Algoritmusok hatékonyságának növelése a bináris keresés elvének alkalmazásával

Mărirea eficienţei algoritmilor prin folosirea principiului căutării binare

Improving the performance of algorithms using the principles of binary search

dr. Ionescu Klára, „Babeş-Bolyai" Tudományegyetem
drd. Pătcaş Csaba, „Babeş-Bolyai" Tudományegyetem

Abstract

Amikor algoritmusaink hatékonyságát vizsgáljuk, két fontos dologra figyelünk: a végrehajtási idő nagyságára és a futtatáshoz szükséges memória méretére. Természetesen, egy algoritmus akkor lesz hatékonyabb egy másiknál, ha az előbbinek a végrehajtáshoz kevesebb időre és kevesebb memóriára van szüksége. Épp ezért, akkor amikor egy készülő algoritmust hatékonyabbá akarunk tenni, vagy a végrehajtási időre koncentrálunk, vagy a memóriára. Szerencsés esetben sikerülhet mindkét szempontból javítani az algoritmus tulajdonságain.

Ebben a cikkben bemutatunk néhány algoritmikai/programozási feladatot, amelyeknek megoldásához olyan algoritmusokat tervezünk, amelyeknek végrehajtási idejét az *Oszd meg és Uralkodj* módszerrel, pontosabban a bináris keresés ötletével javítjuk.

Atunci când analizăm performanţele unui algoritm, ne interesează două aspecte importante: timpul de execuţie şi dimensiunea spaţiului de memorie necesar în timpul executării. Bineînţeles, un algoritm va fi mai eficient decât un altul, dacă primul are nevoie de mai puţin timp pentru executare şi de un spaţiu de memorie mai mic. Din acest motiv, atunci când ne propunem să creştem eficienţa unui algoritm proiectat, ne vom concentra fie asupra timpului de execuţie, fie asupra dimensiunii spaţiului de memorie. Dacă avem noroc, vom putea îmbunătăţi ambele proprietăţi ale algoritmului.

În acest articol vom prezenta câteva probleme de algoritmică/programare în rezolvarea cărora vom proiecta algoritmi ale căror timp de executare le vom îmbunătăţi aplicând metoda *Divide et Impera*, mai exact, ideea căutării binare.

When analyzing the performances of our algorithms, we are interested in two important things: the time of execution and the dimension of the needed memory space. Obviously, an algorithm will be more efficient than another, if the first one has a shorter time of execution and it needs a smaller memory space. Thus, when we want to increase the performance of an algorithm, we focus either on the time of its execution or on the needed memory space. If we are lucky, maybe we succeed in improving both of these properties of our algorithm.

In this article we present a few algorithmic/programming task and we propose algorithms designed with the *Divide and Conquer* method, more exactly with the idea of the binary search, in order to obtain a better execution time.

1. A bináris keresés

Bizonyos feladattípusok esetében, a megoldásként javasolt *lineáris* algoritmus egy *logaritmikus* bonyolultságúval helyettesíthető, ha felhasználjuk a *bináris keresés* elvét. Mielőtt rátérnénk adott feladatok megoldását képező algoritmusok bemutatására, lássuk az *Oszd meg és Uralkodj* elvére alapuló bináris keresés klasszikus algoritmusát!

Feladat$_1$ – Keresés

Adva van egy n egész számból álló, növekvően rendezett sorozat. Állapítsuk meg egy adott szám helyét a sorozatban! Ha a keresett szám nem található meg a sorozatban, a helynek megfelelő változó értéke legyen 0*!*

Megoldás

Legyen a rendezett sorozat $x_1 < x_2 < ... < x_n$. Egy bizonyos értéket keresünk (*keresett*), amelynek a helye ismeretlen. Az *Oszd meg és Uralkodj* módszer alapötletére támaszkodva a sorozatot két részre osztjuk: az x_1, ..., $x_{közép-1}$ és az $x_{közép+1}$, ..., x_n részsorozatokra. Vegyük észre, hogy az $x_{közép}$ elem nem tartozik egyik részsorozathoz

sem. Ezt az elemet külön vizsgáljuk és a vizsgálat eredményétől függően az algoritmus befejeződik vagy az eredeti elképzeléshez hasonlóan folytatódik valamelyik részsorozatra. A következő esetek fordulhatnak elő:

1. *keresett* = $x_{közép}$ ⇒ *keresett* a sorozatban 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, a keresett elem megkeresése átalakul *egyetlen* feladattá: keressük az elemet *vagy* az x_{bal}, ..., $x_{közép-1}$ sorozatban, vagy az $x_{közép+1}$, ..., x_{jobb} sorozatban. Előbb bemutatjuk az algoritmus rekurzív változatát:

```
Algoritmus Bin_keres_Rekurzívan(bal,jobb,hely):
                        { Bemeneti paraméterek: bal, jobb – a vizsgált részsorozat első és utolsó indexe }
                            { Kimeneti paraméterek: hely – a keresett elem indexe az eredeti sorozatban }
                                        { az algoritmust az 1, n bemeneti paraméterértékekre hívjuk meg }
  Ha bal > jobb akkor
    hely ← 0                  { ha keresett nem található meg a sorozatban, a hely változó értéke 0 }
  különben
    közép ← ⌊(bal + jobb)/2⌋        { felosztás: kiszámítjuk a sorozat közepén található elem indexét }
    Ha keresett = x[közép] akkor
      hely ← közép                              { keresett a sorozatban a közép helyen található }
    különben
      Ha keresett < x[közép] akkor
        Bin_keres(bal,közép-1)                   { az x_bal, ..., x_közép-1 részsorozatban keresünk tovább }
      különben
        Bin_keres(közép+1,jobb)                  { az x_közép+1, ..., x_jobb részsorozatban keresünk tovább }
      vége(ha)
    vége(ha)
  vége(ha)
Vége(algoritmus)
```

E feladat esetében is létezik egy iteratív megoldás, amely a végrehajtás idejét és a szükséges memóriát tekintve is hatékonyabb, mivel nincs szükség veremre és veremmel kapcsolatos műveletekre.

```
Algoritmus Bin_Keres_Iteratívan(n,x,keresett,hely):
  bal ← 1
  jobb ← n
  hely ← 0
  Amíg (hely = 0) és (bal ≤ jobb) végezd el:
    közép ← ⌊(bal + jobb)/2⌋
    Ha x[közép] = keresett akkor
      hely ← közép
    különben
      Ha x[közép] > keresett akkor
        jobb ← közép - 1
      különben
        bal ← közép + 1
      vége(ha)
    vége(ha)
  vége(amíg)
Vége(algoritmus)
```

2. Algoritmusok bonyolultságának csökkentése a bináris keresés elvének alkalmazásával

A következőkben bemutatunk néhány feladatot, amelyeknek megoldásaiban felhasználjuk a bináris keresést. Nem olyan feladatokra gondolunk, amelyekben a keresés megjelenik mint explicit részfeladat, hanem olyanokra, amelyekben az eredmény egy olyan érték, amelynek értéktartományát ismerjük, ugyanakkor lehetséges a „találgatás” a bináris keresés algoritmusának alkalmazásával.

Feladat$_2$ – Összeg

Legyen egy n elemű ($3 \le n \le 100000$) *különböző természetes számokat tartalmazó sorozat és az S természetes szám. Válasszunk ki az adott sorozatból három elemet, amelyeknek az összege pontosan S!*

Megoldás

Észrevesszük, hogy részletösszegeket kell számítanunk, de pontosan három elem összegét hasonlítjuk *S*-sel.

A feladat megoldható három egymásba ágyazott **Minden** ciklussal is. Sajnos, az *n* értéke miatt, ez a megoldás nem biztos, hogy befér egy esetleges időhatárba.

Ha a megoldásba beépítjük a bináris keresést, a bonyolultság $O(n^2 \cdot \log n)$ lesz:

- Rendezzük az adott sorozatot.
- Két **Amíg** ciklussal kiválasztunk a sorozatból két elemet (legyen ezeknek indexe *i* és *j*).
- Megkeressük az $S - a_i - a_j$ értéket *a bináris keresés*t alkalmazva.
- A megoldást tovább javítjuk (például, ha a_i értéke meghaladja *S*-t, kilépünk az első **Amíg**-ból stb.).

```
Algoritmus Generál(n,a,S,i,j,k):
                    { Bemeneti paraméterek: n – a sorozat mérete, a – az adott sorozat, S - az adott összeg }
          { Kimeneti paraméterek: i,j,k – három index a sorozatban (a megfelelő indexű elemek összege S) }
  megvan ← hamis
  i ← 1
  Amíg not megvan és (i ≤ n) és (a_i < S) végezd el:
    j ← i + 1
    Amíg not megvan és (j ≤ n) és (a_i + a_j < S) végezd el:
      BinKeres(a,j+1,n,S-a_i-a_j,k)
                                  { ha S-a_i-a_j megtalálható a sorozatban, a j+1 indexű elem után, }
      Ha k ≠ 0 akkor                          { akkor k értéke egy index, különben k értéke 0 }
        megvan ← igaz
      vége(ha)
      j ← j + 1
    vége(amíg)
    i ← i + 1
  vége(amíg)
Vége(algoritmus)
```

Feladat$_3$ – Labirintus

Egy labirintust egy n × n méretű négyzetes tömbbel kódolunk. Az (*i*, *j*) *helyen levő* 1 *érték falat jelent, a* 0 *szabad helyet. Adottak még a kiindulási és az érkezési pontok koordinátái.*

Írjuk ki annak a legnagyobb területű négyzetnek a méretét, amely eltolható a kiindulási ponttól az érkezési pontig, tudva, hogy egy k oldalhosszúságú négyzet minden pillanatban k^2 helyet foglal el az eredeti tömbben és csak vízszintesen, vagy függőlegesen mozgatható úgy, hogy ne haladjon át falon.

Megoldás

Ha a négyzet oldalhossza 1 volna, a feladatot megoldhatnánk a *dinamikus programozás* módszerével, vagy egy egyszerű, szélességi bejáráshoz hasonló algoritmussal.

A nehézséget az okozza, hogy minden lépésben ki kell kerülnünk a falat. Jó lenne, ha egy *k* oldalhosszúságú négyzet eltolása előtt nem kellene megvizsgálnunk *k* szomszédos helyet, (mondjuk a jobb oldallal szomszédosakat, ha jobbra akarunk mozdulni). Kiszámíthatnánk előre, minden pozícióra a legnagyobb oldalhosszúságot, aminek megfelelő négyzetet az illető helyről el lehet tolni mind a négy irányban. Tehát, megpróbálunk eltolni egy 1, 2, 3 stb. oldalhosszú négyzetet, amíg a költöztetés lehetséges, majd kiválasztjuk a legnagyobb méretű négyzetet amit el lehetett tolni.

Ha figyelmesebben elemezzük a fenti gondokat, megállapíthatjuk, hogy ezek kiküszöbölhetők. Olyan algoritmust javasolunk, amely kevesebb memória használatával, ugyanakkor gyorsabban dolgozik. Észrevesszük, hogy egy adott pozícióból valamely irányba tolható négyzet mérete megkapható a kiindulási és a végső pozícióba elhelyezhető négyzetek méreteinek minimumából. Ezek a négyzet-méretek megkaphatók $O(n^2)$ időben, a dinamikus programozás módszerének segítségével.

Az algoritmus javítása abban áll, hogy felhasználjuk a *bináris keresés*t az *oldalhossz megkeresésére*. Észrevesszük, hogy a legnagyobb (elvben eltolható) négyzetnek a mérete *n*, a legkisebbé 1. Tehát az 1 és *n* közötti számok között meg kell találnunk azt a legnagyobbat, amelynek egy eltolható négyzet felel meg. Legelőbb egy *n*/2 oldalhosszú négyzetet próbálunk eltolni; ha ez sikerült, megpróbálunk egy $3 \cdot n/4$ oldalhosszút, egyébként egy *n*/4 oldalhosszút stb. Minden kipróbálandó oldalhossz esetében (összesen log *n* ilyen hosszúságunk lesz) megoldjuk a feladatot egy szélességi bejárással.

Ezáltal a feladatot megoldó algoritmus bonyolultsága $O(n^2 \cdot \log n)$.

Szükségünk lesz két darab egydimenziós segédtömbre, amelyeket konstanstömbökként fogunk használni: `dx=(-1,0,0,1)` és `dy=(0,-1,1,0)`. A tömb elemeit a négyzetek megkeresésekor a következő pozíció megadására használjuk. Egy adott helyről 4 irányba lehet menni, ennek megfelelően a két segédtömb indexeinek jelentése: 1 – fel, 2 – balra, 3 – jobbra, 4 – le.

