1.1. 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]. Ez a fajta magyarázat azonban egy kissé homályos, mindenki értse, ahogy tudja.

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.

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

Tehát egy halmazról van szó a rajta értelmezett műveletekkel együtt. Absztrakt adattípusra példa lehet a korábbi tanulmányokból ismert logikai érték, természetes szám, egész szám, racionális szám, valós szám, de ide tartozik a komplex szám, sorozat, halmaz, dinamikus halmaz is. A logikai érték esetén például csak az a fontos, hogy kétféle értéke lehet: az *igaz* érték vagy a *hamis* érték. Az már mellékes, hogy ezt *i* vagy *h* betűvel vagy a TRUE vagy FALSE szavakkal írjuk le, esetleg hogy a számítógép az 1 vagy 0 bitekkel ábrázolja. Ide tartoznak persze a logikai értékek között végezhető műveletek elsősorban a negáció (a tagadás művelete), konjunkció (az ÉS művelet) és a diszjunkció (a VAGY művelet). Ravaszabb dolog az egész szám esete, mivel 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!)

1. 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.

Definíció: Adatstruktúra
*Adatstruktúrá*nak nevezzük az absztrakt adattípus konkrét megjelenési formáját.

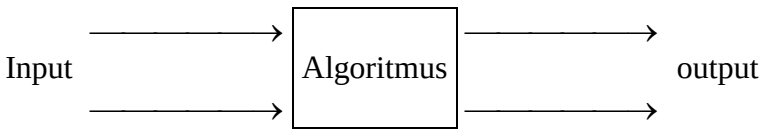
A műveletek az adatstruktúrához emiatt ugyanúgy hozzátartoznak, mint az absztrakt adattípushoz. Adatstruktúrára példa lehet a lista, verem, sor, kupac, elsőbbségi (prioritásos) sor, fa, gráf, hálózat, binomiális kupac, stb., de adatstruktúra a számábrázolás módja is. Az elnevezésekben azonban az absztrakt adattípus és az adatstruktúra nevek gyakran keverednek, hiszen tartalmilag hasonló dolgokról van szó.

1.2. 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.

Definíció: Az algoritmus (csak heurisztikus definíció, nem tudományos)
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.



1. Példa: Algoritmus: 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:

$$(10a+b)^2 = 100a^2 + 20ab + b^2 = 100a^2 + (2 \cdot 10a + b) \cdot b \qquad (1)$$

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.

(Hogyan következik az (1) formulából az éppen lejegyzett algoritmus?)

2. 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 (próba-jegy)	duplázás első lépést kivéve	szorzat	maradék
1	14	**14**	3	$3^2=9$	-	14-9=5
2	42	5**42**	7	$2 \cdot 3=6$	6**7** $\cdot$ 7=469	542-469=73
3	48	73**48**	9	$2 \cdot 37=74$	74**9** $\cdot$ 9=6741	7348-6741=607
4	04	6070**4**	8	$2 \cdot 379=758$	758**8** $\cdot$ 8=60704	60704-60704=0

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

3. 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 (próba-jegy)	duplázás	szorzat	maradék

1	2	2	1	$1^2=1$	-	2-1=1
2	00	1**00**	4	2·1=2	2**4**·4=96	100-96=4
3	00	4**00**	1	2·14=28	28**1**·1=281	400-281=119
4	00	119**00**	4	2·141=282	282**4**·4=11296	11900-11296=604
5	00	6040**0**	2	2·1414=2828	2828**2**·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. (Miért? Próbáljuk ki!) 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 előírás*nak 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 $x_{k+1}=\frac{1}{2}\cdot\left(x_k+\frac{s}{x_k}\right)$ számot és *k* értékét eggyel növeljük.
3. Ha $|x_k-x_{k-1}|<\varepsilon$, akkor megállunk és $x\approx x_k$, egyébként pedig folytatjuk a 2-es pontnál.

