

Algoritmusok bonyolultsága
A bináris keresés elvének alkalmazása
algoritmusok hatékonyságának
növelése céljából

Dr. Ionescu Klára
clara@cs.ubbcluj.ro

Az algoritmus végrehajtásához szükséges idő

- A végrehajtáshoz szükséges idő az algoritmus „*bonyolultságát*” fejezi ki.
- Nem mérünk pontos időt, hiszen ez sok „külső” feltételtől függ.
- Szükség van egy „*mértékegységre*”, ami algoritmusonként változhat: *megnevezünk egy bizonyos műveletet, amit az adott algoritmus valamelyik ciklusában adott számszor végrehajt, és ennek számosságával mérjük az algoritmus teljesítményét.*
- A művelet lehet: egy érték **beolvasása**, egy érték **kiírása**, valamely **aritmetikai, logikai, relációs művelet, összehasonlítás**.

Példák

1. Egy sorozatban meg kell keresnünk az adott **X** értéket: a művelet az **összehasonlítás** lesz.
2. Össze kell szoroznunk két mátrixot: **két valós szám szorzása** lesz a mértékegység.
3. Két nagyon nagy számot kell összeadnunk: **két számjegy feldolgozása** lesz az elemi művelet

A bemeneti adatok méretének szerepe

- Az algoritmus bemeneti adatainak ismert a számosságuk (illetve a határok, amelyek között ez a számosság mozog).
- Az alpműveletet általában minden adatra elvégezzük \Rightarrow ilyenkor *a bemenet mérete megadja a végrehajtások számát is.*

Példák

1. Ha egy *n* elemű sorozatban meg kell keresnünk egy bizonyos elemet, akkor az algoritmus legtöbb *n összehasonlítást* fog elvégezni.
2. Két mátrix összeszorzásakor a műveletek számát *a két mátrix méretének* függvényében számoljuk.
3. Nagy számok összeadásakor *a számok számjegyeinek száma* a méret.
4. A gráfelméleti feladatokban *a csomópontok száma vagy az élek száma* jöhet számításba, esetleg mind a kettő.

Algoritmusok bonyolultsága

- *Az algoritmus bonyolultságát az a függvény írja le, amely az algoritmus futásának idő és/vagy helyigényét adja meg a bevitt adatok számának/méretének függvényében kifejezve.*
- Az időbonyolultságot nem mérjük másodpercben, hanem a **lépések számával**, mivel ez a szám nem függ a számítógép minőségétől.
- **A függvény argumentuma a feldolgozandó adatok száma** (általában: ***n***)
- A hely bonyolultságát is függvénnnyel adjuk meg, ahol az argumentum szintén a feldolgozandó adatok száma.
- A fentiekből következik, hogy a bonyolultságot ***n*** függvényében mindig egy **egész számot** eredményező kifejezéssel adjuk meg.
- Ez egy **becslés**, nem precíz számolások eredménye.

A végrehajtási idő

- A bemeneti adatok tulajdonságainak függvényében ugyanaz az algoritmus **változó** mennyiségű idő alatt hajtódik végre.

Példa: Legyen **S** egy **n** elemű, egész számokból álló sorozat. Keressük meg az **X** adott számmal egyenlő elem indexét a sorozatban! Ha **X** nincs a sorozatban, az eredmény legyen **0**!

- A keresés **leállhat az első összehasonlítás után**, de ha a keresett elem nincs a sorozatban, akkor **n összehasonlítást végzünk**.
- A végrehajtási időt meghatározhatjuk az úgynevezett **legrosszabb eset**ben, a **legjobb eset**ben, vagy beszélhetünk **átlagos végrehajtási idő**ről.

Algorithm Keres(n, S, X):

// bemeneti adatok: n, S, X
// kimeneti adat: i

$i \leftarrow 1$

While $i \leq n$ AND $S_i \neq X$ execute

$i \leftarrow i + 1$

EndWhile

If $i > n$ then

$i \leftarrow 0$

EndIf

return i

EndAlgorithm

Átlagos végrehajtási idő

Alapművelet (az előbbi példában): a sorozat egy elemének összehasonlítása X -szel.

- A bemenetek különbözőségét X pozíciója fejezi ki. Ha X megtalálható a sorozatban, ez lehet az első, a második, ..., az n -edik elem; de az is lehetséges, hogy nincs \Rightarrow összesen $n + 1$ eset lehetséges.
- Ha X megtalálható a sorozatban, akkor átlagosan $T(n) = (n + 1)/2$ összehasonlítást végzünk, vagyis \Rightarrow **átlagban a sorozat felét kell megvizsgálnunk.**
- Így szokványos esetekre kapjuk meg az algoritmus lépéseinek számát, tehát ez az átlag **a várható futási időt fejezi ki.**
- Ugyanakkor: elég nagy a valószínűsége annak, hogy ez soha nem fordul elő, hiszen ez csak egy „jóslás” ...

A legrosszabb és a legjobb eset

A legrosszabb eset a legkedvezőtlenebb esetet jellemzi: *garantáltan nem fordulhat elő nagyobb futási idő.*

- Ha X a sorozat utolsó elemével egyenlő vagy X nincs a sorozatban, „pechesek” vagyunk és a végrehajtási idő a lehető legnagyobb lesz:

$$T(n) = \max\{T_i\}: i = 1, 2, \dots, (n + 1)\} = n.$$

A legjobb eset a legkedvezőbb esetben fejezi ki az algoritmus lépéseinek számát: *garantáltan nem fordulhat elő kisebb futási idő.*

- Ha szerencsések vagyunk, és megtaláljuk X -et az első helyen, a lehetséges legrövidebb futási idő alatt fut le az algoritmus:

$$T(n) = \min\{T_i\}: i = 1, 2, \dots, (n + 1)\} = 1$$

Az algoritmus növekedési rendje

Következtetések:

- Nem érdemes, de nem is lehetséges pontos, precíz értéket keresni a futási idő megadására.
- Az esetek túlnyomó többségében arra kell szorítkoznunk, hogy a végrehajtási idő *nagyságrend*jét határozzuk meg.
- Ha a bemenet méretét n -nel jelöljük, a végrehajtás idejét kifejezhetjük n *függvényeként*.
 - A futási időt kifejező képletnek csak a fő tagját tartjuk meg (például, ha a képlet $an^2 + bn + c$, csak az an^2 tagot tartjuk meg), mivel az alacsonyabb rendű tagok nagy n -re kevésbé lényegesek.
 - Szintén figyelmen kívül hagyjuk a fő tag konstans szorzóját, mivel a nagy bemenetekre ezek elhanyagolhatók.
- Ez a *növekedési rend*, és a $\Theta(g(n))$ függvénnyel jelöljük.

