NEUMANN JÁNOS
INFORMATIKAI KAR

Sergyán Szabolcs

Algoritmusok, adatszerkezetek I.

ÓE-NIK 5014

Budapest, 2016.

Készült az Óbudai Egyetem Neumann János Informatikai Karán az ÓE-NIK 5014. sz. jegyzetszerződés keretein belül 2014-ben.

Szerző: Dr. Sergyán Szabolcs
egyetemi docens
sergyan.szabolcs@nik.uni-obuda.hu

Lektor: Dr. Vámossy Zoltán
egyetemi docens
vamossy.zoltan@nik.uni-obuda.hu

2.1.0. verzió
2016. december 8.

A jegyzet legfrissebb változata letölthető az alábbi címről:
http://users.nik.uni-obuda.hu/sergyan/Programozas1Jegyzet.pdf

Ez a jegyzet LaTeX segítségével készült.

Tartalomjegyzék

Bevezetés

Ez a jegyzet az Óbudai Egyetem Neumann János Informatikai Karán Mérnök informatikus alapszakon tanuló hallgatók számára készült. A jegyzet a Programozás I. első féléves tárgyhoz íródott. A tárgy célja a hallgatók algoritmikus gondolkodásának fejlesztése, a leggyakrabban használt alapvető algoritmusok megismertetése, valamint a C# programozási nyelv alapjainak megismerése. Jelen jegyzet ebből a három célból csak az első kettő elérésében nyújt segítséget a hallgatóknak. Ezért konkrét programozási nyelvvel nem is foglalkozik, a közölt algoritmusok programozási nyelvi implementációját nem adja meg. A C# programnyelv részletes ismertetése számos magyar vagy angol nyelvű szakkönyvben megtalálható, illetve egy konkrétan ezzel foglalkozó későbbi jegyzet témája lehet.

A jegyzet anyaga nagy mértékben lefedi az előadásokon ismertetésre kerülő algoritmusokat. Az előadások látogatása viszont a jegyzet anyagának megértése és elsajátítása mellett is szükséges, hiszen az előadó ott tud rámutatni a legfontosabb összefüggésekre.

A jegyzetben minden algoritmus azonos módszerrel kerül leírásra. Az egyes algoritmusok bemutatása a megoldandó probléma ismertetésével kezdődik. Ezt követi a megoldásra javasolt módszer szöveges kifejtése, a tervezett algoritmus lépéseinek nagy vonalakban történő ismertetése. Ezután pszeudokód formátumban bemutatásra kerül a konkrét algoritmus, a bemeneti és kimeneti változók megadása mellett. A pszeudokód minden egyes utasítását szövegesen is leírjuk, hogy jól érthető legyen, melyik lépésnek pontosan mi a szerepe. Az algoritmus alapos bemutatását követően egy konkrét példán keresztül is végigvezetjük az algoritmus működését. A példák nyomon követését szolgálják az egyes lépések eredményeit bemutató ábrasorozatok. Az algoritmus leírását a futási idő elemzésével zárjuk.

A tananyag elsajátítása érdekében az alábbi feldolgozási módszer követését ajánljuk. A szorgalmi időszakban az egyes előadások előtt szükséges egyszer átolvasni az előadáson ismertetésre kerülő anyagot, hogy az előadás keretében a hallgató valóban az összefüggések megértésére tudja a figyelmét összpontosítani. Az előadást követően az algoritmusok újbóli átnézését javasoljuk akár számos példa megoldásán keresztül. A jegyzetben közölt mintapéldák mellett érdemes más feladatokon is végigkövetni az egyes algoritmusok működését. Így a hallgató meggyőződhet arról, hogy más esetekben is helyes megoldást szolgáltat a megismert módszer. A vizsgákra készülve ajánlott a jegyzet alapos végigolvasása, minden algoritmus működési elvének megértése, az algoritmusok memorizálása. A vizsga előtti napokban célszerű az algoritmusok pszeudokódjait még egyszer tüzetesen átnézni, azokat „vakon” reprodukálni.

Amennyiben az olvasó hibát talál a jegyzetben, kérjük, hogy azt a szerző felé jelezze. Ha valami nehezen érthető, valamely téma mélyebb kifejtést, vagy más megközelítést követő leírást kíván, akkor ezt is jelezze a szerző felé. Közös célunk, hogy olyan jegyzet kerüljön az olvasó kezébe, ami a tananyag megértését és elsajátítását szolgálja.

A jegyzet felépítése

Az 1. fejezetben az algoritmusok alapjait ismertetjük. Elsőként definiáljuk, hogy mit is értünk algoritmus alatt, majd bevezetjük az ezek leírásához szükséges változók, típusok és kifejezések fogalmát. Bemutatásra kerül az is, hogy miként írjuk le ezeket a fogalmakat a jegyzetben használt pszeudokódokban. Ezt követően definiálunk egy összetett adatszerkezetet, a tömböt. A tömbök feldolgozása képezi lényegében a jegyzet teljes anyagát. Bevezetjük az algoritmusokban használt vezérlési szerkezeteket, valamint megadjuk ezek pszeudokóddal történő leírási módját. Ezt követően összegezzük a pszeudokódokkal kapcsolatos megállapításokat, majd megadjuk azt a leírási módot, amit a jegyzet további részeiben követünk. A fejezet végén az algoritmusok futási idejének elemzési módszerét mutatjuk be.

A 2. fejezetben a leggyakrabban előforduló problémákra adunk hatékony megoldási módszereket, melyeket programozási tételeknek nevezünk. Ismertetésre kerül hat egyszerű, valamint hat összetett

programozási tétel. A fejezet végén leírunk néhány olyan esetet, amikor két programozási tétel összeépítésével tudunk egy problémát megoldani.

A programozási gyakorlatban gyakran előforduló probléma, hogy egy tömbben tárolt értékeket rendezni kell. Számos rendező algoritmust dolgoztak ki, melyek közül a legismertebb elemi módszereket mutatjuk be a 3. fejezetben. Az ismertetésre kerülő hét algoritmus közül általában csak négyet használnak, de didaktikai szempontból fontosnak érezzük mindegyik bemutatott algoritmus megismerését.

A 4. fejezetben bevezetjük a rekurzió fogalmát. Bemutatunk néhány algoritmust annak érdekében, hogy a hallgató megértse a rekurzív algoritmusok működését, lássa azok előnyeit és hátrányait. A fejezet végén leírjuk, hogy a korábban már megismert egyszerű programozási tételek miként valósíthatók meg rekurzív módon.

Az 5. fejezetben rendezett tömbök feldolgozásával foglalkozunk. Bemutatjuk, hogy ilyen tömbök esetén miként lehetséges hatékony keresést megvalósítani. Néhány programozási tétel esetén ismertetjük, hogy a tömb rendezettsége mellett hogyan tudjuk a probléma megoldásának futási idejét javítani. A fejezet végén pedig bemutatunk egy módszert halmazok ábrázolására és ismertetjük a halmazokkal kapcsolatos műveletek megvalósításait.

A 6. fejezetben a tömböknek egy olyan feldolgozási módját mutatjuk be, mely lényegesen különbözik a korábbi fejezetekben ismertetett módszerektől. Az „Oszd meg és uralkodj!" néven ismert megközelítés a tömbök rekurzív feldolgozásának egy speciális módját adja. A fejezetben egy konkrét programozási tétel megvalósítását mutatjuk be az új módszerrel, majd két hatékony és széles körben használt rendező algoritmust ismertetünk. A fejezet végén egy speciális kiválasztási problémát is megoldunk.

A jegyzet 7. fejezetében ún. optimalizálási problémákat oldunk meg. A fejezet célja, hogy a hallgatók megismerjenek speciális probléma megoldási módszereket is. Két megközelítés kerül bemutatásra: a dinamikus programozási módszer és a mohó stratégia.

Az utolsó fejezetben egy speciális adatszerkezetre, a kupacra épülő rendezési módszert mutatunk be. Ez a fejezet már előre mutat a Programozás II. tárgy irányába, hiszen további adatszerkezetek ott kerülnek ismertetésre.

A jegyzetben minden fogalmat a magyar elnevezésével írunk le. Annak érdekében, hogy az angol nyelvű szakirodalom megértésében segítsük a hallgatót a legfontosabb fogalmak angol megfelelőit az első előfordulásuknál lábjegyzetben közöljük. A jegyzet végén kis szótárba gyűjtöttük az ismertetett fogalmakat.

A korábbi évek előadás anyagainak kidolgozása, valamint a jegyzet írása során számos magyar és angol nyelvű szakirodalmat ismertünk meg és dolgoztunk fel. Ezeket a jegyzet végén található irodalomjegyzékben gyűjtöttük össze. Ha valaki mélyebben kívánja megismerni a bemutatott algoritmusokat, akkor javasolt az ajánlott szakirodalmak további tanulmányozása.

Köszönetnyilvánítás

A Programozás I., illetve korábban Algoritmusok, adatszerkezetek, objektumok előadásokat három éven keresztül tartottam Dr. Vámossy Zoltánnal közösen. A tárgyat Dr. Csink Lászlótól vettem át, aki kidolgozta a tárgy alapjait. Ezek kis módosításával érlelődött ki az a tananyag, ami a jegyzetben leírásra került. Köszönöm Csink tanár úr tantárgy kialakítási munkáját, a tőle „megörökölt" tananyagot, módszereket.

Dr. Vámossy Zoltán az elmúlt három évben végig követte előadásaimat, melyekre adott értékes észrevételei beépültek az anyagba. Ezen kívül a dinamikus programozási módszer előadását ő tartja, így a jegyzetben közölt anyagot is ő dolgozta ki. A jegyzetet lektorálta, számos javítási javaslatot fogalmazott meg, amelyek segítettek abban, hogy ilyen formában álljon elő a jegyzet. Köszönetet mondok mindezért a munkáért.

A Programozás I. tárgyat követi a második félévben a Programozás II. A két tárgy tematikáját, egymásra épülését Dr. Szénási Sándorral dolgoztuk ki. Ezen mű születésével együtt ő is megírta az Algoritmusok, adatszerkezetek II. jegyzetet. Az egységes tárgyalásmód érdekében számos témában egyeztettünk, öröm volt vele együtt dolgozni. Köszönet a sok segítségért, amit tőle kaptam.

Szeretnék még köszönetet mondani az Alkalmazott Informatikai Intézet munkatársai közül Bedők Dávidnak, Cseri Orsolya Eszternek, Dr. Erdélyi Krisztinának, Kertész Gábornak, Kiss Dánielnek, Légrádi Gábornak, Simon-Nagy Gabriellának, Nagy Tibor Istvánnak, Szabó Zsoltnak, Szántó Balázsnak és Urbán Andrásnak, akik a Programozás I. tárgy laborjait tartották. Számos értékes észrevételt fogalmaztak meg az előadások tananyagával kapcsolatban, amelyek beépültek a jegyzetbe is.

1. fejezet

Algoritmusok alapjai

Algoritmus fogalma

Algoritmusokkal[1] az élet számos területén találkozhatunk. Például, amikor elmegyünk autót mosatni, akkor egy adott algoritmus fut le. Kifizetjük a mosás árát, amiért kapunk egy mágneskártyát. Beállunk az autómosóba, majd a mágneskártyát behelyezzük egy terminálba és megnyomjuk a start gombot. Ekkor elindul az alábbi folyamat.

1. Előmosás. Az autót mosószeres lével bespriccelik.
2. Kefés mosás. A forgó kefék letisztítják az autót.
3. Öblítés. Az autót tiszta vízzel leöblítik.
4. Szárítás. Az autót levegő áramoltatással megszárítják.

Természetesen ez a folyamat lehet ennél sokkal bonyolultabb is, hiszen különböző programokra fizethetünk elő. Ilyenkor az egyes végrehajtandó tevékenységek megvalósulása attól függ, hogy milyen programot választottunk. Nézzünk erre is egy példát:

1. Ha aktívhabos mosásra fizettünk elő, akkor az autót bespriccelik aktív habbal. Különben csak mosószeres lével spriccelik be az autót.
2. Ha alváz mosásra is előfizettünk, akkor az alvázat is végigspriccelik aktív habbal.
3. Ha kerékmosásra előfizettünk, akkor forgó kefék letisztítják az autót, a kerekeknél pedig speciális keréktisztító kefék mossák a kerekeket. Különben csak a forgó kefék letisztítják az autót.
4. Az autót tiszta vízzel leöblítik.
5. Ha előfizettünk viaszvédelemre, akkor az autót forró viasz réteggel bevonják.
6. Az autót levegőáramoltatással megszárítják.

Látható, hogy ez az algoritmus már döntéseket is tartalmaz, hiszen annak függvényében, hogy milyen programot választottunk más-más történik az autómosás során.

Az előbbi példához hasonlóan számos továbbit tudunk mondani arra, hogy a mindennapokban hol és milyen algoritmusokkal találkozhatunk. Érdemes megismernünk, hogy mi volt az első olyan algoritmus, amelyre azt mondták, hogy az valóban algoritmusnak tekinthető. Az első ilyen algoritmust az ókori görögöknél találhatjuk meg. Euklidész alkotta meg azt a módszert, amely két pozitív egész számról meghatározza a legnagyobb közös osztójukat. Ezt az eljárást ma Euklideszi algoritmusnak nevezzük, és a következőt mondja ki.

1. Adott két pozitív egész szám, jelöljük ezeket m-mel és n-nel. A kettő közül legyen m a nagyobbik.
2. Osszuk el m-et n-nel, az osztás maradékát jelöljük r-rel.

[1] Angolul: algorithm

3. Ha r értéke 0, azaz m osztható n-nel, akkor az algoritmus végére értünk. Ilyenkor a két szám legnagyobb közös osztója az n értékével egyezik meg, az algoritmus pedig befejeződik.

4. Ha r értéke nem 0, akkor m-be tároljuk el az n jelenlegi értékét, n-be pedig az r értékét. Majd ugorjunk vissza a 2. pontra.

1.1. Példa. Az előbb bemutatott Euklideszi algoritmus használatával nézzük meg, hogy mi lesz az $m = 150$ és az $n = 36$ számok legnagyobb közös osztója. Az algoritmus 2. pontja alapján el kell osztanunk m-et n-nel. 150-ben 4-szer van meg a 36, a maradék pedig 6. Ezért az r-be eltároljuk a 6-ot. Az algoritmus 3. pontjára lépünk, megvizsgáljuk, hogy r értéke 0-e. Mivel nem nulla, ezért továbblépünk a 4. pontra. Az m-et felülírjuk, mostantól 36 lesz benne eltárolva. Az n értékét is módosítjuk, ez 6 lesz. Ezután visszaugrunk az algoritmus 2. pontjára. Elosztjuk a 36-ot 6-tal. Az eredmény 6 lesz, a maradék pedig 0. Így r-be a 0 értékét tároljuk el. A 3. sorba lépve megvizsgáljuk r értékét. Mivel $r = 0$, ezért leáll az algoritmus futása és eredményül megkapjuk az n-ben eltárolt aktuális értéket tehát a 6-ot. Ez az érték a 150 és a 36 legnagyobb közös osztója. ¶

Az eddigi példák figyelembe vételével jó lenne megfogalmazni, hogy mit is tekintünk algoritmusnak, egy algoritmusnak milyen elvárásokat kell teljesítenie.

Az algoritmus tekinthető egy olyan „gép”-nek, amely valamilyen bemenetekből meghatározott lépéseken keresztül előállít valamilyen kimenetet. A *bemenet*ek[2] természetesen az algoritmus elején már ismertek. Az Euklideszi algoritmus bemenete például a két pozitív egész szám, melyek legnagyobb közös osztóját akarjuk megismerni. A *kimenet*ek[3] a bemenetek által egyértelműen meghatározottak. Olyan nem állhat elő, hogy egy algoritmus ugyanazon bemenetek esetén más-más kimenetet produkál. Ez úgy is kifejezhető, hogy az algoritmus működése *jól meghatározott*, azaz az algoritmus determinisztikus. Fontos még, hogy az algoritmus egyértelmű, jól definiált lépésekből áll, melyek száma minden esetben *véges*. Ha végtelen lépést is megengednénk, akkor az algoritmus leállása nem lenne biztosítva.

[2] Angolul: input
[3] Angolul: output

Változók, típusok és kifejezések

Ahogy azt az Euklideszi algoritmus példáján láttuk, szükséges lehet, hogy az algoritmus bemeneteit, illetve a futása közben előálló értékeket, ezek módosulásait valamilyen módon eltároljuk. Ezt a célt szolgálják az ún. *változó*k[4].

Minden algoritmusban használt változónak van valamilyen *neve*, amellyel egyértelműen azonosítani tudjuk. Azok a változók, amelyek már az algoritmus kezdetén valamilyen értéket tárolnak, az algoritmus bemeneti változói, vagy röviden *bemenet*ei. A bemeneti változók értékei is módosíthatók az algoritmus futása során. Léteznek olyan változók, amelyek csak az algoritmuson belüli műveletek elvégzése miatt szükségesek, ezek az ún. *lokális változó*k[5]. Vannak olyan változók, amelyekben az algoritmus eredményeként előálló értékeket tároljuk, illetve ezeket adja vissza az algoritmus a külvilág felé. Ezek a változók a kimeneti változók, vagy röviden *kimenet*ek.

A változóknak típusa[6] is van. Ez főként az egyes programozási nyelvekben fontos, mert ez alapján tudja a programozási nyelv értelmezője, illetve fordítója, hogy az adott változó számára milyen méretű memóriát kell lefoglalni a biztonságos működés érdekében. Az algoritmusok leírásánál a típusokat lazább módon kezeljük, mint konkrét programozási nyelvi implementációk esetén.

Megjegyzés

Az fordító készíti el az algoritmust leíró, általában magasabb szintű programozási nyelven megírt forrás kódból azt a gépi kódú programot, amit a számítógép utasításként megért és végrehajt.

Jelen jegyzet keretei között csak néhány típust használunk. Az **egész** típusú változókban egész számok tárolhatók. A **logikai** típusú változókban logikai érték tárolható, azaz értékük csak ***igaz*** vagy ***hamis*** lehet. Ezen kívül használni fogjuk még a **T**-vel jelölt típust, ami egy ún. általános típus. Akkor mondjuk, hogy egy változó **T** típusú, ha benne lényegében bármilyen érték eltárolható. Néhány esetben a **T** típusú változók esetén megszorítást teszünk arra, hogy a változóknak összehasonlíthatóknak kell lenniük, azaz értelmezett közöttük a $<$ reláció. Ilyenkor ezt fel is tüntetjük kihangsúlyozva, hogy **T** összehasonlítható.

Minden egyes változónak lehet értéket adni. Ezt például a következő módon jelöljük: $a \leftarrow 5$, ami azt jelenti, hogy az a értékül kapja az 5-öt. Fontos, hogy csak a bal oldalon lévő változó kaphat értéket, azaz az értékadás nem szimmetrikus művelet. Ezen kívül egy változó értéke átadható egy másik változónak is, ilyenkor az érték mindkét változóban tárolva van, az eredeti helyről nem törlődik. Például a b változó értéke 3, majd kiadjuk a $a \leftarrow b$ utasítást. Ilyenkor az a értéke is 3 lesz, a b pedig 3 marad. Egy változóban tárolt érték ki is olvasható, kifejezésekben valamint kiíratásnál felhasználható.

Gyakran előforduló feladat, hogy két azonos típusú változóban eltárolt értéket meg kell cserélünk. Ezt értékadások egymás utáni elvégzésével tehetjük meg. A csere megvalósításához viszont a két megcserélendő értéket tartalmazó változó mellett szükséges egy harmadik változó is, amiben átmenetileg eltároljuk valamelyik értéket. Az a és b változókban tárolt értékek cseréjét az alábbi módon valósíthatjuk meg:

$$
\begin{aligned}
&segéd \leftarrow a \\
&a \leftarrow b \\
&b \leftarrow segéd
\end{aligned}
$$

Látható, hogy két változó cseréjét három értékadással tudjuk csak megoldani és egy átmeneti változó szükséges még hozzá. Mivel gyakran kell cseréket végrehajtani, ezért külön jelölést vezetünk be erre a műveletre. Az $a \leftrightarrow b$ jelöli az a és b változóban tárolt értékek cseréjét, amit a fent látott módon valósítunk meg.

A változókból kifejezések építhetők. Egy számokat tartalmazó kifejezésben számokat tartalmazó változók, konkrét számértékek és ezeket összekapcsoló matematikai műveletek szerepelhetnek. Kifejezés például a következő:

$$\frac{bal - jobb}{2} + 3. \tag{1.1}$$

[4] Angolul: variable
[5] Angolul: local variable
[6] Angolul: type

Ebben a kifejezésben két szám értékű változó a *bal* és a *jobb* szerepel, melyek különbségének feléhez adunk hozzá hármat.

Ahogy már említettük a logikai típusú változók csak ***igaz*** vagy ***hamis*** értéket vehetnek fel. Így egy logikai változónak csak olyan kifejezés adható értékül, amely kifejezés vagy egyértelműen ***igaz*** vagy ***hamis***. Például mondhatjuk azt, hogy az l logikai változónak értékül adjuk a $2/2 = 1$ kifejezést. Mivel 2-ben a 2 pontosan 1-szer van meg, ezért ez a kifejezés ***igaz***.

Logikai értékek, illetve logikai kifejezések között logikai műveleteket is értelmezhetünk, melyek közül hármat használunk ebben a jegyzetben. Logikai értékek tagadására a $\neg$ szimbólummal jelölt *negálás* műveletet használjuk. Az ***igaz*** érték negáltja a ***hamis***, míg a ***hamis*** negáltja a ***igaz***, ahogy az az 1.1 táblázatban is látható.

l logikai értéke	$\neg l$ logikai értéke
hamis	***igaz***
igaz	***hamis***

1.1. táblázat. A negálás logikai művelet értelmezése.

Két logikai érték vagy kifejezés *és* kapcsolatát is értelmezzük, ezt a műveletet a $\wedge$ szimbólummal jelöljük. Két logikai érték és kapcsolata pontosan akkor ***igaz***, ha mindkét érték ***igaz***, ahogy az az 1.2 táblázatban is látható.

l_1 logikai értéke	l_2 logikai értéke	$l_1 \wedge l_2$ logikai értéke	$l_1 \vee l_2$ logikai értéke
hamis	***hamis***	***hamis***	***hamis***
hamis	***igaz***	***hamis***	***igaz***
igaz	***hamis***	***hamis***	***igaz***
igaz	***igaz***	***igaz***	***igaz***

1.2. táblázat. Az és ($\wedge$) valamint a vagy ($\vee$) kapcsolat értelmezése, az l_1 és l_2 logikai típusú változók értékeinek minden lehetséges variációja esetén.

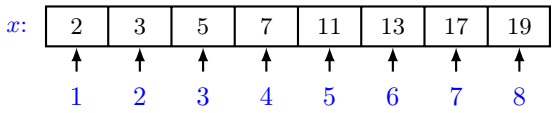
Két logikai érték vagy kifejezés *vagy* kapcsolatát is értelmezzük, ezt a műveletet a $\vee$ szimbólummal jelöljük. Két logikai érték vagy kapcsolata pontosan akkor ***hamis***, ha mindkét érték ***hamis***, minden más esetben ***igaz***, ahogy az az 1.2 táblázatban is látható.

A logikai kifejezések, mint minden más kifejezés balról jobbra kerül kiértékelésre. Így például az $(a = 5) \wedge (b > 3)$ logikai kifejezésben először az $a = 5$, majd a $b > 3$ kifejezés értékelődik ki. Abban az esetben, ha az első kifejezés ***hamis***, akkor az és kapcsolat miatt a teljes kifejezés is biztosan ***hamis*** lesz, tehát a második kifejezés kiértékelése felesleges. Ezt figyelembe veszi az ún. *rövidzár kiértékelés*. Ezek szerint két és kapcsolattal összekötött kifejezésben, ha az első kifejezés értéke ***hamis***, akkor a második kifejezés kiértékelése nem történik meg és a teljes kifejezés értéke ***hamis*** lesz. Hasonlóan két kifejezés vagy kapcsolata esetén, ha az első kifejezés ***igaz***, akkor a második kifejezés nem kerül kiértékelésre, és a teljes kifejezés logikai értéke ***igaz*** lesz.

Tömbök

Gyakran fordul elő, hogy sok azonos típusú adatot akarunk tárolni, illetve ezeken az adatokon valamilyen műveletet elvégezni. Például egy meteorológiai állomáson minden órában eltárolják az aktuális hőmérséklet értéket. Ilyenkor egy nap adatainak tárolására nem célszerű 24 különböző változót létrehozni, heti adatok esetén pedig 168-at. Sokkal célravezetőbb egyetlen változóban összegyűjteni az összes mérési adatot. Ezt valósíthatjuk meg tömbök használatával.

A tömbök használata esetén tehát egyetlen változóban több adatot is el tudunk tárolni. A tömbben tárolt adatoknak azonos típusúaknak kell lenniük. Az 1.1. ábrán láthatunk egy x nevű tömböt, melyben a 20-nál kisebb prím számokat tároltuk el. Ebben az esetben a tömb egész típusú. Minden tömbnek van elemszáma is, példánkban ez 8.



1.1. ábra. A 20-nál kisebb prím számokat tartalmazó tömb és az elemek indexei.

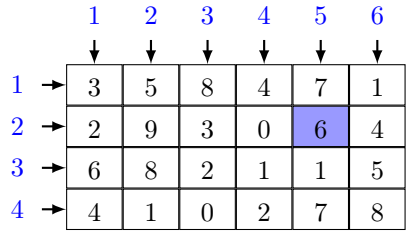
A tömbben tárolt értékek elérése érdekében valamilyen módon hivatkoznunk kell a tömb elemeire. Ezt indexeléssel valósítjuk meg úgy, hogy minden egyes tömbelemhez egyértelműen hozzátartozik egy index. Az indexek logikus rendet követnek: az első tömbelem indexe 1, a másodiké 2, ..., az utolsóé pedig a tömb elemszámával azonos. Az 1.1. ábrán látható, hogy melyik tömbelemnek mi az indexe.

A tömbök indexelése a [ ] indexképző operátor használatával történik. Ha például az x tömb harmadik elemére akarunk hivatkozni, akkor azt a $x[3]$ módon tehetjük meg. Konkrét példánkban $x[3]$ értéke 5. Gyakran fogunk tömbelemekre olyan módon hivatkozni, hogy indexként egy változót használunk. Például $x[i]$ az x tömb i-edik elemét azonosítja. Ilyenkor mindig figyelni kell arra, hogy az alkalmazott indexnek olyan értéke legyen, ami valóban használható indexként. azaz mindenképp pozitív egész számnak kell lennie, és értéke nem haladhatja meg a tömb méretét. Ezek szerint az 1.1. ábrán látható példa esetén csak az 1 és 8 közötti egész számok használhatók indexként.

Algoritmusainkban külön jelezni fogjuk, ha új tömböt hozunk létre. Ilyenkor meg kell adnunk, hogy mi lesz az új tömb neve, milyen típusú adatokat tárolunk benne, illetve hogy mekkora a tömb mérete. Például egy x nevű egészeket tartalmazó nyolc elemű tömböt az $x \leftarrow$ LÉTREHOZ(**egész**)[8] utasítással hozhatunk létre.

Lehetőségünk van többdimenziós tömbök használatára is. Ezek közül most csak a kétdimenziós tömböket ismertetjük. A kétdimenziós tömbök adott számú sorral, illetve oszloppal rendelkeznek. Az elemek indexelése két indexszel, a sor- és az oszlopindexszel történik.

Az 1.2. ábrán megadunk egy 4 sorból és 6 oszlopból álló kétdimenziós tömböt. A kékkel kiemelt elem sorindexe 2, oszlopindexe pedig 5, ezért $x[2, 5]$ módon hivatkozhatunk rá.



1.2. ábra. Kétdimenziós tömb és indexelése.

Kétdimenziós tömb létrehozása is a LÉTREHOZ létrehoz utasítással lehetséges, de méretként meg kell adni a sorok valamint az oszlopok számát is. Így az ábrán látható tömböt az $x \leftarrow$ LÉTREHOZ(**egész**)$[4, 6]$ utasítással hozhatjuk létre.

Vezérlési szerkezetek

Az eddigiekben három műveletet ismertünk meg. Ezek az értékadás, amikor egy változóhoz valamilyen konkrét értéket rendeltünk, illetve a csere, amikor két változóban tárolt értékeket megcserélünk. Harmadik már ismert műveletünk pedig a tömb létrehozása, aminek eredményeként létrejön egy adott típusú és elemszámú új összetett változó. Milyen további műveletekre van szükségünk ahhoz, hogy összetett algoritmusokat építsünk fel? Az igazat megvallva már nem sokra, viszont fontos lenne, hogy az eddig megismert műveleteket valamilyen módon össze tudjuk építeni. Az összeépítések fajtáit *vezérlési szerkezet*eknek nevezzük, mert ezek adják meg, hogy az algoritmus végrehajtása, annak vezérlése miként történjen.

Első vezérlési szerkezetünk a *szekvencia*. Ez utasítások egymás utáni végrehajtását jelenti. Nézzük példaként az alábbi szekvenciát:

$x \leftarrow$ LÉTREHOZ(**egész**)[3]
$x[1] \leftarrow 5$
$x[2] \leftarrow 8$
$x[3] \leftarrow 11$

Négy utasításból áll ez a szekvencia, melyek egymás után kerülnek végrehajtásra. Elsőként létrehozunk egy egész típusú három elemű tömböt, melynek az x nevet adjuk. Ezt követően a tömb első elemébe eltároljuk az 5 értéket. A tömb második helyére a 8 érték kerül, míg a harmadik eleme 11-gyel lesz egyenlő.

Az algoritmusok leírásánál a szekvenciában lévő utasításokat általában egymás alá írjuk. Az utasításokat fentről lefelé haladva sorban hajtjuk végre. Az utasítások sorrendje nagyon fontos, nézzük meg például, hogy mit eredményezne az alábbi szekvencia.

$x[1] \leftarrow 5$
$x[2] \leftarrow 8$
$x[3] \leftarrow 11$
$x \leftarrow$ LÉTREHOZ(**egész**)[3]

Ha ilyen sorrendben írnánk a szekvencia elemeit, akkor hibát követnénk el. Akkor akarnánk a tömb egyes elemeinek értéket adni, amikor a tömb még nem is létezik, hiszen a tömb létrehozása csak a negyedik utasításban történik meg.

Lehetőségünk van arra is, hogy – helytakarékossági okokból – néhány egymást követő utasítást egy sorba írjunk. Ilyenkor az egy sorba kerülő utasításokat pontosvesszővel választjuk el egymástól, ahogy ez az alábbi példában is látható.

$x \leftarrow$ LÉTREHOZ(**egész**)[3]
$x[1] \leftarrow 5$; $x[2] \leftarrow 8$; $x[3] \leftarrow 11$

Második vezérlési szerkezetünk az *elágazás* vagy más néven *szelekció*. Ez a vezérlési szerkezet azt teszi lehetővé, hogy egy utasítást, vagy több utasításból felépülő szekvenciát egy logikai érték igazságának függvényében hajtsunk végre. Az elágazás leírásához szükségünk van meghatározott kulcsszavakra is. Ennek általános formája az alábbi.

ha feltétel **akkor**
    utasítás
elágazás vége

A **ha** kulcsszót követően kell leírnunk a logikai feltételt – ami lehet egy logikai típusú változó, vagy egy logikai értéket eredményező kifejezés –, majd következik az **akkor** kulcsszó. Új sorban behúzással adjuk meg azt az utasítást, amit végrehajt az algoritmus abban az esetben, ha a feltétel ***igaz*** értékű. Az elágazást egy új lezáró sorban behúzás nélkül az **elágazás vége** kulcsszó zárja le. A lezárásra azért van szükség, mert az utasítás helyén akár több utasítás, például egy szekvencia is állhat:

ha feltétel **akkor**
    utasítás1
    utasítás2
    ⋮
    utasításN
elágazás vége

Az elágazásnál az algoritmus futása úgy történik, hogy először kiértékelődik a feltétel. Ha a feltétel ***igaz***, akkor a vezérlés az elágazásban megadott utasítások végrehajtásával folytatódik, majd ezt követően továbbhalad az **elágazás vége** kulcsszót követő részre. Amennyiben viszont a feltétel ***hamis***, akkor a feltétel kiértékelése után a vezérlés rögtön az **elágazás vége** kulcsszót követő részre ugrik.

Nézzük egy konkrét példát:

ha $a < 0$ **akkor**
    $a \leftarrow -a$
elágazás vége
$b \leftarrow \sqrt{a}$

A példában a kiértékelendő feltétel az $a < 0$. Ha az a változó aktuális értéke negatív, akkor a vezérlés az $a \leftarrow -a$ utasításra ugrik, majd innen halad tovább a $b \leftarrow \sqrt{a}$ sorra. Amennyiben a már a feltétel kiértékelésénél se volt negatív, akkor viszont a vezérlés rögtön a $b \leftarrow \sqrt{a}$ sorra ugrik.

Az elágazásból megengedünk ún. kétirányú elágazást is. Ilyenkor más utasítás – vagy utasításokból felépített szekvencia – hajtódik végre, ha az elágazás feltétele ***igaz***, és más ha ***hamis***. A kétirányú elágazást az alább módon adhatjuk meg:

ha feltétel **akkor**
    utasításigazesetben
különben
    utasításhamisesetben
elágazás vége

Ha a feltétel ***igaz***, akkor a **ha** ágban található utasításigazesetben utasítás hajtódik végre, majd a végrehajtást követően a vezérlés az **elágazás vége** kulcsszót követő sorra ugrik. A feltétel ***hamis*** logikai értéke esetén viszont a feltétel kiértékelését követően a vezérlés a **különben** ágban található utasításhamisesetben sorra ugrik. Az ott található utasítás végrehajtását követően az algoritmus futása az **elágazás vége** kulcsszót követő sorban folytatódik.

Nézzünk egy példát a kétirányú elágazásra is:

ha $a < 0$ **akkor**
    $b \leftarrow \sqrt{-a}$
különben
    $b \leftarrow \sqrt{a}$
elágazás vége

Ha az a változó aktuális értéke negatív, akkor az ellentettjének gyöke kerül b-be, egyéb esetben viszont az a gyöke lesz b értéke.

Az elágazások egymásba is ágyazhatók, amiből egy speciális esetet szeretnénk kiemelni. Az alábbi példában megjelenik egy **különben ha** ág:

ha $a > 0$ **akkor**
    $b \leftarrow \sqrt{a}$
különben ha $a < 0$ **akkor**
    $b \leftarrow \sqrt{-a}$
különben
    $b \leftarrow 0$
elágazás vége

Ez teljesen egyenértékű az alábbi egymásba ágyazott elágazásokkal:

ha $a > 0$ **akkor**
    $b \leftarrow \sqrt{a}$
különben
    ha $a < 0$ **akkor**
        $b \leftarrow \sqrt{-a}$
    különben
        $b \leftarrow 0$
    elágazás vége
elágazás vége

A harmadik vezérlési szerkezet a *ciklus*, vagy más néven *iteráció*. Ez arra ad lehetőséget, hogy egy utasítás, vagy utasítások sorozata többször is végrehajtásra kerüljön. Természetesen a végrehajtások száma nem lehet végtelen, mert minden algoritmustól elvárjuk, hogy véges sok lépést követően véget érjen. A ciklusok leállását kétféle módon lehet biztosítani. Vagy csak addig hajtódnak végre a cikluson belüli utasítások, amíg egy meghatározott feltétel ***igaz*** értékű (tesztelős ciklus), vagy előre megadjuk az iterációk számát (számlálós ciklus). Ciklusokból három félét különböztetünk meg.

Első az ún. *elöltesztelős ciklus*. Ennél a ciklusba való belépés előtt kiértékelésre kerül egy feltétel. Ha a feltétel logikai értéke ***igaz***, akkor a cikluson belüli első utasításra ugrik a vezérlés. Végrehajtódik az összes cikluson belüli utasítás, majd ismét kiértékelésre kerül a ciklus feltétele. Ha a feltétel ***igaz***, akkor újra végrehajtódnak a cikluson belüli utasítások. Amint ***hamis***sá válik a ciklus feltétele, akkor a ciklust követő utasításnál folytatódik az algoritmus futása. Az elöltesztelős ciklusnál használt feltételt a fentiek értelmében belépési és bennmaradási feltételnek nevezzük. Az elöltesztelős ciklust az alábbi formában adjuk meg:

ciklus amíg feltétel
    utasítások
ciklus vége

Az Euklideszi algoritmusnál már láttunk is ilyen ciklust, csak még nem neveztük elöltesztelős ciklusnak. Az Euklideszi algoritmus iterációja az alábbi módon írható le, ahol a mod kulcsszó a maradékos osztás operátorát jelöli:

ciklus amíg $r \neq 0$
    $m \leftarrow n$
    $n \leftarrow r$
    $r \leftarrow m \mod n$
ciklus vége

Másik ciklusunk az ún. *hátultesztelős ciklus*. Ennél a ciklusbennmaradási feltétele a ciklus végén van. Ez azt jelenti, hogy a cikluson belüli utasítások egyszer biztosan végrehajtásra kerülnek, majd ezt követően kerül kiértékelésre a ciklus feltétele. Ha a feltétel ***igaz***, akkor az utasítások újra végrehajtódnak, majd megint kiértékelésre kerül a feltétel. Ha viszont a feltétel ***hamis***, akkor a ciklust követő utasításra ugrik a vezérlés. Ebben az esetben a feltétel csak bennmaradási feltétel. A hátultesztelős ciklusok leírási módja:

ciklus
    utasítások
amíg feltétel

A harmadik ciklus az ún. *számlálós ciklus*. Ebben az esetben a ciklusnak van egy *ciklusváltozója*, amely egy megadott értéktől egy másik adott értékig változik úgy, hogy minden felvett értéke mellett egyszer lefutnak a cikluson belüli utasítások. Leírása a következő módon történik:

ciklus *ciklusváltozó* $\leftarrow$ *kezdetiérték*-től *végérték*-ig
    utasítások
ciklus vége

Számlálós ciklusokat tipikuson tömbök bejárására használhatunk. A következő példában egy tömbbe elhelyezzük az első öt darab négyzetszámot:

```
x ← Létrehoz(egész)[5]
ciklus i ← 1-től 5-ig
    x[i] ← i^2
ciklus vége
```

Az eddig említett vezérlési szerkezetekkel, tehát a szekvenciával, szelekcióval és iterációval minden algoritmus leírható. Ezt az állítást külön nem bizonyítjuk.

Algoritmusok leírása, pszeudokód

Az előző fejezetekben már láttuk, hogy mi is az az algoritmus, illetve milyen részekből épül fel. Az eddigieket összefoglalva most bemutatjuk, hogy miként írhatunk le egy konkrét algoritmust. A leírásra az ún. *pszeudokód*ot használjuk. Mutatunk egy példát is, a már megismert Euklideszi algoritmus részletes leírásával, amit az 1.1. algoritmusban adunk meg. A jegyzetben minden algoritmusnak saját sorszáma és neve van, amikkel hivatkozhatunk rájuk.

Mivel minden algoritmusnak van bemenete és kimenete, ezért a leírás ezek pontos megadásával történik. A bemeneti és kimeneti változók nevét és típusát is meghatározzuk. Ezt követi az adott algoritmust megvalósító *függvény* vagy *eljárás* megadása. Azért van szükség függvényekre, illetve eljárásokra, mert ezeken keresztül egy konkrét algoritmust egy másik algoritmusból akár meg is hívhatunk (ld. később). Az algoritmus bemenetei egyben a függvény vagy eljárás bemenetei is lesznek. A függvény abban különbözik az eljárástól, hogy a függvénynek van legalább egy kimenete, míg az eljárásnak nincs ilyen. A függvény kimenetét a **vissza** kulcsszó használatával adhatjuk meg (ld. 1.1. algoritmus 8. sora). Ha a vezérlés a **vissza** kulcsszóhoz kerül, akkor a függvény futása véget is ér.

1.1. Algoritmus Euklideszi algoritmus

Bemenet: m – **egész**, n – **egész**
Kimenet: n – **egész**

```
1: függvény LNKO(m : egész, n : egész)
2:     r ← m mod n
3:     ciklus amíg r ≠ 0
4:         m ← n
5:         n ← r
6:         r ← m mod n
7:     ciklus vége
8:     vissza n
9: függvény vége
```

Felhasznált változók és függvények

- m: Pozitív egész szám.
- n: Pozitív egész szám, amely kezdetben nem nagyobb m-nél. Az algoritmus végén ennek a változónak az értéke a két bemeneti változó legnagyobb közös osztójával egyenlő.
- r: Az aktuális m és n maradékos osztásának eredménye.

A függvényen vagy eljáráson belül a már korábban megismert algoritmus leíró eszközöket használjuk. Ezek az értékadás, tömblétrehozás, illetve a megismert vezérlési szerkezetek.

A függvényt vagy eljárást követően ismertetjük az algoritmusban használt változókat is.

Az Euklideszi algoritmust leírhatjuk függvény helyett eljárás használatával is, ahogy az 1.2. algoritmusban látható. Ilyenkor természetesen nincs a **vissza** kulcsszóval megadott visszatérési értéke az eljárásnak. A kimenetet az eljárás egyik paraméterén keresztül kapjuk meg, amihez a **címszerint** kulcsszót írjuk az adott változó elé.

Az algoritmusokban lehetőségünk van arra is, hogy egy másik algoritmusban definiált függvényt vagy eljárást meghívjunk. Erre láthatunk egy példát az 1.3. algoritmusban. Az algoritmus azt vizsgálja, hogy egy tömb mely elemei relatív prímek egy megadott értékkel. Ennek eldöntéséhez meg kell nézni, hogy a vizsgált x tömb aktuális elemének ($x[i]$) és az *érték* változónak mi a legnagyobb közös osztója. Ehhez meghívjuk az 1.1. Euklideszi algoritmusban definiált LNKO függvényt, ahogy ez az 1.3. algoritmus 4. sorában látható.

1.2. Algoritmus Euklideszi algoritmus (2)

Bemenet: m – **egész**, n – **egész**
Kimenet: n – **egész**

```
1: eljárás LNKO(m : egész, címszerint n : egész)
2:     r ← m mod n
3:     ciklus amíg r ≠ 0
4:         m ← n
5:         n ← r
6:         r ← m mod n
7:     ciklus vége
8: eljárás vége
```

Felhasznált változók és függvények

- m: Pozitív egész szám.
- n: Pozitív egész szám, amely kezdetben nem nagyobb m-nél. Az algoritmus végén ennek a változónak az értéke a két bemeneti változó legnagyobb közös osztójával egyenlő.
- r: Az aktuális m és n maradékos osztásának eredménye.

1.3. Algoritmus Relatív prím vizsgálat

Bemenet: x – **egész tömb**, n – **egész** (*tömb mérete*), *érték* – **egész**
Kimenet: y – **logikai tömb**

```
1: függvény RelatívPrímVizsgálat(x : egész tömb, n : egész, érték : egész)
2:     y ← Létrehoz(logikai)[n]
3:     ciklus i ← 1-től n-ig
4:         ha LNKO(x[i], érték) = 1 akkor
5:             y[i] ← igaz
6:         különben
7:             y[i] ← hamis
8:         elágazás vége
9:     ciklus vége
10:    vissza y
11: függvény vége
```

Felhasznált változók és függvények

- x: Pozitív egész értékeket tartalmazó tömb.
- n: Az x tömb elemszáma.
- *érték*: Pozitív egész szám. Az algoritmusban azt vizsgáljuk, hogy az x tömb relatív prímek-e az *érték* számmal.
- y: Kimeneti logikai típusú tömb. Az y tömb i-edik eleme pontosan akkor ***igaz***, ha az x tömb i-edik eleme relatív prím az *érték*-kel.
- Létrehoz(**logikai**)[n]: Utasítás, mely létrehoz egy n elemű logikai típusú tömböt.
- LNKO($x[i]$, *érték*): Az 1.1. algoritmusban kifejtett LNKO függvényt hívja meg, mely meghatározza, hogy a $x[i]$-nek és az *érték*-nek mi a legnagyobb közös osztója.

Hatékonyság, futási idő elemzése

Láttuk már, hogy miként tudunk leírni egy algoritmust. Felmerül viszont, hogy milyen elvárásaink vannak egy algoritmussal szemben. Röviden tekintsük ezért át az algoritmusokkal szemben támasztott követelményeket.

Fontos az algoritmus *megbízható*sága és *kiszámítható*sága. Ez alatt azt értjük, hogy minden lehetséges bemenet esetén valóban azt a kimenetet állítsa elő, amit elvárunk az algoritmustól. A kiszámíthatóságba beleértjük azt is, hogy egy adott bemenet esetén mindig ugyanazt a kimenetet állítsa elő. Ezt más néven az algoritmus determinisztikus (előre meghatározott) viselkedésének is nevezzük.

A jó algoritmusok *egyszerű*ek is. Ez alatt azt értjük, hogy nem túl körülményesek, könnyen megérthetőek. Az egyszerűség érdekében érdemes az algoritmusokat olyan változó nevekkel ellátni, amik alapján egyértelmű, hogy az adott változóban milyen adatot is tárolunk el. Az egyszerűség érdekében érdemes az algoritmusokat részekre, ún. modulokra osztani. Erre mutat jó példát az 1.3. algoritmus, ahol a legnagyobb közös osztó meghatározását nem írjuk le újra az algoritmuson belül, hanem egy már máshol megvalósított függvényt hívunk meg. Az 1.3. algoritmus olvasáskor nem az köti le a figyelmünket, hogy a legnagyobb közös osztó kiszámítása miként történik, csak azt látjuk, hogy az adott helyen annak számítása megtörténik.

Az algoritmusokkal kapcsolatban elvárás még, hogy *hatékony*ak legyenek. Ez alatt általában két különböző dolgot is értünk. Egyrészt elvárás, hogy az algoritmus számítógépes implementációja során minél kevesebb memória, vagy más háttértár igénye legyen. Ezért, ha egy algoritmus ugyanazt a feladatot meg tudja oldani egy darab n elemű tömb használatával, míg egy másik algoritmusnak ugyanehhez két darab n elemű tömb is szükséges, akkor memória felhasználás tekintetében az elsőt tekintjük hatékonyabbnak. A hatékonyság fogalmában értjük azt is, hogy az algoritmus gyorsan fusson le. Persze ennek meghatározása elég nehéz, sokszor nagyon szubjektív feladat. A fejezet további részében épp ezért azzal foglalkozunk, hogy egy algoritmus futási idejét miként lehet meghatározni vagy megbecsülni.

Hogyan is lehet egy algoritmus futási idejét meghatározni? Első ötletünk talán az, hogy az adott algoritmust egy adott számítógépen, valamilyen programozási nyelven lekódoljuk, majd futtatjuk az elkészült programot. A futó program végrehajtási idejét mérve már meg is tudjuk határozni az adott algoritmus futási idejét. Ez az egyszerűnek tűnő eljárás viszont számos problémát vet fel. Ha az algoritmusunkat más gépen vagy más programozási nyelven kódoltuk volna, akkor különböző futási időt kapunk. Azt se felejtsük el, hogy egy számítógépen egyszerre nem csak egy program fut, tehát miközben a saját programunk futási idejét mérjük, nem tudhatjuk, hogy milyen más programok „zavarnak még be" futás közben. Ezek miatt ezt az empirikus módszert elvetjük, az algoritmusok futási idejét nem így határozzuk meg.

Másik lehetőségünk annak vizsgálata, hogy az adott algoritmus hány darab elemi lépésből áll. Megnézzük tehát, hogy hány értékadás, összehasonlítás, logikai kifejezés kiértékelés, függvény hívás, stb. történik egy algoritmus futása alatt. Ha nézzük például az Euklideszi algoritmust, ott adott bemenetek mellett össze tudjuk számolni ezen műveletek számát. Viszont azt is látnunk kell, hogy minden „elemi" művelet végrehajtása más-más időigénnyel rendelkezik. Például az $m \leftarrow n$ értékadás esetén annyi történik, hogy az m változóba átmásolódik az n változó értéke. Az $r \leftarrow m \mod n$ értékadásnál viszont először ki kell értékelni az $m \mod n$ kifejezést, majd ezt követően kell az előálló értéket az r változóba másolni. Sőt az $m \mod n$ kifejezés kiértékelése sokkal bonyolultabb művelet mint egy másolás, ezért pontosan meg se tudjuk mondani, hogy hányszor több időt igényel.

Mindebből az látható, hogy elég nehéz egzakt dolgot mondani egy algoritmus futási idejéről. Mégis, miként tudjuk akkor egy algoritmus futási idejét megbecsülni? A leggyakrabban használt megközelítés annak vizsgálata, hogy a feldolgozandó adatok mennyiségének növekedésével miként növekszik egy algoritmus futási ideje. Azt vizsgáljuk tehát, hogy ha például kétszeresére növeljük a feldolgozandó adatok számát, akkor hányszorosára növekszik ettől az algoritmus futási ideje. Emiatt azt fogjuk vizsgálni, hogy n darab bemeneti adat esetén a futási idő hogyan függ az n értéktől.

Nézzük egy konkrét példát! Feladatunk, hogy egy n elemű egész számokat tartalmazó tömbben határozzuk meg, hány esetben eredményez a tömb két elemének összege 0 értéket. Ezt a feladatot valósítja meg az 1.4. algoritmus. Az algoritmus működésének lényege, hogy minden lehetséges elempárt előállítunk, amit két darab egymásba ágyazott ciklussal érünk el. Minden egyes elempár esetén megnézzük, hogy az összegük 0-e (ld. 5. sor). Ha igen, akkor a *db* változó értékét – amely kezdetben 0 volt – növeljük eggyel (ld. 6. sor). Az algoritmus végén a *db* változó aktuális értékét adjuk vissza (ld. 10. sor).

1.4. Algoritmus Nullát eredményező elempárok száma

Bemenet: x – **egész tömb**, n – **egész**
Kimenet: db – **egész**

```
1: függvény NullátAdóElempárokSzáma(x : egész tömb, n : egész)
2:     db ← 0
3:     ciklus i ← 1-től (n − 1)-ig
4:         ciklus j ← (i + 1)-től n-ig
5:             ha x[i] + x[j] = 0 akkor
6:                 db ← db + 1
7:             elágazás vége
8:         ciklus vége
9:     ciklus vége
10:    vissza db
11: függvény vége
```

Felhasznált változók és függvények

- x: A feldolgozandó tömb.
- n: Az x mérete.
- db: Azon x-beli elempárok száma, melyek összege 0.

Mit mondhatunk az 1.4. algoritmus futási idejéről? Ehhez vizsgáljuk meg először, hogy hányszor kerül a $x[i] + x[j] = 0$ feltétel kiértékelésre. Ha $i = 1$, akkor j $(n-1)$ darab különböző értéket vesz fel, hiszen $2, 3, \ldots, n$ lehet. Amennyiben $i = 2$, akkor j már csak $(n-2)$-féle lehet. Amikor viszont i már $n-1$ értékű, akkor j csak egy értéket az n-t veszi fel. Így a vizsgált feltétel kiértékeléseinek száma:

$$(n-1) + (n-2) + \ldots + 2 + 1 = \frac{(n-1)+1}{2} \cdot (n-1) = \frac{n\,(n-1)}{2}. \tag{1.2}$$

Hányszor fogjuk elvégezni a $db \leftarrow db + 1$ értékadást? Ez attól függ, hogy hányszor volt igaz a $x[i] + x[j] = 0$ feltétel. Lehetséges, hogy a feltétel mindig igaz volt, lehet viszont, hogy soha. Futási idő szempontjából a legrosszabb eset, ha mindig igaz a feltétel, ilyenkor a $db \leftarrow db+1$ értékadás is $\frac{n(n-1)}{2}$-szer kerül végrehajtásra. Így, ha minden elemi utasítást egy lépésnek tekintünk, akkor legrosszabb esetben a futási idő már

$$2 \cdot \frac{n\,(n-1)}{2} = n\,(n-1)\,. \tag{1.3}$$

Futási idő szerint legjobb esetben viszont a $db \leftarrow db + 1$ értékadás egyszer sem történik meg, így ilyenkor a futási idő

$$\frac{n\,(n-1)}{2}. \tag{1.4}$$

Azt még nem vettük figyelembe, hogy az algoritmus elején van egy $db \leftarrow 0$, illetve a végén egy **vissza** db utasítás, ami minden esetben kettővel növeli a szükséges lépések számát. Emellett még a ciklusok megvalósításának is van valamennyi futási ideje.

Mi az ami ezek alapján biztosan kijelenthető az algoritmus futási idejéről? Ami egyértelműen látszik az annyi, hogy a futási idő biztosan arányos a feldolgozandó tömb méretének négyzetével, hiszen az 1.3. és az 1.4. képletből is ez olvasható ki. Tehát, ha a feldolgozandó tömb méretét kétszeresére növeljük, akkor a futási idő már négyszeres lesz, háromszoros méret esetén pedig kilencszeres.

Hogyan lehet eldönteni, hogy egy algoritmus futási ideje jobb-e egy másik algoritmus futási idejénél? Ha a futási idő elemzésénél azt kapjuk például eredményül, hogy egy algoritmus futási ideje

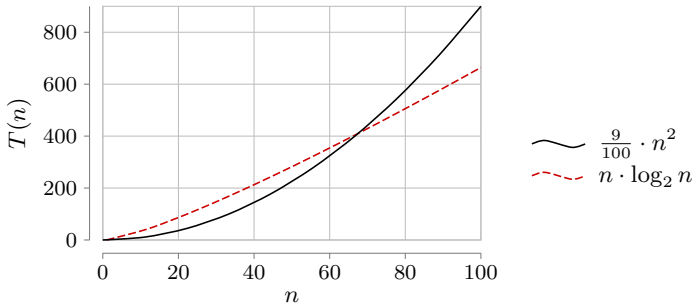
$$T_1\,(n) = \frac{9}{100} \cdot n^2, \tag{1.5}$$

egy másik algoritmusé pedig

$$T_2\,(n) = n \cdot \log_2 n, \tag{1.6}$$

akkor melyik tekinthető hatékonyabbnak? Az 1.3. ábrán megadjuk a két függvény grafikonját. Látható, hogy kisebb n értékek esetén az első, míg nagyobb n értékeknél a második algoritmus eredményez

gyorsabb futást. Mivel számunkra az a fontos, hogy n növekedésével miként változik a futási idő, ezért a nagy n értékekre koncentrálunk. Emiatt a második algoritmus futási idejét tekintjük jobbnak, hiszen minél több adatot kell feldolgozni, annál gyorsabban szolgáltat eredményt az első algoritmushoz képest.



1.3. ábra. A $T_1(n) = \frac{9}{100} \cdot n^2$ és a $T_2(n) = n \cdot \log_2 n$ függvények grafikonjai.

Könnyen belátható, hogy ha az első algoritmus futási idejénél a konstans szorzó nem $\frac{9}{100}$, hanem annál még kisebb lenne, akkor is elég nagy n esetén a $T_1(n)$ már nagyobb lenne $T_2(n)$-nél. Tehát a futási idő nagy n esetén lényegében független a konstans szorzó értéktől. Ezt fejezi ki a nagy ordó jelölés, amit gyakran használunk algoritmusok futási idejének jellemzésére.

Hogyan definiáljuk az O-val jelölt nagy ordót? Azt mondjuk, hogy a $T(n)$ futási idő $O(f(n))$-es, ha létezik olyan c konstans, hogy elég nagy n értékek esetén a

$$T(n) \leq c \cdot f(n). \tag{1.7}$$

Ezek alapján mondhatjuk például, hogy az 1.4. algoritmus futási ideje $O(n^2)$-es.

Az 1.3. táblázatban összefoglaltuk a leggyakrabban előforduló futási időket. A táblázat tetejétől lefelé haladva egyre nagyobb futási időkkel találkozunk.

Futási idő nagyságrendje	Futási idő nagyságrendjének elnevezése
$O(1)$	Konstans
$O(\log n)$	Logaritmikus
$O(n)$	Lineáris
$O(n \log n)$	Logaritmetikus
$O(n^2)$	Négyzetes
$O(n^3)$	Köbös
$O(2^n)$	Exponenciális

1.3. táblázat. Futási idők nagyságrendjei és azok elnevezései

2. fejezet

Programozási tételek

A programozási tételek az algoritmus alkotás során gyakran előforduló problémákra adnak megoldásokat. Azért nevezzük ezeket a módszereket tételeknek, mert matematikailag is bizonyítható az ismertetett módszerek helyessége és hatékony futási idejük. A programozási tételek néhány esetben már első olvasásra is nagyon egyszerűnek tűnnek, esetleg megkérdőjelezhető pontos ismeretük szükségessége is. Fontos szerepük van viszont abban, hogy ezeket a tételeket általában minden programozó ismeri, így a napi rutinban egy adott feladat megoldására ezeket használja. Így kijelenthető, hogy a programozási tételek által adott megoldások használatával az előálló algoritmusok, illetve programok olvashatósága és megértése is sokkal egyszerűbbé válik.

A programozási tételeket jegyzetünkben tömbök használatával mutatjuk be, mert jelenleg csak ezt az adatszerkezetet ismerjük. Tanulmányaink előrehaladtával könnyen meg tudjuk majd oldani, hogy más adatszerkezetek (például listák) esetén is megadjuk a megfelelő tétel megvalósítását, de ez nem képezi ezen jegyzet tárgyát.

A programozási tételek elnevezésének jelentős szerepe van a későbbi algoritmus alkotási gyakorlatban. Az itt bevezetett fogalmak általában mindig ugyanazzal a jelentéssel bírnak. Ezen fogalmak közül szeretnénk kiemelni párat az alábbiakban.

Eldöntés Eldönti, hogy a bemeneti tömbben van-e adott tulajdonságú elem.

Kiválasztás Megadja, hogy hol van a bemeneti tömbben egy adott tulajdonságú elem, feltételezve, hogy van ilyen a tömbben.

Keresés Az előző kettő kombinációja, tehát eldönti, hogy a bemeneti tömbben van-e adott tulajdonságú elem, és ha van, akkor megadja, hogy hol találjuk meg azt.

Megszámlálás Megadja, hogy hány darab adott tulajdonságú elem van a bemeneti tömbben.

Másolás Egy tömb elemeinek egyik tömbből másik tömbbe másolása, melynek során a tömbbeli elemek értékei változhatnak.

Kiválogatás A bementi tömb adott tulajdonságú elemeinek átmásolása egy másik tömbbe.

Szétválogatás A bemeneti tömb különböző tulajdonságú elemeinek átmásolása különböző tömbökbe.

A programozási tételeket jegyzetünkben két nagy csoportra bontjuk. Az első csoportba tartoznak azok a tételek, melyek egy bemeneti tömbből állítanak elő egyetlen (vagy esetleg több) értéket. Ezeket egyszerű programozási tételeknek nevezzük és a 2.1. alfejezetben tárgyaljuk. A második csoportba tartoznak azok a tételek, amelyek bemenete egy vagy több tömb, és a kimenete is egy vagy több tömb. Ezek alkotják az összetett programozási tételeket, melyek tárgyalására a 2.2. alfejezetben kerül sor. A fejezet végén mutatunk pár példát a programozási tételek összeépítésére is (ld. 2.3. alfejezet), majd pár feladatot is megoldunk (ld. 2.4. alfejezet).

Egyszerű programozási tételek

Sorozatszámítás programozási tétel

A sorozatszámítás programozási tétel egy tömb (vagy más néven sorozat) összes eleme között elvégez egy műveletet, majd a művelet eredményét adja vissza. A tétel használható például egy tömb minden elemének összegzésére, az elemek szorzatának kiszámítására. Hasonlóan alkalmas egy sorozat elemeinek uniójaként előálló halmaz elkészítésére. Gyakori alkalmazási terület még szövegeket tartalmazó tömbelemek egyetlen szöveggé történő összefűzése.

> **Megjegyzés**
>
> A sorozatszámítás tételt a szakirodalom egy részében összegzés[a] tételként említik, mivel a tömb elemeinek összegzése is megvalósítható vele. Tárgyalásunkban mi az általánosabb feladatra utaló nevet választottuk.
>
> [a] Angolul: summation

A programozási tételt megvalósító algoritmus pszeudokódját a 2.1. algoritmusban írjuk le. Az algoritmus bemenete a feldolgozandó tömb, melynek természetesen az elemszámát is ismernünk kell. Az algoritmus konkrét megvalósításánál tudnunk kell azt is, hogy a tömb elemei között milyen műveletet kívánunk végrehajtani, ezt jelöljük általánosan az $\oplus$ szimbólummal. A $\oplus$ műveletet nem adjuk át bemenetként az algoritmusnak, mivel általában a konkrét programozás nyelvi implementációkban sem tudunk így eljárni. Az algoritmus kimenete egyetlen eredmény, melynek típusa megegyezik a tömb elemeinek típusával, értéke pedig a tömb összes elemén elvégzett $\oplus$ művelettel kapható meg:

$$\textit{érték} = x[1] \oplus x[2] \oplus \ldots \oplus x[n].$$

2.1. Algoritmus Sorozatszámítás programozási tétel

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*)
Kimenet: *érték* – **T**

```
1: függvény SOROZATSZÁMÍTÁS(x : T tömb, n : egész)
2:     érték ← érték_0
3:     ciklus i ← 1-től n-ig
4:         érték ← érték ⊕ x[i]
5:     ciklus vége
6:     vissza érték
7: függvény vége
```

Felhasznált változók és függvények

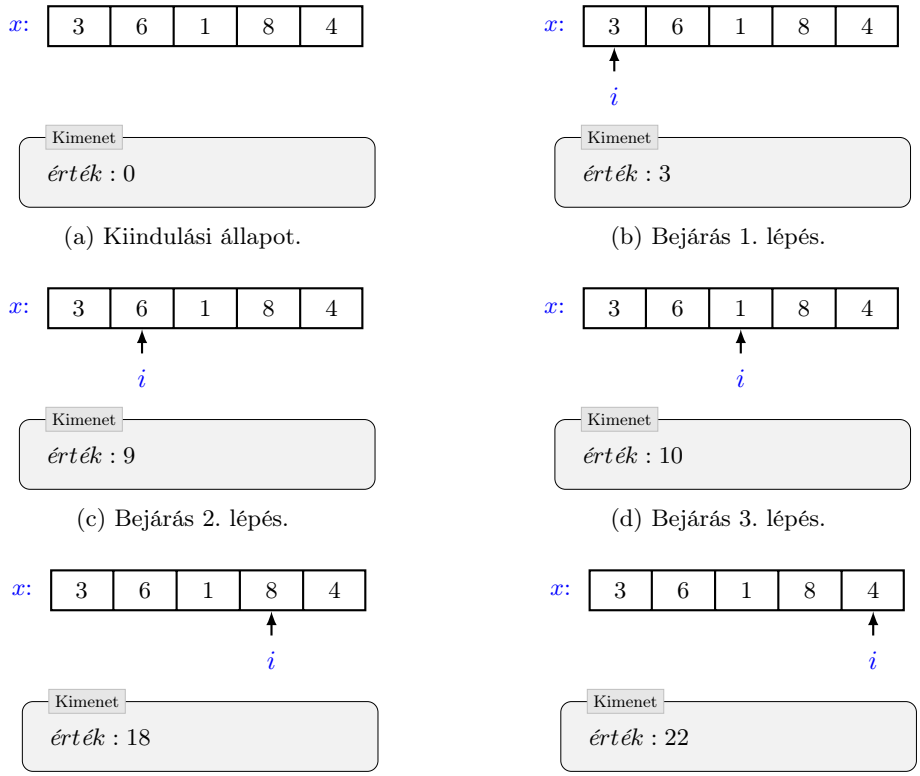
- x: Vizsgált tömb.
- n: A tömb mérete.
- $\oplus$: **T** típusú adatokon értelmezett művelet, amit a tömb összes eleme között hajtunk végre.
- *érték*: Az adott műveletnek ($\oplus$) a tömb összes elemére történő alkalmazását követően előálló eredmény.
- $\textit{érték}_0$: Az alkalmazott művelettől függő kiindulási érték.

Az algoritmus először kezdeti értéket ad az *érték* változónak (2. sor). A kiindulási érték ($\textit{érték}_0$), attól függ, hogy milyen műveletet hajtunk végre a tömb elemei között. Például összeadás esetén 0, szorzás esetén pedig 1 az értéke. Ha az $\oplus$ művelet az unió képzés, akkor az üreshalmazzal, szövegek összefűzése esetén pedig az üres szöveggel inicializáljuk az *érték* változót.

Az algoritmus 3. és 5. sorai közötti ciklussal bejárjuk a teljes x tömböt, így biztosítva, hogy minden elemén el tudjuk végezni a kívánt műveletet. A 4. sorban a *érték* változó korábbi értéke és a tömb aktuális $x[i]$ eleme között elvégezzük az $\oplus$ műveletet, melynek eredményével felülírjuk az *érték* változó korábbi értékét. Itt válik világossá, hogy a 2. sorban miért azt a kezdeti értéket használtuk, amit az adott művelethez választottunk. Összeadás esetén például $i = 1$ esetén a nullához akarjuk hozzáadni a tömb első elemét, szorzásnál viszont az egyet kell megszoroznunk az első elemmel.

A tömb bejárását követően a 6. sorban visszaadjuk a kiszámított *érték*-et.

2.1. Példa. Az algoritmus működését egy konkrét példán is szemléltetjük, ami a 2.1. ábrán látható. A példában egy öt elemű tömb elemeit összegezzük, így a figyelembe vett $\oplus$ művelet itt a jól ismert $+$ művelet.



(a) Kiindulási állapot.

(b) Bejárás 1. lépés.

(c) Bejárás 2. lépés.

(d) Bejárás 3. lépés.

(e) Bejárás 4. lépés.

(f) Bejárás 5. lépés.

2.1. ábra. Sorozatszámítás programozási tétel. A példában egy öt elemű tömb elemeinek összegét számítjuk ki. A kiindulási érték (*érték*$_0$ itt 0, az eredmény pedig az *érték* változóba kerül.

Először az *érték* változó kezdeti értéket kap (2.1a. ábra), ami esetünkben az összeadás miatt 0. Ezt követően a tömb minden elemén végighaladva az *érték* változót mindig az aktuális tömbbeli értékkel növeljük (2.1b-2.1f. ábra). Az utolsó elem figyelembe vételét követően az *érték* változóban pont a tömb elemeinek értéke állt elő (2.1f. ábra). ¶

Futási idő elemzése. A sorozatszámítás programozási tétel futási idejéről könnyen látható, hogy a tömb méretével egyenesen arányos, azaz $O(n)$-es. Ennek oka, hogy minden elemet pontosan egyszer olvasunk ki, és mivel nincs semmiféle feltétel vizsgálat az algoritmusban, így nem állhat elő olyan eset, amikor valamelyik lépés kimaradna az algoritmusból. ♣

Eldöntés programozási tétel

Az eldöntés[1] programozási tételt akkor használjuk, amikor szeretnénk eldönteni, hogy egy tömbben van-e legalább egy adott tulajdonságú elem. Tegyük fel például, hogy egy tömbben egész számokat tárolunk és szeretnénk megtudni, hogy van-e közöttük legalább egy páros szám. Ebben az esetben a párosságot tekintjük a vizsgált tulajdonságnak.

Az algoritmust megalkothatjuk olyan módon, hogy a sorozatszámítás tételhez hasonlóan végigjárjuk az egész tömböt, és amennyiben találunk egy adott tulajdonságú elemet, akkor egy logikai változót – mely kezdetben ***hamis*** volt – ***igaz***ra állítjuk. Így minden elemet meg kell vizsgálnunk, ami számos esetben teljesen felesleges. Ha például már a tömb első elemére teljesül a vizsgált tulajdonság, akkor a tömb többi elemét nem érdemes vizsgálni. Ezen probléma kiküszöbölésére egy olyan algoritmust kell megalkotnunk, amely a tömb elemeinek vizsgálatát abbahagyja abban az esetben, ha már találtunk egy adott tulajdonságú elemet a tömbben.

Az eldöntés problémáját a 2.2. algoritmus oldja meg. Az algoritmus bemenetként megkapja a vizsgált tömböt – amelynek elemszáma is ismert – és a vizsgált tulajdonságot. A tulajdonságot egy ún. tulajdonság[2] függvényként adjuk meg, melyet általában P-vel jelölünk. A tulajdonság függvény logikai értéket ad vissza, ami ***igaz*** vagy ***hamis***lehet. Az algoritmusban a tömb bejárását és a tulajdonság teljesülésének vizsgálatát a 3. és 5. sorok közötti ciklus valósítja meg. A ciklus feltételének vizsgálata előtt a tömb indexelésére használt i változónak kezdeti értéket kell adni, ami a tömb elejéről történő bejárás miatt 1 lesz (2. sor).

2.2. Algoritmus Eldöntés programozási tétel

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai** (*tulajdonság*)
Kimenet: van – **logikai**

```
1: függvény Eldöntés(x : T tömb, n : egész, P : logikai)
2:     i ← 1
3:     ciklus amíg (i ≤ n) ∧ ¬P(x[i])
4:         i ← i + 1
5:     ciklus vége
6:     van ← (i ≤ n)
7:     vissza van
8: függvény vége
```

Felhasznált változók és függvények

- x: Vizsgált tömb.
- n: A tömb mérete.
- P: Tulajdonság függvény, amely minden **T** típusú érték esetén ***igaz*** vagy ***hamis*** értéket ad vissza. Az algoritmus P tulajdonságú elem előfordulását vizsgálja az x tömbben.
- van: Logikai típusú kimeneti változó, amely pontosan akkor lesz ***igaz***, ha van P tulajdonságú elem az x tömbben. Egyéb esetben van értéke ***hamis*** lesz.

A 3. sorban található ciklusfeltételben két dolog együttes teljesülését vizsgáljuk. Egyrészt figyelnünk kell arra, hogy az i változó ne legyen nagyobb a tömb n elemszámánál. Ha i már meghaladná n-t, akkor egyrészt nem tudnánk indexelésre használni – hiszen nem lenne valós indexe a tömbnek –, másrészt azt jelentené, hogy a tömb minden elemét megvizsgáltuk, tehát teljesen felesleges további vizsgálatot végezni. A ciklus féltétel második tagja azt ellenőrzi, hogy az aktuális $x[i]$ tömbelem teljesíti-e az P tulajdonságot. Amennyiben nem teljesíti, akkor tovább kell vizsgálódnunk, ezért a ciklusmagban növeljük az i index értékét (4. sor). Ha viszont $x[i]$ P tulajdonságú, akkor találtunk a tömbben egy P tulajdonságú elemet, így nem kell már további vizsgálatokat végeznünk.

[1] Angolul: decision
[2] Angolul: property

Megjegyzés

A ciklus feltétele kapcsán fontos megemlíteni, hogy a két vizsgált feltétel sorrendje nem felcserélhető. Ennek akkor van jelentősége, ha a tömbben nincs egyetlen P tulajdonságú elem sem. Ilyenkor ugyanis i értéke meg fogja haladni n értékét, konkrétan $n+1$ lesz. Viszont $(n+1)$-gyel nem indexelhetjük a tömböt, mert ez már nem létező index. Feltételek rövidzár kiértékeléséről tudjuk, hogy egy és kapcsolatban ($\wedge$) az első tag ***hamis*** értéke esetén a második tagot már felesleges vizsgálni, mivel ha az első tag ***hamis***, akkor az egész feltétel is ***hamis*** lesz. Így $i = n+1$ esetén a tömböt nem fogjuk hibásan indexelni, mert az első tag akkor már ***hamis*** értékként kiértékelésre került.

A ciklusból két esetben tudunk kilépni. Ha $i \leq n$ igaz, akkor a $\neg P\,(x[i])$ lett ***hamis***, azaz $P\,(x[i])$ ***igaz***. Ekkor a tömb i-edik eleme P tulajdonságú, tehát van a tömbben vizsgált tulajdonságú elem. Ha viszont az első feltétel ***hamis***, azaz $i > n$, akkor a tömb minden elemét megvizsgáltuk már a ciklusban és sehol nem találtunk P tulajdonságú elemet. Ezek alapján kijelenthető, hogy a vizsgált tulajdonságú elem előfordulásáról az i index n-hez való viszonya egyértelmű információt szolgáltat. Így az algoritmus 6. sorában a *van* változónak ezen vizsgálat eredményét adjuk át. Az algoritmus végén a függvény a *van* változó értékével tér vissza (7. sor).

Megjegyzés

Az eldöntés tétel kis átalakítással alkalmassá tehető annak vizsgálatára, hogy egy tömb minden egyes eleme teljesít-e egy adott tulajdonságot. Ehhez a 2.2. algoritmusban két módosítás szükséges. Elsőként a 3. sorban lévő feltételt kell úgy módosítani, hogy addig maradjunk a ciklusban – azaz addig járjuk be a tömb elemeit –, amíg a vizsgált elem P tulajdonságú: $(i \leq n) \wedge P\,(x[i])$. Ha a vizsgált elem nem P tulajdonságú, akkor be kell fejeznünk a tömb bejárását.

A másik módosítási pont a 6. sorban található vizsgálat, mely értéket ad a *van* változónak. Akkor mondhatjuk, hogy minden elem P tulajdonságú a tömbben, ha a ciklusból azért léptünk ki, mert minden elemet megvizsgálva mindig teljesült az $P\,(x[i])$ feltétel, azaz $i \leq n$ lett hamis.

Az átalakított esetet a 2.3. algoritmusban találhatjuk meg teljes egészében.

2.3. Algoritmus Módosított eldöntés programozási tétel

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai** (*tulajdonság*)
Kimenet: *van* – **logikai**

```
1: függvény ELDÖNTÉS_MINDEN(x : T tömb, n : egész, P : logikai)
2:     i ← 1
3:     ciklus amíg (i ≤ n) ∧ P(x[i])
4:         i ← i + 1
5:     ciklus vége
6:     van ← (i > n)
7:     vissza van
8: függvény vége
```

Felhasznált változók és függvények

- x: Vizsgált tömb.
- n: A tömb mérete.
- P: Tulajdonság függvény, amely minden **T** típusú érték esetén ***igaz*** vagy ***hamis*** értéket ad vissza. Az algoritmus azt vizsgálja, hogy x minden eleme P tulajdonságú-e.
- *van*: Logikai változó, amely pontosan akkor ***igaz***, ha minden x-beli elem P tulajdonságú.

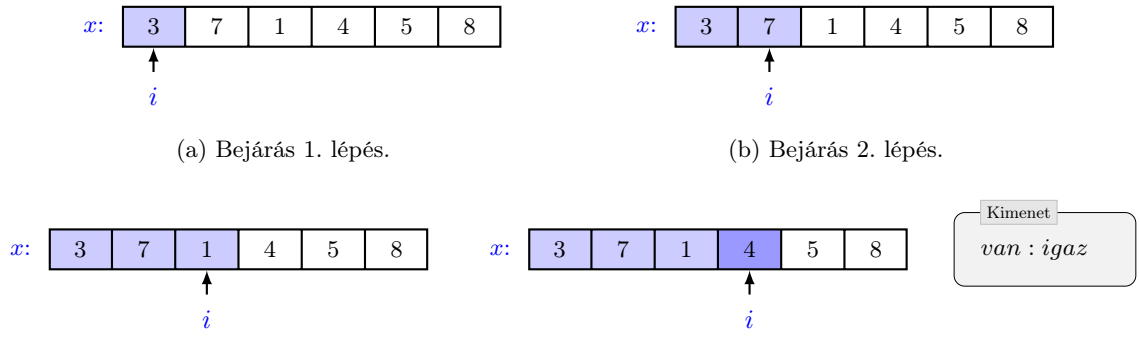
Futási idő elemzése. A futási idő szempontjából azt érdemes vizsgálni, hogy hányszor kell a ciklus feltételét kiértékelni. Ez természetesen függ attól, hogy a tömb elemei milyenek, melyik az első olyan – ha van egyáltalán –, amely teljesíti az P tulajdonságot.

Abban az esetben, ha nincs vizsgált tulajdonságú elem a tömbben, akkor a ciklus feltételt $(n+1)$-szer értékeljük ki, illetve minden egyes tömbelem egyszer lesz vizsgálva.

Ha van P tulajdonságú elem a tömbben, akkor a futási idő attól függ, hogy a legkisebb indexű ilyen elem hol helyezkedik el a tömbben. Ha például már az első elem teljesíti a P tulajdonságot, akkor a ciklusfeltétel csak egyszer kerül kiértékelésre. Ha viszont az n-edik elem az első ilyen, akkor n darab kiértékelés szükséges.

Összességében kijelenthető, hogy az algoritmus ***igaz*** visszatérési értéke esetén átlagosan $\frac{n}{2}$-del arányos a futási idő, míg ***hamis*** visszatérési érték esetén n-nel. Az algoritmus futási ideje így $O(n)$-es. ♣

2.2. Példa. Feladatunk, hogy egy hat elemű, egész számokat tartalmazó tömb esetén eldöntsük, van-e páros szám a tömbben. A feladat eldöntés tétellel történő megoldását a 2.2. ábra szemlélteti. Először megvizsgáljuk a tömb első elemét, hogy páros-e (2.2a. ábra), mivel nem az, ezért továbblépünk a soron következő tömbelemre. A második és harmadik elemet is megvizsgáljuk (2.2b-2.2c. ábra), de ezek sem páros számok. A negyedik elem viszont páros, így további vizsgálat nem szükséges, az algoritmus ***igaz*** értékkel tér vissza (2.2d. ábra). ¶



(a) Bejárás 1. lépés.

(b) Bejárás 2. lépés.

(c) Bejárás 3. lépés.

(d) Bejárás 4. lépés.

2.2. ábra. Eldöntés programozási tétel. A példa azt vizsgálja, hogy van-e páros értékű elem a tömbben.

2.3. Példa. Az eldöntés tétel logikája olyan problémák megoldása esetén is használható, amikor nem egy tömbről kell eldöntetünk, hogy van-e vizsgált tulajdonságú eleme. Példaként tekintsük azt a feladatot, amikor egy pozitív egész számról (N) meg kell vizsgálnunk, hogy prím[3] vagy nem prím. Ennek eldöntése érdekében meg kell vizsgálnunk, hogy van-e kettő és $\sqrt{N}$ között olyan egész érték, amely osztója N-nek. Ha van ilyen, akkor a szám nem lehet prím, mert osztója az 1, a megtalált érték és maga az N szám is, így már több mint két osztója van. A prím tesztelést a 2.4. algoritmussal valósíthatjuk meg. ¶

2.4. Példa. Az eldöntés tétel kis módosítással olyan kérdések megválaszolására is alkalmas, amelyek egyszerre egy tömb több elemét is vizsgálják. Tekintsük azt a feladatot, amikor el kell döntenünk, hogy egy tömb elemei növekvő módon rendezettek-e, azaz

$$x[1] \leq x[2] \leq \ldots \leq x[n]. \tag{2.1}$$

Ennek eldöntése érdekében meg kell vizsgálnunk minden szomszédos elempárt, hogy a nagyobb indexű elem nem kisebb-e a kisebb indexű elemnél, azaz

$$x[i] \leq x[i+1]. \tag{2.2}$$

Ez az összehasonlítás valósítja meg a tulajdonság vizsgálatot A vizsgálatot természetesen csak $1 \leq i \leq n-1$ esetén végezhetjük el, mert $i = n$ esetén már a tömbön kívülre is indexelnénk.

A rendezettség vizsgálatot a 2.5. algoritmussal valósítjuk meg, melynek lefutását egy konkrét példán a 2.3. ábra szemlélteti. ¶

[3]Prím számok azok a pozitív egész számok, melyeknek pontosan kettő különböző osztójuk van.

2.4. Algoritmus Prím teszt

Bemenet: N – **egész** ($N \geq 2$)
Kimenet: *prím* – **logikai**

```
1: függvény PrímTeszt(N : egész)
2:     i ← 2
3:     ciklus amíg (i ≤ √N) ∧ ¬Osztója(i, N)
4:         i ← i + 1
5:     ciklus vége
6:     prím ← (i > √N)
7:     vissza prím
8: függvény vége
```

Felhasznált változók és függvények

- N: 2-nél nem kisebb pozitív egész szám, melynek a prím tulajdonságát vizsgáljuk.
- *prím*: Logikai típusú kimeneti változó, amely pontosan akkor lesz ***igaz***, ha N prím szám.
- Osztója(i, N): Logikai értéket visszaadó függvény, amely pontosan akkor ***igaz***, ha i osztója N-nek. A függvény megvalósítása többféleképpen is történhet, ennek átgondolását az olvasóra bízzuk.

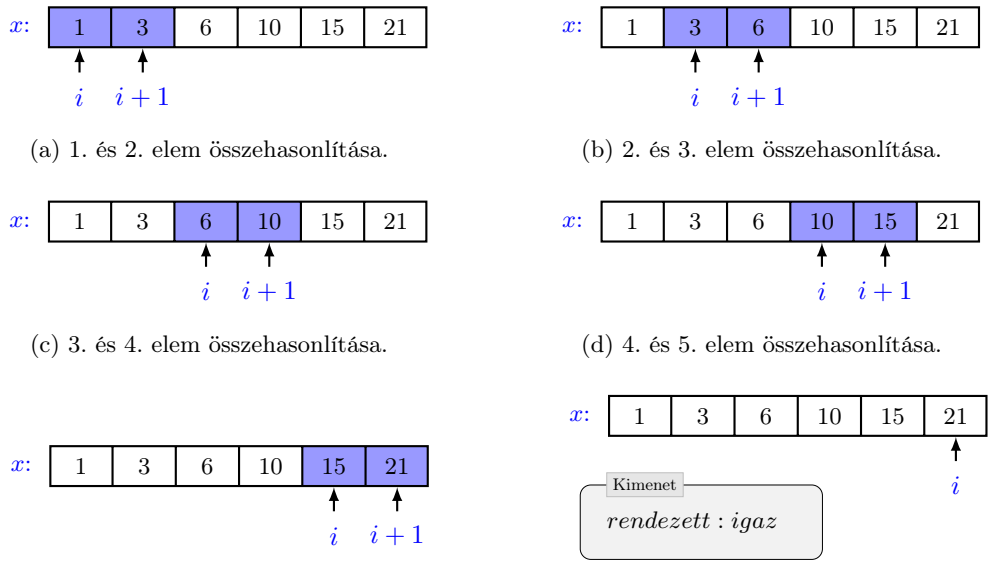
2.5. Algoritmus Növekvő rendezettség vizsgálata

Bemenet: x – **T tömb**, n – **egész**; ahol **T** összehasonlítható
Kimenet: *rendezett* – **logikai**

```
1: függvény Rendezett_E(x : T tömb, n : egész)
2:     i ← 1
3:     ciklus amíg (i ≤ n − 1) ∧ (x[i] ≤ x[i + 1])
4:         i ← i + 1
5:     ciklus vége
6:     rendezett ← (i > n − 1)
7:     vissza rendezett
8: függvény vége
```

Felhasznált változók és függvények

- x: A vizsgált tömb, melynek elemei összehasonlíthatóak.
- n: Az x tömb mérete.
- *rendezett*: Logikai változó, melynek értéke pontosan akkor ***igaz***, ha az x elemei növekvő módon rendezettek.



(a) 1. és 2. elem összehasonlítása.
(b) 2. és 3. elem összehasonlítása.
(c) 3. és 4. elem összehasonlítása.
(d) 4. és 5. elem összehasonlítása.
(e) 5. és 6. elem összehasonlítása.
(f) $i = 6$, a tömb rendezett.

2.3. ábra. Tömb rendezettségének vizsgálata az eldöntés programozási tétellel.

Kiválasztás programozási tétel

A kiválasztás[4] programozási tételt olyan feladatok esetén használjuk, amikor tudjuk, hogy egy tömbben van vizsgált tulajdonságú elem és keressük ennek első előfordulását. Könnyen belátható, hogy a megfelelő algoritmus nagyon hasonlít az eldöntés tétel megvalósítására. Különbség csak abban van, hogy nem a tulajdonság legalább egyszeri teljesülését kell vizsgálnunk, hanem az első előfordulás indexét kell megadnunk, tudva, hogy legalább egy elem esetén biztosan teljesül a tulajdonság.

A tétel megvalósítását a 2.6. algoritmus adja meg, amely nagyon hasonló az eldöntés tétel 2.2. algoritmusához. Három különbség van csak. A ciklus belépési, illetve bennmaradási feltételénél (3. sor) nem kell vizsgálnunk, hogy a tömb indexelésére használt i változó meghaladja-e már a tömb méretét, mivel ez csak akkor állhatna elő, ha nem lenne a tömbben P tulajdonságú elem. A ciklusból akkor lépünk ki, ha megtaláltuk azt az i indexet, amelynél először teljesül az P tulajdonság. A ciklust követően ezért az idx kimeneti változó megkapja az aktuális i értéket (6. sor), amit egyben a függvény visszatérési értéke is (7. sor).

2.6. Algoritmus Kiválasztás programozási tétel

Bemenet: x – **T tömb**, n – **egész**, P – **logikai**
Kimenet: idx – **egész**

```
1: függvény KIVÁLASZTÁS(x : T tömb, n : egész, P : logikai)
2:     i ← 1
3:     ciklus amíg ¬P(x[i])
4:         i ← i + 1
5:     ciklus vége
6:     idx ← i
7:     vissza idx
8: függvény vége
```

Felhasznált változók és függvények

- x: A vizsgált tömb.
- n: Az x tömb mérete.
- P: Tulajdonságfüggvény, amely minden **T** típusú értéke esetén ***igaz*** vagy ***hamis*** értéket ad vissza. Az algoritmus az első P tulajdonságú elem indexét határozza meg.
- idx: Az első P tulajdonságú tömbelem indexe.

Futási idő elemzése. Mivel a kiválasztás programozási tétel az eldöntés tétel egyszerű módosítása, ezért a futási időről is hasonlót mondhatunk, mint az eldöntés tételnél tettük (ld. a 24. oldalon). Azonban a kiválasztás tétel esetén nem kell figyelembe vennünk azt az esetet, amikor nincs a tömbben P tulajdonságú elem, így kijelenthető, hogy az átlagos futási idő $\frac{n}{2}$-del arányos, azaz ez az algoritmus is $O(n)$-es. ♣

[4] Angolul: selection

Lineáris keresés programozási tétel

A lineáris keresés[5] programozási tételt olyan feladatok megoldása esetén használjuk, amikor meg szeretnénk tudni, hogy egy tömbben van-e valamilyen tulajdonságú elem, és ha van, akkor hol található az első ilyen. A feladatot meg lehetne oldani úgy, hogy először egy eldöntéssel megvizsgáljuk a P tulajdonság teljesülését, majd ha szükséges, akkor egy kiválasztás tétellel megkeressük a tömb első P tulajdonságú elemét. Ha így járnánk el, akkor a tömböt kétszer is bejárnánk a keresett elemig, ami teljesen felesleges, mert egy bejárással is megvalósítható a feladat megoldása. Ehhez is csak az eldöntés tétel 2.2. algoritmusának módosítása szükséges.

Változtatni kell egyrészt a kimeneteket, mert P tulajdonságú elem létezése esetén meg kell adnunk azt is, hogy van ilyen elem, valamint az első ilyen elem indexét is. Másrészt a helyes működés érdekében kis mértékben az algoritmust is módosítanunk kell úgy, hogy *van* ***igaz*** értéke esetén visszaadjuk az aktuális indexet is. A tételt a 2.7. algoritmus valósítja meg.

Az algoritmus a 6. sorig teljes egészében megegyezik az eldöntés tételnél már megismertekkel. Ezt követően határozzuk meg annak függvényében, hogy találtunk-e P tulajdonság elemet a tömbben, hogy csak a *van* értékét, vagy emellett a megfelelő indexet (idx) is visszaadjuk a külvilágnak.

> **Megjegyzés**
>
> Pszeudokódban lehetséges, hogy a visszatérési értékek száma más és más, viszont konkrét programozási nyelven általában egy függvénynek csak egy visszatérési értéke lehet. Ilyen esetben úgy is megvalósíthatjuk a lineáris keresés, hogy mindig pontosan egy visszatérési értéke lesz. Ha a keresett értéket megtaláltuk a tömbben, akkor a találat helyét, azaz az idx változót adjuk vissza. Ha viszont nincs benne a keresett érték a tömbben, akkor egy nem valós indexet, például a 0-t adjuk vissza kimenetként.

2.7. Algoritmus Lineáris keresés programozási tétel

Bemenet: x – **T tömb**, n – **egész**, P – **logikai**
Kimenet: van – **logikai**, idx – **egész**

```
1:  függvény LineárisKeresés(x : T tömb, n : egész, P : logikai)
2:      i ← 1
3:      ciklus amíg (i ≤ n) ∧ ¬P(x[i])
4:          i ← i + 1
5:      ciklus vége
6:      van ← (i ≤ n)
7:      ha van akkor
8:          idx ← i
9:          vissza (van, idx)
10:     különben
11:         vissza van
12:     elágazás vége
13: függvény vége
```

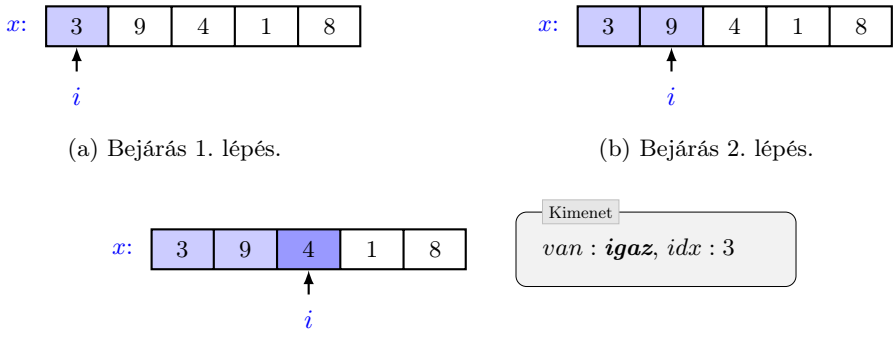
Felhasznált változók és függvények

- x: A vizsgált tömb.
- n: Az x elemszáma.
- P: Logikai értéket visszaadó tulajdonság függvény. Az algoritmusban azt vizsgáljuk, hogy az x tömbnek van-e P tulajdonságú eleme.
- *van*: Logikai érték. Pontosan akkor ***igaz*** az értéke, ha van olyan elem az x tömbben, mely esetén teljesül a P tulajdonság.
- idx: Csak akkor értelmezzük, ha *van* ***igaz***. Ebben az esetben a legkisebb olyan egész számot adja meg, ahányadik eleme a tömbnek teljesíti a P tulajdonságot.

[5] Angolul: linear search

Futási idő elemzése. Mivel a lineáris keresés programozási tétel nagyon hasonló az eldöntés tételhez, csak egy további opcionális értékadásban tér el attól, ezért a futási idejéről is ugyanaz mondható. Így a futási idő itt is $O(n)$-es. ♣

2.5. Példa. Egy öt elemű egész értékeket tartalmazó tömbben vizsgáljuk, hogy van-e benne páros szám, és amennyiben van, akkor hol található az első ilyen. A feladat megoldását a lineáris keresés programozási tétellel a 2.4. ábra szemlélteti. Az első és második elem még nem páros, ezért azok vizsgálatát követően továbblépünk a tömbben (2.4a-2.4b. ábra). A harmadik elem viszont páros, ezért itt véget ér a keresés, az algoritmus visszatér a *van* változó ***igaz*** értékével, az *idx* változó pedig megkapja a tömbbeli aktuális helyet, azaz i aktuális értékét (2.4c. ábra). ¶



(a) Bejárás 1. lépés.

(b) Bejárás 2. lépés.

(c) Bejárás 3. lépés.

2.4. ábra. Lineáris keresés programozási tétel. Keressük, hogy van-e páros elem egy tömbben, és ha van, akkor hol található az első ilyen.

Megjegyzés

Gyakran előfordul, hogy a keresés során egy konkrét érték x tömbbeli előfordulását vizsgáljuk. Ilyenkor a figyelembe vett tulajdonság függvény az $x[i] = érték$ egyenlőség vizsgálatával helyettesíthető. Mivel az 5.1. fejezetben konkrét érték keresését vizsgáljuk majd, ezért a 2.8 algoritmusban megadjuk az érték keresés pszeudokódját is.

2.8. Algoritmus Lineáris keresés programozási tétel (konkrét érték keresése)

Bemenet: x – **T tömb**, n – **egész**, *érték* – **T**
Kimenet: van – **logikai**, idx – **egész**

```
1: függvény LineárisKeresés(x : T tömb, n : egész, érték : T)
2:     i ← 1
3:     ciklus amíg (i ≤ n) ∧ (x[i] ≠ érték)
4:         i ← i + 1
5:     ciklus vége
6:     van ← (i ≤ n)
7:     ha van akkor
8:         idx ← i
9:         vissza (van, idx)
10:    különben
11:        vissza van
12:    elágazás vége
13: függvény vége
```

Felhasznált változók és függvények

- x: A vizsgált tömb.
- n: Az x elemszáma.
- *érték*: Az algoritmusban azt vizsgáljuk, hogy az x tömbben megtalálható-e az *érték*.
- van: Logikai érték. Pontosan akkor ***igaz*** az értéke, ha van az *érték*-kel egyező elem az x tömbben.
- idx: Csak akkor értelmezzük, ha van ***igaz***. Ebben az esetben a legkisebb olyan egész számot adja meg, ahányadik eleme a tömbnek megegyezik az *érték*-kel.

Megszámlálás programozási tétel

Feladatunk, hogy egy tömbben meghatározzuk az adott tulajdonságú elemek számát. Például egy egész számokat tartalmazó tömbben a párosak darabszámát szeretnénk megkapni.

A tételt a 2.9. algoritmussal valósítjuk meg. Bemenetként a tömböt (x), annak elemszámát (n) és a tulajdonság függvényt (P) ismerjük, kimenetként pedig az adott tulajdonságú elemek számát (db) kapjuk vissza.

2.9. Algoritmus Megszámlálás programozási tétel

Bemenet: $x -$ **T tömb**, $n -$ **egész** (*tömb mérete*), $P -$ **logikai** (*tulajdonság*)
Kimenet: $db -$ **egész** (*darabszám*)

```
1: függvény MEGSZÁMLÁLÁS(x : T tömb, n : egész, P : logikai)
2:     db ← 0
3:     ciklus i ← 1-től n-ig
4:         ha P(x[i]) akkor
5:             db ← db + 1
6:         elágazás vége
7:     ciklus vége
8:     vissza db
9: függvény vége
```

Felhasznált változók és függvények

- x: Feldolgozandó tömb.
- n: Az x tömb elemszáma.
- P: Tulajdonság függvény.
- db: A P tulajdonságú elemek száma x-ben.

Első lépésként az algoritmus 2. sorában a darabszámot nullára állítjuk. Ezt követően az algoritmus 3. és 7. sorai közötti ciklussal bejárjuk a teljes tömböt, hogy minden egyes elem esetén megvizsgáljuk a P tulajdonság teljesülését (4. sor).

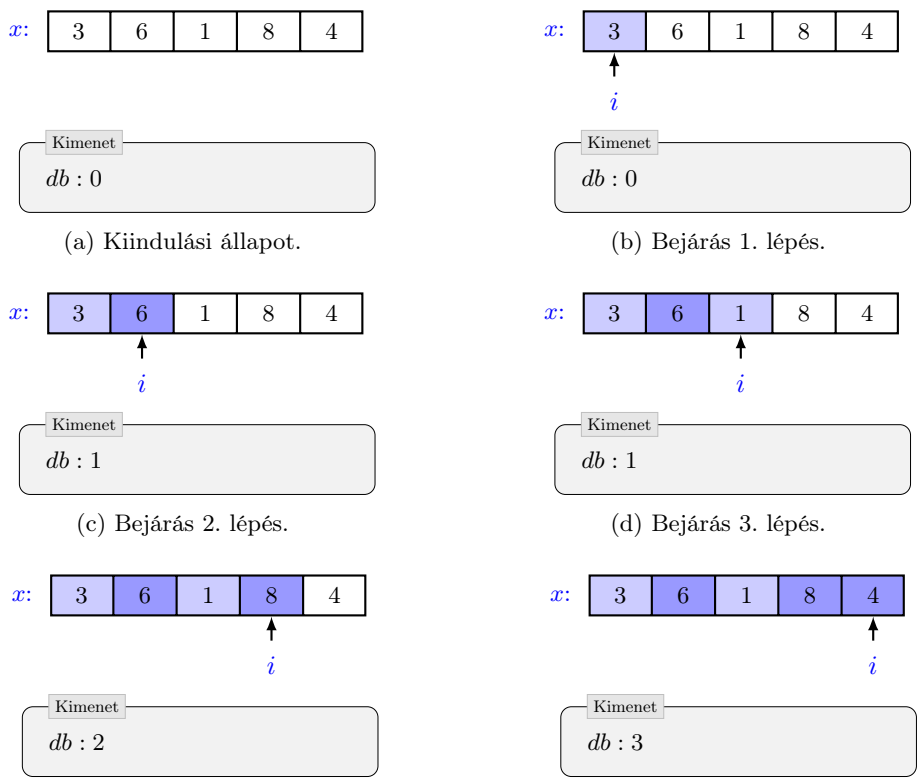
Ha az aktuális elem esetén teljesül a vizsgált tulajdonság, akkor a darabszámot 1-gyel növeljük, amint az az algoritmus 5. sorában látható, egyéb esetben pedig változatlanul hagyjuk.

Az algoritmus végén a darabszám értékével tér vissza a függvény (8. sor).

2.6. Példa. A 2.5. ábra egy 5 elemű egész számokat tartalmazó tömb esetén szemlélteti a páros számok darabszámának meghatározását. Először a db változót incializáljuk (2.5a. ábra), majd bejárjuk a tömböt és minden elemnél vizsgáljuk a tulajdonság teljesülését (2.5b-2.5f. ábra). Ha a vizsgált elem páros, akkor 1-gyel növeljük a db változót (2.5c, 2.5e és 2.5f. ábrák), ha pedig páratlan, akkor db-t változatlanul hagyjuk (2.5b és 2.5d. ábrák). ¶

Futási idő elemzése. Az algoritmus végrehajtása során bejárjuk a teljes tömböt és minden elemre megvizsgáljuk a tulajdonság teljesülését, így egy n elemű tömb esetén n darab tulajdonság vizsgálatot hajtunk végre. A db változót viszont csak abban az esetben növeljük, amikor a vizsgált tulajdonság teljesül. A P tulajdonságfüggvény kiértékelése általában több időt vesz igénybe, mint db növelése. Így a nagyobb futási idejű művelet mindig végrehajtódik, ezért végső soron a futási idő a tömb méretével arányos, azaz $O(n)$-es. ♣

> Megjegyzés
>
> Abban az esetben, ha nincs P tulajdonságú elem a tömbben, akkor a darabszám értéke 0 lesz. Így az algoritmust akár eldöntésre is lehetne használni. Ha a darabszám pozitív, akkor az eldöntés eredménye igaz, egyébként pedig hamis.
>
> Természetesen, ha csak eldöntés a feladatunk, akkor célszerűbb az eldöntés programozási tételt használni, hiszen az az első P tulajdonságú elem megtalálása esetén befejezi a tömb bejárását, míg a megszámlálás programozási tételben minden esetben bejárjuk a teljes tömböt.



(a) Kiindulási állapot.

(b) Bejárás 1. lépés.

(c) Bejárás 2. lépés.

(d) Bejárás 3. lépés.

(e) Bejárás 4. lépés.

(f) Bejárás 5. lépés.

2.5. ábra. Megszámlálás programozási tétel. Az x tömbben meghatározzuk a páros számok darabszámát.

Maximumkiválasztás programozási tétel

A maximumkiválasztás programozási tétel megadja, hogy egy tömbben melyik az az elem, amelynek a legnagyobb az értéke. Természetesen ez a probléma csak olyan esetben merülhet fel, amikor a tömbben egymással összehasonlítható elemeket tárolunk, azaz olyanokat, amelyek között mindig egyértelműen megadható a köztük lévő kisebb-nagyobb viszony.

A feladatot úgy oldjuk meg, hogy az algoritmus elején feltételezzük, hogy a tömb első eleme a legnagyobb. (Ez a feltételezés azt is magában foglalja, hogy nem lehet a tömb elem nélküli, mert akkor már első elemről sem beszélhetünk.) Ezt követően minden elemet megvizsgálunk, hogy az értéke nagyobb-e az őt megelőző elemek maximumánál, és ha igen, akkor ezentúl azt tekintjük a maximális értékű elemnek. A tételt a 2.10. algoritmus írja le.

A 2. sorban a max változó megkapja a kezdeti értékét. A 3. és 7. sorok közötti ciklus biztosítja a tömb bejárását. Vegyük észre, hogy a bejárást a második elemnél kezdjük. A 4. sorban található elágazás feltételében megvizsgáljuk, hogy az aktuális $x[i]$ tömbelem nagyobb-e a már korábban megvizsgált elemek legnagyobbikánál ($x[max]$). Ha nagyobb, akkor új maximumot találtunk, ezért az 5. sorban max új értéket kap, mégpedig az aktuális indexet. A ciklusból kilépve minden elemet megvizsgáltunk már, így a max változó a legnagyobb értékű elem indexét tartalmazza, és ezt adjuk vissza kimenetként a külvilágnak (8. sor).

2.10. Algoritmus Maximumkiválasztás programozási tétel.

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: max – **egész**

```
1: függvény MAXIMUMKIVÁLASZTÁS(x : T tömb, n : egész)
2:     max ← 1
3:     ciklus i ← 2-től n-ig
4:         ha x[i] > x[max] akkor
5:             max ← i
6:         elágazás vége
7:     ciklus vége
8:     vissza max
9: függvény vége
```

Felhasznált változók és függvények

- x: Legalább 1 elemű tömb, melyben a legnagyobb elemet keressük. Az x tömb elemeinek összehasonlíthatónak kell lennie.
- n: Az x tömb mérete
- max: A x legnagyobb értékű elemének indexe. (Ha többször is előfordul a legnagyobb érték, akkor max a legkisebb indexű ilyen elem indexe lesz.)

Megjegyzés

Az algoritmus a legnagyobb értékű tömbbeli elem indexét adja kimenetként, nem pedig a legnagyobb elem értékét. Ennek az az oka, hogy az max index (és az x eredeti tömb) ismeretében a maximális érték bármikor lekérdezhető, viszont ha a maximális értéket ismernénk, akkor csak kiválasztás tétellel tudnánk azt meghatározni, hol is található az az elem a tömbben.

Megjegyzés

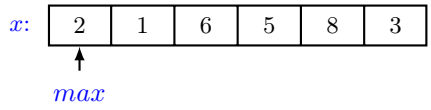
Ha a tömbben többször is előfordul a maximális elem, akkor az algoritmus a legkisebb indexű helyet fogja visszaadni. Ha a 4. sorban található feltételben $\geq$-re módosítanánk a relációt, akkor a visszaadott index, az előforduló maximumok közül a legnagyobb indexű lenne.

Megjegyzés

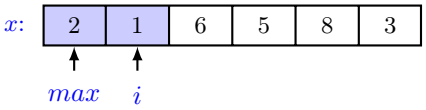
A tétel kis átalakítással minimum érték (illetve ahhoz tartozó index) kiválasztására is alkalmassá tehető. Ehhez szintén a 4. sorban található feltételt kell módosítanunk a reláció <-re cserélésével.

2.7. Példa. Adott egy hat elemű egész számokat tartalmazó tömb. Adjuk meg, hogy melyik elem a legnagyobb értékű. A feladat maximumkiválasztás programozási tétellel történő megoldását a 2.6. ábra szemlélteti.

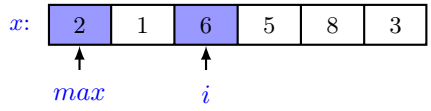
Első lépésben feltesszük, hogy az első elem a legnagyobb, ezért a *max* változó az első elemet indexeli (2.6a. ábra). Ezt követően megvizsgáljuk, hogy a második elem nagyobb-e az elsőnél, és mivel nem nagyobb, ezért *max* értékét változatlanul hagyjuk (2.6b. ábra). A következő lépésben a tömb harmadik elemét vetjük össze az első elemmel (2.6c. ábra). Mivel a harmadik elem nagyobb az elsőnél, ezért a *max* változó értékét módosítjuk, innentől kezdve már a harmadik elemet indexeljük vele. A soron következő lépésekben hasonló módon a negyedik, ötödik és hatodik elemet hasonlítjuk össze a tömb *max* indexű elemével, és ha az aktuális tömbelem nagyobb $x[max]$-nál, akkor *max* értékét megváltoztatjuk (2.6d-2.6f. ábra). Az összes elem vizsgálatát követően *max* értéke 5, azaz az ötödik elem a legnagyobb a tömbben, ezért ezt az értéket adja vissza az algoritmus. ¶



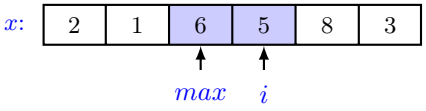
(a) A *max* változónak kezdeti érték adása.



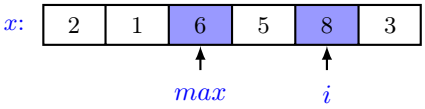
(b) A 2. elem vizsgálata.



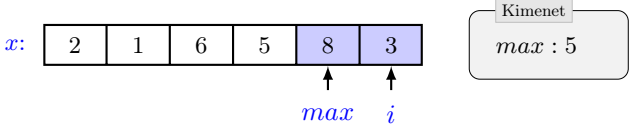
(c) A 3. elem vizsgálata.



(d) A 4. elem vizsgálata.



(e) Az 5. elem vizsgálata.



(f) A 6. elem vizsgálata és a kimenet meghatározása.

2.6. ábra. Maximumkiválasztás programozási tétel. Maximális értékű elem indexének keresése egy hat elemű egészeket tartalmazó tömbben.

Futási idő elemzése. A maximumkiválasztás programozási tétel pszeudokódjából (2.10. algoritmus) látható, hogy egy n elemű tömb feldolgozása során minden esetben $n-1$ darab összehasonlítást végzünk. A cikluson belüli értékadások száma függ attól, hogy a vizsgált feltétel teljesül-e vagy sem. Futási idő szempontjából legrosszabb esetben $(n-1)$-szer hívódik meg az említett értékadó utasítás. Látható hogy a futási idő n-nel arányos, így az algoritmus $O(n)$-es. ♣

Összetett programozási tételek

Másolás programozási tétel

A másolás[6] programozási tételt olyan esetben használjuk, amikor egy tömb minden elemét szeretnénk egy másik tömbbe átmásolni, vagy egy tömb minden elemének egy adott függvény által módosított értékét szeretnénk egy másik tömbbe másolni.

A tétel megvalósítása nagyon egyszerű, ahogy az a 2.11. algoritmusban is látható. Egy számlálós ciklussal végigjárjuk az x tömb minden elemét és az elemek f függvény által módosított értékeit átmásoljuk az y tömb megfelelő helyére.

2.11. Algoritmus Másolás programozási tétel

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), f – **művelet**
Kimenet: y – **T tömb**

```
1: függvény MÁSOLÁS(x : T tömb, n : egész, f : művelet)
2:     y ← LÉTREHOZ(T)[n]
3:     ciklus i ← 1-től n-ig
4:         y[i] ← f (x[i])
5:     ciklus vége
6:     vissza y
7: függvény vége
```

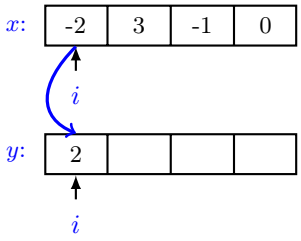
Felhasznált változók és függvények

- x: A feldolgozandó tömb.
- n: Az x tömb mérete.
- f: Függvény, amely az x elemein értelmezett.
- y: Kimeneti, n elemű tömb. Minden egyes eleme csak az x tömb megfelelő elemétől függ, az $y[i] = f(x[i])$ szabály szerint.
- LÉTREHOZ(**T**)[n]: Utasítás, mely létrehoz egy n elemű **T** típusú tömböt.

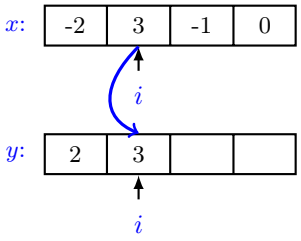
2.8. Példa. Tekintsünk egy négy elemű tömböt. Másoljuk át a tömb egyes elemeinek abszolút értékét egy másik tömbbe. A feladat megoldását a 2.7. ábra szemlélteti. ¶

Futási idő elemzése. Mivel minden esetben a tömb összes elemét egyszer másoljuk át az új tömbbe, ezért az algoritmus futási ideje $O(n)$-es. ♣

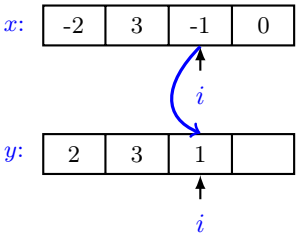
[6] Angolul: copy



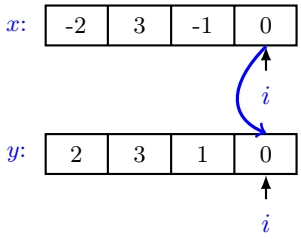
(a) Bejárás 1. lépés.



(b) Bejárás 2. lépés.



(c) Bejárás 3. lépés.



(d) Bejárás 4. lépés.

2.7. ábra. Másolás programozási tétel. Az x tömb minden egyes elemének abszolút értékét átmásolja az y tömb megfelelő helyére.

Kiválogatás programozási tétel

A kiválogatás programozási tételt olyan feladatok esetén használjuk, amikor egy tömb adott tulajdonságú elemeit szeretnénk kigyűjteni, kiválogatni. Például egy egész számokat tartalmazó tömbből ki akarjuk válogatni azokat, amelyek párosak.

A tétel megoldására olyan algoritmust adunk, amely bejárja a bemeneti tömböt és egy másik tömbbe kigyűjti az adott tulajdonságú elemeket. A megoldást a 2.12. algoritmusban írjuk le. Az algoritmus bemenete a feldolgozandó x tömb, melynek ismerjük a méretét, illetve a P tulajdonságfüggvény. Az x tömb P tulajdonságú elemeit egy új y tömbbe gyűjti ki az algoritmus, ez a tömb lesz az egyik visszatérési érték is. Az algoritmus elején még nem ismerjük az y tömb méretét, ezért annyi helyet kell lefoglalni számára, ami a lehetséges maximális mérete. Ez a méret megegyezik az x méretével, hiszen ha x minden eleme P tulajdonságú, akkor minden elemet kiválogatunk y-ba. Az algoritmusnak még egy kimenetet kell szolgáltatnia, ugyanis meg kell határoznia, hogy hány elemet válogattunk ki az y tömbbe. Ezt *db*-vel jelöljük és természetesen megegyezik az x tömb P tulajdonságú elemeinek számával.

> Megjegyzés
>
> Emlékeztetőként jelezzük, hogy a *db* értéket határozza meg a megszámlálás tétel is. Könnyen észrevehető, hogy a kiválogatás 2.12. algoritmusa a megszámlálás tétel 2.9. algoritmusának továbbfejlesztése.

Az algoritmus elején létrehozzuk a kimeneti y tömböt n elemmel (2. sor). Ezt követően nullára állítjuk a *db* változó értékét (3. sor), hiszen kezdetben egyetlen P tulajdonságú elemet sem találtunk még az x tömbben. A 4. és 9. sorok közötti ciklussal bejárjuk az x tömböt. Megvizsgáljuk, hogy a tömb aktuális eleme teljesíti-e a P tulajdonságot (5. sor). Ha igen, akkor növeljük a *db* változó értékét (6. sor) és az y tömb *db*-edik elemébe átmásoljuk az x tömb aktuális elemét (7. sor). A ciklus befejezését követően *db* értéke megegyezik az x tömb P tulajdonságú elemeinek számával, az y tömbbe pedig kiválogattuk az x tömb P tulajdonságú elemeit, így már csak a kimeneteket kell visszaadni az algoritmusnak (10. sor).

2.12. Algoritmus Kiválogatás programozási tétel

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai**
Kimenet: y – **T tömb**, *db* – **egész**

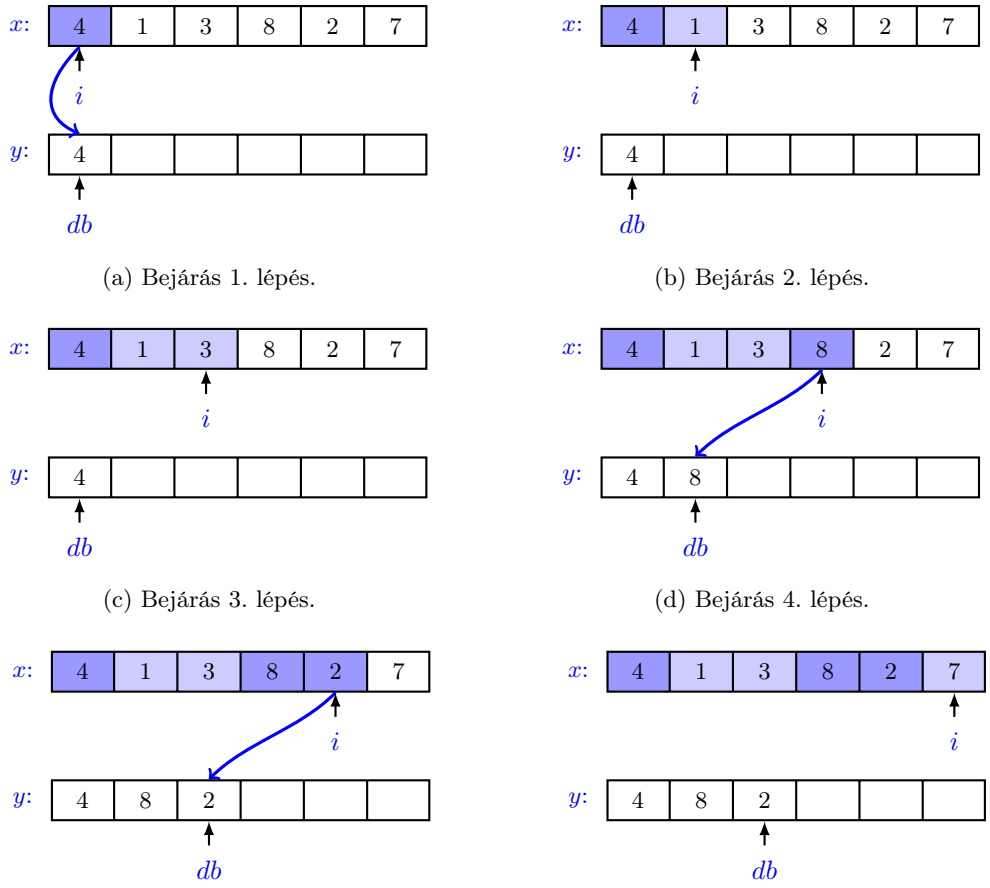
```
1:  függvény KIVÁLOGATÁS(x : T tömb, n : egész, P : logikai)
2:      y ← LÉTREHOZ(T)[n]
3:      db ← 0
4:      ciklus i ← 1-től n-ig
5:          ha P(x[i]) akkor
6:              db ← db + 1
7:              y[db] ← x[i]
8:          elágazás vége
9:      ciklus vége
10:     vissza (y, db)
11: függvény vége
```

Felhasznált változók és függvények

- x: A vizsgált tömb.
- n: Az x tömb mérete.
- P: Logikai értékű tulajdonságfüggvény. Az x tömb P tulajdonságú elemeit válogatja ki az algoritmus az y tömbbe.
- y: Az n elemű kimeneti tömb, melynek elemei az x azon elemei, melyek teljesítik a P tulajdonságot.
- *db*: Az y tömb releváns elemeinek száma.
- LÉTREHOZ(**T**)[n]: Utasítás, mely létrehoz egy n elemű **T** típusú tömböt.

2.9. Példa. Feladatunk, hogy egy hat elemű egész számokat tartalmazó tömbből kiválogassuk a páros számokat egy új tömbbe. A feladat megoldását a 2.8. ábrán láthatjuk. A kiválogatás tétel alkalmazásával végigjárjuk az x tömb elemeit. Az első elem páros szám, ezért a *db* változó értékét egyre változtatjuk és az y tömb első helyére bemásoljuk az x tömb első elemét (2.8a. ábra). A soron következő két elem

nem páros, ezért azokat nem másoljuk át az y tömbbe (2.8b-2.8c. ábrák). Az x tömb negyedik és ötödik eleme páros, ezért db növelését követően ezeket másoljuk y-ba (2.8d-2.8e. ábrák). Mivel x utolsó eleme nem páros, ezért nem történik sem db növelés, sem másolás (2.8f. ábra). ¶



(a) Bejárás 1. lépés.

(b) Bejárás 2. lépés.

(c) Bejárás 3. lépés.

(d) Bejárás 4. lépés.

(e) Bejárás 5. lépés.

(f) Bejárás 6. lépés.

2.8. ábra. Kiválogatás programozási tétel. Egy hat elemű tömbből kiválogatjuk a páros számokat.

Futási idő elemzése. Az algoritmus bejárja az x tömböt és minden elem esetén megvizsgálja, hogy teljesíti-e a P tulajdonságot. Ez pontosan n darab tulajdonságvizsgálatot jelen. A db változó növelése és az értékmásolás az y tömbbe csak abban az esetben történik meg, ha $x[i]$ P tulajdonságú volt. Ezek minimális száma 0, maximális pedig n. Összességében kijelenthető, hogy a futási idő a feldolgozandó tömb méretével egyenesen arányos, tehát az algoritmus $O(n)$-es. ♣

Abban az esetben, ha a kiválogatást követően már nincs szükségünk az eredeti tömbre, akkor a kiválogatást úgy is elvégezhetjük, hogy az eredeti tömb elejére gyűjtjük ki a P tulajdonságú elemeket. Ennek megvalósítását a 2.13. algoritmus adja meg. Az algoritmus bemenete megegyezik a 2.12. algoritmusnál ismertetettekkel. Kimenetként viszont nem egy új tömböt adunk vissza, hanem az eredeti x tömb módosított változatát, valamint a db számlálót. A kimeneti x tömb első db darab elemei azok, amelyek az eredeti x tömbben teljesítették a P tulajdonságot. Természetesen a tömbben bennmaradnak további elemek (a $db + 1$ és n indexek közötti rész), de ezek nem hordoznak semmilyen releváns információt.

A lényeges eltérések a 2.12. algoritmushoz képest a következők. Nem kell helyet foglalnunk az új y-nak, hiszen nincs szükség külön kimeneti tömbre. Amikor az x tömb bejárása során találunk egy P tulajdonságú elemet, akkor azt az x tömb elejére, a db indexű helyre másoljuk át, ahogy ez a 2.13. algoritmus 6. sorában látható. Az x tömb feldolgozását követően csak a db változó értékét adjuk vissza (9. sor), az x tömb módosított változata pedig a cím szerinti paraméterátadás miatt lesz elérhető az algoritmust futtató környezetben.

2.13. Algoritmus Kiválogatás programozási tétel az eredeti tömbben

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai**
Kimenet: x – **T tömb**, db – **egész**

```
1: függvény KiválogatásHelyben(címszerint x : T tömb, n : egész, P : logikai)
2:     db ← 0
3:     ciklus i ← 1-től n-ig
4:         ha P(x[i]) akkor
5:             db ← db + 1
6:             x[db] ← x[i]
7:         elágazás vége
8:     ciklus vége
9:     vissza db
10: függvény vége
```

Felhasznált változók és függvények

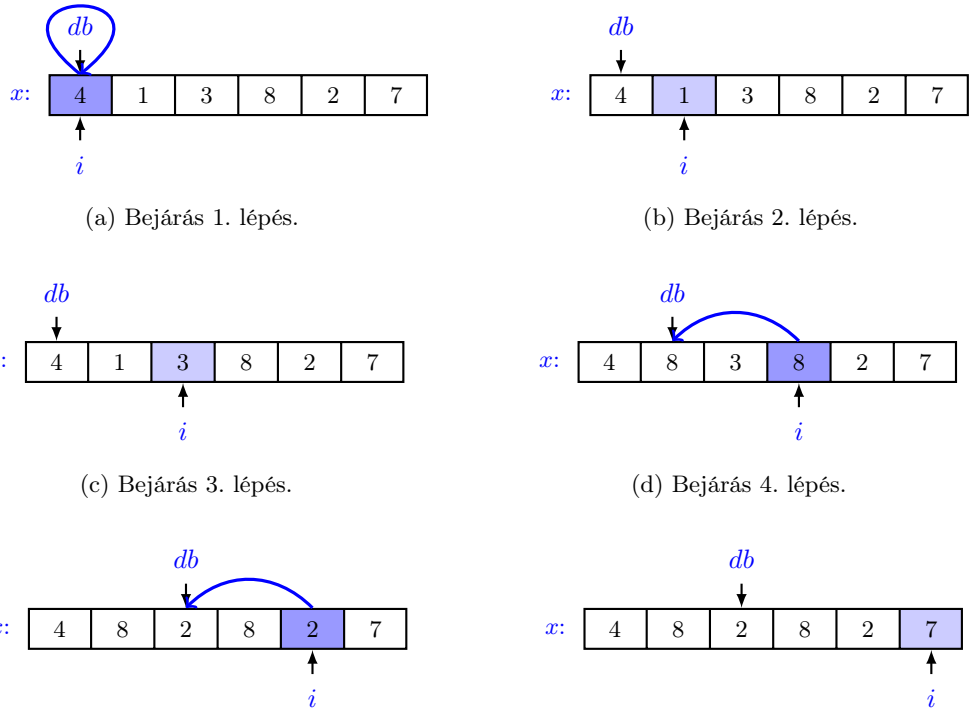
- x: A vizsgált tömb.
- n: Az x tömb mérete.
- P: Logikai értékű tulajdonság függvény. Az x tömb P tulajdonságú elemeit válogatja ki az algoritmus az x tömb elejére.
- db: A kiválogatást követően az x tömbben az első és a db-edik elemek között vannak a releváns elemek, azaz db a releváns elemek száma.