Példa

```
10 2 1 6 5
1 1 1 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
1 1 0 0 1 1 1 1 1 1
1 1 0 0 0 0 0 1 1 1
1 1 0 0 0 0 0 1 1 1
1 1 1 1 0 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1
1 1 1 1 0 0 0 0 1 1
```

Eredmény

```
2
```

Algoritmus Épít_MaxNégyzet(n,a,MaxNégyzet):
{ *Bemeneti paraméterek:* `n` *a labirintus mérete,* `a` *a labirintusnak megfelelő kétdimenziós tömb* }
{ *Kimeneti paraméter:* `MaxNégyzet` *segédtömb, minden pozícióra tárolja a legnagyobb négyzet oldalhosszát,* }
{ *aminek bal felső sarka az adott pozícióba kerülhet* `MaxNégyzet` *méretei*: `[0..n+1,0..n+1]`, }
{ *ahol a* `0` *és* `n+1` *indexű sorok és oszlopok bekeretezik a labirintust ahhoz, hogy ne léphessünk ki belőle* }
{ *feltöltjük a* `MaxNégyzet` *tömböt* `0`*-val* }

```
Minden q=n,1,-1 végezd el:
  Minden w=n,1,-1 végezd el:
    Ha a_qw = 1 akkor
      MaxNégyzet_qw ← 0                                                    { fal }
    különben
      MaxNégyzet_qw ← 1 + min(MaxNégyzet_q+1,w, MaxNégyzet_q,w+1, MaxNégyzet_q+1,w+1)
    vége(ha)
  vége(minden)
vége(minden)
Vége(algoritmus)
```

Algoritmus BF(MaxNégyzet,méret): { *Függvény típusú algoritmus: szélességi bejárás (Breadth First)* }
{ *Bemeneti paraméter:* `MaxNégyzet`, `méret` *– az eltolásra javasolt segédtömb mérete* }
{ *Kimenet: igaz, ha el tudtunk jutni a végső pozícióba ezzel a mérettel* }

```
Ha MaxNégyzet_x1,y1 ≥ méret akkor
{ feltöltjük a volt segédtömböt a hamis értékkel }
                { volt – kétdimenziós segédtömb, amelyben nyilvántartjuk, hogy mely mezőket érintettük }
  volt_x1,y1 ← igaz
  sor.első ← 1                    { sor segédtömb, a szélességi bejáráshoz szükséges várakozási sor }
  sor.utolsó ← 1
  sor.v_1.x ← x1
  sor.v_1.y ← y1
  Amíg (sor.első ≤ sor.utolsó) és (nem volt_x2,y2) végezd el:
    AktX ← sor.v_első.x
    AktY ← sor.v_első.y
    Minden q=1,4 végezd el:
      Ha nem volt_AktX+dx_q,AktY+dy_q és (MaxNégyzet_AktX+dx_q,AktY+dy_q ≤ méret) akkor
        sor.utolsó ← sor.utolsó + 1
        sor.v_utolsó.x ← AktX + dx_q
        sor.v_utolsó.y ← AktY + dy_q
        volt_v_utolsó.x,v_utolsó.y ← igaz
      vége(ha)
    vége(minden)
    sor.első ← sor.első + 1
  vége(amíg)
```

```
    BF ← volt_x2,y2
  különben
    BF ← hamis
  vége(ha)
Vége(algoritmus)

Algoritmus Bináris_Keresés(n,MaxNégyzet,lent):
                { Bemeneti paraméter: a keresés után megoldjuk a feladatot a MaxNégyzet felhasználásával }
                                  { Kimeneti paraméter: lent – a megtalált eltolható négyzet oldalhossza }
  lent ← 1
  fent ← n
  Amíg lent < fent végezd el:
    közép ← [(lent+fent)/2]
    Ha BF(MaxNégyzet,közép) akkor
      lent ← közép
    különben
      fent ← közép - 1
    vége(ha)
  vége(amíg)
  Ha nem BF(MaxNégyzet,lent) akkor
    lent ← 0                                                              { nincs megoldás }
  vége(ha)
Vége(algoritmus)
```

Feladat$_4$ – Út

Adott egy irányítatlan gráf. Minden éléhez hozzárendelünk egy hosszúságot és egy veszélyességi faktort. Írjunk ki egy olyan utat, amely az 1-es csomópontot összeköti az n-edikkel és amely úthoz tartozó élek legnagyobb veszélyességi faktora a lehető legkisebb! Ha több ilyen út létezik, akkor a legrövidebbet kell megtalálnunk.