Növekedési rend

- Az algoritmus futási idejének **növekedési rendje**, (**sebessége**) kifejezi az algoritmus hatékonyságát és
- **eszköz algoritmusok minőségi összehasonlítására.**
 - Pl. ha az **n** bemeneti méret elég nagy, akkor az **összefésülő rendezés**, amelynek futási ideje átlagos esetben **$\Theta(n \lg n)$** jobb, mint a **beszűrő rendezés**, amelynek növekedési rendje **$\Theta(n^2)$** .
- Általában nem számoljuk ki **pontosan** az algoritmus futási idejét, mivel elég nagy bemeneti adatokra a pontos futási idő **multiplikatív állandóinak** és **alacsonyabb rendű tagjainak** a hatása **eltörpül** a futási idő nagyságrendjéhez képest.

Példa

Tekintsük az összegszámító algoritmusunkat:

Algorithm Összegszámítás(N, X):

// bemeneti adatok: az N elemű X sorozat, kimeneti adat: S

$S \leftarrow 0$

For $i = 1, N$ **execute**

$S \leftarrow S + X_i$

EndFor

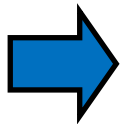
return S

EndAlgorithm

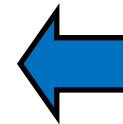
- Megszámoljuk a műveleteket: **1** (S kezdőértéket kap) **+ 1** (i kezdőértéke) **+ n** (i növelése) **+ n** (i összehasonlítása n -nel) **+ n** (az összeadások) **+ 1** (a returnés) = **$3 + 3n$** .
- De a **3**-as konstansból álló tagot nem vesszük figyelembe és a **3**-as szorzót sem, mivel a konstans szorzóktól is eltekintünk.
- Így az algoritmus bonyolultsága: **$\Theta(n)$** ami egyben itt **$O(n)$** is.

Lépések száma a bemeneti adatok számának függvényében

Bemeneti adatok száma	$\log n$	n	$n \cdot \log n$	n^2	n^3	2^n
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
1000	10	1000	10000	10^6	10^9	2^{1000}



Lépések száma



Balról jobbra a függvények értékei egyre nagyobbak.
Kivétel (ha $n = 4$ és $n = 8$): $4^3 > 2^4$

Aszimptotikus hatékonyság

Algoritmusok hatékonyságának elemzésekor feltesszük a kérdést:

Miként növekszik az algoritmus futási ideje, ha nő a bemenet mérete, vagyis $n \rightarrow \infty$.

$\Theta(g(n)) = \{ f(n) : \text{létezik } c_1, c_2 \text{ és } n_0 \text{ pozitív állandó úgy, hogy } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ bármely } n \geq n_0 \text{ esetén} \}$

Más szóval, minden $n \geq n_0$ esetén az $f(n)$ függvény – egy állandó szorzótényezőtől eltekintve – egyenlő $g(n)$ -nel

$\Rightarrow g(n)$ **aszimptotikusan éles korlátja** $f(n)$ -nek.

- Gyakran fogjuk használni a $\Theta(1)$ jelölést az **állandó** végrehajtási időre. Ekkor **az algoritmus végrehajtási ideje nem függ a bemenet méretétől.**

Magyarázat: Mivel bármely állandó egy nulla fokú polinom, ezért bármelyik konstans függvényre fennáll a $\Theta(n^0) = \Theta(1)$ becslés.

Példa

Tekintsük az ismert felcserélés algoritmust:

Algorithm Felcserél(*a*, *b*):

// bemeneti adatok: a, b

// kimeneti adatok: a, b

 segéd ← *a*

a ← *b*

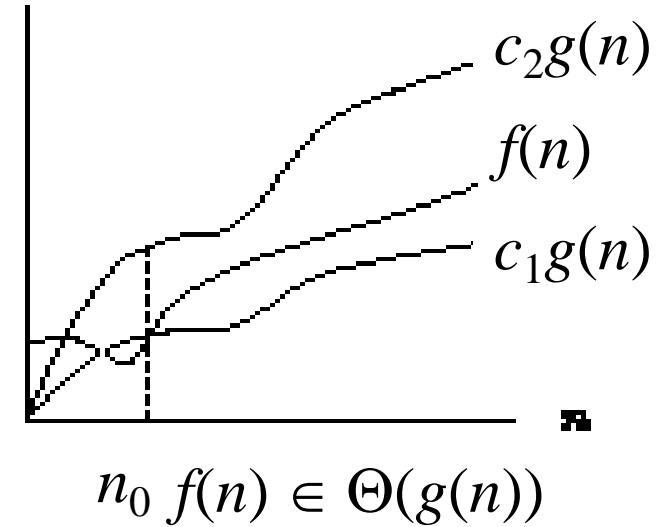
b ← segéd

EndAlgorithm

- Az algoritmus *x* és *y* értékeitől függetlenül **3** művelet végez el.
- Ez a **3** egy *konstans*, tehát az algoritmus bonyolultsága $\Theta(1)$.
- De akkor sem tévedünk, ha $O(1)$ -gyel fejezzük ki, mivel $\Theta(1) = O(1)$.

A Θ -jelölés

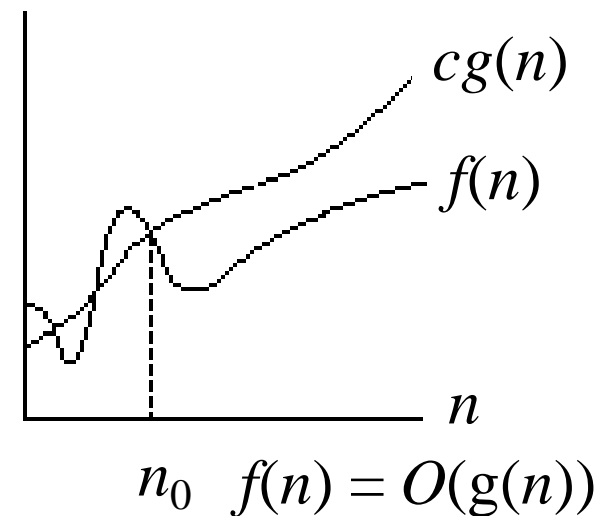
- Minden $n \geq n_0$ esetén az $f(n)$ függvény – egy állandó szorzótényezőtől eltekintve – egyenlő $g(n)$ -nel:
- $g(n)$ aszimptotikusan éles korlátja $f(n)$ -nek.
- Minden $f(n) \in \Theta(g(n))$ aszimptotikusan **nem-negatív** kell legyen \Rightarrow a $g(n)$ függvénynek aszimptotikusan nem-negatívnak kell lennie (else $\Theta(g(n))$ üres).



Az O-jelölés

- Az O -jelölés **aszimptotikus felső korlátot** ad az algoritmus futási idejére.
- Egy adott $g(n)$ függvény esetén $O(g(n))$ -nel jelöljük a függvényeknek azt a halmazát, amelyre

$O(g(n)) = \{ f(n) : \text{létezik } c \text{ és } n_0 \text{ pozitív állandó úgy, hogy } 0 \leq f(n) \leq cg(n) \text{ bármely } n \geq n_0 \text{ esetén} \}$



Példa

Határozzuk meg az n természetes szám legnagyobb páratlan osztóját!