4. Példa: Példaként határozzuk meg az $x=\sqrt{14424804}$ értékét négy tizedesjegyre, azaz legyen $\varepsilon=0{,}0001$! Heurisztikus meggondolásból válasszuk kezdőértéknek az $x_0=2048$ számot! (Mi lehet ezen heurisztika hátterében, amely alapján ez a kezdő közelítés elég jónak tekinthető? Végezzük el a számítást az $x_0=1$ értékre is!)

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

4	3798,000032	$=\frac{1}{2}\cdot\left(3798{,}489832+\frac{14424804}{3798{,}489832}\right)$

Számítsuk ki az $x=\sqrt{2}$ értékét négy tizedes jegyre, 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	$=\frac{1}{2}\cdot\left(1{,}5+\frac{2}{1{,}5}\right)$
2	1,414215686	$=\frac{1}{2}\cdot\left(1{,}416666667+\frac{2}{1{,}416666667}\right)$
3	1,414213562	$=\frac{1}{2}\cdot\left(1{,}414215686+\frac{2}{1{,}414215686}\right)$

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.
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 ← 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...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.

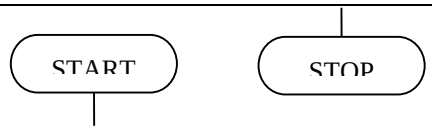
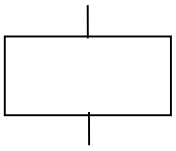
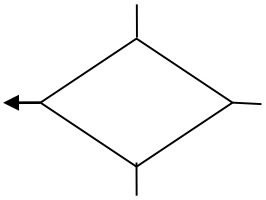
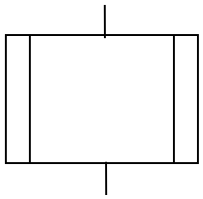
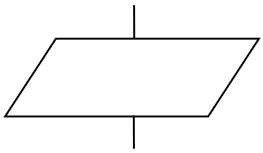
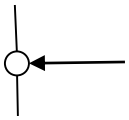
A fenti iteratív négyzetgyökvonási algoritmus pédául így nézhetne ki. A jobboldalon egy praktikusabb változat látható.

	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	$k \leftarrow k + 1$
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	$z \leftarrow 1$
3	// Input: s – nemnegatív szám
4	// Output: z - nemnegatív szám
5	REPEAT $x \leftarrow z$
6	$z \leftarrow (x + s / x) / 2$
7	UNTIL $\lvert z - x \rvert < \varepsilon$
8	RETURN (z)

