Az algoritmus fogalma

Eddig főként az adattípusokkal foglalkoztunk. Az adatainkon általában különféle műveleteket, átalakításokat szoktunk végezni azzal a céllal, hogy ezáltal a kiinduló adatokból közvetlenül nem kiolvasható összefüggéseket, eredményeket kapjunk. Az ehhez szükséges tevékenységeket logikai sorrendbe rakva az algoritmus matematikai fogalmához kerülünk közelebb. Tulajdonképpen a procedúra bevezetésével már részint ezen a területen jártunk. 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 Turing, Church, Markov munkáit.

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.

Példa algoritmusra. Feladat: határozzuk meg egy pozitív s valós számra az $x = \sqrt{s}$ számot, azaz a szám négyzetgyökét valamilyen előre megadott $\varepsilon > 0$ pontossággal. A bemutatandó megoldó algoritmus egy elvileg tetszőleges pozitív számból kiindulva 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 valamely helyiértékes számrendszer jelöléseit használva, 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.

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.

A javasolt algoritmus valóban működik, ezt máris belátjuk. Első lépésként azt mutatjuk meg, hogy a sorozat határértéke valóban $\sqrt{s}$, ha a sorozat konvergál. Tegyük fel tehát a konvergenciát és legyen $\lim_{k \to \infty} x_k = x^*$. Akkor az $x_{k+1} = \frac{1}{2} \cdot \left(x_k + \frac{s}{x_k} \right)$ iterációs formulában mindkét oldalon elvégezve a határátmenetet k szerint az alábbi adódik.

$$\lim_{k\to\infty} x_{k+1} = \lim_{k\to\infty} \frac{1}{2}\cdot\left(x_k + \frac{s}{x_k}\right).$$

Innen kapjuk, hogy

$$x^* = \frac{1}{2}\cdot\left(x^* + \frac{s}{x^*}\right).$$

Átrendezve a formulát az

$$x^* = \frac{s}{x^*}$$

alakra jutunk, amelynek megoldása a pozitív számok között $x^* = \sqrt{s}$. Ezzel megállapíthatjuk, hogy amennyiben konvergál az eljárás, akkor valóban a keresett értékhez konvergál. Most belátjuk a konvergenciát. Az elv az lesz, hogy alkalmazzuk a tételt, amely szerint monoton csökkenő és alulról korlátos sorozatnak mindig van határértéke. Ki kell tehát mutatnunk először, hogy a sorozat alulról korlátos. Mivel $x_{k+1} = \frac{1}{2}\cdot\left(x_k + \frac{s}{x_k}\right)$ két pozitív szám számtani átlaga és a nemnegatív számok számtani közepe mindig nagyobb, mint a mértani közép, legfeljebb egyenlő lehet vele, ha a két szám megegyezik, ezért írhatjuk, hogy $x_{k+1} = \frac{1}{2}\cdot\left(x_k + \frac{s}{x_k}\right) \geq \sqrt{x_k \cdot \frac{s}{x_k}} = \sqrt{s}$. Tehát a sorozat minden tagja (kivéve esetleg az elsőt) nagyobb, mint $\sqrt{s}$, vagy egyenlő vele, vagyis a $\sqrt{s}$ egy alsó korlát. (A sorozat első véges sok tagjának a viselkedése a konvergenciát nem befolyásolja, legfeljebb elhagyjuk őket a sorozatból.) Ezután belátjuk a monoton csökkenést legalább is a második tagtól kezdve. Képezzük két egymást követő tag különbségét.

$x_{k+1} - x_k = \frac{1}{2}\cdot\left(x_k + \frac{s}{x_k}\right) - x_k = \frac{s}{2x_k} - \frac{x_k}{2} = \frac{s - x_k^2}{2x_k} = \frac{\left(\sqrt{s} + x_k\right)\left(\sqrt{s} - x_k\right)}{2x_k} \leq 0$, mert a nevező pozitív, a számlálóban az első tényező pozitív, a második pedig negatív, hiszen $\sqrt{s}$ alsó korlát. Emiatt $x_{k+1} - x_k \leq 0$, azaz $x_{k+1} \leq x_k$, ami a monoton csökkenést jelenti. Az algoritmus egyetlen szépséghibája az az, hogy a kilépési feltétel szerint akkor lépünk ki, amikor két egymást követő közelítés már keveset tér el egymástól, ami nem feltétlen jelenti azt, hogy akkor a $\sqrt{s}$-től is csak kicsit térünk el úgy általában. Maradjunk annyiban, hogy itt a kettő közelítőleg egyszerre teljesül.

Számítsuk ki az $x = \sqrt{2} \approx 1{,}414213562373...$ 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	$= \frac{1}{2}\cdot\left(1{,}414215686 + \frac{2}{1{,}414215686}\right)$

Látjuk, hogy x_2 és x_3 első öt tizedesjegye megegyezik, ezért megállhatunk.

Az algoritmus megadási, lejegyzési módja: a pszeudokód és a folyamatábra.

Az algoritmust *szövegesen* adtuk meg a fenti esetben. Ez egy lehetőség általában az algoritmus lejegyzésére. Elterjedt azonban a pszeudokódos megadás is, mely közelebb viszi a leírást a számítógépes megvalósításhoz anélkül, hogy elkötelezné magát egy konkrét programozási nyelv mellett. A pszeudokód az algoritmusban megfogalmazott gondolatokat tükrözi, nem pedig a kiválasztott programozási nyelv szntaktikai (nyelvtani) szabályait. (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. *Blokkszerkezet*eket fogunk használni, amint az sok programozási nyelvben elterjedt. A blokk összetartozó utasításokat jelent, amelyeket a felsorolásuk sorrendjében végre kell hajtani. A blokkot általában valamilyen módon logikai zárójelek közé teszik, azonban mi a zárójelezés helyett a *bekezdés eltolás*ának módszerét fogjuk használni, azaz a blokk minden sorát ugyanannyival beljebb kezdjük, mint az elsőt. Így nem kell a zárójel, és már messziről látszik, hogy mi tartozik egy blokkhoz.
2. Az értékadás jele a ← jel lesz. Bátran alkalmazzuk tömbök, struktúrák értékadására és többszörös értékadásra is.
3. A magyarázatokat, megjegyzéseket // kezdőjellel fogjuk jelezni. Ez lehet egy teljes megjegyzés sor, vagy lehet egy adott sorhoz hozzáfűzött megjegyzés.
4. Az eljárásokban használt változók, amelyek nem a paraméterlistán keresztül jöttek, lokálisak lesznek, külön nem deklaráljuk őket lokálisnak a leírásban.
5. 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.
6. 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.
7. 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.
8. 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.
9. A pszeudokód nem zárja ki, hogy az algoritmus egyes részeit szöveges módon tüntessük fel.

