Az informatika alapjai

Előadást egyáltalán nem követő, csak a legfontosabb (szükséges de nem elégséges) dolgokat, némi fogalmi alapokat (összezavarás céljából), feladatokat és példa feladatsort tartalmazó jegyzet.

*Ha valamilyen hibát találsz a jegyzetben, kérlek, írd meg a **balti@fw.hu** címre!*
*A jegyzet megtalálható a **http://www.tar.hu/jegyzetek/** weblapon.*

TARTALOMJEGYZÉK

Bevezetés, avagy mi is az a számítógép

Ebben a fejezetben a számítógép fogalmát szeretném körüljárni. Tulajdonképpen ebben a fejezetben tárgyalt dolog nem tartozik szorosan a vizsgaanyaghoz, és senki sem fogja az itt leírtakat számon kérni. Csak egy kis bevezetés szeretne lenni a számítástechnikába…

Akkor induljunk el a kályhától, illetve esetünkben a kínai evőpálcikától! A kínai evőpálcika nemcsak, hogy pálcika alakú, hanem a vége le van csapva. Ily módon teljesen alkalmatlan az étel felszúrására. A japán evőpálcika vége már ki van hegyezve, akár szúrásra is alkalmas. (Igaz, nem tudom, hogy ténylegesen használják-e így a Japánok.) Az európai villát pedig alapból az étel felszúrására tervezték. Szóval itt van ez a három evőeszköz, és annyira különbözőek! A kínai evőpálcika kitalálójának eszébe sem jutott az étel felnyársalása. Számára az étel szent dolog lehetett, amit nem szurkál az ember össze-vissza. Az európai ember praktikus. A villa kitalálójának lelkét az tölthette el, hogy minél biztosabban a szájához emelhesse a falatot. A japán evőpálcikát nem nevezném átmenetnek, inkább öszvérnek. Minden esetre, azzal, hogy kihegyezték a végét, megteremtették annak a lehetőségét, hogy az evőeszközzel szúrni is lehessen. A nyugati ember számára mindkét evőpálcika egykutya, de mint látható ez nem így van. A három eszközben ily módon megjelenik a kitaláló érzelemvilága is. A nyugati ember sosem használta volna a pálcikát evőeszköznek, számára ez túl „bizonytalan” módszer; és a kínai emberünk sem véletlenül választotta a pálcikát evőeszközéül. (Még mielőtt tovalapoznál, ez valóban egy Info. alap jegyzet! ☺)

Az az általánosan elfogadott dolog, hogy bármit és bárhogy gondolhatunk - azaz a gondolatvilágunk teljesen szabad - az a föntiek fényében egy kicsit másképpen látszik! Úgy tűnik, hogy az érzésvilágunk szorosan befolyásolja azt, hogy mit és hogyan fogunk föl, és hogyan gondolunk rá. Több millió európainak több száz éven át eszébe sem jutott, hogy lehet enni pálcikával is. Lehet egy evőpálcikát vagy egy golyóstollat - nem „rendeltetésszerűen” - pl. fülpiszkálásra használni. De ekkor mindkét tárgyat csak egyszerű „fadarabként használjuk”. Ha úgy tetszik - praktikus módon - „lefokozzuk” az eszközünket.

A tárgy funkcióját a forma hordozza. Egy lúdtoll is csak akkor válik alkalmassá írásra, ha megfelelő átalakításokat végzünk rajta. Egy golyóstoll pedig minden ízében az írásra való alkalmasságot hordozza magában. Hisz’ erre tervezték. Ha egy origami mester hajtogat számunkra egy apró papírszéket, akkor nem gondolunk arra, hogy az nem is szék, hogy széknek egyáltalán nem használható.

Tárgy - Funkció - Érzésvilág hármassága valamiféle kapcsolatot alkot. De vajon ez a hármasság, hogyan jelenik meg a számítástechnikában? Mire és hogyan alkalmas egy számítógép?

Azt már az elején megállapíthatjuk, hogy a forma mára már főleg mikroszkopikus méreteket öltött, a szilíciumlapka belsejét szabad szemmel nem tudjuk megnézni. Az elektronikus áramköröket - leegyszerűsítve - megannyi kapcsolók vagy fogaskerekek sokaságaként foghatjuk föl. Korunk számítógépeinek többsége a Neumann-elvek alapján épülnek föl, ezért érdemes ezeket áttekintenünk:

1. A kettes számrendszer használata az adatok tárolásához.
2. Soros működés, azaz egyszerre csak egy műveletet végez el, majd a következőt.
3. Belső memória használata.
4. A belső memóriában tárolódnak az adatok és az utasítások is.
5. Univerzális feladatmegoldó eszköz.

Azzal, hogy az adatok ábrázolására a kettes számrendszert használjuk, a számítógép kétállapotú szerkezetekkel könnyen megvalósítható lett. Elektronikus eszközök használatával, az adatokkal való műveletvégzés leegyszerűsödött. A soros működés magától értetődő. Adva van egy műveletsor, akkor azt műveletenként sorban kell végrehajtani, és többségében a soron következő művelet végrehajtása függ az előző műveletektől, eredményeiktől. Az egyszerre több műveletet elvégző gépek több feladat-végrehajtó egységet tartalmaznak. A belső memória az a „színpad”, ahol a műveletek végrehajtódnak. A tárolt program elve azt jelenti, hogy az utasításokat is ugyanolyan formában ábrázoljuk, mint az adatokat, a tárolás módjában nem téve különbséget közöttük. Ehhez kapcsolódik az univerzális feladatmegoldó eszköz elve, az utasításokkal az adatokon különböző műveleteket végezhetünk. Az utasítások alapműveletek, még a CISC[1] processzoroknál is. Akár a Lego elemek, mindegyiknek meghatározott funkciója van, de ezek alapfunkciók. Úgy kapcsolod egymáshoz őket, azt „építesz” belőlük, amit akarsz.

A számítógép egy olyan automata, amivel adatokat ábrázolunk, és különböző műveleteket végezünk velük. A számítógéppel foglalkozó mérnök és programozó minden dolgot apró részletekre szed, hogy megvalósítható legyen a számítógépen. Gondoljunk csak a képernyőre! A pontokból álló terület az én fejemben áll össze képpé. Az elektronikus-automata nem tudja, hogy én most szöveget szerkesztek, avagy éppen egy FPS[2] játékkal játszok. A megjelenített adatoknak én, az ember, adok értelmet. A gép nem „tudja”, hogy mit is végez valójában, sem azt, hogy mivel. Pont úgy, ahogy a régi mechanikus pénztárgépek. (Talán van még olyan, aki látott ilyet.)

Egy fogaskerekekkel teli gépnél senki nem gondolja azt, hogy tudata lenne. Míg annál az eszköznél, amelynél a fogaskerekeket elektromos áramkörök helyettesítik valamiért többen - még néhány programozó és informatikus is -

[1] *Complex Instructions Set Computer ~ összetett utasításkészletű számítógép.*
[2] *First Person Shooter ~ saját nézőpont típusú játék.*

úgy vélik, hogy egyszer csak, talán a nem is oly távoli jövőben, tudatára tud/fog majd ébredni. Gondolkodni, érezni, mi több, akarni is fog!

Talán van, aki azt mondja magában, hogy ez minden bizonnyal így is lesz, és remek eredményeket értünk el ezen a területen. „A számítógép már legyőzte a sakkvilágbajnokot! Kell nekünk ennél több bizonyíték?" De míg Garri Kaszparov gondolkodással jut el egy-egy lépés megtételéig, addig a gép csak a program által meghatározott műveleteket fogja végrehajtani. Az adott sakk-számítógépnél az addig lejátszott sakknagymester játszmákból létrehoztak egy hatalmas sakkjátszma-adatbázist (ami az igazi alapját képezi az adott sakk-számítógépnek), és egy programozó jó programot készített hozzá. Láthatjuk, hogy a gép lenyűgöző teljesítménye mögött nem a gépet, hanem az embert találjuk.

A másik példa lehet a ma oly' divatos mobiltelefonoknál előforduló hanghívás funkció. De a telefon nem tudja, hogy én az anyósomat vagy éppen az apósomat hívom. Egyszerűen csak két hanghullám egyes tulajdonságainak értékeit hasonlítja össze. Valójában a telefon ezt sem tudja, csak számokat hasonlít össze számokkal. Ha igazán szigorúak vagyunk, akkor ezt sem mondhatjuk. A számítógépet megtervező mérnök és a programozó tudja, hogy ha ez meg ez az utasítás fog végrehajtódni, akkor ez a művelet valamiféleképpen összehasonlítást fog jelenteni. A programozónak kell tudnia azt is, hogy a számítógépen ábrázolt adott értékek az egyes hanghullámok tulajdonságait jelentik és nem valami mást.

Minden olyan emberi viselkedés modellezhető, amit a gép szintjének megfelelő részletességgel ábrázolható. Már a kezdetek kezdetén, a számítógép megjelenésekor néhány emberi képességet pontosságban, gyorsaságban túlszárnyalta a gép. Mivel a számítógépet adatok ábrázolására és ezekkel való műveletvégzésre találták ki ezért - nem meglepő módon - ezen a területen jelentkeztek az első eredmények. Minél több dolgot tudunk majd modellezni, és minél gyorsabb gépek állnak ennek a szolgálatában, annál több területen tűnhet úgy, hogy a gép „jobb", „okosabb", „műveltebb", mint mi. Semmi gondom a mesterséges intelligenciával, de csak akkor, ha így mondják ki, és így is gondolnak rá: MESTERSÉGES *intelligencia*.

A gép - legyen az akár a még oly bonyolult/egyszerű számítógép - nem fogja, mert nem tudja, túlnőni az embert. Akármennyire is próbálja néhány film, könyv, stb. ezt sulykolni, ezzel akár a gyakorlott programozót is megzavarni, megtéveszteni. Ha sikerül elrejteni ezt a marhaságot az emberekben, akkor esetleg teljesül a jóslat, és a „számítógép-mumus" árnya rátelepszik az emberekre. Talán külön életet is élhet, és tovább riogatja az embereket, ezzel teljesítve be a várakozásokat.

Felmerülhet valakiben, hogy mi van akkor, ha nem a Neumanni-elvek alapján építjük föl a gépet, esetleg neuron alapú számítógépet hozunk létre vagy valami más csodamasinát? A alaphelyzet akkor se lenne más, mivel gépről van szó, ezért a funkcióját valahogy meg kell határoznunk, méghozzá pontosan kell meghatároznunk. Nem elég azt mondanunk, hogy: „Tanulj! Akarj! Érezz! Gondolkozz!". Bármilyen eszközt hozunk létre, pontosan meg kell határoznunk a működési módját. Ez a működés akármilyen bonyolult, akárhány visszacsatolást is tartalmaz, mégis egy lépésről-lépésre követhető lineáris folyamat.

Ha modellezzük az akaratot ettől a gép még nem képes akarni; ha modellezzük az emberi érzések megjelenését, ettől a gép még nem fog érezni; ha modellezzük az emberi gondolkodást, ettől a gép még nem fog gondolkodni; ha modellezzük egy leszűkített területen a tanulási képességünket, ettől a gép még nem képes tanulni. Ez továbbra sem lesz több, csak MESTERSÉGES *intelligencia*. Habár biztos, hogy a jövőben nagyon összetett dolgokat is elő fogunk állítani ezen a területen.

Azt már megbeszéltük, hogy a számítógép „formája" mára már mikroszkopikus méreteket öltött. „…A tranzisztor is makroszkopikusan viselkedő rendszer, mert elég nagyméretű, és a folyamatokban nagyon sok elektron vesz részt, úgyhogy a kvantumos hatások kiátlagolódnak. Azonban a tranzisztorok mérete nem csökkenthető a végtelenségig. Ezért a mikroelektronika ma még szédületes ütemű fejlődése rövidesen, pár éven belül kifullad, mert elérjük a tíz nanométer jellemezte elvi korlátot, ahol a kvantumos hatások már nem átlagolódnak ki." - Adja ennek egy jellemzését Végh László „*Egy új természetképről"* című jegyzetében (32. oldal).

De a számítógép túl általános működési elvet mutat. Adatok tárolása, és műveletvégzés az ábrázolt adatokon. Az adatok bármit jelenthetnek számunkra, és még az egyszerű monitoron is számtalanféleképpen jeleníthetjük meg azokat. A legtöbb fizikai eszközünktől abban különbözik a számítógép, hogy a funkció - ami, ha emlékszünk még a fejezet elején elmondottakra, általában a formában jelenik meg - két részre bomlik. A számítógép pontosabban körvonalazott funkcióját, igazi „formáját" a szoftver (software), a programozás határozza meg. Az adatábrázolás területén általánosan használható gép konkrét működését a programozó határozza meg. De ez a - számítástechnikai kifejezéssel élve - réteg még további két részre osztható. A számítógép, ahogy mondani szokták, nem tud semmit. Azt, hogy hogyan kezelje a perifériákat, a memóriát és egyéb eszközöket az ún. operációs rendszer valósítja meg, mivel ezek a dolgok sincsenek pontosan meghatározva. Ha már fut a gépen az adott operációs rendszer, akkor még mindig csak a gépet tudjuk üzemeltetni, és azt határoztuk meg, hogy ez hogyan történjen. Egy konkrét funkciót már egy felhasználói programnak kell megvalósítania. Meg kell még jegyezni, hogy szokás az operációs rendszerekből és a felhasználói programokból kiszakítani azt a csoportot, amik az operációs rendszer működtetését segítik elő. Ezeket a felhasználói programokat rendszerközeli programoknak hívják.

Láttuk, hogy a számítógép működését, a konkrét funkcióját nem a hardver (hardware) határozza meg, hanem a szoftver. Az, ami más tárgynál a formában nyilvánul meg, az a számítástechnikában a hardver és a szoftver együttesében. A számítástechnika fejlődése ezen a vonalon is továbbhalad, és a fejlődés fő iránya akár ez is lehet.

Az igazi korlátot nem a tíznanométeres határ jelenti, hanem az a módszer és mód, ahogy a gépet használjuk, és ez sokkal tágabb kör, mint ahogy az fönt megfogalmazódott.

A számítástechnika fejlődésének elején a fő irány valóban a *hardver* fejlesztése volt. Azt itt részletesen nem fogjuk tárgyalni, hogy hogyan jutottunk el a nem túl bonyolult, de annál nagyobb helyet elfoglaló gépektől az igen komoly, integrált rendszerekig.

A hardver bizonyos fejlettsége után beindult a *szoftver* rohamosabb fejlődése. Ahogy azt az előbb megtárgyaltuk, ez szükségszerű volt. Egy-egy részterületen egyre jobb, tökéletesebb programok jelentek meg. Ma már több szoftvert dobnak ki, mint amennyit használnak. Igen bonyolult hardverek és szoftverek jelentek meg, és nem is olyan rég a számítástechnika összetettségének növekedésével és tömeges elterjedésével egy „újabb" elem is megjelent a számítástechnikában.

Ez az ismerete, az *információ*. Az, hogy az előző két dolgot hogyan, mi módon lehet használni. A számítástechnika fejlődése egy új szakaszba lépet, a harmadik irány az információ. Az elkövetkező években minden bizonnyal sokkal nagyobb hangsúlyt fog kapni a számítástechnika oktatásának területe is. Nem csak a közoktatásra kell gondolnunk. A bonyolultabb rendszerek - programozási nyelvek, operációs rendszerek, adatbázis kezelők, stb. - megjelenésével a szakembernek is bővebb információra van szüksége arra nézve, hogy az adott rendszer, hogyan működik. Az internet segítségével több százezer kisgépet használva - az adott feladatot szétosztva - „virtuális szuperszámítógépet" hozhatunk létre, ami akár a mai nagyszámítógépek kapacitásának sokszorosát is meghaladhatja, és mindezt anélkül, hogy bármiféle nagyobb összeget fektettünk volna a dologba, csupán az együttműködés lehetőségét használva. Természetesen ez az elem ezelőtt is jelen volt, csak nem volt értelme piacosítani, ezért kevésbé tűnhetett föl.

Hardver - Szoftver - Információ, mint fejlődési irányok, nem választhatók külön egymástól. A hardver fejlődése ma is tart, és ki mondhatná azt, hogy tökéletes programot készített. Bárhová is fejlődik a számítástechnika, hacsak „ki nem növi" magát, az általános meghatározása meg fog maradni. Az, hogy adatokat tárolunk, és műveleteket végzünk a számítógép segítségével, a világ egy-egy szeletét képezzük le.

