**A bináris keresés elvének alkalmazása**

**algoritmusok hatékonyságának növelése céljából**

**1. Algoritmusok bonyolultsága**

**1.1. Az algoritmus végrehajtásához szükséges idő**

A végrehajtáshoz szükséges idő az algoritmus „bonyolultságát” fejezi ki (időre vonatkozóan). Itt a *bonyo­lultság*nak semmi köze az algoritmus hosszához, illetve a szerkezetéhez.

Amikor meg szeretnénk határozni az algoritmus végrehajtásához szükséges időt, nem mérünk pontos időt, hiszen ez függ a számítógéptől, a programozási nyelvtől stb. Szükségünk van mégis 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. Például, ha egy sorozatban meg kell keresnünk az adott ***x*** értéket, akkor a művelet az összehasonlítás lesz. Ha össze kell szoroznunk két mátrixot, akkor két valós szám szorzása (vagy szorzások és összeadások száma) lesz a mérték­egység. Ha két nagyon nagy számot kell összeadnunk, akkor két számjegy feldolgozása lesz az elemi művelet.

Egy algoritmusnak az adott feladatot nemcsak egyetlen bemeneti adathalmaz esetében kell megoldania, hanem az összes lehetséges bemenetre. Ezeknek a bemeneteknek (általában) ismert a számosságuk (illetve a határok, amelyek között ez mozog). Az alapműveletet általában minden adatra elvégezzük, és így a bemenet mérete megadja a végrehajtások számát is. Például, ha egy ***n*** elemű sorozatban meg kell keresnünk egy bizo­nyos tulajdonsággal rendelkező elemet, akkor számíthatunk arra, hogy az algoritmus ***n*** összehasonlítást fog elvégezni, ha össze kell szoroznunk két mátrixot, akkor a műveletek számát a két mátrix méretének függvé­nyében számoljuk. Ha két nagyon nagy számot kell összeadnunk, akkor a két szám számjegyeinek száma lesz a méret. A gráfelméleti feladatokban a csomópontok száma vagy az élek száma jöhet számításba, esetleg mind a kettő.

Ugyanakkor vegyük észre, hogy a bemeneti adatok tulajdonságainak függvényében előfordulhat, hogy az algoritmus változó mennyiségű idő alatt hajtódik végre. Például a keresés leállhat az első összehasonlítás után, de ha a keresett elem nincs a sorozatban, akkor ***n*** összehasonlítást végzünk.

Tehát kifejezhetnénk a végrehajtási időt a végrehajtási idők átlagával, de ez nem biztos, hogy túl hasznos lenne, mivel a gyakorlatban elfordul, hogy bizonyos tulajdonságú bemeneti adatok gyakoribbak, mint mások. Ha ismerjük, vagy közelítőleg fel tudjuk becsülni a különböző tulajdonságú bemenetek előfordulásának valószínűségét, akkor az *átlagos végrehajtási idő* jobban kifejezi az algoritmus teljesítményét.

***Példa:*** Legyen ***a*** 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.

|  |
| --- |
| **Algoritmus** Keres(n, a, x):  i ← 1  **Amíg** i ≤ n **és** ai ≠ x) **végezd el:**  i ← i + 1  **vége(amíg)**  **Ha** i > n **akkor**  i ← 0  **vége(ha)**  **térítsd** i  **Vége(algoritmus)** |

**Alapművelet:** a sorozat egy elemének összehasonlítása ***x***-szel.

**Átlagos végrehajtási idő:** a bemenetek különbözőségét ***x*** előfordulásának pozíciója fejezi ki. Ha ***x*** megtalál­ható a sorozatban, ez lehet az első, a második, ..., az ***n***-edik elem. Tehát összesen ***n*** + 1 eset lehetséges.

