

Babeş-Bolyai Tudományegyetem, Kolozsvár

Informatikai-matematika záróvizsga tankönyv



2018-2019

Záróvizsga témák

Informatikai-matematika szak

(a 2019. júliusi vizsgaidőszaktól érvényes)

Algoritmusok és programozás (4 téma)

1. Keresés (szekvenciális és bináris), rendezés (buborékredezés, gyorsrendezés (quicksort), kiválasztásos rendezés).
2. Algoritmusok és specifikációk. Algoritmus írása egy adott specifikáció alapján. Adott egy algoritmus, adjuk meg a végrehajtása eredményét.
3. Objektorientált programozási fogalmak a Python, C++, Java és C# programozási nyelvekben: osztályok és objektumok; egy osztály tagjai és hozzáférési módosítók; konstruktorok és destruktorok.
4. Osztályok közötti kapcsolatok: származtatott osztályok és öröklési viszony; metódusok felülírása; polimorfizmus; dinamikus kötés; absztrakt osztályok és interfészek.

Tartalomjegyzék

1. Algoritmusok és programozás (Ionescu Klára)	5
1.1. Programozási tételek	5
1.2. Lépések finomítása és optimalizálás	21
1.3. Rendező algoritmusok	26
1.4. A visszalépéses keresés módszere (backtracking)	29
1.5. Az oszd meg és uralkodj módszer (divide et impera)	42
2. Objektumorientált programozás (Darvay Zsolt)	49
2.1. Objektumorientált fogalmak	49
2.2. Az objektumorientált programozási módszer	64
2.3. Javasolt feladatok	72

Copyright © 2018 A szerzők

Minden jog fenntartva! E tankönyvet, illetve annak részeit tilos reprodukálni, adat-rögzítő rendszerben tárolni, bármilyen formában vagy eszközzel – elektronikus úton vagy más módon – közölni a szerzők előzetes írásbeli engedélye nélkül.

A szerzők a lehető legnagyobb körültekintéssel jártak el e tankönyv szerkesztésekor. A szerzők nem vállalnak semmilyen garanciát e kiadvány tartalmával, teljességével kapcsolatban. A szerzők nem vonhatóak felelősségre bármilyen baleset, vagy káresemény miatt, amely közvetlen vagy közvetett úton kapcsolatba hozható e tankönyvvel.

1. fejezet Algoritmusok és programozás

1.1. Programozási tételek

A feladatok *feladatosztályok*ba sorolhatók a jellegük szerint. E feladatosztályokhoz készítünk a teljes feladatosztályt megoldó *algoritmussztályt*, amelyeket *programozási tételek*nek nevezünk. Bebizonyítható, hogy ezek a megoldások a szóban forgó feladatok *garantáltan helyes és optimális* megoldásai.

A programozási tételek a feladat bemenete és kimenete szerint négy csoportra oszthatók:

- A. sorozathoz érték rendelése (1 sorozat – 1 érték)
- B. sorozathoz sorozat rendelése (1 sorozat – 1 sorozat)
- C. sorozatokhoz sorozat rendelése (több sorozat – 1 sorozat)
- D. sorozathoz sorozatok rendelése (1 sorozat – több sorozat)

A. Sorozathoz érték rendelése

1.1.1. Sorozatszámítás

Adott az N elemű X sorozat. A sorozathoz hozzá kell rendelnünk egyetlen S értéket. Ezt az értéket egy, az egész sorozaton értelmezett f függvény (pl. elemek összege, szorzata stb.) adja. Ezt a függvényt felbonthatjuk értékpárokra kiszámított függvények sorozatára, így a megoldás az F_0 semleges elemre, valamint egy kétoperandusú műveletre épül. Az S kezdőértéke a semleges elem. A kétoperandusú műveletet végrehajtjuk minden X_i elemre és az S értékre: $S \leftarrow f(X_i, S)$.

Összeg és szorzat

Egyetlen kimeneti adatot számítunk ki, adott számú bemeneti adat feldolgozásának eredményeként, például a bemeneti adatok összegét, esetleg szorzatát kell kiszámítanunk.

Megoldás

A feladat megoldása előtt szükséges tudni, hogy mely érték felel meg a bemeneti adatok halmazára és az elvégzendő műveletre nézve a *semleges elem*nek. Feltételezzük, hogy a bemeneti adatok egész számok, amelyeknek a számossága N^1 .

Algoritmus Összegezés(N, X, S): { Sajátos eset }
{ Bemeneti adatok: az N elemű X sorozat, kimeneti adat: S }

$S \leftarrow 0$

Minden $i = 1, N$ **végezd el:** { minden adatot fel kell dolgoznunk }

$S \leftarrow S + X_i$

vége(minden)

Vége(algoritmus)

¹ A következőkben az algoritmusok implementálása különböző típusú függvényekként szabadon választható. Ha a függvény egyetlen értéket számít ki, akkor ezt nem kötelező kimeneti paraméterként implementálni, hanem térítheti a függvény.

Egy másik megközelítésben el kell döntenünk, hogy az adatok, teljességükben, rendelkeznek-e egy adott tulajdonsággal vagy sem. Más szóval: nem létezik egyetlen adat sem, amely ne lenne T tulajdonságú. Ekkor a bemeneti adathalmaz minden elemét meg kell vizsgálnunk. Mivel a döntés jelentése az összes adatra érvényes, a *talált* változót átkereszteljük *mind*-re.

Algoritmus Döntés_3(N , X , *mind*):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: ha az X sorozatban }
 { minden elem T tulajdonságú, a *mind* értéke igaz, különben hamis }*

$i \leftarrow 1$

Amíg ($i \leq N$) **és** $T(X_i)$ **végezd el:** { *a nem $T(X_i)$ részkiefejezés tagadása* }

$i \leftarrow i + 1$

vége(amíg)

$mind \leftarrow i > N$ { *az $i \leq N$ részkiefejezés tagadása* }

Vége(algoritmus)

1.1.3. Kiválasztás

Adott az N elemű X sorozat és az elemein értelmezett T tulajdonság. Adjuk meg a sorozat egy T tulajdonságú elemének sorszámát! (Előfeltétel: garantáltan létezik ilyen elem.)

Algoritmus Kiválasztás(N , X , *hely*):

{ *Bemeneti adatok: az N elemű X sorozat. }*

{ *Kimeneti adat: hely, a legkisebb indexű T tulajdonságú elem sorszáma* }

hely $\leftarrow 1$

Amíg nem $T(X_{\text{hely}})$ **végezd el:** { *nem szükséges a hely $\leq N$ feltétel, mivel a feladat* }

hely \leftarrow *hely* + 1 { *garantálja legalább egy T tulajdonságú elem létezését* }

vége(amíg)

Vége(algoritmus)

1.1.4. Szekvenciális (lineáris) keresés

Adott az N elemű X sorozat és az elemein értelmezett T tulajdonság. Vizsgáljuk meg, hogy létezik-e T tulajdonságú elem a sorozatban! Ha létezik, akkor adjuk meg az első ilyen elem helyét!

Algoritmus Keres_1(N , X , *hely*):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: hely, a legkisebb indexű T }*

{ *tulajdonságú elem indexe, illetve, sikertelen keresés esetén hely = 0* }

hely $\leftarrow 0$

$i \leftarrow 1$

Amíg (*hely* = 0) **és** ($i \leq N$) **végezd el:**

Ha $T(X_i)$ **akkor**

hely $\leftarrow i$

különben

$i \leftarrow i + 1$

vége(ha)

vége(amíg)

Vége(algoritmus)

Az adott elem tulajdonságát az *Amíg* feltételében is ellenőrizhetjük. Más szóval: amíg az aktuális elem tulajdonsága nem megfelelő, haladunk a sorozatban előre:

Algoritmus Keres_2(N , X , hely):
 { *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: hely, a legkisebb indexű T }*
 { *tulajdonságú elem indexe, illetve, sikertelen keresés esetén hely = 0* }

$i \leftarrow 1$
Amíg ($i \leq N$) és nem $T(X_i)$ végezd el:
 $i \leftarrow i + 1$
vége(amíg)
Ha $i \leq N$ akkor { *ha kiléptünk az Amíg-ból, mielőtt i nagyobbá vált volna N -nél, }*
 hely $\leftarrow i$ { \Rightarrow találtunk adott tulajdonságú elemet az i . pozíción }
 különben
 hely $\leftarrow 0$ { *különbem nem találtunk* }
vége(ha)
Vége(algoritmus)

Ha a feladat azt kéri, hogy keressünk meg minden olyan elemet, amely rendelkezik az adott tulajdonsággal, be kell járnunk a teljes adathalmazt, és vagy kiírjuk azonnal a pozíciókat, ahol megfelelő elemet találtunk, vagy megőrizzük ezeket egy másik sorozatban. Ilyenkor *Minden* típusú struktúrát használunk.

1.1.5. Megszámlálás

Adott, N elemű X sorozatban számoljuk meg a T tulajdonságú elemeket!

Elemzés

Nem biztos, hogy létezik legalább egy T tulajdonságú elem, tehát az is lehetséges, hogy az eredmény 0 lesz. Mivel minden elemet meg kell vizsgálnunk (bármely adat rendelkezhet a kért tulajdonsággal), *Minden* típusú struktúrával dolgozunk. A darabszámot a db változóban tároljuk.

Algoritmus Megszámlálás(N , X , db):
 { *Bemeneti adatok: az N elemű X sorozat* }
 { *Kimeneti adat: db , a T tulajdonságú elemek darabszáma* }

$db \leftarrow 0$
Minden $i = 1, N$ végezd el:
Ha $T(X_i)$ akkor
 $db \leftarrow db + 1$
vége(ha)
vége(minden)
Vége(algoritmus)

1.1.6. Maximumkiválasztás

Adott az N elemű X sorozat. Határozzuk meg a sorozat legnagyobb (vagy legkisebb) értékét!

Megoldás

A megoldásban minden adatot meg kell vizsgálnunk, ezért az algoritmus egy *Minden* típusú struktúrával dolgozik. A max segédváltozó a sorozat első elemétől kap kezdőértéket.

Algoritmus Maximumkiválasztás(N , X , \max):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: \max , a legnagyobb elem értéke* }

$\max \leftarrow X_1$

Minden $i = 2$, n **végezd el:**

Ha $\max < X_i$ **akkor**

$\max \leftarrow X_i$

vége(ha)

vége(minden)

Vége(algoritmus)

A maximumot/minimumot tartalmazó segédváltozónak az adatok közül választunk kezdőértéket, mivel így nem áll fenn a veszély, hogy az algoritmus eredménye egy, az adataink között nem létező érték legyen.

Ha a maximum helyét kell megadnunk, az algoritmus a következő:

Algoritmus MaximumHelye(N , X , hely):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: hely , a legnagyobb elem pozíciója* }

$\text{hely} \leftarrow 1$

{ *hely az első elem pozíciója* }

Minden $i = 2$, n **végezd el:**

Ha $X_{\text{hely}} < X_i$ **akkor**

$\text{hely} \leftarrow i$ { *a maximális elem első előfordulásának helye (pozíciója)* }

vége(ha)

vége(minden)

Vége(algoritmus)

Ha minden olyan indexet meg kell határoznunk, amely indexű elemek egyenlők a legnagyobb elemmel és nem lehetséges/nem előnyös az adott tömböt kétszer bejárni, mert a maximumhoz tartozó adatok egy másik (esetleg bonyolult) algoritmus végrehajtásának eredményei, írhatunk algoritmust, amely csak egyszer járja be a sorozatot:

Algoritmus MindenMaximumHelye(N , X , db , indexek):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: a db elemű indexek sorozat* }

$\max \leftarrow X_1$

$db \leftarrow 1$

$\text{indexek}_1 \leftarrow 1$

Minden $i = 2$, n **végezd el:**

Ha $\max < X_i$ **akkor**

$\max \leftarrow X_i$

$db \leftarrow 1$

$\text{indexek}_{db} \leftarrow i$

különben

Ha $\max = X_i$ **akkor**

$db \leftarrow db + 1$

$\text{indexek}_{db} \leftarrow i$

vége(ha)

vége(ha)

vége(minden)

Vége(algoritmus)

B. Sorozathoz sorozat rendelése

1.1.7. Másolás

Adott az N elemű X sorozat és az elemein értelmezett f függvény. A bemeneti sorozat minden elemére végrehajtjuk a függvényt, az eredményét pedig a kimeneti sorozatba másoljuk.

Algoritmus Másolás(N , X , Y):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az N elemű Y sorozat* }

Minden $i = 1, N$ **végezd el:**

$Y_i \leftarrow f(X_i)$

vége(minden)

Vége(algoritmus)

1.1.8. Kiválogatás

Adott az N elemű X sorozat és az elemein értelmezett T tulajdonság. Válogassuk ki az összes T tulajdonságú elemet!

Elemzés

Az elvárások függvényében különböző megközelítések lesznek érvényesek:

- a. kiválogatás kigyűjtéssel
- b. kiválogatás kiírással
- c. kiválogatás helyben (sorrendváltoztatással vagy megőrizve az eredeti sorrendet)
- d. kiválogatás kihúzással (segédsorozattal vagy helyben)

a. Kiválogatás kigyűjtéssel

A keresett elemeket (vagy sorszámait) kigyűjtjük egy sorozatba. A pozíciók sorozatának (vagy a kigyűjtött elemek sorozatának) hossza legfeljebb az adott sorozatéval lesz megegyező, mivel előfordulhat, hogy a bemeneti sorozat minden eleme adott tulajdonságú. A sorozat számosságát a db változóban tartjuk nyilván.

Algoritmus Kiválogatás_a(N , X , db , pozíciók):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: a db elemű pozíciók sorozat* }

$db \leftarrow 0$

Minden $i = 1, N$ **végezd el:**

Ha $T(X_i)$ **akkor**

$db \leftarrow db + 1$

pozíciók _{db} $\leftarrow i$ { *pozíciók _{db} -ben tároljuk az X_i helyét* }

vége(ha)

vége(minden)

Vége(algoritmus)

b. Kiválogatás kiírással

Ha a feladat „megelégszik” a T tulajdonságú elemek kiírással (nem kéri ezek darabszámát is), az algoritmus a következő:

Algoritmus Kiválogatás_b(N, X):

{ *Bemeneti adatok: az N elemű X sorozat* }

Minden $i = 1, N$ végezd el:

Ha $T(X_i)$ akkor

Ki: X_i

vége(ha)

vége(minden)

Vége(algoritmus)

c. Kiválogatás helyben

Ha a sorozat feldolgozása közben a nem T tulajdonságú elemeket nem óhajtjuk megőrizni, hanem ki szeretnénk zárni ezeket a sorozatból, akkor a feladat specifikációitól függően, a következő lehetőségek közül fogunk választani:

c1. Ha a törlés után nem kötelező, hogy az elemek az eredeti sorrendjükben maradjanak, akkor a törlendő elemre rámásoljuk a sorozat utolsó elemét és csökkentjük 1-gyel a sorozat méretét:

Algoritmus Kiválogatás_c1(N, X):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: a megváltozott elemszámú X sorozat* }

$i \leftarrow 1$

Amíg $i \leq N$ végezd el:

{ *nem alkalmazunk Minden-t, mivel változik az N!!!* }

Ha nem $T(X_i)$ akkor

{ *a T tulajdonságú elemeket tartjuk meg* }

$X_i \leftarrow X_N$

{ *X_i -t felülírjuk X_N -nel* }

$N \leftarrow N - 1$

{ *változik a sorozat hossza* }

különben

$i \leftarrow i + 1$

{ *i csak a különben ágon nő* }

vége(ha)

vége(amíg)

Vége(algoritmus)

c2. Ha az eredeti sorozatra nincs többé szükség, de szeretnénk megőrizni az elemek eredeti sorrendjét, akkor a T tulajdonságú elemeket felsorakoztatjuk a sorozat elejétől kezdve. Így a kiválogatott elemekkel felülírjuk az eredeti adatokat. Nem használunk egy újabb sorozatot, hanem az adott sorozat számára lefoglalt tárrészt használva helyben végezzük a kiválogatást. A db változó ebben az esetben a megváltoztatott sorozatnak a számosságát tartja nyilván:

Algoritmus Kiválogatás_c2(N, X, db):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: a db elemű X sorozat* }

$db \leftarrow 0$

Minden $i = 1, N$ végezd el:

Ha $T(X_i)$ akkor

$db \leftarrow db + 1$

$X_{db} \leftarrow X_i$

vége(ha)

vége(minden)

Vége(algoritmus)

d1. Ha a törlés ideiglenes, akkor a kereséssel párhuzamosan egy logikai tömbben nyilvántartjuk a „törölt” elemeket. A *törölt* tömb elemeinek kezdőértéke *hamis* lesz, majd a törlendő elemeknek megfelelő sorszámú elemek értéke a *törölt* logikai tömbben *igaz* lesz:

Algoritmus Kiválogatás_d1($N, X, \text{törölt}$):
{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az N elemű törölt sorozat* }
Minden $i = 1, N$ végezd el:
 $\text{törölt}_i \leftarrow \text{hamis}$
vége(minden)
Minden $i = 1, N$ végezd el:
 Ha nem $T(X_i)$ akkor { *a T tulajdonságú elemeket tartjuk meg* }
 $\text{törölt}_i \leftarrow \text{igaz}$
 vége(ha)
vége(minden)
Vége(algoritmus)

d2. Egy másik megoldás, amely nem hoz létre új helyen, új sorozatot, helyben végzi a kiválogatást, anélkül, hogy elmozdítaná eredeti helyükről a T tulajdonságú elemeket, a nem T tulajdonságú elemek helyére pedig egy speciális értéket tesz:

Algoritmus Kiválogatás_d2($N, X, \text{törölt}$):
{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az N elemű X sorozat* }
Minden $i = 1, N$ végezd el:
 Ha nem $T(X_i)$ akkor { *a T tulajdonságú elemeket tartjuk meg* }
 $X_i \leftarrow \text{speciális érték}$
 vége(ha)
vége(minden)
Vége(algoritmus)

C. Sorozatokhoz sorozat rendelése

1.1.9. Halmazok

Mielőtt egy *halmazokat* tartalmazó sorozatra vonatkozó műveletet alkalmaznánk, szükséges meggyőződnünk arról, hogy a sorozat valóban *halmaz*. Ez azt jelenti, hogy minden érték csak egyszer fordul elő. Ha kiderül, hogy a sorozat nem halmaz, halmazzá kell alakítanunk.

a. Halmaz-e?

Döntsük el, hogy az adott N elemű X sorozat halmaz-e!

Elemzés

Egy halmaz vagy üres, vagy bizonyos számú elemet tartalmaz. Ha egy halmazt sorozattal implementálunk, az elemei különbözők. A következő algoritmussal eldöntjük, hogy a sorozat csak különböző elemeket tartalmaz-e?

Algoritmus Halmaz_e(N, X, ok):
{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az ok értéke igaz, }*
 $i \leftarrow 1$ { *ha a sorozat halmaz, különben hamis* }
 $\text{ok} \leftarrow \text{igaz}$

Amíg ok és $(i < N)$ **végezd el:**
 $j \leftarrow i + 1$
Amíg $(j \leq N)$ és $(X_i \neq X_j)$ **végezd el:**
 $j \leftarrow j + 1$
vége(amíg)
 $ok \leftarrow j > N$ { ha véget ért a sorozat, nincs két azonos elem }
 $i \leftarrow i + 1$
vége(amíg)
Vége(algoritmus)

b. Halmazzá alakítás

Alakítsuk halmazzá az N elemű X sorozatot!

Elemzés

Ha egy alkalmazásban ki kell zárunk az adott sorozatból a másodsor (harmadszor stb.) megjelenő értékeket, akkor az előbbi algoritmust módosítjuk: amikor egy bizonyos érték megjelenik másodsor, felülírjuk az utolsóval.

Algoritmus HalmazzáAlakít(N, X):

{ *Bemeneti adatok: az N elemű X sorozat.* }

{ *Kimeneti adatok: az új N elemű X sorozat (halmaz)* }

$i \leftarrow 1$
Amíg $i < N$ **végezd el:**
 $j \leftarrow i + 1$
Amíg $(j \leq N)$ és $(X_i \neq X_j)$ **végezd el:**
 $j \leftarrow j + 1$
vége(amíg)
Ha $j \leq N$ **akkor** { *találtunk egy $X_j = X_i$ -t* }
 $X_j \leftarrow X_N$ { *felülírjuk a sorozat N . elemével* }
 $N \leftarrow N - 1$ { *rövidítjük a sorozatot* }
különben
 $i \leftarrow i + 1$ { *haladunk tovább* }
vége(ha)
vége(amíg)
Vége(algoritmus)

1.1.10. Keresztmetszet

Hozzuk létre a bemenetként kapott sorozatok keresztmetszetét!

Elemzés

Keresztmetszet alatt azt a sorozatot értjük, amely az adott sorozatok közös elemeit tartalmazza. Feltételezzük, hogy az adott sorozatok mind különböző elemeket tartalmaznak (halmazok) és nem rendezettek.

Az N elemű X és az M elemű Y sorozat keresztmetszetét a db elemű Z sorozatban hozzuk létre, tehát Z olyan elemeket tartalmaz az X sorozatból, amelyek megtalálhatók az Y -ban is.