2.10. Példa. Oldjuk meg a 2.9. példában ismertetett feladatot úgy, hogy a kiválogatás eredményét az eredeti x tömbben tároljuk el. A megoldás egyes lépéseit a 2.9. ábrán láthatjuk. Először vizsgáljuk a tömb első elemét (2.9a. ábra), amelyik páros, így 1-re növeljük a db változó értékét és a tömb első elemét helyben hagyjuk. A második elem páratlan, így nem teszünk vele semmit (2.9b. ábra). Ugyanez történik a harmadik elem vizsgálatánál (2.9c. ábra). A negyedik elem páros, ezért 2-re növeljük db értékét és a második helyre átmásoljuk a negyedik elemet (2.9d. ábra). Ebben a lépésben elveszítjük a tömb korábbi második elemét. Hasonló történik az ötödik elem vizsgálatakor (2.9e. ábra). Az utolsó elem páratlan ezért nem teszünk semmit, az algoritmus futása véget ér. Látható, hogy az előálló tömb első három eleme mind páros, a további elemek pedig csak ottmaradt értékek a korábbi tömbből, amelyek releváns információt a páros-páratlanságról nem hordoznak. ¶

Megjegyzés

A 2.10. példában láthattuk, hogy a helyben történő kiválogatás esetén elemeket veszítünk az eredeti tömbből. Ha az algoritmusunkat úgy módosítanánk, hogy P tulajdonságú elem megtalálása esetén kicserélnénk két elemet, akkor nem vesztenénk el elemeket az eredeti tömbből, csak az elemek sorrendje változna meg. Így elérhetjük, hogy a tömbünk elejére kerülnek a P tulajdonságú elemek, a végére pedig a nem P tulajdonságúak. Ez egyébként a 2.2.3. alfejezetben tárgyalt szétválogatás tétel feladata.

A 2.14. algoritmusban megadjuk az elemek cserélésével végrehajtott szétválogatást. Az említett csere az algoritmus 6. sorában történik meg. A későbbiekben egy ennél hatékonyabb algoritmust is bemutatunk majd a helyben történő szétválogatásra.



(a) Bejárás 1. lépés.

(b) Bejárás 2. lépés.

(c) Bejárás 3. lépés.

(d) Bejárás 4. lépés.

(e) Bejárás 5. lépés.

(f) Bejárás 6. lépés.

2.9. ábra. Kiválogatás az eredeti tömbben. Egy hat elemű egészeket tartalmazó tömb elejére gyűjtjük ki a páros elemeket. Az algoritmus végén csak az első *db* darab elem hordoz releváns információt.

2.14. Algoritmus Kiválogatás programozási tétel az eredeti tömbben az eredeti elemek megtartásával

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai**
Kimenet: x – **T tömb**, db – **egész**

```
1:  függvény SZÉTVÁLOGATÁS(címszerint x : T tömb, n : egész, P : logikai)
2:      db ← 0
3:      ciklus i ← 1-től n-ig
4:          ha P(x[i]) akkor
5:              db ← db + 1
6:              x[db] ↔ x[i]
7:          elágazás vége
8:      ciklus vége
9:      vissza db
10: függvény vége
```

Felhasznált változók és függvények

- x: A vizsgált tömb.
- n: Az x tömb mérete.
- P: Logikai értékű tulajdonságfüggvény. Az x tömb P tulajdonságú elemeit válogatja szét az algoritmus az x tömb elejére. A P tulajdonságú elemek a tömb elejére, a nem P tulajdonságúak pedig a végére kerülnek.
- db: A szétválogatást követően az x tömbben az első és db-edik elemek között vannak a P tulajdonságúak, a $db + 1$ és n indexek közötti elemek pedig a nem P tulajdonságúak.

Szétválogatás programozási tétel

A szétválogatás programozási tétel a kiválasztás tételhez hasonló feladat megvalósítására alkalmas. Ebben az esetben is adott egy tömb, valamint egy tulajdonság. Viszont a bemeneti tömb elemeit most esetben úgy szeretnénk szétválogatni, hogy egy tömbbe kerüljenek a megadott tulajdonságú elemek, egy másik tömbbe pedig a nem olyan tulajdonságúak. Azaz a szétválogatás tétel használata esetén minden elem bekerül valamely kimeneti tömbbe, így az elemeket tényleg szétválogatjuk a vizsgált tulajdonságnak megfelelően.

A tétel megvalósítását a 2.15. algoritmus mutatja be. Az algoritmus bemenete az x tömb, melynek ismert az n elemszáma, valamint egy logikai értéket visszaadó P tulajdonságfüggvény. Az algoritmus az y_1 tömbbe kiválogatja az x tömb P tulajdonságú elemeit, az y_2 tömbbe pedig a nem P tulajdonságúakat. A két kimeneti tömbön kívül azt is meg kell adnia az algoritmusnak, hogy melyik kimeneti tömbbe hány elem kerül, ezeket jelölik rendre a db_1 és db_2 változók.

Az algoritmus első lépéseként helyet kell foglalni a memóriában az y_1 és y_2 tömbök számára. Mivel nem tudjuk, hogy melyik tömbbe hány elem fog kerülni – hiszen nem ismert még, hogy a bemeneti x tömb hány P tulajdonságú elemet tartalmaz –, ezért mindkét tömbnek a lehetséges legnagyobb méretet foglaljuk le, ami megegyezik az x tömb n méretével (2. és 3. sor). Ezt követően nullára állítjuk a db_1 és db_2 változókat, hiszen még egyetlen elemet sem másoltunk át a kimeneti tömbökbe (4. és 5. sor).

A kimenetek inicializálása után a 6. és a 14. sorok közötti ciklussal végigjárjuk az x tömböt. Megvizsgáljuk, hogy a x tömb aktuális eleme P tulajdonságú-e (7. sor). Amennyien $x[i]$ teljesíti a vizsgált tulajdonságot, akkor az y_1 tömbbe kell bemásolni. Ennek érdekében először növeljük a y_1 tömbben tárolt elemek számlálóját, azaz a db_1 változót (8. sor), majd y_1 megfelelő helyére bemásoljuk az aktuális elemet (9. sor). Ha a vizsgált elem nem P tulajdonságú, akkor hasonló módon a y_2 tömbbe másoljuk az aktuális elemet (10-12. sor). A ciklus lefutását követően minden elem bekerült a neki megfelelő kimeneti tömbbe, ezért az algoritmus végén kimenetként visszaadjuk a két kimeneti tömböt, valamint a tömbökben eltárolt releváns elemek számát (15. sor).

2.11. Példa. Feladatunk, hogy egy hat elemű egész számokat tartalmazó tömböt szétválogassunk két tömbbe úgy, hogy az egyikbe kerüljenek a páros számok, a másikba pedig a páratlan számok. A feladat szétválogatás tétellel történő megoldását a 2.10. ábra szemlélteti. ¶

Futási idő elemzése. A 2.15. algoritmus minden egyes tömbbeli elemet egyszer vizsgál meg. A vizsgált elem pedig minden esetben átkerül valamelyik tömbbe. Így kijelenthető, hogy a futási idő mindig ugyanannyi, mégpedig a bemeneti tömb méretével arányos, azaz $O(n)$-es. ♣

A szétválogatás tétel megismert algoritmusának hátránya, hogy két n elemű kimeneti tömböt foglalunk le minden esetben. Könnyen belátható az is, hogy a két tömbbe összességében n elemet másolunk, tehát a kimeneti tömbök kétszer annyi helyet foglalnak a memóriában, mint amennyire valójában szükség lenne. Ezen probléma feloldására adhatunk egy olyan megoldást is, amely egyetlen n elemű kimeneti tömböt használ, melynek elejére másoljuk a P tulajdonságú elemeket, végére pedig a nem P tulajdonságúakat.

A szétválogatás tétel egy kimeneti tömböt használó megvalósítását a 2.16. algoritmus adja meg. A bemeneti változók nem változnak meg a 2.15. algoritmusban bemutatotthoz képest, a kimeneti változók viszont igen. Egyetlen kimeneti tömbünk lesz (y), illetve a db változóval jelöljük, hogy hány P tulajdonságú elemet találtunk az x tömbben. Az algoritmus végén az y tömb első db darab eleme P tulajdonságú, a $(db+1)$-edik elemtől az utolsóig pedig nem P tulajdonságú.

Először a memóriában helyet kell foglalni a kimeneti y tömb számára, amelynek elemszáma megegyezik a bemeneti x tömb n elemszámával (2. sor).A 2.16. algoritmus végrehajtása során a db változó értéke mindig megegyezik az utoljára beszúrt P tulajdonságú elem indexével az y tömbben, a $jobb$ változó pedig ugyanezt adja meg a nem P tulajdonságú elemek esetén. Így kezdetben a db-nek 0 a $jobb$-nak pedig $n+1$ kezdeti értéket kell adni (ld. 3. és 4. sorok). Az inicializáló lépéseket követően az 5. és 13. sorok közötti ciklussal bejárjuk az x tömböt. A cikluson belül megvizsgáljuk, hogy az aktuális $x[i]$ elem P tulajdonságú-e (6. sor). Ha igen, akkor a vizsgált elemet az y tömb elejére kell másolnunk, ezért megnöveljük db értékét 1-gyel (7. sor), majd az y tömb db-edik helyére bemásoljuk az $x[i]$ elemet (8. sor). Amennyiben az x aktuális eleme nem volt P tulajdonságú, akkor az elágazás 9. sorban található különben ágába lépünk be, hogy az y tömb végére másoljuk be $x[i]$-t. Ekkor a $jobb$ indexelő értékét 1-gyel csökkentenünk kell (10. sor), és $y[jobb]$-ba másoljuk át $x[i]$-t. A ciklus végén, amikor bejártuk a teljes x tömböt, már csak vissza kell adni a kimeneti y és db változókat (14. sor).

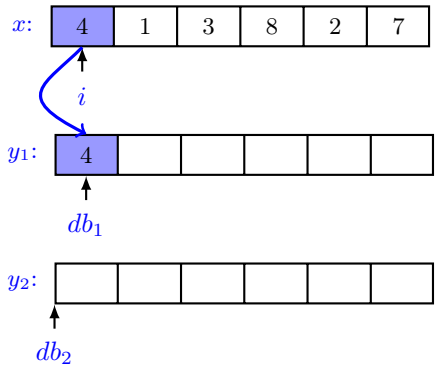
2.15. Algoritmus Szétválogatás programozási tétel

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai**
Kimenet: y_1 – **T tömb**, db_1 – **egész**, y_2 – **T tömb**, db_2 – **egész**

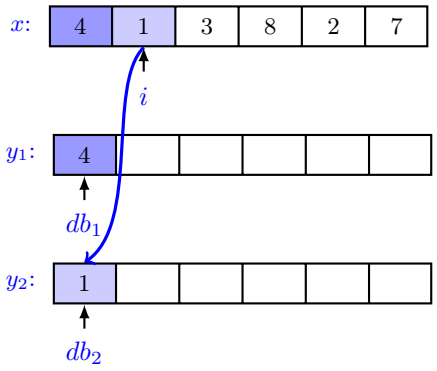
```
1: függvény SZÉTVÁLOGATÁS(x : T tömb, n : egész, P : logikai)
2:     y1 ← LÉTREHOZ(T)[n]
3:     y2 ← LÉTREHOZ(T)[n]
4:     db1 ← 0
5:     db2 ← 0
6:     ciklus i ← 1-től n-ig
7:         ha P(x[i]) akkor
8:             db1 ← db1 + 1
9:             y1[db1] ← x[i]
10:        különben
11:            db2 ← db2 + 1
12:            y2[db2] ← x[i]
13:        elágazás vége
14:    ciklus vége
15:    vissza(y1, db1, y2, db2)
16: függvény vége
```

Felhasznált változók és függvények

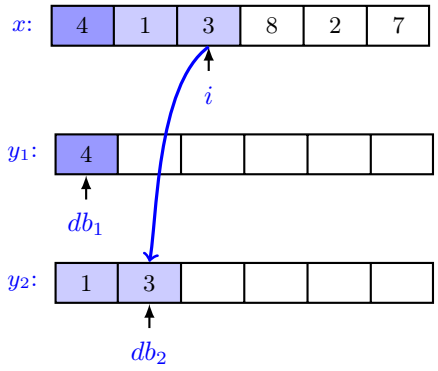
- x: A feldolgozandó tömb.
- n: Az x mérete.
- P: Logikai értékű tulajdonság függvény. A szétválogatás során a P és a nem P tulajdonságú elemeket válogatjuk külön.
- y_1: n elemű tömb, melynek első db_1 darab eleme tartalmazza az x tömb P tulajdonságú elemeit.
- db_1: Az x tömb P tulajdonságú elemeinek száma.
- y_2: n elemű tömb, melynek első db_2 darab eleme tartalmazza az x tömb nem P tulajdonságú elemeit.
- db_2: Az x tömb nem P tulajdonságú elemeinek száma.
- LÉTREHOZ(**T**)[n]: Utasítás, mely létrehoz egy n elemű **T** típusú tömböt.



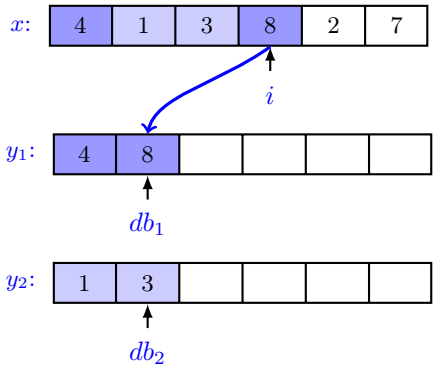
(a) Bejárás 1. lépés.



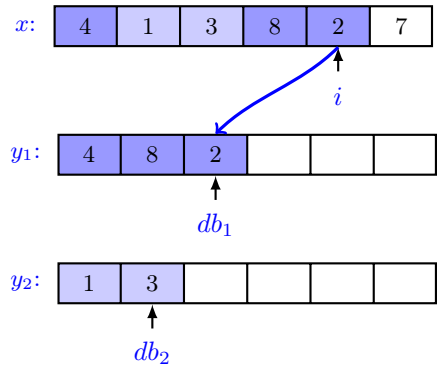
(b) Bejárás 2. lépés.



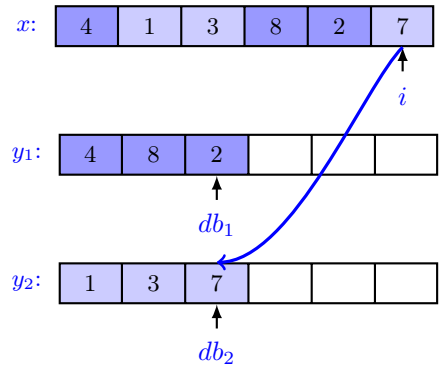
(c) Bejárás 3. lépés.



(d) Bejárás 4. lépés.



(e) Bejárás 5. lépés.



(f) Bejárás 6. lépés.

2.10. ábra. Az x tömb elemeinek szétválogatása. A páros elemek kerülnek az y_1 tömbbe, a páratlanok pedig az y_2-be.

2.16. Algoritmus Szétválogatás programozási tétel egyetlen új kimeneti tömbbe

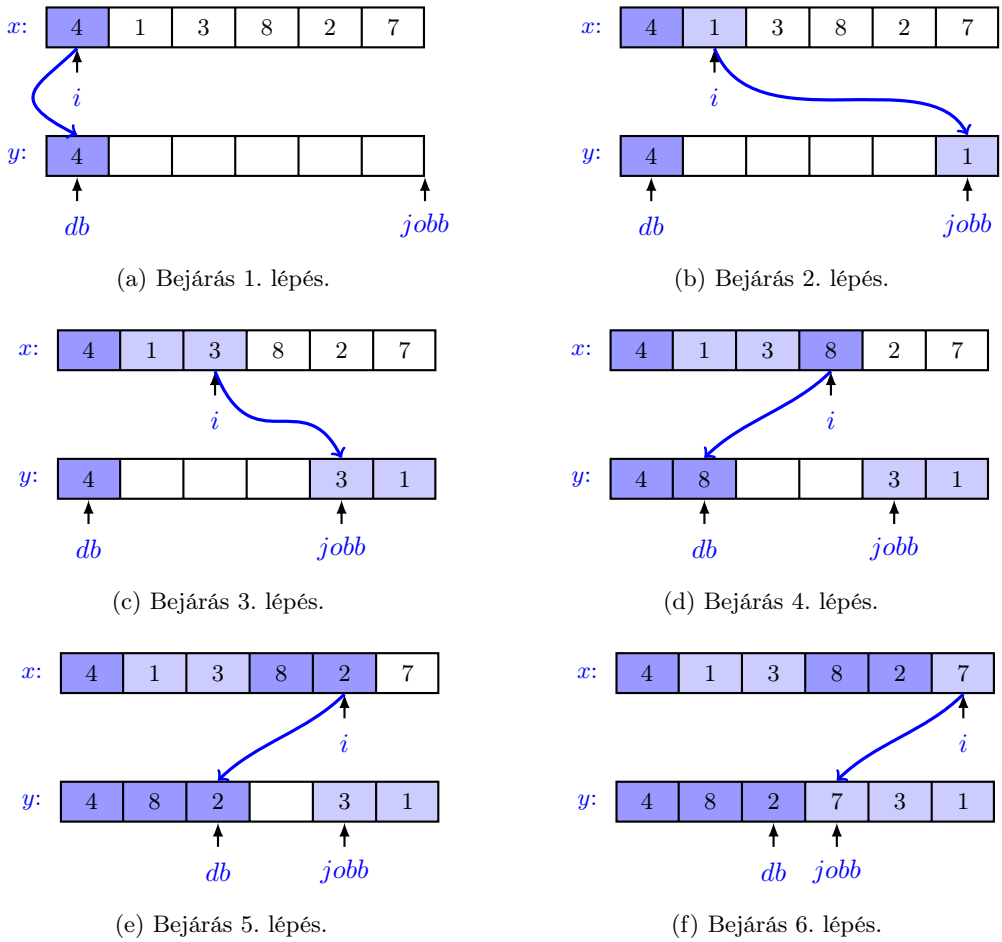
Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai**
Kimenet: y – **T tömb**, db – **egész**

```
1:  függvény SZÉTVÁLOGATÁS(x : T tömb, n : egész, P : logikai)
2:      y ← LÉTREHOZ(T)[n]
3:      db ← 0
4:      jobb ← n + 1
5:      ciklus i ← 1-től n-ig
6:          ha P(x[i]) akkor
7:              db ← db + 1
8:              y[db] ← x[i]
9:          különben
10:             jobb ← jobb − 1
11:             y[jobb] ← x[i]
12:         elágazás vége
13:     ciklus vége
14:     vissza(y, db)
15: függvény vége
```

Felhasznált változók és függvények

- x: A feldolgozandó tömb.
- n: Az x mérete.
- P: Logikai értékű tulajdonság függvény. A szétválogatás során a P és a nem P tulajdonságú elemeket válogatjuk külön.
- y: n elemű tömb, melynek első db darab eleme tartalmazza az x tömb P tulajdonságú elemeit, a $db + 1$ és n indexek közötti elemek pedig az x tömb nem P tulajdonságú elemei.
- db: Az x tömb P tulajdonságú elemeinek száma.
- $jobb$: Indexelő, amely azt jelöli, hogy az y tömb melyik helyére másolunk át egy nem P tulajdonságú elemet az x tömbből.
- LÉTREHOZ(**T**)[n]: Utasítás, mely létrehoz egy n elemű **T** típusú tömböt.

2.12. Példa. Feladatunk, hogy a hat elemű egész számokat tartalmazó x tömb elemeit válogassuk szét az y tömbbe úgy, hogy y elejére kerüljenek a páros számok, a végére pedig a páratlan számok. A páros számok számát a *db* változó adja meg. A feladat 2.16. algoritmussal történő megoldását a 2.11. ábra szemlélteti. ¶



(a) Bejárás 1. lépés.

(b) Bejárás 2. lépés.

(c) Bejárás 3. lépés.

(d) Bejárás 4. lépés.

(e) Bejárás 5. lépés.

(f) Bejárás 6. lépés.

2.11. ábra. Az x tömb elemeinek szétválogatása egyetlen tömbbe. A páros elemek kerülnek az y tömb elejére, a páratlanok pedig a végére.

A szétválogatás memória helyfoglalás szempontjából leghatékonyabb megoldása az, amikor az eredeti tömbön belül végezzük el. Ezt többféle módon is megvalósíthatjuk. Egyik lehetőség a kiválogatás programozási tétel tárgyalásakor már bemutatott 2.14. algoritmus. Ebben az esetben az eredeti tömbön kívül egyetlen plusz elemre van szükség, mivel a csere csak ilyen módon végezhető el. Ennél az algoritmusnál viszont van egy futási idő szempontjából gyorsabb lehetőség is, melyet a 2.17. algoritmus ad meg.

Az algoritmus a szokásos bemenetekkel rendelkezik. Kimenetként egyrészt visszaadja a bemeneti tömböt, melynek elején lesznek a P tulajdonságú elemek, végén pedig a nem P tulajdonságúak. A másik kimeneti változó (*db*) megadja, hogy melyik a tömb utolsó P tulajdonságú eleme, azaz az első *db* elem P tulajdonságú, a többi pedig nem.

Az algoritmus működésének lényege, hogy a tömb elejéről, illetve végéről két irányból végzünk bejárást. Ha a tömb elején nem P tulajdonságú elemet találunk, akkor azt hátra mozgatjuk, ha pedig hátul találunk P tulajdonságú elemet akkor azt előre mozgatjuk. A két irányú bejárás megvalósításához szükségünk van két indexelőre. A tömb elejéről hátrafelé haladó indexelőt *bal*-nak, a hátulról előre felé haladót pedig *jobb*-nak nevezzük az algoritmusban. Ezen indexek értéke az algoritmus kezdetén 1, illetve n lesz (ld. az algoritmus 2. és 3. sorát). Az elemek későbbi mozgatásakor óhatatlanul felül fogunk írni egy elemet, ezért a tömb első elemét kimentjük egy segédváltozóba (4. sor).

A tömb kétirányú bejárását egymásba ágyazott ciklusokkal valósítjuk meg. A külső ciklus az algoritmus 5. és 20. sora között található. A ciklus belépési és bennmaradási feltétele, hogy a bejárás közben

változó indexek még ne érjenek össze, azaz *bal* kisebb legyen *jobb*-nál. Ezt a feltételt a cikluson belüli elágazásokban és a belső ciklusokban is mindig vizsgáljuk, hiszen ha már nem teljesül, akkor nem kell folytatnunk a bejárást, így kiléphetünk a ciklus(ok)ból. Ezen feltétel teljesülését a továbbiakban nem fogjuk minden esetben külön kifejteni.

A 6. és 8. sor közötti ciklus valósítja meg a tömb hátulról történő bejárását. A ciklusban addig vagyunk bent, amíg a *jobb* indexszel meghatározott hátsó elem nem P tulajdonságú, azaz a helyén van. Ilyenkor a ciklus magjában csak annyit teszünk, hogy az indexet 1-gyel balra mozgatjuk, azaz értékét 1-gyel csökkentjük (7. sor).

A ciklusból két ok miatt léphetünk ki. Ha a *bal* index már nem kisebb a *jobb* indexnél, akkor be kell fejeznünk a bejárást, amit a felépített teljes ciklus és elágazás szerkezet megvalósít. Amennyiben viszont $bal < jobb$ még ***igaz***, akkor azért léptünk ki a ciklusból, mert $P\,(x[jobb])$ ***igaz***zá vált, azaz találtunk a tömb hátsó részében egy P tulajdonságú elemet, aminek elől lenne a helye. Ilyen esetben a 9. sorbeli elágazás feltétele ***igaz***, tehát belépünk az elágazásba és a hátul lévő elemet a tömb elejére másoljuk (10. sor). Mivel az $x[bal]$ elem P tulajdonságú, így mostantól már biztos a helyén van, ezért nem kívánjuk a későbbiekben módosítani. Ennek érdekében a *bal* indexet 1-gyel jobbra mozgatjuk (11. sor).

Ezt követően a tömböt az elejétől (konkrétan a *bal* index aktuális értékétől) járjuk be a vége felé, amit a 12. és 14. sorok közötti ciklus valósít meg. Amíg a ciklus feltétele teljesül, azaz a *bal* indexnél található elem P tulajdonságú, addig *bal* értékét 1-gyel növeljük (13. sor).

Ebből a ciklusból is két esetben léphetünk. Ha a *bal* index már nem kisebb a *jobb* indexnél, akkor be kell fejeznünk a bejárást. Ha viszont $bal < jobb$ teljesül (15. sor), akkor azért lépünk ki, mert a *bal* indexű elem nem P tulajdonságú. Ekkor ezt az elemet hátra kell mozgatnunk a *jobb* indexű helyre (16. sor) és a *jobb* index értékét 1-gyel csökkentenünk kell (17. sor).

A fent ismertetett kétirányú bejárást addig folytatjuk, amíg a két index, *bal* és *jobb* össze nem ér. Amennyiben egyenlővé válik a két index értéke, akkor kilépünk a ciklusokból, és az indexek aktuális helyére bemásoljuk a kezdetben külön változóba (*segéd*) kimentett tömbelemet (21. sor). Ebben a pillanatban az x tömb már teljesíti azt az elvárást, hogy a tömb elején minden elem P tulajdonságú, a végén pedig minden elem nem P tulajdonságú, valamint pontosan az eredeti tömb elemeit tartalmazza az átalakított tömb is. Szükséges viszont még megadni, hogy hol is található a választó vonal a P és a nem P tulajdonságú elemek között. Abban biztosak lehetünk, hogy az utoljára tömbbe mozgatott elem előtt minden elem P tulajdonságú, utána pedig mindegyik nem P tulajdonságú, tehát csak az utolsó elemet ($x[bal]$) kell megvizsgálnunk. Amennyiben ez az elem P tulajdonságú (22. sor), akkor az utolsó P tulajdonságú elem helyét megadó *db* változó értékét *bal*-ra állítjuk (23. sor). Ha $x[bal]$ nem P tulajdonságú, akkor az 1-gyel előbb lévő elem lesz az utolsó, tehát *db*-t $(bal - 1)$-re állítjuk (25. sor).

Az algoritmus végén nem kell visszaadnunk külön az átalakított x tömböt, mert cím szerinti paraméter átadással kapta meg ezt a tömböt az algoritmus. Így csak a másik kimeneti változót, a *db*-t adja visszatérési értékként a függvény (27. sor).

2.13. Példa. Egy egész számokat tartalmazó tömb elemeit válogassuk szét úgy, hogy az eredeti tömb elejére kerüljenek a páros számok, végére pedig a páratlan számok! A szétválogatás során csak az eredeti tömböt használhatjuk.

A feladat 2.17. algoritmussal történő megoldását a 2.12. ábra szemlélteti. ¶

Futási idő elemzése. A 2.17. algoritmus futási idejének elemzése során vizsgáljuk meg, hogy hány darab tulajdonságvizsgálat történik, illetve hányszor mozgatjuk az egyes elemeket.

A P tulajdonság teljesülését minden elem esetén pontosan egyszer vizsgáljuk. Az 5. és 20. sorok közötti ciklus az első elem kivételével minden elem esetén egyszer vizsgálja meg az P tulajdonság teljesülését. Az első elemet az algoritmus elején kimentjük a *segéd* változóba, majd az algoritmus végén a visszamozgatást követően megvizsgáljuk ezen elem esetén is a P tulajdonság teljesülését (22. sor). Így összesen n darab tulajdonság vizsgálatot hajtunk végre.

Az elemek mozgatása tekintetében nem tudjuk pontosan megmondani, hogy hány mozgatás történik, de a legrosszabb esetbeli mozgatások számát meg tudjuk határozni. Legrosszabb esetnek azt tekinthetjük, amikor a tömb elemei már kezdetben is szét voltak válogatva, de pont fordított módon, mint azt mi elvártuk volna, azaz a tömb elején a nem P tulajdonságú, a végén pedig a P tulajdonságú elemek vannak. Ebben az esetben a tömb bejárása során minden esetben mozgatunk egy elemet vagy a tömb végéről az elejére, vagy az elejéről a végére. Ezen mozgatások száma $n - 1$ lesz, mivel pontosan ennyi lépés alatt fog a kezdetben egymástól $n - 1$ távolságra lévő *bal* és *jobb* változó ugyanarra az elemre mutatni. Ezen mozgatásokon túl még figyelembe kell vennünk, hogy az algoritmus elején az első elemet

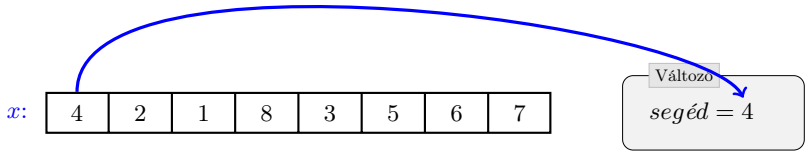
2.17. Algoritmus Szétválogatás programozási tétel az eredeti tömbben

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai**
Kimenet: x – **T tömb**, db – **egész**

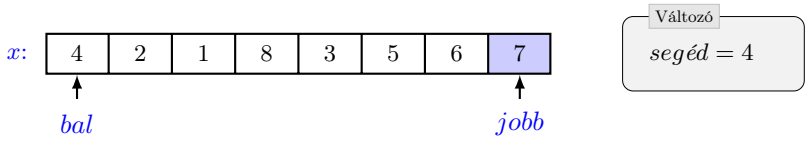
```
1: függvény SZÉTVÁLOGAT(címszerint x : T tömb, n : egész, P : logikai)
2:     bal ← 1
3:     jobb ← n
4:     segéd ← x[1]
5:     ciklus amíg bal < jobb
6:         ciklus amíg (bal < jobb) ∧ ¬P (x[jobb])
7:             jobb ← jobb − 1
8:         ciklus vége
9:         ha bal < jobb akkor
10:            x[bal] ← x[jobb]
11:            bal ← bal + 1
12:            ciklus amíg (bal < jobb) ∧ P (x[bal])
13:                bal ← bal + 1
14:            ciklus vége
15:            ha bal < jobb akkor
16:                x[jobb] ← x[bal]
17:                jobb ← jobb − 1
18:            elágazás vége
19:        elágazás vége
20:    ciklus vége
21:    x[bal] ← segéd
22:    ha P (x[bal]) akkor
23:        db ← bal
24:    különben
25:        db ← bal − 1
26:    elágazás vége
27:    vissza db
28: függvény vége
```

Felhasznált változók és függvények

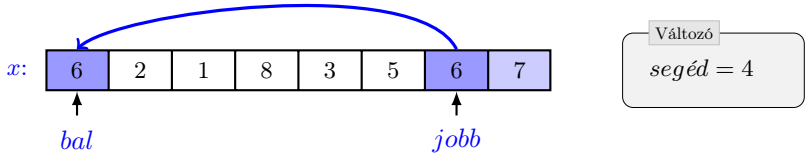
- x: A feldolgozandó tömb. Az algoritmus végén x első db darab eleme P tulajdonságú, a többi pedig nem P tulajdonságú.
- n: Az x mérete.
- P: Logikai értékű tulajdonság függvény.
- db: A P tulajdonságú elemek száma az x tömbben.
- *segéd*: Segédváltozó, melyben az x tömb első elemét tároljuk.
- *bal*: Az x tömb bejárása során az egyik index változó. Kezdeti értéke 1, a bejárás közben pedig folyamatosan növekszik az értéke. Az algoritmus során mindig igaz, hogy az x tömb *bal*-nál kisebb indexű elemei már mind P tulajdonságúak.
- *jobb*: Az x tömb bejárása során a másik index változó. Kezdeti értéke a tömb mérete, a bejárás közben pedig folyamatosan csökken az értéke. Az algoritmus során mindig igaz, hogy az x tömb *jobb*-nál nagyobb indexű elemei már nem P tulajdonságúak.



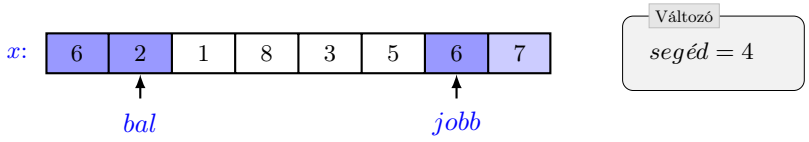
(a) Első elem kimentése segédváltozóba.



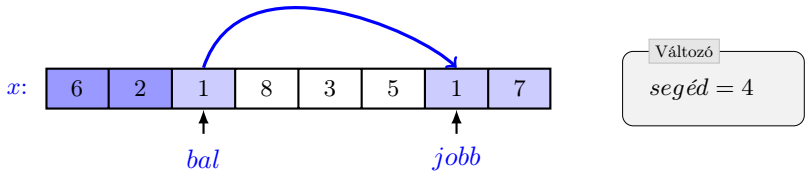
(b) Bejárás 1. lépés.



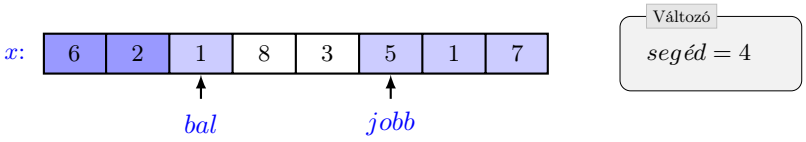
(c) Bejárás 2. lépés.



(d) Bejárás 3. lépés.

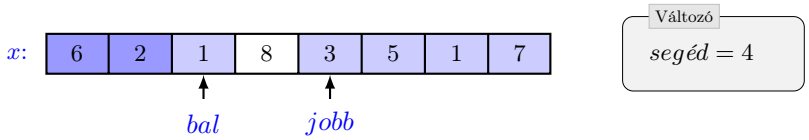


(e) Bejárás 4. lépés.

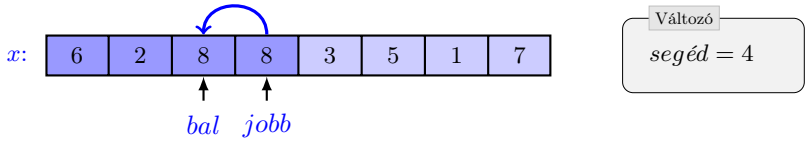


(f) Bejárás 5. lépés.

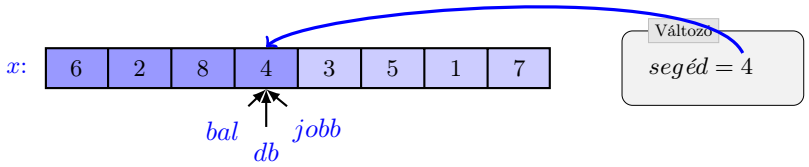
2.12. ábra. Szétválogatás programozási tétel megvalósítása az eredeti tömbben. A páros számok kerülnek a tömb elejére, a páratlanok pedig a végére.



(g) Bejárás 6. lépés.



(h) Bejárás 7. lépés.



(i) Bejárás 8. lépés.

2.12. ábra. Szétválogatás programozási tétel megvalósítása az eredeti tömbben. A páros számok kerülnek a tömb elejére, a páratlanok pedig a végére (folyt.).

átmásoljuk a *segéd* változóba, majd az algoritmus végén onnan visszamozgatjuk a tömb *bal* indexű helyére. Így összességében legrosszabb esetben $n + 1$ darab mozgatás történik.

A fentiek figyelembe vételével kijelenthető, hogy az algoritmus futási ideje minden esetben $O(n)$-es.

A futási időt érdemes még összehasonlítani az ugyanezen feladatot megoldó 2.14. algoritmus futási idejével. Abban az esetben az a lényeges különbség, hogy mindig, amikor egy elemről megállapítjuk, hogy rossz helyen van a tömbben, akkor megcseréljük egy másik elemmel. Az 1. fejezetben láttuk, hogy egy cserét három darab mozgatással tudunk megvalósítani, így a legrosszabb esetben összesen $3 \cdot (n - 1)$ darab mozgatás szükséges, ami a 2.17. algoritmus $(n + 1)$-es maximális mozgatás számánál több. A tulajdonság vizsgálatok száma mindként algoritmusnál azonos, így kijelenthető, hogy futási idő szempontjából a 2.17. algoritmus hatékonyabb. ♣

Metszet programozási tétel

A metszet[7] programozási tételt olyan feladatok megoldása esetén használjuk, amikor két tömbből a közös elemeket ki kell válogatni egy harmadik tömbbe. Például ismert, hogy egy osztály tanulói közül kik járnak énekkarra, illetve kosárlabdázni, és szeretnénk meghatározni azon tanulókat, akik énekkarra és kosárlabdázni is járnak.

A két tömb közös elemeinek kiválogatását két korábban már tárgyalt programozási tétel összeépítésével valósítjuk meg. Az egyik bemeneti tömb minden egyes elemére megvizsgáljuk, hogy benne van-e a másik bemeneti tömbben is. Ez a vizsgálat lényegében egy eldöntés tétel. Az első bemeneti tömb azon elemeit, amelyek bent vannak a másik bemeneti tömbben is (tehát teljesítik ezt a tulajdonságot), kiválogatjuk a kimeneti tömbbe. Ez pedig egy kiválogatás tételt jelent. A probléma ilyen jellegű megvalósítását a 2.18. algoritmus írja le.

2.18. Algoritmus Metszet programozási tétel

Bemenet: $x_1 -$ **T tömb**, $n_1 -$ **egész** (*tömb mérete*), $x_2 -$ **T tömb**, $n_2 -$ **egész** (*tömb mérete*)
Kimenet: $y -$ **T tömb**, $db -$ **egész**

```
 1: függvény Metszet(x1 : T tömb, n1 : egész, x2 : T tömb, n2 : egész)
 2:     y ← Létrehoz(T)[n1]
 3:     db ← 0
 4:     ciklus i ← 1-től n1-ig
 5:         j ← 1
 6:         ciklus amíg (j ≤ n2) ∧ (x1[i] ≠ x2[j])
 7:             j ← j + 1
 8:         ciklus vége
 9:         ha j ≤ n2 akkor
10:             db ← db + 1
11:             y[db] ← x1[i]
12:         elágazás vége
13:     ciklus vége
14:     vissza (y, db)
15: függvény vége
```

Felhasznált változók és függvények

- x_1: Egyik bemeneti tömb.
- n_1: Az x_1 tömb mérete.
- x_2: Másik bemeneti tömb.
- n_2: Az x_2 tömb mérete.
- y: Kimeneti tömb, melynek elemei az x_1 tömb azon elemei, amik benne vannak az x_2 tömbben is.
- db: Az y tömbben a releváns elemek száma, azaz az x_1 tömb azon elemeinek száma, amelyek benne van az x_2 tömbben is.
- Létrehoz(**T**)$[n_1]$: Utasítás, mely létrehoz egy n_1 elemű **T** típusú tömböt.

Az algoritmus bemenete az x_1 és az x_2 tömb, melyeknek ismerjük az n_1 és az n_2 elemszámát is. Az algoritmus kimenetként előállítja az y tömböt, melyben a két bemeneti tömb közös elemei vannak. Az algoritmus megvizsgálja, hogy az x_1 tömb, mely eleme található meg az x_2 tömbben is. Ebből az elvből következik, hogy a kimeneti tömbnek legfeljebb annyi eleme lehet, mint az x_1 tömbnek, ezért a memóriában ekkora helyet kell lefoglalni neki. Szükséges viszont még egy kimeneti változó, hiszen meg kell adnunk, hogy hány közös elemet találtunk a két bemeneti tömbben. Ezt az adatot a db kimeneti változó adja meg, így az y tömb első db darab eleme szolgáltat valódi információt.

Ahogy már említettük a kiválogatás tétel és az eldöntés tétel egybeépítésével valósítjuk meg az algoritmust. Egy kiválogatás tételt adunk meg úgy, hogy a P tulajdonság vizsgálatát egy eldöntés tétel helyettesíti. A kiválogatás tételnek megfelelően először létrehozzuk n_1 elemű tömbként az y tömböt (2. sor), valamint inicializáljuk a db változót (3. sor). A 4. és 13. sorok közötti ciklussal bejárjuk az x_1 tömböt.

[7] Angolul: intersection

Az x_1 tömb minden egyes elemére meg kell vizsgálnunk, hogy benne van-e az x_2 tömbben. Ennek érdekében az 5. és 8. sorok közötti részen bejárjuk az x_2 tömböt az eldöntés tételnél megismert módon. Az x_2 tömb bejárása mindig a tömb elejéről indul, ezért a j index értékét 1-re állítjuk (5. sor). A 6. és 8. sorok közötti ciklusban addig maradunk bent, amíg van még az x_2 tömbnek nem megvizsgált eleme, illetve amíg nem találunk az x_1 tömb aktuális $x_1[i]$ elemével megegyező elemet az x_2 tömbben. A cikluson belül mindig eggyel lépünk tovább az x_2 tömbben (7. sor).

A ciklusból kilépve megvizsgáljuk, hogy mi a kilépés oka. Ha $j \leq n_2$ teljesül (9. sor), akkor az $x_1[i] \neq x_2[j]$ feltétel lett ***hamis***, azaz találtunk egy $x_1[i]$-vel egyenlő elemet az x_2 tömbben. A megtalált elemet át kell másolnunk az y tömbbe. A másolás érdekében az y tömb indexelőjét, a *db* változót eggyel növelnünk kell (10. sor), majd a megtalált elemet már át tudjuk másolni az y tömb megfelelő helyére (11. sor).

Ha az x_1 tömb minden elemére megvizsgáltuk már, hogy benne van-e az x_2 tömbben, akkor vissza kell adni az előállított y tömböt és a *db* változót (14. sor).

Megjegyzés

A metszet fogalmat a matematikai halmazok esetén már megismertük. Lényeges különbség viszont a most tárgyalt metszet és a matematikai metszet között, hogy matematikában a halmazokban nem lehet ismétlődés, az általunk tárgyalt tömbökben viszont igen. Így ha az x_1 tömbben van olyan többször is előforduló elem, amely benne van az x_2 tömbben is, akkor ezt az elemet az összes előfordulásával átmásoljuk az y tömbbe. Ha viszont egy elem az x_1 tömbben egyszer van bent, de az x_2 tömbben többször is előfordul, akkor csak egyszer másoljuk át az y tömbbe.

Ebből következik, hogy a jelenleg tárgyalt metszetünk mint matematikai művelet, nem tekinthető kommutatívnak, azaz nem felcserélhető x_1 és x_2 szerepe.

2.14. Példa. Adott két négy elemű tömb a 2.13. ábrán látható módon. Határozzuk meg a két tömb közös elemeit a metszet tétel alkalmazásával! A megvalósítás egyes lépéseit a 2.13. ábra szemlélteti.

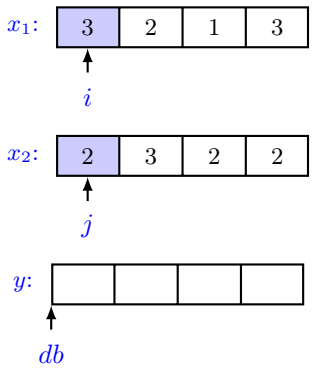
Jól látható, hogy az x_1 tömb többször előforduló azon elemei, amelyek x_2-ben is bent vannak, minden előfordulásukkor bemásolódnak az y tömbbe. Az x_2 többször előforduló és x_1-ben is előforduló elemei viszont csak egyszer másolódnak az y tömbbe. ¶

Futási idő elemzése. A metszet tétel algoritmusának futási ideje a két bemeneti tömb méretétől, n_1-től és n_2-től is függ, hiszen mindkét tömböt be kell járnunk. Az algoritmus során az x_1 tömb minden egyes elemét keressük az x_2 tömbben. Ez a keresés találat esetén legalább egy, legfeljebb n_2, illetve átlagosan $\frac{n_2}{2}$ darab vizsgálatot jelent. Ha a keresett elem nincs bent az x_2 tömbben, akkor az egész tömböt végig kell nézni, tehát n_2 darab vizsgálat szükséges. Ezek alapján kijelenthetjük, hogy az x_1 tömb egy elemének keresése az x_2 tömbben $O(n_2)$ futási idejű.

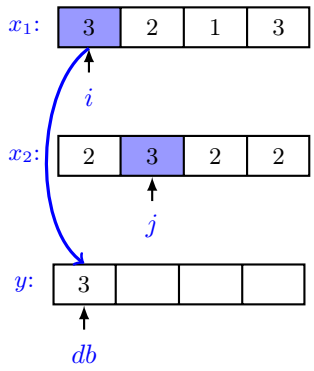
Mivel x_1 minden egyes elemére végre kell hajtanunk a keresést, ezért összességében $n_1 \cdot O(n_2) = O(n_1 \cdot n_2)$ lesz a futási ideje az algoritmusnak. ♣

A metszet tételt az eddigiekben arra használtuk, hogy két tömbből kiválogassuk a közös elemeket. Kis átalakítással lehetőségünk van arra, hogy a közös elemekkel kapcsolatos más jellegű feladatokra adjunk megoldást. Megvizsgálhatjuk például, hogy az x_1 és x_2 tömbnek van-e közös eleme. Ekkor lényegében két eldöntés tételt kell egymásba ágyaznunk, hiszen azt keressük, hogy van-e az x_1 tömbnek olyan eleme, amely benne van az x_2 tömbben is. Erre a feladatra ad megoldást a 2.19. algoritmus.

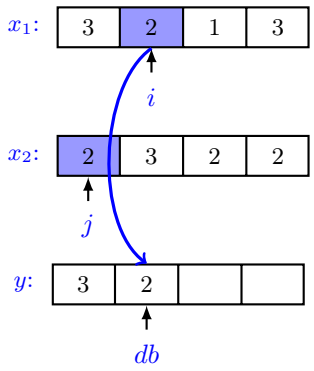
Hasonlóan vizsgálhatjuk azt is, ha tudjuk, hogy van közös eleme a két tömbnek, akkor hol található ez a közös elem az első tömbben. Ez egy kiválasztás és egy eldöntés összeépítését jelenti. Ha az a feladatunk, hogy határozzuk meg a két tömb közös elemeinek számát, akkor pedig egy megszámlálást és egy eldöntést kell egymásba építenünk. Ezen feladatok megvalósítását nem részletezzük, de ajánljuk a hallgatóknak gyakorlásként az algoritmusok önálló megalkotását.



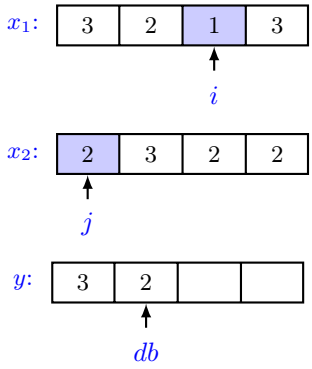
(a) $x_1[1]$ keresése az x_2 tömbben, 1. lépés. Mivel $x_1[1] = x_2[1]$ nem teljesül, így nem másoljuk $x_1[1]$-et az y tömbbe.



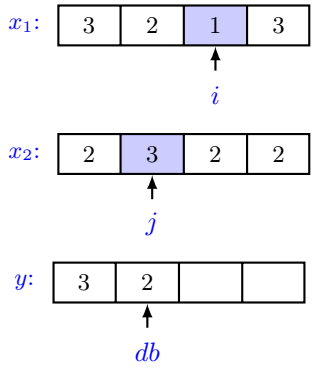
(b) $x_1[1]$ keresése az x_2 tömbben, 2. lépés. Mivel $x_1[1] = x_2[2]$ teljesül, ezért $x_1[1]$-et bemásoljuk az y tömbbe.



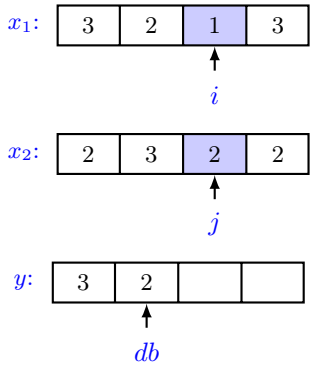
(c) $x_1[2]$ keresése az x_2 tömbben, 1. lépés. Mivel $x_1[2] = x_2[1]$ teljesül, ezért $x_1[2]$-t bemásoljuk az y tömbbe.



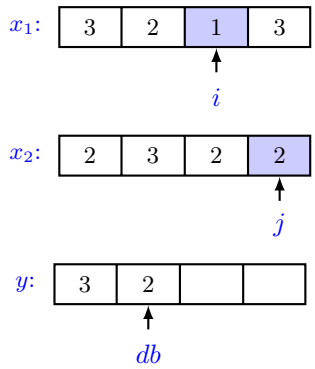
(d) $x_1[3]$ keresése az x_2 tömbben, 1. lépés.



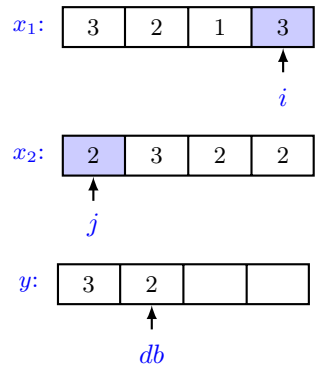
(e) $x_1[3]$ keresése az x_2 tömbben, 2. lépés.



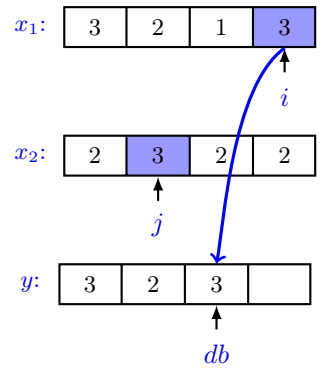
(f) $x_1[3]$ keresése az x_2 tömbben, 3. lépés.



(g) $x_1[3]$ keresése az x_2 tömbben, 4. lépés.



(h) $x_1[4]$ keresése az x_2 tömbben, 1. lépés.



(i) $x_1[4]$ keresése az x_2 tömbben, 2. lépés.

2.13. ábra. Metszet programozási tétel. Az x_1 tömb minden elemére egy eldöntés tétellel megvizsgáljuk, hogy benne van-e az x_2 tömbben. Ha bent van, akkor x_1 vizsgált elemét átmásoljuk az y tömb soron következő helyére.

2.19. Algoritmus Közös elem létezésének vizsgálata

Bemenet: x_1 – **T tömb**, n_1 – **egész** (*tömb mérete*), x_2 – **T tömb**, n_2 – **egész** (*tömb mérete*)
Kimenet: *van* – **logikai**

```
1:  függvény KözösElemEldöntése(x1 : T tömb, n1 : egész, x2 : T tömb, n2 : T tömb)
2:      i ← 1
3:      van ← hamis
4:      ciklus amíg (i ≤ n1) ∧ ¬van
5:          j ← 1
6:          ciklus amíg (j ≤ n2) ∧ (x1[i] ≠ x2[j])
7:              j ← j + 1
8:          ciklus vége
9:          ha j ≤ n2 akkor
10:             van ← igaz
11:         különben
12:             i ← i + 1
13:         elágazás vége
14:     ciklus vége
15:     vissza van
16: függvény vége
```

Felhasznált változók és függvények

- x_1: Egyik bemeneti tömb.
- n_1: Az x_1 tömb mérete.
- x_2: Másik bemeneti tömb.
- n_2: Az x_2 tömb mérete.
- *van*: Logikai változó, mely pontosan akkor ***igaz***, ha van olyan elem, amely az x_1 és az x_2 tömbben is benne van.

Unió programozási tétel

Az unió[8] programozási tételt olyan esetben használjuk, amikor két tömbből ki akarjuk válogatni az összes előforduló elemet, tehát azokat, amik akár az egyikben, akár a másikban benne vannak. Ezt a feladatot úgy tudjuk megvalósítani, hogy az egyik bemeneti tömb minden egyes elemét átmásoljuk a kimeneti tömbbe, majd a másik bemeneti tömbnek csak azokat az elemeit másoljuk át a kimeneti tömbbe, amelyek nincsenek bent a már korábban teljes egészében átmásolt tömbben. Az unió tételt a 2.20. algoritmussal valósíthatjuk meg.

Az algoritmus bemenete az x_1 tömb, melynek ismerjük az elemszámát (n_1), valamint az x_2 tömb az elemszámával (n_2). Az algoritmus kimenete az y tömb, mely tartalmazza a bemeneti tömbök minden elemét, valamint a *db* változó, amely megadja, hogy hány releváns eleme van az y tömbnek.

Az algoritmus 2. sorában létrehozzuk a kimeneti y tömböt, melynek maximális elemszáma a bemeneti tömbök elemszámainak összege. Ezt követően a 3. és 5. sorok közötti ciklusban megvalósítjuk, hogy az x_1 tömb minden eleme bemásolódjon az y tömbbe. A másolás végén az y tömbben n_1 darab elem lesz, ezért a *db* változónak átadjuk ezt az értéket (6. sor).

Az algoritmus lényegi része ezután következik. A 7. és 16. sorok közötti számlálós ciklussal bejárjuk az x_2 tömböt. A tömb minden elemére megvizsgáljuk, hogy benne van-e az x_1 tömbben is. Ha benne van, akkor nem kell bemásolni y-ba, ha viszont nincs bent, akkor a másolás szükséges. Ennek érdekében meg kell vizsgálnunk, hogy $x_2[j]$ benne van-e az x_1 tömbben, amihez egy módosított eldöntés tétel használata szükséges. Először inicializáljuk az i indexelő értékét (8. sor), majd a 9. és 11. sorok közötti ciklussal bejárjuk az x_1 tömböt mindaddig, amíg a tömb végére nem érünk, vagy meg nem találjuk az $x_2[j]$ elemet az x_1 tömbben. A ciklust követően megvizsgáljuk, hogy miért hagytuk abba a x_1 tömb bejárását (12. sor). Ha $i > n_1$, akkor az x_1 tömbben nincs benne az $x_2[j]$ elem, így be kell azt másolni az y tömbbe. Ehhez szükséges az y tömb indexelőjét eggyel növelni (13. sor), majd megtörténhet a tényleges másolás (14. sor). Az algoritmus végén vissza kell adnunk az előállított y tömböt és a bemásolt elemek számát megadó *db* változót (17. sor).

Futási idő elemzése. Az unió tétel futási idejének vizsgálatánál láthatjuk, hogy az x_1 tömböt minden esetben átmásoljuk az y tömbbe, ami n_1 darab értékadást jelent. Az algoritmus további részének futási ideje függ attól, hogy milyen elemeket tartalmaz a két bemeneti tömb. Futási idő szempontjából legrosszabb esetnek az minősül, ha a x_2 tömb egyetlen eleme sincs benne az x_1 tömbben. Ilyenkor ugyanis az x_2 tömb minden elemére elvégzünk egy eldöntést, amely vizsgálja, hogy az adott $x_2[j]$ elem benne van-e x_1-ben. Ez minden j-re n_1 darab vizsgálatot jelent, tehát az x_2 tömb minden elemére $n_2 \cdot n_1$ darab vizsgálatot. Így a futási idő $O(n_1 \cdot n_2)$-es lesz. ♣

2.15. Példa. Adott két egész számokat tartalmazó tömb a 2.14. ábrának megfelelően. Feladatunk, hogy állítsuk elő a két tömb unióját. A feladat megoldásához a 2.20. algoritmust használjuk.

Első lépésként lefoglaljuk a kimeneti y tömb számára szükséges helyet a memóriában. Ez a méret a két bemeneti tömb méretének összege lesz. Ezt követően az x_1 tömb minden elemét átmásoljuk az y tömbbe, majd a *db* indexet 4-re állítjuk (ld. 2.14a. ábra). Az x_1 tömb átmásolását követően feladatunk az x_2 tömb elemeinek keresése az x_1 tömbben, és szükséges esetén másolás az y tömbbe.

Először megvizsgáljuk (az eldöntés tétel használatával), hogy $x_2[1]$ benne van-e az x_1 tömbben. Mivel $x_2[1] = x_1[1]$, ahogy a 2.14b. ábrán is látható, ezért $x_2[1]$-et nem másoljuk át y-ba. Az $x_2[2]$ elem keresésekor először megnézzük, hogy megegyezik-e $x_1[1]$-gyel (ld. 2.14c. ábra). Mivel nem teljesül az egyenlőség, ezért az x_1 tömbben továbblépünk a következő elemre (ld. 2.14d. ábra). Látható, hogy $x_2[2] = x_1[2]$, ezért az $x_2[2]$ elemet nem kell az y tömbbe bemásolni. Következő lépésként megvizsgáljuk, hogy $x_2[3]$ benne van-e az x_1 tömbben. Nem fogunk egyezőséget találni, ezért az x_1 tömb i indexelője meghaladja a tömb méretét (ld. 2.14e. ábra), ami miatt az $x_2[3]$ elemet bemásoljuk az y tömbbe. A 2.14f. ábrán látható, hogy $x_2[4]$-et megtaláljuk x_1-ben. Az x_2 tömb utolsó elemét nem találjuk meg x_1-ben, ezért $x_2[5]$-öt bemásoljuk az y-ba (ld. 2.14g. ábra). ¶

[8] Angolul: union

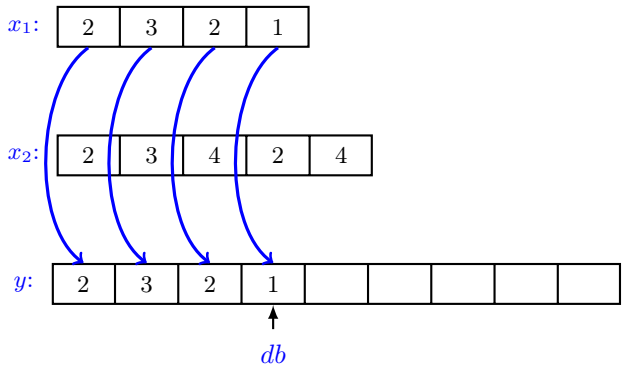
2.20. Algoritmus Unió programozási tétel

Bemenet: x_1 – **T tömb**, n_1 – **egész** (*tömb mérete*), x_2 – **T tömb**, n_2 – **egész** (*tömb mérete*)
Kimenet: y – **T tömb**, db – **egész**

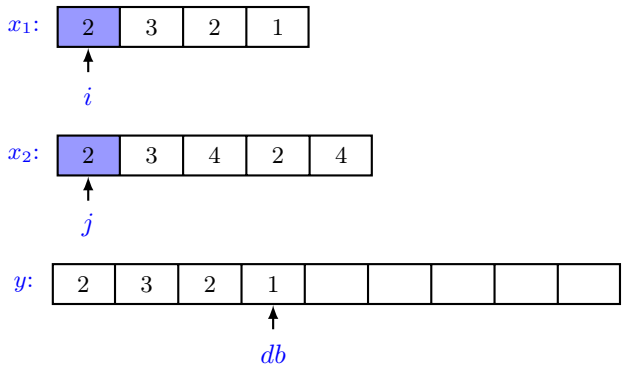
```
1: függvény Unió(x1 : T tömb, n1 : egész, x2 : T tömb, n2 : egész)
2:     y ← Létrehoz(T)[n1 + n2]
3:     ciklus i ← 1-től n1-ig
4:         y[i] ← x1[i]
5:     ciklus vége
6:     db ← n1
7:     ciklus j ← 1-től n2-ig
8:         i ← 1
9:         ciklus amíg (i ≤ n1) ∧ (x1[i] ≠ x2[j])
10:            i ← i + 1
11:        ciklus vége
12:        ha i > n1 akkor
13:            db ← db + 1
14:            y[db] ← x2[j]
15:        elágazás vége
16:    ciklus vége
17:    vissza (y, db)
18: függvény vége
```

Felhasznált változók és függvények

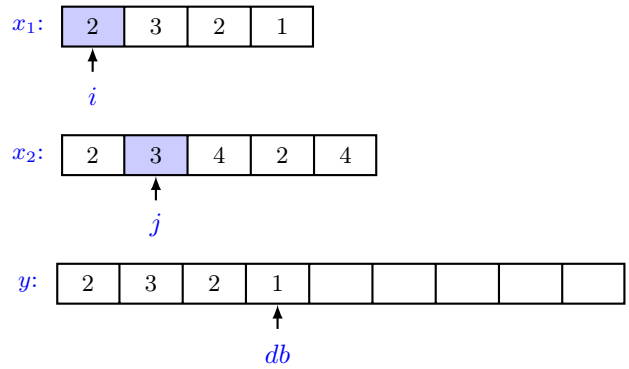
- x_1: Egyik bemeneti tömb.
- n_1: Az x_1 tömb mérete.
- x_2: Másik bemeneti tömb.
- n_2: Az x_2 tömb mérete.
- y: Kimeneti tömb, melynek minden egyes eleme vagy az x_1 vagy az x_2 tömbnek is eleme.
- db: Az y tömbben a releváns elemek száma.
- Létrehoz(**T**)$[n_1 + n_2]$: Utasítás, mely létrehoz egy $n_1 + n_2$ elemű **T** típusú tömböt.



(a) Az y tömböt 9 elemű tömbként hozzuk létre. Az x_1 tömb minden elemét átmásoljuk az y tömb azonos megfelelő helyére, majd a db változót az utolsó bemásolt elem indexére változtatjuk.

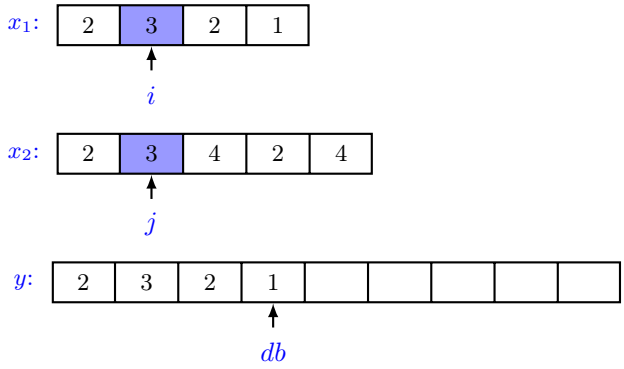


(b) Az $x_2[1]$ elemet keressük az x_1 tömbben. Mivel $x_2[1] = x_1[1]$, ezért nem kell az $x_2[1]$ elemet az y tömbbe bemásolni.

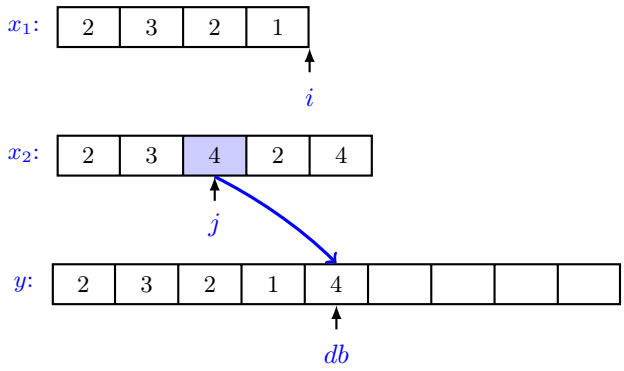


(c) Az $x_2[2]$ elem keresése az x_1 tömbben, 1. lépés.

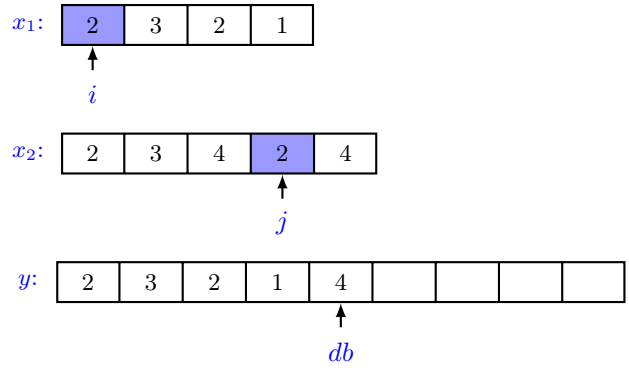
2.14. ábra. Unió programozási tétel.



(d) Az $x_2[2]$ elem keresése az x_1 tömbben, 2. lépés. Mivel $x_2[2] = x_1[2]$, ezért az $x_2[2]$ elemet nem másoljuk be az y tömbbe.

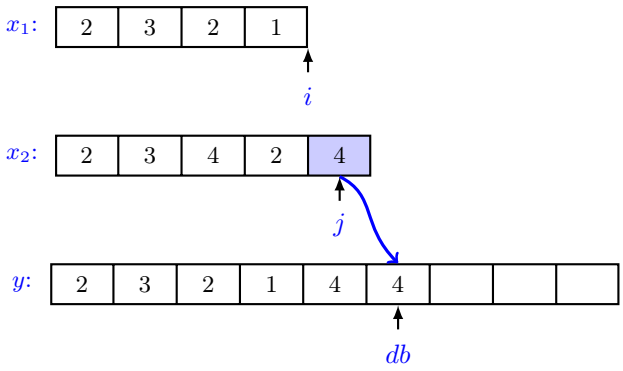


(e) Az $x_2[3]$ elem keresése az x_1 tömbben. Mivel nem található meg a keresett elem az x_1-ben, ezért az i index meghaladja az x_1 méretét. $x_2[3]$-at bemásoljuk az y tömbbe.



(f) Az $x_2[4]$ elem keresése az x_1 tömbben. Mivel $x_2[4] = x_1[1]$, ezért nem másoljuk be a keresett elemet az y tömbbe.

2.14. ábra. Unió programozási tétel (folyt.).



(g) Az $x_2[5]$ elem keresése az x_1 tömbben. Mivel nem található meg a keresett elem az x_1-ben, ezért az i index meghaladja az x_1 méretét. $x_2[5]$-öt bemásoljuk az y tömbbe.

2.14. ábra. Unió programozási tétel (folyt.).

Megjegyzés

Az unió tétellel kapcsolatban is észrevehetjük, hogy a többször előforduló elemeket nem pontosan úgy kezeli, ahogyan azt a matematikai halmazoknál megismert unió műveletnél megismertük. Ha van olyan elem, ami többször is előfordul az x_1 tömbben, akkor azt az összes előfordulásával átmásoljuk az y tömbbe, hiszen a 3. és 5. sorok közötti másolás semmilyen feltételt nem vizsgál, csak másolja az elemeket.

Az x_2 tömbben többször előforduló elemekkel már más a helyzet. Ha a többszörös elem olyan, hogy x_1-ben is megtalálható, akkor x_2-ből egyszer sem másoljuk át az y tömbbe. Ha viszont az x_2-ben többször megjelenő elem nincs benne az x_1 tömbben, akkor minden előfordulását átmásoljuk az y-ba.

A metszet és az unió tételnél is láttuk, hogy az ismétlődő elemek „zavart” okozhatnak az algoritmusok működésében. Ezért felmerül bennünk annak az igénye, hogy legyen olyan algoritmusunk, amely egy tömbből kiszűri az ismétlődéseket. Erre ad megoldást a 2.21. algoritmus.

Az algoritmus bemenete az x tömb, melynek ismerjük az elemszámát. Kimenetként az átalakított x tömböt adjuk vissza, melynek az első db darab eleme nem tartalmaz ismétlődő elemeket, viszont az eredeti tömb minden korábbi elemét egyszeres előfordulással tartalmazza. Természetesen a db változó értékét is szolgáltatnia kell az algoritmusnak.

Az algoritmus a db változó kezdeti értékének meghatározásával indul (2. sor). Ez 1 lesz, mivel ha csak az x első elemét tekintjük, az nem tartalmaz ismétlődést. Ezt követően a tömb összes többi elemére meg kell vizsgálni, hogy az előfordulását megelőző helyen megtalálható-e. Ennek érdekében a tömböt végigjárjuk a második elemtől az utolsó elemig egy számlálós ciklussal (3-12. sorok), amelyben az i értéke mindig a vizsgált tömbelem indexe. Az i-edik elemet össze kell hasonlítanunk az első db darab elemmel, ezt valósítja meg a 4. és 7. sorok közötti rész[9]. Ha a belső ciklusból kilépve a $j \leq db$ feltétel ***igaz***, akkor valamely j esetén az $x[i] \neq x[j]$ feltétel vált ***hamis***sá, azaz találtunk olyan elemet az első db darab elem között, amely megegyezik az $x[i]$ elemmel, ezért $x[i]$-t nem kell ismételten eltárolni a tömbben. Ha viszont $j > db$ teljesül (8. sor), akkor minden megvizsgált j-re az $x[i] \neq x[j]$ feltétel ***igaz*** volt, tehát $x[i]$ még nincs benne az átalakított tömb releváns részében, ezért be kell oda másolni. Ehhez először növeljük a db értékét (9. sor) – hiszen egy új megtartandó elemet találtunk –, majd $x[i]$-t bemásoljuk az adott helyre.

Az algoritmus végén az átalakított x tömböt nem kell külön visszaadnunk, mivel címszerinti paraméterátadást használtunk. A db változót viszont a külvilág felé is elérhetővé kell tennünk, ezért ennek az értékével tér vissza az algoritmus.

2.16. Példa. A 2.15. ábrán látható egy példa, amelyben egy hat elemű tömbből szűrjük ki a többször is előforduló elemeket. ¶

[9] Észrevehetjük, hogy ez valójában egy eldöntés tétel.

2.21. Algoritmus Ismétlődések kiszűrése

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*)
Kimenet: x – **T tömb**, db – **egész**

```
1: függvény IsmétlődésekKiszűrése(címszerint x : T tömb, n : egész)
2:     db ← 1
3:     ciklus i ← 2-től n-ig
4:         j ← 1
5:         ciklus amíg (j ≤ db) ∧ (x[i] ≠ x[j])
6:             j ← j + 1
7:         ciklus vége
8:         ha j > db akkor
9:             db ← db + 1
10:            x[db] ← x[i]
11:        elágazás vége
12:    ciklus vége
13:    vissza db
14: függvény vége
```

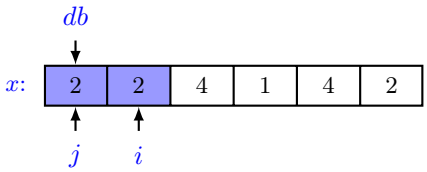
Felhasznált változók és függvények

- x: Tömb, melyet úgy alakít át az algoritmus, hogy az ismétlődő elemek közül pontosan egy maradjon a tömbben. A tömböt oly módon alakítjuk át, hogy a megmaradó elemek a tömb elejére kerüljenek.
- n: Az x tömb mérete.
- db: Az x tömbben az átalakítást követően megmaradó elemek száma.

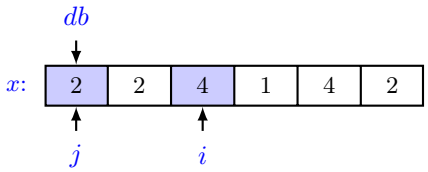
Futási idő elemzése. Futási idő szempontjából legrosszabb esetben a 2.21. algoritmus 5. sorában kezdődő belső ciklus j változója minden esetben $j > db$-vel száll ki a ciklusból. Ez azt jelenti, hogy minden elem pontosan egyszer fordul elő a tömbben. Ilyenkor a belső ciklus magja mindig db-szer kerül végrehajtásra, a db pedig a ciklusból kilépve 1-gyel növekszik. Így a futási idő:

$$T(n) = 1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}, \tag{2.3}$$

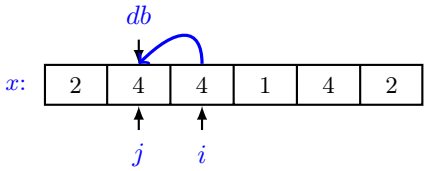
tehát az algoritmus $O\left(n^2\right)$-es. ♣



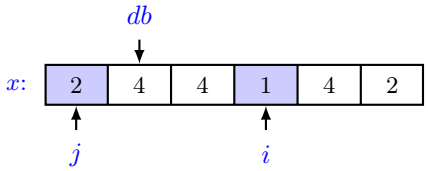
(a) A második elem vizsgálata. Mivel $x[2] = x[1]$, így nem kell eltárolni $x[2]$-t.



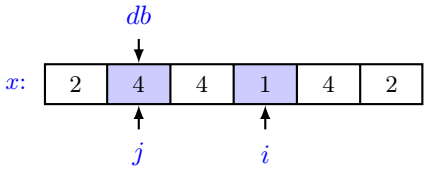
(b) A harmadik elem vizsgálata, 1. lépés.



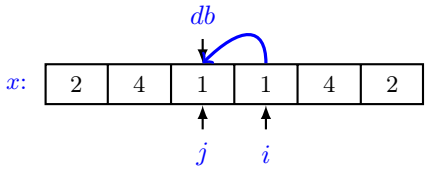
(c) A harmadik elem vizsgálata, 2. lépés. Mivel j nagyobb mint a korábbi db érték, ezért $x[3]$-at $x[2]$-be másoljuk.



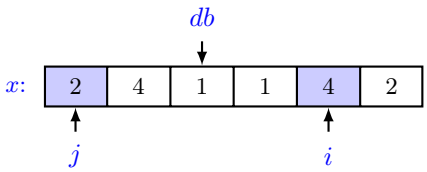
(d) A negyedik elem vizsgálata, 1. lépés.



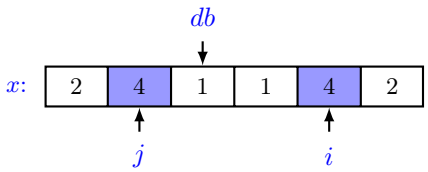
(e) A negyedik elem vizsgálata, 2. lépés.



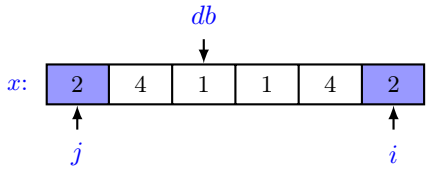
(f) A negyedik elem vizsgálata, 3. lépés. Mivel j nagyobb, mint a korábbi db érték, ezért $x[4]$-et $x[3]$-ba másoljuk.



(g) Az ötödik elem vizsgálata, 1. lépés.



(h) Az ötödik elem vizsgálata, 2. lépés.



(i) A hatodik elem vizsgálata, 1. lépés.

2.15. ábra. Ismétlődések kiszűrése

Összefuttatás programozási tétel

Az összefuttatás programozási tétel ugyanarra a problémára ad megoldást, mint az előző fejezetben bemutatott unió tétel. Adott két tömb és elő kívánunk állítani belőlük egy olyan tömböt, mely a két bemeneti tömb minden elemét tartalmazza, de lehetőleg úgy, hogy az ismétlődéseket (azaz a mindkét tömbben előforduló elemeket) kiszűrjük. Az összefuttatás tétel esetén viszont többlet információnk van a két bemeneti tömbről, azok ugyanis rendezettek. Viszont a rendezettség kihasználásával a megoldó algoritmus futási ideje jelentősen csökkenthető.

Az unió tétel esetén az volt a megoldásunk gondolatmenete, hogy az egyik bemeneti tömb minden elemét átmásoltuk a kimeneti tömbbe, majd a másik bemeneti tömb elemei közül csak azokat, amelyek a korábban bemásolt tömbnek nem elemei. Ehhez a második bemeneti tömb minden egyes eleme esetén végre kellett hajtanunk egy eldöntést, amely megadta, hogy a vizsgált elem benne van-e az első bemeneti tömbben. Emiatt a jelentős számú vizsgálat miatt lett az algoritmusunk futási ideje viszonylag lassú.

A bemeneti tömbök rendezettségét kihasználva viszont a két tömböt „párhuzamosan" is bejárhatjuk. A bejárást kezdjük mindkét tömb elejéről, azaz legyen mindkét tömbben az aktuális elem az első elem. A bejárás során minden egyes elempárnál vizsgáljuk meg, hogy melyikük kisebb. A kisebb értéket másoljuk be a kimeneti tömb soron következő helyére, majd lépjünk tovább abban a bemeneti tömbben, ahonnan a másolást végeztük. A bejárást addig folytassuk, amíg mindkét bemeneti tömb végére nem érünk. Ez az ötlet várhatóan gyorsabb futást eredményez, mint az unió tételnél ismertetett változat, hiszen minden vizsgálat után egy elem bemásolódik a kimeneti tömbbe és eggyel továbblépünk az adott tömbben. Az összefuttatás tétel konkrét megvalósítását a 2.22. algoritmus írja le.

Az algoritmus bemenetként megkapja a két összefuttatandó növekvő módon rendezett tömböt, melyeknek természetesen az elemszámait is ismerjük. Kimenetként szolgáltatja az összefuttatás eredményeként előálló tömböt. Erről a tömbről az algoritmus elején nem tudjuk, hogy hány eleme lesz, ezért lefoglalunk neki a memóriában annyi helyet, amennyi maximálisan szükséges lehet. Ez a méret a két bemeneti tömb méretének összege. Mivel csak az algoritmus végén derül ki, hogy a kimeneti y tömbnek hány releváns eleme van, ezért ezek számát (db) is visszaadja az algoritmus kimenetként.

Első lépésként létre kell hoznunk a kimeneti tömböt (2. sor), majd a tömbök bejárásához szükséges indexeknek adunk kezdeti értéket (3–5. sorok). Az x_1 bementi tömböt i-vel, x_2-t pedig j-vel indexeljük. A db változóban mindig eltároljuk, hogy az adott pontig hány darab elemet másoltunk már át a kimeneti y tömbbe, így egyszerre számlálóként és indexként is használható.

Az inicializáló lépéseket követően következik az algoritmus lényegi része, a bemeneti tömbök párhuzamos bejárása, melyet a 6. és 21. sorok közötti ciklus valósít meg. A ciklus addig fut, amíg valamelyik bementi tömböt teljes mértékben fel nem dolgoztuk, tehát amíg az i és j index sem haladja meg a megfelelő tömb elemszámát. A cikluson belül biztos, hogy egy új elemet fogunk másolni az y tömbbe, vagy az x_1 vagy az x_2 tömbből. Ezért a db változó értékét eggyel növeljük (7. sor). El kell döntenünk, hogy melyik tömbből másoljunk elemet az y tömbbe. Három különböző eset állhat elő, melyek a bemeneti tömbök aktuális elemeinek, azaz $x_1[i]$-nek és $x_2[j]$-nek az egymáshoz való viszonyától függnek. Ha x_1 aktuális eleme a kisebb (8. sor), akkor az első tömbből másolunk elemet az y-ba, majd az x_1 tömb következő elemére lépünk i növelésével. Ha viszont az x_2 aktuális eleme a kisebb (12. sor), akkor az x_2 tömbből történik a másolás és itt lépünk tovább. Harmadik eset, ha a bementi tömbök aktuális elemei megegyeznek (15. sor), akkor tetszés szerint valamelyik bementi tömbből (pl. x_1-ből) másolunk y-ba, majd mindkét tömbben továbblépünk, tehát az i és j indexek értékét is növeljük eggyel.

Amikor a bejárást megvalósító ciklusból kilépünk, akkor biztos, hogy az egyik tömbnek a végére értünk, hiszen vagy $i > n_1$ vagy $j > n_2$. Viszont a még nem teljes mértékben feldolgozott bemeneti tömb fennmaradó elemeit be kell másolnunk a kimeneti tömbbe, hiszen ezek az elemek mind nagyobbak a másik tömb összes eleménél, így helyük van az y tömbben. Ezt a másolást valósítják meg a 22. és 26., valamint a 27. és 31. sorok közötti ciklusok. Ha az x_1 tömbnek vannak még feldolgozatlan elemei, akkor az első, másik esetben pedig a második ciklus valósítja meg a hátralévő elemek y tömbbe másolását.

Az algoritmus végén még egyetlen lépés szükséges, az előállított kimeneti változókat vissza kell adni a külvilág felé (32. sor).

A bemutatott algoritmus megoldja az összefuttatás feladatát, de az algoritmust leíró pszeudokód elég hosszú, valamint első ránézésre bonyolultnak is tűnhet az egymást követő három ciklus miatt. Lehetőségünk van viszont arra, hogy a kódot valamelyest egyszerűsítsük egy kis trükk segítségével. Bővítsük ki mindkét tömböt egy plusz elemmel, melynek értéke biztos, hogy minden a tömbökben előforduló elemnél nagyobb. (Jelöljük ezt az értéket $+\infty$-nel.) A kibővítés miatt a tömbök rendezettsége természetesen

2.22. Algoritmus Összefuttatás programozási tétel

Bemenet: x_1 – **T rendezett tömb**, n_1 – **egész** (*tömb mérete*), x_2 – **T rendezett tömb**, n_2 – **egész** (*tömb mérete*)
Kimenet: y – **T rendezett tömb**, db – **egész**

```
1: függvény Összefuttatás(x1 : T rendezett tömb, n1 : egész, x2 : T rendezett tömb, n2 : egész)
2:     y ← Létrehoz(T)[n1 + n2]
3:     i ← 1
4:     j ← 1
5:     db ← 0
6:     ciklus amíg (i ≤ n1) ∧ (j ≤ n2)
7:         db ← db + 1
8:         ha x1[i] < x2[j] akkor
9:             y[db] ← x1[i]
10:            i ← i + 1
11:        különben
12:            ha x1[i] > x2[j] akkor
13:                y[db] ← x2[j]
14:                j ← j + 1
15:            különben
16:                y[db] ← x1[i]
17:                i ← i + 1
18:                j ← j + 1
19:            elágazás vége
20:        elágazás vége
21:    ciklus vége
22:    ciklus amíg i ≤ n1
23:        db ← db + 1
24:        y[db] ← x1[i]
25:        i ← i + 1
26:    ciklus vége
27:    ciklus amíg j ≤ n2
28:        db ← db + 1
29:        y[db] ← x2[j]
30:        j ← j + 1
31:    ciklus vége
32:    vissza (y, db)
33: függvény vége
```

Felhasznált változók és függvények

- x_1: Egyik rendezett bemeneti tömb.
- n_1: Az x_1 tömb mérete.
- x_2: Másik rendezett bemeneti tömb.
- n_2: Az x_2 tömb mérete.
- y: Rendezett kimeneti tömb, melynek minden egyes eleme vagy az x_1 vagy az x_2 tömbnek is eleme.
- db: Az y tömbben a releváns elemek száma.
- Létrehoz(**T**)$[n_1 + n_2]$: Utasítás, mely létrehoz egy $n_1 + n_2$ elemű **T** típusú tömböt.

továbbra is megmarad. Ennek a trükknek az alkalmazásával elérjük azt, hogy ha az egyik tömbben tárolt valódi elemeket már mind feldolgoztuk, a tömb aktuális eleme (ez a kibővítéskor bevezetett $+\infty$ értékű elem) még összehasonlítható a másik tömb elemeivel, és mivel a $+\infty$ nagyobb minden elemnél, így teljesül, hogy a másik tömb még fel nem dolgozott elemei másolódnak át az y tömbbe. A konkrét megvalósítást 2.23. algoritmus mutatja. A 10. sorban látható ciklusfeltételt is kis mértékben módosítani kell, mert addig kell a ciklusban bent maradni, amíg mindkét tömbben az összes eredeti elemet fel nem dolgoztuk.

2.23. Algoritmus Módosított összefuttatás programozási tétel

Bemenet: x_1 – **T rendezett tömb**, n_1 – **egész** (*tömb mérete*), x_2 – **T rendezett tömb**, n_2 – **egész** (*tömb mérete*)
Kimenet: y – **T rendezett tömb**, db – **egész**

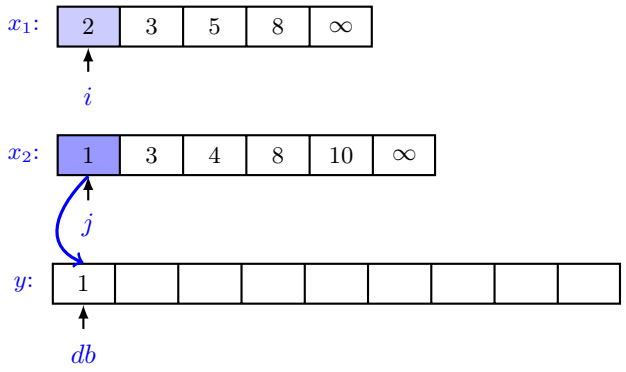
```
1:  függvény MódosítottÖsszefuttatás(x1 : T rendezett tömb, n1 : egész, x2 : T rendezett tömb, n2 : egész)
2:      y ← Létrehoz(T)[n1 + n2]
3:      n1 ← n1 + 1
4:      x1[n1] ← +∞
5:      n2 ← n2 + 1
6:      x2[n2] ← +∞
7:      i ← 1
8:      j ← 1
9:      db ← 0
10:     ciklus amíg (i < n1) ∨ (j < n2)
11:         db ← db + 1
12:         ha x1[i] < x2[j] akkor
13:             y[db] ← x1[i]
14:             i ← i + 1
15:         különben ha x1[i] > x2[j] akkor
16:             y[db] ← x2[j]
17:             j ← j + 1
18:         különben
19:             y[db] ← x1[i]
20:             i ← i + 1
21:             j ← j + 1
22:         elágazás vége
23:     ciklus vége
24:     vissza (y, db)
25: függvény vége
```

Felhasznált változók és függvények

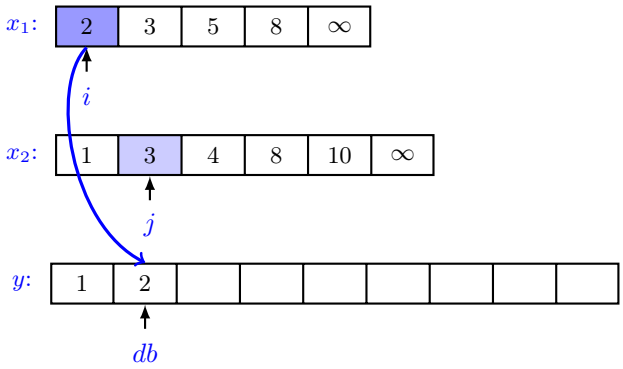
- x_1: Egyik rendezett bemeneti tömb.
- n_1: Az x_1 tömb mérete.
- x_2: Másik rendezett bemeneti tömb.
- n_2: Az x_2 tömb mérete.
- y: Rendezett kimeneti tömb, melynek minden egyes eleme vagy az x_1 vagy az x_2 tömbnek is eleme.
- db: Az y tömbben a releváns elemek száma.
- Létrehoz(**T**)$[n_1 + n_2]$: Utasítás, mely létrehoz egy $(n_1 + n_2)$ elemű **T** típusú tömböt.

Futási idő elemzése. Az összefuttatás tétel bevezetésekor már említettük, hogy fő célunk az unió tétel futási idejének nagymértékű javítása. Vizsgáljuk meg, hogy a bemutatott algoritmusokkal sikerült-e ezt a célt megvalósítanunk! Könnyen látható, hogy a futási idő a két bemeneti tömb méretének összegével arányos, hiszen minden bemeneti tömbbeli elemet legfeljebb egyszer másolunk be a kimeneti tömbbe és minden másolásnál legfeljebb kettő összehasonlítást végzünk csak (ld. 12. és 15. sorok). Így kijelenthető, hogy az összefuttatás tétel futási ideje $O(n_1+n_2)$-es, ami tényleg jelentős javulás az unió tétel $O(n_1 \cdot n_2)$-es futási idejéhez képest. Ne felejtsük viszont el, hogy ezt csak a bemeneti tömbök rendezettsége miatt tudtuk elérni. ♣

2.17. Példa. A 2.16. ábrán láthatunk egy példát arra, hogy a 2.23. algoritmussal miként futtatható össze két rendezett tömb. ¶

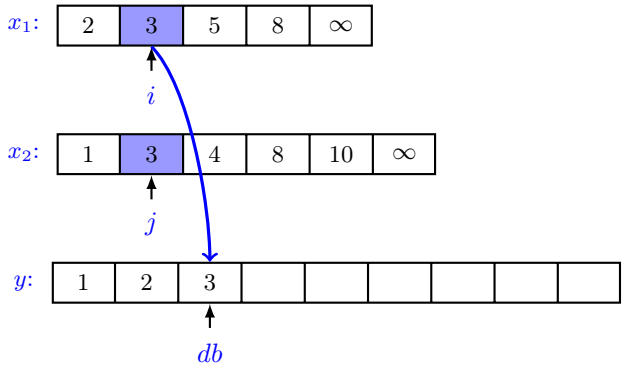


(a) A y tömböt létrehozzuk 9 elemű tömbként, valamint az x_1 és x_2 tömböket kibővítjük egy-egy ∞ értékű új elemmel. Mivel $x_1[1] > x_2[1]$, ezért $x_2[1]$-et másoljuk $y[1]$-be.

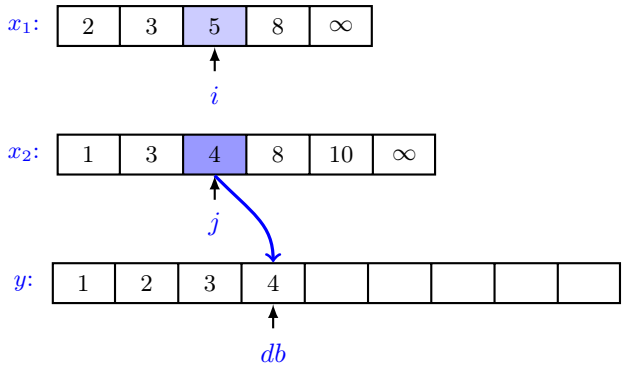


(b) Mivel $x_1[1] < x_2[2]$, ezért $x_1[1]$-et másoljuk $y[2]$-be.

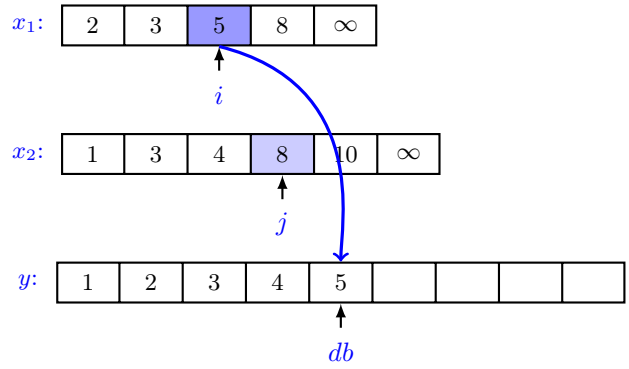
2.16. ábra. Összefuttatas programozási tétel.



(c) Mivel $x_1[2] = x_2[2]$, ezért $x_1[2]$-t másoljuk $y[3]$-ba.

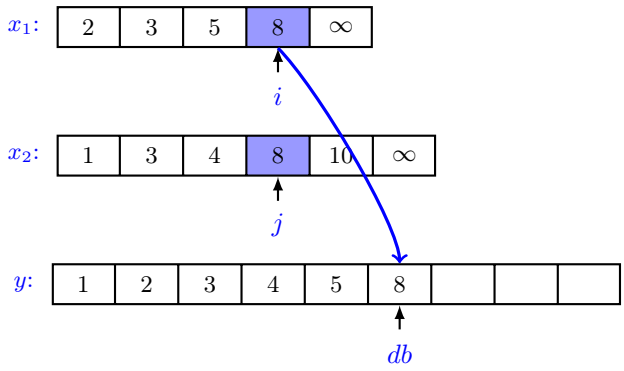


(d) Mivel $x_1[3] > x_2[3]$, ezért $x_2[3]$-at másoljuk $y[4]$-be.

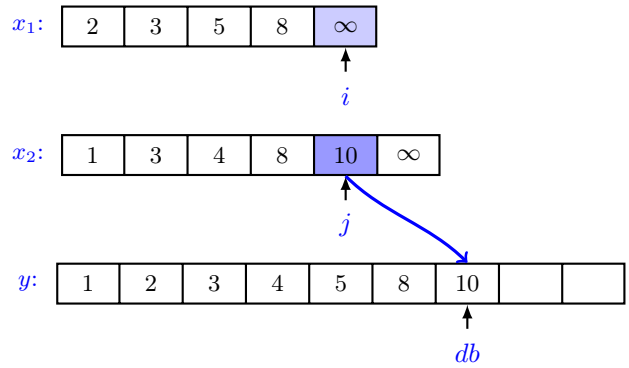


(e) Mivel $x_1[3] < x_2[4]$, ezért $x_1[3]$-at másoljuk $y[5]$-be.

2.16. ábra. Összefuttatás programozási tétel (folyt.).



(f) Mivel $x_1[4] = x_2[4]$, ezért $x_1[4]$-et másoljuk $y[6]$-ba.



(g) Mivel $x_1[5] > x_2[5]$, ezért $x_2[5]$-öt másoljuk $y[7]$-be. A bemeneti tömbök minden eredeti elemét feldolgoztuk, így véget ér az algoritmus futása.

2.16. ábra. Összefuttatás programozási tétel (folyt.).

Programozási tételek összeépítése

A programozási tételek olyan problémákra adnak hatékony, programozók körében általában használt megoldásokat, melyek az algoritmus alkotási munka során gyakran előfordulnak. Számos probléma viszont nem oldható meg egyetlen programozási tétel alkalmazásával, vagy a tételek egymás utáni használatával, hanem szükséges, hogy a tételeknél megismert ötleteket, módszereket összeépítve használjuk. Ilyen feladat lehet például, amikor egy tömb elemei közül ki kell válogatnunk az összes legnagyobb értékű elem indexét. Ehhez egyrészt egy maximumkiválasztást, másrészt egy kiválogatást kell megvalósítanunk. A két tételt természetesen egymást követően is használhatjuk, viszont ebben az esetben a feldolgozandó tömböt kétszer is végig kell járnunk, először a maximális érték meghatározása érdekében, majd a maximális értékű elemek indexének kiválogatása miatt. Ezzel szemben, a két említett programozási tételt össze lehet úgy is építeni, hogy csak egyszer kelljen a feldolgozandó tömböt bejárni.

Programozási tételek összeépítésére csak néhány példát mutatunk be, nem törekszünk a teljességre, hiszen az összeépítési lehetőségek száma olyan magas, hogy minden eset bemutatása túlmutat jelen jegyzet keretein. A bemutatott algoritmusok négy csoportra oszthatók. Elsőként a másolás tétellel való összeépítésre mutatunk be két lehetőséget, egyrészt a sorozatszámítással, másrészt a maximumkiválasztással történő összeépítésre. Ezt követi a megszámolás tétellel való összeépítés esetén egy eset, a kereséssel történő összeépítés tárgyalása. Harmadik csoport a maximumkiválasztással való összeépítés lehetősége egy példán keresztül bemutatva. Végül a kiválogatással történő összeépítésre mutatunk be három lehetséges esetet.

Az egyes összeépítések elnevezése utal arra, hogy melyik tételt melyik tételbe integráljuk. Például a maximumkiválasztás és kiválogatás összeépítése alatt azt az esetet értjük, amikor egy kiválogatáson belül valósítunk meg egy maximumkiválasztást. A kiválogatás és maximumkiválasztás összeépítése viszont arra utal, hogy a maximumkiválasztásba építünk be egy kiválogatást. Tehát az elnevezésnél az első tag a belső, míg a második a külső szerkezeti elemre utal.

Másolás és sorozatszámítás összeépítése

A másolás programozási tétel (ld. 2.2.1. fejezet) egy tömb minden elemét átmásolja egy másik tömbbe úgy, hogy a másolás közben egy adott függvény esetlegesen módosítást hajt végre az elemeken. A sorozatszámítás tétel (ld. 2.1.1. fejezet) pedig egy tömb elemei között végrehajt egy adott műveletet és az így kapott eredményt adja vissza kimenetként. A másolás és sorozatszámítás tétel összeépítését olyan esetekben használjuk, amikor egy tömb minden elemének egy megadott függvénnyel való módosításának eredményeként előálló új tömb minden eleme között végrehajtunk egy műveletet, és ezen művelet eredményét keressük. Például egy tömb elemeinek abszolút értékeit kívánjuk összeadni.

A másolás és sorozatszámítás összeépítését megvalósító algoritmust úgy szeretnénk megalkotni, hogy ne kelljen egymást követően két bejárást elvégezni, egyet a másolás, egyet pedig a sorozatszámítás megvalósítása érdekében. Másrészt olyan megvalósítást kívánunk adni, amelynél nem kell a memóriában átmeneti – az eredmény szempontjából feleslegesnek tűnő – tömböt eltárolni. Egy ilyen megvalósítást mutat be a 2.24. algoritmus.

Az algoritmus bemenete az x feldolgozandó tömb, melynek elemszámát is ismerjük. Bemenetként adjuk meg még a másolás résznél alkalmazandó f függvényt (vagy műveletet). Az algoritmus visszatérési értéke egyetlen érték lesz. Az algoritmus megvalósítása során ismernünk kell még, hogy a sorozatszámítás milyen műveletet hajt végre az elemek között, ezt jelöli a $\oplus$ operátor. Ehhez a művelethez tartozik egy kezdeti érték is ($\mathit{érték}_0$), ahogy ezt a sorozatszámítás tételnél korábban megismertük. A visszaadott *érték* változóra igaz, hogy

$$\mathit{érték} = \mathit{érték}_0 \oplus f(x[1]) \oplus f(x[2]) \oplus \ldots \oplus f(x[n]). \tag{2.4}$$

2.24. Algoritmus Másolás és sorozatszámítás összeépítése

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), f – **művelet**
Kimenet: *érték* – **T**

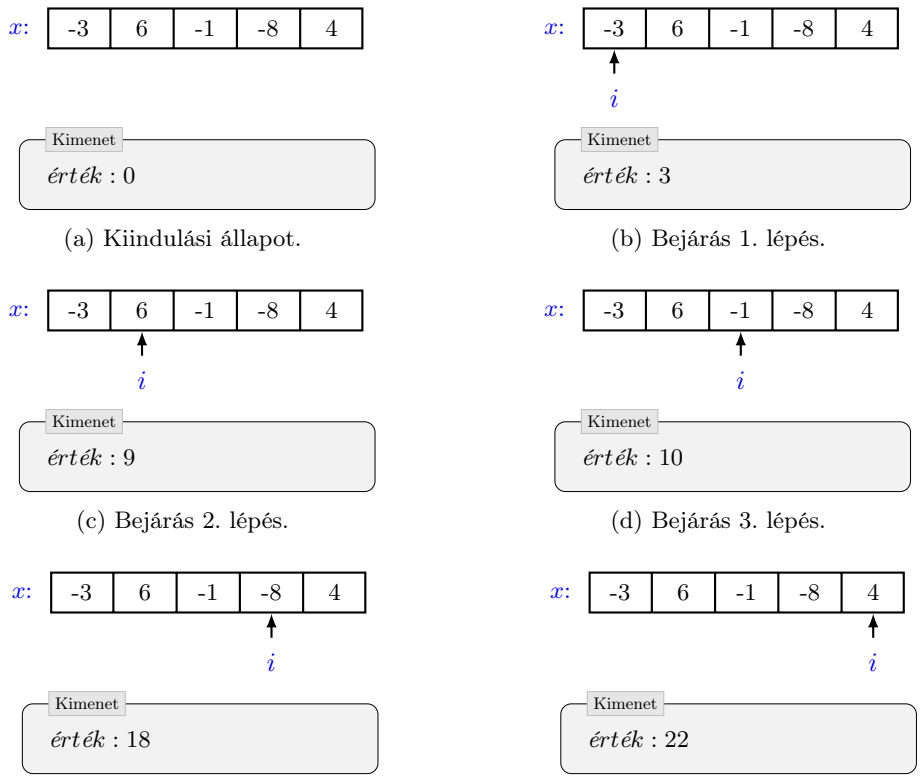
```
1: függvény MÁSOLÁS_SOROZATSZÁMÍTÁS(x : T tömb, n : egész, f : művelet)
2:     érték ← érték_0
3:     ciklus i ← 1-től n-ig
4:         érték ← érték ⊕ f(x[i])
5:     ciklus vége
6:     vissza érték
7: függvény vége
```

Felhasznált változók és függvények

- x: Feldolgozandó tömb.
- n: Az x tömb mérete.
- f: Függvény, amely az x elemein értelmezett.
- $\oplus$: Művelet, amelyet az x tömbből képzett összes $f(x[i])$ elem között hajtunk végre.
- *érték*: Az $\oplus$ műveletnek a tömb összes módosított $f(x[i])$ elemére való alkalmazását követően előálló eredmény.
- $\mathit{érték}_0$: Az alkalmazott $\oplus$ művelettől függő kiindulási érték.

A 2.24. algoritmus lényegében egyetlen sorban különbözik a sorozatszámítás programozási tételt megvalósító 2.1. algoritmustól. A 4. sorban a tömb aktuális $x[i]$ eleme helyett, a tömb aktuális elemének az f által módosított $f(x[i])$ értékét vesszük figyelembe.

2.18. Példa. A 2.17. ábrán látható példában azt követhetjük nyomon, hogy egy egész számokat tartalmazó öt elemű tömb esetén miként lehetséges a számok abszolút értékeinek összegét meghatározni a 2.24. algoritmus használatával. ¶



(a) Kiindulási állapot.

(b) Bejárás 1. lépés.

(c) Bejárás 2. lépés.

(d) Bejárás 3. lépés.

(e) Bejárás 4. lépés.

(f) Bejárás 5. lépés.

2.17. ábra. Másolás és sorzatszámítás összeépítése. A példában egy öt elemű tömb elemei abszolút értékének összegét számítjuk ki, az eredmény az *érték* változóba kerül. Mivel az összeadás műveletet használjuk, ezért $érték_0 = 0$.

Másolás és maximumkiválasztás összeépítése

A másolás és maximumkiválasztás tételek összeépítését olyan esetekben használjuk, amikor egy tömb elemeinek egy függvény által módosított értékei közül szeretnénk a maximális értékű elemet kiválasztani. Ezt megvalósíthatnánk úgy, hogy a másolás tétel (ld. 2.2.1. fejezet) alkalmazásával előállítunk egy új tömböt, melyben az eredeti tömb elemeinek f függvény által módosított értékeit tároljuk el. Ezt követően pedig egy maximumkiválasztás tétel (ld. 2.1.6. fejezet) használatával meghatározhatnánk a módosított értékeket tartalmazó tömbből a maximális értéket és annak első előfordulási helyét. Ha ezt a megközelítést használnánk, akkor két bejáró ciklusra lenne szükségünk, illetve egy átmeneti tömböt is el kéne tárolnunk a memóriában. Ennél futási idő és memória felhasználás szempontjából is hatékonyabb eljárást mutatunk be a 2.25. algoritmusban.

2.25. Algoritmus Másolás és maximumkiválasztás összeépítése

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), f – **művelet**; ahol **T** összehasonlítható
Kimenet: max – **egész**, $maxérték$ – **T**

```
1:  függvény MÁSOLÁS_MAXIMUMKIVÁLASZTÁS(x : T tömb, n : egész, f : művelet)
2:      max ← 1
3:      maxérték ← f(x[1])
4:      ciklus i ← 2-től n-ig
5:          segéd ← f(x[i])
6:          ha maxérték < segéd akkor
7:              max ← i
8:              maxérték ← segéd
9:          elágazás vége
10:     ciklus vége
11:     vissza (max, maxérték)
12: függvény vége
```

Felhasznált változók és függvények

- x: Feldolgozandó tömb.
- n: Az x tömb mérete.
- f: Függvény, amely az x minden egyes elemén értelmezett. Olyan értékeket állít elő, amelyek egymással összehasonlíthatók.
- max: A x tömb azon elemének indexe, mely esetén az $f(x[max])$ érték maximális. Ha több ilyen elem is létezik, akkor ezek közül a legkisebb indexű.
- $maxérték$: $maxérték = f(x[max])$

Az algoritmus bemenete a feldolgozandó x tömb, melynek az elemszámát is ismerjük. Ezen kívül adott még az az f függvény, amelyet meghívunk az x tömb minden elemére, majd az $f(x[i])$ elemek közül választjuk ki a maximális értékű elemet. A max kimeneti változó határozza meg, hogy mi a megtalált maximális elem indexe, azaz $f(x[max])$ nagyobb vagy egyenlő minden más $f(x[i])$ értéknél. Az algoritmus futása során meghatározzuk a maximális értéket, ezért ezt a kiszámított értéket kimenetként is visszaadjuk ($maxérték$).

Az algoritmus kezdetén – a maximumkiválasztás 2.10. algoritmusához hasonlóan – a tömb első elemét tekintjük maximálisnak, ezért a max változót 1-gyel tesszük egyenlővé (ld. 2. sor). Annak érdekében, hogy az f függvényt minél kevesebbszer kelljen meghívni az algoritmus futása során – hiszen ennek a függvénynek a kiértékelése jelentős időbe is telhet –, minden egyes tömbelemre pontosan egyetlen függvényhívást valósítunk meg. Emiatt az $f(x[1])$ értéket el kell tárolnunk a $maxérték$ változóban, mivel kezdetben az első elemet tekintjük maximálisnak (ld. 3. sor).

Az inicializáló lépések után következik a tömb többi elemének a bejárása, amit a 4. és 10. sorok közötti számlálós ciklus valósít meg. Az f függvény korábban már említett kiértékelésének minimalizálása érdekében az aktuális $x[i]$ elem esetén kiszámítjuk és eltároljuk az $f(x[i])$ értéket (ld. 5. sor). A 6. sorban megvizsgáljuk, hogy az aktuális elem nagyobb-e mint a $maxérték$. Ha nagyobb, akkor új maximális elemet találtunk, ezért a max és a $maxérték$ változókat aktualizálnunk kell. Az algoritmus végén visszaadjuk a meghatározott kimeneti értékeket (ld. 11. sor).

Megjegyzés

Érdemes megvizsgálni az algoritmus azon változatát, amikor a kiszámított $f(x[i])$ értékeket nem tároljuk el segédváltozókban, ahogy az a 2.26. algoritmusban látható. A 2.25. algoritmusban egy n elemű tömb feldolgozása során összesen n-szer hívjuk meg az f függvényt, míg a rövidebbnek és ezért egyszerűbbnek tűnő 2.26. algoritmusban $2 \cdot (n-1)$-szer. Emiatt az utóbbi algoritmus futási ideje jelentősen megnövekedhet.

2.26. Algoritmus Másolás és maximumkiválasztás összeépítése (módosított, kevésbé hatékony változat)

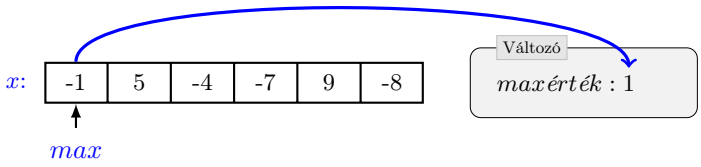
Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), f – **művelet**; ahol **T** összehasonlítható
Kimenet: max – **egész**

```
1: függvény MÁSOLÁS_MAXIMUMKIVÁLASZTÁS_MÓDOSÍTOTT(x : T tömb, n : egész, f : művelet)
2:     max ← 1
3:     ciklus i ← 2-től n-ig
4:         ha f(x[max]) < f(x[i]) akkor
5:             max ← i
6:         elágazás vége
7:     ciklus vége
8:     vissza max
9: függvény vége
```

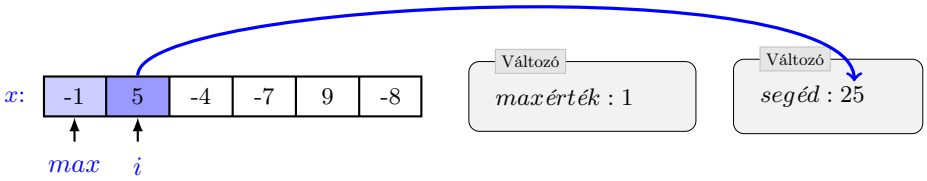
Felhasznált változók és függvények

- x: A feldolgozandó tömb.
- n: Az x tömb mérete.
- f: Függvény, amely az x minden egyes elemén értelmezett. Olyan értékeket állít elő, amelyek egymással összehasonlíthatók.
- max: Az x azon elemének indexe, mely esetén az $f(x[max])$ érték maximális. Ha több ilyen elem is létezik, akkor ezek közül a legkisebb indexű.

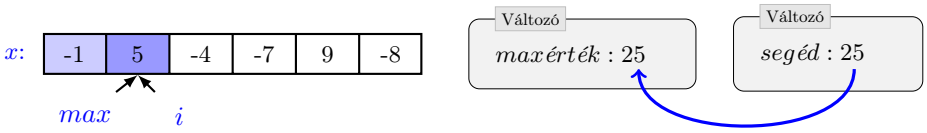
2.19. Példa. Feladatunk, hogy egy hat elemű, egész számokat tartalmazó tömbből kiválasszuk azt az elemet, melynek a négyzete a legnagyobb. A feladat megoldásához a másolás és maximumkiválasztás tételek összeépítésének 2.25. algoritmusát használjuk. A megoldás során az f függvény a négyzetfüggvény. A megoldás lépésenkénti menetét a 2.18. ábra szemlélteti. ¶



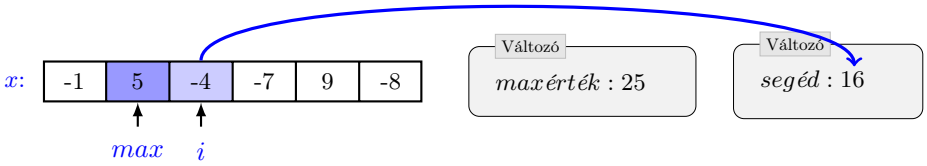
(a) Első lépésként a *max* változó az x tömb első elemét indexeli, a *maxérték* pedig felveszi az első elem négyzetének értékét.



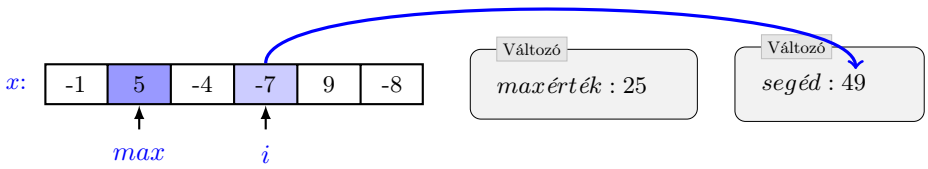
(b) Megkezdődik a tömb elemeinek bejárása, ennek érdekében az i ciklusváltozó értéke 2 lesz. A *segéd* változóba bekerül a második elem négyzetének értéke, ami nagyobb, mint a jelenlegi *maxérték*.



(c) Mivel a *segéd* nagyobb volt, mint a *maxérték* ezért a *max* felveszi az i értékét, a *maxérték* pedig a *segéd* értékét kapja meg.

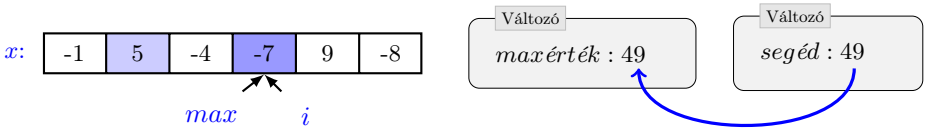


(d) Folytatódik a bejárás a 3. elemmel. A 3. elem négyzete kisebb mint a *max* indexű elem négyzete.

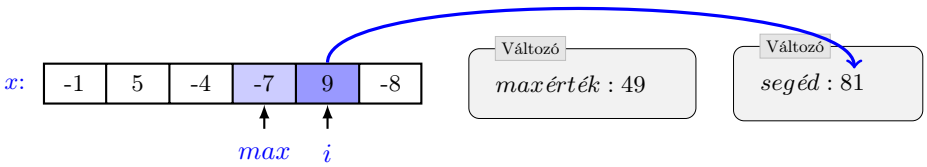


(e) A 4. elem négyzete nagyobb, mint $x[max]$ négyzete.

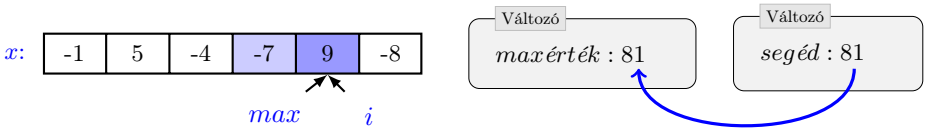
2.18. ábra. Feladatunk, hogy megkeressük az x azon elemét, melynek legnagyobb a négyzete. A megoldáshoz a másolás és maximumkiválasztás tételek összeépítését használjuk úgy, hogy az f függvény a négyzetfüggvény.



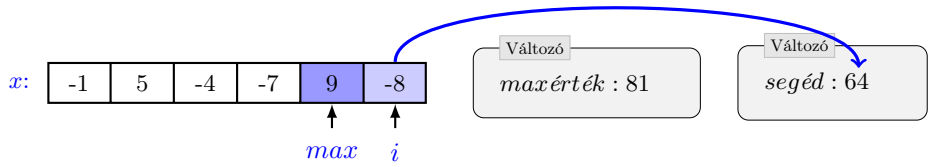
(f) Módosul a *max* és a *maxérték* értéke is.



(g) Az 5. elem négyzete nagyobb a jelenlegi *maxérték*-nél.



(h) Ismét módosítani kell a *max* és a *maxérték* változókat.



(i) Az utolsó elem négyzete nem nagyobb az $x[max]$ négyzeténél. A bejárás véget ér, az algoritmus visszaadja a *max* és a *maxérték* változók aktuális értékét.

2.18. ábra. Feladatunk, hogy megkeressük az x azon elemét, melynek legnagyobb a négyzete. A megoldáshoz a másolás és maximumkiválasztás tételek összeépítését használjuk úgy, hogy az f függvény a négyzetfüggvény (folyt.).

Megszámolás és keresés összeépítése

A 2.1.4. fejezetben megismertük a lineáris keresés programozási tételt, mely arról adott információt, hogy egy tömbben van-e P tulajdonságú elem, és ha van, akkor hol találjuk meg az első ilyet. A megszámolás és keresés programozás tételek összeépítését viszont olyan esetekben használjuk, amikor azt szeretnénk megvizsgálni, hogy egy tömbben van-e legalább k darab P tulajdonságú elem, és ha van, akkor hol találjuk meg a tömbben a k-adikat. Ezt úgy tudjuk megvalósítani, hogy a lineáris keresés programozási tételbe integrálunk egy megszámlálás tételt, azaz a keresés során folyamatosan számoljuk a már megtalált P tulajdonságú elemeket. A feladatot 2.27. algoritmussal oldjuk meg.

2.27. Algoritmus Megszámolás és keresés összeépítése

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai**, k – **egész**
Kimenet: van – **logikai**, idx – **egész**

```
1:  függvény MEGSZÁMOLÁS_KERESÉS(x : T tömb, n : egész, P : logikai, k : egész)
2:      db ← 0
3:      i ← 0
4:      ciklus amíg (i < n) ∧ (db < k)
5:          i ← i + 1
6:          ha P(x[i]) akkor
7:              db ← db + 1
8:          elágazás vége
9:      ciklus vége
10:     van ← (db = k)
11:     ha van akkor
12:         idx ← i
13:         vissza (van, idx)
14:     különben
15:         vissza van
16:     elágazás vége
17: függvény vége
```

Felhasznált változók és függvények

- x: Feldolgozandó tömb.
- n: Az x tömb mérete.
- P: Tulajdonságfüggvény.
- k: Azt vizsgáljuk, hogy van-e az x tömbben legalább k darab P tulajdonságú elem.
- van: Logikai típusú változó, melynek pontosan akkor ***igaz*** az értéke, ha van az x tömbben legalább k darab P tulajdonságú elem.
- idx: Csak akkor kap értéket, ha a van változó ***igaz*** értékű. Ebben az esetben megmutatja, hogy az x hányadik eleme a k-adik P tulajdonságú elem.

Az algoritmus egyik bemenete a feldolgozandó x tömb, melynek ismerjük az elemszámát is. Bemenetként adjuk meg még a vizsgálandó P tulajdonságot, valamint azt a k értéket, ahányadik P tulajdonságú elemet keressük az x tömbben. Az algoritmus van logikai kimenete adja meg, hogy találtunk-e legalább k darab P tulajdonságú elemet az x tömbben. Amennyiben van értéke ***igaz*** az algoritmus végén, akkor visszaadjuk a k-adik P tulajdonságú elem idx indexét is. Míg van ***hamis*** értéke esetén nem adunk vissza semmilyen idx értéket.

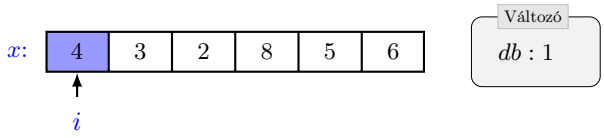
Az algoritmus elején, a 2. sorban létrehozunk egy db változót, mely mindig a már megtalált P tulajdonságú elemek számát tárolja el. Kezdeti értéke természetesen 0 lesz. A 3. sorban létrehozzuk az i változót, melyet a tömb bejárása során a tömb indexelésére használunk majd. Kivételesen kezdeti értékként nullát adunk neki, így a bejárást megvalósító ciklusban első lépésként mindig i értékét fogjuk eggyel növelni (ld. 5. sor).

Az inicializálásokat követően megkezdjük az x tömb bejárását, amit a 4. és 9. sorok közötti ciklussal valósítunk meg. A ciklusba való belépésnek, illetve bennmaradásnak két feltétele van. Egyrészt az $i < n$ feltételnek kell teljesülnie, mivel egyéb esetben a cikluson belül i értékének növelését követően már az x tömbön kívülre indexelnénk i-vel. Másrészt vizsgáljuk, hogy találtunk-e már k darab P tulajdonságú elemet a tömbben. Ha még nem, akkor a db változó kisebb lesz a k változónál, és tovább kell folytatnunk

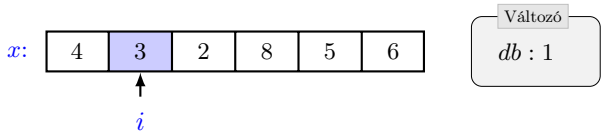
a keresést. Az i index növelését követően megvizsgáljuk, hogy az aktuális $x[i]$ tömbelem P tulajdonságú-e (ld. 6. sor). Amennyiben P tulajdonságú elemet találtunk, akkor a megtalált P tulajdonságú elemek számát tároló db változó értékét eggyel növeljük (ld. 7. sor).

A ciklusból kilépve megvizsgáljuk, hogy találtunk-e a tömbben k darab P tulajdonságú elemet. Erre egyértelmű választ ad a $db = k$ vizsgálat, aminek eredményét el is tároljuk a *van* változóban (ld. 10. sor). Fontos megjegyezni, hogy nem az $i < n$ feltételt vizsgáljuk, mivel ha a tömb utolsó eleme épp a k-adik P tulajdonságú elem a tömbben, akkor bár találtunk k darab P tulajdonságú elemet, az $i < n$ feltétel mégis ***hamis*** értéket ad. Ha a *van* változó ***igaz*** volt (11. sor), akkor a k-adik P tulajdonságú elem pont a jelenlegi i-edik helyen van a tömbben, ezért az *idx* változónak átadjuk i értékét (12. sor), majd visszatérünk a két kimeneti változóval (13. sor). Ha *van* ***hamis*** (14. sor), akkor csak a *van* értékét kell visszaadnunk (15. sor).

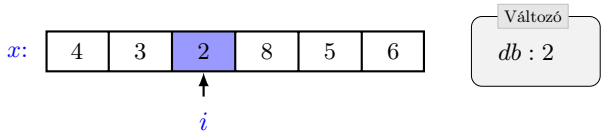
2.20. Példa. A 2.19. ábrán láthatunk egy példát arra, amikor egy tömbben keressük a harmadik páros elemet. ¶



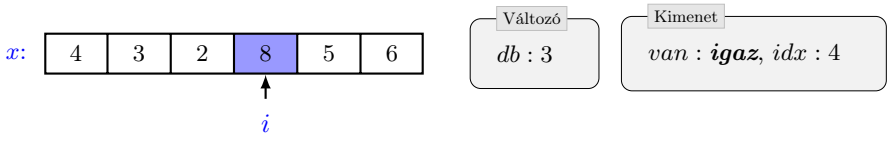
(a) A tömb első eleme páros, ezért db értékét 1-re növeljük.



(b) A második tömbbeli elem nem páros, ezért módosítások nélkül folytatjuk a bejárást.



(c) A harmadik tömbelem páros, ezért db értéke eggyel növekszik.



(d) A negyedik elem is páros, így db értéke már 3 lesz. Mivel megtaláltuk a harmadik páros elemet, ezért az algoritmus futása véget ér.

2.19. ábra. Példa a megszámolás és keresés programozási tételek összeépítésére. Feladatunk megkeresni az x tömb harmadik ($k = 3$) páros elemét, ha van egyáltalán három páros eleme.