Az alábbi pszeudokód 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 és ezután kilépünk az **IF** szerkezetből, egyébként az **ELSE** blokk kerül végrehajtásra átugorva a **THEN** blokkot.
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. A blokk végrehajtása után kilépünk a **CASE**-ből.
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 *igaz*nak lenni. A blokk végrehajtása után kilépünk a **CASE**-ből.
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 ciklusváltozó ∈ Halmaz **DO** Blokk	
	A halmaz elemeit valamely előre rögzített sorrendben soroljuk fel, és ebben a sorrendben végighaladva az elemeken mindegyik elem esetére végigszámoljuk a blokkot. Ha a halmaz üres, akkor egyszer sem hajtjuk végre a blokkot.
INC(változó)	A szám típusú változó értékének megnövelése eggyel
DEC(változó)	A szám típusú változó értékének csökkentése eggyel

Utasítás szerkezete	Utasítás magyarázata
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. Lehet, hogy a ciklusmagot egyszer sem hajtjuk végre. (A feltétel a bennmaradást fogalmazza meg.)
DO Blokk **WHILE** feltétel	(Hátultesztelő iteratív ciklus.) Végehajtjuk a blokkot egyszer mindenképpen, és ezután ha a feltétel *igaz*, akkor visszatérünk a blokkra Ha a feltétel *hamis*, akkor kilépünk a ciklusból. (A feltétel a bennmaradást fogalmazza meg.)
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. (A feltétel a kilépést fogalmazza meg.)
CALL procedúranév (paraméterlista)	
	Procedúrahívás a paraméterlistán felsorolt paraméterekkel. Lehetnek procedúrák, amelyeknél a paraméterlista elmarad a zárójellel együtt.
RETURN (paraméterlista)	Kilépés a procedúrából, visszatérés a hívó (**CALL**) utasítást követő utasításra. A felsorolt paraméterek átadása a hívó rutinnak.
INPUT lista, vagy **INPUT** (lista)	Adatbekérő utasítás a listán megadott rekeszekbe.
OUTPUT lista, vagy **OUTPUT** (lista)	Adatkiviteli utasítás a listán megadott rekeszekből, vagy értékeket.
GOTO címke	Arra a sorra ugrat, amely előtt a megadott címke található. (A címke lehet sorszám, vagy egy név, amit a sor elé írunk kettősponttal elválasztva a sortól.)
STOP	A program futásának leállítása, befejezése.

A fenti iteratív négyzetgyökvonási algoritmus pszeudokóddal pédául így nézhetne ki. A jobboldalon egy praktikusabb változat látható. (Miért praktikusabb?)

	1.2.1 algoritmus
	Négyzetgyökvonás iterációval
	//
	NÉGYZETGYÖK (s, @z)
1.	Input paraméter: $s \in \boldsymbol{R}$, $s \geq 0$
2.	Output paraméter: $z \in \boldsymbol{R}$, $z \geq 0$
3.	$x_0 \leftarrow 1$
4.	$k \leftarrow 0$
5.	**REPEAT** $x_{k+1} \leftarrow (x_k + s / x_k) / 2$
6.	$k \leftarrow k + 1$
7.	**UNTIL** $\lvert x_k - x_{k-1} \rvert < \varepsilon$
8.	$z \leftarrow x_k$
9.	**RETURN** (z)

	1.2.2 algoritmus
	Négyzetgyökvonás iterációval
	//
	NÉGYZETGYÖK (s, @z)
1.	Input paraméter: $s \in \boldsymbol{R}$, $s \geq 0$
2.	Output paraméter: $z \in \boldsymbol{R}$, $z \geq 0$
3.	$z \leftarrow 1$
4.	**REPEAT** $x \leftarrow z$
5.	$z \leftarrow (x + s / x) / 2$
6.	**UNTIL** $\lvert z - x \rvert < \varepsilon$
7.	**RETURN** (z)

Ha ki akarjuk számítani az $u = \sqrt{2}$ értékét ezzel a procedúrával, akkor azt írjuk, hogy

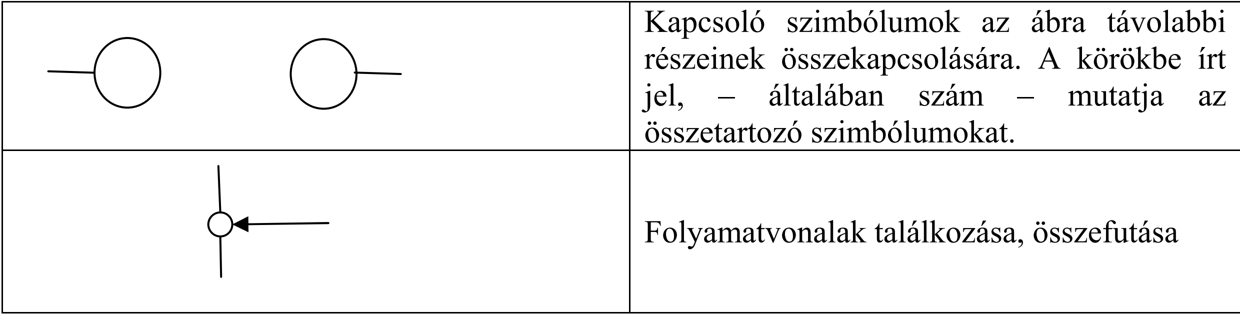
CALL NÉGYZETGYÖK (2, @u). Gyakran zavaró, mint itt is, amikor egy procedúra egy definiált adattípusnak csak és pontosan egy elemét adja vissza. Ilyenkor szemléletesebb a procedúra definícióját függvény formájában megadni, azaz a procedúra nevét használni output paraméterként és annak adni meg a típusát. Ekkor nem is szükséges őt CALL-lal aktivizálni, elegendő a nevét leírni egy értékadó utasításban. Például a négyzetgyökvonás pszeudokódja lehet ilyen is.

	1.2.2 algoritmus
	Négyzetgyökvonás iterációval
	// függvény forma
	NÉGYZETGYÖK (s) $\in \boldsymbol{R}$
1.	Input paraméter: $s \in \boldsymbol{R}, s \geq 0$
2.	$z \leftarrow 1$
3.	**REPEAT** $x \leftarrow z$
4.	$z \leftarrow (x + s / x) / 2$
5.	**UNTIL** $\lvert z - x \rvert < \varepsilon$
6.	NÉGYZETGYÖK $\leftarrow z$
7.	**RETURN**

Ennek a változatnak a használata során, ha ugyanúgy, mint fentebb ki akarjuk számítani az $u = \sqrt{2}$ értékét, akkor elegendő azt írni, hogy u = NÉGYZETGYÖK (2).