Tehát, ha ***n***-nel jelöljük a bemenet méretét, a végrehajtás idejét ***n*** *függvénye*ként fejezzük ki. Ideális körül­mények között meghatározható egy matematikai képlet, amely kifejezi az algoritmus végrehajtásához szüksé­ges ***T*(*n*)** időt, ahol ***n*** a bemeneti adatok mérete. Sajnos, általában ez nem lehetséges. Ezért az esetek túlnyomó többségében arra kell szorítkoznunk, hogy a végrehajtási idő *nagyságrend*jét határozzuk meg. Ilyenkor, kere­sünk egy ***f*(*n*)** függvényt, amelynek segítségével kifejezzük az idő nagyságrendjét.

Bevezetünk egy további egyszerűsítést. Bennünket a futási idő *növekedési* *sebessége* vagy *növekedési rendje* érdekel, ezért a kiszámított képletnek csak a fő tagját vesszük figyelembe (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 állandó együtthatóját, mivel a nagy bemenetekre jellemző számítási haté­konyság meghatározásában az állandó tényezők kevésbé fontosak, mint a növekedés sebessége. Ezt a növeke­dési rendet a Θ(*n*) függvénnyel jelöljük.

Általában akkor tartunk egy algoritmust hatékonyabbnak egy másiknál, ha legrosszabb futási idejének alacsonyabb a növekedési rendje. Az állandó tényezők és alacsonyabb rendű tagok miatt ez az értékelés hibás lehet kis bemenetre. Elég nagy bemenetekre azonban például egy Θ(*n*2) algoritmus gyorsabban fut a leg­rosszabb esetben, mint egy Θ(*n*3) algoritmus.

Az algoritmus futási idejének növekedési rendje, (sebessége) az algoritmus hatékonyságának egyszerű leí­rását adja, valamint lehetővé teszi a szóba jövő algoritmusok relatív teljesítményének összehasonlítását is. Bár néha pontosan meg tudjuk határozni az algoritmus futási idejét, általában nem éri meg az erőfeszítést ez a különleges pontosság. Elég nagy bemeneti adatokra a pontos futási idő multiplikatív állandóinak és alacso­nyabb rendű tagjainak a hatása eltörpül a futási idő nagyságrendjéhez képest.

Ha a bemenet mérete elég nagy, akkor az algoritmus futási idejének csak a nagyságrendje lényeges, ezért az algoritmusnak az *aszimptotikus* hatékonyságát vizsgáljuk. Ekkor csak azzal foglalkozunk, hogy *a bemenet növekedésével miként növekszik az algoritmus futási ideje, ha n* → ∞. Általában az aszimptotikusan hatéko­nyabb algoritmus lesz a legjobb választás, kivéve a kis bemenetek esetét.

Egy adott ***g*(*n*)** függvény esetén **Θ(*g*(*n*))**-nel jelöljük a függvényeknek azt a halmazát, amelyre

Θ(*g*(*n*)) = {*f*(*n*): létezik *c*1, *c*2 és *n*0 pozitív állandó úgy, hogy 0 ≤ *c*1*g*(*n*) ≤ *f*(*n*) ≤ *c*2*g*(*n*) bármely *n* ≥ *n*0 esetén}

Más szóval az ***f*(*n*)** függvény hozzátartozik a **Θ(*g*(*n*))** halmazhoz, ha léteznek ***c*1** és ***c*2** pozitív állandók úgy, hogy ***f*(*n*)** – elég nagy ***n***-re – „beszorítható” ***c*1*g*(*n*)** és ***c*2*g*(*n*)** közé. Bár **Θ(*g*(*n*))** egy halmaz, mégis úgy írjuk, hogy „***f*(*n*) = Θ(*g*(*n*))**”, ha azt akarjuk jelezni, hogy ***f*(*n*)** eleme **Θ(*g*(*n*))**-nek.

Más szóval, minden ***n*** ≥ ***n*0** esetén az ***f*(*n*)** függvény – egy állandó szorzótényezőtől eltekintve – egyenlő ***g*(*n*)**-nel. Ekkor azt mondjuk, hogy ***g*(*n*)** *aszimptotikusan éles korlátja f*(*n*)-nek.