A műveletvégzésbe természetesen bele kell érteni az adatok bevitelét és megjelenítését is. Ezek nélkül nem sok értelme lenne a megalkotott gépnek. A beviteli, megjelenítő és adattároló eszközöket összefoglaló néven perifériáknak hívják. Itt is az érvényesül, hogy nincs meghatározott eszköz, ami ezeket szolgálná, az adatok bevitelét és megjelenítését bármilyen módon megoldhatjuk, csak arra vannak megszorítások, hogy mi módon csatlakoztathatjuk az eszközt a számítógépre, és az adatok átvitele ennek megfelelően történhet. A periféria és a számítógép közötti adatforgalom pontos meghatározását a programozás valósítja meg. Egy adott portra bármilyen ennek megfelelő eszköz illeszthető. Azt, hogy a porton megjelenő adatok mit is jelentenek az adott periféria meghajtó szoftvere, driver-e határozza meg. Egyeseket megtéveszt ez a sokszínűség, és úgy vélik, hogy a be/kiviteli eszközök nem tartoznak szorosan a számítógéphez, és úgy vélik, hogy a periféria nélküli számítógép is tekinthető számítógépnek, tökéletesen működő számítógépnek. Ez a gondolatmenet olyan mintha azt mondanánk, hogy a hangszóró és mikrofon nélküli telefon is tökéletesen működik, hisz' hívásokat képes fogadni. Ki/beviteli eszköz nélküli számítógép számunkra valójában egy fekete doboz lenne. Egyáltalán nem tudnánk, hogy mit csinál, tulajdonképpen lehet, hogy nem is csinálna semmit, mivel adatokat sem vittünk be. Gyakorlatilag egy fél téglának megfelelő funkciót tudna betölteni. Ahhoz, hogy ténylegesen számítógépről beszéljünk, legalább egy beviteli és egy megjelenítő eszközt kell hozzákapcsolnunk. És ekkor még az adatok huzamosabb tárolását nem is oldottuk meg! Még szigorúan véve - a gép oldaláról szemlélve a dolgot - sem mondhatjuk azt, hogy a számítógéphez nincs szükség perifériákra. Ha ezt tennénk, akkor gondolatilag kitakarnánk egy lényeges elemet a rendszerből: magát az embert. Az ember szemszögéből csak a megfelelő ki/beviteli eszközökkel ellátott rendszer alkot számítógépet.

Láthattuk, hogy a számítógép mit is takar, és a programozás többet jelent, mint ahogy azt általában gondolják. Egy-egy apró kis megjelenítési mód is komoly problémákat okozhat a számítógép használatakor. Az operációs rendszerek „elfajulása" szerintem a grafikus felület megjelenése után kicsivel történt. Akkor, amikor valaki kitalálta a ma oly' gyakori nyomógomb használatát. Állandó illúziót keltve a felhasználóban, hiszen a képernyőn nincsen nyomógomb! Nem a grafikus felülettel van a baj, hanem azzal, hogy teljesen feleslegesen illúziókeltésre használják. Nem csak a felhasználót zavarják meg, hanem a még oly komoly programozó is azon töri a fejét, hogy egy-egy gombnak vagy más elemnek így vagy amúgy kellene kinéznie, és milyen színekben játsszon. Ettől aztán néha nagyobb hangsúlyt kap a külcsín, mint a belbecs. Az internet szinte melegágya a semmire nem való, de igen „tetszetős" dolgoknak, bár ma már egyre több a tartalmas hely is. A programozó legalább törekedjen arra, hogy ne szemetet adjon ki a kezéből, hanem jobb és jobban használható programot készítsen.

Minden bizonnyal van, aki nem ért egyet az itt leírtakkal, és páran akár vitába is szállnának velem. Lehet, hogy nekik van igazuk, és sületlenség, amit összehordtam. Nem szükséges követni ezt a sajátságos fölfogást, de érdemes lehetett megismerni, és érdemes lehet továbbgondolni! Ezért kérlek, ne sajnáld azt a pár forintot, hogy ezt a néhány oldalt is lemásold a jegyzetből! Tulajdonképpen azért kezdtem bele ennek a jegyzetnek a megírásába, hogy ezt a bevezetőt megírhassam.

A memória

A számítástechnikában az adatok leképezése kétállapotú elemekkel történik. Egy ilyen kétállapotú elemet neveznek **bit**nek (binary digit ~ bináris számjegy). Az értéke egyértelmű, csak nulla vagy egyes lehet.

Már a számítástechnika fejlődésének elején a biteket csoportokban kezelték. Ez az alapegység a **bájt** (byte) lett, ami alatt nyolc bitből álló csoportot kell érteni. (Volt a nyolctól eltérő csoportosítás is.) Ez a legkisebb egység, ami címezhető, ennél fogva az adatok kezelésének is ez az alapegysége. A csoportosítás alapját az adja, hogy nyolc biten 256 különböző állapot ábrázolható. Ez elégnek bizonyult a különböző processzorutasítások, a különböző karakterek, számok, stb. tárolására. Ott, ahol nem elegendő egy bájt az adat tárolására, több bájtot használnak föl. Például egy lebegőpontos számot tárolhatunk négy bájton, 32 biten. A bájt fogalmához nem tartozik hozzá a különböző segédbitek, ellenőrző bitek.

Az adatok kezelésének alapegysége a bájt lett. Később a processzorok egyszerre több - kettő, négy, nyolc - bájtot is kezeltek, megjelentek a 16, 32, 64 bites processzorok. Ezeket az egységeket **szó**nak (word) nevezik. Egy szó azt az alapegységet jelenti, amit a processzor egyszerre tud kezelni. Nyolcbites processzornál egy szó egy bájt, tizenhat bites processzornál egy szó két bájtnak felel meg. Ezek alapján beszélnek még fél szóról és dupla szóról. Értelemszerűen a szó méretének felét illetve kétszeresét jelentik.

A bájtok csoportosításánál használatosak a **kilo**, **mega**, **giga**, **tera** SI prefixumok is. Azzal az eltéréssel, hogy a kettes számrendszer használata miatt a kilo - 2^{10}, a mega - 2^{20}, a giga - 2^{30}, a tera - 2^{40} bájtnak felel meg. Így 2 GB = $2*2^{10}$ MB = $2*2^{10}*2^{10}$ KB = 2*1024*1024*1024 bájt = 2147483648 bájt. (A winchester gyártók a merevlemez kapacitásának megadásakor a tízes alapot (kilo = 10^3) használják. Ezért a merevlemez kapacitására nagyobb érték jön ki ahhoz képest, mint amit az tárolni képes: 2 GB ≈ 2.1 G.)

A **memória** az a belső színpad, ahol a processzor dolgozik. De akkor, hogyan indul el a gép, hogyan indul el az önteszt? Az addig rendben van, hogy ez is egy program, és be van égetve a ROM-ba. De honnan „tudja” a processzor, hogy ez a program hol helyezkedik el, hol kezdődik? Huzalozással oldották meg, hogy a számítógép bekapcsolásakor a processzor utasításregiszterébe a megfelelő memóriacím töltődik. Az öntesztet is a processzor hajtja végre, de a programot az alaplapon lévő ROM tartalmazza.

A memóriának van egy *fölfelé* és egy *lefele* való kiterjesztése is. A processzorhoz közelebb eső rész a **regiszter**. Ez olyannyira közel van a processzorhoz, hogy a részét képezi. Ezek gyors elérésű, kis tárterületek. Vannak közöttük speciálisak, mint az előbb említett utasítás regiszter, vagy az aritmetikai/logikai műveleteket segítő Akkumulátornak nevezett regiszter, és egyéb speciális regiszterek, és vannak általános célúak is. A processzoron belül is történik adatforgalom, ez alapján megkülönböztetjük a processzor belső és külső buszméretét. Ez a kettő nem feltétlenül egyezik meg. (Például az Intel 80386 SX processzor belül 32 bites, kívül 16 bites.) A regiszterek mérete általában a belső buszmérethez igazodik, egy 32 bites processzor regisztereinek mérete szintén 4 bájt.

A memória *lefelé* való kiterjesztése a **háttértár**. Azon kívül, hogy az adatok huzamosabb tárolását a memória meghosszabbításának is fölfoghatjuk, még egy másik módszerről is beszélhetünk. A memória kiterjesztésének másik módja a **swap-**pelés. Ekkor a memóriában lévő, nem használt adatokat az operációs rendszer ideiglenesen kiírja a háttértárra, így szabadítva föl az operatívtár egy részét az éppen futó alkalmazás számára. Ha a félretett adatokra újra szükség van, akkor azokat az operációs rendszernek vissza kell töltenie a memóriába. Természetesen lehet, hogy egy újabb swap-peléssel kell helyet készíteni az operatívtárban.

Két terület közötti adatátvitelt a gyorsítótár, a **cache** közbeiktatásával segítik. A cache használatával az adatátvitel folyamatosabb lehet, zökkenőmentessé tehető. Erre azért van szükség, mert az egyik tárolási módszer lassabb adatátvitelre képes, mint ahogy azt a másik fogadni/továbbítani képes. Ez az adatok feldolgozásakor azt jelentené, hogy az adatok nem állnának folyamatosan rendelkezésre. A cache terület általában kisebb adatmennyiség tárolására képes, de gyorsabb elérésű, mint az adatok forrásterülete. A cache használatát az teszi lehetővé, hogy az adatok feldolgozásának gyorsasága változó. Amíg az adatok feldolgozása folyik, a cache terület feltöltődik, ezzel biztosítva az adatok egyenletes elérhetőségét. Ha az adatok feldolgozása/olvasása folyamatos, akkor nem lenne idő a cache terület feltöltésére, és az adatáramlás előbb-utóbb megszakadna. Ez a probléma a gyorsítótár méretének növelésével általában megoldható, mivel az adatfeldolgozás a legritkább esetben egyenletes.

A processzorban lévő cache a memóriát olvassa, bizonyos algoritmusok alapján töltődik fel. Ezek a függvények azt próbálják kitalálni[3], hogy az aktuális program futásakor a következő művelet végrehajtásakor melyik memóriaterületekre lesz szükség.

Egy kicsit előre szaladtunk, de az itt tárgyalt fogalmak egy része a számábrázolásnál elő fog kerülni. Az meg elég furcsán nézett volna ki, ha ez a fejezet a számrendszerek és a számábrázolás közzé került volna, de igazából ott lenne a helye! Néha átmentünk Hardver 1-be, de a vizsgán lesznek olyan kérdések, amik ehhez kapcsolódnak.

[3] *A megfogalmazás nem mond ellent az előző fejezetben leírtaknak, ahogy egyszer valaki mondta: „Egy rendszer annyira intelligens amennyire a programozója az.”.*

Számrendszerek, műveletek, átalakítások

Nem helyi értékes számrendszerek

Nem helyi értékes számrendszereknél minden számjegynek - függetlenül a helyétől - rögzített értéke van. Ilyen például a római számrendszer vagy a magyar rovásírás. Nos, ezekkel itt tovább nem is foglalkozunk.

Helyi értékes számrendszerek

Hétköznapjainkban a helyi értékes, tízes alapú - hindu-arab - számrendszert használjuk. Egy-egy számjegy értékét a számon belül elfoglalt helye határozza meg. Ha az adott számrendszerben a-val jelöljük a számrendszer alapját, akkor az a alapú ($a > 1$) számrendszerben egy adott helyi értéken 0-tól ($a - 1$)-ig lehetnek számjegyek. (Tízes számrendszerben: 0-tól 9-ig.) Ha egy helyi értéken a számjegyek száma eléri vagy meghaladja az a-t, akkor a következő helyi értékre plusz 1 átvitele történik. (Tízes számrendszerben, pl. egy összeadás elvégzésekor az egyik helyi értéken, 16-ot kapunk, akkor az adott helyi értéken marad a 6, és a következő helyi érték 1-gyel nő.)

A helyi értékes számrendszerben a helyi értékek az adott alap, az a szám hatványai.

A tízes (decimális) számrendszer helyi értékei:

…	10^3	10^2	10^1	10^0	10^{-1}	…
…	1000	100	10	1	0.1	…

A kettes (bináris) számrendszer helyi értékei:

…	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	…
…	256	128	64	32	16	8	4	2	1	0.5	0.25	…

Bináris számrendszerben a 0 és az 1 számjegyeket használhatjuk. Bármely számrendszerben a fölírt számot (egészek esetében) balról és (tört értéknél) jobbról tetszőlegesen kiegészíthetjük nullákkal, ettől a szám értéke nem fog megváltozni. (Pl. tízes számrendszerben: 1,10 = 001,1 = 1,1000)

Műveletek bináris számrendszerben

A műveletvégzés ugyanúgy történik, ahogy azt tízes számrendszerben végezzük. Az összeadást az előbb tárgyaltuk. A kivonásnál, ha egy adott helyi értéken nem tudjuk elvégezni a műveletet, akkor hozzáadunk a csökkentendő részhez a következő helyi értékből egyet. (Tízes számrendszernél hozzáadunk tizet, ami ugye a következő helyi értéken egyet jelentene.) Ezt a segítségül hívott értéket a következő helyi értéknél ki kell vonnunk, azaz a következő helyi értéken a kivonandót eggyel növeljük. Más műveleteknél előbb decimálisba alakítjuk a számot, tízes számrendszerben elvégezzük a műveletet, majd a kapott eredményt átalakítjuk a kívánt számrendszerbeli számmá.

Példa:

11111101	11010010
00000001 +	01000001 −
11111110	10010001

1. feladat:[4]

Mi lesz a műveletek eredménye?

10010001	10010001
00101111 +	00101111 −

Átalakítás decimálisból binárisba

Az egész és a tört rész átalakítását külön-külön végezzük. Az egész rész átalakításkor egészosztást végzünk a számrendszer alapjával, az osztások maradéka adja meg az új számrendszerbeli számot. Az osztás maradékát kiírjuk, az egészrészt tovább osztjuk addig, amíg nullát nem kapunk. Kettes számrendszerbe való átalakításkor az adott decimális szám egész részét 2-vel osztogatjuk, és figyeljük, hogy az érték páratlan vagy páros, azaz volt-e maradék vagy sem.

[4] *A feladatok megoldását a jegyzet végén, a 30. oldalon találod.*

Példa:

Átalakítás nyolcas számrendszerbe:

183	7	183 = (**22***8) + *7*
22	6	22 = (**2***8) + *6*
2	2	2 = (**0***8) + *2*
0		

Kettes számrendszerbe:

183	1	Páratlan
91	1	Páratlan
45	1	Páratlan
22	0	Páros
11	1	Páratlan
5	1	Páratlan
2	0	Páros
1	1	Páratlan
0		

31	1	Páratlan
15	1	Páratlan
7	1	Páratlan
3	1	Páratlan
1	1	Páratlan
0		

A kapott értéket lentről fölfelé írjuk le! $183_{10} = 10110111_2$. Néhány számot igen könnyű átírni kettes számrendszerbe. Például vegyük a 31-et. Ez egy híján 32, azaz 2^5, ezért az ezt megelőző helyi értékek végig egyesek. $31_{10} = 11111_2$.

A tört rész átalakítása úgy történik, hogy a kapott tört értéket szorozzuk a számrendszer alapjával, a szám egész része a szorzásban nem vesz részt. A szorzásokat addig végezzük, amíg nullát nem kapunk, vagy meg nem unjuk, mivel a tört szám akkor határozható meg pontosan, ha a nevező átírható az alap valamelyik hatványává. Különben végtelen szakaszos törtet kapunk. (Természetesen, ha a tízes számrendszerbeli törtünk végtelen nem szakaszos, akkor a másik számrendszerben is az lesz.) Az új számrendszerbeli számjegyeket a szorzás során keletkező egész részek adják meg.
Példa:

0	325 * 2	0.325 = 3/8
0	750	
1	50	
1	0	
0	0	

0.3:

0	3 * 2
0	6
1	2 ┐
0	4 │
0	8 │
1	6 ┘
1	2 ┐
0	4 │
0	8 │
1	6 ┘

A kapott értéket föntről lefelé kell leírnunk! A $0.325_{10} = 0.011_2$, és a $0.3_{10} \approx 0.01001_2$. Fontos, hogy az ismétlődés meghatározásakor a szorzások eredményeként kapott teljes számokat kell figyelembe vennünk. A 0.3 átalakításakor az ismétlődő rész nem a 6, 2, 4, 8, hanem a 12, 04, 08, 16. A 0.3_{10} számot kettes számrendszerbe átalakítva végtelen szakaszos törtet kaptunk. Az átalakítást elvégezve nem ugyanazt a számot kaptuk, az átírás alatt bizonyos pontatlanság keletkezett!
$183.325_{10} = 10110111.011_2$
$31.3_{10} \approx 11111.01001_2 \approx 31.28125_{10}$

2. feladat:

Írjuk át a 36588.45_{10} számot

2.1 nyolcas,
2.2 tizenkettes,
2.3 hármas számrendszerbe!

