Babeș-Bolyai Tudományegyetem, Kolozsvár

Informatikai-matematika záróvizsga tankönyv

2017

Tematika (algoritmusok és programozás)

1. Keresés (szekvenciális, bináris), rendezés (kiválasztásos, buborékrendezés, quicksort).
2. A visszalépéses keresés (backtracking) és az oszd meg és uralkodj (divide et impera) módszere.
3. Algoritmusok és specifikációk. Algoritmus írása egy adott specifikáció alapján. Adott egy algoritmus, adjuk meg a végrehajtása eredményét.
4. OOP/objektumorientált programozási módszerek a C++, Java és C# programozási nyelvekben: osztályok és objektumok; egy osztály tagjai és hozzáférési módosítók; konstruktorok és destruktorok.

I. Algoritmusok és programozás

1. Programozási tételek

A feladatok feladatosztályokba sorolhatók a jellegük szerint. E feladatosztályokhoz készítünk a teljes feladatosztályt megoldó algoritmusosztályt. Ezeket az algoritmusosztályokat *programozási tételek*nek nevezzük. Bebizonyítható, hogy a megoldások a feladat garantáltan helyes és optimális megoldásai.
A bemenet és a kimenet szerint négy csoportra oszthatók:

1. Sorozathoz érték rendelése (1 sorozat – 1 érték)
2. Sorozathoz sorozat rendelése (1 sorozat – 1 sorozat)
3. Sorozatokhoz sorozat rendelése (több sorozat – 1 sorozat)
4. Sorozatokhoz sorozatok rendelése (1 sorozat – több sorozat)

1.1. Sorozathoz érték rendelése

1.1.1. Sorozatszámítás

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

Összeg és szorzat

Egyetlen kimeneti adatot számítunk ki adott számú bemeneti adat feldolgozásának eredményeként. *Példa:* a bemeneti adatok összegét, illetve 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 elemnek. Feltételezzük, hogy a bemeneti adatok egész számok, amelyeknek a számossága N.

```
Algoritmus Összegszámítás(N, X, S):                    { Sajátos eset!!! }
                                { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: S }
  S ← 0
  Minden i = 1,N végezd el:                  { minden adatot fel kell dolgoznunk }
     S ← S + Xi
  vége(minden)
Vége(algoritmus)
```

```
Algoritmus Feldolgoz(N, X, S):                         { Általános eset!!! }
                                { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: S }
  S ← F0                        { kezdőérték: az elvégzendő műveletre nézve semleges elem }
  Minden i = 1,N végezd el:                  { minden adatot fel kell dolgoznunk }
    S ← f(S, Xi)                                          { f a művelet = funkció }
    vége(minden)
Vége(algoritmus)
```

1.1.2. Döntés

Adott az *N* elemű *X* sorozat és az elemein értelmezett *T* tulajdonság. Döntsük el, hogy van-e a sorozatban *T* tulajdonságú elem!

Elemzés

A sorozat elemei tetszőlegesek, egyetlen jellemzőt kell feltételeznünk róluk: bármely elemről el lehet dönteni, hogy rendelkezik-e az adott tulajdonsággal, vagy nem. A válasz egy üzenet, amelyet az alprogram kimeneti paramétere (logikai változó) értéke alapján ír ki a hívó programegység.

```
Algoritmus Döntés_1(N, X, talált):
        { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: ha az X sorozatban található }
                        { legalább egy T tulajdonságú elem, talált értéke igaz, különben hamis }
  i ← 1                                                          { kezdőérték az indexnek }
  talált ← hamis                                                 { kezdőérték }
  Amíg nem talált és (i ≤ N) végezd el:
    Ha nem T(Xi) akkor
      i ← i + 1
         { amíg nem találunk egy Xi-t, amely rendelkezik a T tulajdonsággal, haladunk előre }
    különben
      talált ← igaz
    vége(ha)
  vége(amíg)
Vége(algoritmus)
```

A fenti algoritmus megírható tömörebben is:

```
Algoritmus Döntés_2(N, X, talált):
        { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: ha az X sorozatban található }
  i ← 1           { legalább egy T tulajdonságú elem, talált értéke igaz, különben hamis }
  Amíg (i ≤ N) és nem T(Xi) végezd el:
    i ← i + 1
  vége(amíg)
  talált ← i ≤ N { kiértékelődik a relációs kifejezés, az érték átadódik a talált változónak }
Vége(algoritmus)
```

Egy másik megközelítésben: el kell döntenünk, hogy az adatok, teljességükben, rendelkeznek-e egy adott tulajdonsággal vagy sem. Másképp kifejezve: nem létezik egyetlen adat sem, amelyiknek ne lenne meg a kért tulajdonsága. 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 ← 1
  Amíg (i ≤ N) és T(Xi) végezd el:
     i ← i + 1
  vége(amíg)
  mind ← i > N
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: már tudjuk, hogy garantáltan létezik ilyen elem.)

```
Algoritmus Kiválasztás(N, X, hely):
                                        { Bemeneti adatok: az N elemű X sorozat. }
                          { Kimeneti adat: a legkisebb indexű T tulajdonságú elem sorszáma: hely }
  hely ← 1
  Amíg nem T(X_hely) végezd el:      { nem szükséges a hely ≤ N feltétel mivel a feladat }
    hely ← 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 van-e *T* tulajdonságú elem a sorozatban! Ha van, akkor adjunk meg egyet!

```
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 ← 0
  Amíg (hely = 0) és (i ≤ N) végezd el:
    Ha T(X_i) akkor
      hely ← i
    különben
      i ← 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 ← 1
  Amíg (i ≤ N) és nem T(X_i) végezd el:
    i ← i + 1
  vége(amíg)
  Ha i ≤ N akkor          { ha kiléptünk az Amíg-ból, mielőtt i nagyobbá vált volna N-nél, }
    hely ← i                        { ⇒ találtunk adott tulajdonságú elemet, különben nem }
  különben
    hely ← 0
  vége(ha)
Vége(algoritmus)
```

Ha a követelményben az áll, hogy minden olyan elemet keressünk meg, 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 sorozatban. Ilyenkor egy *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 számoljuk.

```
Algoritmus Megszámlálás(N, X, db):
  { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: db, a T tulajdonságú elemek száma }
  db ← 0
  Minden i = 1, N végezd el:
    Ha T(Xi) akkor
      db ← 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 (vagy sorszámát)!

Megoldás
A megoldásban minden adatot meg kell vizsgálnunk, ezért az algoritmus, egy *Minden* típusú struktúrával dolgozik. A *max* segédváltozó kezdőértéke a sorozat első eleme.

```
Algoritmus Maximumkiválasztás(N, X, max):
     { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: max, a legnagyobb elem értéke }
  max ← X1
  Minden i = 2, n végezd el:
    Ha max < Xi akkor
      max ← Xi
    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:

```
Algoritmus Maximum_helye(N, X, hely):
   { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: hely, a legnagyobb elem pozíciója }
  hely ← 1                                          { hely az első elem pozíciója }
  Minden i = 2, n végezd el:
    Ha Xhely < Xi akkor
      hely ← i                                      { a maximális elem 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 olyan algoritmust, amely csak egyszer járja be a sorozatot:

```
Algoritmus Minden_max_2(N, X, db, indexek):
```

```
        { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: a db elemű indexek sorozat }
  max ← X1
  db ← 1
  indexek1 ← 1
  Minden i = 2, n végezd el:
    Ha max < Xi akkor
      max ← Xi
      db ← 1
      indexekdb ← i
    különben
      Ha max = Xi akkor
        db ← db + 1
        indexekdb ← i
      vége(ha)
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

1.2. Sorozathoz sorozat rendelése

1.2.1. Másolás

Adott az *N* elemű *X* sorozat és az elemein értelmezett függvény (*f*). A bemenő sorozat minden elemére végrehajtjuk a függvényt, az eredményét pedig a kimenő 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:
    Yi ← f(Xi)
  vége(minden)
Vége(algoritmus)
```

1.2.2. Kiválogatás

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

Elemzés

Az elvárások függvényében több megközelítés érvényes:

a) k
   iválogatás kigyűjtéssel
b) k
   iválogatás kiírással
c) h
   elyben
d) k
   ihúzással

a) Kiválogatás kigyűjtéssel

A keresett elemeket (vagy sorszámaikat) 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 ← 0
  Minden i = 1, N végezd el:
    Ha T(Xi) akkor
      db ← db + 1
      pozíciókdb ← i          { pozíciókdb-ben tároljuk Xi helyét, ha T tulajdonsága van }
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

b) Kiválogatás kiírással

Ha nincs szükség számolásra, és a feladat „megelégszik” a *T* tulajdonságú elemek kiírásával:

```
Algoritmus Kiválogatás_b(N, X, db):
                                        { Bemeneti adatok: az N elemű X sorozat }
  db ← 0
  Minden i = 1, N végezd el:
    Ha T(Xi) akkor
      Ki: Xi
    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 sorrendben 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 ← 1
  Amíg i ≤ N végezd el:        { nem lehet Minden-t alkalmazni, mert változik az N!!! }
    Ha nem T(Xi) akkor                  { a T tulajdonságú elemeket tartjuk meg }
      Xi ← XN                           { Xi-t felülírjuk XN-nel }
      N ← N − 1                         { rövidül a sorozat }
    különben
      i ← i + 1                         { i csak a különben ágon nő }
        vége(ha)
    vége(amíg)
Vége(algoritmus)
```

c2. Kiválogatás segédsorozattal

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_c2(N, X, törölt):
          { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az N elemű törölt sorozat }
```

```
  Minden i = 1, N végezd el:
    törölt_i ← hamis
  vége(minden)
  Minden i = 1, N végezd el:
    Ha nem T(X_i) akkor                    { a T tulajdonságú elemeket tartjuk meg }
      törölt_i ← igaz
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

c3. Kiválogatás helyben, megőrizve az eredeti sorrendet

Ha az eredeti sorozatra nincs többé szükség, de szeretnénk megőrizni az elemek eredeti sorrendjét, akkor a *T* tulajdonságú elemeket felsorakoztatjuk a sorozat elejétől elkezdve. 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 ennek az „új” sorozatnak a számosságát tartja nyilván:

```
Algoritmus Kiválogatás_c3(N, X, db):
                { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: a db elemű X sorozat }
  db ← 0
  Minden i = 1, N végezd el:
    Ha T(X_i) akkor
      db ← db + 1
      X_db ← X_i
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

d) Kiválogatás speciális értékkel

Egy másik megoldás, amely nem hoz létre új helyen, új sorozatot, hanem 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_d(N, X, 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 ← speciális érték
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

1.3. Sorozatokhoz sorozat rendelése

1.3.1. Halmazok

Mielőtt egy – halmazokat tartalmazó sorozatra vonatkozó műveletet alkalmaznánk, szükséges meggyőződnünk afelől, 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? A döntés eredményét az *ok* kimeneti paraméter tartalmazza.