Mivel bármely állandó egy nulla fokú polinom, ezért bármelyik konstans függvényre fennáll a **Θ(*n*0)** vagy **Θ(1)** becslés. Az utóbbi jelölés kissé pontatlan, hiszen nem nyilvánvaló, hogy ekkor mely változó tart a végtelenhez.

A **Θ**-jelölés aszimptotikus alsó és felső korlátot ad a függvényre. Amikor csak az *aszimptotikus felső korlát* jön szóba, akkor az ***O***-jelölést használjuk. Egy adott ***g*(*n*)** függvény esetén ***O*(*g*(*n*))**-nel jelöljük a függvények­nek azt a halmazát, amelyre

*O*(*g*(*n*)) = { *f*(*n*): létezik *c* és *n*0 pozitív állandó úgy, hogy 0 ≤ *f*(*n*) ≤ *cg*(*n*) bármely *n* ≥ *n*0 esetén }.

Az ***O***-jelölés használatával gyakran úgy is meg tudjuk határozni az algoritmus futási idejét, hogy pusztán az algoritmus egészének a szerkezetét vizsgáljuk. Mivel az ***O***-jelölés egy felső korlátot határoz meg, ezért amikor egy algoritmus legrosszabb futási idejére használjuk, akkor ezzel minden bemenet esetében érvényes felső korlátot adunk az algoritmus futási idejére.

**1.2. Az algoritmus által feldolgozott adatok számára szükséges memória mérete**

Az algoritmusok által feldolgozott adatok számára lefoglalt memória mérete ugyanúgy sajátos tulajdonság, mint a végrehajtási idő. Sok esetben a feladat megoldását biztosító program, a bemeneti és kimeneti adatokon kívül, munka közben ideiglenesen létrehozott adatszerkezetekkel is dolgozik. Ha egy algoritmus a végrehajtása közben csak konstans méretű plusz memóriát vesz igénybe, azt mondjuk, hogy *helyben* rendez. Összehason­lítva ezt egy olyan algoritmussal, amely létrehoz egy másolatot a bemeneti sorozatról, természetesen az előbbit hatékonyabbnak tekintjük a tárfelhasználás szempontjából.

**1.3. Algoritmusok egyszerűsége**

Algoritmusok elemzésekor az egyszerűség nem feltétlenül mérvadó tulajdonság. Ha egy algoritmust a legkézenfekvőbb ötlet alapján tervezünk meg (*brute force*), naiv algoritmusról beszélünk. Ezek sokszor, de nem mindig, fölöslegesen terhelik a számítógép erőforrásait. Ugyanakkor egy egyszerű algoritmust könnyeb­ben írunk meg, olvashatóbb, könnyebben javítjuk és könnyebben bizonyítjuk a helyességét. De ne felejtsük el, hogy egy naiv algoritmus nem biztos, hogy nem hatékony.

**1.4. Algoritmusok optimalitása**

Feltételezzük, hogy egy adott feladatnak megvalósítottuk az algoritmusát, és kiszámítottuk a végrehajtásá­hoz szükséges ***T*(*n*)** időt. A következőkben azon fogunk elgondolkozni, hogy vajon létezik-e jobb (kisebb vég­rehajtási idejű) algoritmus, mint amit megterveztünk? Egy algoritmus optimalitásának bizonyítása nehéz fela­dat, elsősorban azért, mert számításba kell vennünk *minden* lehetséges algoritmust, és ki kell mutatnunk, hogy ezeknek a futási ideje nagyobb mint a tárgyalt algoritmusé.

**2. Feladatok**

1. **Keresztmetszet**

Adott két sorozat, amelyeknek elemei *különböző* természetes számok: az ***a*** sorozat elemeinek száma ***n*** (0 <***n*** ≤ 10 000), a ***b*** sorozat elemeinek száma ***m*** (0 < ***m*** ≤ 10 000) és *növekvően rendezett*. Határozzuk meg azt a ***c*** sorozatot, amelynek ***k*** (0 < ***k***≤ 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).