Algoritmus Keresztmetszet(N, X, M, Y, db, Z):

{ *Bemeneti adatok: az N elemű X és az M elemű Y sorozat. }*

$db \leftarrow 0$ { *Kimeneti adatok: a db elemű Z sorozat, X és Y keresztmetszete* }

Minden $i = 1, N$ végezd el:

$j \leftarrow 1$

Amíg ($j \leq M$) és ($X_i \neq Y_j$) **végezd el:**

$j \leftarrow j + 1$

vége(amíg)

Ha $j \leq M$ **akkor**

$db \leftarrow db + 1$

$Z_{db} \leftarrow X_i$

vége(ha)

vége(minden)

Vége(algoritmus)

1.1.11. Egyesítés (Unió)

Hozzuk létre az N elemű X és az M elemű Y sorozatok (halmazok) egyesített halmazát!

Elemzés

Az egyesítés algoritmusá hasonló a keresztmetszetéhez. Nem alkalmazhatunk összefésülést, mivel a sorozatok nem rendezettek! A különbség abban áll, hogy olyan elemeket helyezünk az eredménybe, amelyek legalább az egyik sorozatban megtalálhatók. Előbb a Z sorozatba másoljuk az X sorozatot, majd kiválogatjuk Y -ből azokat az elemeket, amelyeket nem találtunk meg X -ben.

Algoritmus Egyesítés(N, X, M, Y, db, Z):

{ *Bemeneti adatok: az N elemű X és az M elemű Y sorozat. }*

{ *Kimeneti adatok: a db elemű Z sorozat (X és Y egyesítése) }*

Másolás(N, X, Z) { *az X sorozat minden elemét átmásoljuk a Z sorozatba* }

$db \leftarrow N$

Minden $j = 1, M$ **végezd el:**

$i \leftarrow 1$

Amíg ($i \leq N$) és ($X_i \neq Y_j$) **végezd el:**

$i \leftarrow i + 1$

vége(amíg)

Ha $i > N$ **akkor**

$db \leftarrow db + 1$

$Z_{db} \leftarrow Y_j$

vége(ha)

vége(minden)

Vége(algoritmus)

1.1.12. Összefésülés

Adott két rendezett sorozatból állítsunk elő egy harmadikat, amely legyen szintén rendezett!

Elemzés

Az *Egyesítés*(N, X, M, Y, db, Z) és a *Keresztmetszet*(N, X, M, Y, db, Z) algoritmusok négyzetes bonyolultságúak, mivel a halmazokat implementáló sorozatok nem rendezettek. Ez a két művelet megvalósítható lineáris algoritmussal, ha a sorozatok rendezettek. Így az eredményt is rendezett formában fogjuk generálni. Ezek a sorozatok nem mindig halmazok, tehát néha előfordulhatnak azonos értékű elemek is.

Elindulunk mindkét sorozatban és a soron következő két elem összehasonlítása révén eldöntjük, melyiket tegyük a harmadikba. Addig végezzük ezeket a műveleteket, amíg valamelyik sorozatnak a végére nem érünk. A másik sorozatban megmaradt elemeket átmásoljuk az eredményssorozatba. Mivel nem tudhatjuk előre melyik sorozat ért véget, vizsgáljuk mindkét sorozatot.

Algoritmus *Összefésülés_1*(N, X, M, Y, db, Z):

{ *Bemeneti adatok: az N elemű X és az M elemű Y sorozat. A sorozatok nem halmazok. }*

{ *Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel) }*

$db \leftarrow 0$

$i \leftarrow 1$

$j \leftarrow 1$

Amíg ($i \leq N$) és ($j \leq M$) **végezd el:** { *amíg sem X -nek, sem Y -nak nincs vége* }

$db \leftarrow db + 1$

Ha $X_i < Y_j$ **akkor**

$Z_{db} \leftarrow X_i$

$i \leftarrow i + 1$

különben

$Z_{db} \leftarrow Y_j$

$j \leftarrow j + 1$

vége(ha)

vége(amíg)

Amíg $i \leq N$ **végezd el:** { *ha maradt még elem X -ben* }

$db \leftarrow db + 1$

$Z_{db} \leftarrow X_i$

$i \leftarrow i + 1$

vége(amíg)

Amíg $j \leq M$ **végezd el:** { *ha maradt még elem Y -ban* }

$db \leftarrow db + 1$

$Z_{db} \leftarrow Y_j$

$j \leftarrow j + 1$

vége(amíg)

Vége(algoritmus)

Most feltételezzük, hogy az egyes sorozatokban egy elem csak egyszer fordul elő és azt szeretnénk, hogy az összefésült új sorozatban se legyenek „duplák”. Az előző algoritmust csak annyiban módosítjuk, hogy vizsgáljuk az egyenlőséget is. Ha a két összehasonlított érték egyenlő, mind a két sorozatban továbblépünk és az aktuális értéket csak egyszer írjuk be az eredményssorozatba.

Algoritmus Összefésülés_2(N, X, M, Y, db, Z):

{ *Bemeneti adatok: az N elemű X és az M elemű Y sorozat. A sorozatok halmazok* }
 { *Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel)* }

$db \leftarrow 0$

$i \leftarrow 1$

$j \leftarrow 1$

Amíg ($i \leq N$) és ($j \leq M$) **végezd el:**

$db \leftarrow db + 1$

Ha $X_i < Y_j$ **akkor**

$Z_{db} \leftarrow X_i$

$i \leftarrow i + 1$

különben

Ha $X_i = Y_j$ **akkor**

$Z_{db} \leftarrow X_i$

$i \leftarrow i + 1$

$j \leftarrow j + 1$

különben

$Z_{db} \leftarrow Y_j$

$j \leftarrow j + 1$

vége(ha)

vége(ha)

vége(amíg)

Amíg $i \leq N$ **végezd el:**

{ *ha maradt még elem X -ben* }

$db \leftarrow db + 1$

$Z_{db} \leftarrow X_i$

$i \leftarrow i + 1$

vége(amíg)

Amíg $j \leq m$ **végezd el:**

{ *ha maradt még elem Y -ban* }

$db \leftarrow db + 1$

$Z_{db} \leftarrow Y_j$

$j \leftarrow j + 1$

vége(amíg)

Vége(algoritmus)

Szerencsés esetben $X_N = Y_M$. Ekkor a két utolsó *Amíg* struktúra nem hajtódott volna végre egyetlen egyszer sem. Kihasználva ezt az észrevételt, elhelyezünk mindkét sorozat végére egy fiktív elemet (*őrszem*). Tehetjük az X sorozat végére az $X_{N+1} = Y_M + 1$ értéket és az Y sorozat végére az $Y_{M+1} = X_N + 1$ értéket. Ha a két egyesítendő sorozat nem halmaz, az eredmény sem lesz halmaz. Észrevesszük, hogy ebben az esetben az eredményssorozat hossza pontosan $N + M$. Az algoritmus ismétlődőstruktúrája *Minden* típusú lesz.

Algoritmus Összefésül_3(N, X, M, Y, db, Z):

{ *Bemeneti adatok: az N elemű X és az M elemű Y sorozat. A sorozatok nem halmazok* }
 { *Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel)* }

$i \leftarrow 1$

$j \leftarrow 1$


```

 $X_{N+1} \leftarrow Y_M + 1$ 
 $Y_{M+1} \leftarrow X_N + 1$ 
Minden  $db = 1, N + M$  végezd el:
  Ha  $X_i < Y_j$  akkor
     $Z_{db} \leftarrow X_i$ 
     $i \leftarrow i + 1$ 
  különben
     $Z_{db} \leftarrow Y_j$ 
     $j \leftarrow j + 1$ 
  vége(ha)
vége(minden)
Vége(algoritmus)

```

Ha a bemeneti sorozatok halmazokat ábrázolnak és az eredménysorozatnak is halmaznak kell lennie, az algoritmus a következőképpen alakul: a *Minden* struktúra helyett *Amíg*-ot alkalmazunk, hiszen nem tudjuk hány eleme lesz az összefésült sorozatnak (az ismétlődő értékek közül csak egy kerül be az új sorozatba). Ugyanakkor, az őrsemek révén az *Amíg* struktúrát addig hajtjuk végre, amíg mindkét sorozat végére nem értünk.

Algoritmus $\text{Összefésül_4}(N, X, M, Y, db, Z)$:
 { *Bemeneti adatok: az N elemű X és az M elemű Y sorozat. A sorozatok halmazok* }
 { *Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel)* }

```

 $db \leftarrow 0$ 
 $i \leftarrow 1$ 
 $j \leftarrow 1$ 
 $X_{N+1} \leftarrow Y_M + 1$ 
 $Y_{M+1} \leftarrow X_N + 1$ 
Amíg  $(i \leq N)$  vagy  $(j \leq M)$  végezd el:
   $db \leftarrow db + 1$ 
  Ha  $X_i < Y_j$  akkor
     $Z_{db} \leftarrow X_i$ 
     $i \leftarrow i + 1$ 
  különben
    Ha  $X_i = Y_j$  akkor
       $Z_{db} \leftarrow X_i$ 
       $i \leftarrow i + 1$ 
       $j \leftarrow j + 1$ 
    különben
       $Z_{db} \leftarrow Y_j$ 
       $j \leftarrow j + 1$ 
  vége(ha)
vége(ha)
vége(amíg)
Vége(algoritmus)

```

D. Sorozathoz sorozatok rendelése

1.1.13. Szétválogatás

Válogassuk szét az adott N elemű X sorozat elemeit adott T tulajdonság alapján!

Elemzés

A *Kiválogatás*(N, X) algoritmus egy sorozatot dolgoz fel, amelyből kiválogat bizonyos elemeket. Kérdés: mi történik azokkal az elemekkel, amelyeket nem válogattunk ki? Lesznek feladatok, amelyek azt kérik, hogy két vagy több sorozatba válogassuk szét az adott sorozatot.

a. Szétválogatás két új sorozatba

Az adott sorozatból létrehozunk két újat: a T tulajdonsággal rendelkező adatok sorozatát, és a megmaradtak sorozatát. Mindkét új sorozatot az eredetivel azonos méretűnek deklaráljuk, mivel nem tudhatjuk előre az új sorozatok valós méretét. (Előfordulhat, hogy valamennyi elem átvándorol valamelyik sorozatba, és a másik üres marad.) A dby és dbz a szétválogatás során létrehozott Y és Z sorozatba helyezett elemek számát jelöli.

Algoritmus Szétválogatás_1(N, X, dby, Y, dbz, Z):

$dby \leftarrow 0$ { *Bemeneti adatok: az N elemű X sorozat.* }

$dbz \leftarrow 0$ { *Kimeneti adat: a dby elemű Y és a dbz elemű Z sorozat* }

Minden $i = 1, N$ végezd el:

Ha $T(X_i)$ akkor

$dby \leftarrow dby + 1$ { *az adott tulajdonságú elemek, az Y sorozatba kerülnek* }

$Y_{dby} \leftarrow X_i$

különben

$dbz \leftarrow dbz + 1$ { *azok, amelyek nem rendelkeznek az* }

$Z_{dbz} \leftarrow X_i$ { *adott tulajdonsággal, a Z sorozatba kerülnek* }

vége(ha)

vége(minden)

Vége(algoritmus)

b. Szétválogatás egyetlen új sorozatba

A feladat megoldható egyetlen új sorozattal. A kiválogatott elemeket az új sorozat első részébe helyezzük (az elsőtől haladva a vége felé), a megmaradtakat az új sorozat végére (az utolsótól haladva az első felé). Nem fogunk ütközni, mivel pontosan N elemet fogunk N helyre „átrendezni”. A megmaradt elemek az eredeti sorozatban elfoglalt relatív pozícióik fordított sorrendjében kerülnek az új sorozatba.

Algoritmus Szétválogatás_2(N, dby, dbz, X, Y):

$dby \leftarrow 0$ { *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az N elemű Y sorozat;* }

$dbz \leftarrow 0$ { *az első dby elem T tulajdonságú, dbz elem pedig nem T tulajdonságú* }

Minden $i = 1, N$ végezd el:

Ha $T(X_i)$ akkor { *a T tulajdonságú elemek az Y sorozatba kerülnek* }

$dby \leftarrow dby + 1$ { *az első helytől kezdődően* }

$Y_{dby} \leftarrow X_i$

```

    különben
        dbz ← dbz + 1    { a többi elem szintén Y-ba kerül, az utolsó helytől kezdődően }
         $Y_{N-dbz+1} \leftarrow X_i$ 
    vége(ha)
vége(minden)
Vége(algoritmus)

```

c) Szétválogatás helyben

Ha a szétválogatás után nincs már szükségünk többé az eredeti sorozatra, a szétválogatás elvégezhető helyben. A tömb első elemét kivesszük a helyéről és megőrizzük egy segédváltozóban. Az utolsó elemtől visszafelé megkeressük az első olyat, amely adott tulajdonságú, s ezt előre hozzuk a kivett elem helyére. Ezután a hátul felszabadult helyre előlről keresünk egy nem T tulajdonságú elemet, s ha találunk, azt hátrateszük. Mindezt addig végezzük, amíg a tömbben két irányban haladva össze nem találkozunk.

Algoritmus Szétválogatás_3(N, X, db):

```

    { Bemeneti adatok: az  $N$  elemű  $X$  sorozat. Kimeneti adatok: az  $N$  elemű  $X$  sorozat; }
    { az első  $e$  elem  $T$  tulajdonságú,  $n - e$  elem pedig nem  $T$  tulajdonságú }
     $e \leftarrow 1$            { balról jobbra haladva az első  $T$  tulajdonságú elem indexe }
     $u \leftarrow N$        { jobbról balra haladva az első nem  $T$  tulajdonságú elem indexe }
    segéd ←  $X_e$ 
    Amíg  $e < u$  végezd el:
        Amíg ( $e < u$ ) és nem  $T(X_u)$  végezd el:
             $u \leftarrow u - 1$ 
        vége(amíg)
        Ha  $e < u$  akkor
             $X_e \leftarrow X_u$ 
             $e \leftarrow e + 1$ 
        Amíg ( $e < u$ ) és  $T(X_e)$  végezd el:
             $e \leftarrow e + 1$ 
        vége(amíg)
        Ha  $e < u$  akkor
             $X_u \leftarrow X_e$ 
             $u \leftarrow u - 1$ 
        vége(ha)
    vége(ha)
vége(amíg)
 $X_e \leftarrow$  segéd           { visszahozzuk a segéd-be tett elemet }
    Ha  $T(X_e)$  akkor  $db \leftarrow e$ 
    különben  $db \leftarrow e - 1$ 
    vége(ha)
Vége(algoritmus)

```

Megjegyzés

Ha egy sorozatot több részsorozatba szükséges szétválogatni több tulajdonság alapján, egymás után több szétválogatást fogunk végezni, mindig a kért tulajdonság alapján.

Előbb szétválogatjuk az adott sorozatból az első tulajdonsággal rendelkezőket, majd a félretett adatokból szétválogatjuk a második tulajdonsággal rendelkezőket és így tovább.

1.1.14. Programozási tételek összeépítése

Az egészen egyszerű alapfeladatokat kivéve általában több programozási tételt kell használnunk. Ilyenkor – ahelyett, hogy simán egymás után alkalmazzuk ezeket, lehetséges egyszerűbb, rövidebb, hatékonyabb, gazdaságosabb algoritmust tervezni, ha összeépítjük őket.

a. Másolással összeépítés

A másolás bármelyik programozási tétellel egybeépíthető. Ilyenkor az X_i bemeneti adatra való hivatkozást $f(x_i)$ -re cseréljük.

Példa:

Adjuk meg egy számsorozat elemeinek négyzetgyökeiből álló sorozatot!

Megoldás: másolás + sorozatszámítás

b. Megszámlálással összeépítés

A megszámlálást általában egy döntéssel, kiválasztással vagy kereséssel építhetjük össze.

Példa:

Döntsük el, hogy található-e az N elemű X sorozatban legalább K darab T tulajdonságú elem? Adjuk meg a sorozat K -dik T tulajdonságú elemét!

Megoldás: megszámlálás + döntés + kiválasztás

c. Maximumkiválasztással összeépítés

A maximumkiválasztást összeépíthetjük megszámlálással, kiválogatással.

Példa:

Hány darab maximumértékű elem van az adott sorozatban? Generáljuk ezen elemek indexeinek sorozatát!

Megoldás: Lásd a *MindenMaximumHelye*($N, X, db, indexek$) algoritmust.

d. Kiválogatással összeépítés

Olyan feladatoknál, amelyeknek esetében a feldolgozást csak az adott sorozat T tulajdonságú elemeire kell elvégeznünk, alkalmazható a kiválogatással történő összeépítés.

1.2. Lépések finomítása és optimalizálás

Bonyolultabb feladatok esetében a megfelelő algoritmus leírása nem könnyű feladat. Ezért célszerű először a megoldást körvonalazni, és csak aztán részletezni. A feladat elemzése során sor kerül a bemeneti és kimeneti adatok megállapítására, a megfelelő adatszerkezetek kiválasztására és megtervezésére, a feladat követelményeinek szétválasztására. Következik a megoldási módszer megállapítása, a megoldás lépéseinek leírása és a *lépések finomítása*, vagyis az algoritmus részletes kidolgozása. Következik a helyesség bizonyítása és a bonyolultság kiértékelése. A program megírását (kódolást) a tesztelés követi.

A lépések finomítása az algoritmus *kidolgozását* jelenti, amely a kezdeti vázlattól a végleges, precízen leírt algoritmusig vezet. Kiindulunk a feladat specifikációjából és fentről lefele tartó tervezési módszert alkalmazva újabb meg újabb változatokat dolgozunk ki, amelyek eleinte még tartalmaznak bizonyos, anyanyelven leírt magyarázó sorokat, amelyeket csak később írunk át standard utasításokkal. Így, az algoritmusnak több egymás utáni változata lesz, amelyek egyre bővülnek egyik változattól a másikig.

1.2.1. Megoldott feladatok

a. Eukleidész algoritmusa

Határozzuk meg két adott természetes szám legnagyobb közös osztóját (*lnko*) és legkisebb közös többszörösét (*lkkt*) Eukleidész algoritmusával.

Algoritmus Eukleidész_1(*a*, *b*, *lnko*, *lkkt*):

@ kiszámítjuk *a* és *b* *lnko*-ját { *Bemeneti adatok: a, b. Kimeneti adatok: lnko, lkkt* }

@ kiszámítjuk *a* és *b* *lkkt*-ét

Vége(algoritmus)

Lépések finomítása: Ki kell dolgoznunk a kiszámítások módját. Ha a két szám egyenlő, akkor *lnko* az *a* szám lesz. Ha *a* kisebb, mint *b*, nincs szükség felcserélésre: az algoritmus elvégzi ezt az első lépésben. Ezután kiszámítjuk *r*-ben *a* és *b* egészosztási maradékát. Ha a maradék nem 0, a következő lépésben *a*-t felülírjuk *b*-vel, *b*-t *r*-rel, és újból kiszámítjuk a maradékot. Addig dolgozunk, amíg a maradék 0-vá nem válik. Az utolsó osztó éppen az *lnko* lesz. Az *lkkt* értékét megkapjuk, ha *a* és *b* szorzatát elosztjuk az *lnko*-val. Az eredeti két szám értékét az algoritmus „tönkreteszi”, ezért szükséges ezeket elmenteni két segédváltozóba (*x* és *y*).

Algoritmus Eukleidész_1(*a*, *b*, *lnko*, *lkkt*):

{ *Bemeneti adatok: a, b. Kimeneti adatok: lnko, lkkt* }

x ← *a* { *szükségünk lesz a és b értékére az lkkt kiszámításakor* }

y ← *b*

r ← *a mod b* { *kiszámítjuk az első maradékot* }

Amíg *r* ≠ 0 **végezd el:** { *amíg a maradék nem 0* }

a ← *b* { *az osztandót felülírjuk az osztóval* }

b ← *r* { *az osztót felülírjuk a maradékkal* }

r ← *a mod b* { *kiszámítjuk az aktuális maradékot* }

vége(amíg)

lnko ← *b* { *lnko egyenlő az utolsó osztó értékével* }

lkkt ← *x*y div lnko* { *felhasználjuk a és b másolatait* }

Vége(algoritmus)

Az algoritmust megvalósíthatjuk ismételt kivonásokkal. Amíg a két szám különbözik egymástól, a nagyobbikból kivonjuk a kisebbiket, és megőrizzük a különbséget. Az *lnko* az utolsó különbség lesz. Az *lkkt*-t ugyanúgy számítjuk ki, mint az előző változatban.

Algoritmus Eukleidész_2(*a*, *b*, *lnko*, *lkkt*):

$x \leftarrow a$ { *Bemeneti adatok: a, b. Kimeneti adatok: lnko, lkkt* }

$y \leftarrow b$

Amíg $a \neq b$ **végezd el:**

Ha $a > b$ **akkor**

$a \leftarrow a - b$

különben

$b \leftarrow b - a$

vége(ha)

vége(amíg)

$lnko \leftarrow a$

$lkkt \leftarrow [x*y/lnko]$

Vége(algoritmus)

b. Prímszámok

Adva van egy nullától különböző természetes n szám. Döntsük el, hogy az adott szám prímszám-e vagy sem!