```
Algoritmus Halmaz_e(N, X, ok):
    { Bemeneti adatok: az N elemű X sorozat. Kimeneti adatok: az ok értéke igaz, ha a sorozat }
  i ← 1                                                    { halmaz, különben hamis }
  ok ← igaz
  Amíg ok és (i < N) végezd el:
    j ← i + 1
    Amíg (j ≤ N) és (Xi ≠ Xj) végezd el:
        j ← j + 1
    vége(amíg)
    ok ← j > N                          { ha véget ért a sorozat, nincs két azonos elem }
    i ← i + 1
  vége(amíg)
Vége(algoritmus)
```

b) Halmazzá alakítás

Alakítsuk halmazzá az *N* elemű *X* sorozatot!

Elemzés

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

```
Algoritmus Halmaz_2(N, X):
  { Bemeneti adatok: az N elemű X sorozat. Kimeneti adatok: az új N elemű X sorozat (halmaz) }
  i ← 1
  Amíg i < N végezd el:
    j ← i + 1
    Amíg (j ≤ N) és (Xi ≠ Xj) végezd el:
      j ← j + 1
    vége(amíg)
      Ha j ≤ N akkor                                  { találtunk egy Xj = Xi-t }
        Xj ← XN                                  { felülírjuk a sorozat N. elemével }
        N ← N − 1                                    { rövidítjük a sorozatot }
      különben
        i ← i + 1                                                { haladunk tovább }
      vége(ha)
  vége(amíg)
Vége(algoritmus)
```

1.3.2. Keresztmetszet

Hozzuk létre azt a sorozatot, amely a bemenetként kapott sorozatok keresztmetszetét tartalmazza.

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 rendezett sorozatok.

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 ← 0                  { Kimeneti adatok: a db elemű Z sorozat, X és Y keresztmetszete }
  Minden i = 1, N végezd el:
    j ← 1
    Amíg (j ≤ M) és (Xi ≠ Yj) végezd el:
      j ← j + 1
    vége(amíg)
    Ha j ≤ M akkor
      db ← db + 1
      Zdb ← Xi
    vége(ha)
  vége(minden)
Vége(algoritmus)                                                          Április
```

1.3.3. Egyesítés (Unió)

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

Elemzés

Az egyesítés algoritmusa 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*-ból azokat az elemeket, amelyeket nem találtunk meg *X*-ben.

```
Algoritmus Egyesítés(X, Y, Z, M, N, db):
  Z ← X                            { Bemeneti adatok: az N elemű X és az M elemű Y sorozat. }
  db ← N                          { Kimeneti adatok: a db elemű Z sorozat (X és Y egyesítése) }
  Minden j = 1, M végezd el:
    i ← 1
    Amíg (i ≤ N) és (Xi ≠ Yj) végezd el:
      i ← i + 1
    vége(amíg)
    Ha i > N akkor
      db ← db + 1
      Zdb ← Yj
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

1.3.4. Összefésülés

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

Elemzés

Az *Egyesítés* és a *Keresztmetszet* algoritmusok négyzetes bonyolultságúak, mivel a halmazokat implementáló sorozatok nem rendezettek. Ez a két művelet megvalósítható lineáris algoritmussal, ha a sorozatok rendezettek. Természetesen 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 érünk. A másik sorozatban megmaradt elemeket átmásoljuk az eredménysorozatba. 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. }
                                  { Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel) }
  db ← 0                                                  { A sorozatok nem halmazok }
  i ← 1
  j ← 1
  Amíg (i ≤ N) és (j ≤ M) végezd el:
    db ← db + 1
    Ha X_i < Y_j akkor
      Z_db ← X_i
      i ← i + 1
    különben
      Z_db ← Y_j
      j ← j + 1
    vége(ha)
  vége(amíg)
  Amíg i ≤ N végezd el:                                   { ha maradt még elem X-ben }
    db ← db + 1
    Z_db ← X_i
    i ← i + 1
  vége(amíg)
  Amíg j ≤ M végezd el:                                   { ha maradt még elem Y-ban }
    db ← db + 1
    Z_db ← Y_j
    j ← 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 eredménysorozatba.

```
Algoritmus Összefésülés_2(N, X, M, Y, db, Z):
  db ← 0                          { Bemeneti adatok: az N elemű X és az M elemű Y sorozat. }
  i ← 1                           { Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel) }
  j ← 1                                                   { a sorozatok halmazok }

  Amíg (i ≤ N) és (j ≤ M) végezd el:
    db ← db + 1
    Ha X_i < Y_j akkor
      Z_db ← X_i
      i ← i + 1
    különben
```

```
      Ha Xi = Yj akkor
        Zdb ← Xi
        i ← i + 1
        j ← j + 1
      különben
        Zdb ← Yj
        j ← j + 1
      vége(ha)
    vége(ha)
  vége(amíg)
  Amíg i ≤ N végezd el:                    { ha maradt még elem X-ben }
    db ← db + 1
    Zdb ← Xi
    i ← i + 1
  vége(amíg)
  Amíg j ≤ m végezd el:                    { ha maradt még elem Y-ban }
    db ← db + 1
    Zdb ← Yj
    j ← j + 1
  vége(amíg)
Vége(algoritmus)
```

Ha szerencsések lettünk volna $X_N = Y_M$. Ekkor a két utolsó *Amíg* struktúrát nem hajtotta volna végre a program egyetlen egyszer sem. Kihasználjuk 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 eredménysorozat hossza pontosan $N + M$. Az algoritmus ismétlő struktúrája *Minden* típusú lesz.

```
Algoritmus Összefésül_3(N, X, M, Y, db, Z):
  i ← 1                    { Bemeneti adatok: az N elemű X és az M elemű Y sorozat. }
  j ← 1                    { Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel) }
  XN+1 ← YM + 1                                  { A sorozatok nem halmazok }
  YM+1 ← XN + 1


  Minden db = 1, N + M végezd el:
    Ha Xi < Yj akkor
      Zdb ← Xi
      i ← i + 1
    különben
      Zdb ← Yj
      j ← 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: *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 őrszemek révén az Amíg struktúrát addig hajtjuk végre, amíg mindkét sorozat végére nem értünk.

```
Algoritmus Összefésül_4(N, X, M, Y, db, Z):
  db ← 0                          { Bemeneti adatok: az N elemű X és az M elemű Y sorozat. }
  i ← 1                           { Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel) }
  j ← 1                                                          { A sorozatok halmazok }
  X_{N+1} ← Y_M + 1
  Y_{M+1} ← X_N + 1
  Amíg (i < N + 1) vagy (j < M + 1) végezd el:
    db ← db + 1                                          { figyelem! itt vagy (nem és ) }
    Ha X_i < Y_j akkor
      Z_db ← X_i
      i ← i + 1
    különben
      Ha X_i = Y_j akkor
        Z_db ← X_i
        i ← i + 1
        j ← j + 1
      különben
        Z_db ← Y_j
        j ← j + 1
      vége(ha)
    vége(ha)
  vége(amíg)
Vége(algoritmus)
```

1.4. Sorozathoz sorozatok rendelése

1.4.1. Szétválogatás

Adott *N* elemű *X* sorozatot válogassuk szét 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 sorozatba

Az adott sorozatból létrehozunk két újat: a 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 ← 0                                  { Bemeneti adatok: az N elemű X sorozat. }
  dbz ← 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 ← dby + 1          { az adott tulajdonságú elemek, az Y sorozatba kerülnek }
```

```
      Y_dby ← X_i
    különben
      dbz ← dbz + 1                    { azok, amelyek nem rendelkeznek az }
      Z_dbz ← X_i                      { adott tulajdonsággal, a Z sorozatba kerülnek }
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

b) szétválogatás egy 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 kell *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 ← 0   { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az N elemű Y sorozat, }
  dbz ← 0       { ahol 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 ← dby + 1                                  { az első helytől kezdődően }
      Y_dby ← X_i
    különben
      dbz ← dbz + 1 { a többi elem ugyancsak az Y-ba kerül, az utolsó helytől kezdődően }
      Y_N-dbz+1 ← 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 elölről keresünk egy nem *T* tulajdonságú elemet, s ha találunk, azt hátratesszük. 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, ahol }
                    { az első e elem T tulajdonságú, n – e elem pedig nem T tulajdonságú }
  e ← 1                          { balról jobbra haladva az első T tulajdonságú elem indexe }
  u ← 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 ← u – 1
    vége(amíg)
    Ha e < u akkor
      X_e ← X_u
      e ← e + 1
    Amíg (e < u) és T(X_e) végezd el:
```

```
      e ← e + 1
    vége(amíg)
    Ha e < u akkor
        Xu ← Xe
        u ← u - 1
      vége(ha)
    vége(ha)
  vége(amíg)
  Xe ← segéd                    { visszahozzuk a segéd-be tett elemet }
  Ha T(Xe) akkor
    db ← e
  különben
    db ← 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.

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.

2. Lépések finomítása

Bonyolultabb feladatok esetében a megfelelő algoritmus leírása nem könnyű feladat. Ezért célszerű először a megoldást körvonalazni, és csak azután 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*, amíg az algoritmus hatékonysága megfelelő lesz. Végül elmaradhatatlanul következnie kell a helyesség ellenőrzésének, és a programkészítés (kódolás) után a tesztelés.
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. 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 lépésében. Ezután kiszámítjuk *r*-ben *a* egészosztási maradékát *b*-vel. 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*-t úgy kapjuk meg, hogy a két szám szorzatát osztjuk az *lnko*-val. Mivel az eredeti két szám értékét az algoritmus „tönkreteszi", szükséges elmenteni ezeket két segédváltozóba (*x* és *y*) ahhoz, hogy felhasználhassuk ezeket az *lkkt* kiszámításakor.

```
Algoritmus Eukleidész_1(a, b, lnko, lkkt):
                                    { Bemeneti adatok: a, b. Kimeneti adatok: lnko, lkkt }
  x ← a                             { szükségünk lesz a értékére az lkkt kiszámításakor }
  y ← b                             { szükségünk lesz b értékére az lkkt kiszámításakor }
  r ← maradék[a/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 ← maradék[a/b]                       { kiszámítjuk az aktuális maradékot }
  vége(amíg)
  lnko ← b                                { lnko egyenlő az utolsó osztó értékével }
  vége(ha)
  lkkt ← [x*y/lnko]                                { felhasználjuk a és b másolatait }
Vége(algoritmus)
```

Megvalósíthatjuk az algoritmust 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 ← a                          { Bemeneti adatok: a, b. Kimeneti adatok: lnko, lkkt }
  y ← b
  Amíg a ≠ b végezd el:
    Ha a > b akkor
      a ← a - b
    különben
      b ← b - a
    vége(ha)
  vége(amíg)
  lnko ← a
  lkkt ← [x*y/lnko]
Vége(algoritmus)
```

2. Prímszámok

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