**Megoldás**

Adott két sorozat, amelyek nem rendezettek, ugyanakkor halmazokat tárolnak, mivel az elemek különbö­zők. Természetesen, alkalmazhatnánk a *keresztmetszet programozási tétel* néven ismert algoritmust, de így nem használnánk ki a ***b*** sorozat tulajdonságát, vagyis azt, hogy növekvően rendezett. Ebből azonnal követke­zik, hogy ez a megközelítés nem vezet optimális megoldáshoz.

Mivel több lehetséges és hatékony megközelítés is számításba jöhet, ezek közül bemutatunk kettőt.

1. **Megoldás:** A sorozatokat az adott formában dolgozzuk fel és az ***a*** sorozat aktuális értékét a bináris keresés algoritmusával keressük a ***b*** sorozatban. Ha megtaláltuk, elhelyezzük a *c* sorozatba.
2. **Megoldás:** Előbb rendezzük az ***a*** sorozatot, majd egy, az összefésüléshez hasonló algoritmust alkalmazunk (a leghatékonyabb rendező algoritmus a lineáris rendezés).

|  |  |
| --- | --- |
| **Algoritmus** keresztmetszet\_1(n, a, m, b, k, c):  k = 0  **Minden** i = 1, n **végezd el:**  megvan = hamis  bal = 1  jobb = m  **Amíg** **nem** megvan **és** bal ≤ jobb **végezd el:**  közép = (bal + jobb) **DIV** 2  **Ha** a[i] = b[közép] **akkor**  k = k + 1  c[k] = a[i]  megvan = igaz  **különben**  **Ha** a[i] < b[közép])  jobb = közép - 1  **különben**  bal = közép + 1  **vége(ha)**  **vége(ha)**  **vége(amíg)**  **vége(minden)**  **Vége(algoritmus)** | **Algoritmus** keresztmetszet\_2(n, a, m, b, k, c):  rendez(n, a)  i = 1  j = 1  k = 0  a[n + 1] = b[m] + 1  b[m + 1] = a[n] + 1  **Amíg** i ≤ n **és** j ≤ m **végezd el:**  **Ha** a[i] < b[j] **akkor**  i = i + 1  **különben**  **Ha** a[i] > b[j] **akkor**  j = j + 1  **különben**  k = k + 1  c[k] = a[i]  i = i + 1  j = j + 1  **vége(ha)**  **vége(ha)**  **vége(amíg)**  **Vége(algoritmus)** |

1. ***S* összegű elemek kiválasztása**

Legyen egy ***n*** elemű (3 ≤ ***n*** ≤ 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!

**Megoldás**

Részletösszegeket kell számítanunk: három elem összegét hasonlítjuk ***S***-sel. Fennáll a veszély, hogy valaki – tévedésből – exponenciális bonyolultságú algoritmust (*backtracking*) tervez, miközben a feladat megoldható három egymásba ágyazott **Minden** ciklussal is. Az ***n*** nagy értéke miatt (100 000), ez a megoldás időigényes; hiába ügyeskednénk **Amíg** ciklusokkal, illetve a lépésszámok csökkentésével, a bonyolultság továbbra is ***O*(*n*3)**lenne).

* Ha a megoldásba beépítjük a bináris keresést, a bonyolultság ***O*(*n*2log *n*)**lesz:
* Ahhoz, hogy alkalmazhassuk, előbb ***rendezzük*** az adott sorozatot (ennek bonyolultsága pl. ***O*(*n*2)**).
* Két **Amíg** ciklussal kiválasztunk a sorozatból két elemet (legyen ezeknek indexe ***n*1** és ***n*2**). Ennek bonyolultsága ***O*(*n*2)**.
* ***Megkeressük*** az ***S – an*1 *– an*2**értéket *a* ***bináris keresést alkalmazva***. Ennek bonyolultsága ***O*(*log n*)**, de ***n*2** -szer.
* A megoldást tovább javítjuk (például, ha ***an*1** értéke meghaladja ***S***-t, kilépünk az első **Amíg**-ból stb.).
* Tehát az algoritmus bonyolultsága:

***O*(*n*2) + *O*(*n*2 *· log n*) = *O*(*n*2 *· log n*)**

|  |
| --- |
| **Algoritmus** Generál(a, n, S):  i ← 1  **Amíg** i < n-1 **és** a[i] < S **végezd el:**  j ← i + 1  **Amíg** 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, }  { akkor k értéke egy valid index, különben k értéke 0 }  **Ha** k ≠ 0 **akkor**  **Ki:** i, j, k  **vége(ha)**  j ← j + 1  **vége(amíg)**  i ← i + 1  **vége(amíg)**  **Vége(algoritmus)** |

**3. Négyzetszámok darabszáma**

Számoljuk meg egy***n*** (1≤ ***n***≤ 1.000.000) elemű sorozat négyzetszámait!A számok nem nagyobbak egy milliónál.

**Megoldás**

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

|  |
| --- |
| **Algoritmus** Számol\_1(n, a, p):  p ← 0  **Minden** i = 1, n **végezd el:**  **Ha** négyzetgyök(a[i]) = [négyzetgyök(a[i])] **akkor**  p ← p + 1  **vége(ha)**  **vége(minden)**  **Vége(algoritmus)** |

1. Tudjuk, hogy a négyzetgyök kiszámítása elkerülendő, mivel időigényesebb, mint gondolnánk. Főleg, ha sokszor kell alkalmaznunk.

Először is az **Amíg** feltételéből „kiemeljük” az ***ai****-t* ***szám****-ba*, mivel ***ai***-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.

1. Azt hogy egy adott szám négyzetszám-e, hatékonyabban is lehetséges ellenőriznünk:

|  |
| --- |
| **Algoritmus** Számol\_2(n, a, p):  p ← 0 { p a négyzetszámokat számlálja }  **Minden** i = 1, n **végezd el:**  k ← 0 { a k négyzetét hasonlítjuk a[i]-val }  szám ← a[i] { az a[i]-t „megkeresni” nehezebb }  **Amíg** k \* k < szám **végezd el:**  k ← k + 1  **vége(amíg)**  **Ha** k \* k = szám **akkor**  p ← p + 1  **vége(ha)**  **vége(minden)**  **Vége(algoritmus)** |

Ha implementáljuk az első, valamint a második algoritmust, és megmérjük a végrehajtási időt, látni fogjuk, hogy ***az utóbbi sokkal kevesebb időt igényel, m az első!***

De még nem alkalmaztuk a bináris keresés elvét!

Vegyük észre, hogy ***nem érdemes minden k*** (***k \* k < szám***) értékre elvégezni a vizsgálatot, hiszen tulaj­donképpen egy ***rendezett halmazban keresünk! k*** lehetséges értékei az {1, 2, …, 1000}halmazhoz tartozhat­nak, mivel az ellenőrizendő számok kisebbek vagy egyenlők 1 000 000-val.

|  |
| --- |
| **Algoritmus** Számol\_3(n, a, p):  p ← 0  **Minden** i = 1, n **végezd el:**  szám ← a[i]  eleje ← 0  vége ← 1000 { a legnagyobb érték 1 000 000 }  **Amíg** eleje < vége **végezd el:**  k ← (eleje + vége)/2  **Ha** k \* k ≥ szám **akkor**  vége ← k  **különben**  eleje ← k + 1  **vége(ha)**  **vége(amíg)**  **Ha** eleje \* eleje = szám **akkor**  p ← p + 1  **vége(ha)**  **vége(minden)**  **Vége(algoritmus)** |

**4. Síkmértan**

*h*1

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 éri egymást egy pontban, amely valahol a két fal között található.

*h*2

Számítsuk ki ***t***-t ***h*1**, ***h*2** és ***m*** ismeretében (megengedett hibalehetőség ***10*-5**).

