Adatstruktúrák és algoritmusok

Attila Házy, Ferenc Nagy

2011. április 6.

Tartalomjegyzék

1. Bevezetés **7**
1.1. A tárgyról 7
1.2. Alapvető fogalmak, definíciók 10
1.2.1. A számítógép programozásáról 13

2. Az absztrakt adattípus és az algoritmus **23**
2.1. Az absztrakt adat és adattípus 23
2.1.1. A logikai absztrakt adattípus 25
2.1.2. A karakter absztrakt adattípus 33
2.1.3. Az egész szám 36
2.2. Az algoritmus fogalma 38
2.3. Az algoritmus megadási módja: a pszeudokód és a folyamatábra. 43
2.4. Az algoritmus jellemző vonásai (tulajdonságai) 54
2.5. Az algoritmus hatékonysági jellemzői 55
2.6. A növekedési rend fogalma, az ordo szimbolika 58
2.7. A Fibonacci számok 63
2.8. A rekurzív egyenletek és a mester tétel 66

3. Számelméleti algoritmusok **73**
3.1. Alapfogalmak 73
3.2. A legnagyobb közös osztó 75
3.3. A bináris legnagyobb közös osztó algoritmus 78
3.4. Az euklideszi és a kibővített euklideszi algoritmus 80
3.5. A lineáris kongruencia egyenlet 85

3.6. Az RSA-algoritmus 89

4. Elemi dinamikus halmazok 95
4.1. A tömb adatstruktúra 95
4.2. A láncolt lista (mutatós és tömbös implementáció) 103
4.3. A verem és az objektum lefoglalás/felszabadítás 112
4.4. A sor 118

5. Keresés, rendezés egyszerű struktúrában (tömb) 123
5.1. Keresés 123
5.1.1. Lineáris keresés 123
5.1.2. Logaritmikus keresés 126
5.1.3. Hasító táblák 127
5.1.4. Minimum és maximum keresése 142
5.1.5. Kiválasztás lineáris idő alatt 143
5.2. Rendezés 149
5.2.1. A beszúró rendezés 150
5.2.2. Az összefésülő rendezés 152
5.2.3. A Batcher-féle páros-páratlan összefésülés 155
5.2.4. Gyorsrendezés (oszd meg és uralkodj típusú algoritmus) 156
5.2.5. A buborékrendezés 158
5.2.6. A Shell rendezés (rendezés fogyó növekménnyel) 159
5.2.7. A minimum kiválasztásos rendezés 161
5.2.8. Négyzetes rendezés 162
5.2.9. Lineáris idejű rendezők: A leszámláló rendezés 164
5.2.10. A számjegyes rendezés (radix rendezés) 165
5.2.11. Edényrendezés 166
5.2.12. Külső tárak rendezése 167

6. Fák 169
6.1. Gráfelméleti fogalmak, jelölések 169
6.1.1. Gráfok ábrázolási módjai 171
6.2. Bináris kereső fák 173

6.3. Bináris kereső fa inorder bejárása . . . 173
6.4. Bináris kereső fa műveletek . . . 174
6.5. Piros-fekete fák . . . 177
6.5.1. Beszúrás . . . 179
6.6. AVL-fák . . . 180
6.7. 2-3-fák . . . 181
6.8. B-fák . . . 182

7. Gráfelméleti algoritmusok 185
7.1. A szélességi keresés . . . 185
7.2. A mélységi keresés . . . 187
7.3. Minimális feszítőfa . . . 190
7.3.1. Kruskal-algoritmus . . . 192
7.3.2. Prim-algoritmus . . . 197
7.4. Legrövidebb utak . . . 200
7.5. Adott csúcsból induló legrövidebb utak . . . 200
7.5.1. Bellman-Ford algoritmus . . . 203
7.5.2. Dijkstra algoritmusa . . . 206
7.6. Legrövidebb utak minden csúcspárra . . . 209
7.6.1. A Floyd-Warshall-algoritmus . . . 210
7.7. Gráfok tranzitív lezártja . . . 215

8. Dinamikus programozás 217

9. NP-teljesség 221

10. Mellékletek 227
10.1. Az ASCII karakterkészlet . . . 227
10.1.1. Vezérlő jelek . . . 227
10.1.2. Nyomtatható karakterek . . . 229

Irodalomjegyzék 229

1. fejezet

Bevezetés

1.1. A tárgyról

Az adatstruktúrák, algoritmusok tárgy fontos helyet foglal el az informatikában. Egy informatikai alkalmazási rendszer kifejlesztése során három fő szintet szokás megkülönböztetni, amit egy helyjegyfoglalási rendszer példájával illusztrálunk. Mondjuk az InterCity vonatjáratra akarunk helyjegyet venni. Sematikusan az alábbi táblázattal lehetne jellemezni a három fő szintet. A középső szint az, amivel a jelen könyvben foglalkozunk.

A szint neve	A szint jellemzése	A szint fogalomrendszere
Felső szint	Alkalmazói szint	modellalkotás útvonalak, szerelvények, dátumok, helyfoglalások
Középső szint	Modellezési szint, algoritmizálás	file-ok, táblázatok, listák, adatrekordok, stringek, fák
Alsó szint	Csupasz gépszint	objektumok, műveletek, gépi reprezentálásuk, bitek, byte-ok

A felső szint az alkalmazó területe. Ő tudja, neki van meg, hogy milyen útvonalakon, mely napokon közlekedtet szerelvényeket, és milyen ezen szerelvények összetétele a helyfoglalás szempontjából. A helyfoglalási rendszer

kereteit neki kell kijelölni, ő kell, hogy megmondja, hogy mi a fontos a rendszerében, mi az el nem hagyható tény és mi az ami elhanyagolható. Tudjon-e majd a rendszer mondjuk olyan igényt is kielégíteni, hogy ablaknál akar ülni az utas, de csak a menetiránnyal azonos irányban, ne háttal. Az alkalmazó a valóságnak egy modelljéhez a kereteket alkotja meg. A későbbi üzemelő rendszer paramétereit, képességeit, rugalmasságát és használhatóságát ez a szint döntően meghatározza. A középső szinten a modell gyakorlati megvalósítása következik, amely már az egyes adatkezelési, számítási, tárolási módszereket is magába foglalja. Itt tisztázódik a file-ok rendszere. Rögzítik a használandó táblázatokat, listákat és azok szerkezetét, az adatrekordok felépítését. Az egyes esetekben használt keresési módszerek, az adatmódosítások módszerei is kialakításra kerülnek, miután eldöntötték, hogy mit és hogyan tárolnak. Az alsó szint a gépek, berendezések szintje, amelyek fizikailag meg is tudják valósítani, amit a középső szinten elterveztek. Ezen a szinten nincs modell, nincs szerelvény, dátum, útvonal. Az adatok, a tárolási szerkezetek és a rajtuk végzett műveletsorok bitek és byte-ok özöneként és átalakításaiként jelennek meg. Itt már minden a biteken, a byte-okon múlik. Azon, hogy az egyes adatainkat milyen elvek alapján transzformáltuk bitekké és byte-okká, hogy ezek majd akár a legfelső szint fogalomrendszere alapján is értelmezhetők legyenek. Nem kevésbé fontos az esetleg egymástól térben és időben is nagy távolságra lévő eszközök között a kommunikáció lehetősége, ténye és milyensége.

Tekintsünk most egy elemi problémát és annak megoldásait. Legyen adott egy n fős társaság. Az egyes tagok időnként pénzt kérnek kölcsön egymástól. Mindenki felírja, hogy kitől mennyit kért kölcsön, amikor kölcsönkér és kinek mennyit adott kölcsön, amikor kölcsön ad. A társaság tagjai időről-időre összejönnek az összegyűlt tartozásokat kiegyenlíteni. Mindenki összegyűjti a saját listáján mindenkivel kapcsolatban, hogy kinek mennyivel tartozik. Ezután némi kavarodást okozva mindenki megkeres mindenkit, hogy kifizesse a tartozását, ami nem kis időbe telik, ha a társaság létszáma nem lebecsülendő. Ha összegyűjtenénk egy táblázatba a tartozik-követel összesítéseket, akkor egy ilyen tábla valahogy így nézhetne ki mondjuk 5 személy esetén, akiket rendre Aladárnak, Bélának, Cecilnek, Dávidnak és Edének-nek hívnak. A táblázat sora mutatja, hogy ki tartozik, az oszlopa, hogy kinek tartozik.

	Aladár	Béla	Cecil	Dávid	Ede
Aladár	-	5	3	1	2
Béla	1	-	2	3	4
Cecil	5	1	-	1	2
Dávid	2	3	4	-	6
Ede	5	5	1	4	-

Ha ennél a táblánál maradunk, akkor ennek tárolására n fő esetén $n \cdot n - n$ rekesz szükséges, miután mindenki kigyűjtötte a saját nyilvántartásából a többiekkel kapcsolatos meglévő tartozását. A kölcsönök kiegyenlítésére pedig $n(n-1)/2$ találkozót kell létrehozni, ahol a két fél kölcsönösen kiegyenlíti az egymással szembeni adósságát. Mind a két formulában szerepel egy négyzetes tag, ami arra utal, hogy ha a társaság mérete mondjuk 10-szeresre nő, akkor a vele kapcsolatos szervezési és tárolási munka egyaránt körülbelül 100-szorosra emelkedik. Nem éppen bíztató következtetés. Van azonban jobb megoldás is. Nem kell minden alkalommal mindenkinek feljegyezni, hogy kitől mennyit kért, vagy kinek mennyit adott. Elegendő, ha mindenki csak egyetlen számot tárol és azt módosítja kölcsönzés esetén, ez pedig az éppen aktuális össztartozása a többiek felé. Ennek a tárigénye n rekesz és a kigyűjtés sem szükséges. A fenti táblázatból ez a következő módon kapható meg. A sorokban is és az oszlopokban is képezzük az összegeket, majd mindenkinél kivonjuk a tartozásból (sorösszeg) a követel (oszlopösszeg) értékeket:

	Aladár	Béla	Béla	Dávid	Ede	Sorösszeg	Össztartozás
Aladár	-	5	3	1	2	11	11 - 13 = -2
Béla	1	-	2	3	4	10	10 - 14 = -4
Cecil	5	1	-	1	2	9	9 - 10 = -1
Dávid	2	3	4	-	6	15	15 - 9 = 6
Ede	5	5	1	4	-	15	15 - 14 = 1
Oszlopösszeg	13	14	10	9	14		

Tárolni csak az össztartozás oszlopot kell, ami n szám. A tartozások kiegyenlítéséhez szükséges találkozók száma is drasztikusan csökkenthető. A példánál maradva Aladárnak tartoznak 2-vel, ezt Béla megadja Aladárnak. Mostmár Bélának tartoznak 6-tal, azt Cecil adja meg Bélának. Cecilnek tartoznak 7-tel, amit megad neki Dávid, igy Dávidnak lesz 1 hiánya, amit pedig Ede éppen meg tud adni. Ha n fő van, akkor a szükséges találkozók száma legfeljebb $n-1$. Ennél a megoldásnál nem árt, ha Béla és Cecil rendelkeznek némi plusz pénzzel, hogy fizetni tudjanak az elején. Ez elkerülhető azzal, hogy akik tartoznak (Dávid és Ede), azok mindegyike mondjuk Aladárnak adja oda a tartozását és ebből a pénzből Aladár egyenlíti ki a hiányt azoknál, akik pénzre várnak (Aladár, Béla, Cecil). Itt tehát ha a méretek 10-szeresre nőnek, akkor a tárolás és a tartozás kiegyenlítéssel kapcsolatos szervezési munka is csak körülbelül 10-szeresre fog nőni. Érdemes tehát azon elgondolkodni, hogy milyen adatokat milyen formában tárolunk, azokon milyen műveleteket végzünk, hogy a kívánt eredményre jussunk és az a lehető legkisebb erőforrás lekötéssel és energia felhasználásával valósuljon meg.

1.2. Alapvető fogalmak, definíciók

1.1. definíció. Az **alsó egészrész függvény** minden valós számhoz egy egész számot rendel hozzá, éppen azt, amely a tőle nem nagyobb egészek közül a legnagyobb. Az alsó egészrész függvény jele: $\lfloor x \rfloor$, ahol x valós szám. Tömören:

$$\lfloor x \rfloor = \max_{\substack{k \in \mathbb{Z} \\ k \leq x}} k$$

Más szavakkal formálisan: $\lfloor x \rfloor = k$, ahol k olyan egész szám, hogy $k \leq x < k+1$.

1.2. példa.

x	$-5,2$	-5	5	$5,2$
$\lfloor x \rfloor$	-6	-5	5	5

1.3. definíció. A **felső egészrész függvény** minden valós számhoz egy egész számot rendel hozzá, éppen azt, amely a tőle nem kisebb egészek közül a legkisebb. A felső egészrész függvény jele: $\lceil x \rceil$, ahol x valós szám. Tömören:

$$\lceil x \rceil = \min_{\substack{k \in \mathbb{Z} \\ k \geq x}} k$$

Más szavakkal formálisan: $\lceil x \rceil = k$, ahol k olyan egész szám, hogy $k - 1 < x \leq k$.

1.4. példa.

x	$-5,2$	-5	5	$5,2$
$\lceil x \rceil$	-5	-5	5	6

Az alsó és felső egészrész függvények fontosabb tulajdonságai:

1. Ha a egész szám, akkor	$\lfloor a \rfloor = a$	$\lceil a \rceil = a$
2. Ha x valós, a egész szám, akkor	$\lfloor x \pm a \rfloor = \lfloor x \rfloor \pm a$	$\lceil x \pm a \rceil = \lceil x \rceil \pm a$
3. Ha x és y valós számok, akkor	$\lfloor x \pm y \rfloor \geq \lfloor x \rfloor \pm \lfloor x \rfloor$	$\lceil x \pm y \rceil \leq \lceil x \rceil \pm \lceil x \rceil$
4. Ha x valós szám, akkor	$\lfloor -x \rfloor = -\lceil x \rceil$	$\lceil -x \rceil = -\lfloor x \rfloor$
5. Ha $x \leq y$ valós számok, akkor	$\lfloor x \rfloor \leq \lfloor y \rfloor$	$\lceil x \rceil \leq \lceil y \rceil$

1.5. definíció. A **kerekítő függvény** minden valós számhoz a hozzá legközelebb eső egész számot rendeli hozzá. Ha a legközelebbi egész szám nem egyértelmű, akkor a nagyobbat választja. A kerekítő függvény jele: Round(x), ahol x valós szám.

$$\text{Round}(x) = \left\lfloor x + \frac{1}{2} \right\rfloor$$

1.6. példa.

x	-6	$-5,8$	$-5,5$	$-5,2$	-5	5	$5,2$	$5,5$	$5,8$	6
$Round(x)$	-6	-6	-5	-5	-5	5	5	5	6	6

1.7. definíció. A **törtrész függvény** minden valós számhoz azt a számot rendeli hozzá, amely azt mutatja meg, hogy a szám mennyivel nagyobb az alsó egészrészénél. A törtrész függvény jele: $\{x\}$, ahol x valós szám. Tömören:

$$\{x\} = x - \lfloor x \rfloor$$

Mindig fennáll a $0 \leq \{x\} < 1$ egyenlőtlenség.

1.8. példa.

x	$-5,8$	$-5,2$	-5	5	$5,2$	$5,8$
$\{x\}$	$0,2$	$0,8$	0	0	$0,2$	$0,8$

1.9. definíció. Legyen a és b egész szám, $b \neq 0$. Definíció szerint az **egész osztás műveletén** (div) az a/b osztás eredményének alsó egész részét értjük. Tömören:

$$a \operatorname{div} b = \left\lfloor \frac{a}{b} \right\rfloor .$$

1.10. példa.

$$\begin{aligned} -9 \operatorname{div} 4 &= -3 \\ 9 \operatorname{div} 4 &= 2. \end{aligned}$$

1.11. definíció. Legyen a és b egész szám. Definíció szerint az **egész maradék képzését** (mod), az alábbi formulával definiáljuk:

$$a \bmod b = \begin{cases} a, & \text{ha } b = 0 \\ a - b \cdot \lfloor a/b \rfloor = a - b \cdot (a \operatorname{div} b), & \text{ha } b \neq 0 \end{cases}$$

1.2.1. A számítógép programozásáról

A számítógépes programozás területéről több fogalomra lesz szükségünk annak ellenére, hogy igazán egyetlen programozási nyelv mellett sem kötelezzük el magunkat. A számításaink, adatokon végzett tevékenységeink elvégzéséhez gépi utasítások, parancsok rögzített sorozatára lesz szükségünk. Ezeket összefogva **program**nak fogjuk nevezni. A programot valamilyen magas szintű programozási nyelven (az ember gondolkodásmódjához közel álló nyelven) írjuk meg, majd azt a gép nyelvére egy fordítóprogram (**compiler**) segítségével fordítjuk le (remélhetően jól). Ha van *interpreter* program, akkor azzal is megoldható a feladat elvégzésének a gépre történő átvitele. A programok általában **procedúrák** (eljárások) sokaságát tartalmazzák. Ezek a zárt programegységek egy-egy kisebb feladat elvégzésére specializáltak. A program többi részével csak a paramétereik révén tartják a kapcsolatot. Fekete doboznak kell őket tekintenünk. A dobozra rá van írva, hogy miből mit csinál. Vannak (lehetnek) bemenő (**input**) és vannak (lehetnek) kimenő (**output**) paraméterei. A bemenetet alakítják át a kimenetté. Ha ismerjük a procedúra belső szerkezetét – mert mondjuk mi készítettük –, akkor fehér doboz a neve, ha nem ismerjük – mert nem vagyunk kíváncsiak rá, vagy másoktól kaptuk –, akkor fekete doboz szerkezet a neve. Például készíthetünk olyan procedúrát, amely bekéri (input) az a, b, c három valós számot, melyeket egy $ax^2 + bx + c$ kifejezés (itt x valós szám, változó) konstans együtthatóinak tekint, majd eredményül (output) meghatározza a kifejezés valós gyökeinek a számát és ha van(nak) gyök(ök), akkor az(oka)t is megadja. Példa egy lehetséges másik procedúrára: egy file nevének ismeretében a procedúra a file rekordjait valamilyen szempont szerint megfelelő sorrendbe rakja (rendezi). A procedúrák által használt memóriarekeszek – a paramétereket kivéve – a zártságnak köszönhetően **lokális**ak a procedúrára nézve. Csak addig foglaltak, míg a procedúra dolgozik, aktív. A procedúrát munkára fogni az aktivizáló utasítással lehet. Ezt **eljáráshívás**nak is nevezik. Az aktivizált procedúra lehet saját maga az aktivizáló is, ekkor **rekurzív hívás**ról beszélünk, a procedúrát pedig **rekurzív procedúrá**nak nevezzük. A procedúra munkája végén a vezérlés visszaadódik az aktivizáló utasítást követő utasításra. Ezt a mechanizmust a **verem** (stack) révén valósítjuk meg. A verem a memória egy erre a célra kiválasztott része. A procedúra aktivizálásakor ide kerülnek beírásra a procedúra paraméterei és a **visszatérési cím** (az aktivizáló utasítást követő utasítás címe). A procedúrából való visszatéréskor ezen cím és infor-

mációk alapján tudjuk folytatni a munkát, a programot. A visszatéréskor a veremből az aktivizálási információk törlődnek. Ha a procedúra aktivizál egy másik procedúrát, akkor a verembe a korábbiakat követően az új aktivizálási információk is bekerülnek, azt mondjuk, hogy a verem mélyül, a veremmélység szintszáma eggyel nő. Kezdetben a verem üres, a szintszám zérus, procedúrahíváskor a szintszám nő eggyel, visszatéréskor csökken eggyel. A dolog pikantériájához tartozik, hogy a procedúra a lokális változóit is a verembe szokta helyezni, csak ezt közvetlenül nem érzékeljük, mivel a visszatéréskor ezek onnan törlődnek, a helyük felszabadul. Időnként azonban a hatás sajnálatosan látványos, amikor **verem túlcsordulás** (stack overflow) miatt hibajelzést kapunk és a program futása, a feladat megoldásának menete megszakad. Adódhat azonban úgy is, hogy mindenféle hibajelzés nélkül "lefagy a gép". A veremnek a valóságban van egy felső mérethatára, amelyet nagyon nem tanácsos túllépni.

1.12. példa. Nézzünk egy példát a veremhasználatra. Tegyük fel, hogy van még olyan elvetemült informatikus, aki nem tudja, hogy

$$1 + 2 + 3 + \ldots + n = \frac{n \cdot (n+1)}{2},$$

és ezért egy kis procedúrát ír ennek kiszámítására.

Amennyiben az illető a fent említett hibája mellett teljesen normális, akkor igen nagy eséllyel az alábbi módon oldja meg a problémát. A procedúra neve legyen **Summa** és legyen az input paramétere az n, ami jelentse azt, hogy 1-től kezdve meddig történjen az összeadás. Feltételezzük a procedúra jóhiszemű használatát és így az n pozitív egész szám kell legyen. (Nem írjuk meg a procedúrát első lépésben még "bolondbiztosra".) Kirészletezzük egy kissé a procedúra teendőit. Szükség lesz egy gyűjtőrekeszre, ahol az összeget fogjuk képezni és tárolni. Legyen ennek a neve s. A procedúra munkájának végén ez lesz a végeredmény, ezt kapjuk vissza, ez lesz a procedúra output paramétere. Szükség lesz továbbá egy számlálóra, legyen a neve k, amellyel egytől egyesével elszámolunk n-ig és minden egyes értékét az s-hez a számlálás közben hozzáadjuk. Az s-et a munka kezdetén természetesen ki kell nullázni, hiszen nem tudjuk, hogy mi van benne az induláskor.

Ezek után a kósza meggondolások után egy kissé rendezettebb alakban is írjuk le a teendőket.

A Summa procedúra leírása

Összefoglaló adatok a procedúráról:

A procedúra neve	Summa.
Bemenő paraméter	n, megadja, hogy 1-től meddig kell az összeadást elvégezni.
Kijövő paraméter	s, tartalmazza az összeget a végén.
Lokális változó	k, számláló, amely egytől elszámol n-ig egyesével.

A procedúra tevékenysége:

1. lépés:	s kinullázása
2. lépés:	k beállítása 1-re, innen indul a számlálás
3. lépés:	s megnövelése k-val (s-hez hozzáadjuk a k-t és az eredmény s-ben marad.
4. lépés:	Eggyel megnöveljük a k számláló értékét
5. lépés:	Ellenőrizzük, hogy a k számláló nem lépett-e túl az n-nen, a végértéken.
	Ha még nem, akkor folytatjuk a munkát a 3. lépésnél.
	Ha igen, akkor pedig a 6. lépéshez megyünk.
6. lépés:	Készen vagyunk, az eredmény az s-ben található.

Ezután ha szükségünk van, mondjuk, 1-től 5-ig a számok összegére, akkor csak leírjuk, hogy `Summa(Input:5, Output s)`, vagy rövidebben `Summa(5,s)`. Esetleg függvényes alakot használva az `s=Summa(5)` is írható. Az aktivizálás hatására a verembe bekerül az 5-ös szám, valamint az s rekesz memóriabeli címe és a visszatérési cím, hogy a procedúra munkája után hol kell folytatni a tevékenységet. Miután most nincs több teendő, ezért ez a cím olyan lesz, amelyből ez a tény kiderül. Jelezhetjük ezt formálisan mondjuk egy `a STOP utasítás címe`-pal. Valahogy így néz ki a verem formálisan:

5	s címe	STOP utasítás címe

Kezdetben üres volt a verem, most egy szint került bele bejegyzésre. Amikor a procedúra munkája véget ér, akkor ez a bejegyzés a veremből törlődik, így az újra üres lesz. (Tulajdonképpen a számláló számára lefoglalt helyet is fel kellett volna tüntetni a bejegyzésben, de ez a számunkra most nem fontos.)

Minden nagyon szép, minden nagyon jó, mindennel meg vagyunk elégedve, és akkor jön egy rekurzióval megfertőzött agyú ember, aki így gondolkodik. Egytől n-ig összeadni a számokat az ugyanaz, mint az egytől $n-1$-ig összeadott számok összegéhez az n-et hozzáadni. A feladatot visszavezettük saját magára, csak kisebb méretben. Egytől $n-1$-ig persze megint úgy adunk össze, hogy az $n-2$-ig képezett összeghez adjuk az $n-1$-et. Ez a rekurzió. Arra kell vigyázni, hogy valahol ennek a visszavezetésnek véget kell vetni. Amikor már csak egytől egyig kell az összeget képezni, akkor azt nem vezetjük vissza tovább, hiszen ott tudjuk az eredményt, ami triviálisan éppen egy. Tehát a rekurzív agyú ember egy függvényt alkot, mondjuk `RekSumma` néven, és az alábbi módon definiálja azt:

$$RekSumma(n) := \begin{cases} 1 & \text{ha } n = 1 \\ RekSumma(n-1) + n & \text{ha } n > 1 \end{cases}$$

Ha most leírjuk, hogy $s = RekSumma(5)$, akkor ezt úgy kell kiszámolni, hogy:

$$\begin{aligned} s = RekSumma(5) &= RekSumma(4) + 5 \\ &= (RekSumma(3) + 4) + 5 \\ &= ((RekSumma(2) + 3) + 4) + 5 \\ &= (((RekSumma(1) + 2) + 3) + 4) + 5 \\ &= ((1 + 2) + 3) + 4) + 5 \\ &= ((3 + 3) + 4) + 5 \\ &= (6 + 4) + 5 \\ &= 10 + 5 \\ &= 15 \end{aligned}$$

Lássuk ezekután hogyan alakul a verem története. A $RekSumma(5)$ hatására az üres verembe egy bejegyzés kerül:

5	eredmény	Az eredmény s-be írási címe

A továbbiakban pedig a verem az egyes rekurzív hívások hatására a következőképpen alakul:

RekSumma(5):	5	eredmény	Az eredmény s-be írási címe
RekSumma(4):	4	eredmény	Összeadás helye
RekSumma(3):	3	eredmény	Összeadás helye
RekSumma(2):	2	eredmény	Összeadás helye
RekSumma(1):	1	eredmény	Összeadás helye

Itt a rekurzió megakad, további rekurzív hívás már nem lesz, a végleges veremmélység 5, a rekurzív hívások száma 4 (a legelső aktivizálás még nem rekurzív hívás). A legutolsó hívás már tud számolni, és az eredmény 1 lesz, ami a veremben meg is jelenik:

RekSumma(5):	5	eredmény	Az eredmény s-be írási címe
RekSumma(4):	4	eredmény	Összeadás helye
RekSumma(3):	3	eredmény	Összeadás helye
RekSumma(2):	2	eredmény	Összeadás helye
RekSumma(1):	1	1	Összeadás helye

Ezután az utolsó előtti hívásbeli összeadás (1+2) elvégezhető, a hívás befejeződik és a veremből a legutolsó bejegyzés törlődik. A továbbiakban rendre az alábbi veremállapotok állnak elő:

RekSumma(5):	5	eredmény	Az eredmény s-be írási címe
RekSumma(4):	4	eredmény	Összeadás helye
RekSumma(3):	3	eredmény	Összeadás helye
RekSumma(2):	2	3	Összeadás helye

RekSumma(5):	5	eredmény	Az eredmény s-be írási címe
RekSumma(4):	4	eredmény	Összeadás helye
RekSumma(3):	3	6	Összeadás helye

RekSumma(5):	5	eredmény	Az eredmény s-be írási címe
RekSumma(4):	4	10	Összeadás helye

RekSumma(5):	5	15	Az eredmény s-be írási címe

Innen a visszatérés az értékadáshoz, az s-be történő eredmény elhelyezéshez történik, miáltal a verem kiürül. Az elmondottak alapján látszik, hogy a feladat elvégzéséhez szükséges maximális veremmélység 5 és összesen 4 rekurzív hívás történt.

Itt akár fel is lélegezhetnénk, de ekkor egy újabb, még súlyosabb állapotban lévő fazon jelenik meg, aki azt mondja, hogy lehet ezt még szebben is csinálni. Ő a rekurziót arra építi, hogy az összeg képezhető úgy is, hogy az összeadandó számok halmaza első felének összegéhez hozzáadja a halmaz második felének összegét. A felezést további felezéssel számolja, mígcsak az aprózódás révén el nem jut egytagú ősszegekig. Röviden és tömören ő egy másik függvényt definiál, amely kétváltozós, neve `RekSum(m,n)`, és m-től n-ig adja össze a számokat. Ezzel az általánosabb függvénnyel egytől n-ig összeadni `RekSum(1,n)`-nel lehet. Speciálisan a mi fenti problémánk esetében: `RekSum(1,5)` számolandó. Az ő definíciója így néz ki:

$$RekSum(n,m) := \begin{cases} m & \text{ha } n = m \\ RekSum\left(n, \left\lfloor \frac{n+m}{2} \right\rfloor\right) + RekSum\left(\left\lfloor \frac{n+m}{2} \right\rfloor + 1, m\right) & \text{ha } m > n \end{cases}$$

Nézzük csak hogyan is számol ez a ravasz módszer a mi speciális `s=RekSum(1,5)` esetünkben?

$$\begin{aligned} s = RekSum(1,5) &= RekSum(1,3) + RekSum(4,5) \\ &= (RekSum(1,2) + RekSum(3,3)) + (RekSum(4,4) + RekSum(5,5)) \\ &= ((RekSum(1,1) + RekSum(2,2)) + 3) + (4+5)) \\ &= (((1+2)+3)+4)+5 \\ &= (3+3)+(4+5) \\ &= (6+9) \\ &= 15 \end{aligned}$$

Hogyan alakul a verem sorsa ebben az esetben? Az első aktivizáló hívás után a verem:

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe

Ezután következik a `RekSum(1,3)` hívás. A hatása:

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe
RekSum(1,3):	1	3	eredmény	Összeadásjel

Most jön a `RekSum(1,2)` hívás a `RekSum(1,3)`-on belül. A hatás:

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe
RekSum(1,3):	1	3	eredmény	Összeadásjel
RekSum(1,2):	1	2	eredmény	Összeadásjel

Ez megint nem számolható közvetlenül, tehát jön a `RekSum(1,1)`, mire a verem új képe:

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe
RekSum(1,3):	1	3	eredmény	Összeadásjel
RekSum(1,2):	1	2	eredmény	Összeadásjel
RekSum(1,1):	1	1	eredmény	Összeadásjel

Itt már van eredmény, átmenetileg nincs több rekurzív hívás. Az eredmény 1.

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe
RekSum(1,3):	1	3	eredmény	Összeadásjel
RekSum(1,2):	1	2	eredmény	Összeadásjel
RekSum(1,1):	1	1	1	Összeadásjel

A hívás befejezte után a veremből kiürül a legutolsó bejegyzés, visszatérünk az összeadásjelhez, amely után azonban egy újabb rekurzív hívás keletkezik, a `RekSum(2,2)`. Hatására a verem képe:

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe
RekSum(1,3):	1	3	eredmény	Összeadásjel
RekSum(1,2):	1	2	eredmény	Összeadásjel
RekSum(2,2):	2	2	eredmény	Összeadás befejezése

Az innen történő visszatérés után a verem képe:

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe
RekSum(1,3):	1	3	eredmény	Összeadásjel
RekSum(1,2):	1	2	3	Összeadásjel

Az összeadás elvégzéséhez itt azonban egy újabb rekurzív hívás szükséges, a `RekSum(3,3)`.

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe
RekSum(1,3):	1	3	eredmény	Összeadásjel
RekSum(3,3):	3	3	eredmény	Összeadás befejezése

Innen

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe
RekSum(1,3):	1	3	6	Összeadásjel

következik, majd pedig egy újabb hívás, a `RekSum(4,5)`. A veremállapot:

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe
RekSum(4,5):	4	5	eredmény	Összeadásjel

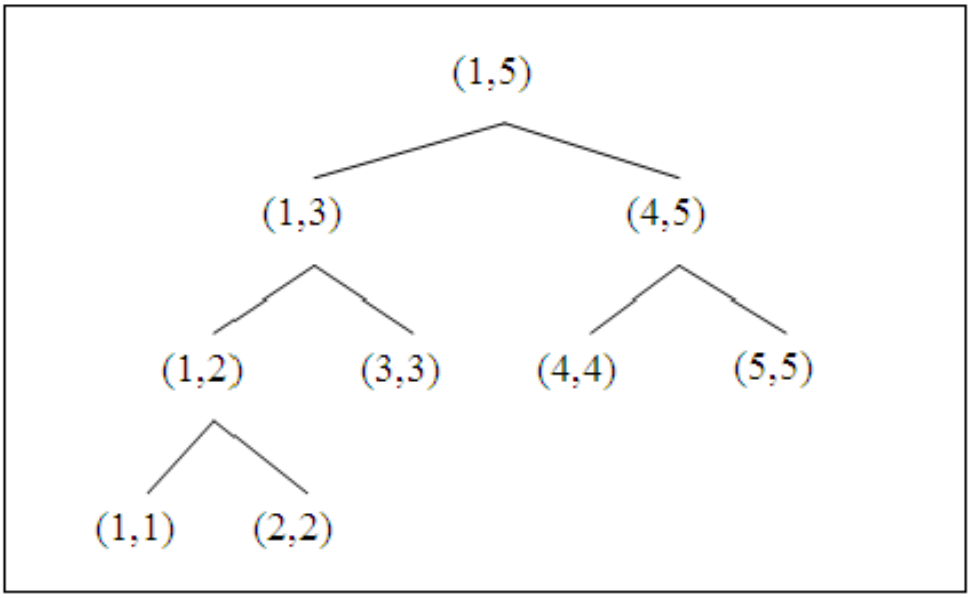
Újabb hívás szükséges a `RekSum(4,4)`. A veremállapot:

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe
RekSum(4,5):	4	5	eredmény	Összeadás vége
RekSum(4,4):	4	4	eredmény	Összeadásjel

Ennek befejezte után és a veremből történő törlést követően még kell egy hívásnak lennie, ez pedig a `RekSum(5,5)`. A veremállapot:

RekSum(1,5):	1	5	eredmény	Az eredmény s-be írási címe
RekSum(4,5):	4	5	eredmény	Összeadás vége
RekSum(5,5):	5	5	eredmény	Összeadás vége

Innentől kezdve a verem már csak ürül, további rekurzív hívásokra nincs szükség. A feladat elvégzéséhez kevesebb szintből álló verem is elég volt, mint az előző esetben, most a maximális veremmélység csak 4 volt. A rekurzív hívások száma azonban megnőtt, összesen nyolc rekurzív hívás volt. Ebben a rekurzióban minden hívás, kivéve a legalsóbb szinten levőket két újabbat eredményezett, de ezek a veremnek ugyanazon szintjét használták. A hívások szerkezetét egy úgynevezett hívási fa sémával tudjuk ábrázolni, melyben csak a paraméter értékeket tüntetjük fel. Íme:



Az ábrán jól látszik a verem négy szintje. A legfelső szint kivételével a többi szinten lévő hívások rekurzívak. Az azonos szinten lévő hívások a verem azonos szintjét használják, csak eltérő időben.

2. fejezet

Az absztrakt adattípus és az algoritmus

2.1. Az absztrakt adat és adattípus

Az adat fogalma az értelmező szótár szerint:
"Az adat valakinek vagy valaminek a megismeréséhez, jellemzéséhez hozzásegítő (nyilvántartott) tény vagy részlet." (Lásd [1]).

Mi adatnak fogunk tekinteni minden olyan információt, amelynek segítségével leírunk egy jelenséget, tanulmányunk tárgyát, vagy annak egy részét. Az adat formai megjelenésére nem leszünk tekintettel, ettől lesz absztrakt. (Absztrakt adat.) Egy rúd hosszát megadhatjuk úgy is, hogy mondjuk százhuszonhét centiméter. Itt nem fontos, hogy a százhuszonhét a 127 formájában van-e megadva, esetleg 1111111_2, vagy $7F_{16}$ alakban. (Egy számítógépes program számára persze ez egyáltalán nem mindegy.) Ez a fejtegetés sem sokkal konkrétabb. Például mi az az információ? Erre a kérdésre a választ nem feszegetjük. Az adat fogalma az alkalmazások, példák és feladatok során lesz tisztább. Tulajdonképpen azt is nehéz megmondani, hogy mi nem lehet adat.

2.1. definíció. Az **absztrakt adat** valamely halmaznak az eleme. Ezen halmaz bármely elemét felhasználhatjuk a munkánkban, számításainkban, az alkalmazott valóságmodellben, objektumainak leírásában, megadásában.

2.2. definíció. Az **absztrakt adattípus** egy leírás, amely absztrakt adatok halmazát és a rajtuk végezhető műveleteket adja meg (definiálja) nem törődve azok konkrét (gépi) realizálásával.

2.3. definíció. n-változós (n-áris) műveletnek nevezzük az $A^n \to A$ a leképzést, függvényt, ahol A az absztrakt adat halmazát jelöli. Azaz ez a leképzés egy absztrakt adat n-eshez szintén egy ugyanolyan absztrakt adatot rendel hozzá.

A műveletek közül kiemelkednek a bináris (binér) műveletek, amelyekben tehát az elnevezés alapján is érthetően a művelet elempárokhoz rendel hozzá egy elemet eredményképpen. Például a valós számok esetén ilyen művelet lehet két szám összeadása. Számunkra fontosak az egyes műveletek tulajdonságai. A tulajdonságok megléte, vagy meg nem léte lehetővé teszi vagy éppen nem teszi lehetővé, hogy a formuláinkat átalakítsuk, egyszerűsítsük. Ilyen tulajdonságok például az asszociativitás, kommutativitás, disztributivitás, idempotencia, stb., melyeket az alkalmas helyeken tárgyalunk.

Példák absztrakt adattípusokra:

- logikai érték,
- természetes szám,
- egész szám,
- racionális szám,
- valós szám,
- komplex szám,
- sorozat,
- halmaz,
- dinamikus halmaz.
  - tömb

- verem
- sor
- lista
- fa
- gráf

2.1.1. A logikai absztrakt adattípus

A logikai absztrakt adattípus egy olyan halmazt ad meg, amelynek két eleme van, a **hamis** és az **igaz**. Jelölésben L=$\{$*hamis, igaz*$\}$. Röviden az elemeket a $\boldsymbol{h}$ (hamis) és az $\boldsymbol{i}$ (igaz) jellel jelöljük.

A típus műveletei lehetnek unáris (unér) és bináris (binér) aszerint, hogy hány operandusuk van. (Lehetnek többoperandusú műveletek is, de ezek a korábbiakkal kifejezhetők.)

Unáris műveletek

Unáris művelet a Negáció, (tagadás, NEM, NOT). Jele: felülvonás a logikai adat neve fölött. Pl.: x és $\overline{x}$. Művelettáblája:

x	$\overline{x}$
h	i
i	h

További három unáris művelet alkotható, amelyek azonban triviálisak. Ezek az identikus művelet, a hamis és az igaz művelet. Ez utóbbi kettő konstans eredményt ad.

Művelettábláik:

x	Identikus	Hamis	Igaz
h	h	h	i
i	i	h	i

Bináris műveletek

<table>
<tr><th>Művelet neve</th><th>Jele</th><th>Művelettáblája</th></tr>
<tr><td>Diszjunkció
(VAGY, OR)</td><td>$x \vee y$</td><td>
<table>
<tr><td>x</td><td>y</td><td>$x \vee y$</td></tr>
<tr><td>h</td><td>h</td><td>h</td></tr>
<tr><td>h</td><td>i</td><td>i</td></tr>
<tr><td>i</td><td>h</td><td>i</td></tr>
<tr><td>i</td><td>i</td><td>i</td></tr>
</table>
</td></tr>
<tr><td>Konjunkció
(ÉS, AND)</td><td>$x \wedge y$</td><td>
<table>
<tr><td>x</td><td>y</td><td>$x \wedge y$</td></tr>
<tr><td>h</td><td>h</td><td>h</td></tr>
<tr><td>h</td><td>i</td><td>h</td></tr>
<tr><td>i</td><td>h</td><td>h</td></tr>
<tr><td>i</td><td>i</td><td>i</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Művelet neve</th><th>Jele</th><th>Művelettáblája</th></tr>
<tr><td>Antivalencia (KIZÁRÓ VAGY, XOR)</td><td>$x \oplus y$</td><td>
<table>
<tr><td>x</td><td>y</td><td>$x \oplus y$</td></tr>
<tr><td>h</td><td>h</td><td>h</td></tr>
<tr><td>h</td><td>i</td><td>i</td></tr>
<tr><td>i</td><td>h</td><td>i</td></tr>
<tr><td>i</td><td>i</td><td>h</td></tr>
</table>
</td></tr>
<tr><td>Ekvivalencia</td><td>$x \leftrightarrow y$</td><td>
<table>
<tr><td>x</td><td>y</td><td>$x \leftrightarrow y$</td></tr>
<tr><td>h</td><td>h</td><td>i</td></tr>
<tr><td>h</td><td>i</td><td>h</td></tr>
<tr><td>i</td><td>h</td><td>h</td></tr>
<tr><td>i</td><td>i</td><td>i</td></tr>
</table>
</td></tr>
<tr><td>Implikáció</td><td>$x \rightarrow y$</td><td>
<table>
<tr><td>x</td><td>y</td><td>$x \rightarrow y$</td></tr>
<tr><td>h</td><td>h</td><td>i</td></tr>
<tr><td>h</td><td>i</td><td>i</td></tr>
<tr><td>i</td><td>h</td><td>h</td></tr>
<tr><td>i</td><td>i</td><td>i</td></tr>
</table>
</td></tr>
<tr><td>Peirce nyíl (NEM VAGY, NOR)</td><td>$x \downarrow y$</td><td>
<table>
<tr><td>x</td><td>y</td><td>$x \downarrow y$</td></tr>
<tr><td>h</td><td>h</td><td>i</td></tr>
<tr><td>h</td><td>i</td><td>h</td></tr>
<tr><td>i</td><td>h</td><td>h</td></tr>
<tr><td>i</td><td>i</td><td>h</td></tr>
</table>
</td></tr>
<tr><td>Scheffer vonás (NEM ÉS, NAND)</td><td>$x \mid y$</td><td>
<table>
<tr><td>x</td><td>y</td><td>$x \mid y$</td></tr>
<tr><td>h</td><td>h</td><td>i</td></tr>
<tr><td>h</td><td>i</td><td>i</td></tr>
<tr><td>i</td><td>h</td><td>i</td></tr>
<tr><td>i</td><td>i</td><td>h</td></tr>
</table>
</td></tr>
</table>

Bináris műveletből 16-ot lehet felírni. A műveletek erősorrendje csökkenő erő szerint (prioritás, zárójelezés nélkül írhatók) a következő: NEM, ÉS, VAGY, KIZÁRÓ VAGY, Ekvivalencia, Implikáció.

A NEM, ÉS, VAGY műveletek tulajdonságai:

1.	Kettős tagadás	$\overline{\overline{x}} = x$	
2.	Kommutativitás	$x \vee y = y \vee x$	$x \wedge y = y \wedge x$
3.	Asszociativitás	$(x \vee y) \vee z = x \vee (y \vee z)$	$(x \wedge y) \wedge z = x \wedge (y \wedge z)$
4.	Disztributivitás	$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
5.	Idempotencia	$x \vee x = x$	$x \wedge x = x$
6.	Konstansok hatása	$x \vee i = i$ $x \vee h = x$	$x \wedge i = x$ $x \wedge h = h$
7.	Elnyelés	$x \vee (x \wedge y) = x$	$x \wedge (x \vee y) = x$
8.	Ellentmondás		$x \wedge \overline{x} = h$
9.	Harmadik kizárása	$x \vee \overline{x} = i$	
10.	De Morgan	$\overline{x \vee y} = \overline{x} \wedge \overline{y}$	$\overline{x \wedge y} = \overline{x} \vee \overline{y}$

Ezen három művelettel (NEM, ÉS, VAGY) az összes többi (a többváltozósak is) kifejezhetők.

2.4. példa.

$$x \oplus y \equiv (\overline{x} \wedge y) \vee (x \wedge \overline{y})$$
$$x \rightarrow y \equiv \overline{x} \vee y$$
$$x \leftrightarrow y \equiv (x \wedge y) \vee (\overline{x} \wedge \overline{y})$$

Szintén kifejezhető az összes művelet csupán a (NEM, VAGY), vagy a (NEM, ÉS), vagy a (NEM, KIZÁRÓ VAGY), vagy a (NEM VAGY) vagy a (NEM ÉS) műveletekkel.

2.5. példa.

$$x \leftrightarrow y \equiv \overline{x \oplus y}$$

A diszjunktív normálforma

2.6. definíció. Elemi konjunkció
Változók vagy tagadottjainak a konjunkciója, melyben a változók legfeljebb egyszer fordulnak elő.

2.7. definíció. Diszjunktív normálforma (DNF)
Elemi konjunkciók diszjunkciója.

Művelettábla alapján DNF előállítása: Ahol az eredmény oszlopban i van, azokat az eseteket diszjunkcióval kötjük össze úgy, hogy a változók konjunkcióiból formulát alkotunk. A formulában i esetén a változó szerepel, h esetén a változó negáltja.

2.8. példa.

x	y	$x \oplus y$
h	h	h
h	i	i
i	h	i
i	i	h

Innen $x \oplus y \equiv (\overline{x} \wedge y) \vee (x \wedge \overline{y})$.

A logikai változó realizálása történhet bitekkel: hamis – 0, igaz – 1.

2.9. definíció. Izomorfizmus
Két algebrai struktúrát ***izomorf***nak nevezünk, ha létezik olyan kölcsönösen egyértelmű megfeleltetés a két struktúra elemei között, amely esetén a műveletek is szinkronizálódnak. Ez azt jelenti, hogy ha az egyik struktúra az $(A, \circ)$, a másik struktúra a $(B, \Box)$, a kölcsönösen egyértelmű megfeleltetés pedig $f : A \to B$, akkor fennáll, hogy $f(a_1 \circ a_2) = f(a_1) \Box f(a_2)$ minden $a_1 \in A$ és $a_2 \in A$ esetén. Az f megfeleltetést nevezzük ***izomorfizmus***nak.

2.10. példa. Legyen $A = \mathbb{R}_+$ a pozitív valós számok halmaza a szorzás művelettel felruházva, és $B = \mathbb{R}$ az összes valós szám halmaza az összeadás művelettel. Akkor az $f : \mathbb{R}_+ \to \mathbb{R}$ megfeleltetés, ahol $f(x) = \log(x)$, izomorfizmust valósit meg, hiszen a logaritmus azonosságai szerint: $\log(xy) = \log(x) + \log(y)$ minden pozitív x és y esetén.

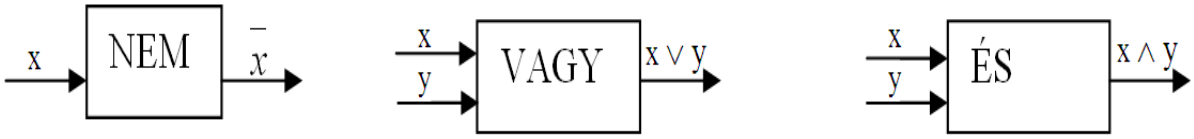
Legyen most az A halmaz az L logikai adattípus a negáció, a konjunkció és a diszjunkció műveletével felruházva. Legyen a B halmaz a $\{0, 1\}$ számokból álló halmaz, amelyen értelmezzük az alábbi három műveletet:

- Ellentett elem képzése unáris művelet : $e(b) = 1 - b$, $b \in B$ a kivonás művelete révén.
- Szorzás bináris művelet: $b_1 \cdot b_2$, $b_1, b_2 \in B$ a számok szorzási műveletének megfelelően.
- Bitösszegzés bináris művelet: $b_1 \oplus b_2 = b_1 + b_2 - b_1 \cdot b_2$ $b_1, b_2 \in B$ a számokra érvényes szokásos összeadás, kivonás és szorzás művelete révén.

Ekkor a most definiált három művelet a negáció, konjunkció, és diszjunkció műveletének megfeleltetve, valamint a logikai mennyiségeknek a biteket a fenti módon megfeleltetve a logikai adattípus és a bit adattípus között izomorfizmust hoztunk létre. Azt mondjuk, hogy a logikai adatokat bitekkel modellezzük. Amilyen törvényszerűséget találunk az egyikben, az izomorfizmus révén megtaláljuk a törvény párját a másikban.

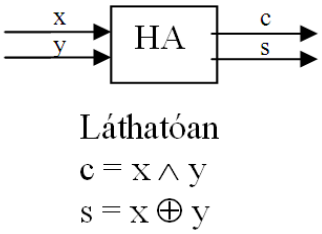
A logikai típus nagyon fontos, mert az értékeket feszültségszintekhez lehet társítani, a műveleteket pedig úgynevezett kapuáramkörökkel valósíthatjuk meg. A kapuáramkör fizikai (technikai) felépítése lényegtelen a számunkra, az változott az idők folyamán (relék, diódák, tranzisztorok, stb.). Sematikusan úgy jelölhetjük őket, mint egy dobozba zárt átalakító szerkezet, amelynek vannak bemenetei és kimenetei. A bemeneteken bemenő jeleket dolgozzák fel a rendeltetésüknek megfelelően, és az eredmény megjelenik a kimeneteken.

Példa kapuáramkörökre:



Kapuáramkörökből felépíthető az úgynevezett félösszeadó (Half Adder). Feladata egyetlen bitpozíción képezni a két bit összegét és az átvitelt, tehát a

művelet mindig kétjegyű eredményt képez, melynek alacsonyabb helyiértékű bitje az összegbit, magasabb helyiértékű bitje az átvitelbit (Carry bit).



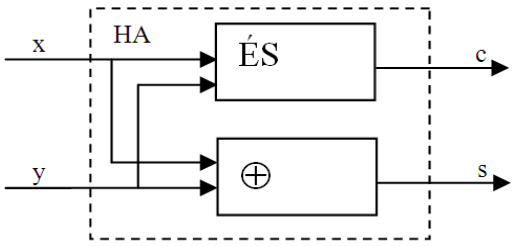
Láthatóan

$c = x \wedge y$

$s = x \oplus y$

Művelettábla

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



A félösszeadó több-bites számok összeadásakor csak fél munkát végez, helyesen csak a legalacsonyabb bitpozíción működik. A további pozíciókon három bitet kell összeadni, a két összeadandó bitet és az előző pozícióról jövő átvitelbitet. A teljes összeadó (Full Adder) ezt valósítja meg.

Műveletáblája:

x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

Felírva a két eredményoszlopra a diszjunktív normálformákat, tulajdonképpen megkapjuk a műveletek egy lehetséges kapuzását.

$$c_{out} = (x \wedge y \wedge \overline{c_{in}}) \vee (\overline{x} \wedge y \wedge c_{in}) \vee (x \wedge \overline{y} \wedge c_{in}) \vee (x \wedge y \wedge c_{in})$$

$$s = (\overline{x} \wedge y \wedge \overline{c_{in}}) \vee (x \wedge \overline{y} \wedge \overline{c_{in}}) \vee (\overline{x} \wedge \overline{y} \wedge c_{in}) \vee (x \wedge y \wedge c_{in})$$

Ezt a látszólag bonyolult formulát le lehet egyszerűsíteni és a teljes összeadó felépíthető két félösszeadó és egy VAGY kapuáramkörből. Előtte azonban egy segéd formulát vezetünk le.