Maximumkiválasztás és kiválogatás összeépítése

A 2.1.6. fejezetben megismert maximumkiválasztás programozási tétel használatával egy tömbben meghatározhatjuk, hogy melyik a legnagyobb értékű elem. Abban az esetben ha viszont több olyan elem is van a tömbben, ami a maximális értékkel egyenlő, akkor ezek közül egyet, a legkisebb indexűt adja meg a 2.10. algoritmus. Ha fontos számunkra, hogy az összes maximális értékű elem indexét ismerjük, akkor ezeket az indexeket ki kell válogatnunk egy újabb tömbbe. Ezt a problémát a maximumkiválasztás és kiválogatás tételek összeépítésével tudjuk megoldani, melyet a 2.28. algoritmussal valósítunk meg.

2.28. Algoritmus Maximumkiválasztás és kiválogatás összeépítése

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: db – **egész**, y – **egész tömb**, $maxérték$ – **T**

```
1:  függvény MAXIMUMKIVÁLOGATÁS(x : T tömb, n : egész)
2:      y ← LÉTREHOZ(egész)[n]
3:      maxérték ← x[1]
4:      db ← 1
5:      y[db] ← 1
6:      ciklus i ← 2-től n-ig
7:          ha x[i] > maxérték akkor
8:              maxérték ← x[i]
9:              db ← 1
10:             y[db] ← i
11:         különben
12:             ha x[i] = maxérték akkor
13:                 db ← db + 1
14:                 y[db] ← i
15:             elágazás vége
16:         elágazás vége
17:     ciklus vége
18:     vissza (db, y, maxérték)
19: függvény vége
```

Felhasznált változók és függvények

- x: Feldolgozandó tömb, melynek elemei összehasonlíthatóak.
- n: Az x tömb elemeinek száma.
- $maxérték$: Az x tömbben található maximális érték.
- db: Az x tömbben a $maxérték$ értékű elemek száma.
- y: Kimeneti n elemű tömb. Az első db darab elemében tároljuk el azon indexeket, ahol az x tömbben a $maxérték$ értékű elemet találjuk. Azaz minden $1 \leq i \leq db$ esetén $x\,[y[i]] = maxérték$.
- LÉTREHOZ(**egész**)[n]: Utasítás, mely létrehoz egy n elemű **egész** típusú tömböt.

Az algoritmus bemenete a feldolgozandó x tömb, melynek elemei összehasonlíthatók, hiszen más esetben nem tudnánk vizsgálni, hogy melyik elem nagyobb a többinél. Természetesen ismerjük az x tömb n elemszámát is. Kimenetként az x tömb maximális értékű elemeinek az indexét adjuk vissza az y tömbben. Az algoritmus elején nem tudjuk, hogy hány maximális értékű elemet fogunk találni, ezért az y mérete az x méretével lesz azonos. Meg kell azonban határoznunk, hogy az y tömbben eltárolt indexek közül hány darab hordoz valódi információt. Emiatt szükségünk van még egy db kimeneti változóra is, melynek értéke az algoritmus lefutása után megegyezik az x tömb maximális értékű elemeinek darabszámával. Az algoritmus végén az n elemű y tömbben az első db darab elem rendelkezik érdemi információval.

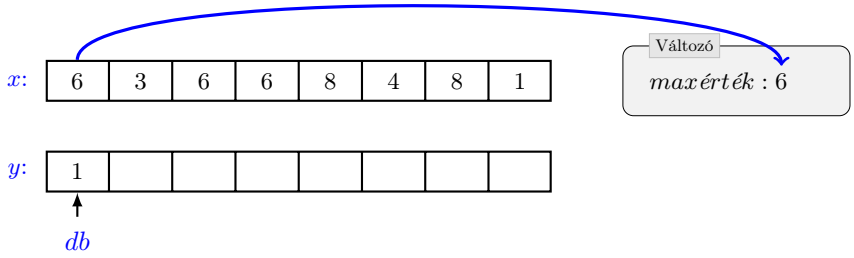
Első lépésként létre kell hoznunk az y tömböt (2. sor), melynek ugyanannyi eleme lesz, mint az x tömbnek. A későbbi lépések logikája nagy mértékben hasonlít a maximumkiválasztás 2.10. algoritmusának logikájához. Kiindulásként az első elemet tekintjük maximálisnak, ezért a $maxérték$ változónak átadjuk az első elem értékét (3. sor). Mivel itt egyetlen maximális értékünk van, ezért a db változót is egyre állítjuk (4. sor), illetve eltároljuk az y tömbbe a jelenlegi maximális értékű elem indexét, azaz az 1-et (5. sor).

Ezt követően be kell járnunk az x tömb többi elemét, amit a 6. és 17. sorok közötti ciklussal valósítunk meg. A bejárás során minden elemet összehasonlítunk az aktuális maximális értékkel ($max érték$), aminek három különböző eredménye lehet. Ha $x[i] < max érték$, akkor az aktuális elem nem maximális értékű, ezért nem kell semmit sem csinálnunk. Ha viszont $x[i] > max érték$ (7. sor), akkor olyan maximális elemet találtunk, ami minden korábbi elemnél biztosan nagyobb, ezért ez a megtalált elem lesz a $max érték$ (5. sor). Az új maximális értéket eddig csak egyszer találtuk meg az x tömbben – mégpedig az aktuális elemnél –, ezért a db változó értékét egyre változtatjuk (9. sor), valamit az y tömbben eltároljuk az aktuális elem indexét (10. sor). Abban az esetben, ha az aktuális $x[i]$ elem megegyezik az eddig vizsgált elemek maximumával (azaz $max érték$-kel) (12. sor), akkor új, de az eddigivel azonos értékű maximumot találtunk az x tömbben. Ilyenkor a megtalált maximális értékű elemek számát növeljük eggyel (13. sor) és az aktuális indexet eltároljuk az y tömbben (14. sor). A bejárás végén szükséges még a kimeneti változókat visszaadni (18. sor).

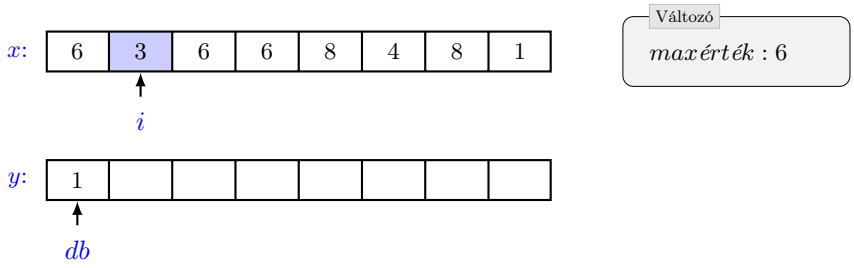
Megjegyzés

A maximumkiválasztás tétel 2.10. algoritmusában nem tároltuk el a maximális értéket, mivel a maximális értékű elem indexének ismeretében a maximális értéket is meg tudjuk határozni. A 2.28. algoritmusban viszont eltároljuk a maximális értéket, bár ez nem feltétlenül szükséges, csak így kissé áttekinthetőbb a kód. Vegyük észre, hogy a $max érték$ változó helyett használhattuk volna az $x\,[y[db]]$ kifejezést is, hiszen az y tömb első db darab eleme olyan indexeket tartalmaz, amely indexű elemei az x tömbnek pont a $max érték$ változóval egyeznek meg. Abban az esetben, ha nem használjuk a $max érték$ változót, akkor az algoritmus 3. és 8. sorai elhagyhatók a kódból, illetve kimenetként sem adjuk meg a maximális értéket.

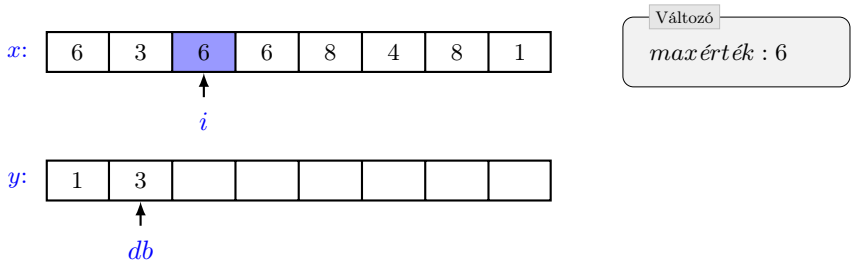
2.21. Példa. A 2.20. ábrán láthatunk egy példát, amelyben egy nyolc elemű tömbből válogatjuk ki a maximális értékű elemek indexeit. Érdemes megfigyelni, hogy a feldolgozás során az y tömbbe három elem is bekerül, majd egy újabb maximális érték megtalálásakor ezek elveszítik jelentőségüket. A bejárás végén három elem is van az y tömbben, de ezek közül csak az első kettő hordoz releváns információt. ¶



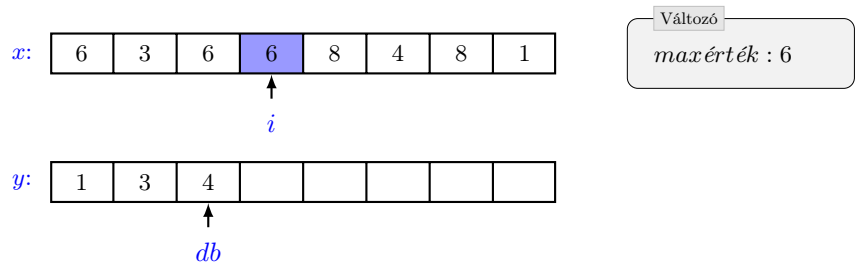
(a) Kiindulásként az első elemet tekintjük maximálisnak, ezért $x[1]$-et átmásoljuk *maxérték*-be, illetve az y tömbbe is bekerül az 1-es index.



(b) A második elem nem nagyobb a jelenlegi *maxérték*-nél, ezért semmit sem kell tenni.

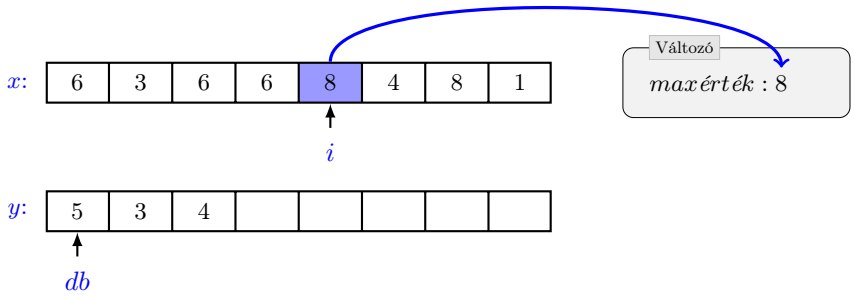


(c) A harmadik elem megegyezik a *maxérték*-kel, így a *db* értékét eggyel növeljük és az y-ba bekerül a hármas index is.

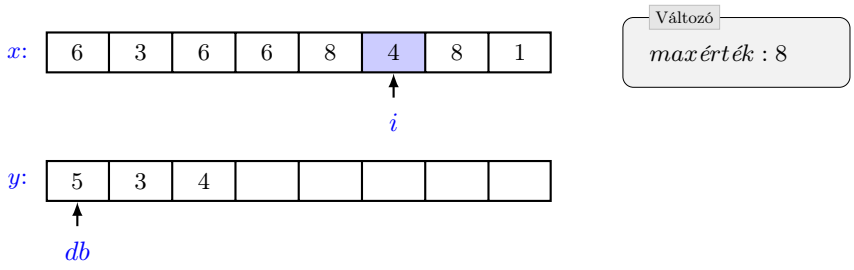


(d) A negyedik elemnél a harmadikhoz hasonlóan járunk el.

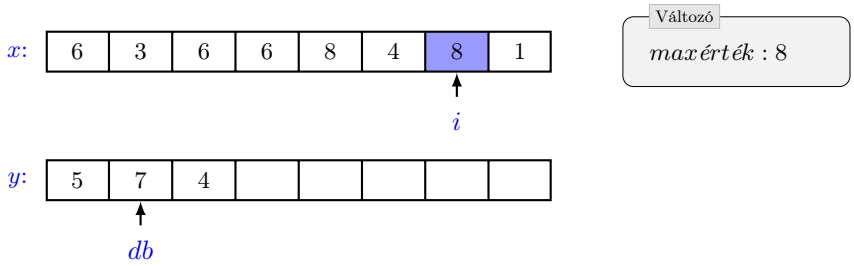
2.20. ábra. Maximumkiválasztás és kiválogatás összeépítése. Egy nyolc elemű tömbből kiválogatjuk a maximális értékű elemek indexeit.



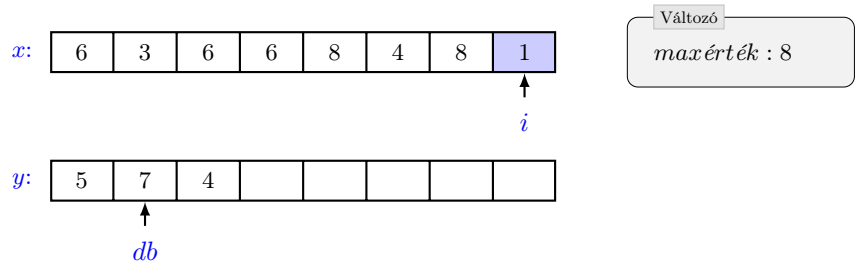
(e) Az ötödik elem nagyobb az eddigi *maxérték*-nél, ezért $x[5]$ kerül a *maxérték*-be. Mivel csak egyetlen maximális értékű elemet találtunk, ezért a *db* változó ismét 1 lesz, és az y tömb első helyére kerül a maximális értékű elem indexe.



(f) A hatodik elem kisebb a *maxérték*-nél, ezért semmi teendő nincs.



(g) A hetedik elem megegyezik a *maxérték*-kel, ezért *db* értékét eggyel növeljük és az y tömbbe bekerül a hetes érték is.



(h) A nyolcadik elem kisebb a *maxérték*-nél, ezért semmit sem teszünk. A bejárás végére az y tömbbe három elem is bekerült, de ebből csak az első kettő hordoz érdemi információt.

2.20. ábra. Maximumkiválasztás és kiválogatás összeépítése. Egy nyolc elemű tömbből kiválogatjuk a maximális értékű elemek indexeit (folyt.).

Kiválogatás és sorozatszámítás összeépítése

A 2.1.1. fejezetben megismert sorozatszámítás tétel meghatározza egy művelet eredményét, ha a műveletet egy tömb elemei között hajtjuk végre. Előállhat viszont olyan feladat is, amikor nem a feldolgozandó tömbünk összes elemére szeretnénk az adott műveletet elvégezni, hanem csak valamilyen adott P tulajdonságú elemekre. Ebben az esetben eljárhatunk úgy, hogy először a 2.2.2. fejezetben megismert kiválogatás tételt alkalmazzuk, azaz kigyűjtjük a P tulajdonságú elemeket egy átmeneti tömbbe, majd az átmeneti tömb minden egyes eleme között hajtjuk végre az adott műveletet. Másik lehetőségünk, hogy az említett két programozási tétel összeépítésével valósítjuk meg a feladatot. Ezt az utóbbi esetet mutatja be a 2.29. algoritmus.

2.29. Algoritmus Kiválogatás és sorozatszámítás összeépítése

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai**
Kimenet: *érték* – **T**

```
1: függvény KIVÁLOGATÁS_SOROZATSZÁMÍTÁS(x : T tömb, n : egész, P : logikai)
2:     érték ← érték_0
3:     ciklus i ← 1-től n-ig
4:         ha P(x[i]) akkor
5:             érték ← érték ⊕ x[i]
6:         elágazás vége
7:     ciklus vége
8:     vissza érték
9: függvény vége
```

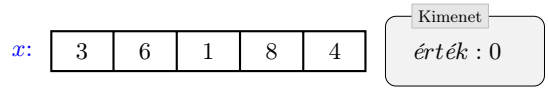
Felhasznált változók és függvények

- x: A feldolgozandó tömb.
- n: Az x tömb mérete.
- P: Tulajdonságfüggvény. Az algoritmusban az x tömb P tulajdonságú elemei között végezzük el az $\oplus$ műveletet.
- $\oplus$: **T** típusú értékek között értelmezett művelet.
- *érték*: Az x tömb minden P tulajdonságú eleme között elvégezve az $\oplus$ műveletet az előálló végeredmény értéke.
- *érték*$_0$: Az $\oplus$ művelethez tartozó kezdeti érték. Például összeadás esetén 0, szorzás esetén pedig 1.

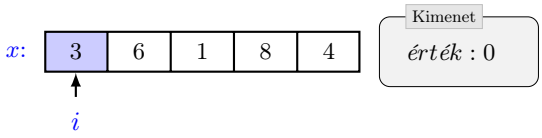
Az algoritmus bemenete a feldolgozandó x tömb, melynek ismerjük az elemszámát is. Ezeken kívül ismerjük még a vizsgálandó P tulajdonságot is. Az algoritmus kimenete az *érték* változó, melyet úgy kapunk meg, hogy a x tömb P tulajdonságú elemei között elvégezzük a megadott $\oplus$ műveletet.

Az algoritmus pszeudokódja majdnem teljes egészében megegyezik a sorozatszámítás 2.1. algoritmusával, ezért részletesen nem ismertetjük az egyes lépéseket, csak az egyetlen különbségre térünk ki. Az *érték* változó 5. sorbeli aktualizálása előtt meg kell vizsgálnunk, hogy az adott $x[i]$ elem teljesíti-e a P tulajdonságot (4. sor). Az *érték* módosítását csak akkor hajtjuk végre, ha P tulajdonságú az aktuális tömbelem.

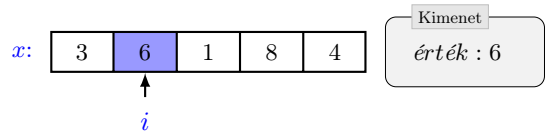
2.22. Példa. A 2.21. ábrán láthatunk egy példát arra, hogy a 2.29. algoritmus használatával hogyan lehet egy tömb páros értékű elemeinek összegét meghatározni. Ebben az esetben a P tulajdonság a párosság, az elvégzendő $\oplus$ művelet az összeadás, míg az *érték*$_0$ kezdeti érték a 0. ¶



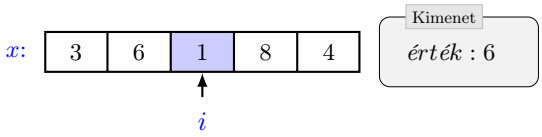
(a) Első lépésként inicializáljuk az *érték* változót. Mivel összeadás az elvégzendő művelet, ezért 0 lesz a kiindulás érték.



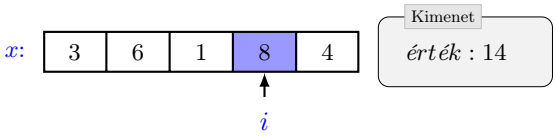
(b) Megkezdjük a tömb bejárást. Mivel az első elem páratlan, ezért nem kell hozzáadni az *érték* változóhoz.



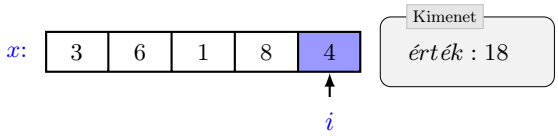
(c) A második elem páros, ezért hozzáadjuk az *érték* változóhoz.



(d) A harmadik elem páratlan, ezért nem teszünk semmit.



(e) A negyedik elem páros, így növeljük vele az *érték* változót.



(f) Az ötödik elem is páros, ezt is hozzáadjuk az *érték* változóhoz. Az algoritmus kimenete az előálló *érték* lesz.

2.21. ábra. Kiválogatás és sorozatszámítás tétel összeépítése. Egy tömb páros értékű elemeinek összegét számítjuk ki.

Kiválogatás és maximumkiválasztás összeépítése

A kiválogatás és maximumkiválasztás programozási tételek összeépítését olyan esetekben használjuk, amikor egy tömb adott P tulajdonságú elemei közül kell a maximális értékűt kiválasztani. Ezt a feladatot megoldhatnánk úgy, hogy először kiválogatjuk egy segédtömbbe a P tulajdonságú elemeket, ahogy ezt a 2.2.2. fejezetben tettük, majd a segédtömbben hajtunk végre egy a 2.1.6. fejezetben megismert maximumkiválasztást. Ehhez viszont egyszer be kéne járnunk a bemeneti tömböt, majd a létrehozott segédtömböt. Amennyiben a két említett programozási tételt összeépítjük, akkor olyan megoldást kaphatunk, amely csak egyszer járja be a bemeneti tömbünket, és a bejárás végén már szolgáltatja is az elvárt eredményeket. Az összeépítést a 2.30. algoritmussal valósítjuk meg.

2.30. Algoritmus Kiválogatás és maximumkiválasztás összeépítése

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai**; ahol **T** összehasonlítható
Kimenet: van – **logikai**, max – **egész**, $maxérték$ – **T**

```
1:  függvény KIVÁLOGATÁS_MAXIMUMKIVÁLASZTÁS(x : T tömb, n : egész, P : logikai)
2:      maxérték ← −∞
3:      ciklus i ← 1-től n-ig
4:          ha P(x[i]) ∧ (x[i] > maxérték) akkor
5:              max ← i
6:              maxérték ← x[i]
7:          elágazás vége
8:      ciklus vége
9:      van ← (maxérték > −∞)
10:     ha van akkor
11:         vissza (van, max, maxérték)
12:     különben
13:         vissza van
14:     elágazás vége
15: függvény vége
```

Felhasznált változók és függvények

- x: A feldolgozandó tömb, melynek elemei összehasonlíthatók.
- n: Az x tömb mérete.
- P: Tulajdonságfüggvény.
- van: Logikai érték, amely pontosan akkor ***igaz***, ha van az x tömbben legalább egy P tulajdonságú elem.
- max: Ha van az x tömbben P tulajdonságú elem, akkor az ilyen elemek közül a legnagyobb értékű indexe. Ha több legnagyobb értékű elem is van a P tulajdonságúak között, akkor a legelőször előforduló (legkisebb indexű) ilyen elem indexe.
- $maxérték$: Ha van az x tömbben P tulajdonságú elem, akkor az ilyen elemek közül a legnagyobb elem értéke.

Az algoritmus bemenete a feldolgozandó x tömb, melynek ismerjük az n elemszámát is. Mivel az elemek közül majd egy maximális elemet akarunk kiválasztani, ezért az elemeknek összehasonlíthatóknak kell lenniük. Ismerjük még a vizsgálandó P tulajdonságot is. Az algoritmus első kimenete a van logikai változó. Erre azért van szükség, mert ha esetleg egyetlen P tulajdonságú elem sincs az x tömbben, akkor nem tudunk közöttük maximális értéket sem meghatározni. Tehát a van változó pontosan akkor lesz ***igaz***, ha van legalább egy P tulajdonságú elem az x tömbben, és csak van ***igaz*** értéke esetén fogjuk a további kimeneti változókat előállítani. Második kimeneti változónk a max index, mely annak az x-beli elemnek az indexét adja meg, amely elem a P tulajdonságú elemek között a legnagyobb értékű. Amennyiben több ilyen maximális elem is van a bemeneti tömbben, akkor ezek közül a legelső indexe lesz a max értéke. Az algoritmusban használni fogjuk a $maxérték$ változót is, ezért azt kimenetként is megadhatjuk.

Az algoritmus első lépése, hogy a $maxérték$ változó értékét $-\infty$-re[10] állítjuk. Erre azért van szükség, mert a tömb bejárása során más gondolatmenetet kell követnünk, mint amit a maximumkiválasztás tétel

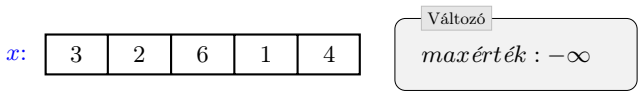
[10] $-\infty$ egy olyan érték, amely minden tömbbeli elem értékénél biztosan kisebb.

2.10. algoritmusánál figyelembe vettünk. Most nem mondhatjuk azt, hogy tekintsük kezdetben maximális elemnek az első elemet, mert lehet, hogy ez az elem nem P tulajdonságú. Azt mondhatnánk esetleg, hogy tekintsük kezdetben maximális elemnek a tömb első P tulajdonságú elemét, de nem tudjuk, hogy melyik ez, sőt lehetséges, hogy egyáltalán nincs a tömbben P tulajdonságú elem. Ezért inkább kezdetben nem tekintünk egyetlen elemet sem maximálisnak, hanem értelmezünk egy olyan maximális értéket tároló *maxérték* változót, amelynek a kezdeti értéke egy olyan kicsi érték, amelynél minden tömbbeli elem biztosan nagyobb.

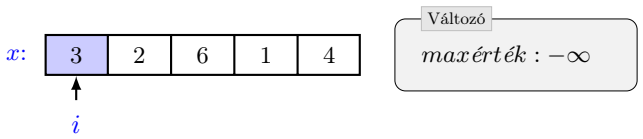
Ezt követően bejárjuk az x tömböt a 3. és 8. sorok közötti ciklussal. A cikluson belül az aktuális $x[i]$ elem esetén megvizsgáljuk, hogy P tulajdonságú-e, és ha az, akkor megnézzük, hogy nagyobb-e az értéke a *maxérték* változó aktuális értékénél (ld. 4. sor). (Vegyük észre, hogy ha épp az első P tulajdonságú elemnél tartunk a bejárás során, akkor az biztos nagyobb lesz, mint a *maxérték* addigi $-\infty$ értéke, tehát tényleg az első P tulajdonságú elemet fogjuk először maximálisnak tekinteni.) Ha olyan elemet találtunk, amely megfelel a megfogalmazott két feltételnek ez azt jelenti, hogy a P tulajdonságú elemek között új maximumot találtunk, ezért megváltoztatjuk a *max* és *maxérték* változók értékeit (5. és 6. sorok).

A ciklusból kilépve először ellenőriznünk kell, hogy találtunk-e az x tömbben P elemet. Ezt eldönthetjük olyan módon, hogy megvizsgáljuk a *maxérték* változó értékét. Ha $maxérték = -\infty$, akkor nem találtunk, hiszen – amint korábban már említettük – találat esetén biztos megváltozott volna a *maxérték* változó értéke. Szóval a *van* kimenetnek átadjuk a $maxérték \neq -\infty$ értékét (9. sor). Ezután megvizsgáljuk a *van* változó értékét (10. sor), ha ***igaz***, akkor három kimeneti változót adunk vissza (11. sor), egyébként pedig csak egyet (13. sor).

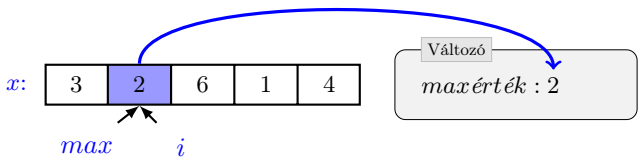
2.23. Példa. A 2.22. ábrán bemutatunk egy példát, ahol egy egész számokat tartalmazó tömbből kiválasztjuk a páros elemek maximumát. ¶



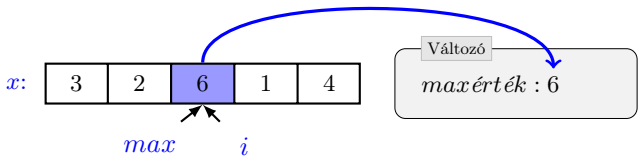
(a) Első lépésként a *maxérték* változó megkapja a $-\infty$ értéket.



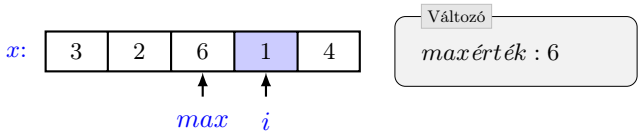
(b) Az első tömbbeli elem nem páros, ezért nem teszünk vele semmit.



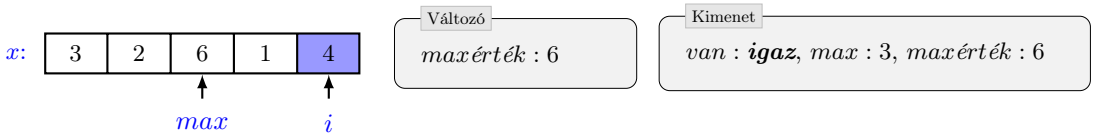
(c) A második elem páros és nagyobb, mint a korábbi *maxérték*, ezért a *max* felveszi az *i* értékét, illetve *maxérték* megkapja az $x[2]$ értékét.



(d) A harmadik elem is páros és nagyobb a korábbi *maxérték*-nél. Így *max* és *maxérték* ismét módosul.



(e) A negyedik elem páratlan, ezért semmit nem kell tennünk.



(f) Az ötödik elem páros, de kisebb mint *maxérték*, ezért nem történik semmi. Mivel a bejárás végére értünk meghatározzuk *van* értékét, ami ***igaz*** lesz.

2.22. ábra. Kiválogatás és maximumkiválasztás tételek összeépítése. Egy egész számokat tartalmazó tömbből ki kell választanunk a páros számok maximumát.

Kiválogatás és másolás összeépítése

A kiválogatás és másolás tétel összeépítése egy – a korábbiak ismeretében már – nagyon egyszerű feladatra ad megoldást. A 2.2.2. fejezetben megismert kiválogatást úgy akarjuk módosítani, hogy ha egy elem P tulajdonságú, azaz kiválogatásra kerül, akkor ne az elemet, hanem annak valamely f függvény által módosított értékét másoljuk be a kimeneti tömbbe. Ehhez a kiválogatás 2.12. algoritmusát egyetlen sorban kell módosítani, amint az a 2.31. algoritmusban látható.

2.31. Algoritmus Kiválogatás és másolás összeépítése

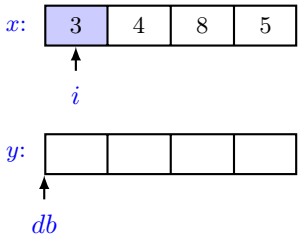
Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), P – **logikai**, f – **művelet**
Kimenet: db – **egész**, y – **T tömb**

```
1:  függvény KIVÁLOGATÁS_MÁSOLÁS(x : T tömb, n : egész, P : logikai, f : művelet)
2:      y ← LÉTREHOZ(T)[n]
3:      db ← 0
4:      ciklus i ← 1-től n-ig
5:          ha P(x[i]) akkor
6:              db ← db + 1
7:              y[db] ← f(x[i])
8:          elágazás vége
9:      ciklus vége
10:     vissza (db, y)
11: függvény vége
```

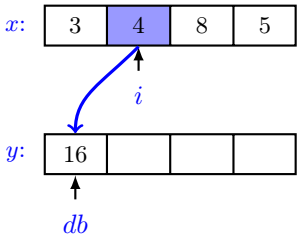
Felhasznált változók és függvények

- x: Feldolgozandó tömb.
- n: Az x tömb elemeinek száma.
- P: Tulajdonságfüggvény.
- f: Művelet, amely **T** tulajdonságú elemeken értelmezett.
- db: Az x tömb P tulajdonságú elemeinek száma.
- y: Kimeneti n elemű tömb. Az y tömb első db darab eleme az x tömb P tulajdonságú elemeiből az f művelettel képzett elem.
- LÉTREHOZ(**T**)[n]: Utasítás, mely létrehoz egy n elemű **T** típusú tömböt.

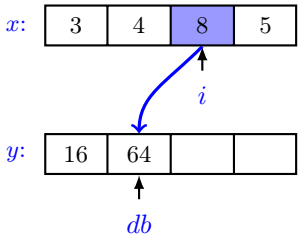
2.24. Példa. A 2.23. ábrán végig követhetjük, hogy a négy elemű egészeket tartalmazó x-ből miként válogatjuk ki a páros elemek négyzetét az y tömbbe. ¶



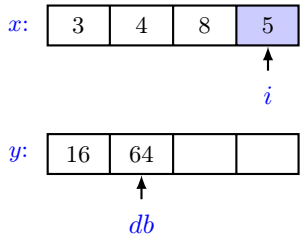
(a) Bejárás 1. lépés.



(b) Bejárás 2. lépés.



(c) Bejárás 3. lépés.



(d) Bejárás 4. lépés.

2.23. ábra. Kiválogatás és másolás tétel összeépítése. Az x tömb páros elemeinek négyzetét másoljuk át az y tömbbe.

Feladatok programozási tételekkel

1. **Egy kétdimenziós egész számokat tartalmazó tömb melyik sorában található a legtöbb nulla?**

   Végigmegyünk a tömb összes során és minden sorban megszámoljuk a nulla elemeket. Ezt követően kiválasztjuk, hogy melyik sorban van a legtöbb nullás.

Bemenet: $x-$**egész kétdimenziós tömb**, $m-$**egész** (*tömb sorszáma*), $n-$**egész** (*tömb oszlopszáma*)
Kimenet: $max-$**egész**

```
1:  függvény LegtöbbNullaSorIndexe(x : egész kétdimenziós tömb, m : egész, n : egész)
2:      max ← 0
3:      maxérték ← 0
4:      ciklus i ← 1-től m-ig
5:          db ← 0
6:          ciklus j ← 1-től n-ig
7:              ha x[i, j] = 0 akkor
8:                  db ← db + 1
9:              elágazás vége
10:         ciklus vége
11:         ha db > maxérték akkor
12:             max ← i
13:             maxérték ← db
14:         elágazás vége
15:     ciklus vége
16:     vissza max
17: függvény vége
```

Felhasznált változók és függvények

- x: Kétdimenziós egész értékeket tartalmazó tömb.
- m: Az x tömb sorainak száma.
- n: Az x tömb oszlopainak száma.
- max: Annak a sornak az indexe, ahol a legtöbb nullás elem van x -ben. Ha több ilyen sor is van, akkor közülük a legelső indexe.
- db: Számláló, mely megadja, hogy x aktuálisan vizsgált sorában hány nulla található.
- $maxérték$: Megadja, hogy x legtöbb nullát tartalmazó sorában hány darab nullás van.

---

2. **Egy számokat tartalmazó kétdimenziós tömbnek van-e olyan sora, ami csak nullákat tartalmaz?**

   Az aktuális soron belül egy eldöntéssel megadjuk, hogy minden elem nulla-e. Egy másik eldöntés megadja, hogy van-e olyan sor, ahol minden elem nulla.

Bemenet: x – **egész kétdimenziós tömb**, m – **egész** (*tömb sorszáma*), n – **egész** (*tömb oszlopszáma*)
Kimenet: van – **logikai**

```
1:  függvény VanECsakNullaSor(x : egész kétdimenziós tömb, m : egész, n : egész)
2:      i ← 1
3:      van ← hamis
4:      ciklus amíg (i ≤ m) ∧ ¬van
5:          j ← 1
6:          ciklus amíg (j ≤ n) ∧ x[i, j] = 0
7:              j ← j + 1
8:          ciklus vége
9:          van ← (j > n)
10:         i ← i + 1
11:     ciklus vége
12:     vissza van
13: függvény vége
```

Felhasznált változók és függvények

- x: Kétdimenziós egész értékeket tartalmazó tömb.
- m: Az x tömb sorainak száma.
- n: Az x tömb oszlopainak száma.
- van: Logikai változó, amely pontosan akkor igaz, ha van olyan sor az x tömbben, aminek minden eleme nulla.

---

3. **Van-e két olyan sor egy számokat tartalmazó kétdimenziós tömbben, melyek ugyanannyi nullát tartalmaznak?**

   Megszámoljuk, hogy melyik sorban hány darab nulla van, és két egymásba ágyazott eldöntéssel megnézzük, hogy van-e két sor ugyanannnyi nullával.

Bemenet: x – **egész kétdimenziós tömb**, m – **egész** (*tömb sorszáma*), n – **egész** (*tömb oszlopszáma*)
Kimenet: van – **logikai**

```
1:  függvény VanEKétUgyanannyiNullaSor(x : egész kétdimenziós tömb, m : egész, n : egész)
2:      db ← Létrehoz(egész)[m]
3:      db[1] ← 0
4:      ciklus j ← 1-től n-ig
5:          ha x[1, j] = 0 akkor
6:              db[1] ← db[1] + 1
7:          elágazás vége
8:      ciklus vége
9:      van ← hamis
10:     i ← 2
11:     ciklus amíg (i ≤ m) ∧ ¬van
12:         db[i] ← 0
13:         ciklus j ← 1-től n-ig
14:             ha x[i, j] = 0 akkor
15:                 db[i] ← db[i] + 1
16:             elágazás vége
17:         ciklus vége
18:         j ← 1
19:         ciklus amíg (j < i) ∧ (db[i] ≠ db[j])
20:             j ← j + 1
21:         ciklus vége
22:         van ← (j < i)
23:         i ← i + 1
24:     ciklus vége
25:     vissza van
26: függvény vége
```

Felhasznált változók és függvények

- x: Kétdimenziós egész értékeket tartalmazó tömb.
- m: Az x tömb sorainak száma.
- n: Az x tömb oszlopainak száma.
- van: Logikai változó, amely pontosan akkor igaz, ha van két olyan sor az x tömbben, amelyek azonos számú nullát tartalmaznak.
- db: m elemű számláló tömb, mely megadja, hogy melyik sorban hány darab nulla található.

---

3. fejezet

Rendezések

Az algoritmus alkotási és programozási gyakorlatban igen gyakran előálló feladat, hogy a rendelkezésre álló adatokat – amiket például egy tömbben tárolunk el – sorba kell rendezni. Emiatt számos ún. rendező algoritmust alkottak meg, melyek közül a leginkább ismert, egyszerűen megérthetőket tárgyaljuk ebben a fejezetben.

A rendezések során mindig egy tömb elemeit fogjuk sorba rendezni, mégpedig növekvő sorrendbe. Ehhez persze szükséges, hogy a tömb olyan elemeket tartalmazzon, melyek egymással összehasonlíthatók, azaz bármely két elem között értelmezett a $<$ reláció. A rendezés során nem fogunk új tömböt létrehozni a rendezendő adatok átmeneti tárolására, hanem az eredeti bemeneti tömbön belül végezzük el a rendezést. Ennek következményeként az előálló rendezett tömb, ami egyben az algoritmus kimenete is lesz, az eredeti tömbben áll elő. Mindez úgy is leírható, hogy a rendezendő n elemű x tömb elemeire a rendezés végére teljesül, hogy

$$x[1] \leq x[2] \leq \ldots \leq x[n]. \tag{3.1}$$

A fejezetben ismertetett rendezések esetén a futási időt három szempont szerint fogjuk elemezni. Egyrészt vizsgálni fogjuk, hogy az adott algoritmus alkalmazása során hány összehasonlításra van szükség. Másrészt elemezni fogjuk a tömbelemek között elvégzendő cserék számát, illetve az elemek másolásának számát. Az elemek közötti cserét a 3.1. algoritmusban részletezett módon valósítjuk meg. A rendezéseknél leírt pszeudokódokban a Csere eljárás helyett a $\leftrightarrow$ operátort használjuk, de ez csak jelölésbeli különbözőség. Látható, hogy egy csere minden esetben három értékadással, azaz három elem másolással valósítható meg, ezért a cserék számának mindig háromszorosa a cserék során végrehajtott értékadások száma.

3.1. Algoritmus Csere

Bemenet: $a - \mathbf{T}$, $b - \mathbf{T}$
Kimenet: $a - \mathbf{T}$, $b - \mathbf{T}$

```
1: eljárás Csere(címszerint a : T, címszerint b : T)
2:     segéd ← a
3:     a ← b
4:     b ← segéd
5: eljárás vége
```

Felhasznált változók és függvények

- a: Egyik cserélendő változó.
- b: Másik cserélendő változó.
- *segéd*: A csere megvalósításához szükséges segédváltozó.

A futási idő elemzése során meg fogjuk vizsgálni a legrosszabb esetet, ami tipikusan a fordítva rendezett tömb esete lesz, hiszen annak érdekében, hogy egy csökkenő módon rendezett tömböt növekvő módon rendezetté alakítsunk a lehető legtöbb cserét, illetve másolást kell végrehajtani. Vizsgálni fogjuk azt az esetet is, amikor a tömb eleve rendezett, mert az algoritmusok ezt nem tudják előre megállapítani, így érdekes lehet, hogy ilyen esetben mit tudunk mondani a futási időről. Természetesen az átlagos esettel is fogunk foglalkozni.

Memória helyfoglalás szempontjából minden ismertetett algoritmus ugyanúgy fog viselkedni. Ahogy már említettük a rendezéseket a bemeneti tömbön belül valósítjuk meg, tehát ehhez n elemű tömb esetén n elem tárolásához szükséges mennyiségű memóriahely kell. Amikor két tömbbeli elemet meg fogunk cserélni, akkor a CSERE eljárás létrehoz egy új átmeneti elemet, amit a csere lefutásának idejére a memóriában tárolnunk kell, ezért összességében $n + 1$ elem tárolásához szükséges memória helyre van szükségünk. Két ismertetett algoritmusban cserék használata nélkül fogunk rendezni, ekkor viszont egy segédváltozót használunk majd, így ebben az esetben is $n+1$ elemnyi memóriára lesz szükségünk. Azzal külön nem foglalkozunk, hogy a tömbök indexelésére használt indexváltozók tárolásra mennyi memóriára van szükségünk, mert az indexek mérete általában elenyésző a tömbelemek méretéhez képest.

A fejezetben tíz különböző rendező algoritmust mutatunk be, melyek négy nagy csoportba oszthatók. Elsőként a 3.1. alfejezetben bemutatjuk az egyszerű cserés rendezést, mely egy alapvető ötletet valósít meg. Ennek az algoritmusnak a továbbfejlesztéseként vezetjük be a minimumkiválasztásos rendezést, melyet a 3.2. alfejezetben mutatunk be. A 3.3. alfejezetben egy másik egyszerű ötleten alapuló rendezést, a buborékrendezést ismertetjük. Ennek a rendezésnek is létezik javított változata, a 3.4. alfejezetben bemutatott javított buborékrendezés. Ötödik rendezési algoritmusként a beillesztéses rendezést ismertetjük a 3.5. alfejezetben. Itt is lesz majd javítási lehetőség, így a 3.6. alfejezetben leírjuk a javított beillesztéses rendezést is. Ezen utóbbit még tovább lehet fejleszteni, így kapjuk meg a 3.7. alfejezetben ismertetett Shell rendezést.

Az említett hét rendező algoritmus közül a szakirodalomban általában csak négyet találunk meg, a minimumkiválasztásos rendezést – amit szokás egyszerűen csak kiválasztásos rendezésnek nevezni –, a javított buborékrendezést, a javított beillesztéses rendezést, illetve a Shell rendezést. A másik három algoritmus tárgyalását didaktikai szempontok miatt tartjuk fontosnak.

Negyedik nagy csoportban két speciális rendezést tárgyalunk, melyek csak akkor használhatók a tömbelemek egy pozitív egész értékű kulcs mezővel is rendelkeznek. A a 3.8. alfejezetben tárgyalt szétosztó rendezés és a 3.9. alfejezetben ismertetett számlálva szétosztó rendezés nem a bemeneti tömbön belül rendez, így ebben is eltérnek a többi tárgyalt rendező algoritmustól. Ezen két rendezésben megismert logika alapján az egyszerű cserés rendezés is módosítható, így kapjuk a 3.10. alfejezetben bemutatott számláló rendezést.

A jegyzet későbbi részében, a 6. fejezetben bemutatunk két további rendező algoritmust is, az összefuttató- és a gyorsrendezést. Viszont ezek olyan algoritmus alkotási eszközöket használnak, melyekre most még nem alapozhatunk.

Egyszerű cserés rendezés

Az egyszerű cserés rendezés ötlete nagyon könnyen megérthető. Először megvizsgáljuk, hogy a rendezendő tömb első eleme kisebb-e a második elemnél. Ha kisebb, akkor megfelelő a viszonyuk, hiszen növekvő módon rendezett tömböt akarunk létrehozni. Ha viszont a második elem kisebb, mint az első, akkor megcseréljük a két elemet, hogy a sorrendjük az elvárásoknak megfelelő legyen. Ezt követően megvizsgáljuk az első és a harmadik elem viszonyát. Ha a harmadik elem kisebb az elsőnél, akkor cserélünk, hiszen előrébb kell lennie a kisebb elemnek. Hasonló vizsgálatot végzünk az első és a negyedik elem között, majd ha szükséges, akkor cserélünk. Addig folytatjuk a lépéseket, amíg el nem jutunk odáig, hogy már az első és az utolsó tömbelemet hasonlítjuk össze. Természetesen csak akkor cserélünk, ha az utolsó elem kisebb az elsőnél. Így végeredményben az első elemet összehasonlítottuk az összes többi elemmel, és ha az első elem nem volt kisebb a másik elemnél, akkor cserét hajtottunk végre. Ennek eredményeként biztos, hogy a tömb legkisebb eleme került az első helyre, tehát a végső helyén van.

Ezt követően a második elemet is hasonló módon összehasonlítjuk az összes őt követővel, és ha szükséges, azaz a nagyobb indexű elem kisebb értékű, mint a második, akkor cserélünk. Miután a második elemet az összes őt követővel összehasonlítottuk, és ha kellett, cseréltünk, biztosak lehetünk abba, hogy a második legkisebb elem került a második helyre, azaz a végleges helyére.

Ezt az eljárást követjük a harmadik elemre, majd a negyedikre, és így tovább egészen az utolsó előtti elemig. A vizsgált elemet mindig összehasonlítjuk az összes őt követővel, és szükség esetén cserélünk, így az algoritmus végére minden elem biztosan az elvárt helyére kerül, így az egész tömb rendezetté válik. Nem szorul talán magyarázatra, hogy miért pont az egyszerű cserés nevet kapta ez a rendezési módszer, hiszen tényleg nem teszünk mást csak cserélgetünk.

Az ismertetett ötlet pszeudokóddal történő leírását a 3.2. algoritmus adja meg. A 2. sorban kezdődő külső számlálós ciklus i ciklusváltozója a tömbnek azt az elemét fogja indexelni, amelyet a többi elemmel összehasonlítunk egy futamon belül. A 3. sorban kezdődő belső ciklus ciklusváltozójával pedig rögzített i index mellett az összes i-nél nagyobb indexű elemet fogjuk indexelni, ezért j értéke $(i+1)$-től n-ig megy. A két ciklus gondoskodik a tömb bejárásáról. A ciklusmagban meg kell valósítanunk az i-edik és az őt követő valamely elem összehasonlítását. Ez történik meg a 4. sorban szereplő elágazásban. Ha az $x[i] > x[j]$ feltétel igaz, azaz az i-edik elem utáni aktuális elem kisebb, mint az i-edik, akkor cserélni kell a két elemet, ahogy az az 5. sorban meg is történik.

3.2. Algoritmus Egyszerű cserés rendezés

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: x – **T rendezett tömb**

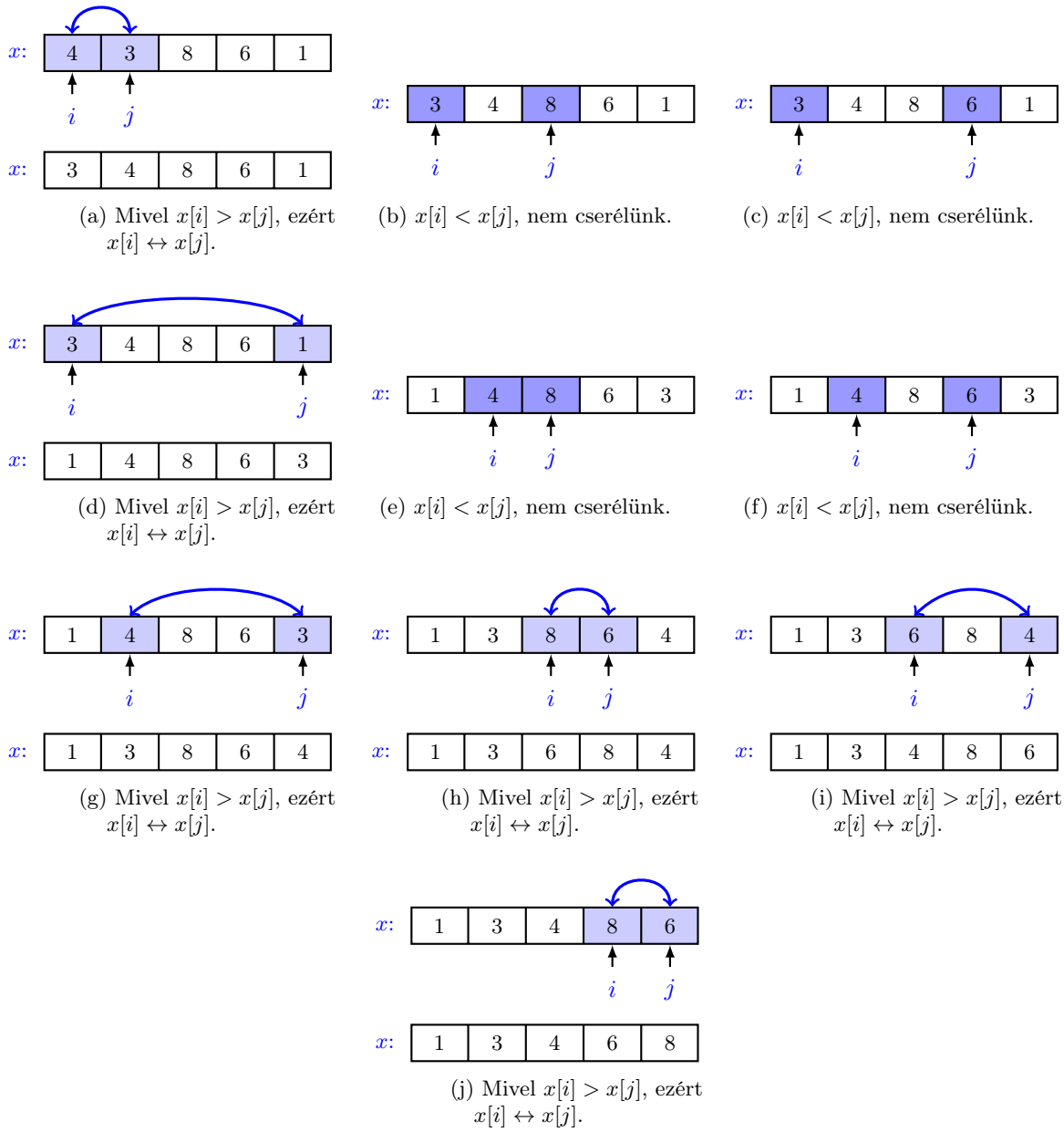
```
1: eljárás EgyszerűCserésRendezés(címszerint x : T tömb, n : egész)
2:     ciklus i ← 1-től (n − 1)-ig
3:         ciklus j ← (i + 1)-től n-ig
4:             ha x[i] > x[j] akkor
5:                 x[i] ↔ x[j]
6:             elágazás vége
7:         ciklus vége
8:     ciklus vége
9: eljárás vége
```

Felhasznált változók és függvények

- x: A rendezni kívánt tömb. Az x tömb elemeinek összehasonlíthatónak kell lennie. Az eljárás a tömböt helyben rendezi.
- n: A paraméterként átadott tömb mérete.

3.1. Példa. Adott a 4, 3, 8, 6, 1 elemeket tartalmazó x tömb. Feladatunk, hogy növekvő módon rendezzük a tömböt az egyszerű cserés rendezés 3.2. algoritmusának használatával. Az algoritmus futása során az i és j indexek alakulását, valamint a tömbben bekövetkező változásokat a 3.1. ábrán szemléltetjük.

Első lépésként elindul az algoritmus 2. sorában található ciklus, így i értéke 1 lesz. Majd a 3. sorban lévő belső ciklushoz kerül a vezérlés, ezért j értéke 2 lesz. Összehasonlítjuk az $x[i]$ és $x[j]$ értékét, és mivel $x[i] > x[j]$, ezért megcseréljük a két megindexelt elemet, ahogy a 3.1a. ábrán látható.



(a) Mivel $x[i] > x[j]$, ezért $x[i] \leftrightarrow x[j]$.

(b) $x[i] < x[j]$, nem cserélünk.

(c) $x[i] < x[j]$, nem cserélünk.

(d) Mivel $x[i] > x[j]$, ezért $x[i] \leftrightarrow x[j]$.

(e) $x[i] < x[j]$, nem cserélünk.

(f) $x[i] < x[j]$, nem cserélünk.

(g) Mivel $x[i] > x[j]$, ezért $x[i] \leftrightarrow x[j]$.

(h) Mivel $x[i] > x[j]$, ezért $x[i] \leftrightarrow x[j]$.

(i) Mivel $x[i] > x[j]$, ezért $x[i] \leftrightarrow x[j]$.

(j) Mivel $x[i] > x[j]$, ezért $x[i] \leftrightarrow x[j]$.

3.1. ábra. Egyszerű cserés rendezés. A példán a 3.2. algoritmusnak megfelelően nyomon követhetjük az i és j indexek alakulását, valamint az elvégzett cserék eredményeit.

Ezután a belső ciklusban j értéke 3-ra növekszik. Mivel most $x[i] < x[j]$, tehát az algoritmus 4. sorában található feltétel hamis, ezért nem kell cserét végrehajtanunk (ld. 3.1b. ábra). A belső ciklusban j értéke 4-re növekszik, viszont ekkor is $x[i] < x[j]$, ezért nem hajtunk végre cserét (ld. 3.1c. ábra). Ezt követően j értéke 5-re növekszik. Mivel ekkor $x[i] > x[j]$, ezért meg kell cserélni az i-edik és j-edik elemet (ld. 3.1d. ábra). Vegyük észre, hogy a tömb első elemét minden más elemmel összehasonlítottuk, ha kellett cseréltünk, így a tömb első helyén a legkisebb elem található.

Mivel a belső ciklus végére értünk, ezért a vezérlés ismét a külső ciklus fejéhez kerül, ahol i értéke 2-re változik. A belső ciklus újra elindul, de most j kiindulási értéke már 3 lesz. Mivel $x[2] < x[3]$, ezért nem történik csere (ld. 3.1e. ábra). A következő lépésben j értéke 4-re növekszik és ekkor sem kell cserét végrehajtani, ahogy az a 3.1f. ábrán is látható. Ezután $j = 5$ esetén $x[i] > x[j]$, ezért cserét kell eszközölni (ld. 3.1g. ábra). A belső ciklusnak ismét a végére értünk, mostanra a tömb második legkisebb eleme is a helyére került.

A külső ciklusban növeljük i értékét 3-ra, majd a belső ciklusban j felveszi a 4 értéket. Mivel $x[3] > x[4]$, ezért cserélni kell (ld. 3.1h. ábra). Ezután j 5-re változik és ismét cserét kell végrehajtani, amint az a 3.1i. ábrán is látható. Mivel a belső ciklus ismét a végére ért, így érdemes észrevenni, hogy a harmadik legkisebb elem is a helyére került.

A külső ciklus utoljára fut le ezt követően $i = 4$ értékkel. Ekkor j csak az 5 értéket veszi fel. Mivel $x[i] > x[j]$, ezért egy cserét még végre kell hajtani (ld. 3.1j. ábra). Minden ciklus végére értünk, így az algoritmus futása véget ér, a tömbünk pedig rendezetté vált. ¶

Futási idő elemzése. Vizsgáljuk meg, mit mondhatunk az egyszerű cserés rendezés futási idejéről. Ahogy a fejezet elején, a rendezések bevezetésénél már említettük, a futási időt az összehasonlítások, cserék és másolások számának vizsgálatával elemezzük, legrosszabb, legjobb és átlagos esetben.

Észrevehetjük, hogy az algoritmus 4. sorában található összehasonlítást pontosan annyiszor végezzük el, ahányszor a belső ciklus lefut. Meg kell tehát határoznunk, hogy ez hányszor történik meg. Amikor $i = 1$, akkor a belső ciklus j változója $n - 1$ különböző értéket vesz fel, tehát ennyiszer fut le a belső ciklus. Az $i = 2$ esetén $(n - 2)$-szer fut a belső ciklus, $i =$ 3-nál pedig $(n - 3)$-szor. Azt kell tehát meghatároznunk, hogy mivel egyenlő az

$$(n-1) + (n-2) + (n-3) + \ldots + 2 + 1 \tag{3.2}$$

összeg. Matematikai ismereteinkből tudjuk, hogy ez pont egyenlő az

$$\frac{n\,(n-1)}{2} \tag{3.3}$$

értékkel. Ez lesz tehát az összehasonlítások száma, bármilyen is volt a rendezés előtt a tömbünk.

Nézzük meg ezt követően, mi a helyzet az algoritmus 5. sorában található cserék számával. Ezek száma már függ attól, hogy az aktuálisan vizsgált két tömbelemnek milyen a viszonya. Ha a tömbünk kezdetben csökkenő módon rendezett volt, akkor minden összehasonlításnál igaz lesz az $x[i] > x[j]$ feltétel, ezért ugyanannyi cserét kell végeznünk, mint amennyi az összehasonlítások száma. Így kijelenthetjük, hogy legrosszabb esetben a cserék száma is $\frac{n(n-1)}{2}$ lesz. Vizsgáljuk meg most a legjobb esetet, azaz amikor a tömbünk eleve növekvő módon rendezett volt. Mivel ilyenkor minden lehetséges i és j esetén $x[i] < x[j]$, így az algoritmus 4. sorában lévő feltétel mindig hamis lesz, tehát soha nem fogunk cserélni. Átlagos esetben a cserék darabszáma $\frac{1}{2} \cdot \frac{n(n-1)}{2}$ lesz.

Mivel a másolások száma a cserék számának háromszorosa, ezért legrosszabb esetben $3 \cdot \frac{n(n-1)}{2}$, legjobb esetben pedig nulla másolás fog végrehajtódni.

Összességében azt mondhatjuk az egyszerű cserés rendezés algoritmusáról, hogy minden esetben a tömb méretével négyzetesen arányos a futási idő, hiszen már az összehasonlítások száma is ilyen. Azaz a futási idő biztos, hogy $O\left(n^2\right)$-es. ♣

Minimumkiválasztásos rendezés

Az egyszerű cserés rendezés működési elve az volt, hogy az első elemtől az utolsó előtti elemig bejárjuk a rendezendő tömbünket, és az aktuális tömbelemet összehasonlítjuk az összes őt követővel. Amennyiben azt találjuk, hogy az aktuális elem utáni valamely elem kisebb az aktuális elemnél, akkor kicseréltük a két elemet. Ennek eredményeként azt tapasztaltuk, hogy először a legkisebb elem került az első helyre, majd a második legkisebb a második helyre, és így tovább, egészen a tömb végéig. Ennek eléréséhez viszont számos cserét kellett végrehajtanunk.

Érdemes azt megvizsgálni, hogy nem lehetne-e hasonló eredményt elérni úgy, hogy a cserék számát jelentősen lecsökkentsük, ezáltal az algoritmusunk futási idejét redukáljuk. Ennek érdekében azt fogjuk tenni, hogy először megvizsgáljuk, hogy a teljes tömbben melyik elem a legkisebb értékű. Ezt az elemet kicseréljük az első elemmel, így az már biztos a helyére kerül. Ezt követően már csak a tömb fennmaradó elemeivel kell foglalkoznunk. Megkeressük a második és az utolsó elem közöttiek közül a legkisebbet, majd ezt kicseréljük a második elemmel. Ennek eredményeként már a tömb első két eleme a helyén lesz. Folytatjuk az eljárásunkat, a harmadik elemtől a tömb végéig megkeressük ismét a legkisebb elemet, majd ezt kicseréljük a harmadik elemmel. Ezt az eljárást követjük addig, amíg az utolsó keresést már csak az utolsó előtti és az utolsó elemből álló résztömbben hajtjuk végre, majd a két elem közül a kisebbiket kicseréljük az utolsó előtti elemmel. Ha ezt az ötletet megvalósítjuk a tömbünk rendezetté válik.

Az ismertetett gondolatot viszonylag egyszerűen meg tudjuk valósítani, mert csak egy ciklusra van szükségünk, amely a tömb első elemétől az utolsó előtti elemig fut. Ezen ciklus belsejében pedig meg kell keresnünk a minimális elemet. A legkisebb elemet a 2.1.6. fejezetben megismert maximumkiválasztás programozási tétel módosításával könnyen meg tudjuk találni. A minimális elem kiválasztását követően már csak egy cserét kell végrehajtanunk. A minimumkiválasztásos rendezés konkrét megvalósítását a 3.3. algoritmus írja le pszeudokóddal.

3.3. Algoritmus Minimumkiválasztásos rendezés

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: x – **T rendezett tömb**

```
1:  eljárás MinimumkiválasztásosRendezés(címszerint x : T tömb, n : egész)
2:      ciklus i ← 1-től (n − 1)-ig
3:          min ← i
4:          ciklus j ← (i + 1)-től n-ig
5:              ha x[min] > x[j] akkor
6:                  min ← j
7:              elágazás vége
8:          ciklus vége
9:          x[i] ↔ x[min]
10:     ciklus vége
11: eljárás vége
```

Felhasznált változók és függvények

- x: A rendezni kívánt tömb. Az x tömb elemeinek összehasonlíthatónak kell lennie. Az eljárás a tömböt helyben rendezi.
- n: A paraméterként átadott tömb mérete.
- min: A tömb egy bejárásán belül a megtalált minimális értékű elem indexe.

A 2. sorban kezdődő külső ciklussal járjuk be a tömbünket az első elemtől az utolsó előtti elemig. Az algoritmus lényege az – ahogy ezt már korábban említettük –, hogy az i-edik és az n-edik elem közötti elemek közül választjuk ki a minimális értékűt, majd ezt a minimális elemet kicseréljük az i-edik elemmel. Tehát a külső ciklus azt az elemet fogja indexelni, ami a vizsgálandó tömbrész első eleme, illetve amely elem a külső ciklus magjának minden egyes lefutása végén már a végleges értékét veszi fel.

A 3. sorban elindul a minimumkiválasztás eljárása, amely egészen a 8. sorig tart. Kijelöljük a kezdetben minimális értékű elemnek tekintett elem indexét, ami az i lesz (ld. 3. sor), majd bejárjuk a vizsgálandó tömbrész fennmaradó elemeit. Ezt a bejárást valósítja meg a 4. sorban kezdődő belső ciklus. A belső ciklus magjában összehasonlítjuk az eddigi minimális elemet az aktuális tömbelemmel, azaz

$x[j]$-vel (5. sor). Ha az aktuális elem kisebb mint az eddigi minimum, akkor új minimumot találtunk, ezért a *min* változóban tárolt indexet felülírjuk az aktuális elem j indexével (ld. 6. sor).

Amikor a belső ciklus végére érünk, akkor a *min* index az i-edik és az n-edik elem közötti elemek minimumának indexét tartalmazza. Mivel a minimális értékű elemet az i-edik helyre szeretnénk helyezni, hiszen ez az algoritmusunk fő célja, ezért kicseréljük az i-edik és a *min*-edik elemet (ld. 9. sor). Ha ezt minden i-re megvalósítjuk, akkor végeredményben a tömbünk rendezetté válik.

Megjegyzés

Az algoritmus 9. sorában lévő cserét abban az esetben is végrehajtjuk, ha a legkisebb elem épp az i-edik, azaz ha $min = i$. Ennek kivédését csak úgy tudjuk megtenni, ha a csere elé beszúrunk egy feltétel vizsgálatot. Így csak akkor cserélünk, ha a $min \neq i$ feltétel teljesül. A felesleges cserét ilyen módon sikerül kiküszöbölnünk, viszont megjelenik egy új feltétel, amit akkor is ki kell értékelni, amikor cserélni kell, tehát ilyenkor pedig a feltétel vizsgálat lesz felesleges. Ezek alapján döntöttünk úgy, hogy inkább minden esetben elvégezzük a néha feleslegesnek tűnő cserét.

Megjegyzés

A szakirodalomban sokszor a minimumkiválasztásos rendezést egyszerűen csak kiválasztó rendezésnek[a] hívják. Mi azért választottuk inkább az itt használt elnevezést, mert így jobban látszik, hogy egy jól ismert programozási tételt használunk a külső ciklus magjában.

[a] Angolul: selection sort

3.2. Példa. Nézzük végig egy konkrét példán hogyan rendez egy tömböt a minimumkiválasztásos rendezés 3.3. algoritmusa. A 3.2. ábrán követhetjük nyomon a 4, 3, 8, 6, 1 elemeket tartalmazó x tömb rendezésének lépéseit.

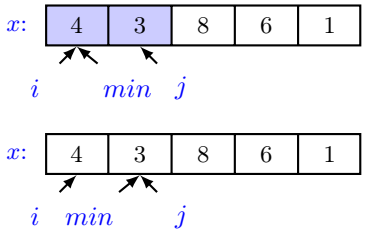
Elsőként belépünk az algoritmus 2. sorában kezdődő ciklusba, az i index értéke 1 lesz. Meg kell keresnünk a tömb legkisebb elemét, hogy ezzel cserélhessük majd ki a jelenlegi első elemet. Ehhez minimumkiválasztást hajtunk végre, ami a *min* index inicializálásával kezdődik, így *min* is felveszi az 1 értéket. Belépünk a 4. sorban kezdődő belső ciklusba, j kezdeti értéke 2 lesz. Következeik az 5. sorbeli feltétel vizsgálat. Mivel a második elem kisebb, mint az első, ezért *min* értékét kettőre változtatjuk, ahogy ez a 3.2a. ábrán is látható.

A belső ciklusban továbblépünk, j értéke háromra nő. Mivel a harmadik elem nagyon, mint $x[min]$, ezért nem kell a *min* értékét módosítani (ld. 3.2b. ábra). Ezután ismét nő j értéke és $x[min] < x[j]$ teljesül, ezért most sem módosítunk semmit (ld. 3.2c. ábra). Következő lépésben a j index 5-re növekszik. Mivel $x[5] < x[min]$, ezért *min* értékét 5-re módosítjuk (ld. 3.2d. ábra). A j értéke elérte az 5-öt, így a belső ciklus végére értünk. Ekkor megcseréljük az i és a *min* indexű elemeket (ld. 3.2e. ábra). Jól látható, hogy a külső ciklusmag első végrehajtásának végére a tömb legkisebb eleme a tömb elejére került.

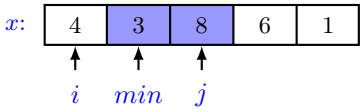
A külső ciklusban i értékét 2-re növeljük, tehát kezdődik a tömb második legkisebb elemének megkeresése, amely a jelenlegi második és utolsó elem közötti elemek minimuma. Ennek érdekében *min* felveszi a 2 értéket, a belső ciklus elején pedig j a 3-at. Mivel $x[2] < x[3]$, ezért *min* értékén az összehasonlítás után nem módosítunk (ld. 3.2f. ábra). A belső ciklusban j 4-re, majd 5-re növekszik, de mindkét elem nagyobb a másodiknál, így nem módosítjuk a *min* indexet (ld. 3.2g. és 3.2h. ábrák). Mivel a belső ciklusnak ismét a végére értünk, ezért végrehajtunk egy cserét az i-edik és a *min* indexű elemek között (ld. 3.2i. ábra). Ez a csere most semmilyen tényleges változást nem fog eredményezni, hiszen $min = i$.

Már az első két elem a helyén van a tömbben, ismét következik a külső ciklusban i növelése. A *min* értéke egyenlő lesz i-vel, azaz 3-mal. A belső ciklusban j kezdeti értéke 4 lesz, majd következik a j-edik és a *min*-edik elem összehasonlítása. Mivel a negyedik elem kisebb a harmadiknál, ezért a *min* index 4-re módosul (ld. 3.2j. ábra). Amikor j-t 5-re növeljük, akkor hasonló esettel találkozunk, ahogy az a 3.2k. ábrán is látható. A belső ciklus végére értünk, ezért meg kell cserélni az i és a *min* indexű elemeket, aminek eredményeként már a tömb első három eleme a rendezettségnek megfelelő helyére került (ld. 3.2l. ábra).

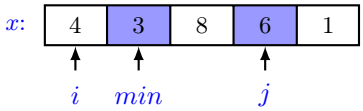
Következik a külső ciklus utolsó lefutása $i = 4$ érték mellett. A *min* is 4 lesz, j pedig 5-öt vesz fel kezdeti értékként a belső ciklusban. Mivel az ötödik elem nagyobb a negyediknél, ezért nem fogjuk a



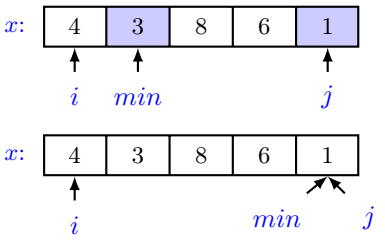
(a) Kezdetben $i \leftarrow 1$ és kiindulásként $min \leftarrow 1$. Mivel $x[min] > x[j]$, ezért $min \leftarrow j$.



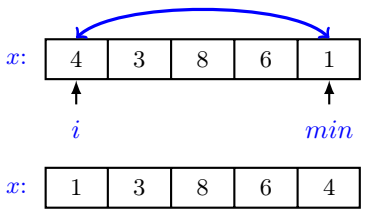
(b) Mivel $x[min] < x[j]$, ezért nincs teendő.



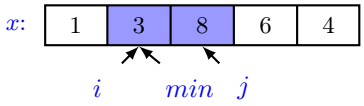
(c) Mivel $x[min] < x[j]$, ezért nincs teendő.



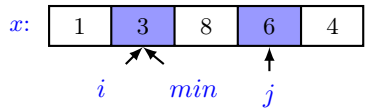
(d) Mivel $x[min] > x[j]$, ezért $min \leftarrow j$.



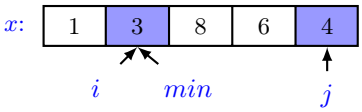
(e) A belső ciklus végére értünk, ezért $x[min] \leftrightarrow x[i]$.



(f) A külső ciklusban továbblépünk, azaz $i \leftarrow 2$ és $min \leftarrow i$. Mivel $x[min] < x[j]$, ezért nincs teendő.

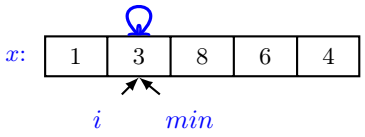


(g) Mivel $x[min] < x[j]$, ezért nincs teendő.

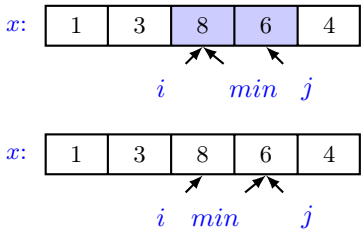


(h) Mivel $x[min] < x[j]$, ezért nincs teendő.

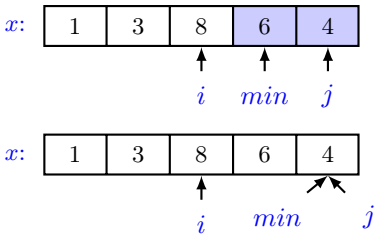
3.2. ábra. Minimumkiválasztásos rendezés. Az ábrán nyomon követhetjük az egyes indexek, valamint a tömb elemeinek alakulását a 3.3. algoritmusnak megfelelően.



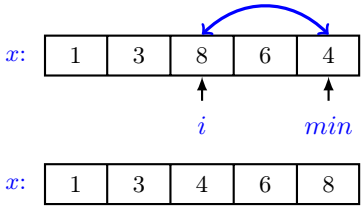
(i) A belső ciklus végére értünk, ezért $x[min] \leftrightarrow x[i]$, még akkor is, ha $min = i$.



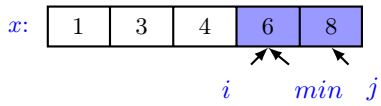
(j) A külső ciklusban továbblépünk, azaz $i \leftarrow 3$ és $min \leftarrow i$. Mivel $x[min] > x[j]$, ezért $min \leftarrow j$.



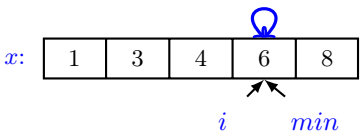
(k) Mivel $x[min] > x[j]$, ezért $min \leftarrow j$.



(l) A belső ciklus végére értünk, ezért $x[min] \leftrightarrow x[i]$.



(m) A külső ciklusban továbblépünk, azaz $i \leftarrow 4$ és $min \leftarrow i$. Mivel $x[min] < x[j]$, ezért nincs teendő.



(n) A belső ciklus végére értünk, ezért $x[min] \leftrightarrow x[i]$, még akkor is, ha $min = i$.

3.2. ábra. Minimumkiválasztásos rendezés. Az ábrán nyomon követhetjük az egyes indexek, valamint a tömb elemeinek alakulását a 3.3. algoritmusnak megfelelően (folyt.).

min indexet módosítani (ld. 3.2m. ábra). Végre kell még hajtanunk egy cserét, ami jelen esetben egy helyben hagyás lesz (ld. 3.2n. ábra). Az algoritmusnak ekkor a végére érünk, hiszen az i változó elérte már az utolsó értékét. Jól látható, hogy az x tömb növekvő módon rendezetté vált. ¶

Futási idő elemzése. Az algoritmus megismerését követően vizsgáljuk meg, hogy tényleg sikerült-e javítanunk a rendezés futási idején az egyszerű cserés rendezéshez képest.

Vizsgáljuk meg elsőként az összehasonlítások számát, azaz hogy hányszor kerül a vezérlés az algoritmus 5. sorához. Ha az i ciklusváltozó értéke 1, akkor a belső ciklus $(n-1)$-szer fut le, tehát a feltételbeli összehasonlítást is $(n-1)$-szer végezzük el. Az $i = 2$ esetén ez a szám $n-2$ lesz. Könnyen látható, hogy i növekedésével a belső cikluson belüli összehasonlítások száma mindig eggyel csökken. Tehát végeredményben az összes összehasonlítások száma

$$(n-1) + (n-2) + \ldots + 1 \tag{3.4}$$

lesz. Így az összehasonlítások száma:

$$\frac{n\,(n-1)}{2}. \tag{3.5}$$

Ez az eredmény nem függ attól, hogy a bemeneti tömbünk milyen volt rendezettség szempontjából, tehát az összehasonlítások száma minden esetben pontosan ugyanannyi, mint az egyszerű cserés rendezésnél. Így itt nem értünk el javulást.

Nézzük most a cserék számát. Cserét csak a külső ciklusmag lefutásának végén végzünk. Így pontosan annyi cserénk lesz, mint ahányszor lefut a külső ciklus. Ez az érték pedig $n-1$. A másolások száma is könnyen meghatározható így már: $3 \cdot (n-1)$. Vegyük észre, hogy a cserék és másolások száma sem függ attól, hogy milyen a rendezendő tömbünk.

Kérdés, hogy sikerült-e javítanunk a rendezés futási idején az egyszerű cserés rendezés futási idejéhez képest. Azt mondhatjuk, hogy átlagosan igen, mivel az egyszerű cserés rendezés cseréinek átlagos száma $\frac{1}{2} \cdot \frac{n(n-1)}{2}$, míg a minimumkiválasztásos rendezésnél mindig csak n darab cserét hajtunk végre. A futási idő csak akkor fog romlani, ha a bemeneti tömb eleve rendezett volt, mert egyszerű cserés esetben ilyenkor egyetlen csere sem lesz, míg a minimumkiválasztásos esetben ekkor is megtörténik az $n-1$ darab csere.

Összességében azt mondhatjuk, hogy a minimumkiválasztásos rendezést akkor érdemes használni, ha a cseréket lassan tudjuk megvalósítani, mert ezek számában sikerült jó eredményt elérni. (Később látni fogjuk, hogy ilyen alacsony csere számot más algoritmussal sem tudunk átlagos esetben elérni.) A minimumkiválasztásos rendezés teljes futási ideje viszont az összehasonlítások száma miatt szintén $O\left(n^2\right)$-es. ♣

Buborékrendezés

A buborékrendezés[1] az egyik legismertebb rendezési eljárás. Ha egy laikust megkérdeznek, hogy milyen rendező algoritmusokról hallott, nagy valószínűség szerint a buborékrendezést említeni fogja. Kérdés persze, hogy az ismertség együtt jár-e a hatékonysággal.

A buborékrendezés az egyszerű cserés rendezéshez hasonlóan egy nagyon egyszerű ötletet valósít meg. Csak összehasonításokat kell végeznünk az elemek között, illetve ha szükséges, akkor cserélnünk. Ami viszont megkülönbözteti az egyszerű cserés rendezéstől az az, hogy az összehasonlításokat és a cseréket is mindig szomszédos elemek között hajtjuk végre.

A rendező algoritmus lényege, hogy összehasonlítjuk a szomszédos elemet, és amennyiben a kisebb indexű elem nagyobb mint a nagyobb indexű elem, akkor megcseréljük őket a növekvő rendezettség elérése érdekében. Az összehasonlításokat az első elemtől kezdjük, azaz először az első és második elemmel foglalkozunk. Ezt követően a második és harmadik elemet vetjük össze, majd jön a harmadik és a negyedik, és egészen így haladunk míg a tömb végére nem érünk. Legvégül tehát az utolsó előtti és az utolsó elemet hasonlítjuk össze, és ha szükséges, cserét is végrehajtunk. Természetesen attól, hogy egyszer így megvizsgáltunk és szükség esetén megcseréltünk minden szomszédos párt, még nem válik teljesen rendezetté a tömbünk. Viszont minden elem feltehetőleg közelebb kerül a végleges helyéhez, mivel a kisebb elemek előrefelé, a nagyobb elemek pedig hátrafelé mozdulnak el a cserék során. Sőt a legnagyobb elem biztos, hogy a végleges helyére kerül, mivel bármely összehasonlításnál amennyiben nála kisebb elemet találtunk a „jobbján", akkor mindig megcseréltük őket. Azt láthatjuk tehát, hogy itt egy bejárás során a legnagyobb elem fog a végleges helyére kerülni. Ez egy lényeges különbség az egyszerű cserés rendezéshez képest, mert ott a legkisebb elem került először a tömb elejére. Az, hogy a legnagyobb elem eljut az első bejárás során a legnagyobb indexű helyre, indokolja a rendezés elnevezését, hiszen a nagy buborékok szállnak felfelé a folyadék felszínére.

Az első bejárást követően hogyan folytatódik tovább az algoritmus futása? Ismét elkezdjük a szomszédos elemek összehasonlítását és esetleges cseréjét a tömb elejéről a vége felé haladva. Viszont a második bejárásnál már nem fogunk a tömb végéig elmenni, hiszen a legutolsó elemmel kár foglalkozni, az már biztos a helyén van. Így egy n elemű tömb esetén a második bejárás során az utolsó összehasonlítás az $(n-2)$-edik és az $(n-1)$-edik elem között fog megtörténni. A második bejárás végére biztosak lehetünk abban, hogy a második legnagyobb elem is biztosan a helyére, az $n-1$ indexű pozícióba kerül.

Az algoritmus egy újabb bejárás folytatódik, amikor is az első pártól indulva hasonlítunk és szükség esetén cserélünk mindaddig, amíg el nem jutunk az $(n-3)$-adik és $(n-2)$-edik elemekből álló párig. A bejárások ilyen sorozatát végezzük addig, amíg az utolsó bejárás során már csak az első és második elemet kell összehasonítanunk és esetleg cserélnünk. Jól látható, hogy az ismertetett algoritmus hatására a rendezendő tömb úgy válik növekvő módon rendezetté, hogy a rendezett állapotot hátulról előre felé haladva éri el. A buborékos rendezés pszeudokóddal történő leírását 3.4. algoritmusban láthatjuk.

3.4. Algoritmus Buborékrendezés

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: x – **T rendezett tömb**

```
1: eljárás BuborékRendezés(címszerint x : T tömb, n : egész)
2:     ciklus i ← n-től 2-ig
3:         ciklus j ← 1-től (i − 1)-ig
4:             ha x[j] > x[j + 1] akkor
5:                 x[j] ↔ x[j + 1]
6:             elágazás vége
7:         ciklus vége
8:     ciklus vége
9: eljárás vége
```

Felhasznált változók és függvények

- x: A rendezni kívánt tömb. Az x tömb elemeinek összehasonlíthatónak kell lennie. Az eljárás a tömböt helyben rendezi.
- n: A paraméterként átadott tömb mérete.

[1] Angolul: bubble sort

Az algoritmus a 2. sorban kezdődő külső ciklussal indul. Az i ciklusváltozó a tömb méretétől (n) halad -1-esével 2-ig. Az i változó mindig azt jelzi, hogy az összehasonlítások során, melyik az utoljára összehasonlítandó elem. Kezdetben ez n, hiszen a tömb végéig haladva minden szomszédos párt meg akarunk vizsgálni. Ahogy haladunk előre az algoritmusban, úgy i értéke folyamatosan csökkenni fog, hiszen a tömb végén a helyükre kerülő elemekkel már nem kell a későbbiekben foglalkozni.

A 3. sorban egy újabb ciklus veszi kezdetét. Ennek j ciklusváltozója 1-től $(i-1)$-ig halad előre. A j változót fogjuk a tömb indexelésére használni, mindig a j-edik és a $(j+1)$-edik elemet vizsgálva. A belső ciklus magjában össze kell hasonlítanunk az aktuális két szomszédos elemet (ld. 4. sor). Amennyiben $x[j] > x[j+1]$, akkor cserét kell végrehajtanunk, ahogy azt az 5. sorban meg is tesszük. Ha $x[j] \leq x[j+1]$, akkor az elvárt rendezettségnek megfelelő a két szomszédos elem viszonya, ezért semmilyen teendőnk nincs.

3.3. Példa. Nézzük végig, hogy miként rendezi a buborékos rendezés 3.4. algoritmusa a 6, 1, 4, 3, 8 elemekből álló tömböt. Az algoritmus egyes lépései nyomon követhetők a 3.3. ábrán is.

Először belépünk a 2. sorban kezdődő külső ciklusba, az i változó felveszi a tömb méretének értékét. Tudjuk, hogy i azt jelöli, hogy az első bejárás során meddig fogjuk a párokat vizsgálni. Az első bejárásnál tehető végigmegyünk egészen az ötödik elemig. Az algoritmusban továbbhaladva belépünk a 3. sorbeli belső ciklusba, a j index 1-től indul. Összehasonlítjuk az első és a második elemet (ld. 4. sor). Mivel az első elem a nagyobb, ezért megcseréljük a két elemet (ld. 5. sor). Ezek a lépések láthatók a 3.3a. ábrán is.

A belső ciklusban j értékét 2-re növeljük, majd összehasonlítjuk a második és a harmadik elemet. Mivel a kisebb indexű elem a nagyobb értékű, ezért megcseréljük a két elemet (ld. 3.3b. ábra). Ezután j értéke 3-ra nő, majd összehasonlítjuk a harmadik és a negyedik elemet. Mivel a harmadik elem a nagyobb, ezért ismét cserélünk (ld. 3.3c. ábra). A belső ciklusban j értékét 4-re változtatjuk, és összehasonlítjuk a negyedik és az ötödik elemet. Mivel a negyedik elem már eleve kisebb volt mint az ötödik, ezért nincs szükség cserére (ld. 3.3d. ábra). A belső ciklus végére értünk. Vegyük észre, hogy a legnagyobb elem biztosan a helyére került, így ezzel az elemmel a további feldolgozás során már nem foglalkozunk.

A külső ciklusban i értékét 4-re csökkentjük, azaz a tömbnek már csak az első négy elemére fókuszálunk mostantól. A belső ciklusban j értéke felveszi az 1 értéket. Összehasonlítjuk az első és a második elemet, és mivel a növekvő rendezettségnek megfelelő a viszonyuk, ezért nem cserélünk (ld. 3.3e. ábra). A j értékét 2-re növeljük, majd összehasonlítjuk a második és a harmadik elemet. Mivel $x[2] > x[3]$, így megcseréljük őket (ld. 3.3f. ábra). Ezután j 3-ra nő, következik a harmadik és negyedik elem összevetése. Mivel a harmadik elem kisebb, ezért nincs szükségünk cserére (ld. 3.3g. ábra). A belső ciklus végén látható, hogy a negyedik elem is a helyén van már. A konkrét példánál azt látjuk, hogy ugyan már minden elem a helyén van, de az algoritmus ettől még nem ér véget.

A külső ciklusban i értéke újra eggyel csökken, mostantól 3 lesz. A belső ciklusban j ismét 1-től kezdi növekedését. Összehasonlítjuk az első és második elemet és nem cserélünk (ld. 3.3h. ábra). A j értéke 2-re nő, összehasonlítjuk a második és harmadik elemet, és most sem végzünk cserét (ld. 3.3i. ábra). A belső ciklusnak ismét vége, a külső ciklusban pedig i 2-re csökken. A belső ciklusban j felveszi az 1 értéket. Összehasonlítjuk az első és a második elemet, amelyek már rendezettek, ezért nem cserélünk (ld. 3.3j. ábra). Ekkor mindkét ciklus végére értünk, a tömbünk rendezetté vált. ¶

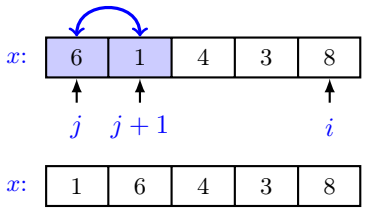
Megjegyzés

Az előző példában láthattuk, hogy a 3.3f. ábrán szemléltetett cserét követően a tömbünk már rendezett lett, de az algoritmusunk ezt nem vette észre, pedig itt befejeződhetett volna. A 3.4. fejezetben meg fogjuk vizsgálni, hogy miként lehetne ezt a hiányosságát az algoritmusnak kiküszöbölni.

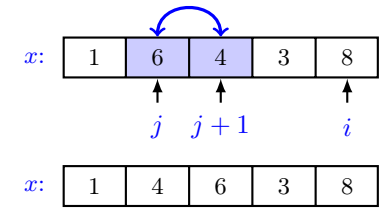
Futási idő elemzése. A futási idő elemzést ismét két részre bontjuk. Először az összehasonlítások számát, majd a cserék és másolások számát vizsgáljuk meg.

A 3.4. algoritmus 4. sorában lévő összehasonlítás a belső ciklus minden lefutásakor kiértékelésre kerül. A belső ciklus $i = n$ esetén $(n-1)$-szer fut le, $i = n-1$ esetén $(n-2)$-szer és így tovább. Az összehasonlítások száma így az előző két fejezetben már megismert érték lesz, azaz

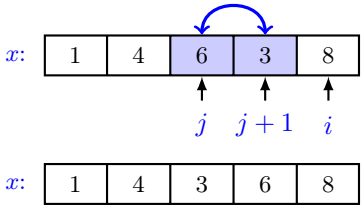
$$(n-1) + (n-2) + \ldots + 1 = \frac{n\,(n-1)}{2}. \tag{3.6}$$



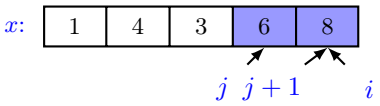
(a) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$.



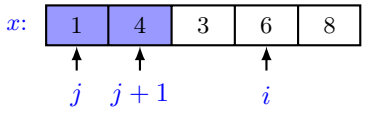
(b) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$.



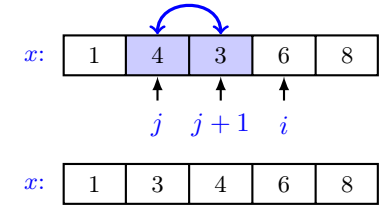
(c) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$.



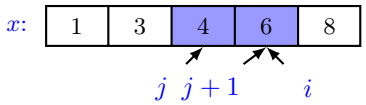
(d) Mivel $x[j] < x[j+1]$, ezért nem cserélünk.



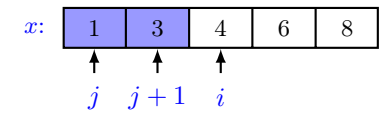
(e) Mivel $x[j] < x[j+1]$, ezért nem cserélünk.



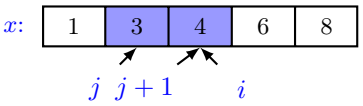
(f) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$.



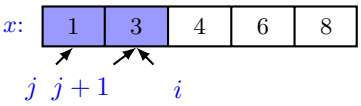
(g) Mivel $x[j] < x[j+1]$, ezért nem cserélünk.



(h) Mivel $x[j] < x[j+1]$, ezért nem cserélünk.



(i) Mivel $x[j] < x[j+1]$, ezért nem cserélünk.



(j) Mivel $x[j] < x[j+1]$, ezért nem cserélünk.

3.3. ábra. Buborékrendezés. A példán a 3.4. algoritmusnak megfelelően nyomon követhetjük az i és j indexek alakulását, valamint az elvégzett cserék eredményeit.

Az összehasonlítások száma most sem függ attól, hogy milyen tulajdonságokkal rendelkezik az algoritmus bemeneti tömbje.

Az algoritmus során az 5. sorban lévő csere csak akkor hajtódik végre, ha az előtte lévő feltétel ***igaz*** volt. Tehát a cserék száma már függ attól, hogy milyen a feldolgozandó tömb. Legrosszabb esetről akkor beszélhetünk, ha a bemeneti tömb csökkenő módon rendezett. Ilyenkor ugyanis minden vizsgált szomszédos pár esetén igaz lesz a 4. sorbeli feltétel, így a csere is végrehajtódik. Ebben az esetben a cserék száma megegyezik az összehasonlítások $\frac{n(n-1)}{2}$ számával. Mivel a csere művelethez három másolást kell alkalmaznunk, ezért legrosszabb esetben $3 \cdot \frac{n(n-1)}{2}$ másolást fogunk végezni. Legjobb esetnek tekinthetjük azt, amikor a tömb eleve növekvő módon rendezett. Ilyenkor az $x[j] > x[j+1]$ feltétel mindig ***hamis***, tehát egyetlen cserét sem hajtunk végre. Persze így másolást sem fogunk eszközölni. Átlagos esetben azt mondhatjuk, hogy a cserék száma

$$\frac{1}{2} \cdot \frac{n\,(n-1)}{2}, \tag{3.7}$$

a másolások száma pedig

$$\frac{3}{2} \cdot \frac{n\,(n-1)}{2}. \tag{3.8}$$

Összességében azt állapíthatjuk meg, hogy a buborékrendezés futási idő szempontjából ugyanolyan mérőszámokkal rendelkezik, mint a 3.1. fejezetben tárgyalt egyszerű cserés rendezés. Így ez a rendező algoritmus is $O\left(n^2\right)$-es futási idejű. ♣

A buborékrendezésről azt állapíthatjuk meg, hogy az ismertsége nincs összefüggésben a hatékonyságával. A következő fejezetben megvizsgáljuk miként lehet javítani ezen a rendező algoritmuson.

Javított buborékrendezés

Az előző fejezetben tárgyalt buborékrendezésnél bemutatott 3.3. példánál láttuk, hogy a buborékrendezés nem „veszi észre”, ha az algoritmus során a tömb rendezetté válik. Próbáljuk ezért erre a képességre megtanítani az algoritmust.

A buborékrendezés egyik jellemzője, hogy egy bejárás során az éppen vizsgált rész legnagyobb eleme biztosan a helyére kerül. Viszont ha a legnagyobb, illetve az azt megelőző néhány elem már eleve a helyén van, akkor a következő bejárásoknál kár lenne ezeket az elemeket ismét figyelembe venni. Hogyan tudjuk azt meghatározni, hogy mely elemek vannak a helyükön? Tudjuk, hogy amikor egy nagy elem a helyén van, akkor ott nem hajtunk végre cserét, hiszen a csere kimozdítaná őt a helyéről. Ezért csak azt kell vizsgálnunk, hogy egy bejárás során hol hajtjuk végre az utolsó cserét. A csere helyénél nagyobb indexű elemek már biztosan a helyükön vannak, ezért a következő bejárásnál azokat nem kell figyelembe venni. Tehát a soron következő bejárást csak a megelőző bejárás utolsó cseréjének helyéig kell végezni.

Ezzel az ötlettel javíthatjuk a buborékrendezés hatékonyságát. A konkrét pszeudokódot a 3.5. algoritmusban írjuk le. Első lépésként az i változónak adunk kezdeti értéket, amely az x tömb n elemszáma lesz (ld. 2. sor). Az i változónak ugyanaz a szerepe, mint a buborékrendezés 3.4. algoritmusának 2. sorában ciklusváltozóként létrehozott i változónak. Tehát i azt adja meg nekünk, hogy meddig kell még a tömböt vizsgálnunk egy konkrét bejárás során. A buborékrendezéshez képest lényeges különbség, hogy most az i értékét nem törvényszerűen egyesével csökkentjük, emiatt nem is számlálós ciklust használunk külső ciklusként. A 3. sorban kezdődő elöltesztelős ciklus fogja azt vizsgálni, hogy a i értéke nem csökkent-e még le túlságosan. Ha 2-nél kisebb lenne az i akkor már nem kell tovább folytatnunk az algoritmust, mivel ez már kevesebb mint kettő elemű résztömb feldolgozását jelentené. Az i értékét a ciklus végén, a 11. sorban módosítjuk majd.

3.5. Algoritmus Javított buborékrendezés

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: x – **T rendezett tömb**

```
1:  eljárás JavítottBuborékRendezés(címszerint x : T tömb, n : egész)
2:      i ← n
3:      ciklus amíg i ≥ 2
4:          idx ← 0
5:          ciklus j ← 1-től (i − 1)-ig
6:              ha x[j] > x[j + 1] akkor
7:                  x[j] ↔ x[j + 1]
8:                  idx ← j
9:              elágazás vége
10:         ciklus vége
11:         i ← idx
12:     ciklus vége
13: eljárás vége
```

Felhasznált változók és függvények

- x: A rendezni kívánt tömb. Az x tömb elemeinek összehasonlíthatónak kell lennie. Az eljárás a tömböt helyben rendezi.
- n: A paraméterként átadott tömb mérete.
- idx: Az utolsó csere helyét tároljuk benne. Ha $x[j]$-t és $x[j+1]$-et cseréljük, akkor az idx értéke j lesz.

Az 5. sorban indul el az a számlálós ciklus, amely egy konkrét bejárást valósít meg. A j ciklusváltozó 1-től $(i-1)$-ig megy, hasonlóan a 3.4. algoritmus 3. sorában lévő ciklushoz. A belső ciklus magjában megvizsgáljuk, hogy két szomszédos elem közül a kisebb indexűnek nagyobb-e az értéke mint a nagyobb indexűnek (ld. 6. sor). Ha $x[j] > x[j+1]$, akkor ki kell cserélnünk a két elemet (ld. 7. sor). Amint már említettük az utolsó csere helyét meg kell jegyeznünk. Ha csere történt, akkor az idx változóban eltároljuk, hogy a jelenlegi helyen (j-nél) történt meg a utolsó csere (ld. 8. sor). Az idx változónak korábban a konkrét bejárás megkezdése előtt, a 4. sorban 0 kezdeti értéket adtunk.

Amikor a belső ciklus végére értünk, akkor megtörtént egy konkrét bejárás. Ekkor meg kell határoznunk, hogy a soron következő bejárás meddig tartson majd. Ennek az utolsó csere helyéig kell majd mennie, ezért az i változó felveszi az idx változóban eltárolt értéket (ld. 11. sor).

Érdemes az algoritmus kapcsán megvizsgálni, hogy mi történik akkor, ha valamely i mellett egyetlen cserét sem hajtunk végre. Ekkor az idx változó kezdetben 0 lesz (4. sor), és mivel egyetlen cserét sem végzünk, így soha nem lépünk a 8. sorba. Tehát a belső ciklusban az idx értéke nem változik meg, végig 0 marad. Így viszont a külső ciklus magjának végén, a 11. sorban az i változó értéke is 0 lesz. Emiatt viszont a külső ciklusból is ki fogunk lépni, hiszen a bennmaradási $i \geq 2$ feltétel már nem lesz ***igaz***. Az látható így, hogy elértük azt a célunkat, hogy az algoritmus felismeri, ha a vizsgált résztömb (elsőtől az i-edik elemig) már rendezett, mivel rendezett tömbben nem végzünk cseréket.

3.4. Példa. Nézzük végig, hogy a javított buborékrendezés 3.5. algoritmusa miként rendezi a 3.3. példában már vizsgált 6, 1, 4, 3, 8 tömböt. A buborékrendezésnél láttuk, hogy számos felesleges lépés volt a rendezés során, mivel a 3.4. algoritmus nem ismerte fel, hogy már rendezetté vált a tömb. A konkrét megvalósítás lépéseit a 3.4. ábrán is nyomon követhetjük.

Az algoritmus elején az i változó értéke 5 lesz (ld. 2. sor). Mivel a külső ciklus belépési feltétele teljesül (ld. 3. sor), ezért belépünk a ciklusba. Az utolsó csere helyét jelző idx változó kezdeti értéke 0 lesz (ld. 4. sor). Belépünk a belső ciklusba és a j változó értéke kiindulásként 1 lesz (ld. 5. sor). Mivel az algoritmus 6. sorában lévő $x[j] > x[j+1]$ feltétel ***igaz*** lesz, ezért megcseréljük az első és a második elemet (ld. 7. sor). Az idx változóban eltároljuk az utolsó csere helyét (ld. 8. sor). Mindez nyomon követhető a 3.4a. ábrán is.

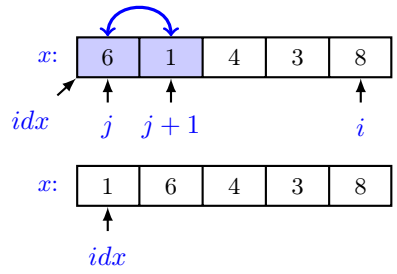
Ezt követően a belső ciklusban a j változó 2-re nő. Összehasonlítjuk a második és harmadik elemet, és mivel a második elem a nagyobb, ezért megcseréljük őket. A cserét követően az idx változó értékét is aktualizáljuk (ld. 3.4b. ábra). A j megint növekszik 1-gyel, majd összehasonlítjuk a harmadik és a negyedik elemet. Mivel a rendezettségük nem megfelelő, ezért cserélünk, és az idx változó értéke is 3 lesz (ld. 3.4c. ábra). A belső ciklus utolsó futásakor j értéke már 4. A negyedik elem kisebb az ötödiknél, ezért nem cserélünk és idx sem változik (ld. 3.3d. ábra). A belső ciklus véget ért, így megvalósítottuk a tömb első bejárását. Amint azt a buborékos rendezésnél is megismertük a legnagyobb elem a helyére, a tömb végére került. A 3.3d. ábrán látható, hogy a negyedik elem is a helyén van már, tehát teljesen felesleges lenne a következő bejárásnál az első négy elemből álló résztömböt végigjárni. Az idx változó jelenlegi értéke 3, hiszen a harmadik és negyedik elem között történt utoljára csere. Az i változó értékét az algoritmus 11. sorában az idx értékére változtatjuk, tehát a következő bejárásnál már csak az első három elemmel foglalkozunk.

Mivel i jelenlegi értéke 3, ezért bent maradunk a külső ciklusban. Az idx értéke ismét 0-ra módosul, majd $j = 1$-gyel elindul a belső ciklus végrehajtása. Mivel az első elem kisebb a másodiknál, ezért nem hajtunk végre cserét (ld. 3.4e. ábra). A j változó értékét 2-re növeljük. A második elem nagyobb a harmadiknál, ezért megcseréljük őket, majd az idx változó értéke 2-re módosul (ld. 3.4f. ábra). A belső ciklus végére értünk, az i értéke is 2-re változik.

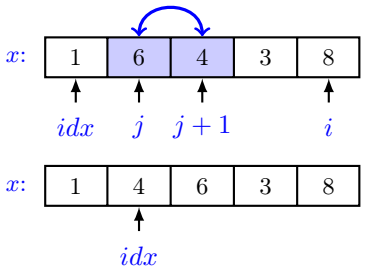
A külső ciklus bennmaradási feltételét ismét kiértékeljük és bennmaradunk a ciklusban. Az idx ismét 0 lesz, a belső ciklus kezdetén pedig j értéke 1 lesz. Mivel az első elem kisebb a másodiknál ezért nem kell cserélnünk (ld. 3.3j. ábra). A belső ciklus végét követően i is felveszi az idx változó 0 értékét. Mivel így már nem teljesül a külső ciklus bennmaradási feltétele, ezért az algoritmus futásának végére érünk. A tömbünk rendezetté vált. ¶

Futási idő elemzése. Vizsgáljuk meg a javított buborékrendezés futási idejét az összehasonlítások, valamint a cserék és másolások számának figyelembevételével.

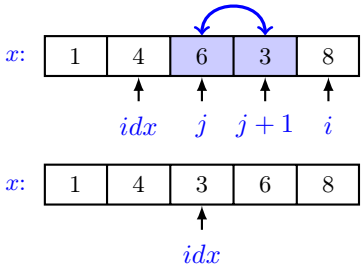
A korábbi három rendező algoritmus esetén láttuk, hogy az összehasonlítások száma mindig $\frac{n(n-1)}{2}$ volt. Most viszont más lesz a helyzet, mert a 3. és 12. sorok közötti külső ciklusnál nem tudjuk pontosan megmondani, hogy hányszor fut le a ciklus. Legrosszabb esetben $(n-1)$-szer hajtódik végre ez a ciklus. Ez pontosan akkor következik be, ha a rendezendő tömbünk kezdetben csökkenő módon volt rendezett. Ilyenkor az összehasonlítások száma a maximális $\frac{n(n-1)}{2}$ érték lesz. Legjobb esetnek azt tekinthetjük, ha egyetlen csere sem történik az algoritmus futása közben, azaz ha a tömbünk eleve rendezett. Ilyenkor a külső ciklusba egyszer lépünk be, a belső ciklusban pedig $(n-1)$-szer végzünk összehasonlítást. Mivel az összehasonlítások eredménye mindig ***hamis*** lesz, ezért nem cserélünk és az idx változó értékét sem módosítjuk. Így viszont az idx 0 marad, így az algoritmus 11. sorában az i is a 0 értéket veszi fel. Ennek viszont az a következménye, hogy a külső ciklus csak egyszer fut le, többször nem. Növekvő módon rendezett tömbök esetén tehát az összehasonlítások száma $n-1$ lesz. Átlagos esetben nem tudjuk pontosan



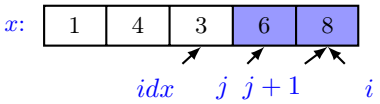
(a) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$. A cserét követően $idx \leftarrow j$.



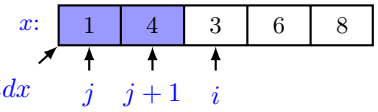
(b) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$. A cserét követően $idx \leftarrow j$.



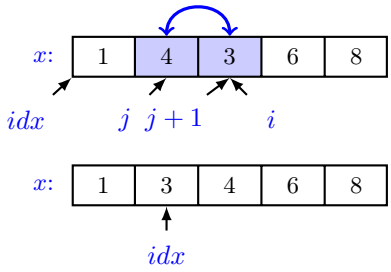
(c) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$. A cserét követően $idx \leftarrow j$.



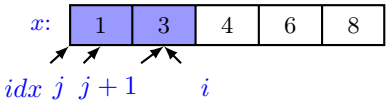
(d) Mivel $x[j] < x[j+1]$, ezért nem cserélünk.



(e) A belső ciklus végén $i \leftarrow idx$. Majd mivel $x[j] < x[j+1]$, ezért nem cserélünk.



(f) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$. A cserét követően $idx \leftarrow j$.



(g) A belső ciklus végén $i \leftarrow idx$. Majd mivel $x[j] < x[j+1]$, ezért nem cserélünk. Az idx változó értéke nulla marad, ezért az algoritmus véget ér.

3.4. ábra. Javított buborékrendezés. A példán a 3.5. algoritmusnak megfelelően nyomon követhetjük az i, j és idx indexek alakulását, valamint az elvégzett cserék eredményeit.

meghatározni, hogy hány összehasonlítás lesz, ám tapasztalat szerint az összehasonlítások száma inkább a legrosszabb esetbeli értékhez van közelebb, mint a legjobbhoz.

Mit mondhatunk a cserék számáról? Az biztos, hogy a cserék száma nem haladhatja meg az összehasonlítások számát, hiszen cserét mindig összehasonlítások után végzünk. Legrosszabb esetben, azaz amikor csökkenő módon rendezett a tömb, minden összehasonlítás után kell cserélni. Ilyenkor tehát a cserék száma $\frac{n(n-1)}{2}$ lesz. Legjobb esetben, vagyis amikor eleve növekvő módon rendezett a tömb, egyetlen cserét sem kell végrehajtanunk. Átlagosan a cserék száma a két szélső érték között lesz. A másolások számára pedig ugyanaz igaz, mint minden más esetben, tehát a cserék számának háromszorosával egyenlő.

Összességében az látható, hogy legrosszabb esetben a javított buborékrendezés futási ideje $O\left(n^2\right)$-es. Legjobb esetben viszont jelentős javulást sikerül elérni, ilyenkor az algoritmus $O\left(n\right)$-es. Bizonyítás nélkül közöljük, hogy átlagos esetben az algoritmus futási ideje $O\left(n^2\right)$-es. ♣

A futási idő elemzés alapján úgy érezhetjük, hogy a javított buborékrendezés algoritmusa elég hatékony, hiszen legjobb esetben a tömb méretével egyenesen arányos a futási idő. Ne feledjük viszont, hogy leggyakrabban az átlagos esettel kell foglalkoznunk, illetve szembesülhetünk a legrosszabb esettel is. Lényeges javulást egyébként a buborékrendezés hatékonyságához képest abban az esetben tudunk elérni, ha a tömbünk már eleve növekvő módon rendezett, vagy csak néhány helyen van eltérés a rendezettségtől.

Beillesztéses rendezés

A beillesztéses rendezés[2] nem abból indul ki mint az eddig tárgyalt rendező algoritmusaink. A korábbi algoritmusok ötlete az volt, hogy cseréket végzünk mindaddig, amíg az elemek a helyükre nem kerülnek. Csak abban volt lényeges különbség az algoritmusaink között, hogy mely elemek között végzünk cseréket. A beillesztéses rendezés más logikát követ, bár cseréket itt is fogunk végezni.

A beillesztéses rendezésnél az n elemű tömbünkből először csak egy elemmel, mégpedig az első elemmel foglalkozunk. Ha egy elemű tömbünk van, akkor az már eleve rendezett. Vegyünk most hozzá az egy elemű tömbhöz még egy elemet, a soron következőt, tehát a másodikat. A második elem lehet, hogy a helyén van, lehet hogy nem. Illesszük úgy a korábbi egy elemű tömbhöz a második elemet, hogy ezután a két elemű tömbünk már rendezett legyen. Nézzük ezután a harmadik elemet. Az első két elemből álló tömb már rendezett, ebbe a rendezett tömbbe illesszük be a megfelelő helyre a harmadik elemet úgy, hogy a három elemű tömb rendezett legyen. Ezt követően jöhet a negyedik elem, amelyet beillesztünk a három elemű rendezett tömbben a megfelelő helyre, így már egy négy elemű rendezett tömböt kapunk. Ezt a gondolatmenetet kövessük egészen addig, amíg utolsó lépésként az $n-1$ elemű rendezett tömbbe illesztjük be az n-edik elemet a megfelelő helyre, ami által az egész tömbünk rendezetté válik. A 3.5. ábrán végig tudjuk tekinteni a leírt folyamatot. Látható, hogy a beillesztések hatására végül az egész tömb rendezetté válik.

A beillesztéses rendezés ötletét már megismertük, de azért egy kérdést még nem válaszoltunk meg. Az aktuális beillesztendő elemet hogyan fogjuk a megfelelő helyre beszúrni? Ennek megvalósítása során szükségünk lesz az elemek cserélgetésére. A beillesztést ugyanis úgy oldjuk meg, hogy a beillesztendő elemet összehasonlítjuk a közvetlenül előtte álló elemmel. Ha a beillesztendő elem nem kisebb a bal szomszédjánál, akkor nem kell semmit tennünk (pl. a 3.5c. ábrán látható esetben). Ha viszont a beillesztendő elem kisebb a bal szomszédjánál, akkor megcseréljük őket, így a beillesztendő elem eggyel előrébb kerül. Ezután ismét összehasonlítjuk a beillesztendő elemet a bal szomszédjával. Vagy azt látjuk, hogy ismét cserélnünk kell, vagy már a helyén van. Az eljárást addig folytatjuk, amíg a beillesztendő elem a helyére nem kerül, azaz vagy nem kisebb már a bal szomszédjánál, vagy a tömb legelejére került.

A leírt ötletet konkrétan a 3.6. algoritmussal valósíthatjuk meg.

3.6. Algoritmus Beillesztéses rendezés

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: x – **T rendezett tömb**

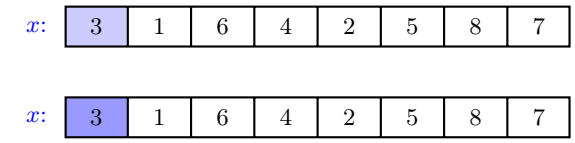
```
1: eljárás BeillesztésesRendezés(címszerint x : T tömb, n : egész)
2:     ciklus i ← 2-től n-ig
3:         j ← i − 1
4:         ciklus amíg (j > 0) ∧ (x[j] > x[j + 1])
5:             x[j] ↔ x[j + 1]
6:             j ← j − 1
7:         ciklus vége
8:     ciklus vége
9: eljárás vége
```

Felhasznált változók és függvények

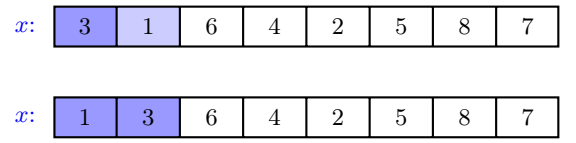
- x: A rendezni kívánt tömb. Az x tömb elemeinek összehasonlíthatónak kell lennie. Az eljárás a tömböt helyben rendezi.
- n: A paraméterként átadott tömb mérete.

Az algoritmus elején a 2. sorban kezdődő külső számlálós ciklusba lépünk be. Az i ciklusváltozóval jelöljük, hogy éppen a tömb hányadik elemét kívánjuk beilleszteni a megfelelő helyére. A fentebb leírt életnek megfelelően mindig feltételezzük, hogy a tömb első $i-1$ eleme már rendezett állapotban van. Az i kiindulási értéke 2, így ekkor még biztos teljesül, hogy a második elemet megelőző egy elemű tömb biztosan rendezett. Az i ciklusváltozó értékét az algoritmus előrehaladtával egészen a tömb méretéig (n) növeljük majd.

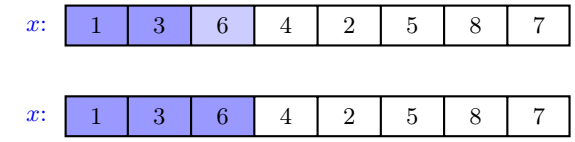
[2] Angolul: insertion sort



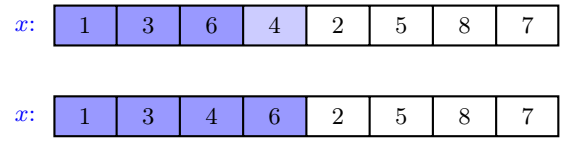
(a) Az első elemet beillesztjük a neki megfelelő helyre, azaz nem teszünk semmit.



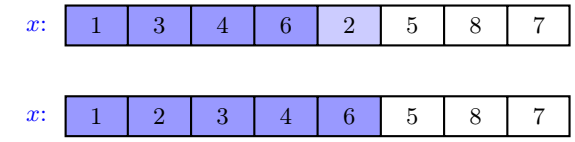
(b) A második elemet beillesztjük az egy elemű rendezett tömbbe úgy, hogy két elemű rendezett tömböt kapjunk.



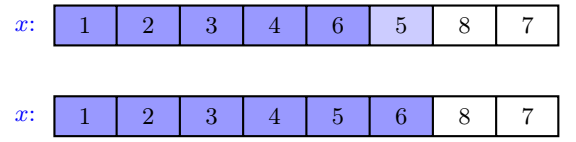
(c) A harmadik elemet beillesztjük a két elemű rendezett tömbbe úgy, hogy három elemű rendezett tömböt kapjunk.



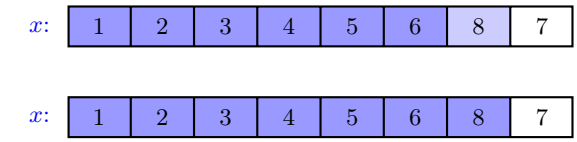
(d) A negyedik elemet beillesztjük a három elemű rendezett tömbbe úgy, hogy négy elemű rendezett tömböt kapjunk.



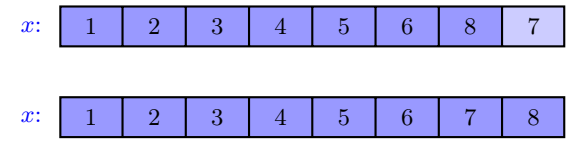
(e) Az ötödik elemet beillesztjük a négy elemű rendezett tömbbe úgy, hogy öt elemű rendezett tömböt kapjunk.



(f) A hatodik elemet beillesztjük az öt elemű rendezett tömbbe úgy, hogy hat elemű rendezett tömböt kapjunk.



(g) A hetedik elemet beillesztjük a hat elemű rendezett tömbbe úgy, hogy hét elemű rendezett tömböt kapjunk.



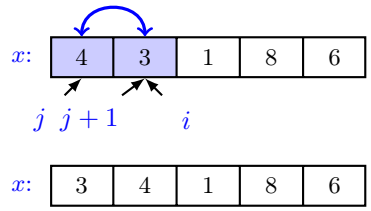
(h) A nyolcadik elemet beillesztjük a hét elemű rendezett tömbbe úgy, hogy nyolc elemű rendezett tömböt kapjunk.

3.5. ábra. Beillesztéses rendezés bevezetése. A beszúrandó (világos kék) elemek a már rendezett (sötét kék) tömbökben a megfelelő helyre kerülnek.

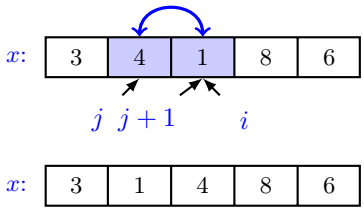
A külső ciklus belsejében kell megoldanunk azt, hogy az i-edik tömbelem a helyére kerüljön. Ahogy említettük ennek érdekében a tőle balra lévő elemmel fogjuk összehasonlítani, és ha szükséges akkor cserélni. Az i-edik elem bal oldalán lévő elemet indexeljük a j változóval, melynek kezdet értéke $i-1$ lesz (ld. 3. sor). A 4. sorban kezdődő elöltesztelős ciklus valósítja meg a beillesztendő elem balján lévő elemmel való összehasonlításokat és cseréket. A ciklus belépési, illetve bennmaradási feltétele egyrészt, hogy a beszúrandó elem balján lévő elem indexe (ezt jelöli a j változó) még valós index legyen, azaz nagyobb legyen 0-nál. Másik feltétel, hogy a beillesztendő elem balján lévő elem nagyobb legyen a beillesztendő elemnél, ugyanis ilyenkor kell cserélni és tovább vizsgálódni. Mivel a beillesztendő elem balján a j-edik elem van és a beillesztendő elem pedig folyamatosan halad a tömb eleje felé, ezért a beillesztendő elemet nem végig i-vel, hanem $(j+1)$-gyel tudjuk indexelni. Ezért a ciklusfeltételben a $x[j] > x[j+1]$ vizsgálatot végezzük el. (Vegyük észre, hogy kezdetben $j+1$ pont az i-vel azonos.)

Ha beléptünk a belső ciklusba, akkor mindkét megvizsgált feltételünk ***igaz*** volt, tehát cserélnünk kell a két szomszédos elemet (ld. 5. sor). A cserét követően a j indexet is csökkentenünk kell 1-gyel, hiszen a beillesztendő elem eggyel előrébb került a tömbben, így a bal oldali szomszédja indexének (azaz j-nek) is 1-gyel kisebbnek kell lennie.

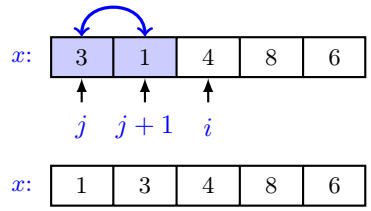
Ha a belső ciklus feltétele már nem ***igaz***, akkor biztosak lehetünk benne, hogy a beillesztendő elem a helyére került, mivel vagy a tömb elején van (ekkor lesz $j=0$), vagy a bal szomszédja nem nagyobb már nála. Ekkor viszont léphetünk tovább a következő beillesztendő elemre, amit a külső ciklus automatikusan megvalósít.



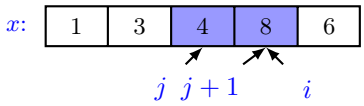
(a) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$.



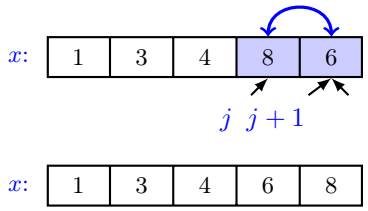
(b) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$.



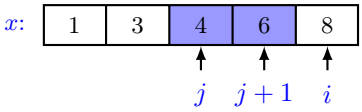
(c) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$.



(d) Mivel $x[j] < x[j+1]$, ezért nem cserélünk.



(e) Mivel $x[j] > x[j+1]$, ezért $x[j] \leftrightarrow x[j+1]$.



(f) Mivel $x[j] < x[j+1]$, ezért nem cserélünk.

3.6. ábra. Beillesztéses rendezés. A példán a 3.6. algoritmusnak megfelelően nyomon követhetjük az i és j indexek alakulását, valamint az elvégzett cserék eredményeit.

3.5. Példa. A 3.6. ábrán nyomon követhetjük, hogy a beillesztéses rendezés 3.6. algoritmusa miként rendezi a 3, 4, 1, 8, 6 elemekből álló x tömböt.

Az algoritmus kezdetén belépünk a 2. sorban induló ciklusba. A beillesztendő elemet jelölő i index értéke 2 lesz. A ciklusbon belül j felveszi az 1 értéket (ld. 3. sor). Ezt követően megvizsgáljuk a 4. sorban kezdődő belső ciklus belépési feltételét. A j index értéke nullánál nagyobb, illetve a j-edik elem nagyobb a $(j+1)$-edik elemnél, így belépünk a ciklusba. A cikluson belül felcseréljük a j és a $j+1$ indexű elemeket (ld. 5. sor), majd j értékét eggyel csökkentjük (6. sor). Ezeket a lépéseket szemlélteti 3.6a. ábra. A vezérlés ezt követően ismét a 4. sorba kerül. Mivel j értéke nullára csökkent, ezért nem lépünk be újra a belső ciklusba.

A külső ciklusban i értéke 3-ra nő, majd j 2 lesz. Összehasonlítjuk a második és harmadik elemet, és mivel a harmadik elem kisebb, ezért megcseréljük őket (ld. 3.6b. ábra). Ezután j 1-re csökken, majd összehasonlítjuk az első és a második elemet. Mivel az első elem a nagyobb, ezért ismét cserélünk (ld. 3.6c. ábra). Ezt követően j értékét tovább csökkentjük, és mivel 0 lesz az értéke, így nem maradunk bent a belső ciklusban. Láthatjuk, hogy a beillesztendő 1 értékű elem a tömb legelejére került.

A külső ciklusban i-t 4-re növeljük, j kezdeti értéke pedig 3 lesz. Mivel a beillesztendő negyedik elem nagyobb, mint a bal szomszédja, ezért helyben hagyjuk, a belső ciklusba be sem lépünk (ld. 3.6d. ábra).

Elérkezünk az utolsó elem beillesztéséhez, azaz i értéke már 5. A j kezdeti értéke 4, és a belső ciklusba belépünk, mivel $x[4] > x[5]$. Megcseréljük a negyedik és ötödik elemet (ld. 3.6e. ábra), majd j-t 3-ra csökkentjük. Mivel a negyedik elem nagyobb a harmadiknál, ezért nem maradunk bent a belső ciklusban (ld. 3.6f. ábra). A külső ciklusból is kilépünk, mivel az i értékét már nem tudjuk tovább növelni. Az algoritmus végére értünk a tömbünk növekvő módon rendezetté vált. ¶

Futási idő elemzése. A futási idő elemzésénél az algoritmus két műveletét fogjuk elemezni, mivel ezek tekinthetők a leglassabbaknak. (A többi lépés ugyanis csak indexekkel dolgozik, ami egész értékekkel végzett művelet, így általában gyorsan megvalósítható.) Egyrészt vizsgálni fogjuk, hogy a 4. sorbeli feltételek közül a tömbelemek viszonyát vizsgáló $x[j] > x[j+1]$ összehasonlítást hányszor hajtjuk végre. Másrészt elemezzük az 5. sorban lévő cserék számát.

Kezdjük az elemzésünket a legjobb esettel, tehát amikor a tömbünk eleve növekvő módon rendezett. Ilyenkor minden i érték mellett egyszer megvizsgáljuk az $x[j] > x[j+1]$ feltétel teljesülését. A feltétel minden esetben ***hamis*** lesz, így a belső ciklusba nem lépünk be. Ezek szerint összehasonlításból $(n-1)$-et, cseréből pedig egyet sem fogunk elvégezni.