A tört részt elég öt jegyig kiszámolni! *A feladatban azért nem szerepel kettes számrendszerbe való átalakítás, mert később még lesz rá módunk!* ☺

Átalakítás binárisból decimálisba

Az átalakítás úgy történik, hogy fölírjuk, melyik helyi értéken szerepel egyes, és az ezeknek megfelelő helyi értékeket összeadjuk.
Példa:

…	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	…
…	256	128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125	…
	0	1	0	0	1	1	1	1	1	0	1	0	0	1	

A 10011111.01001_2 bináris szám decimálisan: 159.28125_{10}. Ekkor az érték pontosan adódik, kivéve azt az esetet, ha tudjuk, hogy végtelen törtről van szó!

3. feladat:

Mennyi az értéke a következő számoknak decimálisan?

3.1 1000000001.1_2

3.2 111111_2

3.3 10110101.0101_2

Átalakítás tizenhatos, nyolcas és négyes számrendszerekben

Az átalakítást az előbb tárgyalt módon is elvégezhetjük. Azaz előbb átváltjuk decimálisba, és a kapott számot váltjuk át a másik számrendszerbe.

…	8^3	8^2	8^1	8^0	8^{-1}	…
…	512	64	8	1	0.125	…
…	6	0	7	7	2	…

$6077.2_8 = (6*512 + 7*8 + 7 + 2*0{,}125) = 3079{,}25_{10}$

De használhatjuk alapként a kettes számrendszert is. Ha az átalakítást binárisból végezzük, akkor sokkal könnyebb dolgunk van, mivel a fönti számok kettő hatványai, ezért az adott kettes számrendszerbeli számokat csoportosítva megkapjuk a másik számrendszerbeli számot. Négyes számrendszerbeli számjegy két bináris számnak felel meg ($4 = 2^2$), egy nyolcas (oktális) számjegy háromnak ($8 = 2^3$), egy tizenhatos (hexa) számjegy négynek ($16 = 2^4$). A tizenhatos számrendszer 10-15 számjegyeit az ábécé nagybetűivel jelöljük: 10-A, 11-B, 12-C, 13-D, 14-E, 15-F. Az átalakításnál a csoportosítást a törtponttól kezdjük, és haladunk balra (egészrész) illetve jobbra (törtrész).

Ha kettes számrendszerből alakítunk át, és a szám végein a csoportosításhoz nincs elég számjegy, akkor ki kell egészítenünk nullákkal. Ha csak kettes, négyes, nyolcas vagy tizenhatos számrendszerben kell átváltanunk, akkor elég csoportosításokat végeznünk. Az átalakítás természetszerűen oda-vissza működik. Ha tizenhatosból nyolcasba kell átváltanunk, akkor jobb, ha átmenetnek nem a tízes, hanem a kettes számrendszert választjuk.

Példa:

$|1110|1110|_2 = EE_{16}$,

$|01|10|11|10_2 = 1232_4$,

$|001|110|111|.100|_2 = 167.1_8$

$FA397C_{16} = 1111|1010|0011|1001|0111|1100_2 = 76434574_8$

4. feladat:

Alakítsuk át a következő számokat

4.1 1111111.1_2 - hexába,

4.2 0111010110.1011_2 - nyolcasba,

4.3 $DC1AF7_{16}$ - kettesbe!

Számábrázolás, tárolási módok

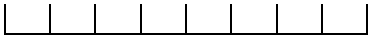
1. Fixpontos
   1.1. Előjel nélküli
   1.2. Előjeles
   1.3. Eltolt
   1.4. Komplementeres
2. Lebegőpontos
3. Binárisan Kódolt Decimális (BCD)
   3.1. Zónázott
   3.2. Pakolt

1. Fixpontos ábrázolás

A fixpontos ábrázolási módoknál a törtpont (tizedesvessző, tizedespont, kettedespont, stb.) helye rögzített. Többségében egész számok tárolására használják, így a törtpont az ábrázolt szám végén van. Ha mégis tároljuk a tört részt, akkor tudjuk, hogy hány biten tároljuk. Ha 32 biten ábrázolunk egy számot, és a tört rész tárolására 24 bitet használunk, akkor az egész rész tárolására már csak 8 bit maradt, a törtpont a 24. bit után áll.

1.1. Előjel nélküli

Előjel nélküli ábrázolásnál csak pozitív számokat tárolunk.



A számok tárolására n db bit áll rendelkezésünkre, így 2^n féle értéket (a permutációk száma) tárolhatunk, 0-tól $(2^n–1)$-ig. A bitek értékei ugyanazok, a szám ábrázolása ugyanúgy történik, mint amikor kettes számrendszerben írjuk föl a számot. A tárolható érték egy bájton: 0…+255. Számegyenesen ábrázolva:

Példa:

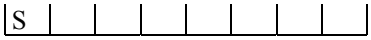
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1 =	0	0	0	0	0	0	0	1
38 =	0	0	1	0	0	1	1	0
128 =	1	0	0	0	0	0	0	0

5. feladat:

5.1 Ábrázoljuk a 168528 / 1536 számot 16 biten, az első 8 biten tároljuk az egészeket,
5.2 256-ot, tört részt nem tárolunk,
5.3 11544 / 384 számot 8 biten, a törtrészt 3 biten tároljuk!

1.2. Előjeles

Ahhoz, hogy negatív számokat is ábrázolni tudjunk, valahogy el kell tárolnunk a szám előjelét. Az előjeles, az eltolt és a komplementeres ábrázolások ezt valósítják meg különbözőféleképpen.



Az utolsó biten (szignum - előjel bit) a szám előjelét tároljuk, 1 ha negatív, 0 ha pozitív a tárolt érték. Ezért az adott szám abszolút értékének tárolására már csak n–1 db bit áll rendelkezésre. A permutációk száma itt is ugyanannyi, ezért ennél az ábrázolási módnál is 2^n $(2*2^{n-1})$ különböző féle értéket tárolhatunk. De, ha így ábrázoljuk a negatív számokat, akkor az az eset áll fönt, hogy a nullából kétféle érték is adódik. Van egy „pozitív” és egy „negatív” nullánk, ezért inkább azt mondhatjuk, hogy 2^n–1 féle értéket ábrázolhatunk.
A számtartomány: $\pm(2^{n-1}–1)$, egy bájton: –127…+127. Számegyenesen:

Példa:

	S	2^6	2^5	2^4	2^3	2^2	2^1	2^0
127 =	0	1	1	1	1	1	1	1
–1 =	1	0	0	0	0	0	0	1
–127 =	1	1	1	1	1	1	1	1

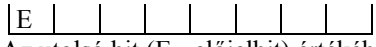
6. feladat:

Ábrázoljuk a következő számokat nyolc biten! A törtrész tárolására egy bitet használunk!

6.1 0
6.2 –45
6.3 –62.5

1.3.Eltolt

Egy másik tárolási módja a negatív számok tárolásához. Az előjeles ábrázolással ellentétben az eltolt és a komplementeres ábrázolásnál nem fordul elő kétféle nulla érték.



Az utolsó bit (E - előjelbit) értékéből itt is megállapíthatjuk, hogy a szám milyen előjelű, de ez a bit nem csak az előjel jelzésére szolgál. Előjel nélküli ábrázolásnál - ahogy az fönt láthattad - a képzeletbeli számegyenesünk nullától kezdődik, az eltolt ábrázolásnál a kezdőpontot negatív irányba eltoltjuk. Valami megmagyarázhatatlan oknál fogva, pontosan az ábrázolható számok felével, azaz 2^{n-1}-nel. Ezért a letárolt nulla érték $–2^{n-1}$-nek felel meg. Így a számegyenes negatív és pozitív oldaláról is 2^{n-1} db számot tárolhatunk. A bitsorozatot úgy számolhatjuk ki, hogy a 2^{n-1}-hez (egy bájton 128-hoz) hozzáadjuk az ábrázolni kívánt számot. Pl. ha a –33 akarjuk ábrázolni, akkor 128 + (–33) = 95. Eltolt ábrázolásnál a 95-nek megfelelő bináris szám –33-nak felel meg. Ennél az ábrázolásnál - nem úgy, mint az előjelesnél - ténylegesen 2^n féle számot ábrázolhatunk. A nulla a pozitív részbe esik, ezért a legnagyobb pozitív szám a $2^{n-1}–1$. Egy bájton: –128…+127.

Példa:

	E	2^6	2^5	2^4	2^3	2^2	2^1	2^0
33 =	1	0	1	0	0	0	0	1
–33 =	0	1	0	1	1	1	1	1
0 =	1	0	0	0	0	0	0	0

Vegyük észre, hogy ha pozitív számokat ábrázolunk, akkor - egy bájt esetén - hét biten ábrázoljuk a számot, ahogy azt előjel nélkül tennénk, és az utolsó bitet is bejelöljük, azaz az ábrázolt pozitív számhoz még hozzáadunk 128-at! Negatív számnál már el kell végeznünk a műveletet, és a kapott értéket kell ábrázolnunk.

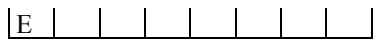
7. feladat:

Ábrázoljuk a következő számokat egy bájton, tört részt nem tárolunk!

7.1 –128
7.2 +127
7.3 +65
7.4 –65

1.4.Komplementeres (fixpontos)

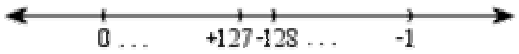
Egy másik mód, amivel az előjeles egész számokat tárolhatunk. Ezzel az ábrázolási móddal egyes aritmetikai műveletek elvégzését is meggyorsíthatjuk. *Figyelem!* Ha fixpontos ábrázolást emlegetnek, akkor komplementeres ábrázolást kell érteni alatta.[5]



Az utolsó bit itt is az ábrázolt szám előjelének jelzésére szolgál. Természetesen nem úgy, ahogy az előjeles ábrázolásnál! Ha pozitív számról van szó, akkor ugyanúgy kell ábrázolni az adott számot, ahogy azt az előjel

[5] *Talán azért nevezik így, mert nem tudják eldönteni, hogy helyesen komplementeresnek, avagy komplemensesnek hívják-e. Egyik számítás-technikus sem nyelvész, ezért valószínű, áthidaló megoldásként nevezik fixpontosnak. De a fixpontos név igazából nem csak ezt az ábrázolási módot jelöli. Lehet, hogy hibásan, én komplementeresnek fogom hívni.*

nélküli tárolásnál tennénk, az utolsó bit ekkor nulla. A negatív szám ábrázolásakor előbb egyes komplemenst képzünk, majd ebből kettes komplemenst.
Az adott helyen letárolható legnagyobb előjel nélküli számból, 2^n-ből (az utolsó bitet is beleértve minden bit csupa egyes) ki kell vonni a tárolni kívánt negatív szám abszolút értékét (egyes komplemens), és hozzá kell adni egyet (kettes komplemens). Ezt azért kell megtenni, hogy ne legyen kétféle nulla értékünk! Azaz eggyel eltoljuk a negatív számokat. A negatív nullából mínusz egy lesz. Láthatjuk, hogy a pozitív számokat normál módon ábrázoljuk (2^{n-1} db biten), a negatív számokat $2^n - |x|$ alakban.
A kettes komplemens képzésnek van egy könnyebb módja is. Ábrázoljuk a tárolni kívánt negatív számot abszolút értékben (ekkor az utolsó bit nulla). A kettes komplemenst úgy kapjuk, hogy az első egyesig bezáróan leírjuk úgy, ahogy szerepel, utána minden számjegynek a fordítottját vesszük. Pl: $|-108| = 01101100_2 \Rightarrow \mathit{10010}\mathbf{100}_2 = -108$.
Komplementeres ábrázolásnál a kivonás helyettesíthető egy összeadással, és az eredményt szintén kettes komplemensben kapjuk meg.
A tárolható értékek egy bájton: –128…+127. Számegyenesen szemléltetve, valahogy így nézne ki a komplementeres tárolási mód:



Példa:

	E	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0 =	0	0	0	0	0	0	0	0
–1 =	1	1	1	1	1	1	1	1
127 =	0	1	1	1	1	1	1	1

Példa műveletekre:

	E	2^6	2^5	2^4	2^3	2^2	2^1	2^0
–31 =	1	1	1	0	0	0	0	1
–31 =	1	1	1	0	0	0	0	1
–62 =	1	1	0	0	0	0	1	0

	E	2^6	2^5	2^4	2^3	2^2	2^1	2^0
120 =	0	1	1	1	1	0	0	0
–1 =	1	1	1	1	1	1	1	1
119 =	0	1	1	1	0	1	1	1

8. feladat:

Ábrázoljuk a következő számokat kettes komplementeres - ún. fixpontos - ábrázolással! Tört részt nem tárolunk.

8.1 –127
8.2 –128
8.3 +64
8.4 –64

2. Lebegőpontos

A matematikában megszokott normál alakos felírásnak a számítástechnikában való megvalósítása a lebegőpontos ábrázolás. Ezért hívják néha lebegőpontos normált alaknak. A számítógépen a tört számok tárolása többségében ezzel a módszerrel történik.
A matematikában használt **normál alak** felírási módja:
$M * 10^k$ - Ahol M a **mantissza** ($0 \le M < 10$), és k a **karakterisztika**.
A számítástechnikában a lebegőpontos ábrázolásnál használt általános képlet:
$M * a^k$ - Ahol a a számrendszer alapja ($a > 1$), az ábrázolásnál kettő hatványai fognak előkerülni: kettő, négy, nyolc, tizenhat. Elég ezeket begyakorolni! M a mantissza, de: $a^{-1} \le M \le 0$. (Úgy szokták írni, hogy M kisebb nullánál, de szerintem, lehet nulla is, hisz csak így tudunk lebegőpontosan nullát ábrázolni.) Egyébként ez a képlet semmi mást nem jelent, csak azt, hogy az adott számrendszerben az a^{-1} helyi értéken, a törtpont előtti első számjegynek kell nullától különböző értéknek lennie. Az ábrázolás általában négy bájton (32 bit) történik, de vannak ettől eltérések.

S	K…	M…

- Az első bit (szignum) a mantissza - ezzel együtt az ábrázolt szám - előjelét jelzi. Ha negatív: 1, ha pozitív: 0, ugyanúgy, mint az előjeles ábrázolásnál. Az eltolt ábrázolásnál az előjel bit értéke fordított!
- A szignumot követő pár db biten a karakterisztikát tároljuk. Ez általában 7 bit, de vannak ettől eltérések (pl. vizsgán). Ez az érték adja meg, hogy az adott számrendszerben a törtpontot hány helyi értékkel és merre kell eltolnunk, hogy megkapjuk a letárolt számot. Ha negatív, akkor balra; ha pozitív, akkor jobbra tolódik a törtpont. A karakterisztika tárolása eltoltan történik, az ábrázolására az ott leírtak érvényesek.
- A mantissza ($a^{-1} \le M \le 0$) tárolására a megmaradó biteket használjuk. Ez általában: 32–(1+7) = 24 bit, azaz 3 bájt. (Kivéve, amikor nem.) A mantissza hossza, illetve a tárolására szánt hely mérete, az ábrázolni kívánt szám pontosságát határozza meg. A mantissza értékét - csak a tört részt, hisz az eltolás miatt az egész rész nulla - mindenféle átalakítás nélkül, nem előjelesként, és nem is előjel nélküliként, mint egyszerű bitsorozatot tároljuk.