Algoritmus Prím(n , *válasz*):

{ *Bemeneti adat: n. Kimeneti adat: válasz* }

@ *Megállapítjuk, hogyn prímszám-e*

Ha n *prímszám* **akkor**

válasz \leftarrow igaz

különben

válasz \leftarrow hamis

vége(ha)

Vége(algoritmus)

Lépések finomítása: Ki kell dolgoznunk azt a módot, ahogyan megállapíthatjuk, hogy a szám prímszám-e. A megoldás első változatában a prímszám definíciójából indulunk ki: egy szám akkor prímszám, ha pontosan két osztója van: 1 és maga a szám. Első ötletünk tehát az, hogy az algoritmus számolja meg az adott n szám osztóit, elosztva ezt sorban minden számmal 1-től n -ig. A döntésnek megfelelő üzenetet az osztók száma alapján írjuk ki.

Algoritmus Prím(n , *válasz*):

osztók_száma \leftarrow 0

{ *Bemeneti adat: n. Kimeneti adat: válasz* }

Minden $osztó = 1, n$ **végezd el:**

Ha $n \bmod osztó = 0$ **akkor**

osztók_száma \leftarrow *osztók_száma* + 1

vége(ha)

vége(minden)

válasz \leftarrow *osztók_száma* = 2

Vége(algoritmus)

1.2.2. Az algoritmus optimalizálása

A lépésenkénti finomításnak elvben vége van, hiszen van egy helyesen működő algoritmusunk. De, miután teszteljük és figyelmesen elemezzük, rájövünk, hogy az algoritmust lehetséges *optimalizálni*. Észre vesszük, hogy az osztások száma fölöslegesen nagy. Ezt a számot lehet csökkenteni, mivel ha 2 és $n/2$ között nincs egyetlen osztó sem, akkor biztos, hogy nincs $n/2$ és n között sem, tehát eldönthető, hogy a szám prím. Sőt elég a szám négyzetgyökéig keresni a lehetséges osztót, hiszen ahogy az osztó értékei nőnek a négyzetgyökig, az $[n/osztó]$ hányados értékei csökkennek szintén a négyzetgyök értékéig. Ha egy, a négyzetgyöknél nagyobb osztóval elosztjuk az adott számot, hányadosként egy kisebb osztót kapunk, amit megtaláltunk volna előbb, ha létezett volna ilyen. Továbbá, a ciklus leállítható amint találtunk egy osztót és a *válasz* hamissá vált. A *Minden* típusú ciklust *Amíg* vagy *Ismételd* típusú ciklussal helyettesítjük. Mivel n nem változik a ciklus magjában, a négyzetgyök kiszámíttatását csak egyszer végezzük el. Azt is tudjuk, hogy az egyetlen páros prímszám a 2. Így elérhetjük, hogy a páros számok lekezelése után csak páratlan számokat vizsgáljunk, és ezeket csak páratlan osztókkal próbáljuk meg elosztani. Ahhoz, hogy az algoritmusunk tökéletesen működjön akkor is, ha $n = 1$, a következőképpen járunk el:

Algoritmus Prím(n , válasz):

{ *Bemeneti adat: n. Kimeneti adat: válasz* }

Ha $n = 1$ **akkor**

prím \leftarrow hamis

különben

Ha n *páros* **akkor**

prím $\leftarrow n = 2$

különben

prím \leftarrow igaz

osztó $\leftarrow 3$

négyzetgyök $\leftarrow [\sqrt{n}]$ { *a négyzetgyök egész része* }

Amíg prím és (osztó \leq négyzetgyök) **végezd el:**

Ha $n \bmod$ osztó = 0 **akkor**

prím \leftarrow hamis

különben

osztó \leftarrow osztó + 2

vége(ha)

vége(amíg)

vége(ha)

vége(ha)

válasz \leftarrow prím

Vége(algoritmus)

Ha ebben az algoritmusban felhasználjuk a matematikából ismert tulajdonságot, éspedig: minden 5-nél nagyobb prímszám $6k \pm 1$ alakú, akkor a vizsgálandó számok száma tovább csökkenthető. Mivel az előbbi állításból következik, hogy prímszámokat keresni csak 6 többszöröseinél 1-gyel kisebb, illetve 1-gyel nagyobb számok között érdemes, a fenti algoritmus a következőképpen változik:

Algoritmus Prím(határ):

Ha $n = 1$ **akkor**

prím \leftarrow hamis

különben

Ha n páros **akkor**

prím \leftarrow $n = 2$

különben

Ha $n \leq 5$ **akkor**

{ $n = 3$ }

prím \leftarrow igaz

különben

Ha $((n - 1) \bmod 6 \neq 0)$ és $((n + 1) \bmod 6 \neq 0)$ **akkor**

prím \leftarrow hamis

különben

osztó \leftarrow 3

...

{ tovább ugyanaz, mint az előző algoritmusban }

Továbbá, ismeretes, hogy a négyzetgyököt számoló függvény ismeretlen lépésszámban határozza meg az eredményt, amely valós szám. Ezt elkerülendő, lemondunk a négyzetgyök kiszámításáról és az *Amíg* feltételét a következőképpen írjuk:

...

Amíg prím és $(osztó * osztó \leq n)$ **végezd el:**

...

Így, nem dolgozunk valós számokkal és nem számítjuk ki fölöslegesen a négyzetgyököt.

Ha sok számról kell eldöntenünk, hogy prím-e, érdemes előbb létrehozni *Eratoszthenész* szita-módszerével prímszámok sorozatát (megfelelő darabszámmal) és az algoritmusban csak ennek a sorozatnak elemeivel osztani.

Algoritmus Prímek(határ, prím):

{ határ-nál kisebb számokat vizsgálunk }

{ a generált prímszámokat a prím logikai tömb alapján lehet értékesíteni }

Minden $i=2, \text{határ}$ **végezd el:**

prím _{i} \leftarrow igaz

{ még nincs kihúzva egy szám sem }

vége(minden)

Minden $i = 2, i * i < \text{határ}$ **végezd el:**

Ha prím _{i} **akkor**

{ ha i még nincs kihúzva }

$k \leftarrow 2 * i$

{ az első kihúzandó szám (i -nek többszöröse) }

Amíg $k \leq \text{határ}$ **végezd el:**

prím _{k} \leftarrow hamis

{ kihúzzuk a k számot }

$k \leftarrow k + i$

{ a következő kihúzandó többszöröse i -nek }

vége(amíg)

vége(ha)

vége(minden)

Vége(algoritmus)

1.2.3. A moduláris programozás alapszabályai

Az eredeti feladatot részfeladatokra bontjuk. Minden rész számára megtervezzük a megoldást jelentő algoritmust. Ezek az algoritmusok legyenek minél függetlenebbek, de álljanak jól definiált kapcsolatban egymással. A részfeladatok megoldásainak összessége tartalmazza a feladat megoldási algoritmusát.

Moduláris dekompozíció: A moduláris dekompozíció a feladat több, egyszerűbb részfeladatra bontását jelenti, amely részfeladatok megoldása már egymástól függetlenül elvégezhető. A módszert általában ismételten alkalmazzuk, azaz a részfeladatokat magukat is felbontjuk. Ezzel lehetővé tesszük azt is, hogy a feladat megoldásán egyszerre több személy is dolgozzon. A módszer egy fával ábrázolható, ahol a fa csomópontjai az egyes dekompozíciós lépéseknek felelnek meg.

Moduláris kompozíció: Olyan szoftverelemek létrehozását támogatja, amelyek szabadon kombinálhatók egymással. Algoritmusainkat a már meglévő egységekből építjük fel.

Modulok tulajdonságai

Moduláris érthetőség: A modulok önállóan is egy-egy értelmes egységet alkossanak, megértésükhöz minél kevesebb „szomszédos” modulra legyen szükség.

Moduláris folytonosság: A specifikáció „kis” változtatása esetén a programban is csak „kis” változtatásra legyen szükség.

Moduláris védelem: Célunk a program egészének védelme az abnormális helyzetek hatásaitól. Egy hiba hatása egy – esetleg néhány – modulra korlátozódjon!

A modularitás alapelvei

- *A modulokat nyelvi egységek támogassák:* A modulok illeszkedjenek a használt programozási nyelv szintaktikai egységeihez.
- *Kevés kapcsolat legyen:* Minden modul minél kevesebb másik modullal kommunikáljon!
- *Gyenge legyen a kapcsolat:* A modulok olyan kevés információt cseréljenek, amennyi csak lehetséges!
- *Explicit interface használata:* Ha két modul kommunikál egymással, akkor annak ki kell derülnie legalább az egyikük szövegéből.
- *Információ elrejtés:* Egy modul minden információjának rejtettnek kell lennie, kivéve, amit explicit módon nyilvánosnak deklaráltunk.
- *Nyitott és zárt modulok:* Egy modult *zárt*nak nevezünk, ha más modulok számára egy jól definiált felületen keresztül elérhető, a többi modul ezt változatlan formában felhasználhatja. Egy modult *nyitott*nak nevezünk, ha még kiterjeszhető, ha az általa nyújtott szolgáltatások bővíthetők vagy, ha hozzávehetünk további mezőket a benne levő adatstruktúrákhoz, s ennek megfelelően módosíthatjuk eddigi szolgáltatásait.

Az újrafelhasználhatóság igényei

A típusok változatossága: A moduloknak többféle típusra is működniük kell, azaz a műveleteket több különböző típusra is definiálni kellene.

Egy típus, egy modul: Egy típus műveletei kerüljenek egy modulba.

1.3. Rendező algoritmusok

Összehasonlításra alapuló rendezések

Legyen egy n elemű a sorozat. Növekvően rendezett sorozatnak nevezzük a bemeneti sorozatot olyan permutációját, amelyben $a_1 \leq a_2 \leq \dots \leq a_n$.

1.3.1. Buborékrendezés (*Bubble-sort*)

A rendezés során páronként összehasonlítjuk a számokat és, ha a sorrend nem megfelelő, akkor az illető két elemet felcseréljük. Ha volt csere, a vizsgálatot újratekintjük. Az algoritmus akkor ér véget, amikor az elemek páronként a megfelelő sorrendben találhatók, vagyis a sorozat rendezett. Mivel a sorozat első bejárása után legalább az utolsó elem a helyére kerül, és a ciklusmag minden újabb végrehajtása után, jobbról balra haladva újabb elemek kerülnek a megfelelő helyre, a ciklus lépésszáma csökkenthető. Az is előfordulhat, hogy a sorozat végén levő elemek már a megfelelő sorrendben vannak, és így azokat már nem kell rendeznünk. Tehát, elegendő a sorozatot csak az utolsó csere helyéig vizsgálni.

Algoritmus BuborékRendezés(n, a):

```

k ← n                                { Bemeneti adatok: n, a. Kimeneti adat: a rendezett a sorozat }
Ismételd
  nn ← k - 1
  rendben ← igaz
  Minden i = 1, nn végezd el:
    Ha  $a_i > a_{i+1}$  akkor
      rendben ← hamis
       $a_i \leftrightarrow a_{i+1}$ 
      k ← i                                { az utolsó csere helye }
    vége(ha)
  vége(minden)
ameddig rendben
Vége(algoritmus)

```

1.3.2. Egyszerű felcseréléses rendezés

Ez a rendezési módszer hasonlít a buborékrendezéshez, de kötelezően elvégez minden páronkénti összehasonlítást (míg a buborékrendezés bonyolultsága a legjobb esetben $\Omega(n)$, ez az algoritmus mindig $O(n^2)$ bonyolultságú). Ha egy elempár sorrendje nem megfelelő, felcseréli őket.

Algoritmus FelcserélésesRendezés(n, a):

```

                                { Bemeneti adatok: n, a; Kimeneti adat: a rendezett a sorozat }
Minden i = 1, n - 1 végezd el:
  Minden j = i + 1, n végezd el:
    Ha  $a_i > a_j$  akkor
       $a_i \leftrightarrow a_j$ 
    vége(ha)
  vége(minden)
vége(minden)
Vége(algoritmus)

```

1.3.3. Minimum/maximum kiválasztásra épülő rendezés

Növekvő sorrendbe rendezés esetén kiválaszthatjuk a sorozat legkisebb elemét. Ezt az első helyre tesszük úgy, hogy felcseréljük az első helyen található elemmel. A következő lépésben hasonlóan járunk el, de a minimumot a második helytől kezdődően keressük. A továbbiakban ugyanezt tesszük, míg a sorozat végére nem érünk.

Algoritmus MinimumkiválasztásosRendezés(n, a):

{ *Bemeneti adatok: n, a ; Kimeneti adat: a rendezett a sorozat* }

Minden $i = 1, n-1$ végezd el:

$indMin \leftarrow i$

Minden $j = i+1, n$ végezd el:

 Ha $a_{indMin} > a_j$ akkor

$indMin \leftarrow j$

 vége(ha)

 vége(minden)

$a_i \leftrightarrow a_{indMin}$

 vége(minden)

Vége(algoritmus)

1.3.4. Beszűrő rendezés

A beszűrő rendezés hatékony algoritmus kisszámú elem rendezésére. Úgy dolgozik, ahogy bridzsezés közben a kezünkben levő lapokat rendezzük: üres bal kézzel kezdünk, a lapok fejjel lefelé az asztalon vannak. Felveszünk egy lapot az asztalról, és elhelyezzük a bal kezünkben a megfelelő helyre. Ahhoz, hogy megtaláljuk a megfelelő helyet, a felvett lapot összehasonlítjuk a már kezünkben levő lapokkal, jobbról balra. A bemeneti elemek helyben rendeződnek: a számokat az algoritmus az adott tömbön belül rakja a helyes sorrendbe, belőlük bármikor legfeljebb csak állandónyi tárolódik a tömbön kívül. Amikor a rendezés befejeződik, az eredeti tömb tartalmazza a rendezett elemeket.

Algoritmus BeszűrőRendezés(n, a):

{ *Bemeneti adatok: n, a . Kimeneti adat: a rendezett a sorozat* }

Minden $j = 2, n$ végezd el:

$segéd \leftarrow a_j$ { *beszűrjük a_j -t az a_1, \dots, a_{j-1} rendezett sorozatba* }

$i \leftarrow j - 1$

Amíg $(i > 0)$ és $(a_i > segéd)$ végezd el:

$a_{i+1} \leftarrow a_i$

$i \leftarrow i - 1$

 vége(amíg)

$a_{i+1} \leftarrow segéd$

 vége(minden)

Vége(algoritmus)

Lineáris rendezések

Az eddigiekben tárgyalt algoritmusok a legrosszabb esetben $O(n^2)$ időben rendeznek n elemet. Ezek az algoritmusok a rendezéshez csak a bemeneti tömb elemein történő összehasonlításokat használják, ezért ezeket az algoritmusokat *összehasonlító rendezéseknek* is nevezzük.

1.3.5. Leszámláló rendezés (ládarendezés, *Binsort*)

A most következő rendező algoritmus *lineáris* idejű. Ez az algoritmus nem az összehasonlítást használja a rendezéshez, hanem kihasználja a rendezendő sorozat bizonyos tulajdonságait, és pedíg azt, hogy az elemek sorszámozható típusúak, olyan értékekkel, amelyek egy segéd tömb indexei lehetnek.

A segéd tömb i -edik elemében azt tartjuk nyilván, hogy hány darab i -vel egyenlő elemet találtunk az eredeti tömbben. A lineáris feldolgozás után felülírjuk az eredeti tömb elemeit a segéd tömb elemeinek értékei alapján.

Algoritmus LádaRendezés(a, n):

```

Minden  $i = 1, k$  végezd el:           { Bemeneti adatok:  $n, a$ ; Kimeneti adat:  $a$  }
     $segéd_i \leftarrow 0$ 
vége(minden)
Minden  $j = 1, n$  végezd el:
     $segéd_{a_j} \leftarrow segéd_{a_j + 1}$ 
vége(minden)
 $q \leftarrow 0$ 
Minden  $i = 1, k$  végezd el:           {  $a$  segéd tömbnek  $k$  eleme van }
    Minden  $j = 1, segéd_i$  végezd el:
         $q \leftarrow q + 1$            {  $a$  segéd $_i$  elemek összege  $n$  }
         $a_q \leftarrow i$            { tehát a feldolgozások száma  $n$  }
    vége(minden)
vége(minden)
Vége(algoritmus)

```

1.3.6. Számjegyes rendezés (*radixsort*)

Ha egész számokat tároló sorozatot szeretnénk rendezni, elképzelhetjük a számokat egymás alá írva és alkalmazhatjuk a fenti algoritmust rendre, minden számjegy-oszlopra. Ha a legnagyobb szám számjegyeinek darabszáma d , a sorozatot d -szer vizsgáljuk. A számjegyes rendezés először a legkevésbé fontos számjegy alapján rendez. A számokat az utolsó számjegyük alapján rendezzük oly módon, hogy ha csak ezt a számjegyet tekintjük, növekvő sorrendet lássunk. Ezután a számokat újra rendezzük a második legkevésbé értékes számjegyük alapján. Ezt mindaddig végezzük, ameddig a számokat mind a d számjegy szerint nem rendeztük.

Algoritmus SzámjegyesRendezés(a, d):

```

Minden  $i = 1, d$  végezd el:
    stabil leszámlálással rendezzük az  $a$  tömböt az  $i$ -edik számjegy szerint
    vége(minden)
Vége(algoritmus)

```

1.4. A visszalépéses keresés módszere (backtracking)

Úgy a visszalépéses keresést, mint az oszd meg és uralkodj módszert alkalmazó algoritmusok implementálhatók iteratívan és rekurzívan. Mivel a következőkben főleg rekurzív algoritmusok következnek, lássuk mi a rekurzió lényege.

1.4.1. Rekurzió

Bármely algoritmus megvalósítható *iteratívan és/vagy rekurzívan*. Mindkét technikának a lényege: *bizonyos utasítások ismételt végrehajtása*. Az iteratív algoritmusokban az ismétlést *ciklusokkal* valósítjuk meg. A rekurzív algoritmusokban az ismétlés azáltal valósul meg, hogy az illető alprogram *meghívja önmagát*, amikor még aktív.

A rekurzió egy különleges programozási stílus, inkább „technika” mint módszer. A rekurzív programok tömören és világosan kódolják az algoritmusokat, bonyolultságuktól függetlenül. A rekurzív programozás, mint fogalom, a matematikai értelmezéshez közelálló módon került közhasználatba.

Rekurzív algoritmust akkor érdemes tervezni, ha a feladat *eredménye rekurzív szerkezetű*, ha a megoldás legjobb módszere a visszalépéses keresés (*backtracking*) vagy az *oszd meg és uralkodj* módszer (*divide et impera*), illetve ha *a feldolgozandó adatok rekurzívan definiáltak* (pl. bináris fák). Ugyanakkor előfordulhat, hogy túlságosan *igénybe veszi a végrehajtási vermet*, és a *futási ideje nagyobb*, mint az iteratív változatnak

Példák

1. A matematikában, egy fogalmat rekurzív módon definiálunk, ha a definíción belül felhasználjuk magát a definiálandó fogalmat. Például, a faktoriális rekurzív definícióját egy adott n szám esetében, a matematikus így fejezi ki:

$$n! = \begin{cases} 1, & \text{ha } n = 0 \\ n \cdot (n-1)!, & \text{ha } n \in \mathbb{N}^* \end{cases}$$

2. A bináris fa Knuth által megfogalmazott definíciója már szorosan kapcsolódik az informatikához: *Egy bináris fa vagy üres, vagy tartalmaz egy csomópontot, amelynek van egy bal meg egy jobb utóda, amelyek szintén bináris fák.*

A programozásban a rekurzió alprogramok formájában jelenik meg, és pedig olyan függvényeket, illetve eljárásokat nevezünk rekurzívoknak, melyek meghívják önmagukat. Ha ez a hívás az illető alprogram összetett utasításában szerepel, *közvetlen* (direkt) rekurzióról beszélünk. Ha egy rekurzív alprogramot egy másik alprogram hív meg, amelyet ugyanakkor az illető alprogram hív (közvetve, vagy közvetlenül) akkor *közvetett* (indirekt) rekurzióról beszélünk. Közvetett rekurzió esetén is arról van szó, hogy egy alprogram meghívja önmagát, hiszen a rekurzív hívás aközben történik, miközben a számítógép azt az összetett utasítást hajtja végre, amely az illető alprogramot alkotja.

Egy alprogram aktív a hívásától kezdődően, addig, amíg a végrehajtás visszatér a hívás helyére. Egy alprogram aktív marad akkor is, ha végrehajtása során más alprogramokat hív meg.

Egy rekurzív alprogram végrehajtása azonos módon történik, mint bármely nem rekurzív alprogramé. A rekurzív eljárások esetében is, hasonlóan a nem rekurzívakhoz, az aktiválás fel-

tételezi a veremhasználatot, ahol a paramétereket, a visszatérés helyének címét, valamint a lokális változókat tárolja (minden aktuális aktiválás idejére) a programozási környezet.