Példa
csomópontok száma = 4, élek száma = 5

él	hossz	veszélyességi faktor
[1, 2]	1	5
[1, 3]	2	5
[1, 4]	1	10
[2, 4]	1	5
[3, 4]	2	5

Eredmény:
Út hossza: 2
Út: 1–2–4

Megoldás

Van megoldás a dinamikus programozás módszerével. „Összerakjuk” a veszélyességi faktort a költséggel: olyan valós számokkal dolgozunk, amelyeknek egész része a veszélyességi faktor, törtrésze pedig a hosszúság. Ennek az algoritmusnak bonyolultsága $O(n \cdot maxVeszélyességiFaktor)$ lenne.

Ennek a feladatnak a megoldásában a *bináris keresés*t a veszélyességi faktor függvényében végezzük. Az éleket a veszélyességi faktor szerint rendezzük és a keresés közben minden kiválasztott értékre meghatározzuk a legrövidebb utat *Dijkstra* algoritmusával úgy, hogy csak a bináris kereséssel kiválasztott értéknél kisebb vagy egyenlő veszélyességi faktorral rendelkező éleket vesszük figyelembe.

Ennek a megoldásnak bonyolultsága $O(n^2 \cdot \log maxVeszélyességiFaktor)$, ami javítható *Dijkstra* algoritmusának hatékonyabb implementálásával.

Az alábbi algoritmus végrehajtása előtt feltöltjük a `hossz` és a `veszély` tömböket `0`-val, majd beolvassuk az éleket és a megfelelő hosszúságokat és veszélyességi faktorokat. Meghatározzuk a veszélyességi faktorok maximumát (`mVeszély`). A `végtelen` egy megfelelően nagy érték, amelyet *Dijkstra* algoritmusában használunk.

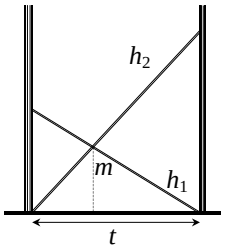
```
Algoritmus Dijkstra(n,m,hossz,veszély,mVeszély):                { Függvény típusú algoritmus }
                  { meghatározza, hogy megoldható-e a feladat a javasolt maximális veszélyességi faktorral }
                  { Kimenet: igaz, ha el tudtunk jutni a végső pozícióba a mVeszély veszélyességi faktorral }
              { Bemeneti paraméterek: n – a gráf csomópontjainak száma, m – a gráf csomópontjainak száma }
                  { hossz – az élek hosszának sorozata, veszély – a veszélyességi faktorok sorozata }
```

```
                { mVeszély – a maximális veszélyességi faktor, a beolvasással párhuzamosan határozzuk meg }
                                                { ez lesz a bináris keresés felső határa az első iterációban }
  { feltöltjük a volt segédtömböt a hamis értékkel }
  Minden q=1,n végezd el:
    Ha (hossz1q ≠ 0) { ha létezik él } és (veszély1q ≤ mVeszély) { csak ezeket vesszük figyelembe }
    akkor
      dq ← hossz1q
      előző_q ← 1                          { az előző tömb megőrzi az úton található csomópontokat }
    különben
      dq ← végtelen
    vége(ha)
  vége(minden)
  volt1 ← igaz
  Minden q=2,n végezd el:
    iMin ← 0
    Minden w=1,n végezd el:
      Ha nem voltw és ((iMin = 0) vagy (diMin > dw)) akkor
        iMin ← w
      vége(ha)
    vége(minden)
    Ha diMin = végtelen akkor              { lehet, hogy nem juthatunk el az összes csomópontba }
      Dijkstra ← dn ≠ végtelen                    { eljutottunk az n-edik csomópontba? }
      { ekkor kilépünk az algoritmusból, a függvény hamis értéket térít }
    vége(ha)
    voltiMin ← igaz
    Minden w=1,n végezd el:
      Ha nem voltw és (hossziMin,w ≠ 0) és (veszélyiMin,w ≤ mVeszély) és
                                                  (dw > diMin + hossziMin,w) akkor
        dw ← diMin + hossziMin,w
        előző_w ← iMin
      vége(ha)
    vége(minden)
  vége(minden)
  Dijkstra ← igaz
Vége(algoritmus)

Algoritmus Út_kiír(csúcs):
  Ha csúcs ≠ 1 akkor
    Út_kiír(előző_csúcs)
  vége(ha)
  Ki: csúcs,' '
Vége(algoritmus)

Algoritmus BinKeres:
  lent ← 1
  fent ← mVeszély
  Amíg lent < fent végezd el:
    m ← [(lent + fent)/2]
    Ha Dijkstra(m) akkor
      fent ← m
    különben
      lent ← m+1
    vége(ha)
  vége(amíg)
  Ha Dijkstra(lent) akkor
    Ki: 'Ut hossza:',dn
    Út_kiír(n)
  különben
    Ki: 'Nincs megoldás'
  vége(ha)
Vége(algoritmus)
```