Legrosszabb esetnek azt tekinthetjük, ha a tömbünk fordított módon rendezett. Ilyenkor az aktuális beillesztendő elem mindig a tömb elejére fog kerülni. Ehhez a maximális számú összehasonlítást és ugyanannyi cserét kell végeznünk. Tehát az összehasonlítások és cserék száma is

$$\frac{n\,(n-1)}{2} \tag{3.9}$$

lesz. A másolások száma a cserék számának háromszorosa, tehát azokból $3 \cdot \frac{n(n-1)}{2}$ darabot kell elvégezni.

Átlagos esetben nem tudunk pontos értéket mondani, mind az összehasonlítások, mind a cserék, illetve másolások száma valahol a két szélsőérték között lesz. Összességében azt mondhatjuk, hogy az algoritmus futási ideje $O\left(n^2\right)$-es. Ha rendezett a tömbünk, akkor viszont $O\left(n\right)$-es a futási idő. ♣

Megjegyzés

A beillesztéses rendezés legnagyobb előnye, hogy a megismert működési elv olyan esetekben is működik, amikor nem tömbökkel, hanem más adatszerkezetekkel[a] dolgozunk. A tömbök egyik jellemzője, hogy a feldolgozandó adatokat (a tömb elemeit) már kezdetben is ismerjük, tudjuk, hogy pontosan hány adatot kell rendeznünk.

Később megjelenő adatszerkezeteknél (például a listáknál) viszont lehetséges, hogy a lista folyamatosan bővül, az adott program futása közben veszünk fel új elemeket a listába. Amennyiben a listánk rendezett, akkor az újonnan megjelenő elemet a beillesztéses rendezés algoritmusának alapötletére támaszkodva illeszthetjük be a listába a megfelelő helyre.

[a] Ld. Szénási Sándor: Algoritmusok, adatszerkezetek II. (ÓE-NIK-503). 2014, Óbudai Egyetem, Neumann János Informatikai Kar

Javított beillesztéses rendezés

Ebben a fejezetben megnézzük, hogy miként lehet javítani az előző fejezetben megismert beillesztéses rendezés futási idején. Egy további ötletet szeretnénk megvizsgálni, de a beillesztéses rendezés alapötletén nem akarunk módosítani. Tehát most is abból fogunk kiindulni, hogy a beillesztendő elem előtti tömbelemek rendezettsége mellett, a megfelelő helyre szúrjuk be a beillesztendő elemet. Viszont az aktuális elemet nem cserék egymás utáni végrehajtásával fogjuk a helyére vinni, hanem más módszert használunk.

Nézzük például a 3.5e. ábrán látható esetet. Az ötödik helyen álló 2-t szeretnénk az előtte lévő négy elemű rendezett tömbben a helyére illeszteni. Ezt megvalósíthatjuk úgy is, hogy a beszúrandó elem előtt álló, nála nagyobb elemeket (3, 4 és 6) eggyel hátrébb toljuk, majd ezt követően a beillesztendő elemet már a helyére tudjuk tenni. Hogyan fogunk tudni egy elemet egy hellyel hátrébb tolni? Ha a hátrébb mozdítandó elem a j-edik elem, akkor az $x[j+1]$ be kell átmásolnunk $x[j]$-t. Ilyen egyszerűen megoldható az elem hátrébb mozgatása. Egy esetben viszont problémával fogunk találkozni. Ha pont a beillesztendő elem előtti elemet akarjuk hátrébb tolni (3.5e. ábrán például a kezdetben negyedik helyen álló 6-ot), akkor a hátrébb másoláskor a beillesztendő elemet felülírnánk, így elvesztenénk a tömbből. Ennek kivédése érdekében azt tesszük, hogy az elemek hátra tologatása előtt kimentjük a beillesztendő elemet egy segédváltozóba. Ezt követően már nyugodtan hátrébb tolhatjuk a beillesztendő elemnél nagyobb elemeket, majd a beillesztendő elemet a segédváltozóból visszamásoljuk a megtalált új helyére.

A javított beillesztéses rendezés konkrét megvalósítását a 3.7. algoritmusban láthatjuk, ami természetesen nagyon hasonlít a beillesztéses rendezés 3.6. algoritmusához. Egy külső ciklussal fogunk végigmenni az aktuálisan beszúrandó elemeken. Egy belső ciklussal oldjuk meg, hogy a beszúrandó elem előtti, nála nagyobb elemek hátrébb kerüljenek. Ami lényeges különbség, hogy nem használunk cseréket, illetve a beillesztendő elem átmeneti tárolására egy segédváltozót alkalmazunk.

3.7. Algoritmus Javított beillesztéses rendezés

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: x – **T rendezett tömb**

```
1:  eljárás JavítottBeillesztésesRendezés(címszerint x : T tömb, n : egész)
2:      ciklus i ← 2-től n-ig
3:          j ← i − 1
4:          segéd ← x[i]
5:          ciklus amíg (j > 0) ∧ (x[j] > segéd)
6:              x[j + 1] ← x[j]
7:              j ← j − 1
8:          ciklus vége
9:          x[j + 1] ← segéd
10:     ciklus vége
11: eljárás vége
```

Felhasznált változók és függvények

- x: A rendezni kívánt tömb. Az x tömb elemeinek összehasonlíthatónak kell lennie. Az eljárás a tömböt helyben rendezi.
- n: A paraméterként átadott tömb mérete.
- *segéd*: Segédváltozó, mely mindig az aktuális beillesztendő elemet tartalmazza.

Az algoritmus 2. sorában belépünk a külső számlálós ciklusba. Az i ciklusváltozó jelöli, hogy éppen melyik elemet kívánjuk az őt megelőző rendezett résztömbben a helyére tenni. Az első beillesztendő elem természetesen a második tömbelem lesz. A cikluson belül a j indexváltozót használjuk a beillesztendő elem előtti tömbrész bejárására. A j változó kezdeti értéke $i-1$ lesz (ld. 3. sor), így a beillesztendő elem bal szomszédjától indul majd ez a bejárás. A beillesztendő elemet kimentjük egy átmeneti változóba (4. sor).

Az 5. sorban kezdődő ciklussal valósítjuk meg a beillesztendő elem előtti elemek tényleges bejárását. Ebbe a ciklusba akkor lépünk be, illetve addig maradunk benne, amíg a j változó még valós index, illetve amíg a j-edik tömbelem még nagyobb a beillesztendő elemnél. Ez utóbbi feltétel jelenti azt, hogy a j-edik elemet még hátrébb kell mozgatnunk. (Érdemes összevetni az 5. sor feltételeit a 3.6. algoritmus

4. sorában lévő feltételekkel.) Ha belépünk a belső ciklusba, akkor hátrébb mozgatjuk a j-edik elemet (6. sor), majd előrébb lépünk a tömbben, azaz j-t 1-gyel csökkentjük (ld. 7. sor).

A belső ciklusból két esetben lépünk ki. Az egyik, ha j értéke nullára csökken. Ilyenkor még a tömb első eleme is nagyobb volt a beillesztendő elemnél. Emiatt az első elemet is hátrébb toltuk egy hellyel. Most következne a soron következő, azaz a nulladik elem vizsgálata ilyen elem viszont nincs a tömbben. Mindez azt jelenti, hogy a beillesztendő elemnek az első helyre kell kerülnie, amely helyről már hátrébb is toltuk a korábbi első elemet. A másik lehetőség a ciklusból való kilépésre, amikor a j-edik elem már nem nagyobb a beillesztendő elemnél. Ilyenkor a beillesztendő elemet a j-edik elem után, azaz a $(j+1)$-edik helyre kell beszúrni.

A ciklusból kilépve a beillesztendő elemet, amelyet átmenetileg a *segéd* változóban tároltunk bemásoljuk a tömb $(j+1)$-edik helyére (9. sor). Ezt követően a külső ciklusban továbblépünk a következő beillesztendő elemre, amit szintén a helyére teszünk az előtte lévő már korábban rendezett résztömbben. Amikor a tömb utolsó elemét is beszúrtuk a megfelelő helyre, kilépünk a külső ciklusból, és az algoritmus végére érünk. A tömbünk pedig rendezetté vált.

Megjegyzés

A 3. fejezetben említettük, hogy az ismertetésre kerülő rendezések mind ún. helyben rendezések. Emiatt az algoritmusok memóriabeli helyigénye megegyezik a rendezendő tömb elemszámával, plusz még egy tömbben tárolt **T** típusú elemet kell eltárolni a memóriába, hiszen a cserék megvalósításához szükséges egy átmeneti változó. A javított beillesztéses rendezésnél viszont nem hajtunk végre cseréket. Ennél az algoritmusnál azonban használunk egy átmeneti változót (*segéd*) a beillesztendő elem tárolására. Így végeredményben ennél az algoritmusnál is $n+1$ darab **T** típusú változót kell egyszerre a memóriában tárolni.

3.6. Példa. Nézzük végig, hogy a javított beillesztéses rendezés 3.7. algoritmusa hogyan rendezni a 4, 3, 1, 8, 6 elemekből álló x tömböt. A rendezés lépéseit a 3.7. ábra is szemlélteti.

Az algoritmus kezdetén belépünk a külső ciklusba (2. sor), az i kezdeti értéke 2 lesz. Először tehát a második elemet szeretnénk a helyére illeszteni az őt megelőző egy elemű résztömbben. A j változó kezdeti értéke 1 lesz (3. sor). Az algoritmus 4. sorában a *segéd* változóba bemásoljuk a beillesztendő elemet (ld. 3.7a. sor). Megvizsgáljuk a belső ciklus belépési feltételeit (ld. 5. sor), amelyek jelenleg teljesülnek, mivel az első elem értéke nagyobb a beillesztendő elem értékénél. Belépünk a belső ciklusba és az első elemet átmásoljuk a második helyre (ld. 6. sor). Természetesen ettől még az első helyen is megmarad a korábbi érték, tehát a 4 érték jelenleg kétszer fordul elő a tömbben. (Vegyük itt észre, hogy ha a beillesztendő elemet nem másoltuk volna be a *segéd* változóba, akkor az első elem másolásánál felülírtuk és így el is vesztettük volna.) A belső ciklusban eggyel csökkentjük még j értékét (ld. 7. sor). A belső ciklusban végbemenő folyamatokat a 3.7b. ábrán is nyomon követhetjük. A vezérlés ismét a belső ciklus bennmaradási feltételének kiértékeléséhez ugrik. Mivel a $j=0$, ezért nem maradunk bent a ciklusban. A ciklusból kilépve, a *segéd* változóban tárolt beillesztendő elemet bemásoljuk a tömb első helyére (ld. 9. sor). Ezzel a beillesztendő elem a helyére került, a tömb első két eleméből álló résztömb rendezetté vált (ld. 3.7c. ábra).

A vezérlés a külső ciklus elejére ugrik, i értékét 3-ra növeljük. A cikluson belül j kezdeti értéke 2 lesz, a harmadik (beillesztendő) elemet kimásoljuk a *segéd* változóba (ld. 3.7d. ábra). Megvizsgáljuk a belső ciklus belépési feltételeit. Mivel a beillesztendő elem kisebb mint a második elem, ezért belépünk a belső ciklusba. A második elemet átmásoljuk a harmadik helyre, majd j értékét eggyel csökkentjük (3.7e. ábra). A ciklusban maradás feltételeit kiértékeljük és látjuk, hogy az első elem is nagyobb, mint a beillesztendő elem. Így bent maradunk a ciklusban és az első elemet is eggyel hátrébb másoljuk, majd j értékét ismét csökkentjük (ld. 3.7f. ábra). A ciklus bennmaradási feltétele most már nem teljesül, mert a j nullára csökkent. Így a belső ciklus utáni sorra ugrik a vezérlés, ahol a beillesztendő elemet az első helyre másoljuk (ld. 3.7g. ábra). Most már az első három elemből álló résztömb is rendezett.

A külső ciklusban i értéke 4-re növekszik. A j változó értékét háromra állítjuk, a negyedik elemet pedig kimásoljuk a *segéd* változóba (ld. 3.7h. ábra). Mivel a harmadik elem kisebb mint a beillesztendő elem, ezért nem lépünk be a belső ciklusba (ld. 3.7i. ábra). A beillesztendő elemet a *segéd* változóból visszaírjuk a kezdeti helyére (ld. 3.7j. ábra).

Következik az $i=5$ eset, azaz az ötödik elemet szeretnénk a helyére illeszteni. Ennek érdekében j-t 4-re állítjuk, az ötödik elemet pedig kimentjük a *segéd*-be (3.7k. ábra). Megvizsgáljuk a belső ciklus

belépési feltételét, ami most ***igaz*** lesz. Belépve a ciklusba a negyedik elemet az ötödikbe másoljuk, majd j értékét 3-re csökkentjük (3.7l. ábra). A belső ciklus bennmaradási feltétele ***hamis*** lesz (ld. 3.7m. ábra), ezért kilépünk a ciklusból. A negyedik helyre visszaírjuk a beillesztendő elemet (ld. 3.7n. ábra). A külső ciklusnak is a végére értünk, a tömbünk rendezetté vált. ¶

Futási idő elemzése. A futási idő vizsgálatánál nézzük meg először, hogy hányszor végezzük el a belső ciklus feltételében lévő $x[j] > segéd$ összehasonlítást. Könnyen belátható, hogy ez pont annyiszor kerül kiértékelésre, mint a beillesztéses rendezésnél vizsgált $x[j] > x[j+1]$ feltétel. Így legrosszabb esetben $\frac{n(n-1)}{2}$, míg legjobb esetben $n-1$ darab összehasonlítást fogunk elvégezni. Minden más esetben valahol a két szélsőérték között lesz az összehasonlítások száma.

A javított beillesztéses rendezésnél egyetlen cserét sem hajtunk végre. Ezért itt csak a másolások számát vizsgáljuk. Az algoritmus 6. sorában lévő másolást annyiszor hajtjuk végre ahányszor beléptünk a belső ciklusba. Ezek száma lényegében ugyanannyi mint a beillesztéses rendezésnél a cserék száma volt, tehát legrosszabb esetben $\frac{n(n-1)}{2}$, legjobb esetben pedig nulla.

Vegyük észre, hogy a külső ciklusban mindig elvégzünk két mozgatást a 4. és a 9. sorokban. Mivel a külső ciklus $(n-1)$-szer fut le, így ezen másolások száma $2 \cdot (n-1)$ lesz.

Így összességében legrosszabb esetben

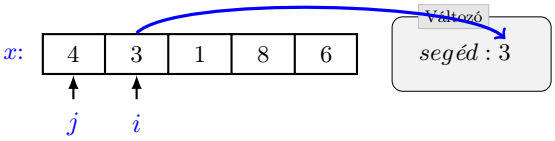
$$2 \cdot (n-1) + \frac{n\,(n-1)}{2} \tag{3.10}$$

darab másolást hajtunk végre. Ez a szám általában kisebb a beillesztéses rendezésnél végrehajtott másolások számánál, hiszen ott legrosszabb esetben $3 \cdot \frac{n(n-1)}{2}$ másolást végeztünk. Legjobb esetben a másolások száma $2 \cdot (n-1)$, ami természetesen több mint a beillesztéses rendezésnél talált nulla darab másolás.

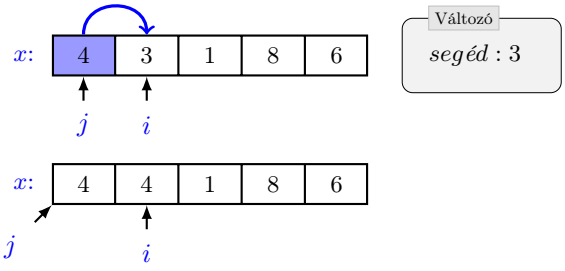
Összegezve a futási időről tett megállapításokat azt mondhatjuk, hogy a javított beillesztéses rendezés futási ideje valamelyest javult más algoritmusok futási idejéhez képest, aminek oka, hogy nem hajtunk végre cseréket. Ettől az algoritmus futási ideje sajnos még mindig $O\left(n^2\right)$-es. ♣

Megjegyzés

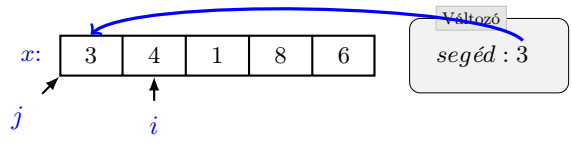
A hat legismertebb rendező algoritmus tárgyalásának végén felmerülhet bennünk a kérdés, hogy a bemutatott algoritmusok közül melyiket érdemes használni. Korrekt válasz csak a konkrét feladat ismeretében adható, de általánosságban kijelenthető, hogy két algoritmus használata ajánlott. Egyik a minimumkiválasztásos rendezés, amelynél a cserék számát sikerült minimalizálnunk. Tehát ha olyan környezetben dolgozunk, ahol a cserék megvalósítása lassú (pl. fájlon belül kell rendezést végrehajtani), akkor érdemes a minimumkiválasztásos rendezést használni. A másik ajánlott algoritmus a javított beillesztéses rendezés, amelynek fő erőssége, hogy általában nem kell a maximális számú összehasonlítást elvégezni, valamint cserék helyett másolásokkal tudunk operálni.



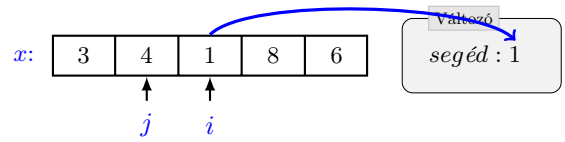
(a) Második elem beillesztése. Az $x[2]$ elemet eltároljuk a *segéd* változóban.



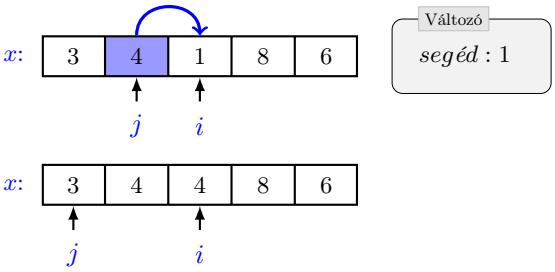
(b) Második elem beillesztése. Mivel $x[1] > segéd$, ezért $x[1]$-et hátrébb mozgatjuk.



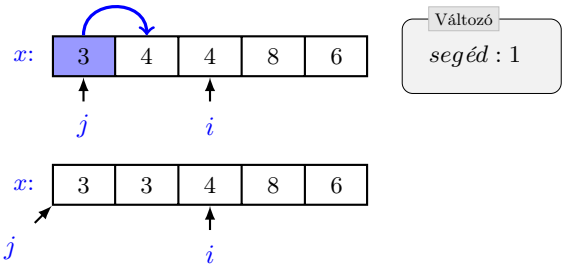
(c) Második elem beillesztése. A *segéd*-ben tárolt értéket az $x[1]$-be mozgatjuk.



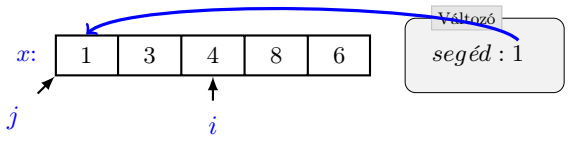
(d) Harmadik elem beillesztése. Az $x[3]$ elemet eltároljuk a *segéd* változóban.



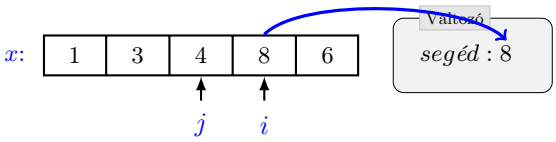
(e) Harmadik elem beillesztése. Mivel $x[2] > segéd$, ezért $x[2]$-t hátrébb mozgatjuk.



(f) Harmadik elem beillesztése. Mivel $x[1] > segéd$, ezért $x[1]$-et hátrébb mozgatjuk.

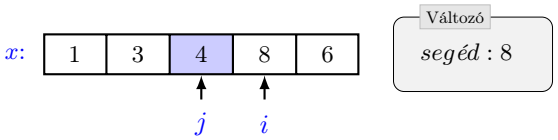


(g) Harmadik elem beillesztése. A *segéd*-ben tárolt értéket az $x[1]$-be mozgatjuk.

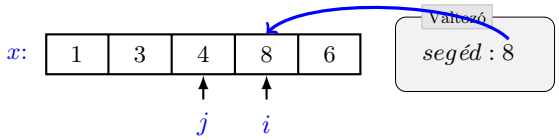


(h) Negyedik elem beillesztése. Az $x[4]$ elemet eltároljuk a *segéd* változóban.

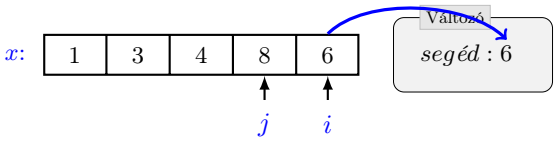
3.7. ábra. Javított beillesztéses rendezés. A példán a 3.7. algoritmusnak megfelelően nyomon követhetjük az i és j indexek alakulását, valamint a *segéd* változó és az aktuális tömbelemek közötti mozgatások eredményeit.



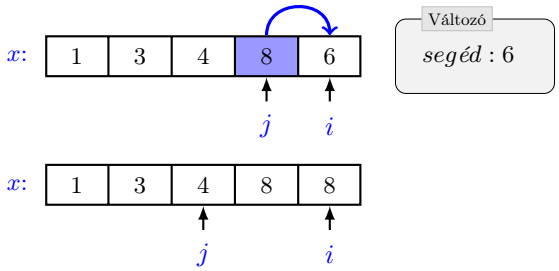
(i) Negyedik elem beillesztése. Mivel $x[3] < segéd$, ezért nem teszünk semmit.



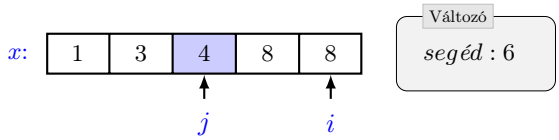
(j) Negyedik elem beillesztése. A *segéd*-ben tárolt értéket az $x[4]$-be mozgatjuk.



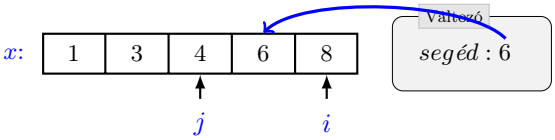
(k) Ötödik elem beillesztése. Az $x[5]$ elemet eltároljuk a *segéd* változóban.



(l) Ötödik elem beillesztése. Mivel $x[4] > segéd$, ezért $x[4]$-et hátrébb mozgatjuk.



(m) Ötödik elem beillesztése. Mivel $x[3] < segéd$, ezért nem teszünk semmit.



(n) Ötödik elem beillesztése. A *segéd*-ben tárolt értéket az $x[4]$-be mozgatjuk.

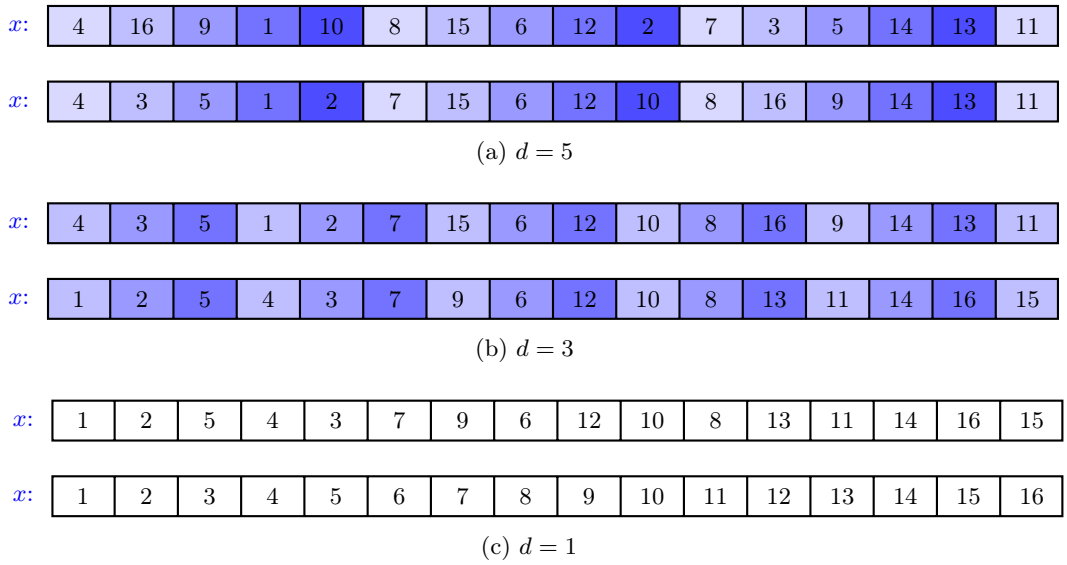
3.7. ábra. Javított beillesztéses rendezés (folyt.). A példán a 3.7. algoritmusnak megfelelően nyomon követhetjük az i és j indexek alakulását, valamint a *segéd* változó és az aktuális tömbelemek közötti mozgatások eredményeit.

Shell rendezés

A Shell rendezést a javított beillesztéses rendezés átalakításával kapjuk. A javított beillesztéses rendezés egyik hátránya, hogy ha egy kis érték a tömb vége felé van kezdetben, akkor sok összehasonlítást és másolást kell annak érdekében végezni, hogy a végső helyére kerüljön. Ennek oka, hogy az elemeket mindig a közvetlen szomszédok mozgatásával juttatjuk el a megfelelő helyükre. A Shell rendezés ötlete az, hogy először az egymástól valamilyen nagyobb távolságra elhelyezkedő elemek között végez összehasonlításokat és mozgatásokat. Így egy a tömb vége felé lévő kis érték nagy lépésközökkel tud eljutni a végső helye közelébe, ezért jóval kevesebb műveletet kell elvégezni a rendezés érdekében.

Shell rendezésnél tehát először választunk valamilyen távolságot (ezt fogjuk d-vel jelölni), majd a tömb ilyen távolságra lévő elemei között végzünk rendezéseket. Ezt szemlélteti például $d = 5$ esetén a 3.8a. ábra. Láthatjuk, hogy az azonos színárnyalattal jelölt, egymástól öt távolságra lévő elemekből alkotott résztömbök rendezetté váltak. Ennek fontos eredménye az, hogy az elemek nagy része a végleges helyének közelébe került. Például a tömb első öt helyére az öt legkisebb elem került. Így ezen elemek sorba rendezése már kevés művelet elvégzésével megvalósítható.

A rendezés folyamán a d távolságértéket folyamatosan csökkenteni fogjuk, így például a következő értéke lehet 3. Ekkor az egymástól három távolságra lévő elemekből alkotott résztömböket rendezzük, ahogy a 3.8b. ábrán is látható. A távolság érték változtatása mindig a $d = 1$ esettel ér véget, mely lényegében megfelel a korábban már megismert javított beillesztéses rendezésnek. Ekkor viszont már az elemek nagy valószínűséggel a végső helyük közelében vannak, így viszonylag kevés művelettel megvalósítható a végleges finomhangolás. A $d = 1$ esetet a 3.8c. ábra szemlélteti.



(a) $d = 5$

(b) $d = 3$

(c) $d = 1$

3.8. ábra. Shell rendezés ötletének bemutatása különböző távolságok használatával. Az egyes színárnyalatok az egymástól d távolságra lévő elemeket emelik ki. Látható, hogy a d távolságra lévő elemekből alkotott résztömbök mindig rendezetté válnak.

Foglaljuk össze, hogyan is működik a Shell rendezés. Kezdetben választunk egy kiindulási távolság értéket (d). A választott távolságra lévő elemekből álló résztömbön belül elérjük, hogy az elemek rendezetté váljanak. Ehhez lényegében a résztömbön belül egy javított beillesztéses rendezést hajtunk végre. A javított beillesztéses rendezés algoritmusán ehhez csak annyit kell módosítanunk, hogy nem a szomszédos elemekkel, hanem az egymástól d távolságra lévő elemekkel foglalkozunk. Amikor a vizsgált résztömbök rendezetté váltak, akkor csökkentjük a d távolság értékét, és ismét rendezzük a kialakult, egymástól d távolságra lévő elemekből álló résztömböket. A d csökkentését addig folytatjuk, amíg el nem éri az egy értéket. Természetesen $d = 1$-gyel is végrehajtjuk a rendezést (a teljes tömbön), és a tömbünk rendezetté válik.

A Shell rendezés konkrét megvalósítását a 3.8. algoritmusban írjuk le.

Első lépésként meghatározzuk, hogy mi legyen a kezdeti távolságérték. Ehhez a rendezendő x tömb n elemszámától függő KezdetiTávolság függvényt hívjuk meg (ld. 2. sor). (A kezdeti távolság értékének

3.8. Algoritmus Shell rendezés

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: x – **T rendezett tömb**

```
1: eljárás ShellRendezés(címszerint x : T tömb, n : egész)
2:     d ← KezdetiTávolság(n)
3:     ciklus amíg d ≥ 1
4:         ciklus i ← (d + 1)-től n-ig
5:             j ← i − d
6:             segéd ← x[i]
7:             ciklus amíg (j > 0) ∧ (x[j] > segéd)
8:                 x[j + d] ← x[j]
9:                 j ← j − d
10:            ciklus vége
11:            x[j + d] ← segéd
12:        ciklus vége
13:        d ← KövetkezőTávolság(d)
14:    ciklus vége
15: eljárás vége
```

Felhasznált változók és függvények

- x: A rendezni kívánt tömb. Az x tömb elemeinek összehasonlíthatónak kell lennie. Az eljárás a tömböt helyben rendezi.
- n: A paraméterként átadott tömb mérete.
- *segéd*: Segédváltozó, mely mindig az aktuális beillesztendő elemet tartalmazza.
- d: Megadja, hogy az egymástól mekkora távolságra lévő elemek között végzünk vizsgálatokat.
- KezdetiTávolság(n): Függvény, amely meghatározza, hogy az adott tömbmérethez milyen kezdeti d távolságérték tartozik.
- KövetkezőTávolság(d): Függvény, amely meghatározza, hogy egy adott d távolságérték után, melyik távolsággal kell dolgoznunk.

módjával a későbbiekben foglalkozunk majd.) Ahogy már korábban említettük az algoritmusnak addig kell futnia, amíg a d értéke 0-ra nem csökken. Ennek érdekében a 3. és 14. sorok közötti ciklusban addig leszünk bent, amíg teljesül a $d \geq 1$ feltétel. A külső cikluson belül két dolgot teszünk. Egyrészt végre kell hajtanunk egy javított beillesztéses rendezést az egymástól d távolságra lévő elemekből álló résztömbökön. Ezt a feladatot valósítja meg az algoritmus 4. és 12. sorok közötti része. Ez az algoritmusrész majdnem teljesen azonos a javított beillesztéses rendezés 3.7. algoritmusával. Csak annyi módosítást eszközöltünk, hogy minden helyen ahol a szomszédos elemre történik utalás (pl. $i-1$ vagy $j+1$), ott a szomszédos elem miatt használt 1 értéket d-re változtattuk. Így tudjuk elérni, hogy az egymástól d távolságra lévő elemekkel foglalkozzunk. A külső cikluson belül még egy lépést kell megtennünk, a módosított javított beillesztéses rendezés végrehajtását követően a d értékét módosítanunk kell. Ezt a KövetkezőTávolság függvénnyel valósítjuk meg.

Az algoritmus megismerését követően nézzük meg, hogy milyen módon határozhatjuk meg a d távolságokat. Erre vonatkozóan öt lehetőséget mutatunk be.

1. Legyenek az alkalmazott d távolságok a $2^k - 1$ alakú számok, azaz az $\{1, 3, 7, 15, 31, 63, 127, \ldots\}$ halmaz elemei. A KezdetiTávolság függvény által megadott első d érték pedig a tömb n elemszámának feléhez legközelebbi $2^k - 1$ alakú szám.

2. A Fibonacci sorozat[3] elemeit használjuk távolságként. Tehát az alkalmazott értékek az $\{1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots\}$ halmaz elemei. A kiindulási d érték ekkor is legyen az $\frac{n}{2}$-hez legközelebbi Fibonacci szám.

3. Donald Shell javaslata, hogy a d értékek legyenek az

$$\left\lfloor \frac{n}{2^k} \right\rfloor$$

alakú számok, ahol $k = 1, 2, \ldots$, $\lfloor \cdot \rfloor$ pedig az alsó egészrészt jelöli. Ilyenkor a kezdeti d érték $\left\lfloor \frac{n}{2} \right\rfloor$ lesz.

4. Vaughan Pratt a $2^p \cdot 3^q$ alakban előállítható számok használatát javasolta, ahol p és q természetes számok. Ilyenkor a lehetséges d értékek a következők lehetnek: $\{1, 2, 3, 4, 6, 8, 9, 12, \ldots\}$. A kezdeti d érték ilyenkor a $2^p \cdot 3^q$ alakú számok közül az elemszám feléhez legközelebbi.

5. Donald Ervin Knuth a

$$\frac{(3^k - 1)}{2}$$

alakú számok használatát javasolta, ahol k pozitív egész számot jelöl, tehát így a d távolságértékek az $\{1, 4, 13, 40, 121, \ldots\}$ halmaz elemei. A kiindulási d ekkor is az n elemszám feléhez legközelebbi megengedett érték.

3.7. Példa. Feladatunk a 14 elemű $4, 9, 1, 10, 8, 6, 12, 2, 7, 3, 5, 14, 13, 11$ x tömb rendezése a Shell rendezés 3.8. algoritmusának használatával. Az algoritmus futását lépésenként végig követjük és közben a 3.9. ábrán láthatjuk, hogy miként alakul át a rendezendő tömb. Helytakarékossági célból a $d = 1$ esetet már nem tárgyaljuk részletesen, mivel ilyenkor a végrehajtandó lépések pontosan megegyeznek a javított beillesztéses rendezés 3.7. algoritmusával. A példában d értékei a $2^k - 1$ alakú számok közül kerülnek ki.

Az algoritmus 2. sorában meghatározzuk a d távolság kezdeti értékét, ami 7 lesz. Mivel $d \geq 1$, ezért belépünk a 3. sorban kezdődő külső ciklusba. A 4. sorban belépünk egy másik ciklusba, melynek i ciklusváltozója 8-tól 14-ig vesz fel értékeket. Az 5. sorban j az 1 értéket veszi fel, majd a *segéd* változóba eltároljuk az $x[8]$ értéket (6. sor). A 7. sorban kezdődő elöltesztelős ciklusba belépünk, mert teljesül a belépési feltétel. A 8. sorban a tömb első elemét átmásoljuk a nyolcadik helyre, majd a 9. sorban j értékét héttel csökkentjük, így értéke -6 lesz. A belső ciklusból kilépünk, mert j értéke negatív lett. A 11. sorban a *segéd*-ben eltárolt értéket bemásoljuk a tömb első helyére. Az eddigi lépéseket jelenítettük meg a 3.9a. ábrán. Látható, hogy az $x[1]$ és $x[8]$ elemekből álló résztömb rendezetté vált.

Ezt követően az i értéke 9-re növekszik, j értéke pedig 2 lesz. A *segéd* változóba kiírjuk az $x[9]$ értéket. Belépünk a belső ciklusba és $x[2]$-t átmásoljuk $x[9]$-be, majd j értékét -5-re csökkentjük. Mivel

[3] Ld. 4.3. fejezet

j negatív lett, ezért kilépünk a belső ciklusból, és $x[2]$-be másoljuk a *segéd*-ben eltárolt értéket. Így az $x[2]$ és $x[9]$ elemekből álló résztömb is rendezetté vált. A lépéseket a 3.9b. ábrán követhetjük nyomon.

Következik az $i = 10$ eset. A j értéke 3 lesz, a *segéd*-be kimentjük az $x[10]$ elemet. Mivel $x[3] < segéd$, ezért nem lépünk be a belső ciklusba. Az $x[10]$-be visszamásoljuk a *segéd* változóban eltárolt értéket. Most az $x[3]$ és $x[10]$ elemekből álló résztömb vált rendezetté (ld. 3.9c. ábra).

Az i változó értékét 11-re növeljük, j értékét 4-re állítjuk és a *segéd* változóba kiírjuk az $x[11]$-et. Mivel a belső ciklus belépési feltételei teljesülnek, ezért belépünk a ciklusba. A negyedik elemet átmásoljuk a tizenegyedik helyre, majd j-t héttel csökkentjük. Mivel j negatív lett, ezért nem maradunk bent a ciklusban, $x[4]$-be visszaírjuk a *segéd*-ben tárolt értéket. A 3.9d. ábrán is látható, hogy az $x[4]$ és $x[11]$ elemből álló résztömb lett rendezett.

Elérkezünk az $i = 12$ esethez. A j változó értéke 5 lesz, a *segéd*-be az $x[12]$-t írjuk ki. Mivel nem teljesül a belső ciklus belépési feltétele, ezért a ciklust átugorjuk, majd $x[12]$-be visszaírjuk a *segéd*-ben tárolt értéket. Az ötödik és tizenkettedik elemből álló résztömb rendezett (ld. 3.9e. ábra).

Az $i = 13$-nál, j kezdetben 6 lesz, a *segéd*-be pedig az $x[13]$ elemet másoljuk át. Mivel most sem teljesül a belső ciklus belépési feltétele, ezért a ciklust követő visszamásolást hajtjuk végre. A hatodik és tizenharmadik elemből álló résztömb is rendezett (ld. 3.9f. ábra).

Következik az $i = 14$ eset. Ekkor j először 7 lesz és a *segéd* változóba $x[14]$-et írjuk ki. Teljesül a belső ciklus belépési feltétele, így $x[14]$-be átmásoljuk $x[7]$-et, majd j értékét nullára csökkentjük. Ekkor már nem teljesül a belső ciklus bennmaradási feltétele, kilépünk ebből a ciklusból és $x[7]$-be írjuk vissza a *segéd*-ben eltárolt értéket. Így az $x[7]$ és $x[14]$ elemekből álló tömb is rendezetté vált (ld. 3.9g. ábra).

Mivel a 4. és 12. sorok közötti ciklus végére értünk, ezért meg kell határoznunk az d távolság soron következő értékét (ld. 13. sor). A következő $2^k - 1$ alakú szám a 3. Mivel d értéke még nem csökkent egy alá, ezért bennmaradunk a külső ciklusban.

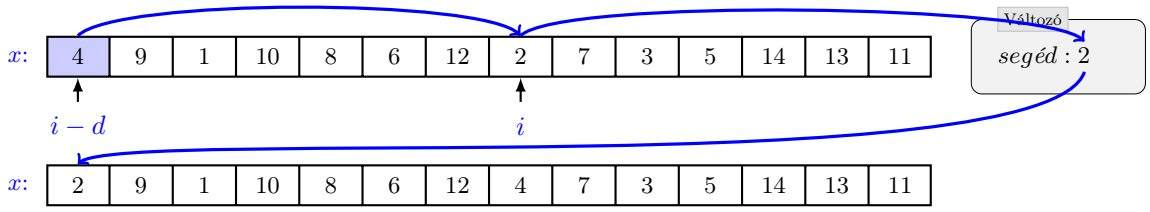
Ismét belépünk a 4. sorban kezdődő ciklusba, i kezdeti értéke 4 lesz. A j 1 lesz, *segéd*-be pedig az $x[4]$-et másoljuk. Mivel nem teljesül a belső ciklus belépési feltétele, ezért a vezérlés a ciklust követő sorra ugrik, ahol $x[4]$-be visszamásoljuk a *segéd* értékét. Az $x[1]$ és $x[4]$ elemekből álló résztömb rendezetté vált (ld. 3.9h. ábra).

A következőkben hasonlóan járunk el miközben az i értéke sorra 5-re, 6-ra, majd így tovább 14-re növekszik. A 3.9i-3.9r. ábrák szemléltetik az egyes fázisokat. A $d = 3$ esetet követően d-t egyre csökkentjük. Ezt az esetet már nem mutatjuk be részletesen, mivel megegyezik a javított beillesztéses rendezés konkrétan tárgyalt algoritmusával. Az eljárás végére a tömbünk teljesen rendezetté válik. ¶

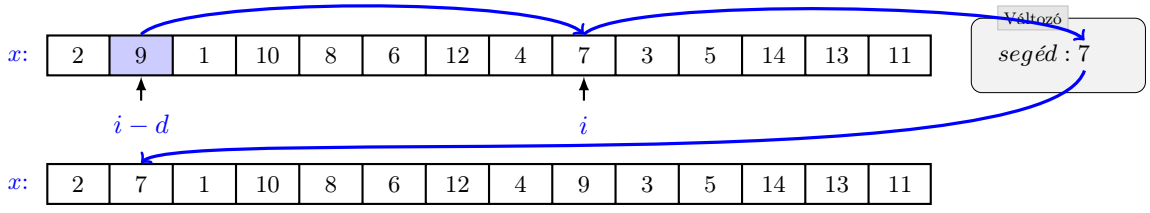
Futási idő elemzése. A futási idő vizsgálatánál a fő kérdés, hogy sikerült-e a javított beillesztéses rendezés futási idején csökkenteni. A Shell rendezés esetén nagyon bonyolult vizsgálni a konkrét összehasonlítási és másolási utasítások számát, ezért erre most nem is vállalkozunk. A futási idő nagy mértékben attól függ, hogy a d távolságértékeket milyen módszerrel határozzuk meg. Legrosszabb esetben bármely távolság esetén az állapítható meg, hogy az algoritmus futási ideje $O\left(n^2\right)$-es, ami sajnos nem jelent javulást.

Mi a helyzet az átlagos esettel? Ha az alkalmazott d távolságok a 2^k-1 alakú számok, vagy a Fibonacci sorozat elemei, illetve a Shell által javasolt $\left\lfloor \frac{n}{2^k} \right\rfloor$ értékek, akkor a futási idő még mindig $O\left(n^2\right)$-es lesz. Ha viszont a Pratt-féle $2^p \cdot 3^q$ alakú számok közül választjuk a megfelelő d értékeket, akkor a futási idő $O\left(n \cdot \log^2 n\right)$-esre csökkenthető. A Knuth által javasolt $\frac{3^k-1}{2}$ alakú számok használata mellett pedig $O\left(n^{\frac{3}{2}}\right)$-es lesz a futási idő.

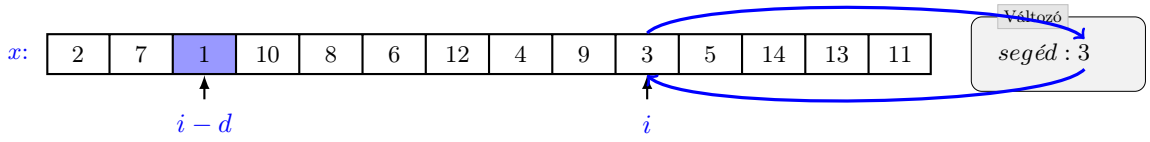
Annak érdekében, hogy jobban el tudjuk képzelni miként is alakul az algoritmus futási ideje az egyes esetekben, a 3.10. ábrán megjelenítettük az egyes futási időket különböző n értékek függvényében. Az ábrán az itt tárgyalt három eseten kívül megjelenítettük az $O\left(n \log n\right)$-es futási időt is, mert később látni fogjuk, hogy bizonyos rendező algoritmusok ilyen futási idővel képesek rendezni. ♣



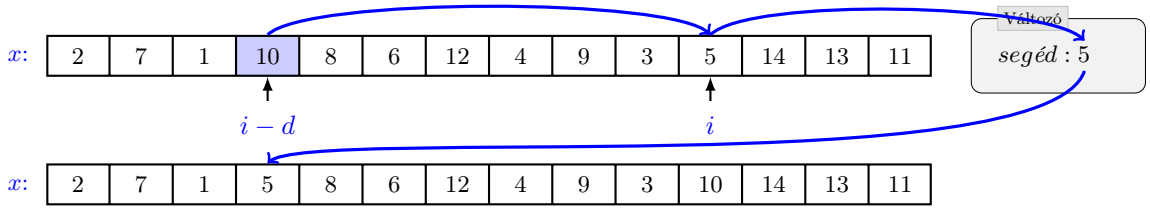
(a) A távolság: $d = 7$. A nyolcadik és az első elemből álló résztömböt rendezzük.



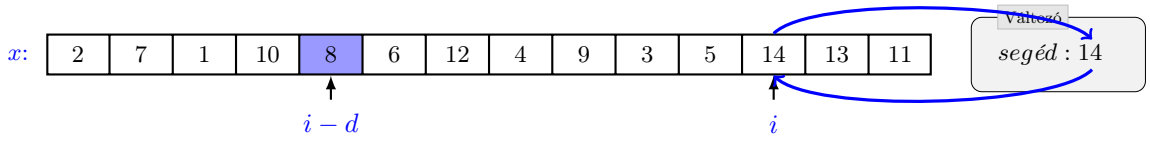
(b) A távolság: $d = 7$. A kilencedik és a második elemből álló résztömböt rendezzük.



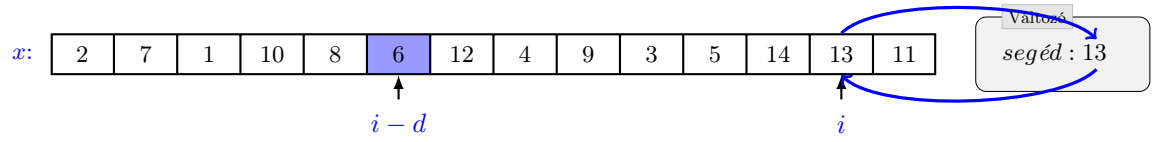
(c) A távolság: $d = 7$. A tizedik és a harmadik elemből álló résztömböt rendezzük.



(d) A távolság: $d = 7$. A tizenegyedik és a negyedik elemből álló résztömböt rendezzük.

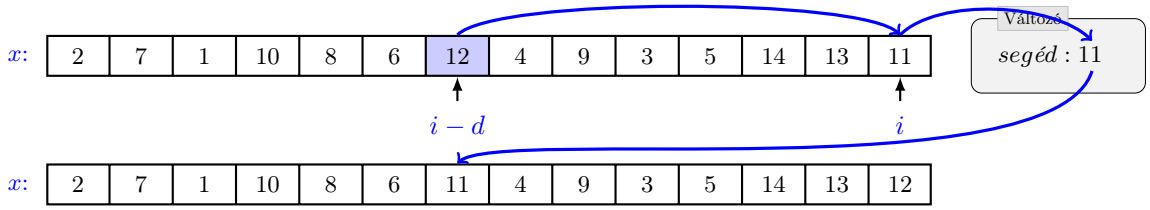


(e) A távolság: $d = 7$. A tizenkettedik és az ötödik elemből álló résztömböt rendezzük.

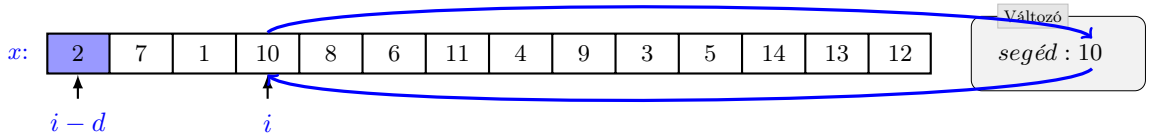


(f) A távolság: $d = 7$. A tizenharmadik és a hatodik elemből álló résztömböt rendezzük.

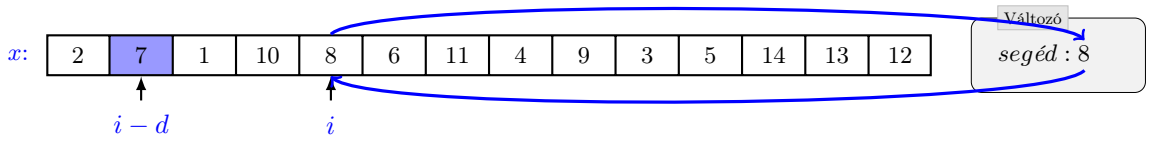
3.9. ábra. Shell rendezés.



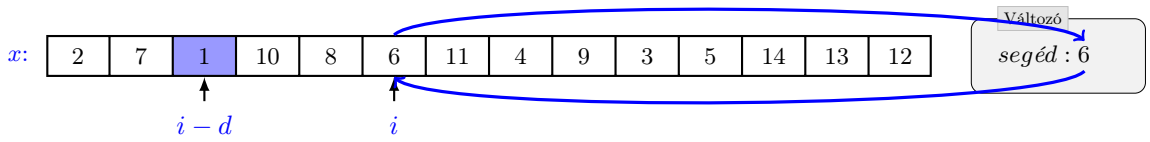
(g) A távolság: $d = 7$. A tizennegyedik és a hetedik elemből álló résztömböt rendezzük.



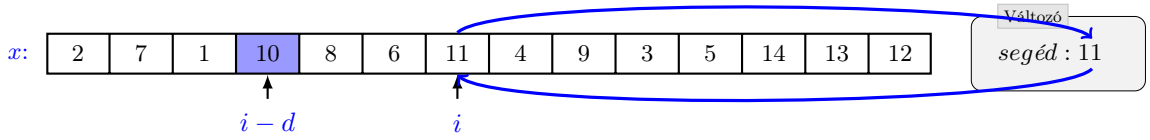
(h) A távolság: $d = 3$. A negyedik és az első elemből álló résztömböt rendezzük.



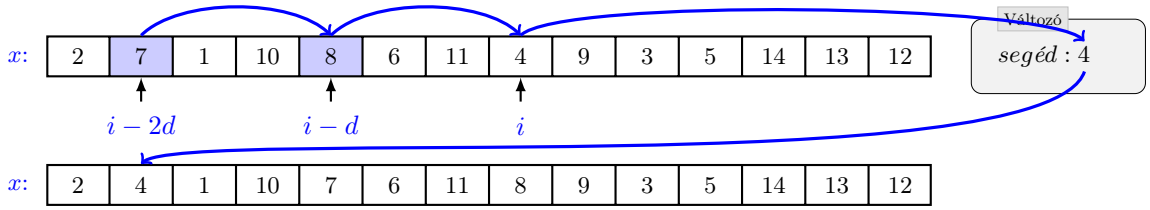
(i) A távolság: $d = 3$. Az ötödik és a második elemből álló résztömböt rendezzük.



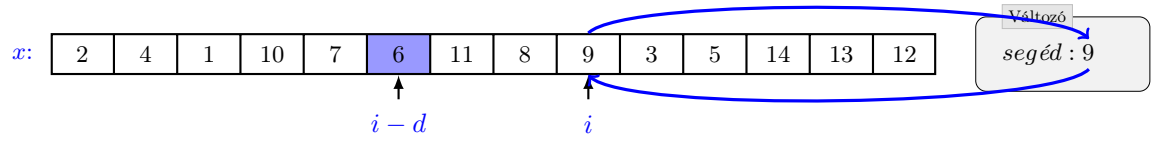
(j) A távolság: $d = 3$. A hatodik és a harmadik elemből álló résztömböt rendezzük.



(k) A távolság: $d = 3$. A hetedik, negyedik és első elemből álló résztömböt rendezzük.

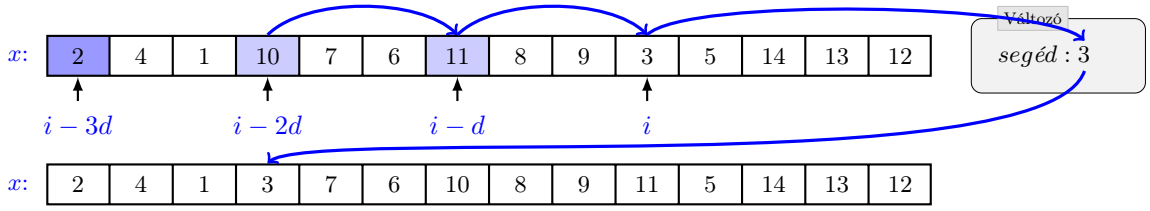


(l) A távolság: $d = 3$. A nyolcadik, ötödik és második elemből álló résztömböt rendezzük.

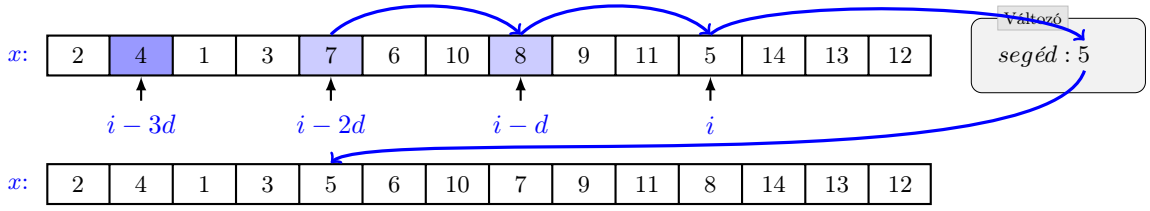


(m) A távolság: $d = 3$. A kilencedik, hatodik és harmadik elemből álló résztömböt rendezzük.

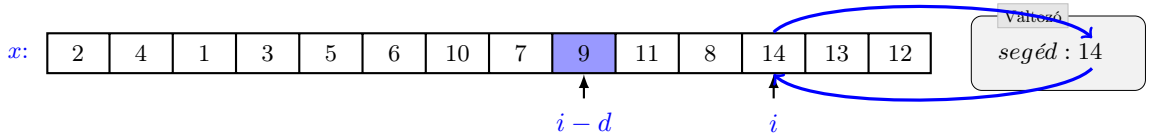
3.9. ábra. Shell rendezés (folyt.).



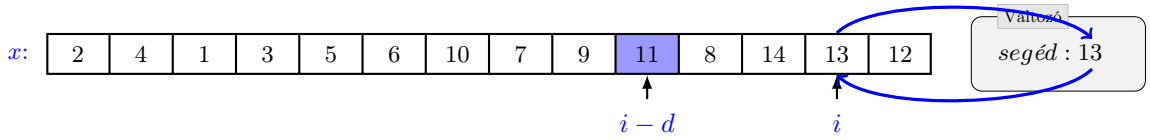
(n) A távolság: $d = 3$. A tizedik, hetedik, negyedik és első elemből álló résztömböt rendezzük.



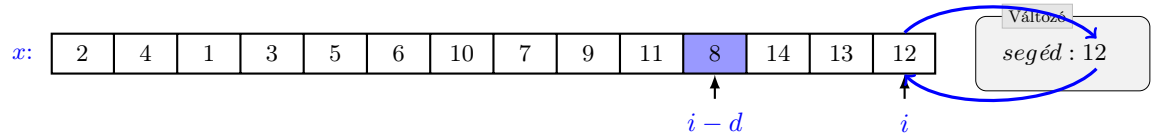
(o) A távolság: $d = 3$. A tizenegyedik, nyolcadik, ötödik és második elemből álló résztömböt rendezzük.



(p) A távolság: $d = 3$. A tizenkettedik, kilencedik, hatodik, és harmadik elemből álló résztömböt rendezzük.

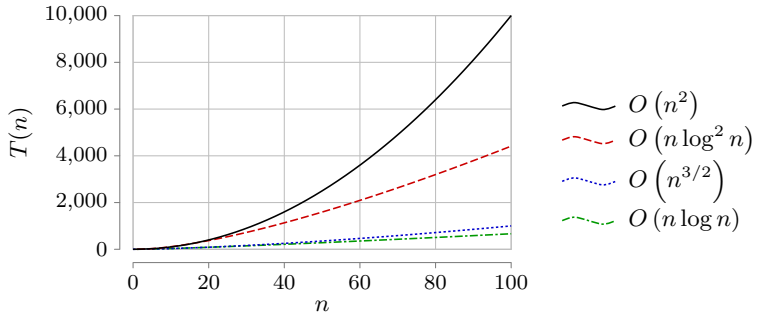


(q) A távolság: $d = 3$. A tizenharmadik, tizedik, hetedik, negyedik és első elemből álló résztömböt rendezzük.



(r) A távolság: $d = 3$. A tizennegyedik, tizenegyedik, nyolcadik, ötödik és második elemből álló résztömböt rendezzük.

3.9. ábra. Shell rendezés (folyt.).



3.10. ábra. Különböző futási idők összehasonlítása.

Szétosztó rendezés

Az eddigi rendezések használatának egyetlen feltétele volt, elvártuk, hogy a rendezendő tömb elemei összehasonlíthatóak legyenek. Ezen felül viszont más megkötésünk nem volt.

A szétosztó rendezés az előzőekkel szemben csak speciális esetben használható. A tömbnek két speciális feltételt is teljesítenie kell.

Első elvárásunk, hogy a tömbnek legyen egy *kulcs* mezője, aminek értéke pozitív egész szám lehet. Azon tömbök esetén, amik pozitív egész számokat tartalmaznak csak, maguk az egyes tömbelemek tekinthetők a kulcsnak. Viszont nem minden tömb ilyen: Ha a tömb elemei szövegek, akkor az egyes elemek nem tartalmaznak a megfogalmazott feltételnek megfelelő kulcsot.

Ha egy tömb elemei kulcsot tartalmaznak, akkor ezt a pszeudokódban külön meg is említjük, hasonlóan ahhoz, ahogy korábban az összehasonlíthatóságot kielmeltük. Az x tömb i-edik eleménél a kulcsra $x[i].kulcs$ módon tudunk hivatkozni.

A szétosztó rendezés alkalmazhatóságának másik feltétele, hogy n elemű tömb esetén a tömbben megtalálható kulcsok mind különbözőek legyenek, illetve 1 és n közé essenek.

Ha a fenti feltételeknek eleget tevő tömböt a kulcs szerint szeretnénk növekvő sorba rendezni, akkor használható a szétosztó rendezés. Az algoritmus előállít egy új kimeneti tömböt, amibe úgy helyei el az eredeti tömb elemeit, hogy azok a kulcsuk által indexelt helyre kerüljenek. A rendezés konkrét megvalósítását a 3.9. algoritmusban mutatjuk be.

3.9. Algoritmus Szétosztó rendezés

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** kulccsal rendelkezik
Kimenet: y – **T rendezett tömb**

```
1: függvény SzétosztóRendezés(x : T tömb, n : egész)
2:     y ← Létrehoz(T)[n]
3:     ciklus i ← 1-től n-ig
4:         y[x[i].kulcs] ← x[i]
5:     ciklus vége
6:     vissza y
7: függvény vége
```

Felhasznált változók és függvények

- x: A rendezendő tömb, melynek elemei kulccsal rendelkeznek. A kulcsok 1 és n közötti egészek. Minden kulcs pontosan egyszer fordul elő a tömbben.
- n: Az x tömb elemeinek száma és egyben a használt kulcsok maximális értéke.
- y: Kimeneti n elemű rendezett tömb.

Mivel a szétosztó rendezés nem helyben, a bemeneti x tömbben rendez, ezért első lépésként létre kell hoznunk a kimeneti y tömböt (ld. 2. sor). Ezt követően a 3. sorban kezdődő számlálós ciklussal végigmegyünk az x minden elemén. Az aktuális tömbelemet a kulcsának megfelelő helyre tesszük a kimeneti y tömbben (ld. 4. sor). Végső lépésként eredményként visszaadjuk a kulcsok szerint rendezett y tömböt (ld. 6. sor).

Futási idő elemzése. A szétosztó rendezés 3.9. algoritmusában összesen egy számlálós ciklus található, amely egyszer vesz figyelembe minden tömbelemet és összesen egy értékadás történik benne, ezért az algoritmus futási ideje: $T(n) = O(n)$. ♣

Számlálva szétosztó rendezés

A számlálva szétosztó rendezés akkor alkalmazható, ha a rendezendő tömbnek van kulcs mezője, viszont itt a kulcsoktól csak azt várjuk el, hogy pozitív egész értékek legyenek. A kulcsok között lehetnek azonosak is, illetve maximális értékük a tömb n elemszámától független m pozitív egész szám. Tehát ebben az esetben a kulcsok 1 és m közé eső számok, de konkrét eloszlásuk nem ismert.

Az algoritmusban először megszámoljuk, hogy melyik kulcs hányszor fordul elő, majd ennek figyelembe vételével osztjuk ki az elemeket a kimeneti y tömbbe úgy, hogy kulcs szerint növekvően rendezett sorrendben szerepeljenek. Ha egy kulcs többször is előfordul, akkor csak annyi az elvárás, hogy az azonos kulcsú elemek egymás mellé kerüljenek, de közöttük további rendezettségi szabályt már nem fogalmazunk meg.

A számlálva szétosztó rendezés megvalósítását a 3.10. algoritmusban adjuk meg. Bemenetként ismernünk kell az x tömböt, illetve az n elemszámot. Ezen túl a kulcsok maximális m értéke is bemeneti változóként jelenik meg. Kimenetként az n elemű y tömböt kapjuk meg.

3.10. Algoritmus Számlálva szétosztó rendezés

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), m – **egész**; ahol **T** kulccsal rendelkezik
Kimenet: y – **T rendezett tömb**

```
1:  függvény SzámlálvaSzétosztóRendezés(x : T tömb, n : egész, m : egész)
2:      db ← Létrehoz(egész)[m]
3:      ciklus i ← 1-től m-ig
4:          db[i] ← 0
5:      ciklus vége
6:      ciklus i ← 1-től n-ig
7:          db[x[i].kulcs] ← db[x[i].kulcs] + 1
8:      ciklus vége
9:      ciklus i ← 2-től m-ig
10:         db[i] ← db[i] + db[i − 1]
11:     ciklus vége
12:     y ← Létrehoz(T)[n]
13:     ciklus i ← 1-től n-ig
14:         y[db[x[i].kulcs]] ← x[i]
15:         db[x[i].kulcs] ← db[x[i].kulcs] − 1
16:     ciklus vége
17:     vissza y
18: függvény vége
```

Felhasznált változók és függvények

- x: A rendezendő tömb, melynek elemei kulccsal rendelkeznek. A kulcsok 1 és m közötti egészek, melyek akár ismétlődhetnek is.
- n: Az x tömb elemeinek száma.
- m: Az x tömbben található kulcsértékek lehetséges legnagyobb értéke.
- y: Rendezett kimeneti tömb, melynek elemszáma n.
- db: m elemű számláló tömb. A 6. sorban kezdődő ciklus lefutását követően az egyes kulcsok előfordulási számát adja meg az x tömbben.

A 3.10. algoritmus 2. sorában létrehozunk egy m elemű db számláló tömböt. Ennek a tömbnek minden elemét kezdetben nullára állítjuk a 3. sorban kezdődő ciklusban. A következő ciklusban (ld. 6. sor) megszámoljuk, hogy az egyes kulcsok hányszor fordulnak elő az x tömbben. A 9. sorban kezdődő ciklus logikailag az egyik legfontosabb része az algoritmusnak. A db tömb elemeit úgy módosítjuk, hogy $db[i]$ határozza meg hány i-nél nem nagyobb kulcsú elem található az x tömbben. Ez azért fontos, mert ha például egy x-beli elem kulcsánál tíz kisebb kulcsú elem van x-ben, akkor az adott elem legkorábban a tizenegyedik helyen állhat a kimeneti y tömbben.

A 12. sorban létrehozzuk az n elemű kimeneti y tömböt. Ezt követően már csak át kell másolni x-ből az elemeket y-ba a megfelelő helyre. Ezt valósítja meg a 13. sorban kezdődő ciklus. Behelyezzük az aktuális $x[i]$ elemet a végleges helyére, majd a db számláló tömb megfelelő elemét még csökkentjük is

eggyel, hiszen ha azonos kulcsú elem később kerül majd feldolgozásra, annak más helyre kerül kerülni y-ban, mint ahova az aktuális $x[i]$ került. Befejezésül visszaadjuk a kimeneti y tömböt (ld. 17. sor.

3.8. Példa.

A 3.11. ábrán nyomon követhető a számlálva szétosztó rendezés működése. Adott egy bemeneti x tömb, melynek elemei pozitív egész számok, így kulcsoknak is tekinthetők (ld. 3.11a. ábra). A 3.10. algoritmus először létrehoz egy *db* számláló tömböt, majd minden elemét nullaként inicializálja. Ezt követően a 6. sorban kezdődő ciklusban megszámoljuk, hogy melyik kulcs hányszor fordul elő x-ben. Ennek eredménye látható a 3.11b. ábrán.

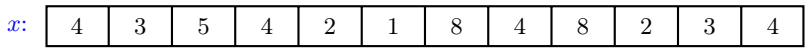
A 9. sorban kezdődő ciklusban meghatározzuk, hogy hány olyan kulcs van az x tömbben, amely egy i értéknél nem nagyobb. Ennek érdekében minden $db[i]$-be a nem nagyobb indexű elemek összege kerül. A módosított *db* tömb a 3.11c. ábrán látható.

Felkészültünk arra, hogy az x tömb elemeit a megfelelő helyre szétosszuk az y tömbbe. A 13. sorban kezdődő ciklus osztja szét az elemeket a megfelelő helyre úgy, hogy közben a *db* számláló tömböt is megfelelően módosítja. Az első iteráció során $x[i]$ értéke 4, ezért a negyedik tömbelemet vesszük figyelembe *db*-ből, $db[x[i]]$ pedig megadja, hogy $x[i]$-t az y kilencedik helyére kell kiosztani (ld. 3.11d. ábra). Ezt követően $db[x[i]]$ értékét még eggyel csökkentjük, hiszen ha lesz még olyan elem x-ben, melynek indexe szintén 4, annak majd y nyolcadik helyére kell kerülnie.

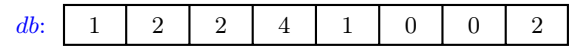
A további lépések nyomon követhetők a 3.11e. ábrától 3.11o. ábráig.

¶

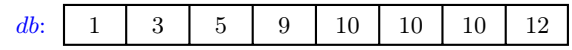
Futási idő elemzése. A 3.10. algoritmusban négy ciklus található. Az első m-szer, második n-szer, a harmadik $(m-1)$-szer, a negyedik pedig n-szer fut le. Minden cikluson belül csak konstans sok lépés van, így az algoritmus futási ideje, ami n-től és m-től is függ: $T(m+n) = O(m+n)$. ♣



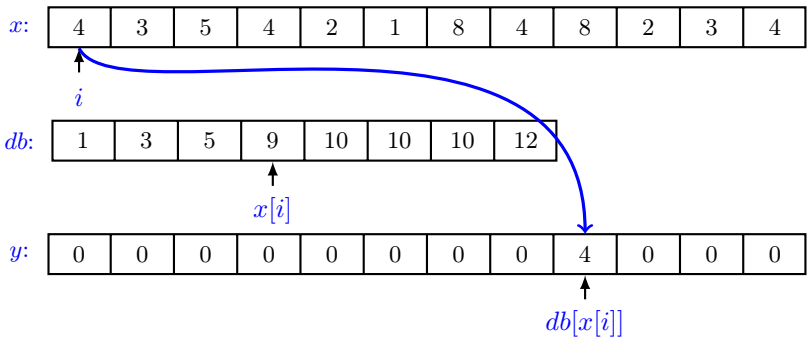
(a) A bemeneti x tömb.



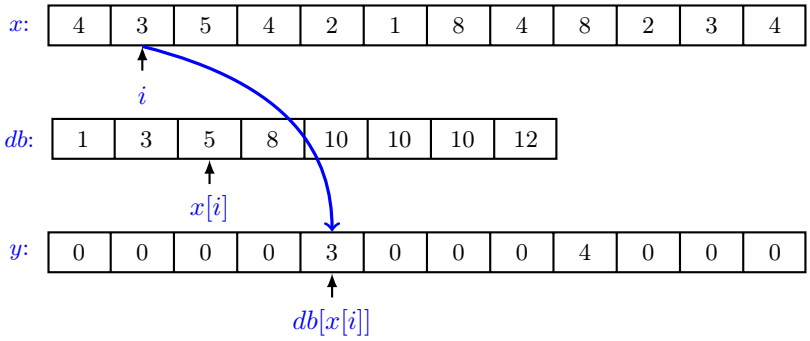
(b) Az egyes kulcsok darabszámát tartalmazó db tömb.



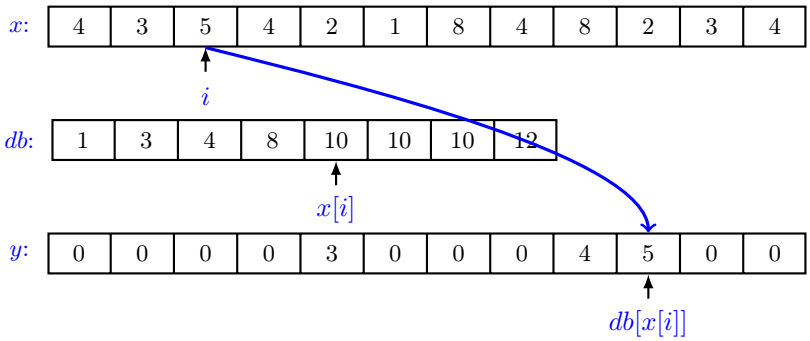
(c) Az egyes kulcsoknál nem nagyobb értékű kulcsok darabszámát tartalmazó módosított db tömb.



(d) $x[1]$ elhelyezése y-ba.

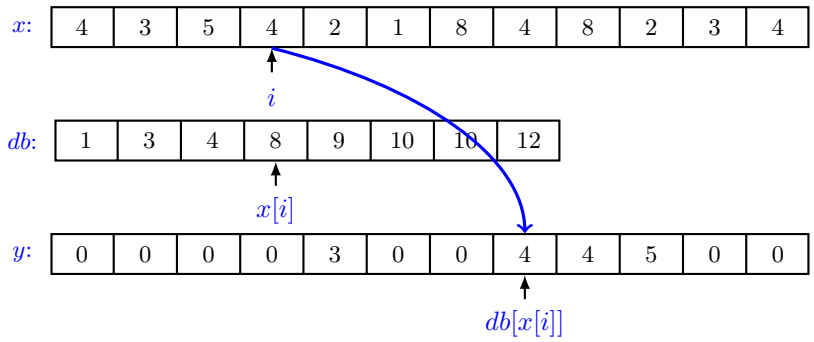


(e) $x[2]$ elhelyezése y-ba.

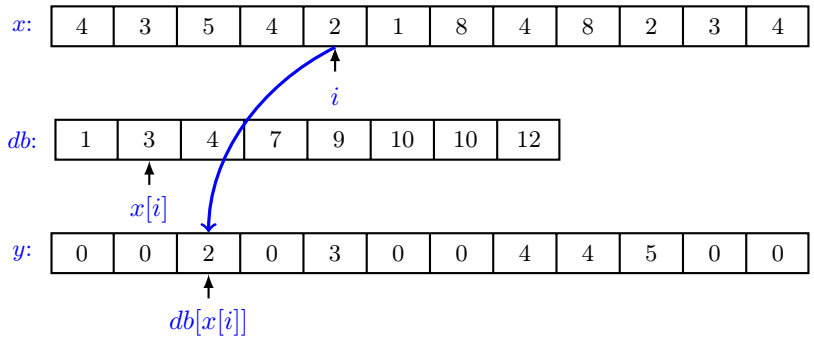


(f) $x[3]$ elhelyezése y-ba.

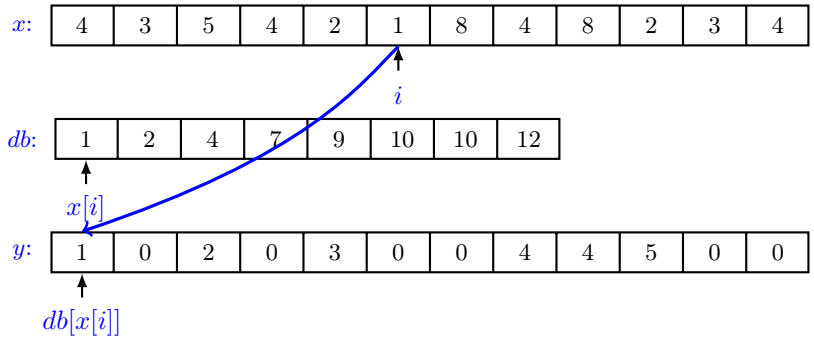
3.11. ábra. Számlálva szétosztó rendezés.



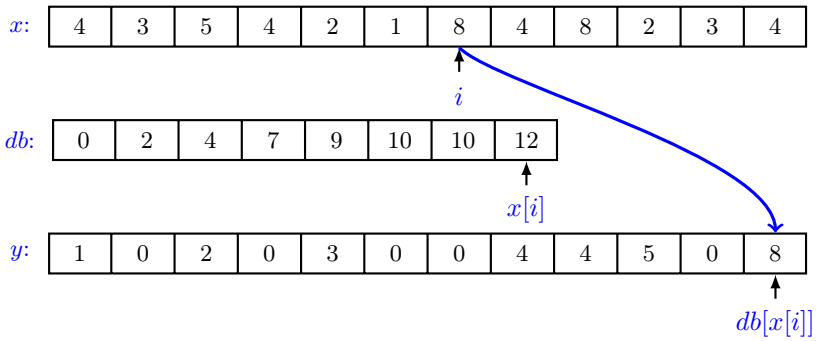
(g) $x[4]$ elhelyezése y-ba.



(h) $x[5]$ elhelyezése y-ba.

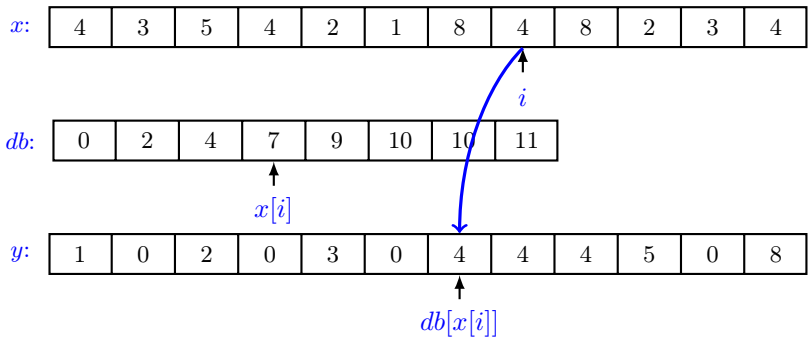


(i) $x[6]$ elhelyezése y-ba.

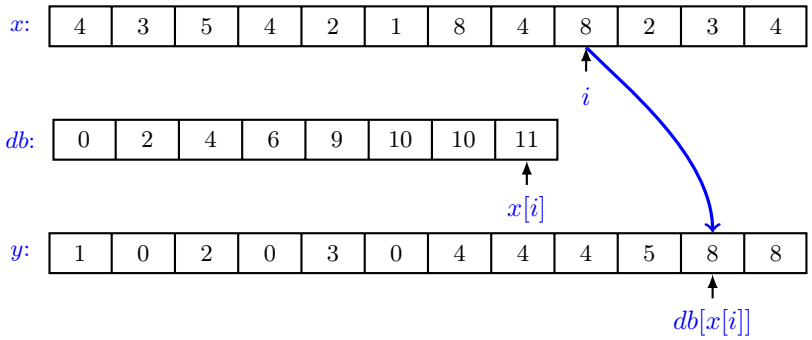


(j) $x[7]$ elhelyezése y-ba.

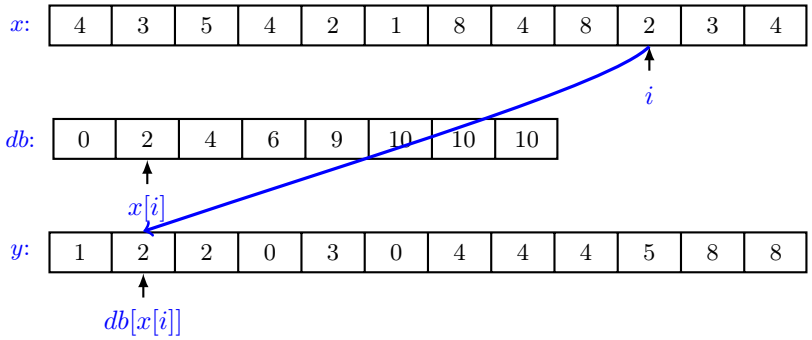
3.11. ábra. Számlálva szétosztó rendezés (folyt.).



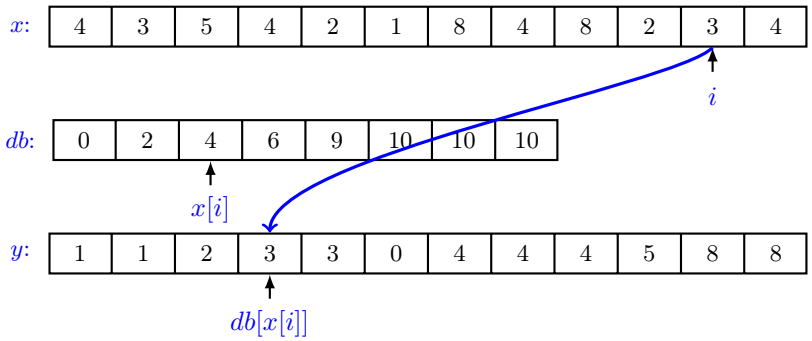
(k) $x[8]$ elhelyezése y-ba.



(l) $x[9]$ elhelyezése y-ba.

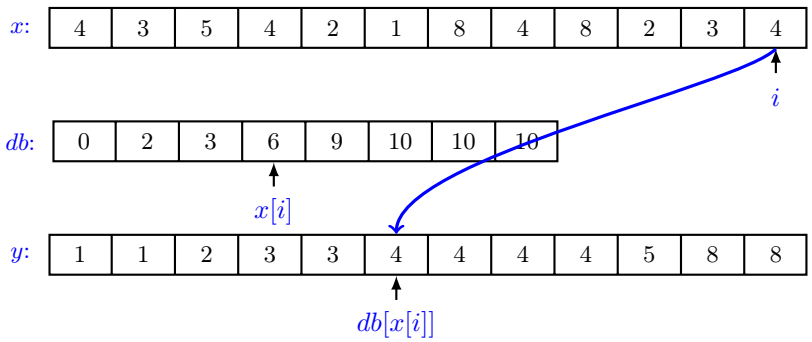


(m) $x[10]$ elhelyezése y-ba.



(n) $x[11]$ elhelyezése y-ba.

3.11. ábra. Számlálva szétosztó rendezés (folyt.).



(o) $x[12]$ elhelyezése y-ba.

3.11. ábra. Számlálva szétosztó rendezés (folyt.).

Számláló rendezés

A számláló rendezés a 3.1. alfejezetben megismert egyszerű cserés rendezés módosított változata. Az egyszerű cserés rendezésnél problémát okozott, hogy nagyon sok cserét végeztünk. A cseréket kiküszöbölhetjük, ha megszámoljuk, hogy melyik elemnél hány kisebb fordul elő. Így ugyanis meghatározhatjuk, hogy mi lesz az elem végleges helye a kimeneti rendezett tömbben. Így lényegében egy számlálást követően egy szétosztást kell megvalósítunk. Ebben az esetben persze kimenetként egy új tömb áll elő, hiszen a szétosztás során nem írhatjuk felül a korábbi tömbelemeket.

Amikor meg akarjuk határozni, hogy egy elem hova kerül majd a kimeneti rendezett tömbben, akkor arra az esetre is fel kell készülni, ha azonos elemek vannak a tömbben. Ezért nem csak azt fogjuk megszámolni, hogy egy adott elemnél hány kisebb van a tömbben, hanem azt is, hogy a vele egyezők közül hány előzi meg őt.

A számláló rendezés megvalósítását a 3.11. algoritmusban írjuk le. Bemenetként a rendezendő x tömböt és annak n elemszámát adjuk meg. Kimenetként egy rendezett y tömböt kapunk, mely szintén n elemű.

3.11. Algoritmus Számláló rendezés

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: y – **T rendezett tömb**

```
 1: függvény SzámlálóRendezés(x : T tömb, n : egész)
 2:     db ← Létrehoz(egész)[n]
 3:     ciklus i ← 1-től n-ig
 4:         db[i] ← 1
 5:     ciklus vége
 6:     ciklus i ← 1-től n − 1-ig
 7:         ciklus j ← i + 1-től n-ig
 8:             ha x[i] > x[j] akkor
 9:                 db[i] ← db[i] + 1
10:             különben
11:                 db[j] ← db[j] + 1
12:             elágazás vége
13:         ciklus vége
14:     ciklus vége
15:     y ← Létrehoz(T)[n]
16:     ciklus i ← 1-től n-ig
17:         y [db[i]] ← x[i]
18:     ciklus vége
19:     vissza y
20: függvény vége
```

Felhasznált változók és függvények

- x: Rendezendő tömb.
- n: Az x elemeinek száma.
- y: Kimeneti n elemű rendezett tömb.
- db: n elemű számláló tömb. A 6. sorban kezdődő ciklus lefutását követően i-edik eleme megadja, hogy hány $x[i]$-nél kisebb, illetve az i-edik eleme előttiek között hány $x[i]$-vel egyező elem van az x tömbben.

Az algoritmus 2. sorában létrehozzuk db számláló tömböt, majd minden elemét inicializáljuk 1 értékkel (ld. 3. sor). Azért egy elemek kezdeti értéke, mert a legkisebb elemnek majd az első helyre kell kerülnie a kimenetben. Ha nullától indexelnénk, akkor db kezdeti értékei nullák lennének.

A 6. sorban kezdődő egymásba ágyazott ciklusok szerkezete megfelel az egyszerű cserés rendezésnél megismertnek. A különbség, hogy itt azt számoljuk meg, hogy a nagyobbik elem előtt hány kisebb elem van, illetve az egyezők között hány korábban előforduló azonos értékű elem található. Könnyen belátható, hogy az előálló db tömbbeli értékek mind különböző 1 és n közé eső egész számok. Így már csak szét kell osztani az x-beli elemeket, ezt tesszük meg a 16. sorban kezdődő ciklusban.

Futási idő elemzése. A 3.11. algoritmus futása során minden x-beli elempárt összehasonlítunk, így $\frac{n\cdot(n-1)}{2}$ összehasonlítást végzünk el. Csere viszont nincs az algoritmusban, tehát a célunkat, hogy a cserék számát csökkentsük, sikerült elérni. Az algoritmus futási ideje: $T(n) = O\left(n^2\right)$. ♣

4. fejezet

Rekurzív algoritmusok

Az eddigiekben tárgyalt algoritmusok csak az 1. fejezetben bevezetett három vezérlési szerkezetet, a szekvenciát, a szelekciót (elágazás) és az iterációt (ciklus) használták. Ezzel a három vezérlési szerkezettel minden eddig problémát meg tudtunk oldani, sőt minden létező algoritmizálási probléma meg is oldható. Néha viszont a gondolkodásmódunkhoz közelebb áll egy negyedik vezérlési szerkezet, mely lényegében az iterációt helyettesítheti.

A most bevezetésre kerülő új vezérlési szerkezet a rekurzió. Ennek lényege, hogy egy függvény vagy eljárás önmagát hívja meg, így a benne lévő utasítások többször is végrehajtódnak.

Konkrét példákat a rekurzióra a fejezet további részében mutatunk. Először egy természetes szám faktoriálisa meghatározásának példáján (ld. 4.1. alfejezet) nézzük végig, hogy miként lehet ugyanazt a feladatot iteratív (ciklust használó) és rekurzív módon megoldani. A faktoriálisnál tapasztaltak alapján a 4.2. alfejezetben megfogalmazzuk a rekurzív algoritmusok általános jellemzőit. Ezt követően három példát veszünk sorra a rekurzió jobb megértése érdekében. Elsőként a Fibonacci sorozat elemeinek meghatározását tekintjük át (ld. 4.3. alfejezet), másodikként pedig egy szám pozitív egész kitevős hatványának kiszámítási módját mutatjuk be (ld. 4.4. alfejezet). Harmadik példánk a 4.5. alfejezetben egy játék, a Hanoi tornyai megoldását mutatja be.

A fejezet lezárásaként (ld. 4.6. alfejezet) sorra vesszük, hogy az egyszerű programozási tételek miként valósíthatók meg rekurzív módon, iteráció használata nélkül. A jegyzetben nem térünk ki annak tárgyalására, hogy miként lehet bármely iteratív algoritmust rekurzívvá, illetve rekurzív algoritmust iteratívvá alakítani. Ezzel általánosan az Algoritmusok, adatszerkezetek II. jegyzet foglalkozik.

Faktoriális számítás

Matematikából jól ismert, hogy az N természetes szám faktoriálisa alatt az első N darab pozitív egész szám szorzatát értjük, a nulla faktoriálisát pedig 1-ként értelmezzük. A faktoriális jelölésére a felkiáltójelet használjuk, tehát az N szám faktoriálisát $N!$-ként jelöljük. Egyetlen képletben összefoglalva az eddigieket:

$$N! = \begin{cases} 1, & \text{ha } N = 0 \text{ vagy } N = 1 \\ 1 \cdot 2 \cdot \ldots \cdot N, & \text{ha } N > 1 \text{ és } n \in \mathbb{N} \end{cases} \tag{4.1}$$

Ha algoritmust szeretnénk alkotni az $N!$ kiszámítására, akkor a sorozatszámítás tételhez hasonló algoritmussal (2.1. algoritmus) tudjuk kiszámítani a kívánt értéket. Lényeges különbség persze, hogy az elemek most nincsenek egy tömbben eltárolva. Tekinthetünk viszont úgy az első N darab pozitív egész számra, mint egy N elemű tömb elemeire. Így a faktoriálist a 4.1. algoritmussal határozhatjuk meg iteratív, azaz ciklust használó módon.

4.1. Algoritmus Faktoriális iteratív kiszámítása

Bemenet: N – **egész** (*természetes szám*)
Kimenet: *érték* – **egész**

```
1: függvény FaktoriálisIteratív(N : egész)
2:     érték ← 1
3:     ciklus i ← 2-től N-ig
4:         érték ← érték · i
5:     ciklus vége
6:     vissza érték
7: függvény vége
```

Felhasznált változók és függvények

- N: Természetes szám, melynek a faktoriálisát meg kívánjuk határozni.
- *érték*: Az algoritmus végén az $N!$ értékét tartalmazó változó.

Érdemes megemlíteni, hogy az algoritmus 3. sorában lévő számlálós ciklus i ciklusváltozójának kezdeti értéke 2. Így $N = 0$, illetve $N = 1$ esetben be sem lépünk a ciklusba, azaz az algoritmus a kezdeti 1 értéket fogja visszaadni.

Könnyen látható, hogy a 4.1. algoritmus futási ideje arányos az N értékével, hiszen a ciklusban lévő utasítást $(N - 1)$-szer hajtjuk végre.

A faktoriális iteratív tárgyalását követően térjünk rá a rekurzív elvű megvalósítás bemutatására. Nézzük meg először miként számíthatnánk ki például a 10 faktoriálisát, ha ismerjük már mondjuk a 9 faktoriálisát. Az iteratív megközelítést használva vennénk az első 10 darab pozitív egész szám szorzatát, és így megkapnánk a 10 faktoriálisát. Ehhez 9 darab szorzást kéne elvégeznünk. Ha viszont ismerjük a 9 faktoriálisát (mert korábban már kiszámítottuk), akkor a 10 faktoriálisát úgy is megkaphatjuk, ha a 9 faktoriálisát megszorozzuk 10-zel. Ez azért igaz, mert a 4.1. képlet alapján

$$9! = 1 \cdot 2 \cdot \ldots \cdot 9 \tag{4.2}$$

és

$$10! = 1 \cdot 2 \cdot \ldots \cdot 9 \cdot 10. \tag{4.3}$$

Ebből már látszik, hogy

$$10! = 10 \cdot 9!. \tag{4.4}$$

Általánosan azt mondhatjuk, hogy az N faktoriálisa kiszámítható úgy, hogy az $N - 1$ faktoriálisát megszorozzuk N-nel. Persze ha csak így definiálnánk a faktoriális fogalmát, akkor például a 2 faktoriálisának meghatározásánál problémába ütköznénk. A 2 faktoriális ugyanis $2 \cdot 1!$ lenne. Ahhoz, hogy ennek értékét meg tudjuk adni ismernünk kell az 1 faktoriálisát. Erről viszont csak annyit tudunk, hogy $1! = 1 \cdot 0!$. Most viszont a 0 faktoriálisát kéne ismernünk. Mondhatjuk erről is, hogy $0! = 1 \cdot (-1)!$-ral? Sajnos nem. Ezért szükséges, hogy például a 0 faktoriálisának konkrét értéket adjunk, ami az 1 lesz.

Mindezek alapján viszont az N természetes szám faktoriálisát az alábbi módon is definiálhatjuk:

$$N! = \begin{cases} 1, & \text{ha } N = 0 \\ N \cdot (N - 1)!, & \text{ha } N \geq 1 \end{cases} \tag{4.5}$$

Ez egy tipikus rekurzív definíció, mert a faktoriálist faktoriálissal adtuk meg. Tehát például a 4 faktoriálisát az alábbi módon tudjuk meghatározni:

$$4! = 4 \cdot 3! = 4 \cdot (3 \cdot 2!) = 4 \cdot (3 \cdot (2 \cdot 1!)) = 4 \cdot (3 \cdot (2 \cdot (1 \cdot 0!))) = 4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))) = 24. \tag{4.6}$$

Az eddigiek alapján már megadhatjuk a faktoriális kiszámításának 4.2. rekurzív algoritmusát, amit a 4.5. képlet alapján alkottunk meg.

4.2. Algoritmus Faktoriális rekurzív kiszámítása

Bemenet: N – **egész** (*természetes szám*)
Kimenet: N faktoriálisa

```
1: függvény FaktoriálisRekurzív(N : egész)
2:     ha N = 0 akkor
3:         vissza 1
4:     különben
5:         vissza N · FaktoriálisRekurzív(N − 1)
6:     elágazás vége
7: függvény vége
```

Felhasznált változók és függvények

- N: Természetes szám, melynek a faktoriálisát meg kívánjuk határozni.

Az algoritmus 2. sorában megvizsgáljuk, hogy az N értéke 0-e. Amennyiben 0, akkor a függvény visszatér az 1 értékkel (ld. 3. sor). Egyébként, tehát amikor $N \geq 1$ a függvény visszatérési értékét úgy határozzuk meg, hogy előtte meghívjuk a FaktoriálisRekurzív függvényt (ld. 5. sor). Itt történik meg ténylegesen a rekurzió, hiszen a FaktoriálisRekurzív függvény egyik saját utasítása meghívja azt a függvényt, ami éppen fut. Fontos viszont azt észrevenni, hogy a hívó függvény paramétere N, míg a hívott függvény paramétere $N - 1$.

4.1. Példa. Egy konkrét példán nézzük végig, miként működik a 4.2. algoritmus, mikor milyen paraméterrel hívjuk meg a FaktoriálisRekurzív függvényt. Az algoritmus futása a 4.1. ábrán is nyomon követhető.

Feladatunk a 4 faktoriálisának meghatározása. Ennek érdekében meghívjuk az algoritmust megvalósító függvényt, paraméterként a 4-et átadva annak: FaktoriálisRekurzív(4). Ezt a hívást szemlélteti a 4.1a. ábra, amelyen kék kitöltéssel jelezzük, hogy a hívott függvénynél van jelenleg a vezérlés. A függvényen belül kiértékelésre kerül az algoritmus 2. sorában található feltétel, ami ***hamis*** eredményt ad. A feltétel különben ágába lép a vezérlés, ahol meghívjuk a függvényt egy új paraméterrel: FaktoriálisRekurzív(3). Ekkor a vezérlés átadódik a hívott függvénynek, ahogy a 4.1b. ábrán is látható.

Az új függvényben ismét kiértékelésre kerül az $N = 0$ feltétel, ami ismét ***hamis*** lesz, ezért a különben ágba ugrik a vezérlés. Itt pedig újabb rekurzív függvényhívás történik: FaktoriálisRekurzív(2). A vezérlés átadódik a 2-vel hívott függvénynek (ld. 4.1c. ábra). Ebben a függvényben is kiértékeljük az $N = 0$ feltételt, ami még mindig ***hamis*** lesz. Ezért meghívjuk a függvényt egy kisebb paraméterrel: FaktoriálisRekurzív(1). A vezérlés átadódik a hívott függvénynek (ld. 4.1d. ábra). Ebben a függvényben is kiértékeljük a 2. sorban lévő feltételt, ami ismét ***hamis*** lesz. A különben ágban meghívjuk a FaktoriálisRekurzív függvényt a 0 paraméterrel, így a vezérlés átadódik a FaktoriálisRekurzív(0) függvénynek (ld. 4.1e. ábra).

Ebben a függvényben viszont az $N = 0$ feltétel ***igaz*** lesz, így a függvény visszaadja az 1 értéket az algoritmus 3. sorában lévő utasításnak megfelelően. Az érték visszaadásával együtt a FaktoriálisRekurzív(0) függvény futása be is fejeződik, a vezérlést visszakapja a FaktoriálisRekurzív(1) függvény (ld. 4.1f. ábra). A FaktoriálisRekurzív(1) függvényen belül a vezérlés abba a sorba tér vissza, ahonnan korábban meghívtuk a FaktoriálisRekurzív(0) függvényt. Ez a függvény 5. sora. Itt elvégezzük az $1 \cdot$ FaktoriálisRekurzív(0) műveletet, majd a függvény visszatér a kiszámított 1 értékkel és a FaktoriálisRekurzív(1) függvény futása is véget ér. Így a vezérlés visszakerül a FaktoriálisRekurzív(2) függvényhez (ld. 4.1g. ábra).

A FaktoriálisRekurzív(2) függvényen belül is elvégezzük az 5. sorban lévő szorzást, majd ez a függvény visszatér a 2 értékkel. A FaktoriálisRekurzív(2) függvény futása is véget ér, a vezérlés visszakerül a FaktoriálisRekurzív(3) függvényhez (ld. 4.1h. ábra). A FaktoriálisRekurzív(3) függvényben

is elvégezzük a szükséges szorzást, majd a függvény visszaadja a 6 értéket. A FaktoriálisRekurzív(3) futása is véget ér, a vezérlés pedig a FaktoriálisRekurzív(3) függvényhez kerül vissza (ld. 4.1i. ábra).

A FaktoriálisRekurzív(4) függvényben is az algoritmus 5. sorában folytatódik a végrehajtás. Elvégezzük a $4 \cdot$ FaktoriálisRekurzív(3) szorzást, majd a függvény visszatér a 24-gyel. Ezt követően véget ér a FaktoriálisRekurzív(4) függvény futása is (ld. 4.1j. ábra). Mivel ez volt az a függvény, amit a „külvilágból" hívtunk, így az algoritmus futása véget ért, sikeresen meghatároztuk a 4 faktoriálisát. ¶

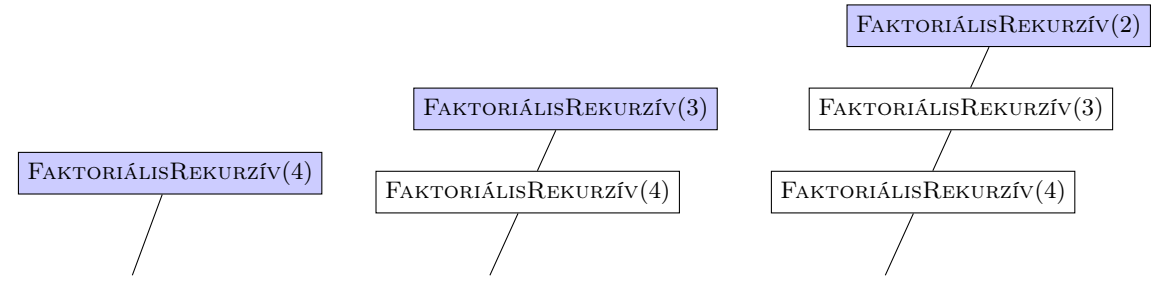
Megjegyzés

A rekurzió végrehajtása során láthatjuk, hogy a FaktoriálisRekurzív függvénynek egyszerre több „példánya" is fut, bár közülük csak egy aktív. Ez nem okoz problémát, mivel a rekurzív hívások során adminisztráljuk, hogy melyik pontról hívtuk meg az új függvényt, így a hívott függvény futása végén visszatérhet oda a vezérlés. Ez viszont idő- és memóriaigényes tevékenység.

Futási idő elemzése. Rekurzív algoritmusok futási idejének vizsgálatánál általában azt vizsgáljuk, hogy hány rekurzív hívás történt. Ez lényegében hasonló megközelítés, mit amit iteratív algoritmusoknál használtunk. Ott sokszor azt vizsgáltuk, hogy egy ciklus hányszor fut le. Mivel a rekurzív algoritmusoknál a ciklusokat helyettesítjük rekurzív hívásokkal, így értelemszerűen a rekurzív hívások számát érdemes vizsgálnunk.

Az N faktoriálisának meghatározásához $(N+1)$-szer hívjuk meg a FaktoriálisRekurzív függvényt. A külvilág persze csak egy hívást eszközöl, majd ezt követi még N darab rekurzív hívás.

Ezek alapján az algoritmus futási ideje $O(N)$-es. Ez azt jelenti, hogy nagyságrendileg ugyanolyan a futási ideje, mint a faktoriális számítás iteratív megvalósításának. Viszont a rekurzív függvényhívások megvalósítása idő- és memóriaigényes, ezért konkrét implementáció esetén azt tapasztaljuk, hogy az iteratív megvalósítás lényegesen gyorsabb futást eredményez. ♣

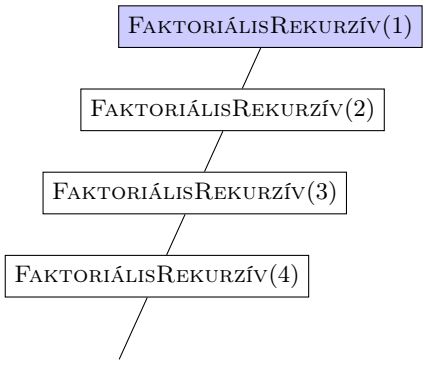


(a) A „külvilágból" meghívásra kerül a FaktoriálisRekurzív(4) függvény.

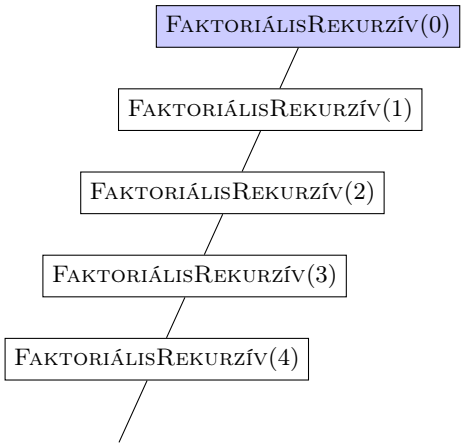
(b) A FaktoriálisRekurzív(4) függvény meghívja a FaktoriálisRekurzív(3) függvényt.

(c) A FaktoriálisRekurzív(3) függvény meghívja a FaktoriálisRekurzív(2) függvényt.

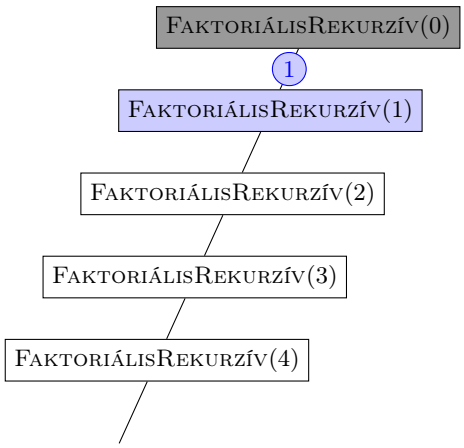
4.1. ábra. A 4 faktoriálisának rekurzív meghatározása.



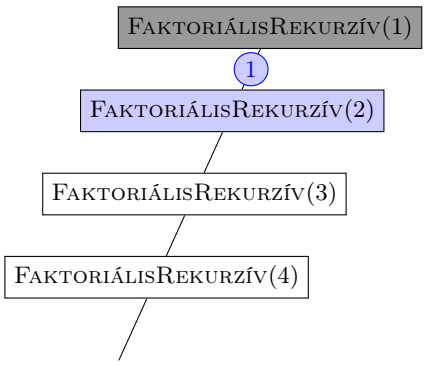
(d) A FaktoriálisRekurzív(2) függvény meghívja a FaktoriálisRekurzív(1) függvényt.



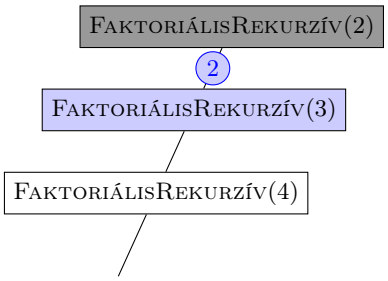
(e) A FaktoriálisRekurzív(1) függvény meghívja a FaktoriálisRekurzív(0) függvényt.



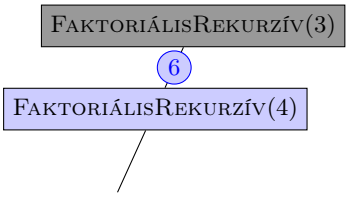
(f) A FaktoriálisRekurzív(0) függvény visszaadja az 1 értéket, majd futása véget ér. A vezérlés visszakerül a FaktoriálisRekurzív(1) függvényhez.



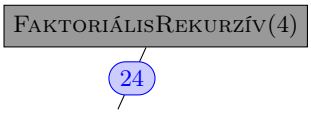
(g) A FaktoriálisRekurzív(1) függvény visszaadja az 1 értéket, majd futása véget ér. A vezérlés visszakerül a FaktoriálisRekurzív(2) függvényhez.



(h) A FaktoriálisRekurzív(2) függvény visszaadja az 2 értéket, majd futása véget ér. A vezérlés visszakerül a FaktoriálisRekurzív(3) függvényhez.



(i) A FaktoriálisRekurzív(3) függvény visszaadja az 6 értéket, majd futása véget ér. A vezérlés visszakerül a FaktoriálisRekurzív(4) függvényhez.



(j) A FaktoriálisRekurzív(4) függvény visszaadja az 24 értéket, majd futása véget ér. A feladat végére értünk.

4.1. ábra. A 4 faktoriálisának rekurzív meghatározása (folyt.).

Rekurzív algoritmusok jellemzői

Az előző alfejezetben a faktoriális számításának példáján láttuk, miként is működnek a rekurzív algoritmusok. Foglaljuk össze, hogy mik ezeknek az algoritmusoknak a legfontosabb jellemzői.

1. A rekurzív algoritmusokat olyan függvényekkel vagy eljárásokkal valósítjuk meg, melyek önmagukat hívják meg. Ez történhet közvetlenül, ahogy azt a faktoriális számításnál is láttuk. A FaktoriálisRekurzív függvény ugyanis önmagát hívta meg.

   > **Megjegyzés**
   >
   > Lehetséges viszont közvetett rekurzív hívás is. Ennek egy példája, ha van egy A függvényünk, amely meghívja a B függvényt, majd a B hívja az A-t. Ekkor az A nem közvetlenül hívja meg önmagát, hanem közvetetten a B-n keresztül.

2. Olyan függvények vagy eljárások lehetnek rekurzívak, melyekbe be van építve egy olyan eset, amikor már nem történik újabb rekurzív hívás. Ez biztosítja, hogy a rekurzív hívások egyszer véget érjenek, azaz ne alakuljon ki végtelen hívási lánc. A leállást biztosító esetet *alapeset*nek nevezzük.

   > **Megjegyzés**
   >
   > A FaktoriálisRekurzív függvénynél az alapeset az $N = 0$ paraméter esetén állt elő. Ekkor ugyanis a függvény nem hívta meg újra önmagát, hanem visszaadott egy konkrét értéket, mégpedig az 1-et.

3. A rekurzív hívások során a függvények vagy eljárások paraméterei folyamatosan változnak. Ha nem változnának, akkor végtelen hívási láncba kerülnénk. A paraméterek változásának olyannak kell lennie, hogy a hívások során közeledjünk az alapeset felé.

   > **Megjegyzés**
   >
   > A FaktoriálisRekurzív függvénynél a rekurzív hívások során a függvény bemeneti paramétere mindig eggyel csökkent, így közeledett az $N = 0$ alapesethez.

4. A rekurzív függvény egyszer kerül eltárolásra a memóriában. Az egyes függvényhívásokat a függvény paraméterei különböztetik meg egymástól. A paraméterek és a hozzájuk tartozó lokális változók viszont minden új függvényhíváskor eltárolásra kerülnek a memória ún. verem részében. A rekurzív hívásoknál a paraméterek és lokális változók eltárolása, azaz a hívások adminisztrálása idő- és memóriaigényes tevékenység. Emiatt általában igaz, hogy egy algoritmus rekurzív megvalósítása lassabb, mint a hasonló logikára épülő iteratív megvalósítás.

   > **Megjegyzés**
   >
   > Konkrét implementáció esetén gyakran találkozhatunk azzal, hogy elfogy az „adminisztráció” számára fenntartott memória hely, mert túl sok rekurzív hívás történt már. Ez a probléma ciklusok alkalmazásával nem szokott előállni.

A fentiek alapján gondolhatnánk, hogy felesleges rekurzív algoritmusokat használni, hiszen ciklusokkal általában hatékonyabb algoritmusokat hozhatunk létre. Ne felejtsük viszont el azt, hogy a rekurzív gondolkodás sok esetben sokkal közelebb áll a probléma egyszerű megoldásához, mint az iteratív megvalósítás.

Fibonacci sorozat

Következő példánk egy olyan algoritmus tárgyalása, mely rekurzív ötleten alapszik, de könnyen adható rá iteratív – sőt többszörös kiértékelést nem igénylő – megvalósítás is.

Fibonacci itáliai matematikus egyszer a nyulak szaporodását vizsgálta. Megállapította, hogy egy nyúlpár a születését követően kétszer szaporodik. Az első szaporodási ciklusban egy nyúlpárnak, majd a második ciklusban is egy nyúlpárnak ad életet. Felmerülhet a kérdés, hogy az N-edik szaporodási ciklusban hány nyúlpár fog ilyen módon megszületni.

Kiinduláskor, azaz a nulladik szaporodási ciklusban egy nyúlpárunk van, akiknek a következő két ciklusban születnek majd utódaik. Ha F_i-vel jelöljük az i-edik ciklusban született nyúlpárok számát, akkor F_0-t az elsőnek tekinthetjük.

Az első szaporodási ciklusban a kezdetben élő egyetlen nyúlpárunk fog szaporodni, mégpedig egy nyúlpárt hoznak a világra. Így $F_1 = 1$. A második szaporodási ciklusban még egyszer szaporulatot ad a kezdetben már létező nyúlpár, illetve szaporodik az előző ciklusban született nyúlpár is. Ezért $F_2 = F_0 + F_1$, azaz $F_2 = 2$. A harmadik ciklusban a kezdetben létező nyúlpár már nem fog újra szaporodni, viszont az első és második szaporodási ciklusban született párok igen. Így $F_3 = F_1 + F_2$, tehát $F_3 = 3$. Innentől kezdve ez így megy a végtelenségig, azaz $F_N = F_{N-2} + F_{N-1}$.

Összefoglalva Fibonacci az alábbi szabályszerűséget találta:

$$F_N = \begin{cases} 1, & \text{ha } N \leq 1 \\ F_{N-2} + F_{N-1}, & \text{ha } N \geq 2 \end{cases} \tag{4.7}$$

Tehát a Fibonacci sorozat nulladik és első eleme 1-gyel egyenlő, minden további eleme pedig a két megelőző elem összegeként áll elő.

A 4.7. képlet alapján egyszerűen megírhatjuk az N-edik Fibonacci szám kiszámításának rekurzív algoritmusát, ahogy ez a 4.3. algoritmusnál látható is.

4.3. Algoritmus Fibonacci sorozat N-edik elemének rekurzív meghatározása

Bemenet: N – **egész**
Kimenet: N-edik Fibonacci szám

```
1: függvény FibonacciRekurzív(N : egész)
2:     ha N ≤ 1 akkor
3:         vissza 1
4:     különben
5:         vissza FibonacciRekurzív(N − 2) + FibonacciRekurzív(N − 1)
6:     elágazás vége
7: függvény vége
```

Felhasznált változók és függvények

- N: Nemnegatív egész, mely megadja, hogy hányadik Fibonacci számot akarjuk meghatározni.

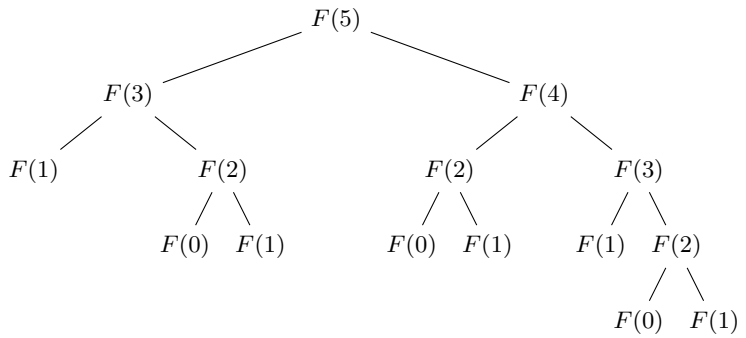
Az algoritmusban semmi más nem történik, mint ami a 4.7. képletben is látható. Megvizsgáljuk, hogy milyen az N értéke (ld. 2. sor). Amennyiben $N \leq 1$, akkor az alapeset áll fenn, tehát 1-et ad vissza a függvény (ld. 3. sor). Egyébként viszont az $(N-2)$-edik és $(N-1)$-edik Fibonacci számok összegét adja vissza a függvény (ld. 5. sor). A kisebb indexű Fibonacci számokat rekurzív hívásokkal kapjuk meg. Egy utasításon belül két rekurzív hívás is történik, de természetesen nem egy időben, hanem egymást követően. Hogy a FibonacciRekurzív$(N-2)$ vagy a FibonacciRekurzív$(N-1)$ függvényt hívjuk-e meg először, az attól függ, hogy a konkrét programozási környezet milyen sorrendben értékeli ki az összeadásban szereplő kifejezéseket. Amennyiben az összeadás kiértékelése balról jobbra történik, akkor először lefut a FibonacciRekurzív$(N-2)$ függvény, majd ezt követően fut a FibonacciRekurzív$(N-1)$ függvény.

4.2. Példa. Nézzük meg, hogy a 4.3. algoritmus használatával miként tudjuk meghatározni az ötödik Fibonacci számot. Ennek érdekében meghívjuk a FibonacciRekurzív függvényt, az $N = 5$ bemenettel. A FibonacciRekurzív(5) függvény futása során meghívja a FibonacciRekurzív(3) függvényt. A FibonacciRekurzív(3) függvény hívja a FibonacciRekurzív(1)-et, ami visszatér az 1 értékkel, majd a vezérlés visszakerül a FibonacciRekurzív(3) függvényhez. A FibonacciRekurzív(3) függvényből meghívásra kerül a FibonacciRekurzív(2) függvény is. A FibonacciRekurzív(2) meghívja

a FibonacciRekurzív(0) függvényt, amely visszaadja az 1 értéket, és a vezérlés visszakerül a FibonacciRekurzív(2) függvényhez. A FibonacciRekurzív(2) függvényből meghívásra kerül a FibonacciRekurzív(1) függvény is. A FibonacciRekurzív(1) függvény visszatér az 1 értékkel, majd a vezérlés ismét visszakerül a FibonacciRekurzív(2) függvényhez. A FibonacciRekurzív(2) függvényben kiszámításra kerül a visszatérési értéke, ami 2 lesz. A vezérlés visszakerül a FibonacciRekurzív(3) függvényhez. Itt is kiszámításra kerül a visszatérési érték, ami 3 lesz. Ekkor a vezérlés visszakerül a FibonacciRekurzív(5) függvényhez. Innen meghívásra kerül a FibonacciRekurzív(4) függvény. Ennek a függvénynek a kiértékelését már nem nézzük végig.

A függvények kiértékelési hierarchiája nyomon követhető a 4.2. ábrán. Az ábrán a FibonacciRekurzív függvényt helytakarékosság miatt csak F-fel jelöltük.

A példában jól látható, hogy az ötödik Fibonacci szám kiszámítása érdekében a negyedik Fibonacci számot egyszer, a harmadik Fibonacci számot kétszer, a második Fibonacci számot háromszor, az első Fibonacci számot ötször, a nulladik Fibonacci számot pedig háromszor kiszámítottuk. Az is felfedezhető, hogy egyszerre legfeljebb a FibonacciRekurzív függvény öt példánya futott, bár összesen tizenöt függvény hívás történt. ¶



4.2. ábra. Az ötödik Fibonacci szám kiszámításának hívási fája. Az F függvény jelöli a 4.3. algoritmus FibonacciRekurzív függvényét.

Futási idő elemzése. Felmerül bennünk a kérdés, hogy az N-edik Fibonacci szám kiszámítása érdekében hányszor kell meghívnunk a FibonacciRekurzív függvényt. Ennek vizsgálata érdekében jelöljük $T(N)$-nel az N-edik Fibonacci szám kiszámítása érdekében bekövetkező függvényhívások számát.

Könnyen belátható, hogy

$$T(0) = 1, \tag{4.8}$$

mivel $N = 0$ az egyik alapeset. Hasonlóan belátható, hogy

$$T(1) = 1. \tag{4.9}$$

A második Fibonacci szám kiszámítása során egyszer meg kell hívnunk a FibonacciRekurzív(2) függvényt, amely meghívja a FibonacciRekurzív(0) és a FibonacciRekurzív(1) függvényeket. Utóbbi kettő hívási számait már ismerjük, hiszen ezek voltak a $T(0)$-lal és $T(1)$-gyel jelölt értékek. Így

$$T(2) = 1 + T(0) + T(1) = 1 + 1 + 1 = 3. \tag{4.10}$$

A harmadik Fibonacci szám kiértékelése során egyszer meg kell hívnunk a FibonacciRekurzív(3) függvényt, amelyből meghívásra kerülnek a FibonacciRekurzív(1) és a FibonacciRekurzív(2) függvények. Emiatt

$$T(3) = 1 + T(1) + T(2) = 1 + 1 + 3 = 5. \tag{4.11}$$

A negyedik Fibonacci szám kiszámítása esetén hasonló gondolatmenettel adódik, hogy

$$T(4) = 1 + T(3) + T(2) = 1 + 5 + 3 = 9. \tag{4.12}$$

Mit mondhatunk általánosan? Amikor az N-edik Fibonacci számot akarjuk kiszámítani, akkor meghívjuk a FibonacciRekurzív(N) függvényt, amelyből meghívódnak a FibonacciRekurzív($N-2$) és a FibonacciRekurzív($N-1$) függvények. Így az általános hívásszámra az igaz, hogy

$$T(N) = 1 + T(N-2) + T(N-1). \tag{4.13}$$

Tehát a Fibonacci számok rekurzív kiszámításához szükséges függvényhívások száma egy rekurzív formulával adható meg, hasonlóan a Fibonacci számok definíciójához. A pontos formula:

$$T(N) = \begin{cases} 1, & \text{ha } N \leq 1 \\ 1 + T(N-2) + T(N-1), & \text{ha } N \geq 2 \end{cases} \tag{4.14}$$

Nem bizonyítjuk, de belátható, hogy a futási idő exponenciális. Ez azt jelenti, hogy

$$T(N) \sim a^N,$$

ahol a egy adott pozitív valós szám. Így találkoztunk az első olyan algoritmussal, amely futási ideje $O(a^N)$-es. ♣

Érdemes azzal is foglalkozni, hogy az N-edik Fibonacci szám kiszámítása iteratív módon miként lehetséges. Erre adunk megoldást a 4.4. algoritmusban. Az algoritmust részletesen nem mutatjuk be, mert az eddigi ismeretek alapján könnyen megérthető.

4.4. Algoritmus Fibonacci sorozat N-edik elemének iteratív meghatározása

Bemenet: N – **egész**
Kimenet: $aktuális$ – **egész**

```
1: függvény FibonacciIteratív(N : egész)
2:     aktuális ← 1
3:     előző ← 1
4:     ciklus i ← 1-től (N − 1)-ig
5:         átmeneti ← aktuális + előző
6:         előző ← aktuális
7:         aktuális ← átmeneti
8:     ciklus vége
9:     vissza aktuális
10: függvény vége
```

Felhasznált változók és függvények

- N: Megadja, hogy hányadik Fibonacci számot akarjuk meghatározni.
- $aktuális$: A függvény visszatérési értéke, melynek értéke az N-edik Fibonacci szám.

Az iteratív algoritmus futási idejét vizsgálva látjuk, hogy egyetlen ciklus szerepel benne, mely $(N-1)$-szer fut le. Így a futási idő ilyenkor $O(N)$-es, ami jóval gyorsabb, mint a rekurzív megvalósítás futási ideje. A rekurzív algoritmus megalkotása viszont egyszerűbb, mint az iteratív változaté, mivel maga a 4.7. képletben megadott definíció is rekurzív volt.

A 4.3. és a 4.4. algoritmusok megadták az N-edik Fibonacci szám értékét. Ha viszont az első N darab Fibonacci számra vagyunk kíváncsiak, akkor ezeket egy tömbbe kell kigyűjtenünk. Ennek iteratív megvalósítását láthatjuk a 4.5. algoritmusban.

4.5. Algoritmus Az első N darab Fibonacci szám megadása

Bemenet: N – **egész**
Kimenet: x – **egész tömb**

```
1: függvény FibonacciKigyűjt(N : egész)
2:     x ← Létrehoz(egész)[N]
3:     x[1] ← 1
4:     x[2] ← 2
5:     ciklus i ← 3-től N-ig
6:         x[i] ← x[i − 2] + x[i − 1]
7:     ciklus vége
8:     vissza x
9: függvény vége
```

Felhasznált változók és függvények

- N: Megadja, hogy az első hány darab Fibonacci számot határozzuk meg. Feltesszük, hogy $N \geq 2$.
- x: N elemű egészeket tartalmazó tömb, melynek i-edik eleme az i-edik Fibonacci szám.
- Létrehoz(**egész**)[N]: Utasítás, mely létrehoz egy N elemű **egész** típusú tömböt.

Hatványozás rekurzívan

Következő feladatunk egy pozitív valós szám (a) pozitív egész kitevős (N) hatványának meghatározása. Erre könnyen adhatunk iteratív megoldást a 2.1.1. fejezetben megismert sorozatszámítás programozási tétel használatával. A hatványozás iteratív megvalósítását a 4.6 algoritmusban láthatjuk.

4.6. Algoritmus a^N iteratív meghatározása

Bemenet: a – **szám**, N – **egész**
Kimenet: *érték* – **szám**

```
1: függvény HatványIteratív(a : szám, N : egész)
2:     érték ← a
3:     ciklus i ← 2-től N-ig
4:         érték ← érték · a
5:     ciklus vége
6:     vissza érték
7: függvény vége
```

Felhasznált változók és függvények

- a: Pozitív valós szám.
- N: Pozitív egész szám.
- *érték*: A függvény visszatérési értéke, mely az algoritmus végén a^N-nel egyenlő.

Futási idő elemzése. Látható, hogy az algoritmuson belüli ciklus $(N-1)$-szer fut le, így az algoritmus futási ideje $O(N)$-es. ♣

Nézzük meg, hogy miként tudunk rekurzív algoritmust adni a feladat megoldására. Tudjuk, hogy az a szám N-edik hatványát kiszámíthatjuk úgy, ha az $(N-1)$-edik hatványt megszorozzuk a-val, azaz

$$a^N = a^{N-1} \cdot a. \tag{4.15}$$

Ahhoz, hogy megvalósítható rekurzív algoritmust adjunk szükséges alapeset megadása is. Ez lehet például az

$$a^1 = a. \tag{4.16}$$

Ezek ismeretében már könnyen megadhatjuk a rekurzív megvalósítást (ld. 4.7. algoritmus).

4.7. Algoritmus a^N rekurzív meghatározása

Bemenet: a – **szám**, N – **egész**
Kimenet: a^N értéke

```
1: függvény HatványRekurzív(a : szám, N : egész)
2:     ha N = 1 akkor
3:         vissza a
4:     különben
5:         vissza a · HatványRekurzív(a, N − 1)
6:     elágazás vége
7: függvény vége
```

Felhasznált változók és függvények

- a: Pozitív valós szám.
- N: Pozitív egész szám.

Futási idő elemzése. Hányszor kell meghívni a HatványRekurzív függvényt az a szám N-edik hatványának kiszámítása érdekében? Könnyen belátható, hogy pontosan N darab függvényhívás szükséges ehhez. Tehát a rekurzív algoritmus futási ideje is $O(N)$-es. ♣

Nem lehetne-e valamilyen módon gyorsítani az algoritmus futási idejét? Egy ötletünk van, ami talán gyorsíthatja a futási időt. Tudjuk, hogy

$$a^k \cdot a^k = a^{2k}. \tag{4.17}$$

Ezt az összefüggést kihasználva hogyan számolnánk ki például egy szám tizenhatodik hatványát. Ha ismernénk a szám nyolcadik hatványát, akkor a nyolcadik hatvány négyzeteként már meg is kapnánk a tizenhatodik hatványt. A nyolcadik hatvány viszont megkapható a negyedik hatvány négyzeteként. A negyedik hatvány pedig a második hatvány négyzeteként. A második hatvány egyszerűen az a szám önmagával vett szorzataként adódik. Tehát azt látjuk, hogy

$$\begin{aligned} a^{16} &= a^8 \cdot a^8 = \\ &= \left(a^4 \cdot a^4\right) \cdot \left(a^4 \cdot a^4\right) = \\ &= \left(\left(a^2 \cdot a^2\right) \cdot \left(a^2 \cdot a^2\right)\right) \cdot \left(\left(a^2 \cdot a^2\right) \cdot \left(a^2 \cdot a^2\right)\right) = \\ &= (((a \cdot a) \cdot (a \cdot a)) \cdot ((a \cdot a) \cdot (a \cdot a))) \cdot (((a \cdot a) \cdot (a \cdot a)) \cdot ((a \cdot a) \cdot (a \cdot a))) . \end{aligned} \tag{4.18}$$

Hány hatványt kell így kiszámítani a tizenhatodik hatvány ismeretéhez. Tudni kell az első, a második, negyedik, a nyolcadik és a tizenhatodik hatványt. Ez összesen csak öt hatvány ismeretét kívánja, szemben a korábbi megközelítéssel, ahol minden hatványt ismerni kellett, azaz tizenhat hatvány ismeretét kívántuk meg.