```
Algoritmus Prím(n, válasz):
                                 { Bemeneti adat: n. Kimeneti adat: válasz }
  @ Megállapítjuk, hogy n prím-e
  Ha n prím akkor
    válasz ← igaz
  különben
    válasz ← hamis
  vége(ha)
Vége(algoritmus)
```

Lépések finomítása: Ki kell dolgoznunk annak a módját, hogy megállapíthassuk, hogy a szám prím-e. A megoldás első változatában a prímszám definíciójából indulunk ki: egy szám akkor prím, ha pontosan két osztója van: 1 és maga a szám. Első ötletünk tehát az, hogy az algoritmus számolja meg az adott 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):
                                 { Bemeneti adat: n. Kimeneti adat: válasz }
  osztók_száma ← 0
  Minden osztó = 1,n végezd el:
    Ha maradék[n/osztó] = 0 akkor
      osztók_száma ← osztók_száma + 1
    vége(ha)
  vége(minden)
  válasz ← osztók_száma = 2}
Vége(algoritmus)
```

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*. Észrevesszük, 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. Mivel a páros számok mind oszthatók 2-vel, és a 2 kivételével nem prímek, „megszabadulunk” a páros számok fölösleges vizsgálatától, és a páratlan számokat csak páratlan osztókkal próbáljuk osztani. 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 ← hamis
  különben
    Ha n páros akkor
      prím ← n = 2
    különben
      prím ← igaz
      osztó ← 3
      négyzetgyök ← [√n]
      Amíg prím és (osztó ≤ négyzetgyök) végezd el:
        Ha maradék[n/osztó] = 0 akkor
          prím ← hamis
        különben
          osztó ← osztó + 2
        vége(ha)
      vége(amíg)
    vége(ha)
  vége(ha)
  válasz ← prím
Vége(algoritmus)
```

Ha ebben az algoritmusban felhasználjuk a matematikából ismert tulajdonságot, éspedig azt, hogy minden 5-nél nagyobb prímszám $6k \pm 1$ alakú, akkor a vizsgálandó számok száma tovább csökkenthető.

2.1. 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.

2.1.1. 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 az alrendszereket 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.

2.1.2. Moduláris kompozíció

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

2.1.3. 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!

2.1.4. 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 zártnak nevezünk, ha más modulok számára egy jól definiált felületen keresztül elérhető, a többi modul ezt változatlan formában felhasználhatja. Egy modult nyitottnak nevezünk, ha még kiterjeszthető, ha az általa nyújtott szolgáltatások bővíthetők vagy, ha hozzávehetünk további mezőket a benne levő adatszerkezetekhez, 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.
Adatstruktúrák és algoritmusok változatossága
Egy típus, egy modul: Egy típus műveletei kerüljenek egy modulba.
Reprezentáció-függetlenség

3. Rendezési algoritmusok

Legyen egy n elemű a sorozat. Rendezett sorozatnak nevezzük a bemeneti sorozat olyan permutációját, amelyben $a_1 \le a_2 \le ... \le a_n$.

3.1. Buborékrendezés (Bubble-sort)

A rendezés során páronként összehasonlítjuk a számokat (az elsőt a másodikkal, a másodikat a harmadikkal és így tovább), és ha a sorrend nem megfelelő, akkor az illető két elemet felcseréljük. Ha volt csere, a vizsgálatot újrakezdjük. 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 ↔ a_i+1
        k ← i
      vége(ha)
    vége(minden)
  ameddig rendben
Vége(algoritmus)
```

3.2. Egyszerű felcseréléses rendezés

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

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

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 Minimumkiválasztás(n, a):
  Minden i = 1,n-1 végezd el:{ Bemeneti adatok: n, a; Kimeneti adat: a rendezett a sorozat }
    ind_min ← i
    Minden j = i+1,n végezd el:
      Ha a_ind_min > a_j akkor
        ind_min ← j
      vége(ha)
    vége(minden)
    a_i ↔ a_ind_min
  vége(minden)
Vége(algoritmus)
```

3.4. Beszúró rendezés

A beszúró rendezés hatékony algoritmus kisszámú elem rendezésére. Úgy dolgozik, ahogy bridzsezés közben a kezünkben lévő 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 lévő 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 Beszúró_Rendezés(n, a):
  Minden j = 2,n végezd el:                          { Bemeneti adatok: n, a }
                                                     { Kimeneti adat: a rendezett a sorozat }
    segéd ← a_j                          { beszúrjuk a_j-t az a_1, ..., a_j–1 rendezett sorozatba }
    i ← j - 1
    Amíg (i > 0) és (a_i > segéd) végezd el:
      a_i+1 ← a_i
      i ← i - 1
    vége(amíg)
    a_i+1 ← segéd
  vége(minden)
Vége(algoritmus)
```

3.5. Leszámláló rendezés (ládarendezés, binsort)

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. Éppen ezért, ezeket az algoritmusokat *összehasonlító rendezés*eknek nevezzük.

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

A segédtömb *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 segédtömb elemeinek értékei alapján.

```
Algoritmus Ládarendezés(a, n):
  Minden i = 1,k végezd el:              { Bemeneti adatok: n, a; Kimeneti adat: a }
    segéd_i ← 0
  vége(minden)
  Minden j = 1,n végezd el:
    segéd_a_j ← segéd_a_j + 1
  vége(minden)
  q ← 0
  Minden i = 1,k végezd el:                          { a segéd tömbnek k eleme van }
    Minden j = 1,segéd_i végezd el:
      q ← q + 1                                  { a segéd_i elemek összege n }
      a_q ← i                                    { tehát a feldolgozások száma n }
    vége(minden)
  vége(minden)
Vége(algoritmus)
```

4. Rekurzió

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.

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:
$$n! = \begin{cases} 1, & \text{ha } n = 0 \\ n \cdot (n-1)!, & \text{ha } n \in \mathrm{N}^* \end{cases}$$
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, éspedig olyan függvényeket, illetve eljárásokat nevezünk rekurzívaknak, melyek meghívják önmagukat. Ha ez a hívás az illető alprogram összetett utasításában benne foglaltatik, 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. Tehát, a rekurzió fogalmát kifejezhetjük úgy is, hogy egy alprogram akkor rekurzív, ha meghívja önmagát, amikor még aktív.

Egy rekurzív alprogram végrehajtása azonos módon történik, mint bármely nem rekurzív alprogramé. A rekurzív eljárások esetében is, hasonlóan a nem rekurzívakhoz, az aktiválás feltételezi 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 túlcsordulás 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; az előző megjegyzést figyelembe véve, 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;

egy rekurzív függvény összetett utasítása, hasonlóan a nem rekurzív függvényekhez, tartalmazni fog egy értékadó utasítást, amely a függvény azonosítójának ad értéket. Ebbe az utasításba kerül, többnyire, az újrahívás.

4.2. Megoldott feladatok

4.2.1. Egy szó betűinek megfordítása

Olvassunk be egymás után több betűt a szóközkarakter megjelenéséig, majd írjuk ki ezeket a betűket fordított sorrendben!

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

```
Algoritmus Fordít:
  Be: betű              { nincs paraméter, mivel az alprogramban olvasunk be és írunk ki }
  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 mi esetünkben: Ki: betű), így természetes módja a feladat megoldásának.

4.2.2. Szavak sorrendjének megfordítása

Olvassunk be *n* szót, majd írjuk ki ezeket a beolvasás fordított sorrendjében! Ne használjunk tömböt!

```
Algoritmus Szavakat_fordít_1(n):
  Be: szó                        { az első hívás aktuális paramétere n = szavak száma }
  Ha n > 1 akkor
    Szavakat_fordít_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 Szavakat_fordít_2(i):
  Be: szó                                  { most az első hívás aktuális paramétere 1 }
  Ha i < n akkor
    Szavakat_fordít_2(i+1)
  különben
    Ki: 'Fordított sorrendben: '
  vége(ha)
  Ki: szó
Vége(algoritmus)
```

4.2.3. Faktoriális

Írjuk ki az *n* adott szám faktoriálisát!

Megoldás

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

```
Algoritmus Fakt(n):                                  { Bemeneti adat: n }
  Ha n = 0 akkor
    Fakt ← 1
  különben
    Fakt ← 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 értékként adjuk át; így kifejezéseket is írhatunk az aktuális paraméter helyére.
A faktoriálist nem előnyös rekurzívan számolni, mivel sokkal időigényesebb, mint az iteratív megoldás, hiszen a *Fakt*(*n*) függvény (*n*+1)-szer fog aktiválódni.

4.2.4. Legnagyobb közös osztó

Számítsuk ki két természetes szám ($n, m \in$ N*) legnagyobb közös osztóját rekurzívan.
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*, maradék[*m*/*n*])-val. Tehát fel lehet írni a következő rekurzív definíciót:

$$Lnko(m,n) = \begin{cases} n, & \text{ha } maradék[m/n] = 0 \\ Lnko\big(n,\ maradék[m/n]\big), & \text{ha } maradék[m/n] \neq 0 \end{cases}$$

```
Algoritmus Lnko(m,n):                                { Bemeneti adatok:m, n }
  mar ← maradék[m/n]
  Ha mar = 0 akkor
    Lnko ← n
  különben
    Lnko ← Lnko(n,mar)
  vége(ha)
Vége(algoritmus)
```

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

4.2.5. Descartes-szorzat

Egy rajzon *n* virágot fogunk 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 a következő Descartes-szorzatot kell generálnunk: $M^n = \underbrace{M \times M \times ... \times M}_{n-szer}$, ha M = {1, 2, ..., *m*}.

```
Algoritmus Descartes_szorzat(i):          { Bemeneti adat: i, az első híváskor = 1 }
  Minden j=1,m végezd el:
    x_i ← j
```

```
    Ha i < n akkor
      Descartes_szorzat_3(i+1)
    különben
      Kiír
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

4.2.6. *k* elemű részhalmazok

Adva vannak az n és a k ($1 \le k \le n$) egész számok. Generáljuk rekurzívan az $\{1, 2, ..., n\}$ halmaz minden k elemet tartalmazó részhalmazát!

Megoldás

Az $\{1, 2, ..., n\}$ halmaz k elemet tartalmazó részhalmaza egy k elemű tömb segítségével kódolható: $x_1, x_2, ..., x_k$. 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). Ha az x sorozatba az elemek szigorúan növekvő sorrendben kerülnek ($x_1 < x_2 < ... < x_k$), egy részhalmazt csak egyszer állíthatunk elő.

Mivel minden x_i szigorúan nagyobb, mint x_{i-1}, az értékei $x_{i-1} + 1$-től nőnek, $n - (k - i)$-ig.

```
Algoritmus Részhalmazok(i):
                                { M = (1, 2, ..., n), x globális ⇒ x_i = 0, i = 0,20, k is globális }
  Minden j=x_{i-1}+1,n-k+i végezd el:
    x_i ← j
    Ha i < k akkor
      Részhalmazok(i+1)
    különben
      Kiír
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

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

4.2.7. Fibonacci-sorozat

Generáljuk a Fibonacci-sorozat első n elemét!