Feladat$_5$ – Deszkák



Két függőleges fal egymástól t távolságra található. Egy h_1 hosszúságú deszkát az egyik fal alapjától a másik falnak támasztunk. Egy h_2 hosszúságú deszkát a másik fal alapjától az első falnak támasztunk. A két deszka m magasságban érinti egymást egy pontban, amely valahol a két fal között található. Számítsuk ki t-t h_1, h_2 és m ismeretében (megengedett hibalehetőség 10^{-5}).

Megoldás

Észrevesszük, hogy a magasság, ahol a két deszka találkozik nő, ha a keresett *t* távolság csökken, és csökken, ha *t* nő.

Átfogalmazzuk a követelményt: keressük meg azt a legnagyobb *t* értéket, amelyre a magasság, ahol a két deszka találkozik ne legyen kisebb mint *m*!

Felhasználjuk a bináris keresést a *t* értékének megkeresésére. A kiindulási érték: Min(h_1, h_2)/2.

Ha ismerjük a *t*, h_1 és h_2 értékeket, az érintkezési pont magasságát kiszámítjuk síkmértan ismeretekkel.

Ha a kiszámított magasság nagyobb mint az adott *m*, akkor növeljük *t*-t, különben csökkentjük.

A `Számol(h1,h2,t,sz)` algoritmus kiszámítja *sz*-ben a magaság értékét *t* aktuális közelítő értékére.

```
Algoritmus Számol(h1,h2,t,sz):
  x ← négyzetgyök(h1*h1-t*t)
  y ← négyzetgyök(h2*h2-t*t)
  sz ← (x*y)/(x+y)
Vége(algoritmus)

Algoritmus Deszkák(m,h1,h2,t):
  megvan ← hamis
  Amíg nem megvan végezd el:
    t ← (min + max)/2
    Számol(h1,h2,t,sz)
    Ha | sz - m | ≤ 0.0001 akkor
      megvan ← igaz
    különben
      Ha sz > m akkor
        min ← t
      különben
        max ← t
      vége(ha)
    vége(ha)
  vége(amíg)
Vége(algoritmus)
```

Feladat$_6$ – Ládák

Költözik a múzeum. A tárgyakat kocka alakú, különböző méretű ládákba csomagolták. Kicsomagoláskor több személy dolgozik egyidőben, és a rendetlenség elkerülése végett, azokba a helyiségekbe, ahol kicsomagolás folyik, felszereltek egy futószalagot, amelyre az üres ládákat helyezik, a nyitott fedelükkel fölfele. A futószalag végéhez egy robotot állítottak, amelynek az a feladata, hogy összeszedje a ládákat és úgy helyezze egyiket a másikba (ha lehet) hogy végül a ládacsomagok száma a lehető legkisebb legyen. A robotot egy program irányítja úgy, hogy:

- A ládákat az érkezésük sorrendjében szedi le a futószalagról.
- Az aktuális ládát csak egy nála nagyobb méretű ládába helyezi.
- Ha nincs olyan megkezdett csomag, amelybe elhelyezhető az aktuális láda, akkor ez a láda egy új csomag első ládája lesz.
- Egy megkezdett csomagba csak egyetlen ládát helyez, vagyis nem helyez két ládát egymás mellé, még akkor sem, ha ez egyébként lehetséges volna.
- Egy elhelyezett ládát, többé nem mozgat.
- Egy megkezdett csomagot nem helyez egy másik csomagba még akkor sem, ha ez egyébként lehetséges volna.
- Egyetlen ládát sem hagy figyelmen kívül.