Mivel a verem mérete véges, bizonyos számú aktiválás után bekövetkezhet a túlsordulás és a program hibaüzenettel kilép. Mivel ezt a hibát feltétlenül el kell kerülnünk, a rekurzív alprogramot csak egy bizonyos feltétel teljesülésekor hívjuk meg újra. A legutolsó aktiválás alkalmával a feltétel hamis, ennek következtében nem történik újrahívás, hanem a feltétel másik ágának megfelelő utasítás (ennek hiányában, a feltétel utáni utasítás) kerül sorra. Új aktiválás csak az újrahívási feltétel teljesülésekor történik. Az újrahívások száma meghatározza a rekurzió mélységét, tehát, egy rekurzív megoldás csak akkor hatékony, ha ez a mélység nem túl nagy.

Ha az újrahívási feltétel egy adott pillanatban nem teljesül, az újraaktiválások sora leáll; ennek következtében a feltétel tagadása a rekurzióból való kilépés feltétele. A feltételnek a rekurzív eljárás paramétereitől kell függnie és/vagy a helyi változóktól, a kilépést a paraméterek és a lokális változók módosulása (egyik hívástól a másikig) biztosítja. Ha ezeket a feltételeket nem tartjuk be, a program hibaüzenettel kilép. Egy újrahívás (közvetlen rekurzió esetén), többször is előfordulhat egy rekurzív eljárásban; ebben az esetben, természetesen, különbözni fognak a visszatérési címek.

A rekurzió késlelteti az eljárás azon utasításainak végrehajtását, amelyek a rekurzív hívás utáni részhez tartoznak. Minden eddigi állítás igaz a rekurzív függvények esetében is, csak a hívás módja más. Egy rekurzív függvényt egy kifejezésből hívunk meg.

1.4.2. Megoldott feladatok

1. Egy szó betűinek megfordítása

Olvassunk be egymás után több betűt a szóköz karakter megjelenéséig, majd írjuk ki ezeket a betűket fordított sorrendben. Ne használjunk tömböt!

Megoldás

A feladat követelményének megfelelően betűk szintjén fogunk dolgozni. A megfordított kiírás azt jelenti, hogy miután beolvastunk egy betűt, nem írjuk ki, csak miután beolvastuk a többi betűt. A fennmaradt rész esetében ugyanígy járunk el; a módszer addig folytatódik, amíg eljutunk az utolsó betűhöz, amikor nincs mit megfordítani.

Algoritmus Fordít: { *nincs paraméter, mivel az alprogramban olvasunk be és írunk ki* }

Be: betű

Ha nem szóköz akkor

Fordít { *meghívja önmagát, hogy megfordíthassa a fennmaradt részt* }

különben

Ki: 'Fordított szó: ' { *ez az utasítás egyszer hajtódik végre* }

vége(ha)

Ki: betű

Vége(algoritmus)

A rekurzió meghatározza az eljárás záró részének az aktiválások fordított sorrendjében való végrehajtását (a fenti példában: **Ki:** betű), így természetes módja a feladat megoldásának.

2. Szavak sorrendjének megfordítása

Olvassunk be n szót, majd írjuk ki ezeket (tömbhasználat nélkül) a beolvasás fordított sorrendjében!

Algoritmus SzavakatFordít_1(n):

Be: szó { az első hívás aktuális paramétere $n = \text{szavak száma}$ }

Ha $n > 1$ **akkor**

 SzavakatFordít_1($n-1$)

különben

Ki: 'Fordított sorrendben: '

vége(ha)

Ki: szó

Vége(algoritmus)

Az eredeti feladat n szó megfordítását valósítja meg, a részfeladatok pedig egyre kevesebb szó megfordítását végzik. Ha fordítva indulunk, vagyis „megfordítjuk” egy szónak a sorrendjét, majd a többiét, akkor az algoritmus a következő:

Algoritmus SzavakatFordít_2(i):

Be: szó { most az első hívás aktuális paramétere 1 }

Ha $i < n$ **akkor**

 SzavakatFordít_2($i+1$)

különben

Ki: 'Fordított sorrendben: '

vége(ha)

Ki: szó

Vége(algoritmus)

3. Faktoriális

Számítsuk ki az adott n szám faktoriálisát!

Megoldás

Felhasználjuk a faktoriális matematikai definícióját, amit a $Fakt(n)$ alprogramban implementálunk. Az első hívás $Fakt(n)$ -nel történik.

Algoritmus Fakt(n):

{ Bemeneti adat: n }

Ha $n = 0$ **akkor**

térítsd 1

különben

térítsd $n * Fakt(n - 1)$

vége(ha)

Vége(algoritmus)

A faktoriális tulajdonképpeni kiszámolása akkor történik, amikor kilépünk egy-egy hívásból. Mivel minden egyes alkalommal más-más n paraméterre van szükség, fontos, hogy ezt értéként adjuk át, így kifejezéseket is írhatunk az aktuális paraméter helyére. Megjegyzendő,

hogy a faktoriális nem előnyös rekurzívan számolni, mivel sokkal időigényesebb, mint az iteratív megoldás, hiszen a $Fakt(n)$ függvény $(n+1)$ -szer fog aktiválódni.

4. Legnagyobb közös osztó

Számítsuk ki két természetes szám $(n, m \in \mathbb{N}^*)$ legnagyobb közös osztóját rekurzívan.

Megoldás

Ha figyelmesen elemezzük *Eukleidész* algoritmusát, észrevesszük, hogy a legnagyobb közös osztó $(Lnko(m, n))$ egyenlő n -nel (ha n osztója m -nek) különben egyenlő $Lnko(n, m \bmod n)$ -nel. Tehát fel lehet írni a következő rekurzív definíciót:

$$Lnko(m, n) = \begin{cases} n, & \text{ha } m \bmod n = 0 \\ Lnko(n, m \bmod n), & \text{ha } m \bmod n \neq 0 \end{cases}$$

Algoritmus Lnko(m, n): { Bemeneti adatok: m, n }

```
mar ← m mod n
Ha mar = 0 akkor
    térítsd n
különben
    térítsd Lnko(n, mar)
vége(ha)
```

Vége(algoritmus)

Az első hívás történhet például egy kiíró utasításból: **Ki**: Lnko(m, n).

5. Descartes-szorzat

Egy rajzon n virágot szeretnénk kiszínezni. A festékeket az 1, 2, ..., m számokkal kódoljuk. Bármely virág, bármilyen színű lehet, de szeretnénk tudni, hány féle módon lehetne ezeket különböző módon kiszínezni. Tulajdonképpen az M^n Descartes-szorzatot kell generálnunk:

Algoritmus DescartesSzorzat(i): { Bemeneti adat: i, az első híváskor = 1 }

```
Minden j = 1, m végezd el:
    xi ← j
    Ha i < n akkor
        DescartesSzorzat(i+1)
    különben
        Kiír
    vége(ha)
```

vége(minden)

Vége(algoritmus)

6. k elemű részhalmazok

Adott két egész szám: n és k ($1 \leq k \leq n$). Generáljuk rekurzívan az $\{1, 2, \dots, n\}$ halmaz minden k elemet tartalmazó részhalmazát!

Megoldás

Az $\{1, 2, \dots, n\}$ halmaz k elemet tartalmazó részhalmaza egy k elemű tömb segítségével kódolható: x_1, x_2, \dots, x_k . A részhalmaz elemei különbözők és nem számít a sorrendjük. Ezért, a részhalmazok generálása során vigyázunk, hogy az x sorozatba ne generáljuk kétszer vagy többször ugyanazt a részhalmazt (esetleg, más sorrendű elemekkel), ugyanakkor ne veszítsünk el egyet sem. Ha az x sorozatba az elemeket szigorúan növekvő sorrendben tesszük ($x_1 < x_2 < \dots < x_k$), egy részhalmazt csak egyszer állíthatunk elő. Mivel minden x_i szigorúan nagyobb, mint x_{i-1} , az értékei $x_{i-1} + 1$ -től kezdődően $n - (k - i)$ -ig nőnek.

Algoritmus Részhalmazok(i): { k és x globális változó, $x_i = 0, i = 0, 1, \dots$ }

Minden $j = x_{i-1} + 1, n - k + i$ **végezd el:**

$x_i \leftarrow j$

Ha $i < k$ **akkor**

 Részhalmazok($i+1$)

különb

 Kiír

vége(ha)

vége(minden)

Vége(algoritmus)

A részhalmazokat generáló algoritmust az i paraméter 1 értékére hívjuk meg.

7. Fibonacci-sorozat

Generáljuk a Fibonacci-sorozat első n elemét!

$$Fib(n) = \begin{cases} 0, & \text{ha } n = 1 \\ 1 & \text{ha } n = 2 \\ Fib(n-1) + Fib(n-2), & \text{ha } n \geq 3 \end{cases}$$

Megoldás

Az n -edik elem kiszámításához szükségünk van az előtte található két elemre. De ezeket szintén az előttük levő elemekből számítjuk ki.

Algoritmus Fibo(n):

Ha $n = 1$ **akkor**

 térítsd 0

különb

Ha $n = 2$ **akkor**

 térítsd 1

különb

 térítsd $Fibo(n-2) + Fibo(n-1)$

vége(ha)

vége(ha)

Vége(algoritmus)

A fenti algoritmus nagyon sokszor hívja meg önmagát ugyanarra az értékre, mivel minden új elem kiszámításakor el kell jutnia a sorozat első eleméhez, amitől kezdődően újra, meg újra generálja ugyanazokat az elemeket. A hívások számát csökkenthetjük, ha a kiszámolt értékeket megőrizzük egy sorozatban. Legyen ez a sorozat f , amelyet globális változóként kezelünk.

Algoritmus Fib(n):

Ha $n > 2$ **akkor**

Fib($n-1$)

$f_n \leftarrow f_{n-1} + f_{n-2}$

különben

$f_1 \leftarrow 0$

Ha $n = 2$ **akkor** $f_2 \leftarrow 1$ **vége(ha)**

vége(ha)

Vége(algoritmus)

8. Az $\{1, 2, \dots, n\}$ halmaz minden részhalmaza

Generáljuk az $\{1, 2, \dots, n\}$ halmaz minden részhalmazát!

Elemzés

A halmazokat az $x_1 < x_2 < \dots < x_i$ sorozattal ábrázoljuk, ahol $i = 1, 2, \dots, n$. Az alábbi algoritmust $i = 1$ -re hívjuk meg. Az x sorozat 0 indexű elemét 0 kezdőértékkel látjuk el. Szükségünk lesz az x_0 elemre is, mivel az algoritmusban a sorozat minden x_i elemét, tehát x_1 -et is az előző elemből számítjuk ki. A j változóban generáljuk azokat az értékeket, amelyeket rendre felvesz az x sorozat aktuális eleme. Ezek a j értékek 1-gyel nagyobbak, mint a részhalmazba utoljára betett elem értéke és legtöbb n -nel egyenlők. Így a részhalmazokat lexikográfikus sorrendben generáljuk. Figyelemre méltó, hogy minden új elem generálása egy új részhalmazhoz vezet.

Algoritmus MindenRészalmaz(i)

Minden $j = x_{i-1} + 1$, n **végezd el:**

$x_i \leftarrow j$

Kiír(i)

MindenRészalmaz($i+1$)

vége(minden)

Vége(algoritmus)

A kilépési feltétel lehetne $x_i = n$, de erre nincs szükség, mivel a *Minden* struktúra végső értéke leállítja a végrehajtást: ha $x_i = n$, a ciklusváltozó kezdőértéke $x_i + 1 = n + 1$, tehát nagyobb, mint n (végső érték), így a *Minden* ciklusmagja nem hajtódik végre és a program kilép az aktuális hívásból. Az algoritmust *MindenRészalmaz*(1) alakban hívjuk meg.

9. Partíciók

Generáljuk az $n \in \mathbb{N}^*$ szám partícióit!

Megoldás

Partíció alatt azt a felbontást értjük, amelynek során az $n \in \mathbb{N}^*$ számot pozitív számok összegként írjuk fel: $n = p_1 + p_2 + \dots + p_k$, ahol $p_i \in \mathbb{N}^*$, $i = 1, 2, \dots, k$, $k = 1, \dots, n$. Két partíciót kétféleképpen tekinthetünk különbözőnek: ha vagy az *előforduló értékek* vagy az *előfordulásuk sorrendje* különbözik vagy, ha csak az *előforduló értékek* különböznek.

A generálás során, rendre kiválasztunk egy lehetséges értéket a partíció első p_1 eleme számára és generáljuk a fennmaradt $n - p_1$ szám partícióit. Ez a különbség az n új értéke lesz, amellyel ugyanúgy járunk el. Egy partíciót legeneráltunk, és kiírhatjuk, ha n aktuális értéke 0.

Az alábbi algoritmust a *Partíció*(1, n) utasítással hívjuk meg először.

Algoritmus Partíció(i, n):
Minden $j = 1, n$ végezd el:
 $p_i \leftarrow j$
Ha $j < n$ **akkor**
 Partíció($i+1, n-j$)
különben
 Kiír(i)
vége(ha)
vége(minden)
Vége(algoritmus)

Az algoritmusok behatóbb tanulmányozása meggyőzött bennünket, hogy tervezésükkor meg kell vizsgálnunk a végrehajtásukhoz szükséges időt. Ha ez az idő elfogadhatatlanul nagy, más megoldásokat kell keresnünk. Egy algoritmus „elfogadható”, ha végrehajtási ideje *polinomiális*, vagyis n^k -nal arányos (adott k -ra és n bemeneti adatra). Ha egy feladat minden lehetséges megoldást kér, és csak exponenciális algoritmussal tudjuk megoldani, a *backtracking* (visszalépéses keresés) módszert alkalmazzuk, amely exponenciális ugyan, de megpróbálja csökkenteni a generálandó próbálkozások számát.

1.4.3. A visszalépéses keresés általános bemutatása

A visszalépéses keresés azon feladatok megoldásakor alkalmazható, amelyeknek eredményét az $M_1 \times M_2 \times \dots \times M_n$ Descartes-szorzatnak azon elemei alkotják, amelyek eleget tesznek bizonyos belső feltételeknek. Az $M_1 \times M_2 \times \dots \times M_n$ Descartes-szorzat a megoldások tere (az eredmény egy x sorozat, amelynek x_i eleme az M_i halmazból való).

A visszalépéses keresés nem generálja a Descartes-szorzat minden $x = (x_1, x_2, \dots, x_n) \in M_1 \times M_2 \times \dots \times M_n$ elemét, hanem csak azokat, amelyeknek esetében remélhető, hogy megfelelnek a belső feltételeknek. Így, megpróbálja csökkenteni a próbálkozásokat.

Az algoritmusban az x tömb elemei egymás után, egyenként kapnak értékeket: x_i számára csak akkor „javasolunk értéket”, ha x_1, x_2, \dots, x_{i-1} már kaptak végleges értéket az aktuálisan generált eredményben. Az x_i -re vonatkozó javaslatot akkor fogadjuk el, amikor x_1, x_2, \dots, x_{i-1} értékei az x_i értékével együtt megvalósítják a *belső feltételeket*. Ha az i -edik lépésben a belső feltételek nem teljesülnek, x_i számára új értéket választunk az M_i halmazból. Ha az M_i halmaz minden elemét kipróbáltuk, visszalépünk az $i-1$ -edik elemhez, amely számára új értéket „javasolunk” az M_{i-1} halmazból. Ha az i -edik lépésben a belső feltételek teljesülnek, az algoritmus folytatódik. Ha szükséges folytatni, mivel a számukat ismerjük és még nem generáltuk mindegyiket, vagy valamilyen másképp kifejezett tulajdonság alapján eldöntöttük, hogy még nem jutottunk eredményhez, a *folytatási feltételek* alapján folytatjuk az algoritmust.

Azokat a lehetséges eredményeket, amelyek a megoldások teréből vették értékeiket úgy, hogy teljesítik a belső feltételeket, és amelyeknek esetében a folytatási feltételek nem kérnek további elemeket, *végeredményeknek* nevezzük.

A belső feltételek és a folytatási feltételek között szoros kapcsolat áll fenn. Ezek kifejezés-módjának szerencsés megválasztása többnyire a számítások csökkentéséhez vezethet. A belső feltételeket egy külön algoritmusban vizsgáljuk: *Megfelel(i)*, ahol i az aktuálisan generált elem

indexe. Ez az alprogram *igaz* értéket térít vissza, ha az x_i elem az eddig generált x_1, x_2, \dots, x_{i-1} elemekkel együtt megfelel a belső feltételeknek, és hamis értéket ellenkező esetben.

Algoritmus *Megfelel(i)*: { *a függvény Megfelel értékét téríti* }
 Megfelel \leftarrow igaz
Ha *a belső feltételek x_1, x_2, \dots, x_i esetében nem teljesülnek akkor*
 Megfelel \leftarrow hamis
vége(ha)

Vége(algoritmus)

A feladat által kért eredményt a következő algoritmussal generáljuk:

Algoritmus *RekurzívBacktracking(i)*:

Minden $m_j \in M_i$ *értékre végezd el*:

$x_i \leftarrow m_j$

Ha *Megfelel(i) akkor* { *megvalósulnak a belső feltételek x_1, x_2, \dots, x_i esetében* }

Ha $i < n$ **akkor**

RekurzívBacktracking($i+1$)

különben

Ki: x_1, x_2, \dots, x_n

vége(ha)

vége(ha)

vége(minden)

Vége(algoritmus)

Az algoritmust az $i = 1$ értékre hívjuk meg először.

A módszer eredményessége nagymértékben függ a folytatási feltételek szerencsés kiválasztásától. Minél hamarabb állítjuk le egy eredmény generálását, annál kisebb a rekurzió mélysége, de a feltételek nem lehetnek túl bonyolultak, mivel ezeket minden aktiválódásnál végrehajtja az algoritmus.

A módszer azoknak a feladatoknak a megoldásakor alkalmazható, amelyekben a követelményeknek megfelelően minden eredményt meg kell állapítanunk. Ha az $M_1 \times \dots \times M_n$ Descartes-szorzat számossága nem túl nagy, valamint a feltételek biztosítanak egy nem túl mély rekurziót, eredményesen alkalmazható.

Összefoglalva, a következő lépéseket kell elvégeznünk:

1. *az eredmény kódolása* – meg kell állapítanunk az x_i elemek *jelentését* az illető feladat esetében, valamint meg kell határoznunk az $M_i, i = 1, 2, \dots, n$ halmazokat.
2. *a belső, majd a folytatási feltételek megállapítása.*
3. *a RekurzívBacktracking(i) vagy iteratív változatának átírása.*

1.4.4. Megoldott feladatok

1. Nyolc királynő a sakktáblán

Írjuk ki az összes lehetséges módját annak, ahogyan 8 királynő elhelyezhető egy sakktáblán, úgy, hogy ne támadják egymást. Két királynő támadja egymást, ha ugyanazon a soron, oszlopon, illetve átlón helyezkedik el.

Megoldás

Minden királynőt egymás után elhelyezünk a neki megfelelő sorba az első oszloppal kezdődően, amíg meg nem találjuk azt az oszlopot, amelyben nem támad más, eddig feltett királynőt. Ha egy királynőt nem lehet elhelyezni, visszatérünk az előzőhöz és számára tovább keresünk megfelelő, nagyobb sorszámú oszlopot.

Az eredményt egy egydimenziós tömbbel ($K_i, i = 1, 2, \dots, 8$) kódoljuk. A tömb K_i elemeinek értéke az oszlop sorszáma, ahova az i -edik királynőt tettük (az i -edik sorban). A sakkjátszóknak 8 oszlopa van, tehát $K_i \in \{1, 2, \dots, 8\}, i = 1, \dots, 8$. Az eddigiekből következik, hogy egy eredmény az $\{1, 2, \dots, 8\}^8$ Descartes-szorzat eleme, azzal a feltétellel, hogy a 8 királynő, amelyek a K_1, K_2, \dots, K_8 oszlopokban találhatóak, ne támadja egymást. A kódolás sajátos módja biztosítja, hogy soronkénti támadási lehetőség nincs, hiszen minden királynő új sorba kerül.

A második észrevétel a feladat rekurzív megfogalmazását teszi lehetővé: elhelyezzük az első királynőt, rendre az első sor első, második, ..., 8-dik oszlopába, majd megoldjuk a feladatot a fennmaradt 7 királynő esetében, de úgy, hogy mindig ellenőrizzük, hogy egy új királynő ne támadjon egyet sem a már elhelyezettek közül.

Általánosan megfogalmazva: az i -edik királynő esetében meg kell határoznunk minden helyet, ahova ezt el lehet helyezni az i -edik sorban úgy, hogy ne támadjon egyet sem azok közül, amelyek az első, második, ..., $i-1$ -edik sorban már el vannak helyezve. Tehát elhelyezzük az i -edik királynőt, majd megoldjuk ugyanezt a feladatot az $i+1$ -edik királynő esetében.

Ha minden királynőt elhelyeztünk, van egy eredmény, amit ki kell írunk. Az elhelyezést a $Királynő(i)$ rekurzív alprogram végzi el, a támadási lehetőséget a $NemTámad(i)$ logikai függvény ellenőrzi.