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.

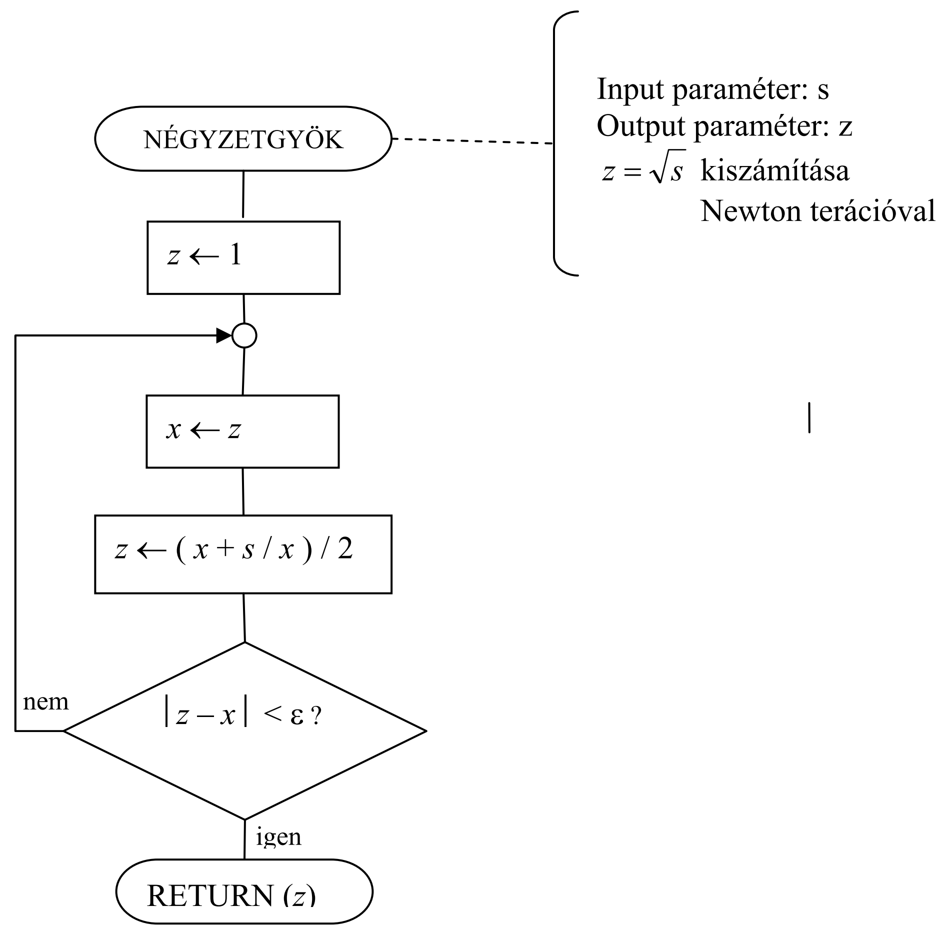
Szimbólum	**Szimbólum magyarázata**
START STOP	*Kezdőszimbólum* (az algoritmus kezdete, pontosan egy van) és a *befejező szimbólum* (az algoritmus megállási helye, legalább egy van)
	Tevékenység (memóriabali műveletek)
	Döntés a szimbólumba írt feltétel milyensége alapján, két-, vagy háromfelé ágaztat el. A kilépő vonalakra rá kell írni, hogy milyen esetben lépünk ott ki.
	Alprogram, procedúra meghívása
	Input – output művelet. A jobb felső sarokba (vagy fölé) írjuk, hogy input vagy pedig output.



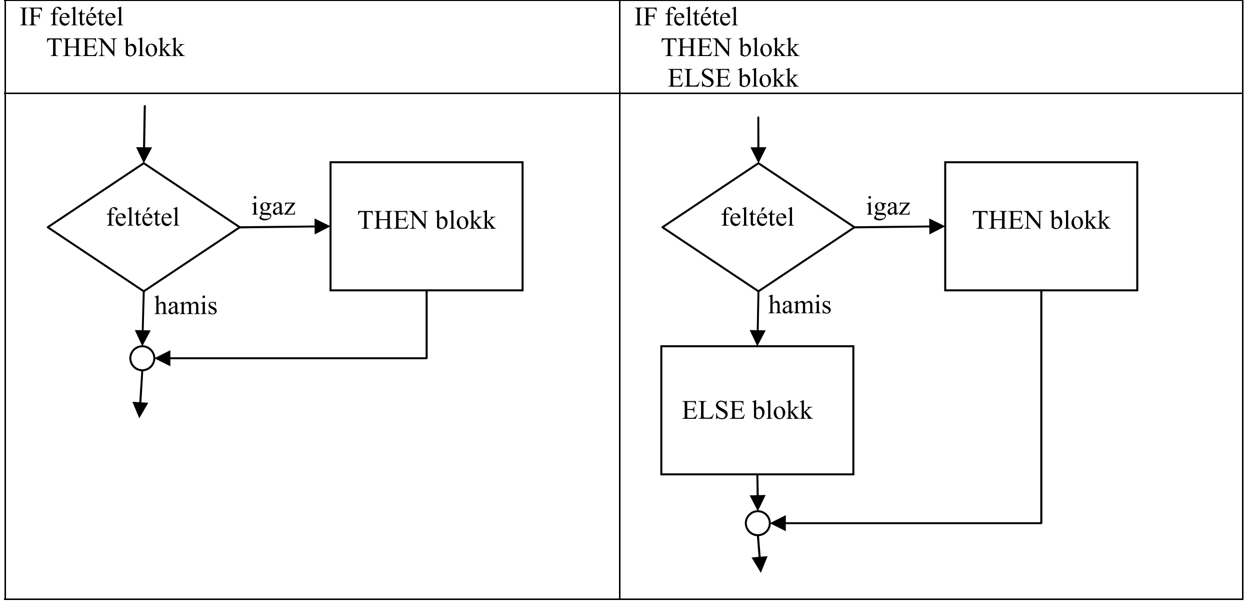
	Kapcsoló szimbólumok az ábra távolabbi részeinek összekapcsolására. A körökbe írt jel, – általában szám – mutatja az összetartozó szimbólumokat.
	Folyamatvonalak találkozása, összefutása