*m*

*t*

***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, m 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 ***Min*(*h*1, *h*2)** között keressük.
* Miután van egy javaslat ***t***-re, ***h*1** és ***h*2** értékeivel kiszámítjuk az érkezési pont magasságát síkmértani ismeretekkel.
* Ha a kiszámított magasság nagyobb m az adott ***m*** akkor növeljük ***t***-t, különben csökkentjük.

|  |
| --- |
| Algoritmus **deszkák(m, h1, h2):**  **megvan ← hamis**  **Ha** h1 > h2 **akkor**  t ← h2  **különben**  t ← h1  **vége(ha)**  min ← 0  max ← t  **Amíg** **nem** megvan **végezd el:**  t ← (min + max) / 2  számol(h1, h2, t, sz)  **Ha** abs(sz - m) ≤ 0.00001 **akkor**  megvan ← igaz  **különben**  **Ha** sz > m **akkor**  min ← t  **különben**  max ← t  **vége(ha)**  **vége(ha)**  **vége(amíg)**  **térítsd** t  **Vége(algoritmus)** |

**5. 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.

Határozzuk meg a ládák számának (0 ≤ ***n*** ≤ 15 000) és méreteiknek (1 ≤ ***ládaMéret*** ≤ 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.
* 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 legtöbb 15 000.
  + Ha a ládák ***növekvő*** sorrendben érkeznek, akkor minden érkező láda új csomagnak felel meg, tehát legtöbb 15 000 ***darab egy elemű részsorozatunk*** lesz.
* A fentieket figyelembe véve egy 15 000 × 15 000méretű 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.

|  |
| --- |
| **Algoritmus** Keres(bal, jobb, új):  **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** { megvan }  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)**  **Algoritmus** Helyez(csomagok, csomagokszáma, új):  { elhelyezzük az új méretű ládát a csomagokszáma sorszámú csomagba }  *helyet kérünk a p mutató által mutatható elem számára*  p^.méret ← új  p^.köv ← csomagok[csomagokszáma]  csomagok[csomagokszáma] ← p  **Vége(algoritmus)** |

**6. Kalitkák**

Az állatkertben a papagájok 1-től ***n***-ig számozott kalitkákban élnek (1 ≤ ***n*** ≤ 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, ... sorszámúakat). A majomnak megtetszik ez a játék. Ezért újra elindul az elejéről és meglátogat minden harmadik kalitkát (vagyis a 3, 6, 9, ... 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.

**Követelmények:**

1. Hány kalitka marad nyitva az utolsó bejárás után, ha ***n*** = 10? (2 pont)
2. Mely sorszámú kalitkák maradnak nyitva az utolsó bejárás után, ha ***n*** = 10? (2 pont)
3. Összesen hányszor látogatja meg a majom a ***k*** sorszámú kalitkát (1 ≤ ***k*** ≤ ***n***) az ***n*** bejárás során? Indokoljátok meg a választ. (4 pont)
4. Mi annak szükséges és elégséges feltétele, hogy a ***k*** sorszámú kalitka (1 ≤ ***k*** ≤ ***n***) nyitva maradjon az ***n*** kalitka utolsó bejárása után? Indokoljátok meg a választ. (4 pont)
5. Hány kalitka marad nyitva az ***n*** kalitka utolsó bejárása után? Indokoljátok meg a választ. (4 pont)
6. Írjatok algoritmust, amely kiszámítja az utolsó bejárás után *nyitva maradt kalitkák számát* (***nyitvaSz***). Az algoritmus bemeneti paramétere a kalitkák ***n*** (1 ≤ ***n*** ≤ 10 000) száma, kimeneti paramétere a ***nyitvaSz*** szám. (14 pont)

**1. *Példa:*** ha ***n*** = 5, akkor ***nyitvaSz*** = 2 (nyitva marad az 1-es és a 4-es sorszámú kalitka).

**2*. Példa:*** ha ***n*** = 12, akkor ***nyitvaSz*** = 3.

**Megoldás**

A majom egy bejárás alkalmával, akkor és csakis akkor látogat meg egy kalitkát, ha a kalitka sorszáma a bejárás sorszámának 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 sorszámának *páratlan* darabszámú osztója van.