A karakterisztika nagysága az ábrázolható legkisebb és legnagyobb számot határozza meg. A legkisebb tárolható érték úgy adódik, hogy a legkisebb ábrázolható mantissza értéket és a legkisebb karakterisztika értéket (negatív szám!) vesszük: $|min| = a^{-1}*a^{-k}$. A legnagyobb tárolható értékhez a legnagyobb ábrázolható mantissza értéket és a legnagyobb karakterisztika értéket vesszük: $|max| \approx 1*a^{k}$. (A maximális érték azért közelítő, mert a mantissza értéke nem lehet egy!) Az ábrázolható tartomány:
$-1*a^{k}\ldots-a^{-1}*a^{-k}$,0, $a^{-1}*a^{-k}\ldots1*a^{k}$
A tárolható tartomány 32 biten, 7 bites karakterisztikával, 2-es alappal:
$-2^{63}\ldots-0.5*2^{-64}$,0, $0.5*2^{-64}\ldots1*2^{63}$
Számegyenesen ábrázolva a tartományt:

Jól látható, hogy ahogy tetszőlegesen nagy szám sem, úgy tetszőlegesen kicsi szám sem (kivéve a nullát) ábrázolható. Túlcsordulásról akkor beszélünk, ha a tárolni kívánt szám abszolút értékben túl nagy; alulcsordulásról, ha abszolút értékben túl kicsi, azaz a tárolni kívánt szám kívül esik az ábrázolható tartományon. Egész számok lebegőpontos tárolása addig pontos, amíg a karakterisztika értéke túl nem lépi a mantisszán tárolható számjegyek számát, ha ezt túllépi, akkor a letárolt egész szám is csonkolódik, a mantisszán nem tárolható számjegyek nullák lesznek.
A mantissza ($a^{-1} \le M < 0 \equiv a^{-1} \ne 0$) alakjából következik, hogy kettes alapnál a mantissza első bitje mindig egy. Ekkor ezt az egy bitet nem szükséges tárolnunk, hisz az értéke mindig egy lesz, egy bit helyet felszabadíthatunk. A felszabaduló egy bitet két dologra is használhatjuk:

- Növelhetjük a pontosságot: Azaz a mantisszán tárolt értéket eltoljuk egy bittel balra. Így plusz egy bittel több értéket tárolunk a mantisszából.
- Növelhetjük az ábrázolható számtartományt: azaz a fölszabaduló egy bitet átadjuk a karakterisztika számára. Ezzel növelve a karakterisztika számtartományát.

Ezt nevezik **rejtett ábrázolás**nak. *Figyelem!* Rejtett ábrázolás csak kettes alapnál fordulhat elő! Más alapot választva nem biztos, hogy a mantissza első bitje egy lesz, ezért nem alkalmazható. (Most jut eszembe, hogy rejtett ábrázolásnál vajon, hogyan tárolják a nulla értéket?)
Ha valaki, valahol, valamikor azt kérné, hogy prezentáljatok valamiféle lebegőpontos rejtett ábrázolást, akkor vegyétek figyelembe a föntieket, azaz, hogy a fölszabaduló egy bittel két dolgot is tehettek, és csak kettes alapnál alkalmazható, hogy a valóságban melyik történik, az megvalósítás függő.

Példa:
Ábrázoljuk az 1/25 értéket lebegőpontosan, 4 bájton, 7 bit karakterisztikával, kettes alappal, rejtett ábrázolással. A szám pozitív, ezért az első bit, a szám előjelbitje 0 lesz. Ez után a kettes számrendszerbe átírt számot úgy kell alakítani, hogy a törtpont utáni első számjegy különbözzön nullától.
$1/25 = 0.04 \approx 0.00001010001111010111_2 = 0.1010001111010111_2*2^{-4}$
Ha más az alap, akkor is jobban járunk, ha már rögtön kettes számrendszerbe váltjuk át a számot, de ekkor lehet, hogy a kapott számot bal oldalt ki kell egészítenünk nullákkal, hogy az eltolást el tudjuk végezni. Pl. egy nyolcas számrendszerbeli számjegy három biten ábrázolható, az előbbi példánál maradva ekkor a mantissza 0.010-val kezdődik. Az eltolás iránya és mértéke adja meg a karakterisztikát, a példánkban ez –4. Eltoltan, hét biten ábrázolva: 64+(–4) = 60. Segítségképpen megjegyzem, hogy a szám pozitív, míg a karakterisztika negatív, ezért a két előjelbit (a szám és a karakterisztika) értéke meg fog egyezni. Rejtett ábrázolást használva a pontosságot fogjuk növelni, a mantissza értékét eltoljuk balra, így a mantisszából 24 bit helyett 25 bitet tárolunk. A memóriában tárolt bitértékek:

S	k…	M…			
0	0111100	01000111	10101110	00010100	= 3C47AE14h

Figyelem! Sose felejtsd el teljesen kiszámolni és kiírni a mantissza értékét! Akkor is, ha az szakaszosan ismétlődik, hisz ha nem írod ki, az már nem ugyanaz a szám. Többeknek ezért nem sikerült a vizsgájuk.

9. feladat:

9.1 Ábrázoljuk a –1/40-et lebegőpontosan, 4 bájton, 7 bit karakterisztikával, 16-os alappal!
9.2 Ábrázoljuk a 25000-et lebegőpontosan, 4 bájton, 7 bit karakterisztikával, 4-es alappal!
9.3 Ábrázoljuk az 1000.125-öt lebegőpontosan, 4 bájton, 6 bit karakterisztikával, rejtett ábrázolással!

3. Binárisan Kódolt Decimális

Az előző fejezetben láthattuk, hogy a törtszámok kettes számrendszerbe való átváltásakor a legritkább esetben kapunk pontos eredményt. Ha az előbb tárgyalt módokon tároljuk a számokat, ez a nagy pontosságot igénylő

számításoknál lényeges lehet. Akár a tizedik helyen álló törtjegy értéke is fontos lehet, miközben már az első tizedes jegy értéke sem pontos. Erre a problémára adhat megoldást a BCD ábrázolási mód.
Ennél a tárolási módnál nem a számot, hanem a számjegyeket tároljuk. A tízes alapú számrendszerben 0-tól 9-ig vannak számjegyek, azaz egy-egy számjegyjegy tárolására 4 bit (fél bájt) elégséges. Ez a kódolási mód is fixpontos, azaz már a kódolás előtt tudjuk, hogy hol helyezkedik el a tizedespont, mégsem ebbe a csoportba soroljuk, mert az ábrázolási mód ettől eltekintve különbözik az ott leírtaktól. A rögzített tizedespont helye meghatározza, hogy mekkora egész és törtértéket tárolhatunk. Ennek a számábrázolási módnak is több fajtája van, attól függően, hogy az ASCII vagy az EBCDIC kódtáblára alapul, és mindkettőn belül van még tömörített ábrázolás is. Itt az EBCDIC zónázott és pakolt (tömörített) ábrázolási formákat tárgyaljuk.

3.1. Zónázott

A zónázott ábrázolásnál minden számjegyet külön bájtban tárolunk. A számjegyet a bájt alsó részében tároljuk, a fölső részt pedig egyesekkel töltjük fel, ami a hexa F-nek felel meg, ezt zónajelnek hívják. Így bájtonként minden számjegy az EBCDIC kódtábla értékeit veszi föl, a nulla EBCDIC kódja 240 (F0h), a kilencé 249 (F9h). A szám előjelét az első bájt zónajelében tároljuk. A negatív előjelnek az A és C, a pozitív előjelnek a B és D felelnek meg, de az A-B párosítást nem használják!

Példa:

1333	=	F1F3F3C3
–256	=	F2F5F6F0D0
–3.0256032	=	F3F0F2F5F6F0F3D2

Az elsőnél nulla, a másodiknál kettő, a harmadiknál hét bájtot használunk a tizedes jegyek tárolására. Zónázott ábrázolásnál legalább annyi bájtra van szükségünk ahány számjegyből áll a tárolni kívánt szám.

3.2. Pakolt

A pakolt ábrázolási módnál elhagyjuk a zónajeleket, és a számjegyek tárolását nem bájtonként, hanem félbájtonként végezzük. Ezt a tömörítést tízes számrendszerbeli szám ábrázolásakor megtehetjük, mivel a szóba jöhető számjegyeket fél bájton is tudjuk tárolni. Az előjel jelzése a zónázotthoz hasonlóan történik, azzal az eltéréssel, hogy a hexa C-t vagy D-t az első fél bájton tároljuk. Előfordulhat, hogy az utolsó fél bájtra nem jut számjegy, ekkor az nulla lesz. (Ha még emlékszel rá, az adatkezelés alapja a bájt, és nem a bit.)

Példa:

1333	=	13330C
–256	=	256D
–3.0256032	=	030256032D

Az elsőnél egy, a másodiknál nulla, a harmadiknál hét számjegyet tárolunk a törtrészből.

10. feladat:

Az ábrázolási módok összefoglalásaként az utolsó feladatsor. Mennyi a memóriában négy bájton tárolt 9807329Dh bitsorozat értéke

10.1 előjel nélküli egész,
10.2 előjeles egész,
10.3 eltolt,
10.4 komplementeres,
10.5 lebegőpontos (4 bit karakterisztika, 8-as alap),
10.6 zónázott és
10.7 pakolt ábrázolási módokban?

Gépi kódú utasítások operandusai

Ezek a matematikai műveletekhez hasonlóan épülnek föl. Egy művelet elvégzésekor megadjuk, hogy melyik műveletet végezzük el és milyen adatokon. (3 + 2, sin 30, ln 10, stb) A gépi kódú utasítások megadásakor a műveleti jel elől szerepel (3 + 2 ⇒ + 3 2). A művelet alatt nem csak aritmetikai/logikai műveletet lehet érteni, hanem más, számítástechnikai utasítást is, például egy tárrészbe tölti a megadott értéket. Ahogy a matematikában, úgy a számítástechnikában is, az utasítás meghatározza, hogy hány operandust követel meg. A számítástechnikában, gépi szinten csak egyszerű aritmetikai és logikai műveletek vannak. Az utasításokat pedig - ugyanúgy, ahogy az operandusokat is - különböző bájtértékek, számok jelzik. Most ezen utasítások a megadási módjait tárgyaljuk.

Négycímes utasítás

utasítás kód | *első operandus* | *második operandus* | *eredmény címe* | *következő utasítás*
Az utasítás megadása után megadjuk a két operandust és azt a tárcímet ahova a művelet eredménye töltődni fog. Ezek után meg kell még adnunk a program következő utasításának tárcímét. Ezt a megadási módot már nem használják.

Háromcímes utasítás

utasítás kód | *első operandus* | *második operandus* | *eredmény címe*
Ha a program utasításait a tárban folytatólagosan helyezzük el, akkor ismerve az adott utasítás hosszát és címét, ki tudjuk számoltatni a következő utasítás címét. Nem kell megadnunk a negyedik címet. Ez a művelet automatikusan történik, a hardver végzi el. Létezik egy Program Counter (PC) nevű regiszter, amiben a következő utasítás memóriacíme tárolódik. A számítógép minden utasítás végrehajtásakor automatikusan kiszámolja a következő utasítás címét és megváltoztatja a PC értékét. Ezzel követve az utasítások sorrendjét. A mai gépek e szerint az elv szerint működnek.

Kétcímes utasítás

utasítás kód | *első operandus* | *második operandus*
A harmadik címet is elhagyhatjuk, ha meghatározzuk, hogy az eredmény címe az első vagy a második operandus helyére töltődjön. (Az utasítás kódja adja meg, hogy az adott utasítás, hogyan működik.) Igaz, ekkor az operandus eredeti értékét elveszítjük.

Egycímes utasítás

utasítás kód | *első operandus*
Ekkor az egyik operandus az Akkumulátorban helyezkedik el, és az eredmény is ide kerül, a művelet megadásakor csak a másik paramétert kell megadnunk. Az utasítás végrehajtása előtt az első operandust az Akkumulátorba kell töltenünk, és a művelet(ek) befejezése után, ha szükségünk van az eredményre, ki kell vennünk az Akkumulátorból. Ennek akkor van igazán értelme, ha az eredménnyel folyamatosan, több műveletet is el szeretnénk végeztetni, ti. a regiszter elérése nagyságrenddel gyorsabb, mint az operatívtáré.

Nem minden utasítás igényel két operandust. Vannak egy operandusú vagy akár operandus nélküli utasítások is. Ezen kívül, nem minden utasítás ad vissza értéket. Egy ugró (JUMP) utasítás lehet egy operandusú is, és a végrehajtása során nem keletkezik visszaadott érték. Az üres utasítás (NOP) semmilyen operandust nem követel meg.

Címzési módok

Az operandusok megadása - akár magasabb szintű nyelveken is - különböző módokon történhet. Attól függően, hogy az adatott hogyan is érjük el, különböző címzési módokat különböztetnek meg.

Közvetlen adat

utasítás | *adat*
Ekkor az operandus helyén maga a feldolgozandó adat szerepel. Ekkor a paraméter egy konstans, állandó érték.

Direkt (közvetlen) címzés

utasítás | *címhivatkozás* ⇒ cím[adat]
Az operandus helyén az operatív memória egy tárcíme van, ezen a címen található az adat.

Regiszteres direkt címzés

utasítás | *regiszterhivatkozás* ⇒ regiszter[adat]
Az operandus helyén egy regiszterhivatkozás szerepel. Ebben a regiszterben található az adat.

Indirekt címzés

utasítás | *címhivatkozás* ⇒ cím[címhivatkozás] ⇒ cím[adat]
Az operandus helyén egy cím van. Ami az operatívtár egy másik címére mutat, és ez a cím adja meg, hogy hol is található a tárban az adat.

Regiszteres indirekt címzés

utasítás | *regiszterhivatkozás* ⇒ regiszter[címhivatkozás] ⇒ cím[adat]
Az operandus helyén egy regiszterhivatkozás szerepel. A regiszter egy címhivatkozást tartalmaz, az adat ezen a tárcímen található.

A következő két címzési módnál egy-egy nagyobb adathalmazt kezelünk. Az adathalmazunk minden egyes eleme azonos nagyságú tárterületet foglal el (pl. egy bájt), és a memóriában folytatólagosan helyezkednek el. Az első elem kezdőcíme megadja a bázist, azt a memóriacímet, ahol az adathalmaz kezdődik. Ettől a bázistól kezdve az adatok elérése indexeléssel történik, az első elem indexe nulla. Az *n.* elem tárcímét a következő képlettel kapjuk meg:

$X_n = b + (i*l)$

Ahol b (base pointer) - a báziscím; l (length) - egy adatelem által lefoglalt memóriaterület mérete; i (index) - az adott elem indexe (indexelés nullától); X_n - az *n.* elem memóriacíme. Az első elem címeként a báziscímet kapjuk vissza.

Indexregiszteres címzés

utasítás | *regiszterhivatkozás* ⇒ regiszter[index]
Indexregiszteres címzésnél az adatok hosszán kívül (length), a báziscím (base pointer) adott, előre meghatározott. Az index értékét egy regiszterben tároljuk. Ezzel a móddal, a regiszter értékét változtatva, az adott memóriaterület minden egyes elemén végig tudunk lépkedni. Valójában ezt a címzési módot valahogy így kellene felírnunk:
utasítás | *regiszterhivatkozás* ⇒ regiszter[index] ⇒ függvény(cím) ⇒ cím[adat]
A függvény értékét a számítógép a címzési mód alapján automatikusan számolja ki, nem nekünk kell vele foglalkoznunk.

Bázisregiszteres címzés

utasítás | *regiszterhivatkozás* ⇒ regiszter[bázis]
Bázisregiszteres címzésnél az adatok hossza és az index adott. Egy regiszterben a báziscímet tároljuk. Ugyanúgy működik, mint az indexregiszteres címzés, azzal a különbséggel, hogy itt a regiszter értékét változtatva a báziscímet változtatjuk meg, így különböző memória területeken elhelyezkedő adatok ugyanazon sorszámú elemét tudjuk elérni.

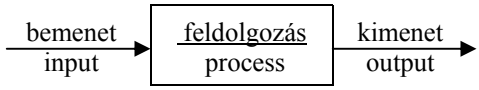
Az index- és bázisregiszteres címzés keverhető, az adatok változtatásával elérhető, hogy az indexregiszteres címzés úgy működjön, mint a bázisregiszteres, és fordítva.