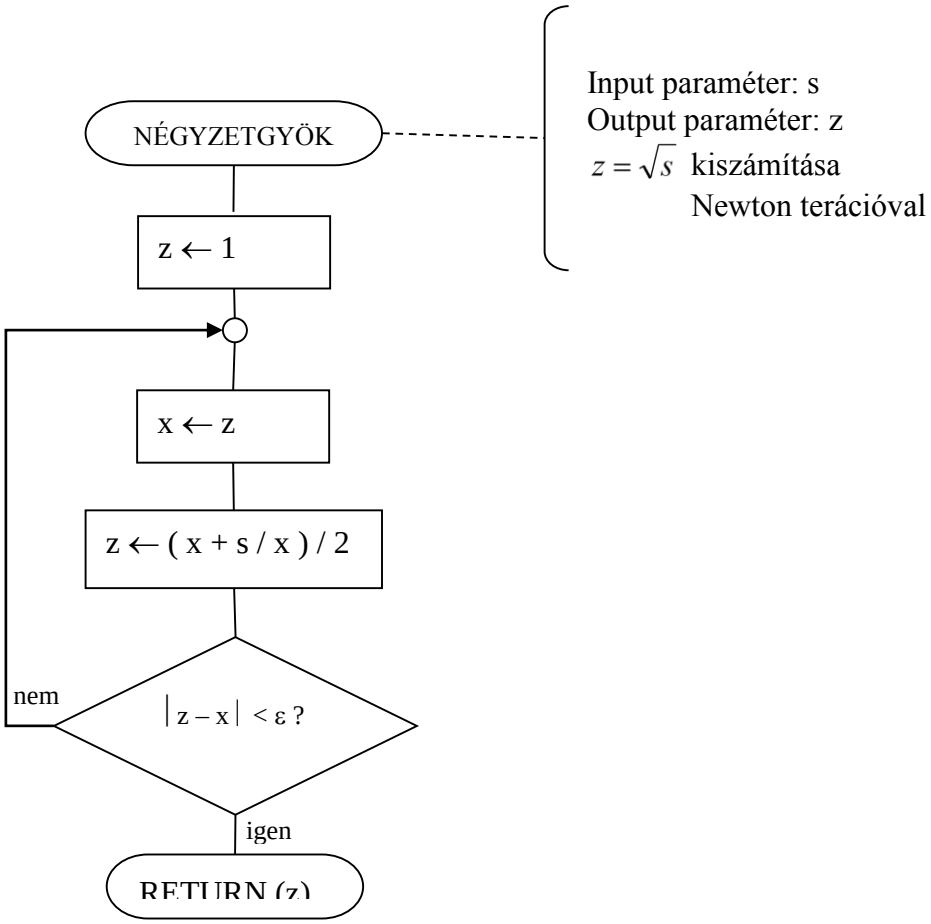
A folyamatábra az algoritmust folyamatában a sík kétdimenziós tulajdonságát kihasználva grafikus szimbólumok felhasználásával teszi szemléletessé. Az alábbi, vízszintes vagy függőleges folyamatvonalak révén egymáshoz kapcsolható szimbólumokat használjuk: A folyamatvonalak összefutását kis körökkel jelöljük.

Szimbólum	**Szimbólum magyarázata**

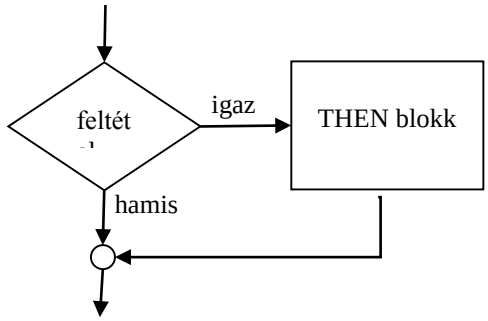
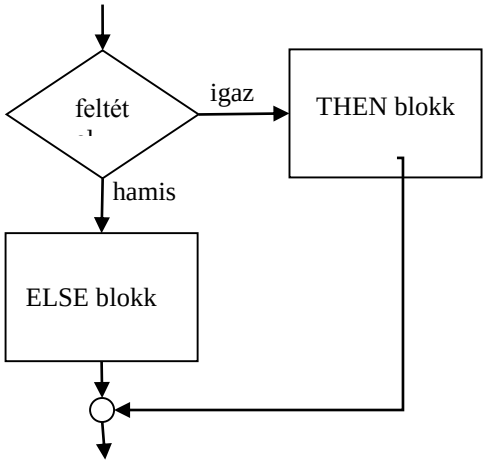
	Kezdőszimbólum (az algoritmus kezdete, pontosan egy van) és a befejező szimbólum (az algoritmus megállási helye, legalább egy van)
	Tevékenység
	Döntés a szimbólumba írt feltétel milyensége alapján
	Alprogram, procedúra
	Input – output
	Kapcsoló szimbólumok az ábra távolabbi részeinek összekapcsolására. A körökbe írt jel, - általában szám – mutatja az összetartozó szimbólumokat.
	Folyamatvonalak találkozása, összefutása