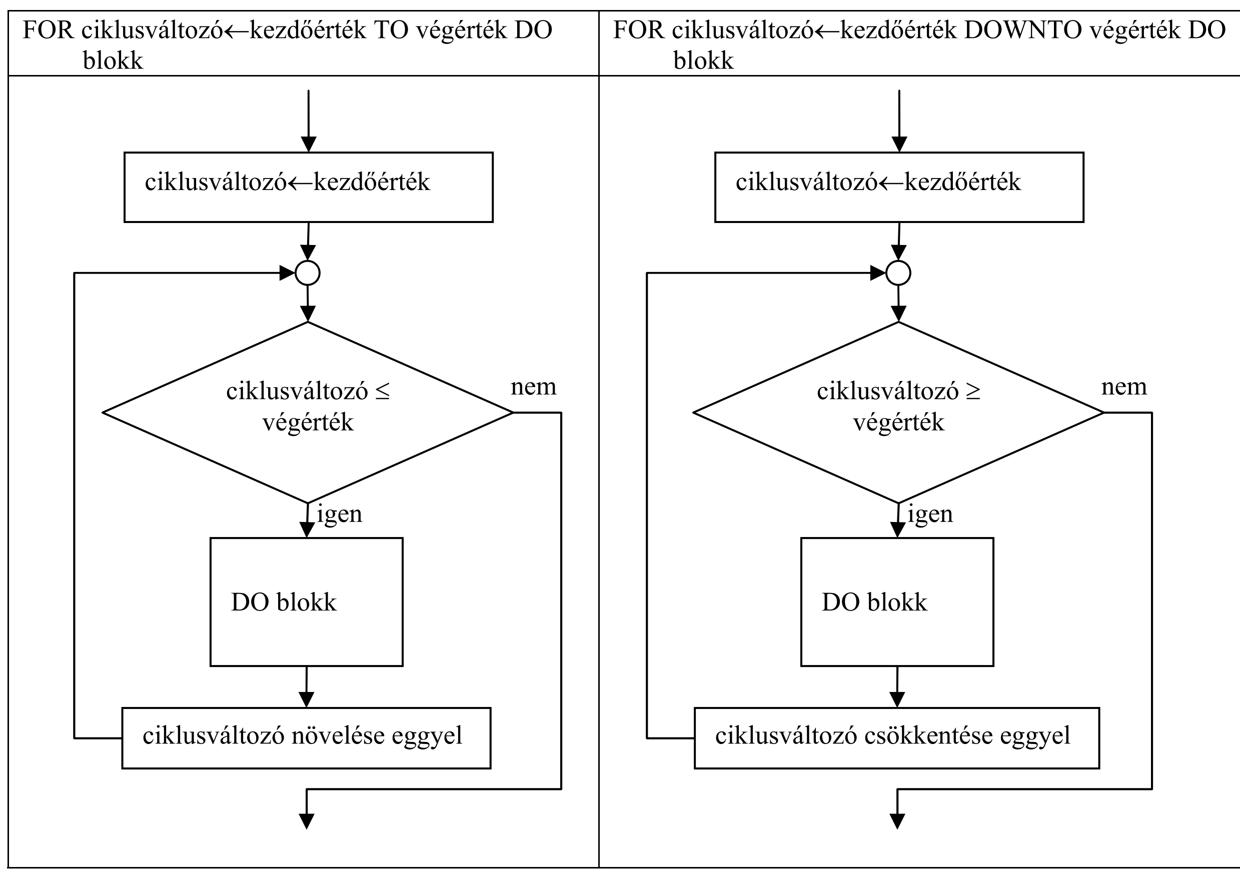
A folyamatvonalakon a haladás iránya balról-jobbra, vagy fentről-lefelé, hacsak a vonalra kitett nyíl másként nem mutat. Megjegyzést a szimbólumokhoz szaggatott vonallal a szimbólumhoz kapcsolt, megfelelően méretezett kezdő szögletes zárójel jellel lehet hozzákapcsolni. A szöveg a szögletes zárójel mögé kerül. Procedúra folyamatábrája esetén a Start szimbólumba a procedúra neve kerül, mellé megjegyzéssel kapcsoljuk a paraméterezést. Ugyanígy a procedúrahívó utasítás szimbólumánál. A Stop szimbólumba ilyenkor a **RETURN** szót írjuk. Amennyiben egy szimbólum egy nagyobb léptékű még ki nem részletezett programrészt szimbolizál, akkor a szimbólumba egy megnevezést írunk, jelezzük, hogy a kirészletezés hol található, majd a kirészletezés Start szimbólumába is ez a megnevezés kerül. A kirészletezés záró szimbóluma azonos a Stop szimbólummal, de a szimbólumba nem írunk semmit.
Más jellegű tevékenységek, adathordozókkal végzett műveletek, előzetes adatfeldolgozások stb. számára is vannak folyamatábra szimbólumok szabványosítva.

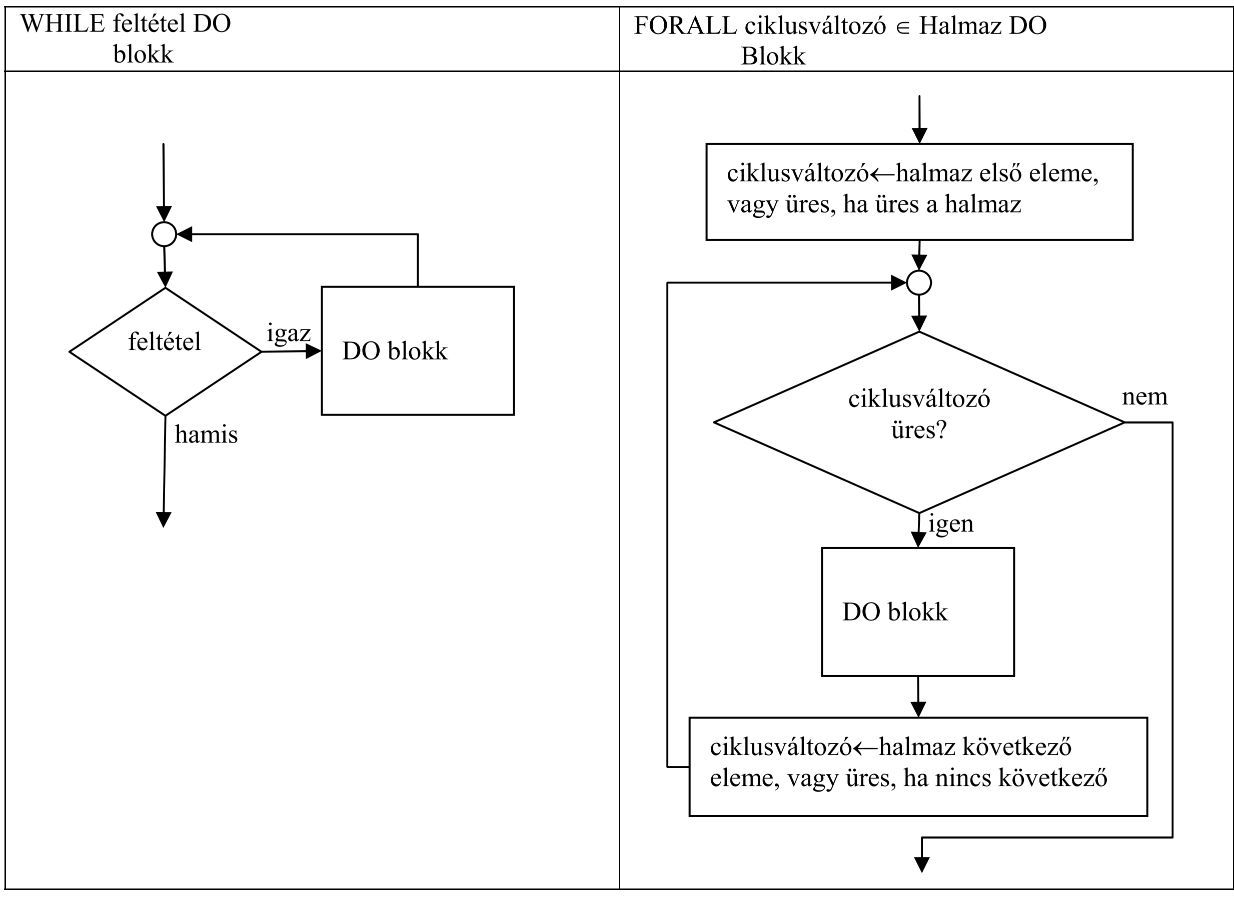
A négyzetgyökvonás fenti praktikusabb változata folyamatábrával:

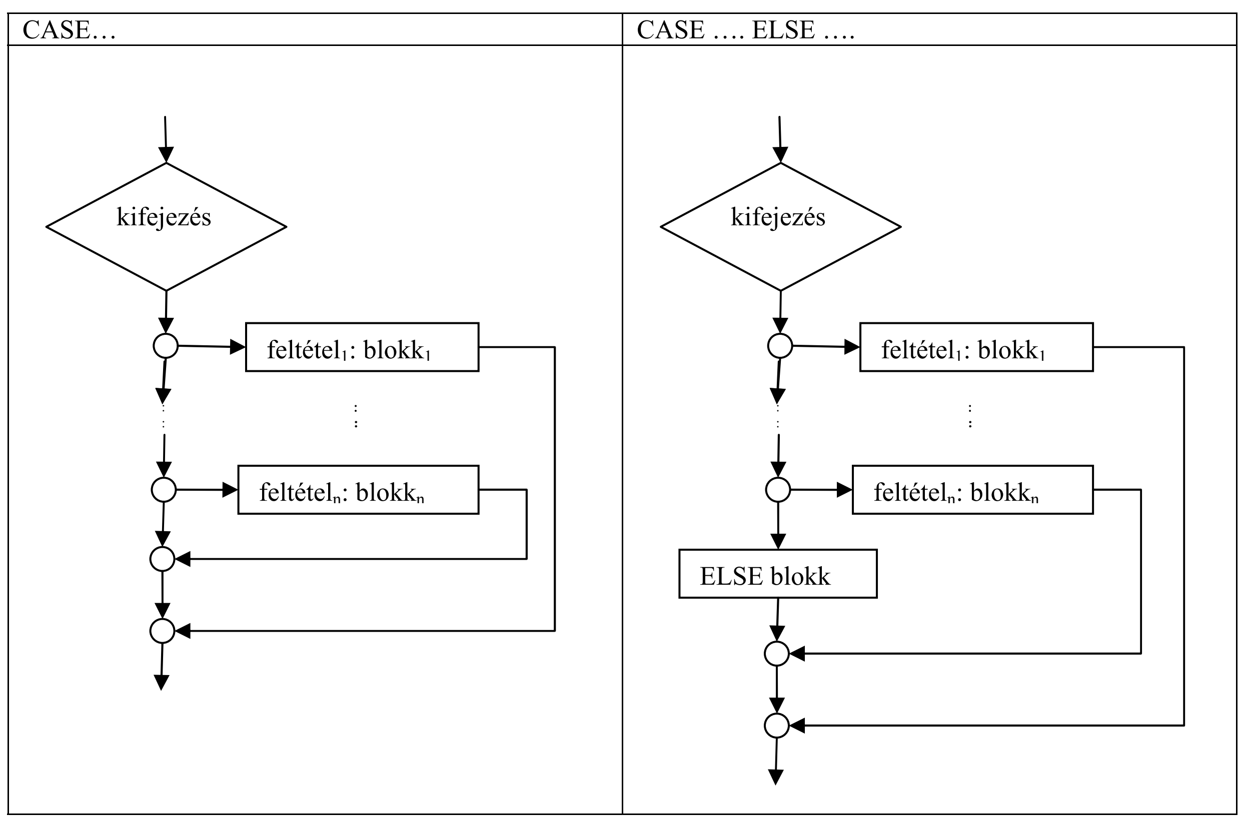


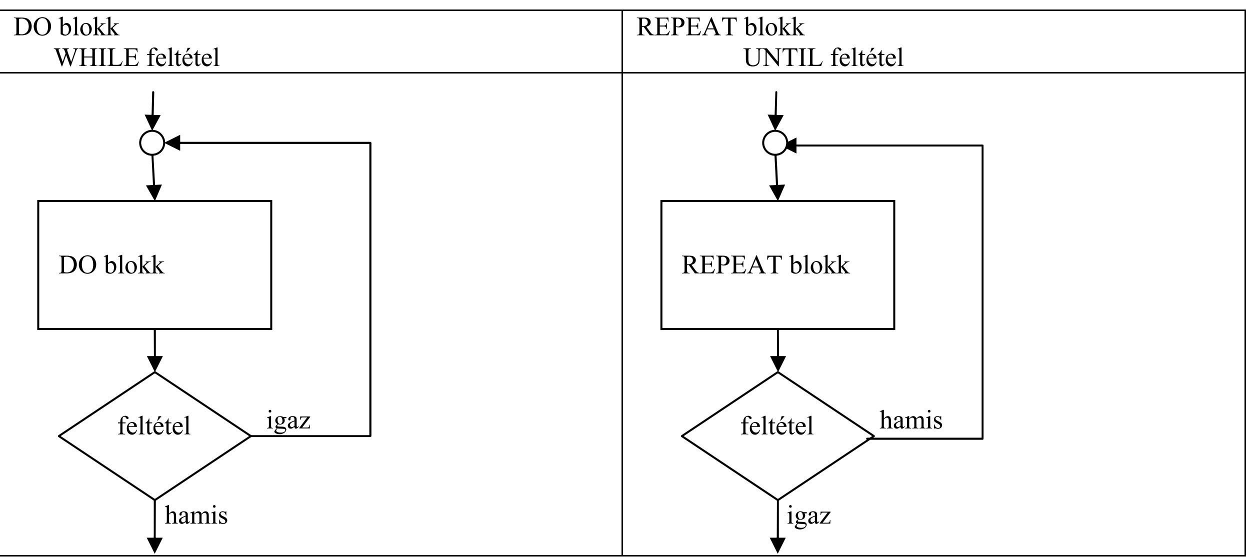
Egy pszeudokód utasítást egy vagy több folyamatábra szimbólum segítségével adhatunk meg. Ugyanakkor a folyamatábra szimbólumok lehetnek nagyléptékűek, azaz egy részfeladatot csak jelzünk egy tevékenység szimbólummal, de csak majd később részletezzük ki, bontjuk le azt. Alább megadjuk fentebb említett strukturális utasításoknak megfelelő folyamatábra részleteket.