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

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
    Fibo ← 0
  különben
    Ha n = 2 akkor
      Fibo ← 1
    különben
      Fibo ← 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 újból, meg újból 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 ← f_n-1 + f_n-2
  különben
    f_1 ← 0
    Ha n = 1 akkor
      f_1 ← 0
    különben
      f_2 ← 1
    vége(ha)
  vége(ha)
Vége(algoritmus)
```

Megjegyzések:

1. A rekurzió hasznos, ha:
   - a feladat eredménye rekurzív szerkezetű,
   - a feladat szövege rekurzív szerkezetű,
   - a megoldás legjobb módszere a visszalépéses keresés (backtracking) módszer,
   - a megoldás legjobb módszere az oszd meg és uralkodj (divide et impera) módszer,
   - a feldolgozandó adatok rekurzívan definiáltak (pl. bináris fák).
2. A rekurzív programozás legszembeötlőbb előnye az, hogy az algoritmus tömör és világos, természetszerűen tükrözi a folyamatot, amit modellez.
3. Ugyanakkor, nem tanácsos rekurziót alkalmazni, ha a feladatot megoldhatjuk egy egyszerűbb iteratív algoritmussal (mivel így a program gyorsabb lesz). Egy iteratív algoritmusnak az ellenőrzése is könnyebb. Tehát, nem alkalmazunk rekurzív algoritmust például a Fibonacci-sorozat elemeinek generálására, kombinációk generálására, illetve más kombinatorikai feladatok megoldására.

4.2.8. Az {1, 2, ..., *n*} halmaz minden részhalmaza

Generáljuk az {1, 2, ..., *n*} halmaz minden részhalmazát!

Elemzés

A halmazokat ebben az esetben is egy $x_1 < x_2 < ... < x_i$ sorozattal ábrázoljuk, ahol $i = 1, 2, ..., n$.

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 az úgynevezett lexikográfikus sorrendben generáljuk. Figyelemre méltó, hogy minden új elem generálása egy új részhalmazhoz vezet.

```
Algoritmus Részhalmaz(i)
  Minden j=x_{i-1}+1,n végezd el:
    x_i ← j
    Kiír(i)
    Részhalmaz(i+1)
  vége(minden)
Vége(algoritmus)
```

A kilépési feltétel ($x_i = n$) el van rejtve a *Minden* típusú struktúrába: ha $x_i = n$, a ciklusváltozó kezdőértéke $x_i + 1$ nagyobb, mint a végső érték, így a *Minden* ciklusmagja nem lesz többet végrehajtva és a program kilép az aktuális hívásból. Az algoritmust *Részhalmazok*(1) alakban hívjuk meg.

4.2.9. Partíciók

Generáljuk az $n \in$N* szám partícióit!

Megoldás

Partíció alatt azt a felbontást értjük, amelynek során az $n \in$ N* számot pozitív számok összegeként írjuk fel: $n = p_1 + p_2 + ... + p_k$, ahol $p_i \in$ N*, $i = 1, 2, ..., k$, $k = 1, ..., n$. Két partíciót kétféleképpen tekinthetünk különbözőnek:

- Két partíció különbözik egymástól, ha vagy az előforduló értékek vagy az előfordulásuk sorrendje különbözik (erre adunk megoldást).
- Két partíció különbözik egymástól, ha az előforduló értékek különböznek (lásd a 8. kitűzött feladatot a fejezet végén).

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 ← j
    Ha j < n akkor
      Partíció(i+1, n-j)
    különben
      Kiír(i)
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

5. A visszalépéses keresés módszere (backtracking)

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 feladatot csak exponenciális algoritmussal tudunk megoldani, alkalmazzuk a backtracking (visszalépéses keresés) módszert, amely exponenciális ugyan, de megpróbálja csökkenteni a generálandó próbálkozások számát.

5.1. A visszalépéses keresés általános bemutatása

A visszalépéses keresés azon feladatok megoldásakor alkalmazható, amelyeknek eredményét az $M_1 \times M_2 \times ... \times M_n$ Descartes-szorzatnak azon elemei alkotják, amelyek eleget tesznek bizonyos belső feltételeknek. Az $M_1 \times M_2 \times ... \times M_n$ 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 $x = (x_1, x_2, ..., x_n) \in M_1 \times M_2 \times ... \times M_n$ elemét, hanem csak azokat, amelyek 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 $x_1, x_2, ..., x_{i-1}$ már kaptak végleges értéket az aktuálisan generált eredményben. Az x_i-re vonatkozó „javaslat”-ot akkor fogadjuk el, amikor $x_1, x_2, ..., x_{i-1}$ 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 amelyek esetében a folytatási feltételek nem kérnek további elemeket, végeredményeknek nevezzük.

A belső feltételek és a folytatási feltételek között szoros kapcsolat áll fenn. Ezek kifejezésmódjának szerencsés megválasztása többnyire a számítások csökkentéséhez vezethet.

A belső feltételeket egy külön algoritmusban vizsgáljuk. Legyen ennek a neve *Megfelel*, és paramétere az aktuálisan generált elem i indexe. Ez az alprogram *igaz* értéket térít vissza, ha az x_i elem az eddig generált $x_1, x_2, ..., x_{i-1}$ elemekkel együtt megfelel a belső feltételeknek, és hamis értéket ellenkező esetben.

```
Algoritmus Megfelel(i):
  Megfelel ← igaz
  Ha a belső feltételek x1, x2, ..., xi esetében nem teljesülnek akkor
    Megfelel ← hamis
  vége(ha)
```

```
Vége(algoritmus)
```

Az eredményt a következő algoritmussal generáljuk:

```
Algoritmus Rekurzív_Backtracking(i):
  Minden mj ∈ Mi értékre végezd el:
    xi ← mj
    Ha Megfelel(i) akkor        { megvalósulnak a belső feltételek x1, x2, ..., xi esetében }
      Ha i < n akkor
        Rekurzív_Backtracking(i+1)
      különben
        Ki: x1, x2, ..., xn
      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 lépésnél végrehajtja az algoritmus.

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

Összefoglalva a lényeget, a következő lépéseket kell elvégeznünk:
- *az eredmény kódolása* – meg kell állapítanunk az x_i elemek jelentését az illető feladat esetében, valamint meg kell határoznunk az M_i, $i = 1, 2, ..., n$ halmazokat.
- a belső, majd a folytatási feltételek megállapítása.
- a *Rekurzív_Backtracking*(i) vagy iteratív változatának átírása.

5.2. Megoldott feladatok

5.2.1. 8 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, $i = 1, 2, ..., 8$) kódoljuk. A tömb K_i elemeinek értéke az oszlop sorszáma, ahova az i-edik királynőt tettük (az i-edik sorban).

A sakktáblának 8 oszlopa van, tehát $K_i \in \{1, 2, \ldots, 8\}$, $i = 1, \ldots, 8$.

Az eddigiekből következik, hogy egy eredmény az $\{1, 2, \ldots, 8\}^8$ Descartes-szorzat eleme. Tehát, ha meg akarjuk oldani a feladatot, tulajdonképpen az $\{1, 2, \ldots, 8\}^8$ Descartes-szorzat

egy részhalmazát kell meghatároznunk, azzal a feltétellel, hogy a 8 királynő, amelyek a K_1, K_2, ..., K_8 oszlopokban találhatók, ne támadja egymást. A kódolás sajátos módja biztosítja, hogy soronkénti támadási lehetőség nincs, hiszen minden királynő új sorba kerül. De például, ha az első két királynő egymást támadja, nem generálunk fölöslegesen $8^6 = 262144$ elemet a $\{1, 2, ..., 8\}^8$ Descartes-szorzatból.
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 elhelyeztük, van egy eredmény, amit ki kell írnunk. Az elhelyezést a *Királynő*(*i*) rekurzív alprogram végzi el, a támadási lehetőséget a *Nem_támad*(*i*) logikai függvény ellenőrzi.

Ahhoz, hogy két királynő ne támadja egymást, a következő relációknak kell teljesülniük: $K_i \neq K_j$, $i - j \neq |K_i - K_j|$, $j = 1, 2, ..., i - 1$.

```
Algoritmus Nem_támad(i):
  Jó ← igaz                                   { Jó = lokális változó, K globális }
  j ← 1
  Amíg (j ≤ i-1) és Jó végezd el:
    Ha (Ki = Kj) vagy (i-j = |Ki - Kj|) akkor
      Jó ← hamis                              { az i. és j. királynők támadják egymást }
    különben
      j ← j + 1
    vége(ha)
  vége(amíg)
  Nem_támad ← Jó
Vége(algoritmus)

Algoritmus Királynő(i):
  Minden j=1,8 végezd el:
    Ki ← j                                    { az i-edik királynőt a j-edik oszlopba tesszük }
    Ha Nem_támad(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).

5.2.2. 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 eredeti sablontól, 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 $x_i \in \{1, 2, ..., m\}$, ahol m a gyermekek száma, ($i = 1, 2, ..., n$).
Belső feltételek: $x_i \neq x_j$, $i \neq j$, $i, j = 1, 2, ..., n$.

Folytatási feltételek: $x_i \neq x_j$, $i \neq j$, $j = 1, 2, ..., i - 1$.

Mivel x_i értéke egy index, a folytatási feltételek kifejezhetők egyszerűbben egyetlen feltétellel. Azt fogjuk kifejezni, hogy az i-edik székre csak olyan gyerek ülhet le, aki pillanatnyilag még áll. Fölhasználunk egy *még_áll* logikai tömböt, ahol $még\_áll_j$ igaz, ha a j-edik gyerek még nem ült le, és hamis ellenkező esetben.

Az $x_i \neq x_j, j = 1, 2, ..., i - 1$ feltételek a következőképpen alakulnak át: $még\_áll_{x_i} = igaz$.

A *még_áll* 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 ül le; a j-edik gyermek felállítása maga után vonja a megfelelő $még\_áll_j$ visszaállítását *igaz*-ra.

Ez a megoldás hatékonyabb, mint az, amelyet a sablon alapján készíthetnénk, hiszen kevesebb összehasonlítást fog végezni.

```
Algoritmus Variáció(i):
  Minden j=1,m végezd el:
    Ha még_áll_j akkor                          { a j-edik gyermek még áll }
      x_i ← j
      még_áll_j ← hamis                         { a j-edik gyermek az i-edik széken ül }
      Ha i < n akkor
        Variáció(i+1)
      különben
        Kiír
      vége(ha)
      még_áll_j ← igaz                          { a j-edik gyermek feláll }
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

5.2.3. Zárójelek

Írjunk ki minden helyesen nyitó és csukó n zárójelet tartalmazó karakterláncot!

Megoldás

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

Belső feltételek: Adott pillanatban ne létezzen több csukó zárójel, mint nyitó, de nyitó sem lehet több, mint $n/2$.