2.11. lemma.

$$c_{out} = (\overline{x} \wedge \overline{y}) \vee (x \wedge y) = \overline{x \oplus y}$$

Bizonyítás. Láttuk, hogy

$$x \oplus y = (\overline{x} \wedge y) \vee (x \wedge \overline{y})$$

Akkor

$$(\overline{x} \wedge \overline{y}) \vee (x \wedge y) = \overline{\overline{(\overline{x} \wedge \overline{y}) \vee (x \wedge y)}} = \overline{\overline{(\overline{x} \wedge \overline{y})} \wedge \overline{(x \wedge y)}} = \overline{(x \vee y) \wedge (\overline{x} \vee \overline{y})} =$$
$$= \underbrace{\overline{(x \wedge \overline{x})}}_{h} \vee (y \wedge \overline{x}) \vee (x \wedge \overline{y}) \vee \underbrace{\overline{(y \wedge \overline{y})}}_{h} = \overline{(y \wedge \overline{x})} \vee (x \wedge \overline{y}) = \overline{x \oplus y}$$

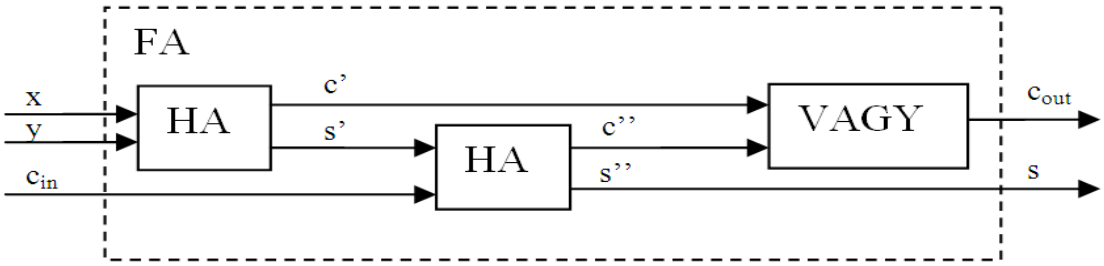
□

Ezután a teljes összeadó levezetése az alábbi lehet:
Jelölje s', c' és s'', c'' az első, valamint a második félösszeadó által adott eredmény összegbitet és az átvitelbitet. Akkor

$$c_{out} = (x \wedge y \wedge \overline{c_{in}}) \vee (\overline{x} \wedge y \wedge c_{in}) \vee (x \wedge \overline{y} \wedge c_{in}) \vee (x \wedge y \wedge c_{in}) =$$
$$= (x \wedge y) \wedge \left(\underbrace{\overline{c_{in}} \wedge c_{in}}_{i} \right) \vee \left[\underbrace{(\overline{x} \wedge y) \vee (x \wedge \overline{y})}_{x \oplus y} \right] \wedge c_{in} = \underbrace{(x \wedge y)}_{c'} \vee \underbrace{\left(\underbrace{x \oplus y}_{s'} \right) \wedge c_{in}}_{c''} = c' \vee c''$$

$$s = (\overline{x} \wedge y \wedge \overline{c_{in}}) \vee (x \wedge \overline{y} \wedge \overline{c_{in}}) \vee (\overline{x} \wedge \overline{y} \wedge c_{in}) \vee (x \wedge y \wedge c_{in}) =$$
$$= \left[\underbrace{(\overline{x} \wedge y) \vee (x \wedge \overline{y})}_{x \oplus y} \right] \wedge \overline{c_{in}} \vee \left[\underbrace{(\overline{x} \wedge \overline{y}) \vee (x \wedge y)}_{\overline{x \oplus y}} \right] \wedge c_{in} = \underbrace{(x \oplus y)}_{s'} \oplus c_{in} = s' \oplus c_{in}$$

A teljes összeadó sematikus ábrája:



2.1.2. A karakter absztrakt adattípus

A karakter absztrakt adattípus a szöveges információ megjelenítést teszi lehetővé. A szövegeinket elemi egységekből építjük fel.

2.12. definíció. Karakter
A karakter a tágabb értelemben szövegesen lejegyzett adat legkisebb, elemi egysége, egy tovább már nem bontható szimbólum.

A karakterek halmazát X-szel fogjuk jelölni.

A karaktereket osztályozhatjuk jellegük szerint. Eszerint a karakterek lehetnek: számjegyek, betűk, írásjelek (pont, vessző, felkiáltó jel, stb.), speciális jel (félgrafikus jel, matematikai jelek, hieroglifák, szimbólumok, stb.), vezérlőjel (csengő, soremelés, lapdobás, kocsi vissza, file vége, escape, stb.).

A karakter absztrakt adattípusra jellemző, hogy az X halmaz elemei között rendezettséget vezetünk be, amely konvención, megállapodáson alapul. Előre rögzítjük a sorrendjüket. (Ezt a sorrendet nevezhetjük ábécé sorrendnek.) A sorrend szerint az egyes karaktereket sorszámmal láthatjuk el. Két műveletet be is vezethetünk ezáltal. Az egyik lehet a Hátrább_álló, a másik lehet az Előbb_álló. A Hátrább_álló a két karakter közül azt adja eredményül, amelyik az X-ben hátrább áll, azaz amelyiknek a kettő közül nagyobb a sorszáma, míg az Előbb_álló azt adja, amelyiknek kisebb a sorszáma. A sorszámot a karakter kódjának nevezzük.

2.13. példa. Előbb_álló('K','C')='C', és Hátrább_álló('K','C')='K'

Ha bevezetünk bináris műveleti jeleket a két műveletre, például legyen az Előbb_álló jele $\lhd$, a Hátrább_álló jele $\rhd$, akkor az előzőleg felírt példa lehet

$$'K' \lhd' C' =' C' \qquad \text{és} \qquad 'K' \rhd' C' =' K'$$

Civilizációnkban rendkívül sok karakterrel találkozhatunk. Az informatikában kezdetben nem sokat használtak fel ezek közül. Szorított a tárolóhely hiánya. Jelentős lépés volt, amikor az akkor legfontosabbnak tekintett jelkészletet egységesítették, szabványosították. Ez volt az ASCII kódtáblázat (American Standard Code for Information Interchange, az információcsere amerikai szabványos kódja), amely hétbites kód volt. Jellemzője, hogy 0-tól 127-ig terjed az egyes karakterek kódja és az első 32 jel (0-31) vezérlőjel. Ilyenek például a csengő (7), a soremelés (10), a lapdobás (12), a kocsi vissza (13), a file vége (26), az escape (27). Vezérlőjel még (törlőjel) a 127-es kódú karakter. A többi jel látható. Ilyenek a helyköz (32), a számjegyek növekvő sorrendben (48-57), az angol ábécé nagybetűi (65-90), az angol ábécé kisbetűi (97-122). A kisbetű kódja 32-vel nagyobb, mint a neki megfelelő nagybetűé. A teljes ASCII kódtáblázat a mellékletben látható. Az ASCII kódok tárolása a byte-os memória és társzervezés következtében az egy byte egy karakter elvet követte. Mivel egy byte-on 256 féle bitmintázat helyezhető el, ezért kihasználatlan maradt a 127-es kód feletti 128-255 számtartomány 128 eleme. Ugyanakkor az informatika nemzetközivé válása miatt szükségessé vált a nemzeti ábécék jeleinek az alkalmazása is, amelyet az ASCII tábla bővítésével igyekeztek megoldani. Az ASCII tábla bővítése sajnos rossz irányba történt, A 128 elemű eredeti készletet jóval több, mint további 128 jellel kellett volna bővíteni. Megtartották az egy byte-os szerkezetet. Ezért a 127-es kód feletti kódokat több célra kellett használni, hogy mindenkinek az igényét kielégítsék. Bevezették a kódlap fogalmát. Definiáltak például Latin-1 kódlapot, amelybe sok ékezetes betű is belefért, persze felrúgva ezzel a betűk ábécé sorrendjét. A magyar ő és ű betű azonban ebbe sem fért bele. Azt a Latin-2 kódlapra tették. Az eredmény az lett, hogy ha a szöveget megjelenítő program nem tudta, hogy a szövegfile eredetileg milyen kódlap alapján készült (honnan tudta volna, amikor ez az információ a file-okban nem szerepel), akkor amennyiben ő más kódlap szerinti megjelenítésre volt beállítva, a szöveg akár olvashatatlanná is vált. Ugyanannak a kódnak többféle karakter is megfelelt a kódlapoknak megfelelően. Másik probléma volt az, hogy vannak nyelvek, amelyeknek több ezer írásjel szimbóluma van, amelyek eleve nem férnek el egy ilyen 256-os táblázatban. Ezeket kétbyte-os ábécében helyezték

el. Azonban ezek a törekvések bár bevezetésre kerültek, a problémákat nem szüntették meg már csak azért sem, mert egy szöveg tartalmazhat különböző nyelven megírt részleteket is.

A megoldást a Unicode (UCS – Universal Character System) szolgáltatja. Ebben a rendszerben minden egyes szimbólumnak saját sorszáma van, azaz a kód (sorszám) egyértelműen azonosítja a szimbólumot. A Unicode 31-bites kód, ezért minden karakter (szimbólum) négy byte-ot foglal el. Milliárdnál is több karakter számára van itt hely biztosítva, ebbe úgy gondoljuk minden használt szimbólumunk belefér. Valójában az informatikában eddig használt szimbólumok kettő byte-on is elférnek. A karakter UCS kódjának a jelölése: U-xxxx, ahol az xxxx legalább négyjegyű hexadecimális szám, a karakter sorszáma. (A hétbites ASCII karakterek kódjai megörződtek a Unicode-ban.) Ez a kódformátum egy kissé pazarlóvá válik a karakterenkénti négy byte alkalmazásával. Ezért egy tömörebb formátumot is bevezettek, amely főként akkor hatékony, ha a hagyományos karaktereinket használjuk, mondjuk európai nyelvek keverékével írunk szöveget. Ez a formátum az UTF-8 (UCS Transformation Format). Egy Unicode file-t átírhatunk UTF-8 formátumúra megőrizve ezzel a karakterek egyediségét és általában jelentős helyet takarítva meg. Az átírás szabályai az alábbiak.

Ha a karakter Unicode kódja legfeljebb hét biten is elfér, akkor a négy byte helyett az alsó nyolc bitet tartjuk meg. Ezt az esetet nevezik az egyedüli byte esetének. Ha hét bitnél hosszabb a kód, akkor az átkódolás során a kódot kettőtől hat byte-ig terjedő hosszan kódoljuk át attól függően, hogy mennyire hosszú a Unicode. Az átkódolás ilyenkor egy byte sorozat, amely egy kezdőbyte-tal indul, melyet egy vagy több nem kezdő byte követ. A kezdő byte felépítése: $1\ldots10x\ldots x$, ahol a byte magasabb helyiértékű végén annyi egyes van, ahány byte-ból áll a sorozat, az x-ek pedig már a Unicode bitjeit tartalmazzák. A nem kezdő byte-ok mindegyike 10xxxxxx alakú, ahol az x-ek a Unicode bitjei. Az alábbi séma mutatja az átkódolás menetét annak függvényében, hogy a Unicode karakter maximálisan hány értékes (nem vezető nulla) bitet tartalmazhat. Felülről lefelé az első alkalmazható szabályt kell alkalmazni.

Unicode	UTF-8
00000000 00000000 00000000 0xxxxxxx	0xxxxxxx
00000000 00000000 00000xxx xxxxxxxx	110xxxxx 10xxxxxx
00000000 00000000 xxxxxxxx xxxxxxxx	1110xxxx 10xxxxxx 10xxxxxx
00000000 000xxxxx xxxxxxxx xxxxxxxx	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
000000xx xxxxxxxx xxxxxxxx xxxxxxxx	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
0xxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

2.14. példa. Például az ó betű Unicode-ja U-00F3, ezért az UTF-8 átírása:

00000000 00000000 00000000 11110011-ből 1100 0011 1011 0011 lesz. Az aláhúzott bitek a unicode-ból kerültek át az UTF-8-ba.

Tulajdonságok:

1. az angol UTF-8 és ASCII file azonos
2. több byte-os sorozat belső byte-ja sem lehet 128-nál kisebb
3. minden szimbólum legfeljebb hat byte.os
4. tipikus magyar szöveg esetén kb. 10% hossznövekedés van
5. nem minden file érvényes UTF-8 file (nem lehet benne 254-es és 255-ös byte)
6. a programokat fel kell készíteni az ilyen file-ok kezelésére (egy karakter nem egy byte)

2.1.3. Az egész szám

Az egész szám esetén nagyon megszoktuk a 10-es alapú számrendszer használatát. Valójában nem törődünk a szám leírásával. Egy szám esetében a

számítógép nem a 10-es, hanem a 2-es alapot használja a szám reprezentálására. Attól még az a szám ugyanaz a szám. Nincs értelme absztrakt egész szám esetén beszélni például egy szám számjegyeiről. Más a helyzet abban a pillanatban, amikor az absztrakt adattípus szerinti adatot konkrétan meg akarjuk jeleníteni. Akkor már nem lényegtelen az ábrázolási forma. Próbáljuk meg például összeszorozni papíron kézzel és ceruzával a hetvenkilencet és a negyvenhetet úgy, hogy a két számot 79 és 47 alakban adjuk meg, valamint úgy is, hogy $LXXIX$ és $XLVII$ alakban. Az első esetre van egy módszer, egy könnyen megjegyezhető séma, amely szerint a kisiskolás is el tudja végezni a számítást, a másik esetben pedig nehéz ilyet adni, holott a szorzat létezik az ábrázolástól függetlenül. Adható persze az ábrázolástól független módszer is két nemnegatív egész szám összeszorzására. Ennek a módszernek a neve ma gyakran úgy olvasható, hogy "orosz paraszt" módszer. Az elnevezés nem teljesen jogos, mert a módszert már az ókori egyiptomiak is ismerték és használták [3]. A módszer lényege, hogy a szorzatot fokozatosan gyűjtjük össze és a zérusból indulunk ki. A szorzót vizsgáljuk. A vizsgálat abból tart, hogy megnézzük, hogy páratlan-e. Ha igen, akkor a szorzandót hozzáadjuk a szorzathoz, ha nem, akkor nem adjuk hozzá. Ezután a szorzót lecseréljük a felének az egész részére, a szorzandót pedig lecseréljük a duplájára. A vizsgálat mindaddig ismétlődik, míg a szorzó zérussá nem válik. (Lássuk be, hogy a módszer mindig a helyes szorzathoz vezet két nemnegatív egész szám esetén!)

2.15. példa. Szorozzuk össze a 79-et és a 47-et az "orosz paraszt" módszerrel!

Szorzandó	Szorzó	Szorzó páratlan?	Szorzat
79	47	*igen*	$0 + 79 = 79$
158	23	*igen*	$79 + 158 = 237$
316	11	*igen*	$237 + 316 = 553$
632	5	*igen*	$553 + 632 = 1185$
1264	2	*nem*	$= 1185$
2528	1	*igen*	$1185 + 2528 = 3713$
	0		$= 3713$

Az absztrakt adattípus egy fekete doboz, amelybe beletesszük az adatot tárolásra és kivesszük, ha szükségünk van rá. Nem érdekes, hogy a doboz hogyan

végzi a tárolást. Ami viszont fontos az az, hogy az absztrakt adattípushoz elválaszthatatlanul hozzátartoznak azok a műveletek, amelyeket az adatokkal végezni lehet.

2.16. definíció. Adatstruktúrának nevezzük az absztrakt adattípus konkrét megjelenési formáját.

A műveletek az adatstruktúrához ugyanúgy hozzátartoznak, mint az absztrakt adattípushoz. Példák adatstruktúrára:

- lista, verem,
- sor, elsőbbségi (prioritásos) sor,
- bináris kupac, binomiális kupac,
- fa, gráf, hálózat.

Adatstruktúra a számábrázolás módja is. Az elnevezésekben azonban az absztrakt adattípus és az adatstruktúra nevek gyakran keverednek, hiszen tartalmilag hasonló dolgokról van szó.

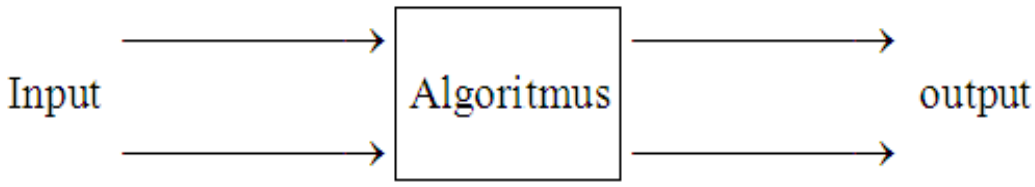
2.2. Az algoritmus fogalma

Az adatainkon általában különféle műveleteket, átalakításokat szoktunk végezni azzal a céllal, hogy ezáltal közvetlenül nem kiolvasható összefüggéseket, eredményeket kapjunk. A tevékenységeket logikai sorrendbe rakva az algoritmus matematikai fogalmához kerülünk közelebb. Az algoritmus mély matematikai fogalom. Mi nem adunk precíz definíciót rá, mivel ebben a könyvben erre nincs szükségünk. Azok számára, akiket a téma mélyebben érdekel ajánlhatjuk a [4, 5, 6] könyveket.

Most megadunk egy heurisztikus (nem tudományos) definíciót az algoritmus fogalmára.

2.17. definíció. Az **algoritmus** egy meghatározott számítási eljárás, a számítási probléma megoldási eszköze.

Az algoritmus pontos előírás, amely megad egy tágan értelmezett számítási folyamatot. Az algoritmus valamely előre meghatározott adathalmaz valamely tetszőleges kiinduló eleméből kezdve az ezen elem által meghatározott eredmény elérésére törekszik. Lehet, hogy a lépések sorozata azzal szakad meg, hogy nincs eredmény. Az algoritmus is tekinthető egy fekete doboznak, melynek a bemenetére adjuk a probléma, a feladat kiinduló adatait, a kimenetén pedig megjelennek a végeredmények, ha vannak, vagy az jelenik meg, hogy nincsenek. Az algoritmus fekete dobozának belső szerkezete azonban érdekelni fog minket ebben a könyvben. Az algoritmusnak véges idő alatt (véges sok lépés után) véget kell érnie.



2.18. példa. Négyzetgyök kiszámítása egy számból papíron kézzel.

Határozzuk meg az $x = \sqrt{s}$ számot megadott számú értékes jegy pontosságig, ahol $s > 0$ valós szám. Egy kézi algoritmus az alábbi formulára építkezve adható:

$$(10a + b)^2 = 100a^2 + 20ab + b^2 = 100a^2 + (2 \cdot 10a + b)b$$

Az algoritmus leírása (ez az algoritmus épít a szám reprezentációjára, azaz arra, hogy helyiértékes számrendszert használunk):

1. Az s szám számjegyeit a tizedesvesszőtől balra és jobbra kettes csoportokra osztjuk.

2. A balszélső (első) csoportnak vesszük az egyjegyű négyzetgyökét és az eredménybe írjuk.

3. A kapott egyjegyű szám négyzetét kivonjuk az első csoportból.

4. A maradék mellé leírjuk a következő kétjegyű csoportot, ha van. (A tizedesvesszőt követően mindig lehet zérust, vagy zéruspárokat írni, ha már nem lennének tizedesjegyek.)

5. Az eddig kapott eredmény kétszereséhez hozzáillesztünk egy próbaszámjegyet, majd az így kapott számot a próbaszámjeggyel megszorozva a szorzatot kivonjuk a 4. pontnál kapott legutóbbi maradék és következő csoport által meghatározott számból. A próbaszámjegy a lehető legnagyobb olyan számjegy legyen, amely még nemnegatív különbséget ad.

6. A próbaszámjegyet az eredményhez hozzáírjuk új számjegyként.

7. Ha a pontosság megfelelő, akkor leállunk, egyébként a 4-es pontnál folytatjuk.

2.19. példa. Konkrét számpélda a kézi négyzetgyökvonás algoritmusára, működésének bemutatása.

Számítsuk ki az $x = \sqrt{14424804}$ értékét! A szám csoportokra osztása: 14 42 48 04. Az algoritmus további lépései az alábbi táblázatban találhatók. A próbajegyeket és a hozzáírt csoportot aláhúztuk.

lépés	csoport	maradék és csoport	számjegy (próbajegy)	duplázás (1. lépést kivéve)	szorzat	maradék
1	14	$\underline{14}$	3	$3^2 = 9$	-	14-9=5
2	42	$5\underline{42}$	7	$2 \cdot 3 = 6$	$6\underline{7} \cdot 7 = 469$	542-469=73
3	48	$73\underline{48}$	9	$2 \cdot 37 = 74$	$74\underline{9} \cdot 9 = 6741$	7348-6741=607
4	04	$6070\underline{4}$	8	$2 \cdot 379 = 758$	$758\underline{8} \cdot 8 = 60704$	60704-60704=0

Kiolvasható, hogy $x = \sqrt{14424804} = 3798$. Az eredmény pontos, mivel a végső maradék zérus.

2.20. példa. Számítsuk ki az $x = \sqrt{2}$ értékét négy tizedes jegyre!

A szám csoportokra osztása: 2, 00 00 00 00.

lépés	csoport	maradék és csoport	számjegy (próbajegy)	duplázás	szorzat	maradék
1	2	$\underline{2}$	1	$1^2 = 1$	-	2-1=1
2	00	$1\underline{00}$	4	$2 \cdot 1 = 2$	$2\underline{4} \cdot 4 = 96$	100-96=4
3	00	$4\underline{00}$	1	$2 \cdot 14 = 28$	$28\underline{1} \cdot 1 = 281$	400-281=119
4	00	$11\underline{00}$	4	$2 \cdot 141 = 282$	$282\underline{4} \cdot 4 = 11296$	11900-11296=604
5	00	$604\underline{00}$	2	$2 \cdot 1414 = 2828$	$2828\underline{2} \cdot 2 = 56564$	60400-56564=3836

$x = \sqrt{2} \approx 1.4142$. Ezen közelítés négyzete a 2-től csak 0,00003896-tal kevesebb.

A tárgyalt algoritmus kellemes, az eredmény jegyei egymás után egyenként jönnek elő. Neuralgikus pont viszont a próbajegyek helyes megválaszásának a kérdése, Igaz, ez a probléma megoldódik, ha a számításokat kettes számrendszerben végezzük. Mégsem ragaszkodunk a négyzetgyökvonás ezen módjához, mivel itt lényeges a szám reprezentációja. Adunk egy másik algoritmust, amely lényegesen jobb mutatókkal rendelkezik. Ez az algoritmus az úgynevezett Newton módszernek egy speciális esete. Ez az algoritmus nem számjegyenként csalja elő a végeredményt, hanem egy számsorozatot képez, amely meglehetősen gyorsan konvergál a végeredményhez. Nem árt persze kellően jó kezdő közelítésből kiindulni. Érdemes megjegyezni, hogy ha a módszer által képzett sorozatban valamely elem már tartalmaz értékes jegyeket a megoldást leíró szám elejéből, akkor minden további elemben az értékes jegyek száma legalább duplájára nő a megelőzőhöz képest. Maga az algoritmus egyszerű és jól programozható, valamint nem igényli a szám reprezentációját. Íme (a leírásban az alkalmazó előre rögzít egy $\varepsilon > 0$ számot, amit pontossági előírásnak nevezünk):

1. Választunk egy tetszőleges pozitív x_0 valós számot és legyen $k = 0$. (Az $x_0 = 1$ mindig megfelelő, csak esetleg a kapott sorozat kezdetben lassan kezd közelíteni a megoldáshoz.)

2. Képezzük az
$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{s}{x_k}\right)$$
számot és k értékét eggyel növeljük.

3. Ha $|x_k - x_{k-1}| \leq \varepsilon$, akkor megállunk és $x \approx x_k$, egyébként pedig folytatjuk a 2-es pontnál.

2.21. példa. Határozzuk meg az $x = \sqrt{14424804}$ értékét négy tizedesjegyre, azaz legyen $\varepsilon = 0,0001$!

Heurisztikus meggondolásból válasszuk kezdőértéknek az $x_0 = 2048$ számot!

k	x_k	x_k kiszámítása
0	2048	-
1	4545,680664	$= \frac{1}{2}\left(2048 + \frac{14424804}{2048}\right)$
2	3859,489842	$= \frac{1}{2}\left(4545,680664 + \frac{14424804}{4545,680664}\right)$
3	3798,489832	$= \frac{1}{2}\left(3859,489842 + \frac{14424804}{3859,489842}\right)$
4	3798,000032	$= \frac{1}{2}\left(3798,489832 + \frac{14424804}{3798,489832}\right)$

2.22. példa. Határozzuk meg az $x = \sqrt{2}$ értékét négy tizedesjegyre, azaz legyen $\varepsilon = 0,0001$!

Mivel $1 < \sqrt{2} < 2$, ezért válasszuk a kezdő közelítést $x_0 = 1,5$-nek!

k	x_k	x_k kiszámítása
0	1,5	-
1	1,416666667	$= \frac{1}{2}\left(1,5 + \frac{2}{1,5}\right)$
2	1,414215686	$= \frac{1}{2}\left(1,416666667 + \frac{2}{1,416666667}\right)$
3	1,414213562	$= \frac{1}{2}\left(1,414215686 + \frac{2}{1,414215686}\right)$

2.3. Az algoritmus megadási módja: a pszeudokód és a folyamatábra.

Az algoritmust szövegesen adtuk meg a fenti esetekben. Ez egy lehetőség általában az algoritmus lejegyzésére. Elterjedt azonban a pszeudokódos megadás is, mely közelebb viszi a leírást a számítgógépes megvalósításhoz anélkül, hogy elkötelezné magát egy konkrét programozási nyelv mellett. (A programozási nyelvek divatja változik - ma már több programozási nyelv van, mint a beszélt emberi nyelvek száma, - de a már lejegyzett algoritmus lényege nem változik.) Alább ismertetünk néhány pszeudokód konvenciót, megállapodást, amely segít az ilyen módon megadott algoritmusok megértésében.

1. Blokkszerkezeteket fogunk használni, amint az sok programozási nyelvben elterjedt. A blokk zárójelezése helyett a bekezdés eltolásának módszerét fogjuk használni.

2. Az alábbi strukturális (strukturált vagy kvázistrukturált) utasításokat fogjuk használni:

Utasítás szerkezete	Utasítás magyarázata
IF feltétel THEN blokk	(Utasításkihagyásos elágazás.) Ha a feltétel igaz, akkor a THEN blokk végrehajtódik, egyébként nem.
IF feltétel THEN blokk ELSE blokk	(Kétirányú elágazás.) Ha a feltétel igaz, akkor a THEN blokk hajtódik végre, egyébként az ELSE blokk.
CASE kifejezés $feltétel_1$: blokk ... $feltétel_n$:blokk	(Többirányú elágazás.) A kifejezéssel kapcsolatos igaz feltétel után megadott blokk hajtódik csak végre. Ha a feltételek listáján egyik sem igaz, akkor egyik blokk sem hajtódik végre. Egyidejűleg legfeljebb egy feltételnek szabad csak igaznak lenni.
CASE kifejezés $feltétel_1$: blokk ... $feltétel_n$:blokk ELSE blokk	(Többirányú elágazás.) A kifejezéssel kapcsolatos igaz feltétel után megadott blokk hajtódik csak végre. Ha a feltételek listáján egyik sem igaz, akkor az ELSE mögötti blokk hajtódik végre. Egyidejűleg legfeljebb egy feltételnek szabad csak igaznak lenni.

FOR ciklusváltozó kezdőérték TO végérték DO Blokk	(Előrehaladó leszámláló, elöltesztelő ciklus.) A ciklusváltozó beáll a kezdőértékre. Ellenőrzésre kerül, hogy a ciklusváltozó nagyobb-e, mint a végérték. Ha kisebb, vagy egyenlő, akkor a DO blokk végrehajtódik és a ciklusváltozó értéke eggyel nő, majd az ellenőrzésnél folytatjuk. Ha nagyobb a ciklusváltozó értéke a végértéknél, akkor kilépünk a ciklusból.
FOR ciklusváltozó kezdőérték DOWNTO végérték DO Blokk	(Visszafelé haladó leszámláló, elöltesztelő ciklus.) A ciklusváltozó beáll a kezdőértékre. Ellenőrzésre kerül, hogy a ciklusváltozó kisebb-e, mint a végérték. Ha nagyobb, vagy egyenlő, akkor a DO blokk végrehajtódik és a ciklusváltozó értéke eggyel csökken, majd az ellenőrzésnél folytatjuk. Ha kisebb a ciklusváltozó értéke a végértéknél, akkor kilépünk a ciklusból.
FORALL elem $\in$ Halmaz DO Blokk	Ciklus, amely a véges Halmaz minden elemére, azok felsorolási sorrendjében végrehajtandó.
WHILE feltétel DO Blokk	(Elöltesztelő iteratív ciklus.) Ha a feltétel igaz, akkor végrehajtódik a DO blokk és visszatérünk a feltétel ellenőrzéséhez. Ha a feltétel hamis, akkor kilépünk a ciklusból.
REPEAT Blokk UNTIL feltétel	(Hátultesztelő iteratív ciklus.) Végrehajtjuk a blokkot, majd ha a feltétel hamis, akkor visszatérünk a blokk ismétlésére. Ha a feltétel igaz, akkor kilépünk a ciklusból.
RETURN(paraméterlista)	Kilépés a procedúrából. A felsorolt paraméterek átadása a hívó rutinnak.

3. Az értékadás jele a $\leftarrow$ jel lesz. Bátran alkalmazzuk tömbök, struktúrák értékadására és többszörös értékadásra is.

4. A magyarázatokat, megjegyzéseket // kezdőjellel fogjuk jelezni. Ez lehet egy teljes megjegyzés sor, vagy lehet egy adott sorhoz hozzáfűzött megjegyzés.

5. Az eljárásokban használt változók lokálisak lesznek.

6. Tömbelemet indexeléssel adunk meg. Lehet egy index (vektor), két index (mátrix), vagy több. Az indexek résztartományát . . .-tal jelöljük. Például $3 \ldots 6$ jelenti a 3,4,5,6 indexeket.

7. Az összetett adatok (objektumok) mezőkkel rendelkeznek, amelyekben az objektum attributumait, tulajdonságait tároljuk. A mezőre a nevével hivatkozunk. A mezőnév mögött szögletes zárójelben feltüntetjük az objektum nevét.

8. A tömbök vagy objektumok mutatók révén lesznek megadva. A `NIL` mutató sehová sem, semilyen objektumra sem mutat. Ha x mutat egy objektumra, y egy másikra, akkor az $x \leftarrow y$ értékadás után az x is és az y is ugyanarra az objektumra mutat, nevezetesen az y által jelzettre. Az x által korábban mutatott objektum ezáltal elvész, mivel a mutatója eltünt.

9. Az eljárások az input paramétereiket érték szerint kapják meg, azaz a paraméterről egy másolat készül (ami a verembe helyeződik el). Az eljárásnak a paramétereken végzett változtatásai a hívó rutinban nem láthatók emiatt, hiszen a veremből ezek a visszatéréskor törlődnek. Objektum paraméter esetén azonban az objektum mutatójának másolata kerül a verembe, nem maga az objektum, ezért az objektum mezőin végzett változtatások a hívó rutinban is láthatóak lesznek a visszatérés után. Nem láthatók viszont magának a mutatónak a megváltozásai. Az input és output paramétereket a paraméterlistán feltüntetjük és megjegyzés sorokban írjuk le azokat. Az output paramétereket a visszatérési `RETURN` utasításban is megadjuk.

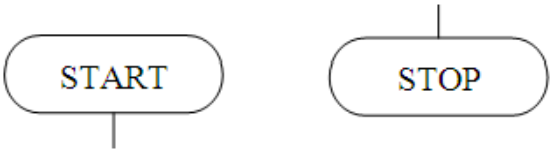
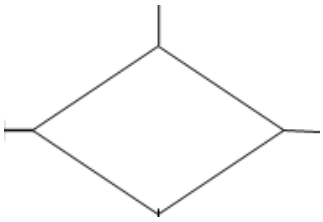
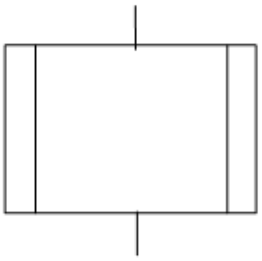
10. A pszeudokód nem zárja ki, hogy az algoritmus egyes részeit szöveges módon tüntessük fel.

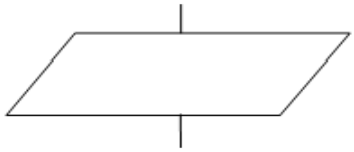
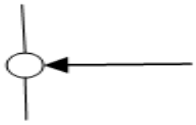
A fenti iteratív négyzetgyökvonási algoritmus pédául így nézhetne ki. A jobboldalon egy praktikusabb változat látható.

	1.2.1 algoritmus
	Négyzetgyökvonás iterációval
1	NÉGYZETGYÖK (s, z)
2	// Input: s - nemnegatív szám
3	// Output: z - nemnegatív szám
4	$x_0 \leftarrow 1$
5	$k \leftarrow 0$
6	REPEAT $x_{k+1} \leftarrow (x_k + s/x_k)/2$
7	$\quad k \leftarrow k + 1$
8	UNTIL $\lvert x_k - x_{k-1} \rvert < \varepsilon$
9	$z \leftarrow x_k$
10	RETURN (z)

	1.2.2 algoritmus
	Négyzetgyökvonás iterációval
1	NÉGYZETGYÖK (s, z)
2	// Input: s - nemnegatív szám
3	// Output: z - nemnegatív szám
4	$z \leftarrow 1$
5	REPEAT $x \leftarrow z$
6	$\quad z \leftarrow (x + s/x)/2$
7	UNTIL $\lvert z - x \rvert < \varepsilon$
8	RETURN (z)

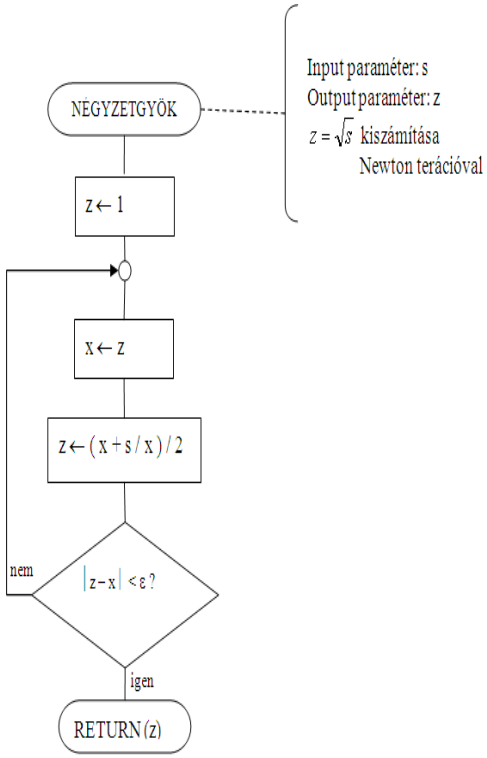
A folyamatábra az algoritmust folyamatában a sík kétdimenziós tulajdonságát kihasználva grafikus szimbólumok felhasználásával teszi szemléletessé. Az alábbi, vízszintes vagy függőleges folyamatvonalak révén egymáshoz kapcsolható szimbólumokat használjuk: A folyamatvonalak összefutását kis körökkel jelöljük.

Szimbólum	**Szimbólum magyarázata**
	Kezdőszimbólum (az algoritmus kezdete, pontosan egy van) és a befejező szimbólum (az algoritmus megállási helye, legalább egy van)
	Tevékenység
	Döntés a szimbólumba írt feltétel milyensége alapján
	Alprogram, procedúra

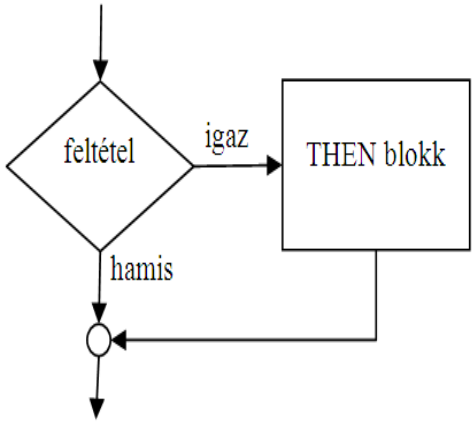
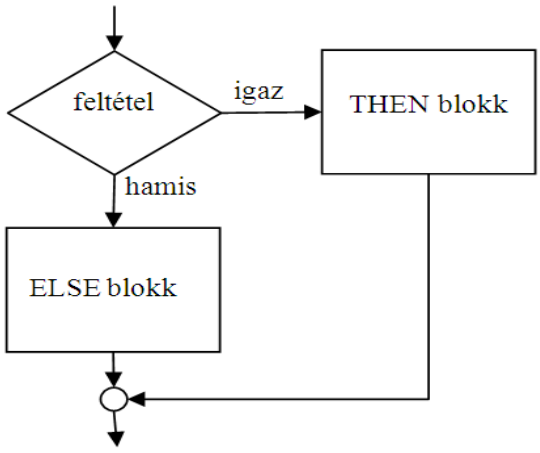
	Input - output
	Kapcsoló szimbólumok az ábra távolabbi részeinek összekapcsolására. A körökbe írt jel, - általában szám - mutatja az összetartozó szimbólumokat.
	Folyamatvonalak találkozása, összefutása

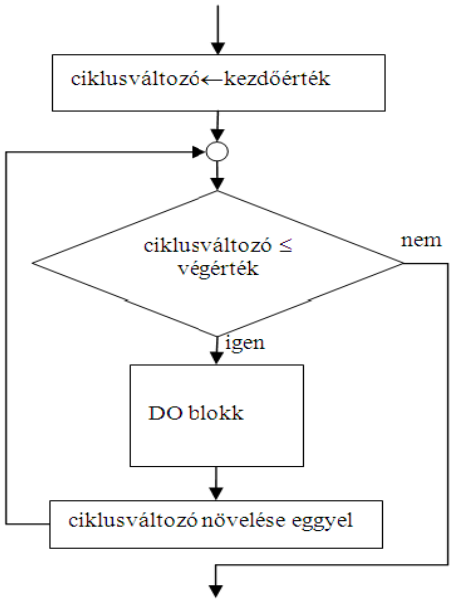
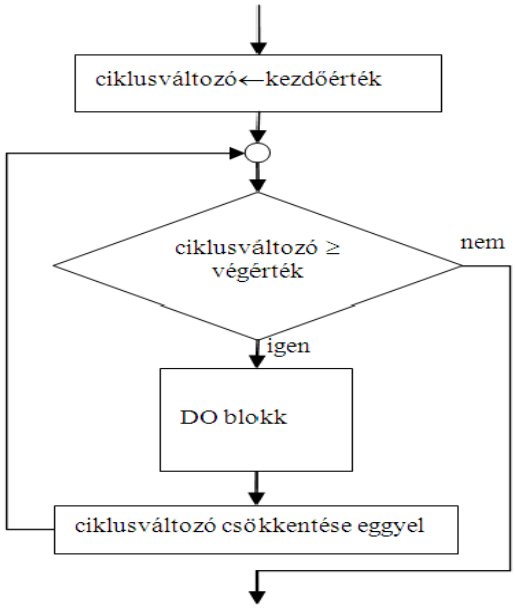
A folyamatvonalakon a haladás iránya balról-jobbra, vagy fentről-lefelé, hacsak a vonalra kitett nyíl másként nem mutat. Megjegyzést a szimbólumokhoz szaggatott vonallal a szimbólumhoz kapcsolt, megfelelően méretezett kezdő szögletes zárójel jellel lehet hozzákapcsolni. A szöveg a szögletes zárójel mögé kerül.

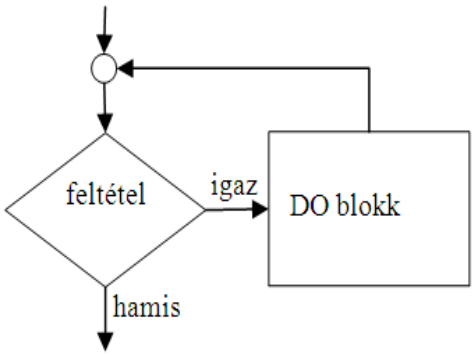
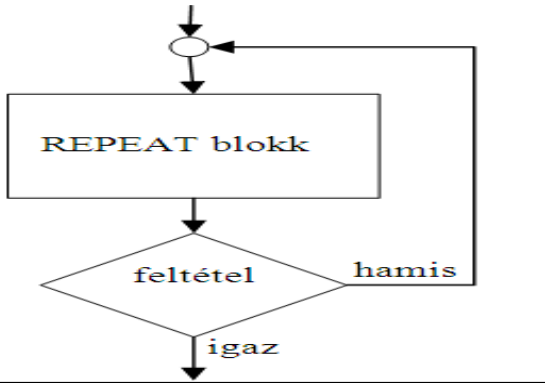
A négyzetgyökvonás fenti praktikusabb változata folyamatábrával:



A strukturális utasításoknak megfelelő folyamatábra részletek:

IF feltétel THEN blokk	IF feltétel THEN blokk ELSE blokk
	

FOR ciklusváltozó ← kezdőérték TO végérték DO blokk	FOR ciklusváltozó ← kezdőérték DOWNTO végérték DO blokk
	

WHILE feltétel DO blokk	REPEAT blokk UNTIL feltétel
	

Az algoritmusok közül kitűnnek a rekurzív algoritmusok és az iteratív algoritmusok. Mindkét fajta algoritmus hatékonyan realizálható számítógépen. Az iteratív algoritmusok hasonló, vagy azonos műveletek sorozatát ismétlik (a latin *iteratio* szó ismétlést jelent). A rekurzív algoritmusokban azt ismerjük

fel, hogy a probléma mérete redukálható kisebb méretre, majd még kisebb méretre, stb. és a kisebb méretű feladat megoldása után visszatérhetünk (a latin *recursio* szó visszatérést jelent) a nagyobb méretűnek a megoldásához, amely ezáltal lényegesen egyszerűbbé válik. Iteratív algoritmus volt a `Summa` algoritmus. és a kézi négyzetgyökvonás algoritmusa. Ugyancsak iteratív algoritmusok az 1.2.1 és 1.2.2 algoritmusok. Rekurzív algoritmus volt a `RekSumma` és a `RekSum` algoritmus. A rekurzív algoritmusok mindig átírhatók iteratív formára is. Az említett algoritmusoknak a pszeudokódját az alábbi módon készíthetjük el procedúra formában:

1	Summa (n, s)
2	$s \leftarrow 0$
3	FOR $k \leftarrow 1$ TO n DO
4	$s \leftarrow s + k$
5	RETURN (s)

1	RekSumma (n, s)
2	IF $n = 1$
3	THEN $s \leftarrow 1$
4	ELSE $s \leftarrow RekSumma(n-1, u)$
5	$s \leftarrow u + n$
6	RETURN (s)

1	RekSum (m , n, s)
2	IF $m = n$
3	THEN $s \leftarrow m$
4	ELSE $RekSum(m, \lfloor (m+n)/2 \rfloor, u)$
5	$RekSum(\lfloor (m+n)/2 \rfloor + 1, n, u)$
6	$s \leftarrow u + v$
7	RETURN (s)

Egy probléma megoldására nem mindig könnyű algoritmust találni még akkor sem, ha ismert, hogy van megoldása a problémának és hogy csak egy

megoldása van. (Ha nincs megoldása a problémának, akkor persze nincs értelme algoritmust keresni.) Megemlíthetők azonban általános elvek, amelyek figyelembe vehetők egy-egy algoritmus kidolgozásakor. Ez persze nem jelenti azt, hogy ezen elvek figyelembe vételével biztosan mindig célba is érünk. Az intuíciónak továbbra is korlátlanok a lehetőségei és a szerepe nem csökken. Ilyen általános elvet, *algoritmus tervezési stratégiá*t hármat említünk a könyvben:

1. **Oszd meg és uralkodj** Ez a stratégia a kiinduló problémát *kisebb méretű, független, hasonló részproblémák*ra bontja, amelyeket *rekurzív*an old meg. A kisebb méretű részproblémák megoldásait egyesítve kapja meg az eredeti probléma megoldását.
2. **Mohó algoritmus** Ezt a heurisztikát általában optimalizálásra használják. A mohó stratégia elve szerint az adott pillanatban mindig az ott legjobbnak tűnő lehetőséget választjuk a részprobléma megoldására, azaz *a pillanatnyi lokális optimumot választjuk. z a választás függhet az előző választásoktól, de nem függ a későbbiektől.* A mohó stratégia nem mindig vezet globális optimumra.
3. **Dinamikus programozás** A stratégia a kiinduló problémát *nemfüggetlen (közös) részproblémákra* bontja, amelyek *egyszer oldódnak meg* és az eredmények az újabb felhasználásig tárolódnak. Általában optimalizálásra használjuk, amikor sok megengedett megoldás van. Lépései:

    1. Jellemezzük az optimális megoldás szerkezetét.
    2. *Rekurzív* módon definiáljuk az optimális megoldás értékét.
    3. Kiszámítjuk az optimális megoldás értékét alulról felfelé módon.
    4. A kiszámított információk alapján megszerkesztjük az optimális megoldást.

2.4. Az algoritmus jellemző vonásai (tulajdonságai)

Minden algoritmusnak vannak jellemző tulajdonságai. Ezek között vannak olyanok, amelyek általánosnak tekinthetők. Ezeket az alábbiakban soroljuk fel:

1. *A kiinduló adatok lehetséges halmaza (D)*
   Ez a halmaz azon adatokat tartalmazza, amelyeket az algoritmus megkaphat mint input adatokat, tehát ezek előjöhetnek egy probléma konkretizálása során.
2. *A lehetséges eredmények halmaza*
   Ezt a halmazt az input adatok halmaza határozza meg. Minden input adathoz tartozik eredmény adat, amit majd az algoritmus meg kell, hogy találjon.
3. *A lehetséges közbülső eredmények halmaza*
   a halmaz a nevében is mutatja, az algoritmus végrehajtása közben keletkező közbülső eredményeket tartalmazza. Menet közben ebből a halmazból nem léphetünk ki.
4. *A kezdési szabály*
   Az algoritmus első (kezdő) műveletét szabja meg.
5. *A közvetlen átalakítási szabályok*
   Azok a szabályok, amelyeket menet közben törvényszerűen használhatunk egy-egy adott szituációban.
6. *A befejezési szabály*
   Az a szabály, amelyből egyértelműen kiderül, hogy az algoritmus végrehajtása végetért.
7. *Az eredmény kiolvasási szabálya*
   Az a szabály, amely alapján a kelekezett adatokból eldönthető, hogy mi az eredmény és az hol, milyen formában található, hogyan nyerhető ki.

2.23. példa. A gyökvonás 1.2.2. algoritmusa (2.3) esetében a 7 pont így nézhetne ki:

1. Kiinduló adatok lehetséges halmaza tetszőleges pozitív szám.
2. A lehetséges eredmények halmaza tetszőleges pozitív szám.
3. A közbülső eredmények tetszőleges pozitív szám.
4. A kezdési szabály a $k = 0$ számláló beállítás és az $x_0 = 1$ kezdőértékből történő indulás.
5. Átalakítási szabály a Newton iterációs formula: $x_{k+1} = (x_k + s/x_k)/2$ és a számláló növelése.
6. A befejezési szabály a pontosság ellenőrzése és annak teljesülése esetén a befejezés.
7. Az eredmény kiolvasható a legutoljára kapott x_k értékből.

2.5. Az algoritmus hatékonysági jellemzői

Amikor egy algoritmust keresünk egy feladat megoldására a következő két kérdés fel kell, hogy vetődjön:

1. Megoldható-e a probléma és ha igen, akkor egy vagy több megoldása van-e? (Ezt hívják a megoldás egzisztencia és unicitás problémájának.)
2. Ha már találtunk a problémára megoldási algoritmust, akkor van-e a meglévőnél hatékonyabb másik megoldási algoritmus? (A megoldási módszer, algoritmus effektívitási problémája.)

A második kérdés csak akkor jogos, ha a megoldás létezik. Meghatározásra szorul az, hogy mit értünk egy megoldó algoritmus hatékonyságán, mikor mondhatjuk, hogy egy probléma egyik megoldó algoritmusa hatékonyabb, mint a másik. Az is tisztázandó, hogy milyen szempont szerint tekintjük a hatékonyságot. A hatékonyságot mérőszámmal lehet jellemezni. Kiválasztva a mérlegelési szempontot, amely alapján az algoritmust vizsgáljuk, az

algoritmus minden bemenő adatára egy mérőszámot konstruálunk. Az az algoritmus a hatékonyabb egy rögzített input esetén, amelyikre ez a mérőszám a jobb eredményt adja. Mérlegelési szempontnak általában az algoritmus időigényét (lépésszámát, műveletszámát) szokás tekinteni, hiszen az időnek vagyunk általában szűkében. Nem elhanyagolható azonban egy másik szempont sem, az algoritmus tárigénye a számítógépes realizáció szempontjából. Egy algoritmus lehet egy bizonyos inputra jobb, mint egy másik algoritmus, egy másik input esetében pedig lehet rosszabb. Ezért a hatékonysági mérőszám fogalmát egy kicsit árnyaltabban kell megközelíteni. Először is be kell vezetni az inputok összehasonlítására alkalmas valamiféle mérőszámot. Teljesen nyílvánvaló, hogy például egy százjegyű számból általában tovább tart négyzetgyököt vonni, mint egy tízjegyűből. Be kell tehát vezetni a probléma méretének a fogalmát.

2.24. definíció. Az algoritmus inputjának a mérete (problémaméret)
Legyen adott egy probléma, amely megoldható egy A algoritmussal. Legyen D az A algoritmus lehetséges inputjainak a halmaza. Legyen $x \in D$ egy input. Az x input méretének nevezzük az x konkrét megadásakor használt bitek számát. Ez egy nemnegatív egész szám, mérőszám. Jelölésben az x input mérete $|x|$.

Meglepő, de ez a bizonytalannak, nem egészen egyértelműnek tűnő méretdefiníció mégis hatékony fogalomnak bizonyul.

2.25. példa. A négyzetgyökvonó algoritmusunk inputja legyen az egyszerűség kedvéért az s pozitív egész szám, amelyből gyököt akarunk vonni. Ekkor az algoritmus inputjának mérete $|s| = \lfloor \log_2(s) \rfloor + 1$, a számot leíró bitek száma.

Vezessünk be most néhány jelölést. Legyen A egy algoritmus, D az algoritmus összes lehetséges input adatainak a halmaza és x egy lehetséges input. Az x input esetén $t_A(x)$-szel fogjuk jelölni az A algoritmus probléma megoldási időigényét (t-time, idő) és $s_A(x)$-szel a tárigényét (s-storage, tár).

2.26. definíció. Az algoritmus időbonyolultsága
A

$$T_A(x) = \sup_{\substack{x \in D \\ |x| \leq n}} t_A(x)$$

számot az A algoritmus időbonyolultságának nevezzük.

Az időbonyolultság megadja, hogy az n-nél nem nagyobb méretű inputok esetén mennyi a legnagyobb időigény.

2.27. definíció. Az algoritmus tárkapacitás bonyolultsága
Az

$$S_A(x) = \sup_{\substack{x \in D \\ |x| \leq n}} s_A(x)$$

számot az A algoritmus tárkapacitás bonyolultságának nevezzük.

A tárkapacitás bonyolultság megadja, hogy az n-nél nem nagyobb méretű inputok esetén mennyi a legnagyobb tárigény.

Mindkét mérőszám hatékonysági mérőszám. A gyakorlatban ma már inkább az elsőt használják algoritmusok hatékonyságának az összehasonlításában, ami nem csökkenti a második szerepének a fontosságát. Elég nagy méretű tárak állnak ma már rendelkezésre, de nincs olyan tár, amit pillanatok alatt ki ne lehetne nőni egy "ügyes" algoritmussal. Láthatóan a bonyolultságok a probléma méretének monoton növekedő függvényei és az adott méretet meg nem haladó méretű esetek közül a legrosszabb esettel jellemzik az algoritmust. Ha ugyanazon probléma megoldására két vagy több algoritmus is létezik, akkor a közülük történő választás megalapozásához ad segítséget az algoritmusok bonyolultsági függvényeinek a vizsgálata, összehasonlítása. Az egyes bonyolultságok összehasonlítása az egyes függvények **növekedési ütemének, rendjének** az összehasonlítását jelenti, mely fogalmat alább definiáljuk. A fenti mérőszámoknak létezik olyan kevésbé pesszimista változata is amikor a legrosszabb eset helyett az inputok szerinti átlagolt értéket vesszük, vagy ami realisztikusabb, hogy ismerve az egyes inputok gyakoriságát (valószínűségét) súlyozott átlagot (várható értéket) számolunk. Ez utóbbi nem tartozik az anyagunkhoz, mivel valószínűség-számítási ismereteket (sajnos) nem tételezünk föl.

2.6. A növekedési rend fogalma, az ordo szimbolika

Egy algoritmus időbonyolultsága (vagy akár a tárkapacitás bonyolultsága) az input méretének monoton növekvő függvénye. Az ilyen függvényeket **növekedést leíró függvények**nek (röviden növekedési függvény) nevezzük.