Algoritmusok

Itt megpróbálom összefoglalni azt, amit erről a területről ismerek. De az igazat megvallva egy sor algoritmust sem szeretnék leírni. (Ezt végül is nem tartottam be, de az algoritmusok néhány helyen eltérnek attól, amit leadtak!) Először egy kis elméletről lesz szó, aztán az előadáson vett keresések és rendezések leírása következik. Szerencsére vagy nem, a vizsgán csak a leadott algoritmusokat kérdezik vissza. Ha valakinek nem megy ez az algoritmizálás mifene, akkor elég azokat az algoritmusokat megjegyeznie.[6]

Bemenet, Feldolgozás, Kimenet

Az algoritmizálás alapját a következőképpen adhatnánk meg:



Nézzük a következő feladatot! Írjuk ki egy pozitív egész szám összes osztóját a képernyőre. A teljes algoritmust tekintve a bemenet: egy pozitív egész szám, kimenet: a megjelenő számsorozat. De be- és kimeneti részek alatt érthetjük a program azon részeit is, amik ezt valósítják meg. A program bemeneti része: ami beolvassa a számot, feldolgozási része: ami kiszámolja az osztókat, kimeneti része: ami a képernyőre írja a számokat. De tovább is bonthatjuk a dolgot, nézetünket leszűkítve egyetlen utasításra. A következő programsornak is létezik be- és kimenete:

$x = n / p;$ - Ekkor a bemenet: n és p, kimenet x, feldolgozás: n / p. Úgy tűnhet, hogy a három művelet közül a legfontosabb dolog a feldolgozás. A valóságban mindhárom rész azonos hangsúlyt kell, hogy kapjon, és az is elképzelhető, hogy a beviteli vagy a kimeneti részek bonyolultabbak. De nem kell feltétlenül bonyolult dologra gondolnunk, az előző programsorban lényeges a *p* értéke, a továbbiakat tekintve, pedig a kimenet, az *x* értéke sem elhanyagolható. Egyáltalán nem állíthatjuk azt, hogy a bemenet ⇒ feldolgozás ⇒ kimenet közül bármelyiket is előtérbe kellene helyeznünk. Ha a program egészét nézzük, akkor a bemeneti/kimeneti részei valósítják meg a kapcsolattartást a felhasználóval. Hiába csinál csoda dolgokat a program, ha nehézkesen lehet „vele" kommunikálni.

Lényeges, hogy egy algoritmus mennyire paraméterezett. A paraméterek használata teszi hajlékonnyá, az algoritmust. Bátran használjunk őket, csak el ne tévedjünk a paraméterek erdejében! ☺ Itt paraméterek alatt nem csak a függvények operandusait kell értened! Hanem belső neveket is. Hisz' valahol azok is valaminek a bemenetei lesznek, és szolgáltatnak majd valamilyen értéket.

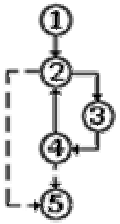
Szekvencia, Szelekció, Iteráció

A fönti három varázsige az algoritmizálás terminus technicus-ai. (Sikerült még egy idegen kifejezést ide biggyesztenem! ☺) De egyáltalán nem kell bonyolult dolgokra gondolni!

Egy algoritmus valamiféle sorrendiséget takar. Csináljuk ezt, csináljuk azt, stb. Ezt a sorrendiséget nevezik **szekvenciának**. Gondolj csak a lineáris (szekvenciális) keresési módra! Ahol egy adott adathalmaz összes elemét, sorban megvizsgáljuk. Az algoritmusban minden olyan utasítássorozat, amikor a leírt műveletek sorban végrehajtódnak.

Erre a szekvenciális dologra épül az elágazás, a **szelekció**. Az utasítások végrehajtása közben van, mikor nem feltétlenül szeretnénk, hogy végrehatódjon egy utasítás (utasításcsoport). Ennek az utasításnak a végrehajtása mindig egy feltételtől függ. A feltétel egy logikai értéket - igent vagy nemet - szolgáltat, ami alapján eldönthetjük, hogy a szóban forgó utasítást végre szeretnénk-e hajtatni. A feltétel igaz értékekor végrehajtódik a szelekciós rész szekvenciális része, ha úgy tetszik, az utasítások sorrendje vagy az egyik vagy a másik ágon folytatódik. A szelekciónak több fajtája is létezik. Van, mikor csak egy utasításról szeretnénk eldönteni, hogy végrehajtódjon-e vagy sem. Úgymond pluszként a végrehajtandók közzé rakjuk. A másik, amikor kettő (vagy több) utasításból szeretnénk, ha egy feltétlenül végrehajtódjon. A szelekciók egymásba skatulyázhatóak, azaz a szelekció szekvenciális részében újabb szelekciók is szerepelhetnek. (Remélem, már kellőképpen követhetetlen! ☺)

A szekvenciára és a szelekcióra épül az ismétlés, az **iteráció**. Az algoritmuson belül egy-egy részfolyamatot ismétlünk. Az ismétlés nem lehet végtelen, valamikor be kell azt fejeznünk. (Akárki akármit mond, végtelen ciklus nincs, illetve programozói hiba. Ha a ciklust jól csináltuk meg, akkor valahol ki is fogunk lépni belőle.) A ciklusból való kilépést egy szelekció határozza meg. Ez a szelekció a ciklus része, minden ciklusvégrehajtáskor kiértékelődik. A szelekció lehet a ciklus elején, végén és néha a közepén is. Így beszélhetünk elöl tesztelő és hátul tesztelő ciklusokról. A hátul tesztelő ciklus egyszer biztos, hogy végrehajtódik, hisz a kilépési feltétel vizsgálata a ciklus végére került. Az elöl tesztelő ciklus nem biztos, hogy végrehajtódik. Már a ciklus elején kiderülhet, hogy nincs is rá



[6] *Az előadáson leadott algoritmusokat, és nem az ebben a jegyzetben szereplőket!*

szükség! (A középen tesztelő ciklust most nem tárgyaljuk.) Az ismétléseket is egymásba ágyazhatjuk, több iterációs szerkezetet is egymásba építhetünk.

Az algoritmizálás, programozás alatt ezt a három utasításcsoportot használjuk, s valójában nincs is más. Eltekintve a paraméterek és különböző függvények alkalmazásától. Nem nagyon bonyolult ez az egész, csak az adott feladatot szét kell szedni szekvenciára, szelekciókra és iterációkra. Figyelve a be- és kimeneti értékekre, amik megadják, hogy mit és hogyan kell paramétereznünk, milyen változókat kell használnunk.

Lehetne tovább feszegetni ezt a szekvencia, szelekció, iteráció dolgot, van még pár dolog, amit le lehetne írni róluk, de most csak a fogalmak megismertetése volt a cél, hogy mindenki érezze, miről is van szó. Az algoritmizáláshoz ennyi elég is lesz.

Algoritmizálás

Készítsünk algoritmust, ami kiírja a képernyőre a 10-es számot! Valószínű, hogy ez a feladat még senkinek nem haladja meg a képességeit!

1. *kiír 10*

Ahhoz, hogy az algoritmusunk használhatóbb legyen, vezessünk be egy változót! A változó fogalma a matematikában megszokott változó fogalomhoz hasonlatos. Valamilyen nevet adunk neki, s egy adott tartományon, típuson belül különböző értékeket vehet föl. Például: $a \in R_+$.

1. *A = 10*
2. *kiír A*

A két algoritmus között első ránézésre nincs sok különbség. Pedig az algoritmusunkon a változtatás igen jelentős. Most az elméleti hátterét nem tárgyaljuk ki, hogy valójában a változó egy izének - szakkifejezéssel a fizikai világ egy objektumának - egy számunkra lényeges tulajdonságát jelöli. Lényeges különbség, hogy a programunkkal meg tudjuk változtatni az A változó értékét. A másik oldalról nézve, ha az A változót az algoritmus további részében többször is használjuk, akkor javítás, módosítás esetén csak egy helyen kell azon változtatnunk. (Kivéve, ha a nevét változtatjuk meg. ☺)

Az algoritmusunk még mindig nagyon hiányos. A felhasználóval még csak egy irányba "kommunikálunk". Változtassunk hát rajta, és kérjük be az A változó értékét, és ezt az értéket írassuk ki!

1. *A = bekér()*
2. *kiír A*

Nos, itt már nem tudjuk, hogy konkrétan mit is fog kiírni az algoritmusunk. Csak azt tudjuk, hogy valamit bekér, és jelenít meg. Aminek így önmagában nem sok értelme van. Csináljunk, hát valamit ezzel az értékkel! Ellenőrizzük le, hogy pozitív avagy negatív-e, és ezt írjuk ki! Egy kétágú elágaztatást csinálunk.

1. *A = bekér()*
2. *Ha A > 0 akkor kiír „A szám pozitív."*
3. *különben kiír „A szám negatív."*

Az algoritmus megadásakor, a szöveget idézőjelek közzé szokás tenni. Ezzel különítve el az algoritmus többi részétől. Bekérünk egy számot. Megvizsgáljuk, hogy nagyobb-e nullánál. Ha igen, akkor kiírjuk, hogy pozitív; ha nem, akkor azt, hogy negatív. Az algoritmusunk a legtöbb számra helyesen működik, kivéve a nullát. A nullára azt írja ki, hogy negatív, de a nulla sem nem negatív, sem nem pozitív.

1. *A = bekér()*
2. *Ha A > 0 akkor kiír „A szám pozitív."*
3. *különben Ha A < 0 akkor kiír „A szám negatív."*
4. *különben kiír „A szám nulla."*

Ennél a megoldásnál szükség van az elágazások egymásba ágyazására. Ha a két ellenőrzést nem ágyaznánk egymásba, hanem külön-külön lennének, akkor az algoritmus úgy működne, hogy a pozitív számokra kiírná, hogy pozitív és azt is, hogy nulla. Egymásba ágyazva a második ellenőrzés csak akkor hajtódik végre, ha az első feltétel nem teljesül. A algoritmusban a sorok számozása teljesen „önkényesen" történik. A számozás egyfajta csoportosítást is jelent, másféle csoportosítás is elképzelhető.

Az algoritmus rövidsége miatt nem írtam ki, hogy vége. A végét két dolog miatt is ki kell írni. Először is, nem feltétlenül az algoritmus végén van a vége, és akár több vége hely is lehet. A vége kiírásnál szokás jelezni, hogy az algoritmus sikeresen vagy sikertelenül fejeződött-e be.

Most az algoritmusunkat bővítsük azzal, hogy folyamatosan kérjük be a számokat, és folyamatosan írjuk ki az előjelüket. Ágyazzuk be az egészet egy ciklusba, a kilépési feltételünk legyen az, hogy ha nem számot kapunk értékül.

1. *A = bekér()*
2. *Ha A ≠ szám akkor vége*
3. *Ha A > 0 akkor kiír „A szám pozitív."*
4. *különben Ha A < 0 akkor kiír „A szám negatív."*
5. *különben kiír „A szám nulla."*
6. *vissza 1-re*

A 6. lépéssel csináltunk egy ciklust. Ami a bekérést/kiírást folytonosan ismételné, ha a 2. pontban nem tennénk egy ellenőrzést, amire az algoritmusnak vége. Esetünkben ez akkor történik meg, ha szám helyett valamilyen más értéket tartalmaz az A változó.

Bevezetésként ennyi az algoritmusok felépítéséről, szerkezetéről. Lehetne még tovább boncolgatni ezt a témát, de egy jegyzet feladata nem is lehet az, hogy teljes mélységében, és részletességgel vegyen át dolgokat. Ezek alapján egy bonyolultabb algoritmus elkészítése sem lehet nehéz. ☺ Segítségképpen még azt szeretném leírni, hogy előbb az algoritmust szóban próbáld megfogalmazni. Nem részleteiben! Hanem, hogy az adott feladatot az algoritmus hogyan is fogja megcsinálni. Ha ez nagy vonalakban összeállt a fejedben, akkor ebből ki lehet szűrni, hogy milyen változókra van szükség, és milyen műveleteket kell velük végezni. Az elágazások és ciklusok alkalmazása ezután már magától megy.

Az előbbi példát fölhasználva:

- A feladat: Kérjünk be egy számot és írjuk ki a szám előjelét. Ismételjük addig, míg más értéket nem kapunk.
- Már első olvasáskor látjuk, hogy egy változóra biztos szükségünk lesz, valahol tárolnunk kell a bevitt értéket. A számoknak többféle előjelük lehet, szükséges valamiféle ellenőrzés. Kisebb feladatok megadásakor az ismétlésre való utalást már maga a feladat megadása tartalmazza. Ahogy itt is: „ismételjük addig…” vagy annál a feladatnál, amit a ZH feladatsor tartalmaz: „…és kérd be újra a számot…” vagy „…kérj be „A” db valós számot…”. Az utóbbinál még azt is tudjuk, hogy a ciklus A-szor fog végrehajtódni, tehát kell egy újabb változó, amiben számoljuk a bekért értékeket.
- A fönti dolgokat nem kell leírni, csak gondold végig, hogy mit is kell tenned ahhoz, hogy szét tudd bontani az adott feladatot.

Keresések

A keresések és rendezések valamilyen adathalmazon dolgoznak. (Valószínű, hogy ez eddig is egyértelmű volt.) Itt most azt mondjuk, hogy a sorozat egyszerű elemekből áll, és a tagokat meg tudjuk számozni, ezzel különböztetve meg őket egymástól, ez a matematikában megszokott indexelés. A gyakorlatban előfordulhat, hogy nem egyszerű, hanem bonyolultabb adatokkal kell dolgoznunk, de az algoritmusok ekkor is ugyanúgy működnek, a lényegét ez nem érinti. A keresési algoritmusoknál az adathalmaz elemein végig kell lépkednünk (ha valamennyin nem is), ezért a keresési algoritmusokban egy, és csak egy iteráció szerepel.

Minimum/maximum keresés

A minimum/maximum keresésnek rendezetlen adathalmazon van értelme. Rendezettnél már tudjuk, hogy melyik a legkisebb/legnagyobb elem. Első lépésként elvégzünk egy ún. előolvasást, azaz vesszük a halmaz első elemét, és azt mondjuk, hogy ő a legkisebb. Néha azt csinálják, hogy pl. a minimum keresésnél a kezdő értéknek azt a legnagyobb értéket adják, amit az adott változótípus fölvehet. Szerintem ez hibás szemlélet, első lépésként inkább csináljunk egy előolvasást!

Tehát az **előolvasás**: a cikluson kívül beolvassuk az első értéket, és ez lesz a kezdő minimum érték. Ezek után végiglépkedünk az összes elemen, és végrehajtunk egy vizsgálatot, hogy kisebb-e a tárolt minimum értéknél. Ha igen, akkor annak az értékét veszi föl a minimum változó. Természetesen, ha a maximum értéket keressük, akkor a vizsgálatnál azt nézzük meg, hogy az adott elem nagyobb-e a tárolt maximumnál. Lehetséges az is, hogy nem a minimum értéket tároljuk, hanem az indexértéket. Ekkor az összehasonlításnál figyelembe kell venni, hogy nem magát az értéket, hanem a rá hivatkozó indexet tároljuk. Az elemeken végighaladva, $n - 1$ összehasonlítást kell elvégeznünk. A –1 azért van, mert az első elemet rögtön a minimum helyre rakjuk.

Ha egyszerre kell meghatároznunk a minimum és maximum értékeket, akkor jobban járunk, ha az összehasonlítást páronként végezzük el. Az adathalmazból egy elempárt veszünk, és összehasonlítjuk őket. A következő lépésben a kisebbet hasonlítjuk a minimummal, a nagyobbat a maximummal. Így két elemre nem négy, hanem három összehasonlítás jut. Ekkor nem $2*(n–1)$ összehasonlítás, hanem $3*(n/2)–2$ történik.

1. $i = 2$
2. $min = K_1$
3. *Ha* $i > n$ *akkor menj 7-re*
4. *Ha* $min > K_i$ *akkor* $min = K_i$
5. $i = i + 1$
6. *vissza 3-ra*
7. *sikeresen vége*

1. $i = 1,\ j = 2$
2. *Ha* $A_i < A_j$ *akkor* $min = A_i \quad max = A_j$
3. *különben* $min = A_j \quad max = A_i$
4. $i = i + 2,\ j = j + 2$
5. *Ha* $j > n$ *akkor menj 9-re*
6. *Ha* $A_i < A_j$ *akkor*
   Ha $A_i < min$ *akkor* $min = A_i$
   Ha $A_j > max$ *akkor* $max = A_j$
7. *különben*
   Ha $A_j < min$ *akkor* $min = A_j$
   Ha $A_i > max$ *akkor* $max = A_i$
8. *vissza 4-re*
9. *Ha* $i = n$ *akkor*
   Ha $A_i < min$ *akkor* $min = A_i$
   Ha $A_i > max$ *akkor* $max = A_i$

Az első algoritmusnál a sorozatban (K_1, K_2, …, K_n) a legkisebb elemet keressük. Feltételezzük, hogy az adathalmaznak legalább egy eleme van, ha ez nem teljesül, akkor ez az algoritmus nem jól működik, hisz a K_1-nek nincs értéke. Az *i* értékét már az elején 2-re állítjuk, mivel a sorozat első eleme lett a minimum. Ha a keresésnél úgy járunk el, hogy a *min* változóban nem az értéket tároljuk, hanem az indexet, akkor az összehasonlításnál ezt is figyelembe kell vennünk, a *min* változót az *i*-hez hasonlóan kell kezelnünk! $K_{min} > K_i$; $min = i$, és az 1. pontban: $min = 1$.