Ahhoz, hogy két királynő ne támadja egymást, a következő relációknak kell teljesülniük: $K_i \neq K_j, i - j \neq |K_i - K_j|, j = 1, 2, \dots, i - 1$. Bemutatjuk a $Megfelel(i)$ algoritmus tömörebb változatát:

Algoritmus NemTámad(i):

Minden $j = 1, i-1$ végezd el:

Ha $(K_i = K_j)$ vagy $(i-j = |K_i - K_j|)$ akkor

térítsd hamis

{ az i . és j . királynők támadják egymást }

vége(ha)

vége(minden)

térítsd igaz

Vége(algoritmus)

Algoritmus Királynő(i):

Minden $j = 1, 8$ végezd el:

$K_i \leftarrow j$

{ az i -edik királynőt a j -edik oszlopba tesszük }

Ha NemTámad(i) akkor

{ az i -edik királynő nem támadja egyiket sem }

Ha $i < 8$ akkor

Királynő($i+1$)

különben

Kiír

vége(ha)

vége(ha)

vége(minden)

Vége(algoritmus)

Az első hívás alakja: $Királynő(1)$.

2. Variációk

Az óvónéni a karácsonyi ünnepélyre készül. A díszterem színpadán n széket lehet egy sorban elhelyezni, de a csoportban m óvódás van ($n < m$). Írjuk ki minden lehetséges módját annak, ahogy az óvódások leülhetnek az n székre.

Megoldás

Eltérünk az előbbi mintától, hiszen fölösleges „javasolni”, hogy üljön le egy már leültetett gyermek.

Az eredmény kódolása: Az x_i az i -edik székre ülő gyerek nevének az indexe. Tehát $x_i \in \{1, 2, \dots, m\}$, ahol m a gyermekek száma, ($i = 1, 2, \dots, n$).

Belső feltételek: $x_i \neq x_j, i \neq j, i, j = 1, 2, \dots, n$. A belső feltételek azt fejezik ki, hogy az i . székre csak olyan gyerek ülhet le, aki pillanatnyilag még áll. Az ellenőrzés egyszerűbb lesz, fölhasználunk egy *mégÁll* logikai tömböt, ahol *mégÁll_j* igaz, ha a j -edik gyerek még nem ült le, és hamis ellenkező esetben. Az $x_i \neq x_j, j = 1, 2, \dots, i - 1$ feltételek a következőképpen alakulnak át: *mégÁll_{x_i}* = igaz.

Folytatási feltétel: $i < n$ (még van szabad szék)

A *mégÁll* tömb elemeinek kezdőértéke *igaz*, mivel még senki nem ült le, majd az ültetési folyamat során a megfelelő elemek *hamis* értéket kapnak. Valahányszor egy ültetési rend megváltozik, a j -edik gyermek feláll az i -edik székről és oda más gyermek ülhet majd le. Ugyanakkor, a j -edik gyermek egy másik ültetési rendben újból leülhet. A j -edik gyermek felállítása maga után vonja a megfelelő *mégÁll_j* visszaállítását *igaz*-ra. Ez a megoldás hatékonyabb, mint az, amelyet a mintaalgorithmus alapján készíthetnénk, mivel kevesebb összehasonlítást végez.

Algorithmus Variáció(i):

Minden $j = 1, m$ végezd el:

Ha mégÁll_j akkor { *a j-edik gyermek még áll* }

$x_i \leftarrow j$

mégÁll_j \leftarrow hamis { *a j-edik gyermeket leültettük az i-edik székre* }

Ha $i < n$ akkor

Variáció($i+1$)

különben

Kiír

vége(ha)

mégÁll_j \leftarrow igaz { *a j-edik gyermeket felállítjuk, hogy később ülhessen más székre* }

vége(ha)

vége(minden)

Vége(algorithmus)

Az első hívás alakja: *Variáció(1)*.

3. Zárójelek

Generáljunk és írjunk ki minden helyesen nyitó és csukó n zárójelet tartalmazó karakterláncot! *Példa:* ha $n = 4$, a helyes karakterláncok:

(())

()()

Megoldás

Az eredmény kódolása: Ha n páros szám, az eredmények az M^n halmaz elemei, ahol $M = \{ '(', ') '\}$ és $x_i \in M, i = 1, \dots, n$. Ha n páratlan, akkor nincs megoldás.

Belső feltételek: Adott pillanatban ne létezzen több csukó zárójel, mint nyitó, és nyitó nem lehet több mint $n/2$. Mivel a megoldások tere kételemű halmaz, és a két elem esetében a belső feltétel különbözik, lemondunk a *Minden* struktúráról és két *Ha* utasítással ellenőrizzük ezeket.

Jelöljük ny -nyel és cs -vel a nyitó, illetve a csukó zárójelek számát. A folytatási feltételek különböznek az x_i elemek értékének függvényében:

$$\begin{cases} ny < \frac{n}{2}, & \text{ha } x_i = '(' \\ z < ny, & \text{ha } x_i = ')' \end{cases} \quad \forall i = \overline{2, n-1}$$

Folytatási feltételek: Mivel bármely eredményben $x_1 = '('$ és $x_n = ')'$, a hívó programegységben elvégezzük az inicializálásokat: $x_1 \leftarrow '('$ és $x_n \leftarrow ')'$. Tehát, az algoritmus a második helytől kezdődően az $(n-1)$ -dik helyig tesz zárójeleket. Amikor az n -edik karakter következne, leállunk. Az első hívás alakja: *Zárójel*(2, 1, 0).

Algoritmus Zárójel(i, ny, cs):

```

Ha  $i = n$  akkor                                     { kilépési feltétel }
    Ki:  $x$                                              {  $x$  egy karakterlánc }
különben
    Ha  $ny < n \text{ div } 2$  akkor
         $x_i \leftarrow '('$ 
        Zárójel( $i+1, ny+1, cs$ )
    vége(ha)
    Ha  $cs < ny$  akkor
         $x_i \leftarrow ')'$ 
        Zárójel( $i+1, ny, cs+1$ )
    vége(ha)
vége(ha)

```

Vége(algoritmus)

4. Labirintus

Egy labirintust egy n soros és m oszlopos L kétdimenziós tömbben tárolunk, amelyben a folyosónak megfelelő elemek értéke 1; ezek az értékek egymás után következnek a labirintust ábrázoló tömbben, egy bizonyos sorban, vagy oszlopban. Egy személyt leengednek ejtőernyővel a labirintusba az (i, j) helyre. Írjunk ki minden olyan utat, amely kivezet a labirintusból! Egy út nem érintheti kétszer ugyanazt a helyet. A labirintusból a tömb szélén léphetünk ki.

Elemzés

Az eredmény kódolása: A feladat minden kivezető utat kér. Legyen egy ilyen út hossza k . Az utat az x_1, x_2, \dots, x_k és y_1, y_2, \dots, y_k sorozatokkal kódolunk, amelyek azokat a sorokat és oszlopokat tartalmazzák, amelyeknek érintésével kifele haladunk a labirintusból. $x_i \in \{1, 2, \dots, n\}$, $y_i \in \{1, 2, \dots, m\}$, $i = 1, 2, \dots, k$.

Belső feltételek: Az útvonalra a következő belső feltételek érvényesek:

a) Folyosón kell haladnia: $L_{x_i, y_i} = 1, i = 1, 2, \dots, k$.

b) Nem léphet kétszer ugyanarra a helyre: $(x_i, y_i) \neq (x_j, y_j), i, j = 1, 2, \dots, k, i \neq j$.

c) Biztosítania kell a labirintusból való kijutást: $x_k \in \{1, n\}$ vagy $y_k \in \{1, m\}$

Folytatási feltételek: Tartalmazzák az a) és b) ellenőrzését minden lépésnél. A b) feltétel az i -edik lépésben: $(x_i, y_i) \neq (x_j, y_j), j = 1, \dots, i-1$.

Az eredményt az $eredm_{ij}$ ($i = 1, 2, \dots, n, j = 1, 2, \dots, m$) tömb segítségével tároljuk, amelyben

$$eredmény_{ij} = \begin{cases} \text{az a lépésszám, amellyel az } (i, j) \text{ helyre léptünk,} & \text{ha lehet lépni az } (i, j) \text{ helyre} \\ 0, & \text{ha nem lehet lépni az } (i, j) \text{ helyre} \end{cases}$$

Egy bizonyos helyről négy irányba léphetünk. Az alábbi kód tartalmaz egy figyelemreméltó egyszerűsítést, ami a folytatási feltételeket illeti. Nem szükséges ellenőriznünk azt, hogy kiléptünk-e a labirintusból, mivel a hívás előtt (a labirintus beolvasása után) az L tömböt körülvevünk egy 0-ból álló kerettel. Így az algoritmus gyorsabbá válik. Az algoritmust a kiindulási hely koordinátáira (i, j) és 1 lépésszámmal hívjuk meg.

Algoritmus Út(i, j , lépés):

Ha ($L_{ij} = 1$) **és** ($eredm_{ij} = 0$) **akkor**

{ próbálunk az (i, j) helyre lépni; ha (i, j) folyosó és még nem jártunk itt }

$eredm_{ij} \leftarrow$ lépés { az (i, j) helyre lépünk }

Ha ($i \in \{1, n\}$) **vagy** ($j \in \{1, m\}$) **akkor**

Kiír { kijáráshoz értünk, kiírjuk az eredménytömböt }

vége(ha)

Út($i-1, j$, lépés+1) { próbálunk más utat is: felfele lépünk }

Út($i, j+1$, lépés+1) { jobbra lépünk }

Út($i+1, j$, lépés+1) { lefele lépünk }

Út($i, j-1$, lépés+1) { balra lépünk }

$eredm_{ij} \leftarrow 0$ { töröljük az utolsó lépést, hogy egy új útvonalon léphessünk újra ide }

vége(ha)

Vége(algoritmus)

Második megoldás

Algoritmus Út(i, j , lépés):

Minden irány = 1, 4 **végezd el:** { kiválasztunk egy irányt }

$úji \leftarrow i + x_{irány}$ { $(úji, újj)$ az új koordináták }

$újj \leftarrow j + y_{irány}$

Ha ($úji \in \{1, 2, \dots, n\}$) **és** ($újj \in \{1, 2, \dots, m\}$) **akkor**

Ha ($L_{úji, újj} = 1$) **és** ($eredm_{úji, újj} = 0$) **akkor**

$eredm_{úji, újj} \leftarrow$ lépés { az $(úji, újj)$ helyre lépünk }

Ha ($úji \in \{1, n\}$) **vagy** ($újj \in \{1, m\}$) **akkor**

Kiír { kiléptünk a labirintus szélén }

vége(ha)

Út($úji, újj$, lépés+1)

$eredm_{úji, újj} \leftarrow 0$ { lemondunk az utolsó lépésről }

vége(ha)

vége(ha)

vége(minden)

Vége(algoritmus)

Az $\hat{U}t(i, j, lépés)$ alprogramban a *lépés* pillanatban megpróbálunk az (i, j) helyről az $(úji, újj)$ helyre lépni. Ezeket két konstans tömb (x, y) segítségével állapítjuk meg úgy, hogy ezek a négy szomszédos hely koordinátáit adják meg: $x = (-1, 0, 1, 0)$, $y = (0, 1, 0, -1)$.

Azt várnánk, hogy az algoritmus a *Ha* utasítás különben ágán hívja meg önmagát. Ha így járnánk el, elvesztenénk azokat az eredményeket, amelyeknek esetében a labirintus szélén tovább lehet menni, és a kilépés egy másik pontban is lehetséges.

Ebben a második megoldásban nem vettük körül a labirintust az első algoritmusban említett kerettel. Ennek következtében szükséges volt ellenőrizni, hogy az új hely, ahova lépni akarunk a labirintuson belül van-e.

Ezt az algoritmust az $\hat{U}t(i, j, 2)$ alakban hívjuk meg, de a hívás előtt $eredm_{ij} \leftarrow 1$, ahol (i, j) a kiindulási hely.

Általánosítva az előbbi feladatban használt rekurzív algoritmust, amely a visszalépéses keresés módosított változata, észrevevesszük, hogy mivel az előrehaladás egy kétdimenziós tömbben történik, az alprogram két paramétere (i, j) annak a helynek a koordinátái, ahova utoljára léptünk.

1.5. Az oszd meg és uralkodj módszer (divide et impera)

Az *oszd meg és uralkodj* módszer (*divide et impera*) alkalmazása akkor ajánlott, amikor a feladatot fel lehet bontani egymástól független részfeladatokra, amelyeket az eredeti feladathoz hasonlóan oldunk meg, de kisebb méretű adathalmaz esetében.

Az eredeti feladatot felbontjuk egymástól független részfeladatokra, amelyek az eredetihez hasonlóak, de kisebb adathalmazra definiáltak. A részfeladatokkal hasonlóan járunk el és a felbontást akkor állítjuk le, amikor a feladat megoldása a lehető legjobban leegyszerűsödött. A maximálisan leegyszerűsített feladatot megoldjuk, majd a részfeladatok eredményeiből fokozatosan felépítjük a következő méretű feladat eredményeit, ezek összerakása által. Az utolsó összerakás az eredeti feladat végeredményét adja meg.

Mivel a részfeladatok csak méreteikben különböznek az eredeti feladattól, a *divide et impera* módszert a legkézenfekvőbbben rekurzívan írjuk le. A felbontás megtörténik a rekurzióba való belépéskor, a részeredmények összerakása pedig a kilépéskor.

1.5.1. Az oszd meg és uralkodj módszer általános bemutatása

A $DivImp(bal, jobb, eredm)$ algoritmus az a_1, a_2, \dots, a_n sorozatot dolgozza fel, tehát $DivImp(1, n, eredm)$ alakban hívjuk meg először. Formális paraméterei a *bal* és a *jobb* (az aktuális részsorozat bal és jobb indexe), valamint *eredm*, amelyben a végeredményt továbbítjuk.

Algoritmus $DivImp(bal, jobb, eredm)$:

```

Ha jobb - bal < ε akkor           { ha a feladat maximálisan leegyszerűsödött }
  Megold(bal, jobb, eredm)       { kiszámítjuk az egyszerű feladat eredm eredményét }
különben
  Feloszt(bal, jobb, közép) { kiszámítjuk a közép indexet, ahol felosztjuk a sorozatot }
  DivImp(bal, közép, eredm1)   { megoldjuk a feladatot a bal részsorozat esetében }
  DivImp(közép+1, jobb, eredm2) { megoldjuk a feladatot a jobb részsorozat esetében }
  Összerak(eredm1, eredm2, eredm) { összerakjuk a részeredményeket }
vége(ha)

```

Vége(algoritmus)

Az oszd meg és uralkodj stratégiát – természetesen – lehet iteratívan is implementálni. Az iteratív algoritmusok mindig gyorsabbak lesznek. A rekurzív változat előnye viszont az átláthatóságában és az egyszerűségében rejlik.

1.5.2. Megoldott feladatok

1. Szorzat

Számítsuk ki n valós szám szorzatát *oszd meg és uralkodj* módszerrel! Egy adott pillanatban csak egy szorzást végezzünk!

Megoldás

Mivel egy adott pillanatban, egy adott művelettel, csak két szám szorzatát tudjuk kiszámítani, a szorzatot részsorzatokra bontjuk: a szorzótényezőket két csoportra osztjuk, kiszámítjuk

egy-egy csoport szorzatát, majd a két csoport kiszámított szorzatát összeszorozzuk. Ezt a felbontást addig lehet újra, meg újra elvégezni, amíg egy csoport legtöbb két szorzótényezőből nem áll.

A $Szorzat(x_1, \dots, x_n)$ részfeladat általános alakja: $Szorzat(x_{bal}, \dots, x_{jobb})$. Minden részfeladat más-más szorzatot számol ki, tehát a feladatok függetlenek egymástól.

Algoritmus Szorzat(bal, jobb):

Ha jobb = bal **akkor** { *Bemeneti adatok: bal, jobb. A függvény a szorzatot téríti* }
 térítsd x_{bal} { *a részsorozat egy elemből áll* }

különben

Ha jobb - bal = 1 **akkor**

 térítsd $x_{bal} * x_{jobb}$ { *a részsorozat két elemű* }

különben

{ *felbontjuk a Szorzat(bal, ..., jobb) feladatot* }

 közepe $\leftarrow (bal+jobb) \text{ div } 2$

 p1 \leftarrow Szorzat(bal, közepe)

 p2 \leftarrow Szorzat(közepe+1, jobb)

 térítsd p1 * p2

{ *összerakjuk a részeredményeket* }

 vége(ha)

 vége(ha)

Vége(algoritmus)

2. Bináris keresés

Adott egy n egész számból álló, növekvően rendezett sorozat. Állapítsuk meg egy adott szám helyét a sorozatban! Ha az illető szám nem található meg a sorozatban, a sorszámnak megfelelő paraméter értéke legyen 0.

Megoldás

Mivel egy bizonyos elemet keresünk, amelynek a helye ismeretlen, az $x_1 < x_2 < \dots < x_n$ sorozat közepén fogjuk először keresni. A következő esetek fordulhatnak elő:

1. $keresett = x_{közép} \Rightarrow$ keresett a sorban a *közép* helyen található;

2. $keresett < x_{közép} \Rightarrow$ mivel a sorozat rendezett, a keresett számot a sorozat első ($x_1, \dots, x_{közép-1}$) felében keressük tovább;

3. $keresett > x_{közép} \Rightarrow$ a keresett számot a sorozat második ($x_{közép+1}, \dots, x_n$) felében keressük tovább.

Következésképpen, ahelyett, hogy a keresett elem megkeresése két részfeladatra bomlana, átalakul egyetlen feladattá: keressük az elemet vagy az $x_{bal}, \dots, x_{közép-1}$ sorozatban, vagy az $x_{közép+1}, \dots, x_{jobb}$ sorozatban. Itt nincs szükség a *divide et impera* harmadik lépésére (a részeredmények összerakására).

Algoritmus BinKeres(x, bal, jobb, keresett, közép):

{ *Bemeneti adatok: x, bal, jobb, keresett. Kimeneti adat: közép* }

Ha bal > jobb **akkor**

 közép $\leftarrow 0$

{ *keresett nincs a sorozatban* }

különben

 közép $\leftarrow (bal+jobb) \text{ div } 2$

Ha keresett < $x_{közép}$ **akkor**

 BinKeres(x, bal, közép-1, keresett, közép)

```

különben
  Ha keresett >  $x_{közép}$  akkor
    BinKeres(x, közép+1, jobb, keresett, közép)
  vége(ha) { ha keresett =  $x_{közép}$  megvan a pozíció }
vége(ha)
vége(ha)
Vége(algoritmus)

```

E feladat esetében is létezik egy iteratív megoldás, amely a végrehajtás idejét tekintve hatékonyabb:

```

Algoritmus BinKeresIteratív(n, x, keresett, közép):
  bal ← 1
  jobb ← n
  megvan ← hamis
  Amíg nem megvan és (bal ≤ jobb) végezd el:
    közép ← (bal+jobb) div 2
    Ha  $x_{közép}$  = keresett akkor
      megvan ← igaz { közép tartalmazza a keresett helyét }
    különben
      Ha  $x_{közép}$  > keresett akkor
        jobb ← közép - 1
      különben
        bal ← közép + 1
    vége(ha)
  vége(amíg)
  Ha nem megvan akkor
    közép ← 0 { ha közép értéke 0 ⇒ keresett nem található }
  vége(ha)
Vége(algoritmus)

```

3. Összefésülésen alapuló rendezés (*MergeSort*)

Rendezzünk növekvő sorrendbe egy egész számokból álló sorozatot összefésüléssel!

Megoldás

Ha két rendezett sorozatból úgy állítunk elő egy harmadikat, hogy ez utóbbi úgyszintén rendezett, összefésülésről beszélünk. De itt nem két rendezett sorozatból kell egy harmadik, ugyan csak rendezettet előállítanunk, hanem egyetlen sorozatot kell rendeznünk. Az adott sorozatot két részre osztjuk, abból a célból, hogy rendezhessük. De ezeket újból felosztjuk, amíg a kapott tömb, amelyet rendeznünk kell, csak egy elemből áll. Az egyelemű tömbök, természetesen rendezettek és megkezdődhet a tulajdonképpeni összefésülés.

```

Algoritmus Összefésül(bal, közép, jobb):
  Minden i = bal, közép végezd el:
     $a_i$  ←  $x_i$ 
  vége(minden)

```

```

Minden  $i = \text{közép} + 1$ , jobb végezd el:
   $b_i \leftarrow x_i$ 
vége(minden)
 $a_{\text{közép}+1} \leftarrow \text{végtelen}$ 
 $b_{\text{jobb}+1} \leftarrow \text{végtelen}$  { strázsák }
 $i \leftarrow \text{bal}$ 
 $j \leftarrow \text{közép} + 1$ 
Minden  $k = \text{bal}$ , jobb végezd el:
  Ha  $a_i < b_j$  akkor
     $x_k \leftarrow a_i$ 
     $i \leftarrow i + 1$ 
  különben
     $x_k \leftarrow b_j$ 
     $j \leftarrow j + 1$ 
  vége(ha)
vége(minden)
Vége(algoritmus)

```

```

Algoritmus Rendez( $\text{bal}$ ,  $\text{jobb}$ ):
  Ha  $\text{bal} < \text{jobb}$  akkor
     $\text{közép} \leftarrow (\text{bal} + \text{jobb}) \text{ div } 2$ 
    Rendez( $\text{bal}$ ,  $\text{közép}$ )
    Rendez( $\text{közép} + 1$ ,  $\text{jobb}$ )
    Összefésül( $\text{bal}$ ,  $\text{közép}$ ,  $\text{jobb}$ )
  vége(ha)
Vége(algoritmus)

```

Az *Összefésül*(bal , közép , jobb) algoritmus eredménye az $x_{\text{bal}}, \dots, x_{\text{jobb}}$ rendezett sorozat, amelybe tulajdonképpen ugyanazon sorozat két részsorozatát, az $x_{\text{bal}}, \dots, x_{\text{közép}}$ és az $x_{\text{közép}+1}, \dots, x_{\text{jobb}}$ részsorozatokat fésültük össze. Ezzel magyarázható annak a szükségessége, hogy az összefésülendő sorozatokat átmásoltuk az a illetve a b sorozatba. A hívó programegységben a *Rendez*($1, n$) algoritmust hívjuk.