Írjunk programot, amely a ládák számának ($0 \le n \le 15000$) és méreteiknek ($1 \le$ *láda_méret* ≤ 10000) ismeretében meghatározza a csomagok lehetséges legkisebb számát, valamint, minden csomag esetében az illető csomagban található ládákat.
Példa: $n = 10$, *méretek* = (4, 1, 5, 10,7, 9, 2 , 8 , 3 , 2)
Eredmény:
Ládacsomagok száma: 4,
Csomagok:
1. csomag = (4, 1)
2. csomag = (5, 2)
3. csomag = (10, 7, 3, 2)
4. csomag = (9, 8)

Megoldás

Ha félretesszük a mesét, észrevesszük, hogy a feladat tulajdonképpen azt kéri, hogy az adott sorozatot bontsuk fel minimális számú növekvő részsorozatra. A feladat megoldható egy *mohó algoritmus*sal, amely mindig a legkisebb olyan ládába csomagol, amelybe lehetséges. Észrevesszük, hogy így a ládacsomagokba utoljára elhelyezett ládák mérete növekvő sorozatot alkot, tehát a megfelelő csomag megkeresése lehetséges bináris kereséssel. Ugyanakkor az is világos, hogy nem egy ismert értéket kell megkeresnünk, hanem egy olyat, amely legkisebb az adott számnál nagyobbak között.

A gondot az adatok tárolása okozza, hiszen, ha a ládák csökkenő (vagy növekvő) sorrendben érkeznek, a következő két a legrosszabb esettel állunk szemben:

1. Ha a ládák csökkenő sorrendben érkeznek, egyetlen csomagba befér minden láda, tehát egyetlen növekvő részsorozatunk lesz, aminek a hossza legtöbb 15000.
2. Ha a ládák növekvő sorrendben érkeznek, akkor minden érkező láda új csomagnak felel meg, tehát legtöbb 15000 darab egy elemű részsorozatunk lesz.

A fentieket figyelembe véve egy 15000 × 15000 méretű tömböt kellene létrehozzunk, ami (ha lehetséges a választott programozási környezetben) nagyon nagy tárpazarlást jelent, hiszen még tárolnunk kell a csomagok hosszát is egy legtöbb 15000 elemű tömbben. A megoldást a dinamikus tárkezelés hozza: minden csomag egy verem típusú lista lesz, amelynek a feje az utoljára elhelyezett láda méretét tartalmazza. A bináris keresést a veremfejek sorozatán végezzük: ha nem találunk olyan ládát, amelybe az aktuális láda elhelyezhető, akkor új csomagot indítunk, különben elhelyezzük az aktuális ládát a megfelelő csomag tetejére. Végül kiírjuk a létrehozott verem típusú listák számát és ezeknek tartalmát. Megjegyezzük, hogy a csomagok tárolását statikusan is elvégezhetjük, egy 15000 elemből álló vektor segítségével, melyben minden ládára tároljuk, hogy melyik ládát helyeztük el benne.

A keresést az alábbi algoritmussal végezzük, amelyben felismerhető a bináris keresés:

```
Algoritmus Keres(bal,jobb,új):
            { keressük a bal..jobb sorszámú csomagok közt azt, amelybe elhelyezhető az új méretű láda }
                { Bemeneti paraméterek: bal, jobb a ládacsomagok tömbszakaszának kezdő és végpontja }
                                                        { az új méretű ládát fogjuk elhelyezni }
  Ha bal > jobb akkor                                                    { sikertelen keresés, }
    jobb  ←  jobb + 1
    csomagokszáma ← jobb
    Helyez(csomagok,csomagokszáma,új)          { új csomagot kezdünk, a jobb sorszámú után }
  különben
    Ha csomagok[bal]^.méret > új akkor                                      { sikeres keresés }
      Helyez(csomagok,bal,új)              { az új méretű ládát a bal sorszámú csomagba tesszük }
    különben
      közép ← [(bal+jobb)/2]                                                { tovább keresünk }
      Ha új < csomagok[közép]^.méret akkor
        Keres(bal,közép,új)
      különben
        Keres(közép+1,jobb,új)
      vége(ha)
    vége(ha)
  vége(ha)
Vége(algoritmus)
```