2.28. példa. A kézi négyzetgyökvonás algoritmusa esetén ha az input az s (egész szám), akkor az input mérete $n = \lfloor \log_2(s) \rfloor + 1$. A négyzetgyököt csak egész jegy pontosságig határozzuk meg. Ebben az esetben $k = \lceil n/2 \rceil$ számú számjegyet kell meghatározni. Minden új számjegy esetén eggyel nő a visszaszorzandó szám jegyeinek a száma, tehát lineárisan nő minden lépésben a műveleti idő. A műveleti idők összege ezáltal $c \cdot (1 + 2 + \cdots + k) = c \cdot (k(k+1))/2$ időegység, ahol c az egy számjegyre eső műveleti idő. Az input méretével ez kifejezve: $T_A(n) = c \cdot (\lceil n/2 \rceil (\lceil n/2 \rceil + 1))/2$. Láthatóan ez az n-től függő függvény monoton növekvő.

Az egyes függvények növekedését a növekedés rendjével jellemezzük, amely valamely előre rögzített függvényhez (etalonhoz) történő hasonlítást jelent. A hasonlítást az alábbi úgynevezett ordo szimbolika által előírt módon végezzük el. Legyen $f, g : \mathbb{N} \to \mathbb{Z}$ két növekedést leíró függvény.

2.29. definíció. Az ordo szimbolika szimbólumai
Azt mondjuk, hogy az $f(n)$ függvény növekedési rendje:

nagy ordo g(n), ha létezik olyan pozitív c konstans és pozitív n_0 probléma küszöbméret, hogy ha n a probléma mérete egyenlő a küszöbmérettel, vagy annál nagyobb, akkor az $f(n)$ függvényérték nemnegatív és a $g(n)$ függvényérték c konstansszorosától nem nagyobb. Tömören:

$f(n) = O(g(n))$, ha létezik $c > 0$ és $n_0 > 0$, hogy minden $n \geq n_0$ esetén $0 \leq f(n) \leq cg(n)$.

nagy omega g(n), ha létezik olyan pozitív c konstans és pozitív n_0 probléma küszöbméret, hogy ha n a probléma mérete egyenlő a küszöbmérettel, vagy annál nagyobb, akkor az $f(n)$ függvényérték legalább akkora, mint a nemnegatív $g(n)$ függvényérték c konstansszorosa. Tömören:

$f(n) = \Omega(g(n))$, ha létezik $c > 0$ és $n_0 > 0$, hogy

minden $n \geq n_0$ esetén $0 \leq cg(n) \leq f(n)$.

nagy teta g(n), ha léteznek olyan pozitív c_1 és c_2 konstansok és pozitív n_0 probléma küszöbméret, hogy ha n a probléma mérete egyenlő a küszöbmérettel, vagy annál nagyobb, akkor az $f(n)$ függvényérték a nemnegatív $g(n)$ függvényérték c_1 és c_2-szerese által meghatározott zárt intervallumból nem lép ki. Tömören:

$f(n) = \Theta(g(n))$, ha létezik $c_1, c_2 > 0$ és $n_0 > 0$,hogy

minden $n \geq n_0$ esetén $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$.

kis ordo g(n), ha minden pozitív c konstanshoz létezik olyan pozitív n_0 probléma küszöbméret, hogy ha n a probléma mérete egyenlő a küszöbmérettel, vagy annál nagyobb, akkor az $f(n)$ függvényérték nemnegatív és a $g(n)$ függvényérték c konstansszorosától kisebb. Tömören:

$f(n) = o(g(n))$, ha minden $c > 0$-hoz létezik egy $n_0 > 0$, hogy

minden $n \geq n_0$ esetén $0 \leq f(n) \leq cg(n)$.

kis omega g(n), ha minden pozitív c konstanshoz létezik olyan pozitív n_0 probléma küszöbméret, hogy ha n a probléma mérete egyenlő a küszöbmérettel, vagy annál nagyobb, akkor az $f(n)$ függvényérték nagyobb, mint a nemnegatív $g(n)$ függvényérték c konstansszorosa. Tömören:

$f(n) = \omega(g(n))$, ha minden $c > 0$-hoz létezik egy $n_0 > 0$, hogy

minden $n \geq n_0$ esetén $0 \leq cg(n) \leq f(n)$.

Az $f(n) = O(g(n))$ és a többi jelölés tulajdonképpen nem szerencsés, mert azt sugalmazza, mintha itt két függvény, az f és a g, valamilyen közelségé-

ről, egyezőségéről lenne szó. Valójában az egyenlőségjel jobboldalán nem egy függvény, hanem egy általa leírt függvényosztály, függvények egy halmaza áll. A baloldalon álló egyetlen függvény nem lesz egyenlő egy halmazzal. Szerencsésebb lenne az egyenlőségjel helyett a halmaz eleme ($\in$) jelet használni, jelezve, hogy az f függvény olyan tulajdonságú, mint a g által definiált függvények. Tradicionális okok miatt azonban megmaradunk az egyenlőségjel használata mellett.

A gyakorlatban gyakran előforduló fontos jellemző növekedések

Konstans	$f(n) = \Theta(1)$
Lineáris	$f(n) = \Theta(n)$
Négyzetes	$f(n) = \Theta(n^2)$
Köbös	$f(n) = \Theta(n^3)$
Polinomiális	$f(n) = \Theta(n^k) \qquad k \in \mathbb{R}$ és $k > 0$
Logaritmikus	$f(n) = \Theta(\log(n))$
Exponenciális	$f(n) = \Theta(a^n) \qquad a > 1$

2.30. definíció. A polinomiálisan gyorsabb növekedés
Azt mondjuk, hogy az $f(n)$ növekedési függvény polinomiálisan gyorsabban nő, mint az n^p polinom ($p \geq 0$), ha létezik olyan $\varepsilon > 0$ valós szám, hogy $f(n) = \Omega\left(n^{p+\varepsilon}\right)$.

2.31. definíció. A polinomiálisan lassabb növekedés
Azt mondjuk, hogy az $f(n)$ növekedési függvény polinomiálisan lassabban nő, mint az n^p polinom ($p \geq 0$), ha létezik olyan $\varepsilon > 0$ valós szám, hogy $f(n) = O\left(n^{p-\varepsilon}\right)$.

2.32. példa. Az előbb tárgyalt $f(n) = T_A(n) = c \cdot (\lceil n/2 \rceil (\lceil n/2 \rceil + 1))/2$ függvény kvadratikus (négyzetes) növekedésű. Ezt a következőképpen láthatjuk be. Igaz az, hogy $x - 1 \leq \lceil x \rceil \leq x + 1$. Ezáltal fennáll a következő egyenlőtlenség

$$c \cdot (n/2-1)((n/2-1)+1)/2 \leq f(n) \leq c \cdot (n/2+1)((n/2+1)+1)/2 \quad (2.1)$$

A feladatunk olyan c_1, c_2 pozitív konstansokat és pozitív n probléma küszöbméretet mutatni, melyekre $n \geq n_0$ esetén

$$c_1 n^2 \leq c \cdot (n/2-1)((n/2-1)+1)/2 \leq f(n) \leq c \cdot (n/2+1)((n/2+1)+1)/2 \leq c_2 n^2$$

fennáll. Először a baloldalra keressük meg a megfelelő konstansokat. Kis átalakítás után az alábbi egyenlőtlenséget kapjuk:

$$0 \leq \left(\frac{c}{8} - c_1\right) n^2 - \frac{2c}{8} n$$

Feltételezve, hogy $n > 0$, leoszthatunk vele. A kapott egyenlőtlenséget n -re megoldva:

$$n \geq \frac{2c}{c - 8c_1}$$

Pozitív értéket itt akkor kapunk, ha $c_1 < c/8$. Kényelmes választás lehet a $c_1 = c/16$. Ekkor (2.1)-ből az $n \geq 4$ adódik. A jobboldalra az algebrai átalakítások után az adódó egyenlőtlenség:

$$0 \leq \left(c_2 \frac{c}{8}\right) n^2 - \frac{6c}{8} n - c$$

Pozitív problémaméret küszöböt akkor remélhetünk, ha az n^2 együtthatójára fennáll, hogy $c_2 \frac{c}{8} > 0$, azaz $c_2 > \frac{c}{8}$. Válasszuk a $c_2 = c/4$ értéket. Az (2.1)-ben szereplő másodfokú kifejezés alakja ekkor:

$$\frac{c}{8} n^2 - \frac{6c}{8} n - c$$

Ennek pozitív gyöke:

$$n = \frac{\frac{6c}{8} + \sqrt{\left(-\frac{6c}{8}\right) - 4\frac{c}{8}(-c)}}{2\frac{c}{8}} = 3 + \sqrt{17} \approx 7,123\ldots$$

Tehát ebben az esetben az $n_0 \geq 8$ választás megfelelő. A két oldal elemzését összevetve a keresett megfelelő konstansok lehetnek: $c_1 = \frac{c}{16}$, $c_2 = \frac{c}{4}$, $n = 8$. Ez alapján teljesül az, hogy ha a probléma mérete legalább 8, akkor $\frac{c}{16}n^2 \leq f(n) \leq \frac{c}{2}n^2$. Tehát valóban az algoritmusunk időigénye $T_A(n) = \Theta(n^2)$

2.33. példa. Bizonyítsuk be, hogy $3n - 3 = \Theta(n)$!

Bizonyítás: Ha az állítás igaz, akkor léteznie kell két pozitív c_1, c_2 konstansnak, melyekre valamely pozitív n_0-tól kezdve igaz, hogy $c_1 n \leq 3n - 3 \leq c_2 n$. Itt n-nel leosztva $c_1 \leq 3 - 3/n$ és $3 - 3/n \leq c_2$ egyenlőtlenségek adódnak. Látszik, hogy $0 < c_1 < 3$ kell legyen. Válasszuk $c = 2$ értéket. Ebben az esetben $2 \leq 3 - 3/n$ miatt $n \geq 3$ adódik. Tehát n lehet például 3. A másik egyenlőtlenségből pedig mivel $3 - 3/n$ mindig kisebb, mint 3, ezért c_2 lehet 3, vagy annál nagyobb szám, n_0 pedig lehet tetszőleges. Összegezve: $c_1 = 2$, $c_2 = 3$, és $n_0 = 3$ megfelelő választás, azaz ha $n \geq 3$, akkor $2n \leq 3n - 3 \leq 3n$.

2.34. példa. Bizonyítsuk be, hogy $2n^2 - n = \omega(n)$!

Bizonyítás: A definícó értelmében legyen c tetszőleges pozitív konstans. Keressünk hozzá pozitív n_0-at, amelytől kezdve $0 < cn\ 2n^2 - n$ fennáll. Itt n-nel leosztva $0 < c < 2n - 1$ adódik. Átrendezéssel $n > (c+1)/2$, amiből az $n = \lceil (c+1)/2 \rceil$ megfelelő választás a definíció kielégítésére, azaz, ha $c > 0$, akkor a $\lceil (c+1)/2 \rceil$-nél nagyobb n problémaméretek esetén $0 < cn\ 2n^2 - n$

2.35. példa. $\log(n) = o(n^p)$ bármely $p > 0$ esetén, azaz a logaritmus függvény lassabban nő, mint bármely pozitív kitevőjű hatványfüggvény. Ennek belátására meg kell mutatnunk, hogy bármely pozitív c konstans esetén valamely n küszöbtől kezdve $\log(n) < cn^p$. Ez minden bizonnyal fennáll, ha belátjuk, hogy $\lim_{n\to\infty} \frac{\log(n)}{n^p} = 0$. Ekkor ugyanis a limesz definíciójának megfelelően a hányadosnak valamely n_0 küszöbtől kezdve c alá kell csökkenni akármilyen kicsi pozitív szám is ez a c. Természetesen az n_0 küszöb c-től függ.

A határérték kiszámításához az n-et először helyettesítjük a valós x-szel és arra számítjuk a limeszt. A $\lim_{x\to\infty} \frac{\log(x)}{x^p}$ típusa $\left(\frac{\infty}{\infty}\right)$, így az analízisből ismert *L'Hospital szabályt* alkalmazva $\lim_{x\to\infty} \frac{\log(x)}{x^p} = \lim_{x\to\infty} \frac{1/x}{px^{p-1}} = \lim_{x\to\infty} \frac{1}{px^p} = 0$. Az is látható innen, hogy $\log(n)$ nemcsak lassabban nő, mint n^p, hanem polinomiálisan lassabban nő, hiszen $0 < \varepsilon < p$ esetén $n^{p-\varepsilon}$-tól is lassabban nő

2.7. A Fibonacci számok

2.36. definíció. Fibonacci sorozat
Fibonacci számsorozatnak nevezzük azt az $F_0, F_1, F_2, \ldots$ számsorozatot, amelyet az alábbi formulapár határoz meg:

$$\left.\begin{array}{rcl} F_0 & = & 1 \\ F_1 & = & 1 \end{array}\right\} \text{(Kezdőfeltétel)} \tag{2.2}$$

$$F_{n+2} = F_{n+1} + F_n \qquad \text{(rekurziós feltétel)} \tag{2.3}$$

A (2.2) és (2.3) formulapár által előállított számok sorozata így kezdődik:

Számok	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	$\cdots$
Jelölés	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	$\cdots$

A probléma a fenti definícióval az, hogy az n indexű elemet nem tudjuk a megelőzőek nélkül kiszámítani a rekurziós formula alapján. Mennyi például F_{100}-nak az értéke? Ennek a problémának a megoldását adja a Binet formula.

2.37. tétel. ***Binet formula***
A Fibonacci számsorozat elemei felírhatók az index függvényében az alábbi úgynevezett Binet formula révén:

$$F_n = \frac{1}{\sqrt{5}}\left(\Phi^n - \overline{\Phi}^n\right) \qquad n = 0, 1, 2, \ldots \tag{2.4}$$

ahol

$$\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618 \qquad \textit{és} \qquad \overline{\Phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$$

Bizonyítás. Keresnünk kell olyan számsorozatot, amely az (2.2), (2.3) feltételeknek megfelel. Könnyebb lesz a dolgunk, ha egyelőre az (2.2) feltételtől eltekintünk. A trükk: keressünk olyan $\{a_n\}$ sorozatokat, amelyek tudják a (2.3) tulajdonságot és alakjuk $a_n = z^n$ valamilyen z számra. A megoldások közül zárjuk ki a z=0 esetet, mint érdektelent, hiszen ez csupa zérust ad és az nekünk biztosan nem jó. A (2.3) tulajdonságot felírva a_n-re

$$z^{n+2} = z^{n+1} + z^n, \qquad n = 0, 1, 2, \ldots \tag{2.5}$$

adódik, ahonnan z^n-nel leosztva a

$$z^2 = z + 1, \qquad n = 0, 1, 2, \ldots \tag{2.6}$$

összefüggésre jutunk. Ennek a másodfokú egyenletnek két valós megoldása van: $z_1 = \Phi$ és $z_2 = \overline{\Phi}$. Tehát az $a_n = z_1^n$ megoldás. Könnyen meggyőződhetünk behelyettesítéssel (2.3)-be, hogy akkor az $a_n = C_1 z_1^n$ is megoldás, ahol C_1 tetszőleges konstans. Ugyanígy mivel $a_n = z_2^n$ megoldás, akkor $a_n = C_2 z_2^n$ is az. Itt C_2 szintén tetszőleges konstans. Azt kaptuk, hogy a megoldások konstansszorosai is megoldások. Vegyük észre továbbá, hogy az

$$a_n = C_1 z_1^n + C_2 z_2^n \tag{2.7}$$

is megoldás.
Összegezve: a z_1^n és a z_2^n úgynevezett alapmegoldások (bázismegoldások) lineáris kombinációi is megoldást adnak. Helyettesítsük be (2.7)-ot (2.2)-be az $n = 0$ és $n = 1$ esetre. Ezzel csempésszük vissza a kezdetben elhanyagolt (2.2) feltételt. Az alábbi kétismeretlenes, két egyenletből álló lineáris egyenletrendszert kapjuk, melyben az ismeretlenek C_1 és C_2.

$$\begin{array}{rcrcl} C_1 & + & C_2 & = & 0 \\ C_1\Phi & + & C_2\overline{\Phi} & = & 1 \end{array}$$

Innen C_1-et és C_2-t kifejezve kapjuk, hogy

$$C_1 = \frac{1}{\sqrt{5}}, \qquad C_2 = -\frac{1}{\sqrt{5}}.$$

□

2.38. tétel. A Fibonacci számok előállítása kerekítéssel
Az F_n Fibonacci szám képezhető az alábbi módon a Binet formula felhasználásával:

$$F_n = \text{Round}\left(\frac{1}{\sqrt{5}}\Phi^n\right) \qquad n = 0, 1, 2, \ldots \tag{2.8}$$

Bizonyítás. Belátjuk, hogy $\frac{1}{\sqrt{5}}\Phi^n$ és F_n között kevesebb, mint $\frac{1}{2}$ az eltérés. Ez azt jelenti, hogy a kerekítendő $\frac{1}{\sqrt{5}}\Phi^n$ szám köré fel tudunk rajzolni egy szimmetrikus intervallumot, amelynek a szélessége kevesebb, mint egy. Ekkor azonban csak egyetlen egész szám lehet ebben az intervallumban, ami így szükségszerűen éppen a Fibonacci szám lesz.

$$\begin{aligned}\left|F_n - \frac{1}{\sqrt{5}}\Phi^n\right| &= \left|\left(\frac{1}{\sqrt{5}}\left(\Phi^n - \overline{\Phi}^n\right)\right) - \frac{1}{\sqrt{5}}\Phi^n\right| = \left|-\frac{1}{\sqrt{5}}\overline{\Phi}^n\right| \\ &\leq \frac{1}{\sqrt{5}}\left|\overline{\Phi}^n\right| \leq \frac{1}{\sqrt{5}}\left|\overline{\Phi}\right|^n \leq \frac{1}{\sqrt{5}} \leq \frac{1}{2}.\end{aligned}$$

Ugyanis $|\overline{\Phi}| \leq 1$ és $\sqrt{5} \geq 2$. Tehát

$$\frac{1}{\sqrt{5}}\Phi^n - \frac{1}{2} \leq F_n \leq \frac{1}{\sqrt{5}}\Phi^n + \frac{1}{2}$$

amiből látszik, hogy F_n egybeesik az egyetlen egésszel, amely a megadott intervallumban van. □

A Fibonacci számok sorozata exponenciálisan növekszik. Ez következik a $F_n = \text{Round}\left(\frac{1}{\sqrt{5}}\Phi^n\right)$ $n = 0, 1, 2, \ldots$ előállításból.

$$F_n \approx 0.447213595 \cdot 1.618033989^n.$$

Írhatjuk, hogy $F_n = \Theta\left(\Phi^n\right)$.

2.8. A rekurzív egyenletek és a mester tétel

Az algoritmusok időbonyolultságának elemzésekor sok esetben nem rendelkezünk közvetlen explicit formulával, amely megadná, hogy a méret függvényében hogyan alakul az algoritmus időigénye a legrosszabb esetben. Ismeretes lehet viszont az egyes méretek időigényei közötti kapcsolat, mivel ezt általában könnyebb felírni. Ennek a kapcsolatnak az ismeretében megpróbálhatjuk meghatározni vagy a függvényt explicite, vagy legalább annak aszimptotikus viselkedését.

2.39. definíció. Rekurzív egyenlet
Rekurzív egyenletnek nevezzük azokat a függvényegyenleteket, amelyekben a T függvény az ismeretlen, a meghatározandó, és a $T(n)$ függvényérték a T függvény n-től kisebb értékű argumentumának helyein felvett értékeinek függvényeként adott.

A rekurzív egyenlet a $T(n)$ függvényértéket a korábbi (n-től kisebb helyen) felvett érték(ek) függvényére vezeti vissza. Ebben az esetben remélhetjük, hogy ha ismert az n első néhány értékére a $T(n)$ értéke, akkor a nagyobb n esetére a $T(n)$ már fokozatosan kiszámítható. Ha megadjuk ezeket az első értékeket, akkor azokat **kezdőfeltételek**nek (vagy **kezdetiérték feltételek**nek) nevezzük.

2.40. példa. Legyen $T(n) = q \cdot T(n-1)$, akkor a geometriai sorozat definícióját kapjuk, mint a rekurzív egyenlet speciális esetét. Ennek megoldása triviálisan látszik, hogy $T(n) = q^{n-1}T(1)$, ami $T(1)$ ismeretében egyértelműen meghatározott.

2.41. példa. Legyen $T(n) = T(n-1)+d$, akkor a számtani sorozat (aritmetikai sorozat) definícióját kapjuk, mint a rekurzív egyenlet speciális esetét. Ennek megoldása triviálisan látszik, hogy $T(n) = T(1)+(n-1)\cdot d$, ami $T(1)$ ismeretében egyértelműen meghatározott.

2.42. példa. Legyen $T(n) = T(n-1) + T(n-2)$ és $T(0) = 0$, $T(1) = 1$, akkor a megoldás a Fibonacci sorozat.

A rekurzív egyenletek megoldására néhány módszert mutatunk. Egyikük a visszavezetési módszer. Nem megyünk bele bonyolult elméleti fejtegetésekbe, lényegét példákon keresztül szemléltetjük.

2.43. példa. Legyen $T(n) = T(n-1) + 1$ és $T(1) = 1$. A visszavezetési módszer abban áll, hogy rendre a $T(n)$ függvényértékeket visszavezetjük a kezdőfeltételre rekurzív behelyettesítéssel. A mi esetünkben ez így néz ki:

$$\begin{aligned} T(1) &= 1 \\ T(2) &= T(1) + 1 = 1 + 1 = 2 \\ T(3) &= T(2) + 1 = 2 + 1 = 3 \\ &\dots \end{aligned}$$

Az ilyen időbonyolultságú algoritmus a méret egységnyi növekedésére az idő egységnyi növekedésével válaszol. A megoldás megsejthetően $T(n) = n$. Ez teljes indukcióval be is bizonyítható. Látható az is, hogy aszimptotikusan $T(n) = \Theta(n)$ azaz az algoritmus lineáris idejű.

2.44. példa. Legyen $T(n) = T(n-1) + n$ és $T(1) = 1$. Megoldása visszavezetéssel:

$$\begin{aligned} T(1) &= 1 \\ T(2) &= T(1) + 2 = 1 + 2 = 3 \\ T(3) &= T(2) + 3 = 3 + 3 = 6 \\ &\dots \end{aligned}$$

Megsejthető és teljes indukcióval belátható, hogy $T(n) = n(n+1)/2$. Ebből pedig következik, hogy aszimptotikusan $T(n) = \Theta(n^2)$, azaz az algoritmus kvadratikus idejű.

2.45. példa. Legyen $T(n) = T(n/2) + 1$ és $T(1) = 1$. Megoldása visszavezetéssel:

$$\begin{aligned} T(1) &= 1 \\ T(2) &= T(1) + 1 = 1 + 1 = 2 \\ T(3) &= ? \text{ A } T \text{ függvény itt nincs értelmezve!} \\ T(4) &= T(2) + 1 = 2 + 1 = 3 \\ T(5) &= ?, T(6) = ?, T(7) = ? \\ T(8) &= T(4) + 1 = 3 + 1 = 4 \\ &\dots \end{aligned}$$

A függvény csak az $n = 2^m$ alakú n-re van értelmezve, ahol $m = 0, 1, 2 \dots$. Ezekre azt írhatjuk, hogy $T(2^m) = T(2^{m-1}) + 1$. Bevezetve az $U(m) = T(2^m)$ jelölést azt kapjuk, hogy $U(m) = U(m-1) + 1, U(0) = 1$. Egy nagyon hasonló feladatot az 2.43. Példában megoldottunk és onnan a megoldás $U(m) = m + 1$. Akkor $T(2^m) = m + 1$. Figyelembe véve, hogy $m = \log_2(n)$, a végeredmény $T(n) = \log_2(n) + 1$ és $T(n) = \Theta(\log_2 n)$, azaz az algoritmus logaritmikus idejű.

2.46. példa. Legyen a rekurzív egyenlet $T(n) = T(\lceil n/2 \rceil) + 1$, a kezdőfeltétel $T(1) = 1$. Megoldása visszavezetéssel:

$$\begin{aligned} T(1) &= 1 \\ T(2) &= T(1) + 1 = 1 + 1 = 2 \\ T(3) &= T(2) + 1 = 2 + 1 = 3 \\ T(4) &= T(2) + 1 = 2 + 1 = 3 \\ T(5) &= T(3) + 1 = 3 + 1 = 4 \\ T(6) &= T(3) + 1 = 3 + 1 = 4 \\ T(7) &= T(4) + 1 = 3 + 1 = 4 \\ T(8) &= T(4) + 1 = 3 + 1 = 4 \\ &\dots \end{aligned}$$

A megoldás a kettő egész hatványainak megfelelő helyeken egybeesik az előző feladat megoldásával. A többi helyen ellentétben az előző feladattal a megoldás értelmezve van. Megsejthető, hogy a megoldás $T(n) = \lceil \log_2(n) \rceil + 1$. Erre is igaz az aszimptotika, hogy $T(n) = \Theta(\log_2 n)$.

2.47. példa. Olyan esetet vizsgálunk, amelyben a megoldást nem tudjuk megsejteni, de az aszimptotikát igen és teljes indukcióval bizonyítunk. Legyen $T(n) = T\left(\left\lceil\frac{n}{4}\right\rceil\right) + T\left(\left\lceil\frac{n}{2}\right\rceil\right) + 2n$. Az aszimptotikus megoldás sejthetően $T(n) = O(n)$, azaz van olyan $c > 0$ és $n_0 > 0$, hogy ha $n \geq n_0$, akkor $0 \leq T(n) \leq c \cdot n$. Helyettesítsük be az aszimptotikus sejtett megoldást az egyenletbe.

$$\begin{aligned} T(n) &= c \cdot \left\lceil\frac{n}{4}\right\rceil + c \cdot \left\lceil\frac{n}{2}\right\rceil + 2n \leq c \cdot \left(\frac{n}{4}+1\right) + c \cdot \left(\frac{n}{2}+1\right) + 2n = \frac{3}{4}cn + 2n + 2c \\ &= cn - \frac{1}{4}cn + 2n + 2c = cn - (\frac{1}{4}cn - 2n - 2c) \end{aligned}$$

Ha most n-et úgy választjuk meg, hogy a zárójelben lévő kifejezés nemnegatív legyen, akkor megkapjuk az áhított $T(n) \leq c \cdot n$ egyenlőtlenséget, miáltal a bizonyítás be is fejeződik. Az $\frac{1}{4}cn - 2n - 2c \geq 0$ feltételezésből $c \geq \frac{2n}{\frac{1}{4}n - 2} = \frac{8n}{n-8}$ következik. Válasszuk most mondjuk az $n = 9$-et, akkor $\frac{8 \cdot 9}{9-8} = 72$. Ebben az esetben a $c \geq 72$ teljesül, tehát ha $n \geq n_0 = 9$, akkor mindig fennáll $T(n) \leq c \cdot n$, ahol a c helyébe a 72, vagy bármilyen tőle nagyobb szám írható. Ezzel beláttuk, hogy a megoldás növekedési rendje valóban $T(n) = O(n)$. Meg lehet mutatni, hogy még $T(n) = \Theta(n)$ is fennáll.

Speciális, de a gyakorlat szempontjából sok fontos esetben a rekurzív egyenletekre az alább megfogalmazott úgynevezett mester tétel ad aszimptotikus megoldást (sok esetben meg nem ad).

2.48. tétel. A mester tétel
Legyenek $a \geq 1, b > 1$ konstansok, $p = \log_b a$, $f : \mathbb{N} \to \mathbb{Z}$ függvény.
Definiálunk egy $g(n) = n^p$ úgynevezett tesztpolinomot.
Legyen a rekurziós összefüggésünk: $T(n) = aT(n/b) + f(n)$. (Az n/b helyén $\lceil n/b \rceil$ vagy $\lfloor n/b \rfloor$ is állhat.) Ezen feltételek esetén igazak az alábbi állítások:

1. *Ha $f(n)$ polinomiálisan lassabb növekedésű, mint a $g(n)$ tesztpolinom, akkor*
$$T(n) = \Theta(g(n)).$$
2. *Ha $f(n) = \Theta(g(n))$, akkor*
$$T(n) = \Theta(g(n) \cdot \log n).$$

3. Ha $f(n)$ polinomiálisan gyorsabb növekedésű, mint a $g(n)$ tesztpolinom, és teljesül az f függvényre az úgynevezett regularitási feltétel, azaz $\exists c < 1$ konstans és $n_0 > 0$ küszöbméret, hogy $n > n_0$ esetén $a \cdot f(n/b) \leq c \cdot f(n)$, akkor

$$T(n) = \Theta(f(n)).$$

A tételt nem bizonyítjuk.

2.49. példa. A mester tétel alkalmazása: $T(n) = 9 \cdot T(n/3) + n$. A mester tétel szerintinek tűnik az eset. Legyen $a = 9, b = 3, f(n) = n, p = \log_3 9 = 2$. A tesztpolinom $g(n) = n^2$. Láthatóan $f(n)$ polinomiálisan lassabban nő, mint $g(n)$, mert minden $0 < \varepsilon < 1$ esetén $f(n) = O(n^{2-\varepsilon})$. Teljesülnek a mester tétel 1. pontjának a feltételei, tehát a tételt alkalmazva: $T(n) = \Theta(n^2)$.

2.50. példa. A mester tétel alkalmazása: $T(n) = T(2n/3) + 1$. A mester tétel szerintinek tűnik az eset. Legyen $a = 1, b = 3/2, f(n) = 1, p = \log_{3/2} 1 = 0$. A tesztpolinom $g(n) = n^0 = 1$. Továbbá $1 = f(n) = \Theta(n^0) = \Theta(1)$. Teljesülnek a mester tétel 2. pontjának a feltételei, tehát a tételt alkalmazva: $T(n) = \Theta(1 \cdot \log n)$.

2.51. példa. A mester tétel alkalmazása: $T(n) = 3T(n/4) + n \log n$. A mester tétel szerintinek tűnik az eset. Legyen $a = 3, b = 4, f(n) = n \log n, p = \log_4 3, 0 < p < 1$. A tesztpolinom $g(n) = n^p$. Ha $0 < \varepsilon < 1 - p$, akkor $f(n) = \Omega(n^{p+\varepsilon})$, azaz f polinomiálisan gyorsabban nő, mint a tesztpolinom, ugyanis $\lim\limits_{n\to\infty} \dfrac{n \cdot \log n}{n^{p+\varepsilon}} = \infty$. A mester tétel 3. pontjának a feltételei fognak teljesülni, ha még kimutatjuk az f regularitását. Ez fennáll a következők szerint $3f(n/4) = 3(n/4)\log(n/4) = (3/4)n \log n - (3/4)n \log 4 \leq (3/4)n \log n = (3/4)f(n)$. Azaz minden pozitív n-re teljesül a regularitás $c = 3/4 < 1$-gyel. A tételt alkalmazva: $T(n) = \Theta(f(n)) = \Theta(n \log n)$.

2.52. példa. A mester tétel nem alkalmazható: $T(n) = 2T(n/2) + n \log n$. A mester tétel szerintinek tűnik az eset. Legyen $a = 2, b = 2, f(n) = n \log n, p = \log_2 2 = 1$. A tesztpolinom $g(n) = n^1 = n$. Az igaz, hogy legalább olyan gyorsan nő, mint a tesztpolinom, mert $\lim_{n\to\infty} \frac{n \cdot \log n}{n} = \infty$, tehát $f(n) = \Omega(n)$. Az viszont nem igaz, hogy polinomiálisan gyorsabban nő, mint a tesztpolinom. Azért nem igaz, mert bármilyen pozitív ε esetén $\lim_{n\to\infty} \frac{n \cdot \log n}{n^{1+\varepsilon}} = 0$. Emiatt a mester tétel gyanított 3. pontja nem alkalmazható. Az aszimptotikus megoldás a mester tétellel nem adható meg. (Behelyettesítéssel be lehet bizonyítani, hogy $f(n) = \Theta(n)$.)

3. fejezet

Számelméleti algoritmusok

3.1. Alapfogalmak

3.1. definíció. Az oszthatóság
Azt mondjuk, hogy a d egész szám osztja az a egész számot, ha az osztásnak zérus a maradéka, azaz, ha létezik olyan k egész szám, hogy $a = k \cdot d$. Jelölésben: $d|a$. A d számot az a **osztó**jának nevezzük. Az a szám a d **többszöröse**.

3.2. definíció. Prímszám
Prímszámnak nevezzük azt az 1-nél nagyobb egész számot, amelynek csak az 1 és saját maga az osztója.

3.3. tétel. ***A maradékos osztás tétele***
Ha a egész szám, n pedig pozitív egész szám, akkor egyértelműen létezik olyan q és r egész szám, hogy

$$a = q \cdot n + r, \qquad \textit{ahol } 0 \leq r < n.$$

A q szám neve ***hányados****, r neve* ***maradék****. A hányados és a maradék felírható:*

$$q = \lfloor a/n \rfloor, \qquad r = a - q \cdot n = a \bmod n.$$

3.4. definíció. *Közös osztó*
Azt mondjuk, hogy a d egész szám az a és b egészek közös osztója, ha d mindkét számot osztja (azaz $d|a$ és $d|b$).

3.5. definíció. Lineáris kombináció
Az s egész számot az a és b egészek (egész) lineáris kombinációjának nevezzük, ha létezik olyan x és y egész szám, hogy $s = x \cdot a + y \cdot b$. Az x és y számokat a lineáris kombináció együtthatóinak nevezzük. Az a és b számok összes lineáris kombinációjának halmazát $L(a, b)$-vel jelöljük.

Speciálisan lineáris kombinációk az $a+b$ és az $a-b$ számok is. Adott s egész előállításakor az x és y együtthatók nem egyértelműek, azaz ha léteznek, akkor több számpár is megfelelhet erre a célra, amint az az alábbi példából is látható.

3.6. példa. Legyen két szám $a = 36, b = 60$. Határozzuk meg az $s = x \cdot a + y \cdot b$ értékeket az $x = -3 \ldots 3$, $y = -3 \ldots 3$ együtthatókra!

	$s = xa + yb$				y			
	tábla	-3	-2	-1	0	1	2	3
	-3	-288	-228	-168	-108	-48	12	72
	-2	-252	-192	-132	-72	-12	48	108
	-1	-216	-156	-96	-36	24	84	144
x	0	-180	-120	-60	0	60	120	180
	1	-144	-84	-24	36	96	156	216
	2	-108	-48	12	72	132	192	252
	3	-72	-12	48	108	168	228	288

3.7. tétel. ***A közös osztó tulajdonságai***
Legyen a d egész az a és b egészek közös osztója. Akkor fennállnak az alábbi állítások:

1. $|d| \leq |a|$, *vagy* $a = 0$.

2. *Ha* $d|a$ *és* $a|d$, *akkor* $d = \pm a$.

3. *A közös osztó osztója az* a *és* b *szám minden lineáris kombinációjának is, azaz* $\forall s \in L(a,b)$*-re* $d|s$.

3.2. A legnagyobb közös osztó

3.8. definíció. *Legnagyobb közös osztó*
Az alább megadott $d*$ egész számot az a és b egész számok legnagyobb közös osztójának nevezzük. Jele: $\text{lnko}(a,b)$.

$$d^* = \text{lnko}(a,b) = \begin{cases} 0 & \text{ha } a = 0 \text{ és } b = 0 \\ \max\limits_{\substack{d|a \\ d|b}} d & \text{egyébként.} \end{cases}$$

3.9. definíció. Relatív prímek
Az a és b egész számokat relatív prímeknek nevezzük, ha $\text{lnko}(a,b) = 1$.

3.10. tétel. A legnagyobb közös osztó elemi tulajdonságai
Legyen a $d*$ *egész az* a *és* b *egészek legnagyobb közös osztója. Akkor fennállnak az alábbi állítások:*

1. $1 \leq d^* \leq \min\{|a|, |b|\}$.

2. $\text{lnko}(a,b) = \text{lnko}(b,a) = \text{lnko}(-a,b) = \text{lnko}(|a|,|b|)$.

3. $\text{lnko}(a,0) = |a|$.

4. $\text{lnko}(a, k \cdot a) = |a|, k \in \mathbb{Z}$.

5. *Ha d közös osztó és $d^* \neq 0$, akkor $d \leq d^*$.*

6. *A legnagyobb közös osztó minden lineáris kombinációnak osztója, azaz $\forall s \in L(a,b)$-re $d^*|s$.*

3.11. tétel. A legnagyobb közös osztó reprezentációs tétele
Ha az a és b egész számok nem mindegyike zérus, akkor a legnagyobb közös osztó megegyezik a két szám pozitív lineáris kombinációinak minimumával.

$$d^* = \text{lnko}(a,b) = \min_{\substack{s \in L(a,b) \\ s>0}} s = a \cdot x^* + b \cdot y^* = s^*.$$

Bizonyítás. A bizonyítás menete az lesz, hogy megmutatjuk, hogy $s^* \leq d^*$ és $d^* \leq s^*$, amiből következik az állítás.

Az $s^* \leq d^*$ megmutatása úgy történik, hogy belátjuk, hogy s^* közös osztó, ami nem lehet nagyobb, mint a legnagyobb közös osztó. Azt, hogy s^* közös osztó azáltal látjuk be, hogy az osztási maradéka zérus. Csak az a számra végezzük el, b-re ugyanígy megy a bizonyítás. Osszuk el tehát a-t s^* -gal és számítsuk ki a maradékot! Legyen q a hányados. Az r maradékra igaz, hogy $0 \leq r < s^*$. Akkor

$$\begin{aligned} 0 &\leq r < s^* \\ 0 &\leq a - q \cdot s^* < s^* \\ 0 &\leq a - q \cdot (a \cdot x^* + b \cdot y^*) < s^* \\ 0 &\leq (1 - q \cdot x^*) \cdot a + (-q \cdot y^*) \cdot b < s^* \end{aligned}$$

Az egyenlőtlenség közepén álló maradék az a és a b lineáris kombinációja. A baloldali egyenlőtlenség miatt nem lehet negatív, a jobboldali egyenlőtlenség miatt pedig kisebb, mint a pozitív lineáris kombinációk közül a legkisebb. Emiatt csak zérus lehet. Tehát az s^* osztja az a számot.

A $d^* \leq s^*$ abból következik, hogy a legnagyobb közös osztó osztja az összes lineáris kombinációt, így s^*-ot is. Ekkor azonban nem lehet nagyobb, mint s^*, hiszen annak osztója. □

A tétel következményei:

1. A közös osztó osztja a legnagyobb közös osztót, ugyanis a legnagyobb közös osztó az a és b egészek lineáris kombinációja a tétel szerint, amit a közös osztó oszt.

2. Tetszőleges n nemnegatív egészre

$$\text{lnko}(n \cdot a, n \cdot b) = n \cdot \text{lnko}(a, b).$$

Ugyanis $n = 0$-ra az állítás triviális, $n > 0$-ra pedig

$$\text{lnko}(n \cdot a, n \cdot b) = n \cdot a \cdot x^* + n \cdot b \cdot y^* = n \cdot (a \cdot x^* + b \cdot y^*) = n \cdot \text{lnko}(a, b).$$

(Lássuk be, hogy az utolsó egyenlőségjel valóban igaz, azaz a lineáris kombinációkban mindkét számpár esetén ugyanaz az x^*, y^* megfelelő választás!)

3.12. tétel. A lineáris kombinációk halmazának jellemzése
Legyen M a $d^* = \text{lnko}(a, b)$ egész többszöröseinek a halmaza. Állítás:

$$L(a, b) \equiv M.$$

Bővebben: az $L(a, b)$ minden eleme d^* egész többszöröse és ha egy s szám egész többszöröse, akkor az az s szám az a és b lineáris kombinációja is.

Bizonyítás. Megmutatjuk, hogy $L \subset M$ és $M \subset L$, amiből következik az állítás. Az $L \subset M$ eset: $d^* | a$ és $d^* | b$, amiből következik, hogy van olyan $k_a, k_b \in \mathbb{Z}$, hogy $a = k_a \cdot d^*, b = k_b \cdot d^*$. Ha $s \in L$, akkor

$$s = a \cdot x + b \cdot y = k_a \cdot d^* \cdot x + k_b \cdot d^* \cdot y = d^* \cdot \underbrace{(k_a \cdot x + k_b \cdot y)}_{=k} = k \cdot d^*,$$

mert $k \in \mathbb{Z}$.

Az $M \subset L$ eset: $d^* = \text{lnko}(a, b) = a \cdot x^* + b \cdot y^*$.
Ha $s \in M$, akkor van olyan $k_s \in \mathbb{Z}$, hogy

$$s = k_s \cdot d^* = k_s \cdot (a \cdot x^* + b \cdot y^*) = k_s \cdot x^* \cdot a + k_s \cdot y^* \cdot b \in L$$

□

3.13. tétel. A legnagyobb közös osztó redukciós tétele
Tetszőleges a és b két egész szám esetén fennáll, hogy

$$\text{lnko}(a, b) = \text{lnko}(a - b, b).$$

Bizonyítás. Legyen $d_1^* = \text{lnko}(a, b)$ és $d_2^* = \text{lnko}(a - b, b)$. Azt fogjuk bizonyítani, hogy $d_1^* | d_2^*$ és $d_2^* | d_1^*$, amiből következik a tétel állítása.

Megmutatjuk, hogy $d_1^* \in L(a - b, b)$ és $d_2^* \in L(a, b)$ is fennáll, amiből a kölcsönös oszthatóság már következik.
$d_1^* \in L(a - b, b)$ megmutatása:

$$\begin{aligned} d_1^* \in L(a, b) \quad &\Rightarrow \quad d_1^* = x_1^* \cdot a + y_1^* \cdot b = x_1^* \cdot (a - b + b) + y_1^* \cdot b \\ &= \quad x_1^* \cdot (a - b) + (x_1^* + y_1^*) \cdot b \in L(a - b, b) \end{aligned}$$

$d_2^* \in L(a, b)$ megmutatása:

$$d_2^* \in L(a - b, b) \Rightarrow d_2^* = x_2^* \cdot (a - b) + y_2^* \cdot b = x_2^* \cdot a + (y_2^* - x_2^*) \cdot b \in L(a, b)$$

□

3.3. A bináris legnagyobb közös osztó algoritmus

Az eddigiek alapján algoritmus konstruálható a legnagyobb közös osztó meghatározására. Az algoritmus neve: `Bináris lnko algoritmus`.

Az algoritmus a két nemnegatív egész bináris felírásának alakjából indul ki. Az utolsó bit alapján a kiinduló problémát fokozatosan egyszerűbbé redukálja, amíg csak az egyik szám zérussá nem válik. Ekkor a legnagyobb közös osztó a másik redukált szám egy szorzóval korrigált értéke lesz. Munka közben az algoritmus csak egyszerű, hatékony gépi műveleteket - egész kivonás és jobbra eltolás (shift) - használ.

Legyen a két szám a és b és legyen $a \geq b$. A $\text{lnko}(a, b)$ kiszámításának feladata ekkor az alábbiak szerint redukálódik az utolsó bit szerint egyszerűbb feladattá:

	2.3.1 algoritmus Bináris legnagyobb közös osztó
1	Bináris_lnko (a, b, d^*)
2	// Input paraméter: $a \in \mathbb{Z}$, $a \geq 0$
3	// $b \in \mathbb{Z}$, $b \geq 0$, $a \geq b$
4	// Output paraméter: $d^* \in \mathbb{Z}$, $d^* \geq 0$, $d^* = \text{lnko}(a, b)$
5	$c \leftarrow 1$
6	**WHILE** $a \neq 0$ és $b \neq 0$ **DO**
7	// a és b paritásértékei p_a és p_b (0 = páros, 1 = páratlan)
8	$p_a = a \bmod 2$
9	$p_b = b \bmod 2$
10	**CASE**
11	$p_a = 0, p_b = 0 : c \leftarrow 2c$
12	$a \leftarrow a/2$
13	$b \leftarrow b/2$
14	$p_a = 0, p_b = 1 : a \leftarrow a/2$
15	$p_a = 1, p_b = 0 : b \leftarrow b/2$
16	$p_a = 1, p_b = 1 : a \leftarrow (a - b)/2$
17	**IF** $a < b$ **THEN** $a \leftrightarrow b$ csere
18	**IF** $a = 0$
19	**THEN** $d^* \leftarrow c \cdot b$
20	**ELSE** $d^* \leftarrow c \cdot a$
21	**RETURN** d^*

3.14. példa. Példa az algoritmusra: $\text{lnko}(3604, 3332) = 22 \cdot 17 = 68$

Lépésszám	a		b	Korrekciós szorzó
0	3604		3332	2
1	1802		1666	2
2	901		833	
3	34	-	833	
4	833		34	
5	833		17	
6	408		17	
7	204		17	
8	102		17	
9	51		17	
10	17		17	
11	0		17	

3.4. Az euklideszi és a kibővített euklideszi algoritmus

3.15. tétel. A legnagyobb közös osztó rekurziós tétele
Tetszőleges a és b két egész szám esetén fennáll, hogy $\text{lnko}(a, b) = \text{lnko}(b, a \bmod b)$.

Bizonyítás. Az $a \bmod b$ az a-ból b ismételt kivonásaival megkapható és így a *redukciós tétel (3.13)* értelmében az állításunk igaz. □

A rekurziós tétel révén készíthető el az euklideszi algoritmus a legnagyobb közös osztó meghatározására. Az algoritmus pszeudokódja:

2.4.1. algoritmus
Euklideszi algoritmus
//
// rekurzív változat
1 Euklidesz (a, b, d^*)
2 // Input paraméter : $a \in \mathbb{Z}$, $a \geq 0$
3 // $b \in \mathbb{Z}$, $b \geq 0$
4 // Output paraméter: $d^* \in \mathbb{Z}$, $d^* \geq 0$
5 $d^* \leftarrow a$
6 IF $b \neq 0$
7 THEN Euklidesz $(b, a \bmod b, d^*)$
8 **RETURN** (d^*)

2.4.2. algoritmus
Euklideszi algoritmus
//
// iteratív változat
1 Euklidesz (a, b, d^*)
2 // Input paraméter : $a \in \mathbb{Z}$, $a \geq 0$
3 // $b \in \mathbb{Z}$, $b \geq 0$
4 // Output paraméter: $d^* \in \mathbb{Z}$, $d^* \geq 0$
5 **WHILE** $b \neq 0$ **DO**
6 $r \leftarrow a \bmod b$
7 $a \leftarrow b$
8 $b \leftarrow r$
9 $d^* \leftarrow a$
10 **RETURN** (d^*)

3.16. példa. Példa az algoritmusra: lnko$(3604, 3332) = 68$, $q = \lfloor a/n \rfloor$, $r = a - q \cdot n = a \bmod n$

Lépésszám	a	b	q	r
0	3604	3332	1	272
1	3332	272	12	68
2	272	68	4	0
3	68	0	-	68

3.17. tétel. Lamé tétele
Ha az euklideszi algoritmusban $a > b \geq 0$ és $b < F_{k+1}$ valamely $k > 0$-ra, akkor a rekurziós hívások száma kevesebb, mint k.

A tételt nem bizonyítjuk

A tétel következménye, hogy ha $F_k \leq b < F_{k+1}$, akkor a rekurziós hívások száma kevesebb, mint k, valamint becslést tudunk adni erre a k-ra közvetlenül a b-ből. A k értékére jól memorizálható becslés az, hogy k vehető a b tizes

számrendszerbeli jegyei ötszörösének. (Valójában megmutatható, hogy itt a kisebb reláció is igaz.) A meggondolás az alábbi:

$$F_k \approx \frac{1}{\sqrt{5}}\Phi^k, \qquad \Phi \approx 1,618, \qquad \log_{10}\Phi \approx 0,2089, \qquad \frac{1}{\log_{10}\Phi} \approx 4,78 \approx 5$$

$$\log_{10} F_k \approx k \cdot \log_{10}\Phi - \log_{10}\sqrt{5}$$

$$k \approx \frac{1}{\log_{10}\Phi} \cdot \log_{10} F_k + \frac{\log_{10}\sqrt{5}}{\log_{10}\Phi} \approx 5 \cdot \log_{10} F_k \approx 5 \cdot \log_{10} b$$

Bizonyítható, hogy az euklideszi algoritmusnak a legrosszabb bemenő adatai a szomszédos Fibonacci számok. Az euklideszi algoritmus időigénye $O(\log b)$ azon feltételezés mellett, hogy az aritmetikai műveletek konstans ideig tartanak függetlenül a benne szereplő számértékek nagyságától. Ha a számok nagyságát is figyelembe vesszük, akkor az időigény $O(\log a \cdot \log b)$. Az euklideszi algoritmus némi bővítéssel alkalmassá tehető arra, hogy a legnagyobb közös osztó lineáris kombinációként történő előállításában szereplő x^* és y^* együtthatókat is meghatározza. Tekintsük az euklideszi táblát.

Lépésszám	a	b	q	r
0	a_0	b_0	q_0	r_0
1	a_1	b_1	q_1	r_1
...	...	...	...	...
k	a_k	b_k	q_k	r_k
$k+1$	a_{k+1}	b_{k+1}	q_{k+1}	r_{k+1}
...	...	...	...	...
n	$d*$	0	-	$d*$

A tábla k. sorában ($k = 0, 1, \ldots, n$) a *rekurziós tétel* alapján érvényes a $d^* = x_k^* \cdot a_k + y_k^* \cdot b_k$ összefüggés, ahol továbbá

$$a_{k+1} = b_k, \qquad b_{k+1} = r_k, \qquad q_k = \lfloor a_k/b_k \rfloor, \qquad r_k = a_k - q_k \cdot b_k.$$

A k és $k+1$ indexű sorok között a kapcsolat:

$$\begin{aligned} d^* &= x_k^* \cdot a_k + y_k^* \cdot b_k = x_k^* \cdot (q_k \cdot b_k + r_k) + y_k^* \cdot b_k \\ &= (x_k^* \cdot q_k \cdot b_k + x_k^* \cdot r_k) + y_k^* \cdot b_k = (x_k^* \cdot q_k + y_k^*) \cdot b_k + x_k^* \cdot r_k \\ &= x_{k+1}^* \cdot a_{k+1} + y_{k+1}^* \cdot b_{k+1} \end{aligned}$$

Kaptunk egy összefüggést az x_k^*, y_k^* és az x_{k+1}^*, y_{k+1}^* együtthatók között az egymást követő sorokra, ha fentről lefelé haladunk a táblában.

$$\begin{aligned} x_{k+1}^* &= x_k^* \cdot q_k + y_k^* \\ y_{k+1}^* &= x_k^* \end{aligned} \tag{3.1}$$

Haladjunk most lentről fölfelé! Akkor (3.1)-ból x_k^* és y_k^* kifejezve:

$$\begin{aligned} x_k^* &= y_{k+1}^* \\ y_k^* &= x_{k+1}^* - q_k \cdot y_{k+1}^* \end{aligned}$$

Az utolsó sor esetén viszont $d^* = 1 \cdot d^* + 0 \cdot 0$, azaz $x_n^* = 1$ és $y_n^* = 0$. Az utolsó sorból indulva így visszafelé sorról-sorra haladva az x_k^* és y_k^* értékek kiszámíthatók. Végül az x_0^* és y_0^* is kiadódik. Ez a módosítás vezet az euklideszi algoritmus kibővítésére, melynek pszeudokódját alább közöljük.

	2.4.3. algoritmus **Kibővített euklideszi algoritmus**
	// rekurzív változat
1	Kibővített_Euklidesz (a, b, d^*, x^*, y^*)
2	// Input paraméterek : $a, b \in \mathbb{Z}, a, b \geq 0$
3	// Output paraméterek: $d^*, x^*, y^* \in \mathbb{Z}, d^* \geq 0$
4	**IF** $b = 0$
5	**THEN** $d^* \leftarrow a$
6	$x^* \leftarrow 1$
7	$y^* \leftarrow 0$
8	**ELSE** Kibővített_Euklidesz $(b, a \bmod b, d^*, x^*, y^*$)
9	$\begin{pmatrix} x^* \\ y^* \end{pmatrix} \leftarrow \begin{pmatrix} y^* \\ x^* - \lfloor a/b \rfloor \cdot y^* \end{pmatrix}$
10	**RETURN** (d^*, x^*, y^*)

	2.4.3. algoritmus **Kibővített euklideszi algoritmus**
	// iteratív változat
1	Kibővített_Euklidesz (a, b, d*, x*, y*)
2	// Input paraméterek : a,b∈Z, a,b≥0
3	// Output paraméterek: d*, x*, y*∈Z, d*≥0
4	$x_0 \leftarrow 1, x_1 \leftarrow 0, y_0 \leftarrow 0, y_1 \leftarrow 1, s \leftarrow 1$
5	**WHILE** $b \neq 0$
6	$r \leftarrow a \bmod b, q \leftarrow a \operatorname{div} b$
7	$a \leftarrow b, b \leftarrow r$
8	$x \leftarrow x_1, y \leftarrow y_1$
9	$x_1 \leftarrow q \cdot x_1 + x_0; y_1 \leftarrow q \cdot y_1 + y_0$
10	$x_0 \leftarrow x, y_0 \leftarrow y$
11	$s \leftarrow -s$
12	$x \leftarrow s \cdot x_0, y \leftarrow -y_0$
13	$(d^*, x^*, y^*) \leftarrow (a, x, y)$
14	**RETURN** (d^*, x^*, y^*)

3.18. példa. Példa kibővített euklideszi algoritmusra

Lépésszám	A	b	q	r	d^*	x^*	y^*
0	3604	3332	1	272	68	-12	$1 - 1 \cdot (-12) = 13$
1	3332	272	12	68	68	1	$0 - 12 \cdot 1 = -12$
2	272	68	4	0	68	0	$1 - 4 \cdot 0 = 1$
3	68	0	-	68	68	1	0

Az algoritmus eredményeképpen $x_0^* = -12$ és $y_0^* = 13$ adódott. Ellenőrzésképpen

$$68 = (-12) \cdot 3604 + 13 \cdot 3332 = -43248 + 43316,$$

ami valóban megfelel az elvárásoknak.