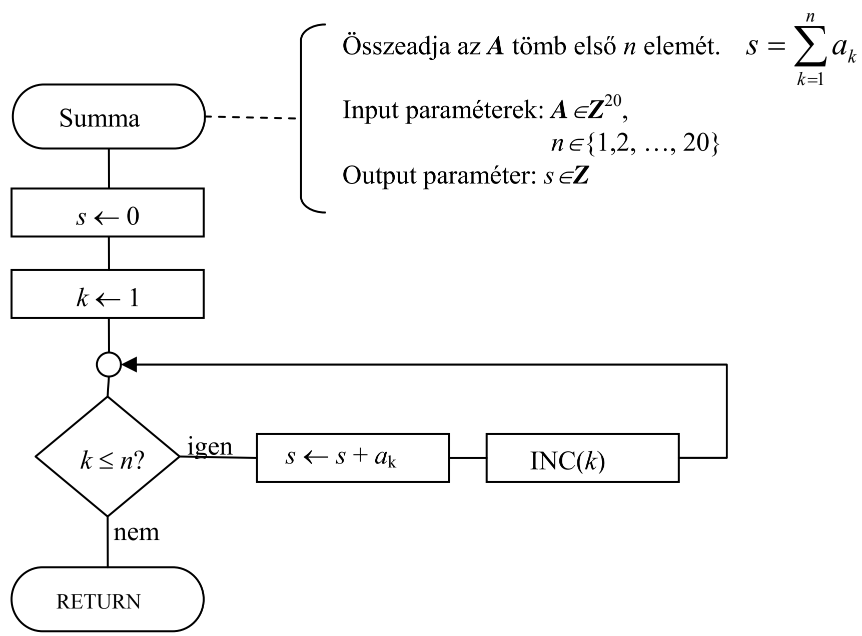

WHILE feltétel DO
blokk
feltétel
igaz
DO blokk
hamis
FORALL ciklusváltozó ∈ Halmaz DO
Blokk
ciklusváltozó←halmaz első eleme, vagy üres, ha üres a halmaz
ciklusváltozó üres?
nem
igen
DO blokk
ciklusváltozó←halmaz következő eleme, vagy üres, ha nincs következő


CASE…
kifejezés
feltétel1: blokk1
feltételn: blokkn
CASE …. ELSE ….
kifejezés
feltétel1: blokk1
feltételn: blokkn
ELSE blokk

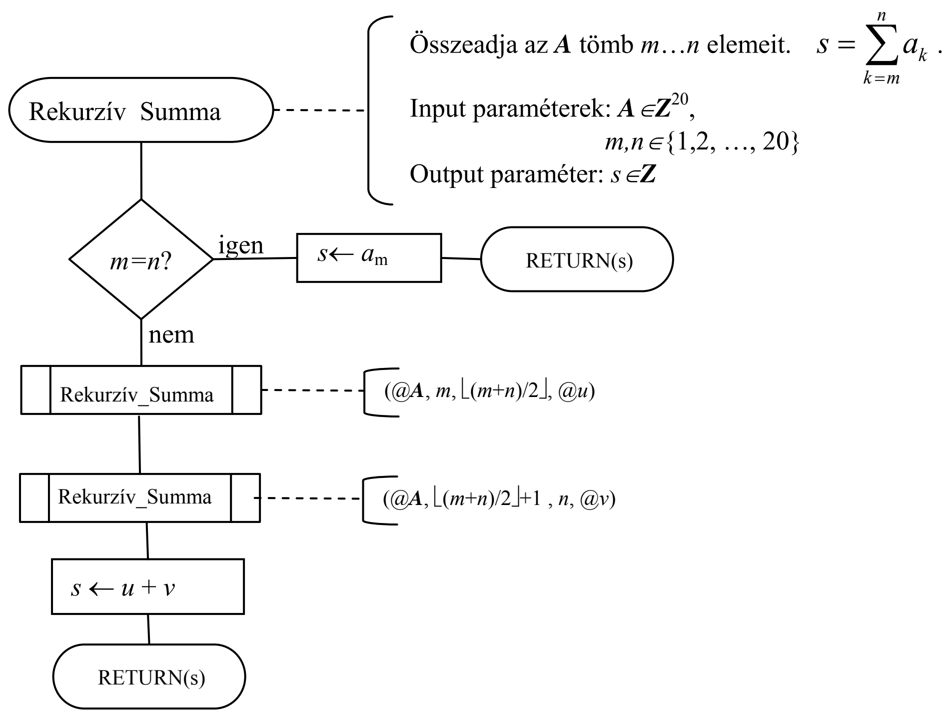


Az algoritmusok közül kitűnnek a *rekurzív* algoritmusok és az *iteratív* algoritmusok. Mindkét fajta algoritmus hatékonyan realizálható számítógépen. (Lehet egy algoritmus persze egyszerre rekurzív is és iteratív is.) Az iteratív algoritmusok hasonló, vagy azonos műveletek sorozatát ismétlik (a latin *iteratio* szóismétlést jelent). A rekurzív algoritmusokban azt ismerjük fel, hogy a probléma mérete redukálható kisebb méretre, majd még kisebb méretre, stb. egészen addig, amíg végül már ismerjük a megoldást 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 például a fenti négyzetgyökvonás algoritmusa. A rekurzív algoritmusok mindig átírhatók iteratív formára is, és ez fordítva is igaz. A korábban tárgyalt tömbelem összegző procedúra pszeudokódja és folyamatábrája lehet az alábbi.