Algorithm legnagyobbPáratlanOsztó(n):

// bemenet: az n természetes szám, kimenet: a Legnagyobb páratlan osztó

$S \leftarrow 0$

While $n \bmod 2 = 0$ **execute**

$n \leftarrow n \text{ DIV } 2$

EndWhile

return n

EndAlgorithm

Az algoritmus bonyolultsága $O(\log n)$.

Magyarázat: \Rightarrow

Magyarázat

- Felírjuk az n számot 2 -es számrendszerben.
- Észrevesszük, hogy az ábrázolás **bitjeinek darabszáma** = 2 -es alapú **logaritmus a számból + 1**.

Példák:

- Ha $n = 7 = 111_{(2)}$, $\log_2 7$ egész része = 2 , hozzáadva 1 -et megkapjuk a 7 bitjeinek darabszámát.
- Legyen $n = 16 = 10000_{(2)}$, $\log_2 16 = 4$, hozzáadva 1 -et megkapjuk a 16 bitjeinek darabszámát.

Magyarázat

- Az algoritmusban addig osztunk (**DIV**) újra meg újra, amíg a **2**-vel történő egész osztás maradéka (**MOD**) egyenlő **0**-val.
- Például, a **7** esetében **0**-szor, a **16** esetében **4**-szer, vagyis a *legrosszabb esetben* annyiszor ahány bitje van ***n***-nek (**-1**).
- Ez akkor fordul elő, amikor a szám **kettőhatvány**; ekkor az **while** addig dolgozik, míg ***n*** értéke **1** lesz. Tehát a lépésszám **az *n* 2-es alapú logaritmusával egyenlő**.
- **Tipp: ha egy ciklus minden lépésében az a változó, amely irányítja a ciklust (legyen ez *n*) osztódik 2-vel, valószínű, hogy a ciklus lépésszáma $\log_2 n$.**
- Az alapot nem mindig tüntetjük fel, mivel ez (szintén) konstans.
- **Fontos: minél nagyobb az alap, annál jobb az algoritmus hatékonysága.**

Példa

- Határozzuk meg az $\{1, 2, \dots, n\}$ halmaz minden permutációját!
- Tudjuk, hogy összesen $n!$ permutációnk lesz, tehát az algoritmus lépéseinek száma nem lehet kisebb, mint $n!$
- Határozzuk meg az $\{1, 2, \dots, n\}$ halmaz minden részhalmazát!
- Tudjuk, hogy összesen 2^n részhalmazunk lesz (az üres halmazt is beleszámítva), tehát az algoritmus lépéseinek száma nem lehet kisebb, mint 2^n .
- De $n! \geq 2^n$, ha $n \geq 4$
- Következik, hogy mindkét algoritmus bonyolultsága exponenciális lesz: $O(2^n)$

Példa

Határozzuk meg az n -nél kisebb és n -nel egyenlő számok osztóinak darabszámát!

Algorithm osztókSzáma(n , d):

// bemenet: az n természetes szám

// kimenet: a d sorozat, ahol $d[i]$ az i osztóinak száma

For $i = 1, n$ **execute**

$d[i] \leftarrow 0$

EndFor

For $i = 1, n$ **execute**

For $j = i, n, i$ **execute**

$d[j] \leftarrow d[j] + 1$

EndFor

EndFor

EndAlgorithm

Lépésszám?

- Számoljuk meg a ciklus lépéseit!
- A külső **For** ciklusnak n lépése van; írhatjuk: $O(n * ?)$
- Ahhoz, hogy a $?$ -t „kiszámítsuk”, felírjuk az i egymás után következő értékeire a lépésszámokat:
- Ha $i = 1$, a lépésszám = n
- Ha $i = 2$, a lépésszám = $n / 2$
- Ha $i = 3$, a lépésszám = $n / 3$
- Ha $i = n$, a lépésszám = n / n
- Ha összeadjuk ezeket:
$$n + n / 2 + n / 3 + \dots + n / n = n * (1 + 1 / 2 + 1 / 3 + \dots + 1 / n) \approx n * \log n$$
- Következik, hogy az algoritmus bonyolultsága: $O(n * \log n)$

Gyakorlat

- Legyen a következő algoritmus. Határozzuk meg a lépésszámot.

Algorithm **hányLépés(n)**

sz \leftarrow 0

i \leftarrow n

While **i** \geq 1 **execute**

j \leftarrow i

While **j** \geq 1 **execute**

sz \leftarrow **sz** + 1

j \leftarrow **j** - 1

EndFor

i \leftarrow **i** DIV 2

EndFor

return **sz**

EndAlgorithm

Megoldás

- Számolunk:
- Ha $i = n$, a lépésszám = n
- Ha $i = n / 2$, a lépésszám = $n / 2$
- Ha $i = n / 2^2$, a lépésszám = $n / 2^2$
- Ha $i = n / 2^k$, a lépésszám = $n / 2^k$, ahol $2^k = n$
- Ha összeadjuk ezeket:

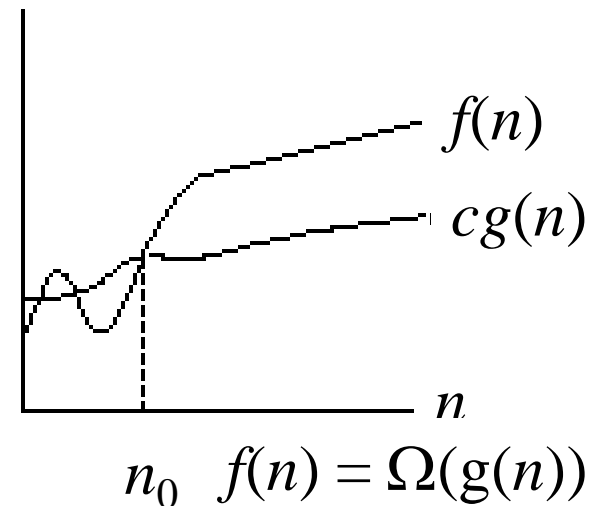
$$n + n / 2 + n / 2^2 + \dots + n / 2^k = 2 * n - 1$$

- Következik, hogy a bonyolultság $O(n)$
- Vagyis nem 100%-os az előbbi tipp ☺ Itt is osztottunk 2-vel, de a logaritmikus szorzótényező „eltűnt”

Az Ω -jelölés

- **Aszimptotikus alsó korlátot** ad a függvényre.
- Egy adott $g(n)$ függvény esetén $\Omega(g(n))$ -nel jelöljük a függvényeknek azt a halmazát, amelyre

$\Omega(g(n)) = \{f(n) : \text{létezik } c \text{ és } n_0 \text{ pozitív állandó úgy, hogy } 0 \leq cg(n) \leq f(n) \text{ bármely } n \geq n_0 \text{ esetén} \}$



Példa: buborékrendezés, mivel, ha „szerencsések” voltunk, és a bemeneti sorozat a kívánt módon rendezett volt a, elég lesz egyszer bejárni: $\Omega(n)$

Az algoritmus által feldolgozott adatok számára szükséges memória mérete

- Gyakran előfordul, hogy egy program, a bemeneti és kimeneti adatokon kívül, *ideiglenesen létrehozott adatszerkezetekkel* is dolgozik.
- Ugyanakkor: ha egy algoritmus a feldolgozás közben, a bemeneti és kimeneti adatokon kívül csak *konstans méretű plusz memóriát* vesz igénybe, azt mondjuk, hogy *helyben* dolgozik.
- Konstans méretű memóriáról beszélünk, ha a lefoglalt memória mérete *nem függ a bemeneti adatok számától* (a függvény, amivel leírjuk nem tartalmazza a bemenet méretét jelölő változót).