Lássuk, mi történik, ha ***n*** = 5. Jelöljük a ***bejárásSorszáma*** változóval a kalitkasor aktuális bejárásának sorszámát. Ha a kalitkák száma ***n***, a ***bejárásSorszáma*** az {1, 2, ..., ***n***} halmazból kap értéket.

Kövessük a példát:

1. Az első bejáráskor (***bejárásSorszáma*** = 1) a majom kinyitja az 1, 2, 3, 4, 5 sorszámú kalitkákat. Tehát most nyitva vannak az **1 2 3 4 5** sorszámú kalitkák. Észrevesszük, hogy az 1, 2, 3, 4, 5 számoknak ***egyetlen*** osztójuk van az {1, ..., ***bejárásSorszáma***} halmazban, ami most {1}. Így az egyetlen osztó az 1.
2. A második bejárás során (***bejárásSorszáma*** = 2) a majom bezár minden második kalitkát, vagyis a 2 és a 4 sorszámúakat. Nyitva maradnak az **1 3 5** sorszámúak, ahol mindegyiknek ***egyetlen*** osztója van az {1, ..., ***bejárásSorszáma***} = {1, 2} halmazban éspedig az 1, miközben a 2-es és a 4-es sorszámúak zárva vannak. Ennek a két számnak egyenként ***két*** osztója van az {1, 2} halmazban, éspedig az 1 és a 2.
3. A harmadik bejárás során (***bejárásSorszáma*** = 3) a majom kinyit minden harmadik kalitkát, ha ez zárva van és bezár minden harmadik kalitkát, ha ez nyitva van. Így bezárja a 3. kalitkát, miközben nyitva maradnak: **1** és **5**. Az 1-nek és az 5-nek ***egyetlen*** osztója van az {1, ..., ***bejárásSorszáma***} = {1, 2, 3} halmazban, miközben a 2, 3, 4 sorszámú kalitkák zárva vannak. Észrevesszük, hogy ezeknek a számoknak egyenként ***két*** osztója van az {1, 2, 3} halmazban: 2 osztói az 1 és a 2, a 3 osztói az 1 és a 3, a 4 osztói pedig az 1 és a 2).
4. A negyedik bejárás során (***bejárásSorszáma*** = 4) a majom kinyitja a 4-es sorszámú kalitkát (zárva volt). Most nyitva vannak az **1**, **4** és **5** sorszámú kalitkák, ahol 1-nek egy osztója van az {1, 2, 3, 4} halmazban, 4-nek három osztója van (1, 2 és 4), 5-nek pedig egy osztója van (***az osztók száma páratlan***). A 2-es és 3-as kalitkák zárva vannak (2-nek és 3-nak két-két osztója van a megfelelő halmazban).
5. Az ötödik bejárás (***bejárásSorszáma*** = 5) végén a majom bezárja az ötödik kalitkát. Nyitva vannak az **1** és a **4** (***páratlan*** számú osztóik vannak az {1, 2, 3, 4, 5} halmazban) és zárva vannak a 2-es, 3-as és 5-ös kalitkák (ahol 2-nek, 3-nak és 5-nek ***két-két*** osztója van).

**Állítás**

Egy természetes számnakakkor és csakis akkor van páratlan számú osztója, ha ***négyzetszám***.

**Bizonyítás**

Ha ***d*** az ***m*** természetes szám osztója, akkor ***m*** / ***d*** is osztója ***m***-nek, mivel:

***d*** \* (***m*** / ***d***) = ***m*** (1)

Következik, hogy egy természetes szám osztói párba állíthatók, vagyis úgy gondolhatnánk, hogy bármely természetes számnak páros darabszámú osztója van. Másfelől, ha van olyan pár, amelyben a két szám egyenlő, következik, hogy az ***m*** szám négyzetszám, mivel, ha behelyettesítjük a ***d***-t (***m*** / ***d***)-vel az (1)-es egyenletben, kapjuk, hogy ***d***\* ***d*** = ***m***. Ezekben az esetekben, csökkentjük a páros számú osztók darabszámát, amiből következik, hogy az osztók száma páratlan.