Ezek szerint, ha van egy páros N számunk, akkor az N-edik hatvány meghatározható az $\frac{N}{2}$-edik hatvány ismeretében, hiszen

$$a^N = a^{\frac{N}{2}} \cdot a^{\frac{N}{2}}. \tag{4.19}$$

Mit tehetünk, akkor ha N páratlan szám. Mivel ilyenkor $N-1$ páratlan, ezért

$$a^N = a^1 \cdot a^{\frac{N-1}{2}} \cdot a^{\frac{N-1}{2}}. \tag{4.20}$$

Így a páros és a páratlan számokat is vissza tudjuk vezetni kisebb kitevős hatványokra. Már csak az szükséges, hogy legyen valamilyen alapesetünk, hogy a rekurzió leállását biztosítsuk. Tudjuk, hogy $a^1 = a$, ami biztosítja az alapesetet.

Összefoglalva az alábbi képletet adhatjuk:

$$a^N = \begin{cases} a, & \text{ha } N = 1, \\ a^{\frac{N}{2}} \cdot a^{\frac{N}{2}}, & \text{ha } N \text{ páros}, \\ a \cdot a^{\frac{N-1}{2}} \cdot a^{\frac{N-1}{2}}, & \text{ha } N > 1 \text{ és } N \text{ páratlan}. \end{cases} \tag{4.21}$$

4.8. Algoritmus a^N felezéses elvű rekurzív meghatározása

Bemenet: a – **szám**, N – **egész**
Kimenet: a^N értéke

```
1:  függvény HatványFelező(a : szám, N : egész)
2:      ha N = 1 akkor
3:          vissza a
4:      különben
5:          ha N páros akkor
6:              segéd ← HatványFelező (a, N/2)
7:              vissza segéd · segéd
8:          különben
9:              segéd ← HatványFelező (a, (N−1)/2)
10:             vissza a · segéd · segéd
11:         elágazás vége
12:     elágazás vége
13: függvény vége
```

Felhasznált változók és függvények

- a: Pozitív valós szám.
- N: Pozitív egész szám.

A 4.8. algoritmusban láthatjuk a 4.21. képletnek megfelelő megvalósítás pszeudokódját. Fontos megemlíteni, hogy az algoritmus 6. és 9. soraiban történik a rekurzív függvényhívás. A függvények visszatérési értékét egy *segéd* változóba mentjük, majd ennek a változónak képezzük a négyzetét. Ha ehelyett például

a **vissza** HatványFelező $\left(a, \frac{N}{2}\right) \cdot$ HatványFelező $\left(a, \frac{N}{2}\right)$ módon adnánk meg a visszatérési értéket, akkor $a^{\frac{N}{2}}$ kiszámítása érdekében kétszer is meghívnánk a függvényt, ami teljesen felesleges.

4.3. Példa. A 4.8. algoritmus használatával határozzuk meg a 2 huszonkettedik hatványát. Az algoritmus futását végigkövethetjük a 4.3. ábrán.

Először meghívjuk a HatványFelező függvényt az $a = 2$ és az $N = 22$ paraméterekkel (ld. 4.3a. ábra). Mivel N értéke még nem 1 és páros, ezért meghívódik a HatványFelező(2,11) függvény és a vezérlés is átkerül a hívott függvényhez (ld. 4.3b. ábra). Mivel $N = 11$ ezért meghívódik a HatványFelező(2,5) függvény (ld. 4.3c. ábra). A vezérlés átkerül a hívott függvényhez. Mivel most $N = 5$, ezért meghívódik a HatványFelező(2,2) függvény (ld. 4.3d. ábra). A vezérlés átkerül a hívott függvényhez. Mivel N páros, ezért meghívódik a HatványFelező(2,1) függvény (ld. 4.3e. ábra).

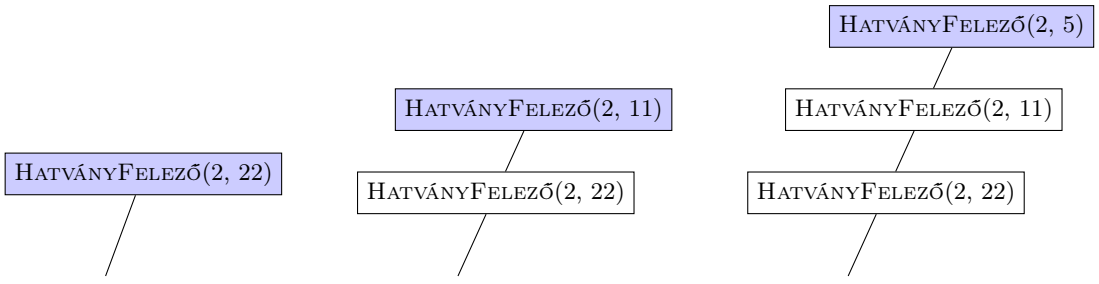
Mivel $N = 1$, ezért a HatványFelező(2,1) függvény visszatér az 1 értékkel. Ennek a függvénynek a futása véget ér, a vezérlés visszakerül a HatványFelező(2,2) függvény 6. sorába (ld. 4.3f. ábra). A HatványFelező(2,2) függvény futása is véget ér, visszaadja a $2 \cdot 2 = 4$ értéket. A vezérlés a HatványFelező(2,5) függvény 9. sorába kerül (ld. 4.3g. ábra). A HatványFelező(2,5) függvény visszatér a $2 \cdot 4 \cdot 4 = 32$ értékkel, a vezérlés a HatványFelező(2,11) függvény 9. sorába kerül (ld. 4.3h. ábra). A HatványFelező(2,11) függvény visszaadja a $2 \cdot 32 \cdot 32 = 2048$ értéket, futása véget ér, a vezérlés pedig a HatványFelező(2,22) függvény 6. sorába kerül (ld. 4.3i. ábra). Visszatértünk a kiinduló függvényhez. Ez a függvény visszaadja a négymillió-százkilencvennégyezer-háromszáznégy értéket, majd véget ér az algoritmusunk (ld. 4.3j. ábra). ¶

Futási idő elemzése. Vizsgáljuk meg, hogy hány függvényhívás szükséges az a^N hatvány felező módszerrel történő meghatározásához. Tudjuk, hogy $N \geq 1$ esetén a HatványFelező(a, N) függvény meghív egy másik függvényt, ahol N értéke $\frac{N}{2}$-re, vagy $\frac{N-1}{2}$-re csökken. Tehát minden egyes függvény egy olyan függvényt hív meg, ahol a változó paraméter felező̈dik vagy még az eredeti érték felénél is kisebbre csökken. A függvény hívások számát tehát meg tudjuk határozni, ha tudjuk, hogy az N értéket hányszor tudjuk úgy megfelezni, hogy értéke még 1-nél ne legyen kisebb. Matematikai tudásunkra építünk: a keresett szám pont az N 2-es alapú logaritmusa.

A lehetséges függvényhívások maximális száma így arányos $\log_2 N$-nel. Pontos felső korlátja:

$$\lceil 1 + \log_2 N \rceil , \tag{4.22}$$

ahol $\lceil \cdot \rceil$ a felső egészrészt jelöli. Így az algoritmusunk futási ideje $O(\log N)$-es, ami gyorsabb mint a korábbi két algoritmus $O(N)$-es futási ideje. ♣

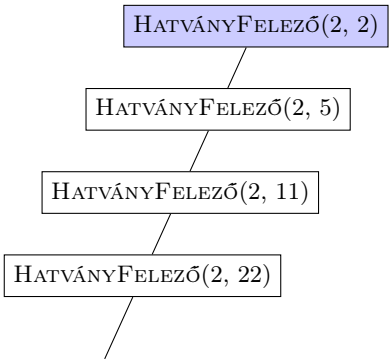


(a) A külvilágból meghívjuk a HatványFelező(2,22) függvényt.

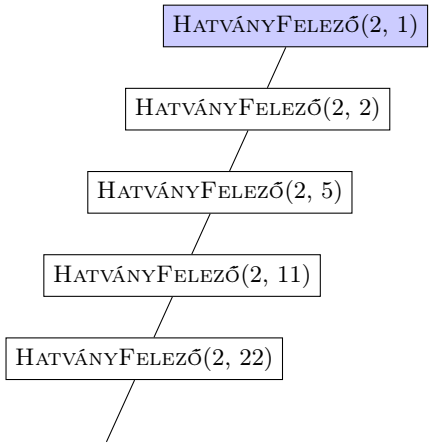
(b) A HatványFelező(2,22) függvény meghívja a HatványFelező(2,11) függvényt.

(c) A HatványFelező(2,11) függvény meghívja a HatványFelező(2,5) függvényt.

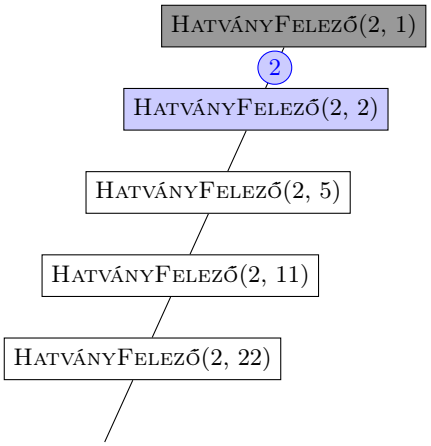
4.3. ábra. 2^{22} meghatározása a 4.8. algoritmus használatával.



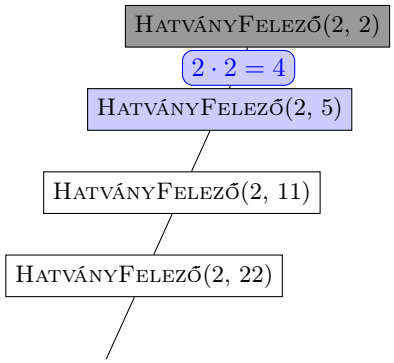
(d) A HatványFelező(2,5) függvény meghívja a HatványFelező(2,2) függvényt.



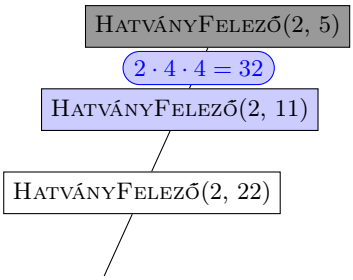
(e) A HatványFelező(2,2) függvény meghívja a HatványFelező(2,1) függvényt.



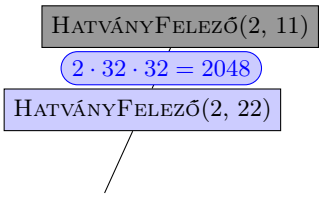
(f) A HatványFelező(2,1) függvény visszatér az 1 értékkel. A vezérlés visszakerül a HatványFelező(2,2) függvényhez.



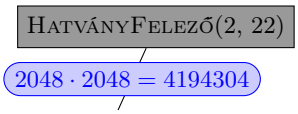
(g) A HatványFelező(2,2) függvény visszatér a 4 értékkel. A vezérlés visszakerül a HatványFelező(2,5) függvényhez.



(h) A HatványFelező(2,5) függvény visszatér a 32 értékkel. A vezérlés visszakerül a HatványFelező(2,11) függvényhez.



(i) A HatványFelező(2,11) függvény visszatér a 2048 értékkel. A vezérlés visszakerül a HatványFelező(2,22) függvényhez.

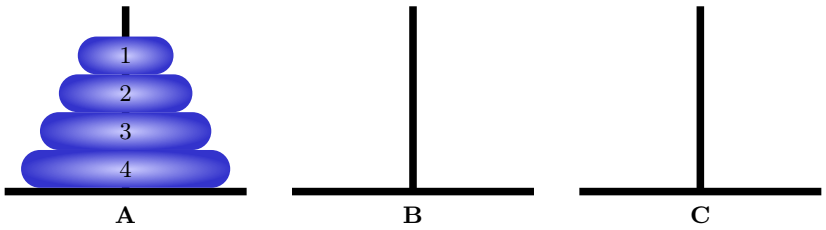


(j) A HatványFelező(2,22) függvény visszatér a 4194304 értékkel. A vezérlés visszakerül a HatványFelező(2,11) függvényhez.

4.3. ábra. 2^{22} meghatározása a 4.8. algoritmus használatával (folyt.).

Hanoi tornyai

A Hanoi tornyai egy játék, amelynek megoldására mutatunk be egy rekurzív elven működő algoritmust. A játék szabályai a következők. Van három rudunk, jelölje ezeket **A**, **B** és **C**, valamint N darab korongunk. A korongok mérete különböző. Kezdetben minden korong az **A** rúdon helyezkedik el úgy, hogy egy korong nem lehet nála kisebb korongon. Négy korong esetén a 4.4. ábra mutatja a kiindulási esetet. Feladatunk, hogy minden korongot áthelyezzünk az **A** oszlopról a **C** oszlopra. Az áthelyezésnek két fontos szabálya van. Az első, hogy csak a legfelső korongot vehetjük le valamely rúdról és a levett korongot át kell helyezni egy másik rúdra. Nem tehetünk olyat, hogy egy korongot bizonyos ideig nem rúdon tárolunk. A másik szabály, hogy egy korongot csak olyan rúdra tehetünk át, amely vagy üres, vagy a rajta lévő felső korong nagyobb, mint a ráhelyezendő korong.



4.4. ábra. Hanoi tornyai feladat alapállása 4 korong esetén. A korongokat az **A** oszlopról kell a **C** oszlopra áthelyezni.

A feladat megoldására olyan algoritmust kell találnunk, amely bármilyen pozitív egész N esetén véges sok lépésben megvalósítható. A rekurzív gondolkodás nagy segítségünkre lesz most.

Feladatunk, hogy N darab korongot a szabályoknak megfelelően áthelyezzünk az **A** rúdról a **C** rúdra úgy, hogy közben a **B** oszlopot is használhatjuk (ld. 4.4. ábra). Tegyük fel, hogy $N-1$ korongot már képesek vagyunk az egyik rúdról a másikra átrakni a szabályok szerint, tehát például 4 korong esetén el tudjuk érni, hogy 3 korong átkerüljön az **A** rúdról a **B** rúdra (ld. 4.5a. ábra). Ha ez megtörtént, akkor az **A** rúdon csak a legnagyobb korong maradt, amit áttehetünk az éppen üres **C** rúdra (ld. 4.5b. ábra). Ezután már csak a **B** rúdon lévő $N-1$ darab korongot kell a **C** rúdra mozgatnunk úgy, hogy közben az **A** rudat is használhatjuk. Ezt láthatjuk a 4.5c. ábrán $N=4$ esetén.

Az ötletünk lényege tehát, hogy N korong mozgatását visszavezetjük $N-1$ korong mozgatására. Ez tipikus rekurzív gondolat. Persze ahhoz, hogy a rekurzió működjön szükséges egy alapeset is. Ilyet is tudunk adni, hiszen ha $N=1$, akkor egyszerűen mozgatjuk a legkisebb korongunkat, amely mindig áthelyezhető bármely rúdról bármely más rúdra.

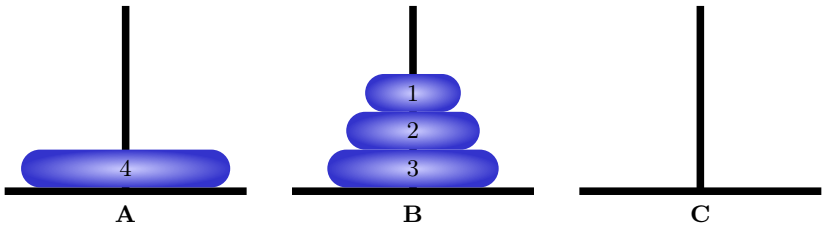
Az ismertetett rekurzív elvet a 4.9. algoritmusban írjuk le pszeudokód segítségével. A HANOI eljárásnak négy bemenete van. Az első paraméter (N) megadja, hogy hány darab korongot akarunk mozgatni. A második paraméter határozza meg azt a rudat, amelyen kezdetben rajta van az N darab korong. Erről a *forrás* rúdról kell átmozgatnunk a korongokat a harmadik paraméterként megadott *cél* rúdra. A negyedik paraméter azt a *segéd* rudat jelöli, amelyet a mozgatások közben segédrúdként használhatunk. Ha feladatunk N darab korong áthelyezése az **A** rúdról a **C** rúdra a **B** rúd segítségével, akkor az eljárást a HANOI(N, **A**, **C**, **B**) módon hívjuk meg.

Az algoritmusban először megvizsgáljuk, hogy milyen az N értéke. Ha $N=1$ (ld. 2. sor), akkor az alapesettel állunk szemben. Ilyenkor az egyes számú korongot kell a *forrás* rúdról a *cél* rúdra mozgatni. Ezt megvalósítja a megfelelően paraméterezett MOZGAT eljárás (ld. 3. sor).

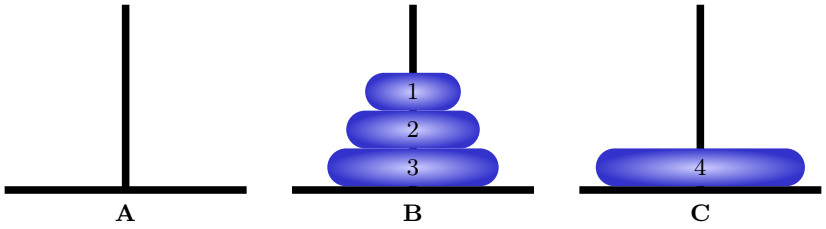
Amennyiben az N értéke nagyobb egynél, akkor rekurzívan meghívjuk az eljárást $N-1$ koronggal, amelyeket a *forrás*-ról a *segéd*-re kell áthelyezni a *cél* segítségével (ld. 5. sor). Amikor a rekurzívan hívott eljárás futása véget ér, akkor a felső $N-1$ korong átkerült a *forrás*-ról a *segéd*-re. A *forrás*-on csak az N-edik korong maradt, amit át tudunk egyszerűen helyezni az üres *cél* rúdra. Ezt valósítja meg a 6. sorban lévő MOZGAT eljárás. Ezt követően a *segéd*-en lévő $N-1$ darab korongot át kell mozgatni a *cél* rúdra, amit a HANOI eljárás megfelelően paraméterezett rekurzív hívásával érünk el (ld. 7. sor).

4.4. Példa. Vizsgáljuk meg, hogy az ismertetett algoritmus miként valósítja meg négy korong **A** rúdról **C** rúdra való mozgatását. A rekurzív eljárások hívási hierarchiáját a 4.6. ábrán láthatjuk.

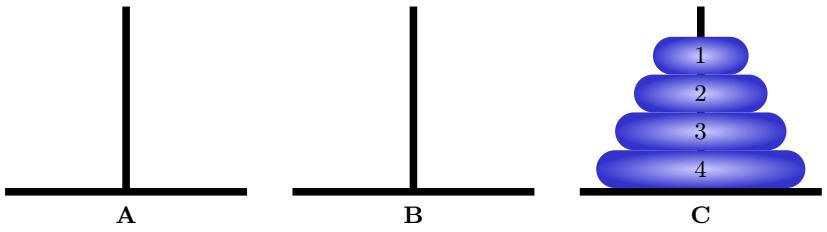
Először meghívjuk a HANOI(4,**A**,**C**,**B**) eljárást, mivel négy korongot akarunk az **A** rúdról a **C** rúdra mozgatni a **B** rúd segítségével. A HANOI(4,**A**,**C**,**B**) eljárás meghívja először a HANOI(3,**A**,**B**,**C**) eljárást,



(a) A három legkisebb korongot már átmozgattuk az **A** rúdról a **B** rúdra.



(b) A negyedik korongot áttesszük az **A** rúdról az üres **C** rúdra.



(c) A **B** rúdról a három korongot áttettük a **C** rúdra úgy, hogy a legnagyobb korong már a **C** rúdon volt.

4.5. ábra. Négy korong **A**-ról **C**-re való átmozgatásának ötlete.

4.9. Algoritmus Hanoi tornyai

Bemenet: N – **egész**, *forrás* – **rúd**, *cél* – **rúd**, *segéd* – **rúd**

```
1: eljárás HANOI(N : egész, forrás : rúd, cél : rúd, segéd : rúd)
2:     ha N = 1 akkor
3:         MOZGAT(1, forrás, cél)
4:     különben
5:         HANOI(N − 1, forrás, segéd, cél)
6:         MOZGAT(N, forrás, cél)
7:         HANOI(N − 1, segéd, cél, forrás)
8:     elágazás vége
9: eljárás vége
```

Eljárás hívása: HANOI(N, **A**, **C**, **B**)

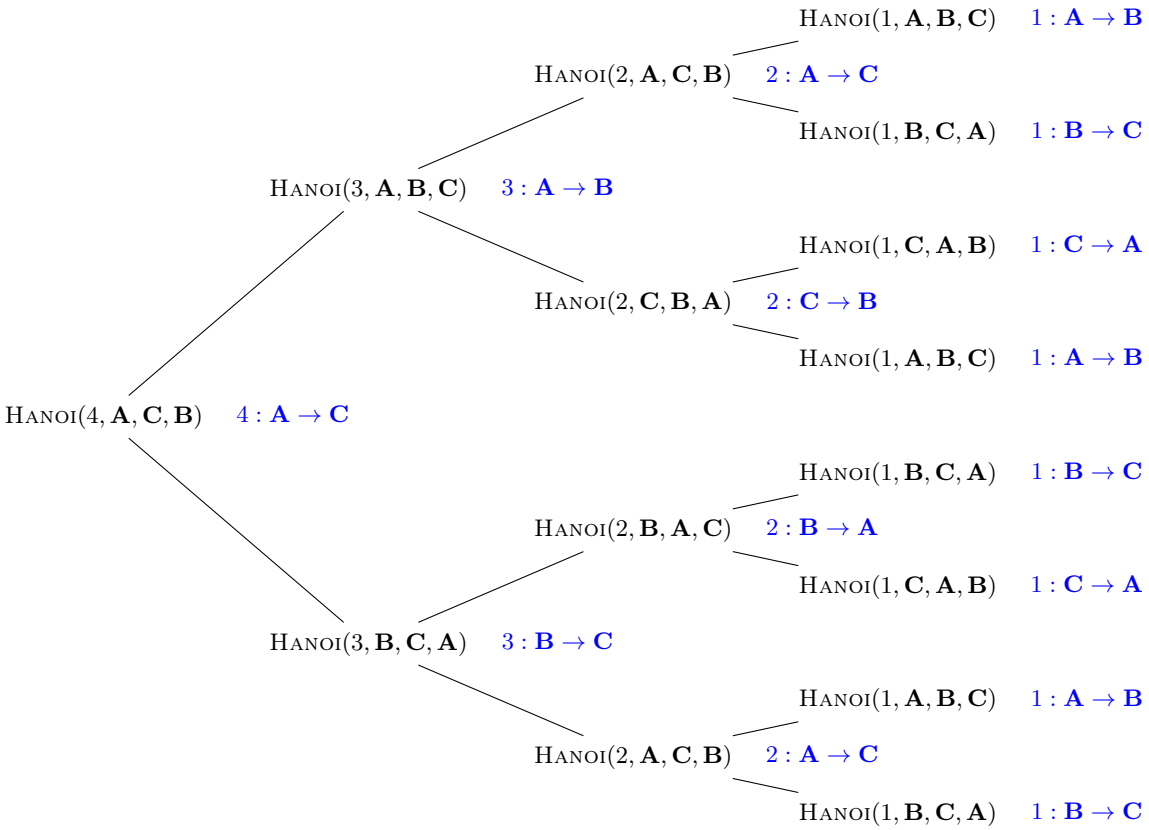
Felhasznált változók és függvények

- N: A mozgatandó korongok száma.
- *forrás*: Az a rúd, amelyről mozgatni akarunk.
- *cél*: Az a rúd, amelyre mozgatni akarunk.
- *cél*: Az a rúd, amelyet segítségül használhatunk a mozgatáshoz.
- MOZGAT(n, *forrás*, *cél*): Eljárás, amely megvalósítja az n-edik korong mozgatását a *forrás* rúdról a *cél* rúdra.

---

mivel elsőként meg kell oldani, hogy három korong átkerüljön az **A**-ról a **B**-re **C**-t segédrúdként használva. A HANOI(3,**A**,**B**,**C**) meghívja a HANOI(2,**A**,**C**,**B**)-t, amely meghívja a HANOI(1,**A**,**B**,**C**)-t. Az utolsóként hívott eljárásban egy egyszerű mozgatás történik csak, az első korong átkerül az **A** rúdról a **B** rúdra. Ezután a vezérlés visszakerül a HANOI(2,**A**,**C**,**B**) eljáráshoz, ahol végrehajtódik az algoritmus 6. sora. Tehát a második korong az **A** rúdról a **C** rúdra kerül. Ezt követően meghívódik a HANOI(1,**B**,**C**,**A**) eljárás, melyen belül az egyes korong átkerül **B**-ről **C**-re.

A teljes futás nyomon követhető a 4.6. ábrán és egyben fentről lefelé nézve látható, hogy milyen mozgatások sorozata szükséges a négy korong áthelyezéséhez. ¶



4.6. ábra. Négy korongos Hanoi algoritmus hívási fája, valamint az egyes eljárásokon belül végrehajtott mozgatások.

Futási idő elemzése. A futási idő vizsgálatánál azt kell megnéznünk, hogy N darab korong egyik rúdról egy másikra történő áthelyezéséhez hány darab eljáráshívásra van szükségünk. Mivel minden egyes eljáráson belül pontosan egy korongmozgatás történik, ezért az eljáráshívások száma pontosan megegyezik a szükséges mozgatások számával is. Az N korong mozgatásához szükséges eljáráshívások számát jelöljük $T(N)$-nel.

Vizsgáljuk meg először, hogy egy korong esetén hány darab eljáráshívásra van szükség. Meghívjuk a HANOI eljárást az $N = 1$ értékkel. Mivel ilyenkor nem történik rekurzív hívás, ezért egy hívás történik csak. Így

$$T(1) = 1. \tag{4.23}$$

Ha $N = 2$, akkor egyszer meghívjuk a HANOI eljárást $N = 2$-vel, amely kétszer meghívja a HANOI-t $N = 1$ értékkel. Így

$$T(2) = 1 + 2 \cdot T(1) = 3. \tag{4.24}$$

$N = 3$ esetén is hasonló a helyzet:

$$T(3) = 1 + 2 \cdot T(2) = 7. \tag{4.25}$$

Mit mondhatunk általánosan? Amikor a HANOI eljárást N-nel hívjuk, akkor ez egy közvetlen hívást jelent. Az eljáráson belül viszont kétszer is megívásra kerül a HANOI eljárás az $N-1$ értékkel, ezért

$$T(N) = 1 + 2 \cdot T(N-1). \tag{4.26}$$

Teljes indukcióval belátható – bár most ezt nem tesszük meg –, hogy általánosan

$$T(N) = 2^N - 1. \tag{4.27}$$

Ezek szerint a Hanoi tornyai feladatot csak $O(2^N)$-es futási idővel tudjuk megoldani. ♣

Egyszerű programozási tételek rekurzív megvalósítása

Sorozatszámítás

A 2.1.1. fejezetben már tárgyaltuk a sorozatszámítás programozási tételt, de ott iteratív megoldást mutattunk be. Ebben a fejezetben pedig a sorozatszámítás feladatának rekurzív megvalósítását mutatjuk be.

Sorozatszámításnál a feladatunk egy x tömb összes eleme között elvégezni egy megadott $\oplus$ műveletet. Az algoritmus eredményül a kiszámított műveleti értéket adja meg. Példaként általában az összegzést szokás említeni, ami a tömb elemeinek összegét határozza meg.

A rekurzív megoldásunk lényege az lesz, hogy amikor az első $jobb$ darab elem közötti műveleti eredményt akarjuk kiszámítani, akkor feltesszük, hogy az első $jobb-1$ darab elem között már ismerjük az $\oplus$ művelet eredményét, így csak ezen érték és a tömb $jobb$ indexű eleme között kell meghatározni az $\oplus$ művelet eredményét. Lényegében az $\oplus$ művelet asszociatív tulajdonságát használjuk, hiszen:

$$x[1]+x[2]+\ldots+x[jobb-1]+x[jobb]=(x[1]+x[2]+\ldots+x[jobb-1])+x[jobb]. \tag{4.28}$$

Hogyan fog leállni a rekurzív hívások láncolata? Tudjuk, hogy az $\oplus$ műveletnek van valamilyen kiindulási értéke ($\mathit{érték}_0$). Amikor az „üres” tömbbel hívjuk majd meg a rekurzív függvényünket, akkor ezt az értéket kell visszaadnia az algoritmusnak.

Az előző ötletre épülő algoritmus pszeudokódját a 4.10. algoritmusban mutatjuk be. Az algoritmust megvalósító függvénynek két bemenete van: a feldolgozandó x tömb, illetve a $jobb$ index. A $jobb$ értékével azt határozzuk meg, hogy éppen az első hány darab elem között akarjuk az $\oplus$ műveletet elvégezni. Amikor a „külvilágból” meghívjuk a rekurzív függvényt, akkor $jobb$ értéke a tömb n mérete lesz, hiszen az egész tömbre el akarjuk végezni a kívánt műveletet. A függvény visszatérési értéke a művelet eredménye lesz.

4.10. Algoritmus Sorozatszámítás programozási tétel rekurzív megvalósítása

Bemenet: x – **T tömb**, $jobb$ – **egész**
Kimenet: A vizsgált résztömb elemeire nézve a $\oplus$ művelet eredménye.

```
1: függvény SorozatszámításRekurzív(x : T tömb, jobb : egész)
2:     ha jobb = 0 akkor
3:         vissza érték_0
4:     különben
5:         vissza SorozatszámításRekurzív(x, jobb − 1) ⊕ x[jobb]
6:     elágazás vége
7: függvény vége
```

Függvény hívása: SorozatszámításRekurzív(x, n)

Felhasznált változók és függvények

- x: Feldolgozandó tömb.
- n: Az x tömb mérete.
- $jobb$: A tömb első elemétől a $jobb$ indexű eleméig tartó résztömböt dolgozza fel az aktuális SorozatszámításRekurzív függvény.
- $\oplus$: A tömb elemei között elvégezendő művelet.
- $\mathit{érték}_0$: Az $\oplus$ művelettől függő kiindulási érték.

Az algoritmus 2. sorában megvizsgáljuk, hogy mennyi a $jobb$ index értéke, azaz hogy éppen melyik résztömb feldolgozását végezzük. Amennyiben $jobb=0$, akkor a SorozatszámításRekurzív függvény az $\oplus$ művelet kiindulási értékét adja vissza (ld 3. sor). Amennyiben $jobb>0$, akkor rekurzívan meghívjuk a SorozatszámításRekurzív függvényt 1-gyel kisebb $jobb$ értékkel, majd az így szolgáltatott eredmény és a tömb $x[jobb]$ eleme között elvégezzük az $\oplus$ műveletet. A művelet eredménye lesz a függvény visszatérési értéke (ld. 5. sor).

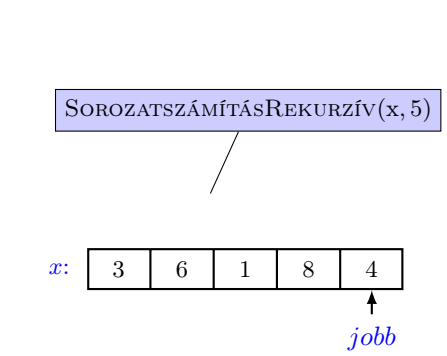
4.5. Példa. Vizsgáljuk meg, miként összegzi a 3,6,1,8,4 elemeket tartalmazó x tömb elemeit a 4.10. algoritmus! Az egyes lépéseket nyomon követhetjük 4.7. ábrán is.

Meghívjuk a SorozatszámításRekurzív függvényt paraméterként átadva neki a feldolgozandó x tömböt és a tömb méretét (5) (ld. 4.7a. ábra). Mivel az átadott $jobb$ paraméter értéke 5, ezért a 2. sorbeli

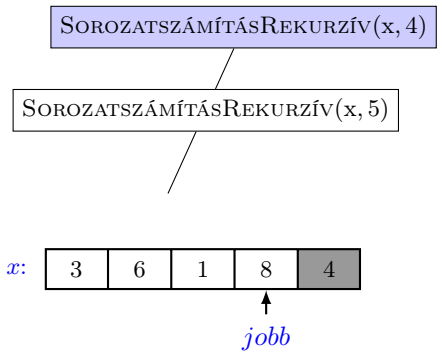
feltétel különben ágába kerül a vezérlés. Az 5. sorban meghívásra kerül a SorozatszámításRekurzív(x, 4) függvény (ld. 4.7b. ábra). Hasonló történik mindaddig, amíg a *jobb* paraméter értéke el nem éri a 0 értéket. Ezeket a hívásokat a 4.7c-4.7f. ábrák szemléltetik.

A SorozatszámításRekurzív(x, 0) függvény, mivel *jobb* értéke 0, ezért az összegzésnél használt $érték_0 = 0$ értéket adja vissza (ld. 4.7g. ábra). A vezérlés visszatér a SorozatszámításRekurzív(x, 1) függvény 5. sorába. Ebben a függvényben a megkapott 0 értékhez hozzáadjuk az $x[1]$ értékét, az így előálló 3 lesz a függvény visszatérési értéke. A vezérlés visszakerül a SorozatszámításRekurzív(x, 2) 5. sorába (ld. 4.7h. ábra). Hasonló történik ahogy sorra visszatér a vezérlés az egyes hívó függvényekhez, ahogy a 4.7i-4.7k. ábrákon láthatjuk. Az algoritmus végén a SorozatszámításRekurzív(x, 5) függvényben előáll a tömb elemeinek összege (ld. 4.7l. ábra). ¶

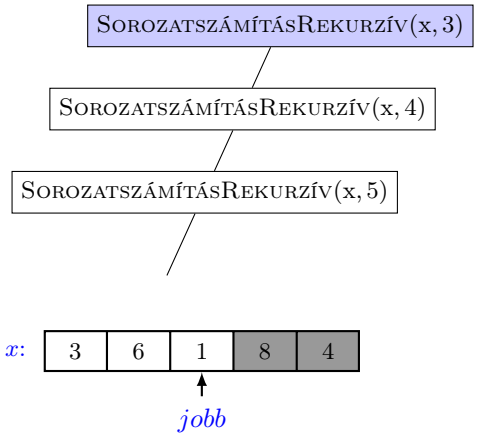
Futási idő elemzése. Amikor a rekurzív algoritmus futási idejét vizsgáljuk, a függvényhívások számát elemezzük. A 4.10. algoritmusból jól látható, hogy egy n elemű tömb feldolgozása során a SorozatszámításRekurzív függvény $(n + 1)$-szer kerül meghívásra. Így a sorozatszámítás rekurzív megvalósításának futási ideje is $O(n)$-es. ♣



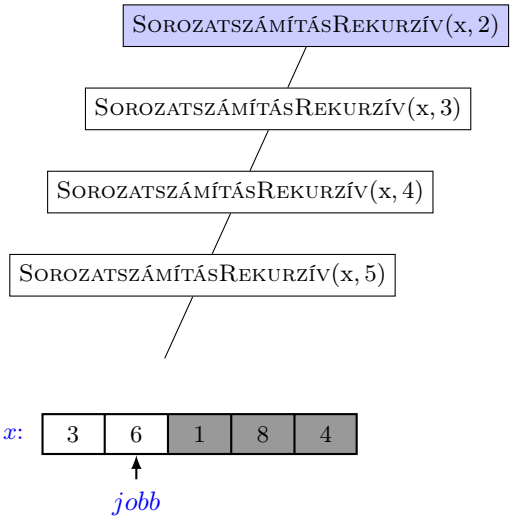
(a) A „külvilágból” meghívásra kerül a SorozatszámításRekurzív(x, 5) függvény, mely a teljes tömböt dolgozza fel.



(b) A SorozatszámításRekurzív(x, 5) függvény meghívja a SorozatszámításRekurzív(x, 4) függvényt, mely a tömb első négy eleméből álló résztömböt dolgozza fel.

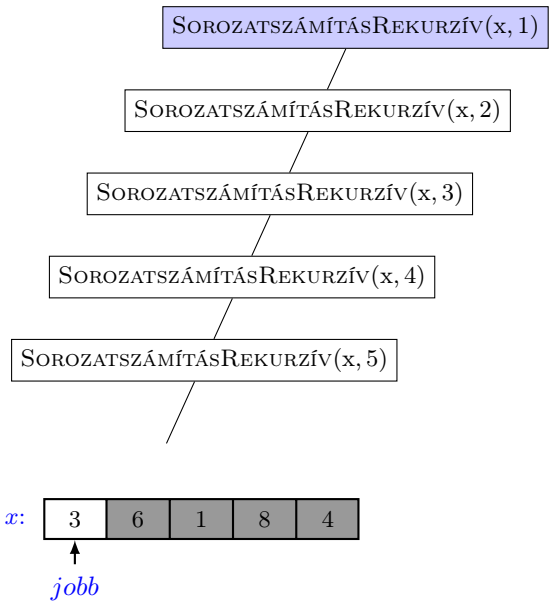


(c) A SorozatszámításRekurzív(x, 4) függvény meghívja a SorozatszámításRekurzív(x, 3) függvényt, mely a tömb első három eleméből álló résztömböt dolgozza fel.

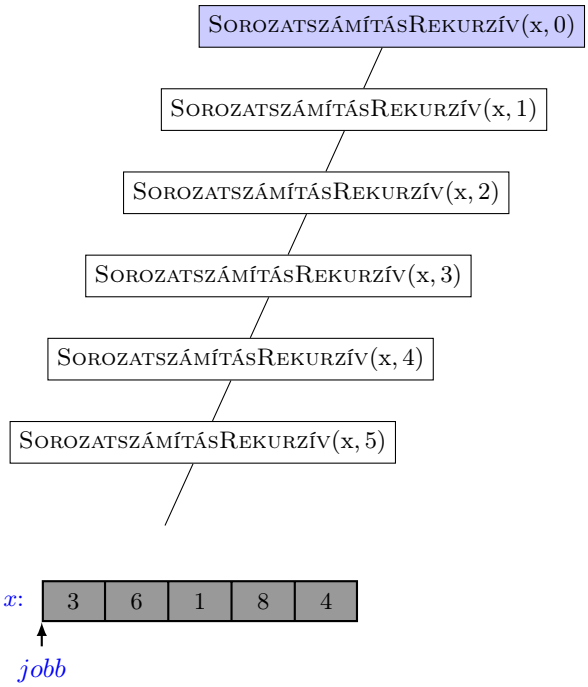


(d) A SorozatszámításRekurzív(x, 3) függvény meghívja a SorozatszámításRekurzív(x, 2) függvényt, mely a tömb első két eleméből álló résztömböt dolgozza fel.

4.7. ábra. Sorozatszámítás programozási tétel rekurzív megvalósítása

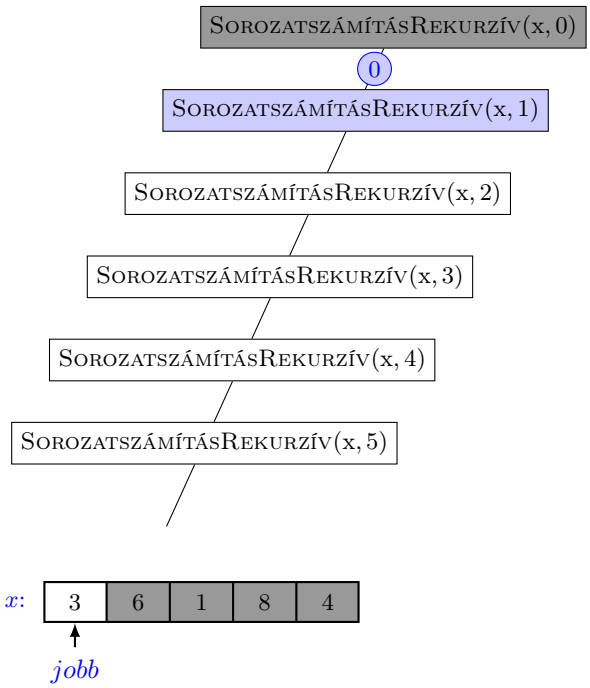


(e) A SorozatszámításRekurzív(x, 2) függvény meghívja a SorozatszámításRekurzív(x, 1) függvényt, mely a tömb első eleméből álló egy elemű résztömböt dolgozza fel.

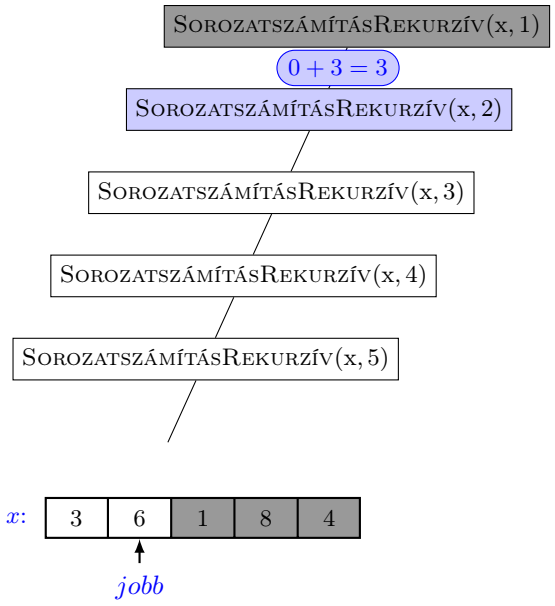


(f) A SorozatszámításRekurzív(x, 1) függvény meghívja a SorozatszámításRekurzív(x, 0) függvényt.

4.7. ábra. Sorozatszámítás programozási tétel rekurzív megvalósítása (folyt.).

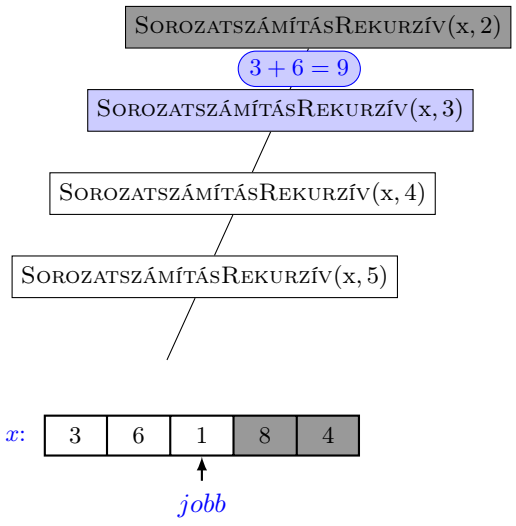


(g) A SorozatszámításRekurzív(x, 0) függvény visszatér az $érték_0$ értékkel, ami most 0. A vezérlés a SorozatszámításRekurzív(x, 1) függvényhez kerül vissza.

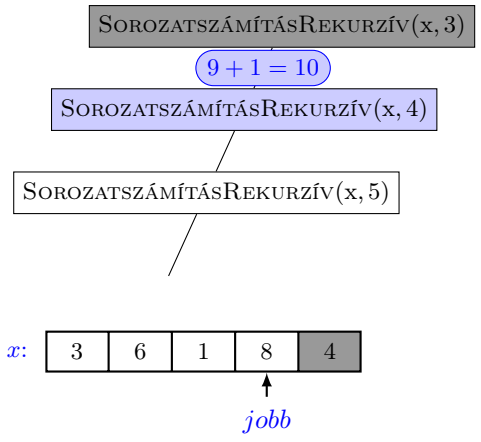


(h) A SorozatszámításRekurzív(x, 1) függvény visszatér a 3 értékkel. A vezérlés a SorozatszámításRekurzív(x, 2) függvényhez kerül vissza.

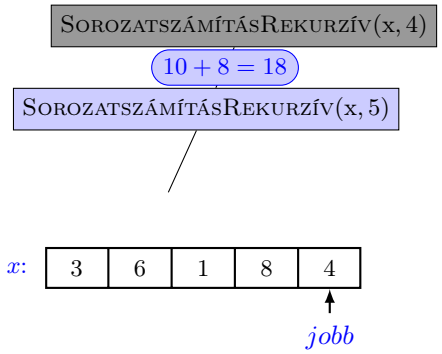
4.7. ábra. Sorozatszámítás programozási tétel rekurzív megvalósítása (folyt.).



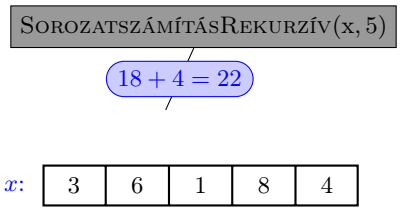
(i) A SorozatszámításRekurzív(x, 2) függvény visszatér a 9 értékkel. A vezérlés a SorozatszámításRekurzív(x, 3) függvényhez kerül vissza.



(j) A SorozatszámításRekurzív(x, 3) függvény visszatér a 10 értékkel. A vezérlés a SorozatszámításRekurzív(x, 4) függvényhez kerül vissza.



(k) A SorozatszámításRekurzív(x, 4) függvény visszatér a 18 értékkel. A vezérlés a SorozatszámításRekurzív(x, 5) függvényhez kerül vissza.



(l) A SorozatszámításRekurzív(x, 5) függvény visszatér a 22 értékkel, ami a tömb elemeinek összege.

4.7. ábra. Sorozatszámítás programozási tétel rekurzív megvalósítása (folyt.).

Lineáris keresés

Az eldöntés (ld. 2.1.2. fejezet), kiválasztás (ld. 2.1.3. fejezet) és lineáris keresés (ld. 2.1.4. fejezet) programozási tételek közül csak az utóbbi rekurzív megvalósítását mutatjuk be. Ennek oka, hogy a lineáris keresés magába foglalja az eldöntést és a kiválasztást is, tehát ennek a rekurzív algoritmusa ismeretében már nagyon egyszerűen meg tudnánk alkotni az eldöntés és kiválasztás rekurzív megvalósításának algoritmusát.

Lineáris keresésnél adott egy x bemeneti tömb, melynek ismerjük az n méretét is. Azt vizsgáljuk, hogy van-e a tömbben P tulajdonságú elem, és ha van, akkor megadjuk az első ilyen elem indexét is. Tehát a lineáris keresésnek két kimenete van. Mivel egyes nyelvi implementációknál egy függvénynek csak egy visszatérési értéke lehet, ezért a lineáris keresést egy olyan rekurzív függvénnyel fogjuk megvalósítani, melynek csak egy visszatérési értéke van. Abban az esetben, ha van P tulajdonságú elem a tömbben, akkor visszaadja a függvény az első ilyen elem indexét. Ha viszont nincs a tömbben P tulajdonságú elem, akkor egy nemlétező indexet, a 0-t adja vissza a függvény.

A rekurzív megvalósítás során kezdetben az egész tömböt vizsgáljuk. Ha az első elem P tulajdonságú, akkor leállhat a futás, ezért visszaadja a függvény az egy értéket. Ha viszont az első elem nem P tulajdonságú, akkor csak a másodiktól az n-edik elemig terjedő résztömbbel kell foglalkoznunk. Ezért rekurzívan meghívjuk a függvényünket úgy, hogy a második elemet tekintjük „elsőnek". Amennyiben a rekurzív hívások során találunk P tulajdonságú elemet, akkor leáll a futás. Ha viszont nincs P tulajdonságú elem a tömbben, akkor már olyan rekurzív hívást eszközölünk, melyben egy nem valós résztömb vizsgálatát kérjük az algoritmustól. Ilyenkor a függvény a 0 visszaadott értékkel fog leállni.

A lineáris keresés rekurzív megvalósítását a 4.11. algoritmusban láthatjuk.

4.11. Algoritmus Lineáris keresés programozási tétel rekurzív megvalósítása

Bemenet: x – **T tömb**, bal – **egész**, n – **egész** (*tömb mérete*), P – **logikai** (*tulajdonság*)
Kimenet: Az első P tulajdonságú elem indexe, illetve ha nincs P tulajdonságú elem, akkor 0.

```
1:  függvény LineárisKeresésRekurzív(x : T tömb, bal : egész, n : egész, P : logikai)
2:      ha bal > n akkor
3:          vissza 0
4:      különben
5:          ha P(x[bal]) akkor
6:              vissza bal
7:          különben
8:              vissza LineárisKeresésRekurzív(x, bal + 1, n, P)
9:          elágazás vége
10:     elágazás vége
11: függvény vége
```

Függvény hívása: LineárisKeresésRekurzív(x, 1, n, P)

Felhasznált változók és függvények

- x: A feldolgozandó tömb.
- n: Az x tömb mérete.
- bal: Az x tömbnek a bal indexű elemétől az utolsó eleméig tartó résztömböt dolgozza fel az aktuális LineárisKeresésRekurzív függvény.
- P: Logikai értéket visszaadó tulajdonságfüggvény. Az algoritmus azt határozza meg, hogy az x tömbnek van-e P tulajdonságú eleme, és ha van, akkor hol található az első ilyen.

Az algoritmus első bemenete az x vizsgált tömb. Második bemenetként megadjuk a bal értéket, amely azt jelzi, hogy hol kezdődik az éppen vizsgált résztömb. Ez a paraméter fog változni a rekurzív hívások során. Kezdeti értéke 1, hiszen kiindulásként a teljes tömböt vizsgáljuk. Ha ennek a bemenetnek az értéke nagyobb lesz a tömb n méreténél, akkor áll elő az az eset, amikor nem találtunk P tulajdonságú elemet a tömbben. Harmadik paraméterként meg kell még adnunk az x tömb méretét. Negyedik bemenetünk a vizsgált P tulajdonságfüggvény lesz. Az algoritmust megvalósító LineárisKeresésRekurzív függvény visszatérési értéke a megtalált P tulajdonságú elem indexe vagy 0 lesz.

Az algoritmus 2. sorában megvizsgáljuk, hogy bal értéke meghaladja-e már a tömb méretét. Ha igen, akkor nem találtunk a tömbben P tulajdonságú elemet, ezért 0-val tér vissza a függvény (ld. 3. sor).

A $bal > n$ feltétel 4. sorban lévő különben ágában két további lehetőséget kell megvizsgálnunk. Ha az éppen vizsgált résztömb első eleme ($x[bal]$) P tulajdonságú (ld. 5. sor), akkor ezen elem indexével tér vissza a függvény (ld. 6. sor). Egyébként viszont egy kisebb résztömböt kell vizsgálni, ezért rekurzívan meghívjuk az ezt végrehajtó függvényt (ld. 8. sor).

4.6. Példa. A 3, 9, 4, 1, 8 tömbben keressük, hogy van-e páros elem, és ha van, akkor hol található az első ilyen. A 4.11. algoritmussal való megvalósítást nyomon követhetjük a 4.8. ábrán.

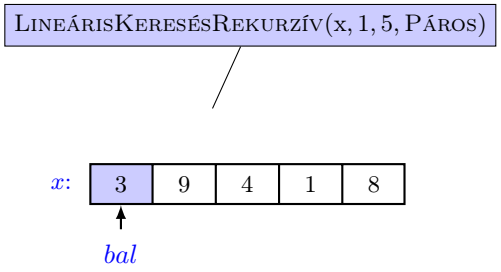
Először meghívjuk a LineárisKeresésRekurzív(x, 1, 5, Páros) függvényt (ld. 4.8a. ábra). Mivel *bal* értéke 1, ami a tömb méreténél kisebb, ezért a 2. sorbeli feltétel különben ágába lépünk. Megvizsgáljuk, hogy $x[1]$ páros-e (ld. 5. sor). Mivel az első elem páratlan, ezért a 8. sorban meghívjuk rekurzívan a kisebb tömb vizsgálatát megvalósító függvényt.

A LineárisKeresésRekurzív(x, 2, 5, Páros) függvényben is hasonló vizsgálatokat végzünk (ld. 4.8b. ábra). Mivel a második elem sem páros, ezért újabb rekurzív hívás történik. A LineárisKeresésRekurzív(x, 3, 5, Páros) függvényben a tömb harmadik eleméről megállapítjuk, hogy páros (ld. 4.8c. ábra). Így a vezérlés az algoritmus 6. sorába kerül. A soron következő LineárisKeresésRekurzív(x, 3, 5, Páros) függvény visszaadja a megtalált indexet, a vezérlés pedig visszakerül a LineárisKeresésRekurzív(x, 2, 5, Páros) függvény 8. sorába. Ez a függvény is visszaadja a 3 értéket, a vezérlés pedig a LineárisKeresésRekurzív(x, 1, 5, Páros) függvénybe kerül vissza (ld. 4.8e. ábra). Ennek a függvénynek a futása is véget ér, visszaadja a megtalált első páros elem indexét (ld. 4.8f. ábra). ¶

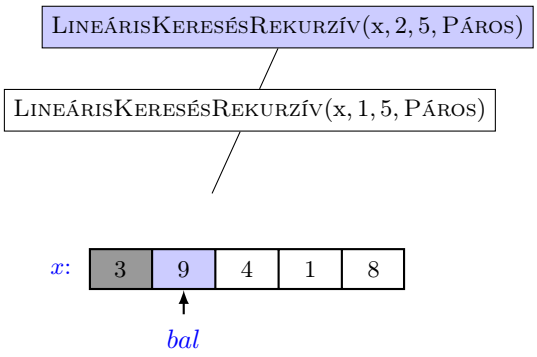
Futási idő elemzése. Az algoritmus futási ideje, azaz a rekurzív hívások száma függ attól, hogy van-e a keresett P tulajdonságú elem a tömbben. Ha van ilyen elem, akkor a futási idő attól függ, hogy hányadik helyen található az első ilyen elem. A függvényhívások száma pontosan az első P tulajdonságú elem indexével lesz egyenlő. Tehát legjobb esetben csak egy hívás szükséges, legrosszabb esetben pedig – amikor az utolsó tömbelem az egyetlen P tulajdonságú – n hívás történik. Átlagosan $\frac{n}{2}$ hívás kell.

Ha nincs P tulajdonságú elem a tömbben, akkor viszont $n + 1$ hívás szükséges.

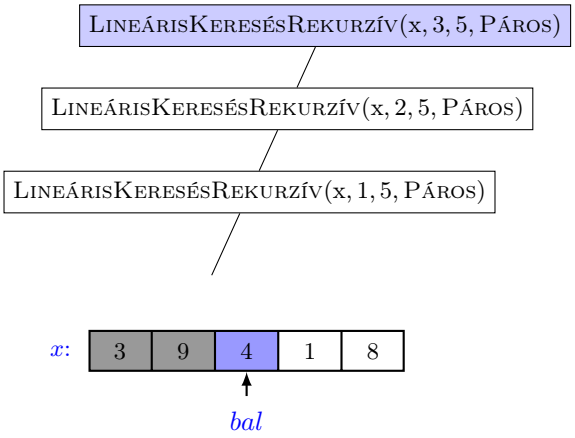
Összességében azt látjuk, hogy az algoritmusunk futási ideje $O(n)$-es. ♣



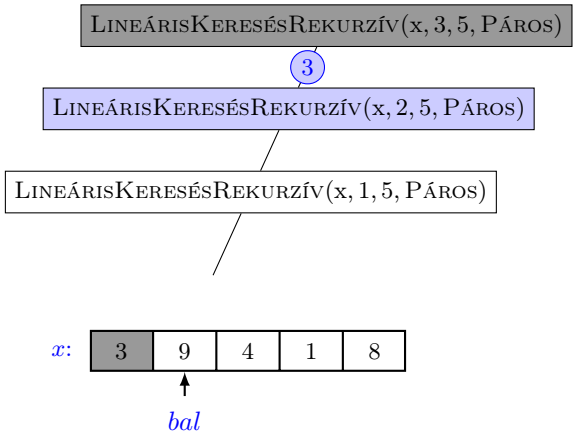
(a) Meghívásra kerül a LineárisKeresésRekurzív (x, 1, 5, Páros) függvény, mely a teljes x tömbben keres páros számot.



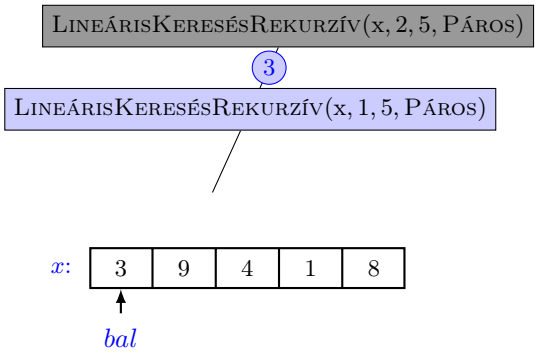
(b) A LineárisKeresésRekurzív (x, 1, 5, Páros) függvény meghívja a LineárisKeresésRekurzív (x, 2, 5, Páros) függvényt, mely az x tömb 2. és 5. eleme közötti résztömbben keres páros számot.



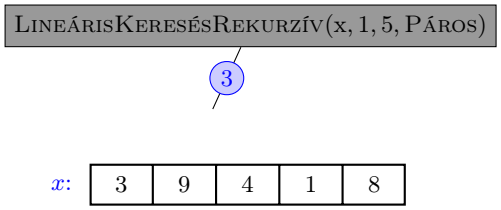
(c) A LineárisKeresésRekurzív (x, 2, 5, Páros) függvény meghívja a LineárisKeresésRekurzív(x, 3, 5, Páros) függvényt, mely az x tömb 3. és 5. eleme közötti résztömbben keres páros számot.



(d) Mivel a 3. elem páros, ezért a LineárisKeresésRekurzív(x, 3, 5, Páros) függvény visszatér a 3 értékkel. A vezérlés visszakerül a LineárisKeresésRekurzív(x, 2, 5, Páros) függvényhez.



(e) A LineárisKeresésRekurzív(x, 2, 5, Páros) függvény is a 3 értékkel tér vissza. A vezérlés a LineárisKeresésRekurzív(x, 1, 5, Páros) függvényhez kerül vissza.



(f) A LineárisKeresésRekurzív(x, 1, 5, Páros) függvény is a 3 értékkel tér vissza. A feladatot megoldottuk, van páros elem a tömbben, mégpedig a 3. elem az első ilyen tulajdonságú.

4.8. ábra. Lineáris keresés rekurzív megvalósítása. Az x tömbben páros számot keresünk.

Megszámlálás

A 2.1.5. fejezetben már megismertük a megszámlálás programozási tételt, melynek feladata egy tömbben megszámolni a P tulajdonságú elemek darabszámát. Most megnézzük, miként lehetséges ezt rekurzív módon megvalósítani.

A feladatot úgy oldjuk meg, hogy megvizsgáljuk az x tömb utolsó elemét. Ha úgy találjuk, hogy $x[n]$ P tulajdonságú, akkor a teljes tömbben a P tulajdonságú elemek száma eggyel több, mint az $n-1$ elemű tömbben. Ha viszont $x[P]$ nem P tulajdonságú, akkor az n elemű tömbnek ugyanannyi P tulajdonságú eleme van, mint az eggyel kevesebb elemű tömbnek.

A rekurzió leállásáról az az alapeset gondoskodik, hogy egy elfajult nulla elemű tömbnek egyetlen P tulajdonságú eleme sincs.

Az ismertetett elvnek megfelelő pszeudokódot a 4.12. algoritmusban adtuk meg.

4.12. Algoritmus Megszámlálás programozási tétel rekurzív megvalósítása

Bemenet: x – **T tömb**, $jobb$ – **egész**, P – **logikai** (*tulajdonság*)
Kimenet: A vizsgált résztömbben az P tulajdonságú elemek száma.

```
1:  függvény MegszámlálásRekurzív(x : T tömb, jobb : egész, P : logikai)
2:      ha jobb = 0 akkor
3:          vissza 0
4:      különben
5:          ha P(x[jobb]) akkor
6:              vissza 1 + MegszámlálásRekurzív(x, jobb − 1, P)
7:          különben
8:              vissza MegszámlálásRekurzív(x, jobb − 1, P)
9:          elágazás vége
10:     elágazás vége
11: függvény vége
```

Függvény hívása: MegszámlálásRekurzív(x, n, P)

Felhasznált változók és függvények

- x: A feldolgozandó tömb.
- n: Az x tömb mérete.
- $jobb$: Az x tömbnek az első elemétől a $jobb$ indexű eleméig tartó résztömböt dolgozza fel az aktuális MegszámlálásRekurzív függvény.
- P: Logikai értéket visszaadó tulajdonság függvény. Az algoritmus azt határozza meg, hogy az x tömbnek hány darab P tulajdonságú eleme van.

A MegszámlálásRekurzív függvény bemenete a feldolgozandó x tömb, az aktuálisan vizsgált résztömb végét jelző $jobb$ index, valamint a vizsgált P tulajdonság. A függvény visszatérési értéke az x első $jobb$ darab eleme között a P tulajdonságúak száma.

Az algoritmus 2. sorában megvizsgáljuk, hogy nulla elemű résztömbbel dolgozunk-e. Ha $jobb = 0$, akkor visszaadjuk a 0 értéket (ld. 3. sor), mivel egy elfajult tömbben nincs elem, tehát a P tulajdonságú elemek száma 0. Így biztosítottuk a rekurzió alapesetét.

A 4. sorban kezdődő különben ágba lépünk akkor, ha legalább egy elemű a vizsgált résztömbünk. Ha a résztömb utolsó eleme ($x[jobb]$) P tulajdonságú (ld. 5. sor), akkor az első és ($jobb-1$)-edik elemek közötti résztömb P tulajdonságú elemeinek számához 1-et hozzá kell adni. Ennek érdekében rekurzívan meghívjuk a MegszámlálásRekurzív(x, $jobb-1$, P) függvényt, és az 1-gyel növelt értékkel áll le az aktuális függvény futása (ld. 6. sor). Ha viszont a $jobb$ indexű elem nem volt P tulajdonságú, akkor az aktuális függvény visszatérési értéke a rekurzívan hívott függvény visszatérési értékével lesz azonos (ld. 8. sor).

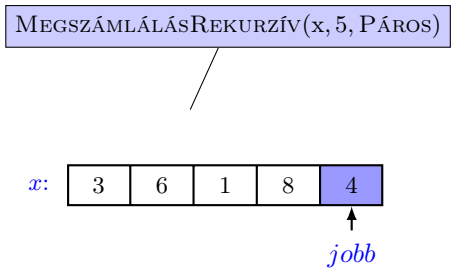
4.7. Példa. A 3, 6, 1, 8, 4 elemeket tartalmazó tömb példáján nézzük meg hogyan is működik a rekurzív megszámlálás. A tömbben a páros számokat keressük. A feldolgozás egyes lépéseit a 4.9. ábrán követhetjük.

A MegszámlálásRekurzív függvényben csak a $jobb$ paraméter fog változni, ezért ennek az értékére térünk csak ki. Először meghívjuk a függvényt a tömb méretével egyező értékkel. Mivel $jobb$ nem 0,

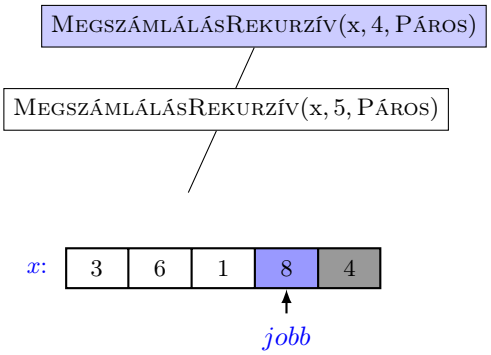
ezért a 4. sorban kezdődő különben ágba lépünk. Megvizsgáljuk az ötödik elem értékét, ami jelen esetben páros lesz (ld. 4.9a. ábra). Így az algoritmus 6. sorába lépünk, ahonnan meghívjuk a MEGSZÁMLÁLÁSREKURZÍV függvényt a $jobb = 4$ értékkel. A negyedik elem is páros, ahogy az a 4.9b. ábrán is látható. A vezérlés ismét a 6. sorba kerül. Meghívjuk a $jobb = 3$ paraméterű függvényt. A harmadik elem páratlan (ld. 4.9c. ábra), ezért a vezérlés a 8. sorba ugrik. A rekurzív hívások további sorát szemléltetik a 4.9d-4.9f. ábrák.

A $jobb = 0$ esetben a függvény visszatér a 0 értékkel, a vezérlés pedig visszatér a MEGSZÁMLÁLÁSREKURZÍV(x, 1, PÁROS) függvény 8. sorába (ld. 4.9g. ábra). A többi függvény visszatérési értékeit és így a páros értékek számának alakulását a 4.9h-4.9l. ábrák szemléltetik. Végeredményül 3-at kapunk, ami tényleg megegyezik a tömb páros értékű elemeinek számával. ¶

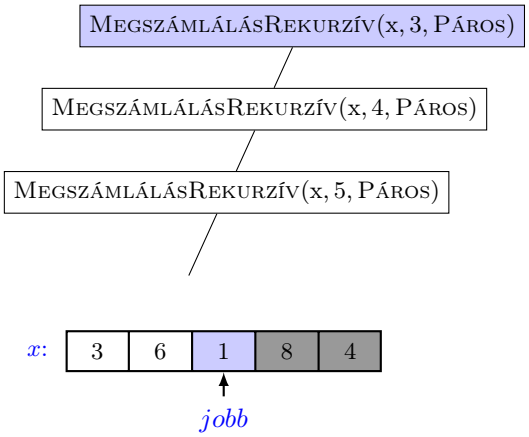
Futási idő elemzése. A futási idő vizsgálatánál most is a függvényhívások számát vesszük figyelembe. Könnyen látható, hogy egy n elemű tömb vizsgálatánál $n + 1$ darab függvényhívás történik, így ez az algoritmus is $O(n)$-es. ♣



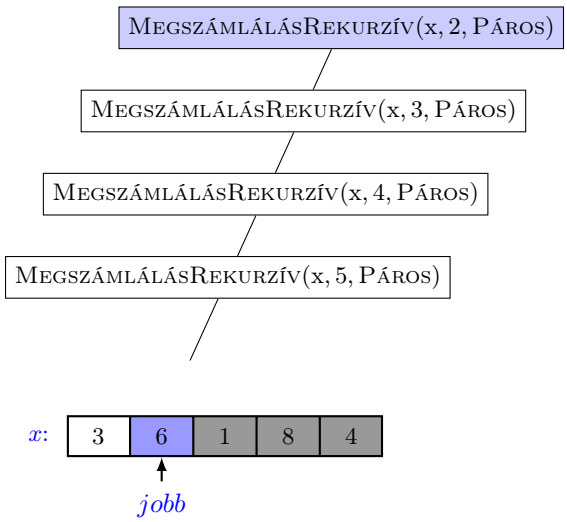
(a) Mehívásra kerül a MEGSZÁMLÁLÁSREKURZÍV(x, 5, PÁROS) függvény, mely meghatározza, hogy hány páros szám van a teljes tömbben.



(b) A MEGSZÁMLÁLÁSREKURZÍV(x, 5, PÁROS) függvény meghívja a MEGSZÁMLÁLÁSREKURZÍV(x, 4, PÁROS) függvényt, amely a tömb első 4 elemében számolja meg a párosakat.

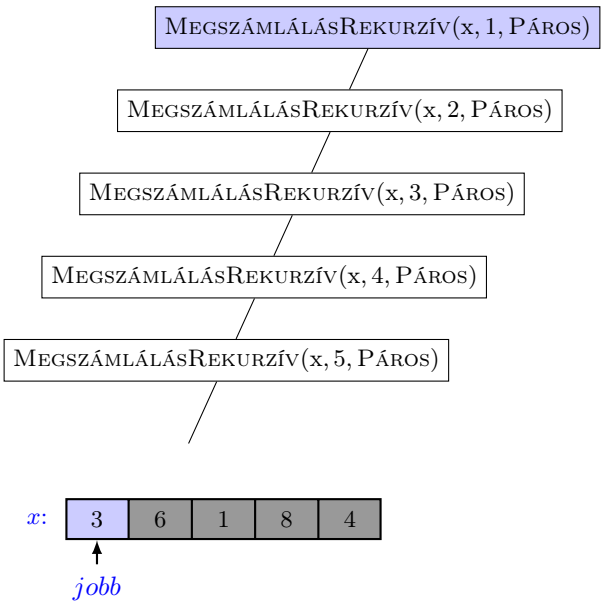


(c) A MEGSZÁMLÁLÁSREKURZÍV(x, 4, PÁROS) függvény meghívja a MEGSZÁMLÁLÁSREKURZÍV(x, 3, PÁROS) függvényt, amely a tömb első 3 elemében számolja meg a párosakat.

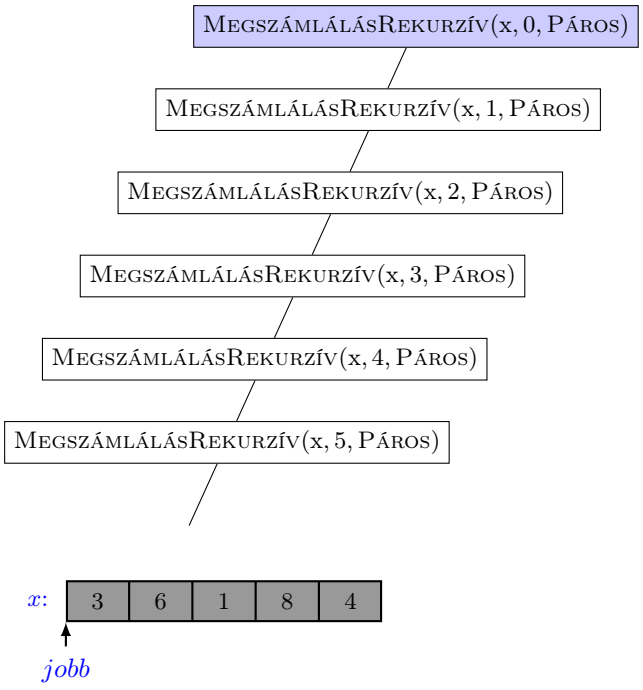


(d) A MEGSZÁMLÁLÁSREKURZÍV(x, 3, PÁROS) függvény meghívja a MEGSZÁMLÁLÁSREKURZÍV(x, 2, PÁROS) függvényt, amely a tömb első 2 elemében számolja meg a párosakat.

4.9. ábra. Megszámlálás programozási tétel rekurzív megvalósítása. A 3, 6, 1, 8, 4 tömbben keressük a páros értékek darabszámát.

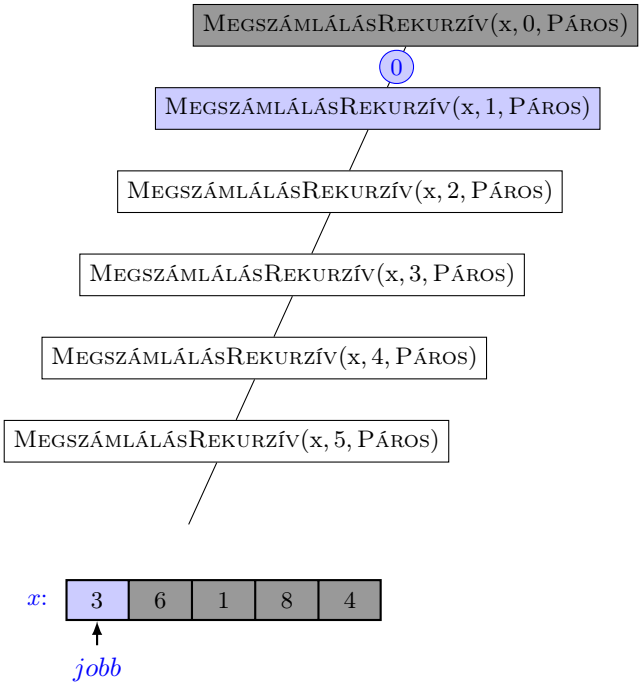


(e) A MegszámlálásRekurzív(x, 2, Páros) függvény meghívja a MegszámlálásRekurzív(x, 1, Páros) függvényt, amely a tömb első 1 elemében számolja meg a párosakat.

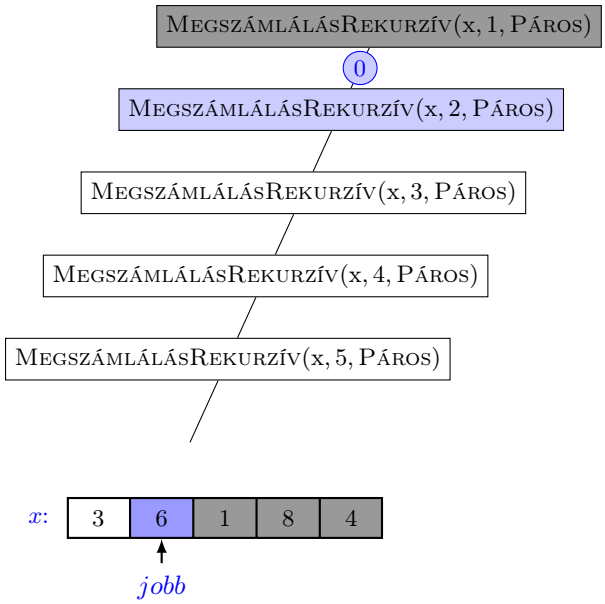


(f) A MegszámlálásRekurzív(x, 1, Páros) függvény meghívja a MegszámlálásRekurzív(x, 0, Páros) függvényt.

4.9. ábra. Megszámlálás programozási tétel rekurzív megvalósítása. A 3, 6, 1, 8, 4 tömbben keressük a páros értékek darabszámát (folyt.).

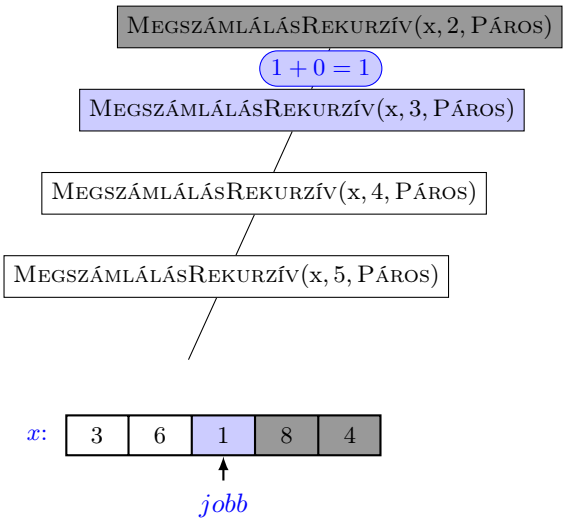


(g) A MegszámlálásRekurzív(x, 0, Páros) függvény visszatér a 0 értékkel. A vezérlés a MegszámlálásRekurzív(x, 1, Páros) függvényhez kerül vissza.

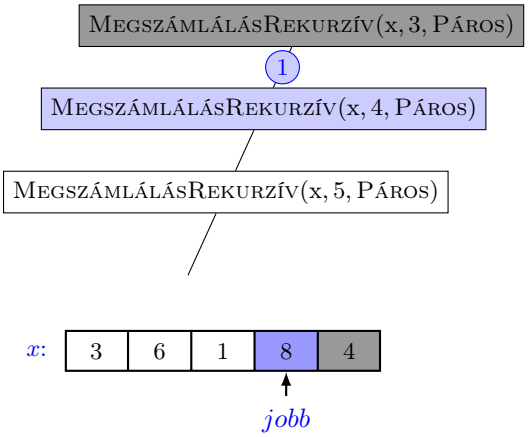


(h) A MegszámlálásRekurzív(x, 1, Páros) függvény visszatér a 0 értékkel. A vezérlés a MegszámlálásRekurzív(x, 2, Páros) függvényhez kerül vissza.

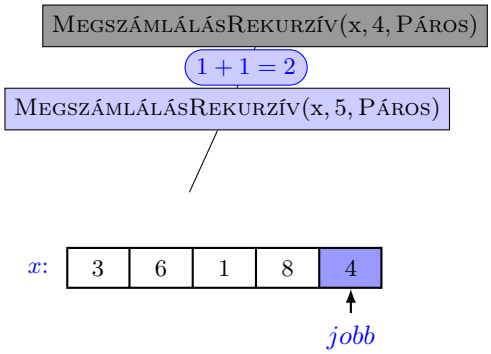
4.9. ábra. Megszámlálás programozási tétel rekurzív megvalósítása. A 3, 6, 1, 8, 4 tömbben keressük a páros értékek darabszámát (folyt.).



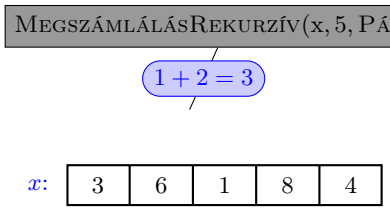
(i) A MegszámlálásRekurzív(x, 2, Páros) függvény visszatér a 1 értékkel. A vezérlés a MegszámlálásRekurzív(x, 3, Páros) függvényhez kerül vissza.



(j) A MegszámlálásRekurzív(x, 3, Páros) függvény visszatér a 1 értékkel. A vezérlés a MegszámlálásRekurzív(x, 4, Páros) függvényhez kerül vissza.



(k) A MegszámlálásRekurzív(x, 4, Páros) függvény visszatér a 2 értékkel. A vezérlés a MegszámlálásRekurzív(x, 5, Páros) függvényhez kerül vissza.



(l) A MegszámlálásRekurzív(x, 5, Páros) függvény visszatér a 3 értékkel, ami a tömbben lévő párosak száma.

4.9. ábra. Megszámlálás programozási tétel rekurzív megvalósítása. A 3, 6, 1, 8, 4 tömbben keressük a páros értékek darabszámát (folyt.).

Maximumkiválasztás

A maximumkiválasztás programozási tétel (ld. 2.1.6. fejezet) egy tömb elemei közül megadja a maximális értékű elem indexét. Miként lehetséges ezt rekurzív módon megvalósítani?

Ha egy elemű a tömbünk, akkor a maximális értékű elem az első elem. Ez lehet majd a rekurziónk alapesete, vagy más néven leállási feltétele. Ha nagyobb tömböt vizsgálunk (pl. egy n eleműt), akkor ha ismerjük az első $n-1$ elem közül a legnagyobb indexét, akkor csak ezt az elemet kell összehasonlítani az n-edik elemmel. A kettő közül a nagyobbik lesz a maximális. Ezt az elgondolást megvalósító algoritmus pszeudokódját a 4.13. algoritmusban írjuk le.

Az algoritmus bemenete a vizsgálandó x tömb, valamint egy *jobb* index. A *jobb* paraméter adja meg, hogy éppen melyik elemet akarjuk az előtte lévők maximumával összehasonlítani. Kezdetben *jobb* értéke megegyezik a tömb n elemszámával. A rekurzív hívások során *jobb* értéke fog egyesével csökkenni. A rekurziónak akkor lesz vége, ha *jobb* értéke eléri az 1-et.

4.13. Algoritmus Maximumkiválasztás programozási tétel rekurzív megvalósítása

Bemenet: x – **T tömb**, *jobb* – **egész**; ahol **T** összehasonlítható
Kimenet: A vizsgált résztömbben a maximális értékű elem indexe.

```
1:  függvény MaximumkiválasztásRekurzív(x : T tömb, jobb : egész)
2:      ha jobb = 1 akkor
3:          vissza 1
4:      különben
5:          max ← MaximumkiválasztásRekurzív(x, jobb − 1)
6:          ha x[jobb] > x[max] akkor
7:              vissza jobb
8:          különben
9:              vissza max
10:         elágazás vége
11:     elágazás vége
12: függvény vége
```

Függvény hívása: MaximumkiválasztásRekurzív(x, n)

Felhasznált változók és függvények

- x: A feldolgozandó tömb, melynek elemei összehasonlíthatók.
- n: Az x tömb mérete.
- *jobb*: Az x tömbnek az első elemétől a *jobb* indexű eleméig tartó résztömböt dolgozza fel az aktuális MaximumkiválasztásRekurzív függvény.
- *max*: Az x tömb első $jobb-1$ darab eleme közül a maximális értékű elem indexe.

Az algoritmus 2. sorában megvizsgáljuk, hogy *jobb* értéke 1-e. Ha igen, akkor visszaadja a MaximumkiválasztásRekurzív függvény az 1 indexet (3. sor). Egyéb esetekben meg kell vizsgálnunk, hogy az aktuális $x[jobb]$ elem hogyan viszonyul a nála kisebb indexű elemek maximumához. Ennek érdekében először meg kell határozni, hogy az x első $jobb-1$ eleme közül melyik a maximális, ezért rekurzívan meghívjuk a MaximumkiválasztásRekurzív függvényt, melynek visszatérési értékét a *max* változóba mentjük (ld. 5. sor). Ezt követően megnézzük, hogy az $x[jobb]$ hogyan viszonyul az őt megelőző elemek maximumához (ld. 6. sor), és ennek megfelelően visszaadjuk vagy a *jobb* aktuális értékét (7. sor), vagy a megelőző elemek maximumának indexét (9. sor).

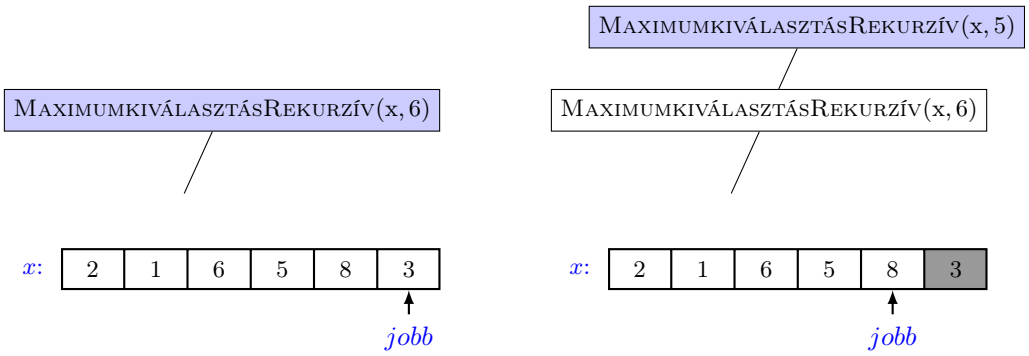
> Megjegyzés
>
> Az algoritmus megadható úgy is, hogy a *jobb* indexnél korábbi elemek maximumát nem mentjük ki egy külön változóba (*max*), hanem a 6. sorban és a 9. sorban is meghívjuk a MaximumkiválasztásRekurzív függvényt. Így viszont növekedne a rekurzív függvényhívások száma, amit el akarunk kerülni.

4.8. Példa. Határozzuk meg a 2, 1, 6, 5, 8, 3 elemeket tartalmazó x tömb legnagyobb értékű elemének indexét a 4.13. algoritmus használatával! A megoldás során előálló függvényhívásokat, a vizsgált résztömb alakulását és a visszatérési értékeket nyomon követhetjük a 4.10. ábrán.

Először meghívjuk a MaximumkiválasztásRekurzív(x, 6) függvényt (ld. 4.10a. ábra). Mivel *jobb* értéke nem 1, ezért rögtön hívjuk a MaximumkiválasztásRekurzív(x, 5) függvényt (ld. 4.10b. ábra). A további függvényhívásokat ábrázoltuk a 4.10c-4.10f. ábrákon.

Amikor *jobb* értéke 1, akkor a MaximumkiválasztásRekurzív(x, 1) függvény visszatér az 1 értékkel. A vezérlés visszakerül a MaximumkiválasztásRekurzív(x, 2) függvényhez. Ebben a függvényben összehasonlítjuk az második és a *max* indexű elemet (ld. 4.10h. ábra). Mivel az első elem nagyobb ezért a MaximumkiválasztásRekurzív(x, 2) függvény is 1-et fog visszaadni a MaximumkiválasztásRekurzív(x, 3) függvénynek. Itt is elvégezzük az aktuális összehasonlítást (ld. 4.10h. ábra). A harmadik elem viszont nagyobb az első elemnél, ezért a MaximumkiválasztásRekurzív(x, 3) függvény a 3-at fogja visszaadni a MaximumkiválasztásRekurzív(x, 4) függvénynek. Az algoritmus további futását a 4.10i-4.10l. ábrák szemléltetik. Az algoritmus végén azt kapjuk vissza, hogy az x tömb ötödik eleme a legnagyobb. ¶

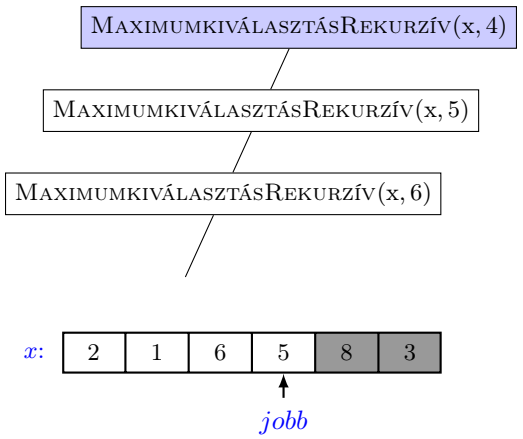
Futási idő elemzése. A 4.13. algoritmus végrehajtása során mindig n darab függvényhívást végzünk, így a maximumkiválasztás rekurzív megvalósítása is $O(n)$-es futási idejű algoritmus. ♣



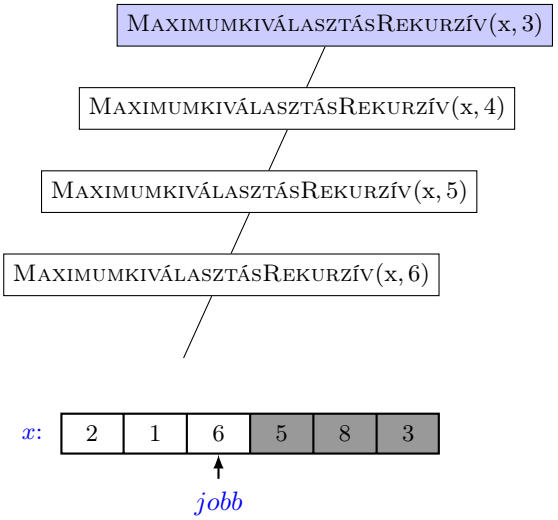
(a) Meghívásra kerül a MaximumkiválasztásRekurzív(x, 6) függvény, mely a teljes tömb maximális elemét választja ki.

(b) A MaximumkiválasztásRekurzív(x, 6) függvényből meghívásra kerül a MaximumkiválasztásRekurzív(x, 5) függvény, mely a tömb első öt elemének maximális elemét választja ki.

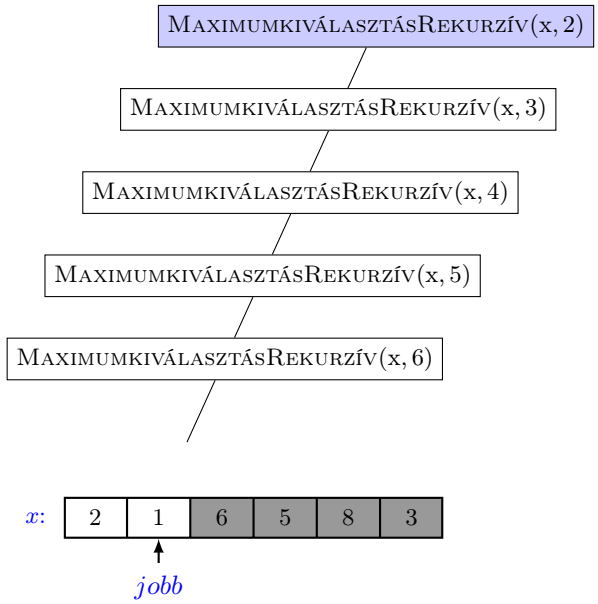
4.10. ábra. Maximumkiválasztás programozási tétel rekurzív megvalósítása. A 2, 1, 6, 5, 8, 3 elemei között keressük a legnagyobb értékűt.



(c) A MaximumkiválasztásRekurzív(x, 5) függvényből meghívásra kerül a MaximumkiválasztásRekurzív(x, 4) függvény, mely a tömb első négy elemének maximális elemét választja ki.

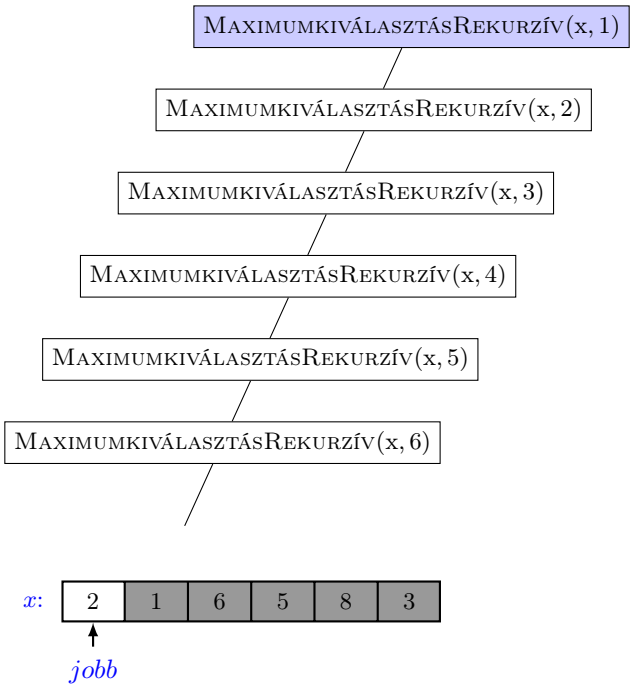


(d) A MaximumkiválasztásRekurzív(x, 4) függvényből meghívásra kerül a MaximumkiválasztásRekurzív(x, 3) függvény, mely a tömb első három elemének maximális elemét választja ki.

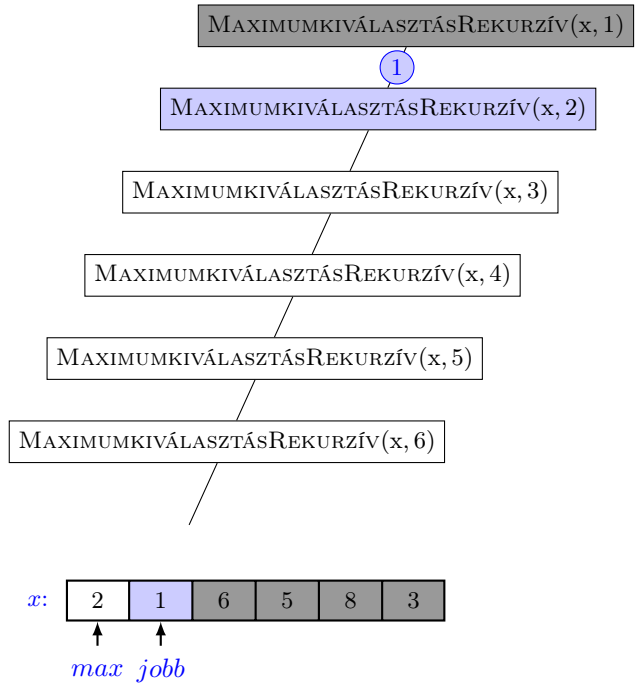


(e) A MaximumkiválasztásRekurzív(x, 3) függvényből meghívásra kerül a MaximumkiválasztásRekurzív(x, 2) függvény, mely a tömb első két elemének maximális elemét választja ki.

4.10. ábra. Maximumkiválasztás programozási tétel rekurzív megvalósítása. A 2, 1, 6, 5, 8, 3 elemei között keressük a legnagyobb értékűt (folyt.).

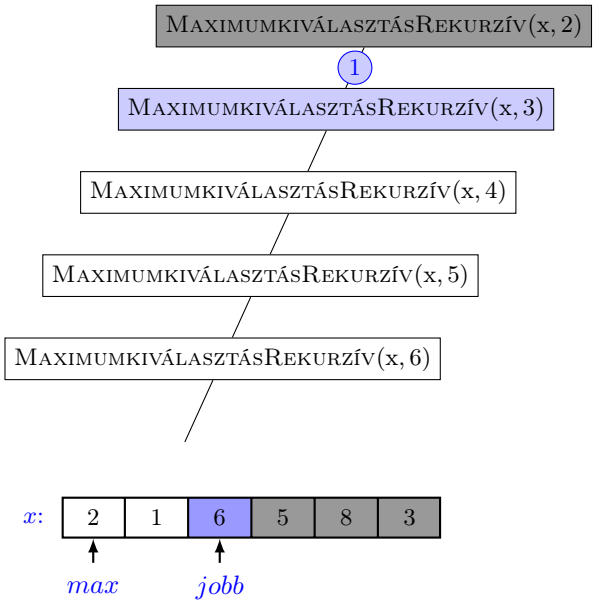


(f) A MaximumkiválasztásRekurzív(x, 2) függvényből meghívásra kerül a MaximumkiválasztásRekurzív(x, 1) függvény, mely a tömb első elemét választja ki maximális elemként.

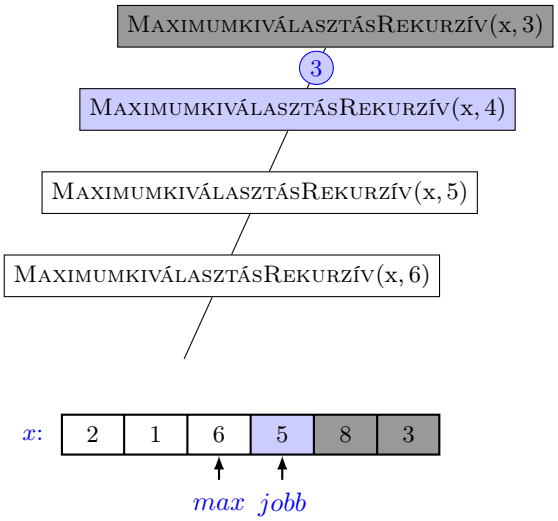


(g) A MaximumkiválasztásRekurzív(x, 1) függvény visszaadja az 1-et, a vezérlés pedig a MaximumkiválasztásRekurzív(x, 2) függvényhez kerül vissza.

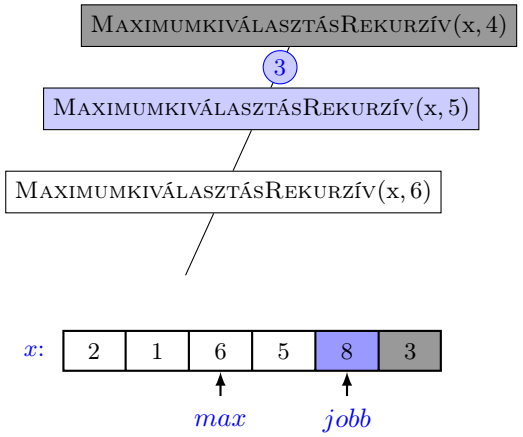
4.10. ábra. Maximumkiválasztás programozási tétel rekurzív megvalósítása. A 2, 1, 6, 5, 8, 3 elemei között keressük a legnagyobb értékűt (folyt.).



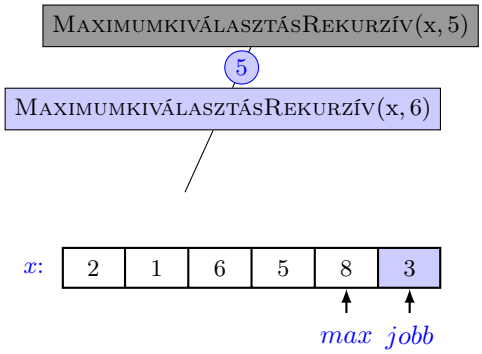
(h) A MaximumkiválasztásRekurzív(x, 2) függvény visszaadja az 1-et, a vezérlés pedig a MaximumkiválasztásRekurzív(x, 3) függvényhez kerül vissza.



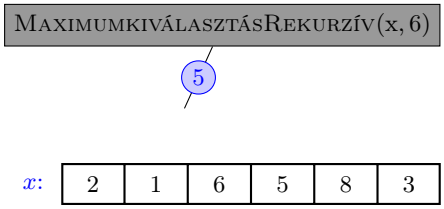
(i) A MaximumkiválasztásRekurzív(x, 3) függvény visszaadja az 3-at, a vezérlés pedig a MaximumkiválasztásRekurzív(x, 4) függvényhez kerül vissza.



(j) A MaximumkiválasztásRekurzív(x, 4) függvény visszaadja az 3-at, a vezérlés pedig a MaximumkiválasztásRekurzív(x, 5) függvényhez kerül vissza.



(k) A MaximumkiválasztásRekurzív(x, 5) függvény visszaadja az 5-öt, a vezérlés pedig a MaximumkiválasztásRekurzív(x, 6) függvényhez kerül vissza.



(l) A MaximumkiválasztásRekurzív(x, 6) függvény visszaadja az 5-öt, ami a tömb maximális elemének indexe.

4.10. ábra. Maximumkiválasztás programozási tétel rekurzív megvalósítása. A 2, 1, 6, 5, 8, 3 elemei között keressük a legnagyobb értékűt (folyt.).

5. fejezet

Rendezett tömbök

Ebben a fejezetben növekvő módon rendezett tömbökkel foglalkozunk. Azt már korábban, a 3. fejezetben láttuk, hogy a tömböket miként tehetjük rendezetté. Ezt most nem vizsgáljuk, csak kihasználjuk.

Először áttekintjük, hogy miként lehetséges kereséseket végrehajtani rendezett tömbökben (ld. 5.1. alfejezet). A már ismert lineáris keresést módosítjuk és vizsgáljuk a futási idő változását (ld. 5.1.1. alfejezet). Ezt követően megismerkedünk egy olyan módszerrel, a logaritmikus kereséssel, amely csak rendezett tömbök esetén használható (ld. 5.1.2. alfejezet). Ez az újfajta megközelítés a futási idő jelentős javulásához vezet.

Az 5.2. alfejezetben megvizsgáljuk, hogy egyes programozási tételeket miként lehetséges módosítani, futási idő szempontjából hatékonyabbá tenni rendezett tömbök esetén. A kereséssel rokon eldöntés (ld. 5.2.1. alfejezet) és kiválasztás (ld. 5.2.2. alfejezet) tételek rendezett változatát ismertetjük, valamint megismerkedünk a kiválogatás(ld. 5.2.3. alfejezet) és a megszámlálás (ld. 5.2.4. alfejezet) tételek módosított algoritmusaival.

A fejezet harmadik részében egy speciális adatszerkezet, a halmazok tömbökben történő ábrázolását mutatjuk be (ld. 5.3. alfejezet). A halmazokat ismétlődő elemeket nem tartalmazó növekvő módon rendezett tömbökként reprezentáljuk. Algoritmust adunk annak vizsgálatára, hogy egy rendezett tömb halmaznak tekinthető-e (ld. 5.3.1. alfejezet). Megmutatjuk, hogy egy rendezett tömböt miként lehetséges halmazzá alakítani (ld. 5.3.2. alfejezet). A kereséseknél megismert módszer alapján bemutatjuk az elemtartalmazás vizsgálatának algoritmusát (ld. 5.3.3. alfejezet). Bemutatjuk a részhalmaz tulajdonság vizsgálatának hatékony eljárását (ld. 5.3.4. alfejezet). A halmazok tárgyalásának végén négy halmazművelet algoritmusát ismertetjük. Ezek az unió (ld. 5.3.5. alfejezet), a metszet (ld. 5.3.6. alfejezet), a különbség (ld. 5.3.7. alfejezet) és a szimmetrikus differencia (ld. 5.3.8. alfejezet).

Keresések rendezett tömbökben

A keresés feladatával már megismerkedtünk a 2.1.4. fejezetben. Ott azt láttuk, hogy egy ismert méretű x tömbben keresünk adott (P) tulajdonságú elemet. Mivel nem tudjuk, hogy van-e egyáltalán P tulajdonságú elem x-ben, ezért a keresés egyik visszatérési értéke erről ad információt (*van* logikai változó). Ha találtunk P tulajdonságú elemet az x tömbben, akkor azt is tudni akarjuk, hogy hol találtunk ilyet. Erről ad információt az *idx* visszatérési érték.

Rendezett tömbök esetén kis mértékben módosítunk a feladaton annak érdekében, hogy a rendezettséget valóban ki tudjuk használni. Nem azt fogjuk vizsgálni, hogy van-e adott tulajdonságú elem a tömbben, hanem egy konkrét értéket keresünk, amit az *érték* változóval jelölünk (ld. 2.8. algoritmus). Ha van a keresett *érték*-kel megegyező elem a tömbben, akkor az előtte lévő elemek mind kisebbek nála, az utána következők pedig mind nagyobbak nála, hiszen növekvő módon rendezett a tömb.

Megjegyzés

Utóbbi kijelentésünk csak akkor igaz, ha a keresett *érték* pontosan egyszer fordul elő a tömbben. Lehetséges viszont, hogy többször is benne van. Ekkor abban biztosak lehetünk, hogy az összes előfordulás szomszédos helyeken van. Az *érték*-nél kisebb elemek pedig az *érték* első előfordulása előtt, a nagyobb elemek pedig az *érték* utolsó előfordulása után találhatók a tömbben.

Lineáris keresés

A 2.1.4. fejezetben megismert lineáris keresés algoritmusában a tömb elejétől indulva elemről elemre haladva járjuk be a vizsgált tömbünket, ahogy azt a 2.8. algortimusban láttuk. A bejárás akkor ér véget, ha megtaláltuk a keresett elemet, vagy ha a tömb végére értünk.

Rendezett tömbök esetén, amikor egy konkrét *érték*-et keresünk javíthatunk a 2.8. algoritmuson. Ugyanis a tömb bejárása nem csak a fentebb említett két esetben érhet véget, hanem akkor is, ha az aktuálisan vizsgált tömbelem nagyobb a keresett *érték*-nél. Ilyenkor már biztos nem fogjuk a keresett *érték*-et megtalálni a tömbben, hiszen csak nála nagyobb elemeket vizsgálnánk a további bejárás során.

Ennek értelmében rendezett tömbök esetén a lineáris keresést az 5.1. algoritmussal valósíthatjuk meg.

5.1. Algoritmus Lineáris keresés rendezett tömbben

Bemenet: x – **T rendezett tömb**, n – **egész** (*tömb mérete*), *érték* – **T**; ahol **T** összehasonlítható
Kimenet: *van* – **logikai**, *idx* – **egész**

```
1:  függvény LineárisKeresésRendezettben(x : T rendezett tömb, n : egész, érték : T)
2:      i ← 1
3:      ciklus amíg (i ≤ n) ∧ (x[i] < érték)
4:          i ← i + 1
5:      ciklus vége
6:      van ← (i ≤ n) ∧ (x[i] = érték)
7:      ha van akkor
8:          idx ← i
9:          vissza (van, idx)
10:     különben
11:         vissza van
12:     elágazás vége
13: függvény vége
```

Felhasznált változók és függvények

- x: Növekvő módon rendezett tömb.
- n: Az x tömb mérete.
- *érték* : Keresett érték.
- *van*: Pontosan akkor ***igaz***, ha az *érték* benne van az x-ben.
- *idx*: Ha *érték* benne van az x tömbben, akkor (az első előfordulása) az *idx*-edik helyen található meg.

A 2.8. algoritmust két helyen kell módosítanunk. Egyrészt a 3. sorbeli ciklusfeltétel második tagját változtatjuk meg. Nem azt vizsgáljuk, hogy $x[i] \neq$ *érték* teljesül-e, hiszen teljesen felesleges akkor folytatni a bejárást, ha *érték* $> x[i]$. Ehelyett csak azt nézzük, hogy a keresett *érték* kisebb-e mint az aktuális $x[i]$ tömbelem. Ha kisebb, akkor még van értelme tovább keresni, ezért bennmaradunk a bejárást megvalósító ciklusban.

A ciklust követően megvizsgáljuk, hogy miért léptünk ki a ciklusból és ez alapján eldöntjük, hogy megtaláltuk-e a keresett *érték*-et. Ehhez korábban a 2.8. algoritmusban elég volt azt vizsgálni, hogy $i \leq n$ ***igaz***-e. Az most is igaz, hogy $i > n$ esetén nem találtuk meg a keresett *érték*-et, tehát a *van* kimenetnek ***hamis***-nak kell lennie. Viszont az $i \leq n$ feltétel úgy is lehet ***igaz***, ha nem találtuk meg a tömbben a keresett *érték*-et, ugyanis a bejárást félbehagyjuk akkor is, ha az aktuális tömbelem már nagyobb a keresett *érték*-nél.

Ezek miatt a ciklust követően azt vizsgáljuk, hogy nem haladtunk-e túl a tömb utolsó elemén is $(i \leq n)$, illetve, hogy az aktuális $x[i]$ tömbelem megegyezik-e a keresett *érték*-kel. Ha mindkét vizsgált feltétel ***igaz***, akkor találtuk meg a keresett *érték*-et az x tömbben. Fontos megjegyezni azt is, hogy az $i \leq n$ feltételt nem hagyhatjuk el a vizsgálatból. Ha például a keresett *érték* az utolsó tömbelemnél is nagyobb – azaz *érték* $> x[n]$ –, akkor a ciklusból az $i \leq n$ feltétel ***hamis***-sá válása miatt lépünk ki. Így viszont $i = n + 1$, tehát az x tömböt nem tudjuk ezzel az i-vel indexelni.

Futási idő elemzése. Először ismételjük át, hogy mit tudunk a lineáris keresés 2.8. algoritmusának futási idejéről. Ha a keresett *érték* benne van a tömbben, akkor a bejárást megvalósító ciklusban addig vagyunk bent, amíg meg nem találjuk az *érték*-et. A futási idő tehát ilyenkor attól függ, hogy hányadik

helyen van a keresett *érték* az x tömbben. Ha viszont nem található meg a keresett *érték* akkor a teljes tömböt be kell járnunk. Tehát a lineáris keresés 2.8. algoritmusa ilyenkor n elem vizsgálatát kívánja.

Mi a helyzet a módosított 5.1. algoritmussal? Ha a keresett *érték* bent van az x tömbben, akkor megtalálásához pontosan annyi vizsgálat szükséges, ahányadik helyen található. Tehát ebben az esetben nem módosul a futási idő. Viszont, ha nincs bent a keresett elem a tömbben, akkor ennek kiderítéséhez nem feltétlenül szükséges az egész tömböt végigjárnunk. Amint ugyanis olyan elemet találunk, amely már nagyobb a keresett *érték*-nél, akkor abbahagyjuk a bejárást. Tehát a futási idő lehet jóval kevesebb is a tömb n elemszámánál.

Összességében persze ilyenkor is az igaz, hogy legrosszabb, illetve átlagos esetben is a futási idő $O(n)$-es. ♣

Logaritmikus keresés

A logaritmikus keresésnél[1] feladatunk ugyanaz, mint a korábbi lineáris keresésnél volt. Azt keressük, hogy egy növekvő módon rendezett tömbben benne van-e a keresett érték, és ha benne van, akkor melyik helyen. A lineáris keresésnél láttuk, hogy rendezetlen és rendezett tömbben is meg tudjuk oldani a problémát. A logaritmikus keresés viszont csak rendezett tömbök esetén alkalmazható.

A logaritmikus keresésnél a következő ötlet alapján járunk el. Kezdetben vizsgáljuk az egész tömbünket, melynek középső elemét összehasonlítjuk a keresett értékkel. Ha egyezőséget találunk, akkor nincs további teendőnk. Ha viszont a középső elem nagyobb mint a keresett érték, akkor a keresést már csak a középső elem előtti elemek között kell folytatnunk. Hasonlóan, ha a középső elem kisebb a keresett értéknél, akkor a középső elem utáni elemek között kell a továbbiakban keresnünk. Mindkét esetben igaz, hogy a vizsgált tömbnek csak az egyik felében folytatjuk tovább a keresést. Emiatt ezt a keresési módszert felező keresésnek, illetve bináris keresésnek[2] is szokás nevezni. A keresés akkor ér véget, ha megtaláljuk valahol a keresett értéket, vagy ha „elfogy" a tömbünk.

A logaritmikus keresés megvalósítását az 5.2. algoritmusban mutatjuk be. Az algoritmus bemenetei és kimenetei megegyeznek a lineáris keresésnél már megismertekkel.

5.2. Algoritmus Logaritmikus keresés iteratív megvalósítása

Bemenet: x – **T rendezett tömb**, n – **egész**, *érték* – **T**; ahol **T** összehasonlítható
Kimenet: *van* – **logikai**, *idx* – **egész**

```
1:  függvény LogaritmikusKeresés(x : T rendezett tömb, n : egész, érték : T)
2:      bal ← 1
3:      jobb ← n
4:      center ← ⌊(bal+jobb)/2⌋
5:      ciklus amíg (bal ≤ jobb) ∧ (x[center] ≠ érték)
6:          ha x[center] > érték akkor
7:              jobb ← center − 1
8:          különben
9:              bal ← center + 1
10:         elágazás vége
11:         center ← ⌊(bal+jobb)/2⌋
12:     ciklus vége
13:     van ← (bal ≤ jobb)
14:     ha van akkor
15:         idx ← center
16:         vissza (van, idx)
17:     különben
18:         vissza van
19:     elágazás vége
20: függvény vége
```

Felhasznált változók és függvények

- x: Növekvő módon rendezett tömb.
- n: Az x tömb mérete.
- *érték*: Ezt az értéket keressük az x-ben.
- *van*: Pontosan akkor ***igaz***, ha az *érték* benne van az x-ben.
- *idx*: Ha *érték* benne van az x-ben, akkor az *idx*-edik helyen található meg. (Nem biztos, hogy csak ott.)
- *bal*: Az aktuálisan vizsgált résztömb bal szélének indexe.
- *jobb*: Az aktuálisan vizsgált résztömb jobb szélének indexe.
- *center*: Az aktuálisan vizsgált résztömb középső elemének indexe.

[1] Angolul: logarithmic search
[2] Angolul: binary search

Az algoritmusban nyomon kell követnünk, hogy az x tömbnek éppen melyik résztömbjében keresünk. Az aktuálisan vizsgált résztömb legkisebb indexét jelöli a *bal* változó, legnagyobb indexét pedig a *jobb* változó. Ezek kezdeti értékei rendre 1 és n lesznek (ld. 2-3. sorok). A vizsgált résztömb középső elemével valósítjuk meg a „felezést", ezért meghatározzuk a középső elem *center* indexét (ld. 4. sor).

Ezután belépünk az 5. és 12. sorok közötti elöltesztelős ciklusba. Ebben a ciklusban valósítjuk meg a tömb „felezését". A ciklusba akkor lépünk be, illetve addig maradunk bent (ld. 5. sor), amíg legalább egy eleme van a vizsgált résztömbnek ($bal \leq jobb$) és a résztömb középső eleme nem a keresett érték ($x[center] \neq érték$). A cikluson belül megvizsgáljuk, hogy a középső elem hogyan viszonyul a keresett értékhez. Ha $x[center] > érték$ (ld. 6. sor), akkor a továbbiakban csak a középső elem előttieket kell figyelembe vennünk. Emiatt a *jobb* értékét a *center* elé áthelyezzük (ld. 7. sor). Ha viszont $x[center] < érték$, akkor a keresett értéket a középsőt követő elemek között kell keresni. Így a *bal* értéke módosul a középsőt követő indexre (ld. 9. sor). Végül meghatározzuk, hogy hol található a lekicsinyített vizsgált résztömb középső eleme (ld. 11. sor).

A ciklusból kilépve megnézzük, hogy mi a kilépés oka. Ha $bal \leq jobb$, akkor biztos, hogy a ciklus bennmaradási feltételének második tagja lett ***hamis***, tehát megtaláltuk a keresett elemet. Ha viszont $bal > jobb$, akkor „elfogyott" a tömbünk, tehát nem találtuk meg a keresett elemet. Ennek megfelelően adunk értéket a *van* változónak (ld. 13. sor). Az algoritmus további soraiban már csak a függvény megfelelő visszatérési értékeit adjuk meg, majd véget ér az algoritmus futása.

Megjegyzés

Vegyük észre, hogy a 8. sorban nem vizsgáljuk meg, hogy a $x[center] < érték$ feltétel teljesül-e. Ugyanis a ciklusba csak akkor lépünk be, ha $x[center] \neq érték$, majd a 6. sorban megvizsgáljuk, hogy $x[center] > érték$ teljesül-e. Így a különben ágba pont akkor lépünk be, ha az $x[center] < érték$ feltétel ***igaz***. A feltételt viszont ténylegesen nem értékeljük ki, mert ez időigényes művelet lenne.

5.1. Példa. Az 5.1. ábrán adott 15 elemű x tömbben keressük meg a 12 értéket a logaritmikus keresés 5.2. algoritmusával.

A *bal* értéke kezdetben 1 lesz (2. sor), *jobb* pedig a tömb méretének megfelelően 15 (3. sor). A 4. sorban meghatározzuk a *center* index értékét, ami 8 lesz, ahogy az az 5.1a. ábrán is látható. Belépünk az 5. sorban kezdődő ciklusba. Mivel a nyolcadik elem nagyobb a keresett értéknél (6. sor), ezért *jobb* 7-re módosul (7. sor). Új *center* indexet határozunk meg (ld. 11. sor), ez most 4 lesz (ld. 5.1b. ábra).

Megvizsgáljuk a ciklus bennmaradási feltételét, ami jelenleg teljesül. A negyedik elem kisebb a keresett *érték*nél (8. sor), ezért a *bal* értékét módosítjuk (9. sor). A *center* értéke 6-ra módosul (ld. 5.1c. ábra). A ciklus bennmaradási feltétele viszont már nem teljesül, mert a hatodik elem megegyezik a keresett *érték*kel. Kilépve a ciklusból a *van* értéke ***igaz***-zá válik, *idx* pedig felveszi a megtalált elem indexét.

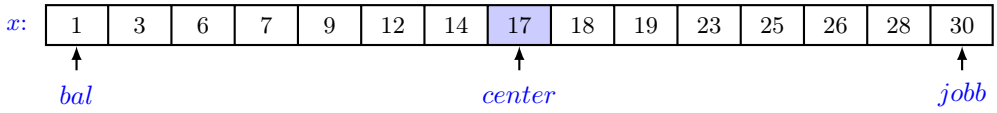
Látható, hogy háromszori vizsgálattal sikerült megtalálnunk a keresett elemet. ¶

5.2. Példa. Az 5.2. ábrán adott egy 15 elemű tömb. Az 5.2. algoritmus használatával keressük, hogy a 4 benne van-e és melyik helyen a rendezett tömbben.

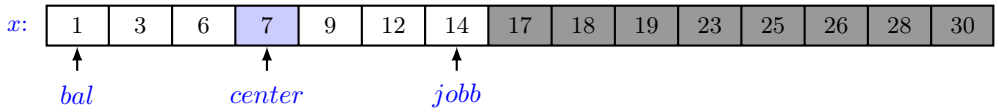
Kezdetben az egész tömböt vizsgáljuk, ezért $bal = 1$ és $jobb = 15$, *center* pedig felveszi a 8 értéket (ld. 5.2a. ábra). A vezérlés belép az 5. sorban kezdődő ciklusba. Mivel a nyolcadik tömbelem nagyobb a keresett értéknél, ezért *jobb* 7-re módosul. A *center* értéke 4-re módosul (ld. 5.2b. ábra). A bennmaradási feltétel ismét teljesül, ezért nem lép ki a vezérlés a ciklusból. A negyedik elem is nagyobb a keresett értéknél, ezért ismét a *jobb* index változik, a harmadik elemre fog mutatni, a *center* pedig a második elemre mutat (ld. 5.2c. ábra). A ciklus bennmaradási feltétele még most is teljesül. Mivel a második elem kisebb a keresett értéknél, ezért a *bal* indexváltozó módosul, a harmadik elemre mutat rá. Mivel a *bal* és a *jobb* indexváltozók értéke azonos, így a *center* értéke is meg fog velük egyezni (ld. 5.2d. ábra). A következő ciklusban mivel a harmadik elem nagyobb, mint a keresett érték, ezért a *jobb* változó értéke 2-re csökken (ld. 5.2e. ábra). Ekkor viszont a bennmaradási $bal \leq jobb$ feltétel ***hamis***-sá válik, ezért kilép a vezérlés a ciklusból, a *van* változó értéke pedig felveszi a ***hamis*** értéket. ¶

Futási idő elemzése. Vizsgáljuk meg, hogy milyen futási időt eredményez a logaritmikus keresés 5.2. algoritmusa. Vizsgálatunk arra irányul, hogy hányszor kell az algoritmusban található ciklust végrehajtani.

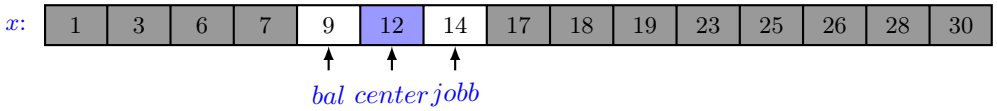
Legjobb esetben a ciklusfeltétel egyszer kerül kiértékelésre, hiszen ha a keresett érték pont a középső elemnél található, akkor rögtön megtaláljuk azt. Ilyenkor a ciklusba nem is lép be a vezérlés.



(a) Az egész tömböt vizsgáljuk, így $bal = 1$ és $jobb = 15$. A középső elem a nyolcadik, amelynek értéke nagyobb a keresett 12 értéknél.



(b) Már csak az első és a hetedik elem közötti résszel foglalkozunk. A középső elem ebben az esetben a negyedik, amelynek értéke kisebb a keresett 12 értéknél.



(c) Az ötödik és a hetedik elem közötti résztömböt vizsgáljuk, melynek középső eleme a hatodik. Ennek értéke megegyezik a keresett 12 értékkel, ezért véget ér az algoritmus.

5.1. ábra. Logaritmikus keresés. Az x tömbben keressük a 12 értéket. A harmadik vizsgálatnál megtaláljuk a keresett értéket a hatodik elemnél.

Legrosszabb esetben csak akkor találjuk meg a keresett értéket, amikor a vizsgált résztömb egy elemű, azaz a *bal* és a *jobb* értéke már megegyező, mégpedig mindkettő a keresett értéket tartalmazó tömbelemre mutat. Ilyen esetben hányszor kellett a ciklust végrehajtani? Mivel a ciklus minden egyes lefutásánál a vizsgált tömb mérete a felére – vagy még annál is kisebbre – csökken, ezért a fő kérdés az, hogy hányszor lehet az n elemű tömböt megfelezni. Ennek száma pedig $\lceil \log_2 n \rceil$. Tehát például egy 15 elemű tömb esetén legrosszabb esetben is csak 4 ciklus végrehajtására van szükség a keresett elem megtalálásához, feltéve, hogy a keresett elem benne van a tömbben.

Mi a helyzet akkor, ha nincs benne a keresett elem a tömbben? Ekkor az előbb tárgyalt legrosszabb esethez képest eggyel több ciklus futás szükséges, hiszen elő kell állnia annak a helyzetnek, hogy a vizsgált résztömb mérete már 1-nél is kisebb. Ezt jelzi az algoritmusban a $bal > jobb$ eset. Így a futási idő ilyenkor $\lceil 1 + \log_2 n \rceil$.

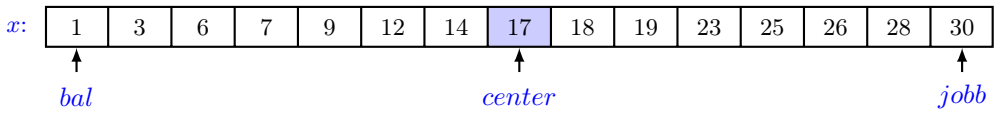
Összességében kijelenthető, hogy a logaritmikus keresés algoritmusának futási ideje legrosszabb esetben $\lceil 1 + \log_2 n \rceil$, átlagos esetben pedig $\lceil \log_2 n \rceil$. Tehát az algoritmus $O(\log n)$ futási idejű. Ezt könnyű megjegyezni, mert erről kapta az algoritmus a nevét is.

Vegyük észre, hogy egy 1000 elemű tömb esetén a logaritmikus keresés minden esetben legfeljebb 11 ciklus végrehajtással eredményre vezet. A korábban megismert lineáris keresésnél viszont legrosszabb esetben 1000, átlagos esetben pedig 500 vizsgálatra volt szükség. Jól látható, hogy a logaritmikus keresés jelentős futási idő javulást eredményez, de ne felejtsük el, hogy csak rendezett tömbök esetén használható. ♣

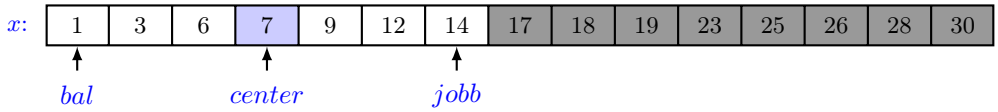
A fejezet további részében azt tekintjük át, hogy a logaritmikus keresés megismert algoritmusát, hogyan lehet rekurzív módon leírni. Természetesen az algoritmus ötlete nem módosul, csak az 5.2. algoritmusban lévő ciklust kell rekurzív függvényhívásokkal helyettesíteni. A rekurzív megvalósítás pszeudokódját az 5.3. algoritmus írja le.