Példa:

Legyen egy program, amely megadja egy adott keresztnév esetében a névnapot. Ehhez tárolnia kell a „kalendáriumot”, amelynek a mérete rögzített és nem függ a *keresett* névnapok számától.

Algoritmusok egyszerűsége

- Az egyszerűség nem feltétlenül mérvadó tulajdonság.
- Ha egy algoritmust a **legkézenfekvőbb ötlet alapján** tervezünk meg (**brute force**), sokszor (néha, gyakran) fölöslegesen terheljük a számítógép erőforrásait.
- Magyarul: **naiv algoritmus**nak hívjuk.
- Ugyanakkor egy egyszerű algoritmust könnyebben írunk meg, olvashatóbb, könnyebben javítjuk és könnyebben bizonyítjuk a helyességét.
- Ne gondoljuk, hogy minden naiv algoritmus visszaél az erőforrásokkal, hiszen: **léteznek optimális naiv algoritmusok!**

Algoritmusok optimalitása

- Feltételezzük, hogy egy adott feladatnak megvalósítottuk az algoritmusát, és kiszámítottuk a végrehajtásához szükséges időt.

Kérdés: vajon létezik-e jobb (kisebb végrehajtási idejű) algoritmus, mint amit megterveztünk?

- **Az optimalitás bizonyítása**kor számításba kell vennünk **minden** lehetséges algoritmust, és ki kell mutatnunk, hogy ezeknek a futási ideje nagyobb mint a tárgyalt algoritmusé.
 - Feltételezzük, hogy egy adott feladat megoldására talált minden algoritmus legkevesebb **$T(n)$** időt igényel.
 - Ha egy időegység alatt az algoritmus egy műveletet végez, akkor a **$T(n)$** az elvégzendő műveletek számát jelenti.
 - Egy algoritmus akkor optimális az adott algoritmushalmazon belül, ha legrosszabb esetben **$T(n)$** műveletet végez, vagyis minimális lépésszámot: ez azt jelenti, hogy a leghatékonyabb!!!

Példa

Határozzuk meg egy n elemű sorozat minimumát!

Algorithm Minimum(n , a , \min):

// Bemeneti adatok: n , a , Kimeneti adat: \min

$\min \leftarrow a_1$

For $i = 2, n$ **execute**

If $a_i < \min$ **then**

$\min \leftarrow a_i$

EndIf

EndFor

EndAlgorithm

Elemzés

Az algoritmusban az $a = (a_1, a_2, \dots, a_n)$ sorozat elemeivel pontosan $n - 1$ összehasonlítást végzünk.

Állítás: A fenti algoritmus optimális.

Bizonyítás: Indukciót használva kimutatjuk, hogy bármely, összehasonlításokra épülő algoritmus legkevesebb $n - 1$ műveletet végez.

- Ha $n = 1$, **nem** végzünk egyetlen összehasonlítást sem ($0 = n - 1$ összehasonlítás).
- Feltételezzük, hogy bármely algoritmus, amely megoldja a feladatot n szám esetében, legkevesebb $n - 1$ összehasonlítást végez, és tárgyaljuk azt a feladatot, amely $n + 1$ szám minimumát kéri.

Bizonyítás (folyt.)

- Legyen az első összehasonlítás, amely (anélkül, hogy vesztenénk az általánosságból) összehasonlítja a_1 -t a_2 -vel, és megállapítja, hogy $a_1 < a_2$.
- A továbbiakban meg kell oldanunk a feladatot az $(a_1, a_3, \dots, a_{n+1})$ sorozat esetében.
- De ezt a feladatot (a sorozatnak n eleme van) $n - 1$ összehasonlítással végeztük, tehát az $n + 1$ elemű sorozat minimumát összesen n összehasonlítás után kaptuk meg.

Híres algoritmusok bonyolultsága

- Szekvenciális keresés: $O(n)$ (helyben dolgozik)
- Kiválogatás: $O(n)$ (dolgozhat helyben vagy használhat segédsorozatot)
- Összefésülés: $O(n + m)$ (nem dolgozik helyben)
- Bináris keresés: $O(\log n)$ (helyben dolgozik)
- Buborékrende­zés: $O(n^2)$ és $\Omega(n)$ (helyben dolgozik)
- Láda­rende­zés: $O(n)$ (nem dolgozik helyben)
- Gyors­rende­zés: $\Theta(n \log n)$ és $O(n^2)$ (helyben dolgozik)
- Összefésülő rende­zés: $O(n \log n)$ (nem dolgozik helyben)
- Kupac­rende­zés: $O(n \log n)$ (helyben dolgozik)

Oszd meg és uralkodj módszer. Bevezetés

Az *Oszd meg és uralkodj* módszer akkor **ajánlott** amikor:

- a feladatot fel lehet bontani **egymástól független részfeladatokra**,
 - amelyeket az **eredeti feladathoz hasonlóan oldunk meg**, de kisebb méretű adathalmaz esetében.
1. Az eredeti feladatot **felbontjuk** egymástól **független részfeladatokra**: az eredetihez hasonlóak, de kisebb adathalmazra definiáltak.
 2. A részfeladatokkal hasonlóan járunk. A felbontást **akkor állítjuk le**, amikor a feladat megoldása a lehető **legegyszerűbb**.
 3. Ezt a legegyszerűbb feladatot **megoldjuk**.
 4. A részfeladatok eredményeiből fokozatosan felépítjük mindig a következő méretű feladat eredményeit, ezek **összerakása** által. Az utolsó összerakás az eredeti feladat eredményét adja.

Megjegyzések

1. A részfeladatok csak méreteikben különböznek az eredeti feladattól \Rightarrow a *Divide et Impera* módszert **rekurzívan** fejezzük ki.
2. A **felbontás** megtörténik a rekurzióba való **belépéskor**, a **részeredmények összerakása** pedig a **kilépéskor**.