Folytatási feltételek: Amikor elhelyeztük az n-edik karaktert is, az eredmény kiírható.
Jelöljük *ny*-nyel és *cs*-vel a nyitó, illetve a csukó zárójelek számát. A folytatási feltételek különböznek az x_i elemek értékének függvényében:

$$\begin{cases} ny < \dfrac{n}{2} & \text{, ha } x_i = '(' \\ cs < ny & \text{, ha } x_i = ')' \end{cases} \quad \forall i = 2,3,\ldots,n$$

Mivel bármely eredményben x_1 = '(' és x_n = ')', a hívó programegységben elvégezzük az inicializálásokat: $x_1 \leftarrow$ '(' és $x_n \leftarrow$ ')'. Az első hívás alakja: *Zárójel*(2, 1, 0).

```
Algoritmus Zárójel(i, ny, cs):
  Ha i = n akkor Ki: x                                  { kilépési feltétel }
  különben
    Ha ny < [n/2] akkor
      xi ← '('
      Zárójel(i+1, ny+1, cs)
    vége(ha)
    Ha cs < ny akkor
      xi ← ')'
      Zárójel(i+1, ny, cs+1)
    vége(ha)
  vége(ha)
Vége(algoritmus)
```

5.2.4. Labirintus

Egy labirintust egy $L_{(n\times m)}$ kétdimenziós tömbben tárolunk, amelyben a folyosónak megfelelő elemek értéke 1; ezek az értékek egymás után következnek a labirintusnak megfelelő tömbben, egy bizonyos sorban, vagy oszlopban. Egy személyt ejtőernyővel leengednek 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 kéri. Egy utat az $x_1, x_2, \ldots, x_k$ és $y_1, y_2, \ldots, y_k$ sorozatokkal kódolunk, amelyek azokat a sorokat és oszlopokat tartalmazzák, amelyeknek érintésével kifele haladunk a labirintusból. $x_i \in \{1, 2, \ldots, n\}$, $y_i \in \{1, 2, \ldots, m\}$, $i = 1, 2, \ldots, k$.

Belső feltételek: Az útvonalra a következő belső feltételek érvényesek:

a) A folyosón kell haladnia: $L(x_i, y_i) = 1$, $i = 1, 2, \ldots, k$.

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

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: $(x_i, y_i) \neq (x_j, y_j)$, $j = 1,\ldots, i - 1$.

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

$$eredm_{ij} = \begin{cases} \text{az a lépésszám mellyel az } (i,j) \text{ helyre léptünk} \\ 0, \text{ ha az útvonal nem halad át az } (i,j) \text{ helyen} \end{cases}$$

Egy bizonyos helyről négy irányba léphetünk.

```
Algoritmus Út(i, j, lépés):
  Ha (Lij = 1) és (eredmij = 0) akkor
                    { próbálunk az (i, j) helyre lépni; ha (i, j) folyosó és még nem jártunk itt }
    eredmij ← lépés                                  { az (i, j) helyre lépünk }
    Ha (i ∈ {1,n}) vagy (j ∈ {1,m}) akkor
      Kiír                                { kijárathoz 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 ← 0          { töröljük az utolsó lépést, hogy új utat választhassunk }
  vége(ha)
Vége(algoritmus)
```

Ez a kód tartalmaz egy figyelemreméltó egyszerűsítést, ami a folytatási feltételeket illeti. Nem ellenőriztük azt, hogy kiléptünk-e a labirintusból, mivel a hívás előtt (a labirintus beolvasása után) az *L* tömböt körülvettük egy 0-ból álló kerettel. Így az algoritmus gyorsabbá válik. Az algoritmust a kiindulási hely koordinátáira (*i*, *j*) és 0 lépésszámra hívjuk meg.

Második megoldás

A következő változatban az *Út* 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.
Ezeknek a tömböknek az értékei: $x = (-1, 0, 1, 0)$, $y = (0, 1, 0, -1)$.

```
Algoritmus Út(i, j, lépés):
  Minden irány=1,4 végezd el:                    { kiválasztunk egy irányt }
    úji ← i + x_irány                          { (úji, újj) az új koordináták }
    újj ← j + y_irány
    Ha (úji ∈ {1, 2, ..., n}) és (újj ∈ {1, 2, ..., m}) akkor
      Ha (L_úji,újj = 1) és (eredm_úji,újj = 0) akkor
        eredm_úji,újj ← lépés                    { az (úji, újj) helyre lépünk }
        Ha (úji ∈ {1,n}) vagy (újj ∈ {1,m}) akkor
          Kiír                                 { kiléptünk a labirintus szélén }
        vége(ha)
        Út(úji, újj, lépés+1)
        eredm_úji,újj ← 0                      { lemondunk az utolsó lépésről }
      vége(ha)
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

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.

Az algoritmusban nem vettük körül a labirintust az előbbi változat megoldásában említett kerettel. Ennek következtében szükség volt ellenőrizni, hogy az új hely, ahova lépni akarunk a labirintuson belül van-e.

Az előbbi algoritmust az *Út*(*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, észrevesszük, hogy mivel az előrehaladás egy kétdimenziós tömbben történik, az alprogramnak mindig két paramétere lesz (*i*, *j*), amelyek annak a helynek a koordinátái, ahova lépni készülünk. Mivel általában az utat is meghatározzuk, szükséges a *lépés* paraméter is.

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

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

Megoldjuk a maximálisan leegyszerűsített feladatot.

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 legkézenfekvőbben rekurzívan írjuk le. A felbontás megtörténik a rekurzióba való belépéskor, a részeredmények összerakása pedig a kilépéskor.

6.1. Az oszd meg és uralkodj módszer általános bemutatása

A *DivImp*(*bal*, *jobb*, *eredm*) algoritmus az a_1, a_2, ..., 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 egyszerű }
                                  { kiszámítjuk az egyszerű feladat eredm eredményét }
    Megold(bal,jobb,eredm)
  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.

6.2. Megoldott feladatok

6.2.1. 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 részszorzatokra bontjuk. Ezt úgy valósítjuk meg, hogy 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, ..., x_n) részfeladat általános alakja: *Szorzat*(x_{bal}, ..., x_{jobb}). Minden részfeladat más-más szorzatot számol ki, tehát a feladatok függetlenek egymástól. Mivel előnyösebb, ha egy szorzatot egy függvénnyel számolunk ki és nem egy eljárással, a *DivImp*(*bal*, *jobb*, *eredm*) algoritmust a következőképp írjuk át:

```
Algoritmus Szorzat(bal ,jobb):                    { függvény típusú algoritmus }
  Ha jobb = bal akkor            { Bemeneti adatok: bal, jobb. Kimeneti adat: Szorzat }
    Szorzat ← x_bal                                 { a részsorozat egy elemből áll }
  különben
    Ha jobb - bal = 1 akkor
      Szorzat ← x_bal * x_jobb                      { a részsorozat két elemű }
    különben                                { felbontjuk a Szorzat(bal, ..., jobb) feladatot }
      közepe ← [(bal+jobb)/2]
      p1 ← Szorzat(bal, közepe)
      p2 ← Szorzat(közepe+1, jobb)
      Szorzat ← p1 * p2                             { összerakjuk a részeredményeket }
    vége(ha)
  vége(ha)
Vége(algoritmus)
```

6.2.2. Minimumszámolás

Állapítsuk meg *n* egész szám közül a legkisebbet!

```
Algoritmus Minimum(bal, jobb):                    { függvény típusú algoritmus }
                                  { Bemeneti adatok: bal, jobb. Kimeneti adat: Minimum }
  Ha jobb = bal akkor                             { a részsorozat egy elemből áll }
    Minimum ← x_bal
  különben
    Ha jobb - bal = 1 akkor                       { a részsorozat két elemből áll }
      Ha x_bal < x_jobb akkor
        Minimum ← x_bal
      különben
        Minimum ← x_jobb
      vége(ha)
    különben              { felbontjuk a Minimum(bal, jobb) feladatot részfeladatokra: }
      közepe ← [(bal+jobb)/2]
      min1 ← Minimum(bal, közepe)
      min2 ← Minimum(közepe+1, jobb)
      Ha min1 < min2 akkor
        Minimum ← min1                              { összerakjuk a részeredményeket }
      különben
        Minimum ← min2
      vége(ha)
    vége(ha)
  vége(ha)
Vége(algoritmus)
```

6.2.3. 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 sorszámnak megfelelő paraméter értéke legyen 0.

Megoldás

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

1. *keresett* = $x_{közép}$ ⇒ keresett a sorban a *közép* helyen található;
2. *keresett* < $x_{közép}$ ⇒ mivel a sorozat rendezett, a keresett számot a sorozat első (x_1, ..., $x_{közép-1}$) felében keressük tovább;
3. *keresett* > $x_{közép}$ ⇒ a keresett számot a sorozat második ($x_{közép+1}$, ..., 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}, ..., $x_{közép-1}$ sorozatban, vagy az $x_{közép+1}$, ..., x_{jobb} sorozatban. Itt nincs szükség a divide et impera harmadik lépésére (a részeredmények összerakására).

```
Algoritmus Bin_keres(x, bal, jobb, keresett, közép):
                                  { Bemeneti adatok: x, bal, jobb, keresett. Kimeneti adat: közép }
  Ha bal > jobb akkor
    közép ← 0                                          { keresett nincs a sorozatban }
  különben
    közép ← [(bal+jobb)/2]
    Ha keresett < xközép akkor
      Bin_keres(x, bal, közép-1, keresett, közép)
    különben
      Ha keresett > xközép akkor
        Bin_keres(x, közép+1, jobb, keresett, közép)
      vége(ha)
    vége(ha)
  vége(ha)                                  { ha keresett = x_közép megvan a pozíció }
Vége(algoritmus)
```

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

```
Algoritmus Bin_Keres_Iteratív(n, x, keresett, közép):
  bal ← 1
  jobb ← n
  megvan ← hamis
  Amíg nem megvan és (bal ≤ jobb) végezd el:
    közép ← [(bal+jobb)/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(ha)
  vége(amíg)
```

```
  Ha nem megvan akkor
    közép ← 0                    { ha közép értéke 0 ⇒ keresett nem található }
  vége(ha)
Vége(algoritmus)
```

6.2.4. Összefésülésen alapuló rendezés (*MergeSort*)

Rendezzünk növekvő sorrendbe egy egész számokból álló sorozatot összefésüléssel! (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, ugyancsak rendezettet előállítanunk, hanem egyetlen sorozatot kell rendeznünk.)

Megoldás

Az adott sorozatot két részre osztjuk, abból a célból, hogy rendezhessük. De ezeket újból felosztjuk, amíg a kapott tömb, amelyet rendezni 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=közép+1,jobb végezd el:
    b_i ← x_i
  vége(minden)
  a_közép+1 ← végtelen
  b_jobb+1 ← végtelen                                   { strázsák }
  i ← bal
  j ← közép + 1
  Minden k=bal,jobb végezd el:
    Ha a_i < b_j akkor
      x_k ← a_i
      i ← i + 1
    különben
      x_k ← b_j
      j ← j + 1
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