A rekurzív algoritmus bemenete a rendezett x tömb, melyben keresünk egy konkrét *érték*-et. Bemenetként adjuk meg azt is, hogy az x tömb mely résztömbjében végezzük éppen a keresést. Ehhez a *bal* és a *jobb* indexek ismerete szükséges. A függvény első hívásakor a teljes tömbben történik a keresés, ezért $bal = 1$ és $jobb = n$, ahol n az x tömb mérete.

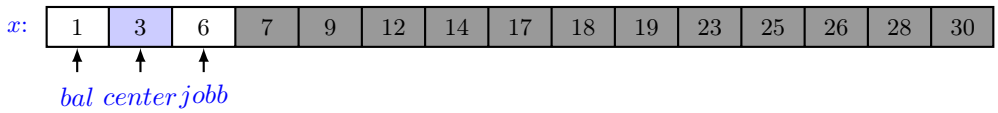
Az algoritmusban négy lényegesen különböző esetet kell kezelni. Elsőként megnézzük, hogy az aktuálisan vizsgált résztömb létezik-e. Ha $bal > jobb$ (ld. 2. sor), akkor nemlétező résztömbbel kéne dolgoznunk, ami azt jelenti, hogy ebben biztos nincsen benne a keresett *érték*. Ilyen esetben az algoritmust meg-



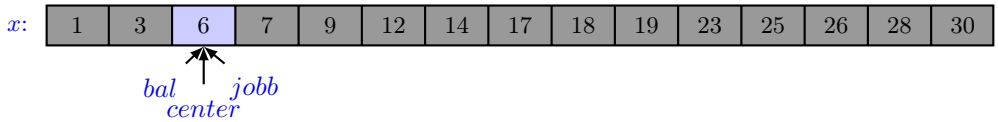
(a) Első lépésben a teljes tömböt vizsgáljuk, így $bal = 1$ és $jobb = 15$. A középső elem a nyolcadik, amelyik a keresett 4 értéknél nagyobb.



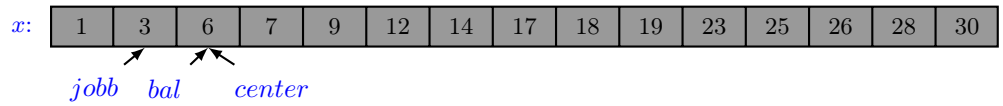
(b) A második iterációban már csak a tömb első felét vizsgáljuk, tehát $jobb = 7$. A középső elem a negyedik, amely nagyobb mint a keresett 4 érték.



(c) Harmadik lépésben az első és harmadik elemek közötti résztömbbel foglalkozunk. A középső elem most a második, amely kisebb a keresett 4 értéknél.



(d) A negyedik iterációban $bal = 3$ és $jobb = 3$, ezért a középső elem is a harmadik. Mivel a harmadik elem nagyobb a keresett 4 értéknél, ezért továbbfolytatjuk a keresést.



(e) Ötödik lépésben a $jobb = 2$, ami kisebb mint a bal. A keresett elemet nem találtuk meg a tömbben.

5.2. ábra. Logaritmikus keresés. Az x tömbben keressük a 4 értéket.

5.3. Algoritmus Logaritmikus keresés rekurzív megvalósítása

Bemenet: x – **T rendezett tömb**, *bal* – **egész**, *jobb* – **egész**, *érték* – **T**; ahol **T** összehasonlítható
Kimenet: Az *érték*-kel megegyező elem indexe, illetve ha nincs ilyen, akkor 0.

```
1: függvény LogaritmikusKeresésRekurzív(x : T rendezett tömb, bal : egész, jobb : egész, érték : T)
2:     ha bal > jobb akkor
3:         vissza 0
4:     különben
5:         center ← ⌊(bal+jobb)/2⌋
6:         ha x[center] = érték akkor
7:             vissza center
8:         különben
9:             ha x[center] > érték akkor
10:                vissza LogaritmikusKeresésRekurzív(x, bal, center − 1, érték)
11:            különben
12:                vissza LogaritmikusKeresésRekurzív(x, center + 1, jobb, érték)
13:            elágazás vége
14:        elágazás vége
15:    elágazás vége
16: függvény vége
```

Függvény hívása: LogaritmikusKeresésRekurzív(x, 1, n, *érték*)

Felhasznált változók és függvények

- x: Növekvő módon rendezett tömb.
- n: Az x tömb mérete.
- *bal*: Az aktuálisan vizsgált résztömb első elemének indexe.
- *jobb*: Az aktuálisan vizsgált résztömb utolsó elemének indexe.
- *center*: Az aktuálisan vizsgált résztömb középső elemének indexe.
- *érték*: A keresett érték.

valósító függvény a 0 értéket adja vissza (ld. 3. sor). Minden más esetben valóban létező résztömböt vizsgálunk, amit a 4. sorban kezdődő különben ág valósít meg.

A különben ágban először meghatározzuk a vizsgált résztömb középső elemének indexét (ld. 5. sor). Ezután megnézzük, hogy a középső $x[center]$ elem hogyan viszonyul a keresett *érték*-hez. Ez három lehetőséget jelent.

Amennyiben $x[center] = érték$ (ld. 6. sor), akkor megtaláltuk a keresett elemet, ezért ennek indexével ($center$) tér vissza a függvény (ld. 7. sor).

Ha $x[center] > érték$ (ld. 9. sor), akkor a keresést a bal és $center - 1$ indexek által meghatározott résztömbben kell folytatni. Emiatt rekurzívan meghívjuk a LogaritmikusKeresésRekurzív függvényt a megfelelő bemeneti paraméterekkel (ld. 10. sor). A hívott függvény által visszaadott érték vagy a megtalált elem indexe lesz, vagy 0, ami azt jelenti, hogy nem találtuk meg a keresett *érték*-et. Mindkét esetben a hívott függvény visszatérési értékét fogja az aktuális függvény is visszaadni az őt hívó környezet felé.

Utolsó lehetőség, amikor $x[center] < érték$ (ld. 11. sor). Ekkor is hasonlóan járunk el, mint az előző esetben, csak más résztömböt vizsgálunk (ld. 12. sor).

5.3. Példa. Nézzük meg, hogy az 5.1. példában már látott feladatot miként oldhatjuk meg rekurzív módon, az 5.3. algoritmus használatával. Az egyes lépéseket az 5.3. ábrán követhetjük nyomon.

A tizenöt elemű tömbben a 12-t keressük, ezért a külvilágból az alábbi hívás történik: LogaritmikusKeresésRekurzív(x, 1, 15, 12). Mivel a bal index kisebb mind a $jobb$ index, ezért az algoritmus 4. sorában kezdődő különben ágba lép a vezérlés. A középső elem $center$ indexe 8 lesz (ld. 5.3a. ábra).

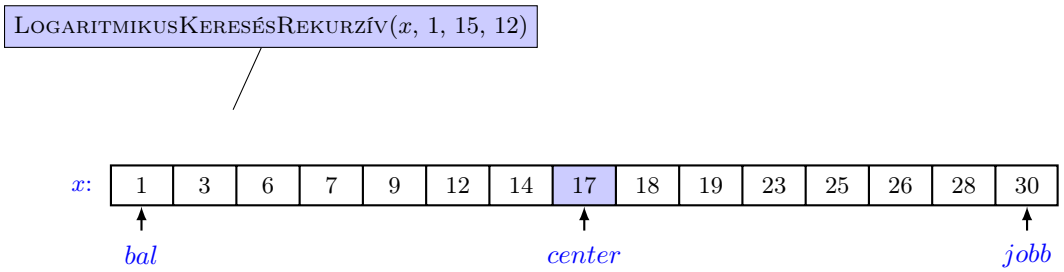
Mivel $x[center] > érték$ (ld. 6. sor), ezért rekurzívan meghívódik a LogaritmikusKeresésRekurzív függvény $bal = 1$ és $jobb = 7$ értékekkel. Mivel $bal < jobb$, ezért ismét meghatározásra kerül a középső elem indexe, ami 4 lesz (ld. 5.3b. ábra). Mivel $x[center] < érték$ (ld. 11. sor), ezért újabb rekurzív hívás következik. Ekkor $bal = 5$ és $jobb = 7$. Az új függvényben is meghatározzuk a középső elem indexét, ami 6 lesz (ld. 5.3c. ábra). A hatodik elem megegyezik a keresett 12 *érték*-kel (ld. 6. sor), ezért az aktuális függvény visszaadja a 6 értéket (ami a megtalált elem indexe), majd a futása véget ér (ld. 5.3d. ábra). A vezérlés visszakerül a másodikként elindult függvényhez. Itt nem történik más, mint a megkapott 6 értéket visszaadja ez a függvény is, majd a futása ennek is véget ér (ld. 5.3e. ábra). A vezérlés visszakerül a legkorábban hívott függvényhez. Ez a függvény is visszaadja a 6 értéket, majd ennek a futása is leáll (ld. 5.3f. ábra). Az algoritmus futása véget ér, eredményül megkaptuk a keresett elem indexét. ¶

5.4. Példa. A fejezet korábbi feladataiban már vizsgált x tömbben keressük a 4 értéket az 5.2. feladathoz hasonlóan. A keresésnél a rekurzív 5.3. algoritmust használjuk. Az 5.4. ábrán nyomon követhető, hogy miként alakul a rekurzív hívások sorozata.

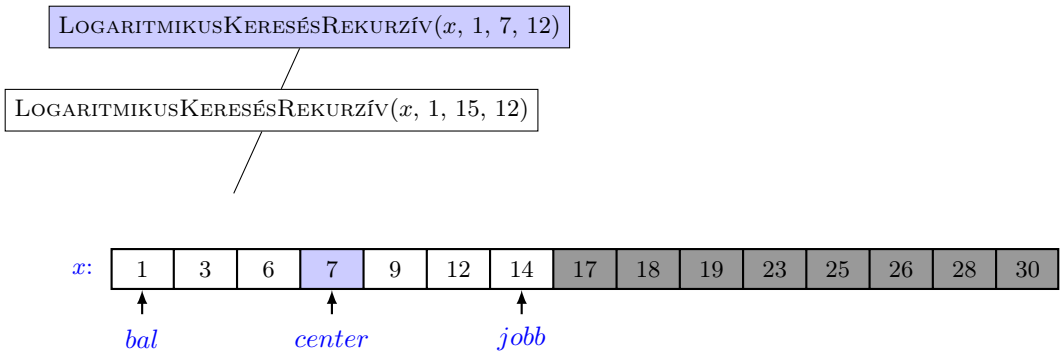
Kezdetben meghívásra kerül a LogaritmikusKeresésRekurzív függvény a $bal = 1$ és $jobb = 15$ paraméterekkel (ld. 5.4a. ábra). Ebben az esetben a középső elem nagyobb a keresett 4 értéknél, ezért rekurzív függvényhívás történik. A bal értéke nem változik a $jobb$ viszont 7-re módosul (ld. 5.4b. ábra). A középső elem ismét nagyobb a keresett értéknél, ezért újabb rekurzív hívás következik. Most a $jobb$ 3-ra csökken (ld. 5.4c. ábra). A középső elem kisebb a keresettnél, ezért bal növekszik 3-ra (ld. 5.4d. ábra). Mivel az aktuális középső elem nagyobb a keresett értéknél, ezért $jobb$ változik, mégpedig 2-re (ld. 5.4e. ábra).

Mivel $jobb < bal$, ezért a függvény a 0 értéket adja vissza (ld. 5.4f. ábra). Ezt követően minden korábban meghívott függvény a 0 értékekkel tér vissza, ahogy ez az 5.4g-5.4j. ábrákon is látható. ¶

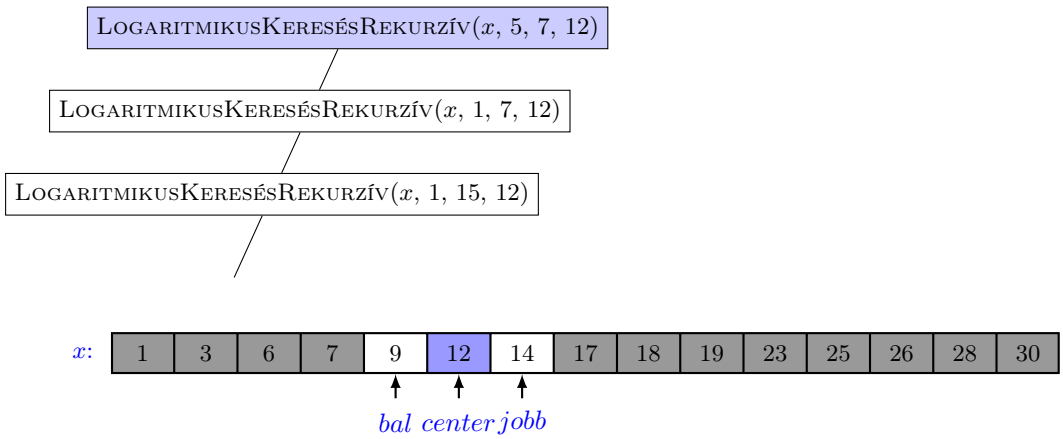
Futási idő elemzése. A rekurzív megvalósítás futási ideje nagyságrendileg megegyezik az iteratív esettel, mivel a függvényhívások száma azonos a ciklus végrehajtási számmal. ♣



(a) Az első függvényhívásnál $bal = 1$ és $jobb = 15$. A középső elem nagyobb a keresett értéknél.

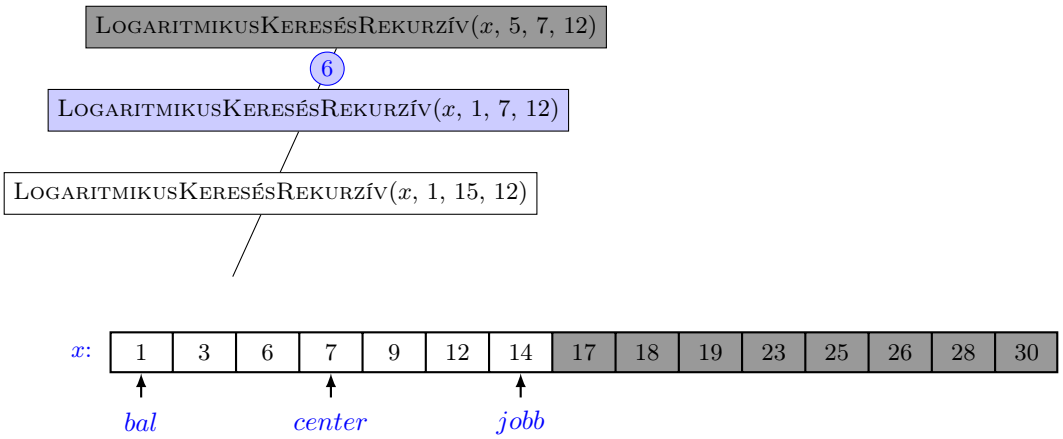


(b) Rekurzívan meghívásra kerül a függvény $bal = 1$ és $jobb = 7$ értékkel. A középső elem kisebb a keresett értéknél.

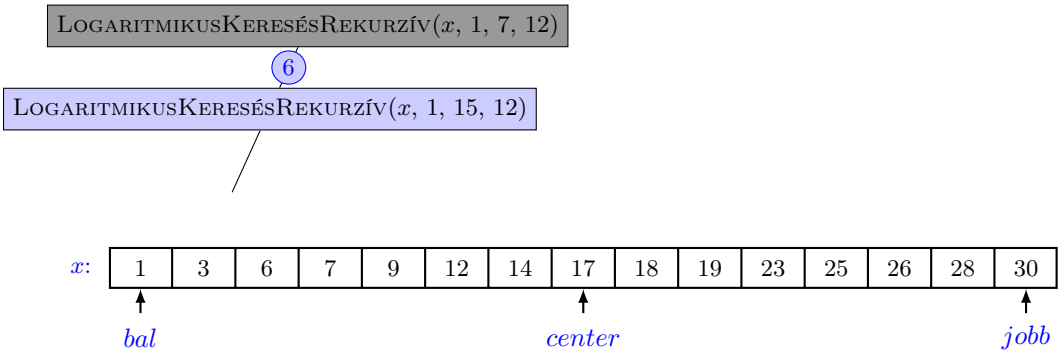


(c) Rekurzívan meghívásra kerül a függvény $bal = 5$ és $jobb = 7$ értékkel. A középső elem megegyezik a keresett értékkel.

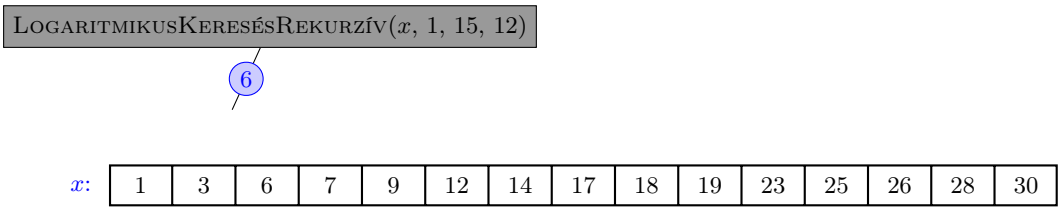
5.3. ábra. Logaritmikus keresés rekurzív megvalósítása. Az x tömbben keressük a 12 értéket.



(d) Az utoljára hívott függvény visszaadja a megtalált elem indexét (6), majd a futása véget ér.

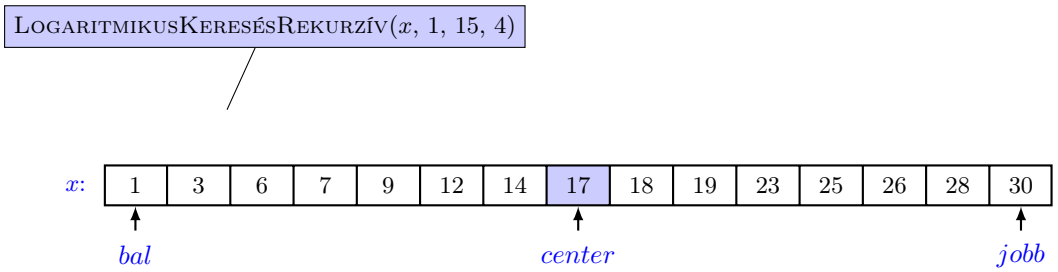


(e) Az aktuálisan futó függvény visszaadja a megtalált elem indexét (6), majd a futása véget ér.

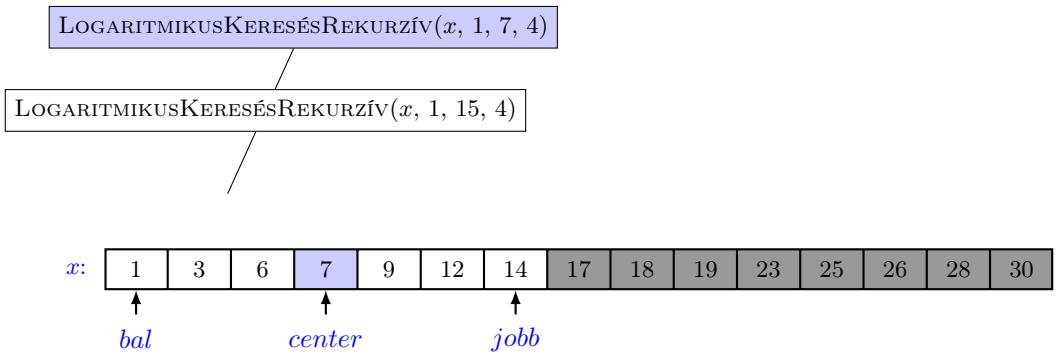


(f) A külvilágból hívott függvény visszaadja a megtalált elem indexét (6), majd a futása véget ér.

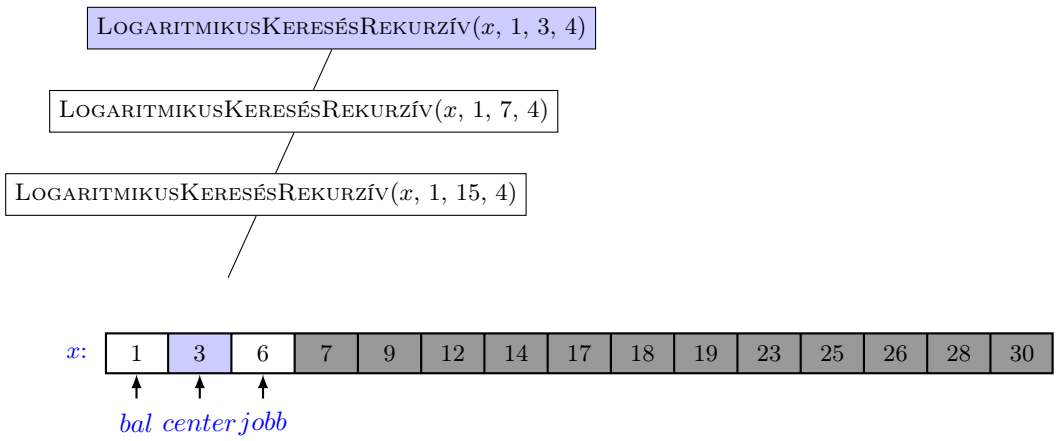
5.3. ábra. Logaritmikus keresés rekurzív megvalósítása (folyt.). Az x tömbben keressük a 12 értéket.



(a) A külvilágból a függvény hívása a $bal = 1$ és $jobb = 15$ paraméterekkel történik. A középső elem nagyobb a keresettnél.

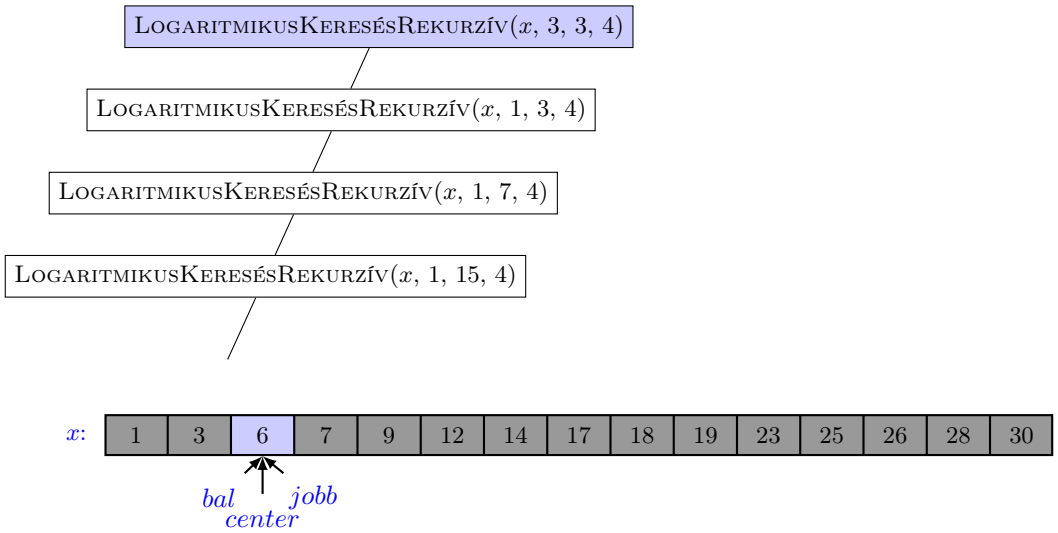


(b) Az első rekurzív hívásnál a *jobb* paraméter 7-re módosul. A középső elem most is nagyobb a keresettnél.

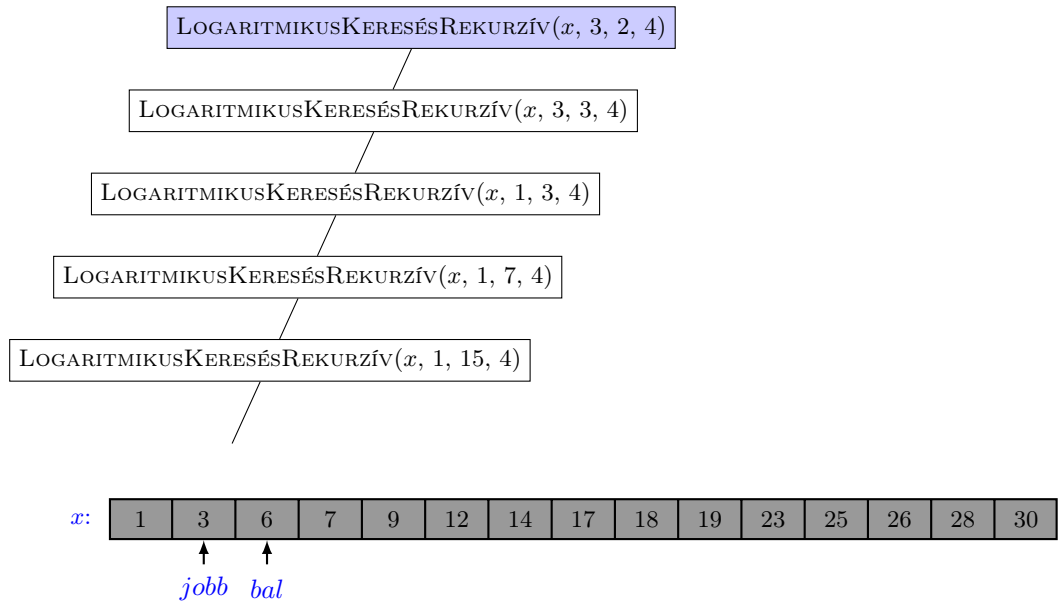


(c) A második rekurzív hívásnál a *jobb* 3-ra csökken. A középső elem kisebb a keresettnél.

5.4. ábra. Logaritmikus keresés rekurzív megvalósítása. Az x tömbben keressük a 4 értéket.

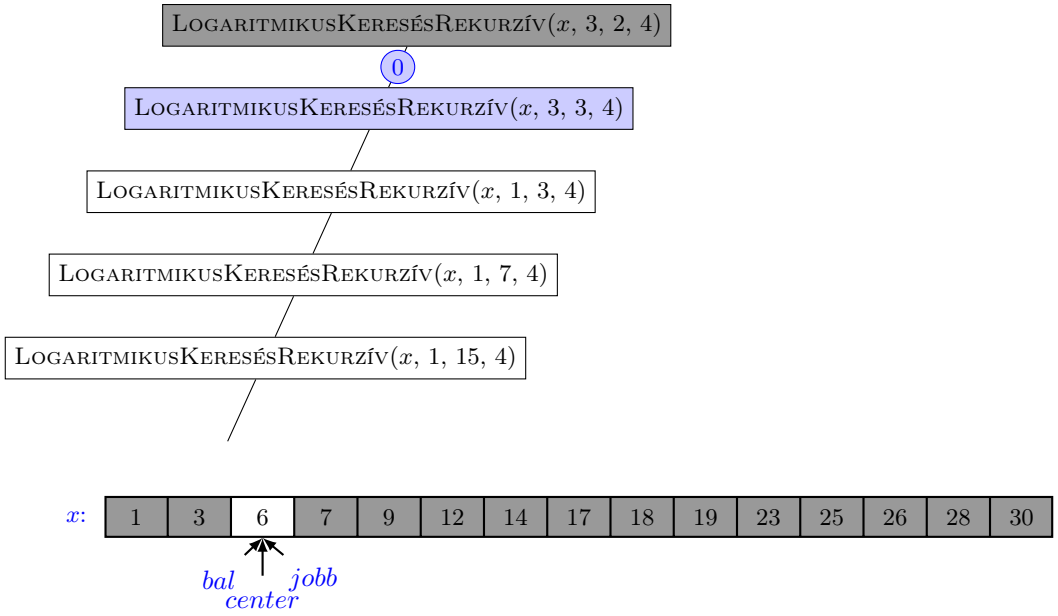


(d) A harmadik rekurzív hívásnál *bal* 3 lesz. A vizsgált résztömb egy eleművé válik, amely elem nagyobb a keresettnél.

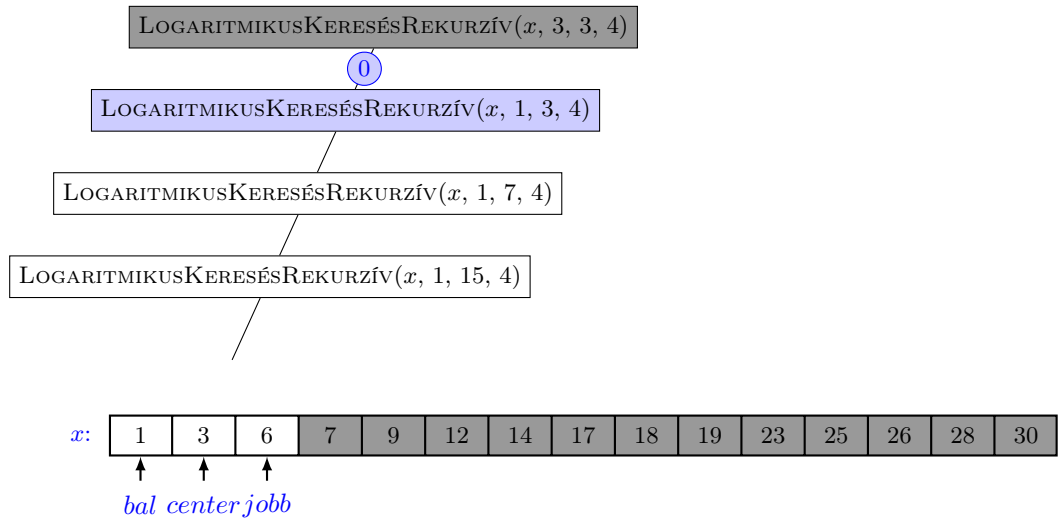


(e) A negyedik rekurzív hívásnál *jobb* 2 lesz. Mivel $jobb < bal$, nem lesz további rekurzív hívás.

5.4. ábra. Logaritmikus keresés rekurzív megvalósítása (folyt.). Az x tömbben keressük a 4 értéket.

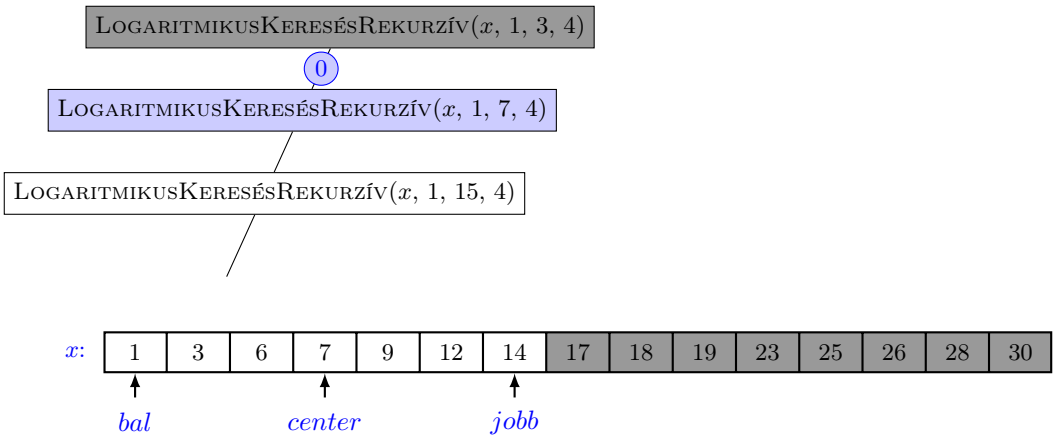


(f) Az utoljára hívott függvény visszatér a 0 értékkel.

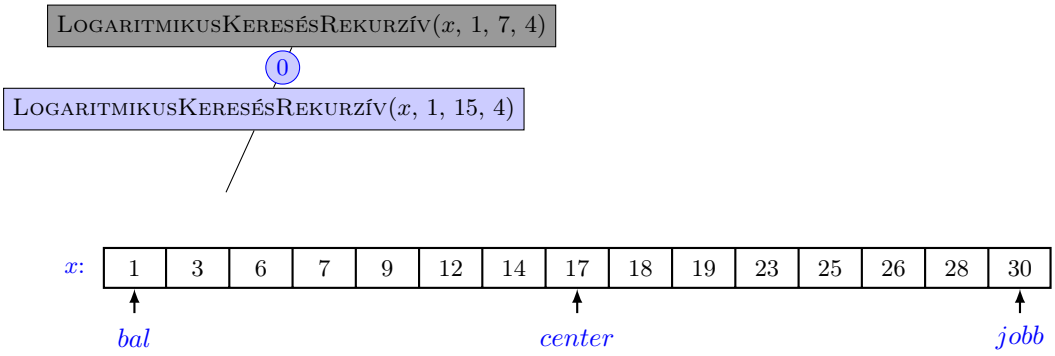


(g) Az utoljára hívott függvény visszatér a 0 értékkel.

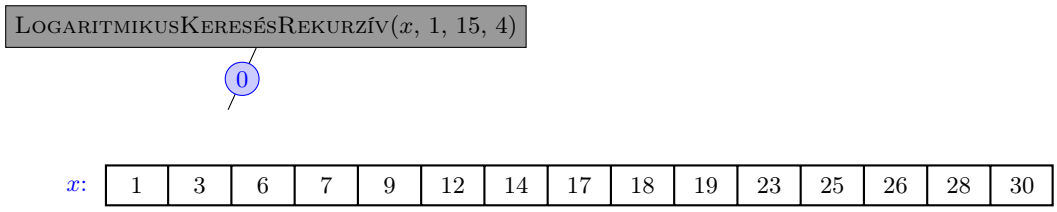
5.4. ábra. Logaritmikus keresés rekurzív megvalósítása (folyt.). Az x tömbben keressük a 4 értéket.



(h) Az utoljára hívott függvény visszatér a 0 értékkel.



(i) Az utoljára hívott függvény visszatér a 0 értékkel.



(j) A külvilágból hívott függvény visszatér a 0 értékkel.

5.4. ábra. Logaritmikus keresés rekurzív megvalósítása (folyt.). Az x tömbben keressük a 4 értéket.

Programozási tételek rendezett tömbökben

Ebben a fejezetben áttekintjük, hogy bizonyos programozási tételeket miként lehet futási idő szempontjából optimalizálni rendezett tömbök esetén. Az 5.1. fejezetben láttuk, hogy a keresés megvalósítása például a logaritmikus keresés használatával a leghatékonyabb.

A fejezetben ismertetésre kerülő programozási tételek mind a logaritmikus keresés ötletére építenek. Megmutatjuk, hogy miként kell eldöntés, kiválasztás, kiválogatás és megszámolás céljára módosítani a logaritmikus keresést. Ezen túl néhány esetben ismertetjük, hogy miként lehet nem csak egyetlen értéket, hanem egy adott intervallumba eső értéket keresni egy rendezett tömbben.

Eldöntés

Rendezett tömbök esetén az eldöntés programozási tétel arról ad információt, hogy a vizsgált x tömbben megtalálható-e egy konkrét *érték*. A problémát megoldó 5.4. algoritmus csak kis mértékben tér el a logaritmikus keresés 5.2. algoritmusától, ezért sorról sorra nem ismertetjük.

5.4. Algoritmus Eldöntés programozási tétel rendezett tömbben

Bemenet: x – **T rendezett tömb**, n – **egész** (*tömb mérete*), *érték* – **T**; ahol **T** összehasonlítható
Kimenet: *van* – **logikai**

```
1:  függvény EldöntésRendezettben(x : T rendezett tömb, n : egész, érték : T)
2:      bal ← 1
3:      jobb ← n
4:      center ← ⌊(bal+jobb)/2⌋
5:      ciklus amíg (bal ≤ jobb) ∧ (x[center] ≠ érték)
6:          ha x[center] > érték akkor
7:              jobb ← center − 1
8:          különben
9:              bal ← center + 1
10:         elágazás vége
11:         center ← ⌊(bal+jobb)/2⌋
12:     ciklus vége
13:     van ← (bal ≤ jobb)
14:     vissza van
15: függvény vége
```

Felhasznált változók és függvények

- x: Növekvő módon rendezett tömb, melynek elemei egymással összehasonlíthatóak.
- n: Az x tömb mérete.
- *érték*: A tömb elemeivel azonos típusú érték. Az eldöntés során azt vizsgáljuk, hogy ez az *érték* benne van-e az x tömbben.
- *van*: Logikai változó, amely pontosan akkor ***igaz***, ha *érték* megtalálható az x tömbben.
- *bal*: Az aktuálisan vizsgált résztömb első elemének indexe.
- *jobb*: Az aktuálisan vizsgált résztömb utolsó elemének indexe.
- *center*: Az aktuálisan vizsgált résztömb középső elemének indexe.

Nézzünk viszont egy bonyolultabb problémát. Azt szeretnénk megmondani, hogy egy rendezett tömbben van-e olyan érték, amely két szélsőérték ($alsóhatár$ és $felsőhatár$) közé esik. Ennek eldöntéséhez az 5.4. algoritmus kis mértékű módosítása szükséges. A módosított kódot az 5.5. algoritmusban adjuk meg.

Az algoritmus 5. sorbeli belépési és bennmaradási feltételét módosítani kell. Itt nem az a fontos, hogy a konkrét *érték*-et megtaláljuk, hanem, hogy a két szélsőérték közötti-e az elem. Hasonlóan a 6. sorban nem azt kell vizsgálnunk, hogy az aktuális középső elem a keresett *érték*-nél nagyobb-e, hanem azt, hogy a $felsőhatár$ értéknél nagyobb-e. A 8. sorba akkor kerül a vezérlés, ha az $x[center] < alsóhatár$.

5.5. Példa. Az 5.5. ábrán láthatunk egy példát, ahol egy 10 elemű x rendezett tömb esetén szeretnénk eldönteni, hogy van-e a tömbben $alsóhatár = 17$ és $felsőhatár = 23$ közötti érték. A feladatot a logaritmikus keresésen alapuló eldöntés módosított 5.5. algoritmusával oldjuk meg.

Kezdetben a teljes tömböt vizsgáljuk, így a $bal = 1$ és a $jobb = 10$. A tömb középső eleme az ötödik. Mivel az ötödik elem kisebb az $alsóhatár$-nál (ld. 5.5a. ábra), ezért *bal* értéke 6-ra módosul. Ekkor a középső elem a nyolcadik, ami viszont nagyobb a $felsőhatár$-nél (ld. 5.5b. ábra). Ekkor a *jobb* változik 7-re. A középső elem a hatodik, amely viszont benne van a keresett tartományban (ld. 5.5c. ábra). Mivel találtunk megfelelő elemet, ezért az algoritmus a $van =$ ***igaz*** értékkel tér vissza. ¶

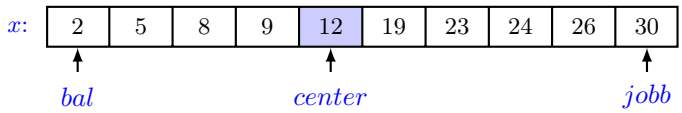
5.5. Algoritmus Módosított eldöntés programozási tétel rendezett tömbben

Bemenet: x – **T rendezett tömb**, n – **egész** (*tömb mérete*), $alsóhatár$ – **T**, $felsőhatár$ – **T**; ahol **T** összehasonlítható
Kimenet: van – **logikai**

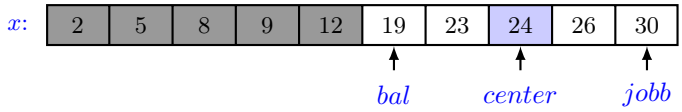
```
1:  függvény MódosítottEldöntésRendezettben(x : T rendezett tömb, n : egész, alsóhatár : T, felsőhatár : T)
2:      bal ← 1
3:      jobb ← n
4:      center ← ⌊(bal+jobb)/2⌋
5:      ciklus amíg (bal ≤ jobb) ∧ ¬((alsóhatár ≤ x[center]) ∧ (x[center] ≤ felsőhatár))
6:          ha x[center] > felsőhatár akkor
7:              jobb ← center − 1
8:          különben
9:              bal ← center + 1
10:         elágazás vége
11:         center ← ⌊(bal+jobb)/2⌋
12:     ciklus vége
13:     van ← (bal ≤ jobb)
14:     vissza van
15: függvény vége
```

Felhasznált változók és függvények

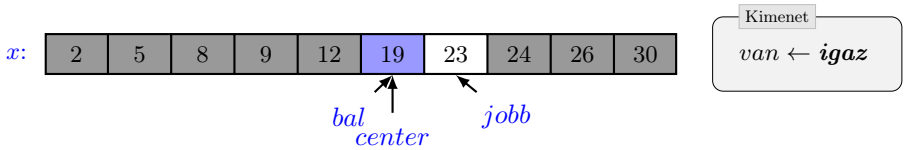
- x: Növekvő módon rendezett tömb, melynek elemei összehasonlíthatók.
- n: Az x tömb elemszáma.
- $alsóhatár$: Ennél az értéknél nem kisebb elemet keresünk az x tömbben.
- $felsőhatár$: Ennél az értéknél nem nagyobb elemet keresünk az x tömbben.
- van: Logikai változó, amely pontosan akkor ***igaz***, ha van az $alsóhatár$ értéknél nem kisebb és a $felsőhatár$ értéknél nem nagyobb elem az x tömbben.
- bal: Az aktuálisan vizsgált résztömb első elemének indexe.
- $jobb$: Az aktuálisan vizsgált résztömb utolsó elemének indexe.
- $center$: Az aktuálisan vizsgált résztömb középső elemének indexe.



(a) A középső elem kisebb, mint az $alsóhatár$.



(b) A középső elem nagyobb, mint a $felsőhatár$.



(c) A középső elem az $alsóhatár$ és a $felsőhatár$ között van, ezért a kimeneti érték ***igaz*** lesz.

5.5. ábra. Módosított eldöntés rendezett tömbben. Az x tömbben 17 és 23 közötti értéket keresünk.

Kiválasztás

A kiválasztás programozási tétel esetén tudjuk, hogy a rendezett tömbben van olyan elem, mely a keresett *érték*-kel megegyezik. A kérdés csak az, hogy melyik ez az elem. Ezt a problémát is a logaritmikus keresés 5.2. algoritmusának módosításával oldhatjuk meg. A módosított kódot az 5.6. algoritmusban mutatjuk be.

5.6. Algoritmus Kiválasztás programozási tétel rendezettben

Bemenet: x – **T rendezett tömb**, n – **egész**, *érték* – **T**; ahol **T** összehasonlítható
Kimenet: idx – **egész**

```
1:  függvény KiválasztásRendezettben(x : T rendezett tömb, n : egész, érték : T)
2:      bal ← 1
3:      jobb ← n
4:      center ← ⌊(bal+jobb)/2⌋
5:      ciklus amíg x[center] ≠ érték
6:          ha x[center] > érték akkor
7:              jobb ← center − 1
8:          különben
9:              bal ← center + 1
10:         elágazás vége
11:         center ← ⌊(bal+jobb)/2⌋
12:     ciklus vége
13:     idx ← center
14:     vissza idx
15: függvény vége
```

Felhasznált változók és függvények

- x: Növekvő módon rendezett tömb, melynek elemei összehasonlíthatóak.
- n: Az x tömb elemeinek száma.
- *érték*: Az x tömb elemivel azonos típusú érték. Annak az elemnek az indexét keressük, amely az *érték*-kel megegyezik. (Tudjuk, hogy van ilyen elem.)
- idx: Az x tömb azon elemének indexe, amely megegyezik az *érték*-kel.
- *bal*: Az aktuálisan vizsgált résztömb első elemének indexe.
- *jobb*: Az aktuálisan vizsgált résztömb utolsó elemének indexe.
- *center*: Az aktuálisan vizsgált résztömb középső elemének indexe.

Kiválogatás

A most tárgyalásra kerülő kiválogatásnál módosítunk a 2.2.2. fejezetben megismert kiválogatás programozás tétel feladatán. Nem azt fogjuk vizsgálni, hogy melyek azok a tömbbeli elemek, amelyek egy adott tulajdonságnak megfelelőek, hanem azt nézzük, hogy melyek azok az elemek, amelyek egy konkrét *érték*-kel megegyeznek. Ilyen elem lehet, hogy nincs is a tömbben, lehet, hogy csak egy van, de lehet, hogy több is van. Ha több azonos értékű elem is van a tömbben, akkor azok a rendezettség miatt biztosan egymás közvetlen szomszédságában helyezkednek el. Természetesen nem az most a célunk, hogy ugyanazt az értéket akár többször is kiválogassuk egy új tömbbe, hanem azt akarjuk megadni, hogy az eredeti tömbben melyik két index között található meg a keresett *érték*.

A probléma hatékony megoldását az 5.7. algoritmusban adjuk meg. Az algoritmus bemenetként kapja meg a vizsgált tömböt, melynek a méretét is ismerjük, valamint a keresett értéket. Kimenetként információt kapunk arról, hogy a keresett érték benne van-e a tömbben, és ha igen, akkor mely indexek között.

Az algoritmus 13. soráig ugyanazt tesszük, mint a logaritmikus keresés 5.2. algoritmusában, azaz vizsgáljuk, hogy a keresett *érték* megtalálható-e az x tömbben. Amennyiben megtalálható, akkor a *van* változó ***igaz*** értékű lesz. Ilyenkor meg kell vizsgálni, hogy a megtalált elem bal, illetve jobb szomszédai is megegyeznek-e az *érték*-kel. Ezt a vizsgálatot valósítja meg a 15. és 22. sorok közötti rész. Az algoritmus találat esetén visszatér a *van* változó megfelelő értékével, valamint a meghatározott indexhatárok értékeivel. Ha nincs találat, akkor csak erről szolgáltat információt.

5.6. Példa. Az 5.6. ábrán látható rendezett tömb esetén szeretnénk meghatározni, hogy van-e 3-as érték a tömbben, és ha van, akkor mely két index közötti tartományban. A feladatot a kiválogatás 5.7. algoritmusával oldjuk meg.

Kezdetben a logaritmikus keresésnél megismert keresési módszert alkalmazva egy darab 3-ast szeretnénk megtalálni. Szerencsére ezt rögtön az első feltétel vizsgálatnál megleljük, mert a középső elem 3-as értékű (ld. 5.6a. ábra). Ezt követően kell a megtalált elem bal és jobb szomszédjait végigjárva megkeresni az összes 3-as érték előfordulásait. Ennek érdekében a *bal* indexet egyenlővé tesszük a megtalált elem *center* indexével. Megvizsgáljuk a *bal*-t megelőző elem értékét (ld. 5.6b. ábra). Mivel ez az érték is 3-as, ezért *bal* értékét eggyel csökkentjük. Ugyanezt tesszük az 5.6c. és 5.6d. ábrán látható esetekben is. Amikor a *bal* értéke 3-ra csökken, akkor az őt közvetlenül megelőző elem már nem 3-as (ld. 5.6e. ábra), ezért a *bal* indexszel folytatott bejárást befejezzük. Ezután megvizsgáljuk a *center*-edik elemet követő elemeket. Ennek érdekében a *jobb* indexet tesszük egyenlővé a *center*-rel. Mivel a *jobb*-adik elemet követő elem is 3-as (ld. 5.6f. ábra), ezért *jobb* értékét eggyel növeljük. Újabb vizsgálat következik, de most a *jobb* indexű elemet követő elem már nem 3-as (ld. 5.6g. ábra), ezért ezt a bejárást is befejezzük. Végeredményül megtaláltuk azt a tartományt, amelyben csak 3-asok vannak. ¶

Természetesen a kiválogatás esetén is módosíthatjuk a keresésünket oly módon, ahogy ezt az eldöntésnél tettük. Itt is kereshetünk olyan elemeket, amelyek egy *alsóhatár* és egy *felsőhatár* közé esnek. A kiválogatás ezen esetének megvalósítását az 5.8. algoritmusban mutatjuk be.

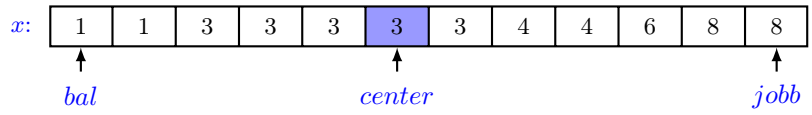
5.7. Algoritmus Kiválogatás programozási tétel rendezett tömbben

Bemenet: x – **T rendezett tömb**, n – **egész**, *érték* – **T**; ahol **T** összehasonlítható
Kimenet: *van* – **logikai**, *bal* – **egész**, *jobb* – **egész**

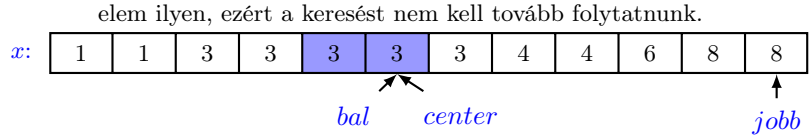
```
1:  függvény KiválogatásRendezettben(x : T rendezett tömb, n : egész, érték : T)
2:      bal ← 1
3:      jobb ← n
4:      center ← ⌊(bal+jobb)/2⌋
5:      ciklus amíg (bal ≤ jobb) ∧ (x[center] ≠ érték)
6:          ha x[center] > érték akkor
7:              jobb ← center − 1
8:          különben
9:              bal ← center + 1
10:         elágazás vége
11:         center ← ⌊(bal+jobb)/2⌋
12:     ciklus vége
13:     van ← (bal ≤ jobb)
14:     ha van akkor
15:         bal ← center
16:         ciklus amíg (bal > 1) ∧ (x[bal − 1] = érték)
17:             bal ← bal − 1
18:         ciklus vége
19:         jobb ← center
20:         ciklus amíg (jobb < n) ∧ (x[jobb + 1] = érték)
21:             jobb ← jobb + 1
22:         ciklus vége
23:         vissza (van, bal, jobb)
24:     különben
25:         vissza van
26:     elágazás vége
27: függvény vége
```

Felhasznált változók és függvények

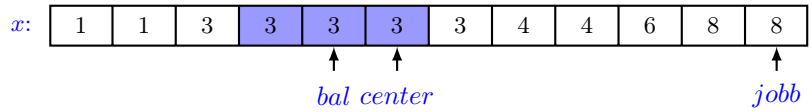
- x: Növekvő módon rendezett tömb, melynek elemei összehasonlíthatóak.
- n: Az x tömb elemeinek száma.
- *érték*: Az x tömb elemeivel azonos típusú érték. Azt keressük, hogy van-e ilyen elem a tömbben, és ha van, akkor mely helyeken található.
- *van*: Logikai változó, amely pontosan akkor ***igaz***, ha van legalább egy *érték*-kel azonos elem az x tömbben.
- *bal*: Az aktuálisan vizsgált résztömb első elemének indexe. Amennyiben *van* értéke ***igaz***, akkor az algoritmus végén az első *érték*-kel egyező tömbbeli elem indexe.
- *jobb*: Az aktuálisan vizsgált résztömb utolsó elemének indexe. Amennyiben *van* értéke ***igaz***, akkor az algoritmus végén az utolsó *érték*-kel egyező tömbbeli elem indexe.
- *center*: Az aktuálisan vizsgált résztömb középső elemének indexe.



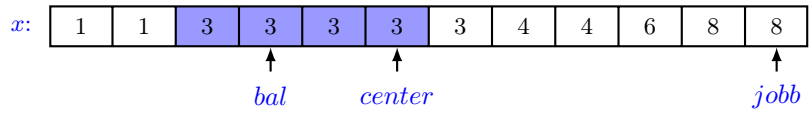
(a) Első lépésben egy darab 3-ast kell találnunk a tömbben. Mivel a középső elem ilyen, ezért a keresést nem kell tovább folytatnunk.



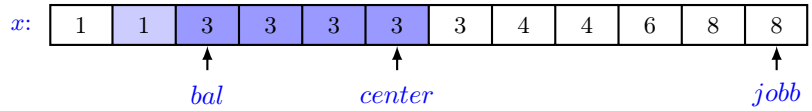
(b) A *bal* indexet egyenlővé tesszük a *center*-rel, majd megvizsgáljuk a bal szomszédját.



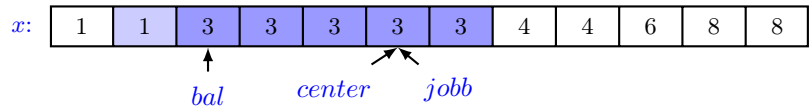
(c) A *bal* indexet eggyel csökkentettük, majd ismét vizsgáljuk a bal szomszédját.



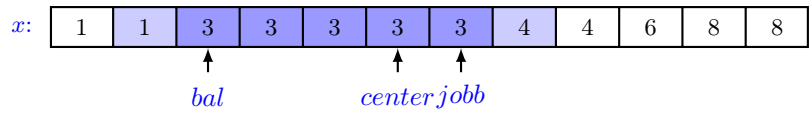
(d) A *bal* indexet eggyel csökkentettük, majd ismét vizsgáljuk a bal szomszédját.



(e) A *bal* indexet ismét csökkentettük, viszont a bal szomszéd már nem 3-as.



(f) A *jobb* indexet egyenlővé tesszük a *center*-rel, majd megvizsgáljuk a jobb szomszédját.



(g) A *jobb* indexet eggyel növeltük, viszont a jobb szomszédja nem 3-as. Visszatérünk a $van = \textbf{\textit{igaz}}$, $bal = 3$ és $jobb = 7$ értékekkel.

5.6. ábra. Kiválogatás rendezett tömbök esetén. A 12 elemű x tömbben keressük azt a tartományt, amelyben 3-asok találhatók.

5.8. Algoritmus Módosított kiválogatás programozási tétel rendezett tömbben

Bemenet: $x-$**T rendezett tömb**, $n-$**egész**, $alsóhatár-$**T**, $felsőhatár-$**T**; ahol **T** összehasonlítható
Kimenet: $van-$**logikai**, $bal-$**egész**, $jobb-$**egész**

```
1: függvény MódosítottKiválogatásRendezettben(x : T rendezett tömb, n : egész, alsóhatár
   : T, felsőhatár : T)
2:     bal ← 1
3:     jobb ← n
4:     center ← ⌊(bal+jobb)/2⌋
5:     ciklus amíg (bal ≤ jobb) ∧ ¬((alsóhatár ≤ x[center]) ∧ (x[center] ≤ felsőhatár))
6:         ha x[center] > felsőhatár akkor
7:             jobb ← center − 1
8:         különben
9:             bal ← center + 1
10:        elágazás vége
11:        center ← ⌊(bal+jobb)/2⌋
12:    ciklus vége
13:    van ← (bal ≤ jobb)
14:    ha van akkor
15:        bal ← center
16:        ciklus amíg (bal > 1) ∧ (x[bal − 1] ≥ alsóhatár)
17:            bal ← bal − 1
18:        ciklus vége
19:        jobb ← center
20:        ciklus amíg (jobb < n) ∧ (x[jobb + 1] ≤ felsőhatár)
21:            jobb ← jobb + 1
22:        ciklus vége
23:        vissza (van, bal, jobb)
24:    különben
25:        vissza van
26:    elágazás vége
27: függvény vége
```

Felhasznált változók és függvények

- x: Növekvő módon rendezett tömb, melynek elemei összehasonlíthatóak.
- n: Az x tömb elemeinek száma.
- $alsóhatár$ és $felsőhatár$: Az x tömb elemeivel azonos típusú értékek. Azt keressük, hogy van-e $alsóhatár$ és $felsőhatár$ közötti elem a tömbben. Ha van, akkor mely helyeken találhatók ilyenek.
- van: Logikai változó, amely pontosan akkor ***igaz***, ha van legalább egy $alsóhatár$ és $felsőhatár$ közötti elem az x tömbben.
- bal: Az aktuálisan vizsgált résztömb első elemének indexe. Amennyiben van értéke ***igaz***, akkor az algoritmus végén az első $alsóhatár$-nál nem kisebb első tömbbeli elem indexe.
- $jobb$: Az aktuálisan vizsgált résztömb utolsó elemének indexe. Amennyiben van értéke ***igaz***, akkor az algoritmus végén az utolsó $felsőhatár$-nál nem nagyobb utolsó tömbbeli elem indexe.
- $center$: Az aktuálisan vizsgált résztömb középső elemének indexe.

Megszámlálás

Rendezett tömbök esetén a megszámlálás során azt szeretnénk meghatározni, hogy egy konkrét érték hányszor fordul elő a rendezett tömbben. Mivel az azonos értékek egymás közvetlen szomszédságában találhatók meg, ezért azt kell meghatároznunk, hogy melyik két határindex között találhatók a keresett értékű elemek, majd ebből egyszerűen kiszámítható az elemek darabszáma.

A megszámlálás megoldását az 5.9. algoritmusban ismertetjük, amely elvében nagyon hasonló az előző algoritmusokhoz.

5.9. Algoritmus Megszámlálás programozási tétel rendezett tömbben

Bemenet: x – **T rendezett tömb**, n – **egész**, *érték* – **T**; ahol **T** összehasonlítható
Kimenet: db – **egész**

```
1:  függvény MegszámlálásRendezettben(x : T rendezett tömb, n : egész, érték : T)
2:      bal ← 1
3:      jobb ← n
4:      center ← ⌊(bal+jobb)/2⌋
5:      ciklus amíg (bal ≤ jobb) ∧ (x[center] ≠ érték)
6:          ha x[center] > érték akkor
7:              jobb ← center − 1
8:          különben
9:              bal ← center + 1
10:         elágazás vége
11:         center ← ⌊(bal+jobb)/2⌋
12:     ciklus vége
13:     ha bal ≤ jobb akkor
14:         bal ← center
15:         ciklus amíg (bal > 1) ∧ (x[bal − 1] = érték)
16:             bal ← bal − 1
17:         ciklus vége
18:         jobb ← center
19:         ciklus amíg (jobb < n) ∧ (x[jobb + 1] = érték)
20:             jobb ← jobb + 1
21:         ciklus vége
22:         db ← jobb − bal + 1
23:     különben
24:         db ← 0
25:     elágazás vége
26:     vissza db
27: függvény vége
```

Felhasznált változók és függvények

- x: Növekvő módon rendezett tömb, melynek elemei összehasonlíthatóak.
- n: Az x tömb elemeinek száma.
- *érték*: Az x tömb elemeivel azonos típusú érték. Azt keressük, hogy hány darab ilyen elem van a tömbben.
- db: Az *érték*-kel egyező elemek száma az x tömbben.
- bal: Az aktuálisan vizsgált résztömb első elemének indexe. Amennyiben van az *érték*-kel egyező elem a tömbben, akkor az algoritmus végén az első ilyen tömbbeli elem indexe.
- $jobb$: Az aktuálisan vizsgált résztömb utolsó elemének indexe. Amennyiben van az *érték*-kel egyező elem a tömbben, akkor az algoritmus végén az utolsó ilyen tömbbeli elem indexe.
- $center$: Az aktuálisan vizsgált résztömb középső elemének indexe.

Halmazok

A matematikából tudjuk, hogy a halmaz egy alapfogalom, amelyet nem lehet definiálni. Viszont az eldönthető, hogy valami eleme-e egy halmaznak vagy sem. Tudjuk azt is a halmazokról, hogy ha valami benne van egy halmazban, akkor nem lehet benne kétszer vagy többször is, tehát egy halmazban nem létezik ismétlődés.

Vizsgáljuk meg, hogyan lehet a matematikából ismert halmazokat az algoritmusok és adatszerkezetek világában reprezentálni úgy, hogy hatékonyan lehessen megvalósítani a halmazokon értelmezett műveleteket. A halmazokat olyan növekvő módon rendezett tömbökként ábrázoljuk – bár a matematikai értelemben vett halmazelemek között nincs rendezettség –, amelyekben nem fordul elő ismétlődés. Természetesen ez a reprezentáció csak olyan halmazok esetén alkalmazható, amikor az elemek között értelmezhető a < reláció.

A halmazokkal kapcsolatban az alábbi problémákat tekintjük át részletesen. Megnézzük, hogy miként lehet egy növekvő módon rendezett tömbről eldönteni, hogy az halmaznak tekinthető-e (ld. 5.3.1. alfejezet). Algoritmust adunk arra, hogy egy rendezett tömb milyen módon alakítható át halmazzá (ld. 5.3.2. alfejezet). Megadjuk az elemtartalmazás (ld. 5.3.3. alfejezet) és a részhalmaz tulajdonság (ld. 5.3.4. alfejezet) eldöntésének hatékony algoritmusait. A fejezet lezárásaként ismertetjük a halmazműveletek megvalósítását az unió (ld. 5.3.5. alfejezet), metszet (ld. 5.3.6. alfejezet), különbség (ld. 5.3.7. alfejezet) és szimmetrikus differencia (ld. 5.3.8. alfejezet) esetében. Mind a négy halmazművelet a korábban már megismert összefuttatás programozási tétel (ld. 2.2.6. fejezet) elve alapján került kidolgozásra.

Halmaztulajdonság vizsgálata

Ha egy tömbről azt szeretnénk eldönteni, hogy halmaz-e, akkor két tulajdonságot kell megvizsgálnunk. Az első a rendezettség, a második pedig az ismétlődés mentesség. Az eldöntés programozási tétel tárgyalásánál (ld. 2.1.2. fejezet) már megmutattuk a rendezettség vizsgálatát megvalósító algoritmust (ld. 2.5. algoritmus), így most csak az ismétlődésnélküliség vizsgálatával foglalkozunk. Ennek megvalósítása is az eldöntés tétel módosításával valósítható meg hatékonyan, amit az 5.10. algoritmusban mutatunk be.

5.10. Algoritmus Halmaztulajdonság vizsgálata

Bemenet: x – **T rendezett tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: l – **logikai**

```
1: függvény HalmazE(x : T rendezett tömb, n : egész)
2:     i ← 2
3:     ciklus amíg (i ≤ n) ∧ (x[i] ≠ x[i − 1])
4:         i ← i + 1
5:     ciklus vége
6:     l ← (i > n)
7:     vissza l
8: függvény vége
```

Felhasznált változók és függvények

- x: Rendezett tömb.
- n: Az x tömb mérete. Tudjuk, hogy $n \geq 2$.
- l: Logikai változó. Pontosan akkor ***igaz***értékű, ha az x tömbben nincs ismétlődés.

Az algoritmus bemenete az x növekvő módon rendezett tömb, melynek ismerjük a méretét is (n). A kimenet egy l logikai változó, amely pontosan akkor ***igaz*** értékű, ha a rendezett tömbben nincsenek ismétlődő elemek.

A rendezettség miatt az esetleges ismétlődések csak közvetlen szomszédok esetén fordulhatnak elő, ezért az eldöntés programozási tételt úgy kell módosítanunk, hogy a szomszédok egyenlőségét vizsgáljuk (ld. 3. sor). Az x tömb akkor tekinthető halmaznak, hogy nincs benne ismétlődés, ami akkor áll elő, ha az összes szomszédos elemet már megvizsgáltuk, és így az i változó értéke már meghaladja a tömb n elemszámát (ld. 6. sor).

Halmaz létrehozása

Tudjuk már, hogy miként lehet egy rendezett tömbről eldönteni, hogy van-e benne ismétlődő elem, azaz tekinthető-e halmaznak. Nézzük meg, hogyan lehet egy ismétlődő elemeket tartalmazó rendezett tömbből az ismétlődéseket kiszűrni. Általános, nem rendezett tömbök esetén megoldottuk már ezt a problémát a 2.21. algoritmus használatával. Viszont, ha kihasználjuk a rendezettséget, akkor sokkal hatékonyabb megoldást is adhatunk, ahogy ez az 5.11. algoritmusban látható.

Az algoritmus bemenete az x növekvő módon rendezett tömb, melynek ismert az n elemszáma is. Kimenetként az ismétlődés nélküli a halmazt kapjuk, melyben a releváns elemek száma db.

5.11. Algoritmus Halmaz létrehozása

Bemenet: x – **T rendezett tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: a – **T halmaz**, db – **egész**

```
1:  függvény HalmazLétrehozás(x : T rendezett tömb, n : egész)
2:      a ← Létrehoz(T)[n]
3:      db ← 1
4:      a[db] ← x[1]
5:      ciklus i ← 2-től n-ig
6:          ha x[i] ≠ a[db] akkor
7:              db ← db + 1
8:              a[db] ← x[i]
9:          elágazás vége
10:     ciklus vége
11:     vissza (a, db)
12: függvény vége
```

Felhasznált változók és függvények

- x: Rendezett tömb.
- n: Az x tömb mérete.
- a: Halmaz, mely az x tömb elemeit tartalmazza ismétlődés nélkül.
- db: Az a halmazban tárolt elemek száma.
- Létrehoz(**T**)[n]: Utasítás, mely létrehoz egy n elemű **T** típusú tömböt.

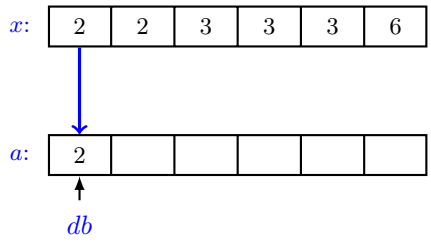
Az algoritmus működésének lényege az lesz, hogy az x tömböt az elejétől indulva bejárjuk és megnézzük, hogy az x aktuális eleme megegyezik-e az a utolsó elemével. Ha megegyezik, akkor nem másoljuk át a halmazba, ha nem akkor viszont betesszük a halmaz végére.

Első lépésként létre kell hozni a kimeneti a halmazt. Mivel nem tudjuk, hogy az algoritmus végén pontosan hány elem lesz benne, ezért az x bemeneti tömbbel azonos méretű tömbként inicializáljuk (ld. 2. sor). Mivel az x tömb első eleme még biztos nincs az a halmazban – hiszen a halmaz üres –, ezért az első elemet feltételvizsgálat nélkül átmásoljuk és a halmaz aktuális db elemszámát 1-re állítjuk (ld. 3-4. sor). Ezt követően bejárjuk az x tömb többi elemét, amit az 5. és 10. sorok közötti számlálós ciklussal valósítunk meg. A cikluson belül a 6. sorban megvizsgáljuk, hogy a tömb aktuális $x[i]$ eleme különbözik-e a halmaz aktuálisan utolsó $a[db]$ elemétől. Ha egyezőség van, akkor semmit nem kell tennünk, különbözőség esetén viszont egy új elemet kell a halmazba tenni. Ennek érdekében a halmaz elemeinek számát növeljük eggyel (ld. 7. sor), majd bemásoljuk az aktuális tömbelemet a halmazba (ld. 8. sor). A ciklus végeztével az előálló halmazt és annak elemszámát kimenetként visszaadjuk (ld. 11. sor).

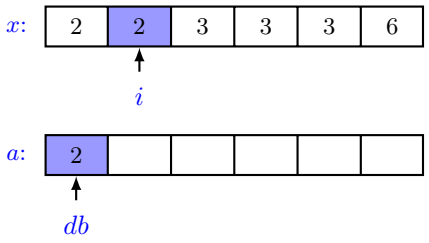
5.7. Példa. Az 5.7. ábrán láthatunk egy hat elemű tömböt, mely számos ismétlődő elemet is tartalmaz. Az ábrán végigkövethetjük, hogy az 5.11. algoritmus használatával miként alkothatjuk meg az x-nek megfelelő a halmazt.

Első lépésként létrehozunk egy hat elemű tömböt az a halmaz számára (ld. 2. sor), majd a tömb első elemét átmásoljuk a halmaz első helyére a 3-4. soroknak megfelelően (ld. 5.7a. ábra). Ezután megkezdjük a tömb bejárását az 5. sorban kezdődő ciklussal. Mivel a tömb második eleme, megegyezik a halmaz első elemével (ld. 6. sor), ezért nem másoljuk be még egyszer az ismétlődő elemet a halmazba (ld. 5.7b. ábra). Folytatjuk a bejárást, a harmadik elem különbözik a halmaz utolsó elemétől, ezért a halmaz elemszámát növeljük (ld. 7. sor), majd átmásoljuk az aktuális tömbelemet (ld. 8. sor), amit az 5.7c. ábra szemléltet. A tömb negyedik és ötödik eleme már benne van a halmazban (ld. 5.7d-5.7e. ábrák), ezért nem kerülnek

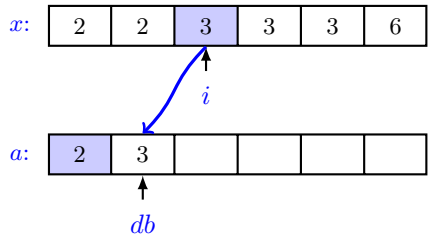
be ismételten a halmazba. A hatodik elem viszont még nem halmazbeli, ezért bemásoljuk a halmazba (ld. 5.7f. ábra). A bejárás végére érve létrejött egy ismétlődés nélküli halmaz, melyben az x minden kezdeti eleme egyes multiplicitással szerepel. ¶



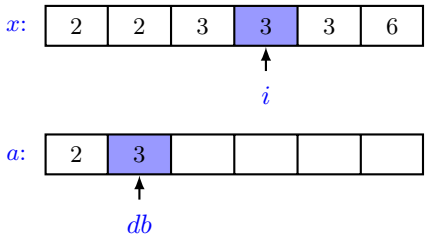
(a) Az x tömb első elemét automatikusan átmásoljuk az a halmazba.



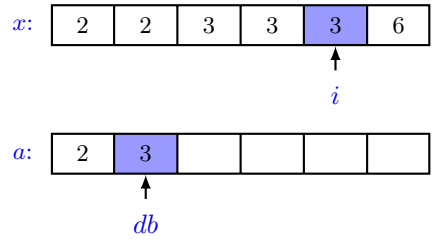
(b) Az x második eleme megegyezik az a első elemével, ezért nem helyezzük ismét a halmazba.



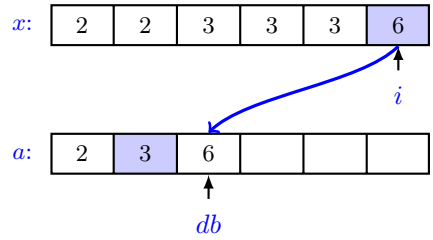
(c) Az x harmadik eleme még nincs a halmazban, ezért a soron következő helyre másoljuk.



(d) Az x negyedik eleme már benne van az a-ban.



(e) Az x ötödik eleme is benne van az a-ban.



(f) Az x hatodik eleme még nincs a halmazban, ezért a soron következő helyre másoljuk.

5.7. ábra. Halmaz létrehozása rendezett tömbből.

Futási idő elemzése. Mivel az 5.11. algoritmus egyetlen ciklust tartalmaz, ezért könnyen látható, hogy a futási idő az x elemszámával arányos, tehát $O(n)$-es. Vessük ezt össze a 2.21. algoritmus futási idejével, ami $O(n^2)$-es volt. A rendezettség kihasználása tehát jelentős futási idő javulást eredményez. ♣

Tartalmazás vizsgálata

Halmazokkal kapcsolatban fontos feladat, hogy el tudjuk dönteni, valami benne van-e egy halmazban vagy sem. Jelenlegi ismereteink mellett ez nem jelent nehéz feladatot, hiszen egy rendezett tömbben – amilyen egy halmaz is – az 5.4. algoritmus használatával el tudjuk dönteni, hogy a keresett elem a vizsgált tömbben van-e. Az 5.12. algoritmus halmazok esetén írja le a tartalmazás vizsgálatot. Mivel semmi új ismeret nincs ebben az algoritmusban, ezért a részletesebb ismertetéstől eltekintünk.

5.12. Algoritmus Tartalmazás vizsgálat

Bemenet: a – **T halmaz**, n – **egész** (*halmaz mérete*), *érték* – **T**; ahol **T** összehasonlítható
Kimenet: l – **logikai**

1: **függvény** TARTALMAZZAE(a : **T halmaz**, n : **egész**, *érték* : **T**)
2: $bal \leftarrow 1$
3: $jobb \leftarrow n$
4: $center \leftarrow \left\lfloor \frac{bal+jobb}{2} \right\rfloor$
5: **ciklus amíg** $(bal \leq jobb) \wedge (a[center] \neq érték)$
6: **ha** $a[center] > érték$ **akkor**
7: $jobb \leftarrow center - 1$
8: **különben**
9: $bal \leftarrow center + 1$
10: **elágazás vége**
11: $center \leftarrow \left\lfloor \frac{bal+jobb}{2} \right\rfloor$
12: **ciklus vége**
13: $l \leftarrow (bal \leq jobb)$
14: **vissza** l
15: **függvény vége**

Felhasznált változók és függvények

- a: Halmaz.
- n: Az a halmaz mérete.
- *érték*: A halmaz elemeivel azonos típusú érték. Azt vizsgáljuk, hogy ez az *érték* benne van-e az a halmazban.
- l: Logikai változó, amely pontosan akkor ***igaz***, ha *érték* megtalálható az a halmazban.
- bal: Az aktuálisan vizsgált részhalmaz első elemének indexe.
- $jobb$: Az aktuálisan vizsgált részhalmaz utolsó elemének indexe.
- $center$: Az aktuálisan vizsgált részhalmaz középső elemének indexe.

Részhalmaz

A részhalmaz vizsgálatnál feladatunk annak meghatározása, hogy egy a halmaz minden egyes eleme benne van-e egy másik b halmazban. Ezt a feladatot megoldhatnánk úgy, hogy az a halmaz minden egyes elemére megnézzük, hogy benne van-e a b halmazban, tehát két egymásba ágyazott eldöntés tételt (ld. 2.1.2. fejezet) alkalmaznánk. Így a futási idő a két halmaz méretének szorzatával lenne arányos. Viszont ez a megoldás nem használja ki, hogy a halmazban lévő elemek növekvő módon rendezettek.

Egy másik megoldási lehetőség, ha a két halmazt párhuzamosan járjuk be a halmazok elejéről indulva. Amennyiben a bejárás során az a aktuális $a[i]$ eleme kisebb, mint a b aktuális $b[j]$ eleme, akkor nincs értelme tovább vizsgálódnunk, mert a b halmaz j-nél nagyobb indexű elemei már mind nagyobbak $a[i]$-nél, tehát biztos nem egyenlők az $a[i]$-vel. Ha viszont $a[i] = b[j]$, akkor továbbléphetünk az a következő elemére, miközben a b-ben is továbbhaladunk. Amennyiben $a[i] > b[j]$, akkor pedig továbbra is keresni kell még az $a[i]$-t a b halmazban, ezét csak b-ben haladunk tovább. Ha a bejárás során túljutunk az a összes elemén, akkor az a minden elemét megtaláltuk b-ben, tehát a részhalmaza b-nek.

A leírt ötletet az 5.13. algoritmussal valósítjuk meg. Az algoritmus bemenete a két halmaz, valamint azok mérete. Kimenet egy logikai változó, amely pontosan akkor igaz, ha az a részhalmaza b-nek.

5.13. Algoritmus Részhalmaz vizsgálat

Bemenet: a – **T halmaz**, m – **egész** (a *mérete*), b – **T halmaz**, n – **T halmaz** (b *mérete*)
Kimenet: l – **logikai**

```
1:  függvény RÉSZHALMAZ_E(a : T halmaz, m : egész, b : T halmaz, n : egész)
2:      i ← 1
3:      j ← 1
4:      ciklus amíg (i ≤ m) ∧ (j ≤ n) ∧ (a[i] ≥ b[j])
5:          ha a[i] = b[j] akkor
6:              i ← i + 1
7:          elágazás vége
8:          j ← j + 1
9:      ciklus vége
10:     l ← (i > m)
11:     vissza l
12: függvény vége
```

Felhasznált változók és függvények

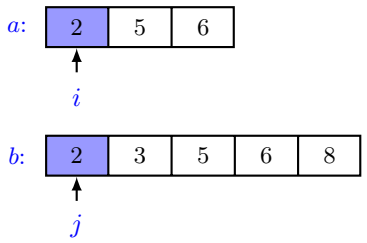
- a: Egyik halmaz.
- m: a mérete.
- b: Másik halmaz.
- n: b mérete.
- l: Pontosan akkor ***igaz***, ha $a \subseteq b$.

Az algoritmus elején inicializáljuk az a halmaz i, illetve a b halmaz j indexét. Mivel mindkét halmazt az elejétől járjuk be, ezért az indexek kezdeti értéke 1 lesz (ld. 2. és 3. sorok). A tömbök bejárását a 4. és 9. sorok közötti ciklussal valósítjuk meg. A ciklus belépési és bennmaradási feltétele három részből tevődik össze. Az i és j indexnek is tényleges tömbindexnek kell lennie, valamint az a halmaz aktuális eleme nem lehet kisebb a b aktuális eleménél. A ciklusban megvizsgáljuk, hogy a két halmaz aktuális elemei egyenlők-e (ld. 5. sor). Ha egyenlők, akkor továbblépünk az a halmazban (ld. 6. sor). Az egyenlőségtől függetlenül továbbhaladunk a b halmazban (ld. 8. sor). A ciklusból való kilépést követően megvizsgáljuk a kilépés okát. Amennyiben az a halmaz indexe meghaladja az a méretét, akkor minden a-beli elem benne van b-ben is, tehát a részhalmaza b-nek. Ezért az l kimeneti változót az $i > m$ relációtól függően értelmezzük (ld. 10), majd értékét kimenetként adjuk vissza (ld. 11. sor).

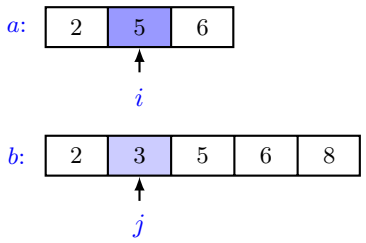
> Megjegyzés
>
> Ha az a halmaz több elemet tartalmaz mint b (azaz $m > n$), akkor biztos, hogy a nem részhalmaza b-nek. Ezért az algoritmus elejére, akár betehetünk egy méretvizsgálatot is. Viszont $m < n$ esetben ez egy felesleges vizsgálat, ezért nem jelenítettük meg az 5.13. algoritmusban.

5.8. Példa. Az 5.8. ábrán bemutatjuk, hogy miként dönti el az 5.13. algoritmus két halmazról, hogy az egyik részhalmaza-e a másiknak.

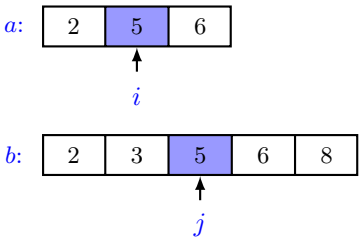
Kezdetben az i és j indexek értéke 1 lesz (ld. 2. és 3. sor). Mivel a 4. sorban kezdődő ciklus belépési feltétele ***igaz***, ezért belépünk a ciklusba. A cikluson belül az $a[1] = b[1]$ feltétel teljesül (ld. 5.8a. ábra), ezért i és j értéke is növekszik 1-gyel (ld. 6. és 8. sorok). Az $i = 2$ és $j = 2$ esetben a bejárást biztosító ciklus bennmaradási feltétele ***igaz***, de az $a[2] = b[2]$ feltétel ***hamis*** (ld. 5.8b. ábra), ezért csak a j index értékét növeljük 1-gyel. Még mindig bennmaradunk a ciklusban, ahol az $a[2] = b[3]$ egyenlőség teljesül (ld. 5.8c. ábra), ezért az i és a j értékét is növeljük. A ciklus bennmaradási feltétele még mindig ***igaz***, valamint az aktuális elemek is egyenlőek (ld. 5.8d. ábra), ezért továbbnöveljük mindkét indexet. Ekkor viszont az i index már meghaladja az a halmaz méretét, ezért kilépünk a ciklusból és az l változó ***igaz*** értéket vesz fel (ld. 10. sor). ¶



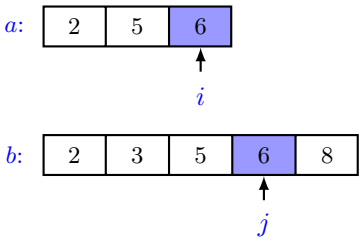
(a) Mivel $a[1] = b[1]$, ezért mindkét tömbben továbblépünk.



(b) Mivel $a[2] > b[2]$, ezért folytatjuk az eldöntést, de csak a b halmazban lépünk tovább.



(c) Mivel $a[2] = b[3]$, ezért mindkét tömbben továbblépünk.



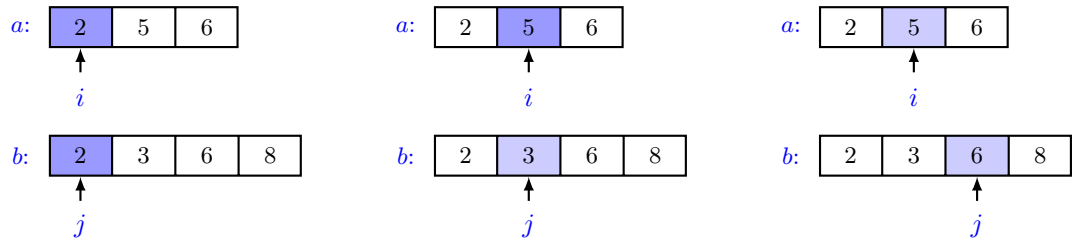
(d) Mivel $a[3] = b[3]$, ezért mindkét tömbben továbblépünk. Az a végére értünk, így a részhalmaza b-nek.

5.8. ábra. Részhalmaz vizsgálat. Az a halmaz részhalmaza a b halmaznak.

5.9. Példa. Az 5.9. ábrán látható esetben végigkövethetjük, hogy az 5.13. algoritmus miként dönti el, hogy a nem részhalmaza b-nek.

Mindkét halmaz első eleme azonos (ld. 5.9a. ábra), ezért továbblépünk a második elemekre. Az a második eleme nagyobb a b második eleménél (ld.5.9b. ábra), ezért csak a b halmaz indexét növeljük. Viszont $a[2] < b[3]$, ezért kilépünk a bejárást megvalósító ciklusból és a kimeneti l változó ***hamis*** értéket kap. ¶

Futási idő elemzése. Az 5.13. algoritmus futási idejének meghatározásánál a 4. sorban kezdődő ciklusra kell fókuszálnunk. Mivel a cikluson belül a j index mindig eggyel növelődik, ezért a ciklus legfeljebb n-szer fut le. Tehát a részhalmaz tulajdonságot vizsgáló algoritmusunk futási ideje a második halmaz méretével arányos, azaz $O(n)$-es. Ez a futási idő sokkal jobb, mint a fejezet elején említett $O(m \cdot n)$-es idő. ♣



(a) Mivel $a[1] = b[1]$, ezért mindkét tömbben továbblépünk.

(b) Mivel $a[2] > b[2]$, ezért folytatjuk az eldöntést, de csak a b halmazban lépünk tovább.

(c) Mivel $a[2] < b[3]$, ezért a nem lehet részhalmaza b-nek.

5.9. ábra. Részhalmaz vizsgálat. Az a nem részhalmaza a b halmaznak.

Unió

A halmazok uniójának megvalósítása teljes mértékben megegyezik a korábban már tárgyalt összefuttatás programozási tétellel (ld. 2.2.6. fejezet), ezért csak az algoritmus pszeudokódját adjuk meg (ld. 5.14. algoritmus).

5.14. Algoritmus Halmazok uniója

Bemenet: a_1 – **T halmaz**, n_1 – **egész** (*halmaz mérete*), a_2 – **T halmaz**, n_2 – **egész** (*halmaz mérete*)
Kimenet: b – **T halmaz**, db – **egész**

```
1:  függvény HalmazUnió(a1 : T halmaz, n1 : egész, a2 : T halmaz, n2 : egész)
2:      b ← Létrehoz(T)[n1 + n2]
3:      i ← 1
4:      j ← 1
5:      db ← 0
6:      n1 ← n1 + 1
7:      a1[n1] ← +∞
8:      n2 ← n2 + 1
9:      a2[n2] ← +∞
10:     ciklus amíg (i < n1) ∨ (j < n2)
11:         db ← db + 1
12:         ha a1[i] < a2[j] akkor
13:             b[db] ← a1[i]
14:             i ← i + 1
15:         különben
16:             ha a1[i] > a2[j] akkor
17:                 b[db] ← a2[j]
18:                 j ← j + 1
19:             különben
20:                 b[db] ← a1[i]
21:                 i ← i + 1
22:                 j ← j + 1
23:             elágazás vége
24:         elágazás vége
25:     ciklus vége
26:     vissza (b, db)
27: függvény vége
```

Felhasznált változók és függvények

- a_1: Egyik halmaz.
- n_1: Az a_1 halmaz mérete.
- a_2: Másik halmaz.
- n_2: Az a_2 halmaz mérete.
- b: Kimeneti halmaz, melynek minden egyes eleme vagy az a_1 vagy az a_2 halmaznak eleme.
- db: A b halmaz releváns elemeinek száma.
- Létrehoz(**T**)$[n_1 + n_2]$: Utasítás, mely létrehoz egy $(n_1 + n_2)$ elemű **T** típusú tömböt.

Metszet

Két halmaz metszetében a halmazok közös elemei lesznek benne. Ennek érdekében az uniót megvalósító algoritmust kell úgy módosítanunk, hogy csak az egyező elemek kerüljenek be a kimeneti halmazba. A konkrét megvalósítást az 5.15. algoritmussal adjuk meg.

5.15. Algoritmus Halmazok metszete

Bemenet: a_1 – **T halmaz**, n_1 – **egész** (*halmaz mérete*), a_2 – **T halmaz**, n_2 – **egész** (*halmaz mérete*)
Kimenet: b – **T halmaz**, db – **egész**

```
1:  függvény HalmazMetszet(a1 : T halmaz, n1 : egész, a2 : T halmaz, n2 : egész)
2:      b ← Létrehoz(T)[min(n1, n2)]
3:      i ← 1
4:      j ← 1
5:      db ← 0
6:      ciklus amíg (i ≤ n1) ∧ (j ≤ n2)
7:          ha a1[i] < a2[j] akkor
8:              i ← i + 1
9:          különben ha a1[i] > a2[j] akkor
10:             j ← j + 1
11:         különben
12:             db ← db + 1
13:             b[db] ← a1[i]
14:             i ← i + 1
15:             j ← j + 1
16:         elágazás vége
17:     ciklus vége
18:     vissza (b, db)
19: függvény vége
```

Felhasznált változók és függvények

- a_1: Egyik halmaz.
- n_1: Az a_1 halmaz mérete.
- a_2: Másik halmaz.
- n_2: Az a_2 halmaz mérete.
- b: Kimeneti halmaz, melynek minden egyes eleme az a_1 és az a_2 halmaznak is eleme.
- db: A b halmaz releváns elemeinek száma.
- Létrehoz(**T**)[min(n_1, n_2)]: Utasítás, mely létrehoz egy **T** típusú tömböt, melynek mérete az a_1 és a_2 halmazok méretének minimuma.

Különbség

Az a_1 és a_2 halmaz különbségének az elemei az a_1 halmaz azon elemei, amelyek nincsenek benne az a_2 halmazban. A különbség a b halmazban jelenik meg. Ezek meghatározása is az unióhoz hasonló módon tehető meg, ahogy az 5.16. algoritmusban látható.

5.16. Algoritmus Halmazok különbsége

Bemenet: a_1 – **T halmaz**, n_1 – **egész** (*halmaz mérete*), a_2 – **T halmaz**, n_2 – **egész** (*halmaz mérete*)
Kimenet: b – **T halmaz**, db – **egész**

```
1:  függvény HalmazKulonbseg(a1 : T halmaz, n1 : egész, a2 : T halmaz, n2 : egész)
2:      b ← Létrehoz(T)[n1]
3:      i ← 1
4:      j ← 1
5:      db ← 0
6:      ciklus amíg (i ≤ n1) ∧ (j ≤ n2)
7:          ha a1[i] < a2[j] akkor
8:              db ← db + 1
9:              b[db] ← a1[i]
10:             i ← i + 1
11:         különben ha a1[i] > a2[j] akkor
12:             j ← j + 1
13:         különben
14:             i ← i + 1
15:             j ← j + 1
16:         elágazás vége
17:     ciklus vége
18:     ciklus amíg i ≤ n1
19:         db ← db + 1
20:         b[db] ← a1[i]
21:         i ← i + 1
22:     ciklus vége
23:     vissza (b, db)
24: függvény vége
```

Felhasznált változók és függvények

- a_1: Egyik halmaz.
- n_1: Az a_1 halmaz mérete.
- a_2: Másik halmaz.
- n_2: Az a_2 halmaz mérete.
- b: Kimeneti halmaz, melynek minden egyes eleme az a_1 halmaznak eleme, de az a_2 halmaznak nem.
- db: A b halmaz releváns elemeinek száma.
- Létrehoz(**T**)$[n_1]$: Utasítás, mely létrehoz egy n_1 méretű **T** típusú tömböt.

Szimmetrikus differencia

Két halmaz szimmetrikus differenciájának elemei azok az elemek, amelyek a két halmazból pontosan egyben vannak benne. Ezen elemek kiválogatása is az unió módosításával történik. A konkrét pszeudokódot az 5.17. algoritmusban mutatjuk be.

5.17. Algoritmus Halmazok szimmetrikus differenciája

Bemenet: a_1 – **T halmaz**, n_1 – **egész** (*halmaz mérete*), a_2 – **T halmaz**, n_2 – **egész** (*halmaz mérete*)
Kimenet: b – **T halmaz**, db – **egész**

```
1: függvény HalmazSzimmetrikusDifferencia(a1 : T halmaz, n1 : egész, a2 : T halmaz, n2 : egész)
2:     b ← Létrehoz(T)[n1 + n2]
3:     i ← 1
4:     j ← 1
5:     db ← 0
6:     ciklus amíg (i ≤ n1) ∧ (j ≤ n2)
7:         ha a1[i] < a2[j] akkor
8:             db ← db + 1
9:             b[db] ← a1[i]
10:            i ← i + 1
11:        különben ha a1[i] > a2[j] akkor
12:            db ← db + 1
13:            b[db] ← a2[j]
14:            j ← j + 1
15:        különben
16:            i ← i + 1
17:            j ← j + 1
18:        elágazás vége
19:    ciklus vége
20:    ciklus amíg i ≤ n1
21:        db ← db + 1
22:        b[db] ← a1[i]
23:        i ← i + 1
24:    ciklus vége
25:    ciklus amíg j ≤ n2
26:        db ← db + 1
27:        b[db] ← a2[j]
28:        j ← j + 1
29:    ciklus vége
30:    vissza (b, db)
31: függvény vége
```

Felhasznált változók és függvények

- a_1: Egyik halmaz.
- n_1: Az a_1 halmaz mérete.
- a_2: Másik halmaz.
- n_2: Az a_2 halmaz mérete.
- b: Kimeneti halmaz, melynek minden egyes eleme az a_1 és az a_2 halmazok közül pontosan az egyiknek eleme.
- db: A b halmaz releváns elemeinek száma.
- Létrehoz(**T**)$[n_1 + n_2]$: Utasítás, mely létrehoz egy $n_1 + n_2$ méretű **T** típusú tömböt.

6. fejezet

„Oszd meg és uralkodj!” elvű algoritmusok

Az „Oszd meg és uralkodj!”[1] elvű algoritmusok lényegesen különböznek az eddig megismertektől. Az előző fejezetekben minden algoritmus olyan volt, amely egy tömb bejárásán alapult. A bejárás lehetett olyan, amely a tömb elejétől a tömb végéig haladva, minden elemet egyszer érintett. Ez a bejárás megvalósulhatott ciklusok vagy rekurzív hívások használatával is. Néhány esetben a bejárás kétirányú volt (pl. a 2.17. algoritmus esetén). Láttunk olyan algoritmusokat is, amikor a bejárás úgy történt, hogy a bejárására használt index értéke feleződött (pl. a logaritmikus keresésnél).

Ezekkel szemben az „Oszd meg és uralkodj!” elvű algoritmusoknál nem történik bejárás, illetve nem a bejáráson van a hangsúly, hanem teljesen más elven működnek. Az „Oszd meg és uralkodj!” elvű algoritmusok mindig három fázisból állnak. Első lépésként a megoldandó problémát felosztják kisebb részproblémákra. Például, ha rendezni szeretnénk egy tömböt, akkor nem a teljes tömb rendezésére koncentrálnak, hanem a tömböt kettő vagy több résztömbre osztják. Második fázisként „uralkodnak” a részproblémákon, azaz a részproblémákat megoldják. Ez a megoldás úgy történik, hogy a teljes problémát megoldó algoritmust alkalmazzák a kisebb részproblémára rekurzív módon. Majd miután az egyes részproblémák már megoldottak, akkor harmadik lépésként a megoldott részproblémák egyesítése következik.

Talán már érezhető ebből a leírásból is, hogy azt igazából nem határozzuk meg, hogy a részproblémák megoldása hogyan történik meg, hiszen azt a rekurzivitás biztosítja majd. Miként működhetnek akkor ezek az algoritmusok?

Az „Oszd meg és uralkodj!” elv bemutatása és működésének megértése érdekében négy feladatot ismertetünk részletesen. Elsőként a már többször tárgyalt maximumkiválasztás problémáját oldjuk meg az új módszerrel (ld. 6.1. fejezet). Ezt követően bemutatunk egy hatékony rendező algoritmust, az összefésülő rendezést, mely az új elvet használja (ld. 6.2. fejezet). Harmadikként egy másik rendezést, a gyakran használt gyorsrendezést ismerjük meg (ld. 6.3. fejezet). Végezetül a gyorsrendezés ötletére építve bemutatunk egy hatékony módszert a k-adik legkisebb tömbbeli elem kiválasztására (ld. 6.4. fejezet).

[1] Angolul: divide and conquer

Maximumkiválasztás

A maximumkiválasztás feladatot jól ismerjük, hiszen már két megoldást is adtunk rá (ld. 2.1.6. és 4.6.4. fejezetek). Most viszont az eddigiektől teljesen különböző megközelítést alkalmazó megoldást mutatunk be.

Egy tömb legnagyobb értékű elemének helyét úgy fogjuk megkeresni, hogy kiindulásként nem az egész tömböt vizsgáljuk, hanem annak az első és a második felét külön-külön. Vagyis így felosztjuk a problémát két részproblémára. Ezt követően megkeressük a kisebb méretű tömbök legnagyobb értékű elemeinek indexeit. Ez a keresés rekurzív módon történik, tehát a résztömböket újabb résztömbökre fogjuk osztani. Mint minden rekurziónál itt is fontos a rekurzió leállását biztosító alapesetről gondoskodni. Ha egy elemű már a vizsgált résztömbünk, akkor nem fogunk további felosztást és rekurzív hívást (uralkodás) végezni, hiszen az egy elemű résztömb legnagyobb eleme az egyetlen benne lévő elem. Miután meghatároztuk az eredeti tömb mindkét felében a legnagyobb elemet, illetve annak indexét, már csak el kell döntenünk, hogy a két féltömbben található maximumok közül melyik a nagyobb. Amelyik nagyobb, az lesz az egész tömb maximuma, tehát annak az indexét adjuk meg eredményül.

A feladatból jól látható az „Oszd meg és uralkodj!" elvű algoritmusok működésének három fázisa. Első lépésként megtörténik az eredeti tömb felosztása két résztömbbé. Második lépésként rekurzív módon meghatározzuk az egyes résztömbök legnagyobb elemeinek helyét, azaz a résztömbökre megoldjuk ugyanazt a feladatot, amit a teljes tömbre is meg kell oldanunk. Ez az uralkodás fázisa. Harmadik lépésként pedig egyesítjük a két résztömbre kapott megoldást, ami jelen esetben a két maximum közül a nagyobbik kiválasztását jelenti.

A feladatot megoldó konkrét algoritmus pszeudokódját a 6.1. algoritmusban mutatjuk be. A algoritmus bemenete a vizsgálandó x tömb, illetve a *bal* és a *jobb* indexértékek. Ezek az indexek jelzik, hogy éppen melyik résztömbbel foglalkozunk. Kezdetben természetesen *bal* értéke 1, míg *jobb* a tömb n méretével egyezik meg. Az algoritmus kimenete a legnagyobb értékű tömbelem indexe lesz.

6.1. Algoritmus Felező maximumkiválasztás

Bemenet: x – **T tömb**, *bal* – **egész**, *jobb* – **egész**; ahol **T** összehasonlítható
Kimenet: Az x tömb *bal* és *jobb* indexei közötti résztömbje maximális elemének indexe.

```
1:  függvény FelezőMaximumkiválasztás(x : T tömb, bal : egész, jobb : egész)
2:      ha bal = jobb akkor
3:          vissza bal
4:      különben
5:          center ← ⌊(bal+jobb)/2⌋
6:          balmax ← FelezőMaximumkiválasztás(x, bal, center)
7:          jobbmax ← FelezőMaximumkiválasztás(x, center + 1, jobb)
8:          ha x[balmax] ≥ x[jobbmax] akkor
9:              vissza balmax
10:         különben
11:             vissza jobbmax
12:         elágazás vége
13:     elágazás vége
14: függvény vége
```

Függvény hívása: FelezőMaximumkiválasztás(x, 1, n)

Felhasznált változók és függvények

- x: A tömb.
- n: Az x tömb mérete.
- *bal*: Az aktuálisan vizsgált résztömb első elemének indexe.
- *jobb*: Az aktuálisan vizsgált résztömb utolsó elemének indexe.
- *center*: Az aktuálisan vizsgált résztömb középső elemének indexe.
- *balmax*: A *bal* és *center* indexek közötti tömbrész maximális elemének indexe.
- *jobbmax*: A *center* + 1 és *jobb* indexek közötti tömbrész maximális elemének indexe.

A rekurzív algoritmusban először gondoskodunk az alapeset kezeléséről. Ha a vizsgált tömbrész egy elemű, azaz $bal = jobb$ (ld. 2. sor), akkor annak az egy elemnek az indexét adja vissza az algoritmus (ld. 3. sor). Minden más esetben fel kell osztani a tömböt két (közel) egyenlő méretű részre. Ennek érdekében meghatározzuk a *bal* és *jobb* indexek közötti középső elem indexét (ld. 5. sor). A közép helyének ismeretében már meghívhatjuk rekurzívan az előálló résztömbökre az algoritmust. A bal oldali rész eredménye kerül a *balmax* változóba (ld. 6. sor), míg a jobb oldali részé a *jobbmax*-ba (ld. 7. sor). Ezután már csak az egyesítés fázisa van hátra. Megvizsgáljuk, hogy a bal- vagy a jobboldali rész maximuma-e a nagyobb (ld. 8. sor), és ennek függvényében visszaadjuk a megfelelő kimenetet (ld. 9. és 11. sorok).

Megjegyzés

Érdemes megfigyelni, hogy abban az esetben, amikor a legnagyobb elem többször is előfordul a tömbben, akkor az elem első előfordulásának indexét adja vissza az algoritmus. Ezt úgy értük el, hogy a 8. sorban lévő feltételnél az egyenlőség teljesülése esetén is a baloldali résztömb maximumát adjuk vissza kimenetként. Így tényleg a legbaloldalibb maximum helyét fogjuk végső eredményként megkapni.

6.1. Példa. Nézzük végig egy konkrét példán a 6.1. algoritmus működését. A 6.1. ábrán látható hat elemű tömb maximális értékű elemének indexét szeretnénk meghatározni.

Először meghívjuk a FelezőMaximumkiválasztás rekurzív függvényt a $bal = 1$ és $jobb = 6$ paraméterekkel. Mivel $bal \neq jobb$, ezért az algoritmus különben ágába lépünk (ld. 4. sor). Meghatározzuk a középső elem *center* indexét (ld. 5. sor), ami 3 lesz (ld. 6.1a. ábra). Ezt követően meghívódik a FelezőMaximumkiválasztás függvény a $bal = 1$ és $jobb = 3$ paraméterekkel. Ebben a függvényben is kiszámítjuk *center* értékét, ami 2 lesz (ld. 6.1b. ábra). Ismét meghívódik a FelezőMaximumkiválasztás függvény a $bal = 1$ és $jobb = 2$ paraméterekkel. A középső elem indexe most 1 lesz. Újabb rekurzív hívás következik a $bal = 1$ és $jobb = 1$ értékekkel (ld. 6.1d.ábra). Mivel $bal = jobb$ (ld. 2. sor), ezért a *bal* értékét adja vissza a függvény (ld. 3. sor), ahogy ez a 6.1e. ábrán látható. A visszaadott érték bekerül az aktív rekurzív függvény *balmax* változójába. Újabb rekurzív hívás következik a $bal = 2$ és $jobb = 2$ paraméterekkel (ld. 6.1f. ábra). A meghívott függvény a 2 értékkel tér vissza, ami az aktuális függvény *jobbmax* változójába kerül (ld. 6.1g. ábra). Mivel a *balmax* helyen tárolt érték nagyobb a *jobbmax* helyen lévő értéknél, ezért az aktuális függvény a $balmax = 1$ értékkel tér vissza (ld. 6.1h. ábra). Ezután meghívásra kerül a FelezőMaximumkiválasztás függvény a $bal = 3$ és $jobb = 3$ paraméterekkel (ld. 6.1i. ábra). A hívott függvény a 3 értékkel tér vissza, ami az aktuális függvény *jobbmax* változójába kerül (ld. 6.1j. ábra). Mivel $x[jobbmax] > x[balmax]$, ezért a *jobbmax* indexszel tér vissza a függvény (ld. 6.1k. ábra). Az algoritmus futásában most értünk el oda, hogy az eredeti tömb első felében ismertté vált a legnagyobb elem indexe, ez a jelenlegi *balmax* változóban van eltárolva.

Következik a jobboldali résztömb feldolgozása. Ennek érdekében meghívásra kerül a FelezőMaximumkiválasztás függvény a $bal = 4$ és $jobb = 6$ paraméterekkel. A középső elem az ötödik lesz (ld. 6.1l. ábra). Újabb rekurzív hívás következik a $bal = 4$ és $jobb = 5$ paraméterekkel, ahol a *center* index 4 lesz (ld. 6.1m. ábra). A következő rekurzív hívás a $bal = 4$ és $jobb = 4$ paraméterekkel történik (ld. 6.1n. ábra). A függvény visszatér a 4 értékkel (ld. 6.1o. ábra). Meghívásra kerül a FelezőMaximumkiválasztás függvény a $bal = 5$ és $jobb = 5$ paraméterekkel (ld. 6.1p. ábra). Ez a függvény visszatér az 5 értékkel, ami az aktuális függvény *jobbmax* változójába kerül (ld. 6.1q. ábra). Mivel $x[jobbmax] > x[balmax]$, ezért a *jobbmax* értékkel tér vissza a függvény (ld. 6.1r. ábra). Ezután meghívásra kerül a FelezőMaximumkiválasztás függvény a $bal = 6$ és $jobb = 6$ paraméterekkel (ld. 6.1s. ábra). Ez a függvény visszatér a 6 értékkel, ami a *jobbmax* változóba kerül (ld. 6.1t. ábra). Mivel a *balmax* helyen lévő érték nagyobb a *jobbmax* indexűnél, ezért az aktuális függvény a *balmax* értékével tér vissza (ld. 6.1u. ábra). Az eredeti tömb első felének legnagyobb eleme a harmadik (*balmax*), második felének maximuma pedig az ötödik (*jobbmax*) elem. Mivel $x[jobbmax] > x[balmax]$, ezért végső eredményként a *jobbmax* értékét kapjuk meg (ld.6.1v. ábra). ¶

Futási idő elemzése. A maximumkiválasztás 2.10. iteratív és 4.13. rekurzív algoritmusainak futási idejéről már láttuk, hogy $O(n)$-esek. Milyen a futási ideje a más elven működő felező maximumkiválasztás 6.1. algoritmusának?

Vizsgáljuk meg, hogy hány rekurzív függvényhívás történik. Kezdetben meg kell hívnunk a FelezőMaximumkiválasztás függvényt a teljes tömbre. Ez a függvény meghívja önmagát két résztömbre.

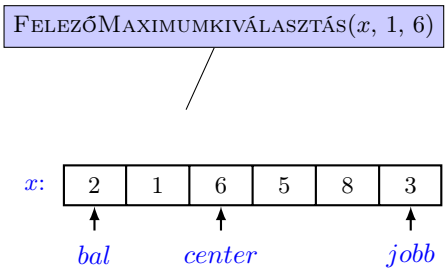
Ez minden esetben így történik addig, amíg egy elemű résztömbökre nem hívjuk meg a függvényt. Egy elemű résztömbből pedig pontosan n darab van. Ezek alapján a rekurzív hívások száma:

$$1 + 2 + \ldots + \frac{n}{2} + n. \tag{6.1}$$

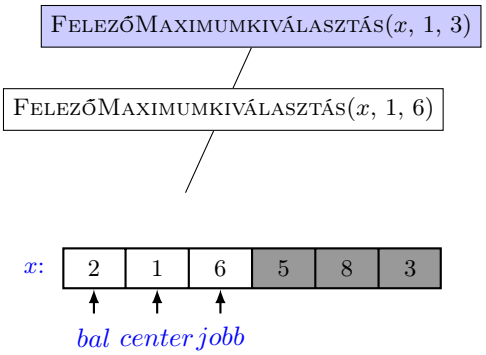
A fenti összeg egy mértani sorozat összege, ahol a kvóciens $q = 2$ az elemek száma pedig közelítőleg $1 + \log_2 n$. Ezek figyelembe vételével kijelenthető, hogy

$$1 + 2 + \ldots + \frac{n}{2} + n = 1 \cdot \frac{2^{1+\log_2 n} - 1}{2 - 1} = 2n - 1 = O(n). \tag{6.2}$$

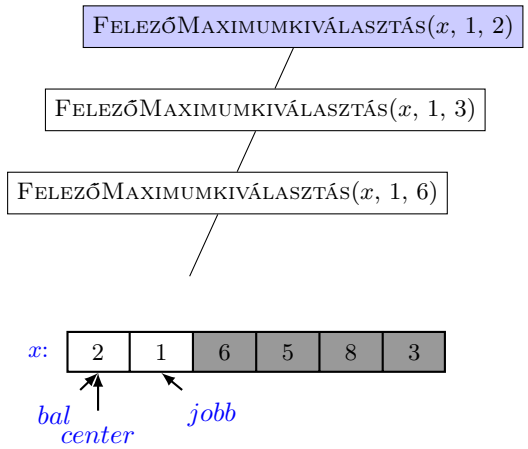
Tehát a futási idő az „Oszd meg és uralkodj!” elvű felező maximumkiválasztás esetén is $O(n)$-es. ♣



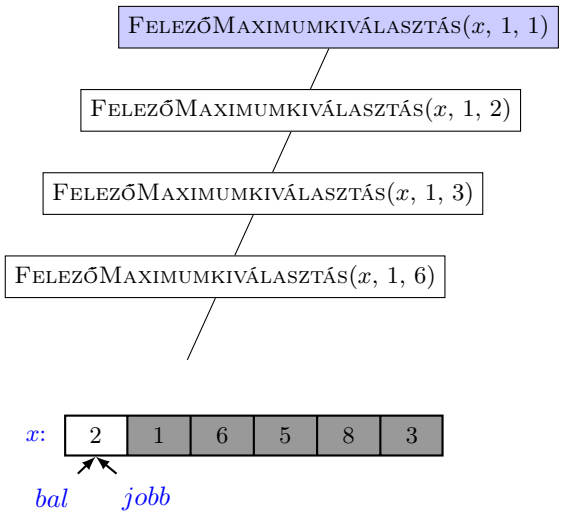
(a) Függvény hívása az 1. és 6. elemek közötti résztömbre.



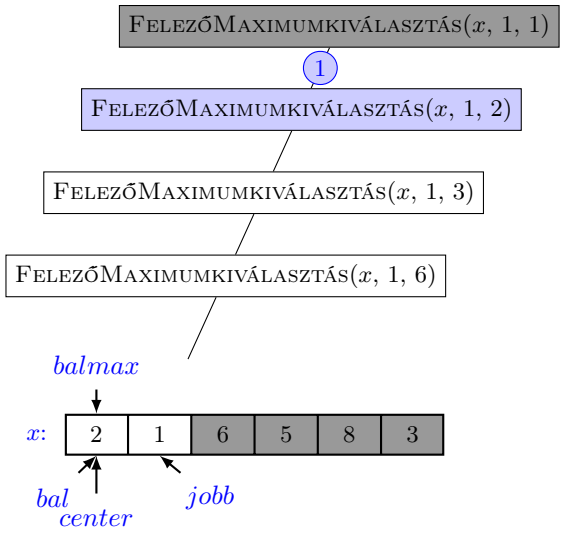
(b) Függvény hívása az 1. és 3. elemek közötti résztömbre.



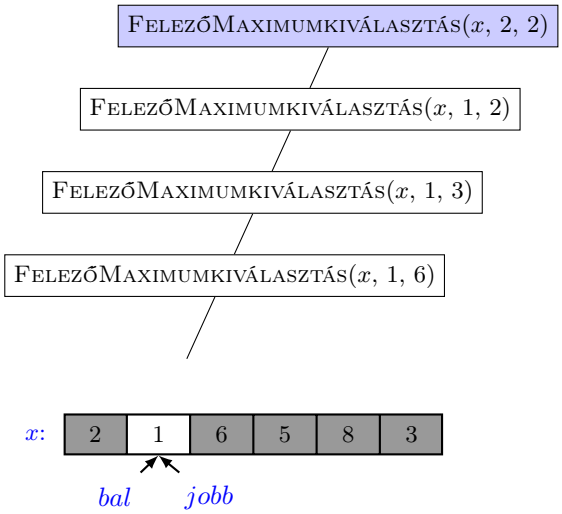
(c) Függvény hívása az 1. és 2. elemek közötti résztömbre.



(d) Függvény hívása az 1. elemből álló résztömbre.

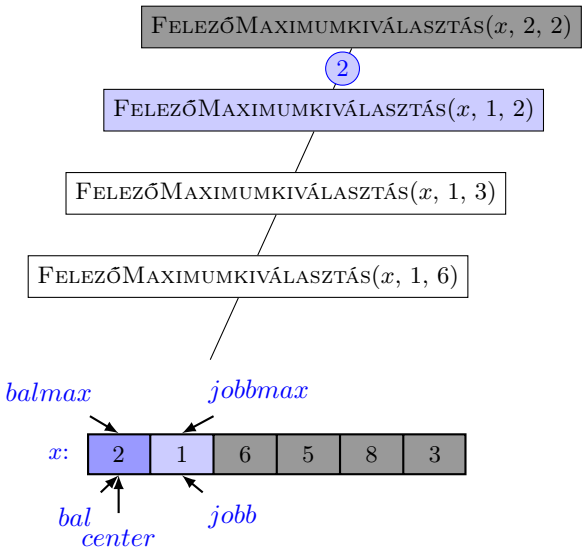


(e) Az 1. elemből álló résztömb maximumának megadása.

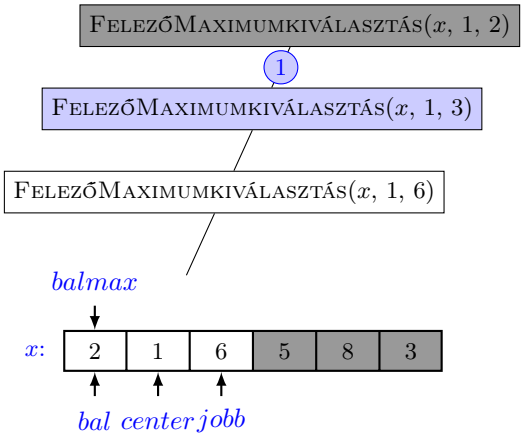


(f) Függvény hívása az 2. elemből álló résztömbre.

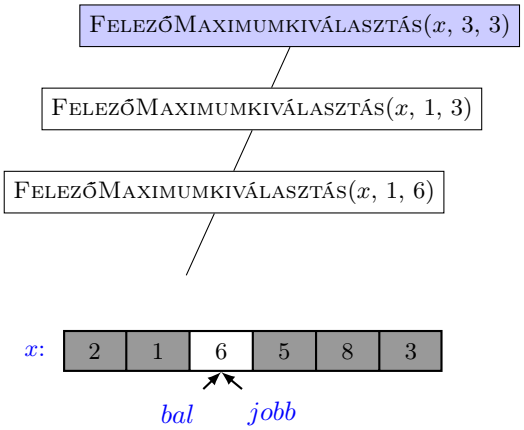
6.1. ábra. Felező maximumkiválasztás.



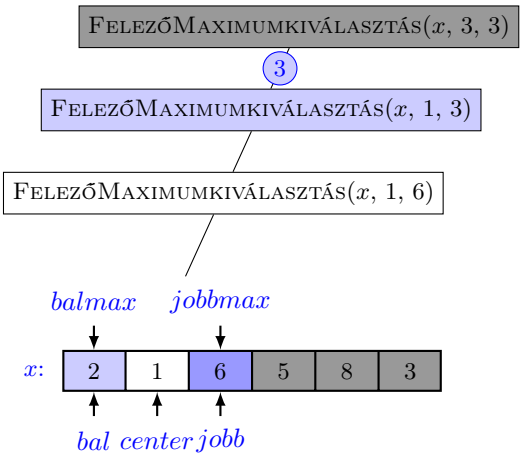
(g) Az 2. elemből álló résztömb maximumának megadása. A két eredmény összehasonlítása.



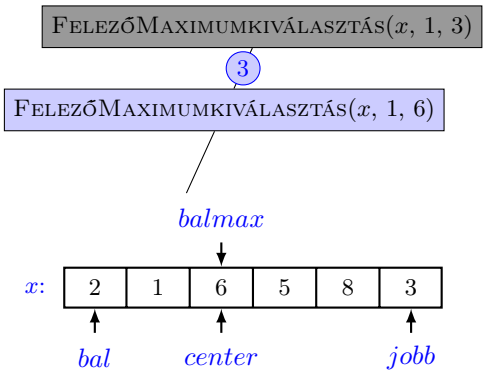
(h) Az 1. és 2. elemek közötti résztömb maximumának megadása.



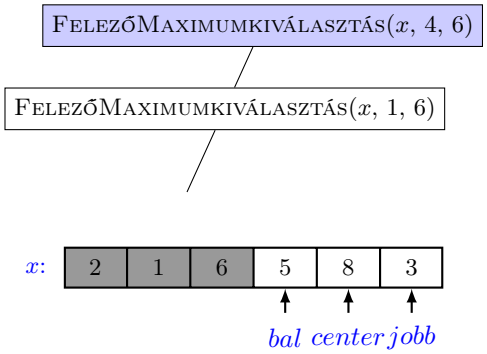
(i) Függvény hívása az 3. elemből álló résztömbre.



(j) A 3. elemből álló résztömb maximumának megadása. A két eredmény összehasonlítása.

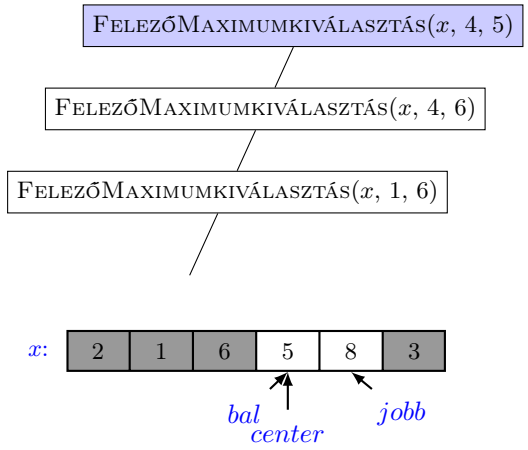


(k) Az 1. és 3. elemek közötti résztömb maximumának megadása.

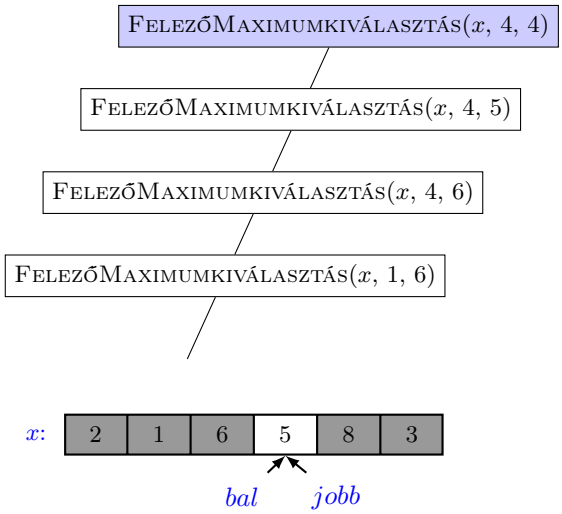


(l) Függvény hívása a 4. és 6. elemek közötti résztömbre.

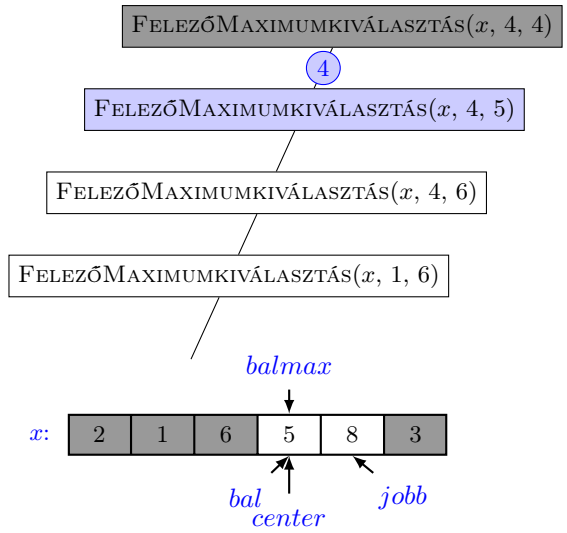
6.1. ábra. Felező maximumkiválasztás (folyt.).



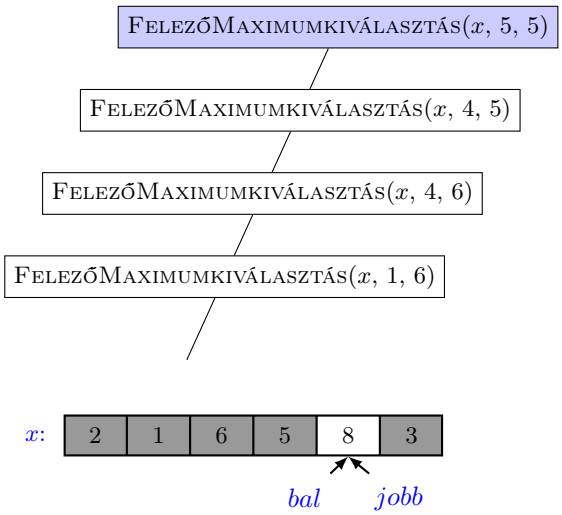
(m) Függvény hívása a 4. és 5. elemek közötti résztömbre.



(n) Függvény hívása az 4. elemből álló résztömbre.

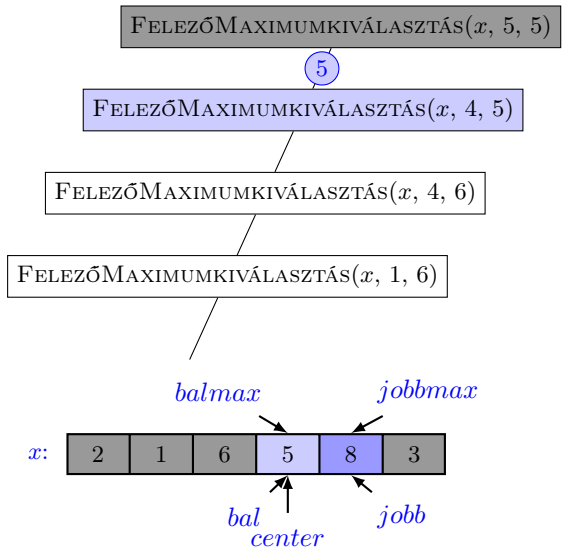


(o) A 4. elemből álló résztömb maximumának megadása.

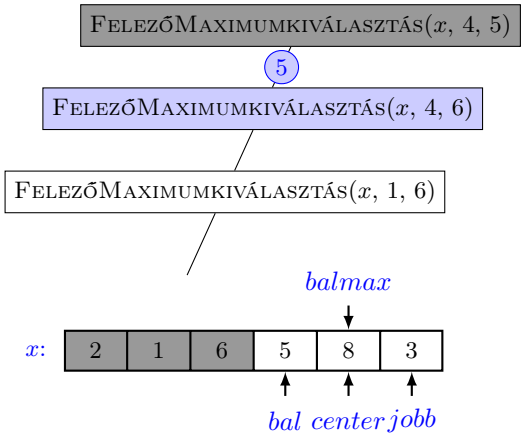


(p) Függvény hívása az 5. elemből álló résztömbre.

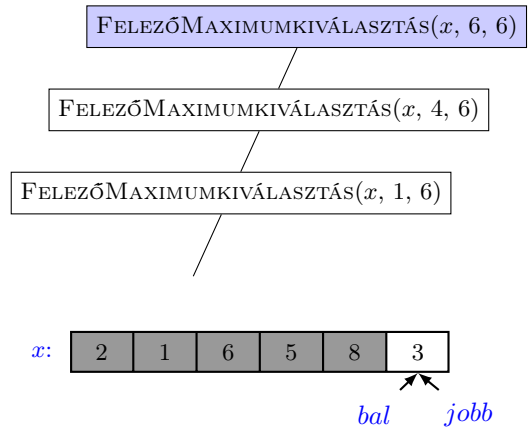
6.1. ábra. Felező maximumkiválasztás (folyt.).