A módszer általános bemutatása

- A **DivImp(bal, jobb)** algoritmus az a_1, a_2, \dots, a_n sorozatot dolgozza fel, tehát **DivImp(1, n)** alakban hívjuk meg először.
- Formális paraméterei **bal** és **jobb**, amelyek az aktuális részsorozat bal és jobb indexei.
- Ha nem szeretnénk globális változókkal dolgozni, ehhez hozzáadódnak: **n** és **a**.
- Az *Oszd meg és uralkodj* stratégiát lehet **iteratívan is** implementálni (például, *bináris keresés*).
 - Ezek az algoritmusok mindig gyorsabbak lesznek.

```
Algorithm DivImp(bal, jobb, eredmény):  
    If jobb - bal <  $\epsilon$  then  
        Megold(bal, jobb, eredmény)  
    else  
        Feloszt(bal, jobb, közép)  
        DivImp(bal, közép, eredmény1)  
        DivImp(közép+1, jobb, eredmény2)  
        Összerak(eredmény1, eredmény2, eredmény)  
    EndIf  
EndAlgorithm
```

A „leghíresebb” és egyben a leggyakrabban alkalmazott Oszd meg és Uralkodj típusú algoritmus a **„bináris keresés”**.

Bináris keresés

Adott egy n egész számból álló, növekvő sorrendbe **rendezett** sorozat: $x_1 < x_2 < \dots < x_n$.
Állapítsuk meg egy adott szám helyét a sorozatban! Ha az illető szám nem található meg a sorozatban, írjunk ki egy megfelelő üzenetet!

Elemzés

Az elemet a sorozat közepén fogjuk először keresni.

1. **$keresett = x_{közép}$** \Rightarrow **$keresett$** a sorozatban a **$közép$** helyen van

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

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

A feladat **átalakul ugyan két feladattá**, de **csak az egyiket kell megoldani**.

Nincs szükség a divide et impera harmadik lépésére (a részeredmények összerakására).

```
Algorithm BinKeres(bal, jobb):  
    If bal > jobb then                                // közép = a keresett helye, ha megtalálható  
        return -1                                    // ha keresett nincs a sorozatban  
    else  
        közép ← (bal + jobb) div 2  
        If keresett >  $x_{\text{közép}}$  then  
            return BinKeres(közép + 1, jobb)  
        else  
            If keresett <  $x_{\text{közép}}$  then  
                return BinKeres(bal, közép - 1)  
            else  
                return közép  
            EndIf  
        EndIf  
    EndIf  
EndAlgorithm
```

Megjegyzés: ha a számok nem különbözők, és *keresett* többször is előfordul, az algoritmus a keresett *tetszőleges* előfordulását téríti.

Algorithm IteratívBinKeres(n , x , keresett):

$bal \leftarrow 1$

$jobb \leftarrow n$

While $bal \leq jobb$ **execute**

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

If $x_{közép} = keresett$ **then**

return $közép$

// ha megtaláltuk, vége

else

Ha $x_{közép} > keresett$ **then**

$jobb \leftarrow közép - 1$

else

$bal \leftarrow közép + 1$

EndIf

EndIf

EndWhile

return -1

// csak, ha nem találtuk meg

EndAlgorithm

A bináris keresés alkalmazásai

- Következik néhány feladat, amelyeknek megoldásaiban felhasználjuk a bináris keresést.
- Ezekben a *keresés* nem jelenik meg, mint explicit részfeladat, hanem az eredmény egy olyan érték, amelynek értéktartományát ismerjük, ugyanakkor lehetséges a „találgatás” a bináris keresés algoritmusának alkalmazásával.
- Tehát: a bináris keresés algoritmusa eredményesen alkalmazható algoritmusok *optimalizálásának* érdekében.
- Bizonyos feladattípusok esetében, a megoldásként javasolt *lineáris* algoritmus *egy logaritmikussal helyettesíthető*, ha felhasználjuk a *bináris keresés elvét*.

1. Keresztmetszet (Mat-Info verseny, 2016)

- Legyen két sorozat, amelyeknek elemei különböző természetes számok: az **a** sorozat elemeinek száma **n** ($0 < n \leq 10\,000$), a **b** sorozat elemeinek száma **m** ($0 < m \leq 10\,000$) és növekvően rendezett.
- Határozzuk meg azt a **c** sorozatot, amelynek **k** ($0 < k \leq 10\,000$) eleme lesz, és amely a két sorozat minden közös elemét egyszer tartalmazza.

Példa: ha **n** = 4, **a** = (5, -7, -2, 3), **m** = 5 és **b** = (-2, 3, 5, 7, 8), a **c** sorozatnak **k** = 3 eleme van: **c** = (5, -2, 3).

Elemzés

- Ismerjük a klasszikus algoritmust, amely két *nem* rendezett sorozat keresztmetszetét határozza meg (programozási tétel). Ennek bonyolultsága $O(n*m)$.
- „Észrevesszük”, hogy a **b** sorozat rendezett. De a fent említett algoritmus ezt nem „kéri”...
- De ne maradjon a tulajdonság kihasználatlanul! Így a **b** sorozatban rendre keressük az **a** sorozat elemeit... **bináris keresés**sel!

Algorithm közösElemek(n, a, m, b, k, c):

$k \leftarrow 0$

// egyelőre a c sorozatnak nincs egyetlen eleme sem

For $i = 1, n$ **execute**

// az a elemeket keressük rendre b-ben

megvan \leftarrow hamis

bal $\leftarrow 1$; jobb $\leftarrow m$

// a b rendezett, lehetséges a bináris keresés

While nem megvan és bal \leq jobb **execute**

közép $\leftarrow (bal + jobb) / 2$

If $a_i = b_{közép}$ **then**

// megtaláltuk: a_i közös

$k \leftarrow k + 1$

$c_k \leftarrow a_i$

// elhelyezzük a c sorozatban

megvan \leftarrow igaz

// a keresés leáll

else

If $a_i < b_{közép}$ **then**

jobb \leftarrow közép - 1

// tovább keresünk a b sorozat bal felében

else

bal \leftarrow közép + 1

// tovább keresünk a b sorozat jobb felében

EndIf

EndIf

EndWhile

EndFor

EndAlgorithm

2. S összegű elemek kiválasztása

Legyen egy n elemű ($3 \leq n \leq 100\,000$), különböző természetes számokat tartalmazó sorozat és az S természetes szám.

Válasszunk ki az adott sorozatból **három elemet, amelyeknek az összege S !** Adjunk meg minden megoldást!

Elemzés

- Részletösszegeket kell számítanunk: három elem összegét hasonlítjuk S -sel.
- A feladat megoldható három egymásba ágyazott **For** ciklussal is (az n értéke miatt, ez a megoldás időigényes; hiába ügyeskednénk **While** ciklusokkal, illetve a lépésszámok csökkentésével, a bonyolultság továbbra is $O(n^3)$ lenne).