A második algoritmus a közös minimum/maximum keresés. Előzetesen feltételeztük, hogy a sorozatnak legalább két db eleme van. A végén - a kilences pontban - a miatt kell még elvégeznünk egy ellenőrzést, mert előfordulhat, hogy páratlan elemszámunk volt. Ekkor az utolsó elemet kihagynánk az ellenőrzésből. Ezért attól függően, hogy páros vagy páratlan volt-e az elemszám $3*(n/2)–1$ vagy $3*(n/2)+1$ ellenőrzést kell elvégeznünk.

Teljes keresés (Szekvenciális keresés)

Most nem minimum/maximumot, hanem egy konkrét értéket keresünk. Ennél a keresésnél még mindig rendezetlen adathalmazon keresünk. A teljes keresés igen hasonlít a minimum/maximum keresésre, de itt már előfordulhat az is, hogy a keresett elem nincs benne az adathalmazban.

Adott a keresett érték (egyértelmű), keressük annak az elemnek az indexét, ami egyenlő a keresett elemmel. Sorban végiglépkedünk az adathalmazon, és minden elemet megnézzük, hogy egyezik-e azzal, amit keresünk. Ha igen, akkor sikeresen vége. A sikertelenséget csak akkor jelenthetjük be, ha minden elemet megvizsgáltunk. Ekkor összesen n db összehasonlítást végeztünk el. Az elem megtalálása ellenben már az első lépésben is megtörténhet.

A keresést a **strázsa** módszerrel egy kicsit gyorsíthatjuk. Jelenleg a ciklus minden egyes végrehajtásakor két dolgot ellenőrzünk. Azt, hogy az adott elem egyezik-e a keresettel, és azt, hogy az utolsó elemet vizsgáljuk-e, azaz az adathalmaz végére értünk-e. Ha az adathalmazba az utolsó utáni (n + 1) helyre berakjuk a keresett elemet, akkor a ciklusból elhagyhatjuk a sorozat végének ellenőrzését. Ez az őrszemnek állított elem, a strázsa mindenképpen meg fogja állítani a ciklust. Ekkor a ciklus lefutása után még el kell végezni egy vizsgálatot. (Amit kivettünk a ciklusból.) Ha sikeresen ért véget a keresés, akkor az index $\leq$ n, ha nem volt sikeres, akkor az index = n + 1. A strázsa használatával sikertelenség esetén n + 1 db összehasonlítást végzünk el.

1. $i = 0$
2. $i = i + 1$
3. *Ha* $K_i = K$ *akkor sikeresen vége.*
4. *Ha* $i \leq n$ *akkor vissza 2-re*
5. *különben sikertelenül vége*

1. $K_{n+1} = K \quad i = 0$
2. $i = i + 1$
3. *Ha* $K_i \neq K$ *akkor vissza 2-re*
4. *Ha* $i = n$ *akkor sikertelenül vége*
5. *különben sikeresen vége*

Az első a strázsa nélküli megoldás. A 3. pontban vizsgáljuk, hogy megtaláltuk-e az elemet, és 4.-ben, hogy elértük-e a sorozat utolsó elemét. A másik megoldás a strázsás. Első lépésként a sorozat n+1 helyére berakjuk a keresett elemet. A ciklus csupán két sorból áll. Az utolsó két sor egy kétágú szelekció.

Lineáris keresés (Szekvenciális keresés rendezett sorozatra)

Ez az algoritmus rendezett adathalmazon működik. Ennek ellenére igen hasonlatos a teljes kereséshez. Végiglépkedünk az elemeken, de a ciklusban nem azt vizsgáljuk, hogy az elem megegyezik-e a keresettel, hanem azt, hogy kisebb-e nála. (Növekvő sorrendbe rendezett elemekről van szó.) Ha nem kisebb, akkor vége a ciklusnak. A ciklus után még végre kell hajtanunk egy vizsgálatot, mivel a ciklus két dolog miatt fejeződhetett be. Az elem vagy megegyezett a keresettel (sikeres vége), vagy nagyobb volt nála (sikertelenül vége). Rendezett adathalmazon

ezzel a módszerrel előbb megállapíthatjuk, hogy sikertelen volt-e a keresés, mint ahogy azt a teljes kereséssel tehetnénk. Nem tudjuk megmondani sem sikeres, sem sikertelen vég esetére, mennyi összehasonlítás fog történni.

Ennél a módszernél is alkalmazható a **strázsa**. A módszert nem írnám le még egyszer, csak megjegyzem, hogy ugyanúgy, mint ahogy a strázsás teljes keresésnél tettük, itt is a ciklus után még meg kell vizsgálnunk, hogy a strázsát találtuk-e meg. Ezt a vizsgálatot elhagyhatjuk, ha strázsának a keresett elemnél nagyobbat teszünk meg. Ebben az esetben biztosnak kell lenünk abban, hogy mindenképpen be tudunk illeszteni az adathalmazba a keresett elemnél nagyobb értéket.

1. $K_{n+1} = K'$ $i = 0$
2. $i = i + 1$
3. Ha $K_i < K$ akkor vissza 2-re
4. Ha $K = K_i$ akkor sikeresen vége
5. különben sikertelenül vége

1. $K_{n+1} = K$ $i = 0$
2. $i = i + 1$
3. Ha $K_i < K$ akkor vissza 2-re
4. Ha $K = K_i$ és $i < n$ akkor sikeresen vége
5. különben sikertelenül vége

Az első egy olyan strázsás keresés, ahol a strázsa nagyobb, mint a keresett elem (K' > K). A második megoldásnál a strázsa maga az elem, de ekkor a ciklus végén plusz egy ellenőrzést el kell végeznünk.

Bináris keresés (Logaritmikus keresés)

Az algoritmus rendezett adathalmazon keres, és szükséges feltétel még, hogy az elemeket közvetlenül el tudjuk érni, mivel nem lépkedünk, hanem ugrálunk az adathalmaz elemei között.

A keresés során két csoportra bontjuk a sorozatunkat. Kivesszük a középső elemet (ez fogja két részre osztani az adott adathalmazt), és megvizsgáljuk, hogy a keresett elemet választottuk-e ki. Ha igen, akkor a keresésnek sikeresen vége, ha nem, akkor a keresett elem vagy kisebb nála, vagy nagyobb. Ezzel meghatároztuk, hogy melyik halmazban lehet az elem. Azaz az elemek felét ki is „dobtuk", biztosan tudva, hogy abban a részben nincs a keresett elem. Ezzel az eljárással rendre megfelezzük az adathalmazt, (n–1)/2. Ténylegesen nem kell megfeleznünk, csak fölveszünk egy alsó (lower - l) és egy fölső (high - h) indexet, és ezen indexek értékeit változtatjuk. Első lépésben az l = 1, h = n. Ha a keresett elem kisebb a középsőnél a h = középső indexe–1 lesz, ha nagyobb nála, az l = középső indexe+1 lesz. Akkor mondhatjuk, hogy sikertelen a keresés, ha a halmaz elemszáma 1 és az az elem sem egyezik meg a keresettel. Máshogy megfogalmazva akkor, ha l > h.

Nézzünk meg egy 1023 elemű rendezett adathalmazt. Vegyük a középső elemét, ezzel két egyenlő részre osztottuk: 1023, (1023–1)/2 = 511, (511–1) = 255, (255–1)/2 = 127, (127–1)/2 = 63, (63–1)/2 = 31, (31–1)/2 = 15, (15–1)/2 = 7, (7–1)/2 = 3, (3–1)/2 = 1. Tíz lépés alatt eljutottunk az egyelemű halmazig. Tehát maximum ennyi lépés alatt találunk meg egy elemet egy 1023 elemű adathalmazban, és pontosan ennyi lépés szükséges ahhoz, hogy azt mondhassuk, hogy a keresett elem nincs benne. Így alkalmazva a bináris keresést, a maximális összehasonlítások száma $\log_2 n$, $2^{10} = 1024$. Most már érthető, hogy miért hívják bináris vagy logaritmikus keresésnek.

Fölmerülhet valakiben, hogy megtehetjük-e, hogy az adatcsoportból nem a középső elemet választjuk ki. Azaz az adathalmazt nem felezzük, hanem például harmadoljuk. Ekkor is két részre osztjuk a sorozatot, de a két rész közel sem egyenlő nagyságú. Ha a keresett elem nincs meg, és a kisebbik részben található, akkor gyorsabban juthatunk eredményre, de néha ekkor is a nagyobbik csoportot kell átvizsgálnunk, ezzel lassítva a keresést. A bináris keresés akkor működik kiegyensúlyozottan, általában akkor a leggyorsabb, ha a sorozatot felezzük.

1. $l = 1$, $u = n$
2. Ha $l > u$ akkor sikertelenül vége
3. $i = (l + u) / 2$
4. Ha $K = K_i$ akkor sikeresen vége
5. Ha $K < K_i$ akkor $u = i - 1$
6. különben $l = i + 1$
7. vissza 2-re

Függvény: Keresés (K[], bal, jobb, K)

1. Ha bal > jobb akkor visszaad:–1
2. $i = (bal + jobb) / 2$
3. Ha $K = K_i$ akkor visszaad: i
4. Ha $K < K_i$ akkor
   visszaad Keresés(K[], bal, i - 1, K)
5. különben
   visszaad Keresés(K[], i + 1, jobb, K)

Az első algoritmus a bináris keresés iteratív (ismétléses) megvalósítása. Néha szokták használni a rekurzív (~újrahívó) változatot is, de ennek a vizsgán kívül nem sok értelme van. A rekurzív megoldás ebben az esetben nem átláthatóbb, mint az iteratív társa. A rekurzív algoritmus, ha nem találta meg az elemet, önmagát hívja meg más paraméterekkel, végül vagy a keresett elem indexét vagy –1-et ad vissza.

Rendezések

A rendezések alapja az, hogy az elemeket összehasonlítjuk egymással, és ha szükséges megcseréljük őket. Első lépésben nem tudjuk az összes elem helyét megkerestetni, ezért minden rendezésre jellemző, hogy két ciklust, egy belső és egy külső ciklust tartalmaz. Ha úgy tetszik, a két ciklussal az adathalmazt két dinamikusan változó részhalmazra bontjuk.

Egy rendezés nagy vonalakban úgy néz ki, hogy a külső ciklusban általában értékadások történnek, és valahol még szerepel benne a belső ciklus. A külső ciklus kilépési feltétele határozza meg, hogy a rendezés mikor fejeződik be. A belső ciklusban az elemeken vagy még inkább az elemek egy részhalmazán lépkedünk végig. Ez a rész tartalmazza a tagok összehasonlítását. Az elemek cseréje a rendezési módszertől függően vagy a belső vagy a külső ciklusban szerepel. Egy rendezés akkor dolgozik gyorsabban, ha ugyanazon adathalmaz rendezéséhez kevesebb összehasonlítást és cserét kell elvégeznie.

A rendezések leírásakor azt vettem alapul, hogy egyszerű típusú adatokat növekvő sorrendbe rendezzünk.

Közvetlen kiválasztásos rendezés

Ez a rendezés a minimum/maximum kereséssel van kapcsolatban. A külső ciklus azt adja meg, hogy az adathalmaz melyik helyére keressük a megfelelő elemet. A belső ciklusban megkeressük a legkisebb elemet, és amikor befejeződik berakjuk a megfelelő helyre. Ennél a rendezésnél az elemek cseréje a külső ciklusban van. A külső ciklus n-1-szer ismétlődik, ti. az utolsó helyre már nem kell megkeresni a hozzá tartozó tagot. A belső ciklus kezdőértéke a külső ciklus aktuális értéke, így a rendezett részen már nem keresünk. Ha csökkenő sorrendbe rendezünk, akkor maximum értéket kell keresnünk. A rendezést még úgy is módosíthatjuk, hogy visszafelé rendezzük az adatokat. Első lépésben nem a legelső elemet hanem a legutolsót határozzuk meg. Ekkor a külső ciklus visszafelé fog haladni, és e szerint kell megváltoztatni a belső ciklus szerkezetét is. Ha így változtatjuk meg, akkor növekvő rendezéshez maximumot kell keresnünk. Első ránézésre azt mondanánk, hogy ennél a rendezésnél n-1 csere történik. Ha jobban megnézzük, akkor észrevesszük, hogy a belső ciklusban - a minimum keresésekor - igen sok csere történhet. Ez a rendezés egyenletes, egyenletesen rossz teljesítményt ad.

1. $i = 1$
2. $min = i, \quad j = i$
3. $j = j + 1$
4. *Ha* $A_j < A_{min}$ *akkor* $min = j$
5. *Ha* $j < n$ *akkor vissza 3-ra*
6. $t = A_i \quad A_i = A_{min} \quad A_{min} = t$
7. $i = i + 1$
8. *Ha* $i < n$ *akkor vissza 2-re*
9. *vége*

A belső ciklus (3-5) a külső ciklus értéke utáni indextől indul, elsőnek fölveszi a külső ciklus értékét ($j = i$), de rögtön első lépésként növeljük is egyel. Mindkét ciklus hátultesztelő, de a belső a ciklusváltozóját elöl változtatja meg, míg a külső a ciklus végén. A belső ciklus lefutása után (6) a legkisebb elemet a helyére - az i. helyre - rakjuk.

Közvetlen beszúrásos rendezés

Ezt a rendezést az elnevezés hasonlósága miatt néha összekeverik az előzővel. Pedig pont fordítottan működik. Míg az előző rendezésnél a helyhez kerestük az elemet, addig a közvetlen beszúrásos rendezésnél az elemhez keressük a helyet.

A rendezés a lineáris kereséssel van kapcsolatban. Sorra vesszük az elemeket, és a már rendezett részhalmazban megkeressük a helyét. A külső ciklus 2-től n-ig halad. (Az első elem helye már adott.) A belső ciklusban a rendezett adathalmaz elemein végighaladva minden olyan tagot átlépünk, ami nagyobb nála. A belső ciklus meghatározza az elem helyét. A rendezés alatt meg kell oldani, hogy a fennmaradó elemeket eltoljuk, hiszen helyet kell készítenünk a beszúrandó elemnek. Ekkor a belső ciklus visszafelé lépked, és az elemeket sorra eltoljuk. Ha az adathalmazba egy elem beszúrása megoldható a tagok rakosgatása nélkül, akkor sima strázsás lineáris keresést kell végeznünk a már rendezett adathalmaz elejétől. Egyébként akkor is lineáris keresést végezünk, amikor pakolgatjuk az elemeket.

Ha nem tudjuk megoldani az elem beszúrását, akkor a belső ciklusban az összes mögötte állót el kell tolnunk. Ilyen esetben a belső ciklus visszafelé halad, ekkor (n – 1)-nél sokkal több cserét kell elvégeznünk.

1. $i = 1$
2. $i = i + 1$
3. $j = i - 1, \quad x = A_i$
4. *Ha* $A_j \leq x$ *akkor menj 8-ra*
5. $A_{j+1} = A_j$
6. $j = j - 1$
7. *Ha* $j > 0$ *akkor vissza 4-re*
8. $A_{j+1} = x$
9. *Ha* $i \neq n$ *akkor vissza 2-re*
10. *vége*

Buborékrendezés

Az előző két rendezésnél láthattuk, hogy egy-egy elem rendezésekor sok mellékes csere történik. A közvetlen kiválasztásosnál a legkisebb elemet kell megjegyeznünk, a közvetlen beszúrásosnál, ha nem tudjuk máshogy megoldani az elem beszúrását, a mögötte lévőket kell átpakolnunk. A buborékrendezés ezen próbál változtatni.

A rendezést a minimum kiválasztással lehet kapcsolatba hozni, bár nem úgy, ahogy a közvetlen kiválasztásos rendezést. Vesszük az első elemet, és az mondjuk, hogy ez a legnagyobb. Ez után sorban végiglépkedünk az adathalmaz elemein, és ha ennél az elemnél kisebbet találunk, akkor megcseréljük őket, ha nagyobbat, akkor ettől kezdve az lesz a legnagyobb elem. Így járva el, a legnagyobb elem a ciklus végén a helyére kerül, mint egy buborék felszáll a sorozat „tetejére". Amikor egy-egy cserét elvégzünk, az adott elem mindig közelebb kerül a helyéhez. Nézzük a következő sorozatot:

1 7 2 3 4 9 5 6 8

Első lépésben az 1-es lesz a legnagyobb elem, összehasonlítjuk a következővel, és a 7-es lesz a legnagyobb. Ezek után az összehasonlításokat az adathalmazban csere is követi egészen a 9-esig.