4. Gyorsrendezés (*QuickSort*)

Fölhasználva a *quicksort* algoritmust, rendezzünk növekvő sorrendbe n egész számot!

Megoldás

A gyorsrendezés az oszd meg és uralkodj módszeren alapszik, mivel az eredeti sorozatot úgy rendezi, hogy két rendezendő részsorozatra bontja. A részsorozatok rendezése egymástól függetlenül történik. A részeredmények összerakása hiányzik (hasonlóan a bináris kereséshez). Amikor az x_1, \dots, x_n sorozatot készülünk rendezni, előbb előkészítünk két részsorozatot (x_1, \dots, x_{m-1} és x_{m+1}, \dots, x_n) úgy, hogy az x_1, \dots, x_{m-1} részsorozat elemei kisebbek legyenek, mint az x_{m+1}, \dots, x_n részsorozat elemei. Közöttük található az x_m , amely nagyobb, mint az x_1, \dots, x_{m-1} részsorozat bármely eleme, és kisebb, mint az x_{m+1}, \dots, x_n részsorozat összes eleme.

Azt az elemet, amely meghatározza a helyet, ahol az adott tömb két részre oszlik, *strázsának* (*örszem*) nevezzük. Ennek a helynek a meghatározása kulcskérdés az algoritmus végrehajtása során. A strázsá m helyét úgy határozzuk meg, hogy az x_1, \dots, x_m tömbben legyenek azok az

elemek, amelyek kisebbek, mint a strázsza és az x_{m+1}, \dots, x_n tömbben azok, amelyek nagyobbak annál.

Gyakran választjuk strázsának az x_1 -et. Elindulunk a tömb két szélső elemétől és felcseréljük egymás közt azokat az elemeket, amelyek nagyobbak, mint a strázsza (és a tömb első részében található) azokkal, amelyek kisebbek, mint a strázsza (és a tömb második részében található). Ahol ez a bejárás véget ér, ott fogjuk két részre osztani a tömböt. Egy ilyen feldolgozás során egy elem a végleges helyére kerül.

A részsorozatok rendezése érdekében ezeket hasonló módon bontjuk fel. A felbontás addig folytatódik, amíg a rendezendő részsorozat hossza 1 lesz.

Algoritmus QuickSort(bal, jobb):

```

Ha bal < jobb akkor           { meghatározzuk azt az  $m$  helyet, ahol a sorozatot }
                                { két részre bontjuk, miközben az  $x_m$  elem a végleges helyére kerül }
    m ← Strázsza_helye(bal, jobb)
    QuickSort(bal, m)           { hasonlóan járunk el az  $(x_{bal}, \dots, x_m)$  részsorozattal }
    QuickSort(m+1, jobb)       { valamint az  $(x_{m+1}, \dots, x_{jobb})$  részsorozattal }
vége(ha)
Vége(algoritmus)

```

Látható, hogy a rekurzív hívásoknak megfelelően, az algoritmus meghívja önmagát egy bal meg egy jobb részsorozat rendezése érdekében. Dacára annak, hogy az algoritmus nem tartalmaz összehasonlításokat és felcseréléseket, a sorozat aközben rendeződik, miközben keressük a strázsza m helyét:

Algoritmus StrázszaHelye(bal, jobb):

```

strázsza ←  $x_{bal}$            { Bemeneti adatok: bal, jobb. Kimeneti adat: a strázsza helye }
i ← bal-1
j ← jobb+1           { megkeressük azt a  $j$  indexet, amelyre  $bal \leq j < jobb$  }
Ismételd
    Ismételd           { megkeressük azt a  $j$  indexet (jobbról balra), amelyre  $x_j > strázsza$  }
        j ← j - 1
    ameddig  $x_j \leq strázsza$ 
        Ismételd       { megkeressük azt az  $i$  indexet (balról jobbra), amelyre  $x_i < strázsza$  }
            i ← i + 1
        ameddig  $x_i \geq strázsza$ 
        Ha i < j akkor
             $x_i \leftrightarrow x_j$            { felcseréljük ezt a két nem megfelelő tulajdonságú elemet }
        vége(ha)
    ameddig i ≥ j       { addig folytatjuk a keresést és felcserélést, amíg i kisebb, mint j }
    térítsd j           { megtaláltuk az új strázsza helyét }
Vége(algoritmus)

```

Megjegyzés

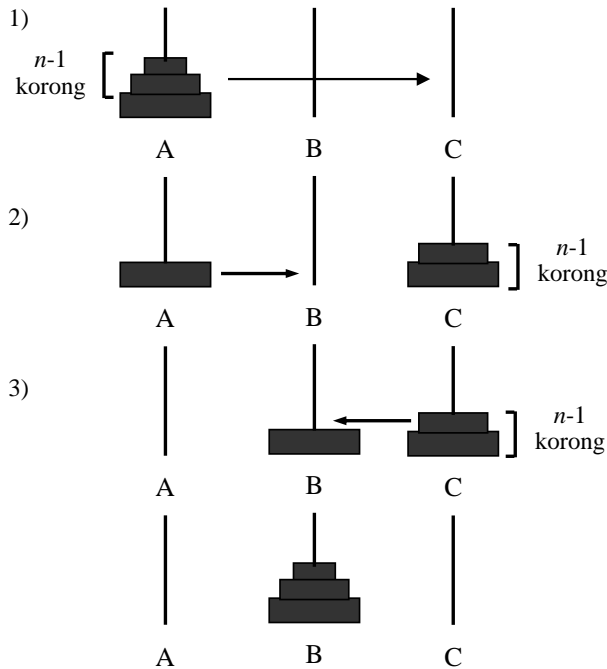
Ez az algoritmus főleg akkor gyors, ha a sorozat nem rendezett, egyébként előfordulhat, hogy az algoritmus négyzetes bonyolultságúvá válik. Ha fennáll ez a veszély, tanácsos a rendezés előtt véletlenszerűen átrendezni a bemeneti sorozatot, ezzel biztosítva a különböző permutációk azonos valószínűségét. Ez a módosítás nem javít a legrosszabb futási időn, de biztosítja, hogy a futási idő független lesz a bemeneti elemek sorrendjétől.

5. Hanoi tornyok

Adva van három rúd A, B, C. Az elsőre fel van fűzve n darab, különböző átmérőjű korong úgy, hogy a korongok az átmérőjük csökkenő sorrendjében helyezkednek el egymás fölött. A másik két rúd üres. Írjuk ki minden lehetséges módját annak, ahogyan a korongokat átköltöztethetjük az A rúdról a B-re, ugyanolyan sorrendben, ahogyan az A-n helyezkedtek el. Közben fel lehet használni, ideiglenesen a C rudat. Egy mozgatás csak egy korongot érinthet, és csak kisebb átmérőjű korongot helyezhetünk egy nagyobb átmérőjű korong fölé.

Megoldás

A módszer újból a *divide et impera*. Az n korong átköltöztetése az A rúdról a B-re felbontható három, ehhez hasonló feladatra:



A három részfeladat méretét a költöztetendő korongok száma határozza meg: $n - 1$, 1 és $n - 1$. A részfeladatok függetlenek, mivel az eredeti rudak konfigurációi, valamint az időközben váltakozva ideiglenesen használt rudaké különbözők. A feladat felbontása ugyanígy folytatódik, míg olyan részfeladathoz nem érünk, amelynek mérete 1 . Ennek megoldása egyetlen korong költöztetését jelenti.

A részeredmények összerakása ebben az esetben is hiányzik.

Algoritmus Hanoi(n , A, B, C):

Ha $n = 1$ **akkor**

Költöztess egy korongot A-ról B-re

különben

Hanoi($n-1$, A, C, B)

Hanoi(1 , A, B, C)

Hanoi($n-1$, C, B, A)

vége(ha)

Vége(algoritmus)

Az algoritmust a *Hanoi*(n , A, B, C) utasítással hívjuk, ahol az aktuális paraméterek értékei 'A', 'B', 'C'. A *Költöztess egy korongot A-ról B-re* lehet pl. egy kírás: **Ki**: A, '-', B.

2. fejezet

Objektumorientált programozás

2.1. Objektumorientált fogalmak

2.1.1. Adatvédelem moduláris programozással

Az eljárásközpontú programozás keretében a kódot igyekszünk eljárásokra és függvényekre bontani. A C és a C++ programozási nyelvekben az eljárásokat és függvényeket egyetlen névvel jellemezzük. Mindkét esetben *függvényekről* beszélünk, de megkülönböztetünk olyan függvényeket, amelyek visszatérítenek egy értéket és olyanokat, amelyek nem. Az eljárásoknak azok a függvények felelnek meg, amelyek nem térítenek vissza semmit. Ebben az esetben a *void* kulcsszóval jelezzük a visszaadandó érték típusának a hiányát.

A nagyobb alkalmazások írásakor felmerül annak a szükségessége, hogy az általunk használt adatok védelmét megvalósítsuk. Ez azt jelentené, hogy csak a függvényeknek egy részével lehessen hozzáférni az adatokhoz. Azért van erre szükség, mert ez által jelentősen csökken a hibalehetőségek száma. Az adatok és a rájuk vonatkozó függvények egyetlen egységet fognak képezni. Így az adatok módosítása csak ezekkel a függvényekkel lesz megvalósítható, másokkal nem.

Az adatok védelmére már a C programozási nyelv is lehetőséget teremtett a *moduláris programozás* által. Ha egy állomány globális hatókörében, tehát a függvényeken, osztályokon és névtereken kívül, egy statikus változót vezetünk be, akkor ezt a változót a deklaráció helyétől az illető állomány (modul) végéig bármely függvényben használhatjuk. Ezzel ellentétben viszont más állományban még akkor sem tudunk hivatkozni az illető változóra, ha abban egy *extern* típusú deklarációt helyezünk el.

A továbbiakban egy olyan példát ismertetünk, amely az adatok védelmét a moduláris programozás segítségével teszi lehetővé. Egy egész elemekből álló vektorokra vonatkozó modult hozunk létre. A vektor elemeit egy *int* típusra hivatkozó mutató segítségével tároljuk. Meg kell adnunk a vektor méretét is, tehát az elemek számát. Ezt a két adatot a függvényeken kívül deklarált statikus változókkal vezetjük be. Az adatok

feldolgozását a következő négy függvénnyel végezzük: *epit*, *felszabadit*, *negyzetre* és *kiir*. Az első függvény egy egész elemekből álló tömb és egy egész szám (a méret) segítségével létrehozza a vektort. Ha a vektorra már nincs szükség, a második függvénnyel szabadíthatjuk fel a lefoglalt memóriaterületet. A *negyzetre* függvény a vektor összes elemét négyzetre emeli, és az utolsó függvény kiírja az elemeket. Az alábbi állományban mutatjuk be ennek a modulnak egy lehetséges megvalósítását.

2.1. kódszöveg. A *vektor* modul.

```

1  #include <iostream>
2  using namespace std;
3  static int* elem;
4  static int meret;
5  void epit(int* az_elem, int a_meret)
6  {
7      meret = a_meret;
8      elem = new int[meret];
9      for(int i = 0; i < meret; i++)
10         elem[i] = az_elem[i];
11 }
12 void felszabadit()
13 {
14     delete [] elem;
15 }
16 void negyzetre()
17 {
18     for(int i = 0; i < meret; i++)
19         elem[i] *= elem[i];
20 }
21 void kiir()
22 {
23     for(int i = 0; i < meret; i++)
24         cout << elem[i] << ' ';
25     cout << endl;
26 }
```

Egy külön állományba helyezzük a fő függvényt. Ez a következő lehet:

2.2. kódszöveg. A fő függvényt tartalmazó állomány.

```

1  void epit( int*, int);
2  void felszabadit();
3  void negyzetre();
4  void kiir();
5  //extern int* elem;
6  void main()
7  {
8      int x[] = {1, 2, 3, 4, 5};
9      epit(x, 5);
10     negyzetre();
11     kiir();
12     felszabadit();
```

```

13     int y[] = {1, 2, 3, 4, 5, 6};
14     epit(y, 6);
15     //elem[1]=10;
16     negyzetre();
17     kiir();
18     felszabadit();
19 }

```

Végrehajtva a programot az alábbi kimenetet kapjuk:

```

1 4 9 16 25
1 4 9 16 25 36

```

A *vektor* modul függvényeinek meghívása előtt a deklarációkat elhelyeztük a fő függvényt tartalmazó állományban. A *main* függvényben előbb egy öt elemből álló *y* vektorral, majd ezt követően egy hat elemből álló *y* vektorral végeztünk műveleteket.

Hangsúlyozzuk, hogy a *vektor* modul bevezetése nem tette lehetővé azt, hogy egyszerre két vektorral tudjunk dolgozni. Például nem tudunk olyan vektorokra vonatkozó műveletet értelmezni, mint az összeadás, amelyben egyszerre több vektorra volna szükség. Figyeljük meg, hogy az *x* vektor által lefoglalt memóriaterületet fel kellett szabadítani még mielőtt az *y* vektort létrehoztuk volna. Ez egy nagy hátránya ennek a megközelítésnek, éppen ezért a következő pontban azt fogjuk vizsgálni, hogy milyen módon tudunk egy olyan saját adattípust létrehozni, amely megengedi, hogy egyszerre több példánnyal dolgozzunk. Ugyanakkor viszont nem szeretnénk lemondani a védettségéről sem, és ez által jutunk el az *osztály* (§2.1.3) fogalmának a bevezetéséhez.

Vegyük észre ugyanakkor azt is, hogy a *vektor* modul valóban biztosítja az adatok védelmét. Ha a vektort az *elem* mutató segítségével direkt módon próbáljuk módosítani, a 15. sorból eltávolítva a megjegyzés jelét, akkor fordítási hibát kapunk. Ha ugyanezt megteesszük az 5. sorban, ez által elhelyezve egy *extern* típusú deklarációt a kódban, akkor ez az állomány önmagában lefordítható lesz, viszont a *szerkesztéskor* jelez hibát a rendszer. Ahhoz, hogy ez a hiba se jelenjen meg, el kell távolítanunk a *static* kulcsszót a 2.1. kódszöveg 3. sorából. Ekkor már valóban módosítható lesz az illető elem, de ez pontosan azt jelenti, hogy nincs védettség. Futtatáskor a kimenet így módosul:

```

1 4 9 16 25
1 100 9 16 25 36

```

Levonhatjuk tehát a következtetést, hogy a moduláris programozás esetén a védettséget valóban a statikus változók valósítják meg.

A moduláris programozás módszerét az adatok védelmén kívül *adatrejtésre* is használhatjuk. Ennek lényege az, hogy a felhasználó csak azt a felületet kell ismerje, amin keresztül feldolgozhatóak az adatok.

2.1.2. Absztrakt adattípusok

Az előző pontban egy példát adtunk a védettség megvalósítására moduláris programozással. Megállapítottuk, hogy az adatoknak és függvényeknek ilyen jellegű megadása nem tette lehetővé azt, hogy egyszerre több példánnyal, például két vektorral,

dolgozzunk. Ezért szükségszerűen jelenik meg az az igény, hogy az adatokat és függvényeket, egy különálló modulhoz hasonlóan, továbbra is egyetlen egységben tároljuk, de legyen lehetőség arra is, hogy több példányt hozzunk létre.

Természetesen merül fel az a lehetőség, hogy a hagyományos struktúra rendeltetésének a kiterjesztése által próbáljuk meg elérni a célunkat. A C++ programozási nyelvben egy struktúrán belül a hagyományos adatokon kívül elhelyezhetünk függvénydeklarációkat, illetve definíciókat is. Ilyen módon egy új típust vezetünk be, amit gyakran *absztrakt adattípusnak* (elvont adattípusnak, vagy *felhasználói típusnak*) nevezünk. Tekintsük az alábbi *taxi* elvont adattípusra vonatkozó forráskódot.

2.3. kódszöveg. A *Taxi* felhasználói típus.

```
1  #include <iostream>
2  using namespace std;
3  struct Taxi {
4      int fizetni;
5      int indulas_ar;
6      int menet_ar;
7      int varakozas_ar;
8      bool van_utas;
9      void Kezdes();
10     bool Beul();
11     int Kiszall();
12     void Megy(int km);
13     void All(int perc);
14 };
15 void Taxi::Kezdes()
16 {
17     indulas_ar = 10;
18     menet_ar = 10;
19     varakozas_ar = 3;
20     fizetni = 0;
21     van_utas = false;
22 }
23 bool Taxi::Beul()
24 {
25     if ( van_utas ) return false;
26     van_utas = true;
27     fizetni = indulas_ar;
28     return true;
29 }
30 int Taxi::Kiszall()
31 {
32     if ( !van_utas ) return 0;
33     van_utas = false;
34     return fizetni;
35 }
36 void Taxi::Megy(int km)
37 {
38     if ( van_utas )
```

```

39         fizetni += menet_ar * km;
40     }
41     void Taxi::All(int perc)
42     {
43         if ( van_utas )
44             fizetni += varakozas_ar * perc;
45     }
46     void main()
47     {
48         Taxi t1, t2;
49         t1.Kezdes();
50         t2.Kezdes();
51         t1.Beul();
52         t1.Megy(4);
53         t2.Beul();
54         t1.All(3);
55         t2.Megy(6);
56         t1.Megy(5);
57         cout << "t1-nek fizetni: ";
58         cout << t1.Kiszall() << endl;
59         cout << "t2-nek fizetni: ";
60         // t2.fizetni = 500;
61         cout << t2.Kiszall() << endl;
62     }

```

A program kimenete a következő lesz:

```

t1-nek fizetni: 109
t2-nek fizetni: 70

```

Megjegyezzük, hogy a 2.3. kódszöveg 3-14 soraiban bevezetett struktúra az adatokon kívül függvénydeklarációkat is tartalmaz. Az elvont adattípusokon belül megadott adatokat *adattagoknak*, a függvényeket pedig *tagfüggvényeknek* nevezzük. A tagfüggvényekre az adattagokhoz hasonlóan a tagkiválasztó operátorral (a *pont* operátor), illetve a struktúra-mutató operátorral (a *->* operátor) hivatkozhatunk.

A struktúrán belül elhelyezhetünk függvénydefiníciókat is, de ez általában csak a nagyon egyszerű függvények esetén ajánlott. Ha egy függvény definíciója a struktúrán belül van, akkor *inline függvényként* kezeli a rendszer. Ha csak a függvény deklarációja kerül a struktúra belsejébe, akkor a definíciót, a névterekhez hasonló módon, úgy adjuk meg, hogy a függvény nevét a struktúra neve és a hatókör operátor előzi meg.

A 2.3. kódszöveg fő függvényéből, illetve a program kimenetéből egyértelműen levonható az a következtetés, hogy a *Taxi* adatszerkezetnek egyszerre több példányával tudunk műveleteket végezni. Az adatok védelme azonban nem valósul meg ebben az esetben. Meggyőződhetünk erről, ha a 60. sorból eltávolítjuk a megjegyzés jelét, és úgy fordítjuk le a kódot. A kimenet a következő lesz:

```

t1-nek fizetni: 109
t2-nek fizetni: 500

```

Tehát a fizetendő összeg módosítható direkt módon, függvénymeghívás nélkül. Ez azt jelenti, hogy nincs biztosítva az adatok védelme. A következő pontban azt vizsgáljuk meg, hogy az absztrakt adattípus fogalma hogyan terjeszthető ki úgy, hogy lehetőséget teremtsen az adatvédelemre.

2.1.3. Osztálydeklaráció

Az előző pontban megállapítottuk, hogy a felhasználói típus bevezetése lehetővé teszi azt, hogy az adatszerkezetnek egyszerre több példányával tudjunk műveleteket végezni. Ugyanakkor, az adatvédelem nem valósul meg egyszerűen az által, hogy adatokat és függvényeket egyetlen struktúra részeként adunk meg. Annak érdekében, hogy ezt a hiányosságot kiküszöböljék, bevezették az *osztály* fogalmát.

Az *osztály* egy olyan absztrakt adattípus, amely lehetőséget teremt az adattagok és tagfüggvények védelmére. Az *osztálydeklaráció* az előző pontban ismertetett felhasználói típus bevezetéséhez hasonló, azzal a különbséggel, hogy a *struct* kulcsszót a *class* (osztály) fogja helyettesíteni. Az osztály tagjaira való hivatkozás a tagkiválasztó operátorral, illetve a struktúra-mutató operátorral történhet, ugyanúgy mint az egyszerű struktúrák, vagy az előző pontban ismertetett elvont adattípusok esetén. Ezt a kérdést a §2.1.4. pontban tárgyaljuk részletesebben.