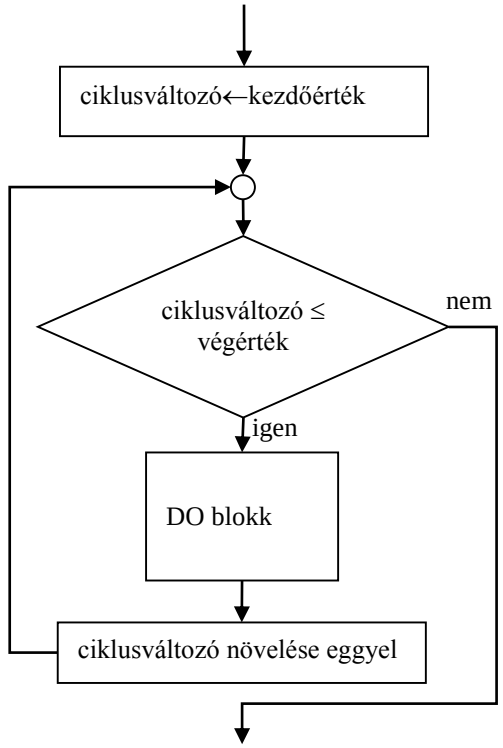
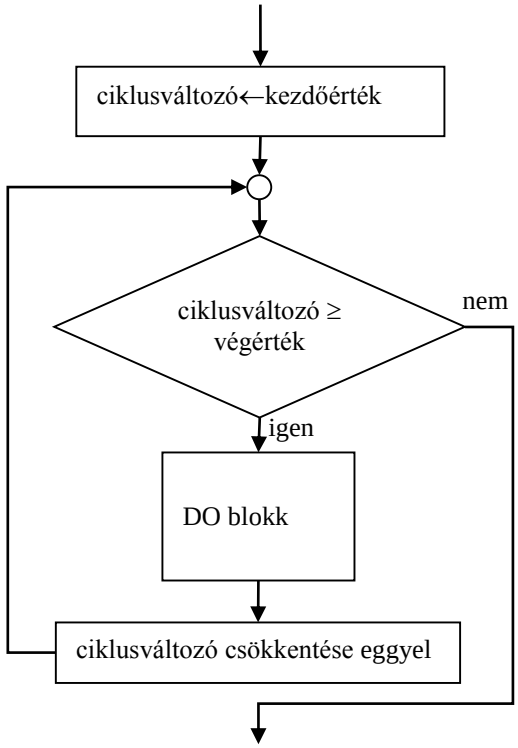
A folyamatvonalakon a haladás iránya balról-jobbra, vagy fentről-lefelé, hacsak a vonalra kitett nyíl másként nem mutat. Megjegyzést a szimbólumokhoz szaggatott vonallal a szimbólumhoz kapcsolt, megfelelően méretezett kezdő szögletes zárójel jellel lehet hozzákapcsolni. A szöveg a szögletes zárójel mögé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

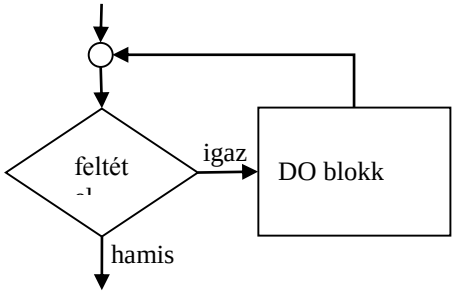
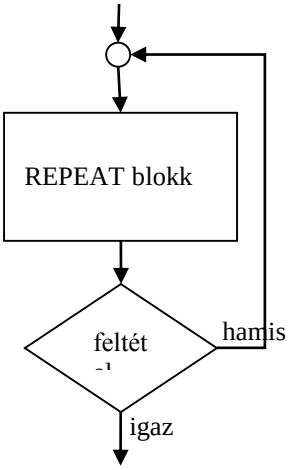
A négyzetgyökvonás fenti praktikusabb változata folyamatábrával:


NÉGYZETGYÖK
Input paraméter: s
Output paraméter: z
$z = \sqrt{s}$ kiszámítása
Newton terációval
z ← 1
x ← z
z ← (x + s / x) / 2
| z – x | < ε ?
nem
igen
RETURN (z)