1 2 3 4 7 9 5 6 8

Ekkor a legnagyobb értéke a 9 lesz, az összehasonlítás és csere ezzel fog tovább folytatódni.

1 2 3 4 7 5 6 8 9

A ciklus végére a 7-es közelebb van a helyéhez, és a legnagyobb elem a helyére került. A ciklust minden elemére eljátszva a rendezett adathalmazt kapjuk.

Az algoritmus futási ideje javítható, ha a már rendezett elemekre nem hajtjuk végre az összehasonlítást. Ekkor elegendő, ha a belső ciklus (n–k+1)-ig (k - a külső ciklus aktuális értéke) hajtódik végre. Vagy megoldható úgy is, hogy megfordítjuk a belső ciklust, és n-től halad a külső ciklus aktuális értékéig. Ekkor a buborékok „lefelé szállnak", növekvő rendezést alkalmazva minimumot kell keresnünk.

Ha megnézzük a sorozatunkat, látható, hogy a következő ciklusban már rendezett lesz. A rendezés hatékonyságán azzal is javíthatunk, ha alkalmazunk egy *voltcsere* változót. Ezt a változót a belső ciklus végrehajtása előtt egy meghatározott értéket kap, és amikor cserét végzünk, megváltoztatjuk. A külső ciklus kilépési feltétele pedig az, amikor a *voltcsere* értéke az eredeti maradt. A fönti sorozatot rendezve a belső ciklus háromszor fog lefutni.

A buborékrendezés - bár más alapokon működik, mint az előző kettő - mégsem dolgozik igazán gyorsan. Néha kifejezetten lassú. Köszönheti ezt annak, hogy a rendezés alatt egy-egy elem csak közelebb kerül a helyéhez, és nem a valódi helyét foglalja el. (Kivéve az aktuális legnagyobb/legkisebb elemet.) A módszer ezzel szaporítja a felesleges cserék számát. Az is igaz, hogy ha már rendezett sorozatot kell rendezni, akkor a buborékrendezés (ha alkalmaztuk a *voltcsere* változót) minden más rendezéshez képest előbb végez a sorozat ellenőrzésével.

1. *[Ciklus i = (n-1), …, 1-re] hajtsuk végre 2-t*
2. *[Ciklus j = 1, …, i-re] hajtsuk végre 3-t*
3. *Ha* $A_j > A_{j+1}$ *akkor*
   cseréljük meg $A_j \Leftrightarrow A_{j+1}$

1. *voltcsere = 1, i = 1*
2. *Amíg voltcsere = 1 és* $i \le n$ *hajtsuk végre 3, 4-et*
3. *i = i + 1, voltcsere = 0*
4. *j = n, n-1, …, i-re hajtsuk végre 5-t*
5. *Ha* $A_j < A_{j-1}$ *akkor*
   $t = A_j$, $A_j = A_{j-1}$, $A_{j-1} = t$, *voltcsere = 1*

Az első algoritmuson látszik igazán, hogy ezt a rendezést az egyszerű megvalósíthatósága miatt szokták alkalmazni, és kisebb adathalmazok esetén elhanyagolható, hogy lassabban rendez, mint más bonyolultabb rendezések. Nagyon egyszerű a szerkezete, csak két ciklusból és egy ellenőrzésből áll. A külső ciklus csökkenő, ez adja meg a belső ciklus végét, így a belső ciklus minden végrehajtásakor rövidebb lesz, azért, hogy a már helyére rakott elemeket a további ciklusokban ne ellenőrizzük. A belső ciklusban végiglépkedünk az elemeken, és páronként összehasonlítjuk őket. Úgy vesszük, hogy az aktuális elem a legnagyobb, ha ez teljesül, akkor cserélünk (növekvőbe rendezünk), ha nem, akkor a következő ellenőrzéskor, már a rákövetkező elemet tekintjük a legnagyobbnak. A sorozat elemeit 1,…, n-ig indexeltük, ha ettől eltérünk, akkor némileg módosítanunk kell a ciklusváltozók értékeit. A második algoritmus a buborékrendezés voltcserés megvalósítása. Most a belső ciklus halad visszafelé. Növekvő rendezéshez akkor cserélünk, ha az aktuális elem kisebb, mint az azt megelőző. Természetesen a *voltcsere* változót alkalmazhatjuk az első algoritmushoz is, és kivehetjük a másodikból, ettől még ugyanolyan jól fog működni mindkettő.

Gyorsrendezés (Quick sort)

Nézzük a következő természetes permutációt:

0 1 2 3 4 5 6 | 7 8 9

Osszuk ezt két részre! Ha az egyik részhalmazban megváltoztatjuk az elemek sorrendjét, attól a másik részhalmaz elemei változatlanok maradnak. Ellenben, ha a két részhalmaz között végzünk cserét, akkor ugyanúgy megváltozik mindkét részhalmaz.

0 3 9 1 4 5 6 | 7 8 2

A gyorsrendezés ezt használja ki, és a rendezni kívánt sorozat két részhalmaza közötti eltéréseket hozza rendbe. A belső ciklus lefutásakor a két részhalmaz elemei általában továbbra is rendezetlenek, de rajtuk is elvégezve a rendezést végül - eljutva az egyelemű halmazokig - a teljes sorozat rendezett lesz. Gyorsaságát két dolognak köszönheti. Jó esetben a rendezés külső ciklusa $\log_2$ n-szer hajtódik végre, és egy csere elvégzésével két elem kerül a neki megfelelő halmazba. A külső ciklus csak akkor hajtódik végre $\log_2$ n-szer, ha a kitüntetett elem rendre mindig a sorozat középső eleme, ez pedig csak speciális esetben fordul elő, ezért a külső ciklus ennél némiképp többször fut le, legrosszabb esetben n-szer.

Az algoritmus megvalósításakor a problémát az jelenti, hogy nem tudjuk, hogy hol van a két részhalmaz határa. (Nem tudjuk, hogy hol van a középső elem, hiszen a sorozat rendezetlen.) Ezért általában úgy valósítják meg a gyorsrendezést, hogy kivesznek egy elemet az adathalmazból, és ehhez hasonlítják a többi elemet. Azaz megkeressük a kiválasztott elem helyét. (Ténylegesen ezt az elemet nem kell kivennünk a sorozatból! Ha az elem rendre a sorozat első vagy utolsó eleme, akkor el is tudjuk szigetelni a többitől, ezért a ciklus végére a helyére kerül.) Nézzünk egy példát! A kitüntetett elemünk rendre az adathalmaz első eleme lesz. Elindulunk a sorozat bal és jobb oldaláról, és sorra összehasonlítjuk őket a kiválasztott elemmel.

7 4 8 0 5 6 1 9 2 3
→ ←

Addig keresünk a bal oldalon, amíg nem találunk egy oda nem illőt. Egy olyan elemet, ami nagyobb, mint a kiválasztott elem. A sorozat jobb oldalán addig keresünk, amíg nem találunk egy kisebb elemet. Ha az indexek nem lépték át egymást, akkor megcseréljük az elemeket. Ha a jobb oldali index átlépte a baloldalit, vagy ugyanarra az elemre mutat, akkor megtaláltuk a két halmaz határát, megtaláltuk a kiválasztott elem helyét. A kiválasztott elem helyét a jobb oldali index adja meg, ezzel az elemmel kell megcserélni. (Mivel a kitüntetett elem a bal oldalon van, és a jobb index csúszott át a bal oldalra, a bal oldali pedig a jobb oldali részhalmaz egy elemére mutat.) A ciklus végére a kiválasztott elem a helyére kerül, meghatározzuk az adathalmaz két részhalmazát, és a részhalmaz elemei között nincs keveredés. Ezt minden részhalmazra elvégezve a rendezett sorozatot kapjuk.

7 4 3 0 5 6 1 | 2 | 9 8
← →
2 4 3 0 5 6 1 | 7 | 9 8

Ennél a megvalósításnál nem tudjuk előre, hogy a két részhalmaznak hány eleme van, mivel az indexhatárt a kitüntetett elem határozza meg, és ennek a helyét keressük. A keresés közben a két részhalmaz elemeit, ha szükséges megcseréljük. A példában az első ciklust elvégezve az adathalmazt egy 7 és egy 2 elemű részre osztjuk. (A kiválasztott elemet egyik részhalmazba sem kell betennünk, mivel már a helyére raktuk.) A gyorsrendezés ezen megvalósításának ez a gyenge pontja. Ha a kiválasztott elem rendre mindig az adathalmaz első vagy utolsó eleme, akkor az n elemű adathalmazt *1* és (*n–1*) elemű részre osztja. Ekkor az algoritmus a közvetlen beszúrásosnak megfelelő sebességgel dolgozik.

Az algoritmusnak ezt a hibáját - hogy kiegyensúlyozatlanul működik, azaz már rendezett sorozatot (Ebben az esetben a fordított sorrendű és az azonos elemű sorozatok is ide értendők.) kifejezetten lassan rendez - ki lehet küszöbölni azzal, hogy az adathalmazból nem az első/utolsó hanem a középső elemet választjuk ki. Az elv nem változik, ekkor sem csinálunk mást, mint az előbb, csak a középső elem a kitüntetett, és ennek megfelelően kell az algoritmust megváltoztatni. Ekkor sem tudjuk, hogy hol lesz a két halmaz határa, de már rendezett sorozat rendezésekor nem fordul elő az, hogy az adathalmazt rendre *1* és (*n–1*) elemű részre osztjuk. A Borland cég ezt a gyorsrendezést használja. A fejlesztőkörnyezeteiben (Turbo Pascal, Delphi, Borland C, stb.) a példaprogramok között ott szokott lenni.

A gyorsrendezés előbb tárgyalt megoldásainál úgy tekintjük, hogy ismerjük az elemet, de nem tudjuk a helyét. Ha úgy tetszik, erre az elemre nézve közvetlen beszúrásos rendezést végzünk, csak két indexet használunk, és közben cserélgetjük az elemeket is. A másik módszer az lehetne, amikor a helyhez (pl. az adathalmaz középső helyéhez) keressük a megfelelő értéket. Ez a gyorsrendezés matematikai hátterének egy másik szemlélet szerinti megvalósítása. Ennél a megvalósításnál nem fordul elő, hogy egy-egy sorozatot aránytalanul hosszú ideig rendezne. Összehasonlításképpen a Hoare[7] gyorsrendezéséhez képeset rendezetlen adathalmazt kb. 1,6-szor lassabban, míg rendezett adathalmazt kb. 250-szer gyorsabban rendez. A Borland cég gyorsrendezése rendezett sorozatokkal szintén gyorsan végez, és a teljesítményében kisebb visszaesés figyelhető meg. (A gyorsrendezésnek ezen kívül még számos más változata is létezik.)

[7] *Charles Anthony Richard Hoare 1962-ben publikálta rendezési módszerét.*

Eljárás: Rendezés (A[], bal, jobb)

1. *Ha bal ≥ jobb akkor vége*
2. *határ = A_{jobb} , i = bal, j = jobb - 1*
3. *Amíg A_i < határ hajtsuk végre 5-öt*
4. *i = i + 1*
5. *Amíg A_j > határ hajtsuk végre 7-et*
6. *j = j – 1*
7. *Ha i ≥ j akkor menj 10-re*
8. *különben cseréljük meg $A_i \Leftrightarrow A_j$*
9. *vissza 3-ra.*
10. *cseréljük meg $A_i \Leftrightarrow A_{jobb}$*
11. *Rendezés (A[], bal, i - 1)*
12. *Rendezés (A[], i + 1, jobb)*

Hoare gyorsrendezés. A második pontban jelöljük ki a kiválasztott elemet, ami most a részsorozat jobb szélső eleme, ezután indítunk egy-egy keresést i és j indexekkel jobbról is és balról is, a kiválasztott elemnél nagyobb illetve kisebb elemeket keressük. A ciklus akkor fejeződik be, ha megtaláltuk a kiválasztott elem helyét, azaz a két index átlépte egymást, vagy ugyanarra a helyre mutat. Ekkor a kiválasztott elemet a 10. pontban a helyére rakjuk. Azért az i index adja meg a helyét, mert ezt indítottuk a bal oldalról, és a ciklus végére ez az index mutat a jobboldali részhalmaz első helyére. A következő két lépésben a sorozat két részhalmazán is elvégezzük a gyorsrendezést, a kiválasztott elem a továbbiakban már nem vesz részt a rendezésben.

Eljárás: Rendezés (A[], bal, jobb)

1. *i = bal, j = jobb, középső = $A_{(bal + jobb)/2}$*
2. *Amíg i ≤ j hajtsuk végre 3, 5, 7-t.*
3. *Amig A_i < középső hajtsuk végre 4-t*
4. *i = i + 1*
5. *Amig A_j > középső hajtsuk végre 6-t*
6. *j = j – 1*
7. *Ha i < j akkor*
   cseréljük meg $A_i \Leftrightarrow A_j$, j = j – 1, i = i + 1
8. *Ha bal < j akkor Rendezés (A[], bal, j)*
9. *Ha i < jobb akkor Rendezés (A[], i, jobb)*

Eljárás: Rendezés (A[], bal, jobb)

1. *Ha bal ≥ jobb akkor vége*
2. *i = bal, j = jobb, közép = (i + j) / 2*
3. *Amíg A_i < $A_{közép}$ hajtsuk végre 4-et*
4. *i = i + 1*
5. *Amíg A_j > $A_{közép}$ hajtsuk végre 6-ot*
6. *j = j – 1*
7. *Ha i = j menj 14-re*
8. *Ha $A_i \neq A_j$ akkor hajtsuk végre 9, 10, 11-t különben 12-t*
9. *Cseréljük meg $A_i \Leftrightarrow A_j$*
10. *Ha i = közép akkor i = bal*
11. *Ha j = közép akkor j = jobb*
12. *Ha i = közép akkor i = i – 1 különben j = j + 1*
13. *vissza 3-ra*
14. *Rendezés (A[], bal, közép-1)*
15. *Rendezés (A[], közép+1, jobb)*

Az első algoritmusnál az elemhez keressük a helyet, és a középső elemet választjuk ki. Az algoritmus utolsó két sora is fontos, mivel nem igazi öngyilkos rekurzív algoritmusról van szó, mert ha már nem kell, meg sem hívja önmagát. A részsorozat rendezését nem akkor fejezi be, amikor a kiválasztott elemet áthelyeztük, hanem akkor, amikor a két index átlépi egymást. A két halmaz határát a következő függvényhíváshoz nem a kiválasztott elem határozza meg, hanem a két index (i és j) aktuális állása. (Az előző rendezésnél is így volt, de ott tudtuk, hogy a kiválasztott elem a ciklus lefutása után a helyére kerül. Ennél az algoritmusnál nem feltétlenül.) Ezért van a 2. pontban egyenlőség is megengedve, ha ezt nem tesszük, akkor végtelen ciklusba szalad az algoritmus. A kiválasztott elemet nem tudjuk kiemelni a sorozatból, ezért előfordulhat, hogy nem pontosan a helyére kerül, de mindenképpen abba a halmazba, amelyikbe való.

A második algoritmusnál a helyhez keressük az elemet. Ezt az algoritmust valószínű, hogy nem fogják számon kérni, most találtam ki a tavaszi szünetben. Én még nem találkoztam ezzel a változattal, ettől persze még lehetséges, hogy a spanyolviaszt találtam föl. Szóval nem kell energiát fektetned a megtanulásába. Az algoritmus a fönt leírtak szerint működik, nem az elemhez keresi a helyet, hanem a helyhez az elemet. A 8. pont különben ága és ezzel együtt a 12. pont elhagyható, ha tudjuk, hogy egy érték maximum kétszer ismétlődhet a sorozatban. Ha többszöri ismétlődést is megengedünk, akkor ez a különben rész akadályozza meg, hogy végtelen ciklus jöjjön létre.

Informatika alapjai ZH - 2001.12.04.