Mivel az osztály egy felhasználói típus, fontos különbséget tennünk maga az osztály, és ennek példányai között. Egy osztály példányait *objektumoknak* nevezzük. Tehát az objektum általában egy változó, amelynek a típusát az osztálya határozza meg.

Azok a függvénydefiníciók, amelyek az osztályon belül vannak *inline függvényt* eredményeznek ugyanúgy, mint az előző pontban bevezetett felhasználói típusok esetén. Az osztályon kívül elhelyezett függvénydefiníciók is hasonlóak lesznek, tehát az osztály nevét és a hatókör operátort írjuk a függvénynév elé.

Egy osztályon belül a tagok védelme az *elérhetőség* szabályozása által valósul meg. Az adattagok és tagfüggvények elérhetőségét a *private* (privát), *protected* (védett) és *public* (nyilvános) kulcsszavakkal szabályozhatjuk. Mivel a tagok elérhetőségét változtathatják meg, *hozzáférés módosítóknak* is nevezzük őket. A hozzáférés módosítókat mint címkéket használjuk, azaz mindig kettőspont követi őket. Az így kapott címkék több részre osztják az osztály törzsét, ez által szabályozva azt, hogy melyek a nyilvános, védett, illetve privát tagok. Például a *public* címkét követő összes adattag és tagfüggvény nyilvános lesz, egészen a következő címkéig. Jegyezzük meg azt is, hogy osztályok esetén alapértelmezés szerint a tagok privát elérhetőségűek.

A nyilvános tagok elérhetősége nincs korlátozva. Ezeket tetszőleges függvényben használhatjuk, ahol az illető osztály egy példányával dolgozunk. A privát és védett tagok elérhetősége korlátozott. Egyelőre nem teszünk különbséget köztük, csak később az alosztályok (§2.2.2) tanulmányozásakor foglalkozunk ezzel a kérdéssel.

Az objektumokra épülő programozás egyik alapelve az, hogy a nem nyilvános tagokat csak az illető osztály tagfüggvényeiben lehet elérni. Ez a szigorú követelmény bizonyos fokig enyhítve van a C++ programozási nyelvben. Enek megfelelően a privát és védett tagok elérhetősége az illető osztály tagfüggvényeire és *barát (friend) függvényeire* korlátozódik. A barát függvény nem tagfüggvénye az illető osztálynak, de ennek ellenére megengedjük, hogy hozzáférjen a privát és védett tagokhoz. Az előbb

említett alapelvet figyelembe véve megállapíthatjuk, hogy ajánlott a barát függvények számát a minimálisra csökkenteni.

Az osztályok létrehozásakor mindig egy sajátos tagfüggvényt hív meg a rendszer, amit *konstruktor*nak nevezünk. Általában ezt a függvényt használjuk arra, hogy az adattagokat kezdeti értékkel lássuk el. A C++ nyelvben a konstruktor neve mindig megegyezik az osztály nevével, de a függvénynevek túlterhelése lehetővé teszi, hogy egy osztály több konstruktorral rendelkezzen. A konstruktorokkal a §2.1.5. pontban foglalkozunk részletesebben.

Az objektum létrehozása a hagyományos változók bevezetéséhez hasonló, tehát előbb az osztály nevét kell megadni, ami egy típusnév, és ezt követően az objektum nevét. Ha egyszerre több objektumot szeretnénk létrehozni, akkor ezeket vesszővel választhatjuk el. Mivel minden egyes új objektum egy konstruktormeghívást is jelent, ezért a deklaráláskor az objektumnév után kerek zárójelben meg kell adni a konstruktor aktuális paramétereit is.

Jegyezzük meg, hogy az előző pontban bevezetett *struct* kulcsszóval jellemzett felhasználói típus is tulajdonképpen egy osztály, tehát használhatók az elérhetőséget szabályozó címkék. A lényeges különbség az, hogy a *struct* kulcsszó esetén a tagok alapértelmezett elérhetősége nyilvános, míg a *class* esetén privát.

2.1.4. A tagokra való hivatkozás és a *this* mutató

Az előző pontokban láttuk, hogy egy felhasználói típus tagjaira való hivatkozást a tagkiválasztó, illetve a struktúra-mutató operátorral (a *.* és *->* operátorok) végezhetjük. A struktúra-mutató operátort akkor kell használni, ha egy objektumra hivatkozó mutatóval rendelkezünk, ellenkező esetben a tagkiválasztó operátorral dolgozunk.

A továbbiakban moduláris programozás (§2.1.1) esetén ismertetett 2.1. kódszöveget módosítjuk úgy, hogy osztályokra vonatkozzon, majd ezt követően vizsgáljuk a tagokra való hivatkozást.

2.4. kódszöveg. A *vektor* osztály.

```

1  #include <iostream>
2  using namespace std;
3  class vektor {
4  public:
5      vektor(int* az_elem, int a_meret);
6      ~vektor() { delete [] elem; }
7      void negyzetre();
8      void kiir();
9  private:
10     int* elem;
11     int meret;
12 };
13 vektor::vektor(int* az_elem, int a_meret)
14 {
15     meret = a_meret;
16     elem = new int[meret];
17     for(int i = 0; i < meret; i++)

```

```

18         elem[i] = az_elem[i];
19     }
20     void vektor::negyzetre()
21     {
22         for(int i = 0; i < meret; i++)
23             elem[i] *= elem[i];
24     }
25     void vektor::kiir()
26     {
27         for(int i = 0; i < meret; i++)
28             cout << elem[i] << ' ';
29         cout << endl;
30     }
31     void main()
32     {
33         int x[] = {1, 3, 5, 7, 9};
34         vektor v(x, 5);
35         vektor *p = &v;
36         v.kiir();
37         p->negyzetre();
38         p->kiir();
39         v.kiir();
40     }

```

A fenti kódszöveg fő függvényében előbb a v vektort vezettük be, majd a p mutatót, amely a v vektorra hivatkozik. Ez azt is jelenti, hogy a p segítségével előidézett változtatások a v vektorban is tükröződnek. Valóban a kimenet a következő lesz:

```

1 3 5 7 9
1 9 25 49 81
1 9 25 49 81

```

Tehát az elemenként négyzetreemelt vektor jelenik meg kétszer a képernyőn. Figyeljük meg, hogy a v esetén a tagkiválasztó operátort, a p esetén pedig a struktúra-mutató operátort használtuk.

Figyeljük meg, hogy a tagfüggvények belsejében direkt módon hivatkozhatunk az osztály tagjaira, nincs szükség tagkiválasztó, vagy struktúra-mutató operátorra. Mégis, felmerül a kérdés, hogy milyen módon azonosítja a rendszer az illető adattagot, tudva azt, hogy egy osztálynak több objektumát is létrehoztuk. A megoldás a *this* mutató használatában rejlik, mivel a tagfüggvények belsejében a tagokra való hivatkozás ezzel a mutatóval történik.

Pontosabban arról van szó, hogy minden egyes objektumon belül a rendszer létrehozza a *this* mutatót, amely az aktuális objektumra mutat. Például a 2.4. kódszöveg fő függvényében bevezetett v objektum esetén a *this* ennek az objektumnak a címe. Ha pedig az ugyanott definiált p mutatót tekintjük, akkor a *this* megegyezik p -vel.

Ennek alapján már könnyen azonosíthatóak a különböző objektumok tagjai. Az illető osztály tagfüggvényeiben a rendszer egyszerűen elvégez egy helyettesítést, azaz minden *tag* helyett *this->tag* lesz. Például a 2.4. kódszöveg *negyzetre* tagfüggvénye így alakul:


```

void vektor::negyzetre()
{
    for(int i = 0; i < this->meret; i++)
        this->elem[i] *= this->elem[i];
}

```

Hangsúlyozzuk, hogy nem kell mi megadnunk a fenti esetben a *this* mutatót, ezt automatikusan elhelyezi a rendszer. Mégis, a *this* mutatót explicit módon is használhatjuk, ha erre szükség van.

2.1.5. A konstruktor

Az előző pontok alapján tudjuk, hogy egy objektum létrehozását a konstruktorral végezzük. Továbbá, a konstruktor neve meg kell egyezzen az osztály nevével. Mégis, mivel a függvények túlterhelhetők, egy osztálynak több konstruktora is lehet, feltéve ha a paraméterlisták különböznek. Fontos, hogy a konstruktor nem térít vissza értéket. A konstruktor deklarációja nem tartalmazhat semmit a visszatérítendő típus helyén, még a *void* kulcsszót sem.

Az alábbi példa több konstruktor együttes használatát szemlélteti. Egy olyan osztályt hozunk létre, amely különböző személyek családnevét és keresztnévét tárolja.

2.5. kódszöveg. A *szemely.h* fejléc.

```

1  #include <iostream>
2  using namespace std;
3  class szemely {
4      char* cs_nev;
5      char* sz_nev;
6  public:
7      szemely();          //alapértelmezett konstruktor
8      szemely(char* cs_n, char* sz_n);
9      szemely(const szemely& sz);    // másoló konstruktor
10     ~szemely();
11     void kiir();
12 };
13 szemely::szemely() {
14     cs_nev = new char[1];
15     *cs_nev = 0;        // 0 és '\0' ugyanaz
16     sz_nev = new char[1];
17     *sz_nev = 0;
18     cout << "Alapertelmezett konstruktor\n";
19 }
20 szemely::szemely(char* cs_n, char* sz_n)
21 {
22     cs_nev = new char[strlen(cs_n)+1];
23     sz_nev = new char[strlen(sz_n)+1];
24     strcpy(cs_nev, cs_n);
25     strcpy(sz_nev, sz_n);
26     cout << "Hagyomanyos konstruktor\n";
27 }

```

```

28     személy::szemely(const személy& x)
29     {
30         cs_nev = new char[strlen(x.cs_nev)+1];
31         strcpy(cs_nev, x.cs_nev);
32         sz_nev = new char[strlen(x.sz_nev)+1];
33         strcpy(sz_nev, x.sz_nev);
34         cout << "Masolo konstruktor\n";
35     }
36     személy::~szemely() {
37         cout << "Destruktor\n";
38         delete[] cs_nev;
39         delete[] sz_nev;
40     }
41     void személy::kiir() {
42         if ( strlen(cs_nev) > 0 )
43             cout << cs_nev << ' ' << sz_nev << endl;
44         else
45             cout << "Nincs adat\n";
46     }

```

Ez a forráskód három konstruktort tartalmaz. Ezek közül a 8. sorbeli konstruktordeklarációt hagyományosnak tekinthetjük abban az értelemben, hogy az adattagok (családnév és keresztnév) kezdeti értékkel való ellátását valósítja meg. Figyeljük meg, hogy két sajátos konstruktor is szerepel a fenti kódban. Az egyik az *alapértelmezett konstruktor*, vagy más néven *alapértelmezés szerinti konstruktor*, a másik a *másoló konstruktor*.

Ha a konstruktor formális paramétereinek listája üres, akkor beszélünk alapértelmezett konstruktorról. Az alapértelmezés szerinti konstruktoroknak fontos szerepe van azoknak az objektumoknak a létrehozásában, amelyek nem rendelkeznek kezdeti értékekkel megadható aktuális paraméterekkel. Pontosabban, ha egy osztálynak van alapértelmezett konstruktora, akkor létrehozható olyan objektum, amely nem tartalmaz inicializáló aktuális paraméterekből álló listát. Ez akkor is lehetséges, ha olyan konstruktorunk van, amelynek az összes formális paramétere kezdeti értékkel van ellátva. Tehát az ilyen konstruktort is alapértelmezett konstruktoroknak nevezhetjük.

A konstruktorokon kívül a 2.5. kódszöveg tartalmaz egy sajátos tagfüggvényt, a *destruktor*t, melyet az objektumok megszűnésekor hív meg a rendszer.

Tekintsük a 2.5. kódszöveget felhasználó alábbi fő függvényt:

2.6. kódszöveg. A *szemely* osztály objektumainak létrehozása.

```

1     #include "szemely.h"
2     void main() {
3         személy BF("Bolyai", "Farkas");
4         BF.kiir();
5         személy *FGy = new személy("Farkas", "Gyula");
6         FGY->kiir();
7         személy A; //alapértelmezett konstruktor
8         A.kiir();
9         személy Gyula(*FGy); // masoló konstruktor
10        Gyula.kiir();

```

```

11     delete FGy;
12 }

```

Ennek a kódnak a kimenete a következő lesz:

```

Hagyományos konstruktor
Bolyai Farkas
Hagyományos konstruktor
Farkas Gyula
Alapértelmezett konstruktor
Nincs adat
Masolo konstruktor
Farkas Gyula
Destruktor
Destruktor
Destruktor
Destruktor

```

Megfigyelhetjük, hogy először a *BF* objektumot hoztuk létre a hagyományos konstruktorral. Ezt követően a szabad tárban jön létre egy objektum, amelyre az *FGy* mutatóval hivatkozhatunk. Itt is a hagyományos konstruktort hívtuk meg a rendszer, mivel a *new* operátor után az osztály nevet és, kerek zárójelben, az aktuális paraméterek listáját adtuk meg. Az *A* objektumot az alapértelmezett, a *Gyula* objektumot pedig a másoló konstruktorral hoztuk létre.

A alapértelmezett konstruktor mindkét adattagba az üres karakterláncot másolja. Mivel ennek a hossza zéró, a *kiir* tagfüggvény a „*Nincs adat*” üzenetet jeleníti meg. Feltételeztük, hogy ha a családnév üres, akkor a keresztnevet sem adtuk meg.

Egy osztályt úgy is deklarálhatunk, hogy nem adunk meg konstruktort. Jegyezzük meg, hogy ha nincs, a programozó által bevezetett konstruktor, akkor a rendszer létrehoz egy alapértelmezett konstruktort, és ezt hívja meg minden alkalommal, amikor egy új objektum keletkezik. Ez a konstruktor nem ad kezdeti értékeket az adattagoknak.

Ha a programozó létrehozott egy vagy több konstruktort, akkor a rendszer nem generál alapértelmezett konstruktort. Ha ezen konstruktorok közül egyik sem alapértelmezett, és szeretnénk olyan objektumot létrehozni, amely nem tartalmaz aktuális paraméterekből álló listát, akkor kötelesek vagyunk egy alapértelmezett konstruktort definiálni.

A másoló konstruktor célja az, hogy egy objektumot kezdeti értékekkel lásson el egy ugyanolyan típusú objektum segítségével. Általában az

```
osztálynév(const osztálynév & objektum);
```

alakban deklaráljuk, ahol a *const* kulcsszó arra utal, hogy a paraméterként megadott objektum nem változik.

Ha a programozó nem definiál másoló konstruktort, akkor a rendszer létrehoz egy másoló konstruktort, amely az adattagok *bitenkénti másolását* végzi. Ez azt jelenti, hogy megfelelteti egymásnak a rendszer az adattagokat, majd a forrás adattag biteit rendre átmásolja a cél adattagba. A bitenkénti másolás általában akkor ad helyes eredményt, ha az osztálynak nincsen mutató típusú adattagja.

Például a 2.5. és 2.6. kódszövegek esetén, ha nem definiáltunk volna másoló konstruktort, akkor futási időben hibát észleltünk volna. Pontosabban, kétszer próbálta volna meg felszabadítani ugyanazt a memóriaterületet a rendszer. Ennek a hibának az oka abban rejlik, hogy a *Gyula* objektum létrehozásakor egy bitenkénti másolást végzett a rendszer, tehát a **FGy* objektum *cs_nev* és *sz_nev* adattagjait másolta át. Mivel mindkét adattag értéke egy cím, ezért ezt a címet másoltuk át, tehát a *Gyula* objektum *cs_nev* és *sz_nev* adattagjai ugyanarra a memóriaterületre fognak mutatni, ahova a **FGy* objektum adattagjai. Ez viszont nem az, amit meg szeretnénk volna tenni, mivel így, ha az egyik objektum megszűnik, a másiknak is fel lesz szabadítva a memóriaterülete és fordítva. E helyett a másoló konstruktort terheljük túl, amely új memóriaterületet foglal le, és erre másolja a családnevet és keresztnévet.

Jegyezzük meg, hogy a rendszer akkor hívja meg a másoló konstruktort, ha:

- ugyanolyan típusú objektummal adunk kezdőértéket;
- egy függvénynek a paramétere egy objektum;
- egy függvény objektumot térít vissza.

Ezért, ha van mutató típusú adattag, akkor a másoló konstruktort definiálnunk kell akkor is, ha nincs szándékunkban a kezdőértékadást ugyanolyan típusú objektummal végezni.

A 2.6. kódszövegben a *new* operátorral dinamikus módon hoztuk létre az egyik objektumot. A *new* utáni típust követően kerek zárójelt használtunk, és ezen belül adtuk meg a konstruktor aktuális paramétereit.

Lehetőség van arra, hogy egy osztály törzsében osztály típusú tagokat helyezünk el. A következő példa keretében azt vázoljuk fel, hogy ha egy osztályon belül *n* darab különböző osztály típusú tagot helyezünk el, akkor hogyan alakul az illető osztály konstruktora.

```
class oszt {
    oszt_1 ob_1;
    oszt_2 ob_2;
    ...
    oszt_n ob_n;
};
```

Ebben az esetben az *oszt* osztály konstruktorának a fejléce a következőképpen adható meg:

```
oszt(argumentumlista) : objektumlista
```

az *objektumlista* pedig az

```
ob_1(arglista_1), ob_2(arglista_2), ..., ob_n(arglista_n)
```

alakú kell legyen. Természetesen, sem itt, sem az osztálydeklarációban a három pont nem része a szintaxisnak, csak jelzi a folytatást. Az *argumentumlista* az *oszt* osztály konstruktorában a formális paraméterek listája. Továbbá, minden egyes *i* értékre 1-től *n*-ig az *arglista_i* az *ob_i* osztály konstruktorában az aktuális paraméterek listája. Az

egyres objektumok aktuális paramétererei az *argumentumlistából* alkotott kifejezések lesznek.

Jegyezzük meg, hogy az objektumlistából hiányoznak azok az objektumok, amelyek nem rendelkeznek a programozó által bevezetett konstruktorral. Ezen kívül hiányozhatnak az objektumlistából azok az objektumok is, amelyekre az alapértelmezett konstruktort szeretnénk meghívni.

Egy másik fontos észrevétel a következő. Ha egy osztálynak egyik adattagja egy objektum, akkor először ennek az objektumnak a konstruktorát hívja meg a rendszer, majd ezt követően lesz végrehajtva az osztály konstruktorának a törzse.

A továbbiakban a 2.5. kódszöveget úgy módosítjuk, hogy eltávolítjuk a konstruktorokból és a destruktorból a kiírásokat, vagyis a 18., 26., 34. és 37. sorokat töröljük. Legyen az így kapott állomány neve *szemely2.h*. Ezt felhasználva a következő példa *házaspárok* adatait tárolja, mégpedig úgy, hogy osztály típusú tagokat használ.

2.7. kódszöveg. Osztály típusú tagok.

```

1  #include "szemely2.h"
2  class hazaspar {
3      személy ferj;
4      személy feleseg;
5  public:
6      hazaspar() // alapértelmezett konstruktor
7      {
8      }
9      hazaspar(személy& aferj, személy& afelesege);
10     hazaspar(char* cs_ferj, char* sz_ferj,
11             char* cs_felesege, char* sz_felesege):
12     ferj(cs_ferj, sz_ferj), feleseg(cs_felesege, sz_felesege)
13     {
14     }
15     void kiir();
16 };
17 inline hazaspar::hazaspar(személy& aferj, személy& afelesege):
18 ferj(aferj), feleseg(afelesege)
19 {
20 }
21 void hazaspar::kiir()
22 {
23     cout << "ferj: ";
24     ferj.kiir();
25     cout << "felesege: ";
26     feleseg.kiir();
27 }
28 void main() {
29     személy Ady("Ady", "Endre");
30     személy Csinszka("Boncza", "Berta");
31     hazaspar Hpar(Ady, Csinszka);
32     Hpar.kiir();
33     hazaspar Petofi("Petofi", "Sandor", "Szendrei", "Julia");
34     Petofi.kiir();

```

```

35     hazaspar XY;
36     XY.kiir();
37 }

```

A program kimenete a következő lesz:

```

ferj:  Ady Endre
feleseg:  Boncza Berta
ferj:  Petofi Sandor
feleseg:  Szendrei Julia
ferj:  Nincs adat
feleseg:  Nincs adat

```

A 2.7. kódszöveg három konstruktorral rendelkezik. Az alapértelmezett konstruktor definíciója is az osztályon belülrre került, ezért ez helyben kifejtett függvény (*inline* függvény) lesz. Mivel a konstruktor fejlécét úgy adtuk meg, hogy hiányzik a kettőspont, és az azt követő objektumlista, ezért ez a konstruktor az összes osztály típusú tagnak az alapértelmezett konstruktorát hívja meg. Erre utal az is, hogy a fő függvényben az XY objektum kiírásakor a „Nincs adat” üzenet jelenik meg.

A 9. sorban egy konstruktordeklaráció szerepel, a definíció most az osztályon kívülre került. Mivel azt szeretnénk, hogy ez is helyben kifejtett függvény legyen az *inline* minősítőt használjuk a függvénydefinícióban. Ez a konstruktor a személy osztály másoló konstruktorával hozza létre a *ferj* és *feleseg* tagokat.