Ezúttal megadjuk a teljes megoldást egy Pascal programmal:

```
type csomag=^lada;                          { mutató, amely egy veremmel ábrázolt csomagra mutat }
     lada=record
            meret:Word;                                          { a láda mérete }
            kov:csomag                       { a csomagban az aktuális láda alatti láda címe }
          end;
     csomagokTipusa=array [1..15000] of csomag;                  { a veremfejek sorozata }

var csomagokszama:Word;                                          { csomagok/vermek száma }
    csomagok:csomagokTipusa;

procedure Helyez(var csomagok:csomagokTipusa; csomagokszama,uj:Word);
      { elhelyezzük az uj méretű ládát a csomagokszama sorszámú csomagba }
var p:csomag;
begin
  New(p);
  p^.meret:=uj;
  p^.kov:=csomagok[csomagokszama];
  csomagok[csomagokszama]:=p
end;

procedure Keres(bal,jobb,uj:Word);
            { keressük a bal..jobb sorszámú csomagok közt azt, amelybe elhelyezhető az uj méretű láda }
var kozep:Word;
    p:csomag;
begin
  if bal>jobb then begin                                         { sikertelen keresés, }
    Inc(jobb);
    csomagokszama:=jobb;
    Helyez(csomagok,csomagokszama,uj)          { új csomagot kezdünk, a jobb sorszámú után }
  end else begin
    if csomagok[bal]^.meret>uj then                              { sikeres keresés }
      Helyez(csomagok,bal,uj)              { az uj méretű ládát a bal sorszámú csomagba tesszük }
    else begin
      kozep:=(bal+jobb) div 2;                                   { tovább keresünk }
      if uj<csomagok[kozep]^.meret then
        Keres(bal,kozep,uj)
      else
        Keres(kozep+1,jobb,uj)
    end
  end
end;

procedure Beolvas(var csomagokszama:Word; var csomagok:csomagoktipusa);
var n,i:Word;
    uj:Word;
begin
  ReadLn(n);
  csomagokszama:=0;                                              { még nincs egy csomag sem }
  for i:=1 to n do
    csomagok[i]:=nil;                                            { vagyis minden csomag üres }
  for i:=1 to n do begin
    Read(uj);                                                    { a soron következő láda mérete }
    Keres(1,csomagokszama,uj)                                    { megkeressük a helyét }
  end
end;

procedure Kiir(csomagokszama:Word; var csomagok:csomagoktipusa);
var i:Word;
    fej:csomag;
begin
  WriteLn(csomagokszama);
  for i:=1 to csomagokszama do begin
    fej:=csomagok[i];
    while fej<>nil do begin
      Write(fej^.meret,' ');
```

```
      fej:=fej^.kov
    end;
    WriteLn
  end
end;

Begin
  Beolvas(csomagokszama,csomagok);
  Kiir(csomagokszama,csomagok)
End.
```

3. Következtetések

A 3. és 4. feladatok megoldásai a következő általános szabályhoz vezetnek: ha egy maximális értéket kell meghatároznunk, amelynek olyan követelményeknek kell eleget tennie, amelyek ha teljesülnek egy bizonyos értékre, akkor biztosan teljesülnek az ennél kisebbekre, akkor a bináris kereséssel és a feltételek utólagos ellenőrzésével log *n* lépésben eredményt kapunk.

A szabály alkalmazható minimum esetében is.

Az elv alkalmazása az algoritmus végrehajtási idejének csökkentését eredményezi, de ehhez előbb be kell látnunk, hogy ez lehetséges, majd meg kell találnunk az alkalmazás módját.

Szakirodalom

1. Cormen T., Leiserson C., Rivest R. – *Algoritmusok*, Műszaki Könyvkiadó, Budapest, 1997.
2. Ionescu K. – *Bevezetés az algoritmikába,* Presa Universitară Clujeană, 2005.
3. Stroe M. – *Metode de reducere a complexităţii*, GInfo, Agora Media, Tg. Mureş, 13/7 (2003), 28–29 (román nyelven).