	Summa (@***A***, n, @s)
1.	Input paraméterek: $\boldsymbol{A} \in \boldsymbol{Z}^{20}$,
2.	$n \in \{1,2, \ldots, 20\}$
3.	Output paraméter: $s \in \boldsymbol{Z}$
4.	$s \leftarrow 0$
5.	**FOR** $k \leftarrow 1$ **TO** n **DO**
6.	$s \leftarrow s + a_k$
7.	**RETURN**(s)



	Rekurzív_Summa (@***A***, *m* , *n*, @*s*)
1.	Input paraméterek: $A \in Z^{20}$,
2.	$m,n \in \{1,2, …, 20\}$ és $m \le n$,
3.	Output paraméter: $s \in Z$
4.	**IF** $m=n$
5.	**THEN** $s \leftarrow a_m$
6.	**ELSE** Rekurzív_Summa (@***A***, *m*, $\lfloor (m+n)/2 \rfloor$, @*u*)
7.	Rekurzív_Summa (@***A***, $\lfloor (m+n)/2 \rfloor+1$, *n*, @*v*)
8.	$s \leftarrow u+v$
9.	**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. Három ilyen általános elvet, *algoritmus-tervezési stratégiát* említünk itt meg.

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.

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 algoritmusa esetében a 7 pont így nézhetne ki:
1. A kiinduló adatok lehetséges halmaza: tetszőleges valós pozitív szám.
2. A lehetséges eredmények halmaza: tetszőleges valós pozitív szám.
3. A közbülső eredmények: tetszőleges valós 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.

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 hatékonysági, *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, de erre itt most nem térünk ki.

Most ezután nézzünk néhány apró kis algoritmust különböző problémák, feladatok megoldására. Csak egy lehetséges pszeudokódot közlünk mindegyik probléma megoldására. Készítsük el a megfelelő folyamatábrát! Vizsgáljuk meg, hogy melyik az áttekinthetőbb!

Algoritmus mondat szavainak a megszámlálására.

Mondatnak tekintünk egy karaktersztringet. Szavak a sztringben az egymást követő nem helyköz karakterek sorozata. A szavakat egymástól egy vagy több helyköz válsztaja el egymástól. Az első előtt és az utolsó szó megengedetten állhatnak fölösleges szóközök. megengedett az üres sztring is, amely zérus számú szót tartalmaz.

	// Mondat szavainak megszámlálása (procedúramentes megoldás).
	Használt változók: $s \in$ Ascii*, // az input mondat stringje,
	// betűi: $s_1, s_2, \ldots, s_n$, ahol n=Hossz[s]
	szószám$\in \boldsymbol{N}$ // a megtalált szavak száma
	$i \in \boldsymbol{N}$ // ciklusváltozó, betűszámláló
	állapot$\in \boldsymbol{A}$=({ *köz_állapot*, *szó_állapot* },∅)
	// A vizsgálat két állapota menet közben

1.	**INPUT** (*s*)
2.	*állapot*←*köz*
3.	*szószám*←0
4.	**FOR** *i*←1 **TO** Hossz[*s*] **DO**
5.	**IF** *állapot*=*köz_állapot*
6.	**THEN IF** s_i≠' '
7.	**THEN** **INC**(*szószám*)
8.	*állapot*←*szó_állapot*
9.	**ELSE IF** s_i=' '
10.	**THEN** *állapot*←*köz_állapot*
11.	**OUTPUT** (*szószám*)
12.	**STOP** // ---------------- Program vége ------------------

Ugyanaz a probléma procedúra és főprogram segítségével megoldva.

	// ---------- **Szószámláló** procedúra ---------- Mondat szavainak megszámlálása
	Szószámláló (@*s*, @*szószám*)
1.	Input paraméter: $s \in$ Ascii* // a mondat stringje // betűi: $s_1, s_2, \ldots, s_n$, ahol n=Hossz[s]
2.	Output paraméter: *szószám*$\in \boldsymbol{N}$ // a megtalált szavak száma
3.	Használt lokális változók: $i \in \boldsymbol{N}$ // ciklusváltozó, betűszámláló
4.	*állapot*$\in \boldsymbol{A}$=({ *köz_állapot*, *szó_állapot* },∅) // A vizsgálat két állapota menet közben
5.	*állapot*← *köz_állapot*
6.	*szószám*←0
7.	**FOR** *i*←1 **TO** Hossz[*s*] **DO**
8.	**IF** *állapot*= *köz_állapot*
9.	**THEN IF** s_i≠' '
10.	**THEN** **INC**(*szószám*)
11.	állapot← *szó_állapot*
12.	**ELSE IF** s_i=' '
13.	**THEN** *állapot*← *köz_állapot*
14.	**RETURN**(*szószám*) // ---------- Szószámláló procedúra vége ----------

	// ======= Főprogram =======
1.	Használt változók: $x \in \text{Ascii}^*$, // az input mondat stringje
2.	$k \in \boldsymbol{N}$ // a megtalált szavak száma
3.	**INPUT** (*x*)
4.	**CALL** Szószámláló (@*x*; @*k*)
5.	**OUTPUT** (*k*)
6.	**STOP** // ---------- Főprogram vége ----------

Algoritmus annak eldöntésére, hogy egy sztring lehet-e azonosító.
Lehet-e egy sztring *azonosító*? Az azonosító egy sztring, amely betűvel kezdődik, utána betűk, vagy számjegyek állhatnak. Megengedett az üres sztring inputként.

	// ---------- **Lehet_e_azonosító** procedúra ---------- Azonosító ellenőrzés
	Lehet_e_azonosító (@*s*, :@*Lehet*)
1.	Input paraméter: $s \in \text{Ascii}^*$ // a vizsgálandó sztring
2.	Output paraméter: $Lehet \in \boldsymbol{L}$ // *igaz*, ha lehet, *hamis*, ha nem
3.	Használt lokális változók: $i \in \boldsymbol{N}$ // ciklusváltozó, betűszámláló
4.	*Számjegyek* ← {'0',…,'9'} // a számjegyek halmaza
5.	*Betűk* ← {'A',…,'Z','a',…,'z'} // betűk halmaza
6.	**IF** Hossz[*s*] = 0
7.	**THEN** *Lehet* ← *hamis*
8.	**ELSE** *Lehet* ← *igaz*
9.	*i* ← 1
10.	**WHILE** ($i \le$ Hossz[*s*]) ÉS *Lehet* **DO**
11.	**IF** *i* = 1
12.	**THEN IF** $s_1 \in$ *Betűk*
13.	**THEN** *Lehet* ← *hamis*
14.	**ELSE IF** $s_i \notin$ (*Betűk* ∪ *Számjegyek*)
15.	**THEN** *Lehet* ← *hamis*
16.	**INC**(*i*)
17.	**RETURN** (*Lehet*) // ---------- Lehet_e_azonosító procedúra vége ----------