```
Algoritmus Rendez(bal, jobb):
  Ha bal < jobb akkor
    közép ← [(bal+jobb)/2]
    Rendez(bal, közép)
    Rendez(közép+1, jobb)
    Összefésül(bal, közép, jobb)
  vége(ha)
Vége(algoritmus)
```

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

6.2.5. Gyorsrendezés (QuickSort)

Fölhasználva a *quiksort* algoritmust, rendezzünk növekvő sorrendbe n egész számot! A gyorsrendezés az oszd meg és uralkodj módszeren alapszik, mivel az eredeti sorozatot úgy rendezi, hogy két rendezendő részsorozatra bontja.

Megoldás

A részsorozatok rendezése egymástól függetlenül történik. A részeredmények összerakása ebből az algoritmusból is hiányzik (mint a bináris keresésből).

Amikor az $x_1, \ldots, x_n$ sorozatot készülünk rendezni, előbb előkészítünk két részsorozatot ($x_1, \ldots, x_{m-1}$ és $x_{m+1}, \ldots, x_n$) úgy, hogy az $x_1, \ldots, x_{m-1}$ részsorozat elemei kisebbek legyenek, mint az $x_{m+1}, \ldots, x_n$ részsorozat elemei. Közöttük található az x_m, amely nagyobb mint az $x_1, \ldots, x_{m-1}$ részsorozat bármely eleme, és kisebb mint az $x_{m+1}, \ldots, x_n$ részsorozat összes eleme.

Azt az elemet, amely meghatározza a helyet, ahol az adott tömb két részre oszlik, strázsának (őrszem) nevezzük. Ennek a helynek a meghatározása kulcskérdés az algoritmus végrehajtása során. A strázsa m helyét úgy határozzuk meg, hogy az $x_1, \ldots, x_m$ tömbben legyenek azok az elemek, amelyek kisebbek, mint a strázsa és az $x_{m+1}, \ldots, x_n$ 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 strázsa (és a tömb első részében találhatók) azokkal, amelyek kisebbek, mint a strázsa (és a tömb második részében találhatók). 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 egy elem (x_m) a végleges helyére kerül }
    m ← Strázsa_helye(bal, jobb)
    QuickSort(bal, m)              { hasonlóan járunk el az (x_bal, ..., x_m) részsorozattal }
    QuickSort(m+1, jobb)           { valamint az (x_m+1, ..., x_jobb) }
  vége(ha)
Vége(algoritmus)
```

Látható, hogy a rekurzív hívásoknak megfelelően, az algoritmus meghívja önmagát egy bal meg egy jobb részsorozat rendezése érdekében. De hol a rendezés, hiszen ez az algoritmus nem tartalmaz összehasonlításokat és felcseréléseket? Ezeket aközben végezzük, miközben keressük a strázsa m helyét:

```
Algoritmus Strázsa_helye(bal, jobb):
  strázsa ← x_bal          { Bemeneti adatok: bal, jobb. Kimeneti adat: Strázsa_helye }
  i ← bal-1
  j ← jobb+1                       { megkeressük azt a j-t, amelyre bal ≤ j < jobb }
  Ismételd
    Ismételd                { megkeressük azt a j-t (jobbról balra), amelyre x_j < strázsa }
      j ← j - 1
    ameddig x_j ≤ strázsa
    Ismételd                { megkeressük azt az i-t (balról jobbra), amelyre x_i > strázsa }
      i ← i + 1
    ameddig x_i ≥ strázsa
    Ha i < j akkor
```

```
      xi ↔ xj                    { felcseréljük ezt a két nem megfelelő tulajdonságú elemet }
    vége(ha)
  ameddig i ≥ j                  { addig folytatjuk a keresést és felcserélést, amíg i kisebb, mint j }
  Strázsa_helye ← j              { megtaláltuk az új strázsa helyét }
Vége(algoritmus)
```

Megjegyzés
Ez az algoritmus főleg abban az esetben gyors, amikor a tömb elemei nem rendezettek!

6.2.6. 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.

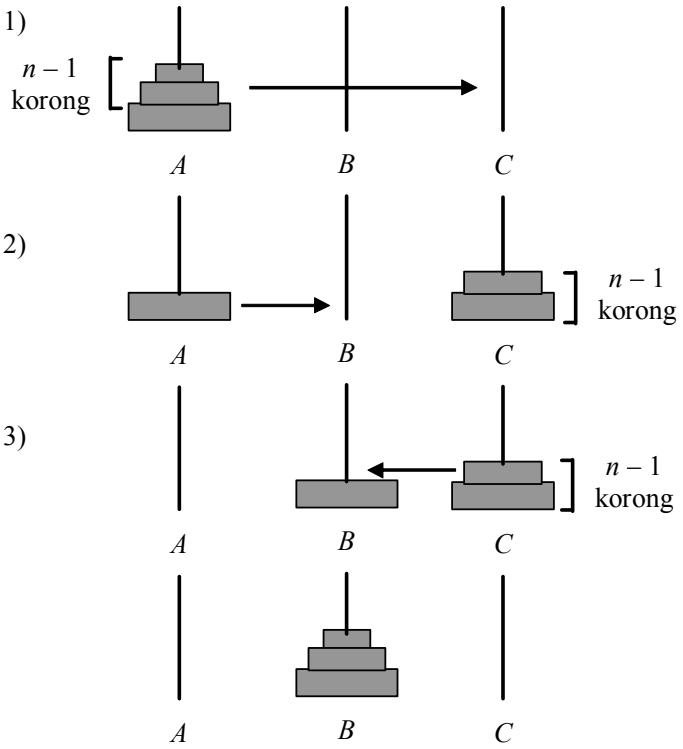
Megoldás

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

1) *n* – 1 korong átköltöztetése az A rúdról a C-re,

2) a megmaradt korong áthelyezése B-re,

3) *n* – 1 korong áthelyezése C-ről B-re.



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 ideiglenesnek 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
    Hanoi_1(n-1,A,C,B)
    Tedd a korongot A-ról B-re
```

```
    Hanoi_1(n-1,C,B,A)
  vége(ha)
Vége(algoritmus)
```

Ennek az algoritmusnak a hívása Hanoi(*n*, A, B, C) alakú, ahol A, B, C a három rudat jelképezi, és ha a hívó programegységben az aktuális paraméterek értékei 'A', 'B', 'C', akkor a *Tedd a korongot A-ról B-re* egy egyszerű kiírás: **Ki:** `A, '-', B`.

7. Mohó algoritmusok (greedy módszer)

A greedy módszert (mohó algoritmusokat) optimum-számításokra használjuk. E feladatok eredményei részhalmazai vagy elemei annak a Descartes-szorzatnak, amelyre a célfüggvény eléri minimumát vagy maximumát.

A mohó algoritmus mindig egyetlen eredményt határoz meg. Ezt az eredményt fokozatosan építjük fel: a feladatokban általában adott egy L halmaz, amelynek meg kell határoznunk egy M részhalmazát, amely megfelel bizonyos követelményeknek (T tulajdonságnak), és amely általában a végeredmény. Az M halmaz eredetileg az üres halmaz. Ehhez, egymás után hozzáadunk L-beli elemeket, amelyeket úgy választunk ki, hogy lokális optimumot biztosítanak. Ezek az elemek azok, amelyek a legtöbbet ígérők az aktuális lépésben, és amelyek megfelelnek a feladatnak az adott pillanatban.

A stratégia mohó jellegének következtében kapta ez az algoritmus a greedy (mohó) elnevezést. Mivel a stratégia egy helyi optimum kiválasztására épül, nem biztosítja a megoldás globális optimalitását, tehát nem mindig határozza meg a legjobb megoldást. Nem lehetünk biztosak a megoldásban, de ha sikerül *bebizonyítani*, hogy az adott feladat esetében a mohó algoritmus optimumot határoz meg, akkor biztonságosan alkalmazható. Ha viszont olyan feladatunk van, amelynek pontos megoldását csak exponenciális algoritmussal tudjuk megadni, sok esetben akkor is alkalmazható, de természetesen számításba vesszük, hogy az eredmény közelítő érték. Ilyenkor *heurisztikus* mohó algoritmusról beszélünk.

Legyen az L halmaz az $\{a_1, a_2, ..., a_n\}$ sorozat és T egy tulajdonság, amelyet az L részhalmazaira definiáltunk: T: $T(L) \rightarrow \{0, 1\}$, ahol $T(\varnothing) = 1$ (igaz, vagyis teljesül T), ha $T(X)$, akkor $\Rightarrow$ $T(Y)$, bármely $Y \subset X$ részhalmaz esetében. Egy $S \subset L$ részhalmazt eredménynek nevezünk, ha $T(S) = 1$. Minden lehetséges eredményből azt szeretnénk kiválasztani, amely optimalizálja a T: $T(L) \rightarrow R$ adott függvényt. A mohó algoritmus nem generál minden lehetséges részhalmazt (ami exponenciális végrehajtási időhöz vezetne), hanem megpróbál közvetlenül az optimális megoldás felé haladni.

A módszer egyszerű, a programok gyorsak, még nagyméretű adatszerkezetek esetében is. Az egyszerűség abban áll, hogy minden pillanatban, csak az adott kontextusnak megfelelő részfeladatot tekintjük. A módszer különbözik a backtracking (visszalépéses keresés) módszertől mivel, ha egy elemről kiderül, hogy hiába volt sokat ígérő, akkor nem kerül be a megoldásba és soha nem térünk vissza ehhez az elemhez. Fordítva, ha egy elem bekerült egy adott pillanatban egy megoldásba, nem fogjuk kivenni onnan.

7.1. A mohó algoritmus általános bemutatása

A módszer általános alakját két változatban ismertetjük. (A feladat megoldását az M halmaz tartalmazza, a megoldásokat az L – lehetséges megoldások halmazából – válogatjuk):

```
Algoritmus Greedy_1(L,M):
  M ← ∅
  Amíg M nem megoldás és L ≠ ∅ végezd el:
    Választ(L,x)                          { kiválasztjuk a legtöbbet ígérő elemet L-ből }
    L ← L\{x}                             { töröljük a legtöbbet ígérő elemet L-ből }
    Ha T(M ∪ {x}) = 1 akkor               { ha lehetséges }
      M ← M ∪ {x}                         { ezt hozzáadjuk M-hez }
    vége(ha)
  vége(amíg)
```

```
Vége(algoritmus)
```

Megjegyzések

1. Ha a kiválasztott elemet töröljük *L*-ből, akkor biztosítottuk az algoritmus számára, hogy *L* minden elemét csak egyszer dolgozzuk fel (töröljük, függetlenül attól, hogy betesszük az eredménybe vagy sem).
2. Mivel a bemeneti adatoktól függően nem mindig találunk eredményt, a hívó programegységben meg kell vizsgálnunk, hogy az *M* halmaz valóban eredmény-e:

Az ilyen típusú feladatok megoldása során gyakran bizonyul előnyösnek, ha a tulajdonképpeni feldolgozás előtt előbb rendezzük a feldolgozandó adatokat (az *L* halmazt). A rendezett sorozat elemeit ($\{a_1, a_2, ..., a_n\}$) egymás után vizsgáljuk és a követelményektől függően betesszük az eredménybe vagy sem (nincs szükség ezek törlésére *L*-ből, mivel egy megvizsgált elemhez nem térünk vissza). Az algoritmus ebben a változatban a következő lesz:

```
Algoritmus Greedy_2(n,a,M):
  Feldolgoz(n,a)                                  { ez a feldolgozás gyakran rendezés }
  M ← ∅
  i ← 1
  Amíg M nem megoldás és (i ≤ n) végezd el:
    Ha T(M ∪ {a_i}) akkor                                  { ha lehetséges }
      M ← M ∪ {a_i}                                  { a_i-t hozzáadjuk M-hez }
    vége(ha)
    i ← i + 1
  vége(amíg)
Vége(algoritmus)
```

A fenti algoritmusok lineárisak (eltekintve a *Választ*(*L*, *x*) és a *Feldolgoz*(*n*, *a*) algoritmusok bonyolultságától)! A tulajdonképpeni nehézséget a *Választ*(*L*, *x*), valamint a *Feldolgoz*(*n*, *a*) jelenti, mivel ezekbe „rejtjük” el a célfüggvényt.

7.2. Megoldott feladatok

7.2.1. Összeg

Adott egy *n* elemű, valós számokból álló sorozat. Határozzuk meg az adott sorozat azon részsorozatát, amelynek összege a lehető legnagyobb.

Megoldás

Alkalmazzuk a *Greedy*_1(*L*, *n*) algoritmust, ahol a *Választ*(*L*, *x*) alprogramnak megfelelően az adott sorozatból kiválasztjuk a szigorúan pozitív elemeket. Ezúttal könnyű belátni, hogy az algoritmus garantáltan maximális összegű részsorozatot határoz meg, hiszen, ha az összeghez hozzáadnánk egy negatív értéket, akkor az kisebbé válna. Ha egy 0 értékű elemet adunk az összeghez, az nem változik. Ebből az észrevételből következik, hogy, ha a sorozat tartalmaz 0 értékeket is, akkor több megoldás is létezik.