Megoldás

- Ha a megoldásba beépítjük a bináris keresést, a bonyolultság $O(n^2 \cdot \log n)$ lesz:
- Ahhoz, hogy alkalmazhassuk, előbb **rendezzük** az adott sorozatot (ennek bonyolultsága pl. $O(n^2)$).
- Két **While** ciklussal kiválasztunk a sorozatból két elemet (legyen ezeknek indexe n_1 és n_2). Ennek bonyolultsága $O(n^2)$.
- **Megkeressük** az $S - a_{n_1} - a_{n_2}$ értéket a **bináris keresést alkalmazva**. Ennek bonyolultsága $O(\log n)$, de n^2 -szer.
- A megoldást tovább javítjuk (például, ha a_{n_1} értéke meghaladja S -t, kilépünk az első **While**-ből stb.).
- Tehát az algoritmus bonyolultsága:
$$O(n^2) + O(n^2 \cdot \log n) = O(n^2 \cdot \log n)$$

Algorithm Generál(a, n, S):

i \leftarrow 1

While **i** < n-1 és **a_i** < **S** **execute**

j \leftarrow **i** + 1

While **j** < n és **a_i** + **a_j** < **S** **execute**

BinKeres(**a**, **j**+1, **n**, **S** - **a_i** - **a_j**, **k**)

*// ha **S** - **a_i** - **a_j** megtalálható a sorozatban,*

*// akko **k** értéke egy valid index, különben **k** értéke 0*

If **k** \neq 0 **then**

Ki: **i**, **j**, **k**

EndIf

j \leftarrow **j** + 1

EndWhile

i \leftarrow **i** + 1

EndWhile

EndAlgorithm

3. Négyzetszámok darabszáma

Számoljuk meg egy n ($1 \leq n \leq 1\,000\,000$) elemű sorozat négyzetszámait!

A számok nem nagyobbak $1\,000\,000$ -nál.

Megoldás

Részfeladat: ellenőriznünk kell, hogy egy szám négyzetszám-e? Tudjuk már, hogy a következő algoritmus nem lesz kielégítő hatékonyságú.

Algorithm Számol_1(n, a, p):

p \leftarrow 0

For **i** = 1, n **execute**

If négyzetgyök(a_i) = [négyzetgyök(a_i)] **then**

p \leftarrow **p** + 1

EndIf

EndFor

EndAlgorithm

1. Tudjuk, hogy a négyzetgyök kiszámítása elkerülendő, mivel időigényesebb, mint gondolnánk. Főleg, ha sokszor kell alkalmaznunk.
2. Először is az **While** feltételéből „kiemeljük” az a_i -t *szám-ba*, mivel a_i -t *megkeresni a memóriában*, (az *i* index, a tömb első elemének címe és az *a* tömb típusának megfelelő elemhossz alapján) több idő, mint egy egyszerű változóban tárolt értéket keresni.
3. Azt hogy egy adott szám négyzetszám-e, hatékonyabban is tudjuk ellenőrizni:

Algorithm Szamol_2(n, a, p):

p \leftarrow 0

// p a négyzetszámokat számlálja

For i = 1, n **execute**

k \leftarrow 0

// a k négyzetét hasonlítjuk a_i-val

szám \leftarrow a_i

// az a_i-t „megkeresni” nehezebb

While k * k < szám **execute**

k \leftarrow k + 1

EndWhile

If k * k = szám **then**

// a k értéke = [négyzetgyök(szám)]

p \leftarrow p + 1

EndIf

EndFor

EndAlgorithm

Elemzés

- Ha implementáljuk az első, valamint a második algoritmust, és megmérjük a végrehajtási időt, látni fogjuk, hogy **az utóbbi sokkal kevesebb időt igényel, mint az első!**
- De még nem alkalmaztuk a bináris keresés elvét!
- Vegyük észre, hogy **nem érdemes minden k ($k * k < \text{szám}$) értékre elvégezni a vizsgálatot**, hiszen tulajdonképpen egy **rendezett halmazban keresünk!**
- **k** lehetséges értékei az **$\{1, 2, \dots, 1000\}$** halmazhoz tartozhatnak, mivel az ellenőrizendő számok kisebbek vagy egyenlők **1 000 000**-val.

Algorithm Számol_3(n, a, p):

p \leftarrow 0

For i = 1, n **execute**

szám \leftarrow a_i

eleje \leftarrow 0

vége \leftarrow 1000

// a Legnagyobb érték 1 000 000

While eleje < vége **execute**

k \leftarrow (eleje + vége)/2

If k * k \geq szám **then** vége \leftarrow k

else eleje \leftarrow k + 1

EndIf

EndWhile

If eleje * eleje = szám **then**

// eleje = [négyzetgyök(szám)]

p \leftarrow p + 1

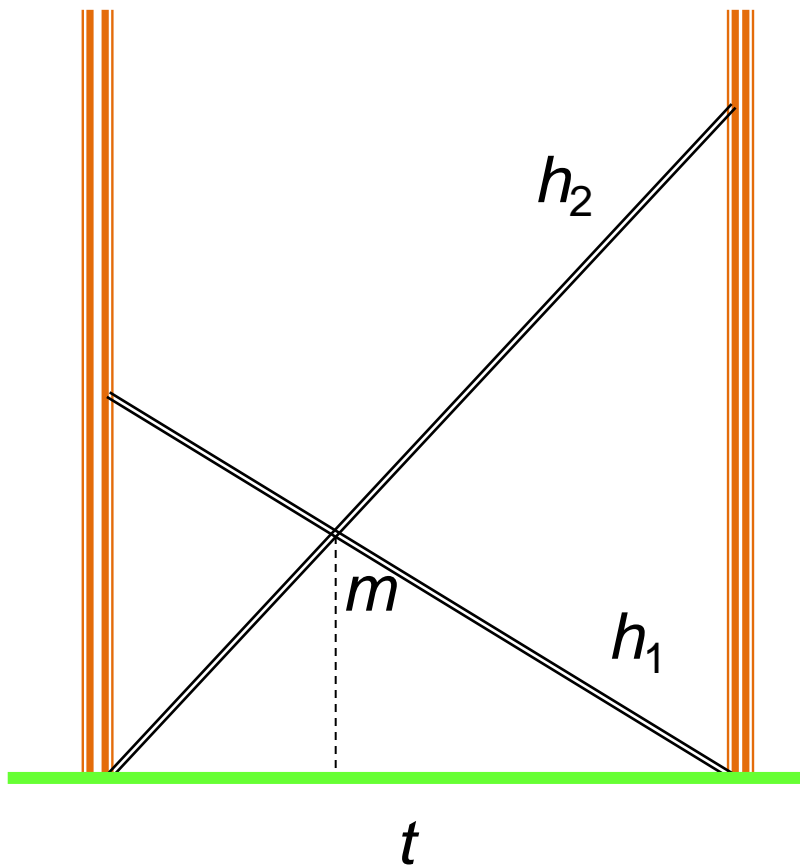
EndIf

EndFor

EndAlgorithm

4. Síkmértan

- 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}).