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 0. fejezetben leírt *Summa* algoritmus. és az 1.2 fejezetben leírt kézi négyzetgyökvonás algoritmusa. Ugyancsak iteratív algoritmusok az 1.2.1 és 1.2.2 algoritmusok. Rekurzív algoritmus volt a 0. fejezetbeli *RekurzívSumma* és a *RekSum* algoritmus. A rekurzív algoritmusok mindig átírhatók iteratív formára is. A 0. fejezetbeli említett algoritmusoknak a pszeudokódját az alábbi módon készíthetjük el procedúra formában

```
1  Summa ( n, s )
2  s ← 0
3  FOR   k ← 1 TO n
          DO
4         s ← s + k
5  RETURN(s)
```

```
1  RekurzívSumma ( n, s )
2  IF   n=1
3       THEN     s ← 1
4       ELSE     s ← RekurzívSumma ( n-1, u )
5                s ← u+n
6  RETURN ( s )
```

```
1  RekSum ( m , n, s )
2  IF   m=n
3       THEN   s ← m
4       ELSE   RekSum ( m , ⌊(m+n)/2⌋, u)
6              RekSum ( ⌊(m+n)/2⌋+1 , n, v)
6              s ← u+v
7  RETURN ( s )
```

(Írjuk át a *RekurzívSumma* és a *RekSum* rekurzív algoritmusokat iteratívra!)
(Írjuk át az 1.2.2. algoritmust rekurzív algoritmusra!)

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 stratégiát* hármat említünk a könyvben:

1. *Oszd meg és uralkodj* — Ez a stratégia a kiinduló problémát kisebb méretű, független, hasonló részproblémákra bontja, amelyeket rekurzívan 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. Ez 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.

1.3. 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*
   Ez 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.

4. Példa: A gyökvonás 1.2.2. algoritmusa 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.

1.4. 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.

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 *méret*ének 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.

1. 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 *időigény*ét (*t*-time, idő) és $s_A(x)$-szel a *tárigény*ét (*s*-storage, tár).

Definíció: Az algoritmus időbonyolultsága
A $T_A(n) = \sup_{\substack{x \in D \\ |x| \le n}} t_A(x)$ számot az *A* algoritmus *időbonyolultság*ának nevezzük.

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

Definíció: Az algoritmus tárkapacitás bonyolultsága
Az $S_A(n) = \sup_{\substack{x \in D \\ |x| \le n}} s_A(x)$ számot az *A* algoritmus *tárkapacitás bonyolultság*ának 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 (Miért?) 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.

1.5. 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 függvény*eknek (röviden növekedési függvény) nevezzük.

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 $f,g : \boldsymbol{N} \rightarrow \boldsymbol{Z}$ két növekedést leíró függvény.

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

1. *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 $\exists\, c>0$ és $n_0>0$, hogy ha $n \geq n_0$, akkor $0 \leq f(n) \leq c \cdot g(n)$

2. *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 $\exists\, c>0$ és $n_0>0$, hogy ha $n \geq n_0$, akkor $0 \leq c \cdot g(n) \leq f(n)$

3. *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 $\exists\, c_1,c_2>0$ és $n_0>0$, hogy ha $n \geq n_0$, akkor
   $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

4 *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 $\forall c>0$-ra $\exists\, n_0>0$, hogy ha $n \geq n_0$, akkor $0 \leq f(n) < c \cdot g(n)$

5. *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 $\forall c>0$-ra $\exists\, n_0>0$, hogy ha $n \geq n_0$, akkor $0 \leq c \cdot g(n) < f(n)$

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 közelségéről, 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	$f(n) = \Theta(1)$
Logaritmikus	$f(n) = \Theta(\log n)$
Lineáris	$f(n) = \Theta(n)$
Négyzetes	$f(n) = \Theta(n^2)$

Köbös	$f(n) = \Theta(n^3)$
Polinomiális	$f(n) = \Theta(n^k)$, $k \in R$ és $k>0$
Exponenciális	$f(n) = \Theta(a^n)$, $a>1$

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 $f(n) = \Omega(n^{p+\varepsilon})$.

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 $f(n) = O(n^{p-\varepsilon})$.

1.6. 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.

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.