3.5. A lineáris kongruencia egyenlet

3.19. definíció. *Kongruencia*
Az a és b egész számokat *kongruens*nek mondjuk az n *modulus* szerint, ha az n szerinti osztás utáni maradékaik megegyeznek, vagy ami ugyanaz: ha $n \,|(a-b)$. Jelölésben: $a \equiv b \bmod n$.

3.20. tétel. A kongruenciákon végezhető műveletek tétele
Legyen $a \equiv b \bmod n$ *és* $c \equiv d \bmod n$. *Akkor igazak az alábbi állítások:*

1. $a \pm c \equiv b \pm d \bmod n$
2. $a \cdot c \equiv b \cdot d \bmod n$
3. $\dfrac{a}{k} \equiv \dfrac{b}{k} \bmod n$ *ha* $k \,|a$, $k \,|b$ *és* $\operatorname{lnko}(k,n) = 1$
4. $a \equiv b \bmod m$ *ha* $m \,|n$

3.21. definíció. *A lineáris kongruencia egyenlet*
Az

$$a \cdot x \equiv b \bmod n, \qquad a, b \in Z, n \in Z^+ \tag{3.2}$$

egyenletet, melyben $x \in Z$ az ismeretlen, *lineáris kongruencia egyenlet*nek nevezzük.

3.22. tétel. A lineáris kongruencia egyenlet megoldhatósági tétele
Legyen az (3.2) *egyenletre* $d^* = \operatorname{lnko}(a,n) = a \cdot x^* + n \cdot y^*$. *Az* (3.2) *lineáris kongruencia egyenletnek akkor és csak akkor van megoldása, ha* $d^* \,|b$. *Ha van megoldás, akkor végtelen sok van, de ezeket egy* d^* *számú megoldást tartalmazó úgynevezett megoldás alaprendszerből megkaphatjuk az* n *egész számú többszöröseinek a hozzáadásával. Az alaprendszer elemeit a* $0 \le x < n$ *intervallumból választjuk ki. Az alaprendszer megoldásai az alábbi módon írhatók fel:*

$$\begin{aligned} x_0 &= x^* \cdot (b/d^*) \bmod n \\ x_i &= x_0 + i \cdot (n/d^*) \bmod n \qquad (i = 1, 2, \ldots, d^* - 1) \end{aligned}$$

Bizonyítás. Legyen $q_1 = \left\lfloor \frac{ax}{n} \right\rfloor$, $q_2 = \left\lfloor \frac{b}{n} \right\rfloor$, $q = q_2 - q_1$. Akkor a lineáris kongruencia egyenlet $ax - q_1 n = b - q_2 n$ alakra írható át, amiből az $ax + qn = b$ egyenlet adódik, vagyis hogy a b az a és az n lineáris kombinációja. Ha azt akarjuk, hogy legyen megoldás, akkor $b \in L(a, n)$ fenn kell álljon, ahol $L(a, n)$ az a és n lineáris kombinációinak a halmaza. Ha ez nem áll fenn, akkor nincs megoldás. A lineáris kombinációban lévő elemeket viszont a $d^* = \text{lnko}(a, n)$ legnagyobb közös osztó osztja, és csak azokat osztja a lineáris kombinációk halmazának jellemzési tétele szerint. Legyen most b olyan, hogy $d^* | b$. Akkor van olyan k egész szám, hogy $b = k \cdot d^*$. A legnagyobb közös osztó viszont az a és az n lineáris kombinációja, azaz van olyan x^* és y^* egész, hogy $d^* = a \cdot x^* + n \cdot y^*$. Ez a formula viszont egyenértékű az $a \cdot x^* \equiv d^* mod\, n$ lineáris kongruencia egyenlettel, ha az n szerinti maradékokat nézzük. Beszorozva itt k-val $a \cdot x^* k \equiv d^* k \bmod n$ adódik, amiből azonnal látható, hogy az $x_0 = x^* k = x^* (b/d^*) \bmod n$ megoldás. További megoldásokat kapunk, hogyha képezzük az $x_i = x_0 + i \cdot (n/d^*) \bmod n$, $i = 1, 2, \ldots, d^* - 1$ számokat, ugyanis a lineáris kongruencia egyenletbe történő behelyettesítés után az $ax_0 + a \cdot i \cdot (n/d^*)$, $i = 1, 2, \ldots, d^* - 1$ jelenik meg a baloldalon, ahol a második tag osztható n-nel, mert a d^* az a-t osztja, így az n megmarad, tehát ez a tag nem módosítja az első tag általi maradékot. Ezeket a megoldásokat *alapmegoldások*nak nevezzük. Nyílvánvaló, hogy ha n egész többszörösét hozzáadom az alapmegoldásokhoz, akkor újra megoldást kapok, csak az már nem lesz alapmegoldás (nem viselkedik maradékként). □

A lineáris kongruencia egyenlet megoldására algoritmus konstruálható, ugyanis a kívánt x^* a kibővített euklideszi algoritmusból megkapható.

	2.5.1. algoritmus
	Lineáris kongruencia megoldó
1	Lineáris_kongruencia_megoldó (a, b, n, X)
2	// Input paraméterek: a,b,n∈Z, n>0
3	// Output paraméter : X – egyindexes tömb
4	// indexelés 0-tól
5	Kibővített_Euklidesz (a, n, d^*, x^*, y^*)
6	Hossz[X]← 0
7	**IF** $d^* \mid b$
8	**THEN** $x_0 \leftarrow x^* \cdot (b/d^*) \bmod n$
9	Hossz[X]← d^*
10	**FOR** $i \leftarrow 1$ **TO** $d^* - 1$ **DO**
11	$x_i \leftarrow x_0 + i \cdot (n/d^*) \bmod n$
12	**RETURN** (X)
13	// Hossz[X]=0 jelenti, hogy nincs megoldás

3.23. példa. Oldjuk meg a $3604 \cdot x \equiv 136 \bmod 3332$ egyenletet!

Láttuk, hogy $\text{lnko}\,(3604, 3332) = 68 = -12 \cdot 3604 + 13 \cdot 3332$. A 136 osztható 68-cal, így az egyenletnek van megoldása. Az alaprendszer 68 különböző elemet tartalmaz. Most

$$b/d^* = 136/68 = 2, \qquad n/d^* = 3332/68 = 49, \qquad x^* = -12 + 68 = 56.$$

A megoldások:

$$\begin{aligned} x_0 &= 56 \cdot 2 = 112, \\ x_1 &= 112 + 49 = 161, \\ x_2 &= 112 + 2 \cdot 49 = 210, \\ &\dots \\ x_{67} &= 112 + 67 \cdot 49 = 3395 \equiv 63 \bmod 3332. \end{aligned}$$

3.24. definíció. *A multiplikatív inverz*
Legyen a lineáris kongruencia egyenlet

$$ax \equiv 1 \bmod n, \qquad a \in \mathbb{Z}, n \in \mathbb{Z}^+, \text{lnko}\,(a,n) = 1$$

alakú (azaz a és n legyenek relatív prímek). Az egyenlet egyetlen alapmegoldását az a szám n szerinti *multiplikatív inverzének* nevezzük. Jelölése:

$$x = a^{-1} \bmod n.$$

A multiplikatív inverz meghatározása történhet a lineáris kongruencia megoldó 2.5.1. algoritmus segítségével. Természetesen a FOR ciklus alkalmazására az eljárásban nem lesz szükség.

3.25. példa. Oldjuk meg: $5^{-1} =? \bmod 8$.

Az $5x \equiv 1 \bmod 8$ megoldását keressük.

Lépésszám	n	a	q	r	d^*	x^*	y*		
0	8	5	1	3	1	2	$-1-1\cdot 2$	=	-3
1	5	3	1	2	1	-1	$1-1\cdot(-1)$	=	2
2	3	2	1	1	1	1	$0-1\cdot 1$	=	-1
3	2	1	2	0	1	0	$1-2\cdot 0$	=	1
4	1	0	-	1	1	1			0

Láthatóan $\text{lnko}(5,8) = 1$, tehát van multiplikatív inverz. $1 = 2\cdot 8 + (-3)\cdot 5 = 16 - 15$.

Az a együtthatója $--3$, aminek a 8 szerinti maradéka $--3+8 = 5$. Tehát az 5 multiplikatív inverze 8-ra nézve éppen saját maga.

Ellenőrzés: $5 \cdot 5 = 25 = 3 \cdot 8 + 1$.

3.6. Az RSA-algoritmus

Sok esetben – többek között a majd ismertetésre kerülő RSA algoritmusban – szükség van egészek hatványa valamely modulus szerinti maradékának meghatározására. Legyen $a, b, n \in \mathbb{Z}^+$. A feladat $c = a^b \bmod n$ meghatározása lehetőleg elfogadható idő alatt. Ilyennek bizonyul a *moduláris hatványozás* algoritmusa. Ötlete a b szám bináris felírásából jön. Legyenek a b bitjei: $b_k, b_{k-1}, \ldots, b_1, b_0$. A legmagasabb helyiértékű bit 1-es. Ha b-nek ki akarjuk számítani az értékét, akkor ezt megtehetjük a 2 hatványaival történő számítással, $b = b_k \cdot 2^k + b_{k-1} \cdot 2^{k-1} + \ldots + b_1 \cdot 2^1 + b_0 \cdot 2^0$. Ugyanezt az eredményt megkaphatjuk a gazdaságosabb Horner séma szerint:

$$b = (\ldots((b_k) \cdot 2 + b_{k-1}) \cdot 2 + \ldots + b_1) \cdot 2 + b_0$$

Itt láthatóan csak kettővel való szorzást és egy nulla vagy egy hozzáadását kell végezni, melyek számítástechnikailag hatékony műveletek. Ez annál inkább hasznos, mivel még a b értékét sem kell kiszámítani az algoritmusban, hiszen az adott, hanem csak az egyes bitjeit kell elérni, ami eltolásokkal hatékonyan megvalósítható. A b szám a kitevőben van, ezért a hatványozás során a kettővel való szorzásnak a négyzetreemelés az egy hozzáadásának pedig az alappal történő szorzás felel meg. Minden lépés után vehető a modulo n szerinti maradék, így a használt számtartomány mérete mérsékelt marad.

A megfelelő algoritmus pszeudokódja:

	2.6.1. algoritmus
	Moduláris hatványozó
1	Moduláris_hatványozó (a, b, n, c)
2	// Input paraméterek: $a, b, n \in \mathbb{Z}, a, b, n > 0$
3	// Output paraméter: $c \in \mathbb{Z}, c \geq 0$
4	$p \leftarrow 0$
5	$c \leftarrow 1$
6	**FOR** $i \leftarrow k$ **DOWNTO** 0 **DO**
7	$p \leftarrow 2p$
8	$c \leftarrow c^2 \bmod n$
9	**IF** $b_i = 1$
10	**THEN** $p \leftarrow p + 1$
11	$c \leftarrow (c \cdot a) \bmod n$
12	**RETURN** (c)

Az algoritmusban ténylegesen a p értékét nem kell kiszámítani, mert az végül a b értékét adja majd.

3.26. példa. $118^{2005} \bmod 137, b = 2005_{10} = (11111010101)_2, a = 118, n = 137.$

k	b_k	$c^2 \bmod n$					$(c \cdot a) \bmod n$				
10	1	12	=	1	$\equiv$	1	$1 \cdot 118$	=	118	$\equiv$	118
9	1	118^2	=	13924	$\equiv$	87	$87 \cdot 118$	=	10266	$\equiv$	128
8	1	128^2	=	16384	$\equiv$	81	$81 \cdot 118$	=	9558	$\equiv$	105
7	1	105^2	=	11025	$\equiv$	65	$65 \cdot 118$	=	7670	$\equiv$	135
6	1	135^2	=	18225	$\equiv$	4	$4 \cdot 118$	=	472	$\equiv$	61
5	0	61^2	=	3721	$\equiv$	22					
4	1	22^2	=	484	$\equiv$	73	$73 \cdot 118$	=	8614	$\equiv$	120
3	0	120^2	=	14400	$\equiv$	15					
2	1	15^2	=	225	$\equiv$	88	$88 \cdot 118$	=	10384	$\equiv$	109
1	0	109^2	=	11881	$\equiv$	99					
0	1	99^2	=	9801	$\equiv$	74	$74 \cdot 118$	=	8732	$\equiv$	101

Az RSA algoritmus fel fogja tételezni, hogy nagy prímszámaink vannak. Ilyenek keresésére egy eszköz lehet (nem a leghatékonyabb és nem abszolút biztos) az alábbi tételen alapuló algoritmus.

3.27. tétel. A Fermat tétel
Ha p prím, akkor

$$a^{p-1} \equiv 1 \bmod p, \qquad a = 1, 2, \ldots, p-1.$$

A tételt nem bizonyítjuk.
A tételre épülő prímszám ellenőrzési algoritmus egy egyszerű, de nem teljesen megbízható változatának a pszeudokódja:

	2.6.2. algoritmus
	Fermat féle álprímteszt
1	Fermat_teszt (n, p)
2	// Input paraméter: $n \in \mathbb{Z}, n > 1$
3	// Output paraméter: p logikai érték
4	// igaz – lehet prím
5	// hamis – nem prím
6	Moduláris_hatványozó $(2, n-1, n, c)$
7	$p \leftarrow (c = 1)$
8	**RETURN** (p)

Ha ez az algoritmus azt mondja, hogy a szám összetett, akkor az biztosan nem lesz prím. Ha azt mondja, hogy lehet, hogy prím, akkor nagy eséllyel valóban prímet vizsgált, ugyanis 10000-ig terjedően a számok között csak 22 olyan van, amely nem prím és a teszt esetlegesen prímnek minősíti. Ilyenek a $341, 561, 645, 1105, \ldots$. Ötven bites számok esetén már csak a számok egy milliomod része lehet ilyen, 100 biteseknél pedig ez az arány 1:1013. Ezen hibák egy része kiszűrhető azzal, hogy a 2 helyett más alapot is beveszünk a moduláris hatványozásba, például a 3-at, stb. Sajnos azonban vannak olyan számok, amelyek mindegyik alap esetén prímnek maszkírozzák magukat ennél az algoritmusnál. Ezek az úgynevezett *Carmichael számok*. A Carmichael számok relatíve nagyon kevesen vannak. (Valójában végtelen sok ilyen szám van. Ilyenek: $561, 1105, 1729, \ldots$. Az első egy milliárd szám között csak 255 ilyen van.)

3.28. példa. Döntsük el, hogy a 11 és a 12 prímek-e?

$2^{10} =? \bmod 11, 10 = (1010)_2$

3	1	$1^2 = 1$	$1 \cdot 2 = 2$
2	0	$2^2 = 4$	
1	1	$4^2 = 16 \equiv 5$	$5 \cdot 2 = 10$
0	0	$10^2 = 100 \equiv 1$	

$2^{10} \equiv 1 \bmod 11$
Tehát a 11 nagy eséllyel prím.

$2^{11} =? \bmod 12, 11 = (1011)_2$

3	1	$1^2 = 1$	$1 \cdot 2 = 2$
2	0	$2^2 = 4$	
1	1	$4^2 = 16 \equiv 4$	$4 \cdot 2 = 8$
0	1	$8^2 = 64 \equiv 4$	$4 \cdot 2 = 8$

$2^{11} \equiv 8 \bmod 12.$
Tehát a 12 nem prím.

Ezen előkészületek után térjünk rá a fejezet céljára a nyilvános kulcsú titkosításra A titkosítás alapja az eredeti szöveg átalakítása, kódolása. A nyílvános kulcsok használata azt jelenti, hogy minden résztvevőnek van egy nyílvános, mindenki számára hozzáférhető kulcsa (P, személyes, Private) és egy titkos, más által nem ismert kulcsa (S, titkos, Secret). Legyen M az üzenet. Legyen a két résztvevő A és B. A küldi B-nek az M üzenetet titkosítva. Az elküldött titkosított szöveg $C = P_B(M)$, B megkapja a C üzenetet és a titkos kulcsával dekódolja $M = S_B(C)$. A kulcsok egymás inverzei, és úgy vannak kialakítva, hogy a P kulcs révén könnyű legyen titkosítani, de a kulcs ismeretében nagyon nehezen lehessen - praktikusan lehetetlen legyen - az S kulcsot meghatározni. A digitális aláírás ilyenkor történhet úgy, hogy a küldő a titkosított C szöveg mellé akár nyíltan odaírja a saját Q azonosítóját (aláírását), majd annak az $R = S_A(Q)$ titkosítottját. Ezután B a Q alapján tudva, hogy kit nevez meg az aláírás, annak privát kulcsával dekódolja R-et. $Q* = P_A(R)$. Ha $Q* = Q$, akkor nem történt átviteli hiba, vagy hamisítás, egyébként igen. Persze Q az M-mel együtt is kódolható. Ez annak felel meg, mintha az első esetben nyílt levelezőlapon lenne az aláírásunk, a másodikban pedig mintha borítékba tettük volna.

Alább közöljük az RSA (Rivest – Shamir - Adleman) nyílvános kulcsú titkosítás algoritmusát. Az algoritmus feltételez két nagy prímszámot. (A gyakorlatban legalább 100-200 jegyűekre van szükség, hogy a titkosítás praktikusan feltörhetetlen legyen.) A P kulcs felépítése $P = (e, n)$, ahol n a két prím szorzata, e pedig egy kis páratlan szám. Az S kulcs $S = (d, n)$.

	2.6.3. algoritmus
	RSA kulcsok meghatározása
1	RSA_kulcsok_meghatározása (p, q, e, P, S)
2	// Input paraméterek: p, q, e
3	// Output paraméterek: P, S
4	**IF** p vagy q nem prím vagy $e<3$ vagy e páros
5	**THEN RETURN** („Nincs kulcs”)
6	$n \leftarrow p \cdot q$
7	$f \leftarrow (p-1) \cdot (q-1)$
8	**IF** $\text{lnko}(e, f) \neq 1$
9	**THEN RETURN** („Nincs kulcs”)
10	$d \leftarrow e^{-1} \bmod f$
11	**RETURN** $(P = (e, n), S = (d, n))$

A szöveg titkosítása a $C = P(M) = M^e \bmod n$ alapján történik. Dekódolása pedig az $M = S(C) = C^d \bmod n$ alapján. A szöveg darabolásának bitméretét az n szabja meg.

Az eljárás helyességét nem bizonyítjuk.

3.29. példa. Számpélda RSA algoritmusra (nem életszerű, mivel a prímek kicsik)

Legyen a titkos választás:

$p = 11, q = 29, n = p{\cdot}q = 11{\cdot}29 = 319, p = 11, f = (p-1){\cdot}(q-1) = 10{\cdot}28 = 280$

A kibővített euklideszi algoritmust alkalmazzuk.

f	e	$\lfloor f/e \rfloor$	$f \bmod e$	d^*	x	y
280	3	93	1	1	1	- 93
3	1	3	1	1	0	1
1	0	-	1	1	1	0

Láthatóan $\text{lnko}(f, e) = 1$ és e multiplikatív inverze $d = e^{-1} = -93$. Ez utóbbi helyett 280-at hozzáadva vesszük a 187-et. Ezek után akkor

$$P = (3; 319) \text{ közölhető kulcs } P(M) = M^3 \bmod 319$$
$$S = (187; 319) \text{ titkos kulcs } S(C) = C^{187} \bmod 319$$

Legyen az üzenetünk 100. Egy darabban titkosítható, mivel ez kisebb, mint 319. Titkosítsuk, majd fejtsük meg az üzenetet.

Titkosítás: $C = 100^3 \bmod 319 \qquad 3_{10} = 11_2$

1	1	$12 = 1 \equiv 1$	$1 \cdot 100 = 100 \equiv 100$
0	1	$1002 = 10000 \equiv 111$	$111 \cdot 100 = 11100 \equiv 254$

Tehát a titkosított érték: $C = P(M) = 254$

Megfejtés: $M = 254^{187} \bmod 319 \qquad 187_{10} = 10111011_2$.

7	1	$12 = 1 \equiv 1$	$1 \cdot 254 = 254 \equiv 254$
6	0	$2542 = 64516 \equiv 78$	
5	1	$782 = 6084 \equiv 23$	$23 \cdot 254 = 5842 \equiv 100$
4	1	$1002 = 10000 \equiv 111$	$111 \cdot 254 = 28194 \equiv 122$
3	1	$1222 = 14884 \equiv 210$	$210 \cdot 254 = 53340 \equiv 67$
2	0	$672 = 4489 \equiv 23$	
1	1	$232 = 529 \equiv 210$	$210 \cdot 254 = 53340 \equiv 67$
0	1	$672 = 4489 \equiv 23$	$23 \cdot 254 = 5842 \equiv 100$

Tehát a megfejtés: $M = S(C) = 100$

4. fejezet

Elemi dinamikus halmazok

4.1. A tömb adatstruktúra

Egy adastruktúra számtalan adatot tartalmazhat. Mondhatjuk, hogy egy adathalmazt tárolunk egy struktúrában. Számunkra a dinamikus halmazok lesznek fontosak.

4.1. definíció. *Dinamikus halmaz* Az olyan halmazt, amely az őt felhasználó algoritmus során változik (bővül, szűkül, módosul) dinamikus halmaznak nevezzük.

A dinamikus halmazok elemei tartalmazhatnak az információs adatmezőiken felül kulcsmezőt, és mutatókat (pointereket), amelyek a dinamikus halmaz más elemeire mutatnak. (pl: a következő elemre). Felsorolunk a dinamikus halmazokon néhány általánosságban értelmezett műveletet. Konkrét esetekben ezek közül egyesek el is maradhatnak, vagy továbbiak is megjelenhetnek. Az S jelöli a szóban forgó halmazt, k kulcsot ad meg és x mutató a halmaz valamely elemére. Feltételezzük, hogy a kulcsok között értelmezett a kisebb, nagyobb, egyenlő reláció.

Lekérdező műveletek	
KERES (*S, k, x*)	adott k kulcsú elem x mutatóját adja vissza, vagy NIL, ha nincs.
MINIMUM (*S, x*)	A legkisebb kulcsú elem mutatóját adja vissza
MAXIMUM (*S, x*)	A legnagyobb kulcsú elem mutatóját adja vissza
KÖVETKEZŐ (*S, x, y*)	az x elem kulcsa utáni kulcsú elem mutatóját adja vissza,NIL, ha x után nincs elem
ELŐZŐ (*S, x, y*)	az x elem kulcsa előtti kulcsú elem mutatóját adja vissza,NIL, ha x előtt nincs elem

Módosító műveletek	
BESZÚR (*S, x*)	az S bővítése az x mutatójú elemmel
TÖRÖL (*S, x*)	az x mutatójú elemet eltávolítja S-ből

Az egyes műveletek végrehajtásukat tekintve lehetnek *statikusak* (passzívak), vagy dinamikusak (aktívak) aszerint, hogy a struktúrát változatlannak hagyják-e vagy sem. A módosító műveletek alapvetően dinamikusak, a lekérdezők általában statikusak, de nem ritkán lehetnek szintén dinamikusak. (A dinamikus lekérdezés olyan szempontból érdekes és fontos, hogy ha egy elemet a többitől gyakrabban keresnek, akkor azt a struktúrában a keresés folyamán a megtalálási útvonalon közelebbi helyre helyezi át a művelet, ezzel megrövidíti a későbbi keresési időt erre az elemre, vagyis a művelet változást eredményez a struktúrában.)

4.2. definíció. *A sorozat adatstruktúra*
Sorozatnak nevezzük az objektumok (elemek) olyan tárolási módját (adatstruktúráját), amikor az elemek a műveletek által kijelölt lineáris sorrendben követik egymást. Tipikus műveletek: keresés, beszúrás, törlés.

A sorozat egyik lehetséges *implementációja* – gyakorlati megvalósítása, megvalósítási eszköze – a tömb. A *tömb* azonos felépítésű (típusú) egymást fizikailag követő memóriarekeszeket jelent. Egy rekeszben egy elemet, *adatrekord*ot helyezünk el. Az egyes tömbelemek helyét az indexük határozza

meg. Az elemek fontos része a kulcsmező, melyet **kulcs**$[A_x]$ révén kérdezhetünk le az A tömb x indexű eleme esetén. Számunkra lényegtelen lesz, de a gyakorlat szempontjából alapvetően fontos része az adatrekordnak az információs (adat) mezőkből álló rész. A tömböt szokás *vektor*nak is nevezni. Ha a lineáris elhelyezésen kívül egyéb szempontokat is figyelembe veszünk, akkor ezt az egyszerű szerkezetet el lehet bonyolítani. Ha például az elemek azonosítására indexpárt használunk, akkor *mátrix*ról vagy *táblázat*ról beszélünk. Ilyen esetben az első index a sort, a második az oszlopot adja meg. (Itt tulajdonképpen olyan vektorról van szó, amelynek elemei maguk is vektorok.)

A struktúrának és így az implementációnak is lehetnek *attribútuma*i – jellemzői, hozzákapcsolt tulajdonságai. A tömb esetében ezeket az alábbi táblázatban adjuk meg.

Attributum	Leírás
fej[A]	A tömb első elemének indexe. NIL, ha a tömbnek nincs eleme.
vége[A]	A tömb utolsó elemének indexe. NIL, ha a tömbnek nincs eleme.
hossz[A]	A tömbelemek száma. Zérus, ha a tömbnek nincs eleme.
tömbméret[A]	annak a memóriaterületnek a nagysága tömbelem egységben mérve, ahová a tömböt elhelyezhetjük. A tömb ezen terület elején kezdődik.

Vizsgáljuk meg most a műveletek algoritmusait!

A keresési algoritmus. Az A tömbben egy k kulcsú elem keresési algoritmusa pszeudokóddal lejegyezve következik alább. Az algoritmus NIL-t ad vissza, ha a tömb üres, vagy a tömbben nincs benne a keresett kulcsú elem. A tömb elejétől indul a keresés. Ha a vizsgált elem egyezik a keresett elemmel, akkor azonnal viszatérünk az indexével. (Realizáció szempontjából úgy is elképzelhetjük a dolgot, hogy a tömb elemeinek indexelése 1-gyel kezdődik és a NIL eredményt a 0 indexszel jelezzük.) Ha nem egyezik, akkor az INC függvénnyel növeljük eggyel az index értékét (rátérünk a következő elemre)

és újra vizsgálunk. Addig növeljük az indexet, míg az érvényes indextartományból ki nem lépünk vagy meg nem találjuk a keresett kulcsú elemet. A legrosszabb eset az, ha az elem nincs benne a tömbben, – ekkor ugyanis az összes elemet meg kell vizsgálni – így az algoritmus időigénye: $T(n) = \Theta(n)$, ahol $n = \textbf{hossz}[A]$, a tömbelemek száma.

	3.1.1. algoritmus
	Keresés tömbben
	// $T(n) = \Theta(n)$
1	KERESÉS_TÖMBBEN (A, k, x)
2	// Input paraméter: A - a tömb
3	// k – a keresett kulcs
4	// Output paraméter: x - a k kulcsú elem pointere (indexe), ha van ilyen elem, vagy NIL, ha nincs
5	// Lineárisan keresi a k kulcsot.
6	//
7	$x \leftarrow$ **fej**[A]
8	**IF** **hossz**[A] $\neq 0$
9	**THEN** **WHILE** $x \leq$ **vége**[A] és **kulcs**$[A_x] \neq k$ **DO**
10	INC(x)
11	**IF** $x >$ **vége**[A]
12	**THEN** $x \leftarrow$ **NIL**
13	**RETURN** (x)

Az új elem beszúrásának algoritmusa az A tömb adott x indexű helyére szúrja be az új elemet. Az ott lévőt és a mögötte állókat egy hellyel hátrább kell tolni. Emiatt az időigény $T(n) = \Theta(n)$.

	3.1.2. algoritmus
	Beszúrás tömbbe
	// $T(n) = \Theta(n)$
1	BESZÚRÁS_TÖMBBE (*A, x, r, hibajelzés*)
2	// Input paraméter: A – a tömb
3	// x – a tömbelem indexe, amely elé történik a beszúrás, ha a tömb nem üres és az x index létező elemre mutat. Üres tömb esetén az x indexnek nincs szerepe, a beszúrandó elem az első helyre kerül.
4	// r – a beszúrandó elem (rekord)
5	// Output paraméter: *hibajelzés* - a beszúrás eredményességét jelzi
6	//
7	**IF** **hossz**[A] $\neq 0$
8	**THEN IF** **fej**[A] $\leq x \leq$ **vége**[A]) és (**tömbméret**[A] $>$ **hossz**[A])
9	**THEN FOR** $i \leftarrow$ **vége**[A] **DOWNTO** x **DO**
10	$A_{i+1} \leftarrow A_i$
11	$A_x \leftarrow r$
12	INC(**hossz**[A])
13	INC(**vége**[A])
14	*hibajelzés*: $\leftarrow$ „sikeres beszúrás”
15	**ELSE** *hibajelzés*: $\leftarrow$ „nem létező elem,vagy nincs az új elemnek hely”
16	**ELSE** **fej**[A] $\leftarrow$ **vége**[A] $\leftarrow$ **hossz**[A] $\leftarrow 1$
17	$A_1 \leftarrow r$
18	**RETURN** (*hibajelzés*)

Ezzel az algoritmussal nem tudunk az utolsó elem után beszúrni. A problémát egy erre a célra megírt külön algoritmussal is megoldhatjuk. Legyen ennek CSATOL_TÖMBHÖZ a neve. Az olvasóra bízzuk pszeudokódjának megírását. Írjunk pszeudokódot arra az esetre, amikor a beszúrás az adott indexű elem mögé történik! Ennek is van egy szépséghibája!

Egy adott indexű elem törlésének algoritmusa az elem felszabaduló helyének megfelelően az elem mögött állókat feltömöríti, egy hellyel előre lépteti. Az algoritmusban használni fogjuk a DEC függvényt, melynek hatása az, hogy csökkenti eggyel az argumentuma értékét. Ennek révén a hossz és a vége attributumokat korrigáljuk. Legrosszabb esetben az összes megmaradt elemet mozgatni kell, ezért az időigény $T(n) = \Theta(n)$.

	3.1.3. algoritmus
	Törlés tömbből
	// $T(n) = \Theta(n)$
1	TÖRLÉS_TÖMBBŐL (*A, x, hibajelzés*)
2	// Input paraméter: A - a tömb
3	// x – a törlendő tömbelem indexe, ha a tömb nem üres és az x index létező elemre mutat. A hátrébb álló elemeket egy hellyel előre léptetjük. A tömb megrövidül.
4	// Output paraméter: *hibajelzés* - a beszúrás eredményességét jelzi
5	//
6	**IF** **hossz**[A] $\neq 0$
7	**THEN** **IF** **fej**[A] $\leq x \leq$ **vége**[A]
8	**THEN** **FOR** $i \leftarrow x$ **TO** **vége**[A]-1 **DO**
9	$A_i \leftarrow A_{i+1}$
10	**DEC**(**hossz**[A])
11	**DEC**(**vége**[A])
12	*hibajelzés*: ← „sikeres törlés”
13	**ELSE** *hibajelzés*: ← „nem létező elem”
14	**ELSE** *hibajelzés*: ← „üres tömb”
15	**RETURN** (*hibajelzés*)

A lineáris törlési időt konstansra csökkenthetjük, ha a tömbelemek eredeti sorrendjének megörzése nem fontos azáltal, hogy a törlésre ítélt elem helyére a tömb utolsó elemét helyezzük és a tömböt lerövidítjük.

Az eddigiek során nem használtuk ki, hogy a tömbben a kulcsok rendezetten (növekvő sorrend, vagy csökkenő sorrend) követik egymást. Nem is használhattuk ki, hiszen ezt nem tételeztük fel. Ez ismeretlen volt a számunkra. Most azonban élünk azzal a feltételezéssel, hogy a tömbelemek kulcsai növekvő sorrendben vannak.

A keresés időigénye, amely rendezettlen tömbben lineáris volt, feljavítható logaritmikusra az úgynevezett bináris keresés révén. A bináris keresés alapötlete az, hogy a k kulcs keresésekor a kulcsösszehasonlítást a tömb középső elemének kulcsával kezdjük. Ha egyezést tapasztalunk, akkor az eljárás véget ér. Ha a k kulcs értéke kisebb, mint a megvizsgált elem kulcsa, akkor a tömb középső elemének indexétől kisebb indexű elemek között folytatjuk a keresést. Ha a k kulcs értéke nagyobb, mint a megvizsgált elem kulcsa, akkor a tömb középső elemének indexétől nagyobb indexű elemek között folytatjuk a keresést. A méret ezáltal feleződött. A további keresés ugyanilyen elv alapján megy tovább. Minden lépésben vagy megtaláljuk a keresett kulcsú elemet, vagy fele méretű résztömbben folytatjuk a keresést. Ha a résztömb mérete (hossza) zérusra zsugorodik, akkor a keresett kulcs nincs a tömbben.

Megadjuk a rekurzív algoritmust is és az iteratívat is. Mindkettőben a felezést *nyílt index-intervallumra* végezzük, ami azt jelenti, hogy az index-intervallum végek nem tartoznak a keresési index-intervallumhoz. Ez az ötlet jó hatással van az algoritmus szerkezetére. Az iteratív esetben az algoritmus bemenő paraméterei természetes módon a tömb és a keresett kulcs, az algoritmus az egész tömbben keres. A rekurzív algoritmus bemenő paraméterei a rekurzivitás sajátosságai révén szintén természetes módon a tömb, a keresési nyílt index-intervallum két vége valamint a keresett kulcs. Tehát alaphelyzetben ez az algoritmus csak a tömb egy összefüggő részében keres, nem a teljes tömbben. Ha a teljes tömbben akarunk keresni, akkor az algoritmust a

BINÁRIS_KERESÉS_TÖMBBEN (A, **fej**[A] - 1, **vége**[A] + 1, k)

sorral kell aktivizálni. Itt feltételeztük, hogy a tömb nem üres.

	3.1.4. algoritmus
	Bináris keresés tömbben (rekurzív változat)
	// $T(n) = \Theta(\log n)$
1	BIN_KER_TÖMB (A, i, j, k, x)
2	// Input paraméter: A - a tömb
3	// i a keresési nyílt intervallum kezdőindexe. A keresésben ez az index még nem vesz részt.
4	// j a keresési nyílt intervallum végindexe. A keresésben ez az index már nem vesz részt.
5	// k – a keresett kulcs
6	// Output paraméter: x - a k kulcsú elem indexe. **NIL**, ha a kulcs nincs a tömbben.
7	//
8	$x \leftarrow \lfloor \frac{i+j}{2} \rfloor$
9	**IF** $i = x$
10	**THEN** $x \leftarrow$ **NIL**
11	**ELSE IF** $k =$ ***kulcs***$[A_x]$
12	**THEN RETURN** (x)
13	**ELSE IF** $k >$ ***kulcs***$[A_x]$
14	**THEN** BIN_KER_TÖMB (A, x, j, k, z)
15	**ELSE** BIN_KER_TÖMB (A, i, x, k, z)
16	$x \leftarrow z$
17	**RETURN** (x)

	3.1.5. algoritmus
	Bináris keresés tömbben (iteratív változat)
	// $T(n) = \Theta(\log n)$
1	BINÁRIS_KERESÉS_TÖMBBEN (A, k, x)
2	// Input paraméter: A – a tömb
3	// k – a keresett kulcs
4	// Output paraméter: x - a k kulcsú elem indexe. **NIL**, ha a kulcs nincs a tömbben.
5	//
6	**IF** **hossz**[A]=0
7	**THEN** $x \leftarrow$ **NIL**
8	**ELSE** $i \leftarrow$ **fej**[A] - 1,
9	$j \leftarrow$ **vége**[A] + 1,
10	$x \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
11	**WHILE** $i < x$ **DO**
12	**IF** **kulcs**$[A_x] = k$
13	**THEN RETURN** (x)
14	**IF** $k >$ **kulcs**$[A_x]$
15	**THEN** $i \leftarrow x$
16	**ELSE** $j \leftarrow x$
17	$x \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
18	**RETURN** (x)

A másik két művelet hatékonyságára a rendezettség nincs jótékony hatással. A beszúrásnál a helycsinálás továbbra is lineáris ideig fog tartani, és ez lesz a helyzet a törlésnél is a tömörítés idejével.

4.2. A láncolt lista (mutatós és tömbös implementáció)

Egy másik lehetőség a sorozat implementálására a *láncolt lista*.

4.3. definíció. *A láncolt lista adatstruktúra*
A *láncolt lista* (linked list) olyan dinamikus halmaz, melyben az objektumok, elemek lineáris sorrendben követik egymást. A lista minden eleme mutatót tartalmaz a következő elemre. Műveletei: keresés, beszúrás, törlés.

A láncolt listáról nem feltételezzük, hogy az egymást követő elemei a memóriában valamilyen szabályszerűségnek megfelelően követik egymást. Az elemek lehetnek bárhol, az egyes elemeket mutatón keresztül érhetjük el, nem az index révén.. Ha valamely elemet – például az elsőt - megtaláltuk, akkor a rákövetkezőt is megtaláljuk a benne lévő mutató alapján. Az elem számára helyet foglalni elegendő csak a keletkezése pillanatában. Megszünésekor helyét felszabadíthatjuk. Ennek feltétele, hogy elérhető legyen a dinamikus memória gazdálkodás. A láncolt listák a mutatók és a kulcsok alapján osztályozhatók az alábbi egymást ki nem záró szempontok szerint.

Láncoltság:	**Egyszeresen láncolt**, ha csak egy mutató van minden elemben (előre láncolt). A listán az elejétől végig lehet menni a végéig. **Kétszeresen láncolt**, ha van visszafelé mutató mutató is. Ekkor a lista végétől is végig lehet menni a listán az elejéig.
Rendezettség:	**Rendezett** a lista, ha a kulcsok valamilyen sorrend szerint (növekvő, csökkenő) követik egymást. **Nem rendezett** a lista, ha a kulcsok sorrendjében nincs rendezettség.
Ciklikusság:	**Ciklikus** a lista, ha listavégek elemeinek mutatói nem jelzik a véget, hanem a lista másik végelemére mutatnak. **Nem ciklikus**, ha a listavégi elemek jelzik a lista véget.

Az L listának, mint struktúrának az attributuma a **fej**[L], ami egy a lista első elemére mutató mutató. Ha **fej**[L] = **NIL**, akkor a lista üres. A listaelemek attributumai az alábbi táblázatban következnek. A tárgyalásban kétszeresen láncolt, nem rendezett, nem ciklikus listáról van szó. A listaelemet az x mutató révén érhetjük el.

Attributum	Leírás
kulcs[x]	az elem kulcsa
elő[x]	Az x mutató által mutatott elemet megelőző elemre mutató mutató. Ha **elő**[x] = **NIL**, akkor az x mutató által mutatott elem a lista eleje.
köv[x]	Az x mutató által mutatott elemet követő elemre mutató mutató. Ha **köv**[x] = **NIL**, akkor az x mutató által mutatott elem a lista vége.

Tekintsük át most a műveletek pszeudokódjait. Elsőként a keresés. A keresésnél a listafej információ alapján a lista kezdetét megtaláljuk és a ***köv*** mutatók révén végig tudunk lépkedni a listaelemeken, miközben vizsgáljuk a kulcsok egyezőségét. A legrosszabb eset az, amikor az elem nincs a listában. Ilyenkor minden elem vizsgálata sorrakerül és ez adja a lineáris időt.

	3.2.1. algoritmus
	Láncolt listában keresés
	// $T(n) = \Theta(n)$
1	LISTÁBAN_KERES (L, k, x)
2	// Input paraméter: L – a lista
3	// k – a keresett kulcs
4	// Output paraméter: x - a kulcselem pointere. **NIL**, ha a kulcs nincs a listában.
5	//
6	$x \leftarrow$ **fej**[L]
7	**WHILE** $x \neq$ **NIL** és **kulcs**[x] $\neq k$ **DO**
8	$x \leftarrow$ **köv**[x]
9	**RETURN** (x)

A beszúrást konstans idő alatt azáltal tudjuk elvégezni, hogy az új elemet mindig a lista legelső eleme elé szúrjuk be. Ekkor mindegy, hogy hány elem van a listában, a beszúrási idő ugyanannyi.

	3.2.2. algoritmus
	Láncolt listába beszúrás (kezdőelemként)
	// $T(n) = \Theta(1)$
1	LISTÁBA_BESZÚR (L, x)
2	// Input paraméter: L – a lista
	// x a beszúrandó elemre mutató mutató
3	// Az új elemet a lista elejére teszi
4	//
5	**köv**[x] ← **fej**[L]
6	**elő**[x] ← **NIL**
7	**IF** **fej**[L] ≠ **NIL**
8	**THEN elő**[**fej**[L]] ← x
9	**fej**[L] ← x
10	**RETURN**

Szintén megoldható konstans idő alatt a beszúrás a lista tetszőleges eleme elé, vagy mögé, amennyiben az elemre mutató mutató meg van adva. Ha a mutató helyett az elem kulcsa van megadva, akkor lineáris idejű a beszúrás, mivel a beszúrás tényleges elvégzése előtt az elemet meg kell keresni a listában.

Törlésnél az elem nem szűnik meg létezni, csak a listából kiláncolódik, a listán lépkedve többé nem érhető el. Elérhető viszont más úton akkor, ha valahol maradt valamilyen mutató, amely rá mutat. A konstans idő abból adódik, hogy az algoritmus input adataként a törlendő elem mutatóját adjuk meg, így az elemet nem kell keresni. Ha az inputban a törlendő elem kulcsa szerepelne, akkor a kiláncolás előtt előbb a kulcs alapján az elemet meg kellene keresni, ami miatt az idő lineárissá nőne.

	3.2.3. algoritmus
	Láncolt listából törlés
	// $T(n) = \Theta(1)$
1	LISTÁBÓL_TÖRÖL(L,x)
2	// Input paraméter: L – a lista
3	// x a törlendő elemre mutató mutató
4	//
5	**IF** **elő**[x] $\neq$ **NIL**
6	**THEN köv**[**elő**[x]] $\leftarrow$ **köv**[x]
7	**ELSE fej**[L] $\leftarrow$ **köv**[x]
8	**IF** **köv**[x] $\neq$ **NIL**
9	**THEN elő**[**köv**[x]] $\leftarrow$ **elő**[x]
10	**ELSE köv**[**elő**[x]] $\leftarrow$ **NIL**
11	**RETURN**

Némi kényelmetlenséget jelent a lista kezelésénél a listavégek állandó vizsgálata, valamint hogy a végeken az algoritmus lépései eltérnek a középen alkalmazott lépésektől. Ennek a feloldása úgynevezett *szentinel* (őrszem, strázsa) alkalmazásával megoldható.

Legyen **nil**[L] egy mutató, amely az alábbi szerkezetű elemre mutat:

elő	***kulcs***	***köv***
A lista végére mutat	Speciális, „érvénytelen szerkezetű”	a lista elejére mutat

Ez az elem testesíti meg a nem létező **NIL** elemet. Ezzel az elemmel tulajdonképpen egy ciklikus listát valósítunk meg, melyben egy olyan kulcsú elem van, amelyről azonnal eldönthető, hogy a valódi listához nem tartozhat hozzá. Bármilyen irányban haladunk a listán, mindig jelezni tudjuk a kulcs vizsgálata révén, hogy a lista elejére, vagy a végére értünk. Ennek a listának a sajátossága, hogy **köv**[**nil**[L]] a lista első elemére mutat, **elő**[**nil**[L]] pedig az utolsó elemére. A lista utolsó eleme esetében **köv**[x] = **nil**[L], az első eleme esetében pedig **elő**[x] = **nil**[L]. A **fej**[L] attributumra nincs szükség!

Ennek megfelelően az algoritmusaink az alábbi módon változnak, egyszerűsödnek.

	3.2.4. algoritmus
	Szentineles listában keresés
	// $T(n) = \Theta(n)$
1	SZENTINELES_LISTÁBAN_KERES (L, k, x)
2	// Input paraméter: L – a lista
3	// k – a keresett kulcs
4	// Output paraméter: x - a kulcselem pointere. **nil**[L], ha a kulcs nincs a listában.
5	//
6	$x \leftarrow$**köv**[**nil**[L]]
7	**WHILE** $x \neq$ **nil**[L] és **kulcs**[x] $\neq k$ **DO**
8	$x \leftarrow$ **köv**[x]
9	**RETURN** (x)

	3.2.5. algoritmus
	Szentineles listába beszúrás
	// $T(n) = \Theta(1)$
1	SZENTINELES_ LISTÁBA_BESZÚR (L, x)
2	// Input paraméter: L – a lista
3	// x a beszúrandó elemre mutató mutató
4	// Az új elemet a lista elejére teszi
5	//
6	**köv**[x] $\leftarrow$ **köv**[**nil**][L]
7	**elő**[**köv**[**nil**[L]]] $\leftarrow x$
8	**köv**[**nil**[L]] $\leftarrow x$
9	**elő**[x] $\leftarrow$ **nil[L]**
10	**RETURN**

	3.2.6. algoritmus
	Szentineles listából törlés
	// $T(n) = \Theta(1)$
1	SZENTINELES_LISTÁBÓL_TÖRÖL (L, x)
2	/// Input paraméter: L – a lista
3	// x a törlendő elemre mutató mutató
4	//
5	**köv**[**elő**[x]] ← **köv**[x]
6	**elő**[**köv**[x]] ← **elő**[x]
7	**RETURN**

A rendezett lista rendezettségi tulajdonságait sajnos nem tudjuk kihasználni algoritmus gyorsításra. Ha a dinamikus memória gazdálkodás nem elérhető, vagy nem kívánunk vele élni, vagy egyszerűen nem vonzódunk a mutatók használatához (bár ez utóbbi nem úri passzió kérdése egy informatikai rendszer kifejlesztésekor), akkor a láncolt lista adatstruktúra realizálható, implementálható tömbbel is. Ekkor minden tömbelemet kiegészítünk „mutató" mezőkkel, amelyek valójában tömbelem indexeket fognak tárolni. A listafej is egy különálló, egész számot tároló rekesz lesz, amely megmutatja, hogy a tömb melyik eleme számít a lista kezdő elemének. A műveletek realizálásának pszeudokódjában a beszúrásnál most azt is figyelni kell, hogy van-e még fel nem használt rekesz a tömbben, vagyis, hogy a rendelkezésre álló memória korlátos. Természetesen a valódi mutatók használatakor is figyelni kell a memóriát, hiszen minden új elem megjelenése új memóriaigényt támaszt. A tömbös realizációhoz javasolható a következő séma. Legyen a *fej* annak a rekesznek a neve, melyben a lista első elemének a tömbelem indexe található. A **NIL** mutatót szimbolizálja a zérus. A listaelemek tárolására rendelkezésre bocsátott tömb neve legyen A, elemeinek indexelése induljon 1-től és tartson *maxn*-ig. Minden tömbelem, mint rekord álljon a kulcsmezőből, az információs mezőkből, valamint a „mutató" mezőkből (előző elemre és a következő elemre mutatók). A zérus realizálja itt is a **NIL** mutatót.

4.4. példa. Álljon a listánk a következő kulcsú elemekből: 423, 356, 764, 123, 987, 276, 839. Tömbös realizácóval a következőképpen nézhet ki egy ilyen láncolt lista egy tömbben, amelynek mondjuk 10 eleme van. A re-

kordokból (sorokból) az információs mezőket kihagyjuk, mert a tárgyalás szempontjából lényegtelenek.

fej	3

A			
index	**elő**	**kulcs**	**köv**
1	6	764	7
2			
3	0	423	6
4			
5	7	987	9
6	3	356	1
7	1	123	5
8			
9	5	276	10
10	9	839	0

Ha most a fenti listába be kellene szúrni a 247-es kulcsot, akkor ez többféle módon is megtehető. Be lehet szúrni - ha semmilyen megkötés sincs – az új elemet a lista elejére az első elem elé. Másik lehetőség, hogy megmondják, hogy például szúrjuk be a 356-os kulcsú elem mögé. Van ezeken kívül még sok lehetőség. Nézzük az említett két esetet. Az új elemet egyelőre helyezzük el mondjuk a tömb 2-es indexű helyén, ami jelenleg üres és a listához nem tartozik hozzá. A lista első eleme az új elem lesz, tehát az ő **elő** mutatója zérus lesz, és a régi első elem **elő** mutatója az új elemre fog mutatni, tehát 2-re változik. Meg kell még adni az új első elem **köv** mutatóját, amely a régi első elemre mutat, tehát értéke 3 lesz, ami korábban a **fej** mutató volt. A **fej** mutatónak pedig az új első elemre kell mutatni, tehát értéke 2 lesz. Látható, hogy a művelethez a mutatókat illetően négynek a megváltoztatására volt szükség. Ezek után az új lista tömbös megjelenése alább látható a baloldali táblázatban. A változásokat vastagítva és aláhúzva jelenítettük meg. A másik esetben az új elem **elő** mutatója a 356-osra fog mutatni, amely a 6-os helyen van, a **köv** mutatója pedig a 356-os **köv** mutatóját kapja, vagyis 1-et. A 356-os **köv** mutatója is és a 356 mögött korábban álló elem (a 764-es kulcsú) **elő** mutatója is az új elemre mutat, tehát mindkettő értéke 2. Vigyázni kell a mutatók megváltoztatásánál. A 764-es **elő** mutatóját

hamarabb kell megváltoztatni, mint a 356-os **köv** mutatóját, mert ellenkező esetben a 356-os **köv** mutatója már nem a 764-esre mutat. Itt is négy mutató változott. Az eredmény az alábbi jobboldali táblázatban látható.

fej	**2**

A			
index	**elő**	***kulcs***	**köv**
1	6	764	7
2	**0**	**247**	**3**
3	**2**	423	6
4			
5	7	987	9
6	3	356	1
7	1	123	5
8			
9	5	276	10
10	9	839	0

fej	**3**

A			
index	**elő**	***kulcs***	**köv**
1	**2**	764	7
2	**6**	**247**	**1**
3	0	423	6
4			
5	7	987	9
6	3	356	**2**
7	1	123	5
8			
9	5	276	10
10	9	839	0

4.5. példa. Az 1. példabeli listából töröljük a 356-os kulcsú elemet. A kulcs alapján először az elemet meg kell keresni, hogy ismerjük a helyét (a

tömbelem indexet). A **fej**-töl elindulva az első elem a 3-as indexű, az ő kulcsa 423, ami nem jó nekünk. A 3-as **köv** mutatója 6, és a 6-os indexű kulcsa 356, amit kerestünk. Tehát a listából a 6-os indexűt kell törölni, ami a lista láncából történő kiláncolást jelenti, tehát maga az elem nem semmisül meg. A kiláncoláshoz megnézzük, hogy melyik a megelőző elem. Ez az **elő** mutató szerint a 3-as. Ezért a 3-as **köv** mutatóját lecseréljük a 6-os **köv** mutatójára, azaz 1-re, és a 6-os **köv** mutatója szerinti 1-es indexű elem **elő** mutatóját lecseréljük a 6-os **elő** mutatójára, azaz 3-ra. A törléshez tehát elegendő volt két mutatót megváltoztatni. Az eredmény az alábbi táblázatban látható:

fej	**3**

A			
index	**elő**	***kulcs***	**köv**
1	**3**	764	7
2			
3	0	423	**1**
4			
5	7	987	9
6	3	356	1
7	1	123	5
8			
9	5	276	10
10	9	839	0

4.3. A verem és az objektum lefoglalás/felszabadítás

4.6. definíció. *A verem adatstruktúra*
A verem (stack) olyan dinamikus halmaz, amelyben előre meghatározott az az elem, melyet a TÖRÖL eljárással eltávolítunk. Ez az elem mindig az időben a legutoljára a struktúrába elhelyezett elem lesz. Műveletei: beszúrás (push), törlés (pop).