Észrevétel

A magasság, ahol a két deszka találkozik nő, ha a keresett t távolság csökken és fordítva.

Megoldás

- 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 megtalálása érdekében.
- A t lehetséges legkisebb értéke **0** (a két fal egymás mellett van), legnagyobb értéke pedig a rövidebb deszka hossza.
- Így a t értékét **0** és **$\text{Min}(h_1, h_2)$** között keressük.
- Miután van egy javaslat t -re, h_1 és h_2 értékeivel kiszámítjuk az érintkezési pont magasságát síkmértani ismeretekkel.
- Ha a kiszámított magasság nagyobb mint az adott m akkor növeljük t -t, else csökkentjük.

Algorithm deszkák(m , $h1$, $h2$):

$megvan \leftarrow hamis$

If $h1 > h2$ **then** $t \leftarrow h2$

else $t \leftarrow h1$

EndIf

$min \leftarrow 0$; $max \leftarrow t$

While **nem** $megvan$ **execute**

$t \leftarrow (min + max) / 2$

 számol($h1$, $h2$, t , sz)

If $abs(sz - m) \leq 0.00001$ **then**

$megvan \leftarrow igaz$

else

If $sz > m$ **then** $min \leftarrow t$

else $max \leftarrow t$

EndIf

EndIf

EndWhile

return t

EndAlgorithm

5. Ládák

- *Költözik a múzeum. A tárgyakat kocka alakú, különböző méretű ládákbba 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 leszedje a ládákat a futószalagról és úgy helyezze egyiket a másikba (ha lehet) hogy végül a ládacsomagok száma a lehető legkisebb legyen. A robotot egy program irányítja úgy, hogy:*

Ládák

- *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.*

Ládák

- Határozzuk meg a ládák számának ($0 \leq n \leq 15\,000$) és méreteiknek ($1 \leq \text{ládaMéret} \leq 10\,000$) ismeretében 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

- A feladat tulajdonképpen azt kéri, hogy **az adott sorozatot bontsuk fel minimális számú növekvő részsorozatra**.
- A feladat megoldható egy **mohó algoritmussal**, amely mindig a legkisebb olyan ládába csomagol, amelybe lehetséges.
- Észrevétel: a ládacsomagokba utoljára elhelyezett ládák mérete **növekvő sorozatot** alkot, tehát a megfelelő csomag megkeresése lehetséges **bináris kereséssel**.
- Ugyanakkor: nem egy ismert értéket kell megkeresnünk, hanem egy olyat, amely legkisebb az adott számnál nagyobbak között.

Megoldás

- Gond: az adatok tárolása, hiszen, ha a ládák csökkenő (vagy növekvő) sorrendben érkeznek, a következő két úgynevezett *„legrosszabb esettel”* állunk szemben:
 - Ha a ládák *csökkenő* sorrendben érkeznek, *egyetlen csomag*ba befér minden láda, tehát *egyetlen növekvő részsorozatunk lesz*, aminek a hossza legtovább **15 000**.
 - Ha a ládák *növekvő* sorrendben érkeznek, akkor minden érkező láda új csomagnak felel meg, tehát legtovább **15 000 darab egy elemű részsorozatunk** lesz.

Megoldás

- A fentieket figyelembe véve egy **15 000 × 15 000** 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,
- else elhelyezzük az aktuális ládát a megfelelő csomag tetejére.

```
Algorithm Keres(bal, jobb, új):  
    If bal > jobb then                                     // sikertelen keresés  
        jobb ← jobb + 1  
        csomagokszáma ← jobb  
        Helyez(csomagok, csomagokszáma, új) // új csomagot kezdünk, a jobb után  
    else  
        If csomagok[bal]^méret > új then                 // megvan  
            Helyez(csomagok, bal, új) // az új ládát a bal csomagba tesszük  
        else  
            közép ← (bal + jobb)/2                       // tovább keresünk  
            If új < csomagok[közép]^méret then  
                Keres(bal, közép, új)  
            else  
                Keres(közép + 1, jobb, új)  
            EndIf  
        EndIf  
    EndIf  
EndIf  
Vége(algoritmus)
```

Algorithm Helyez(csomagok, csomagokszáma, új):

// elhelyezzük az új méretű ládát a csomagokszáma sorszámu csomagba

helyet kérünk a p mutató által mutatható elem számára (new)

$p^{\text{méret}} \leftarrow \text{új}$

$p^{\text{köv}} \leftarrow \text{csomagok}[\text{csomagokszáma}]$

$\text{csomagok}[\text{csomagokszáma}] \leftarrow p$

EndAlgorithm

Következtetések

- 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.

6. Kalitkák

- Az állatkertben a papagájok 1-től n -ig számozott kalitkákbán élnek ($1 \leq n \leq 10\,000$).
- Egy adott pillanatban egy játékos majom kinyit minden kalitkát. Megijed a következményektől, visszatér az első kalitkához és bezár minden második kalitkát (így bezárja a $2, 4, 6, \dots$ sorszámúakat).
- A majomnak megtetszik ez a játék. Ezért újra elindul az elejéről és meglátogat minden harmadik kalitkát (vagyis a $3, 6, 9, \dots$ sorszámúakat) és bezárja a kalitkát, ha az nyitva van, illetve kinyitja, ha azt zárva találja. A negyedik bejáráskor meglátogat minden negyedik kalitkát, és hasonlóan jár el (megváltoztatva a meglátogatott kalitka állapotát).
- A majom megismétli a játékot, míg az utolsó bejáráskor (az n . bejárás) bezárja az n . kalitkát, ha ez nyitva van vagy kinyitja, ha zárva van

Megoldás

- A majom egy bejárás alkalmával, akkor és csakis akkor látogat meg egy kalitkát, **ha a kalitka sorszáma a bejárás sorszámanak többszöröse**.
- Következik, hogy **egy adott kalitka meglátogatásának darabszáma egyenlő a kalitka sorszáma különböző osztóinak darabszámával**.
- A kalitka akkor és csakis akkor marad nyitva, **ha a sorszámanak páratlan darabszámú osztója van**.
- A feladat megoldása visszavezethető azoknak a kalitkáknak a megszámlálására, amelyeknek **sorszáma négyzetszám**.
- Egy **n** természetes szám négyzetszám, ha $\sqrt{n} * \sqrt{n} = n$.
- **Az n -nél kisebb (vagy egyenlő) négyzetszámok darabszáma egyenlő $\lfloor \sqrt{n} \rfloor$ -nel**.
- Ez az érték meghatározható a **bináris keresés** módszerével, amikor keressük a $\lfloor \sqrt{n} \rfloor$ -et az **1, 2, ..., $n/2$** rendezett számsorozatban.