1. 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 $T(n)=q^{n-1}\cdot T(1)$, ami $T(1)$ ismeretében egyértelműen meghatározott.

2. 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.

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

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).

1. Tétel: A mester tétel

Legyenek $a\geq 1$, $b>1$ konstansok, $p=\log_b a$, $f: N\to Z$ függvény.

Definiálunk egy $g(n)=n^p$ úgynevezett tesztpolinomot.

Legyen a rekurziós összefüggésünk: $T(n)=a\cdot T(n/b)+f(n)$. (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
   $\boxed{T(n)=\Theta(g(n))}$
2. Ha $f(n)=\Theta(g(n))$, akkor $\boxed{T(n)=\Theta(g(n)\cdot\log n)}$
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 $n\geq n_0$ esetén $a\cdot f(n/b)\leq c\cdot f(n)$, akkor $\boxed{T(n)=\Theta(f(n))}$.

FELADATOK

1. Egy – ma még nem létező - szuper kvantumszámítógép egy gépi utasítást annyi idő alatt hajt végre, amennyi idő alatt a fény megtesz egy hidrogén atommag átmérőjének megfelelő távolságot, amely idő 10^{-24} másodperc. Tegyük fel, hogy az algoritmusunk időbonyolultsága az alábbi táblázatban megadott $f(n)$ érték, ahol ez a gépi kódú műveletek száma az input méretének (n) függvényében. Határozzuk meg, hogy az egyes időbonyolultságok és időtartamok esetén mennyi a maximális problémaméret, amely még a gépünkkel megoldható elvileg. Töltsük ki a táblázatot!

idő / $f(n)$	1 mp	1 perc	1 óra	1 nap	1 hónap (30 nap)	1 év (12 hónap)	1 évszázad
$\log n$							
$\sqrt{n}$							
n							
$n \cdot \log n$							
n^2							
n^3							
2^n							
$n!$							

2. Adott az előző szuper kvantumszámítógép, melyen egy probléma megoldási algoritmusa $f_{sq}(n)$ időbonyolultságú, és van egy másik egyszerű számítógép, melyen az egy gépi kódú utasítás végrehajtási ideje a könnyebbség kedvéért 10^{-6} másodperc és rajta ugyanazon problémának egy másik megoldási algoritmusa van beprogramozva, melynek időbonyolultsága $f_e(n)$. Határozzuk meg az alábbi táblázat szerinti esetekre, hogy milyen input méretadatok esetén melyik gépet (a szuper kvantum – „*sq*”, vagy az egyszerű – „*e*”) érdemesebb a probléma megoldására használni! Töltsük ki a táblázatot!

Probléma sorszáma	$f_{sq}(n)$	probléma méret „*sq*”	$f_e(n)$	probléma méret „*e*”
1.	$n!$		$10^6 n^2$	
2.	n^3		$10^8 \log n$	
3.	2^n		$10^6 \sqrt{n}$	
4.	$n!$		$\log n$	
5	n^3		$10^8 n^2$	

3. Bizonyítsuk be, hogy:
   a. $\frac{n^2}{2} - 3n = \Theta(n^2)$
   b. $6n^3 \neq \Theta(n^2)$
   c. $2^{n+1} = O(2^n)$

d. $2^{2n} \neq O\left(2^n\right)$

e.

4. Oldjuk meg az alábbi rekurzív egyenleteket!

   a. $T(n) = 2T\left(\frac{n}{2}\right) + n, \quad T(1) = 1$

   b. $T(n) = 2T\left(\sqrt{n}\right) + \log n, \quad T(2) = 1$

   c. $T(n) = T(n-1) + n, \quad T(1) = 1$ (Rekurzív visszahelyettesítéssel.)

5. Adjunk aszimptotikus megoldást az alábbi rekurzív egyenletekre a mester tétellel!

   a. $T(n) = 4T\left(\frac{n}{2}\right) + n$

   b. $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

   c. $T(n) = 4T\left(\frac{n}{2}\right) + n^3$