```
Algoritmus Összeg(n, a, k, Pozitívak):
  k ← 0                          { Bemeneti adatok: n, a. Kimeneti adatok: k, Pozitívak }
  Minden i=1,n végezd el:
    Ha a_i > 0 akkor
      k ← k + 1
      Pozitívak_k ← a_i
```

```
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

7.2.2. Az átlagos várakozási idő minimalizálása

Egy ügyvédi irodába egyszerre érkezik n személy, akiknek az intéznivalóit az ügyvéd ismeri, és így azt is tudja, hogy egy-egy személlyel hány percet fog eltölteni. Állapítsuk meg azt a sorrendet, amelyben fogadnia kellene a személyeket ahhoz, hogy az átlagos várakozási idő minimális legyen.

Megoldás

Az átlagos várakozási idő az n személy várakozási idejének számtani középarányosa, tehát az átlagos várakozási idő csökkentése a várakozási idők összegének csökkentését jelenti.

A minimális várakozási időösszeget a személyekkel való tárgyalási idők növekvő sorrendben való rendezése eredményezi.

Dacára annak, hogy ez természetesnek tűnik, be kell bizonyítanunk, hogy a mohó algoritmus jó megoldási módszer (…).

A mohó algoritmus alkalmazása optimális eredményt biztosít. Ahhoz, hogy minimalizáljuk az átlagos várakozási időt, minimalizálnunk kell a várakozási idők összegét. Egy személy addig várakozik, amíg az összes előtte fogadott személlyel tárgyal az ügyvéd. Ha csak két személy érkezett volna az irodába, akkor az lenne előnyösebb (az átlagos várakozási idő szempontjából), ha előbb a kevesebb időt igénylő személlyel tárgyalna az ügyvéd. Az eredmény tehát a személyek sorszámainak egy olyan permutációja, amelynek megfelelően az ügyvéd minden lépésben a legkevesebb időt igénylő személyt fogadja: $M = (k_1, k_2, ..., k_n) \in \{(x_1, x_2, ..., x_n) \mid x_i \in \{1, 2, ..., n\}, x_i \neq x_j \; \forall \; i, j = 1, 2, ..., n, i \neq j\}$.

Az L eredetileg az $\{1, 2, ..., n\}$ halmaz. A legtöbbet ígérő x elem az L-ből annak a személynek a sorszáma, akinek a fogadási ideje minimális azok között, akik még az L-hez tartoznak. Ezt hozzáadjuk az M-hez és kizárjuk az L-ből. Az x kizárását az L-ből úgy valósítjuk meg, hogy 0 értéket másolunk rá. Minden lépésnél csak 0-tól különböző értéket választunk az L-ből.

A következő implementáció előbb inicializálja az M halmazt az 1, 2, ..., n értékekkel, és növekvő sorrendbe rendezi az időket, megfelelően módosítva az M halmaz elemeit. A rendezés után: $M = k_1, k_2, ..., kn$ és $t_1 \leq t_2 \leq ... \leq t_n$. A kiírást az M halmazban található indexpermutáció alapján végezzük.

```
Algoritmus Sorrend(n, t, M, átlag):
                                   { Bemeneti adatok: n, t, M. Kimeneti adatok: átlag, M }
  Minden i=1,n végezd el:
    M_i ← i
  vége(minden)              { növekvően rendezzük a t sorozatot és módosítjuk az M-et is }
  Növekvő_sorrendbe_rendezés(n,M,t)
  vi_min ← 0
  vi ← 0
  Minden i=1,n-1 végezd el:
    vi ← vi + t_i                          { a t sorozat már növekvően rendezett }
    vi_min ← vi_min + vi
  vége(minden)
```

```
  átlag ← vi_min / n
Vége(algoritmus)
```

7.2.3. Buszmegállók

Egy közszállítási vállalat olyan gyorsjáratot szeretne indítani, amely csak a város főutcáján közlekedne, és a már létező n megálló közül használna néhányat. Ezeket a megállókat úgy kell kiválasztanunk, hogy két megálló között a távolság legkevesebb x méter legyen (gyorsjáratról van szó), és a megállók száma legyen a lehető legnagyobb (minél több utas használhassa). Adott a főutcán már meglevő egymás után található megállók közti távolságok sorozata.

Megoldás

Az L halmazt a létező megállók sorszámai alkotják: $L = \{1, 2, \ldots, n\}$. Ismerjük az n megálló közötti $n - 1$ távolságot: $a_1, a_2, \ldots, a_{n-1}$.

Meg kell határoznunk azt a maximális elemszámú $M \subseteq L$ részhalmazt ($M = \{i_1, i_2, \ldots, i_k\}$), amelyben a sorszámok növekvő sorrendben követik egymást (a főutcán található megállóknak egymás utáni sorszámaik vannak), és amelynek megfelelően bármely két kiválasztott megálló között a távolság legkevesebb x méter ($a_{ij+1} - a_{ij} \geq x, j = 1, 2, \ldots, k - 1$).

```
Algoritmus Megállók(n,a,M):
  i ← 1                                  { Bemeneti adatok: n, a. Kimeneti adat: M }
  M1 ← 1                              { az eredménybe betett utolsó megállótól mért távolság }
  táv_az_utolsótól ← 0
  Minden j=2,n végezd el:
    Ha aj-1 + táv_az_utolsótól ≥ x akkor
      i ← i + 1
      Mi ← j
      táv_az_utolsótól ← 0
    különben
      táv_az_utolsótól ← táv_az_utolsótól + aj-1
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

Látható, hogy az első megállót betettük a megoldásba, majd megkerestük azt a megállót, amelyik megfelelő távol található az elsőtől. Ha találtunk ilyent, betettük a megoldásba. Ezt addig folytattuk, amíg bejártuk az összes, már létező megállót.

7.2.4. Autó bérbeadása

Egy szállítási vállalat autókat kölcsönöz. Egy bizonyos jármű iránt igen nagy az érdeklődés, ezért az igényeket egy évre előre jegyzik. Az igényt két számmal jelöljük, amelyek az év azon napjainak sorszámait jelölik, amellyel kezdődően, illetve végződően igénylik az illető autót.

Állapítsuk meg a bérbeadást úgy, hogy a lehető legtöbb személyt szolgáljuk ki. Adott a személyek száma n, ($n \leq 100$) és az igényelt intervallumok ($a_i, b_i, i = 1, 2, ..., n, a_i < b_i \leq 365$). Írjuk ki azt a számot, amely a lehetséges legnagyobb az igénylő személyek számából és a bérbeadási időintervallumokat.

A következő algoritmusban az L halmaz: $\{2, 3, ..., n\}$ mivel M kezdőértéke $\{1\}$ (az első igény – a minimális b_1 – mindig része lesz a megoldásnak, amelyet a greedy stratégia biztosít). Az L halmazt az algoritmus *Minden* típusú struktúrával számítja ki, amelyben sorra veszi a b_i szerint rendezett igényléseket.

```
Algoritmus Autó_kölcsönzés(n, a, b, max, M):
  Növekvő_sorrendbe_rendezés(n, a, b)
  M1 ← 1                          { Bemeneti adatok: n, a, b. Kimeneti adat: max, M }
  max ← 1
  Minden i=2,n végezd el:
    j ← Mmax
    Ha ai > bj akkor
      max ← max + 1
      Mmax ← i
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

7.2.5. Hátizsák

Egy tolvaj betört egy hentesüzletbe, ahol n áru közül válogat. Minden árunak ismeri a súlyát és az értékét. Mivel a hátizsákjába legtöbb *S* súly fér, szeretne úgy válogatni, hogy a nyeresége maximális legyen. Ha egy áru nem fér be egészében a hátizsákba, a tolvaj levághat belőle egy akkora darabot, amekkora befér a hátizsákba, de ebben az esetben az áru értéke a súlyával arányosan csökken.

Megoldás

A feladat a szakirodalomban „töredékes hátizsák” vagy „folytonos hátizsák” elnevezés alatt ismeretes.

Észrevehető, hogy mivel meg volt engedve, hogy levághatunk az árukból, a hátizsák teljesen megtölthető, és ha minden lépésben azt az árut választjuk, amelynek az *érték*/*súly* aránya maximális, akkor a hátizsákba csomagolt árumennyiség összértéke is maximális lesz.

Bevezetjük a következő jelöléseket: Az eredmény az $x = (x_1, \ldots, x_n)$ sorozat lesz, ahol $x_i \in [0, 1]$, $i = 1, 2, ..., n$ azt fejezi ki, hogy az i-edik árunak mekkora darabját csomagoljuk be. Ezen kívül: $súly_1 \cdot x_1 + súly_2 \cdot x_2 + ... + súly_n \cdot x_n \leq S$.

Az optimális eredmény az, amely maximalizálja az $f(x) = érték_1 \cdot x_1 + érték_2 \cdot x_2 + ... + érték_n \cdot x_n$ függvényt.

Abban a sajátos esetben, amikor minden árut be lehet csomagolni a hátizsákba, $x = (1, 1, ..., 1)$. Ezért a továbbiakban feltételezzük, hogy $súly_1 + ... + súly_n > S$.

A greedy stratégiának megfelelően, az árukat az *erték*/*súly* arány szerint csökkenő sorrendbe rendezzük. Az árukat ebben a sorrendben csomagoljuk a hátizsákba, amíg az meg nem telik. Ha egy áru nem fér a hátizsákba, levágunk belőle egy akkora darabot, ami befér.

```
Algoritmus Hátizsák(n, S, súly, érték, sorszám, x):
  Csökkenő(n, súly, érték, sorszám)
  Hely ← S                        { Hely a hátizsákban még szabad helyet jelöli }
  i ← 1
  Amíg (i ≤ n) és (Hely > 0) végezd el:
    Ha súlyi ≤ Hely akkor
      xi ← 1
      Hely ← Hely - súlyi
    különben
      xi ← Hely / súlyi
      Hely ← 0
      Minden j=i+1,n végezd el:
```

```
        xj ← 0
      vége(minden)
    vége(ha)
    i ← i + 1
  vége(amíg)
Vége(algoritmus)
```

Az algoritmus végrehajtásának eredménye az x sorozat: $x = (1, ..., 1, x_j, 0, ..., 0)$ ahol $x_j \in [0, 1)$. Ennek alapján kiírhatjuk a becsomagolt áruk sorszámait (vigyázzunk, hogy az eredeti sorszámokat írjuk ki) és a hátizsák tartalmának értékét.