A harmadik konstruktor a családnevekkel és személynevekkel hozza létre az osztály típusú tagokat. Ezért a *szemely* osztály hagyományos konstruktorát hívja meg a rendszer mindkét adattagra.

2.1.6. A destruktork

Az eddigi pontok alapján tudjuk, hogy ha egy objektum megszűnik, akkor a rendszer automatikusan végrehajt egy sajátos tagfüggvényt, amit *destruktor*nak nevezünk. A továbbiakban részletesebben vizsgáljuk a destruktort.

A destruktork neve mindig a ~ karakterrel kezdődik, és ez után az osztály neve következik. A konstruktorhoz hasonlóan a destruktork sem térít vissza értéket, és még a *void* típust sem szabad megadni a visszatérítendő érték típusaként.

Felmerül a kérdés, hogy mikor hívódnak meg az egyes destruktorkok. Ez a hatókörtől függ. Egy globális objektum destruktorka a *main* függvény végén az *exit* függvény részeként lesz végrehajtva. Ezért nem szabad az *exit* függvényt meghívni a destruktorkban, mivel ez végtelen ciklust eredményezhet.

Egy helyi objektum destruktorkát akkor hívja meg a rendszer, ha annak a blokknak a végére értünk, amelyben be volt vezetve.

Végül tekintsük azt az esetet is, amikor a *new* operátorral hoztunk létre a szabad tárban egy objektumot. Ezeket dinamikus módon létrehozott objektumoknak is nevezzük. Ekkor a destruktork a *delete* operátoron keresztül hívja meg a rendszer. Valóban ekkor lesz felszabadítva a *new* operátor által lefoglalt memóriaterület.

A továbbiakban egy olyan példa keretében szemléltetjük a destruktork működését, amely minden esetben kiírja, hogy éppen mit végzett, azaz milyen konstruktorkt vagy destruktorkt hívott meg. A kiírást most a *printf* függvényvel végezzük.

2.8. kódszöveg. A destruktork.

```
1  #include <cstdio>
2  #include <cstring>
3  using namespace std;
4  class kiiras {
5      char* nev;
6  public:
7      kiiras(char* n);
8      ~kiiras();
9  };
10 kiiras::kiiras(char* n)
11 {
12     nev = new char[strlen(n)+1];
13     strcpy(nev, n);
14     printf("Letrehoztam:  %s\n", nev);
15 }
16 kiiras::~kiiras()
17 {
18     printf("Felszabaditottam:  %s\n", nev);
19     delete nev;
20 }
21 void fuggv()
22 {
23     printf("Fuggvenymeghivas.\n");
24     kiiras helyi("HELYI");
25 }
26 kiiras globalis("GLOBALIS");
27 void main() {
28     kiiras* dinamikus = new kiiras("DINAMIKUS");
29     fuggv();
30     printf("Folytatodik a fo fuggveny.\n");
31     delete dinamikus;
32 }
```

Végrehajtva a programot, a következő kimenetet kapjuk:

```
Letrehoztam: GLOBALIS
Letrehoztam: DINAMIKUS
Fuggvenymeghivas.
Letrehoztam: HELYI
Felszabaditottam: HELYI
Folytatodik a fo fuggveny.
Felszabaditottam: DINAMIKUS
Felszabaditottam: GLOBALIS
```

A forráskódban egy *kiiras* nevű osztályt vezetünk be, és létrehoztuk ennek három objektumát. Figyeljük meg, hogy a globális objektumot hozta először létre a rendszer, ugyanakkor ennek a destruktora lesz utolsónak végrehajtva. A helyi objektum destruktora a függvényből való kilépéskor, a dinamikus objektumé pedig a *delete* operátor részeként hívódik meg.

2.2. Az objektumorientált programozási módszer

2.2.1. Elméleti alapok

Az objektum adattagokat és tagfüggvényeket tartalmaz. Ha nem használunk barát függvényeket a védett tagok csak a tagfüggvényekben érhetőek el. Ezt a tulajdonságot *egybezártságnak (zártságnak)* nevezzük.

A gyakorlatban viszont nem csak különálló objektumokkal találkozunk. A különböző objektumok közti kapcsolatok is fontosak. Egy osztály örökölheti egy másik osztály tagjait. Az eredeti osztály neve *alaposztály*, vagy *bázisosztály*. Az örökléssel létrehozott osztályt *származtatott osztálynak* nevezzük. Az adattagok, és a tagfüggvények is öröklődnek. Ha egy osztály több alaposztállyal rendelkezik, akkor *többszörös öröklésről* beszélünk. Az *öröklés* egy másik fontos tulajdonsága az objektumoknak. Az objektumok egy hierarchiát alkothatnak.

Az öröklött tagfüggvények túlterhelhetőek. Nem csak a függvény neve, hanem a paraméterlistája is ugyanaz lehet. Az objektumhierarchia különböző szintjein ugyanannak a műveletnek más és más értelme lehet. Ezt a tulajdonságot *polimorfizmusnak* nevezzük.

2.2.2. Származtatott osztályok deklarálása

A C++ programozási nyelvben a származtatott osztályokat az alábbi módon adjuk meg:

```
class oszt : alaposztálylista {
    // új adattagok és tagfüggvények
};
```

ahol az alaposztálylista vesszővel elválasztott elemei

```
public alaposztály
protected alaposztály
private alaposztály
```

alakúak kell legyenek. Ha minden egyes esetben a *public* hozzáférésmódosítót használjuk, akkor a

```
class oszt : public oszt_1, ..., public oszt_n {
    // ...
};
```

alakú szerkezetet kapjuk, ahol az *oszt* osztály az *oszt_1*, ..., *oszt_n* osztályok származtatott osztálya. Jegyezzük meg, hogy a konstruktorok és destruktorok nem öröklődnek. A származtatott osztály konstruktorát az

```
oszt(paraméterlista) :
    oszt_1(lista1), ..., oszt_n(lista_n)
{
    // ...
}
```


módon definiáljuk. A következő pontban olyan példákat adunk származtatott osztályra, amelyek lehetőséget teremtenek a *virtuális tagfüggvények* bevezetésére is.

2.2.3. Virtuális tagfüggvények

Tekintsük egy olyan példát származtatott osztályra, amelyben az *alap* nevű osztályban két függvényt deklarálunk, és a második meghívja az elsőt. Ugyanakkor a származtatott osztályban csak az elsőt írjuk felül.

2.9. kódszöveg. Virtuális tagfüggvény.

```

1  #include <iostream>
2  using namespace std;
3  class alap {      // az alaposztály
4  public:
5      void f1();
6      void f2();
7  };
8  class szarm : public alap {
9  public:
10     void f1();
11 };
12 void alap::f1()
13 {
14     cout << "alap:  f1\n";
15 }
16 void alap::f2()
17 {
18     cout << "alap:  f2\n";
19     f1();      // az f2 meghívja az f1-et.
20 }
21 void szarm::f1()
22 {
23     cout << "szarmaztatott:  f1\n";
24 }
25 void main() {
26     szarm s;
27     s.f2();
28 }
```

Figyeljük meg, hogy csak az *f1* tagfüggvényt írtuk felül, az *f2* öröklődik az alaposztálytól. A fő függvényben a származtatott osztálynak hoztuk létre egy objektumát és az erre az *f2* függvényt hívtuk meg. Felmerül a kérdés, hogy ilyen módon melyik *f1* függvény lesz végrehajtva?

A 2.9. kódszöveg esetén az *f1* függvény kiválasztása fordítási időben történt, ezért az alaposztály *f1* tagfüggvénye lesz végrehajtva. Ezt a tulajdonságot *statikus kötésnek* nevezzük.

Ha a végrehajtandó függvény kiválasztása futási időben történik, akkor *dinamikus kötésről* beszélünk. A dinamikus kötést virtuális tagfüggvények segítségével valósíthatjuk meg. Az *f1* tagfüggvényt kell virtuálisnak deklarálni. Ezt úgy tehetjük meg,

hogy a *virtual* minősítőt használjuk a függvény alapszálybeli deklarációjában. Ebben az esetben az alapszályt a

```
class alap {
public:
    virtual void f1();
    void f2();
};
```

alakban adjuk meg. Így a származtatott osztálybeli *f1* függvény lesz végrehajtva.

Figyeljük meg, hogy a *virtual* kulcsszót elég egyszer megadni, az alapszálybeli deklarációban. Ebben az esetben a származtatott osztályban deklarált túlterhelt tagfüggvény is virtuális lesz. Ha egy függvényt virtuálisnak deklaráltunk az alapszályban, akkor az osztályhierarchia tetszőleges származtatott osztályában virtuális lesz.

A továbbiakban tekintsünk egy másik példát, amelyben felmerül a virtuális tagfüggvények megadásának a szükségszerűsége. Vezessük be a racionális számokra vonatkozó *tort* nevű osztályt, amely két egész típusú adattaggal rendelkezik, melyek a számlálónak és nevezőnek felelnek meg. Az osztály kell rendelkezzen egy olyan konstruktorral, amely a számlálót és a nevezőt kezdeti értékekkel látja el. Alapértelmezetten a számláló értéke legyen 1, a nevezője pedig 0. Továbbá, az osztálynak kell legyen egy *szorzat* és egy *szoroz* nevű tagfüggvénye is. Az első a két tört szorzatát számolja ki, a második pedig az aktuális objektumot módosítja úgy, hogy azt megszorozza a paraméterként megadott objektummal. Ugyanakkor a *tort* osztálynak kell legyen egy olyan tagfüggvénye is, amely az illető racionális számot írja ki.

A fenti osztályt felhasználva egy olyan *tort_kiir* nevű osztályt is létre kell hozni, amely a *szorzat* tagfüggvényt úgy módosítja, hogy a művelet elvégzésén kívül maga a művelet is jelenjen meg a szabványos kimeneten. A *szoroz* tagfüggvényt nem írjuk felül, de a műveletnek ebben az esetben is meg kell jelennie.

2.10. kódszöveg. A szorzat virtuális tagfüggvény bevezetése a racionális számokra vonatkozó osztály esetén.

```
1  #include <iostream>
2  using namespace std;
3  class tort {
4  protected:
5      int szamlalo;
6      int nevezo;
7  public:
8      tort(int szamlalol = 0, int nevezol = 1);
9      /*virtual*/ tort szorzat(tort& r);
10     tort& szoroz(tort& r);
11     void kiir();
12 };
13 tort::tort(int szamlalol, int nevezol)
14 {
15     szamlalo = szamlalol;
16     nevezo = nevezol;
17 }
18 // két tört szorzatát számolja ki, de nem egyszerűsít
```

```
19     tort tort::szorzat(tort& r)
20     {
21         return tort(szamlalo * r.szamlalo, nevezo * r.nevezo);
22     }
23     // az aktuális objektumot módosítja
24     tort& tort::szoroz(tort& q)
25     {
26         *this = this->szorzat(q);
27         return *this;
28     }
29     void tort::kiir()
30     {
31         if ( nevezo )
32             cout << szamlalo << " / " << nevezo;
33         else
34             cerr << "helytelen tort";
35     }
36     class tort_kiir: public tort {
37     public:
38         tort_kiir( int szamlalol = 0, int nevezol = 1 );
39         tort szorzat( tort& r);
40     };
41     inline tort_kiir::tort_kiir(int szamlalol, int nevezol) :
42     tort(szamlalol, nevezol)
43     {
44     }
45     tort tort_kiir::szorzat(tort& q)
46     {
47         tort r = tort(*this).szorzat(q);
48         cout << "(";
49         this->kiir();
50         cout << ")" * "(";
51         q.kiir();
52         cout << ")" = ";
53         r.kiir();
54         cout << endl;
55         return r;
56     }
57     int main()
58     {
59         tort p(3,4), q(5,2), r;
60         r = p.szoroz(q);
61         p.kiir();
62         cout << endl;
63         r.kiir();
64         cout << endl;
65         tort_kiir p1(3,4), q1(5,2);
66         tort r1, r2;
67         r1 = p1.szorzat(q1);
68         r2 = p1.szoroz(q1);
```

```

69     p1.kiir();
70     cout << endl;
71     r1.kiir();
72     cout << endl;
73     r2.kiir();
74     cout << endl;
75     return 0;
76 }

```

A programot végrehajtva az alábbi kimenetet kapjuk:

```

15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8

```

Figyeljük meg, hogy a kapott eredmény nem megfelelő, mivel a művelet kiírása csak egy alkalommal jelent meg. Ahhoz, hogy az elvárt eredményt kapjunk, a *szorzat* tagfüggvényt virtuálisnak kell deklarálni, és ezt úgy tehetjük meg, hogy a 2.10. kódszöveg 9. sorából eltávolítjuk a megjegyzés jelét. Ha ezt meg tesszük, akkor a kimenet így módosul:

```

15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8

```

tehát valóban kétszer jelenik meg a műveletre vonatkozó kiírás.

2.2.4. Absztrakt osztályok

Egy alaposztálynak lehetnek olyan általános tulajdonságai, amelyekről tudunk, de nem tudjuk őket definiálni csak egy származtatott osztályban. Ebben az esetben egy olyan virtuális tagfüggvényt deklarálhatunk, amely nem lesz definiálva az alaposztályban. Azokat a tagfüggvényeket, amelyek deklarálva vannak, de nincsenek definiálva egy adott osztályban, *tiszta virtuális tagfüggvényeknek* nevezzük.

A tiszta virtuális tagfüggvényt a szokásos módon deklaráljuk, de a fejléc után az = 0 karaktereket írjuk. Ez jelzi, hogy a tagfüggvényt nem fogjuk definiálni.

Azokat az osztályokat, amelyek tartalmaznak legalább egy tiszta virtuális tagfüggvényt, *absztrakt osztályoknak* nevezzük. Az absztrakt osztályoknak nem hozhatjuk létre objektumát.

A tiszta virtuális tagfüggvényeket felül kell írni a származtatott osztályban, ellenkező esetben az illető osztály is absztrakt lesz.

Tekintsük a következő példát

2.11. kódszöveg. Absztrakt osztály.

```
1  #include <iostream>
2  using namespace std;
3  class allat {
4  protected:
5      double suly;      // kg
6      double eletkor;   // ev
7      double sebesseg;  // km / h
8  public:
9      allat( double su, double k, double se);
10     virtual double atlagos_suly() = 0;
11     virtual double atlagos_eletkor() = 0;
12     virtual double atlagos_sebesseg() = 0;
13     int kover() { return suly > atlagos_suly(); }
14     int gyors() { return sebesseg > atlagos_sebesseg(); }
15     int fiatal() { return 2 * eletkor < atlagos_eletkor(); }
16     void kiir();
17 };
18 allat::allat( double su, double k, double se)
19 {
20     suly = su;
21     eletkor = k;
22     sebesseg = se;
23 }
24 void allat::kiir()
25 {
26     cout << ( kover() ? "kover, " : "sovany, " );
27     cout << ( fiatal() ? "fiatal, " : "oreg, " );
28     cout << ( gyors() ? "gyors" : "lassu" ) << endl;
29 }
30 class galamb : public allat {
31 public:
32     galamb( double su, double k, double se):
33     allat(su, k, se) {}
34     double atlagos_suly() { return 0.5; }
35     double atlagos_eletkor() { return 6; }
36     double atlagos_sebesseg() { return 90; }
37 };
38 class medve: public allat {
39 public:
40     medve( double su, double k, double se):
41     allat(su, k, se) {}
42     double atlagos_suly() { return 450; }
43     double atlagos_eletkor() { return 43; }
44     double atlagos_sebesseg() { return 40; }
45 };
46 class lo: public allat {
47 public:
48     lo( double su, double k, double se):
```

```

49     allat(su, k, se) {}
50     double atlagos_suly() { return 1000; }
51     double atlagos_eletkor() { return 36; }
52     double atlagos_sebesseg() { return 60; }
53 };
54 void main() {
55     galamb g(0.6, 1, 80);
56     medve m(500, 40, 46);
57     lo l(900, 8, 70);
58     g.kiir();
59     m.kiir();
60     l.kiir();
61 }

```

A programot futtatva az alábbi kimenetet kapjuk:

```

kover, fiatal, lassu
kover, oreg, gyors
sovany, fiatal, gyors

```

Figyeljük meg, hogy annak ellenére, hogy az *allat* osztályt absztraktnak deklarál-
tuk, hasznos volt ennek bevezetése, mivel egyes tagfüggvényeket már az alaposztály
szintjén definiálni lehetett. Ezek öröklődtek a származtatottakba és így nem kellett őket
minden egyes esetben külön-külön megírni.

2.2.5. Az interfész fogalma

A C++ programozási nyelvben az interfész fogalma nincsen értelmezve abban a
formában, ahogyan az létezik a Java és C# programozási nyelvekben. De tetszőleges
olyan absztrakt osztályt, amely csak tiszta virtuális függvényeket tartalmaz interfész-
nek tekinthetünk. Természetesen ebben az esetben nem fogunk deklarálni adattagokat
sem az osztályon belül. Az előző pontban bevezetett *allat* nevű osztály adattagokat
is és nem virtuális függvényeket is tartalmaz, ezért ez nem tekinthető interfésznek. A
továbbiakban egy *Jarmu* nevű absztrakt osztályt adunk meg, amely csak tiszta virtuális
tagfüggvényekkel rendelkezik. Ugyanakkor ennek az osztálynak két származtatottját
is létrehozuk.

2.12. kódszöveg. Absztrakt osztály, amely interfésznek tekinthető.

```

1     #include <iostream>
2     using namespace std;
3     class Jarmu
4     {
5     public:
6         virtual void Indul() = 0;
7         virtual void Megall() = 0;
8         virtual void Megy(int km) = 0;
9         virtual void All(int perc) = 0;
10    };
11    class Bicikli : public Jarmu
12    {

```

```
13     public:
14         void Indul();
15         void Megall();
16         void Megy(int km);
17         void All(int perc);
18     };
19     void Bicikli::Indul() {
20         cout << "Indul a bicikli." << endl;
21     }
22     void Bicikli::Megall() {
23         cout << "Megall a bicikli." << endl;
24     }
25     void Bicikli::Megy(int km) {
26         cout << "Biciklizik " << km << " kilometert." << endl;
27     }
28     void Bicikli::All(int perc) {
29         cout << "A bicikli all " << perc << " percet." << endl;
30     }
31     class Auto : public Jarmu
32     {
33     public:
34         void Indul();
35         void Megall();
36         void Megy(int km);
37         void All(int perc);
38     };
39     void Auto::Indul() {
40         cout << "Indul az auto." << endl;
41     }
42     void Auto::Megall() {
43         cout << "Megall az auto." << endl;
44     }
45     void Auto::Megy(int km) {
46         cout << "Az auto megy " << km << " kilometert." << endl;
47     }
48     void Auto::All(int perc) {
49         cout << "Az auto all " << perc << " percet." << endl;
50     }
51     void BejarUt(Jarmu *j)
52     {
53         j->Indul();
54         j->Megy(3);
55         j->All(1);
56         j->Megy(2);
57         j->Megall();
58     }
59     int main()
60     {
61         Jarmu *b = new Bicikli;
62         BejarUt(b);
```

```
63     Jarmu *a = new Auto;
64     BejarUt(a);
65     delete a;
66     delete b;
67 }
```

A fő függvényben egy *Bicikli* és egy *Auto* típusú dinamikus objektumot deklarálunk. Ha ezekre az objektumokra a *BejarUt* nevű tagfüggvényt hívjuk meg, különböző eredményt kapunk, annak ellenére, hogy a függvénynek csak egy olyan paramétere van, amely a *Jarmu* absztrakt osztályra hivatkozó mutató.

2.3. Javasolt feladatok

1. Írjunk programot a Python, C++, Java, C# programozási nyelvek egyikében, amely
 - a) egy **Diak** nevű osztályt definiál, amely tartalmaz egy *nev* karakterlánc típusú attribútumot és egy egész elemekből álló tömbként megadott *jegyek* attribútumot, az adott nevű diák jegyeivel. Továbbá, adjunk meg konstruktorokat, az attribútumokat beállító és lekérdező metódusokat, valamint egy olyan metódust, amely a diák médiáját számolja ki.
 - b) Írjunk olyan függvényt, amely paraméterként kap egy **Diak** típusú objektumot és igazat térít vissza, ha a diák összes jegye nagyobb mint 4.
 - c) Írjuk le a **Diak** osztályon belül megadott metódusok, valamint a b) pontbeli függvény specifikációját.
2. Írjunk programot a Python, C++, Java, C# programozási nyelvek egyikében, amely
 - a) egy **Diak** nevű osztályt definiál, amely tartalmaz egy *nev* karakterlánc típusú attribútumot és egy egész elemekből álló tömbként megadott *jegyek* attribútumot, az adott nevű diák jegyeivel. Továbbá, adjunk meg konstruktorokat, az attribútumokat beállító és lekérdező metódusokat, valamint egy olyan metódust, amely a diák médiáját számolja ki.
 - b) Írjunk olyan függvényt, amely paraméterként kap egy **Diak** típusú objektumot és kiírja a diák nevét, valamint a jegyeit csökkenő sorrendben.
 - c) Írjuk le a **Diak** osztályon belül megadott metódusok, valamint a b) pontbeli függvény specifikációját.