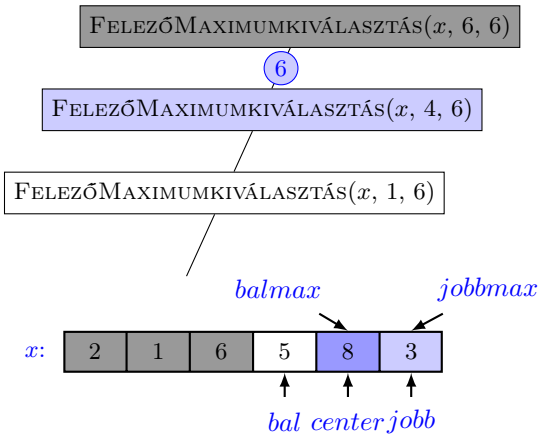
(q) Az 5. elemből álló résztömb maximumának megadása. A két eredmény összehasonlítása.



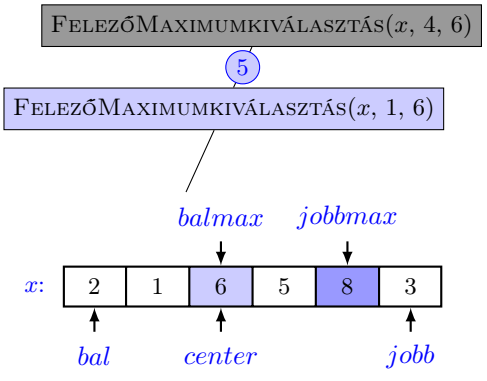
(r) A 4. és 5. elemek közötti résztömb maximumának megadása.



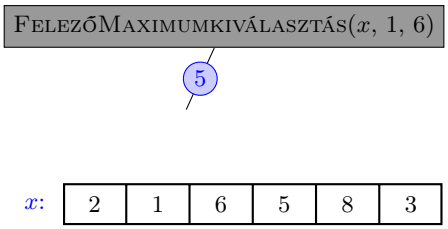
(s) Függvény hívása az 6. elemből álló résztömbre.



(t) A 6. elemből álló résztömb maximumának megadása. A két eredmény összehasonlítása.



(u) A 4. és 6. elemek közötti résztömb maximumának megadása. A két eredmény összehasonlítása.



(v) Az 1. és 6. elemek közötti résztömb maximumának megadása.

6.1. ábra. Felező maximumkiválasztás (folyt.).

Összefésülő rendezés

Rendező algoritmusokkal már a 3. fejezetben megismerkedtünk. Az ott látott rendezések cserékkel, beillesztésekkel operáltak, miközben valamilyen szabály alapján bejártuk a rendezendő tömböket. Az összefésülő rendezés[2] viszont teljesen más elven működik.

Az „Oszd meg és uralkodj!" elvet követő összefésülő rendezés ötlete – melyet Neumann János fejlesztett ki 1945-ben – nagyon hasonlít az előző fejezetben megismert felező maximumkiválasztáshoz. A rendezendő tömbünket úgy rendezzük, hogy szétvágjuk két (közel) egyenlő részre, majd az egyes részeket rendezzük. Ha pedig van két rendezett tömbünk, azokat az összefuttatás programozási tétel (ld. 2.2.6. fejezet) használatával már könnyen össze tudjuk fésülni egyetlen rendezett tömbbé.

Hogyan rendezzük a résztömböket? Pontosan ugyanúgy, mint ahogy a teljes tömböt. Azaz rekurzívan megoldjuk a rendezést ugyanazzal az algoritmussal, amivel a teljes tömböt is rendeztük. Ha csak egy elemű a rendezendő tömb, akkor pedig nem kell rendezni.

A fenti elvet megvalósító összefuttató rendezés megvalósítása nagyon egyszerű, ahogy azt a 6.2. algoritmusban láthatjuk.

6.2. Algoritmus Összefésülő rendezés

Bemenet: x – **T tömb**, bal – **egész**, $jobb$ – **egész**; ahol **T** összehasonlítható
Kimenet: x – **T rendezett tömb**

```
1: eljárás ÖsszefésülőRendezés(címszerint x : T tömb, bal : egész, jobb : egész)
2:     ha bal < jobb akkor
3:         center ← ⌊(bal+jobb)/2⌋
4:         ÖsszefésülőRendezés(x, bal, center)
5:         ÖsszefésülőRendezés(x, center + 1, jobb)
6:         Összefésül(x, bal, center, jobb)
7:     elágazás vége
8: eljárás vége
```

Eljárás hívása: ÖsszefésülőRendezés(x, 1, n)

Felhasznált változók és függvények

- x: Rendezendő tömb, mely az algoritmus végére rendezetté válik.
- bal: A tömb éppen rendezés alatt lévő részének alsó indexe.
- $jobb$: A tömb éppen rendezés alatt lévő részének felső indexe.
- $center$: A rendezés alatt lévő résztömb közepének indexe.
- n: A tömb mérete.
- Összefésül: Két rendezett tömböt összefuttat egy rendezett tömbbé (ld. 6.3. algoritmus).

Az algoritmust megvalósító eljárás bemenete a rendezendő x tömb, valamint a bal és $jobb$ indexek, amelyek meghatározzák, hogy az eredeti tömb melyik résztömbjével foglalkozunk éppen. Kezdetben $bal = 1$ és $jobb = n$. Az algoritmus helyben rendezést valósít meg, így futásának végére az x rendezetté válik. Ezen kívül további kimenetre nincs szükség, ezért tudjuk az algoritmust függvény helyett eljárással megvalósítani.

Ha egy elemű tömböt kell rendeznünk, akkor nincs semmi teendőnk, mivel egy egy elemű tömb már eleve rendezett. Ha viszont legalább két eleme van a vizsgált résztömbnek, akkor $bal < jobb$ (ld. 2. sor). Ilyenkor meghatározzuk a résztömb középső elemének indexét (ld. 3. sor). Ezután gondoskodunk a bal oldali résztömb rendezéséről egy megfelelő rekurzív hívással (ld. 4. sor), majd a jobb oldali résztömböt is rendezetté tesszük (ld. 5. sor). Ha van két rendezett résztömbünk, akkor ezek összefuttatását kell még megvalósítanunk, amihez egy önálló eljárást használunk (ld. 6. sor).

Az Összefésül eljárás lényegében az összefuttatás programozási tétel 2.23. algoritmusát valósítja meg kis módosítással. Változtatásra azért van szükség, mert most nem két bemeneti tömböt akarunk egy kimeneti tömbbe összefuttatni, hanem egy adott x tömb két résztömbjét akarjuk az x ugyanazon helyén összefuttatni. Mivel ezt adatvesztés nélkül nem tudjuk megtenni, ezért az x tömb megfelelő résztömbjeit ideiglenesen kiírjuk két átmeneti tömbbe. Az y_1 tömbbe kerül a bal és $center$ indexek közötti résztömb,

[2] Angolul: merge sort

6.3. Algoritmus Összefésülés

Bemenet: x – **T tömb**, *bal* – **egész**, *center* – **egész**, *jobb* – **egész**; ahol **T** összehasonlítható
Kimenet: x – **T tömb**

```
1:  eljárás ÖSSZEFÉSÜL(címszerint x : T tömb, bal : egész, center : egész, jobb : egész)
2:      n1 ← center − bal + 1
3:      n2 ← jobb − center
4:      y1 ← LÉTREHOZ(T)[n1 + 1]
5:      ciklus i ← 1-től n1-ig
6:          y1[i] ← x[bal + i − 1]
7:      ciklus vége
8:      y2 ← LÉTREHOZ(T)[n2 + 1]
9:      ciklus j ← 1-től n2-ig
10:         y2[j] ← x[center + j]
11:     ciklus vége
12:     y1[n1 + 1] ← +∞
13:     y2[n2 + 1] ← +∞
14:     i ← 1
15:     j ← 1
16:     ciklus k ← bal-től jobb-ig
17:         ha y1[i] ≤ y2[j] akkor
18:             x[k] ← y1[i]
19:             i ← i + 1
20:         különben
21:             x[k] ← y2[j]
22:             j ← j + 1
23:         elágazás vége
24:     ciklus vége
25: eljárás vége
```

Felhasznált változók és függvények

- x: A rendezendő tömb. A tömb *bal* és *center* közötti része, valamint *center* + 1 és *jobb* közötti része az algoritmus elején rendezett. Az algoritmus végére a tömb *bal* és *jobb* közötti része válik rendezetté.
- *bal*: Az x tömb összefésülendő résztömbjének kezdő indexe.
- *jobb*: Az x tömb összefésülendő résztömbjének záró indexe.
- *center*: Index, amely megadja, hogy a két rendezett résztömbnek hol van a határa.
- y_1: Segédtömb, melybe a kezdeti x *bal* és *center* közötti részét másoljuk át, illetve a végére teszünk egy elég nagy értéket.
- n_1: Az x tömb *bal* és *center* indexek közötti elemeinek száma.
- y_2: Segédtömb (y_1-hez hasonlóan) a kezdeti x *center* + 1 és *jobb* közötti elemeinek tárolására.
- n_2: Az x tömb *center* + 1 és *jobb* indexek közötti elemeinek száma.
- LÉTREHOZ(**T**)[n]: Utasítás, mely létrehoz egy n elemű **T** típusú tömböt.

az y_2 tömbbe pedig a $center+1$ és $jobb$ indexek közötti résztömb. Ezt követően már csak a két résztömböt kell az x tömb bal és $jobb$ indexű elemei közötti helyre összefésülni.

6.2. Példa. A 6.2. ábrán végig követhetjük, hogy az összefésülő rendezés 6.2. algoritmusa miként rendez egy hat elemű tömböt.

Első lépésben meghívjuk az ÖsszefésülőRendezés eljárást a $bal = 1$ és $jobb = 6$ paraméterekkel. Mivel $bal \neq jobb$, ezért meghatározásra kerül a középső elem $center$ indexe (ld. 6.2a. ábra). Rekurzívan meghívódik a ÖsszefésülőRendezés eljárás a $bal = 1$ és $jobb = 3$ paraméterekkel, a középső elem indexe pedig 3 lesz (ld. 6.2b. ábra). Ismét rekurzív hívás következik a $bal = 1$ és $jobb = 2$ paraméterekkel, $center$ pedig 1 lesz (ld. 6.2c. ábra). Újabb rekurzív hívás jön a $bal = 1$ és $jobb = 1$ értékekkel. Mivel $bal = jobb$ ezért semmi nem történik, hiszen egy 1 elemű résztömb eleve rendezett (ld. 6.2d. ábra). Ezt követően a $bal = 2$ és $jobb = 2$ paraméterekkel is meghívásra kerül az ÖsszefésülőRendezés eljárás. Mivel most is egy elemű a vizsgált résztömb, ezért semmi nem történik (ld. 6.2e. ábra). Ezután következik a két egy elemű résztömb összefésülése egy rendezett két elemű résztömbbé (ld. 6.2f. ábra). Következik a harmadik elemből álló egy elemű résztömb rendezése (ld. 6.2g. ábra), majd két rendezett résztömb összefésülése (ld. 6.2h. ábra). Ekkor az eredeti tömb első fele már rendezett.

Most következik a második féltömb rendezése a 6.2i-6.2o. ábráknak megfelelően. Utolsó lépésként pedig a két három elemű rendezett résztömböt összefésüljük egyetlen rendezett tömbbé (ld. 6.2p. ábra). ¶

Futási idő elemzése. Az összefésülő rendezés futási idejének elemzését kezdjük az Összefésül eljárás futási idejének elemzésével. Mennyi a futási ideje egy bal és $jobb$ közötti tömbrész esetén ennek az eljárásnak? Az összefuttatás programozási tétel futási idejének ismeretében könnyű belátni, hogy a futási idő a résztömb méretével arányos.

Mit mondhatunk akkor a teljes összefésülő rendezés futási idejéről? Amikor az eredeti n elemű tömbre meghívjuk az algoritmust, akkor történik két rekurzív hívás megközelítőleg $\frac{n}{2}$ méretű tömbökkel. Ezt követően pedig egy összefésülés jön egy n elemű tömbben. Így a $T(n)$ futási időről kimondhatjuk, hogy

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n). \tag{6.3}$$

Mennyi az $\frac{n}{2}$ elemű tömb esetén a futási idő? A gondolatmenet hasonló, tehát

$$T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\right). \tag{6.4}$$

Ezek alapján viszont a 6.3. képlet így alakul:

$$T(n) = 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot O\left(\frac{n}{2}\right) + O(n) = 4 \cdot T\left(\frac{n}{4}\right) + O(n) + O(n). \tag{6.5}$$

Folytathatnánk tovább a felezgetést meghatározva $T\left(\frac{n}{4}\right)$, majd $T\left(\frac{n}{8}\right)$, stb. értékét. Mindenhol azt kapnánk, hogy

$$T\left(\frac{n}{k}\right) = 2 \cdot T\left(\frac{n}{2k}\right) + O\left(\frac{n}{k}\right). \tag{6.6}$$

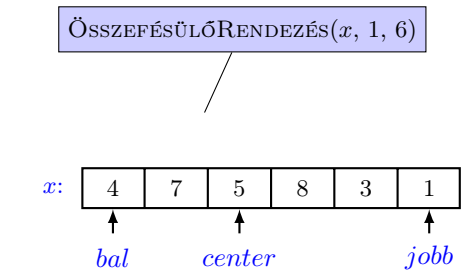
A fő kérdés az, hogy mikor nem tudjuk már továbbosztani a tömb aktuális méretét. Erre tudjuk viszont a választ: egy elemű tömb esetén. $T(1)$ ugyanis 1-gyel egyenlő.

Hány felezgetés kell addig elvégeznünk, amíg egy n elemű tömböt 1 elemű résztömbökre tudunk szétszabdalni? Közelítőleg $\log_2 n$.

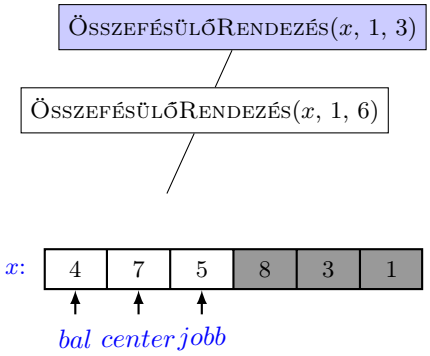
Mindezek alapján kijelenthető, hogy

$$T(n) = n \cdot T(1) + \underbrace{O(n) + \ldots + O(n)}_{\sim \log_2 n \text{ darab}} \approx n\,(1 + \log_2 n) = O(n \log n). \tag{6.7}$$

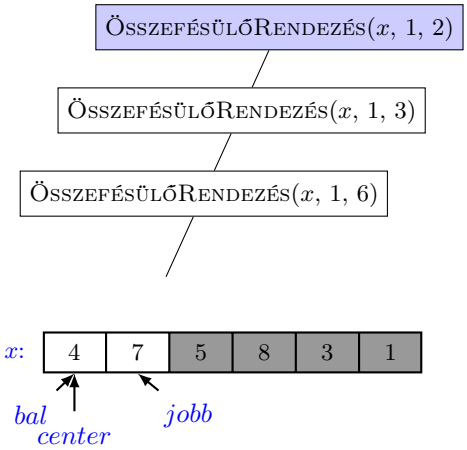
Az összefuttató rendezés futási ideje tehát $O(n \log n)$, amely az eddigi rendezéseknél jobb futási időt jelent. Az algoritmus hátránya viszont, hogy az összefuttatás megvalósítása miatt az eredeti tömb méretével megegyező méretű átmeneti tömbfoglalásra is szükség van, tehát jelentős memória igénye van. ♣



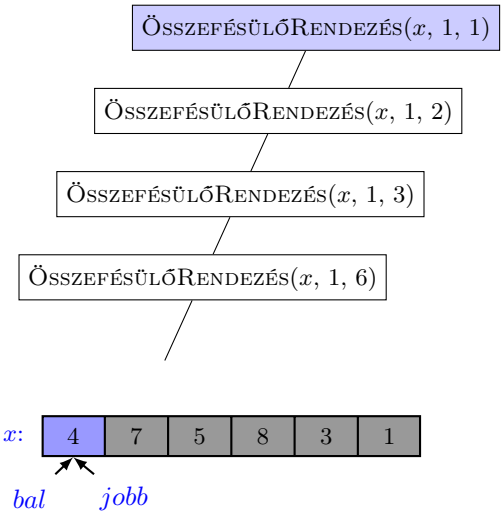
(a) Rendező eljárás hívása az 1. és 6. elemek közötti résztömbre.



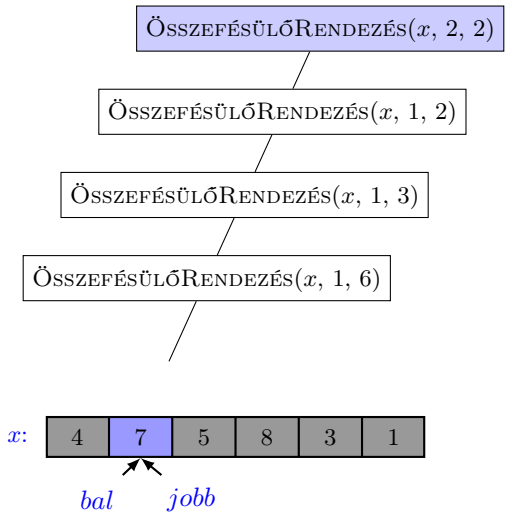
(b) Rendező eljárás hívása az 1. és 3. elemek közötti résztömbre.



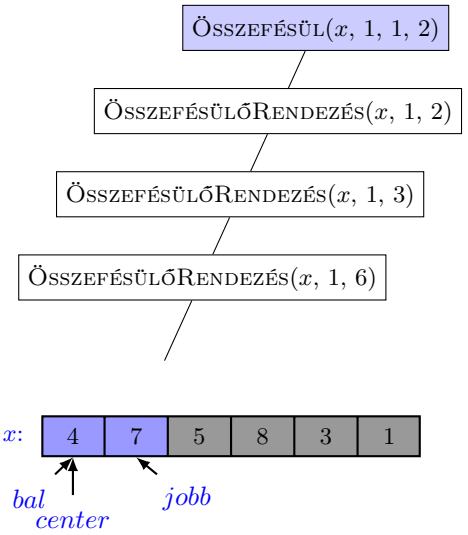
(c) Rendező eljárás hívása az 1. és 2. elemek közötti résztömbre.



(d) Rendező eljárás hívása az 1. elemből álló résztömbre. Az egy elemű tömb már eleve rendezett is.

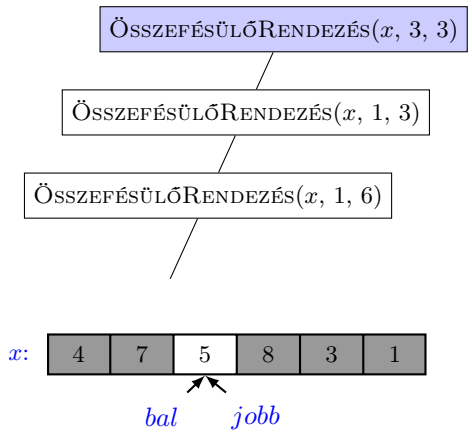


(e) Rendező eljárás hívása az 2. elemből álló résztömbre. Az egy elemű tömb már eleve rendezett is.

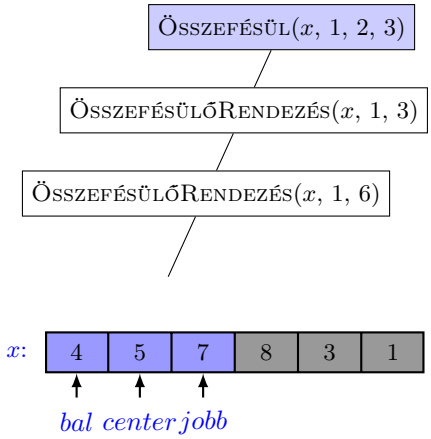


(f) Két rendezett résztömb összefésülése egy rendezett résztömbbé.

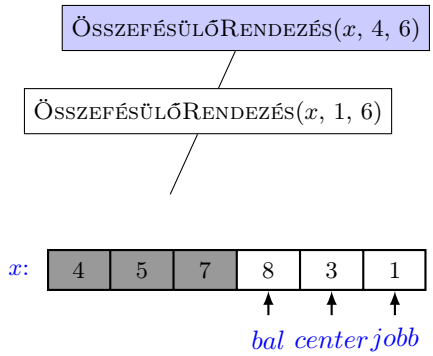
6.2. ábra. Összefésülő rendezés.



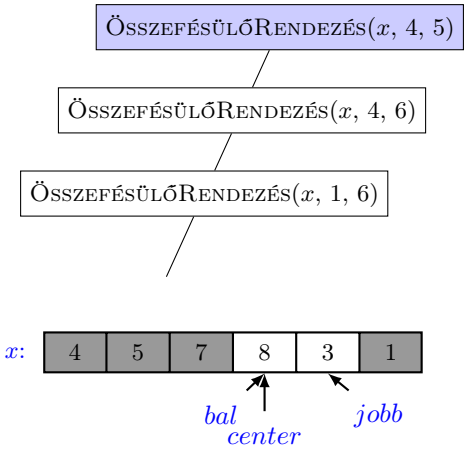
(g) Rendező eljárás hívása az 3. elemből álló résztömbre. Az egy elemű tömb már eleve rendezett is.



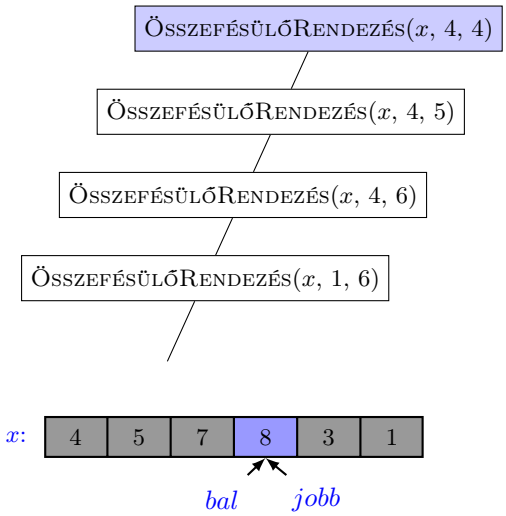
(h) Két rendezett résztömb összefésülése egy rendezett résztömbbé.



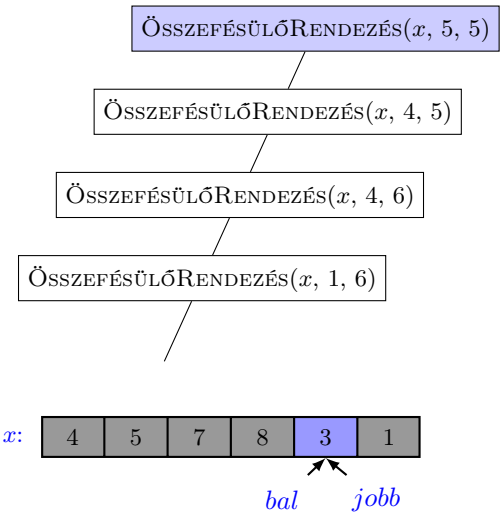
(i) Rendező eljárás hívása a 4. és 6. elemek közötti résztömbre.



(j) Rendező eljárás hívása a 4. és 5. elemek közötti résztömbre.

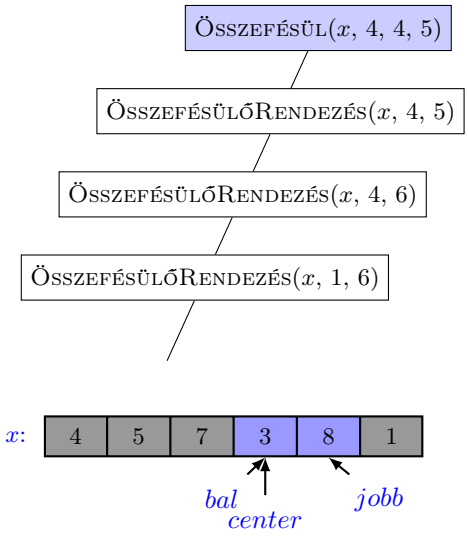


(k) Rendező eljárás hívása a 4. elemből álló résztömbre. Az egy elemű tömb már eleve rendezett is.

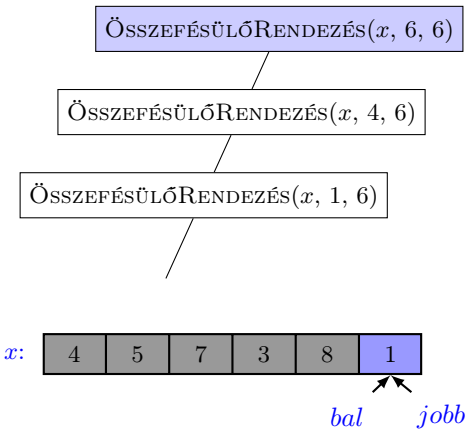


(l) Rendező eljárás hívása az 5. elemből álló résztömbre. Az egy elemű tömb már eleve rendezett is.

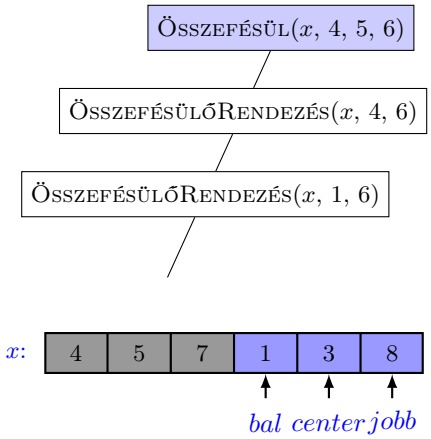
6.2. ábra. Összefésülő rendezés (folyt.).



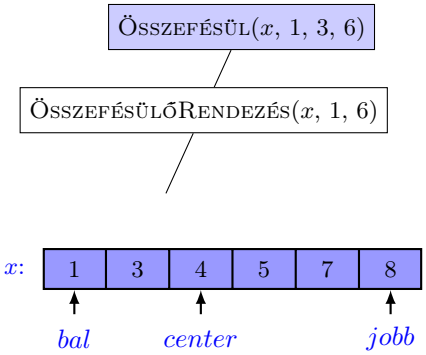
(m) Két rendezett résztömb összefésülése egy rendezett résztömbbé.



(n) Rendező eljárás hívása a 6. elemből álló résztömbre. Az egy elemű tömb már eleve rendezett is.



(o) Két rendezett résztömb összefésülése egy rendezett résztömbbé.



(p) Két rendezett résztömb összefésülése egy rendezett tömbbé.

6.2. ábra. Összefésülő rendezés (folyt.).

Gyorsrendezés

A gyorsrendezés[3] is egy „Oszd meg és uralkodj!” elven működő rendező algoritmus, de itt nem tudjuk előre, hogy a rendezendő tömböt milyen méretű résztömbökre osztjuk fel. Tehát ebben az esetben nem lesz igaz, hogy a tömböt mindig felezzük, ahogy ezt a felező maximumkiválasztásnál vagy az összefésülő rendezésnél tettük.

A gyorsrendezés legfontosabb lépése, hogy a tömb elemeit egy támpont elem[4] segítségével szétválogatja. A támpont például az eredeti tömb első eleme lehet. A tömb elemeinek szétválogatása pedig úgy történik, hogy a kiválasztott támpont elem olyan helyre kerül a tömbben, hogy előtte csak nála nem nagyobb, mögötte pedig csak nála nagyobb elemek vannak. Erre láthatunk egy példát a 6.3a. és 6.3b. ábrán.

x:	8	4	1	10	17	2	3	5	9	23

(a) Eredeti tömb, melyben az első elem a támpont.

x:	5	4	1	3	2	8	17	10	9	23

(b) Szétválogatott tömb. A támpont előtt csak nála nem nagyobb, mögötte pedig csak nála nagyobb elemek vannak.

x:	1	2	3	4	5	8	9	10	17	23

(c) Rendezett tömb. A támpont előtti és utáni elemek is rendezve lettek.

6.3. ábra. Támpont alapú szétválogatás és rendezés.

Ha szét tudjuk a tömböt válogatni az előbb leírt módon, akkor már csak annyi a feladatunk, hogy a támpont előtti, illetve utáni résztömböt külön-külön rendezzük. Ezt követően viszont az egész tömb is rendezetté válik, ahogy ez a 6.3c. ábrán is látható. Már csak az a kérdés, hogy miként rendezzük a támpont előtti és utáni elemekből álló két résztömböt. Erre egyszerű a válasz: rekurzív módon alkalmazzuk rájuk a gyorsrendezést.

Az eddigiek alapján az látható, hogy az „Oszd meg és uralkodj!” elv a gyorsrendezésnél másként alakul mint a korábbi két esetben. A felosztás a támpont alapú szétválogatással történik meg. Az uralkodás a támpont előtti és utáni résztömbök rekurzív rendezését jelenti. Az egyesítésre pedig itt nincs szükség, mert már az uralkodás fázis végére rendezetté válik az egész tömb.

Egyetlen kérdés maradt már csak megválaszolatlanul. Mi biztosítja, hogy a rekurzív gyorsrendezés leálljon, azaz mi a rekurzió alapesete? Mivel az egy elemű tömbök eleve rendezettek, ezért az egy (illetve nulla) elemű tömböknél nem végzünk újabb rekurzív hívást.

A gyorsrendezés konkrét megvalósítását a 6.4. algoritmusban mutatjuk be. Az algoritmust megvalósító GyorsRendezés eljárás bemenete a rendezendő x tömb, valamint az aktuális résztömb indexhatárait jelölő *bal* és *jobb* paraméterek. Az első eljáráshívás esetén $bal = 1$ és $jobb = n$. Az eljárásnak nincs külön kimenete, hanem a címszerinti paraméterátadással megadott x tömbben történik meg a rendezés. Emiatt kijelenthető, hogy a gyorsrendezés is egy helyben rendező algoritmus.

Első lépésben meghívjuk a Szétválogat függvényt (ld. 2. sor), mely megvalósítja a támpont alapú szétválogatást. Mivel a Szétválogat (ld. 6.5. algoritmus) függvény a helyben történő szétválogatást (ld. 2.17. algoritmus) valósítja meg, ezért részletes ismertetésétől eltekintünk. Csak két lényeges különbséget említünk meg. Az egyik, hogy a szétválogatás nem a teljes x tömbben történik, hanem csak a *bal* és *jobb* indexek közötti résztömbben. Másrészt a szétválogatásnál használt P tulajdonság, a támpont elemhez képesti kisebb egyenlőség.

A Szétválogat függvény – mivel cím szerinti paraméter átadással adjuk át az x tömböt – átrendezi az x tömböt úgy, hogy az $x[bal]$ támpont elem előtt csak nála nem nagyobb, mögötte pedig csak nála nagyobb elemek lesznek. Kimenetként pedig visszaadja a támpont elem új helyét a tömbben.

[3] Angolul: quicksort
[4] Angolul: pivot element

6.4. Algoritmus Gyorsrendezés

Bemenet: x – **T tömb**, bal – **egész**, $jobb$ – **egész**; ahol **T** összehasonlítható
Kimenet: x – **T rendezett tömb**

```
1: eljárás GyorsRendezés(címszerint x : T tömb, bal : egész, jobb : egész)
2:     idx ← Szétválogat(x, bal, jobb)
3:     ha idx > bal + 1 akkor
4:         GyorsRendezés(x, bal, idx − 1)
5:     elágazás vége
6:     ha idx < jobb − 1 akkor
7:         GyorsRendezés(x, idx + 1, jobb)
8:     elágazás vége
9: eljárás vége
```

Eljárás hívása: GyorsRendezés(x, 1, n)

Felhasznált változók és függvények

- x: A rendezendő tömb. Az algoritmus végén x növekvő módon rendezett lesz.
- bal: A tömb éppen rendezés alatt lévő részének alsó indexe.
- $jobb$: A tömb éppen rendezés alatt lévő részének felső indexe.
- idx: A rendezés alatt lévő tömbrész első elemének – a támpont elemnek – a helye a Szétválogat függvény lefutását követően.
- n: Az x tömb mérete.
- Szétválogat: Egy tömböt úgy alakít át, hogy a tömb első elemét (támpont elem) úgy teszi helyre (ez lesz az idx index), hogy minden idx előtti elem nem nagyobb és minden idx utáni elem nagyobb az támpont elemnél.

Ezután megnézzük, hogy a vizsgált résztömb támpont előtti része hány elemet tartalmaz. Ha legalább két elemű a támpont előtti rész (ld. 3. sor), akkor rendezni kell még ezt a résztömböt. Ennek érdekében rekurzívan meghívjuk a GyorsRendezés eljárást a megfelelő paraméterekkel (ld. 4. sor). Hasonlóan megnézzük, hogy legalább két elemű-e a támpont elem utáni vizsgálandó résztömb (ld. 6. sor). Ha igen, akkor rekurzívan azt a tömbrészt is rendezzük (ld. 7. sor).

6.3. Példa. A 6.4. ábrán végigkövethetjük, hogy miként rendez a 6.4. algoritmus egy tizennégy elemű tömböt.

Először a teljes tömböt rendeznünk kell, ezért a GyorsRendezés eljárást meghívjuk a $bal = 1$ és $jobb = 14$ paraméterekkel. Az eljáráson belülről meghívásra kerül a Szétválogat függvény ugyanazokkal a paraméterekkel. A függvény szétválogatja a tömböt úgy, hogy a kezdeti első elem (támpont) a negyedik helyre kerül, előtte csak nála kisebb, mögötte pedig csak nála nagyobb elemek vannak a tömbben. A függvény a 4 értéket adja vissza (ld. 6.4a. ábra). A támpont elem már a helyére került, viszont az első három elemből álló bal oldali és az utolsó tíz elemből álló jobb oldali résztömb rendezése még hátra van.

Először rekurzívan meghívásra kerül a GyorsRendezés eljárás a $bal = 1$ és $jobb = 3$ paraméterekkel. Ez az eljárás gondoskodik az első három elemet tartalmazó résztömb rendezéséről. Ennek érdekében meghívódik a Szétválogat függvény a megfelelő paraméterekkel. A jelenlegi támpont a harmadik helyre kerül (ld. 6.4b. ábra). Mivel az éppen vizsgált résztömb jobb szélére került a támpont, ezért csak a bal oldali résztömbbel kell a továbbiakban foglalkozni.

Meghívásra kerül a GyorsRendezés eljárás a $bal = 1$ és $jobb = 2$ paraméterekkel. Az eljárás elején meghívódik a Szétválogat függvény ugyanezekkel a paraméterekkel. Az aktuális támpont elem az első helyre kerül (valójában ott marad). Mivel a vizsgált két elemű résztömb első helyén van a támpont, ezért bal oldali résztömb nem is jön létre, a jobb oldali pedig egy elemű, ezért nem kell további vizsgálat. Így az első két elemből álló résztömb már rendezett (ld. 6.4c. ábra).

A rekurzívan meghívott eljárások futása véget ér, a vezérlés visszaadódik a külvilágból meghívott eljáráshoz. Az eredeti tömb bal oldalát már feldolgoztuk, most következik a jobb oldal rendezése. Ennek érdekében meghívódik a GyorsRendezés eljárás a $bal = 5$ és $jobb = 14$ paraméterekkel. A szétválogatás eredményeként az aktuális támpont elem a nyolcadik helyre kerül (ld. 6.4d. ábra).

6.5. Algoritmus Gyorsrendezés szétválogatása

Bemenet: x – **T tömb**, bal – **egész**, $jobb$ – **egész**; ahol **T** összehasonlítható
Kimenet: x – **T tömb**, idx – **egész**

```
1:  függvény SZÉTVÁLOGAT(címszerint x : T tömb, bal : egész, jobb : egész)
2:      segéd ← x[bal]
3:      ciklus amíg bal < jobb
4:          ciklus amíg (bal < jobb) ∧ (x[jobb] > segéd)
5:              jobb ← jobb − 1
6:          ciklus vége
7:          ha bal < jobb akkor
8:              x[bal] ← x[jobb]
9:              bal ← bal + 1
10:             ciklus amíg (bal < jobb) ∧ (x[bal] ≤ segéd)
11:                 bal ← bal + 1
12:             ciklus vége
13:             ha bal < jobb akkor
14:                 x[jobb] ← x[bal]
15:                 jobb ← jobb − 1
16:             elágazás vége
17:         elágazás vége
18:     ciklus vége
19:     idx ← bal
20:     x[idx] ← segéd
21:     vissza idx
22: függvény vége
```

Felhasznált változók és függvények

- x: A rendezés alatt lévő tömb.
- bal: Kezdetben az x tömb éppen vizsgált résztömbjének alsó indexe, majd a bejárást segítő változó.
- $jobb$: Kezdetben az x tömb éppen vizsgált résztömbjének felső indexe, majd a bejárást segítő változó.
- $segéd$: Segédváltozó, melyben eltároljuk a vizsgált résztömb első elemét, azaz a kezdeti $x[bal]$-t.
- idx: Az eltárolt segédváltozó helye a függvény végén. Az idx-nél kisebb indexű elemek mind kisebbek vagy egyenlők a $segéd$-del, míg az idx-nél nagyobb indexű elemek mind nagyobbak a $segéd$-nél.

Következik az ötödik és hetedik elemek közötti tömbrész rendezése. (Az algoritmus nem veszi észre, hogy ez már most is rendezett.) Ennek érdekében meghívásra kerül a GyorsRendezés eljárás a $bal = 5$ és $jobb = 7$ paraméterekkel. A támpont elem az ötödik helyre kerül (ld. 6.4e. ábra). A támpont bal oldalán nincs rendezendő rész, de a jobb oldalán egy két elemű résztömb található.

Emiatt meghívásra kerül a GyorsRendezés eljárás a $bal = 6$ és $jobb = 7$ paraméterekkel. A támpont elem a hatodik helyen nyeri el végső helyzetét. Bal oldalán nincs rendezendő rész, a jobb oldalán is csak egy egy elemű résztömb árválkodik, ezért nem kerül sor innen további rekurzív hívásra (ld. 6.4f. ábra).

Visszatér a vezérlés az ötödök és tizennegyedik elemek közötti rész feldolgozásához, ahol a támpont bal oldalát már rendeztük. Következik a jobb oldal rendezése a $bal = 9$ és $jobb = 14$ paraméterek használatával. A támpont elem a tizedik helyre kerül. Így a támpont bal oldalán csak egy egy elemű, már eleve rendezett tömböt találunk (ld. 6.4g. ábra).

Az algoritmus a tizenegyedik és tizennegyedik helyen álló elemek közötti résztömb rendezésével folytatódik. A támpont a tizenkettedik helyre kerül, aminek a bal oldalán csak egy elem van, ezért csak a jobb oldal rendezése szükséges a későbbiekben (ld. 6.4h. ábra).

Az utolsó két elemű résztömb feldolgozása során a támpont a tizenharmadik helyre kerül, aminek csak egy egy elemű jobb szomszédja van, ezt nem kell már tovább rendezni (ld. 6.4i. ábra). A rekurzívan hívott eljárások futása véget ér, az elsőként hívott eljárás is leáll. A teljes tömb rendezetté vált. ¶

Futási idő elemzése. A gyorsrendezés futási ideje attól függ, hogy az éppen aktuális támpont elem hova kerül a vizsgált résztömbben. Legjobb esetnek azt tekinthetjük, ha az aktuális résztömb közepére kerül, ilyenkor két közel azonos méretű továbbiakban rendezendő résztömb keletkezik. Ahhoz, hogy ebben az esetben meg tudjuk határozni a futási időt, először fel kell elevenítenünk a helyben szétválogató 2.17. algoritmus futási idejéről tanultakat. A szétválogatás a tömb méretével arányos futási idejű, tehát n elemű tömb esetén $O(n)$-es.

Ahhoz, hogy az n elemű tömbben meghatározzuk a támpont helyét végre kell hajtanunk egy szétválogatást. Ez $O(n)$ idő alatt megvalósul. Ha a támpont középre kerül, akkor ezt követően két darab közel $\frac{n}{2}$ méretű tömböt kell rendezni. Ha egy n méretű tömb rendezésének ideje $T(n)$, akkor ugyanazzal egy fele akkor tömböt $T\left(\frac{n}{2}\right)$ idő alatt lehet rendezni. Ezek alapján látható, hogy

$$T(n) = O(n) + 2 \cdot T\left(\frac{n}{2}\right). \tag{6.8}$$

Az összefuttató rendezés futási idejénél már látott levezetés alapján ebből következik, hogy állandó felezgetés mellett a teljes futási idő:

$$T(n) = O\left(n \log n\right). \tag{6.9}$$

Tehát a gyorsrendezés legjobb esetben hasonló futási idejű, mint az összefuttató rendezés.

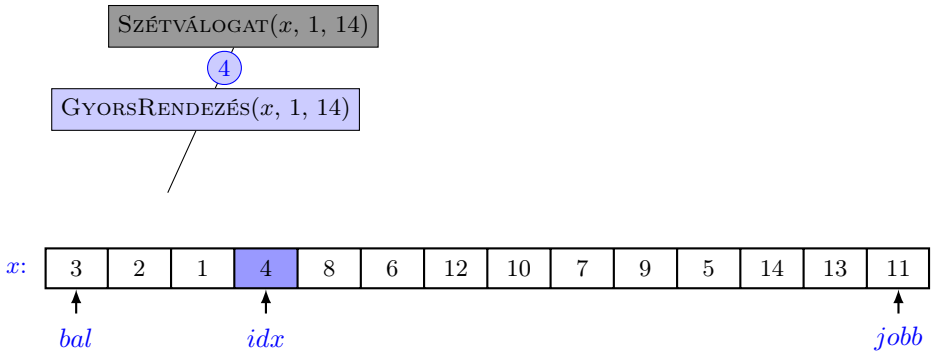
Mi a helyzet a legrosszabb esettel? Mi tekinthető egyáltalán futási idő szempontjából a legrosszabbnak? Legrosszabb az, ha a támpont elem minden esetben a tömb szélére, az elejére vagy a végére kerül. Ez könnyen előáll például akkor, ha eleve rendezett a tömb, vagy ha fordítva rendezett. Ilyenkor ugyanis első lépésben egy n elemű tömböt kell szétválogatni. Ezután csak egy tömbbel kell tovább foglalkozni – hiszen a támpontnak csak az egyik oldalán vannak még rendezendő elemek –, de ennek mérete $n - 1$. Hasonlóan a még további rendezendő és ezzel együtt szétválogatandó tömb mérete mindig csak eggyel fog csökkenni. Ilyenkor a futási idő a következőképpen alakul:

$$T(n) \sim n + (n-1) + (n-2) + \ldots + 2 = \frac{(2+n)(n-1)}{2} = O\left(n^2\right). \tag{6.10}$$

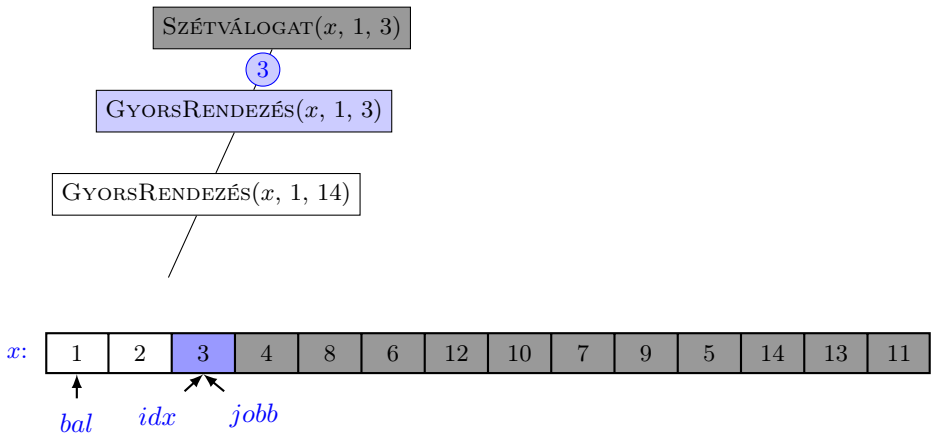
Ezek szerint a gyorsrendezés legjobb esetben versenyképes az összefuttató rendezéssel, legrosszabb esetben viszont a korábban megismert – nem hatékony – rendező algoritmusokhoz hasonló futási idővel rendelkezik.

Az átlagos esetet bizonyítás nélkül közöljük. A gyorsrendezés átlagos esetben $O(n \log n)$ futási idejű.

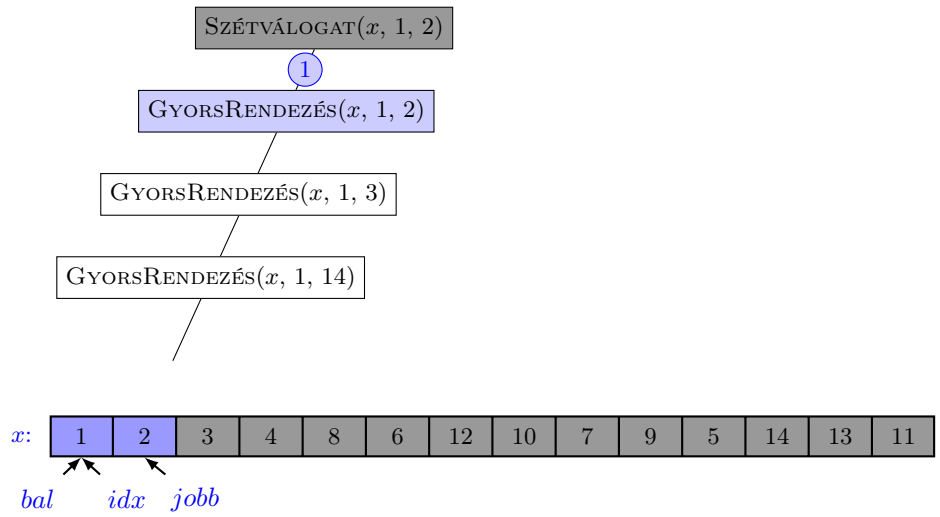
Fontos megemlíteni, hogy a gyorsrendezés memória igénye viszont lényegében csak a rendezendő tömb méretével azonos, hiszen a szétválogatásnál egyetlen többletértéket kell átmenetileg eltárolnunk. ♣



(a) Az 1. és 14. elem közötti résztömb rendezése. A támpont elem a 4. helyre kerül.

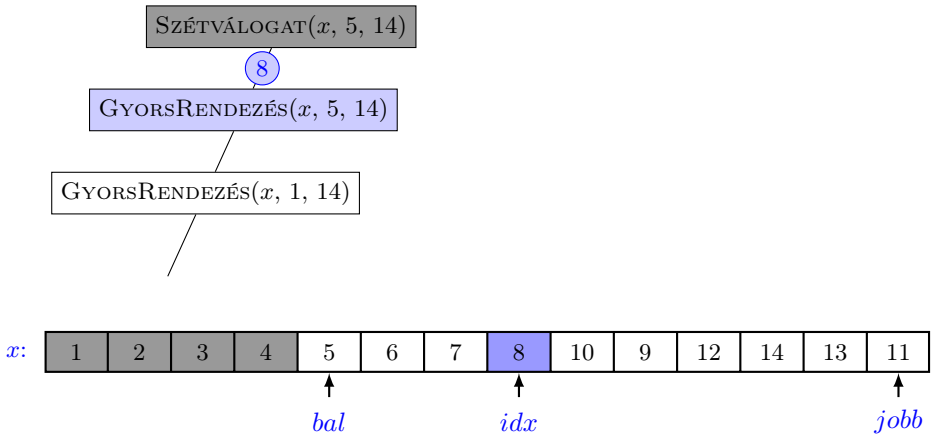


(b) Az 1. és 3. elem közötti résztömb rendezése. A támpont a 3. helyre kerül, ezért csak a bal oldali résztömböt kell rendezni.

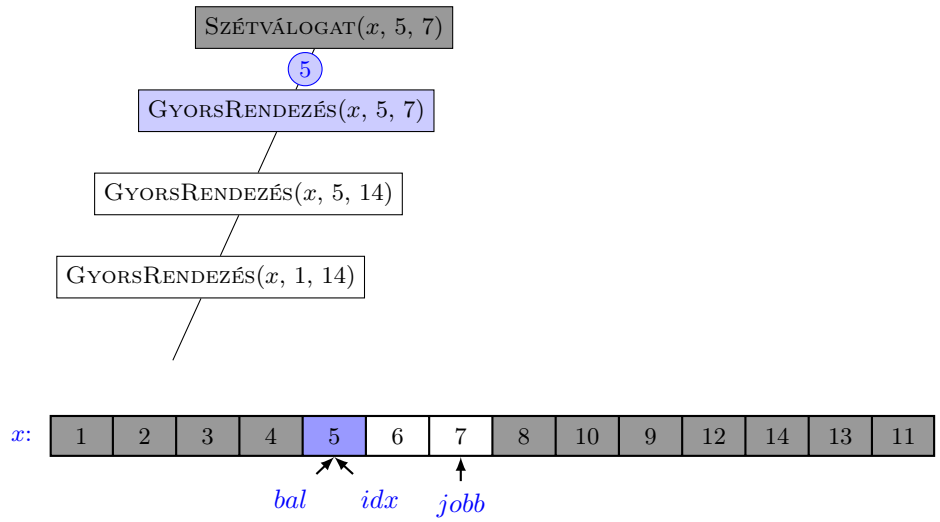


(c) Az 1. és 2. elem közötti résztömb rendezése. A támpont elem az 1. helyre kerül. További rendezés itt nem szükséges.

6.4. ábra. Gyorsrendezés.

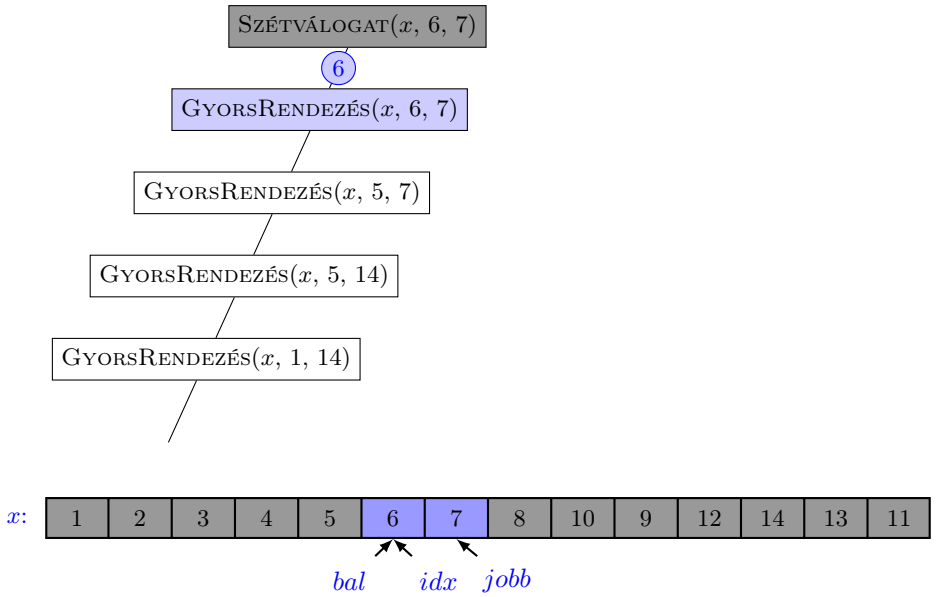


(d) Az 5. és 14. elem közötti résztömb rendezése. A támpont elem a 8. helyre kerül.

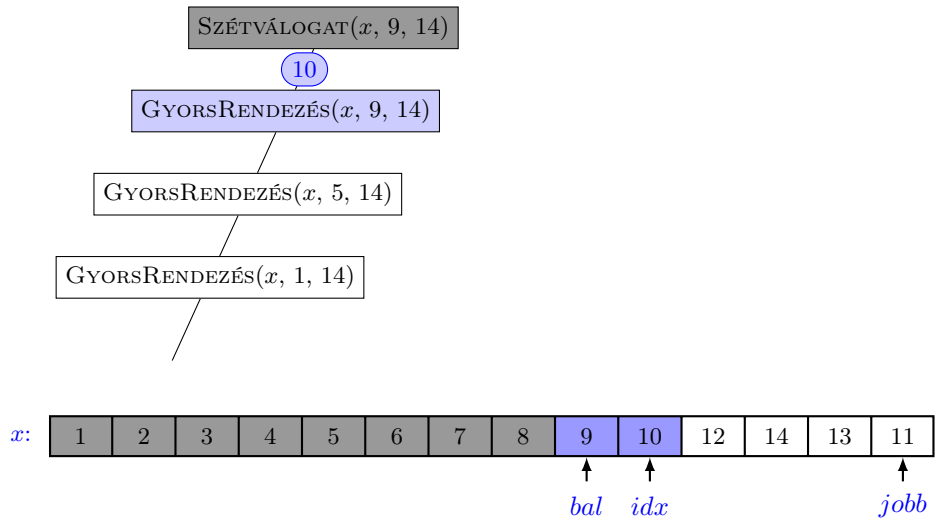


(e) Az 5. és 7. elem közötti résztömb rendezése. A támpont elem az 5. helyre kerül, ezért csak a jobb oldali részt kell a továbbiakban rendezni.

6.4. ábra. Gyorsrendezés (folyt.)

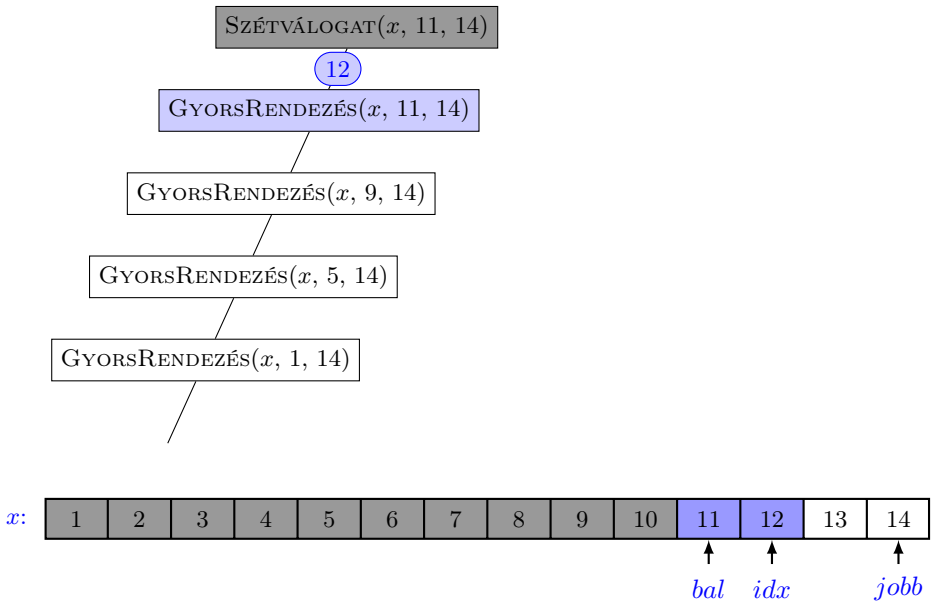


(f) A 6. és 7. elem közötti résztömb rendezése. A támpont elem a 6. helyre kerül. További rendezés itt már nem szükséges.

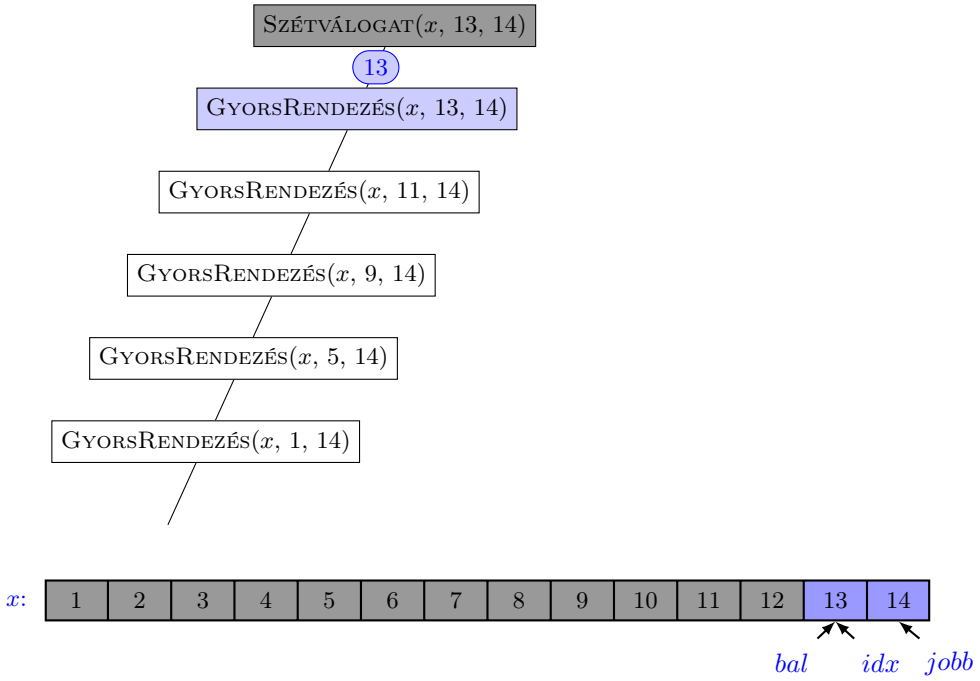


(g) A 9. és 14. elem közötti résztömb rendezése. A támpont a 10. helyre kerül. A bal oldali résztömb egy elemű, ezért csak a jobb oldali részt kell a továbbiakban rendezni.

6.4. ábra. Gyorsrendezés (folyt.)



(h) A 11. és 14. elem közötti résztömb rendezése. A támpont a 12. helyre kerül. Mivel a bal oldali részt egy elemű, ezért csak a jobb oldali részt kell a továbbiakban rendezni.



(i) A 13. és 14. elem közötti résztömb rendezése. A támpont a 13. helyre kerül. Mivel a jobb oldali résztömb is csak egy elemű, ezért további rendezés már nem szükséges.

6.4. ábra. Gyorsrendezés (folyt.)

Megjegyzés

Miként lehet a gyorsrendezésnél arra törekedni, hogy a futási idő minden esetben $n \log n$-nel legyen arányos? Erre többféle módszer is van, amelyek közül csak egyet szeretnénk megemlíteni. Rendezett tömböknél az a probléma, hogy ha mindig az első elemet választjuk támpontnak, akkor a lehető legrosszabb felosztását kapjuk az aktuális résztömbnek. Ezen javíthatunk, ha a támpontot a vizsgált résztömbből véletlenszerűen választjuk ki.

A k-adik legkisebb elem kiválasztása

Feladatunk, hogy egy tömbben megkeressük a valahányadik legkisebb elemet. Ha például az első legkisebb elem kell, akkor ezt egyszerűen meghatározhatjuk a minimumkiválasztás használatával. Hasonlóan egy n elemű tömb n-edik legkisebb elemét a maximumkiválasztással adhatjuk meg. Ha viszont nem egy szélsőértékre van szükségünk, hanem egy általános k-adik legkisebb elemre (például az ötödikre), akkor nehezebb dolgunk van.

A problémát eddigi ismereteink alapján is megoldhatjuk. Amennyiben a feladatunk az ötödik legkisebb elem meghatározása, akkor eljárhatunk a következő lépéseket követő módszerrel. Minimumkiválasztással meghatározzuk először a legkisebb elemet, majd ezt elhagyjuk a tömbből. Ezt követően a maradék elemek között ismét meghatározzuk a legkisebbet, ami az eredeti tömb második legkisebb eleme. Majd ezt az elemet is elhagyjuk a tömbből. Ezt folytathatjuk mindaddig, amíg az ötödik legkisebb elemhez nem jutunk. Futási idő szempontjából egy minimumkiválasztás arányos a vizsgált tömb méretével. Tehát, ha például a tömb középső elemét (mediánját) kell megadnunk, akkor a futási idő $O\left(n^2\right)$-es.

Másik lehetséges megközelítés, ha növekvő sorrendbe rendezzük a tömb elemeit, majd kiválasztjuk a k-adik elemet. A rendezést jelenlegi ismereteink szerint leggyorsabban $O(n \log n)$ idő alatt végezhetjük el. Kérdéses, hogy van-e ennél gyorsabb lehetőség a k-adik legkisebb elem kiválasztására.

A bemutatásra kerülő módszer a gyorsrendezésnél már megismert szétválogatáson (ld. 6.5. algoritmus) alapul. A vizsgált tömbünket először szétválogatjuk, így megkapjuk, hogy a támpont elem hova kerül a tömbben. Jelölje ennek a helynek az indexét idx. Ha a k érték egyenlő az idx-szel, akkor a k-adik legkisebb elem maga a támpont elem, hiszen a szétválogatásnál a támpont elem a végleges helyére kerül egy rendezett tömbben. Ha k kisebb az idx-nél, akkor a további keresést csak az idx-edik elem előtti résztömbben kell folytatnunk, mert ott találhatók a támpontnál kisebb értékű elemek. Ha viszont k nagyobb az idx-nél, akkor a támpontot követő elemek között kell keresnünk. Itt viszont már nem a k-adik, hanem a $(k - idx)$-edik elemet kell megtalálnunk.

Az előbbi „Oszd meg és uralkodj!" elvű ötletet a 6.6. algoritmusban írjuk le. Az algoritmus bemenete a vizsgált x tömb, az aktuálisan vizsgált résztömb bal és $jobb$ határindexei, valamint a k érték. Kimenetként a k-adik legkisebb elem értékét kapjuk meg. Az algoritmust megvalósító K-ADIKLEGKISEBBELEM rekurzív függvényt $bal = 1$ és $jobb = n$ paraméterekkel hívjuk meg.

> **Megjegyzés**
>
> Azért nem a k-adik legkisebb elem helyét határozzuk meg, mert a SZÉTVÁLOGAT függvényben a tömb elemeinek helye változik. Viszont a K-ADIKLEGKISEBBELEM függvény az őt hívó külvilág felé nem ad vissza módosított tömböt, így az eredeti tömbben valószínűleg máshol lesz a megtalált elem, mint a SZÉTVÁLOGAT függvény által módosított tömbben.

Az algoritmust megvalósító függvény 2. sorában megvizsgáljuk, hogy a vizsgált résztömb egy elemű-e. Ha igen, akkor nem kell további keresést végeznünk, ezért visszaadjuk az egyetlen elem értékét (ld. 3. sor). Amennyiben több elemű a vizsgált résztömb (ld. 4. sor), akkor az aktuális résztömb első elemét ($x[bal]$) támpontnak tekintve elvégzünk egy szétválogatást. A szétválogatás megadja a támpont helyét, valamint a tömböt átalakítja a korábban már megismert módon (ld. 5. sor). Megvizsgáljuk, hogy a támpont elem a vizsgált résztömb hányadik helyére került, amit konkrétan az $idx - bal + 1$ határoz meg. Ha ez a hely megegyezik a k-val (ld. 6. sor), akkor megtaláltuk a keresett értéket, ezért visszaadjuk azt kimenetként (ld. 7. sor). Amennyiben a k érték kisebb a támpont résztömbön belüli helyénél (ld. 8. sor), akkor további keresést kell végrehajtanunk a résztömb támpont előtti elemei között. Ennek érdekében rekurzív hívást hajtunk végre a megfelelő paraméterekkel (ld. 9. sor). A rekurzívan hívott függvény visszatérési értéke lesz az aktuális függvény visszatérési értéke is. Amennyiben viszont a k érték nagyobb a támpont résztömbön belüli helyénél (ld. 10. sor), akkor a jobb oldali elemek között kell további keresést végezni. Viszont ekkor már nem a k-adik, hanem a $(k - (idx - bal + 1))$-edik elemet kell kiválasztani, tehát a rekurzív hívás paramétereit is ennek megfelelően adjuk meg (ld. 11. sor).

6.4. Példa. Érdemes végigkövetnünk az algoritmus működését egy konkrét példán is. Tekintsük a 6.5. ábran látható tizennégy elemű résztömböt, amelyből az ötödik legkisebb elemet szeretnénk kiválasztani.

Meghívjuk a K-ADIKLEGKISEBBELEM függvényt a $bal = 1$, $jobb = 14$ és $k = 5$ paraméterekkel. Mivel a $bal \neq jobb$ ezért a vezérlés a SZÉTVÁLOGAT függvény hívására ugrik. A szétválogatás átrendezi a

6.6. Algoritmus k-adik legkisebb elem kiválasztása

Bemenet: x – **T tömb**, bal – **egész**, $jobb$ – **egész**, k – **egész**; ahol **T** összehasonlítható
Kimenet: k-adik legkisebb tömbelem értéke

```
1: függvény K-ADIKLEGKISEBBELEM(x : T tömb, bal : egész, jobb : egész, k : egész)
2:     ha bal = jobb akkor
3:         vissza x[bal]
4:     különben
5:         idx ← SZÉTVÁLOGAT(x, bal, jobb)
6:         ha k = idx − bal + 1 akkor
7:             vissza x[idx]
8:         különben ha k < idx − bal + 1 akkor
9:             vissza K-ADIKLEGKISEBBELEM(x, bal, idx − 1, k)
10:        különben
11:            vissza K-ADIKLEGKISEBBELEM(x, idx + 1, jobb, k − (idx − bal + 1))
12:        elágazás vége
13:    elágazás vége
14: függvény vége
```

Függvény hívása: K-ADIKLEGKISEBBELEM(x, 1, n, k)

Felhasznált változók és függvények

- x: Tömb, melyből a k-adik legkisebb elem értékét akarjuk kiválasztani.
- bal: Az x tömb aktuálisan vizsgált részének alsó indexe.
- $jobb$: Az x tömb aktuálisan vizsgált részének felső indexe.
- k: A k-adik legkisebb elemet akarjuk kiválasztani az x tömbből.
- idx: A gyors rendezésnél megismert SZÉTVÁLOGAT függvény kimenete.
- n: Az x tömb elemszáma.

tömböt úgy, hogy az aktuális támpont elem a negyedik helyre kerül (ld. 6.5a. ábra). Ezek alapján az ötödik legkisebb elemet a támponttól jobbra lévő elemek között tudjuk megtalálni, viszont már nem az ötödik, hanem az első legkisebb elem kiválasztása a cél.

Meghívásra kerül ismételten a K-ADIKLEGKISEBBELEM függvény a $bal = 5$, $jobb = 14$ és $k = 1$ paraméterekkel. Következik a SZÉTVÁLOGAT függvény hívása, amely a támpontot a 8. helyre (a vizsgált résztömbön belül a 4. helyre) teszi (ld. 6.5b. ábra). Mivel az első legkisebb elemet keressük, ezért a résztömb támpont előtti elemei között kel a keresést folytatnunk.

Ismét a K-ADIKLEGKISEBBELEM függvény hívása következik, de most a $bal = 5$, $jobb = 7$ és $k = 1$ paraméterekkel. A szétválogatás az aktuális támpontot az 5., az aktuális résztömbön belül pedig az 1. helyre teszi. Így megtaláltuk a keresett elemet, ezért a rekurzívan hívott függvények ennek értékével térnek vissza (ld. 6.5c. ábra). ¶

Futási idő elemzése. A futási idő szempontjából vizsgáljuk meg először a legkedvezőbb esetet. Ez természetesen az, ha az első szétválogatásnál a támpont rögtön a k-adik helyre kerül. Ilyenkor egyetlen szétválogatást kell végezni az n elemű tömbben, aminek futási ideje $O(n)$-es.

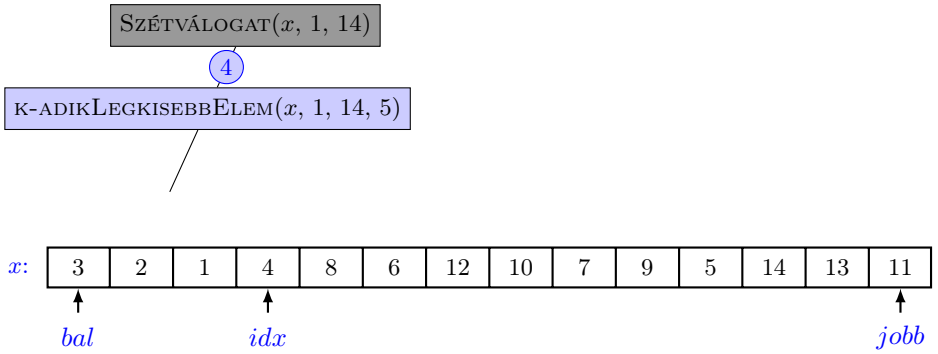
Vizsgáljuk most azt az esetet, amikor több rekurzív hívás, így több szétválogatás is történik. Legrosszabb esetben egészen addig kell a rekurziót folytatnunk, amíg a vizsgált résztömb egy eleműve nem válik. Ha közben a még vizsgálat alatt lévő tömb mérete mindig feleződik, azaz a támpont középre kerül, akkor a futási időről a következőt mondhatjuk:

$$T(n) = O(n) + O\left(\frac{n}{2}\right) + O\left(\frac{n}{4}\right) + \ldots + O(1) = O\left(n + \frac{n}{2} + \frac{n}{4} + \ldots + 1\right). \tag{6.11}$$

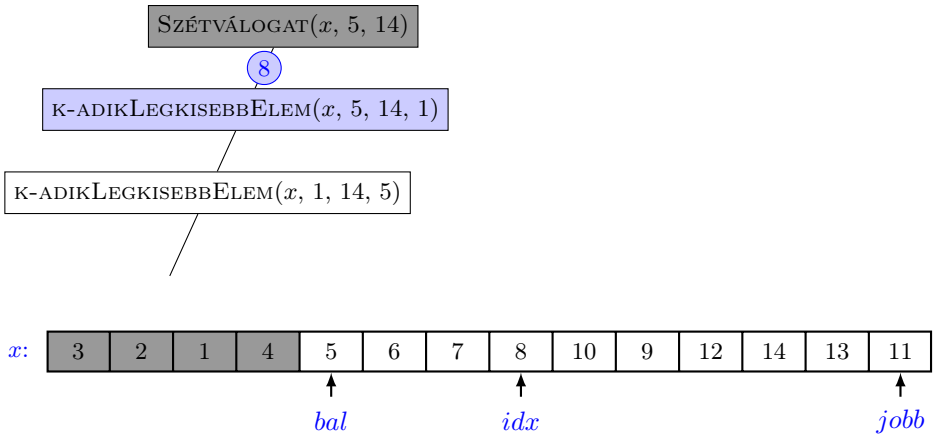
Az $n + \frac{n}{2} + \frac{n}{4} + \ldots + 1$ összeg egy mértani sorozat összege, melynek első eleme 1, kvóciense 2, elemszáma pedig megközelítőleg $1 + \log_2 n$. Így viszont

$$T(n) = O\left(1 \cdot \frac{2^{1+\log_2 n} - 1}{2 - 1}\right) = O(2n - 1) = O(n). \tag{6.12}$$

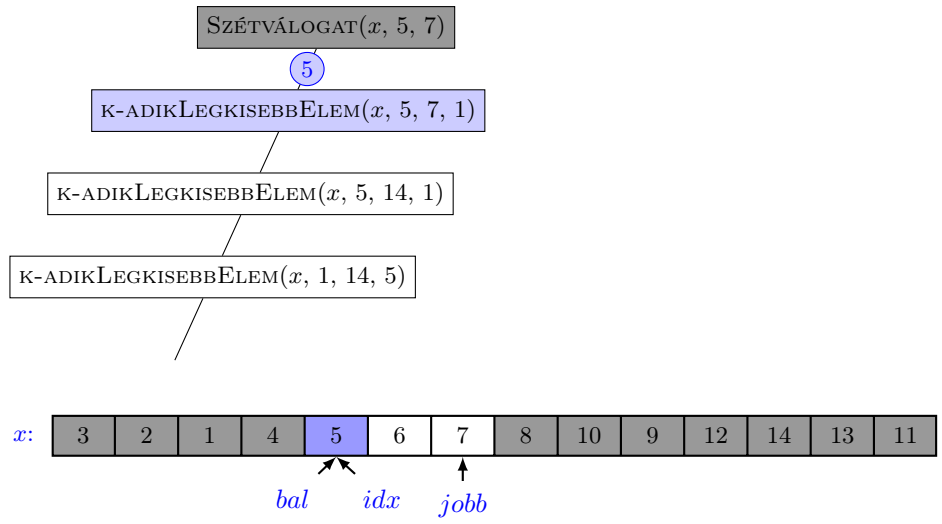
Az algoritmus futásának legrosszabb esete természetesen most is az, ha a támpont elem mindig a vizsgált tömb valamelyik szélére kerül, ilyenkor a futási idő $O\left(n^2\right)$-es.



(a) Az 1. és 14. elemek között végzünk keresést. A támpont elem a 4. helyre kerül, ezért az ötödik legkisebb elemet a támpont jobb oldalán kell tovább keresnünk.



(b) Az 5. és 14. elemek között keresünk, de most már az első legkisebb elemet. A támpont a 8. helyre kerül, ezért a bal szomszédjai között folytatjuk majd a vizsgálatot.



(c) Az 5. és 7. elemek között keressük az első legkisebb elemet. Mivel a támpont az 5. helyre kerül, ami a vizsgált résztömb első helye, ezért megtaláltuk a keresett elemet.

6.5. ábra. Ötödik legkisebb elem kiválasztása.

Bizonyítás nélkül közöljük, hogy a k-adik legkisebb elem kiválasztását megvalósító algoritmusunk átlagos futási ideje $O(n)$-es, tehát a fejezet elején ismertetett két másik megközelítésnél jelentősen jobb futási idővel rendelkezik. ♣

7. fejezet

Optimalizálási problémák

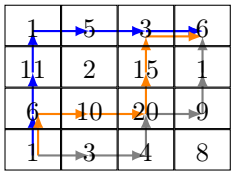
Az optimalizálási problémák céljukban jelentősen különböznek minden eddig tárgyalt algoritmustól. A korábbi fejezetek algoritmusainál mindig volt egy konkrét problémánk, amelynek a megoldását kerestük. Kettős cél lebegett a szemünk előtt. Egyrészt egy olyan megoldást kívántunk adni, ami minden lehetséges bemenet esetén helyesen oldja meg a felmerült problémát. Másrészt arra koncentráltunk, hogy a kifejlesztett algoritmus a lehető leggyorsabb, illetve legkisebb memória igényű legyen.

Az optimalizálási problémáknál nem az a kifejezett célunk, hogy megoldást találjunk egy adott feladatra. (Persze azért nem árt az sem, ha megoldást tudunk adni.) Ehelyett az adott probléma sok lehetséges megoldása közül szeretnénk valamilyen szempontból a legjobbat, az optimálisat megadni.

Tekintsünk egy konkrét feladatot. Egy mezőn kincseket ástak el. A kincseket rendezetten egy négyzetrács ismert pontjaiban helyezték el. Minden kincsnek ismerjük a pontos helyét és az értékét is, ahogy ezt a 7.1a. ábra szemlélteti. A „kincsmező” bal alsó sarkából elindulva el kell jutnunk a jobb felső sarkába. Bejárásunk során csak fel, illetve jobbra haladhatunk. Ahol kincset találunk, ott azt be is gyűjtjük.

1	5	3	6
11	2	15	1
6	10	20	9
1	3	4	8

(a) A „kincsmező”. Zöld mező: kiindulási pont, piros mező: érkezési pont.



(b) Néhány lehetséges bejárás.

7.1. ábra. Kincsek begyűjtése. A „kincsmező” bal alsó sarkából indulva kell eljutnunk a jobb felső sarokba úgy, hogy közben csak felfelé vagy jobbra léphetünk.

Feladatunk most nem az, hogy találjunk egy lehetséges olyan bejárást, amivel eljuthatunk a bal alsó sarokból a jobb felsőbe. (A korábbi fejezetekben megismert algoritmusok ilyen jellegű problémákra adtak megoldásokat.) Bejárási lehetőség egyébként számos létezik, ezek közül néhányat különböző színű nyilakkal feltüntettünk a 7.1b. ábrán. Optimalizálási probléma megoldása során azt kell meghatároznunk, hogy a létező bejárások közül melyik az, amelyik esetén a lehető legtöbb kincset tudjuk begyűjteni. Ezt úgy is fogalmazhatjuk, hogy van a probléma során egy függvényünk és olyan megoldást keresünk, amely ezen függvény szélsőértékét (maximumát vagy minimumát) képes előállítani. Esetünkben ez a függvény a bejárás során összegyűjtött kincsek összértéke és ennek a függvénynek a maximalizálása a célunk. A 7.1b. ábra három lehetséges bejárása során, ha a kék nyilakat követjük, akkor 33 kincset gyűjtünk be. A narancssárga nyilak által mutatott bejárás esetén 61 kincsünk lesz, míg a szürke nyilak mentén haladva 44 kincset tudunk összegyűjteni. Így a három lehetőség közül biztos, hogy a narancssárga nyilak mentén fogunk haladni. Persze ez még nem biztosítja, hogy ez az optimális megoldása a feladatnak.

Az optimális megoldás keresése során gondolkodhatnánk úgy, hogy először meghatározzuk az összes lehetséges megoldást, majd mindegyik megtalált megoldás esetén kiszámítjuk az optimalizálandó függvény értékét. Ezt követően már csak az az esetet kell egy maximumkiválasztással megtalálni, amikor a

függvény értéke a legnagyobb (vagy a legkisebb). Ezt a megközelítést *nyers erő*[1] módszernek nevezzük, mert „gondolkodás nélkül", nyers erővel nekiesünk a feladatnak és megoldjuk. Ilyenkor általában nagyon lassan működő algoritmusokat kapunk eredményül.

A fejezetben két konkrét megközelítést mutatunk be, melyek optimalizálási problémák megoldására használhatók. Elsőkét a 7.1. fejezetben a *Dinamikus programozás*[2] módszert ismertetjük. Ez a módszer a megoldandó feladatot részfeladatokra bontja – ilyen szempontból hasonlít az „Oszd meg és uralkodj!" elvű algoritmusokra –, majd ezek alapján határozza meg az optimális megoldást. Második tárgyalt megközelítés a 7.2. fejezetben részletezett *Mohó algoritmus*[3]. Ez a módszer a problémát részproblémákra osztja, de nem minden részproblémát old meg, csak az éppen optimálisnak tűnőt.

[1] Angolul: brute force
[2] Angolul: dynamic programming
[3] Angolul: greedy algorithm

Dinamikus programozás

A dinamikus programozás az „Oszd meg és uralkodj!” elvhez hasonlóan a megoldandó feladatot részfeladatokra bontással oldja meg. Lényeges különbség viszont, hogy az „Oszd meg és uralkodj!” elv a feladatot egymástól független részfeladatokra bontotta (fentről lefele építkezett), melyeket rekurzívan megoldott, majd a megoldásokat egyesítette. A dinamikus programozás viszont akkor használható, ha a részproblémák egymástól nem függetlenek, azaz közös részproblémák vannak. Ilyenkor is lehetne persze az „Oszd meg és uralkodj!” elvet használni, de a közös részproblémákat az többször is megoldaná. Ezzel szemben a dinamikus programozásnál minden egyes részproblémát csak egyszer oldunk meg (lentről felfelé építkezünk), a megoldást pedig egy táblázatban tároljuk el. A részprobléma megoldását felhasználó további feladatrészek ebből a táblázatból tudják kiolvasni a szükséges értékeket.

Megjegyzés

Fontos megemlíteni, hogy a módszer neve egyáltalán nem fejezi ki a dinamikus programozás lényegét. Egyrészt nem számítógépes programot írunk a dinamikus programozás használata során, hanem egy táblázaton alapuló módszert adunk meg. Másrészt a felépítendő táblázat elemei sem dinamikusan változnak, hanem egy értéket egyszer meghatározunk, utána pedig már nem módosítjuk.

A fejezetet három fő részre osztjuk. Először egy konkrét példát mutatunk be, az ún. *0-1 hátizsák feladat*ot[4] (ld. 7.1.1. alfejezet). Ezt a feladatot megoldjuk a dinamikus programozás módszerével és közben rámutatunk a módszer legfontosabb elemeire. A 7.1.2. alfejezetben összefoglaljuk a dinamikus programozás elvét. A fejezet zárásaként a 7.1.3. alfejezetben a *leghosszabb közös részsorozat*[5] probléma megoldását mutatjuk be.

A dinamikus programozási technikánál általában táblázatokban tárolunk el adatokat. Ezek tekinthetők kétdimenziós tömböknek is. A kétdimenziós tömb annyiban különbözik az eddig megismert tömböktől, hogy nem csak egy, hanem két index szükséges egy tömbbeli elem eléréséhez. Minden tömbbeli elem rendelkezik egy sor- és egy oszlopindexszel. Az x tömb i-edik sorában és j-edik oszlopában lévő elemre az $x[i, j]$ szintaktikával hivatkozhatunk. Amikor kétdimenziós tömböt hozunk létre, akkor is szükséges a LÉTREHOZ függvényt meghívni, de a tömb méreténél két értéket, a tömb sorainak és oszlopainak számát is meg kell adni.

A fejezetben a dinamikus programozás egyszerűbb leírása érdekében a kétdimenziós tömbök indexelése ebben a fejezetben nem 1-től, hanem 0-tól indul. Ez igaz mind a sor-, mind az oszlopindexekre. Az ilyen speciális tulajdonságú kétdimenziós tömböket jelen jegyzetben röviden táblának, a létrehozáshoz szükséges függvényt pedig TÁBLALÉTREHOZ-nak nevezzük. Az algoritmusok leírásánál mindig megadjuk, hogy az adott táblában milyen típusú elemeket tárolunk el, hasonlóan a tömbök korábban bevezetett leírásához.

[4] Angolul: 0-1 knapsack problem
[5] Angolul: longest common subsequence

0-1 hátizsák probléma

Egy rabló ellátogat a királyi kincstárba. Nagy mennyiségű kincset talál ott, de a „hátizsákjába" csak egy súlykorlátot nem meghaladó mennyiségű kincset tehet, mivel egyébként nem bírja el azokat. Hogyan tud a rabló úgy kiválasztani kincseket, hogy a kiválasztottak beférjenek a hátizsákba, és emellett az összértékük a lehető legnagyobb legyen?

A feladat megoldása érdekében először formalizáljuk a feladatot.

Adott egy n elemű K halmaz, mely a kincstárban lévő kincsek halmaza. Az K halmaz elemei érték és súlypárok. Tehát az K halmaz i-edik eleme: (p_i, w_i), ahol p_i az i-edik kincs értéke, w_i pedig az i-edik kincs súlya. A rabló hátizsákjának kapacitása legfeljebb c. (Az egyszerűség kedvéért minden p_i és w_i érték, valamint c is pozitív egész szám.) Ki kell választani az K halmaznak egy olyan S részhalmazát, hogy

$$\sum_{j \in S} w_j \leq c, \tag{7.1}$$

azaz a kiválasztott kincsek összsúlya a hátizsák kapacitását nem haladja meg, valamint

$$\sum_{j \in S} p_j \tag{7.2}$$

maximális.

Hogyan lehetne megoldani a feladatot? Első ötletünk, hogy megvizsgáljuk az összes lehetséges kiválasztások halmazát. Mivel egy n elemű halmaznak 2^n darab részhalmaza van, ezért ez 2^n lehetőséget jelent. Megnézzük, hogy ezen lehetőségek közül melyik „fér bele" a hátizsákba, majd ezek közül kiválasztjuk a legnagyobb összértékűt. Ez az ötlet az ún. nyers erő módszert követi, aminek futási ideje $O(2^n)$-es, tehát elég lassú. Tudunk ennél hatékonyabb módszert is találni?

Vezessünk be egy új jelölést, aminek segítségével leírhatunk egy más jellegű megközelítést. Jelölje $F[i, x]$ az első i darab kincs közül a legjobb (azaz legnagyobb összértékű) kiválasztás összértékét, ahol a kincsek összsúlya legfeljebb x. Mivel n darab kincsünk van, ezért $1 \leq i \leq n$, x lehetséges értékei pedig: $0, 1, 2, \ldots, c$.

Benne lehet-e az $F[i, x]$ kiválasztásban az i-edik kincs? Ez elsődlegesen attól függ, hogy milyen a súlya. Ha $w_i > x$, akkor biztos nincs benne, mert így az összsúly is meghaladná x-et, ami nem lehetséges. Ha viszont $w_i \leq x$, akkor sem biztos, hogy benne van az $F[i, x]$ kiválasztásban az i-edik kincs. Meg kell vizsgálni, hogy az $F[i-1, x]$ és az $F[i-1, x-w_i] + p_i$ érték hogyan viszonyul egymáshoz? Az $F[i-1, x]$ jelenti az első $i-1$ kincs közül a legjobb választás összértékét úgy, hogy az összsúly legfeljebb x. Az $F[i-1, x-w_i]$ pedig jelenti az első $i-1$ kincs közül a legjobb választást úgy, hogy az összsúlyuk legfeljebb $x - w_i$. Ha ehhez hozzávesszük az i-edik kincset is, akkor a kiválasztott kincsek összértéke $F[i-1, x-w_i] + p_i$ lesz, az összsúly pedig legfeljebb $(x - w_i) + w_i$, azaz legfeljebb x. Ezek szerint az $F[i-1, x]$ az az eset, amikor legfeljebb x súlynyi kincset az első $i-1$ kincs közül választottunk, $F[i-1, x-w_i] + p_i$ pedig azt jelenti, amikor az i-edik kincset biztosan kiválasztottuk, valamint az első $i-1$ kincs közül is választottunk annyit, hogy az i-edik kincset hozzávéve az összsúly x-nél ne legyen nagyobb. Tehát akkor kijelenthető, hogy $w_i \leq x$ esetben azt kell megnéznünk, hogy $F[i-1, x]$ és $F[i-1, x-w_i] + p_i$ közül melyik eredményez nagyobb összértéket, a súly pedig mindkét esetben maximum x lesz.

Mindezek alapján kijelenthető, hogy $F[i, x]$ az alábbi képlettel definiálható:

$$F[i, x] = \begin{cases} F[i-1, x], & \text{ha } w_i > x, \\ \max\{F[i-1, x], F[i-1, x-w_i] + p_i\}, & \text{ha } w_i \leq x. \end{cases} \tag{7.3}$$

Ez egy rekurzív formula, amit úgy tehetünk teljessé, ha pontosan definiáljuk az $i = 0$ alapesetet is. Mivel ez azt jelenti, hogy nem választottunk ki kincset a halmazból, ezért $F[0, x] = 0$. Könnyen belátható az is, hogy $F[i, 0] = 0$, mivel ez azon kiválasztás összértékét jelöli az első i darab kincs közül, amikor az összsúly legfeljebb 0. A feladat megoldása érdekében az $F[n, c]$ értéket kell kiszámítanunk és máris tudjuk, hogy mekkora az optimális választás mellett a kiválasztott kincsek összértéke.

Bár rekurzív képletet adtunk, de a megoldásra nem rekurzív algoritmust fogunk írni. Ennek az az oka, hogy ugyanazt az értéket többször is ki kéne értékelnünk, ahogy azt például a Fibonacci sorozat rekurzív megvalósításánál (ld. 4.3. fejezet) tettük. Ehelyett az $F[i, x]$ értékeket egy táblázatban tároljuk el úgy, hogy minden értéket pontosan egyszer számítunk csak ki. A táblázatot úgy töltjük fel, hogy először a

rekurzív alapesetnek megfelelő értékeket tároljuk el. Ezek egyrészt minden lehetséges x érték mellett az $F[0, x]$ értékek, mivel ha egyetlen tárgyat sem választunk ki, akkor a kiválasztott tárgyak összértéke 0. Tehát minden megengedett x értéke esetén tudjuk, hogy $F[0, x] = 0$. Emellett könnyen látható, hogy minden lehetséges i esetén $F[i, 0] = 0$, mivel a legfeljebb 0 összsúlyú kiválasztások esetén sincs egyetlen kincs sem a zsákban, hiszen minden kincsnek 0-nál nagyobb a súlya. Az alapesetek megadását követően már csak sorfolytonosan be kell járni a táblát és a 7.3. képletnek megfelelően meg kell határozni az aktuális $F[i, x]$ értéket. A konkrét megvalósítást a 7.1. algoritmusban adjuk meg.

7.1. Algoritmus 0-1 hátizsák probléma

Bemenet: p – **egész tömb**, w – **egész tömb**, n – **egész** (*tömb mérete*), c – **egész**
Kimenet: F – **egész tábla**

```
 1: függvény 0-1Hátizsák(p : egész tömb, w : egész tömb, n : egész, c : egész)
 2:     F ← TáblaLétrehoz(egész)[n + 1, c + 1]
 3:     ciklus x ← 0-től c-ig
 4:         F[0, x] ← 0
 5:     ciklus vége
 6:     ciklus i ← 1-től n-ig
 7:         F[i, 0] ← 0
 8:     ciklus vége
 9:     ciklus i ← 1-től n-ig
10:         ciklus x ← 1-től c-ig
11:             ha w_i ≤ x akkor
12:                 F[i, x] ← max(F[i − 1, x], F[i − 1, x − w_i] + p_i)
13:             különben
14:                 F[i, x] ← F[i − 1, x]
15:             elágazás vége
16:         ciklus vége
17:     ciklus vége
18:     vissza F
19: függvény vége
```

Felhasznált változók és függvények

- p: Tömb, melyben az egyes kincsek értékét tároljuk.
- w: Tömb, melyben az egyes kincsek súlyát tároljuk.
- n: A p és w tömbök mérete.
- c: A hátizsák kapacitása.
- F: Tábla, melynek mérete $(n + 1) \times (c + 1)$. $F[i, x]$ jelenti az első i darab kincs közül kiválasztott kincsek legnagyobb összértékét, ha a kiválasztott kincsek összsúlya legfeljebb c.
- TáblaLétrehoz(**egész**)$[n + 1, c + 1]$: Utasítás, mely létrehoz egy $(n + 1) \times (c + 1)$ méretű **egész** típusú táblát.

Az algoritmus bemenete az egyes kincsek értékeit tartalmazó p tömb, valamint a súlyok w tömbje. Mindkét tömb elemszáma azonos, ezt adja meg az n bemeneti változó. Természetesen a p és w tömb megfelelő elemei ugyanazon kincs értékét és súlyát jelölik. Bemeneti változóként meg kell még adni a hátizsák c kapacitását. A tömb kimenete a 7.3. képletnek megfelelően előállított tábla, melynek mérete $(n + 1) \times (c + 1)$-es.

A 7.1. algoritmusban először létrehozzuk a kimeneti táblát (ld. 2. sor). Ezután feltöltjük a kezdeti 0 értékekkel az F tábla nulladik sorának (ld. 3-5. sorok) és nulladik oszlopának (ld. 6.-8. sorok) minden egyes elemét. A 9. sorban kezdődő két egymásba ágyazott ciklusban a 7.3. képletnek megfelelően meghatározzuk az F tábla egyes elemeinek értékét. Az algoritmus végén az F táblát kapjuk meg kimenetként.

Futási idő elemzése. Könnyen látható, hogy a 7.1. algoritmus futási ideje $O(n \cdot c)$-s, tehát jóval gyorsabb, mint a fejezet elején említett $O(2^n)$-es futási idejű nyers erő módszer. ♣

7.1. Példa. A 7.1. algoritmus működését nyomon követhetjük egy konkrét példán is. A kincsek száma: $n = 4$. Az egyes kincsek konkrét érték és súly értékeit a 7.1. táblázatban adtuk meg, a hátizsák

kapacitása pedig legyen 5. A 7.2. ábrán végigkísérhető az algoritmus működése. Az ábrán kék színnel jelezzük a tábla sor- és oszlopindexeit. ¶

i	p_i	w_i
1	3	2
2	4	3
3	5	4
4	6	5

7.1. táblázat. A 7.1. feladat kincseinek konkrét értékei és súlyai.

		x					
		0	1	2	3	4	5
i	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
	2	0	0	3	4	4	7
	3	0	0	3	4	5	7
	4	0	0	3	4	5	7

7.2. ábra. 0-1 hátizsák probléma megoldása dinamikus programozással. Az F tábla elemeinek előállítása.

Az F tábla ismeretében már meghatározható, hogy pontosan melyik kincseket kell kiválasztanunk. Az $F[n, c]$ elem – azaz a jobb alsó sarokban lévő elem – megadja, hogy az n elem közül kiválasztva a legnagyobb összértéket eredményező legfeljebb c összsúlyú elemeket mivel egyenlő az összérték. Az F tábla jobb alsó sarkából indulva egy jól meghatározott bejárással megadható, hogy mely kincsek kiválasztása szükséges. A 7.1. algoritmusból tudjuk ugyanis, hogy mikor került egy elem a kiválasztottak közé. Tehát azt kell csak vizsgálni, hogy egy kincset hozzávettünk-e a legjobb kiválasztáshoz vagy sem. Ezt pedig onnan tudjuk megmondani, hogy egy adott elem „fölötti" elemnek mi az értéke az F táblázatban.

A kiolvasás megvalósítását a 7.2. algoritmussal adjuk meg. Az algoritmus bemenete az F tábla, melynek mérete $(n + 1) \times (c + 1)$. Kimenete pedig az optimális kiválasztásban szereplő kincsek indexeit tartalmazó S halmaz.

A kimeneti S halmaz kezdetben üres (ld. 2. sor). Az F tábla bejárását a bal alsó sarokból kezdjük, ezért i kezdetben a kincsek n számával (ld. 3. sor), x pedig a c kapacitással egyenlő (ld. 4. sor). Az F tábla bejárását addig folytatjuk, amíg i és x is pozitív, ezt valósítja meg az 5. sorban kezdődő ciklus. Megvizsgáljuk, hogy az F tömb aktuális $F[i, x]$ eleme megegyezik-e a „fölötte" lévővel (ld. 6. sor). Ha nem egyezik meg, akkor az i-edik kincset hozzávettük az optimális megoldáshoz, ezért az S halmazba betesszük az i indexet (ld. 7. sor). Mivel az i-edik elem súlya w_i, ezzel csökkenteni kell a betehető x összsúlyt (ld. 8. sor). Minden esetben eggyel korábbi sorra kell visszalépni, ezért az i-t csökkenteni kell eggyel (ld. 10. sor). A ciklusból kilépve az S-ben pont az optimális megoldáshoz tartozó kincsek indexei lesznek, ezért S-t visszaadjuk eredményként (ld. 12. sor).

7.2. Példa. A 7.1. példánál létrehozott F tömb alapján határozzuk meg, hogy mely elemek kiválasztása lesz optimális. Az algoritmus működését szemlélteti a 7.3. ábra. A bejárást az $F[4, 5]$-ből indítjuk. Mivel $F[4, 5] = F[3, 5]$, ezért a negyedik kincs nincs az optimális kiválasztásban. Ugyanez igaz a harmadik kincs esetében is, mert $F[3, 5] = F[2, 5]$. Viszont $F[2, 5] \neq F[1, 5]$, ezért a második kincs benne van a kiválasztott elemek S halmazában. A második elem súlya $w_2 = 3$, ezért az x értéke 2-re csökken. Mivel $F[1, 2] \neq F[0, 2]$, ezért az első kincset is hozzávesszük a kiválasztottak S halmazához. További vizsgálat nem kell, mert elfogytak a kincsek. Így az első és második kincset választva kapjuk a legnagyobb összértékű kiválasztást $c = 5$ zsákkapacitás mellett. ¶

Futási idő elemzése. Könnyen belátható, hogy a 7.2. algoritmus futási ideje a kincsek számával arányos, tehát $O(n)$-es. Így a 0-1 hátizsák probléma optimális megoldásának teljes futási ideje is csak $O(n \cdot c)$-s. ♣

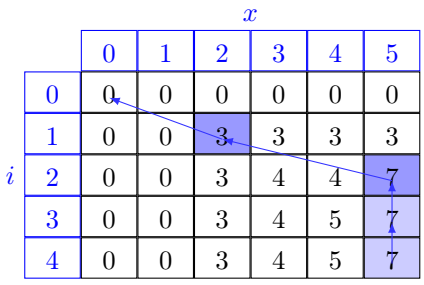
7.2. Algoritmus Kiválasztott elemek kiolvasása

Bemenet: F – **egész tábla**, n – **egész**, c – **egész**
Kimenet: S – **egész halmaz**

```
1: függvény KIOLVAS(F : egész tábla, n : egész, c : egész)
2:     S ← ∅
3:     i ← n
4:     x ← c
5:     ciklus amíg (i > 0) ∧ (x > 0)
6:         ha F[i, x] ≠ F[i − 1, x] akkor
7:             S ← S ∪ {i}
8:             x ← x − w_i
9:         elágazás vége
10:        i ← i − 1
11:    ciklus vége
12:    vissza S
13: függvény vége
```

Felhasznált változók és függvények

- F: Tábla, melynek mérete $(n+1) \times (c+1)$. $F[i, x]$ jelenti az első i darab kincs közül kiválasztott kincsek legnagyobb összértékét, ha a kiválasztott kincsek összsúlya legfeljebb c.
- n: A kincsek száma.
- c: A hátizsák kapacitása.
- S: Az optimális kiválasztásban szereplő kincsek indexeinek halmaza.



i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

7.3. ábra. 0-1 hátizsák probléma megoldása dinamikus programozással. Az optimális kiválasztás előállítása.

Érdemes végignézni, hogy milyen lépéseket hajtottunk végre a 0-1 hátizsák probléma ismertetett megoldása során. Először jellemeztük az optimális megoldás szerkezetét. Második lépésben rekurzív módon definiáltuk az optimális megoldás értékét. Ezután a rekurzív definíció alapján kiszámítottuk az optimális megoldás értékeit, de nem rekurzív algoritmussal, hanem ún. alulról felfelé történő módon. Végül a kiszámított információk alapján meghatároztunk egy optimális megoldást.

A dinamikus programozás elve

Egy példán keresztül megismertük a dinamikus programozást, amely az alábbi négy lépésből állt:

1. Az optimális megoldás szerkezetének jellemzése.
2. Az optimális megoldás értékének rekurzív módon történő definiálása.
3. Az optimális megoldás értékének kiszámítása ún. alulról felfelé történő módon.
4. Az előzőek alapján egy optimális megoldás megadása.

Felmerül viszont kérdésként, hogy milyen esetekben alkalmazható a dinamikus programozás elve. Az optimalizálási problémának két tulajdonságot kell ahhoz teljesítenie, hogy a dinamikus programozás módszerével megoldható legyen. Első az optimális részstruktúrák, második pedig az átfedő részproblémák tulajdonság.

Akkor mondjuk, hogy egy feladat *optimális részstruktúrájú*, ha a probléma egy optimális megoldása önmagán belül a részfeladatok optimális megoldásait is tartalmazza. A 0-1 hátizsák probléma megoldásánál ez teljesült, hiszen pont ezt használtuk ki a 7.3. képlet megadásánál.

A dinamikus programozás alkalmazhatóságának másik feltétele, hogy a rekurzív megoldó algoritmus (vagy képlet) mindig ugyanazokat a részfeladatokat oldja meg. Ilyenkor az optimalizálási feladat *részfeladatai átfedőek*. Ez egy lényeges különbség az „Oszd meg és uralkodj!" elvnél tárgyalt feladatokhoz képet. Azok ugyanis pont olyan részfeladatokat tartalmaztak, melyek egymástól teljesen függetlenek voltak, nem voltak közöttük átfedőek.

Az átfedő részfeladatok gyors megoldása érdekében a dinamikus programozásnál nem rekurzív függvényeket használunk, hanem az ún. *feljegyzéses módszer*t[6] alkalmazzuk. Ennek lényege, hogy a részproblémák megoldásait egy táblázatban tároljuk el. Ennek a táblázatnak minden egyes értékét csak egyszer számoljuk ki. Mivel a táblázatot úgy alkotjuk meg, hogy először a legelemibb részproblémákat oldjuk meg, majd ezekből építjük fel az egyre bonyolultabb problémák megoldásait, a dinamikus programozást „alulról felfelé" haladó módszernek tekintjük.

[6] Angolul: memoization

Leghosszabb közös részsorozat

Ahhoz, hogy két sorozat leghosszabb közös részsorozatát meg tudjuk határozni először definiálnunk kell néhány fogalmat.

Az $X = \langle x_1, x_2, \ldots, x_n \rangle$ egy sorozat, mely véges hosszú és a benne szereplő $x_1, x_2, \ldots, x_n$ elemek sorrendje adott. Ebből látható, hogy egy n elemű sorozat eltárolható egy n elemű tömbben.

Az X sorozat egy részsorozatát úgy kapjuk, ha X-ből valahány elemet törlünk, a megmaradó elemek sorrendjét pedig nem változtatjuk meg. Például az $\langle 1, 4, 8, 9, 12, 15, 17 \rangle$ sorozatnak részsorozata az $\langle 1, 8, 9, 15, 17 \rangle$ sorozat, amit a második és ötödik elem elhagyásával kaptunk.

Megjegyzés

Az X sorozatnak részsorozata maga az X sorozat is, mert ezt úgy kapjuk meg, hogy semmit nem törlünk X-ből.

Az X sorozatnak részsorozata az üres sorozat is, mert ha X-ből minden elemet törlünk, akkor azt kapjuk.

Az eddigi definíciók alapján már meghatározhatjuk a jelen fejezetben tárgyalásra kerülő pontos feladatunkat. Adott két sorozat: $X = \langle x_1, x_2, \ldots, x_n \rangle$ és $Y = \langle y_1, y_2, \ldots, y_m \rangle$. Keressük meg a leghosszabb olyan részsorozatot, amely részsorozata X-nek és Y-nak is. Például az $X = \langle c, a, d, b, r, z \rangle$ és az $Y = \langle a, s, b, z \rangle$ sorozatok leghosszabb részsorozata az $\langle a, b, z \rangle$ sorozat.

A feladat ismeretében elkezdhetünk a megoldással foglalkozni. Természetesen most is nekieshetnénk a feladat megoldásának nyers erő módszerrel. Ekkor előállítanánk az X összes részsorozatát, amiből 2^n darab van. Majd ugyanígy meghatároznánk az Y összes részsorozatát, amelyből pedig 2^m darab létezik. Ezután megnéznénk, hogy X mely részsorozata egyezik meg Y valamely részsorozatával, majd ezek közül kiválasztanánk a leghosszabbat. Nem kell sokáig bizonygatni, hogy futási idő szempontjából ez egy nagyon nem hatékony megközelítés lenne. Válasszunk ezért más stratégiát!

Követendő módszerünk két egymástól jól elkülönülő lépésből fog állni. Először ezt határozzuk meg, hogy milyen hosszú lehet a leghosszabb közös részsorozat, majd ennek ismeretében határozunk meg egy konkrét, ilyen hosszúságú részsorozatot.

A leghosszabb közös részsorozat hosszának meghatározásánál nem a teljes X és Y sorozatot fogjuk minden esetben vizsgálni, hanem azoknak csak az első valahány elemével foglalkozunk egy adott lépésben. A vizsgálat érdekében bevezetjük az $F[i, j]$ jelölést. $F[i, j]$ alatt értjük az X sorozat első i darab eleméből és az Y sorozat első j darab eleméből álló sorozatok leghosszabb közös részsorozatának hosszát. Az egyszerűbb leírás érdekében bevezetjük az X^i és Y^j jelölést, ami a megfelelő sorozat első i illetve j darab eleméből álló részsorozatot jelöli. A feladat megoldása során az a célunk, hogy meg tudjuk határozni az $F[n, m]$ értéket, hiszen ez pont a teljes X és Y leghosszabb közös részsorozatának hosszát adja meg. $F[n, m]$ ismeretében rátérhetünk majd egy konkrét olyan közös részsorozat megkeresésére, melynek hossza megegyezik $F[n, m]$-mel.

Nézzük meg most, hogy milyen szabályszerűséget tudunk megadni az $F[i, j]$ értékekre. Ha $i = 0$ vagy $j = 0$, azaz X-nek vagy Y-nak a 0 hosszúságú elejét tekintjük, akkor a leghosszabb közös részsorozat hossza se lehet több 0-nál. Tehát ilyenkor $F[i, j] = 0$. Ha az X sorozat i-edik eleme (x_i) és az Y sorozat j-edik eleme (y_j) megegyezik, akkor ez a megegyező elem benne van az X^i és Y^j leghosszabb közös részsorozatában. A leghosszabb közös részsorozat hosszáról pedig az mondható, hogy az X^i és az Y^j leghosszabb közös részsorozatának hossza eggyel nagyobb, mint az X^{i-1} és az Y^{j-1} leghosszabb közös részsorozatának hossza. Tehát $x_i = y_j$ esetén az $F[i, j] = F[i-1, j-1]+1$. Vizsgálandó eset még, amikor az $x_i \neq y_j$. Ilyenkor az X^i és az Y^j leghosszabb közös részsorozatának hossza az X^i és az Y^{j-1}, vagy az X^{i-1} és az Y^j leghosszabb közös részsorozat hossza közül a nagyobbik lesz. Az $F[i, j]$-re vonatkozó megállapításainkat az alábbi rekurzív képletben foglalhatjuk össze:

$$F[i,j] = \begin{cases} 0, & \text{ha } i = 0 \text{ vagy } j = 0 \\ F[i-1, j-1] + 1, & \text{ha } i, j > 0 \text{ és } x_i = y_j \\ \max\{F[i, j-1], F[i-1, j]\}, & \text{ha } i, j > 0 \text{ és } x_i \neq y_j \end{cases} \tag{7.4}$$

Természetesen az $F[i, j]$ értékeket most sem rekurzív függvény használatával állítjuk elő, hanem egy táblát használunk az adatok tárolására, amivel gyorsabb futás válik lehetségessé. A tábla előállítását a 7.3. algoritmus valósítja meg.

7.3. Algoritmus Leghosszabb közös részsorozat hossza

Bemenet: X – **T tömb**, n – **egész** (*tömb mérete*), Y – **T tömb**, m – **egész** (*tömb mérete*)
Kimenet: F – **egész tábla**

```
1: függvény LKRHossza(X : T tömb, n : egész, Y : T tömb, m : egész)
2:     F ← TáblaLétrehoz(egész)[n + 1, m + 1]
3:     ciklus j ← 0-től m-ig
4:         F[0, j] ← 0
5:     ciklus vége
6:     ciklus i ← 1-től n-ig
7:         F[i, 0] ← 0
8:     ciklus vége
9:     ciklus i ← 1-től n-ig
10:        ciklus j ← 1-től m-ig
11:            ha x_i = y_j akkor
12:                F[i, j] ← F[i − 1, j − 1] + 1
13:            különben
14:                F[i, j] ← max{F[i − 1, j], F[i, j − 1]}
15:            elágazás vége
16:        ciklus vége
17:    ciklus vége
18:    vissza F
19: függvény vége
```

Felhasznált változók és függvények

- X: Az egyik sorozat elemeit tartalmazó tömb.
- n: Az X sorozat elemeinek száma.
- Y: A másik sorozat elemeit tartalmazó tömb.
- m: Az Y sorozat elemeinek száma.
- F: Tábla, melynek mérete $(n + 1) \times (m + 1)$. $F[i, j]$ megadja, hogy az X^i és az Y^j sorozatoknak milyen hosszú a leghosszabb közös részsorozata. $F[n, m]$ határozza meg az X és Y sorozatok leghosszabb közös részsorozatának hosszát.
- TáblaLétrehoz(**egész**)[n+1, m+1]: Utasítás, mely létrehoz egy $(n + 1) \times (m + 1)$ méretű **egész** típusú táblát.

Az algoritmus bemenete az X és az Y sorozat, melyek elemei tömbben vannak eltárolva. A két bemeneti sorozatnak ismerjük az elemszámát is, melyek n és m. Kimenetként az F táblát adja vissza az algoritmus, melynek mérete $(n+1) \times (m+1)$. A táblát most is 0-tól indexeljük.

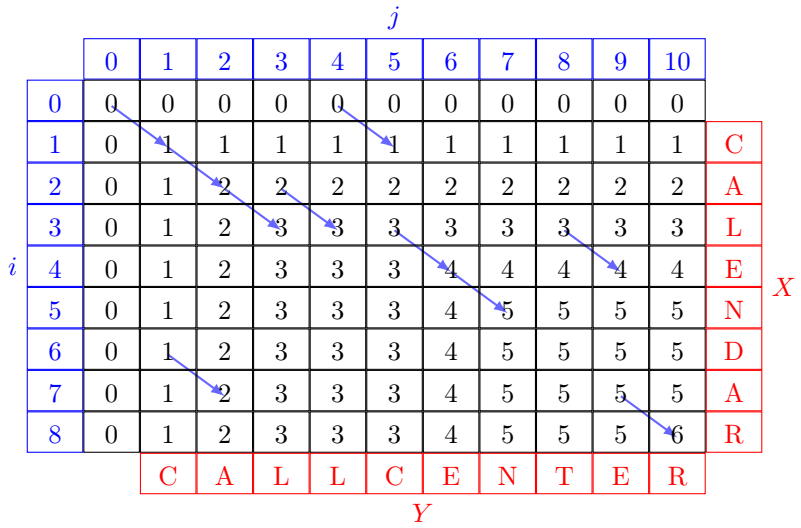
Az algoritmus nem tesz mást, mint sorfolytonos bejárással előállítja a 7.4. képlet alapján az $F[i,j]$ értékeket. A 3. sorban kezdődő ciklussal feltöltjük az F nulladik sorának minden elemét 0 értékkel. Hasonló módon az F nulladik oszlopának elemei a 6. sorban kezdődő ciklusban kapnak 0 értéket. A tényleges sorfolytonos bejárást a 9. sorban kezdődő két egymásba ágyazott ciklus valósítja meg. A külső ciklus végighalad az összes soron az elsőtől az n-edikig, míg a belső ciklus közben az elsőtől az m-edik oszlopig az aktuális i-edik sor minden elemét bejárja. A ciklusok magjában megvizsgáljuk, hogy az X sorozat i-edik eleme megegyezik-e az Y sorozat j-edik elemével (ld. 11. sor). Amennyiben megegyezik a két vizsgált elem, akkor a 7.4. képletnek megfelelően az $F[i,j]$ érték az $F[i-1,j-1]$ értéknél eggyel nagyobb lesz. Amennyiben nem teljesül az egyezőség (ld. 13. sor), akkor viszont az $F[i-1,j]$ és az $F[i,j-1]$ értékek közül a nagyobbikat kell eltárolni $F[i,j]$-ben. Az algoritmus végén vissza kell adni a feltöltött F táblát (ld. 18. sor).

Futási idő elemzése. Mivel a 7.3. algoritmusban két egymásba ágyazott ciklussal valósítottuk meg az F tábla feltöltését, ezért az algoritmus futási ideje minden esetben $O(n \cdot m)$-es, azaz a két vizsgált sorozat hosszának szorzatával arányos. ♣

7.3. Példa. Feladatunk, hogy megadjuk a leghosszabb közös részsorozatát a következő két sorozatnak: $X = \langle \mathrm{C}, \mathrm{A}, \mathrm{L}, \mathrm{E}, \mathrm{N}, \mathrm{D}, \mathrm{A}, \mathrm{R} \rangle$ és $Y = \langle \mathrm{C}, \mathrm{A}, \mathrm{L}, \mathrm{L}, \mathrm{C}, \mathrm{E}, \mathrm{N}, \mathrm{T}, \mathrm{E}, \mathrm{R} \rangle$. Ehhez első lépésként elő kell állítani a 7.4. képlet által meghatározott F táblát a 7.3. algoritmus használatával. A feladat további részének megoldását, egy konkrét leghosszabb közös részsorozat meghatározását a 7.4. példában mutatjuk be.

Először helyet kell foglalni a memóriában az F tábla számára. Az F mérete 9×11-es lesz, mivel az X sorozat 8, az Y sorozat pedig 10 elemű. Ezt követően a tábla nulladik sorát és nulladik oszlopát feltöltjük csupa 0 értékekkel. Ezután jön a tábla „értékes” elemeinek meghatározása. Mivel az X sorozat első eleme megegyezik az X sorozat első elemével, ezért $F[1,1]$ az $F[0,0]$ értéknél 1-gyel nagyobb lesz. Az X sorozat első eleme nem egyezik meg az Y sorozat második elemével, ezért $F[1,2]$ értéke az $F[0,2]$ és az $F[1,1]$ közül a nagyobbikkal lesz egyenlő. Hasonlóan járunk el minden más elemnél is a 7.4. képlet szerint, ahogy az a 7.3. algoritmusban is látható.

Az előálló F tábla a 7.4. ábrán látható. Az ábrán a feketével jelzett $F[i,j]$ értékek mellett, kék színnel feltüntettük a tábla sorainak és oszlopainak indexeit, valamint piros színnel az X és Y sorozat megfelelő elemeit. Világoskék nyilak mutatják azokat az eseteket, amikor az X sorozat i-edik eleme megegyezett az Y sorozat j-edik elemével, ezért az $F[i,j]$ elem értéke az $F[i-1,j-1]$ értékénél 1-gyel nagyobb lett. ¶



7.4. ábra. Az $X = \langle \mathrm{C}, \mathrm{A}, \mathrm{L}, \mathrm{E}, \mathrm{N}, \mathrm{D}, \mathrm{A}, \mathrm{R} \rangle$ és $Y = \langle \mathrm{C}, \mathrm{A}, \mathrm{L}, \mathrm{L}, \mathrm{C}, \mathrm{E}, \mathrm{N}, \mathrm{T}, \mathrm{E}, \mathrm{R} \rangle$ sorozatok leghosszabb közös részsorozatának meghatározásához szükséges F tábla előállítása.

Most már tudjuk, hogy miként lehet hatékonyan meghatározni az X^i és Y^j részsorozatok leghosszabb közös részsorozatainak hosszait, amit az F tábla $F[i,j]$ elemében tárolunk el. Megállapítottuk azt is, hogy az $F[n,m]$ érték megadja az X és Y sorozatok leghosszabb közös részsorozatának hosszát. Miként lehet viszont egy konkrét leghosszabb közös részsorozatot meghatározni?

A kérdés megválaszolásában a 7.3. algoritmus elemzése nyújt segítséget. Igaz, hogy a 7.3. algoritmus kimenetként csak az aktuális hosszértékeket tartalmazó F táblát állítja elő, de azért ha tovább gondolkodunk ennél több információ is található benne. A tábla előállítása során akkor tudunk csak értéknövekedést elérni, amikor az X sorozat i-edik eleme megegyezik az Y sorozat j-edik elemével. Ilyenkor pedig kijelenthető, hogy az X^{i-1} és az Y^{j-1} sorozatok leghosszabb közös részsorozatához, ha hozzátesszük az x_i (vagy az azzal megegyező y_j) elemet, akkor az X^i és Y^j egy leghosszabb közös részsorozatát kapjuk meg. Ezen logika alapján azt kell vizsgálnunk, hogy melyek azok a helyek az F táblában, ahol igaz, hogy értéknövekedés történt. Minden ilyen (i,j) helyen igaz, hogy

$$F[i,j] = F[i-1,j-1] + 1. \tag{7.5}$$

A kérdés csak az, hogy hogyan lehet az előbbi felismerés alapján megtalálni az X és Y sorozatok egy leghosszabb közös részsorozatát. Tekintsünk ehhez először egy konkrét esetet, a 7.3. példában előállított F táblát és a hozzá tartozó $X = \langle \mathrm{C,A,L,E,N,D,A,R} \rangle$ és az $Y = \langle \mathrm{C,A,L,L,C,E,N,T,E,R} \rangle$ sorozatokat. Látható, hogy az $F[n,m]$ elem 1-gyel nagyobb az $F[n-1,m-1]$ elemnél és $x_8 = y_{10}$. Így kijelenthető, hogy az X és Y közös utolsó eleme biztos benne lesz az X és Y leghosszabb közös részsorozatában. Ez azért igaz, mert az R betű nem lehet benne az X^{n-1} és az Y^{m-1} leghosszabb közös részsorozatában (ami az F alapján 5 hosszú), így ehhez hozzávéve az R betűt, máris egy 6 hosszú részsorozatát kapjuk X-nek és Y-nak. 6 hosszú részsorozatot egyébként az F tábla alapján most nem is kaphatunk másként csak a lezáró R betűvel.

Hogyan tudjuk meghatározni, hogy mi lesz az X és Y leghosszabb közös részsorozatának ötödik eleme. Ehhez kell a táblában találni egy olyan (i,j) helyet, ahol $F[i,j] = 5$, de $F[i-1,j-1] = 4$. Ha megtaláltuk ezt a helyet, akkor az x_i elem (vagy a vele megegyező y_j) lesz az X és Y leghosszabb közös részsorozatának ötödik eleme. Ilyen hely több is van a konkrét példában, de csak egy helyen teljesül, hogy $x_i = y_j$. Ez alapján kijelenthető, hogy az ötödik sor hetedik eleménél találunk ilyen esetet. Ebből viszont az is következik, hogy X és Y leghosszabb közös részsorozatának ötödik eleme az N betű lesz. Kérdés viszont, hogy az előbb megtalált $(8,10)$ koordinátájú helytől miként juthatunk el egy algoritmussal a mostani $(5,7)$ koordinátájú helyhez. Úgy is feltehetjük a kérdést, hogy az F táblát milyen módon kell ehhez bejárnunk.

A keresett bejárás viszont nem lesz más, mint hogy az F tábla jobb alsó sarkából (az (n,m) koordinátájú helytől) indulva bejárjuk az F táblát úgy, hogy közben visszakövetkeztetünk az F tábla előállítási módjára. Elindulunk az (n,m) koordinátájú helyről. Megvizsgáljuk, hogy az adott helyen $x_n = y_m$ teljesül-e. Ha teljesül, akkor biztos, hogy az $(n-1,m-1)$ koordinátájú helyre kell továbblépnünk, mert egyezőség esetén ezt diktálja az F előállítási szabálya. Ha viszont $x_n \neq y_m$, akkor megnézzük, hogy az $F[n-1,m]$ és $F[n,m-1]$ értékek közül melyik nagyobb. Ha az $F[n-1,m] > F[n,m-1]$, akkor az $F[n,m]$ értéke biztos $F[n-1,m]$-mel egyenlő, ezért az $(n-1,m)$ koordinátájú pontra lépünk vissza. Egyéb esetben viszont az $(n,m-1)$ helyre lépünk. Ezt a bejárási szabály követjük mindaddig, amíg meg nem találjuk a leghosszabb közös részsorozat minden elemét.

A fenti ötletet megvalósító algoritmus pontos leírását a 7.4. algoritmusban mutatjuk be. Az algoritmus bemenete a 7.3. algoritmus által előállított F tábla, valamint az X és Y sorozatok, melyek méreteit is ismerjük. Kimenetként az X és Y egy leghosszabb közös részsorozatát tartalmazó S tömböt adjuk vissza.

Az algoritmus első lépéseként létrehozzuk a kimeneti S tömböt, melynek a mérete pont az $F[n,m]$ értékkel egyezik meg (ld. 2. sor). Az F tábla bejárásához két indexváltozó szükséges. A sorokat az i, az oszlopokat pedig a j indexeli. Az i változó kezdeti értéke n (ld. 3. sor), az j változóé pedig m (ld. 4. sor), hiszen a bejárást az F jobb alsó sarkából kezdjük. Az S tömböt is szükséges indexelnünk. Mivel az S-t hátulról előrefelé haladva töltjük fel, ezért az *idx* index kezdeti értéke az S tömb mérete lesz (ld. 5. sor).