Be kell bizonyítanunk, hogy az algoritmus optimális eredményt határoz meg.

1. Objektumorientált fogalmak

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.

1.1. 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:

1.2. 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 *x* 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 védettségről sem, és ez által jutunk el az *osztály* (§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 megtesszük 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 az 1.1. 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.

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.

Természetszerűen 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.

1.3. kódszöveg. A *Taxi* felhasználói típus.

```
#include <iostream>
using namespace std;
struct Taxi {
   int fizetni;
   int indulas_ar;
   int menet_ar;
   int varakozas_ar;
   bool van_utas;
   void Kezdes();
   bool Beul();
   int Kiszall();
   void Megy(int km);
   void All(int perc);
};
void Taxi::Kezdes()
{
   indulas_ar = 10;
   menet_ar = 10;
   varakozas_ar = 3;
   fizetni = 0;
   van_utas = false;
}
bool Taxi::Beul()
{
   if ( van_utas ) return false;
   van_utas = true;
   fizetni = indulas_ar;
   return true;
}
int Taxi::Kiszall()
{
   if ( !van_utas ) return 0;
   van_utas = false;
   return fizetni;
}
void Taxi::Megy(int km)
{
   if ( van_utas )
      fizetni += menet_ar * km;
}
void Taxi::All(int perc)
{
   if ( van_utas )
      fizetni += varakozas_ar * perc;
}
void main()
{
   Taxi t1, t2;
   t1.Kezdes();
   t2.Kezdes();
   t1.Beul();
   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 az 1.3. 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.

Az 1.3. 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.

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 az §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 pri-

vát 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) 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 *konstruktornak* 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 az §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.

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 (§1.1) esetén ismertetett 1.1. kódszöveget módosítjuk úgy, hogy osztályokra vonatkozzon, majd ezt követően vizsgáljuk a tagokra való hivatkozást.

1.4. 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 az 1.4. 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 az 1.4. 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 megadjuk a fenti esetben a *this* mutatót, ezt automatikusan elhelyezi a rendszer. Mégis, a *this* mutatót explicit módon is használhatjuk, ha erre szükség van.

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 keresztnevét tárolja.

1.5. kódszöveg. A *szemely.h* fejállomány.

```
#include <iostream>
using namespace std;
class szemely {
   char* cs_nev;
   char* sz_nev;
public:
   szemely();      //alapértelmezett konstruktor
   szemely(char* cs_n, char* sz_n);
   szemely(const szemely& sz);    // másoló konstruktor
   ~szemely();
   void kiir();
};
szemely::szemely() {
   cs_nev = new char[1];
   *cs_nev = 0;      // 0 és '\0' ugyanaz
   sz_nev = new char[1];
   *sz_nev = 0;
   cout << "Alapertelmezett konstruktor\n";
}
szemely::szemely(char* cs_n, char* sz_n)
{
   cs_nev = new char[strlen(cs_n)+1];
   sz_nev = new char[strlen(sz_n)+1];
   strcpy(cs_nev, cs_n);
   strcpy(sz_nev, sz_n);
   cout << "Hagyomanyos konstruktor\n";
}
szemely::szemely(const szemely& x)
{
   cs_nev = new char[strlen(x.cs_nev)+1];
   strcpy(cs_nev, x.cs_nev);
   sz_nev = new char[strlen(x.sz_nev)+1];
   strcpy(sz_nev, x.sz_nev);
   cout << "Masolo konstruktor\n";
}
szemely::~szemely() {
   cout << "Destruktor\n";
   delete[] cs_nev;
   delete[] sz_nev;
}
void szemely::kiir() {
   if ( strlen(cs_nev) > 0 )
      cout << cs_nev << ' ' << sz_nev << endl;
   else
      cout << "Nincs adat\n";
}
```

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 konstruktornak fontos szerepe van azoknak az objektumoknak a létrehozásában, amelyek nem rendelkeznek kezdeti értékeket megadó aktuális paraméterekkel. Pontosabban, ha egy osztálynak van alapértelmezett konstruktora, akkor létrehozható olyan objektum, amely nem tartalmaz inicializáló ak-

tuális 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 konstruktornak nevezhetjük.

A konstruktorokon kívül az 1.5. kódszöveg tartalmaz egy sajátos tagfüggvényt, a *destruktort*, melyet az objektumok megszűnésekor hív meg a rendszer.

Tekintsük az 1.5. kódszöveget felhasználó alábbi fő függvényt:

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

```
1  #include "szemely.h"
2  void main() {
3     szemely BF("Bolyai", "Farkas");
4     BF.kiir();
5     szemely *FGy = new szemely("Farkas","Gyula");
6     FGy->kiir();
7     szemely A; //alapértelmezett konstruktor
8     A.kiir();
9     szemely Gyula(*FGy); // masoló konstruktor
10    Gyula.kiir();
11    delete FGy;
12 }
```

Ennek a kódnak a kimenete a következő lesz:

```
Hagyomanyos konstruktor
Bolyai Farkas
Hagyomanyos konstruktor
Farkas Gyula
Alapertelmezett 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 hívta meg a rendszer, mivel a *new* operátor után az osztály nevet és, kerek zárójelben, az aktuális paraméterek listáját adtuk meg. Az *A* objektumot az alpértelmezett, 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 bitjeit rendre átmásolja a cél adattagba. A bitenkénti másolás általában akkor ad helyes eredményt, ha az osztálynak nincsen mutató típusú adattagja.

Például az 1.5. és 1.6. 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 szerettünk volna tenni, mivel így, ha az egyik objektum megszűnik, a másiknak is fel lesz szabadítva a memóriaterülete és fordítva. E helyett a másoló konstruktort terheltük túl, amely új memóriaterületet foglal le, és erre másolja a családnevet és keresztnevet.

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.

Az 1.6. 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 helyezzünk 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 folyatatást. 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 egyes 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 az 1.5. 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.

1.7. kódszöveg. Osztály típusú tagok.

```
1  #include "szemely2.h"
2  class hazaspar {
3     szemely ferj;
4     szemely feleseg;
5  public:
```

```
6       hazaspar() // alapértelmezett konstruktor
7       {
8       }
9       hazaspar(szemely& aferj, szemely& afeleseg);
10      hazaspar(char* cs_ferj, char* sz_ferj,
11         char* cs_feleseg, char* sz_feleseg):
12      ferj(cs_ferj, sz_ferj), feleseg(cs_feleseg, sz_feleseg)
13      {
14      }
15      void kiir();
16   };
17   inline hazaspar::hazaspar(szemely& aferj, szemely& afeleseg):
18   ferj(aferj), feleseg(afeleseg)
19   {
20   }
21   void hazaspar::kiir()
22   {
23      cout << "ferj:  ";
24      ferj.kiir();
25      cout << "feleseg:  ";
26      feleseg.kiir();
27   }
28   void main() {
29      szemely Ady("Ady","Endre");
30      szemely 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
```

Az 1.7. kódszöveg három konstruktorral rendelkezik. Az alapértelmezett konstruktor definíciója is az osztályon belülre 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.

1.6. A destruktor

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

A destruktor neve mindig a $\sim$ karakterrel kezdődik, és ez után az osztály neve következik. A konstruktorhoz hasonlóan a destruktor 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 destruktorok. Ez a hatókörtől függ. Egy globális objektum destruktora a *main* függvény végén az *exit* függvény részeként lesz végrehajtva. Ezért nem szabad az *exit* függvényt meghívni a destruktorban, mivel ez végtelen ciklust eredményezhet.

Egy helyi objektum destruktorát 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 destruktort 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 destruktor működését, amely minden esetben kiírja, hogy éppen mit végzett, azaz milyen konstruktort vagy destruktort hívott meg. A kiírást most a *printf* függvénnyel végezzük.

1.8. kódszöveg. A destruktor.

```
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 vezettünk 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. Az objektumorientált programozási módszer

2.1. Elméleti alapok

Az objektum adattagokat és tagfüggvényeket tartalmaz. Ha nem használunk barát függvényeket a védett tagok csak a tagfüggvényekben érhetők 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 öröklléssel 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. 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.

2.3. 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.

1.9. 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?

Az 1.9. 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 alaposzálybeli deklarációjában. Ebben az esetben az alaposztályt 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 alaposztálybeli deklarációban. Ebben az esetben a származtatott osztályban deklarált túlterhelt tagfüggvény is virtuális lesz. Ha egy függvényt virtuálisnak deklaráltunk az alaposztályban, 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 nevezőjé 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.

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

```
#include <iostream>
using namespace std;
class tort {
protected:
    int szamlalo;
    int nevezo;
public:
    tort(int szamlalo1 = 0, int nevezo1 = 1);
    /*virtual*/ tort szorzat(tort& r);
    tort& szoroz(tort& r);
    void kiir();
};
tort::tort(int szamlalo1, int nevezo1)
{
    szamlalo = szamlalo1;
    nevezo = nevezo1;
}
// két tört szorzatát számolja ki, de nem egyszerüsít
tort tort::szorzat(tort& r)
{
    return tort(szamlalo * r.szamlalo, nevezo * r.nevezo);
}
// az aktuális objektumot módosítja
tort& tort::szoroz(tort& q)
{
    *this = this->szorzat(q);
    return *this;
}
void tort::kiir()
{
    if ( nevezo )
       cout << szamlalo << " / " << nevezo;
    else
       cerr << "helytelen tort";
}
class tort_kiir:  public tort {
public:
    tort_kiir( int szamlalo1 = 0, int nevezo1 = 1 );
    tort szorzat( tort& r);
};
inline tort_kiir::tort_kiir(int szamlalo1, int nevezo1) :
tort(szamlalo1, nevezo1)
{
}
tort tort_kiir::szorzat(tort& q)
{
    tort r = tort(*this).szorzat(q);
    cout << "(";
    this->kiir();
    cout << ") * (";
    q.kiir();
    cout << ") = ";
    r.kiir();
    cout << endl;
    return r;
}
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 kapjuk, a *szorzat* tagfüggvényt virtuálisnak kell deklarálni, és ezt úgy tehetjuk meg, hogy az 1.10. kódszöveg 9. sorából eltávolítjuk a megjegyzés jelét. Ha ezt megtesszük, 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.

2.4. 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 tagfügvényeket felül kell írni a származtatott osztályban, ellenkező esetben az illető osztály is absztrakt lesz.

Tekintsük a következő példát

1.11. 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 deklaráltuk, 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.

2.5. 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 interfésznek 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 létrehozzuk.

1.12. 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 deklaráltunk. 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.

3. Javasolt feladatok

1. Írjunk programot a C++, Java, C# 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 C++, Java, C# 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.