A feladatsort Farkas János készítette. Ez a feladatsor és az ezt következő feladatsor jó viszonyítási alapot ad, hogy a vizsgán mire is lehet számítani. Aki könnyű szerrel oldja meg a két kérdéssort, az bátran vághat neki a vizsgának. De az önálló megoldásuk segíthet az ismeretek rendszerezésében is! ☺

1. Mennyi az (a – b) kivonás eredménye 19-es számrendszerben, ha
   $a = 10011_{38}$
   $b = AB71C_{14}$?

2. Ábrázold a (14342012 / –1623) racionális számot lebegőpontosan, 48 biten, 8 bit karakterisztikával, 8-as alappal! *Az osztás eredményét elég öt tizedes jegyig figyelembe venni!*

3. Algoritmus:
   Kérj be egy egész számot az „A” változóba! Ha az érték negatív, jelezd, hogy hibás érték, és kérd be újra a számot! Ha az érték nem negatív, akkor kérj be „A” db valós számot, és számold ki azok átlagát, majd az eredményt írasd ki! Használd a már létező **bekér**(*típus*) és a **kiír**(*változó*) szubrutinokat, ahol a bekér() a billentyűzetről bekér egy adott típusú értéket, a kiír() pedig kiírja a képernyőre az adott változó értékét!

4. Adott egy processzor, mely egy 256 bájt méretű tárrészen dolgozik, egy regisztere van, és rendelkezik egy rendszerveremmel. A memória címzésére egy bájtot használ, a regisztert pedig REG-nek hívják. A processzor minden utasítása és az operandussal rendelkező utasításainak operandus hossza is egy bájt. A processzor utasításkészletéből az alábbi részletet ismertetjük:

   `[00h]` A program vége.
   `[17h]` Az operandusként megadott tárhelyre rakja a REG értékét.
   `[18h]` A REG-be rakja az operandusként megadott című tárhelyen lévő értéket.
   `[A2h]` PUSH OP, a rendszerverembe rakja az operandusként megadott értéket.
   `[A3h]` PUSH REG, a rendszerverembe rakja a REG értékét.
   `[A4h]` OUT OP, kiírja a képernyőre az operandus értékét.
   `[BAh]` POP REG, a REG-be rakja a rendszerverem egy elemét, és eltávolítja azt a veremből.
   `[BBh]` Kivesz egy elemet a veremből és az 1-es komplemensét teszi vissza.
   `[CCh]` JMP OP, feltétel nélküli vezérlésátadás, a program futása a megadott memóriacímen folytatódik.
   `[FEh]` OUT REG, kiírja a képernyőre a REG értékét (fixpontos számként, decimálisan).
   `[FFh]` OUT [OP], kiírja a képernyőre az operandusként megadott címen lévő értéket (fixpontos számként, decimálisan).

   Memóriarészlet:

```
<A2>  0001 1000        0001 1000  <A3>
<A4>  1100 1100        1010 0011  <A5>
<A6>  1011 1011        1100 1100  <A7>
<A8>  1010 1101        0010 1101  <A9>
<AA>  1111 1101        1100 0111  <AB>
<AC>  1101 0010        1011 1010  <AD>
<AE>  0001 0111        1011 0001  <AF>
<B0>  1111 1111        1111 0001  <B1>
<B2>  0000 0000        0010 1111  <B3>

<CA>  0101 1111        1101 0011  <CB>
<CC>  0100 1110        0001 1111  <CD>
<CE>  1010 1010        0011 1101  <CF>
```

   Mi lesz a futás eredménye a képernyőn, ha a futtatást az A4h címen kezdjük?

A feladatsor megoldása

1. $CF78H_{19}$
2. |1|10000101|010001010000100.10111010110000010010101 0 (A törtpontot csak jelzésként tettem ki. Te ne használd! Az adott tárrész értéke hexában: C2A284BAC12Ah)
3.
   1. *A =* ***bekér****(egész szám)*
   2. *Ha A < 0 akkor*
      kiír*(„Nem jó érték!”), vissza 1-re.*
   3. *db = 0, átlag = 0*
   4. *Ha A = db akkor*
      menj 8-ra.
   5. *szám =* ***bekér****(valós szám)*
   6. *átlag = átlag + szám, db = db +1*
   7. *vissza 4-re*
   8. *Ha db = 0 akkor*
      kiír(„Nem volt adatbevitel.”), vége.
      különben
      átlag = átlag/db, kiír(átlag), vége.
4.
   1. `JMP <A3>`
   2. `regiszter ⇐ <CC> = 0100 1110`
   3. `verem ⇐ regiszter = 0100 1110`
   4. `verem ⇒ 0100 1110 ⇒ 1011 0001 ⇒ verem`
   5. `JMP <AD>`
   6. `regiszter ⇐ verem = 1011 0001`
   7. `<B1> ⇐ regiszter = 1011 0001 (B1h!)`
   8. `Fixpontos számként kiírja a <B1> címen tárolt értéket: -79.`

Informatika alapjai vizsgafeladat - 1998.12.16.

*A feladatsort a **http://dragon.klte.hu/~gkusper/** honlapon találtam. A weblapon található megoldásokat kiegészítettem ott, ahol szerintem más a helyes válasz. Az utolsó két kérdés volt a beugró, amiket tökéletesen kell megoldani!*

1. Ha egy bináris tört számot tizenhatos számrendszerbe akarunk konvertálni, akkor
   a) a szám elejéről kezdve hármas csoportokat képezünk (ha kell az elején nullákkal kiegészítjük) és egy-egy ilyen csoportból alkotunk egy-egy tizenhatos számrendszerbeli számjegyet.
   b) a szám végéről kezdve négyes csoportokat képezünk (ha kell az elején és a végén nullákkal kiegészítjük) és egy-egy ilyen csoportból alkotunk egy-egy tizenhatos számrendszerbeli számjegyet.
   c) a bináris ponttól jobbra és balra indulva hármas csoportokat képezünk (ha kell az elején és a végén nullákkal kiegészítjük) és egy-egy ilyen csoportból alkotunk egy-egy tizenhatos számrendszerbeli számjegyet.
   d) a bináris ponttól jobbra és balra indulva négyes csoportokat képezünk (ha kell az elején és a végén nullákkal kiegészítjük) és egy-egy ilyen csoportból alkotunk egy-egy tizenhatos számrendszerbeli számjegyet.

2. A bájt
   a) 8 bitből áll.
   b) 8 bitből és a paritásbitből áll.
   c) 256 féle információt jeleníthet meg.
   d) 255 féle információt jeleníthet meg.
   e) tartalma két oktális számjeggyel írható le.

3. Hány KB-nak felel meg 7 GB?

4. Egy kettes számrendszerbeli szám kettes komplemensét úgy kapjuk, hogy
   a) minden helyi értéken a legnagyobb számjegyre (1) kiegészítő számjegyet veszünk, majd a kapott számhoz hozzáadunk egyet.
   b) minden helyi értéken az inverz (ellentett) kódot képezzük.
   c) a legalacsonyabb helyi értéktől kezdve, balra haladva az első 1-essel bezárólag meghagyjuk a jegyeket, majd a továbbiakat ellentettjükre változtatjuk.

5. Az alábbi csoportok közül melyek azok, amelyek csak INPUT perifériát tartalmaznak:
   a) egér, nyomtató, CD.
   b) egér, mágnesszalag, billentyűzet.
   c) billentyűzet, ROM, egér.

6. Az alábbi tízes számrendszerbeli lebegőpontos számok közül melyik ábrázolható pontosan, mint kettesre normált lebegőpontos szám?
   a) 2.1
   b) 2.75
   c) 2.3

7. Mennyi lesz a decimális értéke annak a számnak, melynek egybájtos előjeles egész ábrázolásának hexadecimális alakja AB.
   a) –85
   b) 85
   c) –171
   d) 171
   e) –43
   f) 43

8. Négy bájt egymás melletti tartalma hexadecimálisan 480B720D. Milyen a szám előjele?
   a) fixpontosan: negatív.
   b) fixpontosan: pozitív.
   c) tizenhatosra normált lebegőpontos: negatív.
   d) tizenhatosra normált lebegőpontos: pozitív.
   e) pakolt decimális: pozitív.
   f) zónázott decimális: negatív.

9. Két szöveg összehasonlításánál
   a) a hosszabb a nagyobb.
   b) a rövidebb a nagyobb.
   c) ha a legelső karakterek kódja egyenlő, akkor a két szöveg egyenlő.
   d) ha a legutolsó karakterek kódja egyenlő, akkor a két szöveg egyenlő.

10. Lebegőpontosan m*a^k alakú számokat ábrázolunk, ahol
   a) m és k egész.
   b) k és a egész és rögzített.
   c) a valós egész vagy tört és rögzített.
   d) m a mantissza, k a karakterisztika, a az alap.

11. A gyorsított szekvenciális keresés
   a) csak rendezett értékhalmazon alkalmazható.
   b) csak rendezetlen értékhalmazon alkalmazható.
   c) esetén egy adott érték megtalálásához minden értéket meg kell vizsgálnunk.
   d) esetén ahhoz, hogy azt mondhassuk, hogy egy adott érték nincs benne az értékhalmazban, minden elemet végig kell vizsgálni.
   e) úgy indul, hogy kiválaszt egy tetszőleges (általában a középső) elemet.

12. Regiszteres indirekt címzés esetén
   a) az operandust az utasításban megadott regiszterben találhatjuk.
   b) az operandus címét az utasításban megadott regiszterben találjuk.
   c) az operandus indexét az utasításban megadott regiszterben találjuk.
   d) egy báziscímhez viszonyított eltolási értéket találunk az utasításban megadott regiszterben.

13. A CPU
   a) a Concurrent Processing Unit rövidítése.
   b) a számítógép erőforrásait osztja meg a futó programok között.
   c) tartalmaz regisztereket.
   d) a megszakítások jelzésére szolgál.

14. Az utasításszámláló
   a) a memória egy speciális rekesze.
   b) egy regiszter.
   c) egy regiszter, amely a bekapcsolás óta a processzor által végrehajtott utasítások darabszámát tartalmazza.
   d) egy regiszter, amely a bekapcsolás óta a processzor által sikeresen végrehajtott utasítások darabszámát tartalmazza.
   e) a ROM része.

15. A tömb és a lista (egyirányú lista) adatszerkezetek között
   a) az a hasonlóság, hogy mindkettőben bármely elem elérése azonos időt vesz igénybe.
   b) az a különbség, hogy a tömb mérete fix, a lista mérete változtatható.
   c) az a különbség, hogy a lista két tömbnek felel meg.

16. A protokoll
   a) a csomópont része.
   b) az algoritmus egy leírási, megadási módszere.
   c) az algoritmuselméletben a nem definiált helyzetek elkerülésének egyik eszköze.
   d) az adatátviteli csatorna szinonimája.

17. Írja le pontokba szedett magyar „utasításokkal” (pszeudokód) a gyorskeresés (bináris keresés) algoritmusát.

18. Ábrázolja a –96.4 decimális értéket 4 bájton lebegőpontosan 16-os alapra normáltan 7 bites karakterisztikával! A bájtok tartalmát hexadecimálisan adja meg!

A feladatsor megoldása

1. d
2. a, c
3. 7 GB = $7*2^{10}$ MB = $7*2^{10}*2^{10}$ KB = 7*1024*1024 KB = 7340032 KB
4. a, c
5. Nincs jó megoldás
6. b
7. a *(Szerintem a jó válasz: e. Akkor lenne az 'a' a jó, ha fixpontos ábrázolásról lenne szó, de itt előjeles egészről beszélnek!)*
8. b *(Szerintem a jó válasz: b, d.)*
9. Nincs jó megoldás
10. d *(Az alap $a \in R$, $a > 1$, ezért nem jó a c válasz. A tárolás szempontjából az $a=e$ lenne a legoptimálisabb, de nehézkes a gépen irracionális számot ábrázolni. A gépi megvalósításnál $a \in R$, vagy még inkább $a \in 2^n$, $n \in N$.)*
11. d
12. b
13. c
14. b
15. b
16. Nincs jó megoldás
17. Adott K keresése [K_1, K_2, … ,K_n] rendezett sorozatban (l - lower, u - upper).
    1. *l = 1, u = n*
    2. *Ha u < l akkor sikertelenül vége,*
       különben i = egészrész((l + u) / 2)
    3. *Ha K==Ki akkor sikeresen vége*
    4. *Ha K < Ki akkor u = i – 1,*
       különben l = i + 1
    5. *Folytatás a 2. pontnál.*
18. C2606666h *(A vizsgán ne felejtsd el valamiféleképpen jelölni az adott számrendszer alapját! Különben nem biztos, hogy az egyébként jó megoldást el is fogadják!)*

A feladatok megoldásai

1.
   1.1. 11000000
   1.2. 01100010
2.
   2.1. 107354.34631_8
   2.2. 19210.54972_{12}
   2.3. 1212012010.011001_3
3.
   3.1. 1025.5
   3.2. 63
   3.3. 181.3125
4.
   4.1. 7F.8h
   4.2. 726.54_8
   4.3. $11011100|00011010|11110111_2$
5.
   5.1. 01101101|10111000
   5.2. 0000000100000000 - A tárolás alapegysége a bájt, ezért a 256-hoz két bájtot kell felhasználni
   5.3. nem lehet ábrázolni
6.
   6.1. 10000000, 00000000 *(Mindkét felírás jó.)*
   6.2. 11011010
   6.3. 11111101
7.
   7.1. 00000000
   7.2. 11111111
   7.3. 11000001
   7.4. 00111111
8.
   8.1. 10000001
   8.2. 10000000
   8.3. 01000000
   8.4. 11000000
9.
   9.1. BF666666h
   9.2. 4861A800h
   9.3. 55E82000h, vagy 4AF41000h. Mindkettő jó, az elsőnél a mantissza értékét eltoltuk, ezzel növelve a pontosságot. (Ami ebben az esetben nem igazán számított.) A másodiknál a karakterisztika tárolási helyét növeltük 7-re. (Ami szintén nem sok vizet zavart.)
10.
   10.1 $+2550608541_{10}$ - Csak akkor, ha tört részt nem tárolunk!
   10.2 -403124893_{10} - Csak akkor, ha tört részt nem tárolunk!
   10.3 $+403124893_{10}$
   10.4 -1744358755_{10}
   10.5 –1.07254209069651551544666290283203e-7
   10.6 Zónázottan nem értelmezhető a tárolt érték, nincsenek zónajelek.
   10.7 Pakolt ábrázolásban többféle számot is jelenthet, attól függően, hogy hol van a tizedesvessző. Az ábrázolt érték lehet –0,9807329; –9,807329; –98,07329; stb. Az biztos, hogy a szám negatív.

Ajánlott irodalom

- **Racskó Péter: Bevezetés a számítástechnikába**
  Egyik nagy előnye a könyvnek, hogy olcsó, kb. 900 Ft, ami ugye mégsem 4000. Talán nem véletlenül, ti. elég sok benne a nyomdahiba, elírás, és néhol a szerkesztő is összekavart egy-két dolgot. De ettől eltekintve érdemes átlapozni, jó pár plusz információt tartalmaz, amiről az előadáson nem esik szó.
- **Angster Erzsébet: Programozás tankönyv I-II**
  Sajnos a könyv a Turbo Pascal-ra épít, de az elméleti részeket igazán hasznos átnézni, akár mindenféle programozói tudás nélkül is. Ezen kívül a második kötetben lévő részek a következő szemeszterben is elő fognak kerülni.
- **Halassy Béla: Ember - Információ - Rendszer**
  Annak ellenére ajánlom a könyvet, hogy ilyen mondatokat is olvashatsz benne: „…a számítástechnika szakma, az informatika hivatás.” Nem sokat kell morfondíroznunk azon, hogy vajon az író informatikus-e. Ezt a nézőpontot követve a kőművesség sem lehet hivatás, csak az építészmérnökség. Pedig valószínű, hogy mindkét oldalon vannak csapnivaló *szakemberek*. De nem ezért a pár félresikerült mondatért érdemes elolvasni a könyvet, csak mint negatívumot (ajánlásképpen) emeltem ki belőle.
- **Halassy Béla: Az adatbázis-tervezés alapjai és titkai**
  A számítástechnika területén ennél jobb könyv nemigen jelent meg. Ami a címével ellentétben nem csak adatbázis-tervezésről szól, hanem szemléletről is. (Lehet, hogy főleg arról?) Csodálkoztam, amikor erről a könyvről kérdeztem egy informatikus-hallgatót, és még hallani sem hallotta Halassy Béla nevét. Pedig ennek a könyvnek alapműnek kellene lenni, az informatikusoknak mindenképpen! A szövege olvasmányos, igaz a leírtakat nehéz követni, meg kell emészteni.