Az ilyen törlési eljárást Utolsóként érkezett – Elsőként távozik (Last In – First Out, LIFO) eljárásnak nevezzük.

A verem jele S, attributuma a **tető**[S], amely egy mutató, a legutóbb betett (beszúrt) elemre mutat. Itt a tömbös realizációt mutatjuk be. A vermet egy S tömb valósítja meg. A kezdő tömbindex az 1-es. Az üres verem esetén a **tető**[S] tartalma zérus. Beszúrásnál a **tető**[S] tartalma nő eggyel, és az elem a **tető**[S] által mutatott indexű rekeszbe kerül. Törlésnél csak a **tető**[S] tartalma csökken eggyel elem nem semmisül meg. Figyelnünk kell, hogy lehetetlen esetben ne végezzük el a műveletet. Nincs beszúrás, ha betelt a tömb, nincs törlés, ha üres a verem. Érdemes ezeket a vizsgálatokat külön eljárásként megírni. Alább következnek a megfelelő pszeudokódok.

	3.3.1. algoritmus
	Verem megtelt-e
	// $T(n) = \Theta(1)$
	// tömbös realizáció
1	VEREM_MEGTELT (S)
2	// Input paraméter: S – a vermet tároló tömb
3	// Az algoritmus IGAZ-at ad vissza, ha a verem megtelt és HAMIS-at, ha nem
4	//
5	**IF tető**[S]=**tömbméret**[S]
6	**THEN RETURN (IGAZ)**
7	**ELSE RETURN (HAMIS)**

	3.3.2. algoritmus
	Verembe beszúrás (push)
	// $T(n) = \Theta(1)$
	// tömbös realizáció
1	VEREMBE (*S, x, hibajelzés*)
2	// Input paraméter: S – a vermet tároló tömb
3	// x – a beszúrandó elem
4	// Output paraméter: *hibajelzés* - jelzi az eredményességet
5	//
6	IF VEREM_MEGTELT (S)
7	THEN hibajelzés ← „túlcsordulás”
8	ELSE INC(**tető**[S])
9	$S_{\textbf{tető}[S]} \leftarrow x$
10	hibajelzés ← „rendben”
11	**RETURN** (*hibajelzés*)

	3.3.3. algoritmus
	Verem üres-e
	// $T(n) = \Theta(1)$
	// tömbös realizáció
1	ÜRES_VEREM (S)
2	// Input paraméter: S – a vermet tároló tömb
3	// Az algoritmus IGAZ-at ad vissza, ha a verem üres és HAMIS-at, ha nem
4	//
5	**IF tető**[S]=0
6	**THEN RETURN** (IGAZ)
7	**ELSE RETURN** (HAMIS)

	3.3.4. algoritmus
	Veremből törlés (pop)
	$T(n) = \Theta(1)$
	// Tömbös realizáció
1	VEREMBŐL (*S, x, hibajelzés*)
2	// Input paraméter: S – a vermet tároló tömb
3	// Output paraméter: x – *a kivett elem*
4	// *hibajelzés* - jelzi az eredményességet
5	//
6	**IF** ÜRES_VEREM(S)
7	**THEN** *hibajelzés* ← „alulcsordulás”
8	**ELSE** $x \leftarrow S_{\mathbf{tető}[S]}$
9	**DEC(tető[S])**
10	hibajelzés ← „rendben”
10	**RETURN** (*x, hibajelzés)*

4.7. példa. Helyezzük el egy kezdetben üres verembe a megadott sorrendben érkező 342, 416, 112 kulcsú elemeket, majd ürítsük ki a veremet! A verem maximális mérete hat elem.

A verem feltöltése. Beszúrások (push)			
tető 0	**tető 1**	**tető 2**	**tető 3**
1	**1** 342	**1** 342	**1** 342
2	**2**	**2** 416	**2** 416
3	**3**	**3**	**3** 112
4	**4**	**4**	**4**
5	**5**	**5**	**5**
6	**6**	**6**	**6**

A verem kiürítése. Törlések (pop)		
tető 2	**tető 1**	**tető 0**
1 342	**1** 342	**1** 342
2 416	**2** 416	**2** 416
3 112	**3** 112	**3** 112
4	**4**	**4**
5	**5**	**5**
6	**6**	**6**

A törléseknél láthatóan az elemek nem törlődtek fizikailag, csak már nem elérhetők. Újabb beszúrások esetén természetesen felülíródnak az új elemmel és akkor végképpen elvesznek.

A verem realizálható olyan egyszeresen láncolt, nemrendezett, nemciklikus listával is, ahol a beszúrás mindig a lista első eleme elé történik, és a törlés mindig az első elemet törli.

Egy szép alkalmazása a veremnek és a listának együtt a memóriaterület (objektum) lefoglalása és felszabadítása. Legyen m rekeszünk az adatrekordok tárolására. Legyen $n < m$ rekesz lefoglalva listás szerkezettel. Szabadon áll m - n rekesz további elemek számára. Ezeket a szabad rekeszeket egy egyszeresen láncolt listában tartjuk nyilván. A **szabad** *globális mutató* mutat az első szabad helyre. Mindig az első szabad helyet foglaljuk le, vagy a felszabadult hely a szabadok közé az első helyre kerül. Tehát a szabad rekeszek egy vermet alkotnak. A felszabadult rekesz a verembe kerül, rekesz igénylés esetén pedig a veremből elégítjük ki az igényt. Két művelet jelentkezik, az objektum lefoglalása és az objektum felszabadítása. Pszeudokódjaik:

	3.3.5. algoritmus
	Objektum lefoglalás
	// $T(n) = \Theta(1)$
1	OBJEKTUMOT_LEFOGLAL (x)
2	// Output paraméter: x a lefoglalt hely mutatója
3	// $x \leftarrow$ **szabad**
4	IF **szabad** $\neq$ **NIL**
5	THEN **szabad** $\leftarrow$ **köv**[x]
6	**RETURN** (x)

	3.3.6. algoritmus
	Objektum felszabadítás
	// $T(n) = \Theta(1)$
1	OBJEKTUMOT_FELSZABADÍT(x)
2	// Input paraméter: x – mutató, amely a felszabadítandó elemre mutat
3	**köv**[x] $\leftarrow$ ***szabad***
4	**szabad** $\leftarrow x$
5	**RETURN**

4.8. példa. Legyen adott 6 rekesz – egy hatelemű tömb - tárolási célra. A tömb kezdetben üres. Végezzük el a megadott sorrendben a felsorolt kulcsokkal a műveleteket. Ha a kulcs előtt egy T betű áll, akkor azt törölni kell, ha nem áll semmi, akkor be kell szúrni. Az elemeket kétszeresen láncolt listában tartjuk nyilván, a beszúrás az egyszerűség kedvéért történjen mindig a lista elejére és az objektum lefoglalás/felszabadítás vermes módszerét alkalmazzuk. A kulcsok felsorolása: 987, 654, 365, 247, T654, 123, T247, 235.

fej	0	**Szabad**	1
	elő	**kulcs**	**köv**
1			2
2			3
3			4
4			5
5			6
6			0

fej	1	**Szabad**	2
	elő	**kulcs**	**köv**
1	0	987	0
2			3
3			4
4			5
5			6
6			0

fej	2	**Szabad**	3
	elő	**kulcs**	**köv**
1	2	987	0
2	0	654	1
3			4
4			5
5			6
6			0

fej	3	**Szabad**	4
	elő	**kulcs**	**köv**
1	2	987	0
2	3	654	1
3	0	365	2
4			5
5			6
6			0

fej	4	**Szabad**	5
	elő	**kulcs**	**köv**
1	2	987	0
2	3	654	1
3	4	365	2
4	0	247	3
5			6
6			0

fej	4	**Szabad**	2
	elő	**kulcs**	**köv**
1	3	987	0
2	3	654	5
3	4	365	1
4	0	247	3
5			6
6			0

fej	2	**Szabad**	5
	elő	**kulcs**	**köv**
1	3	987	0
2	0	123	4
3	4	365	1
4	2	247	3
5			6
6			0

fej	2	**Szabad**	4
	elő	**kulcs**	**köv**
1	3	987	0
2	0	123	3
3	2	365	1
4	2	247	5
5			6
6			0

fej	4	**Szabad**	5
	elő	**kulcs**	**köv**
1	3	987	0
2	4	123	3
3	2	365	1
4	0	235	2
5			6
6			0

4.4. A sor

4.9. definíció. *A sor adatstruktúra*
A sor (queue) olyan dinamikus halmaz, amelyben előre meghatározott az

az elem, melyet a TÖRÖL eljárással eltávolítunk és az az elem is amelyet a BESZÚR eljárással a halmazba beteszünk. Törlésre mindig az elemek közül a legrégebben beszúrt kerül. A beszúrt elem lesz a legfrissebb elem. Műveletek: beszúrás, törlés.

Az ilyen törlést Elsőként érkezik – Elsőként távozik (First In – First Out, FIFO) eljárásnak nevezzük. Ha az elemeket lineárisan egymás mellé felsorakoztatjuk az érkezésük sorrendjében, akkor szemléletesen mondhatjuk, hogy törlésre mindig az első helyen álló elem kerül, a beszúrás pedig az utolsó elemet követő helyre történik. A sornak, mint adatstruktúrának többféle realizációja lehetséges. Itt most a tömbös realizációval foglalkozunk. Legyen a tömb neve Q és legyen a **tömbméret**[Q] attributum értéke n, azaz a tömb n elemű. Az elemek indexelése legyen $1, 2, 3, \ldots, n$. Tömbös realizáció esetén a tömb a méreténél legalább eggyel kevesebb elemű sort képes csak tárolni. A sor attributumai:

Attributum	Leírás
fej[Q]	A sor első elemének a tömbelem indexe.
vége[Q]	A sor utolsó elemét követő tömbelem indexe. Az új elem ide érkezik majd, ha még van hely.

A sor elemei a tömbben a **fej**[Q], **fej**[Q]+1, **fej**[Q]+2,. . . ,**vége**[Q]-1 indexű helyeket foglalják el. Ebben az index felsorolásban az indexek ciklikusan követik egymást, azaz az n index után az 1 index következik, ha erre szükség van. A műveletek szempontjából a törlés esetében nyilvánvaló, hogy üres sorból nem lehet elemet törölni. Az üres sor felismerhető abból, hogy **fej**[Q]=**vége**[Q]. (Kezdetben **fej**[Q]=**vége**[Q]=1.). Ha üres sorból veszünk ki elemet, az hiba (alulcsordulás). A beszúrás esetében nem szabad azt elvégezni, ha a sor már megtelt. Ez a jelenség felismerhető abból, hogy **fej**$[Q]$ = **vége**$[Q] + 1$. A beszúrás tele sorba hibát eredményez (túlcsordulás). Nézzük a műveletek pszeudokódjait:

	3.4.1 algoritmus
	Sorba beszúrás
	// $T(n) = \Theta(1)$
	// Tömbös realizáció
1	SORBA (*Q, x, hibajelzés*)
2	//Input paraméter: Q – a tömb
3	// x – a beszúrandó elem
4	//Output paraméter: *hibajelzés* - jelzi az eredményességet
5	IF **fej**[Q]=**vége**[Q]+1
6	**THEN** *hibajelzés*←„tele sor”
7	**RETURN**(*hibajelzés*)
8	Q[**vége**[Q]] ←x
9	IF **vége**[Q] = **tömbméret**[Q]
10	THEN **vége**[Q] ← 1
11	ELSE INC(**vége**[Q])
12	hibajelzés←„sikeres beszúrás”
13	**RETURN** (*hibajelzés*)

	3.4.2 algoritmus
	Sorból eltávolítás
	// $T(n) = \Theta(1)$
	// Tömbös realizáció
1	SORBÓL (*Q, x, hibajelzés*)
2	//Input paraméter: *Q – a tömb*
3	//Output paraméter: *x – az eltávolított elem*
4	// *hibajelzés – jelzi az eredményességet*
5	// IF **fej**[Q]=**vége**[Q]
6	**THEN** *hibajelzés*← „üres sor”
7	**RETURN** (*hibajelzés*)
8	x ← Q[**fej**[Q]
9	**IF fej**[Q] = **tömbméret**[Q]
10	**THEN fej**[Q] ← 1
11	**ELSE** INC(**fej**[Q])
12	Hibajelzés←„sikeres eltávolítás”
13	**RETURN** (*x, hibajelzés*)

4.10. példa. Végezzük el az alábbi műveleteket egy üres sorból kiindulva. A következő felsorolásban egy kulcsérték annak beszúrását jelenti. A T betű a sorból eltávolítást szimbolizálja. A tömb 5 elemű. A felsorolás: 345, 231, 768, T, 893, T, 259, 478. Az eredmény alább látható. Minden egyes lépésben megmutatjuk a sor állapotát. Vastagítottuk a sorhoz hozzátartozó elemeket.

		kezdet		345		231		768		T		893		T		259		478
fej		1		1		1		1		2		2		3		3		3
vége		1		2		3		4		4		5		5		1		2
1				**345**		**345**		**345**		345		345		345		345		**478**
2						**231**		**231**		**231**		**231**		231		231		231
3								**768**		**768**		**768**		**768**		**768**		**768**
4												**893**		**893**		**893**		**893**
5																**259**		**259**

Sor realizálható láncolt listával is. Törölni mindig a lista első elemét töröljük, beszúrni pedig mindig a lista utolsó eleme után szúrunk be. Elegendő egyszeresen láncolt listával dolgozni, viszont ilyenkor érdemes a lista végének a mutatóját is külön tárolni. Ha kétszeresen láncolt listát használunk, akkor már a ciklikus lista használata a hatékonyabb és ekkor a listavég mutatót nem kell külön tárolni.

5. fejezet

Keresés, rendezés egyszerű struktúrában (tömb)

5.1. Keresés

5.1.1. Lineáris keresés

A tömb adatstruktúrában a keresés műveletét részint megtárgyaltuk a 3.1. fejezetben. Két esetet különböztettünk meg, a rendezetlen és a rendezett tömb esetét. Rendezetlen tömbben egy adott k kulcsú elem megkereséséhez nem áll rendelkezésre semmilyen információ azon kívül, hogy az elemek lineárisan követik egymást. Hiába érhetők el az elemek tetszőleges sorrendben, minden elemet meg kell vizsgálni, hogy a kulcs megegyezik-e a keresett k kulccsal, ugyanis azt sem tételeztük föl, hogy például a kulcsok számok. A kulcsok természete lehet olyan is, hogy mondjuk a rendezésükről szó nem lehet. (Nevezzünk meg ilyen kulcsokat!) Ez pedig azt jelenti, hogy a lineáris keresésnél jobb növekedési rendű időbonyolultsággal rendelkező algoritmus nem adható. A keresés rendezetlen tömbben lineáris idejű, azaz a keresési algoritmus időbonyolultsága $T(n) = \Theta(n)$. Ez egy aszimptotikus $T(n) \approx c \cdot n$ összefüggést jelent (pontosítva: $\lim_{n\to\infty} \frac{T(n)}{n} = c$), melyben c egy pozitív konstans. Nem mindegy azonban ennek a konstansnak a konkrét értéke. Azt megtehetjük, hogy a lineáris keresési algoritmust némiképpen módosítva ezt a konstanst lejjebb szorítjuk. Tekintsük például a 3.1.1. keresés_tömbben algoritmust. Legyen az algoritmus i számmal számozott sorának a végre-

hajtási ideje c_i. Tételezzük fel a számolási idő szempontjából a legrosszabb esetet, hogy a keresett elem nincs a tömbben. Ekkor a keresés ideje:

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + n \cdot (c_9 + c_{10}) + c_{11} + c_{12} \\ &= c_7 + c_8 + n \cdot (c_9 + c_{10}) + c_{11} + c_{12} \end{aligned}$$

Nem túl sokat torzítunk a valóságon, ha feltételezzük, hogy az értékadás és egy relációvizsgálat valamint logikai művelet (ÉS) körülbelül azonosan $\bar{c}$ ideig tart. Akkor $c_7 = c_8 = c_{10} = c_{11} = c_{12} = \bar{c}$, $c_9 = 3\bar{c}$ és így

$$T(n) = \bar{c} + \bar{c} + n \cdot (3\bar{c} + \bar{c}) + \bar{c} + \bar{c} = 4\bar{c}n + 4\bar{c}$$

A $T(n) = \Theta(n)$-nek megfelelő aszimptotikus kifejezésben szereplő $c$konstans értéke $4\bar{c}$-nek vehető. Módosítsuk most úgy a 3.1.1. algoritmust, hogy a keresés kezdetén a keresett k kulcsot a tömb végéhez hozzáfüggesztjük. Feltesszük, hogy erre van elegendő hely. Ebben az esetben az elem biztosan benne lesz a tömbben. A keresésből a tömbelem indexének végvizsgálata kihagyható. A keresés mindig sikeres lesz, csak ha a visszakapott index nagyobb, mint az eredeti tömb utolsó elemének indexe, akkor valójában az elem nincs a tömbben. Íme a megváltoztatott algoritmus pszeudokódja:

	4.1.1.1. algoritmus
	Módosított keresés tömbben
	// $T(n) = \Theta(n)$
1	KERESÉS_TÖMBBEN (A,k, x)
2	// Input paraméter: A - a tömb
3	// k – a keresett kulcs
4	// Output paraméter: x - a k kulcsú elem pointere (indexe), ha van ilyen elem vagy NIL, ha nincs
5	// Lineárisan keresi a k kulcsot.
6	//
7	$x \leftarrow$ **fej**[A]
8	INC(**vége**[A])
9	**kulcs**[$A_{\textbf{vége}[A]}$] $\leftarrow k$
10	**WHILE kulcs**[A_x] $\neq k$ **DO**
11	INC(x)
12	DEC(**vége**[A])
13	**IF** $x >$ **vége**[A]
14	**THEN** $x \leftarrow$ **NIL**
15	**RETURN** (x)

Most a legrosszabb eset ideje

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + n \cdot (c_{10} + c_{11}) + c_{12} + c_{13} + c_{14} \\ &= c_7 + c_8 + c_9 + n \cdot (c_{10} + c_{11}) + c_{12} + c_{13} + c_{14}. \end{aligned}$$

Itt azt feltételezhetjük, hogy $c_7 = c_8 = c_9 = c_{10} = c_{11} = c_{12} = c_{13} = c_{14} = \bar{c}$, amiből

$$T(n) = \bar{c} + \bar{c} + \bar{c} + n \cdot (\bar{c} + \bar{c}) + \bar{c} + \bar{c} + \bar{c} = 2\bar{c}n + 6\bar{c}$$

Itt az aszimptotikus kifejezés konstansa $2\bar{c}$, tehát ezzel a kis trükkel ha a futási idő jellegét (linearitását) nem is, de a konkrét idejét közel felére sikerült csökkenteni a 3.1.1. algoritmus idejéhez képest.

5.1.2. Logaritmikus keresés

Rendezett tömb esetén láttuk a 3.1. fejezetben, hogy a lineáris időnél jobbat is el tudunk érni a bináris kereséssel (3.1.4. algoritmus), amely logaritmikus időt ad. A bináris keresés mellett vele konkuráló érdekes algoritmus a *Fibonacci keresés* algoritmusa. Feltételezzük, hogy a kulcsokat (és az adatrekordokat) tartalmazó tömb neve A, mérete n, és a tömbelemek indexelése egytől indul. A rekordok a kulcsok növekvő sorrendje szerint követik egymást, a kulcsok pedig mind különbözőek. Alább megadjuk szövegesen a Fibonacci keresés algoritmusát. A felírást azon feltétel mellett tesszük meg, hogy $n+1$ legyen egyenlő az F_{k+1} Fibonacci számmal. (Az algoritmus módosítható tetszőleges pozitív egész n esetére is.)

A Fibonacci keresés algoritmusa:

1.	Kezdeti beállítások: $i \leftarrow F_k$, $p \leftarrow F_{k-1}$, $q \leftarrow F_{k-2}$	
2.	Összehasonlítás:	Ha $k < \mathbf{kulcs}[A_i]$, akkor a 3. pont következik Ha $k > \mathbf{kulcs}[A_i]$, akkor a 4. pont következik Ha $k = \mathbf{kulcs}[A_i]$, akkor sikeres befejezés.
3.	Az i csökkentése:	Ha $q = 0$, akkor sikertelen befejezés. Ha $q \neq 0$, akkor $i \leftarrow i - q$, $\begin{pmatrix} p \\ q \end{pmatrix} \leftarrow \begin{pmatrix} q \\ p-q \end{pmatrix}$ és a 2.pont következik.
4.	Az i növelése:	Ha $p = 1$, akkor sikertelen befejezés. Ha $p \neq 1$, akkor $i \leftarrow i + q$, $p \leftarrow p - q$, $q \leftarrow q - p$ és a 2. pont következik.

Az eddigi keresési algoritmusok csak a rendezettség tényét használták ki, lényegtelen volt a kulcsok milyensége. Ha föltételezzük, hogy a kulcsok számok, akkor használhatjuk az úgynevezett *interpolációs keresést*. A módszer hallgatólagosan feltételezi, hogy a kulcsok növekedésükben körülbelül egyenletes

eloszlásúak (majdnem számtani sorozatot alkotnak). Az átlagos keresési idő: $T(n) = \Theta(\log \log n)$. Az elv azon alapszik, hogy a feltételezéseink mellett a keresett k kulcs a sorban az értékének megfelelő arányosság szerinti távolságra van a keresési intervallum balvégétől. Azaz ha a balvég indexe b, a jobbvégé j, a megfelelő kulcsok k_b és k_j, akkor a következő vizsgálandó elem indexe $b + \frac{(j-b)\cdot(k-k_b)}{k_j-k_b}$. Ha a keresett kulcs megegyezik ezen elem kulcsával, akkor az algoritmus sikeresen befejeződik. Ha a k kulcs értéke kisebb, akkor az intervallum jobbvégét, ha a k kulcs nagyobb, akkor a balvégét cseréljük le erre a közbülső elemre és az új intervallummal folytatjuk a keresést. Az algoritmust nem részletezzük.

5.1.3. Hasító táblák

A hasító táblák algoritmusai tömböt használnak a kulcsok (rekordok) tárolására, de nem az eddig megszokott értelemben, vagyis a tömböt általában nem töltik fel teljesen és a rekordok nem feltétlenül hézagmentesen helyezkednek el a tömbben. Az algoritmusok a keresésre, módosításra, beszúrásra és a törlésre vannak kihegyezve, tehát ezek a műveletek végezhetők el a struktúrán hatékonyan. Például a legkisebb kulcs megkeresése a struktúrában már nem olyan hatékony, mint a fent nevezettek. Az alapvető problémát az okozza, és ez az oka ezen adatstruktúra bevezetésének, hogy a kulcsok elméletileg lehetséges U halmaza - az úgynevezett *kulcsuniverzum* – számottevően bővebb, mint a konkrétan szóbajöhető kulcsok halmaza, amelyet ráadásul még csak nem is ismerünk pontosan. Egy példával világítjuk ezt meg. Legyen adott egy cég, amelyről ismert, hogy legfeljebb 5000 alkalmazottja van. Minden alkalmazottról bizonyos adatokat nyilván kell tartani a különböző adminisztrációs feladatok elvégzéséhez. Ezen adatok egyike a TAJ-szám, amely kilencjegyű, előjel nélküli egész szám. Ezt az adatot néztük ki magunknak kulcs céljára, mivel a TAJ-szám egyértelműen azonosítja a személyt. Ha csak ennyit tudunk a TAJ számról – és most nem is akarunk annak mélyebb ismereteiben elmélyedni, - akkor ez 10^9 lehetséges kulcsot jelent. Ennyi eleme van a kulcsuniverzumnak. Ebből az írdatlan mennyiségű kulcsból nekünk viszont csak körülbelül 5000 kell. Azaz a kulcsuniverzumnak csak egy viszonylag szűk részhalmaza, (a teljes halmaz körülbelül $0,0005\%$-a). Azt viszont nem tudjuk, hogy melyik részhalmaz. A kulcsokat majd a munkatársak hozzák magukkal. Ráadásul a személyi mozgás, fluktuáció révén ezek a kulcsok

változhatnak is. Teljességgel nyílvánvaló, hogy értelmetlen lenne az adatbázisunkban egy milliárd rekord számára helyet biztosítani. Elég, ha egy kis ráhagyással mondjuk körülbelül 6000 rekordnak foglalunk le helyet (20% körüli ráhagyás). Ezen a helyen kell az 5000 rekordot úgy elhelyezni, hogy a rekordok keresése, módosítása, beszúrása, törlése hatékony legyen. Azt a táblázatot (tömböt), ahol a rekordokat, vagy a rekordokra mutató mutatókat (pointereket) elhelyezzük, *hasító táblázat*nak, hasító táblának nevezzük az angol *hash table* elnevezés után. A hasító tábla elemeinek indexelése nulláról indul. A tábla elemeit *rés*nek is szokás nevezni. Külön érdemes kihangsúlyozni a módosítás műveletét, amely tulajdonképpen két részből áll, egy keresésből, majd a megtalált rekord módosításából. Ha ez a módosítás a rekord kulcsmezejét érinti, akkor a rekordot a táblából először törölni kell, majd a módosítás elvégzése után újra be kell szúrni az új kulcsnak megfelelően.

*Közvetlen címzésű táblázat*ról beszélünk, ha a kulcsuniverzum az $U = \{0, 1, \ldots, M-1\}$ számok halmaza, ahol az M egy mérsékelt nagyságú szám. A tárolási célra használandó tábla (tömb) mérete legyen m, amit most válasszunk $M = m$-nek. Ekkor a kulcs egyúttal az index szerepét játszhatja, azaz a kulcsuniverzum minden kulcsa egyidejűleg tárolható. Ha valamely kulcsot nem tároljuk, akkor a helye, a rés üres lesz. Az üres rést az jelenti, hogy a rés tartalma NIL. (Pointeres változat.) A keresés, beszúrás, törlés algoritmusai ekkor rém egyszerűek, pszeudokódjaik következnek alább. Mindegyik művelet időigénye konstans, $T(n) = \Theta(1)$. A tömb neve T, utalásképpen a táblázatra.

	4.1.3.1. algoritmus
	Közvetlen címzésű keresés hasító táblában
	// $T(n) = \Theta(1)$
1	KÖZVETLEN_CÍMZÉSŰ_KERESÉS (T, k, x)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// k – a keresett kulcs
4	// Output paraméterek: x – a keresett elem indexe, NIL, ha nincs
5	//
6	$x \leftarrow T_k$
7	**RETURN** (x)

	4.1.3.2. algoritmus
	Közvetlen címzésű beszúrás hasító táblába
	// $T(n) = \Theta(1)$
1	KÖZVETLEN_CÍMZÉSŰ_BESZÚRÁS (T, x)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// x – mutató a beszúrandó elemre
4	//
5	$T_{\mathbf{kulcs}}[x] \leftarrow x$
6	**RETURN**

Az ismertetett eset nagyon szerencsés és nagyon ritka. Általában M értéke lényegesen nagyobb, mint a ténylegesen tárolható kulcsok m száma. A memóriaigény leszorítható $\Theta(m)$-re úgy, hog az átlagos időigény $\Theta(1)$ maradjon a *láncolt hasító tábla* alkalmazásával. Ebben a táblában minden elem egy listafej mutatója, amely kezdetben az üres táblázat esetén mindenütt **NIL**. Most nem tételezzük fel, hogy az U kulcsuniverzum a $0, 1, \ldots, m-1$ számok halmaza lenne, de feltételezzük, hogy ismerünk egy úgynevezett *hasító függvényt*, amely az U kulcsuniverzum elemeit képezi bele ebbe a $0, 1, dots, m-1$ számhalmazba, az indexek halmazába: $h : U \rightarrow \{0, 1, \ldots, m-1\}$. Ez a

függvény egyáltalán nem lesz injektív, azaz nem fog feltétlenül különböző kulcsokhoz különböző számértéket rendelni, hiszen az U elemszáma sokkal több, mint a $0, 1, dots, m-1$ indexhalmazé. (Ezt a viszonyt az $M >> m$ jelöléssel szoktuk jelezni.) A célunk a hasító függvénnyel az, hogy a k kulcsú rekord a tábla $h(k)$ indexű réséből indított láncolt listába kerüljön. Ezzel a stratégiával oldjuk fel az úgynevezett *ütközési problémát*, ami akkor lép fel, ha két különböző kulcs ugyanarra az indexre (résre) képeződik le. (Az ütközésnek nem kicsi az esélye. Ha egy tízemeletes ház földszintjén négyen belépnek a liftbe és mindenki a többitől függetlenül választ magának egy emeletet a tíz közül, akkor $10 \cdot 10 \cdot 10 \cdot 10 = 10000$-féleképpen választhatnak. Ebből a 10000-ből csak $10 \cdot 9 \cdot 8 \cdot 7 = 5040$ olyan van, amikor mindenki a többitől eltérő emeletet választott. Ha továbbá minden ilyen választást azonos esélyűnek tekintünk, akkor annak esélye, hogy legalább két ember ugyanazt az emeletet választotta eszerint $\frac{1000-5040}{10000} = 0,496$. Tehát majdnem 50% eséllyel lesznek olyanok, akik ugyanarra az emeletre mennek. A híres von Mises féle születésnap probléma esetén elegendő legalább 23 embernek összejönni, hogy legalább 50% eséllyel legyen köztük legalább kettő olyan, akik azonos napon ünneplik a születésnapjukat.) Egy elemnek a listában történő elhelyezése történhet a lista elejére történő beszúrással, vagy készíthetünk rendezett listát is, ha a kulcsok rendezhetők. Az egyes műveletek pszeudokódjai alább következnek. Az egyes műveletek idejeivel kapcsolatban bevezetünk egy fogalmat, az úgynevezett *telítettségi arányt*, vagy *telítettségi együthatót*.

5.1. definíció. *A telítettségi arány*
Az $\alpha = \frac{n}{m}$ számot a hasító tábla telítettségi arányának nevezzük, ahol m a tábla réseinek a száma, n pedig a táblába beszúrt kulcsok száma.

A telítettségi arány láncolt hasító tábla esetén nemnegatív szám, amely lehet 1-nél nagyobb is. Szokásos elnevezése még a *kitöltési arány* is.

	4.1.3.4. algoritmus
	Láncolt hasító keresés
	// $T(n) = \Theta(1+\alpha)$
1	LÁNCOLT_HASÍTÓ_KERESÉS (*T, k, x*)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// k a keresett kulcs
4	// Output paraméterek: x - a k kulcsú rekord mutatója, **NIL** ha a rekord nincs a struktúrában
5	A k kulcsú elem keresése a $T_{h(k)}$ listában, melynek mutatója x lesz.
6	**RETURN** (x)

	4.1.3.5. algoritmus
	Láncolt hasító beszúrás
	// $T(n) = \Theta(1+\alpha)$
1	LÁNCOLT_HASÍTÓ__BESZÚRÁS (*T, x*)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// x – mutató a beszúrandó elemre
4	
5	Beszúrás a $T_{h(\mathbf{kulcs}[x])}$ lista elejére
6	**RETURN**

	4.1.3.6. algoritmus
	Láncolt hasító törlés
	// $T(n) = \Theta(1+\alpha)$
1	LÁNCOLT_HASÍTÓ_TÖRLÉS (*T, x*)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// x – mutató a törlendő elemre
4	//
5	x törlése a $T_{h(\mathbf{kulcs}[x])}$ listából
6	**RETURN**

Vezessünk most be két jelölést. A megvizsgált kulcsok átlagos számát jelölje C_n a sikeres keresés esetén és $C_n^{'}$ a sikertelen keresések esetén.

5.2. tétel. A láncolt hasító tábla időigénye
Ha α a kitöltési arány, akkor a láncolt hasító táblában

$$C_n = \Theta\left(1 + \alpha\right)$$

és

$$C_n^{'} = \Theta\left(1 + \alpha\right).$$

A láncolt hasító tábla mérete nem korlátozza a struktúrában elhelyezett rekordok számát. Természetesen ha a rekordok száma igen nagy, akkor az egyes résekhez tartozó listák mérete is igen nagy lehet. Nem ritkán azonban ismeretes egy felső korlát a rekordok számára és azok (vagy a kulcsaik, vagy a mutató a rekordra) elhelyezhetők magában a táblázatban. Minden táblabeli elem (rés) legalább két mezőből fog állni az alábbi tárgyalásmódban, egy kulcsmezőből és egy mutatóból, amely a következő elemre mutat. Minden réshez tartozik egy foglaltsági bit, amely szerint a rés lehet szabad, vagy lehet foglalt. Közöljük két algoritmus pszeudokódját. Az első a megadott kulcsú elemet keresi a táblában. Ha megtalálta, akkor visszaadja az elem indexét, ha nem találta meg, akkor **NIL**-t ad vissza. A második a megadott kulcsú elemet beszúrja a táblába, ha az elem nincs a táblában és van még ott üres hely. Ha az elem benne lenne a táblában, akkor az algoritmus visszatér. Az algoritmus jellegzetessége, hogy a különböző résekhez tartozó listák egymásba nőnek. Az üres helyek adminisztrálása céljából bevezetünk egy r változót, amely mindig azt fogja mutatni, hogy az r és a magasabb indexű helyeken a táblaelemek már foglaltak. Az r a tábla attributuma lesz. Üres táblára $r = m$, minden rés szabad és a **köv** mutatók mindegyike **NIL**.

	4.1.3.7. algoritmus
	Összenövő listás hasító keresés
	// $T(n) = \Theta(1+\alpha)$
1	ÖSSZENÖVŐ_LISTÁS_HASÍTÓ_KERESÉS (T, k, x)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// k - a keresett kulcs
4	// Output paraméterek: x - a k kulcsú rekord mutatója, **NIL** ha a rekord nincs a struktúrában
4	//
5	$i \leftarrow h(k)$
6	**IF** T_i foglalt
7	**THEN REPEAT**
8	**IF** $k =$ **kulcs**$[T_i]$
9	**THEN** $x \leftarrow i$
10	**RETURN** (x)
11	**IF** **kulcs**$[T_i] \neq$ **NIL**
12	**THEN** $i \leftarrow$ **köv**$[T_i]$
13	**UNTIL** **köv**$[T_i] =$ **NIL**
14	$x \leftarrow$ **NIL**
15	**RETURN**(x)

	4.1.3.8. algoritmus
	Összenövő listás hasító beszúrás
	// $T(n) = \Theta(1+\alpha)$
1	ÖSSZENÖVŐ_LISTÁS_HASÍTÓ_BESZÚRÁS(*T, k, hibajelzés*)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// k - a beszúrandó kulcs
4	// Output paraméterek: *hibajelzés* – a művelet eredményességét jelzi
5	$i \leftarrow h(k)$
6	**IF** T_i szabad
7	**THEN** **kulcs**$[T_i] \leftarrow k$
8	**köv**$[T_i] \leftarrow$ **NIL**
9	*hibajelzés* ←„Sikeres beszúrás”
10	**RETURN** (*hibajelzés*)
11	**ELSE** **REPEAT**
12	**IF** $k =$ **kulcs**$[T_i]$
13	**THEN** *hibajelzés*←„Sikeres beszúrás”
14	**RETURN** (*hibajelzés*)
15	**IF** **kulcs**$[T_i] \neq$ **NIL**
16	**THEN** $i \leftarrow$ **köv**$[T_i]$
17	**UNTIL köv**$[T_i] =$ **NIL**
18	// Nincs a táblában, be kell szúrni
19	**IF** $R \leq 0$
20	**THEN** *hibajelzés* ←„Betelt a tábla”
21	**RETURN** (*hibajelzés*)
22	**REPEAT**
23	DEC (r)
24	**IF** T_r szabad
25	**THEN** **kulcs**$[T_r] \leftarrow k$
26	**köv**$[T_r] \leftarrow$ **NIL**
27	**köv**$[T_i] \leftarrow r$
28	*hibajelzés* ← „Sikeres beszúrás”
29	**RETURN** (*hibajelzés*)
30	**UNTIL** $r \leq 0$
31	*hibajelzés* ← „Betelt a tábla”
32	**RETURN** (*hibajelzés*)

A törlés műveletét itt nem tárgyaljuk, hanem külön diszkusszió tárgyává tesszük a feladatok között. A keresés és beszúrás műveletének átlagos idejére érvényesek az alábbi közelítő formulák:

$$C_n \approx 1 + \frac{1}{8\alpha}\left(e^{2\alpha} - 1 - 2\alpha\right) + \frac{1}{4}\alpha.$$

$$C_n' \approx 1 + \frac{1}{4}\left(e^{2\alpha} - 1 - 2\alpha\right).$$

Eddig nem szóltunk a hasító függvényről közelebbit. Egy jó hasító függvény kielégíti az *egyszerű egyenletességi feltételt*, ami azt jelenti, hogy minden kulcs egyforma eséllyel képződik le az m rés bármelyikére, amely az ütközések elleni harcban fontos. Ezen felül lényeges, hogy a függvény értéke nagyon gyorsan számítható legyen. Az igazán nem komoly probléma, hogy a kulcsok sokfélék lehetnek, hiszen általában könnyen konvertálhatók számértékké. Például ha a kulcs szöveges, akkor tekinthetjük a szöveg egyes betűinek az ASCII kódját és minden ilyen számértéket egy magasabb alapú számrendszer számjegyeinek vesszük. Ha a szöveg csak (latin) nagybetűket tartalmaz, akkor minden betűhöz hozzárendelhetjük az ábécében elfoglalt helyének az eggyel csökkentett sorszámát. A – 0, B – 1, C– 2, D – 3, E – 4, ..., Z – 25. Ekkor a „ZABA" szöveghez hozzárendelhető szám 26-os számrendszerben $25 \cdot 26^3 + 0 \cdot 26^2 + 1 \cdot 26 + 0 = 439426$. Két nagy módszer osztályt szokás kiemelni a hasító függvények kiválasztásakor, az osztó módszerű és a szorzó módszerű függvényeket.

Az osztó módszer esetében a $h(k) = k \bmod m$ formulával dolgozunk. Nem árt azonban némi óvatosság az m kiválasztásánál. Ha m páros, akkor $h(k)$ paritása is olyan lesz, mint a k kulcsé, ami nem szerencsés. Ha m a 2-nek hatványa, akkor $h(k)$ a k kulcs utolsó bitjeit adja. Általában prímszámot célszerű választani. Knuth javaslata alapján kerülendő az az m, amely osztja az $r_k \pm a$ számot, ahol k és a kicsi számok, r pedig a karakterkészlet elemeinek a száma. Például ha $r = 256$ és a kulcsok az ASCII táblázatbeli karakterek lehetnek és az $m = 2^16+1 = 65537$ Fermat-féle prímszámot választjuk, akkor mondjuk háromkarakteres kulcsok esetén a $C_1C_2C_3$ kulcsot tekinthetjük egy 256-os számrendszerbeli háromjegyű számnak is. Ha most itt m-mel osztunk,

$m = 256^2 + 1$, akkor az osztás eredménye $(C_2C_3 - C_1)_{256}$ lesz, ami azt jelenti, hogy az eredmény úgy adódik, hogy a szám első jegyét levonjuk a hátsó két jegy által alkotott számból. Ezáltal *egymáshoz közeli kulcsok egymáshoz közelre* képződnek le.

A szorzásos módszer esetében a $h(k) = \lfloor m \cdot (kA \bmod 1) \rfloor$ formulát használjuk, ahol A egy alkalmas módon megválasztott konstans, $0 < A < 1$. Igen jó tulajdonságokkal rendelkezik az

$$A = \Phi^{-1} = \frac{\sqrt{5} - 1}{2} \approx 0,618033988...$$

számérték, amelyet a Fibonacci számokkal kapcsolatban már megismerhettünk.

Nyílt címzések

A nyílt címzésű hasító táblákban nincsenek táblázaton kívül tárolt elemek, listák. A táblaelemeket (a rekordokat) a $0, 1, ..., m - 1$ indexekkel indexeljük. Az ütközések feloldására azt a módszert használjuk, hogy beszúráskor amennyiben egy rés foglalt, akkor valamilyen szisztéma szerint tovább lépünk a többi résre, míg üreset nem találunk és oda történik a beszúrás. Keresésnél szintén ha a számított résben nem a keresett kulcs van, akkor a beszúrási szisztéma szerint keressük tovább. Formálisan ezt azáltal érjük el, hogy a hasító függvényünket, amely eddig csak a kulcstól függött, most kétváltozósra terjesztjük ki, a második változója a próbálkozásunk sorszáma lesz. Ez a szám a $0, 1, 2, ..., m - 1$ számok valamelyike lehet. Azaz a függvényünk: $h : U \times (0, 1, \ldots, m - 1) \to (0, 1, \ldots, m - 1)$ és egy rögzített k kulcs esetén a $h(k, 0), h(k, 1), \ldots, h(k, m - 1)$ egy úgynevezett *kipróbálási sorozat*ot produkál. Ezek az indexek a $0, 1, ..., m - 1$ indexhalmaznak egy permutációját kell, hogy adják. Ezzel biztosítjuk, hogy ha van még hely a táblában, akkor a beszúrást minden esetben meg lehessen csinálni. Tekintsük ezután a keresés, beszúrás és törlés pszeudokódjait a nyílt címzésű hasítótábla esetén

	4.1.3.4. algoritmus
	Nyílt címzésű hasító keresés
	// $T(n) = \Theta(1+\alpha)$
1	NYÍLT_CÍMZÉSŰ_HASÍTÓ_KERESÉS (*T, k, x*)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// k - a keresendő kulcs
4	// Output paraméterek: x – a k kulcsú rekord mutatója, **NIL**, ha nincs
5	//
6	$i \leftarrow 0$
7	**REPEAT**
8	$j \leftarrow h(k,i)$
9	**IF** **kulcs**$[T_j] = k$ és **foglaltság**$[T_j]$=Foglalt
10	**THEN** $x \leftarrow j$
11	**RETURN** (x)
12	**INC** (i)
13	**UNTIL** $T_j =$ **NIL** vagy $i = m$
14	$x \leftarrow$ **NIL**
15	**RETURN** (x)

	4.1.3.5. algoritmus
	Nyílt címzésű hasító beszúrás
	// $T(n) = \Theta(1+\alpha)$
1	NYÍLT_CÍMZÉSŰ _HASÍTÓ_BESZÚRÁS (*T, k, hibajelzés*)
2	//
3	// Input paraméterek: T – a tömb zérus kezdőindexszel
4	// k - a beszúrandó kulcs
5	// Output paraméterek: *hibajelzés* – a művelet eredményességét jelzi
6	//
7	$i \leftarrow 0$
8	**REPEAT**
9	$j \leftarrow h(k,i)$
10	**IF** **foglaltság**$[T_j]$=szabad vagy **foglaltság**$[T_j]$=törölt
11	**THEN** **kulcs**$[T_j] \leftarrow k$
12	*hibajelzés* ← „ Sikeres beszúrás”
13	**RETURN** (*hibajelzés*)
14	**ELSE** **INC** (i)
15	**UNTIL** $i = m$
16	*Hibajelzés*← „tábla betelt”
17	**RETURN** (*hibajelzés*)

	4.1.3.6. algoritmus
	Nyílt címzésű hasító törlés
	// $T(n) = \Theta(1+\alpha)$
1	NYÍLT_CÍMZÉSŰ_HASÍTÓ_TÖRLÉS (T, k)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// k - a törlendő kulcs
4	
5	NYÍLT_CÍMZÉSŰ_HASÍTÓ_KERESÉS (T, k, x)
6	**IF** $x \neq$**NIL**
7	**THEN** ***foglaltság***$[T_j]$ $\leftarrow$ törölt
8	**RETURN**

Törlésnél nem megfelelő a **NIL** beírás, helyette a rés foglaltságát TÖRÖLT-re kell állítani, mivel a **NIL** a későbbi kereséseket megzavarhatja. A kipróbálási sorozat végét jelzi ott, ahol annak valójában még nincs vége. Ennek az a következménye, hogy sok beszúrás és törlés után már szinte minden rés vagy foglalt, vagy törölt lesz, ami a keresés sebességét lerontja. Ilyenkor a teljes táblát a benne lévő kulcsokkal újra hasítjuk. A másik lehetőség, hogy a láncolt listás megoldást választjuk, ahol a törlések nem okoznak gondot. Most három gyakran alkalmazott módszer típust említünk meg a nyílt címzésű hasításra.

Lineáris kipróbálás

Ez a módszer a

$$h(k,i) = ((h_0(k)) + i) \bmod m$$

hasító függvényt használja, ahol az $i = 0, 1, 2, ..., m-1$ lehet. A formula alapján látható, hogy egy h_0 alap hasító függvényből indul ki és a kipróbálási sorozat a $0, 1, 2, ..., m-1$ számokkal módosítja a kipróbálási indexet, amely nem függ a k kulcstól, tehát minden kulcsra azonos. (A kipróbálási sorozat csak a résen keresztül függ a kulcstól, az azonos résre képeződő kulcsok esetén azonos.) A hatás pedig az, hogy ha a vizsgált rés nem megfelelő, akkor tovább lépünk a magasabb indexek felé. A továbblépés ciklikus, azaz a tábla

végére érve a másik végén folytatjuk a kipróbálást. A módszernek van egy kellemetlen mellékhatása, az *elsődleges klaszterezés*. *Klaszternek* nevezzük a kipróbálási sorozat mentén az egymást követő kitöltött rések összességét. Ezen halmaz elemeinek a száma a klaszter mérete. Nem szerencsés, ha a klaszerek mérete nagy, mert ez lelassítja a hasító tábla műveleteit. Egy példa:

X	X	X				X				X		X	X	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

A hasító tábla X-szel jelölt elemei a foglaltak. A maximális klasztermérete 3, de van két egyelemű és egy kételemű klaszter is. Ha egy új kulcs a 0 indexű helyre kerülne, akkor a negyedik megvizsgált rés lenne csak megfelelő a tárolásra. Tegyük fel azonban, hogy egy új elem a 14-es résbe kerül, majd a következő a 12-esbe. Ez utóbbinak már egy 6 elemű klaszteren kell végigmennie, hogy megfelelő helyet találjon magának. A klaszterek összeolvadnak!

Négyzetes kipróbálás

A hasító függvény a négyzetes kipróbálás esetén

$$h(k,i) = \big(h_0(k) + c_1 \cdot i + c_2 \cdot i^2\big) \bmod m$$

alakú. Az i értékei itt is a $0, 1, 2, ..., m-1$. A c értékeket úgy kell megválasztani, hogy a kipróbálási sorozat kiadja az összes rést. A kipróbálási sorozat itt is csak a réstől függ, holott az $1, 2, ..., m-1$ számnak $(m-1)!$ permutációja van. A klaszterezés jelensége itt is fellép, de nem olyan súlyos, mint a lineáris esetben. Itt *másodlagos klaszterezésről* beszélünk, mivel a klaszter elemei nem egymás mellett, hanem a kipróbálási sorozat mentén helyezkednek el. Egy lehetőség például egy négyzetes kipróbálásra, ha a $c_1 \cdot i + c_2 \cdot i^2$ korrekció úgy alakul, hogy az $i = 0, 1, 2, ...$ értékekre a zérus, majd egy, azután három stb. értéket vesz fel, szabályát tekintve minden i értéknél az addigi i értékek összege lesz. Természetesen ez az eset nem minden táblaméret (m) esetén megfelelő. Ha azonban a táblaméret 2-nek hatványa, akkor valóban kiadja az összes rést.

Dupla hasítás

A dupla hasítás a

$$h(k,i) = (h_0(k) + i \cdot h_1(k)) \bmod m$$

hasító függvényt használja. Láthatóan minden réshez általában m különböző sorozatot ad, azaz a kipróbálási sorozatok száma m^2, ami arra ad reményt, hogy a módszer az előzőekhez képest jobb tulajdonságokkal bír. Mindenesetre a h_1 megválasztásakor arra kell ügyelni, hogy az mindig m-hez relatív prímet szolgáltasson. Ellenkező esetben a kipróbálási sorozat csak a tábla elemeinek a d-edrészét adná, ahol d az m és a szolgáltatott h_1 érték legnagyobb közös osztója. Ha m 2-hatványa és h_1 páratlan értékeket ad, az megfelel a feltételeknek. Ugyancsak megfelelő, ha m prímszám és h_1 mindig kisebb, mint m, de pozitív. Legyen m prím és legyen

$$h_0(k) = k \bmod m$$

$$h_1(k) = 1 + (k \bmod (m-1))$$

Válasszuk az $m = 701$-et. A kulcsok legyenek négyjegyű számok. Az 1000-es kulcsra $h_0(1000) = 1000 \bmod 701 = 299$, $h_1(1000) = 1000 \bmod 700 = 301$ és $h(k,i) = (299 + i \cdot 301) \bmod 701$, $i = 0, 1, 2, \ldots, 700$. A kipróbálási sorozat: 299, 600, 200, 501, 101, ... A 9999-es kulcsra $h_0(9999) = 9999 \bmod 701 = 185$, $h_1(9999) = 1 + (9999 \bmod 700) = 199$ és $h(k,i) = (185 + i \cdot 199) \bmod 701$, $i = 0, 1, 2, \ldots, 699$. A kipróbálási sorozat: 185, 384, 583, 81, 280, ...

5.3. tétel. A nyílt címzésű hasító tábla időigénye
Ha α a kitöltési arány, akkor a nyílt címzésű hasító táblában

$$C_n \leq \frac{1}{1-\alpha}$$

és

$$C_n' \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}.$$

5.1.4. Minimum és maximum keresése

Legyen most n kulcs elhelyezve egy tömbben, a kulcsok legyenek egy rendezett halmaz elemei (bármelyik kettő legyen összehasonlítható). Feladatunk a legkisebb és a legnagyobb kulcs megkeresése. A legkisebb kulcs megkeresése lineáris idejű és $n-1$ összehasonlítást igényel.

	4.1.4.1. algoritmus
	Minimumkeresés tömbben
	// $T(n) = \Theta(n)$
1	MINIMUMKERESÉS(A, min)
2	// Input paraméter: A – a tömb
3	// Output paraméter: min - a minimum értéke
4	//
5	min $\leftarrow A_1$
6	**FOR** $i \leftarrow 2$ **TO hossz**[A] **DO**
7	**IF** min $> A_i$
8	**THEN** min $\leftarrow A_i$
9	**RETURN** (min)

A legnagyobb elemet már $n-2$ összehasonlítással is megtaláljuk ezt követően. Összesen ez $2n-3$ összehasonlítást jelent. A két kulcs meghatározási ideje lineáris és az aszimptotika konstansa 2. Ezen a konstanson tudunk némiképpen javítani az alábbi módszer alkalmazásával.

Legyen az elemek száma páros. Először az első két elemet hasonlítjuk össze (ez 1 összehasonlítás), a kisebbet minimumként, a nagyobbat a maximumként tároljuk. Ezután már csak elempárokkal dolgozunk ($\frac{n-2}{2}$ van). Összehasonlítjuk az elempár elemeit egymással (mindegyik 1 összehasonlítás), majd a kisebbet a minimummal, a nagyobbat a maximummal (további 2 összehasonlítás). Ha az addigi minimumot, vagy maximumot változtatni kell, akkor

megtesszük. Összesen az összehasonlítások száma:

$$1 + 3 \cdot \frac{n-2}{2} = 1 + \frac{3}{2}n - 3 \cdot \frac{2}{2}$$

ami $\frac{3}{2}n - 2$ és ez kevesebb, mint $2n$-3

Ha páratlan számú elem van, akkor $\frac{n-3}{2}$ további elempár van az elsőt követően és marad még egy egyedüli elem a legvégén. Ezt az utolsó elemet mind az addigi minimummal, mind a maximummal össze kell hasonlítani legrosszabb esetben. Az összehasonlítások száma ennek megfelelően:

$$1 + 3 \cdot \frac{n-3}{2} + 2 = \frac{3}{2}n - 3 \cdot \frac{3}{2} + 3 = \frac{3}{2}n - \frac{3}{2} < 2n - 3$$

Az aszimptotika konstansa 2-ről 3/2-re csökkent.