Algorithm kalitkák(n):

nyitvaSz = 0

For i = 1, (i * i <= n) **execute**

nyitvaSz = nyitvaSz + 1

EndFor

return nyitvaSz

EndAlgorithm

Algorithm kalitkak(n):

bal = 1

jobb = n DIV 2

While bal < jobb **execute**

k = (bal + jobb) DIV 2

If k * k >= n **then**

jobb = k

else

bal = k + 1

EndIf

EndWhile

If bal * bal <= n **then**

return bal

else

return bal - 1

EndIf

EndAlgorithm

7. Egészítsétek ki (Felvételi – 2019)

Adott az n elemű ($3 \leq n \leq 100$), növekvően rendezett x sorozat, amely 30 000-nél kisebb különböző természetes számokat tartalmaz.

A $\text{legközelebbi}(x, \text{bal}, \text{jobb}, \text{ér})$ algoritmus meghatározza az x sorozat legnagyobb értékű elemének pozícióját, amely a bal és jobb pozíciók között helyezkedik el ($1 \leq \text{bal} < \text{jobb} \leq n$) és, amelynek az értéke kisebb, vagy egyenlő ér -rel. Ha nem létezik ilyen elem, a $\text{legközelebbi}(x, \text{bal}, \text{jobb}, \text{ér})$ algoritmus 0-t térít vissza.

- A $\text{modulusz}(a)$ algoritmus az a egész szám abszolút-értékét téríti vissza.
- A $\text{számol}(n, x, \text{adottSz})$ algoritmus meghatározza azt az elemét az x sorozatnak, amely a legközelebb áll adottSz -hoz. Ha két elem azonosan közel van adottSz értékéhez, az algoritmus a nagyobb számot határozza meg.

Egészítsétek ki

- Legyen $n = 5$, $x = (5, 9, 11, 15, 99)$ és $adottSz = 12$. Állapítsátok meg melyik kifejezéssel helyettesíthető a „. . .” a $legközelebbi(x, bal, jobb, ér)$ algoritmusban, ahhoz, hogy a $sza\acute{m}ol(n, x, adottSz)$ algoritmus **11**-et térítsen vissza.
- A. $x[közép - 1] \leq ér \text{ AND } ér < x[közép]$
- B. $x[közép - 1] \leq ér \text{ OR } ér < x[közép]$
- C. $x[közép - 1] < ér \text{ AND } ér \leq x[közép]$
- D. $x[közép] \leq ér \text{ AND } ér < x[közép - 1]$

Algorithm legközelebbi(x, bal, jobb, ér)

If ér > x[jobb] **then**

return jobb

EndIf

If ér < x[bal] **then**

return bal - 1

EndIf

 közép ← (bal + jobb) DIV 2

If ... **then**

return közép - 1

else

If ér < x[közép] **then**

return legközelebbi(x, bal, közép - 1, ér)

else

return legközelebbi(x, közép + 1, jobb, ér)

EndIf

EndIf

EndAlgorithm

Algorithm számol(n, x, adottSz)

i \leftarrow legközelebbi(x, 1, n, adottSz)

If i = 0 **then**

return x[i + 1]

else

If modulusz(x[i]- adottSz) < modulusz(x[i + 1] - adottSz) **then**

return x[i]

else

return x[i + 1]

EndIf

EndIf

EndAlgorithm

Megoldás

- Az algoritmusban felismerjük a **bináris keresést**
- Különbség: itt nem egy adott értékű elemet keresünk, hanem egy másikat, amelynek az értéke a legközelebb áll az adott számhoz.
- A **szamol(n, x, adottSz)** algoritmus meghívja a **legközelebbi(x, bal, jobb, ér)** algoritmust, amely téríti az adott szám valamelyik szomszédos értékű elemének indexét.
- A továbbiakban, a térített indexnek megfelelő elem és az adott szám különbsége alapján eldönti, hogy melyik érték a legközelebbi, így ezzel a gonddal a **legközelebbi(x, bal, jobb, ér)** algoritmus nem foglalkozik.
- Következik, hogy azt az indexet térítjük, amelytől balra egy olyan elem található, amely kisebb, mint az adott szám, miközben jobbra egy nagyobb.

Megoldás

- Ugyanakkor, a hiányzó feltételt tartalmazó utasítás fölöslegesnek tűnik, mivel sejtjük, hogy az algoritmus működése során, vagy valamelyik önmeghívás, vagy a két leállási feltétel közül valamelyik hajtódik majd végre.
- Megvizsgáljuk, mi történne, ha a **közép** kiszámítása utáni **Ha** (a felkínált feltételektől függetlenül) és ennek az **If**-nek a **then** ága hiányozna.
- Lássuk, hogyan hajtódik végre az algoritmus, ha (**$n = 5$** , **$x = (5, 9, 11, 15, 99)$** és **$adottSz = 12$**):
- Az egyetlen eset, amelynek megfelelően a tárgyalt **Ha** utasítást végrehajtja a program, a **B.** eset. Ekkor a térített érték **2**, vagyis **$x_2 = 9$** . De mégsem ez lesz a végeredmény, erről gondoskodik a **számol($n, x, adottSz$)** alprogram.
- Mivel beláttuk, hogy a hiányzó feltételt tartalmazó utasítás fölösleges, válasz lehetne **A.**, **B.**, **C.**, **D.**, tehát, mindegyiket helyesnek nyilvánítjuk.

Megoldás

bal	jobb	közép	ér	Végrehajtott utasítás	Magyarázat
1	5	3	12	Ha $ér > x[jobb]$ akkor	Mivel $12 < 99$ a Ha-nak vége
				Ha $ér < x[bal]$ akkor	Mivel $12 > 5$ a Ha-nak vége
				Ha $ér < x[közép]$ akkor	Mivel $12 > 11$, a különben ág következik (újra hívás)
4	5	4		Ha $ér > x[jobb]$ akkor	Mivel $12 < 99$ a Ha-nak vége
				Ha $ér < x[bal]$ akkor	Mivel $12 < 15$, téríti bal – 1-et
					vagyis 3-at ($x_3 = 11$), vége

Bonyolultsági osztályok

- **P:** polinomidőben megoldható feladatok ($O(n^k)$).
- **NP:** polinomidőben ellenőrizhető feladatok (például a lineáris programozási problémák (George Dantzig: simplex módszer, amely exponenciális, de Leonid Khachian orosz matematikus felfedezett egy polinomiális algoritmust.)
- **co-NP:** olyan feladatok, amelyeknek esetében létezik példa, amelyre polinomidőben kimutatható, hogy az algoritmus eredmény NEM helyes.
- **NP-nehéz:** a megoldás algoritmus módosítható egy NP probléma megoldására (például az utazóügynök probléma)
- **NP-teljes:** az NP osztályhoz tartozó feladatok, amelyek egyben nem NP-nehezek
Létezik számukra exponenciális megoldás és nem tudjuk, hogy létezik-e polinomiális megoldás.