Gyors algoritmus keresésre valós számok rendezett tömbjében.
Adott egy tömb (vektor), amelyben valós számok (lebegőpontos számok) helyezkednek el növekvő sorrendben. A tömb elemeinek indexelése az egyes számmal kezdődik. Adott egy valós szám. Megadandó a tömb azon elemének indexe, amely elem megegyezik az adott számmal, vagy zérus, ha az adott szám nincs benne a tömbben.
Algoritmusként az úgynevezett bináris keresést vagy logaritmikus keresést fogjuk használni. A bináris keresés alapötlete az, hogy az adott elem keresésekor az elem összehasonlítását a tömb középső eleménél kezdjük. Ha egyezést tapasztalunk, akkor az eljárás véget ér. Ha a keresett számérték kisebb, mint a megvizsgált elem számértéke, akkor a tömb középső elemének indexétől kisebb indexű elemek között folytatjuk a keresést. Ha a keresett elem értéke nagyobb, mint a megvizsgált elem kulcsa, akkor a tömb középső elemének indexétől nagyobb indexű elemek között folytatjuk a keresést. A vizsgálandó hossz ezáltal feleződött. A további keresés ugyanilyen elv alapján megy tovább. Minden lépésben vagy megtaláljuk a keresett számértéket, vagy fele hosszúságú résztömbben folytatjuk a keresést. Ha a résztömb hossza zérusra zsugorodik, akkor a keresett kulcs nincs a tömbben.

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.

	// ---------- **Bináris_keresés** procedúra ---------- Bináris keresés valós számok rendezett tömbjében nyílt intervallumos módszerrel
	Bináris_keresés (@***A***, *Elem*, @*Index*)
1.	Input paraméter: $\boldsymbol{A} \in \boldsymbol{R}^n$, // a valós számok tömbje, $a_i \le a_j$, ha $i<j$.
2.	$Elem \in \boldsymbol{R}$ // a keresett elem, valós szám
3.	Output paraméter: $Index \in \boldsymbol{N}$ // a megtalált elem indexe, zérus, ha nincs meg
4.	
5.	Használt lokális változók: $alsó, felső \in \boldsymbol{N}$ // keresési indexintervallum alsó és felső határai
6.	$középső \in \boldsymbol{N}$ // vizsgált elem indexe
7.	$Nincs_meg \in \boldsymbol{L}$ // *igaz*, ha még nem, *hamis*, ha már megtaláltuk
8.	
9.	*alsó* ← 0
10.	*felső* ← Hossz[***A***] + 1
11.	$középső \leftarrow \left\lfloor \frac{alsó + felső}{2} \right\rfloor$
12.	*Nincs_meg* ← *igaz*
13.	**WHILE** (*alsó* < *középső*) ÉS *Nincs_meg* **DO**
14.	**IF** Elem = $a_{középső}$
15.	**THEN** *Index* ← *középső*
16.	*Nincs_meg* ← *hamis*
17.	**ELSE** **IF** *Elem* > $a_{középső}$
18.	**THEN** *alsó* ← *középső*
19.	**ELSE** *felső* ← *középső*
20.	$középső \leftarrow \left\lfloor \frac{alsó + felső}{2} \right\rfloor$
21.	**IF** *Nincs_meg*
22.	**THEN** *Index* ← 0
23.	**RETURN** (*Index*) // ---------- Bináris_keresés procedúra vége ----------

Hogyan módosulna az algoritmus, ha zárt intervallumokkal dolgoznánk? Írjuk át a procedúrát rekurzívra!

FELADATOK

1. Hogyan lehetne a négyzetgyökvonási algoritmushoz valóban jó kezdő közelítést adni? Miért nem mindig elég jó az $x_0 = 1$ érték?
2. Tanulmányozzuk az alábbi algoritmusokat!
   a. Bizonyítsuk be, hogy az $x_{n+1} = 1 + \frac{1}{x_n}$ iteratív algoritmus bármilyen pozitív x_0 kezdőérték esetén a $\Phi = \frac{1+\sqrt{5}}{2}$ értékhez konvergál! Osszuk fel a pozitív számok halmazát tartományokra melyekből indítva az algoritmusunkat a 10^{-3} pontosság ($|x_n - \Phi| < 10^{-3}$ teljesül) eléréséhez szükséges iterációs lépések száma rendre 0, 1, 2, stb. Melyik a „legrosszabb" tartomány és hány iterációs lépés jellemzi? Tegyük fel, hogy nem ismerjük a Φ értékét és a pontosságról az alapján döntünk, hogy a két legutolsó iteráció eredménye az előírt pontosságnál nem tér el erősebben egymástól. (Ez az elv nem alkalmazható általában minden iterációs módszernél!) Hogyan alakul a válasz az előzőekben kitűzött feladatra ebben az esetben?
   b. Határozzuk meg $\frac{1+\sqrt{5}}{2}$ értékét a 2.a formula alapján az $x_0 = 1$ kezdőpontból indítva kilenc tizedesjegy pontosságig! Határozzuk meg ugyanezt az értéket úgy, hogy a $\sqrt{5}$ értékét a négyzetgyökvonási algoritmussal számoljuk ugyancsak az $x_0 = 1$ pontból indítva, majd a kellő pontosság elérése után egyet hozzáadunk az eredményhez és 2-vel osztunk! Melyik módszer a gyorsabb?
3. Készítsünk algoritmust LOTTO számok előállítására. (90 számból találomra 5 különböző szám kiválasztása). Egy x szám találomra kiválasztása történjen az $x \leftarrow$RANDOM(90) utasítással. (A RANDOM(n) egy függvény, amely az 1,2,…,n egész számok közül választ egyet találomra. Véletlenszám generátor.) Rajzoljuk meg a folyamatábrát! Írjuk le a pszeudokódot!
4. Legyen adott az a, b, c három tetszőleges valós szám! Tekintsük őket egy $ax^2+bx+c=0$ egyenlet együtthatóinak. Készítsünk algoritmust az egyenlet gyökeinek a megkeresésére, ha vannak valós gyökök. Minden lehetséges esetet vizsgáljunk le. Rajzoljuk meg a folyamatábrát! Írjuk le a pszeudokódot!