Az F tábla bejárását addig kell végeznünk, amíg az S tömböt teljesen fel nem töltöttük. Ezt a 6. sorban kezdődő ciklusban valósítjuk meg, amelyben addig kell bennmaradnunk, amíg az *idx* érték pozitív. A cikluson belül megvizsgáljuk, hogy az aktuális (i,j) hely esetén $X[i]$ és $Y[j]$ megegyezik-e (ld. 7. sor). Ha megegyeznek, akkor az S tömbbe felvesszük az $X[i]$ elemet, majd mindhárom indexet 1-gyel csökkentjük. Ha $X[i] \neq Y[j]$, akkor csak azt kell eldönteni, hogy az F táblában felfelé vagy balra lépjünk tovább. Amennyiben $F[i-1,j] > F[i,j-1]$ (ld. 12. sor), akkor az előző sorra lépünk, különben pedig

7.4. Algoritmus Leghosszabb közös részsorozat előállítása

Bemenet: F – **egész tábla**, X – **T tömb**, n – **egész**, Y – **T tömb**, m – **egész**
Kimenet: S – **T tömb**

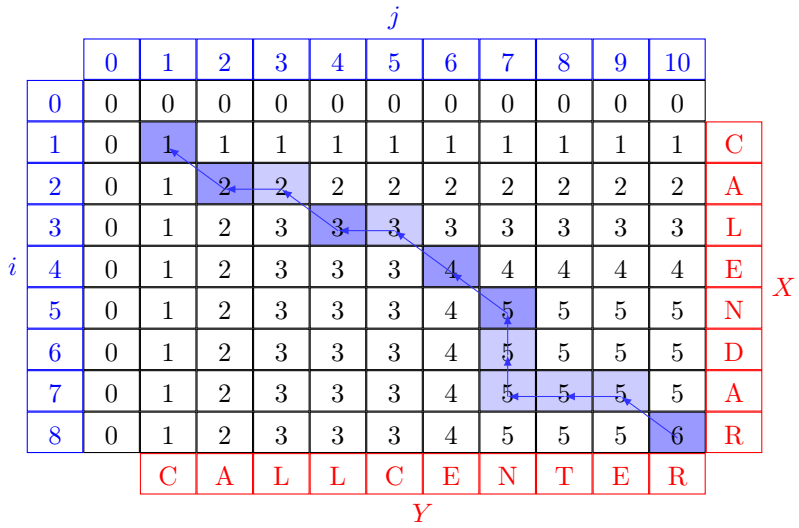
```
1: függvény LKRElőállítás(F : egész tábla, X : T tömb, n : egész, Y : T tömb, m : egész)
2:     S ← Létrehoz(T)[F[n, m]]
3:     i ← n
4:     j ← m
5:     idx ← F[n, m]
6:     ciklus amíg idx > 0
7:         ha X[i] = Y[j] akkor
8:             S[idx] ← X[i]
9:             idx ← idx − 1
10:            i ← i − 1
11:            j ← j − 1
12:        különben ha F[i − 1, j] > F[i, j − 1] akkor
13:            i ← i − 1
14:        különben
15:            j ← j − 1
16:        elágazás vége
17:    ciklus vége
18:    vissza S
19: függvény vége
```

Felhasznált változók és függvények

- X: Az egyik sorozat elemeit tartalmazó tömb.
- n: Az X sorozat elemeinek száma.
- Y: A másik sorozat elemeit tartalmazó tömb.
- m: Az Y sorozat elemeinek száma.
- F: Tábla, melynek mérete $(n+1) \times (m+1)$. $F[i,j]$ megadja, hogy az X^i és az Y^j sorozatoknak milyen hosszú a leghosszabb közös részsorozata. $F[n,m]$ határozza meg az X és Y sorozatok leghosszabb közös részsorozatának hosszát. Az F táblát a 7.3. algoritmusban bemutatott LKRHossza függvénnyel állítjuk elő.
- S: Az X és az Y sorozatok egyik leghosszabb közös részsorozata. Az S elemei egy $F[n,m]$ elemszámú tömbben vannak eltárolva.
- Létrehoz(**T**)$[F[n,m]]$: Utasítás, mely létrehoz egy $F[n,m]$ elemű **T** típusú tömböt.

(ld. 14. sor) az előző oszlopba lépünk. A ciklusból kilépve már csak vissza kell adni a kimeneti S tömböt (ld. 18. sor), mely tartalmazza az X és Y egy leghosszabb közös részsorozatát.

7.4. Példa. A 7.5. ábrán szemléltetjük, hogy a 7.4. algoritmus használatával milyen bejárással kapjuk meg az $X = \langle \mathrm{C}, \mathrm{A}, \mathrm{L}, \mathrm{E}, \mathrm{N}, \mathrm{D}, \mathrm{A}, \mathrm{R} \rangle$ és az $Y = \langle \mathrm{C}, \mathrm{A}, \mathrm{L}, \mathrm{L}, \mathrm{C}, \mathrm{E}, \mathrm{N}, \mathrm{T}, \mathrm{E}, \mathrm{R} \rangle$ sorozatok egy leghosszabb közös részsorozatát a 7.3. példában előállított F tábla használatával. A nyilak mutatják a bejárás menetét, a sötétebb hátterű elemek pedig azok, amelyek bekerülnek a leghosszabb közös részsorozatba. Így az előállított leghosszabb közös részsorozat az $S = \langle \mathrm{C}, \mathrm{A}, \mathrm{L}, \mathrm{E}, \mathrm{N}, \mathrm{R} \rangle$ sorozat lesz. ¶



7.5. ábra. Az $X = \langle \mathrm{C}, \mathrm{A}, \mathrm{L}, \mathrm{E}, \mathrm{N}, \mathrm{D}, \mathrm{A}, \mathrm{R} \rangle$ és $Y = \langle \mathrm{C}, \mathrm{A}, \mathrm{L}, \mathrm{L}, \mathrm{C}, \mathrm{E}, \mathrm{N}, \mathrm{T}, \mathrm{E}, \mathrm{R} \rangle$ sorozatok egy leghosszabb közös részsorozatának előállítása a 7.3. példában meghatározott F tábla felhasználásával. A leghosszabb közös részsorozat: $S = \langle \mathrm{C}, \mathrm{A}, \mathrm{L}, \mathrm{E}, \mathrm{N}, \mathrm{R} \rangle$.

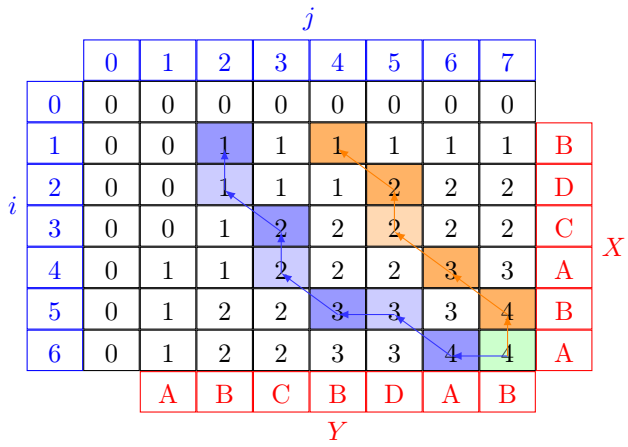
Megjegyzés

Sok esetben nem csak egy leghosszabb közös részsorozata van két sorozatnak, hanem több is. Természetesen a hossza mindegyik leghosszabb közös részsorozatnak ugyanaz.

A 7.6. ábrán mutatunk egy példát, ahol két legnagyobb közös részsorozatát is meghatározzuk az $X = \langle \mathrm{B}, \mathrm{D}, \mathrm{C}, \mathrm{A}, \mathrm{B}, \mathrm{A} \rangle$ és az $Y = \langle \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{B}, \mathrm{D}, \mathrm{A}, \mathrm{B} \rangle$ sorozatnak. A kék színnel jelölt esetben a 7.4. algoritmussal határoztuk meg az $S_1 = \langle \mathrm{B}, \mathrm{C}, \mathrm{B}, \mathrm{A} \rangle$ esetet. A narancssárga esetben viszont kis mértékben módosítottunk a 7.4. algoritmuson. Az algoritmus 12. sorában a feltétel vizsgálatnál megengedjük az egyenlőséget is. Így már az $S_2 = \langle \mathrm{B}, \mathrm{D}, \mathrm{A}, \mathrm{B} \rangle$ sorozatot kapjuk eredményül.

Futási idő elemzése. A 7.4. algoritmus futási ideje könnyen meghatározható, ha megvizsgáljuk, hogy az algoritmusbeli ciklus legfeljebb hányszor fut le. Mivel az i érték legfeljebb $(n-1)$-szer, az j pedig legfeljebb $(m-1)$-szer csökkenthető, így az algoritmus futási ideje $O(n+m)$-es.

A teljes leghosszabb közös részsorozat meghatározás futási ideje a 7.3. és a 7.4. algoritmusok futási idejének összege, tehát $O(n \cdot m)$-es. ♣



7.6. ábra. Az $X = \langle \mathrm{B}, \mathrm{D}, \mathrm{C}, \mathrm{A}, \mathrm{B}, \mathrm{A} \rangle$ és az $Y = \langle \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{B}, \mathrm{D}, \mathrm{A}, \mathrm{B} \rangle$ sorozatok legnagyobb közös részsorozatának meghatározása. Két leghosszabb közös részsorozatot is találhatunk: $S_1 = \langle \mathrm{B}, \mathrm{C}, \mathrm{B}, \mathrm{A} \rangle$ és $S_2 = \langle \mathrm{B}, \mathrm{D}, \mathrm{A}, \mathrm{B} \rangle$.

Mohó algoritmusok

A második optimalizálási módszer, amit megismerünk, a mohó megközelítés. A módszer lényege nagyon egyszerű: minden döntésünket mohó módon hozzuk meg. Ez azt jelenti, hogy bármikor döntési helyzetbe kerül az algoritmus, az éppen legjobbnak tűnő megoldást választja. Ezeket a döntéseket később sem vizsgálja felül. Mivel nem végzünk minden esetre kiterjedő elemzéseket, ezért a módszer nem biztos, hogy az optimális megoldást találja meg. Viszont a futási idő tekintetében gyors algoritmusok várhatók.

A fejezet elején egy konkrét problémán keresztül ismerjük meg a mohó stratégiát. Ez a példa a pénztári kifizetés eljárása (ld. 7.2.1. alfejezet). Ezt követően a dinamikus programozásnál már megismert 0-1 hátizsák problémának adjuk meg a mohó megoldását (ld. 7.2.2. alfejezet). A dinamikus programozás és a mohó algoritmus összehasonlítása érdekében a 7.2.3. alfejezetben megadjuk a fejezet elején ismertetett kincs gyűjtő feladat megoldását mindkét megközelítés használatával. A 7.2.4. alfejezetben összefoglaljuk a mohó algoritmusok legfontosabb jellemzőit. A fejezet végén a 7.2.5. alfejezetben ütemezési feladatokat és azok megoldásait ismertetjük, melyek esetén a mohó stratégia optimális megoldást eredményez.

Pénzkifizetés

Első mohó módszerként ismerkedjünk meg a pénztári kifizetés problémájával. Egy pénztárban gyakran előforduló feladat, hogy egy konkrét összeget kell kifizetni valaki számára, amihez a rendelkezésre álló címletek közül a lehető legkevesebbet szabad használni. Adjunk konkrét algoritmust ennek a problémának a megoldására!

Mohó megközelítést használva nagyon egyszerűen kell eljárnunk. Az eljárás alapötlete, hogy minél nagyobb címletet használunk a kifizetéshez annál kevesebb címletre van szükségünk. Vegyük például azt az esetet, amikor 20.000 Ft-ot kell kifizetni. Ezt megoldhatjuk egyetlen 20.000 Ft-os címlettel, vagy 2 db 10.000 Ft-ossal is. Ha a nagyobb címletet használjuk, akkor értelemszerűen kevesebb címletre van szükségünk. Kövessük tehát azt az ötletet, hogy mindig a lehető legnagyobb címletet adja oda a pénztáros, majd a még kifizetendő összeget is a legnagyobb címlet felhasználásával fizeti ki. Mindezt addig teszi, amíg mindent ki nem fizetett. Ez a módszer azért mohó, mert a legnagyobb címletet használja mindig.

A módszer konkrét leírását a 7.5. algoritmusban adjuk meg. Az algoritmus egyik bemenete a kifizetendő összeg, melyet x-szel jelöljük. (Azt feltesszük, hogy ez az összeg maradék nélkül kifizethető a rendelkezésre álló címletek felhasználásával.) Másik bemenetünk egy rendezett tömb, melyben a rendelkezésre álló címletek értékeit adjuk meg. (A jelenleg Magyarországon forgalomban lévő címletek esetén $c = \{5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000\}$.) Természetesen a c tömb n méretét is ismerjük. Az algoritmus kimenete az n elemű db tömb. A db tömb i-edik eleme meghatározza, hogy a $c[i]$ címletből hány darabra van szükség az x összeg kifizetésekor. Tehát

$$x = \sum_{i=1}^{n} db[i] \cdot c[i]. \tag{7.6}$$

Az algoritmus olyan db tömböt határoz meg, melyben tárolt értékek összege – azaz a címletek száma – minimális miközben a 7.6. egyenlet is teljesül. A db elemeinek összege ugyanis pont a felhasznált címletek számával egyezik meg.

7.5. Algoritmus Pénzkifizetés mohó algoritmusa

Bemenet: x **– egész**, c **– egész rendezett tömb**, n **– egész**
Kimenet: db **– egész tömb**

```
1:  függvény PÉNZKIFIZETÉS(x : egész, c : egész tömb, n : egész)
2:      db ← LÉTREHOZ(egész)[n]
3:      ciklus i ← 1-től n-ig
4:          db[i] ← 0
5:      ciklus vége
6:      j ← n
7:      ciklus amíg x > 0
8:          ciklus amíg c[j] > x
9:              j ← j − 1
10:         ciklus vége
11:         db[j] ← db[j] + 1
12:         x ← x − c[j]
13:     ciklus vége
14:     vissza db
15: függvény vége
```

Felhasznált változók és függvények

- x: A kifizetendő összeg.
- c: A rendelkezésre álló címletek növekvő módon rendezett tömbje.
- n: A c tömb mérete.
- db: Kimeneti n elemű tömb. A tömb i-edik elemének értéke megadja, hogy a $c[i]$ címletből hány darabot kell kifizetni, tehát $x = \sum_{i=1}^{n} db[i] \cdot c[i]$.
- LÉTREHOZ(**egész**)$[n]$: Utasítás, mely létrehoz egy n elemű **egész** típusú tömböt.

A 7.5. algoritmus elején létrehozunk egy n elemű tömböt a felhasznált címletek darabszámának tárolására (ld. 2. sor). Ennek a tömbnek minden elemét 0-val tesszük egyenlővé, amit a 3. sorban kezdődő ciklussal valósítunk meg. Mivel számon kell tartanunk, hogy éppen melyik címletet használjuk a kifizetéshez, ezért ezt egy j indexváltozóval fogjuk jelölni. A j kezdeti értéke a legnagyobb címletnek megfelelő index, tehát a c tömb mérete lesz (ld. 6. sor). Mivel a teljes x összeget ki kell fizetni, ezért a 7. sorban kezdődő ciklusban addig keresünk megfelelő címleteket, amíg a még kifizetendő x összeg pozitív. A cikluson belül első lépésként meg kell keresni azt a címletet, ami a felhasználhatóak közül a legnagyobb. Ennek megtalálását egy ciklussal valósítjuk meg (ld. 8. sor), amelyben a j indexet addig csökkentjük, amíg az aktuális címlet nagyobb a még kifizetendő összegnél. Ha találtunk olyan címletet, amit fel tudunk használni, akkor az adott címletnek megfelelő darabszámot a *db* tömbben eggyel növeljük (ld. 11. sor). Mivel a megtalált címletet már fel is használjuk pénzkifizetésre, ezért a még kifizetendő összeget csökkentjük a címlet értékével (ld. 12. sor). Az algoritmus végén a *db* tömböt visszaadjuk eredményként (ld. 14. sor).

7.5. Példa. Fizessünk ki 79.485 Ft-ot a lehető legkevesebb címlet felhasználásával!

A 7.5. algoritmus használatával azt kapjuk, hogy 3 db 20.000-es, 1 db 10.000-es, 1 db 5.000-es, 2 db 2.000-es, 2 db 200-as, 1 db 50-es, 1 db 20-as, 1 db 10-es és 1 db 5-ös címlettel lehet a kifizetést megvalósítani. Nem bizonyítjuk, de igaz, hogy ennél kevesebb címlettel nem lehet megoldani a feladatot. ¶

7.6. Példa. Feladatunk most adott összértékű bélyeg felragasztása egy borítékra úgy, hogy a lehető legkevesebb bélyeget használjuk. A postán fellelhető bélyegek címleteit a c tömbben tároljuk. $c = \{10, 100, 210, 340, 700, 1.000, 3.500\}$. Hogyan tudjuk megoldani a feladatot, ha 1.400 Ft-nyi bélyeget kell a borítékra tenni?

A 7.5. algoritmus alapján azt kapjuk, hogy a feladat 1 db 1.000-es, 1 db 340-es és 6 db 10-es bélyeggel oldható meg. Könnyen látható, hogy ez a megoldás nem optimális, hiszen 2 db 700 Ft-os bélyeggel is megoldható a feladat. ¶

A fenti két példából látható, hogy ugyanaz az algoritmus egyik esetben optimális megoldást szolgáltat, míg a másik esetben nem. Ennek fő oka az, hogy a c címleteket tartalmazó tömb más az egyik, illetve a másik esetben.

A mohó algoritmusokról egyelőre azt látjuk, hogy a döntés mohó módon történik – esetünkben a lehető legnagyobb választható címlet mellett döntünk –, viszont nem minden esetben kapunk optimális megoldást. Az is igaz persze, hogy bár a 7.6. példánál kapott megoldás nem az optimális, de nem is a legrosszabb. Ezt úgy fejezhetjük ki, hogy a mohó algoritmus nem mindig eredményez globális optimumot, de a probléma egy lokális optimumát megadja.

0-1 hátizsák probléma

A 7.1.1. fejezetben már megismerkedtünk a 0-1 hátizsák feladattal, és láttuk miként lehetséges a megoldása dinamikus programozás használatával. Most ugyanezt a feladatot mohó algoritmussal oldjuk meg.

A mohó megközelítéshez először meg kell fogalmaznunk, hogy milyen szempontból kívánunk mohók lenni. Mivel célunk a hátizsákba kerülő összérték maximalizálása a súlykorlát figyelembe vétele mellett, ezért több lehetőség is felmerülhet. Az első, hogy a tárgyak értékét vesszük csak figyelembe a kiválasztásnál. Ilyenkor kiválasztjuk a legnagyobb értékű tárgyat és behelyezzük a zsákba, ha lehetséges. Ezután kiválasztjuk a még nem vizsgált tárgyak közül a legnagyobb értékűt és ezt is behelyezzük a zsákba, amíg lehetséges. Ezt az eljárást követjük addig, amíg meg nem telik a zsák, vagy el nem fogynak a tárgyak. Megvalósítás szempontjából szükséges, hogy a tárgyaink az értékük szerint csökkenő sorrendben legyenek és máris könnyen megalkotható az algoritmus. Mielőtt leírnánk a konkrét algoritmust ejtsünk szót a mohóság további lehetőségeiről. Dönthetünk úgy is, hogy nem a tárgyak értékét tekintjük a legfontosabb jellemzőnek, hanem a súlyukat. Ekkor először a legkisebb súlyú tárgyat választjuk ki, majd az egyre nagyobb súlyúakat. Igaz, hogy ilyenkor nem figyeljük a feladat szempontjából fontos értéket, viszont a lehető legtöbb tárgyat tudjuk a zsákba tenni, melyek összértéke is nagyobb lehet, mint például egy nagy értékű, de nagyon nagy súlyú tárgynak. Harmadik lehetséges megközelítés, ha az érték és súly arányt vesszük figyelembe, tehát elsőként a legnagyobb érték-súly arányú tárgyat választjuk ki. Ezen utolsó megközelítést fogjuk használni, tehát ha p a tárgyak értékeinek n elemű tömbje, w pedig a megfelelő súlyok tömbje, akkor teljesül az alábbi relációsorozat:

$$\frac{p[1]}{w[1]} \geq \frac{p[2]}{w[2]} \geq \ldots \geq \frac{p[n]}{w[n]}. \tag{7.7}$$

A leírt ötlet konkrét megvalósítását a 7.6. algoritmusban adjuk meg. Az algoritmus bemenetei a kincsek értékeit tartalmazó p tömb, valamint a megfelelő kincsek súlyait tartalmazó w tömb. A két tömb elemeire teljesül a 7.7. egyenlőtlenségsorozat. Ismerjük a két tömb azonos n elemszámát is, valamint bemenetként adjuk még meg a hátizsák c kapacitását. Az algoritmus kimenete a kiválasztott kincsek indexeit tartalmazó S halmaz.

7.6. Algoritmus 0-1 hátizsák probléma mohó megoldása

Bemenet: p – **egész tömb**, w – **egész tömb**, n – **egész** (*tömb mérete*), c – **egész**
Kimenet: S – **egész halmaz**

```
1:  függvény MohóHátizsák(p : egész tömb, w : egész tömb, n : egész, c : egész)
2:      S ← ∅
3:      i ← 1
4:      ciklus amíg (c > 0) ∧ (i ≤ n)
5:          ha w[i] ≤ c akkor
6:              S ← S ∪ {i}
7:              c ← c − w[i]
8:          elágazás vége
9:          i ← i + 1
10:     ciklus vége
11:     vissza S
12: függvény vége
```

Felhasznált változók és függvények

- p: Tömb, melyben az egyes kincsek értékét tároljuk.
- w: Tömb, melyben az egyes kincsek értékét tároljuk. (Fontos kiemelnünk, hogy a p és w tömb elemeire teljesül a 7.7. egyenlőtlenségsorozat.)
- n A p és w tömbök mérete.
- c: A hátizsák kapacitása.
- S: Halmaz, melynek elemei a hátizsákba helyezendő tárgyak indexei.

A 7.6. algoritmus 2. sorában inicializáljuk a kimeneti S halmazt, mely kezdetben üres, hiszen még egyetlen kincset sem választottunk ki. Ezután ki kell választanunk mohó módon a kincseket. Mivel a kincsek érték-súly arány szerint csökkenő módon rendezettek, ezért csak annyit kell tennünk, hogy

végighaladunk a kincseken egy bejárással. A bejárás során az i változóval indexelünk, melynek kezdeti értéke 1 (ld. 3. sor). A bejárást a 4. sorban kezdődő ciklussal valósítjuk meg. A ciklusban addig kell bennmaradnunk, amíg van még szabad hely a hátizsákban ($c > 0$), illetve amíg van még megvizsgálandó kincsünk ($i \leq n$). A cikluson belül megvizsgáljuk, hogy az aktuális kincs befér-e a hátizsákba (ld. 5. sor). Ha befér, akkor a kincset betesszük a hátizsákba, azaz az indexét eltároljuk az S halmazban (ld. 6. sor), illetve a hátizsák szabad kapacitását csökkentjük a betett kincs súlyával (ld. 7. sor). A bejárás során minden esetben szükséges a következő tárgyra továbblépni, ezért az i értékét a 9. sorban 1-gyel növeljük. A bejárás végeztével az S halmazt adjuk vissza kimenetként (ld. 11. sor).

7.7. Példa. Vizsgáljuk meg, hogy a 7.1. táblázatban lévő kincsek közül melyeket helyezi egy $c = 5$ kapacitású hátizsákba a 7.6. mohó algoritmus. Vegyük észre, hogy a kincsek 7.1. táblázatban megadott indexelése pont megfelelő számunkra, mivel épp a $\frac{p}{w}$ arány szerinti csökkenő rendezettségben vannak.

A 7.6. algoritmus először megvizsgálja, hogy az első tárgy súlya kisebb-e a $c = 5$ kapacitásnál. Mivel kisebb, ezért az S halmazba bekerül az első tárgy indexe ($S = \{1\}$ lesz), a szabad kapacitás c értéke pedig 3-ra csökken. Ezután következik a második tárgy vizsgálata. Mivel az is befér a hátizsákba, ezért az $S = \{1, 2\}$-re változik, a hátizsák szabad kapacitása pedig 0-ra csökken. Mivel $c = 0$, ezért kilépünk a bejárást megvalósító ciklusból, és az algoritmus visszaadja az S halmazt.

Jól látható, hogy ebben az esetben a mohó algoritmus ugyanazt az eredményt szolgáltatja, mint a dinamikus programozás esetében, ahogy azt a 7.1. példában láttuk. ¶

7.8. Példa. Tekintsünk egy másik példát, melyben a 7.2. táblázatban megadott értékű és súlyú kincseket használjuk. A kincsek most is az érték-súly arány szerint csökkenő módon rendezettek. Milyen kiválogatást eredményez ebben az esetben a 7.6. mohó algoritmus, ha a hátizsákunk kapacitása $c = 50$?

i	p_i	w_i
1	60	10
2	100	20
3	120	30

7.2. táblázat. A 7.8. feladat kincseinek konkrét értékei és súlyai.

Az első tárgy befér a hátizsákba, ezért be is helyezzük ($S = \{1\}$), a hátizsák szabad kapacitását pedig $c = 40$-re csökkentjük. A második kincs is befér még a hátizsákba, ezért $S = \{1, 2\}$ lesz, a szabad kapacitás pedig 20-ra csökken. A harmadik tárgy viszont már nem fér be a hátizsákba, mivel a súlya meghaladja a hátizsák szabad kapacitását.

Könnyű belátni, hogy a 7.6. algoritmus ebben az esetben nem szolgáltatott optimális megoldást. A kiválasztott kincsek összértéke 160. Ha viszont az első és harmadik tárgyat választjuk, melyek befférnek a hátizsákba, az összérték akkor 180. A második és harmadik tárgy is befér a zsákba, az összérték ebben az esetben már 220.

Az algoritmus ott „hibázott”, hogy mohó módon az első tárgyat rögtön betette a zsákba, míg egy kicsit előrelátóbb szemlélettel mérsékletet gyakorolva ezt nem kellett volna megtennie. De a mohó algoritmusok nem tekintenek előre, csak a pillanatnyilag legjobbnak tűnő esetet nézik.

A 7.7. ábrán látható, hogy dinamikus programozást használva a 7.1. algoritmus milyen F táblát állít elő, ha az x értékek 10-esével változnak. Az ábra szemlélteti a 7.2. algoritmus által szolgáltatott optimális megoldást is. Látható tehát, hogy a dinamikus programozási megközelítés ebben az esetben is megtalálta az optimális megoldást. ¶

Futási idő elemzése. Mivel a 7.6. algoritmusban egyetlen ciklus van, mely legfeljebb n-szer fut le, ezért az algoritmus futási ideje $O(n)$-es. Így megállapítható, hogy a mohó megközelítés gyorsabb futási időt eredményez, mint a dinamikus programozási módszer használata, viszont nem minden esetben eredményez optimális megoldást. ♣

i \ x	0	10	20	30	40	50
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160	180	220

7.7. ábra. A 7.8. példa megoldása dinamikus programozással. Eredményül az optimális $S = \{2, 3\}$ kiválasztást kapjuk.

Mohó algoritmus szemben a dinamikus programozással

A két tárgyalt optimalizálási módszerünket már összehasonlítottuk a 0-1 hátizsák probléma kapcsán. Vizsgáljuk meg most, hogy a fejezet elején említett kincsbegyűjtési problémát melyik megközelítés miként oldja meg.

A megoldandó feladat a következő. Egy m sorból és n oszlopból álló mezőn minden egész koordinátájú rácspontban kincs van elhelyezve. A kincsek értékeit a C $m \times n$ méretű kétdimenziós tömbben tároljuk. A mező bal alsó sarkából, azaz az $(1,1)$ indexű helyről el akarunk jutni a jobb felső sarokba (az (m,n) koordinátájú helyre) úgy, hogy a bejárás közben csak jobbra és felfelé haladhatunk. Amely mezőn áthaladunk, ott kiássuk a kincset. Kérdés, hogy milyen útvonalon haladjunk annak érdekében, hogy a legtöbb kincset gyűjtsük össze.

Oldjuk meg először a feladatot a dinamikus programozás használatával. Ehhez létrehozunk egy F $m \times n$ méretű kétdimenziós tömböt. Az $F[i,j]$ értéke megmutatja, hogy ha az (i,j) koordinátájú helyig jutunk a bejárás során, akkor mennyi kincset tudunk maximálisan összegyűjteni. Tehát a teljes bejárás során összegyűjtött kincsek mennyiségét az $F[m,n]$ értéke adja meg. Kérdés, hogy miként határozhatók meg az $F[i,j]$ értékek.

Induljunk ki a legegyszerűbb esetből. Az $(1,1)$ koordinátájú helyre csak egyféleképpen juthatunk el, hiszen onnan indulunk. Ezért $F[1,1] = C[1,1]$. Miként érhetünk el a legalsó sor további elemeihez. Ennek is csak egyetlen módja van: az $(1,j)$ koordinátájú helyre csak az $(1,j-1)$ koordinátájú helyről léphetünk, ha $2 \leq j \leq n$. Ezért az összegyűjtött kincs mennyiségére ilyenkor igaz, hogy $F[1,j] = F[1,j-1] + C[i,j]$. Hasonlóan az első oszlop elemeiről kijelenthető, hogy $F[i,1] = F[i-1,1] + C[i,j]$, ha $2 \leq i \leq m$. Mi a helyzet akkor, ha nem az első sorban és nem is az első oszlopban vagyunk? Ilyenkor az (i,j) koordinátájú helyre érkezhetünk a bal oldali vagy az alsó szomszédjától. A két hely közül onnan érdemesebb érkezni, ahol nagyobb az addig összegyűjtött kincs mennyisége. Ez alapján kijelenthető, hogy $F[i,j] = \max\{F[i-1,j], F[i,j-1]\} + C[i,j]$, ha $2 \leq i \leq m$ és $2 \leq j \leq n$. Összefoglalva a megállapításainkat:

$$F[i,j] = \begin{cases} C[i,j], & \text{ha } i = 1 \text{ és } j = 1, \\ F[i,j-1] + C[i,j], & \text{ha } i = 1 \text{ és } j \geq 2, \\ F[i-1,j] + C[i,j], & \text{ha } i \geq 2 \text{ és } j = 1, \\ \max\{F[i-1,j], F[i,j-1]\} + C[i,j], & \text{ha } i \geq 2 \text{ és } j \geq 2. \end{cases} \tag{7.8}$$

A 7.7. algoritmusban adjuk meg az így definiált F kétdimenziós tömb előállításának algoritmusát. Az algoritmust részletesebben nem írjuk le, mert semmi más nem történik benne, mint hogy a 7.8. képlet alapján sorfolytonos bejárást követve előállítja az F tömb értékeit.

A 7.1. ábrán bemutatott példa esetén az F tömb a 7.8b. ábrán látható lesz.

1	5	3	6
11	2	15	1
6	10	20	9
1	3	4	8

(a) A C tömb.

19	39	55	61
18	20	52	53
7	17	37	46
1	4	8	16

(b) A F tömb.

19	39	55	61
18	20	52	53
7	17	37	46
1	4	8	16

(c) Az optimális bejárás.

7.8. ábra. Kincsek begyűjtése dinamikus programozással.

Hogyan lehet az F tömb ismeretében megtalálni a legtöbb kincset eredményező bejárás útvonalát. Ehhez az (m,n) koordinátájú pontból visszalépkedve kell megnézni, hogy egy adott mező bal vagy alsó szomszédja-e a nagyobb. A nagyobb értékű helyre kell visszalépni, majd tovább folytatni az ilyen jellegű visszalépkedést. A konkrét megvalósítást a 7.8. algoritmusban írjuk le. Az algoritmus kimenete egy P tömb lesz, melynek elemei a bejárás során érintett mezők koordinátáit tartalmazzák.

Vizsgált példánk esetén az optimális útvonal a $P = \{(1,1), (2,1), (2,2), (2,3), (3,3), (3,4), (4,4)\}$ lesz, ahogy az a 7.8c. ábrán is látható.

Megállapíthatjuk, hogy a dinamikus programozás használatával megkaptuk az optimális megoldást. Ehhez szükségünk volt egy átmeneti F kétdimenziós tömbre, tehát a memória igény a C tömb méretének duplája volt. Futási idő tekintetében az alkalmazott algoritmusok $O(m \cdot n)$-esek.

7.7. Algoritmus Összegyűjtött kincsek összege (dinamikus programozás)

Bemenet: C – **egész tömb**, m – **egész** (C *sorainak száma*), n – **egész** (C *oszlopainak száma*)
Kimenet: F – **egész tömb**

```
1:  függvény KincsÖsszeg(C : egész tömb, m : egész, n : egész)
2:      F ← Létrehoz(egész)[m, n]
3:      F[1, 1] = C[1, 1]
4:      ciklus j ← 2-től n-ig
5:          F[1, j] = F[1, j − 1] + C[i, j]
6:      ciklus vége
7:      ciklus i ← 2-től m-ig
8:          F[i, 1] = F[i − 1, 1] + C[i, j]
9:      ciklus vége
10:     ciklus i ← 2-től m-ig
11:         ciklus j ← 2-től n-ig
12:             F[i, j] = max (F[i − 1, j], F[i, j − 1]) + C[i, j]
13:         ciklus vége
14:     ciklus vége
15:     vissza F
16: függvény vége
```

Felhasznált változók és függvények

- C: Kincsek értékét tartalmazó kétdimenziós tömb.
- m: A C tömb sorainak száma.
- n: A C tömb oszlopainak száma.
- F: A C tömbbel azonos méretű kétdimenziós tömb. Az $F[i, j]$ értéke megadja, hogy mennyi a maximálisan összegyűjthető kincs mennyisége az (i, j) koordinátájú mezőig jutva a bejárásban.
- Létrehoz(**egész**)$[m, n]$: Utasítás, mely létrehoz egy $m \times n$ méretű kétdimenziós **egész** típusú tömböt.

7.8. Algoritmus Bejárási út kiolvasása (dinamikus programozás)

Bemenet: F – **egész tömb**, m – **egész** (F *sorainak száma*), n – **egész** (F *oszlopainak száma*)
Kimenet: P – **egész tömb**

```
1:  függvény BejárásiÚtKiolvas(F : egész tömb, m : egész, n : egész)
2:      P ← Létrehoz(egész)[m + n − 1]
3:      i ← m
4:      j ← n
5:      k ← m + n − 1
6:      ciklus amíg (i ≥ 2) ∧ (j ≥ 2)
7:          P[k] ← (i, j)
8:          k ← k − 1
9:          ha F[i − 1, j] > F[i, j − 1] akkor
10:             i ← i − 1
11:         különben
12:             j ← j − 1
13:         elágazás vége
14:     ciklus vége
15:     ciklus amíg i ≥ 2
16:         P[k] ← (i, j)
17:         k ← k − 1
18:         i ← i − 1
19:     ciklus vége
20:     ciklus amíg j ≥ 2
21:         P[k] ← (i, j)
22:         k ← k − 1
23:         j ← j − 1
24:     ciklus vége
25:     P[1] ← (1, 1)
26:     vissza P
27: függvény vége
```

Felhasznált változók és függvények

- F: A 7.7. algoritmus által meghatározott kétdimenziós tömb.
- m: Az F tömb sorainak száma.
- n: Az F tömb oszlopainak száma.
- P: Tömb, melynek elemszáma $m + n - 1$. A P tömb adja meg egy optimális bejárás koordináta sorozatát.
- Létrehoz(**egész**)$[m + n - 1]$: Utasítás, mely létrehoz egy $m + n - 1$ elemű **egész** típusú tömböt.

Vizsgáljuk meg, hogy miként tudjuk a feladatot mohó módon megoldani. Ha a mohó megközelítést követjük, akkor a bejárásnál egyszerűen elindulunk az $(1,1)$ koordinátájú helyről. Itt rövidlátó módon ránézünk a jobb oldali és a felső szomszédra. A kettő közül amelyik nagyobb, arra folytatjuk a bejárást. Minden mezőn ezt a stratégiát követjük, tehát csak a jobb és felső szomszédot vizsgálva hozunk döntést a továbbhaladásról. Mohó módon mindig az éppen jobbnak tűnő irányt választjuk. Amikor elérjük a felső sort vagy a jobb szélső oszlopot, akkor már csak egyenes úton el kell jutnunk a jobb felső célmezőbe.

Az ismertetett ötletnek megfelelő konkrét megvalósítást a 7.9. algoritmusban írjuk le. Az algoritmus a bal alsó sarokból eljut a legfelső sorig vagy a jobb szélső oszlopig. Ezt követően végigjárja a jobb szélső oszlopot, illetve a felső sort. Az utolsó lépésben a jobb felső sarkot is hozzáveszi a bejárást megadó P koordináta sorozathoz.

7.9. Algoritmus Kincsek begyűjtése (mohó algoritmus)

Bemenet: C – **egész tömb**, m – **egész** (C *sorainak száma*), n – **egész** (C *oszlopainak száma*)
Kimenet: P – **egész tömb**

```
1:  függvény MohóKincsGyűjtés(C : egész tömb, m : egész, n : egész)
2:      P ← Létrehoz(egész)[m + n − 1]
3:      i ← 1
4:      j ← 1
5:      k ← 0
6:      ciklus amíg (i < m) ∧ (j < n)
7:          k ← k + 1
8:          P[k] ← (i, j)
9:          ha C[i + 1, j] > C[i, j + 1] akkor
10:             i ← i + 1
11:         különben
12:             j ← j + 1
13:         elágazás vége
14:     ciklus vége
15:     ciklus amíg i < m
16:         k ← k + 1
17:         P[k] ← (i, j)
18:         i ← i + 1
19:     ciklus vége
20:     ciklus amíg j < n
21:         k ← k + 1
22:         P[k] ← (i, j)
23:         j ← j + 1
24:     ciklus vége
25:     k ← k + 1
26:     P[k] ← (i, j)
27:     vissza P
28: függvény vége
```

Felhasznált változók és függvények

- C: Kincsek értékét tartalmazó kétdimenziós tömb.
- m: A C tömb sorainak száma.
- n: A C tömb oszlopainak száma.
- P: Tömb, melynek elemszáma $m + n - 1$. A P tömb adja meg egy optimális bejárás koordináta sorozatát.
- Létrehoz(**egész**)$[m + n - 1]$: Utasítás, mely létrehoz egy $m + n - 1$ elemű **egész** típusú tömböt.

Ha a korábban dinamikus programozással megoldott feladatra alkalmazzuk a 7.9. mohó algoritmust, akkor eredményül a $P = \{(1,1),(2,1),(3,1),(3,2),(3,3),(4,3),(4,4)\}$ koordináta sorozatot kapjuk, ahogy az a 7.9. ábrán is látható. Az így összegyűjtött kincsek mennyisége csak 44 lesz, ami elmarad a dinamikus programozási megközelítésnél kapott 61 értéktől. Ebből látszik, hogy a mohó megközelítés

nem az optimális megoldást adja eredményül. (Érdemes azért megemlíteni, hogy nem a legkevesebb kincset eredményező megoldást kaptuk.)

1	5	3	6
11	2	15	1
6	10	20	9
1	3	4	8

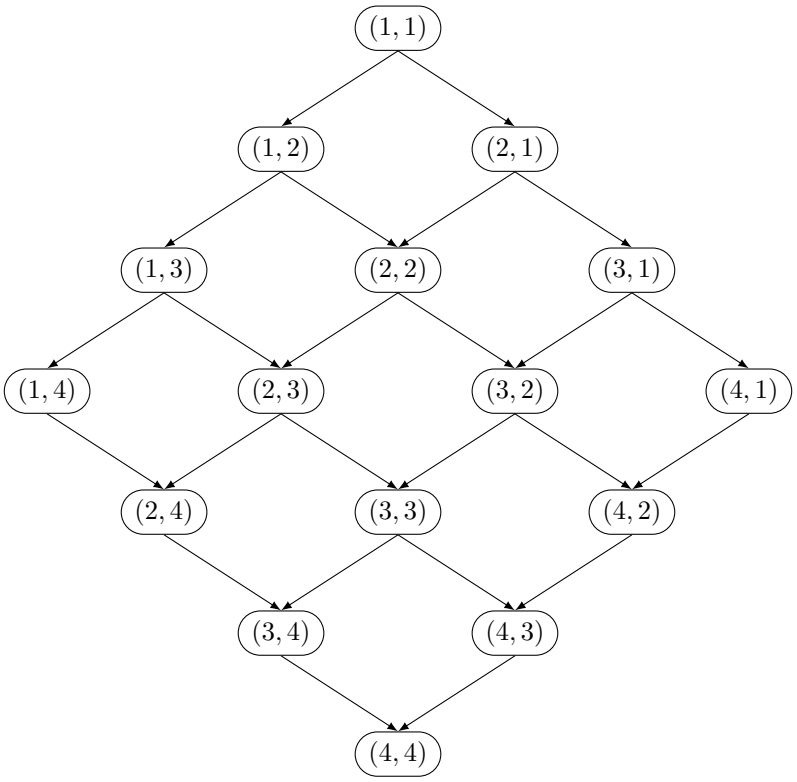
7.9. ábra. Kincsek begyűjtése mohó algoritmussal.

A mohó megközelítés tehát nem eredményezett optimális megoldást. Memória igénye viszont kisebb a mohó algoritmusnak, hiszen nem kellett egy segédtömböt eltárolnunk. Futási idő tekintetében is jobb eredményt kapunk, mint a dinamikus programozásnál, hiszen a 7.9. algoritmus futási ideje csak $O(m+n)$-es.

Érdemes észrevenni még egy különbséget a két használt módszer között. A konkrét bejárási útvonal meghatározása különböző irányban történt ugyanis. A dinamikus programozásnál az út végétől haladtunk vissza az eleje felé, míg a mohó megközelítés az út elejétől haladt a vége felé. Ezt úgy is ki szokás fejezni, hogy a mohó algoritmus felülről lefelé halad, míg a dinamikus programozás alulról felfelé haladva határozza meg az optimális megoldást.

Mit is jelent a felülről lefelé, illetve az alulról felfelé elnevezés? Ennek megértésében segít a 7.10. ábra, melyen feltüntettük az $(1, 1)$ koordinátájú pontból a $(4, 4)$ koordinátájú pontba jutás összes lehetőségét. Minden egyes csúcsból kiinduló lefelé haladó élek mutatják a lehetséges továbbhaladási irányt. A mohó megközelítés használatánál elindulunk a felső csúcsból és döntünk, hogy az alatta lévő két csúcs közül melyik felé menjünk tovább. A döntést követően továbblépünk, majd megint megvizsgáljuk az adott csúcs alatti két csúcsot. Ezt így tesszük addig, amíg el nem jutunk a legalsó csúcsba. A bejárás közben lesznek olyan csúcsok, amiket teljesen kihagyunk.

A dinamikus programozás használatánál minden egyes csúcsba eljutunk, hiszen minden mező esetén kiszámítjuk, hogy addig eljutva mennyi az összegyűjthető kincsek maximuma. Majd amikor már minden csúcsnál ismerjük a kincsösszegeket, akkor alulról indulva visszafelé, tehát felfelé haladva határozunk meg egy konkrét optimális bejárást. Így itt a tényleges optimális megoldás megadása alulról felfelé haladva történik úgy, hogy közben minden egyes csúcsnál pontosan tudjuk az addig összegyűjthető kincsek összértékét.



7.10. ábra. A kincsek mezőjének bejárási lehetőségei.

Mohó stratégia

Foglaljuk össze, hogy milyen jellemzőit találtuk meg eddig a mohó stratégiának. A mohó algoritmusok úgy alkotják meg egy probléma optimális megoldását, hogy választások sorozatát hajtják végre. Az algoritmus minden egyes döntési helyzetben azt a lehetőséget választja, ami az adott helyzetben optimálisnak tűnik. Ez a módszer nem mindig ad globális, a teljes feladatra vonatkozó optimális megoldást, de néhány esetben igen.

A mohó algoritmusok általában olyankor alkalmazhatók, amikor két tulajdonság teljesül. Ezek a mohó választási tulajdonság, illetve az optimális részproblémák tulajdonság.

A *mohó választási tulajdonság* azt jelenti, hogy a globális optimális megoldás elérhető lokális optimumok választásával. A mohó stratégia alkalmazásánál minden döntési helyzetben a lokális optimumot választjuk, ezért ez a tulajdonság szükséges feltétel a mohó stratégiával való megoldhatósághoz. A mohó választási tulajdonság képezi a lényeges különbséget a mohó algoritmusok és a dinamikus programozás között. Ez az, amit az előző alfejezetben fentről lefelé, illetve alulról felfelé módszerként említettünk. A mohó stratégia lokális optimumokon keresztül fentről lefelé haladva oldja meg a problémát. A dinamikus programozás a részproblémák optimumainak ismeretében alulról felfelé haladva ad optimális megoldást a problémára.

A mohó stratégia alkalmazhatóságának másik feltétele az *optimális részproblémák tulajdonság* teljesülése. Ahogy a dinamikus programozásnál már láttuk, ez a tulajdonság azt jelenti, hogy az optimális megoldás felépíthető a részproblémák optimális megoldásából. A mohó stratégia csak abban az esetben ad optimális megoldást, ha teljesül az optimális részproblémák tulajdonság. Minden más esetben csak ún. lokális optimumot határoz meg.

Ütemezési feladatok

A mohó stratégiára kívánunk még néhány példát mutatni. Ütemezési feladatokat ismertetünk és oldunk meg, melyek mind optimális megoldást eredményeznek a mohó megközelítés alkalmazása mellett.

Esemény kiválasztási probléma

Az esemény kiválasztási probléma esetén erőforrás ütemezést kívánunk megvalósítani egymással versengő feladatok között. Egyetlen erőforrás áll rendelkezésünkre és azt szeretnénk minél több eseményhez hozzárendelni.

A pontos feladat a következőképpen fogalmazható meg. Adott n darab eseményünk, melyeknek ismerjük a kezdési és befejezési időpontjait. A kezdési időpontok az s tömbben, a befejezési időpontok pedig az f tömbben vannak eltárolva. Ezek az időpontok fixek, nem módosíthatók a feladat megoldása során. Értelemszerűen tudjuk azt is, hogy egy feladat kezdési időpontja soha nem haladja meg ugyanazon feladat befejezési időpontját, azaz $s[i] \leq f[i]$ minden $1 \leq i \leq n$ esetén. Ha egy eseményt kiválasztunk, azaz hozzárendeljük az adott erőforráshoz, akkor az $\big[s[i], f[i]\big[$ intervallumot foglalja el. Tehát a kezdési időpont benne van az időintervallumában, a befejezési időpont viszont nincs benne. Az i-edik és j-edik eseményt *kompatibilisnek* nevezzük, ha az $\big[s[i], f[i]\big[$ és $\big[s[j], f[j]\big[$ intervallumok nem fedik egymást, azaz $s[i] \geq f[j]$ vagy $s[j] \geq f[i]$. A feladatunk az, hogy kiválasszuk páronként kompatibilis eseményeknek egy legnagyobb elemszámú halmazát. Ezt úgy is mondhatjuk, hogy az adott erőforráshoz minél több páronként kompatibilis eseményt szeretnénk hozzárendelni. Fontos megemlíteni, hogy nem az a célunk, hogy az erőforrás a legtöbb ideig szolgáljon ki eseményeket, hanem ez, hogy minél több eseményt kiszolgáljon.

Mivel mohó megközelítést kívánunk követni, először el kell döntenünk, hogy miben is kívánunk mohók lenni. A stratégiánk az lesz, hogy az események befejezési időpontjában leszünk mohók. Mégpedig azért ebben, mert úgy gondoljuk, hogy ha először a legkorábban befejeződő eseményt választjuk ki, akkor még a lehető legtöbb esemény marad meg, amik közül további választást eszközölhetünk. Ennek érdekében először úgy rendezzük az eseményeket, hogy a befejezési időpontjuk szerint növekvő rendezettségben legyenek. Ezután kiválasztjuk a legkorábban véget érő eseményt, és hozzárendeljük az erőforráshoz. Majd kiválasztjuk a soron következő azon eseményt, amely az eddig kiválasztottal kompatibilis, tehát a kezdési időpontja nem kisebb a kiválasztott esemény befejezési időpontjával. Ezt a metódust követjük mindaddig, amíg az események sorozatának végére nem érünk.

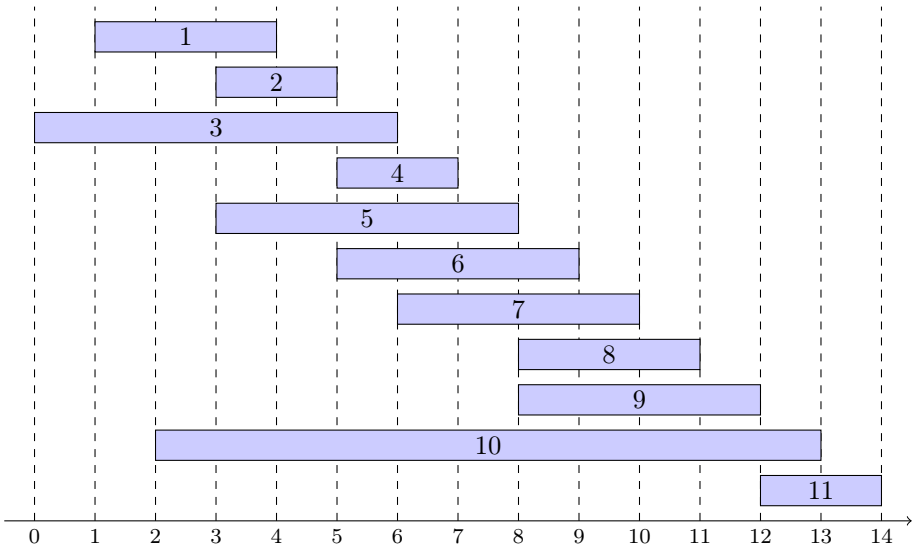
Az esemény kiválasztást a 7.10. algoritmus valósítja meg. Az algoritmus bemenete az s tömb, melyben az események kezdési időpontjai vannak. Az f növekvő módon rendezett tömbben adjuk meg a megfelelő események befejezési időpontjait. Az s és f tömb is n elemet tartalmaz. Az algoritmus kimenete a kiválasztott események indexeit tartalmazó A tömb.

Az algoritmus elején kiválasztjuk az első eseményt, ezért az A halmazba bekerül az 1-es index (ld. 2. sor). Az aktuálisan utoljára kiválasztott elem indexét az *utolsó* változóban tároljuk el. Ez azért szükséges, mert a következő kiválasztott esemény az lesz, amely ezen esemény befejezésénél nem kezdődik korábban. Az *utolsó* változó kezdeti értéke 1 lesz (ld. 3. sor). Meg kell vizsgálnunk a további eseményeket, hogy azok kompatibilisek-e az utoljára kiválasztott eseménnyel. Ennek érdekében szükséges egy bejárást megvalósító ciklus. Mivel az összes fennmaradó eseményt meg kell vizsgálni, ezért számlálós ciklust használunk (ld. 4. sor). A cikluson belül megvizsgáljuk, hogy az i-edik esemény kezdési időpontja hogyan viszonyul az utoljára kiválasztott esemény befejezési időpontjához (ld. 5. sor). Ha nem kezdődik korábban, akkor kompatibilis az utoljára kiválasztott eseménnyel – és minden más kiválasztott eseménnyel is –, ezért betesszük az A halmazba (ld. 6. sor) és az *utolsó* változó értékét is módosítjuk (ld. 7. sor). Amikor elfogyott minden esemény, tehát a ciklus futása véget ér, akkor már csak vissza kell adni a kiválasztott események indexeinek A halmazát (ld. 10. sor).

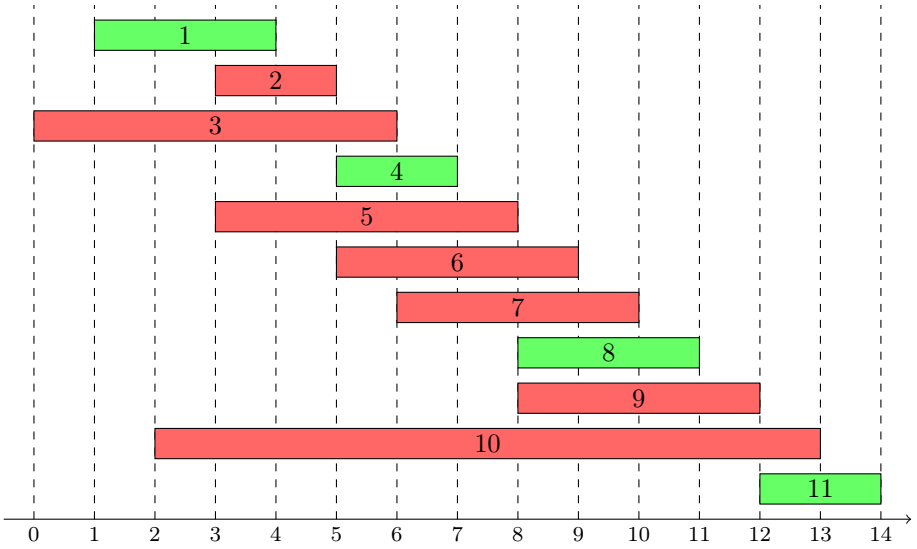
7.9. Példa. A 7.11a. ábrán megadtunk 11 eseményt, melyek a befejezési időpontjaik szerint rendezettek. Feladatunk, hogy kiválasszuk közülük a lehetőt legtöbb olyat, melyek páronként kompatibilisek.

A 7.10. algoritmus használatával a 7.11b. ábrán láthatjuk, hogy mely események kerültek kiválasztásra. Esetünkben tehát 4 eseményhez tudjuk ugyanazt az erőforrást hozzárendelni. Vegyük észre, hogy nem csak ez a megoldás létezik. Számos olyan más megoldást is tudunk adni, melyek 4 kölcsönösen kompatibilis eseményt tartalmaznak. Ilyen például a második, negyedik, nyolcadik és tizenegyedik események kiválasztása is. ¶

Futási idő elemzése. Az esemény kiválasztási problémát megoldó 7.10. algoritmus futási ideje $O(n)$-es, mivel egyetlen számlálós ciklus található benne. ♣



(a) Befejezési időpont szerint rendezett események.



(b) Kiválasztott események.

7.11. ábra. Esemény kiválasztás.

7.10. Algoritmus Esemény kiválasztás

Bemenet: s – **idő tömb**, f – **idő rendezett tömb**, n – **egész** (*tömbök mérete*)
Kimenet: A – **egész halmaz**

```
1: függvény EseményKiválasztás(s : idő tömb, f : idő tömb, n egész)
2:     A ← {1}
3:     utolsó ← 1
4:     ciklus i ← 2-től n-ig
5:         ha s[i] ≥ f[utolsó] akkor
6:             A ← A ∪ {i}
7:             utolsó ← i
8:         elágazás vége
9:     ciklus vége
10:    vissza A
11: függvény vége
```

Felhasznált változók és függvények

- s: Az események kezdeti időpontjait tartalmazó tömb.
- f: Az események befejezési időpontjait tartalmazó tömb. Az f tömb növekvő módon rendezett, emiatt alkalmazható a mohó stratégia.
- n: Az s és f tömb elemszáma.
- A: A kiválasztott események indexeit tartalmazó halmaz.
- $utolsó$: Az aktuálisan utolsóként kiválasztott esemény indexe.

Felmerülhet a kérdés, hogy a 7.10. algoritmus valóban globálisan optimális megoldást eredményez-e minden lehetséges bemenet esetén. Ennél az egy algoritmusnál adunk egy bizonyítást arra, hogy a mohó stratégia most tényleg optimális megoldást eredményez.

A bizonyítás első lépéseként belátjuk, hogy az automatikusan kiválasztott első esemény tényleg benne lehet az optimális megoldásban. Jelöljük a lehetséges megoldások halmazát S-sel. Tegyük fel, hogy az $A \subset S$ egy optimális megoldás, és legyenek az A-beli események a befejezési idő szerint növekvő módon rendezettek. Tegyük fel, hogy az A-beli első esemény a k-adik. Ha $k = 1$, akkor beláttuk, hogy az első esemény benne lehet egy optimális megoldásban. Ha $k \neq 1$, akkor tekintsük a $B \subset S$ megoldást, amelyet úgy kapunk, hogy A-ból elhagyjuk a k-t és hozzávesszük az 1-et. Tehát $B = (A \setminus \{k\}) \cup \{1\}$. Mivel $f[1] \leq f[k]$, így a B-beli események páronként kompatibilisek és B pontosan annyi elemet tartalmaz, mint A. Ez viszont azt jelenti hogy B is optimális megoldás. Tehát minden esetben van olyan optimális megoldás, melyben az első elem benne van.

A bizonyítás második lépéseként azt kell belátnunk, hogy az első elemet követő elemek választása esetén is optimális megoldást kapunk. Ennek érdekében azt kell megvizsgálnunk, hogy az $A' = A \setminus \{1\}$ optimális megoldása-e az $S' = \{i \in S : s[i] \geq f[1]\}$ eseményeket tartalmazó problémának. Tegyük fel, hogy találunk olyan B' megoldását az S' problémának, amely több eseményt tartalmaz mint A'. Ebben az esetben az első eseményt hozzáadva B'-höz, az S problémának olyan megoldást kapjuk, mely több eseményt tartalmaz mint A. Ez viszont ellentmond annak, hogy A optimális megoldás. Tehát minden mohó választás után olyan problémánk marad mint az eredeti. Így az eredetileg kiválasztott eseménnyel kompatibilis események közül újra az elsőt választhatjuk mohó módon.

Esemény elkülönítési probléma

Az esemény elkülönítési problémánál adott n darab esemény, melyeknek ismerjük a kezdési és befejezési időpontjaikat. Ezek rendere az s és az f tömbben vannak eltárolva. Ezek az időpontok a feladat megoldása során nem módosíthatók. Minden egyes eseményhez hozzá kell rendelnünk egy erőforrást. Feladatunk, hogy minden eseményt hozzárendeljünk erőforrásokhoz úgy, hogy a lehető legkevesebb erőforrást használjuk. (Az biztos, hogy van annyi erőforrásunk, amennyi szükséges.)

A problémát most úgy fogjuk megoldani, hogy az eseményeket kezdési időpontjuk szerint rendezzük. Ezt követően az első eseményt hozzárendeljük az első erőforráshoz. A következő eseménynél megnézzük, hogy hozzárendelhető-e az első erőforráshoz, azaz kompatibilis-e az első erőforráshoz már hozzárendelt eseménnyel. Ha hozzárendelhető, akkor ezt meg is tesszük. Ha nem, akkor viszont hozzárendel-

jük a második erőforráshoz A soron következő eseménynél is megvizsgáljuk, hogy az első erőforráshoz hozzárendelhető-e. Ha igen, akkor így teszünk, ha viszont nem, akkor továbblépünk a második erőforrásra. Ha ahhoz hozzárendelhető, akkor ezt tesszük, ha ahhoz sem, akkor a harmadik erőforrást is használatba vesszük. Ezen logika mentén járunk el végig, tehát minden erőforrást sorban megvizsgálunk, hogy az adott elemet hozzá tudjuk-e rendelni. Ha egyikhez sem, akkor új erőforrást vonunk be a feladat megoldásába.

A konkrét megvalósítást a 7.11. algoritmusban mutatjuk be. Az algoritmus bemenete az események kezdeti időpontjait tartalmazó s tömb, valamint a befejezési időpontok f tömbje. Mindkét tömb n elemet tartalmaz. Mivel az események a kezdési idők alapján rendezettek, ezért s rendezett tömb. Kimenetként egy n elemű A tömböt állítunk elő, melynek minden egyes eleme meghatározza, hogy a megfelelő eseményhez melyik erőforrást rendeltük hozzá.

7.11. Algoritmus Esemény elkülönítés

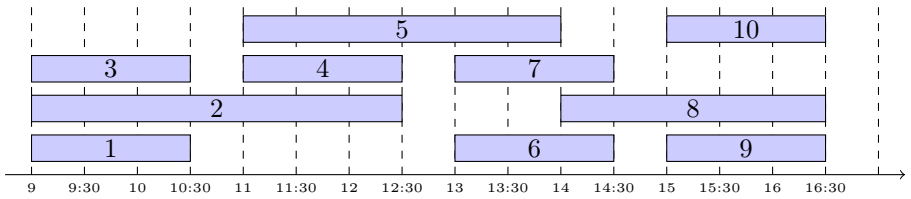
Bemenet: s – **idő rendezett tömb**, f – **idő tömb**, n – **egész** (*tömbök mérete*)
Kimenet: A – **egész tömb**

```
1:  függvény EseményElkülönítés(s : idő rendezett tömb, f : idő tömb, n : egész)
2:      A ← Létrehoz(egész)[n]
3:      utolsó ← 0
4:      ciklus i ← 1-től n-ig
5:          j ← 1
6:          ciklus amíg (j ≤ utolsó) ∧ ¬KompatiblisEseményErőforrással(A, s, f, i, j)
7:              j ← j + 1
8:          ciklus vége
9:          ha j ≤ utolsó akkor
10:             A[i] ← j
11:         különben
12:             utolsó ← utolsó + 1
13:             A[i] ← utolsó
14:         elágazás vége
15:     ciklus vége
16:     vissza A
17: függvény vége
```

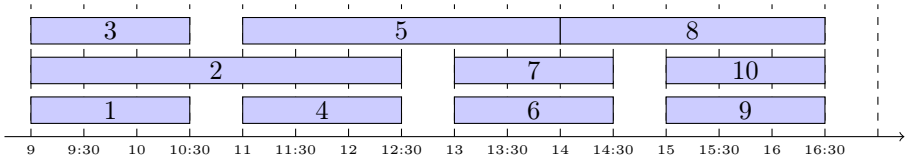
Felhasznált változók és függvények

- s: Az események kezdési időpontjainak növekvő módon rendezett tömbje.
- f: Az események befejezési időpontjainak tömbje.
- n: Az s és f tömbök elemeinek száma.
- A: n elemű tömb, melynek i-edik eleme megadja, hogy az i-edik eseményt hányadik erőforráshoz rendelktük hozzá.
- *utolsó*: A használatba vett erőforrások legnagyobb indexe.
- Létrehoz(**egész**)[n]: Utasítás, mely létrehoz egy n elemű **egész** típusú tömböt.
- KompatiblisEseményErőforrással(A, s, f, i, j): A függvény megadja, hogy az i-edik esemény kompatibilis-e a j-edik erőforráshoz már hozzárendelt eseményekkel. Ennek meghatározásához szükséges a kezdési (s) és befejezési (f) időpontok, valamint az erőforrás hozzárendelések A tömbjének ismerete.

Az *utolsó* változóban tároljuk el, hogy hány darab erőforrást vontunk már használatba. Kezdetben ez a szám 0 lesz (ld. 3. sor). A 4. sorban kezdődő számlálós ciklus végighalad az összes eseményen. A cikluson belül megvizsgáljuk, hogy melyik erőforrást lehet az i-edik eseményhez rendelni. Kiindulásként az első erőforrással próbálkozunk majd, ezért a j változót az 1 értékkel inicializáljuk (ld. 5. sor). A 6. sorban kezdődő ciklusban megvizsgáljuk, hogy a már használatba vett erőforrások közül valamelyikhez hozzárendelhető-e az i-edik esemény. Ha lehetséges a hozzárendelés (ld. 9. sor), akkor ezt meg is tesszük (ld. 10. sor). Egyéb esetben pedig új erőforrást veszünk használatba (ld. 12. sor), majd ezt összekapcsoljuk az i-edik eseménnyel (ld. 13. sor). Az algoritmus végén az A tömböt adjuk vissza kimenetként (ld. 16. sor).



(a) Az események egy lehetséges elkülönítése.



(b) Az események egy optimális elkülönítése.

7.12. ábra. Esemény elkülönítési probléma.

7.10. Példa. Adott 10 eseményünk, melyeket a lehető legkevesebb erőforráshoz kívánjuk hozzárendelni. A 7.12a. ábrán megadjuk egy lehetséges elkülönítését az eseményeknek. Minden egyes sor egy erőforrást jelöl. Látható, hogy ebben az esetben 4 erőforrás szükséges.

A 7.11. algoritmus használatával egy másik lehetséges elkülönítés adódik, ami a 7.12b. ábrán látható. Ebben az esetben csak 3 erőforrás szükséges a feladat megvalósításához. ¶

Futási idő elemzése. A 7.11. algoritmus futási ideje nagymértékben függ attól, hogy hány erőforrás használata szükséges. Ez azért van így, mert a belső ciklus maximális végrehajtási száma a használatba vett erőforrások számával egyezik meg. Legrosszabb eset az, ha az eseménypárok nem kompatibilisek. Ilyenkor ugyanis n darab erőforrás szükséges, így a két egymásba ágyazott ciklus miatt az algoritmus futási ideje $O(n^2)$-es. ♣

Ütemezés késés minimalizálással

A késés minimalizálással történő ütemezésnél azt kell meghatároznunk, hogy rögzített időtartamú, határidővel rendelkező feladatokat hogyan tudunk úgy ütemezni, hogy a lehető legkevesebb késés álljon elő. Adott tehát n darab feladat, melyek egyetlen és ugyanazon erőforrást igényelik. Minden feladatnak ismerjük a teljesítéséhez szükséges időtartamot. Ezeket az időtartamokat a t tömbben tároljuk el. Minden feladatnak van határideje is, melyek értéke a d tömbben van eltárolva. Egy feladat késését kell tudnunk értelmezni. Ha a feladatot a határideje előtt tudjuk teljesíteni, akkor a késés mértéke 0. Viszont, ha csak a határidő után tudjuk befejezni, akkor a befejezési időpont és a határidő különbsége lesz a késés. Ha $f[i]$ jelöli az i-edik feladat befejezési időpontját (ez előre nem ismert) és $l[i]$ az i-edik feladat késését, akkor

$$l[i] = \max\{0, f[i] - d[i]\}. \tag{7.9}$$

Célunk, hogy az összes feladat késésének maximumát minimalizáljuk. Ez azt jelenti, hogy jobb nekünk sok kicsi késés, mint egy nagy.

A megalkotandó algoritmusnál arra fogunk koncentrálni, hogy a feladatokat a határidejük szerint rendezetten ütemezzük. Azért így járunk el, mert ha a legkorábbi határidővel rendelkező feladatot tudjuk elsőként végrehajtani, akkor a késése biztos a lehető legkisebb lesz. Feltesszük tehát, hogy az események a határidejük szerint rendezettek. Ezt követően nincs más dolgunk, mint a feladatokat egymás után helyezni úgy, hogy páronként kompatibilisek legyenek, hiszen csak egy erőforrás áll rendelkezésünkre.

A problémát a 7.12. algoritmussal oldjuk meg. Az algoritmus bemenete a határidők d rendezett tömbje és az időtartamok t tömbje. Mindkét tömb n elemű. Kimenetként a kezdési és befejezési idők s és f tömbjeit állítja elő az algoritmus.

Az algoritmus elején létrehozzuk a kimeneti s és f tömböket (ld. 2. és 3. sorok). Az *utolsó* változó jelzi, hogy a soron következő feladat mikor kezdődhet el legkorábban. Kezdeti értéke a 0 időpillanat (ld. 4. sor). Ezután a határidő szerint feladatokat kell időzíteni, amit az 5. sorban kezdődő ciklussal valósítunk meg. Az aktuális feladat kezdési ideje az *utolsó* változóban tárolt érték lehet (ld. 6. sor).

7.12. Algoritmus Ütemezés késés minimalizálással

Bemenet: d – **idő rendezett tömb**, t – **idő tömb**, n – **egész** (*tömbök mérete*)
Kimenet: s – **idő tömb**, f – **idő tömb**

```
1:  függvény KésésMinimalizálás(d : idő rendezett tömb, t : idő tömb, n : egész)
2:      s ← Létrehoz(idő)[n]
3:      f ← Létrehoz(idő)[n]
4:      utolsó ← 0
5:      ciklus i ← 1-től n-ig
6:          s[i] ← utolsó
7:          f[i] ← s[i] + t[i]
8:          utolsó ← f[i]
9:      ciklus vége
10:     vissza (s, f)
11: függvény vége
```

Felhasznált változók és függvények

- d: Határidők növekvő módon rendezett tömbje.
- t: Időtartamok tömbje. A $d[i]$ határidejű feladat időtartama a $t[i]$.
- n: A d és t tömb elemszáma.
- s: A kezdési időpontok kimeneti tömbje.
- f: A befejezési időpontok kimeneti tömbje.
- *utolsó*: Az aktuálisan utolsóként időzített feladat befejezési időpontja.
- ← Létrehoz(**idő**)[n]: Utasítás, mely létrehoz egy n elemű **idő** típusú tömböt.

Ennek a feladatnak a befejezési ideje a kezdési idejének és az időtartamának az összege (ld. 7. sor). A következő feladat majd az aktuális feladat befejezési idejekor kezdődhet el, ezért aktualizáljuk az *utolsó* változó értékét is (ld. 8. sor). Miután minden feladatot ütemeztünk a kezdési és befejezési időpontok tömbjeit adja vissza az algoritmus (ld. 10. sor).

7.11. Példa. Adott 6 feladat, melyek időtartamait és határidejeit a 7.3. táblázatban adjuk meg. Úgy kell ütemeznünk a feladatokat, hogy az egyes feladatok késésének maximuma minimális legyen.

	1	**2**	**3**	**4**	**5**	**6**
$\mathbf{t_i}$	3	2	1	4	3	2
$\mathbf{d_i}$	6	8	9	9	14	15

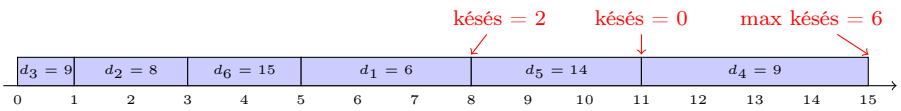
7.3. táblázat. Feladatok időtartama és határidejeik.

A 7.13a. ábrán egy ütemezés, mely esetén a maximális késés 6 időegység.

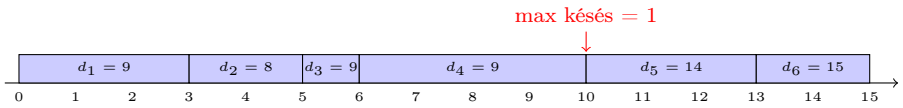
A 7.12. algoritmus alkalmazásával a 7.13b. ábrán látható ütemezést kapjuk eredményül, melynél a legnagyobb késés csak 1 időegység.

¶

Futási idő elemzése. Mivel a 7.12. algoritmusban egyetlen ciklus van, mely pontosan n-szer fut le, ezért az algoritmus futási ideje $O(n)$-es. ♣



(a) Egy lehetséges ütemezés.



(b) Optimális ütemezés.

7.13. ábra. Ütemezés késés minimalizálással.

8. fejezet

Kupacrendezés

A kupacrendezés[1] a 3. fejezetben megismert rendezésekhez hasonlóan helyben történő rendezést valósít meg egy tömbben. A rendezés megvalósításához viszont egy speciális adatszerkezetet kell megvalósítani, melyet kupacnak nevezünk.

A fejezet 8.1. alfejezetében definiáljuk azokat a fogalmakat, amelyek ahhoz szükségesek, hogy megértsük milyen adatszerkezetet tekintünk kupacnak. Ezt követően a 8.2. alfejezetben megismerkedünk a KUPACOL rekurzív eljárással, amely a kupactulajdonság fenntartását biztosítja. A 8.3. alfejezetben ismertetjük, hogy milyen módon lehet egy tetszőleges tömbből kupacot építeni. Mindezek ismeretében a 8.4. alfejezetben adjuk meg a kupacrendezést megvalósító algoritmust.

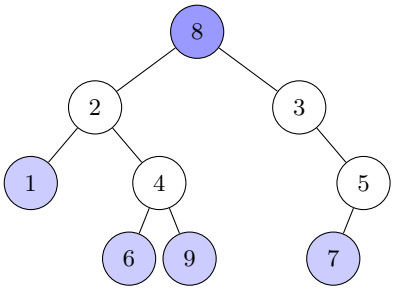
[1] Angolul: heapsort

Definíciók

Annak érdekében, hogy a kupac fogalmát értelmezni tudjuk, először néhány más adatszerkezetet kell definiálnunk. Ezeket most nem teljes precizitással tesszük meg, az Algoritmusok, adatszerkezetek II. jegyzetben adjuk meg ezek pontosabb és részletesebb bemutatását.

Bináris fának[2] nevezünk egy olyan adatszerkezetet, amelyben minden egyes elemnek (más szóval csúcsnak) legfeljebb két gyermeke van. A gyermek elemek között megkülönböztetjük a bal- és jobboldali gyermeket. A bináris fának egyetlen gyökere van, melyet ábrázolásnál – a valódi fákkal ellentétben – felül jelenítünk meg. Azokat a csúcsokat, amelyeknek egyetlen gyermeke sincs, leveleknek nevezzük. Értelem szerint ezeket ábrázoljuk a fa legalján. A bináris fa magassága alatt a levelekből gyökérbe vezető utak közül a leghosszabb hosszát értjük.

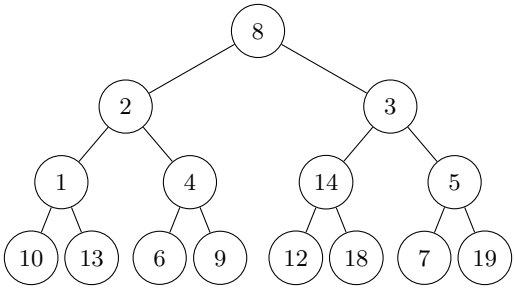
A 8.1. ábrán bemutatunk egy bináris fát. Ennek gyökerében a 8 van eltárolva, melyet kék színnel emeltünk ki. A gyökér elemnek két gyermeke van: baloldali gyermeke a kettes és a jobboldali gyermeke a hármas értéket tartalmazó csúcs. Látható a példából, hogy minden csúcsnak legfeljebb két gyermeke van, de például a hármas és ötös értéket tartalmazó csúcsoknak csak egy-egy. A hármas értékű csúcsnak csak jobboldali, míg az ötös értéket tartalmazó csúcsnak csak baloldali gyermeke van. Azok a csúcsok, amelyeknek egyetlen gyermeke sincs, a levelek. Ezeket világoskék kitöltéssel jelöltük. A bemutatott bináris fa magassága 3, mivel a legalsó levelektől három lépésben jutunk el a gyökérig.



8.1. ábra. Bináris fa. A fa tetején található a fa gyökere (kék kitöltés), a fa alján pedig a levelek (világoskék kitöltés). A fa hármas magasságú.

A bináris fákban szintekről is beszélhetünk. A 8.1. ábrán látható bináris fa első szintjén – mint minden más bináris fa esetén is – csak a gyökér elem található. A második szinten két elem van: a kettes és a hármas értékeket tartalmazó. A harmadik szinten találjuk az egyes, négyes és ötös értékeket, a negyedik szinten pedig a hatos, kilences és hetes értékeket.

Egy bináris fát *teljes bináris fának* tekintünk, ha minden szintjén teljesen kitöltött, azaz az első szinten egy elem, a másodikon két elem, a harmadikon négy elem, az n-ediken pedig 2^{n-1} elem található. Könnyen belátható, hogy egy teljes bináris fában pontosan $2^n - 1$ elem van, ha az n-edik szint a legalsó. A teljes bináris fákról az is kijelenthető, hogy minden elemnek pontosan kettő vagy nulla gyermeke van, és minden nulla gyermekkel rendelkező csúcs (azaz levél) a legalsó szinten van. Teljes bináris fára mutatunk be példát a 8.2. ábrán.

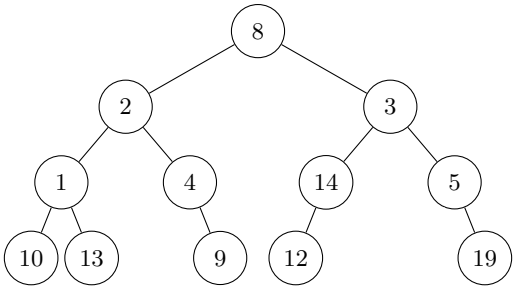


8.2. ábra. Teljes bináris fa. A fa minden szintje teljesen kitöltött.

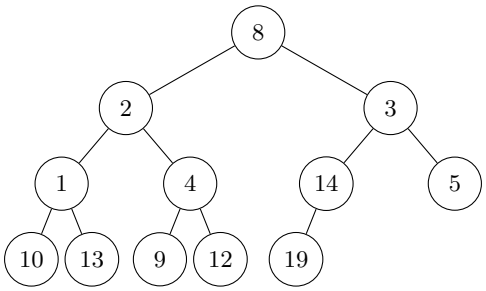
[2] Angolul: binary tree

Majdnem teljes bináris fáról beszélünk, ha csak az alsó szinten hiányzik elem a teljes bináris fából. A *majdnem teljes balról kitöltött bináris fa* pedig olyan majdnem teljes bináris fa, melynek az alsó szintje balról jobbra haladva van feltöltve.

A 8.3. ábrán egy majdnem teljes bináris fát mutatunk be, a 8.4. ábrán pedig egy majdnem teljes balról kitöltött bináris fa látható.

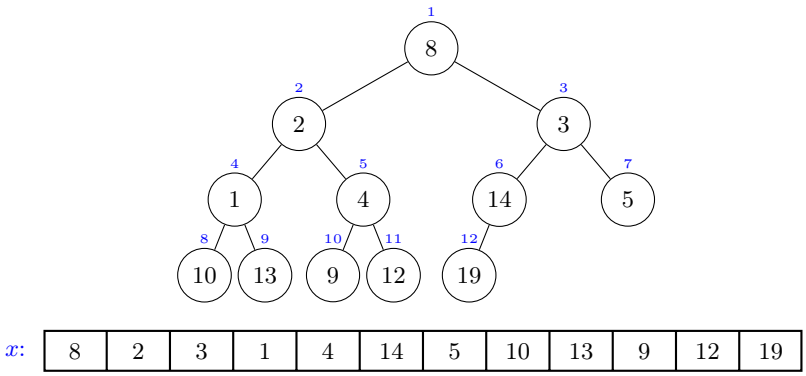


8.3. ábra. Majdnem teljes bináris fa. A fának csak a legalsó szintjén hiányoznak elemek.



8.4. ábra. Majdnem teljes balról kitöltött bináris fa. A fának csak a legalsó szintjén hiányoznak elemek, viszont a legalsó szint balról van feltöltve.

A továbbiakban csak a majdnem teljes balról kitöltött bináris fákkal foglalkozunk. Ezek a fák tömbökkel reprezentálhatók úgy, hogy a fa gyökere a tömb első eleme, majd a második szint elemei balról jobbra haladva a tömb második és harmadik elemei, a harmadik szint elemei balról jobbra haladva a tömb negyedik, ötödik, hatodik és hetedik elemei, és így tovább. A 8.5. ábrán megadtunk egy majdnem teljes balról kitöltött bináris fát a hozzárendelt indexekkel együtt. Az indexek ismeretében a fa már egyértelműen reprezentálható egy tömbbel. Belátható, hogy minden majdnem teljes balról kitöltött bináris fának egyértelműen megfeleltethető egy tömb, illetve minden tömbnek egyértelműen megfeleltethető egy majdnem teljes balról kitöltött bináris fa.



8.5. ábra. Majdnem teljes balról kitöltött bináris fa és tömb megfeleltetése.

Vizsgáljuk meg, hogy egy majdnem teljes balról kitöltött bináris fában mi az i-edik elem baloldali, illetve jobboldali gyermekének indexe. A 8.5. ábra alapján azt látjuk, hogy az i-edik elem baloldali

gyermeke a $2 \cdot i$ indexű elem, jobboldali gyermekének indexe pedig $2 \cdot i + 1$, feltéve, hogy van az i-edik elemnek baloldali és jobboldali gyermeke. Ezt általánosan is be tudjuk bizonyítani.

Tegyük fel, hogy a k-adik szinten van az i indexű elem. Ha van az i-edik elemnek baloldali gyermeke, akkor a k-adik szint biztosan teljesen fel van töltve. Tudjuk, hogy a k-adik szint végénél a $2^k - 1$ indexű elem található. Így a k-adik szinten az i indexű elem jobb oldalán összesen $2^k - 1 - i$ darab elem található. Az i-edik elem bal oldalán pedig $i - \left(2^{k-1} - 1\right) - 1$ darab elem van a k-adik szinten. A $k+1$-edik szinten az i-edik elem baloldali gyereke előtt pontosan kétszer annyi elem áll, mint ahány elem a k-adik szinten volt az i-edik elem előtt, azaz $2 \cdot \left(i - \left(2^{k-1} - 1\right) - 1\right)$. Mivel az i-edik elemből úgy jutunk el az ő baloldali gyermekéhez, hogy feltöltjük a k-adik szinten i-től jobbra lévő helyeket, majd a $k+1$-edik szinten az i-edik elem baloldali gyermekétől balra lévő helyeket, aztán pedig még egyet jobbra lépünk. Ezért az i-edik elem baloldali gyermekének indexe:

$$i + \left[2^k - 1 - i\right] + \left[2 \cdot \left(i - \left(2^{k-1} - 1\right) - 1\right)\right] + 1 = 2 \cdot i.$$

A jobboldali gyermek indexe eggyel nagyobb mint a baloldali gyermek indexe, ezért az $2 \cdot i + 1$ lesz minden esetben, ha létezik jobboldali gyermek. Mindebből következik, hogy az i-edik elem szülőjének indexe pedig $\lfloor n/2 \rfloor$.

Vegyük észre azt is, hogy egy n elemű majdnem teljes balról kitöltött bináris fának a levelei az $\lfloor n/2 \rfloor + 1$ indexű elemtől az n indexű elemig tartanak, hiszen az n indexű elem szülője a legnagyobb olyan indexű elem, amely nem levél. A fa gyökérelemének indexe minden esetben az 1.

Érdemes lenne azt is megvizsgálni, hogy egy n elemű majdnem teljes balról kitöltött bináris fának hány szintje van. Ha a szintek számát k-val jelöljük, akkor biztos igaz, hogy

$$2^{k-1} - 1 < n \leq 2^k - 1.$$

Ebből viszont következik, hogy

$$k - 1 < \log_2(n+1) \leq k.$$

Mivel a szintek száma egész szám, ezért

$$k = \lceil \log_2(n+1) \rceil.$$

Az eddigi definíciók mind szükségesek voltak ahhoz, hogy definiálni tudjuk a kupacot. *Kupac*nak[3] nevezzük azt a majdnem teljes balról kitöltött bináris fát, melyben minden elemre igaz, hogy az adott elem nagyobb értékű mindkét gyermeke értékénél. Ezt a tulajdonságot kupactulajdonságnak nevezzük.

Mivel a kupacok majdnem teljes balról kitöltött bináris fák, ezért tömbökkel reprezentálhatók. Így a kupactulajdonság úgy is kimondható, hogy egy n elemű kupac tömb reprezentációjában minden 1 és $\lfloor n/2 \rfloor$ közé eső i-re igaz, hogy

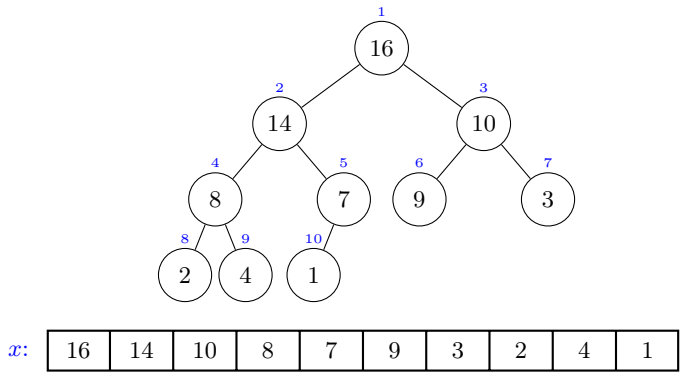
$$\begin{aligned} x[i] &\geq x[2 \cdot i], \\ x[i] &\geq x[2 \cdot i + 1]. \end{aligned}$$

Utóbbi természetesen csak akkor áll fenn az $\lfloor n/2 \rfloor$-edik elem esetén, ha n páratlan érték, azaz a kupac utolsó eleme a szülőjének jobboldali gyermeke, egyéb esetben ugyanis kiindexelnénk a tömbből.

A 8.6. ábrán bemutatunk egy kupacot bináris fa és tömb reprezentációval is.

Összefoglalva: kupacnak tekintjük azt a majdnem teljes balról kitöltött bináris fát, mely teljesíti a kupactulajdonságot és tömbként reprezentálható.

[3] Angolul: heap

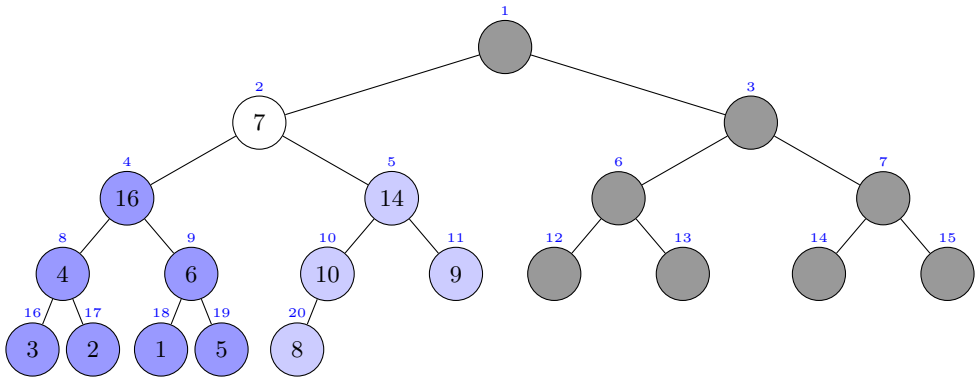


8.6. ábra. Kupac bináris fa és tömb reprezentációja.

Kupacolás

A kupacrendezés megvalósításához először szükséges pár segédalgoritmus megvalósítása. Az első ezek közül a kupacolás, vagy más néven a kupactulajdonság fenntartása.

A kifejlesztendő algoritmus egy olyan tömböt kap bemenetként, melynek ha nézzük a majdnem teljes balról kitöltött bináris fa reprezentációját, akkor az i-edik elem baloldali gyermekéből, mint gyökérből kiinduló részfa kupacnak tekinthető, valamint az i-edik elem jobboldali gyermekéből, mint gyökérből kiinduló részfa is teljesíti a kupacokkal szembeni elvárásokat. Például a 8.7. ábrán a 4-es és 5-ös indexű elemekből, mint gyökérből kiinduló részfák kupacnak tekinthetők. Viszont a 2-es indexű elemből, mint gyökérből kiinduló részfa már nem kupac, hiszen a kettes indexű elem gyermekei kisebbek nála.



8.7. ábra. A 4-es és az 5-ös indexű elemből, mint gyökérből kiinduló részfák kupacnak tekinthetők, viszont a 2-es indexű elemre már nem teljesül a kupac tulajdonság.

A kupacolás erre a problémára nyújt megoldást, tehát ha a $2 \cdot i$ indexű elemből, mint gyökérből induló részfa kupac, és a $2 \cdot i + 1$ indexű elemből, mint gyökérből induló részfa is kupac, akkor biztosítja, hogy az i-edik elemből, mint gyökérből induló részfa is kupac legyen. Ezt úgy éri el az algoritmus, hogy kiválasztja az i, $2 \cdot i$ és $2 \cdot i + 1$ indexű elemek közül a legnagyobb értékűt és ezt cserével az i-edik helyre teszi. Ha eleve az i-edik elem volt a három vizsgált elem közül a legnagyobb, akkor nincs további teendő. Viszont ha az i-edik elem valamely gyermeke volt a legnagyobb, akkor az adott gyermek értéke kerül az i-edik helyre, az i-edik elem pedig az adott gyermek helyére. Ilyenkor azonban az adott gyermek elemből mint gyökérből induló részfa már nem biztos, hogy kupac, ezért arra rekurzívan meg kell hívni a kupacolást. A rekurzív hívások záros időn belül véget érnek, mert a hívások során eljutunk olyan elemhez, ami már levél, tehát nincs gyermeke.

A 8.1. algoritmusban írjuk le a kupacolás konkrét megvalósítását. Az algoritmust megvalósító eljárás bemenete az az x tömb, amelyben a kupacolást végre akarjuk hajtani. Természetesen az x tömb n méretét is ismerjük. Meg kell adni a kupac k elemszámát is, amely általában megegyezik a tömb méretével. Ennek szükségességét később fogjuk megérteni. Ezeken kívül még az i index az eljárás bemenete. Az algoritmusban feltételezzük, hogy a $2 \cdot i$ és $2 \cdot i + 1$ indexű elemekből mint gyökérből induló részfák már kupacok, vagy nem léteznek ezek az elemek.

Az algoritmus 2. és 3. soraiban meghatározzuk az i indexű elem bal- és jobboldali gyermekeinek indexeit. Ezek nem biztos, hogy valódi indexek, ezért ezt majd vizsgálnunk kell. Ezt követően el kell döntenünk, hogy a fókuszban lévő három elem közül (i, $2 \cdot i$ és $2 \cdot i + 1$ indexűek) melyik a legnagyobb. Ennek érdekében a 4. sorban megnézzük, hogy létezik-e baloldali gyermek a kupacban ($bal \leq k$ feltétel), illetve ha létezik, akkor nagyobb-e az $x[i]$ elemnél. Ha létezik és nagyobb, akkor a baloldali gyermeket tekintjük maximálisnak, ezért az indexét eltároljuk a max változóban (ld. 5. sor). Egyéb esetben az i-edik elem lesz az aktuális maximum (ld. 7. sor). Ezt követően megvizsgáljuk, hogy benne van-e a kupacban az i-edik elem jobboldali gyermeke ($jobb \leq k$ feltétel) és nagyobb-e az eddigi maximumnál (ld. 9. sor). Ha igen, akkor a jobboldali a gyermek a legnagyobb a vizsgált három elem közül, ezért ennek az indexét tároljuk el a max változóban (ld. 10. sor). A 12. sorban megvizsgáljuk, hogy az i-edik elem volt-e a legnagyobb. Ha igen, akkor nincs szükség további teendőkre, hiszen a gyermekeiből mint gyökerekből kiinduló részfákról tudjuk, hogy kupacok, és az i-edik elemből mint gyökérből induló részfa is teljesíti a kupac tulajdonságot. Ha viszont valamelyik gyermeke a három vizsgált elem közül a

8.1. Algoritmus Kupactulajdonság fenntartása

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*), k – **egész** (*kupac mérete*), i – **egész**; ahol **T** összehasonlítható
Kimenet: x – **T tömb**

```
1:  eljárás KUPACOL(címszerint x : T tömb, n : egész, k : egész, i : egész)
2:      bal ← 2 · i
3:      jobb ← 2 · i + 1
4:      ha bal ≤ k ∧ x[bal] > x[i] akkor
5:          max ← bal
6:      különben
7:          max ← i
8:      elágazás vége
9:      ha jobb ≤ k ∧ x[jobb] > x[max] akkor
10:         max ← jobb
11:     elágazás vége
12:     ha max ≠ i akkor
13:         x[i] ↔ x[max]
14:         KUPACOL(x, n, k, max)
15:     elágazás vége
16: eljárás vége
```

Felhasznált változók és függvények

- x: A feldolgozandó tömb, amelyből kupacot akarunk készíteni.
- n: Az x tömb elemszáma.
- k: Az x tömb első k darab elemét vesszük figyelembe a kupac kialakítás során.
- i: Az aktuális index, amely alatti elemekre az algoritmus biztosítja a kupactulajdonság teljesülését, ha az i-edik elem alatti két részkupac már eleve kupactulajdonságú.
- *bal*: Az i-edik elem bal oldali gyerekének indexe.
- *jobb*: Az i-edik elem jobb oldali gyerekének indexe.
- *max*: Az i, a *bal* és *jobb* indexű elemek közül a legnagyobb elem indexe.

legnagyobb ($max \neq i$), akkor az adott gyermeket és az i-edik elemet megcseréljük (ld. 13. sor). Ekkor az i-edik elemnél már minden alatta lévő elem kisebb lesz, viszont a csere folytán új értéket kapó gyermekre már nem biztos, hogy teljesül a kupac tulajdonság. Ezért az adott gyermekre vonatkozóan rekurzívan meghívjuk a kupacolás eljárást (ld. 14. sor).

8.1. Példa. Nézzük meg, hogy a 8.8a. ábrán látható példa esetén miként tudjuk a kupac tulajdonságot fenntartani a 2-es indexű elem esetén. Látható, hogy a bal- és jobboldali gyermekekből mint gyökérből kiinduló részfák már eleve teljesítik a kupac tulajdonságot, viszont a 2-es indexű elemre már nem teljesül a kupacokkal szemben választott követelmény.

Meghívjuk a KUPACOL eljárást olyan módon, hogy a bemenetek közül n és k megegyezik, i értéke pedig 2. Az eljárás kiválasztja a 2-es, 4-es és 5-ös indexű elemek közül a legnagyobbat, ami most az 5-ös indexű. A második és ötödik elem kicserélésre kerül. Így a 2-es indexű helyre kerül a legnagyobb elem, illetve a baloldali részfa már biztos rendben van (ld. 8.8b. ábra). Viszont a jobboldali részfában a csere miatt elromolhatott a kupac tulajdonság teljesülése, ezért rekurzív módon meghívásra kerül a KUPACOL eljárás az $i = 5$ értékkel.

A rekurzívan hívott eljárás kiválasztja az 5-ös, 10-es és 11-es indexű elemek közül a legnagyobbat. Ez a tizedik elem, amit megcserélünk az ötödik elemmel. Így az ötödik elem a helyére került és a jobboldali részfája már rendben van, viszont a csere miatt most a baloldali részfában sérülhetett a kupactulajdonság teljesülése (ld. 8.8c. ábra). Emiatt ismét rekurzív hívás következik az $i = 10$ paraméterrel.

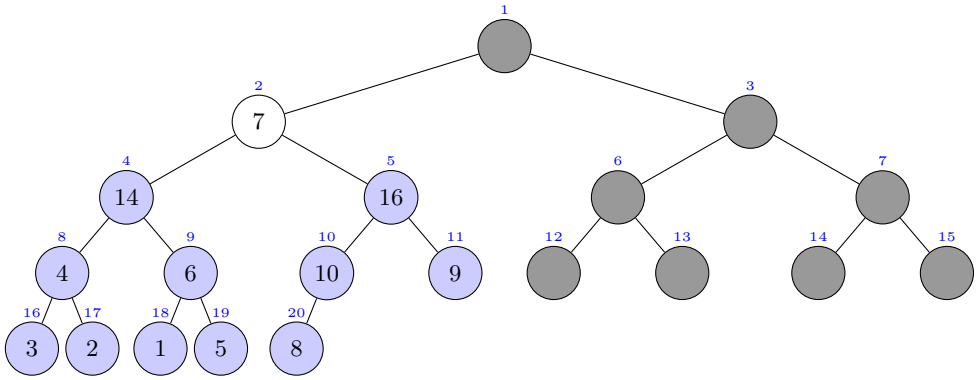
A 10-es indexű elemnek csak baloldali gyermeke van, ezért az eljárás a 10-es és 20-as indexű elemek közül választja ki a nagyobbikat. Mivel a huszadik elem nagyobb, ezért megtörténik a csere, aminek következtében a 10-es indexű helyen már biztos jó elem áll (ld. 8.8d. ábra). Viszont még egy rekurzív hívás történik, hiszen a 20-as indexű elemből mint gyökérből induló részfában elromolhatott a kupactulajdonság teljesülése.

Mivel a 20-as indexű elemnek nincsenek gyermekei, ezért az eljárás a 20-as indexű elemet fogja legnagyobbnak tekinteni, és nem történik csere, illetve újabb rekurzív hívás.

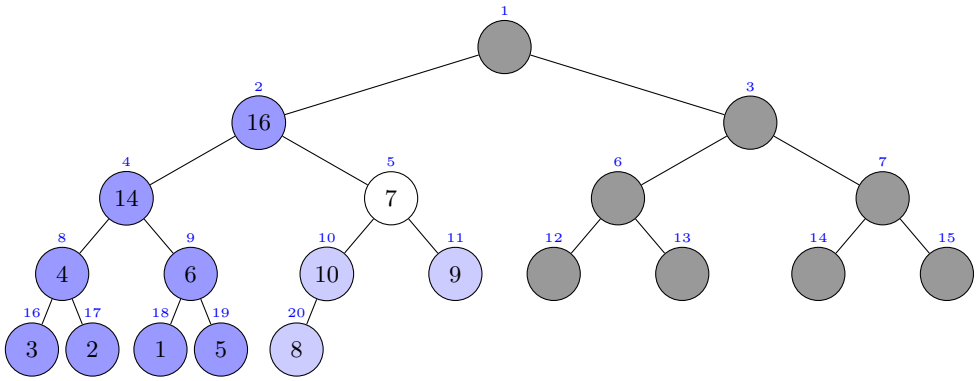
A kupacolás véget ért, a 2-es indexű elemből mint gyökérből induló részfára már teljesül a kupactulajdonság. ¶

Futási idő elemzése. Határozzuk meg, hogy mennyi a 8.1. algoritmusban részletezett KUPACOL eljárás futási ideje. A három vizsgált elem közül a legnagyobb kiválasztása $O(1)$ lépésben megvalósítható. Majd a csere, ha szükséges, csak három értékadást jelent. Ezért valójában azt kell megvizsgálni, hogy legfeljebb hány rekurzív hívás történik.

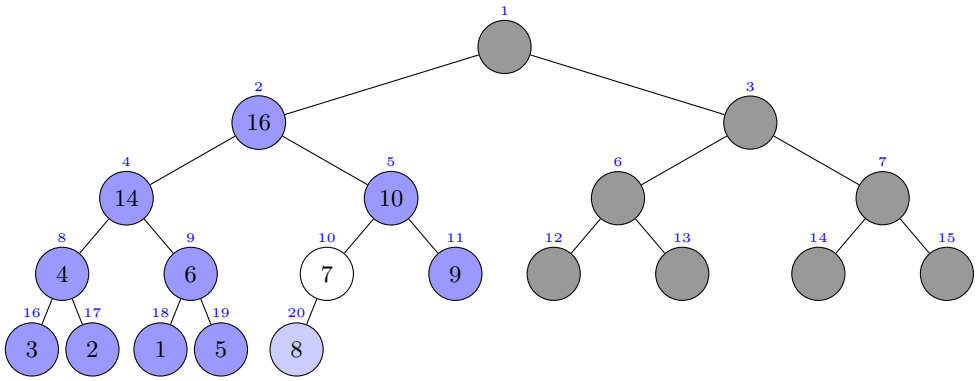
Mivel minden rekurzív hívásnál a bináris fa eggyel lejjebbi szintjére kerülünk, ezért a hívások száma legfeljebb a fa mélysége lehet. Erről viszont tudjuk, hogy n elemű fa esetén $\lceil \log_2(n+1) \rceil$. Így a kupacolás futási ideje: $T(n) = O(\log n)$. ♣



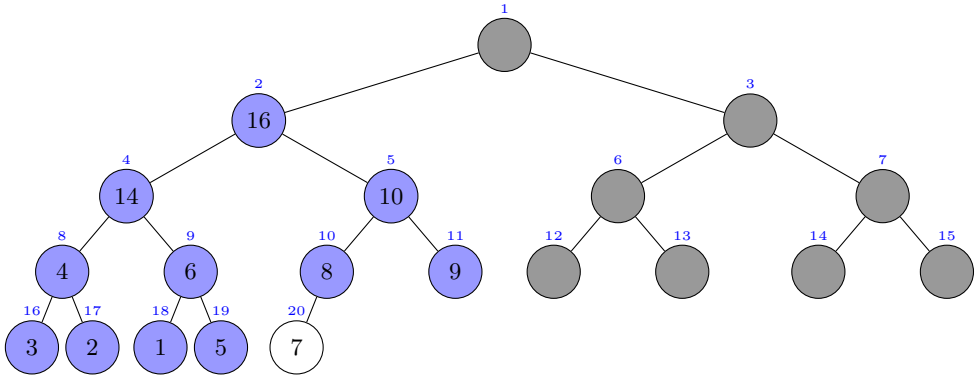
(a) A 2-es indexű elemre vonatkozóan kell kupacolni.



(b) A 2-es indexű elemre vonatkozóan kell kupacolni.



(c) Az 5-ös indexű elemre vonatkozóan kell kupacolni.



(d) A 10-es indexű elemre vonatkozóan kell kupacolni.

8.8. ábra. Kupacolás.

Kupac építése

A kupacrendezés megvalósítása felé vezető úton következő lépésünk annak meghatározása, hogy egy tetszőleges tömböt miként lehet kupaccá átalakítani. Ennek érdekében azt fogjuk megvizsgálni, hogy a tömb mely elemeiből mint gyökérből kiinduló részfákra teljesül már eleve a kupactulajdonság, illetve a többi elemre milyen módszerrel biztosítható az elvár tulajdonság teljesülése.

Tudjuk, hogy minden tömbnek van egyértelmű majdnem teljes balról kitöltött bináris fa reprezentációja. Könnyen látható, hogy egy bináris fa levél elemeiből mint gyökérből kiinduló részfákra eleve teljesül a kupactulajdonság, hiszen a nemlétező gyermek elemeknél nem kisebb a bennük tárolt elemek értéke. Azt is tudjuk, hogy egy majdnem teljes balról kitöltött bináris fa minden $\lfloor n/2 \rfloor + 1$ és n közötti indexű eleme levél. Így az x tömb $\lfloor n/2 \rfloor + 1$-edik és n-edik eleme közötti elemekre már kezdetben is teljesül a kupactulajdonság.

Hogyan lehetne a többi, nemlevél elem esetén elérni, hogy teljesüljön a belőlük mint gyökérből induló részfára a kupactulajdonság. Amennyiben az alattuk lévő elemek már eleve kupacok, akkor ezt kupacolással (ld. 8.1. algoritmus) el tudjuk érni. Ennek érdekében a nemlevél elemeket a tömb vége felől előre haladva fogjuk bejárni. Akkor ugyanis először a tömb $\lfloor n/2 \rfloor$-edik elemével foglalkozunk. Ennek minden gyermeke levél, tehát kupactulajdonságú. Kupacolás után már az $\lfloor n/2 \rfloor$-edik elemből mint gyökérből induló részfára is teljesül a kupac tulajdonság. Ahogy lépünk előre mindig olyan elemhez jutunk, melynek gyermekei már kupacok, hiszen nagyobb indexűek. Így viszont az egész tömb is kupaccá tehető.

A konkrét megvalósítást a 8.2. algoritmusban mutatjuk be. Az algoritmus bemenete egy n elemű x tömb, kimenete pedig a kupaccá alakított tömb.

Az algoritmus 2. sorában kezdődő ciklussal végigjárunk a nemlevél elemeken. A ciklusban csak annyit kell tennünk, hogy az aktuális elemre meghívjuk a KUPACOL eljárást úgy, hogy a kupacméretet és a tömbméretet azonosnak tekintjük (ld. 3. sor).

8.2. Algoritmus Kupac építése

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: x – **T kupac**

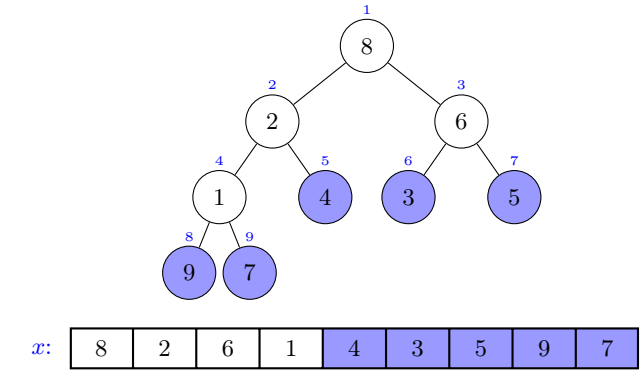
```
1: eljárás KupacotÉpít(címszerint x : T tömb, n : egész)
2:     ciklus i ← ⌊n/2⌋-től 1-ig
3:         Kupacol(x, n, n, i)
4:     ciklus vége
5: eljárás vége
```

Felhasznált változók és függvények

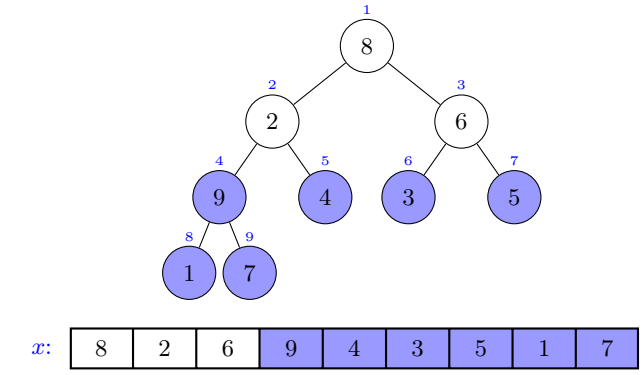
- x: A feldolgozandó tömb, amelyből kupacot építünk.
- n: Az x tömb elemeinek száma.

8.2. Példa. Alkalmazzuk a kupacépítés 8.2. algoritmusát a **??**. ábrán adott tömbre. Kezdetben az utolsó öt elemre már teljesül a kupactulajdonság (ld. 8.9a. ábra). Elindul az algoritmus 2. sorában kezdődő ciklus, i kezdeti értéke négy lesz. A kupacolás hatására a negyedik és nyolcadik elem cseréjét követően már teljesül a negyedik elemre a kupactulajdonság (ld. 8.9b. ábra). A ciklusban továbblépünk a harmadik elemre (ld. 8.9c. ábra). Erre az elemre már eleve teljesül a vizsgált tulajdonság, így léphetünk a második elemre. A kettes indexű elem esetén két cserével érhető el, hogy a a kupactulajdonság fennálljon (ld. 8.9d. ábra). A ciklusban végül az első elemre is meghívjuk a KUPACOL eljárást, aminek eredményeként egy cserét követően az egész tömb kupaccá válik (ld. 8.9e. ábra). ¶

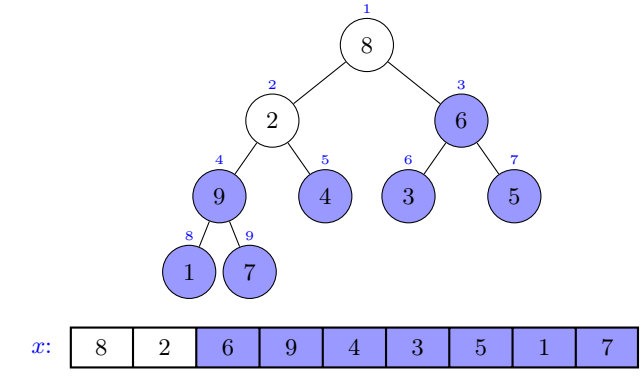
Futási idő elemzése. A KUPACOTÉPÍT eljárásban a ciklus pontosan $\lfloor n/2 \rfloor$-szer fut le. A cikluson belül a KUPACOL eljárást hívjuk meg, melynek futási ideje $O(\log n)$-es, így a kupacépítés futási ideje: $T(n) = O(n \log n)$-es. ♣



(a) A kezdeti tömb és annak majdnem teljes balról kitöltött bináris fa reprezentációja.

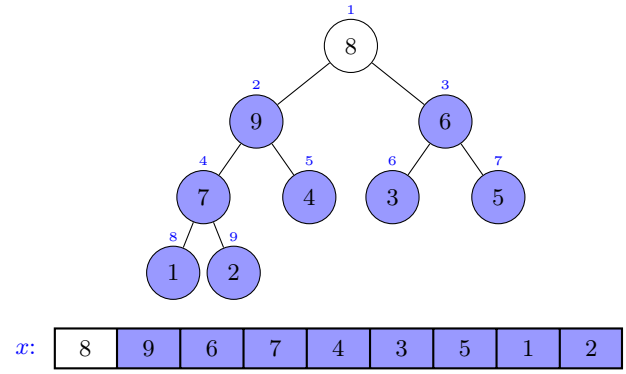


(b) A negyedik elemre kupacolunk. Egy cserét követően teljesül a kupactulajdonság.

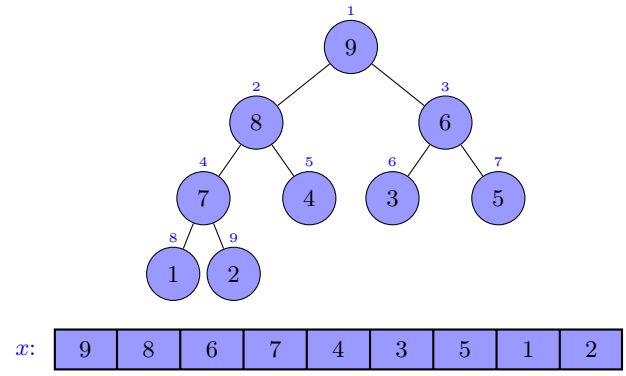


(c) A harmadik elemre kupacolunk. Most nincs szükség cserére.

8.9. ábra. Kupacépítés.



(d) A második elemre kupacolunk. Most két cserét kell elvégeznünk.



(e) Az első elemre kupacolunk. Egy cserét követően az egész tömb kupaccá válik.

8.9. ábra. Kupacépítés (folyt.).

Kupac rendezése

Elérkeztünk oda, hogy végre képesek vagyunk kupacok segítségével rendezni egy tömböt. Tudjuk, hogy egy kupacban az első elem mindig a legnagyobb, hiszen csak így teljesülhet rá a kupactulajdonság. Egy növekvően rendezett tömbben viszont a tömb végére szeretnénk vinni a legnagyobb elemet. Ennek érdekében cseréljük meg a tömb első és utolsó (n-edik) elemét. Így a legnagyobb elem biztos a helyére kerül és többé már nem is kell vele foglalkozni.

Emiatt a továbbiakban tekintsük a tömbnek csak az első $n-1$ darab elemét. Ez majdnem kupacnak tekinthető, csak az első elemből mint gyökérből induló részfára nem teljesül biztosan a kupactulajdonság, hiszen a korábbi cserével ezt elronthattuk. Hívjuk meg ezért az első elemre a KUPACOL eljárást. Figyeljünk arra, hogy a tömb mérete ekkor n, de a kupacként vizsgált része már csak $n-1$ elemű. (Emiatt van arra szükség, hogy a KUPACOL eljárásban külön adjuk meg a tömb és a kupac méretét.) Az $n-1$ elemű kupac legnagyobb eleme biztos az első helyre kerül. Cseréljük meg az első és az $(n-1)$-edik elemet, majd a továbbiakban csak az első $n-2$ darab elemmel foglalkozzunk.

Iteratívan vigyük mindig az aktuálisan vizsgált kupac első elemét hátra, majd csökkentsük a kupac méretét eggyel és kupacoljunk ismét az első elemre. Tegyük ezt mindaddig, amíg a második elem is a helyére nem kerül. Ekkor már biztos, hogy az első elem is a helyén lesz, így a tömbünk rendezetté válik.

A 8.3. algoritmusban írjuk le a rendezést megvalósító eljárást. Bemenetként az x tömböt és annak n elemszámát adjuk meg, kimenetként pedig a rendezett tömböt kapjuk vissza.

Az algoritmus 2. sorában a 8.2. algoritmusban ismertetett kupacépítést hívjuk meg. Miután már van kupacunk egy ciklussal bejárjuk a kupacot az n-edik elemtől a második elemig (ld. 3. sor). A cikluson belül megcseréljük az első és az i-edik elemet (ld. 4. sor), majd kupacolunk az első elemre úgy, hogy kupacnak csak a tömb első $i-1$ darab elemét tekintjük (ld. 5. sor). A ciklusból kilépve rendezett tömböt kapunk vissza.

8.3. Algoritmus Kupacrendezés

Bemenet: x – **T tömb**, n – **egész** (*tömb mérete*); ahol **T** összehasonlítható
Kimenet: x – **T rendezett tömb**

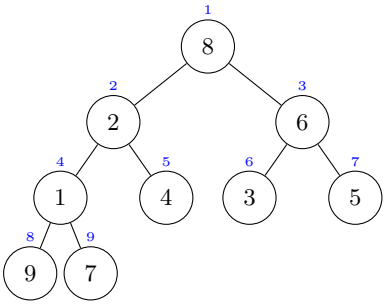
```
1: eljárás KUPACRENDEZES(címszerint x : T tömb, n : egész)
2:     KUPACOTÉPÍT(x, n)
3:     ciklus i ← n-től 2-ig
4:         x[1] ↔ x[i]
5:         KUPACOL(x, n, i − 1, 1)
6:     ciklus vége
7: eljárás vége
```

Felhasznált változók és függvények

- x: A feldolgozandó tömb, amelyet rendezünk.
- n: Az x tömb elemeinek száma.

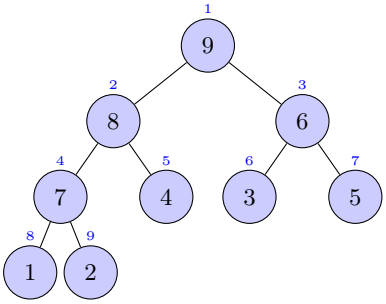
8.3. Példa. A 8.10. ábrán végig követhetjük, hogy miként rendezi a kupacrendezés 8.3. algoritmusa az adott x tömböt. Az eredeti tömböt a 8.10a. ábrán adtuk meg. Először felépítjük a kupacot, aminek eredményét a 8.10b. ábrán láthatjuk. Ezt követően megcseréljük az első és a kilencedik elemet, majd az első nyolc elemből képzett résztömböt az első elemre történő kupacolással visszaalakítjuk kupaccá (ld. 8.10c. ábra). Az új kupac első elemét a nyolcadik helyre mozgatjuk, majd ismét kupacolunk az első elemre, miközben már csak az első hét elemet vesszük figyelembe (ld. 8.10d. ábra). Hasonló logikát követve járunk el, ahogy a 8.10e-8.10j. ábrákon látható. Az eljárás végén a tömbünk rendezetté válik. ¶

Futási idő elemzése. Vizsgáljuk meg a kupacrendezés 8.3. algoritmusának futási idejét. Először meghívjuk a KUPACOTÉPÍT(e)ljárást, aminek futási ideje $O(n \log n)$-es. Ezt követően $(n-1)$-szer fut le a ciklus, melyben egy cserét hajtunk végre és meghívjuk az $O(n \log n)$ futási idejű KUPACOL eljárást. Ezek alapján látható, hogy a kupacrendezés futási ideje: $T(n) = O(n \log n)$-es. ♣



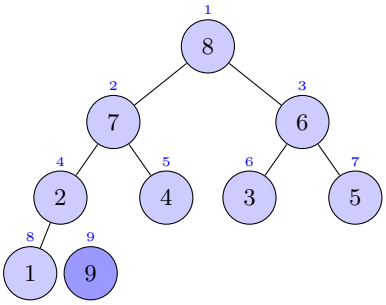
x:	8	2	6	1	4	3	5	9	7

(a) A kezdeti tömb és annak majdnem teljes balról kitöltött bináris fa reprezentációja.



x:	9	8	6	7	4	3	5	1	2

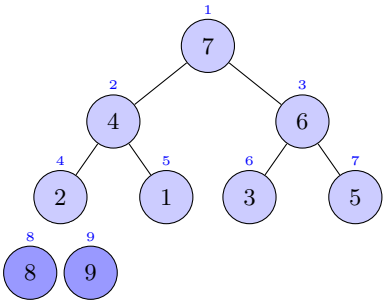
(b) Az eredeti tömbből felépített kupac.



x:	8	7	6	2	4	3	5	1	9

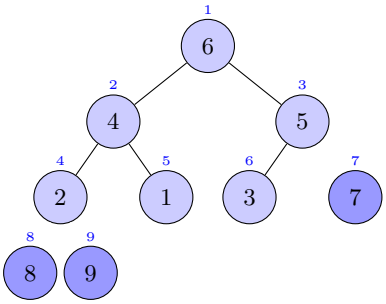
(c) A legnagyobb elem a helyére került. Az első nyolc elemet az első elemből indulva kupacoljuk.

8.10. ábra. Kupacrendezés.



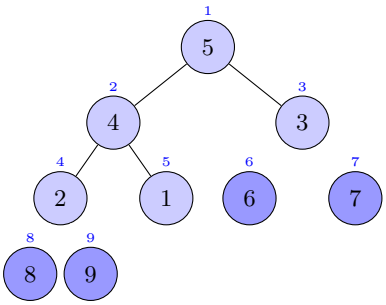
x:	7	4	6	2	1	3	5	8	9

(d) A nyolcadik elem a helyére került. Az első hét elemet kupacoljuk az első elemből indulva.



x:	6	4	5	2	1	3	7	8	9

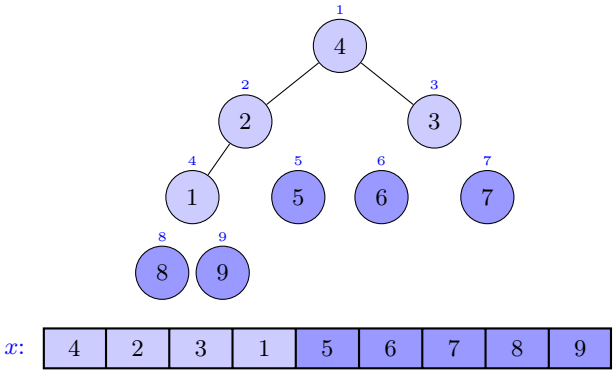
(e) A hetedik elem a helyére került. Az első hat elemet kupacoljuk az első elemből indulva.



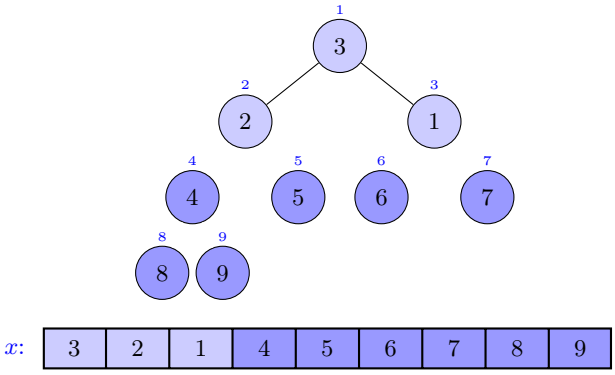
x:	5	4	3	2	1	6	7	8	9

(f) A hatodik elem a helyére került. Az első öt elemet kupacoljuk az első elemből indulva.

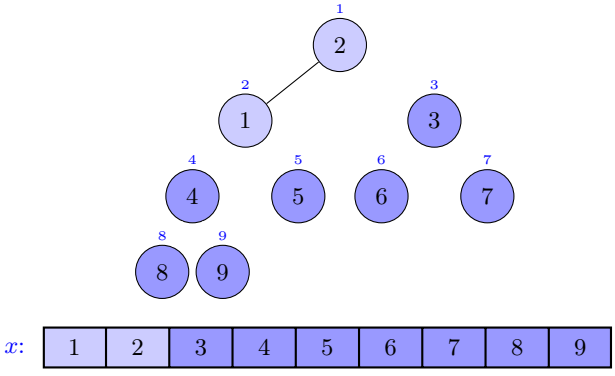
8.10. ábra. Kupacrendezés (folyt.).



(g) Az ötödik elem a helyére került. Az első négy elemet kupacoljuk az első elemből indulva.

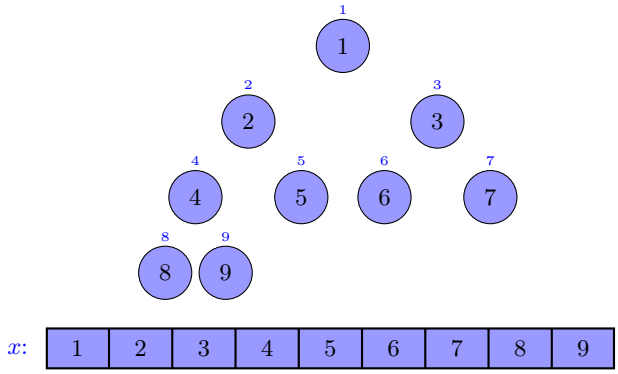


(h) A negyedik elem a helyére került. Az első három elemet kupacoljuk az első elemből indulva.



(i) A harmadik elem a helyére került. Az első két elemet kupacoljuk az első elemből indulva.

8.10. ábra. Kupacrendezés (folyt.).



(j) A második elem a helyére került. Így az első elem is biztos a helyén van, tehát a tömb rendezett lett.

8.10. ábra. Kupacrendezés (folyt.).

Magyar-angol szótár

0-1 hátizsák feladat	0-1 knapsack problem
Algoritmus	Algorithm
Bemenet	Input
Beillesztéses rendezés	Insertion sort
Bináris fa	Binary tree
Bináris keresés	Binary search
Buborékrendezés	Bubble sort
Dinamikus programozás	Dynamic programming
Eldöntés	Decision
Feljegyzéses módszer	Memoization
Gyorsrendezés	Quicksort
Kimenet	Output
Kiválasztás	Selection
Kiválasztó rendezés	Selection sort
Kupac	Heap
Kupacrendezés	Heapsort
Leghosszabb közös részsorozat	Longest common subsequence
Lineáris keresés	Linear search
Logaritmikus keresés	Logarithmic search
Lokális változó	Local variable
Másolás	Copy
Metszet	Intersection
Mohó algoritmus	Greedy algorithm
Nyers erő	Brute force
Oszd meg és uralkodj	Divide and conquer
Összefésülő rendezés	Merge sort
Összegzés	Summation
Támpont elem	Pivot element

Típus	Type
Unió	Union
Változó	Variable

Angol-magyar szótár

0-1 knapsack problem	0-1 hátizsák feladat
Algorithm	Algoritmus
Binary search	Bináris keresés
Binary tree	Bináris fa
Brute force	Nyers erő
Bubble sort	Buborékrendezés
Copy	Másolás
Decision	Eldöntés
Divide and conquer	Oszd meg és uralkodj
Dynamic programming	Dinamikus programozás
Greedy algorithm	Mohó algoritmus
Heap	Kupac
Heapsort	Kupacrendezés
Input	Bemenet
Insertion sort	Beillesztéses rendezés
Intersection	Metszet
Linear search	Lineáris keresés
Local variable	Lokális változó
Logarithmic search	Logaritmikus keresés
Longest common subsequence	Leghosszabb közös részsorozat
Memoization	Feljegyzéses módszer
Merge sort	Összefésülő rendezés
Output	Kimenet
Pivot element	Támpont elem
Quicksort	Gyorsrendezés
Selection	Kiválasztás
Selection sort	Kiválasztó rendezés

Summation	Összegzés
Type	Típus
Union	Unió
Variable	Változó