5.1.5. Kiválasztás lineáris idő alatt

5.4. definíció. *A kiválasztási probléma*
Legyen adott egy A halmaz (n különböző szám), és egy i index $1 \leq i \leq n$. Meghatározandó az A halmaz azon x eleme, melyre nézve pontosan $i-1$ darab tőle kisebb elem van az A halmazban.

Speciális esetben ha $i = 1$, akkor a minimumkeresési problémát kapjuk. Mivel a minimumkeresési probléma $n-1$ összehasonlítással megoldható és ennyi kell is, ezért a probléma lineáris idő alatt megoldható. Ha növekvő sorrendbe rendezéssel próbáljuk megoldani a problémát, akkor mint később látni fogjuk $\mathrm{O}(n \cdot \log n)$ lépésben a probléma mindig megoldható. Nem szükséges azonban a rendezéshez folyamodni, mert a probléma rendezés nélkül is megoldható, ráadásul lineáris időben, amiről alább lesz szó.

5.5. definíció. *Medián*
Mediánnak nevezzük az adatsor azon elemét, amely a rendezett sorban a középső helyet foglalja el. Ha páratlan számú elem van az adatsorban, akkor $n = 2k - 1$ és így a medián indexe a rendezés után k. Ha páros számú elem van az adatsorban, akkor $n = 2k$, és ekkor két középső elem van a k és a $k+1$ indexű a rendezés után. (Alsó medián, felső medián.)

Ha nem említjük, akkor az alsó mediánról beszélünk.

5.6. példa. A 123, 234, 345, 444, 566, 777, 890 rendezett adatsorban a medián a 444, míg a 123, 234, 345, 444, 566, 777, 890, 975 sorban két medián van, a 444 (alsó medián) és az 566 (felső medián).

A lineáris idejű kiválasztási algoritmusnak szüksége lesz agy segédalgoritmusra, amely egy előre megadott számnak megfelelően az adatsort két részre osztja úgy, hogy az első részbe kerülnek azok az adatok, amelyek nem nagyobbak, a második részbe a nem kisebbek kerülnek.

	4.1.5.1. algoritmus
	Résztömb felosztása előre adott érték körül
	// $T(n) = \Theta(n)$
1	FELOSZT(A,p,r,x, q)
2	// Input paraméter: A – a tömb
3	// p - a felosztandó rész kezdőindexe
4	// r - a felosztandó rész végindexe
5	// x - az előre megadott érték, amely a felosztást szabályozza
6	// Output paraméter: A – a megváltozott tömb
7	// q – a felosztás határa $A_{p..q}, A_{q+1..r}$
8	//
9	$i \leftarrow p-1$
10	$j \leftarrow r+1$
11	**WHILE** IGAZ **DO**
12	**REPEAT** $j \leftarrow j-1$
13	**UNTIL** $A_j \leq x$
14	**REPEAT** $i \leftarrow i+1$
15	**UNTIL** $A_i \geq x$
16	**IF** $i < j$
17	**THEN** Csere $A_i \leftrightarrow A_j$
18	**ELSE** $q \leftarrow j$
19	**RETURN** (A, q)

Az előre adott x értéket az $A_{p..r}$ résztömb elemei közül jelöljük ki.

5.7. példa. Az algoritmus munkáját az alábbi tömbön szemléltetjük. Itt most $p = 1, r = 15, x = 5$.

	9	7	2	1	8	3	5	2	9	5	3	1	2	3	6			
i																j		kezdőállapot
	i													j				Csere
	3	7	2	1	8	3	5	2	9	5	3	1	2	9	6			
		i											j					Csere
	3	2	2	1	8	3	5	2	9	5	3	1	7	9	6			
					i							j						Csere
	3	2	2	1	1	3	5	2	9	5	3	8	7	9	6			
							i				j							Csere
	3	2	2	1	1	3	3	2	9	5	5	8	7	9	6			
									i	j								Csere
	3	2	2	1	1	3	3	2	5	9	5	8	7	9	6			
									j	i								eljárás vége

Az eljárás befejeztével a tömb $1, \ldots, 9$ indexű elemei nem nagyobbak, mint 5 a $10, \ldots, 15$ indexű elemek pedig nem kisebbek, mint 5.

	4.1.5.2. algoritmus
	Kiválasztás lineáris időben
	// $T(n) = \Theta(n)$
1	KIVÁLASZT (A, i, x)
2	// Input paraméter: A – a tömb
3	// i - a növekvő sorrendbe rendezés esetén a keresett elem indexe
4	// Output paraméter: x – a keresett elem
5	//
6	Ha $n = 1$, akkor x maga az A_1 elem. **RETURN** (x)
7	Ha $n \neq 1$, akkor osszuk fel a tömböt $n/5$ darab 5-elemű csoportra. (Esetleg a legutolsó csoportban lesz 5-nél kevesebb elem.)
8	Az összes $n/5$ csoportban megkeressük a mediánt.
9	A KIVÁLASZT algoritmus rekurzív alkalmazásával megkeressük az $n/5$ darab medián mediánját (medmed)
10	A FELOSZT algoritmus segítségével a **mediánok mediánja (medmed) körül** felosztjuk a bemeneti tömböt két részre. Legyen k elem az alsó és $n-k$ a felső részben.
11	A KIVÁLASZT algoritmus rekurziójával keressük az i-dik elemet a felosztás alsó részében, ha $i \leq k$, vagy pedig az $i-k$-adikat a felső részben egyébként.
12	*RETURN* (x)

5.8. tétel. A KIVÁLASZT algoritmus időigénye
A KIVÁLASZT algoritmus lineáris idejű.

Bizonyítás. $\lceil \frac{n}{5} \rceil$ csoport alakult ki. Mindegyikben meghatároztuk a mediánt. Ezen mediánok mediánját is meghatározzuk. Az adatok között a mediánok mediánjánál nagyobb elemek számát meg tudjuk becsülni az alábbi meggondolással. Mivel a medián középen lévő elem, így az a mediánok mediánja, amely $\lceil \frac{n}{5} \rceil$ medián közül kerül ki. Ezen mediánok fele biztosan nagyobb, mint a mediánok mediánja, azaz legalább $\lceil \frac{n}{5} \rceil \cdot \frac{1}{2} - 1$ ilyen elem van (saját magát nem számítjuk bele). Minden ilyen medián csoportjában akkor

legalább három elem nagyobb a medánok mediánjánál, kivéve az esetleg 5-nél kevesebb elemű utolsó csoportot, amit szintén elhagyunk. Ezek alapján legalább $3 \cdot \left(\left\lceil \frac{n}{5} \right\rceil \cdot \frac{1}{2} - 2 \right) \geq \frac{3}{10}n - 6$ elem biztosan nagyobb, mint a mediánok mediánja. (Ugyanennyi adódik a kisebb elemek számára is.)

Az 11-es sorban a KIVÁLASZT algoritmus a fentiek szerint a felosztás másik részében legfeljebb $n - \left(\frac{3}{10}n - 6\right) = \frac{7}{10}n + 6$ elemmel dolgozhat.

A KIVÁLASZT algoritmus egyes lépéseinek az időigénye:

Sor	Időigény
7.	$O(n)$
8.	$O(n)$
9.	$T(n/5)$
10.	$O(n)$
11.	$T(7n/10 + 6)$

Az időigényeket összegezve érvényes: a

$$T(n) \leq T\left(\left\lceil \frac{1}{5}n \right\rceil\right) + T\left(\frac{7}{10}n + 6\right) + \mathrm{O}(n)$$

összefüggés. Legyen itt $\mathrm{O}(n)$ konstansa a. Feltételezzük, hogy a megoldás $T(n) \leq c \cdot n$ egy bizonyos n küszöbtől kezdve, és behelyettesítéssel ezt fogjuk bizonyítani.

$$\begin{aligned} T(n) &\leq c \cdot \left\lceil \tfrac{1}{5}n \right\rceil + c \cdot \left(\tfrac{7}{10}n + 6\right) + a \cdot n \leq c \cdot \left(\tfrac{1}{5}n + 1\right) + c \cdot \left(\tfrac{7}{10}n + 6\right) + a \cdot n \\ &= c \cdot \left(\tfrac{9}{10}n + 7\right) + a \cdot n = c \cdot n - \left(c \cdot \tfrac{1}{10}n - 7c - a \cdot n\right) \end{aligned}$$

Válasszuk n-et úgy, hogy a zárójel nem negatív legyen. Ekkor $c \geq \frac{10a \cdot n}{n - 70}$.

Ha ezen felül $n \geq 140$, akkor a $c \geq 20a$ választás megfelelő a kiinduló feltételezésünk teljesüléséhez. □

5.2. Rendezés

5.9. definíció. *A reláció*
Valamely A halmaz esetén a $\varrho \subset A \times A$ részhalmazt az A halmazon értelmezett *relációnak* nevezzük. Azt mondjuk, hogy az A halmaz a és b eleme a ϱ relációban van, ha $(a, b) \in \varrho$. Röviden ezt így írjuk: $a \varrho b$.

5.10. példa. Legyen $A = \mathbb{R}$, és a reláció a kisebb „$<$” jel. Az $a \varrho b$ reláció azokat a számpárokat jelenti, amelyekre fennáll az $a < b$ összefüggés.

5.11. definíció. *A rendezési reláció*
A ϱ relációt *rendezési relációnak* nevezzük az A halmazon, ha

1. *reflexív*, azaz $a \varrho a$ teljesül minden $a \in A$ esetén;
2. $a \varrho b$ és $b \varrho a$ akkor és csak akkor áll fenn, ha $a = b$
3. *tranzitív*, azaz $a \varrho b$ és $b \varrho c$ maga után vonja az $a \varrho c$ teljesülését;
4. *antiszimmetrikus*, azaz vagy az $a \varrho b$, vagy a $b \varrho a$ fennáll minden $a, b \in A$ esetén.

5.12. példa. A valós számok közötti „$\leq$” reláció rendezési reláció.

Rendezésről olyan adattípus esetén beszélhetünk, amelyre értelmezve van egy rendezési reláció. Tekintsük a sorozatot és annak is vegyük a tömbös realizációját. A rendezés a sorozat elemeinek olyan felsorolását jelenti, amelyben az egymást követő elemek a megadott relációban vannak. A tárgyalást valós számokon (leginkább egészek) visszük végig és relációnak a kisebb, egyenlő relációt ($\leq$) tekintjük. ami nem csökkenti az általánosságot. A rendezés mind a mai időkig fontos informatikai probléma. Gyakran jelenik meg mint egy nagyobb probléma része. A rendezési algoritmusokkal kapcsolatban több szempont szerinti igény léphet föl. Ilyenek például az alábbiak:

a.	*Helyben rendezés*, azaz a rendezés eredménye az eredeti helyén jelenjen meg, legfeljebb konstans méretű többletmemória felhasználása révén.
b.	*Gyorsaság.* A rendezési idő legyen minél rövidebb.
c.	*Adaptivitás.* Az algoritmus használja ki a kulcsok között már meglévő rendezettséget.
d.	*Stabilitás.* A rendezés őrizze meg az azonos kulcsú rekordok esetén a rekordok egymáshoz képesti eredeti sorrendjét. (Például telefonszámlák készítésekor az azonos kulcsú előfizetői hívások időrendi sorrendje maradjon meg.)
e.	Az algoritmus csak a kulcsokat rendezze a rekordokra mutató pointerekkel, vagy az összes rekordot mozgassa.
f.	*Belső rendezés* legyen (csak a belső memóriát vegye igénybe a rendezéshez), vagy *külső rendezés* legyen (háttértárakat is igénybe vehet).
g.	*Összehasonlítás*on alapuljon a rendezés, vagy azt ne vegye igénybe az algoritmus. (Ez utóbbi esetben a kulcsokra további megszorításokat kell tenni.)
h.	Optimális legyen a rendezési algoritmus, vagy sem. (Nem biztos, hogy az adatok az optimális algoritmus által megkívánt módon vannak megadva.)
i.	Az összes rendezendő adatnak rendelkezésre kell-e állnia a rendezés teljes folyamata alatt, vagy sem.
j.	A rendezésnek csak a befejeztével van eredmény, vagy menet közben is a már rendezett rész tovább nem változik.

Nem lehet kizárni a nem optimális algoritmusokat sem az alkalmazásokból, mert egy probléma megoldásában nem csak a rendezési algoritmus optimalitása az egyetlen szempont a problémamegoldás hatékonyságára. (Hiába gyors az algoritmus, ha az adatok nem a kívánt formában állnak rendelkezésre, és a konverzió lerontja a hatékonyságot.)

5.2.1. A beszúró rendezés

A beszúró rendezés alapelve nagyon egyszerű. A sorozat második elemétől kezdve (az első önmagában már rendezett) egyenként a kulcsokat a sorozat

eleje felé haladva a megfelelő helyre mozgatjuk összehasonlítások révén. A sorozatnak a vizsgált kulcsot megelőző elemei mindig rendezettek az algoritmus során.

5.13. példa. Az alábbi kulcsok esetén nyíl mutatja a mozgatandó kulcsot és a beszúrás helyét.

	8		4		2		3		1		6		5		9		7
↑			↑														
	4		8		2		3		1		6		5		9		7
↑					↑												
	2		4		8		3		1		6		5		9		7
		↑					↑										
	2		3		4		8		1		6		5		9		7
↑									↑								
	1		2		3		4		8		6		5		9		7
								↑			↑						
	1		2		3		4		6		8		5		9		7
								↑					↑				
	1		2		3		4		5		6		8		9		7
															↑		
	1		2		3		4		5		6		8		9		7
												↑					↑
	1		2		3		4		5		6		7		8		9

	4.2.1.1. algoritmus
	Beszúró rendezés
	// $T(n) = \Theta(n^2)$
1	BESZÚRÓ_RENDEZÉS (A)
2	// Input paraméter: A - a rendezendő tömb
3	// Output paraméter: A - a rendezett tömb
4	//
5	**FOR** $j \leftarrow 2$ **TO** ***hossz***$[A]$ **DO**
6	$kulcs \leftarrow A_j$
7	// Beszúrás az $A_{1\ldots j-1}$ rendezett sorozatba
8	$i \leftarrow j-1$
9	**WHILE** $i > 0$ és $A_i > kulcs$ **DO**
10	$A_{i+1} \leftarrow A_i$
11	DEC(i)
12	$A_{i+1} \leftarrow kulcs$
13	**RETURN** (A)

Feladat: Számítsuk ki, hogy a beszúró rendezésre a $T(n) = \Theta(n^2)$!

Feladat: Készítsük el a beszúró rendezésre a pszeudokódot, ha a kulcsok egy kétszeresen láncolt listával vannak megadva!

5.2.2. Az összefésülő rendezés

Az összefésülő rendezés alapelve az összefésülés műveletén alapszik, amely két *rendezett* tömbből egy új rendezett tömböt állít elő. Az összefésülés folyamata: Mindkét tömbnek megvizsgáljuk az első elemét. A két elem közül a kisebbiket beírjuk az eredménytömb első szabad eleme helyére. A felszabaduló helyre újabb elemet veszünk abból a tömbből, ahonnan előzőleg a kisebbik elem jött. Ezt a tevékenységet folytatjuk mindaddig, míg valamelyik kiinduló tömbünk ki nem ürül. Ezután a még vizsgálat alatt lévő elemet,

valamint a megmaradt másik tömb további elemeit sorba az eredménytömbhöz hozzáírjuk a végén. Az eredménytömb nem lehet azonos egyik bemeneti tömbbel sem, vagyis az eljárás nem helyben végzi az összefésülést.

5.14. példa. Példa összefésülésre
Legyen ÖSSZEFÉSÜL $(A,\ p,\ q,\ r)$ az az eljárás, amely összefésüli az $A_{p..q}$ és az $A_{q+1..r}$ résztömböket, majd az eredményt az eredeti $A_{p..r}$ helyre másolja vissza. Az eljárás lineáris méretű további segédmemóriát igényel. Az összefésülés időigénye $\Theta(n)$, ha összesen n elemünk van. (Egy menetben elvégezhető és az kell is hozzá.)

5.15. definíció. *Az oszd meg és uralkodj elv*
Az oszd meg és uralkodj elv egy algoritmus tervezési stratégia A problémát olyan *kisebb méretű, azonos részproblémákra* osztjuk föl, amelyek *rekurzívan* megoldhatók. Ezután *egyesítjük* a megoldásokat.

Az összefésülő rendezés oszd meg és uralkodj típusú algoritmus, melynek az egyes fázisai:

Felosztás:	A tömböt két $\frac{n}{2}$ elemű részre osztjuk
Uralkodás:	Rekurzív összefésüléses módon mindkettőt rendezzük . (Az 1 elemű már rendezett)
Egyesítés:	A két részsorozatot összefésüljük.

	4.2.2.1. algoritmus
	Összefésülő rendezés (Merge Sort)
	// $T(n) = \Theta(n \cdot \log n)$
1	ÖSSZEFÉSÜLŐ_RENDEZÉS (*A, p, r*)
2	// Input paraméter: *A* - a tömb, melynek egy részét rendezeni kell
3	// *p* - a rendezendő rész kezdőindexe
4	// *r* - a rendezendő rész végindexe
5	// Output paraméter: *A* - a rendezett résszel rendelkező tömb
6	//
7	**IF** $p < r$
8	**THEN** $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
9	ÖSSZEFÉSÜLŐ_RENDEZÉS (*A, p, q*)
10	ÖSSZEFÉSÜLŐ_RENDEZÉS (*A, q* + 1, *r*)
11	ÖSSZEFÉSÜL (*A, p, q, r*)
12	**RETURN** (A)

A teljes tömb rendezését megoldó utasítás: ÖSSZEFÉSÜLŐ_RENDEZÉS(*A,1,hossz[A]*).

5.16. példa. Példa összefésülésre

Az összefésülő rendezés időigénye

Felosztás: $\Theta(1)$

Uralkodás: $2 \cdot T\left(\frac{n}{2}\right) \Rightarrow T(n) = \begin{cases} \Theta(1), & ha \quad n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n), & ha \quad n > 1 \end{cases}$

Egyesítés: $\Theta(n)$

Az algoritmus időigénye megkapható a mester tétel 2. pontja alapján: $T(n) = \Theta(n \cdot \log n)$.

5.2.3. A Batcher-féle páros-páratlan összefésülés

Az eljárás csak az összefésülést teszi hatékonyabbá. Nem önálló rendező módszer. Nagy előnye, hogy párhuzamosíthatók a lépései. Legyen két rendezett sorozatunk, az n elemű A sorozat és az m elemű B sorozat.

$$A = \{a_1, \ldots, a_n\}$$
$$B = \{b_1, \ldots, b_m\}$$

A két sorozat összefésülése adja a $C = \{c_1, \ldots, c_{n+m}\}$ sorozatot. Az összefésülés módja a következő: Mindkét kiinduló sorozatból kettőt képezünk, a páratlan indexű és a páros indexű elemek sorozatait:

$$A_1 = \{a_1, a_3, a_5, \ldots\}; B_1 = \{b_1, b_3, b_5, \ldots\}$$
$$A_2 = \{a_2, a_4, a_6, \ldots\}; B_2 = \{b_2, b_4, b_6, \ldots\}$$

Összefésüljük az A_1, B_2 sorozatokat, eredménye az U sorozat.

Összefésüljük az A_2, B_1 sorozatokat, eredménye a V sorozat.

Összefésüljük az U és V sorozatokat, eredmény a C sorozat.

5.17. tétel. A Batcher-féle összefésülés tétele
A Batcher összefésülés során

$$c_{2i-1} = \min\{u_i, v_i\}$$

és

$$c_{2i} = \max\{u_i, v_i\},$$

ahol $1 \leq i \leq \dfrac{n+m}{2}$.

Bizonyítás. Fogadjuk el kiindulásként igaznak azt a feltevést, hogy C elejéből páros számú elemet véve azok között azonos számú U és V elem van. Ekkor

$$\{c_1, \ldots, c_{2(i-1)}\} = \{u_1, \ldots, u_{i-1}\} \cup \{v_1, \ldots, v_{i-1}\}$$

és

$$\{c_1, \ldots, c_{2i}\} = \{u_1, \ldots, u_i\} \cup \{v_1, \ldots, v_i\}$$

Ebből viszont

$$\{c_{2i-1}, c_{2i}\} = \{u_1, v_i\},$$

ahonnan $c_{2i-1} < c_{2i}$ miatt adódik a tétel állítása.

A feltételezésünk bizonyítása:
Legyen

$$\{c_1, \ldots, c_{2k}\} = \{a_1, \ldots, a_k\} \cup \{b_1, \ldots, b_{2k-s}\}.$$

Ezek közül U-ba kerül $\left\lceil \frac{s}{2} \right\rceil$ elem az A-ból (az A páratlan indexű elemei) és $\left\lfloor \frac{2k-s}{2} \right\rfloor$ elem a B-ből (a B páros indexű elemei), valamint V-be kerül $\left\lfloor \frac{s}{2} \right\rfloor$ elem az A-ból (az A páros indexű elemei) és $\left\lceil \frac{2k-s}{2} \right\rceil$ elem a B-ből (a B páratlan indexű elemei). Innen az U-beliek száma

$$\left\lceil \frac{s}{2} \right\rceil + \left\lfloor \frac{2k-s}{2} \right\rfloor = k$$

és a V-beliek száma

$$\left\lfloor \frac{s}{2} \right\rfloor + \left\lceil \frac{2k-s}{2} \right\rceil = k$$

□

5.2.4. Gyorsrendezés (oszd meg és uralkodj típusú algoritmus)

Felosztás:	Az $A_{p..r}$ tömböt két nemüres $A_{p..q}$ és $A_{q+1..r}$ részre osztjuk úgy, hogy $A_{p..q}$ minden eleme kisebb egyenlő legyen, mint $A_{q+1..r}$ bármely eleme. (A megfelelő q meghatározandó.)
Uralkodás:	Az $A_{p..q}$ és $A_{q+1..r}$ résztömböket rekurzív gyorsrendezéssel rendezzük.
Egyesítés:	Nincs rá szükség, mivel a tömb már rendezett. (A saját helyén rendeztük.)

	4.2.4.1. algoritmus
	Gyorsrendezés (Quick Sort)
	// $T(n) = \Theta(n^2)$, átlagos:$T(n) = \Theta(n \cdot \log n)$
1	GYORSRENDEZÉS(*A,p,r*)
2	// Input paraméter: A – a tömb, melynek egy részét rendezeni kell
3	// p - a rendezendő rész kezdőindexe
4	// r - a rendezendő rész végindexe
5	// Output paraméter: A - a rendezett résszel rendelkező tömb
6	//
7	**IF** $p < r$
8	**THEN** FELOSZT (*A, p, r,* A_p*, q*) //Lásd 4.1.5.1 algoritmus
9	GYORSRENDEZÉS (*A, p, q*)
10	GYORSRENDEZÉS (*A, q+1, r*)
11	**RETURN** (A)

A gyorsrendezés időigénye:

A legrosszabb eset: a felosztás minden lépésben $n-1, 1$ elemű

$$T(1) = \Theta(1),$$
$$T(n) = T(n-1) + \Theta(n).$$

Innen

$$T(n) = T(n-1) + \Theta(n) = T(n-2) + \Theta(n-1) + \Theta(n) = \dots$$
$$= T(1) + \Theta(1) + \Theta(2) + \dots + \Theta(n) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

A legjobb eset: $\frac{n}{2}$, $\frac{n}{2}$ a felosztás, ekkor

$$T(1) = \Theta(1),$$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n), \text{ ha } n > 1$$

Ez megegyezik az összefésülő módszer formulájával, tehát

$$T(n) = \Theta(n \cdot \log n).$$

Megjegyzés: ha a felosztás aránya állandó pl. $\frac{9}{10}, \frac{1}{10}$, akkor a rekurziós formula:

$$T(n) = T\left(\frac{9}{10}\right) + T\left(\frac{1}{10}\right) + \Theta(n).$$

Bizonyíthatö, hogy ekkor is

$$T(n) = \Theta(n \cdot \log n).$$

Ezen túlmenően az átlagos értékre is $T(n) = \Theta(n \cdot \log n)$ adódik.

5.2.5. A buborékrendezés

A buborékrendezésnél az egymás mellett álló elemeket hasonlítjuk össze, és szükség esetén sorrendjüket felcseréljük. Ezt mindaddig folytatjuk, míg szükség van cserére.

	4.2.5.1. algoritmus
	Buborékrendezés (Bubble Sort)
	// $T(n) = \Theta(n^2)$
1	BUBORÉKRENDEZÉS(A)
2	// Input paraméter: A - a rendezendő tömb
3	// Output paraméter: A - a rendezett tömb
4	//
5	$j \leftarrow 2$
6	**REPEAT** *nemvoltcsere* $\leftarrow$ igaz
7	**FOR** $i \leftarrow$ **hossz**$[A]$ **DOWNTO** j **DO**
8	**IF** $A_i < A_{i-1}$
9	**THEN** csere $A_i \leftrightarrow A_{i-1}$
10	*nemvoltcsere* $\leftarrow$ hamis
11	INC(j)
12	**UNTIL** *Nemvoltcsere*
13	**RETURN** (A)

Időigény a legrosszabb esetben: $T(n) = \dfrac{n \cdot (n-1)}{2}$.

Az algoritmusra jellező a sok csere, az elem lassan kerül a helyére.

5.2.6. A Shell rendezés (rendezés fogyó növekménnyel)

A Shell rendezés a buborékrendezésnél tapasztalt lassú helyrekerülést igyekszik felgyorsítani azáltal, hogy egymástól távol álló elemeket hasonlít és cserél fel. A távolságot (itt *növekménynek* nevezik) fokozatosan csökkenti, míg az 1 nem lesz. Minden növekmény esetén beszúrásos rendezést végez az adott növekménynek megfelelő távolságra álló elemekre. Mire a növekmény 1 lesz, sok elem már majdnem a helyére kerül.

A növekmények felépítése. Használjunk t számú növekményt. Legyenek ezek $h_{1...t}$.

A követelmény: $h_t = 1$, és $h_{i+1} < h_i \qquad i = 1, \ldots t-1$.
A szakirodalomban javasolt növekményadatok: $t = \lceil \log n \rceil - 1$

$h_{i-1} = 2h_i$	$\ldots, 32, 16, 8, 4, 2, 1$
$h_{i-1} = 3h_i + 1$	$\ldots 121, 40, 13, 4, 1$
$h_{i-1} = 2h_i - 1$	$\ldots 31, 15, 7, 3, 1$

	4.2.6.1. algoritmus
	Shell rendezés
	// $T(n) = \Theta(n^{1,5})$
1	SHELL_RENDEZÉS(A)
2	// Input paraméter: A - a rendezendő tömb
3	// Output paraméter: A - a rendezett tömb
4	//
5	**FOR** $s \leftarrow 1$ **TO** t **DO**
6	$m \leftarrow h_s$
8	**FOR** $j \leftarrow m+1$ **TO** hossz[A] **DO**
9	$i \leftarrow j$ - m
10	$k \leftarrow$ ***kulcs***$[A_j]$
	$r \leftarrow A_j$
15	**WHILE** $i > 0$ és $k < A_j$ **DO**
16	$A_{i+m} \leftarrow A_i$
17	$i \leftarrow i - m$
18	$A_{i+m} \leftarrow r$
19	**RETURN** (A)

Megjegyzés: A hossz[A] a rendezendő elemek számát jelöli.

Időigény: alkalmas növekmény választással leszorítható $T(n) = \Theta(n^{1,5})$-re.

5.2.7. A minimum kiválasztásos rendezés

Ebben a rendezésben **Hossz**$[A] - 1$-szer végigmegyünk a tömbön. Minden alkalommal eggyel magasabb indexű elemtől indulunk. Megkeressük a minimális elemet, és azt az aktuális menet kezdő elemével felcseréljük.

	4.2.7.1. algoritmus
	Minimum kiválasztásos rendezés
	// $T(n) = \Theta(n^2)$
1	MINIMUM_KIVÁLASZTÁSOS_RENDEZÉS(A)
2	// Input paraméter: A - a rendezendő tömb
3	// Output paraméter: A - a rendezett tömb
4	//
5	**FOR** $i \leftarrow 1$ **TO** *hossz[A]*-1 **DO**
6	// minimumkeresés
7	$k \leftarrow i$
8	$x \leftarrow A_i$
9	**FOR** $j \leftarrow i+1$ **TO** *hossz[A]* **DO**
10	**IF** $A_j < x$
11	**THEN** $k \leftarrow j$
12	$x \leftarrow A_j$
13	// az i. elem és a minimum felcserélése
14	$A_k \leftarrow A_i$
15	$A_i \leftarrow x$
16	**RETURN** (A)

Időigény: összehasonlításszám $T(n) = \Theta(n^2)$.

5.2.8. Négyzetes rendezés

Felosztjuk az n elemű A tömböt (közel) $\sqrt{n}$ számú részre (alcsoportra). Mindegyikben (közel) $\sqrt{n}$ elemet helyezünk el, majd mindegyikből kiemeljük (eltávolítjuk) a legkisebbet. A kiemeltekből egy főcsoportot képzünk. Kiválasztjuk a főcsoport legkisebb elemét és azt az eredménytömbbe írjuk, a főcsoportból pedig eltávolítjuk (töröljük). Helyére abból az alcsoportból ahonnan ő származott újabb legkisebbiket emelünk be a főcsoportba. Az eljárást folytatjuk, míg az elemek el nem fogynak a főcsoportból.

Időigény: összehasonlításszám $T(n) = \Theta(n \cdot \sqrt{n}) = \Theta(n^{1,5})$.

Továbbfejlesztett változat, amikor $\sqrt[3]{n}$ számú elem van egy fő-főcsoportban és $\sqrt[3]{n}$ számú főcsoport van, mindegyikben $\sqrt[3]{n}$ számú elemmel, melyek mindegyikéhez egy $\sqrt[3]{n}$ elemszámú alcsoport tartozik. A rendezés elve az előző algoritmuséhoz hasonló.

Időigény: $T(n) = \Theta(n \cdot \sqrt[3]{n}) = \Theta\left(n^{\frac{4}{3}}\right)$

A Stirling formula és az Alsó korlát összehasonlító rendezésre tétel

5.18. tétel. A Stirling formula
Igaz az alábbi összefüggés az n!-ra:

$$\frac{n^n}{e^n} < n! < \frac{(n+1)^{n+1}}{e^n}, \qquad n = 3, 4, 5, \ldots$$

Bizonyítás. Az egyenlőtlenséget a logaritmusra látjuk be. A logaritmus

függvény konkáv és emiatt írható:

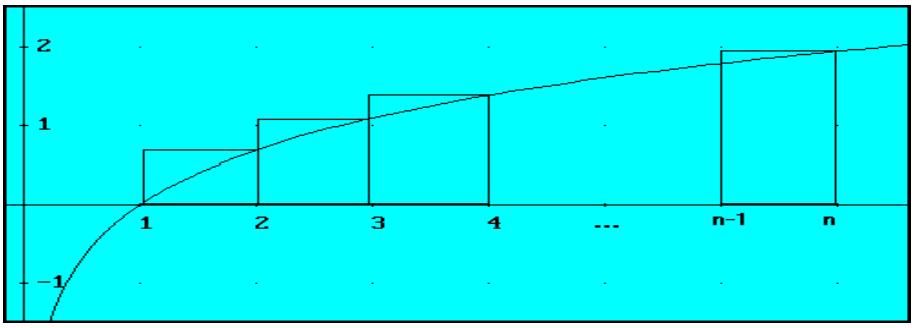
$$\ln(n!) = 1 \cdot \ln 2 + 1 \cdot \ln 3 + 1 \cdot \ln 4 + \ldots + 1 \cdot \ln n$$

$$\ln(n!) > \int_1^n \ln x dx = \int_1^n 1 \cdot \ln x dx = [x \cdot \ln x]_1^n - \int_1^n x \cdot \frac{1}{x} dx$$
$$= n \ln n - [x]_1^n = n \ln n - (n-1) = n \ln n - n + 1 > n \ln n - n$$

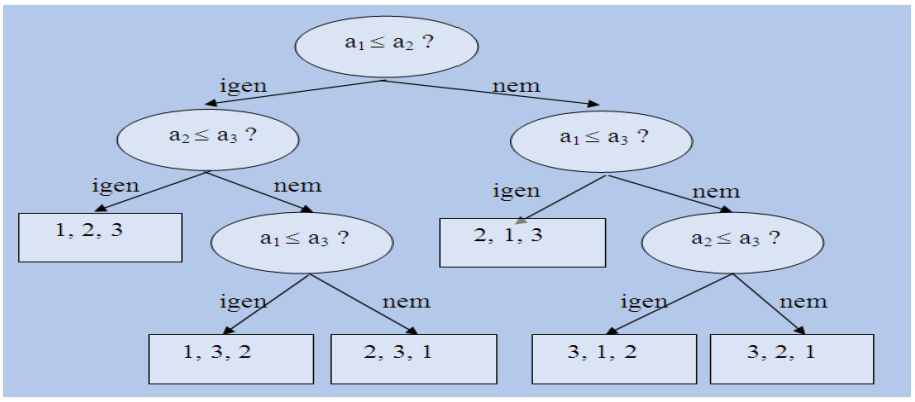
$$\ln(n!) = 1 \cdot \ln 2 + 1 \cdot \ln 3 + 1 \cdot \ln 4 + \ldots + 1 \cdot \ln n$$

$$\ln(n!) < \int_1^{n+1} \ln x dx = \int_1^n 1 \cdot \ln x dx = [x \cdot \ln\ x]_1^{n+1} - [x]_1^{n+1}$$
$$= (n+1)\ln(n+1) - (n+1-1) = (n+1)\ln(n+1) - n$$

□



Az összehasonlító módszerek döntési fája



5.19. tétel. Alsó korlát összehasonlító rendezésre
Bármely n elemet rendező döntési fa magassága $T(n) = \Omega(n \cdot \log n)$

Bizonyítás. Egy h magasságú döntési fa leveleinek száma legfeljebb $2h$. Mivel minden permutációt rendezni kell tudnia az algoritmusnak, és összesen $n!$ permutáció lehetséges, ezért a döntései fának legalább $n!$ levele kell legyen.

Tehát $n! \leq 2^h$fennáll. Logaritmálva: $h \geq \log(n!)$. A Stirling formula szerint $n! > \left(\frac{n}{e}\right)^n$. Behelyettesítve: $h \geq \log\left(\left(\frac{n}{e}\right)^n\right) = n \log n - n \log e$.

Tehát: $h = \Omega(n \cdot \log n)$ □

5.2.9. Lineáris idejű rendezők: A leszámláló rendezés

A lineáris idejű rendezők nem használják az összehasonlítást.

A leszámláló rendezés (= binsort, ládarendezés) bemenete 1 és k közötti egész szám.

Időigény: $T(n) = \Theta(n + k)$. Ha $k = \Theta(n)$, akkor a rendezési idő is $T(n) = \Theta(n)$, ahol $n = hossz[A]$.

Az elemeket az $A_{[1..n]}$ tömbben helyezzük el. Szükség van további két tömbre: $B_{[1..n]}$ az eredményt tárolja majd, $C_{[1..k]}$ segédtömb.

A rendezés lényege, hogy A minden elemére meghatározza a nála kisebb elemek számát. Ez alapján tudja az elemet a kimeneti tömb megfelelő helyére tenni. Stabil eljárás: az azonos értékűek sorrendje megegyezik az eredetivel

	4.2.10.1. algoritmus
	Leszámláló rendezés
	// $T(n) = \Theta(n)$
1	LESZÁMLÁLÓ_RENDEZÉS (A, k, B)
2	// Input paraméter: A - a rendezendő tömb
3	// k – kulcs felső korlát, pozitív egész
4	// Output paraméter: B - a rendezett tömb
5	//
6	**FOR** $i \leftarrow 1$ **TO** k **DO**
7	$C_i \leftarrow 0$
8	**FOR** $j \leftarrow 1$ **TO** hossz[A] **DO**
9	**INC**(C_{A_j})
10	// C_i azt mutatja, hogy hány i értékű számunk van
11	**FOR** $i \leftarrow 2$ **TO** k **DO**
12	$C_i \leftarrow C_i + C_{i-1}$
13	// C_i most azt mutatja, hogy hány i-től nem nagyobb számunk van
14	**FOR** $j \leftarrow hossz[A]$ **DOWNTO** 1 **DO**
15	$B_{C_{A_j}} \leftarrow A_j$
16	**DEC**(C_{A_j})
17	**RETURN** (B)

5.2.10. A számjegyes rendezés (radix rendezés)

Azonos hosszúságú szavak, stringek rendezésére használhatjuk. (Dátumok, számjegyekből álló számok, kártyák, stb.) Legyen d a szó hossza, k pedig az egy karakteren, mezőben előforduló lehetséges jegyek, jelek száma, n pedig az adatok száma.

Időigény: $T(n) = \Theta(d \cdot (n + k))$

	4.2.11.1. algoritmus
	Számjegyes rendezés
	// $T(n) = \Theta(d \cdot (n+k))$
1	SZÁMJEGYES_RENDEZÉS (A)
2	// Input paraméter: A - a rendezendő tömb
4	// Output paraméter: A - a rendezett tömb
4	//
5	**FOR** $i \leftarrow d$ **DOWNTO** 1 **DO**
6	Stabil módszerrel rendezzük az A tömböt az i. számjegyre
7	**RETURN** (A)

5.2.11. Edényrendezés

Feltételezzük, hogy a bemenet a $[0, 1)$ intervallumon egyenletes eloszlású számok sorozata. Felosztjuk a $[0, 1)$ intervallumot n egyenlő részre (edények). A bemenetet szétosztjuk az edények között, minden edényben egy listát kezelve. Az azonos edénybe esőket beszúrásos módon rendezzük. A végén a listákat egybefűzzük az elsővel kezdve.

Várható időigény: $T(n) = \Theta(n)$

	4.2.12.1. algoritmus
	Edényrendezés
	// $T(n) = \Theta(n)$
1	EDÉNYRENDEZÉS (A, L)
2	// Input paraméter: A - a rendezendő tömb, n elemű
3	// Output paraméter: L - a rendezett elemek listája
4	// Menet közben szükség van egy n elemű B tömbre, mely listafejeket tárol. Indexelése 0-val indul.
5	**n ←hossz[A]**
6	**FOR** $i \leftarrow 1$ **TO** n **DO**
7	Beszúrjuk az A_i elemet a $B_{\lfloor n \cdot A_i \rfloor}$ listába
8	**FOR** $i \leftarrow 0$ **TO** $n-1$ **DO**
9	Rendezzük a B_i listát beszúrásos rendezéssel
10	Sorban összefűzzük a $B_0, B_1, \ldots, B_{n-1}$ listákat. képezve az L listát
11	**RETURN** (L)

5.2.12. Külső tárak rendezése

Külső tárak rendezésénél az elérési és a mozgatási idő szerepe drasztikusan megnő. Az összefésüléses módszerek jönnek elsősorban számításba.

5.20. definíció. *A k hosszúságú futam file-ban*
Egy file k szomszédos rekordjából álló részét **k hosszúságú futam**nak nevezzük, ha benne a rekordkulcsok rendezettek (pl.: növekedő sorrendűek).

Először alkalmas k-val ($k = 1$ mindig megfelel) a rendezendő file-t két másik file-ba átmásoljuk úgy, hogy ott k hosszúságú futamok jöjjenek létre. Ezután a két file-t összefésüljük egy-egy elemet véve mindkét file-ból. Az

eredményfile-ban már $2k$ lesz a futamhossz.(esetleg a legutolsó rövidebb lehet). Ezt ismételgetjük a teljes rendezettségig mindig duplázva k értékét. Legyen a rekordok száma n. Egy menetben n rekordmozgás van a szétdobásnál és n az összefésülésnél. A menetek száma legfeljebb $\lceil \log n \rceil$.

Az időigény: $T(n) = \Theta(n \cdot \log n)$.

Külső tárak rendezésének gyorsítása

Az $n \log n$ nem javítható az összehasonlítások miatt. A szorzó konstansokat lehet csökkenteni.

Változó futamhosszak:	a file-ban meglévő természetes módon kialakult futamhosszakat vesszük figyelembe. A futamhatárok figyelésének adminisztrálása bejön, mint további költség.
Több részfile használata	esetén a szétdobás nem két, hanem több részre történik.Külön adminisztráció összefésülésnél a kiürült file-ok figyelése.
Polifázisú összefésülés	alkalmazásakor nem folytatjuk végig minden menetben az összefésüléseket, hanem a célfile szerepét mindig a kiürült file veszi át és ide kezdjük összefésülni a többit.

Egy eset polifázisú összefésülésre, amikor kínosan lassú a módszer. A tábla belsejében a file-ok futamszáma szerepel, zárójelben a futamok mérete. Kezdetben az első file 1 rekordból áll és egy a futamhossz, a második file 5 rekordból áll és szintén egy a futamhossz.

6. fejezet

Fák

6.1. Gráfelméleti fogalmak, jelölések

6.1. definíció. Legyen V egy véges halmaz, E pedig V-beli rendezetlen elempárok véges rendszere. Ekkor a $G = (V, E)$ párt **gráfnak** nevezzük.

Vezessük be a következő jelöléseket egy $G = (V, E)$ gráf esetében:

- $n = |V|$ (csúcsok száma)
- $e = |E|$ (élek elemszáma)

Egy gráf nagyon sok probléma szemléltetésére szolgálhat, a legegyszerűbb például az úthálózat, telefonhálózat, de akár házassági problémát is ábrázolhatunk vele.
A gráfokat többféle szempontból is szokás csoportosítani. A legjelentősebb szempont az irányítottság.

6.2. definíció. A $G = (V, E)$ rendezett párt **irányított gráfnak (digráfnak)** nevezzük. A rendezett pár elemeire tett kikötések:

- V véges halmaz, a G-beli csúcsok halmaza.
- E bináris reláció a V halmazon, az élek halmaza.
- E=$\{(u, v)$ rendezett pár $\mid u \in V, v \in V\} \subset V \times V$. Hurkok megengedettek.

Hurok az (a, a) él.

6.3. definíció. A $G = (V, E)$ rendezett párt **irányítatlan gráfnak** nevezzük. A rendezett pár elemeire tett kikötések:

- V véges halmaz, a G-beli csúcsok halmaza.
- E bináris reláció a V halmazon, az élek halmaza.
- E=$\{(u, v)$ rendezett pár $\mid u \in V, v \in V\} \subset V \times V$. Hurok nem megengedett.

Mint ahogy már fentebb utaltunk rá, a csúcsok közötti kapcsolat sokszor jelentheti út létezését vagy kommunikáció lehetőséget. Ilyenkor gyakran költségek vagy súlyok tartoznak az élekhez, amelyek az út esetében időt vagy akar pénzt is jelenthetnek (gondoljunk csak az autópályákra, amelyek használatáért fizetni kell).

6.4. definíció. Az a gráf (irányított vagy irányítatlan), amelynek minden éléhez egy számot (súlyt) rendelünk hozzá, **hálózatnak (súlyozott gráfnak)** nevezzük.

6.5. megjegyzés. A súlyt rendszerint egy súlyfüggvény segítségével adunk meg: $w : E \to \mathbb{R}$, egy (u, v) él súlya $w(u, v)$.

Az ilyen gráfok sok helyen előfordulnak, például optimalizálási feladatokban, mint az utazó ügynök probléma. Az élhez rendelt érték lehet az él költsége, súlya vagy hossza az alkalmazástól függően.

6.6. definíció. A gráf egymáshoz csatlakozó éleinek olyan sorozatát, amely egyetlen ponton sem megy át egynél többször, **útnak** nevezzük.

6.7. definíció. Legyen adott egy $G = (V, E)$ irányított vagy irányítatlan gráf a $k(f)$,
$f \in E$ élsúlyokkal. A G gráf egy u-ból v-be menő **útjának hossza** az úton szereplő élek súlyának összege.

6.1.1. Gráfok ábrázolási módjai

Két módszert szokás használni egy $G = (V, E)$ gráf ábrázolására: az egyikben szomszédsági listákkal, a másikban szomszédsági mátrixszal adjuk meg a gráfot.

Rendszerint a szomszédsági listákon alapuló ábrázolást választják, mert ezzel ***ritka*** gráfok tömören ábrázolhatók.

6.8. definíció. Egy gráfot **ritkának** nevezünk, ha $|E|$ sokkal kisebb, mint $|V|^2$.

Ugyanakkor a csúcsmátrixos ábrázolás előnyösebb lehet ***sűrű*** gráfok esetén, vagy ha gyorsan kell eldönteni, hogy két csúcsot összeköt-e él.

6.9. definíció. Egy gráfot **sűrűnek** nevezünk, ha $|E|$ megközelíti $|V|^2$-et.

Szomszédsági listás

$G = (V, E)$ ***szomszédsági listás ábrázolása*** során egy Adj tömböt használunk. Ez |V| darab listából áll, és az Adj tömbben minden csúcshoz egy lista tartozik. Minden $u \in V$ csúcs esetén az $Adj[u]$ szomszédsági lista tartalmazza az összes olyan v csúcsot, amelyre létezik $(u, v) \in E$ él. Azaz: $Adj[u]$ elemei az u csúcs G-beli szomszédjai. (Sokszor nem csúcsokat, hanem megfelelő mutatókat tartalmaz a lista.) A szomszédsági listákban a csúcsok sorrendje rendszerint tetszőleges.

Ha G irányított gráf, akkor a szomszédsági listák hosszainak összege $|E|$, hiszen egy (u, v) élt úgy ábrázolunk, hogy v-t felvesszük az $Adj[u]$ listába. Ha G irányítatlan gráf, akkor az összeg $2|E|$, mert (u, v) irányítatlan él ábrázolása során u-t betesszük v szomszédsági listájába, és fordítva. Akár irányított, akár irányítatlan a gráf, a szomszédsági listás ábrázolás azzal a kedvező tulajdonsággal rendelkezik, hogy az ábrázoláshoz szükséges tárterület $\theta(V + E)$.

A szomszédsági listákat könnyen módosíthatjuk úgy, hogy azokkal súlyozott gráfokat ábrázolhassunk. Például, legyen $G = (V, E)$ súlyozott gráf w súlyfüggvénnyel. Ekkor az $(u, v) \in E$ él $w(u, v)$ súlyát egyszerűen a v csúcs

mellett tároljuk u szomszédsági listájában. A szomszédsági listás ábrázolás könnyen alkalmassá tehető sok gráfváltozat reprezentálására.

A szomszédsági listás ábrázolás hátránya, hogy nehéz eldönteni szerepel-e egy (u, v) él a gráfban, hiszen ehhez az $Adj[u]$ szomszédsági listában kell v-t keresni. Ez a hátrány kiküszöbölhető csúcsmátrix használatval, ez azonban aszimptotikusan növeli a szükséges tárterület méretét.

Szomszédsági mátrixos

Ha egy $G = (V, E)$ gráfot ***szomszédsági mátrixszal*** (vagy más néven ***csúcsmátrixszal***) ábrázolunk, feltesszük, hogy a csúcsokat tetszőleges módon megszámozzuk az 1,2, ...,$|V|$ értékekkel. A G ábrázolásához használt $A = (a_{ij})$ csúcsmátrix mérete $|V| \times |V|$, és

$$a_{ij} = \begin{cases} 1, & \text{ha } (i, j) \in E, \\ 0, & \text{különben} \end{cases}$$

A csúcsmátrix $\theta(V^2)$ tárterültetet foglal el, függetlenül a gráf éleinek számától. Gyakran kifizetődő a csúcsmátrixból csak a főátlóban és az efölött szereplő elemeket tárolni, ezzel majdnem felére csökkenthetjük az ábrázoláshoz szükséges tárterület méretét.

A szomszédsági listás ábrázoláshoz hasonlóan csúcsmátrixokkal is reprezentálhatunk súlyozott gráfokat. Ha $G = (V, E)$ súlyozott gráf w súlyfüggvénnyel, akkor az $(u, v) \in E$ él $w(u, v)$ súlyát a csúcsmátrix u sorában és v oszlopában tároljuk. Nem létező él esetén a mátrix megfelelő elemét NIL-nek választjuk, noha sokszor célszerű ehelyett 0 vagy végtelen értéket használni.

A szomszédsági listák együttesen aszimptotikusan kevesebb tárterületet igényelnek, mint a csúcsmátrix, azonban a használat során hatékonyságban ugyanennyivel elmaradnak attól, így ha a gráf mérete nem túl nagy, akkor kedvezőbb a hatékonyabb és egyszerűbb csúcsmátrixos ábrázolást használni. Ha a gráf nem súlyozott, akkor a csúcsmátrixos ábrázolás tovább javítható. Ebben az esetben a mátrix elemei lehetnek bitek, így jelentősen csökkenthetjük a szükséges tárterület méretét.

6.2. Bináris kereső fák

6.10. definíció. *A bináris kereső fa*
A **bináris kereső fa** egy bináris fa, amely rendelkezik az alábbi **bináris kereső fa tulajdonság**gal:

- Legyen x a bináris kereső fa tetszőleges csúcsa.
- Ha y az x baloldali részfájában van, akkor $kulcs[y] \leq kulcs[x]$.
- Ha y az x jobboldali részfájában van, akkor $kulcs[x] \leq kulcs[y]$.

A jellemző műveletek bináris kereső fában:
KERES, MINIMUM, MAXIMUM, ELŐZŐ, KÖVETKEZŐ,BESZÚR, TÖRÖL.

A műveletek általában a fa magasságával függenek össze, amely lehet $\log n$, de lehet n is.

6.3. Bináris kereső fa inorder bejárása

	5.1.1. algoritmus
	Inorder fabejárás
	// $T(n) = \Theta(n)$
1	INORDER_FA_BEJÁRÁS(x)
2	**IF** $x \neq$ **NIL**
3	**THEN** INORDER_FA_BEJÁRÁS($bal[x]$)
4	Print($kulcs[x]$)
5	INORDER_FA_BEJÁRÁS($jobb[x]$)

INORDER_FA_BEJÁRÁS(gyökér[T]) bejárja az egész fát.

Az **inorder** bejárással növekvő sorrendben tudjuk a kulcsokat kiiratni. **Preorder** bejárás esetén kulcskiírás a részfák előtt, **postorder** bejárás esetén a részfák után történik.

6.4. Bináris kereső fa műveletek

	5.1.2. algoritmus
	Keresés bináris fában
	// Rekurzív változat
	// $T(n) = \Theta(h)$
1	**FÁBAN_KERES** (x, k)
2	**IF** $x = NIL$ vagy $k = kulcs[x]$
3	**THEN RETURN** (x)
4	**IF** $k < kulcs[x]$
5	**THEN RETURN** (FÁBAN_KERES($bal[x], k$))
6	**ELSE RETURN** (FÁBAN_KERES($jobb[x], k$))

	5.1.3. algoritmus
	Keresés bináris fában
	// Iteratív változat
	// $T(n) = \Theta(h)$
1	**FÁBAN_ITERATÍVAN_KERES (x,k)**
2	**WHILE** $x \neq NIL$ vagy $k \neq kulcs[x]$ **DO**
3	**IF** $k < kulcs[x]$
4	**THEN** $x \leftarrow bal[x]$
5	**ELSE** $x \leftarrow jobb[x]$
6	**RETURN**(x)

	5.1.4. algoritmus
	Minimális elem keresés bináris fában
	// $T(n) = \Theta(h)$
1	**FÁBAN_MINIMUM** (x)
2	**WHILE** $bal[x] \neq NIL$ **DO**
3	$x \leftarrow bal[x]$
4	**RETURN** (x)

	5.1.5. algoritmus
	Maximális elem keresés bináris fában
	// $T(n) = \Theta(h)$
1	**FÁBAN_MAXIMUM** (x)
2	**WHILE** $jobb[x] \neq NIL$ **DO**
3	$x \leftarrow jobb[x]$
4	**RETURN** (x)

	5.1.6. algoritmus
	Következő elem keresés bináris fában
	// $T(n) = \Theta(h)$
1	**FÁBAN_KÖVETKEZŐ** (x)
2	**IF** $jobb[x] \neq NIL$
3	**THEN RETURN** (FÁBAN_MINIMUM($jobb[x]$))
4	$y \leftarrow szülő[x]$
5	**WHILE** $y \neq NIL$ és $x = jobb[y]$ **DO**
6	$x \leftarrow y$
6	$y \leftarrow szülő[y]$
7	**RETURN**(y)

	5.1.7. algoritmus
	Bináris fába beszúrás
	// $T(n) = \Theta(h)$
1	**FÁBA_BESZÚR**(T, z)
2	$y \leftarrow NIL$
3	$x \leftarrow$ *gyökér*$[T]$
4	**WHILE** $x \neq NIL$ **DO**
5	$y \leftarrow x$
6	**IF** *kulcs*$[z] <$ *kulcs*$[x]$
7	**THEN** $x \leftarrow$ *bal*$[x]$
8	**ELSE** $x \leftarrow$ *jobb*$[x]$
9	*szülő*$[z] \leftarrow y$
10	**IF** $y = NIL$
11	**THEN** *gyökér*$[T] \leftarrow z$
12	**ELSE IF** *kulcs*$[z] <$ *kulcs*$[y]$
13	**THEN** *bal*$[y] \leftarrow z$
14	**ELSE** *jobb*$[y] \leftarrow z$

5.1.8. algoritmus	
Bináris fából törlés	
//	$T(n) = \Theta(h)$
FÁBÓL_TÖRÖL(T, z)	
Három esetre bontjuk a törlést:	
z-nek nincs gyereke:	egyszerűen kivágjuk
z-nek egy gyereke van:	kivágjuk úgy, hogy a szülője és a gyereke között kapcsolatot hozunk létre.
z-nek két gyereke van:	kivágjuk z azon legközelebbi rákövetkezőjét, amelynek már nincs baloldali gyereke és ezt a rákövetkezőt z helyére illesztjük.

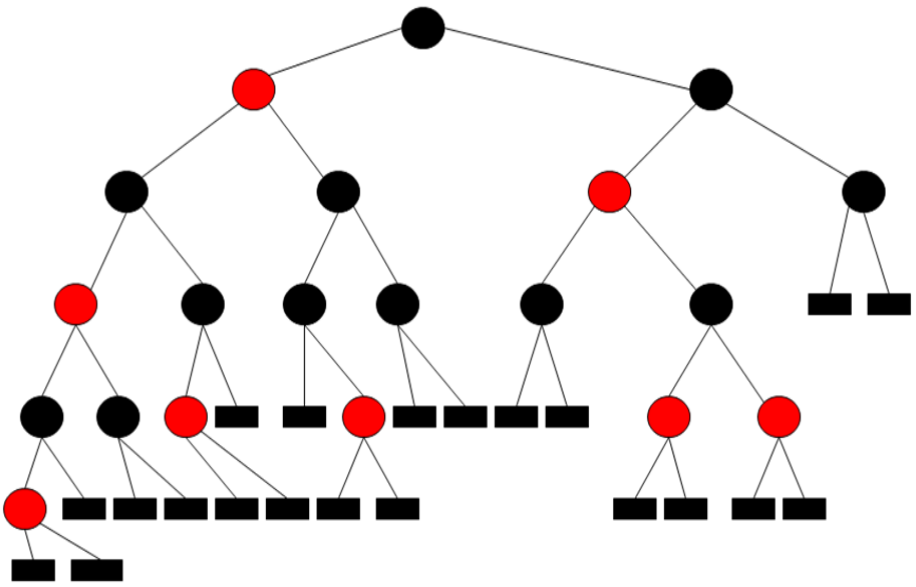
6.5. Piros-fekete fák

6.11. definíció. *A piros-fekete fa*

A **piros-fekete fa** olyan bináris keresőfa, melynek minden csúcsa egy extra bit információt tartalmaz (a csúcs színét, amely piros, vagy fekete) és rendelkezik az alábbi **Piros-fekete fa tulajdonság**okkal:

- Minden csúcs színe piros, vagy fekete.
- Minden levél (NIL) színe fekete, a levél nem tartalmaz kulcsot.
- Minden piros csúcsnak mindkét fia fekete.
- Bármely két, azonos csúcsból induló, levélig vezető úton ugyanannyi fekete csúcs van.

6.12. példa.



6.13. definíció. *A piros-fekete fa fekete magassága*
Egy piros-fekete fában egy x csúcs **fekete magasság**ának nevezzük az x csúcsból kiinduló, levélig vezető úton található, x-et nem tartalmazó fekete csúcsok számát.

A **fa fekete magasság** a gyökércsúcs fekete magassága.

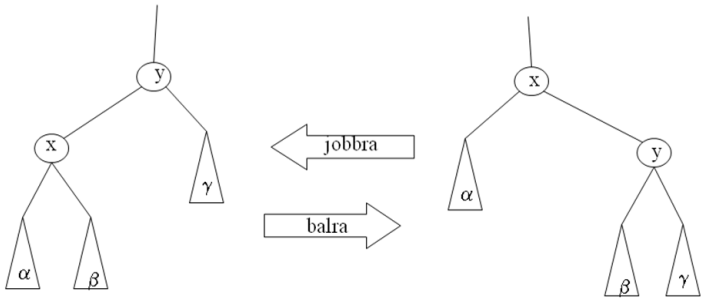
6.14. tétel.
Bármely n belső csúcsot tartalmazó piros-fekete fa magassága legfeljebb

$$2\log(n+1).$$

MŰVELETEK:

A bináris kereső fák műveletei piros-fekete fában $O(\log n)$ idő alatt elvégezhetők. Műveletek: BALRA_FORGAT(T, x), JOBBRA_FORGAT(T, x), PF_FÁBA_BESZÚR(T, x), PF_FÁBÓL_TÖRÖL(T, x).

6.15. példa. A forgatások hatását az alábbi ábrával szemléltetjük.



A műveletek közül most csak a beszúrást írjuk le részletesebben.

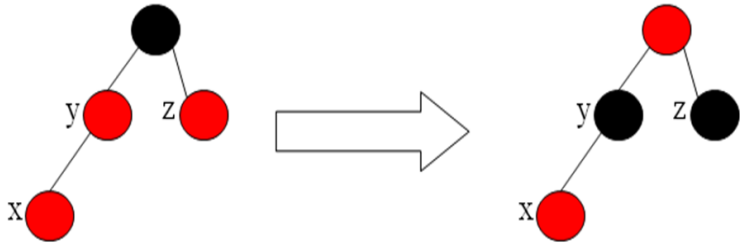
6.5.1. Beszúrás

1. Megkeressük az új elem helyét és oda piros színnel beszúrjuk.

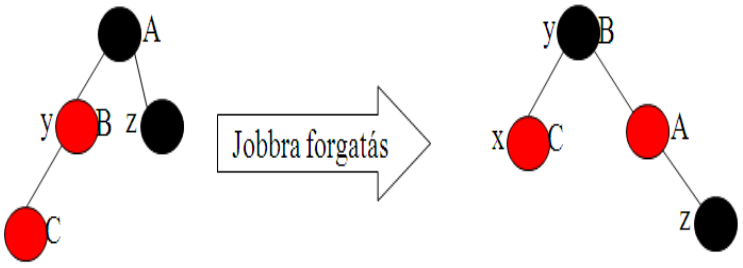
2. Csak a 3. Piros-fekete tulajdonság sérülhet abban az esetben, ha a beszúrt elem szülője piros. A cél, hogy az ilyen csúcsot feljebb és feljebb vigyük a fában úgy, hogy a többi tulajdonság ne sérüljön. Ha nem tudunk már feljebb menni, akkor forgatunk. (Feltehetjük, hogy a gyökér színe mindig fekete.)

3. Jelölések: x a vizsgált piros csúcs, y az x piros szülője (y szülője biztosan fekete) z az y testvére (x-nek nagybácsija). Hat eset adódik, amely a szimmetria (y bal- vagy jobbgyerek) miatt 3-ra redukálódik:

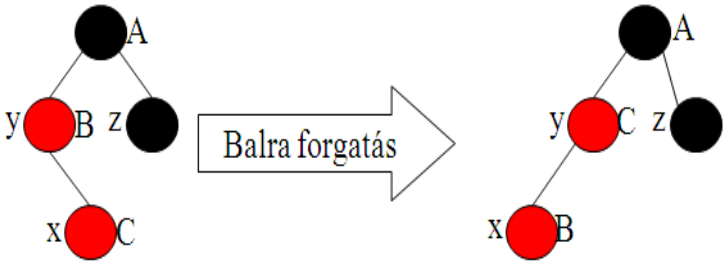
   (a) z **piros esetén** y és z feketére *szülő*$[y]$ pirosra változtatása után a problémás csúcs már csak a *szülő*$[x]$ lehet, ez lesz az új x vizsgálandó csúcs.



   (b) z **fekete és** x y**-nak baloldali gyereke esetén** jobbra forgatást és színcseréket végzünk: y feketére, volt szülője pirosra vált.



(c) z **fekete és** x y**-nak jobboldali gyereke esetén** olyan helyzetet állítunk elő, hogy x y-nak baloldali gyereke legyen balra forgatással (x és y szerepcsere). Ezután a (b) eset áll elő.



6.6. AVL-fák

Az AVL-fákat G.M. **A**delszon-**V**elszkij, E.M. **L**andisz vezették be 1962-ben.

6.16. definíció. *AVL-fa*
Egy bináris keresőfát AVL-fának nevezünk, ha a fa minden x csúcsára teljesül az alábbi úgynevezett AVL tulajdonság: (kiegyensúlyozottsági feltétel).

$$|h\left(bal\left(x\right)\right) - h\left(jobb\left(x\right)\right)| \leq 1$$

$Bal(x)$ az x csúcs baloldali, $jobb(x)$ a jobboldali részfáját jelöli, h pedig a részfa magassága.

Jelölje k a fa szintjeinek számát. (Ez eggyel nagyobb, mint a magasság). Jelölje G_k a k szintű fa csúcsainak minimális számát. Ekkor minden k>2 esetén teljesül, hogy

$$G_k = 1 + G_{k-1} + G_{k-2}$$

6.17. tétel.
$G_k = F_{k+2} - 1, k > 1$, *ahol* F_{k+2} *a* $k+2$*-dik Fibonacci szám.*

Bizonyítás. $k = 1$ és $k = 2$ esetén a tétel nyílvánvalóan igaz.

Teljes indukcióval $k > 2$-re kapjuk:

$$G_k = 1 + G_{k-1} + G_{k-2} = 1 + F_{k+1} - 1 + F_k - 1 = (F_{k+1} + F_k) - 1 = F_{k+2} - 1$$

□

6.18. következmény.
Egy n csúcsot tartalmazó AVL-fa szintjeinek k számára fennáll a

$$k \leq 1.44 \log 2(n+1)$$

egyenlőtlenség, azaz a szintszám és így a magasság is $O(logn)$ nagyságú.

Bizonyítás. A tétel alapján $n \geq F_{k+2} - 1$, azaz $F_{k+2} \leq n + 1$. A Fibonacci számokra igaz, hogy $\Phi_{n-2} \leq F_n \leq \Phi_{n-1}$, ahol $\Phi = (1 + \sqrt{5})/2 \approx 1,61803....$
Tehát $\Phi_k \leq F_{k+2} \leq n + 1$.
Innen logaritmust véve $k \leq \log \Phi(n+1) \approx 1.44 \log 2(n+1)$. □

6.19. tétel.
Egy n csúcsból álló AVL-fa esetén egy új csúcs beszúrását követően legfeljebb egy (esetleg dupla) forgatással az AVL tulajdonság helyreállítható. A beszúrás költsége ezzel együtt $O(logn)$.
Törlés után legfeljebb $1.44 \log 2n$ (szimpla vagy dupla) forgatás helyreállítja az AVL tulajdonságot.

6.7. 2-3-fák

6.20. definíció. *2-3-fa*
A 2-3-fa egy gyökeres fa az alábbi tulajdonságokkal:

1. A rekordok a fa leveleiben helyezkednek el, a kulcs értéke szerint balról jobbranövekvő sorrendben. Egy levél egy rekordot tartalmaz.

2. Minden belső (nem levél) csúcsnak 2 vagy három gyereke van. A csúcs szerkezete: m_1, k_1, m_2, vagy m_1, k_1, m_2, k_2, m_3 ahol ($k_1 < k_2$. Az m_1 mutató szerinti részfában minden kulcs kisebb, mint k_1, az m_2 mutató szerinti részfa legkisebb kulcsa k_1, és minden kulcs ott kisebb, mint k_2 (ha van a csúcsban), az m_3 mutató (ha van) szerinti részfában a legkisebbkulcs k_2.

3. A fa levelei a gyökértől egyforma távolságra vannak.

6.21. tétel.
Ha a 2-3-fának k szintje van, akkor a levelek száma legalább $2k - 1$.
Fordítva: ha a tárolt rekordok száma n, akkor $k \leq \log 2n + 1$

Keresés 2-3-fában: A gyökértől elindulva a belső csúcsokban 1 vagy 2 összehasonlítás után tovább lehet menni egy szinttel lejjebb. A műveletigény $\Theta(\log n)$.

6.8. B-fák

Definíció: A B-fa egy olyan gyökeres fa, amely rendelkezik a következő tulajdonságokkal:
Minden csúcsnak a következő mezői vannak:

1. Az $n[x]$ az x csúcsban tárolt kulcsok darabszáma. Az $n[x]$ darab kulcs (nemcsökkenő sorrendben) $kulcs_1[x] \leq kulcs_2[x] \leq \ldots \leq kulcs_{n[x]}[x]$. A $levl[x]$ egy logikai változó, melynek az értéke $IGAZ$, ha x levél, és $HAMIS$, ha x egy belső csúcs.

2. Ha x belső csúcs, akkor tartalmazza a $c_1[x], c_2[x], \ldots, c_{n[x]+1}$ mutatókat, melyek az x gyerekeire mutatnak. A levél csúcsoknak nincsenek gyerekeiés így e mezők definiálatlanok.

3. A $kulcs_i[x]$ értékek meghatározzák a kulcsértékek azon tartományait, amelyekbe a részfák kulcsai esnek. Ha ki egy olyan kulcs, amelyik a $c_i[x]$ gyökerű részfában van, akkor $k_1 \leq kulcs_1[x] \leq k_2 \leq kulcs_2[x] \leq \ldots \leq k_n[x] \leq kulcs_{n[x]}[x] \leq k_{n[x]+1}$.

4. Minden levélnek azonos a mélysége, ez az érték a fa magassága

5. A csúcsokban található kulcsok darabszámára adott egy alsó és egy felső korlát. Ezeket a korlátokat egy rögzített t egész számmal ($t \geq 2$) lehet kifejezni, és ezt a számot a **B-fa minimális fokszámának** nevezzük.A. Ezért minden nem gyökér csúcsnak legalább $t-1$ kulcsa van. Minden belső csúcsnak legalább t gyereke van. Ha a fa nem üres, akkor a gyökércsúcsnak legalább egy kulcsának kell lennie. Minden csúcsnak legfeljebb $2t-1$ kulcsa lehet. Tehát egy belső csúcsnak legfeljebb $2t$ gyereke lehet. Azt mondjuk, hogy egy csúcs telített, ha pontosan $2t-1$ kulcsa van.

Ha $t = 2$, akkor a B-fa neve **2-3-4 fa.**

6.22. tétel. *Ha $n \geq 1$, és T egy olyan n-kulcsos B-fa, amelynek magassága h és minimális fokszáma $t \geq 2$, akkor*

$$h \leq \log t((n+1)/2).$$

7. fejezet

Gráfelméleti algoritmusok

7.1. A szélességi keresés

1. G éleit vizsgálja és rátalál minden s-ből elérhető csúcsra.

2. Kiszámítja az elérhető csúcsok legrövidebb (legkevesebb élből álló) távolságát s-től.

3. Létrehoz egy s gyökerű „szélességi fát”, amelyben az s-ből elérhető csúcsok vannak.

4. A csúcsoknak szint tulajdonít (fehér, szürke, fekete). Kezdetben minden csúcs fehér, kivéve s-et, amely szürke. Szürke lesz egy csúcs, ha elértük és fekete, ha megvizsgáltuk az összes belőle kiinduló élt.

5. A szélességi fa kezdetben az s csúcsból áll. Ez a gyökér.

6. Ha egy fehér v csúcshoz értünk az u csúcsból, akkor azt felvesszük a fába (u, v) éllel és u lesz a v szülője.

Attributumok

$szín[u]$	az u csúcs színe
$\pi[u]$	az u csúcs előd je
$táv[u]$	az u távolsága s-től
Q	a szürke csúcsok sora

	SZÉLESSÉGI_KERESŐ(G, s) $O(V+E)$
1	**FOR** $\forall \in$V[G]\[s] csúcsra **DO**
2	$szin[u] \leftarrow$FEHÉR
3	$táv[u] \leftarrow \infty$
4	$\pi[u] \leftarrow$ NIL
5	$Szin[s] \leftarrow$ SZÜRKE
6	$táv[s] \leftarrow 0$
7	$\pi[s] \leftarrow$ NIL
8	$Q \leftarrow \{s\}$
9	**WHILE** $Q \neq \emptyset$ **DO**
10	$u \leftarrow fej[Q]$
11	**FOR** $\forall v \in szomszéd[u]$-ra **DO**
12	**IF** $szin[v] =$ FEHÉR
13	**THEN** $szin[v] \leftarrow$SZÜRKE
14	$táv[v] \leftarrow táv[u]+1$
15	$\pi[v] \leftarrow u$
16	SORBA(Q,v)
17	SORBÓL(Q)
18	$szin[u] \leftarrow$ FEKETE

7.1. definíció. $\delta(s, v)$ jelölje a legrövidebb úthosszat s-ből v-be, ha létezik út s-ből v-be, egyébként δ(s,v) legyen ∞.

7.2. lemma. *Legyen $G = (V, E)$ digráf vagy gráf és $s \in V$ tetszőleges csúcs. Ekkor bármely $(u, v) \in E$ él esetén $\delta(s, v) \leq \delta(s, u) + 1$.*

7.3. lemma. *Legyen $G = (V, E)$ gráf és tegyük fel, hogy a szélességi keresés algoritmust alkalmaztuk egy $s \in V$ kezdőcsúccsal. Ekkor a szélességi keresés által kiszámított* táv *értékek minden $v \in V$ csúcsra kielégítik a* táv[v] $\geq \delta(s, v)$ *egyenlőtlenséget.*

7.4. lemma. *Tegyük fel, hogy a szélességi keresést alkalmaztuk a $G = (V, E)$ gráfra és a futás során a Q sor a $v_1, \ldots, v_r$ csúcsokat tartalmazza. (v_1 az első, v_r az utolsó). Ekkor $táv[v_r] \leq táv[v_1] + 1$ és $táv[v_i] \leq táv[v_{i+1}]$ bármely $i = 1, \ldots, r-1$ értékre.*

7.5. tétel. *Legyen $G = (V, E)$ gráf és tegyük fel, hogy a szélességi keresés algoritmust alkalmaztuk egy $s \in V$ kezdőcsúccsal. Ekkor a szélességi keresés minden s-ből elérhető csúcsot elér és befejezéskor $távolság[v] = \delta(s, v)$, $s \in V$. Továbbá bármely s-ből elérhető $v \neq s$ csúcsra az s-ből v-be vezető legrövidebb utak egyikét megkapjuk, ha az s-ből $\pi[v]$-be vezető legrövidebb utat kiegészítjük a $(\pi[v], v)$ éllel.*

7.6. definíció. $G_\pi = (V_\pi, E_\pi)$ fát **előd részfá**nak nevezzük, ha

$$V_\pi = \{v \in V \ : \ \pi[v] \neq NIL\} \cup \{s\}$$

és

$$E_\pi = \{(\pi[v], v) \in E \ : \ v \in V_\pi \backslash \{s\}\}$$

7.7. definíció. A G_π előd részfa **szélességi fa**, ha V_π elemei az s-ből elérhető csúcsok és bármely $v \in V_\pi$ csúcsra egyetlen egyszerű út vezet s-ből v-be G_π-ben

7.8. lemma. *A szélességi keresés olyan π értékeket határoz meg, amelyekre a $G_\pi = (V_\pi, E_\pi)$ előd részfa egy szélességi fa.*

Eljárás az s-ből v-be vezető legrövidebb út csúcsai kiírására:

	UTAT_NYOMTAT(G, s, v)
1	**IF** $v = s$
2	**THEN** PRINT(s)
3	**ELSE IF** $\pi[v] =$ NIL
4	**THEN** PRINT(„nincs út s és v között”)
5	**ELSE** UTAT_NYOMTAT$(G, s, \pi[v])$
6	PRINT(v)

7.2. A mélységi keresés

1. G éleit vizsgálja, mindig az utoljára elért, új kivezető élekkel rendelkező v csúcsból kivezető, még nem vizsgált éleket deríti fel. Az összes ilyen él megvizsgálása után visszalép és azon csúcs éleit vizsgálja, amelyből v-t elértük.

2. Az összes csúcsot megvizsgálja.

3. Létrehoz egy „mélységi erdőt”, amely az előd részgráf fáiból áll.

4. A csúcsoknak szint tulajdonít (fehér, szürke, fekete). Kezdetben minden csúcs fehér, szürke lesz, mikor elértük, és fekete, mikor elhagytuk.

5. Minden csúcshoz két időpontot rendel, az elérési $d[v]$ és az elhagyási időpontot $f[v]$.

7.9. definíció. A $G_\pi = (V, E_\pi)$ gráfot **előd részgráf**nak nevezzük, ha

$$E_\pi = \{(\pi[v], v) \in E \ : \ v \in V \text{ és } \pi[v] \neq NIL\}.$$

	MÉLYSÉGI_KERESŐ(G) $\Theta(V+E)$
1	**FOR** $\forall u \in V[G]$ csúcsra **DO**
2	$szin[u] \leftarrow$ FEHÉR
3	$\pi[u] \leftarrow$ NIL
4	idő $\leftarrow 0$
5	**FOR** $\forall u \in V[G]$ csúcsra **DO**
6	**IF** $szin[u]$ =FEHÉR
7	**THEN** MK_BEJÁR(u)

	MK_BEJÁR(u) $\Theta(V+E)$
1	$szin[u] \leftarrow$ SZÜRKE
2	$d[u] \leftarrow$ idő $\leftarrow$ idő+1
3	**FOR** $\forall v \in$ *szomszéd*$[u]$ csúcsra **DO**
4	**IF** $szin[v]$ = FEHÉR
5	**THEN** $\pi[v] \leftarrow u$
6	MK_BEJÁR(v)
7	$szin[u] \leftarrow$ FEKETE
8	$f[u] \leftarrow$ idő $\leftarrow$ idő+1

7.10. tétel (Zárójelezés tétele).
Mélységi keresést alkalmazva egy $G = (V, E)$ (irányított, vagy iráyítatlan) gráfra a következő 3 feltétel közül pontosan 1 teljesül bármely u és v csúcsra:

- *a $[d[u], f[u]]$ és a $[d[v], f[v]]$ intervallumok diszjunktak,*
- *a $[d[v], f[v]]$ intervallum tartalmazza a $[d[u], f[u]]$ intervallumot, és az u csúcs a v csúcs leszármazottja a mélységi fában,*
- *a $[d[u], f[u]]$ intervallum tartalmazza a $[d[v], f[v]]$ intervallumot, és a v csúcs az u csúcs leszármazottja a mélységi fában.*

7.11. következmény (Leszármazottak intervallumainak beágyazása).
A v csúcs akkor és csak akkor leszármazottja az u csúcsnak az irányított, vagy irányítatlan G gráf mélységi erdejében, ha $d[u] < d[v] < f[v] < f[u]$.

7.12. tétel (Fehér út tétele).
Egy $G = (V, E)$ gráfhoz tartozó mélységi erdőben a v csúcs akkor és csak akkor leszármazottja az u csúcsnak, ha u elérésekor a $d[u]$ időpontban a v csúcs elérhető u-ból olyan úton, amely csak fehér csúcsokat tartalmaz.

A mélységi keresés révén a mélységi kereséstől függően a bemeneti gráf éleit osztályozhatjuk. Éltípusok: egy (u, v) él

1. **Fa él**, ha a G_π mélységi erdő éle.
2. **Visszamutató él**, ha v megelőzője u-nak egy mélységi fában.
3. **Előre mutató él**, ha v leszármazottja u-nak egy mélységi fában.
4. **Kereszt él**, ha a fenti három osztályba nem sorolható be

Irányított gráf akkor és csak akkor körmentes, ha a mélységi keresés során nem találtunk visszamutató éleket.

7.13. tétel. *Egy irányítatlan G gráf mélységi keresésekor bármely él vagy fa él, vagy visszamutató él.*

Egy irányított $G = (V, E)$ gráf topologikus rendezése a csúcsainak sorba rendezése úgy, hogy ha G-ben szerepel az (u, v) él, akkor u előzze meg v-t a sorban

	TOPOLOGIKUS_RENDEZÉS(G) $\Theta(V + E)$
1	MÉLYSÉGI_KERESŐ(G) hívása, minden csúcsra meghatározzuk az $f[u]$ elhagyási időt.
2	Az egyes csúcsok elhagyásakor szúrjuk be azokat egy láncolt lista elejére
3	**RETURN**(a csúcsok listája)

7.14. tétel. *A TOPOLOGIKUS_RENDEZÉS(G) egy irányított, körmentes gráf topologikus rendezését állítja elő.*

7.3. Minimális feszítőfa

7.15. definíció. Egy irányítatlan gráf feszítőfája a gráfnak az a részgráfja, amely fagráf és tartalmazza a gráf összes cúcspontját.

7.16. definíció. A fa súlya a

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

számérték .

7.17. definíció. Minimális feszítőfáról beszélünk, ha $w(T)$ értéke minimális az összes T feszítőfára nézve. A minimális feszítőfa nem feltétlenül egyértelmű.

7.18. definíció. Legyen A egy minimális feszítőfa egy része. A-ra nézve **biztonságos** egy él, ha A-hoz hozzávéve A továbbra is valamely minimális feszítőfa része marad.

	Minimális_Feszítő_Fa(G, w)
1	$A \leftarrow \emptyset$
2	**WHILE** A nem feszítőfa **DO**
3	keresünk egy biztonságos (u, v) élt az A-ra nézve
	$A \leftarrow A \cup \{(u, v)\}$
4	RETURN (A)

7.19. definíció. Egy irányítatlan $G = (V, E)$ **gráf vágása** a V kettéosztása egy S és egy $V \setminus S$ halmazra

7.20. definíció. Az (u, v) **él keresztezi az** $(S, V \setminus S)$ **vágást**, ha annak egyik végpontja S-ben, másik végpontja $V \setminus S$-ben található.

7.21. definíció. Egy **vágás kikerüli az** A **halmazt**, ha az A egyetlen éle sem keresztezi a vágást.

7.22. definíció. Egy **él könnyű egy vágásban**, ha a vágást keresztező élek közül neki van a legkisebb súlya.

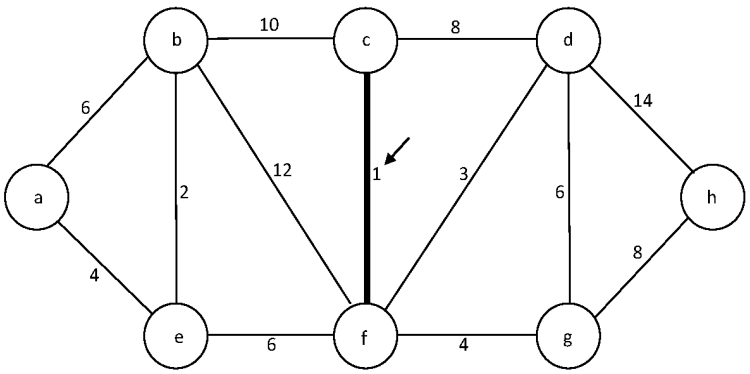
7.23. tétel. *Legyen $G = (V, E)$ egy összefüggő, irányítatlan gráf $w : E \to \mathbb{R}$ súlyfüggvénnyel. Legyen A egy olyan részhalmaza E-nek, amelyik G valamelyik minimális feszítőfájának is része. Legyen $(S, V \setminus S)$ tetszőleges A-t kikerülő vágása a G-nek. Legyen (u, v) könnyű él az $(S, V \setminus S)$ vágásban. Ekkor az (u, v) él biztonságos az A-ra nézve.*

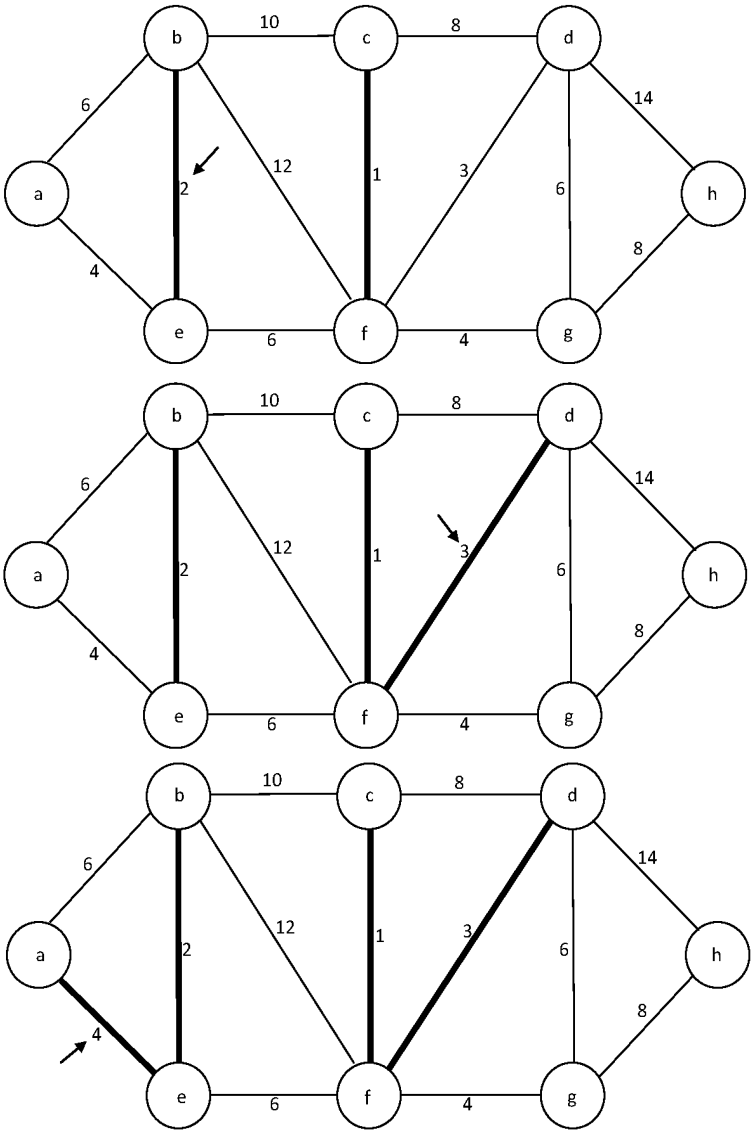
7.24. következmény. *Legyen $G = (V, E)$ egy összefüggő, irányítatlan gráf gráf $w : E \to \mathbb{R}$ súlyfüggvénnyel. Legyen A egy olyan részhalmaza E-nek, amelyik G valamelyik minimális feszítőfájának is része. Legyen C egy összefüggő komponens a $G_A = (V, A)$ erdőben. Ha (u, v) a C-t és a G_A valamely másik komponenesét összekötő könnyű él, akkor az (u, v) él biztonságos az A-ra nézve.*

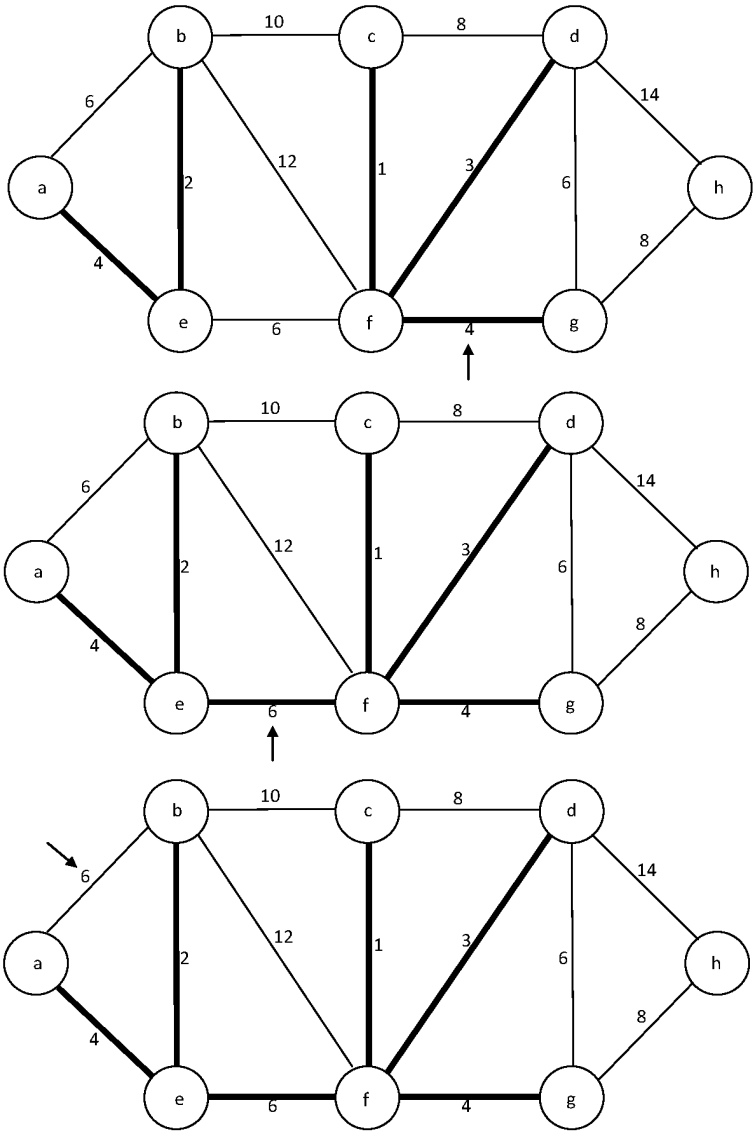
7.3.1. Kruskal-algoritmus

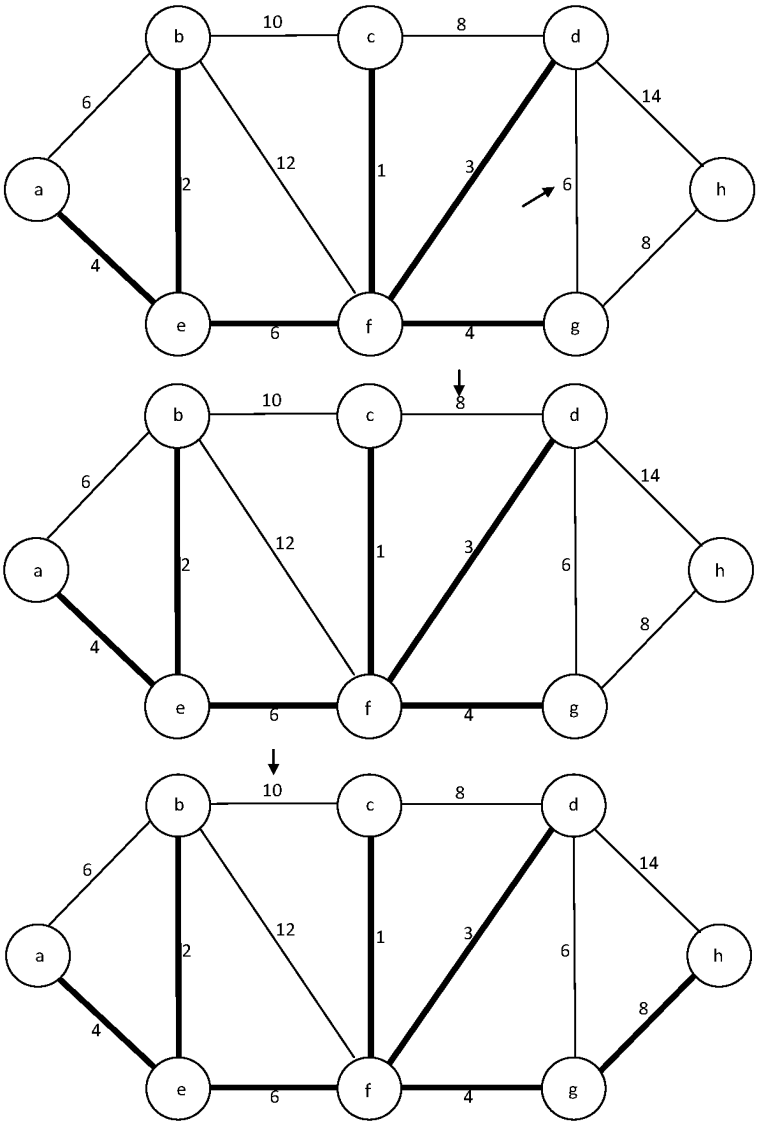
	MFF_KRUSKAL(G, w) O(E log E)
1	$A \leftarrow \emptyset$
2	**FOR** $\forall v \in V[G]$-re **DO**
3	HALMAZT_KÉSZÍT(v)
4	Rendezzük E éleit a súly szerint növekvő sorrendben
5	**FOR** $\forall (u, v) \in E$ élre az élek súly szerint növekvő sorrendjében **DO**
6	**IF** HALMAZT_KERES$(u) \neq$ HALMAZT_KERES(v)
7	**THEN** $A \leftarrow A \cup \{(u, v)\}$
8	EGYESÍT(u, v)
9	RETURN(A)

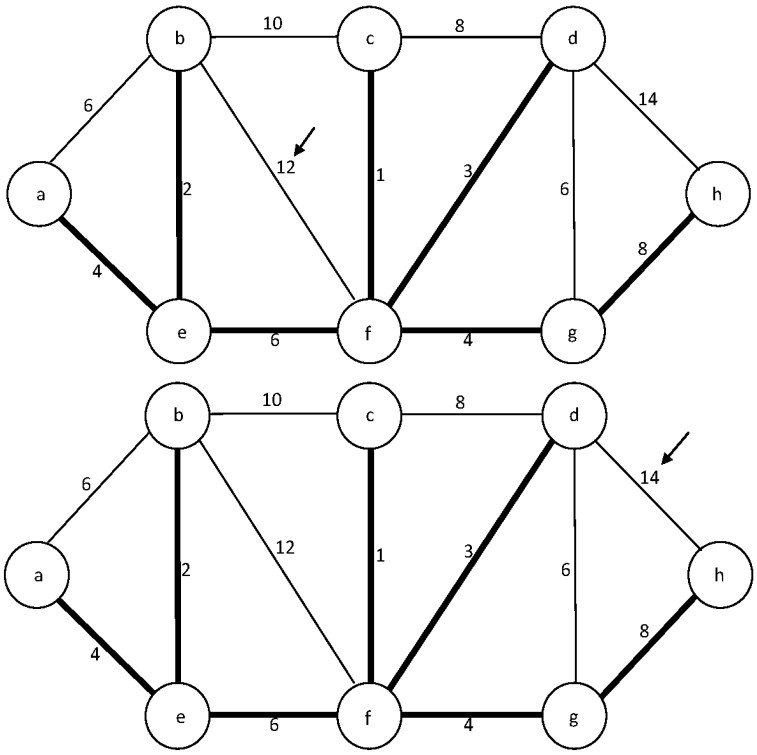
7.25. példa.




b
c
d
a
e
f
g
h
10
8
6
14
12
1
3
6
2
4
8
6
4
b
c
d
a
e
f
g
h
10
8
6
14
12
1
3
6
2
4
8
6
4
b
c
d
a
e
f
g
h
10
8
6
14
12
1
3
6
2
4
8
6
4


b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
2
6
8
4
6
4
b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
2
6
8
4
6
4
b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
2
6
8
4
6
4

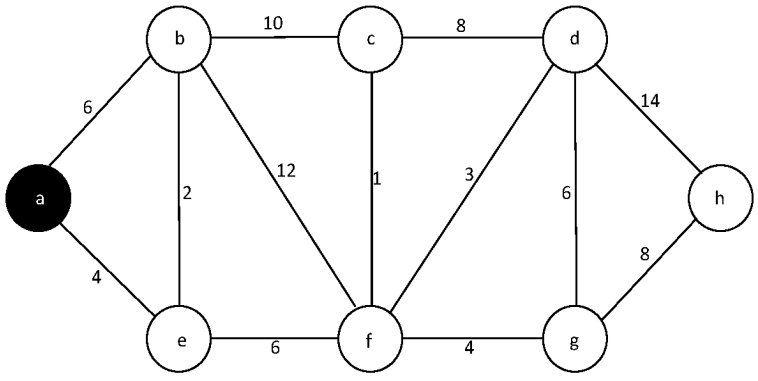

b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
6
2
8
4
6
4
b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
6
2
8
4
6
4
b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
6
2
8
4
6
4

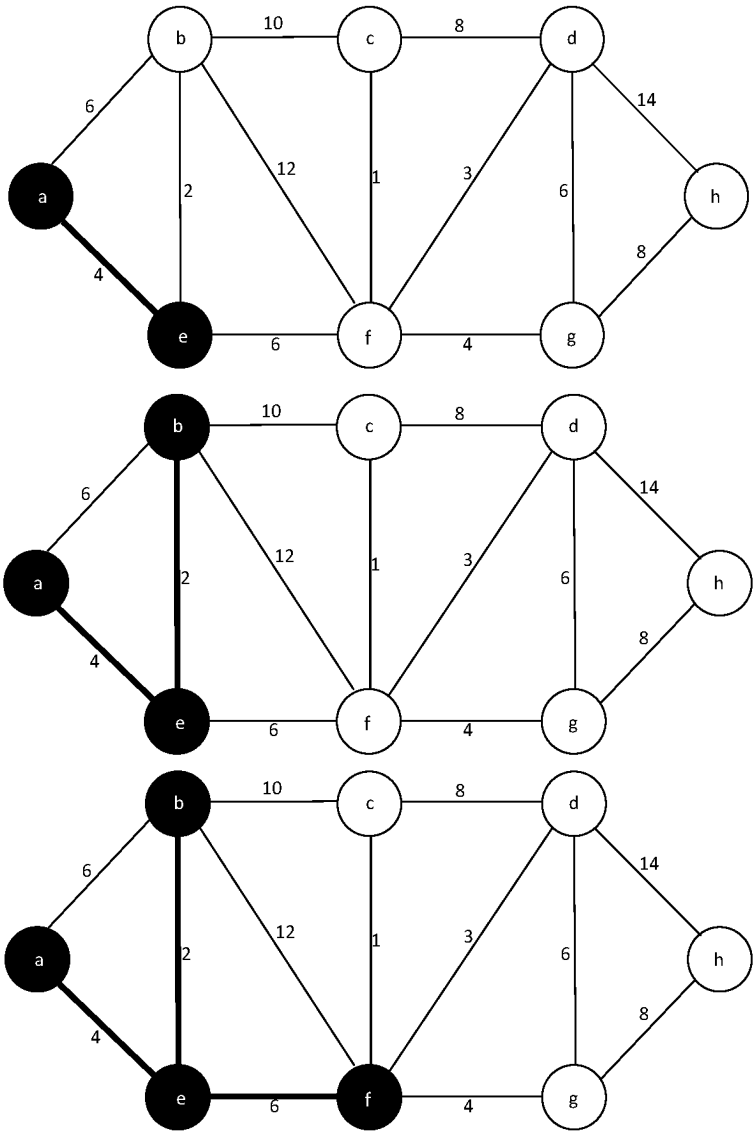

b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
2
6
8
4
6
4
b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
2
6
8
4
6
4

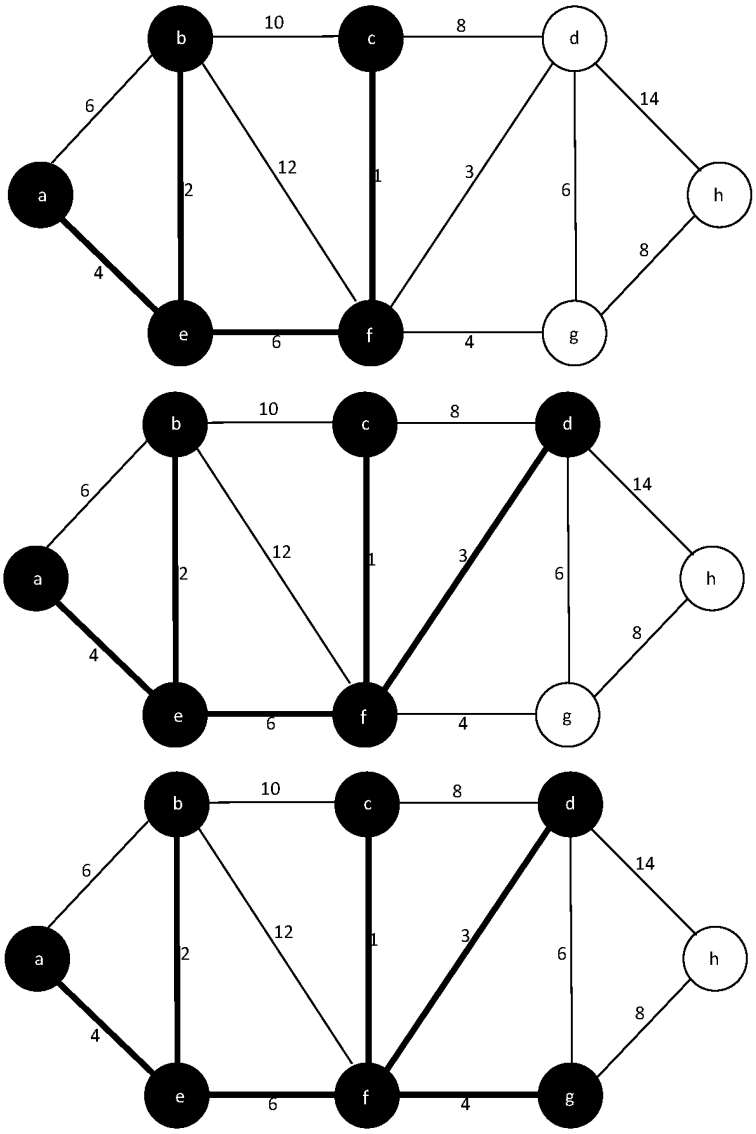
7.3.2. Prim-algoritmus

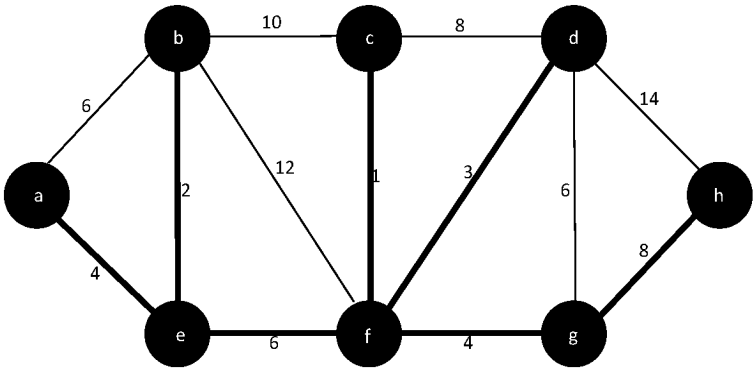
	MFF_PRIM(G, w) O(E log V)
1	$Q \leftarrow V[G]$
2	**FOR** $\forall v \in Q$-ra **DO**
3	$\quad kulcs[v] \leftarrow \infty$
4	$kulcs[r] \leftarrow 0$
5	$\pi[r] \leftarrow$ NIL
6	**WHILE** $Q \neq \emptyset$ **DO**
7	$\quad u \leftarrow$ KIVESZ_MIN(Q)
8	$\quad$ **FOR** $\forall v \in$ *szomszéd*$[u]$-ra **DO**
9	$\quad\quad$ **IF** $v \in Q$ és $w(u, v) \leq kulcs[v]$
10	$\quad\quad\quad$ **THEN** $\pi[v] \leftarrow u$
11	$\quad\quad\quad\quad kulcs[v] \leftarrow w(u, v)$
12	RETURN(π)

7.26. példa.




b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
2
6
8
4
6
4
b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
2
6
8
4
6
4
b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
2
6
8
4
6
4


b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
2
6
8
4
6
4
b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
2
6
8
4
6
4
b
c
d
a
h
e
f
g
10
8
6
14
12
1
3
2
6
8
4
6
4



7.4. Legrövidebb utak

A v csúcs ***legrövidebb út távolsága*** s-től legyen $\delta(s, v)$, amelyet az s-ből v-be vezető egyes utak élszámainak minimumaként definiálunk, ha van ilyen út, és ∞, ha nem vezet út s-ből v-be.

7.27. definíció. Egy $\delta(s, v)$ hosszúságú s-ből v-be vezető utat s és v közötti **legrövidebb útnak** nevezzük.

7.28. lemma. *Legyen $G = (V, E)$ irányított vagy irányítatlan gráf és $s \in V$ tetszőleges csúcs. Ekkor bármely $(u, v) \in E$ élre:*

$$\delta(s, v) \leq \delta(s, u) + 1.$$

Bizonyítás. Ha u elérhető s-ből, akkor v is. Ebben az esetben, az s-ből v-be vezető legrövidebb út nem lehet hosszabb, mint ha az s-ből u-ba vezető legrövidebb utat kiegészítjük az (u, v) éllel, így az egyenlőtlenség teljesül. Ha u nem érhető el s-ből, akkor $\delta(s, u) = \infty$, ezért az egyenlőtlenség biztosan igaz. □

7.5. Adott csúcsból induló legrövidebb utak

Egy gépkocsivezető a lehető legrövidebb úton szeretne eljutni az egyik városból (A-ból) a másikba (B-be). Hogyan határozhatjuk meg ezt a legrövidebb

utat, ha rendelkezünk az ország teljes autós térképével, amelyen minden, két szomszédos útkereszteződés közötti távolságot bejelöltek?

Egyik lehetséges megoldás az, ha módszeresen előállítjuk az összes, A-ból B-be vezető utat azok hosszával együtt, és kiválasztjuk a legrövidebbet. Könnyű azonban azt belátni, hogy még ha kört tartalmazó utakkal nem is foglalkozunk, akkor is több millió olyan lehetőség marad, amelyek többsége egyáltalán nem érdemel figyelmet. Például, egy C-n keresztül vezető A és B közötti út nyilvánvalóan szerencsétlen útvonalválasztás lenne, ha C túl hosszú kitérőt jelent.

A ***legrövidebb utak problémában*** adott egy élsúlyozott, irányított $G = (V, E)$ gráf, ahol a $w : E \to \mathbb{R}$ súlyfüggvény rendel az élekhez valós értékeket. A $p = \langle v_0, v_1, \ldots, v_k \rangle$ út ***súlya*** az utat alkotó súlyainak összege:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Definiáljuk az u-ból v-be vezető A ***legrövidebb út súlyát*** az alábbi módon:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}, & \text{ha vezet út} u\text{-ból } v\text{-be,} \\ \infty, & \text{különben.} \end{cases}$$

Az u csúcsból v csúcsba vezető ***legrövidebb úton*** egy olyan p utat értünk, amelyre $w(p) = \delta(u, v)$ teljesül.

Az A-ból B-be vezető utak példájában az autós térképet egy gráf segítségével modellezhetjük: a csúcsok jelentik a kereszteződéseket, az élek szimbolizálják a kereszteződések közötti útszakaszokat, az élek súlyai pedig az útszakaszok hosszaira utalnak. Célunk olyan legrövidebb útnak a megtalálása, amely A-ból adott indul ki, és B-be érkezik.

Az élsúlyok a távolságokétól eltérő metrikákat is kifejezhetnek. Gyakran használják idő, költség, büntetés, veszteség vagy más olyan mennyiség megjelenítésére, amely egy út mentén lineárisan halmozódik, és amelyet minimalizálni szeretnénk.

Fokozatos közelítés

Ennek a fejezetnek az algoritmusai a ***fokozatos közelítés*** módszerét alkalmazzák. Minden $v \in V$ csúcsnál nyilvántartunk egy $d[v]$ értéket, amely egy felső korlátot ad az s kezdőcsúcsból a v-be vezető legrövidebb út súlyára. A $d[v]$-t egy ***legrövidebb-út becslésnek*** nevezzük.

Kezdetben a legrövidebb-út becsléseket és a szülőkre mutató π értékeket a következő $\theta(V)$ futási idejű eljárás állítja be.

	Egy-forrás-kezdőérték$(G,\ s)$
1	**FOR** minden $v \in V[G]$-re **DO**
2	$d[v] \leftarrow \infty$
3	$\pi[v] \leftarrow$ NIL
4	$d[s] \leftarrow 0$

A kezdeti értékek beállítása után $v \in V$-re $\pi[v] =$ NIL, és minden $v \in V \setminus \{s\}$-re $d[v] = \infty$ áll fenn.

Egy (u, v) él segítségével történő ***közelítés*** technikáját alkalmazzák. Egy ellenőrzésből áll, amelyik összeveti a v csúcshoz ez idáig legrövidebbnek talált utat az u csúcson keresztül vezető úttal, és ha ez utóbbi rövidebb, akkor módosítja a $d[v]$ és $\pi[v]$ értékeket. A közelítő lépés csökkentheti a $d[v]$ legrövidebb-út becslés értékét, és átállíthatja a $\pi[v]$ mezőt az u csúcsra. Az alábbi kód az (u, v) él közelítő lépését írja le.

	Közelít$(u,\ v,\ w)$
1	**IF** $d[v] > d[u] + w(u, v)$
2	**THEN** $d[v] \leftarrow d[u] + w(u, v)$
3	$\pi[v] \leftarrow u$

Ennek a fejezetnek mindegyik algoritmusa meghívja az EGY-FORRÁS-KEZDŐÉRTÉK eljárást, majd az élekkel egymás után végez közelítéseket. Sőt a közelítés az egyetlen olyan lépés, amely megváltoztatja a legrövidebb-út becsléseket és a szülő értékeket. A fejezet algoritmusai abban különböznek egymástól, hogy hányszor és milyen sorrendben végzik el az élekkel a KÖZELÍT műveletet. Dijkstra algoritmusa minden éllel pontosan egyszer közelít. A Bellman-Ford-algoritmus az egyes élekkel többször végez közelítést.

7.5.1. Bellman-Ford algoritmus

A ***Bellman-Ford-algoritmus*** az adott kezdőcsúcsból induló legrövidebb utak problémáját abban az általánosabb esetben oldja meg, amikor az élek között negatív súlyúakat is találhatunk. Adott egy $w : E \to \mathbb{R}$ súlyfüggvénnyel súlyozott irányított $G = (V, E)$ gráf, ahol a kezdőcsúcs s. A Bellman-Ford-algoritmus egy logikai értéket ad vissza annak jelölésére, hogy van vagy nincs a kezdőcsúcsból elérhető negatív kör. Ha van ilyen kör, az algoritmus jelzi, hogy nem létezik megoldás. Ha nincs ilyen kör, akkor az algoritmus előállítja a legrövidebb utakat és azok súlyait.

A Bellman-Ford-algoritmus a fokozatos közelítés technikáját alkalmazza, bármelyik $v \in V$ csúcsnál az s kezdőcsúcsból odavezető legrövidebb út súlyára adott $d[v]$ becslését ismételten csökkenti mindaddig, amíg az eléri annak tényleges $\delta(s, v)$ értékét. Az algoritmus akkor és csak akkor tér vissza IGAZ értékkel, ha a gráf nem tartalmaz a kezdőcsúcsból elérhető negatív köröket.

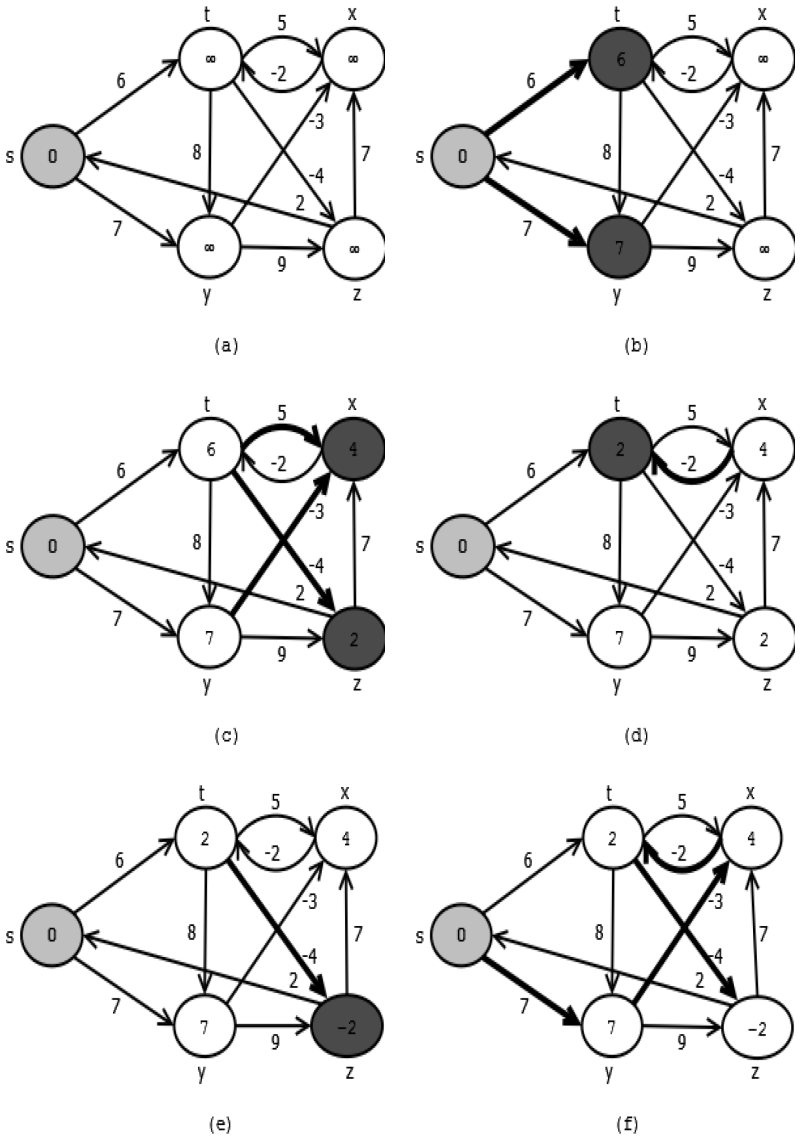
	BELLMAN–FORD(G, w, s)	
1	Egy-forrás-kezőérték (G,s)	O(VE)
2	**FOR** $i \leftarrow 1$ **TO** $\mid V[G] \mid -1$ **DO**	
3	**FOR** $\forall(u, v) \in E[G]$-re **DO**	
4	KÖZELÍT(u, v, w)	
5	**FOR** $\forall(u, v) \in E[G]$-re **DO**	
6	**IF** $d[v] > d[u] + w(u, v)$	
7	**THEN** RETURN(HAMIS)	
8	RETURN (IGAZ)	

	KIG-LEGRÖVIDEBB-ÚT(G, w, s)	
	A G csúcsainak topologikus rendezése	$\Theta(V + E)$
1	EGY_FORRÁS_KEZDŐÉRTÉK(G, s)	
2	**FOR** $\forall u$ csúcsra azok topologikus sorrendjében **DO**	
3	**FOR** $\forall v \in$ *szomszéd*$[u]$-ra **DO**	
4	KÖZELIT(u, v, w)	

A következő ábra a Bellman-Ford-algoritmus működését egy 5 csúcsból álló

gráfon mutatja be. A d és π kezdeti beállítása után az algoritmus $|V| - 1$ menetet végez a gráf éleivel. Minden menet a 2-4. sorok for ciklusának egy iterációja, amely a gráf minden élével egyszer végez közelítést. A (b)-(f) ábrák az algoritmus állapotait mutatják az élekkel végzett négy menet mindegyike után. $|V| - 1$ menet után az 5-8. sorok negatív kört keresnek, és a megfelelő logikai értéket adják vissza.

A Bellman-Ford-algoritmus futási ideje $O(V, E)$, mivel az 1. sor előkészítése $\theta(V)$ idejű, a 2-4. sorokban az élekkel végzett $|V| - 1$ menet mindegyike $\theta(E)$ ideig tart, és az 5-7. sorok **for** ciklusa $O(E)$ idejű.



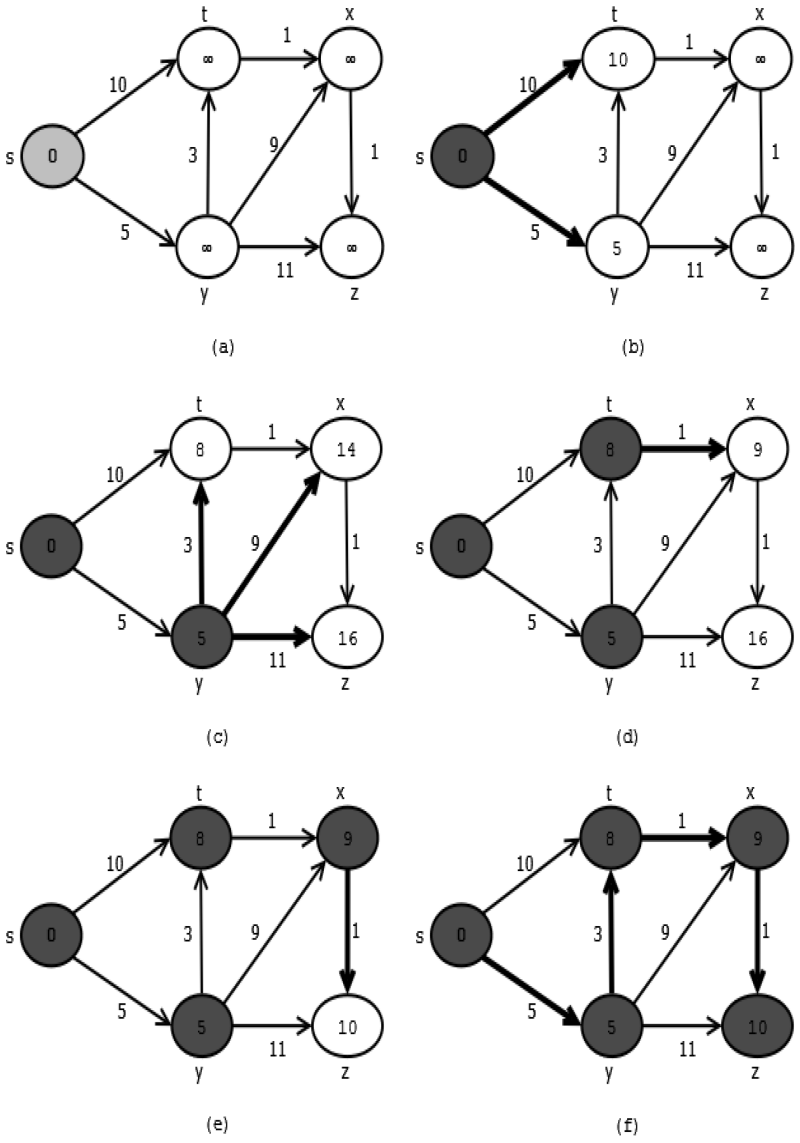
A kezdőcsúcs az s csúcs. A d értékeket beleírtuk a csúcsokba, és a vastagított élek jelzik a szülő értékeket: ha (u, v) él vastagított, akkor $\pi[u] = u$. **(a)** Az első menet előtti helyzet. **(b)-(f)** Az egymást követő menetek után kialakult helyzetek. Az **(f)** rész a végleges d és π értékeket mutatja. A Bellman-Ford-algoritmus ebben a példában IGAZ értékkel tér vissza.

7.29. tétel. ***Bellman-Ford-algoritmus helyessége***
Egy $G = (V, E)$ súlyozott, irányított gráfon futtassuk a BELLMAN-FORD *eljárást, ahol a súlyfüggvény a $w : E \to \mathbb{R}$, és a kezdőcsúcs az s. Ha G nem tartalmaz s-ből elérhető negatív köröket, akkor az algoritmus* IGAZ *értéket ad vissza, tovább minden $v \in V$ csúcsra $d[v] = \delta(s, v)$, és a G_π szülő részgráf egy s gyökerű legrövidebb-utak fa lesz. Ha G-ben van egy s-ből elérhető negatív kör, akkor az algoritmus* HAMIS *értékkel tér vissza.*

7.5.2. Dijkstra algoritmusa

Dijkstra algoritmusa az adott kezdőcsúcsból induló legrövidebb utak problémáját egy élsúlyozott, irányított $G = (V, E)$ gráfban abban az esetben oldja meg, ha egyik élnek sem negatív a súlya. Ebben az alfejezetben ennek megfelelően feltesszük, hogy minden $(u, v) \in E$ élre $W(u, v) \geq 0$. A Dijkstra algoritmusának futási ideje, egy jó megvalósítás mellett, gyorsabb, mint a Bellman-Ford-algoritmusé.

A Dijkstra-algoritmus azoknak a csúcsoknak az S halmazát tartja nyilván, amelyekhez már meghatározta az s kezdőcsúcsból odavazető legrövidebb-út súlyát. Az algotimus minden lépésben a legkisebb legrövidebb-út becslésű $u \in V \setminus S$ csúcsot választja ki, beteszi az u-t az S-be, és minden u-ból kivezető éllel egy-egy közelítést végez. Az alábbi megvalósításban egy Q minimum-elsőbbségi sort alkalmazunk a $V \setminus S$-beli csúcsok nyilvántartására, amelyeket azok *táv* értékeivel indexeltünk. Az algoritmus feltételezi, hogy a G gráf egy szomszédsági listával van megadva.



A kezdőcsúcs a bal oldali s csúcs. A legrövidebb-út becsléseket a csúcsok belsejében tüntettük fel, és vastagított élek jelzik a szülő csúcsokat: ha (u, v) vastag, akkor $\pi[v] = u$. A sötétebb szürke színű csúcsok az S halmazban vannak, és a fehér színűek a $Q = V \setminus S$ prioritási sorban. (**a**) Ez a 4-8. sorok **while** ciklusának első iterációja előtti helyzet. A sötét színű csúcs a legkisebb d értékű csúcs, és az 5. sor ezt választja ki u csúcsként. (**b**)-(**f**) A **while** ciklus soron következő iterációi utáni helyzetek. Minden részben a

sötétebb színnel jelölt csúcsot mint u csúcsot választja ki a következő iteráció 5. sora. Az (**f**) rész a végleges *táv* és π értékeket mutatja.

	Dijkstra(G, s)	
1	Egy-forrás-kezőérték (G,s)	O(V^2)
2	$S \leftarrow \emptyset$	
3	$Q \leftarrow V[G]$	
4	**WHILE** $Q \neq \emptyset$ **DO**	
5	$u \leftarrow$ KIVESZ_MIN(Q)	
6	$S \leftarrow S \cup \{u\}$	
7	**FOR** $\forall v \in$ *szomszéd*$[u]$-ra **DO**	
8	KÖZELÍT(u, v, w)	

A Dijkstra-algoritmus az ábrán bemutatott módon végzi az élekkel a fokozatos közelítést. Az 1. sor a *táv* és a π értékeinek szokásos kezdeti beállítását végzi, majd a 2. sor az S halmazt teszi üressé. A 4-8. sorok **while** ciklusának minden iterációja előtt fennáll a $Q = V \setminus S$ invariáns állítás. A 3. sor a Q minimum-elsőbbségi sort készíti elő úgy, hogy az kezdetben minden V-beli csúcsot tartalmazzon; mivel ekkor az S még üres, az invariáns állítás a 3. sor után teljesül. A 4-8. sorok **while** ciklusában egy u csúcsot veszünk ki az $Q = V \setminus S$ halmazból (legelőször $u = s$), és hozzáadjuk az S halmazhoz, tehát az invariáns állítás továbbra is igaz marad. Az u csúcs tehát a legkisebb legrövidebb-út becslésű csúcs a $V \setminus S$-ben. Ezután a 7-8. sorok közelítést végeznek az u-ból kivezető (u, v) élekkel, ezáltal módosítjuk a $d[v]$ becslést és a $\pi[v]$ szülőt, feltéve, hogy a v-hez az u-n keresztül most talált út rövidebb, mint az ott eddig nyilvántartott legrövidebb út. Figyelembe véve azt, hogy a 3. sor után már egyetlen csúcsot sem teszünk bele a Q-ba, valamint azt, hogy mindegyik csúcsot egyetlenegyszer veszünk ki a Q-ból és tesszük át az S-be, a 4-8. sorok **while** ciklusa pontosan $|V|$-szer hajtódik végre.

A Dijkstra-algoritmus mohó stratégiát alkalmaz, hiszen minden a „legkönnyebb”, a „legközelebbi” csúcsot választja ki a $V \setminus S$-ből, hogy azután betegye az S halmazba. A mohó stratégiák általában nem mindig adnak optimális eredményt, de amint a következő tétel és annak következménye mutatja, a Dijkstra-algoritmus szükségszerűen a legrövidebb utakat állítja elő.

7.30. tétel. ***Dijkstra-algoritmus helyessége***
Ha Dijkstra algoritmusát egy nemnegatív w súlyfüggvénnyel súlyozott, s kezdőcsúcsú irányított $G = (V, E)$ gráfban futtatjuk, akkor annak a befejeződésekor minden $u \in V$ csúcsra teljesül, hogy táv$[u] = \delta(s, u)$.

7.31. következmény. *Ha Dijkstra algoritmusát egy nemnegatív w súlyfüggvénnyel súlyozott, irányított s kezdőcsúcsú $G = (V, E)$ gráfban futtatjuk, akkor annak befejeződésekor a G_π szülő részgráf egy s gyökerű legrövidebb-utak fa lesz.*

7.6. Legrövidebb utak minden csúcspárra

Ebben a fejezetben célunk egy gráf valamennyi rendezett csúcspárjára a két csúcs közti legrövidebb út megkeresése. Ha például egy autóstérképhez elkészítjük a városok egymástól mért távolságainak táblázatát, éppen ezt a feladatot oldjuk meg. Csakúgy, mint korábban $G = (V, E)$ egy súlyozott irányított gráfot jelöl, amelynek élhalmazán egy $w : E \to \mathbb{R}$ valós értékű súlyfüggvény van megadva. A gráf minden $u, v \in V$ csúcspárjára keressük az u-ból v-be vezető legrövidebb (legkisebb súlyú) utat, ahol egy út súlya az úthoz tartozó élek súlyának az összege. Az eredményt általában táblázatos formában keressük; a táblázat u-hoz tartozó sorában és v-hez tartozó oszlopában álló elem az u-ból v-be vezető legrövidebb út hossza.

A csúcspárok közti legrövidebb utakat természetesen megkereshetjük úgy, hogy az előző fejezetben látott valamelyik egy kezdőcsúcsból kiinduló legrövidebb utakat kereső algoritmust egyenként végrehajtunk a lehetséges $|V|$ különböző gyökércsúcsot választva. Ha a távolságok nemnegatívak, Dijkstra algoritmusát alkalmazhatjuk.

Ha negatív élsúlyokat is megengedünk a gráfban, Dijkstra algortmusa nem alkalmazható, helyette a lassabb Bellman-Ford-algoritmust kell minden csúcsra egyszer végrehajtanunk. Célunk tehát, hogy a negatív élsúlyok esetére hatékonyabb algoritmusokat adjunk. Eközben megismerjük az összes csúcspár közit legrövidebb út probléma és a mátrixszorzás kapcsolatát is.

Az egy csúcsból kiinduló legrövidebb utakat megadó algoritmusok általában a gráf éllistás megadását igénylik. A bemenő adat tehát egy $Wn \times n$-es mátrix

lesz. A mátrix egy n csúcsú irányított $G = (V, E)$ gráf élsúlyait tartalmazza: $W = (w_{ij})$, ahol

$$w_{i,j} = \begin{cases} 0, & \text{ha } i = j, \\ \text{az irányított } (i,j) \text{ él hossza}, & \text{ha } i \neq j \text{ és } (i,j) \in E \\ \infty, & \text{ha } i \neq j \text{ és } (i,j) \notin E \end{cases} \tag{7.1}$$

A gráfban megengedünk negatív súlyú éleket, azonban egyelőre feltesszük, hogy negatív összsúlyú körök nincsenek.

A fejezetbeli algoritmus kimenete az összes párra adott legrövidebb úthosszakat tartalmazó táblázat egy D $n \times n$-es mátrix lesz. A mátrix d_{ij} eleme az i csúcsból a j csúcsba vezető legrövidebb út súlyát tartalmazza. A végeredményként kapott d_{ij} értékek tehát megegyeznek a $\delta(i, j)$-vel, az i csúcsból a j csúcsba vezető legrövidebb út súlyával.

Csak akkor mondhatjuk, hogy a kitűzött feladatot megoldottuk, ha a legrövidebb utak súlya mellett magukat az utakat is meg tudjuk adni. E célból egy $\prod = (\pi_{ij})$ ***megelőzési mátrixot*** is meg kell adnunk, amelyben $\pi_{ij} = \text{NIL}$, ha $i = j$ vagy ha nem vezet i és j között út; ellenkező esetben π_{ij} a j-t megelőző csúcs az egyik i-ből j-be vezető legrövidebb úton.

Mielőtt az algoritmust ismertetnénk, állapodjunk meg néhány, szomszédsági mátrixokkal kapcsolatos, jelölésben. Először is a $G = (V, E)$ gráfnak általában n csúcsa lesz, azaz $n = |V|$. Másodszor a mátrixokat nagybetűvel, egyes elemeiket pedig alulindexelt kisbetűvel fogjuk jelölni, tehát például W és D elemei w_{ij}, d_{ij} lesznek. Bizonyos esetekben iterációk jelölésére a mátrixokat zárójelezett felsőindexszel látjuk el, úgy mint $D^{(m)} = (d_{ij}^{(m)})$. Végül egy adott A $n \times n$-es mátrix esetén *sorok-száma*[A] tartalmazza n értékét.

7.6.1. A Floyd-Warshall-algoritmus

A következőkben dinamikus programozási feladatként értelmezzük a legrövidebb utak keresését. Egy $G = (V, E)$ irányított gráfon a keletkező ún. ***Floyd-Warshall-algoritmus*** futási ideje $\theta(n^3)$. A bemenő gráfban megengedünk negatív élsúlyokat, negatív összsúlyú köröket azonban nem.

A legrövidebb utak szerkezete

Dinamikus programozásunk a legrövidebb utak „belső” csúcsait tekinti, ahol

egy egyszerű $p = \langle v_1, v_2, \ldots, v_l \rangle$ út ***belső*** csúcsa p minden v_1-től és v_l-től különböző csúcsa, azaz a $\{v_2, v_3, \ldots, v_{l-1}\}$ halmaz minden eleme.

A szerkezet jellemzése a következő észrevételen alapul. Legyen a G gráf csúcshalmaza $V = \{1, 2, \ldots, n\}$, és tekintsük valamely k-ra az $\{1, 2, \ldots, k\}$ részhalmazt. Legyen p a legrövidebb i-ből j-be vezető olyan út, melynek belső csúcsait az $\{1, 2, \ldots, k\}$ részhalmazból választhatjuk. (A p út egyszerű, hiszen G nem tartalmaz negatív összsúlyú köröket.) A Floyd-Warshall-algoritmus a p út és az olyan legrövidebb utak kapcsolatát alkalmazza, melynek belső csúcsait az $\{1, 2, \ldots, k-1\}$ részhalmazból választjuk. E kapcsolat két esetre osztható attól függően, hogy k belső csúcsa-e p-nek vagy sem:

- Ha k a p útnak nem belső csúcsa, akkor a p út minden belső csúcsa az $\{1, 2, \ldots, k-1\}$ halmaz eleme. Így a legrövidebb i-ből j-be vezető és belső csúcsként csak az $\{1, 2, \ldots, k-1\}$ halmaz elemeit használó út szintén legrövidebb út lesz, ha a belső csúcsok az $\{1, 2, \ldots, k\}$ halmazból kerülhetnek ki.

- Ha k belső csúcs a p úton, akkor felbontjuk a p utat két $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ útra. A p_1 egy olyan legrövidebb út i és k között, melynek belső csúcsai az $\{1, 2, \ldots, k\}$ halmaz elemei. Sőt k nem is lehet p_1 út belső csúcsa, így p_1 egyben olyan legrövidebb út is, melynek belső csúcsai $\{1, 2, \ldots, k-1\}$-beliek. Hasonlóképpen p_2 egy legrövidebb, belső csúcsként csak $\{1, 2, \ldots, k-1\}$-et használó k-ból j-be vezető út.

Az összes csúcspár közti legrövidebb utak rekurzív megadása

A fenti megfigyelés alapján egy rekurzióval adhatunk egyre javuló becsléseket a legrövidebb utak súlyára. Legyen $d_{ij}^{(k)}$ a legrövidebb olyan i-ből j-be vezető út hossza, melynek minden belső csúcsa az $\{1, 2, \ldots, k\}$ halmaz eleme. A $k = 0$ érték esetén egy olyan i-ből j-be vezető útnak, melyen a belső csúcsok sorszáma legfeljebb 0 lehet, egyáltalán nem lehet belső csúcsa. Egy ilyen útnak tehát legfeljebb egyetlen éle lehet és így $d_{ij}^{(0)} = w_{ij}$. A további értékeket a következő rekurzió szolgáltatja:

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{ha } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right), & \text{ha } k \geq 1. \end{cases} \tag{7.2}$$

A végeredményt a $D^{(n)} = (d_{ij}^{(n)})$ mátrix tartalmazza, hiszen az egyes legrövidebb utak belső csúcsai a $\{1, 2, \ldots, n\}$ halmaz elemei, és így $d_{ij}^{(n)} = \delta(i, j)$

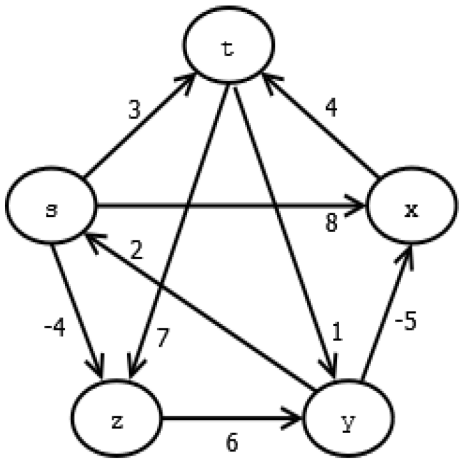
minden $i, j \in V$ esetén.

Az úthosszak kiszámítása alulról felfelé haladva

A (7.2) rekurzió segítségével a $d_{ij}^{(k)}$ értékeket alulról felfelé, k értéke szerinti növekvő sorrendben számíthatjuk. Az algoritmus bemenő adatai a (7.1) egyenlőség által definiált $Wn \times n$-es mátrix lesz, eredményül pedig a legrövidebb úthosszak $D^{(n)}$ mátrixát adja.

	Floyd-Warshall(W)	
1	$n \leftarrow$ sorok_száma$[W]$	$\Theta(n^3)$
2	$D^{(0)} \leftarrow W$	
3	**FOR** $k \leftarrow 1$ **TO** n **DO**	
5	**FOR** $i \leftarrow 1$ **TO** n **DO**	
6	**FOR** $j \leftarrow 1$ **TO** n **DO**	
7	$d_{ij}^{(k)} \leftarrow \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$	
8	**RETURN** $D^{(n)}$	

A Floyd-Warshall-algoritmus futási idejét a 3-6. sorok háromszorosan egymásba ágyazott **for** ciklusa határozza meg. A 6. sor minden egyes alkalommal $O(1)$ időben végrehajtható, így a teljes futási idő $\theta(n^3)$. A megadott programkód tömör és csak egyszerű adatszerkezeteket igényel. A θ-jelölésbeli állandó tehát kicsi, és a Floyd-Warshall-algoritmus még közepesen nagy méretű gráfok esetén is hatékony.



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \Pi^{(0)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \Pi^{(1)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \Pi^{(2)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

A $D^{(k)}$ és $\Pi^{(k)}$ mátrixok, ha a Floyd-Warshall-algoritmust az ábrán látható gráfon futtatjuk.

A legrövidebb utak megadása

A Floyd-Warshall-algoritmusban több módszer is kínálkozik arra, hogy a legrövidebb utak éleit is megkapjuk. Megtehető, hogy a legrövidebb úthosszak D mátrixának ismeretében $O(n^3)$ idő alatt megkonstruáljuk a Π megelőzési mátrixot. A Π megelőzési mátrix ismeretében pedig a MINDEN-PÁRHOZ-UTAT-NYOMTAT eljárás megadja a kívánt két csúcs közti legrövidebb út éleit.

	Minden-párhoz-utat-nyomtat(Π, i, j)
1	**IF** $i = j$
2	**THEN** PRINT i
3	**ELSE IF** $\pi_{ij} = NIL$
4	**THEN** PRINT i „-ből” j„-be nem vezet út”
5	**ELSE** Minden-párhoz-utat-nyomtat(Π, i, π_{ij})
6	PRINT j

A Π megelőzési mátrixot számíthajuk a Floyd-Warshall-algoritmus menetében a $D^{(k)}$ mátrixokkal egyidejűleg is. Pontosabban mátrixok egy $\Pi^{(0)},\Pi^{(1)},\ldots,\Pi^{(n)}$ sorozatát számíthatunk, melyre $\Pi=\Pi^{(n)}$. E sorozatban $\pi_{ij}^{(k)}$-t úgy definiáljuk, mint egy olyan legrövidebb i-ből j-be vezető úton a j-t megelőző csúcsot, melynek belső csúcsai a $\{1, 2, \ldots, k\}$ halmaz elemei.

Megadjuk a $\pi_{ij}^{(k)}$ értékeit definiáló rekurziót. (A $\Pi^{(k)}$ mátrixok számítása az ábra példájában követhető.) Ha $k = 0$, akkor i-től j-ig tartó úton nincs közbülső csúcs és így

$$\pi_{ij}^{(0)} = \begin{cases} NIL, & \text{ha } i = j \text{ vagy } w_{ij} = \infty, \\ i, & \text{ha } i \neq j \text{ és } w_{ij} < \infty. \end{cases}$$

A $k \geq 1$ esetben pedig egy úton a j-t megelőző csúcsnak választható ugyanaz a csúcs, amely j-t egy legrövidebb k-ból induló és belső csúcsként csak az $\{1, 2, \ldots, k-1\}$ halmaz elemeit használó úton előzi meg. Ha pedig a legrövidebb i-ből j-be vezető és az $\{1, 2, \ldots, k\}$ halmaz elemeit használó út k-t nem tartalmazza, akkor a j-t megelőző csúcs megegyezik a $k-1$ érték esetén választott megelőző csúccsal. Tehát $k \geq 1$ mellett a rekurzív egyenlet

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \text{ha } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)}, & \text{ha } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

7.7. Gráfok tranzitív lezártja

7.32. definíció. A G gráf tranzitív lezártja az a $G^* = (V, E^*)$ gráf, melyre $E^* = \{(i,j) \ : \$ létezik G -ben i-ből j-be út$\}$.

	TRANZITIV_LEZÁRT(G) $\Theta(n^3)$
1	$n \leftarrow \mid V[G] \mid$
2	**FOR** $i \leftarrow 1$ **TO** n **DO**
3	**FOR** $j \leftarrow 1$ **TO** n **DO**
4	**IF** $i = j$ vagy $(i,j) \in E[G]$
5	**THEN** $t_{ij}^{(0)} \leftarrow 1$
6	**ELSE** $t_{ij}^{(0)} \leftarrow 0$
7	**FOR** $k \leftarrow 1$ **TO** n **DO**
8	**FOR** $i \leftarrow 1$ **TO** n **DO**
9	**FOR** $j \leftarrow 1$ **TO** n **DO**
10	$t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right)$
11	**RETURN**($T(n)$)

8. fejezet

Dinamikus programozás

A dinamikus programozás és az oszd meg és uralkodj elv közötti legfontosabb különbségeket mutatja a következő táblázat.

Oszd meg és uralkodj	**Dinamikus programozás**
Független részproblémákat oldunk meg	A részproblémák nem függetlenek (közösek), egyszer oldódnak meg, újabb felhasználásig tárolódnak.
A megoldásokat egyesítjük	Általában optimalizálásra használjuk, amikor sok megengedett megoldás van

A dinamikus programozás lépései:

1. Jellemezzük az optimális megoldás szerkezetét.
2. Rekurzív módon definiáljuk az optimális megoldás értékét.
3. Kiszámítjuk az optimális megoldás értékét alulról felfelé módon.
4. A kiszámított információk alapján megszerkesztjük az optimális megoldást.

8.1. definíció. Mátrixok szorzatát teljesen zárójelezettnek nevezzük, ha a szorzat vagy egyetlen mátrixból áll, vagy pedig két, zárójelbe tett teljesen zárójelezett mátrix szorzata

8.2. példa. Legyenek az A, B, C mátrixok méretei $2 \times 3, 3 \times 4$, és 4×5. Számítsuk ki a $D = ABC$ mátrixot.

Ekkor D mérete 2×5. A műveletszám (szorzások száma) két mátrix összeszorzásakor: pqr, ha a méretek $p \times q$ és $q \times r$.

Első módszer: $((AB)C)$, műveletszám $2 \cdot 3 \cdot 4 + 2 \cdot 4 \cdot 5 = 24 + 40 = 64$.

Második módszer: $(A(BC))$, műveletszám $3 \cdot 4 \cdot 5 + 2 \cdot 3 \cdot 5 = 60 + 30 = 90$.

Legyenek az összeszorzandó mátrixok: $A_1, A_2, \ldots, A_n$ és legyen az A_i mátrix mérete $p_{i-1} \cdot p_i$ $(i = 1, \ldots, n)$.

Legyen $P(n)$ az n mátrix zárójelezéseinek a száma.

$$P(n) = \begin{cases} 1, & ha \;\; n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k), & ha \;\; n \geq 2 \end{cases}$$

Innen:

$P(1) = 1$
$P(2) = P(1) \cdot P(1) = 1$
$P(3) = P(1) \cdot P(2) + P(2) \cdot P(1) = 1 \cdot 1 + 1 \cdot 1 = 2$
$P(4) = P(1) \cdot P(3) + P(2) \cdot P(2) + P(3) \cdot P(1) = 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 1 = 5$
$\ldots$

$P(n) = C_n - 1$, ahol $C_n = \dfrac{\binom{2n}{n}}{n+1}$ (Catalan számok, exponenciális a növekedésük).

Az optimális zárójelezés szerkezete:

Legyen $A_{i\ldots j} = A_i A_{i+1} \ldots A_j$. Az optimális eset az $A_1, A_2, \ldots, A_n$ szorzatot k-nál vágja szét $A_{1\ldots n} = A_{1\ldots k} \cdot A_{k+1\ldots n}$. Költség = $A_{1\ldots k}$ költsége + $A_{k+1\ldots n}$ költsége + az összeszorzás költsége. $A_{1\ldots k}$ és $A_{k+1\ldots n}$ zárójelezése is optimális kell legyen.

Legyen m_{ij} az $A_{i\ldots j}$ kiszámításának minimális költsége

$$m_{ij} = \begin{cases} 0, & \text{ha } i = j \\ \min\limits_{i \leq k < j} \{m_{ik} + m_{k+1,j} + p_{i-1}p_k p_j\} & \text{ha } i < j \end{cases}$$

Legyen s_{ij} az a k index, ahol az $A_{i\ldots j}$ szorzat ketté van vágva.

Legyen $p = (p_0, p_1, \ldots, p_n)$

	MÁTRIX_SZORZÁS_SORREND(p) $O(n^3)$
1	$n \leftarrow$ hossz[p]-1
2	**FOR** $i \leftarrow 1$ **TO** n **DO**
3	$\quad m_{ii} \leftarrow 0$
4	**FOR** $l \leftarrow 2$ **TO** n **DO**
5	$\quad$ **FOR** $i \leftarrow 1$ **TO** $n - l + 1$ **DO**
6	$\quad\quad j \leftarrow i + l - 1$
7	$\quad\quad m_{ij} \leftarrow \infty$
8	$\quad\quad$ **FOR** $k \leftarrow i$ **TO** $j - 1$ **DO**
9	$\quad\quad\quad q \leftarrow m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j$
10	$\quad\quad\quad$ **IF** $q < m_{ij}$
11	$\quad\quad\quad\quad$ **THEN** $m_{ij} \leftarrow q$
12	$\quad\quad\quad\quad\quad s_{ij} \leftarrow k$
13	**RETURN** (m, s)

$A_1 \cdot A_2 \cdot A_3 \cdot A_4$
(2x3) (3x4) (4x5) (5x6)

(1,2)	0+0+2·3·4	24
(2,3)	0+0+3·4·5	60
(3,4)	0+0+4·5·6	120
(1,3)	0+60+2·3·5	90
	24+0+2·4·5	64
(2,4)	0+120+3·4·6	192
	60+0+3·5·6	150
(1,4)	0+150+2·3·6	186
	24+120+2·4·6	192
	64+0+2·5·6	124

$((A_1 \cdot A_2) \cdot A_3) \cdot A_4$

m		1	2	3	4
	2	3	4	5	6
1	3	0	24	64	124
2	4		0	60	150
3	5			0	120
4	6				0

s		1	2	3	4
1			1	2	3
2				2	3
3					3
4					

	MÁTRIX_LÁNC_SZORZÁS(A, s, i, j)
1	**IF** $j > i$
2	**THEN** $X \leftarrow$ MÁTRIX_LÁNC_SZORZÁS(A, s, i, s_{ij})
3	$Y \leftarrow$ MÁTRIX_LÁNC_SZORZÁS$(A, s, s_{ij} + 1, j)$
4	**RETURN** (MÁTRIXSZORZÁS(X, Y))
5	**ELSE RETURN**(A_i)

9. fejezet

NP-teljesség

A következőkben vázlatosan betekintünk az algoritmusok bonyolultság elméletébe. Az előző fejezetekben tárgyalt algoritmusok általában legrosszabb esetben polinomiális idejűek voltak. Vajon minden probléma megoldására adható polinomiális idejű algoritmus? Sajnos a válasz az, hogy nem. A polinomiális időben megoldható problémákat tekintjük jól kezelhetőknek. Ezen problémák zárt osztályt képeznek, azaz, ha az egyikük eredményét egy másik ilyen probléma bemenetére adjuk, akkor az összetett probléma is polinomiális idejű lesz.

9.1. definíció. Az absztrakt probléma egy kétváltozós reláció a probléma eseteinek (bemeneteinek) I és a probléma megoldásainak S halmazán.

Így egy inputhoz több output is tartozhat.

9.2. definíció. Döntési (eldöntési) probléma az olyan absztrakt probléma, melynek a megoldása "igen", vagy "nem".

Ez azt jelenti, hogy a döntési probléma egy $I \to \{0;1\}$ leképezés, ahol a nulla szimbolizálja a"nem"-et, az 1 az "igen"-t. Optimalizálási problémák például átalakíthatók döntésivé azon átfogalmazással, hogy az optimum kisebb-e egy előre megadott értéknél.

9.3. definíció. Egy absztrakt objektumokból álló S halmaz kódolása egy e leképezés S-ről a bináris sorozatokra.

Az algoritmusok a probléma eseteinek kódolását kapják inputként.

9.4. definíció. Konkrét az absztrakt probléma, ha esetei a bináris sorozatok.

9.5. definíció. Egy algoritmus egy konkrét problémát $O(T(n))$ idő alatt megold, ha minden n hosszúságú i esetre a megoldás $O(T(n))$ lépést igényel.

9.6. definíció. Egy konkrét probléma polinomiális időben megoldható, ha létezik olyan algoritmus, amely $O(n^k)$ idő alatt megoldja valamely $k \in R, k \geq 0$ számra.

9.7. definíció. P bonyolultsági osztálynak nevezzük a polinomiális időben megoldható problémák halmazát.

9.8. definíció. Azt mondjuk, hogy egy $f : \{0 > 1\}^* \rightarrow \{0 > 1\}^*$ függvény polinomiális időben kiszámítható, ha létezik olyan A polinomiális algoritmus, amely tetszőleges $x \in \{0 > 1\}^*$ bemenetre az *f(x)*-et adja eredményül.

9.9. definíció. Azt mondjuk, hogy az e_1 és e_2 kódolások polinomiálisan kapcsoltak, ha léteznek olyan $f(x_{12})$ és $f(x_{21})$ polinomiális időben kiszámítható függvények, hogy bármely $i \in I$ esetre $f(x_{12})(e_1(i)) = e_2(i)$ és $f(x_{21})(e_2(i)) = e_1(i)$.

Polinomiálisan kapcsolt kódolások esetén mindegy, hogy melyiket használjuk a probléma polinomiális időben való megoldhatóságának az eldöntésére. Erős kapcsolat áll fenn a döntési problémák és a formális nyelvek között. A formális nyelvek valamely szimbólumok véges halmazát, amit ábécének neveznek,

használják fel a formális nyelv megadására. Az ábécé elemeiből véges sorozatokat (szavakat) képezve, a nyelvet az így képezhető összes véges sorozatok halmazának részhalmazaként adják meg. Ha az ábécét a Σ jelöli, akkor a sorozatokat a Σ^*. Ha az ábécé a zérus és az egy jel, akkor a sorozatok halmaza az összes véges bináris jelsorozat, amihez hozzávesszük még az üres szót, amit ε-nal jelölünk. Mivel a nyelv eszerint egy halmaz (részhalmaz), ezért érvényesek rá és általában a formális nyelvekre a halmazelméleti műveletek, mint például az unió, a metszet, a komplementer képzés. További művelet a konkatenáció, amely a benne szereplő nyelvek szavainak egymás után írását, összekapcsolását jelenti. Jelölésben az L_1 és L_2 nyelvek konkatenáltja L_1L_2. Az L nyelv lezártjának nevezik az $L^* = \{\varepsilon\} \cup L \cup L^2 \cup \ldots$ nyelvet. A döntési problémák esetén a döntési probléma esethalmaza Σ^*, a bináris jelsorozatok halamaza. Magát a döntési problémát pedig ezen halmazból az a nyelv jelöli ki, amely azon sorozatokból áll, amelyre a probléma egyet ad. Ha Q a probléma, akkor a neki megfelelő nyelv. $L = \{x \in \Sigma^* : Q(x) = 1\}$.

9.10. definíció. Azt mondjuk, hogy az A algoritmus elfogadja az x szót, ha $A(x) = 1$, és elutasitja azt, ha $A(x) = 0$.

9.11. definíció. Azt mondjuk, hogy az A algoritmus elfogadja az L nyelvet, ha a nyelv minden szavát elfogadja.

Ez nem jelenti azt, hogy a nyelvhez nem tartozó szavak közül ne fogadna el egyet sem.

9.12. definíció. Azt mondjuk, hogy az A algoritmus eldönti az L nyelvet, ha a nyelv minden szavát elfogadja, a többit pedig elutasítja.

9.13. definíció. Azt mondjuk, hogy az A algoritmus polinomiális időben elfogadja az L nyelvet, ha a bármely n hosszúságú $x \in L$ szót $O(n^k)$ időben elfogad valamely k konstansra.

9.14. definíció. Azt mondjuk, hogy az A algoritmus polinomiális időben eldönti az L nyelvet, ha a bármely n hosszúságú x szót $O(n^k)$ időben elfogad, ha a szó L-beli, vagy elutasít, ha nem L-beli. Itt k konstans.

9.15. definíció. Azt mondjuk, hogy az L nyelv polinomiális időben elfogadható/eldönthető, ha létezik algoritmus, amely polinomiális időben elfogadja/eldönti.

A P bonyolultsági osztály azon nyelvek halmaza a $\{0;1\}$ fölött, amelyek polinomiális időben elfogadhatók. Könnyebb általában egy probléma megoldását ellenőrizni, mint azt megtalálni. Az ellenőrzés egfajta tanúsítvány, ami a megoldás helyességét bizonyítja.

9.16. definíció. Ellenörző algoritmusnak nevezzük az olyan kétbemenetű algoritmust, amelynek egyik bemenete a döntési probléma bemenete, a másik pedig, amit tanúnak neveznek, egy bináris szó.

9.17. definíció. Azt mondjuk, hogy az A ellenörző algoritmus bizonyítja az x szót, ha létezik olyan y tanú, hogy $A(x,y) = 1$, és A bizonyítja az L nyelvet, ha bizonyítja L minden szavát és minden szó, amit bizonyít L-ben van.

9.18. definíció. Azon nyelvek osztályát, amelyek polinomiális időben bizonyíthatók, NP bonyolultsági osztálynak nevezzük.

Az NP megnevezés a "nondeterministic polinomial (time)" rövidítése. Annyi bizonyos, hogy $P \subseteq NP$, hiszen az ellenörző algoritmus lehet maga az $L \in P$ nyelv polinomiális idejű eldöntő algoritmusa azáltal, hogy az y bemenetet ignorálja. Azt nem tudjuk, hogy $P \subset NP$, vagy $P = NP$ áll-e fönn, bár intuitíve az első látszik igaznak. A válasz egyelőre nem ismeretes. NP probléma például a Hamilton-kör probléma, amely azt óhajtja eldönteni, hogy van-e egy irányítatlan gráfban Hamilton-kör, azaz olyan kör, amely a gráf mindegyik csomópontján csak egyszer megy át. A problémák bonyolultságának összehasonlítását könnyíti meg a visszavezethetőségi elv. Ez az elv a problémát átfogalmazza egy másik problémává, olyanná, amelynek megoldásából már következik az eredeti probléma megoldása.

9.19. definíció. Azt mondjuk, hogy az L_1 nyelv polinomiálisan visszavezethető az L_2 nyelvre (jelölése $L_1 \leq_p L_2$), ha létezik olyan $f : \{0;1\}^* \to \{0;1\}^*$ polinom időben kiszámítható függvény, hogy minden $x \in \{0;1\}^*$ esetén $x \in L_1 \Leftrightarrow f(x) \in L_2$.

Az f függvény neve visszavezető függvény, a kiszámító algoritmusáé pedig visszavezető algoritmus. Teljesül az állítás, miszerint, ha egy probléma visszavezethető polinomiálisan egy másikra és a másik P-beli, akkor az eredeti is P-beli volt.

9.20. definíció. Azt mondjuk, az $L \subseteq \{0;1\}^*$ nyelv NP-teljes, ha teljesül az következő két feltétel

1. $L \in NP$
2. Minden $L' \in NP$-re $L' \leq_p L$.

Az NP-teljes nyelvek halmazát NPC-vel jelöljük. Ha egy nyelv a második tulajdonságot teljesíti, de az elsőt nem, akkor NP nehéznek nevezzük.

9.21. tétel.
Ha létezik polinomiális időben megoldható NP-teljes probléma, akkor P=NP, azaz ha létezik NP-ben polinomiális időben nem megoldható probléma, akkor egyetlen NP-teljes probléma sem polinomiális.

A tétel nyílvánvaló, mert ha van olyan L nyelv, amelyre $L \in P \cap NPC$, akkor az NP-teljesség 2. pontja alapján bármely $L' \in NP$ esetén $L' \leq_p L$ és emiatt akkor $L' \in P \cap NPC$ is fennáll. A jelenlegi állapotok alapján kicsi az esélye, hogy valaki előáll egy NP-probléma polinomiális megoldásával, ami azt jelentené, hogy $P = NP$. egy probléma NP-teljessége arra utal, hogy a probléma nehezen kezelhető. Végül néhány NP-teljes probléma következzen.

- Irányítatlan gráf Hamilton-köre. (HAM probléma.)
- Boole-hálózat kielégíthetőségi problémája, azaz egy logikai kapukból (ÉS, VAGY, NEM) álló hálózatnak van-e olyan bemenete, amelyre 1-et ad a kimeneten. (C-SAT probléma.) Ugyanis, ha ilyen nem lenne, akkor helyettesíthető lenne egy konstans nullát adó elemmel, megtakarítva ezzel a bonyolult felépítést.
- A 3-SAT probléma. A pontosan három elemű zárójeleket tartalmazó konjunktív normálformák kielégíthetőségének problémája.

- Az irányítatlan gráf klikk-problémája, azaz létezik-e a gráfban k-méretű klikk, olyan részgráf, amelyben minden csúcspár szomszédos.

- A minimális lefedő csúcshalmaz probléma. Irányítatlan gráf lefedő csúcshalmaza a csúcsainak olyan részhalmaza, hogy minden élre az él két végpontján lévő csúcsok egyike, vagy mindkettő ezen csúcshalmazban van. A probléma a minimális lefedő csúcshalmaz méretének megkeresése.

- A részletösszeg probléma. Adott a természetes számok egy S véges részhalmaza. Egy adott t természetes számra létezik-e olyan $S' \subseteq S$ halmaz, melyben a számok összege pontosan a t szám.

- Az utazó ügynök probléma. Adott n város, amelyet az ügynöknek tetszőleges sorrendben végig kell látogatnia. Ismert bármely két város között a közvetlen utazási költség, egész szám, ami lehet akár negatív is. Megkeresendő a minimális költségű körutazás.

Egy NP-probléma esetén nagy méreteknél kevés az esély polinomiális algoritmus megtalálására. (Eddig még nem sikerült.) Ilyenkor célszerű közelítő megoldást adó algoritmusokkal foglalkozni.

10. fejezet

Mellékletek

10.1. Az ASCII karakterkészlet

10.1.1. Vezérlő jelek

Hexa	Decimális	Karakter	Jelentés
00	0	NUL	NULL
01	1	SOH	Start of heading
02	2	STX	Start of text
03	3	ETX	End of text
04	4	EOT	End of transmission
05	5	ENQ	Enquiry
06	6	ACK	Acknowledgement
07	7	BEL	Bell
08	8	BS	Backspace
09	9	HT	Horizontal tab
0A	10	LF	Line feed
0B	11	VT	Vertical tab
0C	12	FF	Form feed
0D	13	CR	Carriage return
0E	14	SO	Shift out
0F	15	SI	Shift in

Hexa	**Decimális**	**Karakter**	**Jelentés**
10	16	DLE	Data link escape
11	17	DC1	Device control 1
12	18	DC2	Device control 2
13	19	DC3	Device control 3
14	20	DC4	Device control 4
15	21	NAK	Negative acknowledgement
16	22	SZN	Synchronous idle
17	23	ETB	End of transmission block
18	24	CAN	Cancel
19	25	EM	End of medium
1A	26	SUB	Substitute
1B	27	ESC	Escape
1C	28	FS	File separator
1D	29	GS	Group separator
1E	30	RS	Record separator
1F	31	US	Unit separator
7F	127	DEL	Delete

10.1.2. Nyomtatható karakterek

Hexa	Dec	Karakter	Hexa	Dec	Karakter	Hexa	Dec	Karakter
20	32	Space	40	64	@	60	96	‘
21	33	!	41	65	A	61	97	a
22	34	"	42	66	B	62	98	b
23	35	#	43	67	C	63	99	c
24	36	$	44	68	D	64	100	d
25	37	%	45	69	E	65	101	e
26	38	&	46	70	F	66	102	f
27	39	’	47	71	G	67	103	g
28	40	(	48	72	H	68	104	h
29	41	)	49	73	I	69	105	i
2A	42	*	4A	74	J	6A	106	j
2B	43	+	4B	75	K	6B	107	k
2C	44	,	4C	76	L	6C	108	l
2D	45	-	4D	77	M	6D	109	m
2E	46	.	4E	78	N	6E	110	n
2F	47	/	4F	79	O	6F	111	o
30	48	0	50	80	P	70	112	p
31	49	1	51	81	Q	71	113	q
32	50	2	52	82	R	72	114	r
33	51	3	53	83	S	73	115	s
34	52	4	54	84	T	74	116	t
35	53	5	55	85	U	75	117	u
36	54	6	56	86	V	76	118	v
37	55	7	57	87	W	77	119	w
38	56	8	58	88	X	78	120	x
39	57	9	59	89	Y	79	121	y
3A	58	:	5A	90	Z	7A	122	z
3B	59	;	5B	91	[	7B	123	{
3C	60	<	5C	92	\	7C	124	|
3D	61	=	5D	93	]	7D	125	}
3E	62	>	5E	94	^	7E	126	~
3F	63	?	5F	95	_			

Irodalomjegyzék

[1] *Magyar értelmező kisszótár*, Akadémiai Kiadó, Budapest, 1975.

[2] Sain Márton: *Nincs királyi út!* Matematikatörténet. Gondolat Kiadó Budapest, 1986.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest: *Algoritusok*, Műszaki Könyvkiadó, Budapest, 1999.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein : *Új algoritusok*, Scolar Kiadó, Budapest, 2003.

[5] Ivanyos G., Szabó R. Rónyai L: *Algoritmusok*, TYPOTEX Kiadó KFT., 2008.

[6] Iványi A. (alkotó szerkesztő): *Informatikai Algoritmusok I.*, ELTE Eötvös Kiadó, Budapest, 2004.

[7] Iványi A. (alkotó szerkesztő): *Informatikai Algoritmusok II.*, ELTE Eötvös Kiadó, Budapest, 2005.

[8] D. E. Knuth: *A számítógép-programozás művészete*, AnTonCom Infokommunikációs Kft., 2009.