Ha ***d*** ≠ ***m*** / ***d***, marad a páros darabszámú osztószám (a nem négyzetszámok esetében).

Ilyen körülmények között a feladat megoldása visszavezethető azoknak a kalitkáknak a megszámolására, amelyeknek sorszáma négyzetszám. Egy ***n*** természetes szám négyzetszám, ha \* = ***n***. Másfelől *az* ***n****-nél kisebb (vagy egyenlő) négyzetszámok darabszáma egyenlő* ⎣⎦-nel. Ez az érték meghatározható a *bináris keresés módszerével*, amikor keressük a ⎣-et az 1, 2, ..., ***n***/2 rendezett számsorozatban.

A kalitkákat „zárhatjuk/nyithatjuk” a következő táblázatban:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***Bejárássorszáma*** | ***Kalitkák*** | | | | | ***Osztók halmaza*** |
| 1 | 2 | 3 | 4 | 5 |  |
| kiindulópont | zárva | zárva | zárva | zárva | zárva |  |
| 1 | nyitva | nyitva | nyitva | nyitva | nyitva | 1 |
| 2 | nyitva | zárva | nyitva | zárva | nyitva | 1, 2 |
| 3 | nyitva | zárva | zárva | zárva | nyitva | 1, 2, 3 |
| 4 | nyitva | zárva | zárva | nyitva | nyitva | 1, 2, 3, 4 |
| 5 | **nyitva** | zárva | zárva | **nyitva** | zárva | 1, 2, 3, 4, 5 |

Lássuk a válaszokat a feladatban feltett kérdésekre:

1. Ha ***n*** = 10, nyitva marad 3 kalitka.
2. Ha ***n*** = 10, nyitva maradnak az 1, 4, 9 sorszámú kalitkák.
3. A ***k*** sorszámú kalitkát összesen annyiszor látogatja meg a majom az ***n*** bejárás során, ahány osztója van ***k***-nak az {1, 2, ..., ***n***} halmazban.
4. Egy ***k*** sorszámú kalitka akkor és csakis akkor marad nyitva az ***n*** kalitka utolsó bejárása után, ha:

* ***k***-nak *páratlan darabszámú osztója* van az {1, 2, ..., ***n***} halmazban

vagy – másképp kifejezve:

* ***k*** *négyzetszám*, mivel minden olyan szám, amelynek páratlan darabszámú osztója van négyzetszám.

1. A nyitva maradt kalitkák darabszáma egyenlő-nel, mivel minden olyan szám, amelynek páratlan darabszámú osztója van, négyzetszám. Ugyanakkor az ***n***-nél kisebb négyzetszámok darabszáma:  .
2. Az algoritmus több elképzelés alapján is megtervezhető. A következőkben bemutatjuk az egyik legkézenfekvőbbet (**V1**), majd egy hatékonyabbat (**V2**).
3. a nyitva maradt kalitkák számát meghatározzuk az ***n***-nél kisebb négyzetszámok megszámlálása által (egyetlen ismétlő struktúrával);
4. a nyitva maradt kalitkák számát a  alapján határozzuk meg; az optimális algoritmus ezt az értéket a bináris keresés algoritmusával számítja ki:

|  |  |
| --- | --- |
| kalitkakV1( n){  nyitvaSz = 0  **Minden** ( i = 1 i \* i <= n i++){  nyitvaSz++  }  **térítsd** nyitvaSz  } | kalitkakV2( n){  bal = 1  jobb = n / 2  **Amíg** (bal < jobb){  k = ((bal + jobb) / 2)  **Ha** (k \* k >= n)  jobb = k  **különben**  bal = k + 1  }  **Ha** (bal \* bal <= n)  **térítsd** bal  **különben**  **térítsd** bal - 1  } |

1. ha a fenti ötletek közül egyik sem „ugrik be”, szimulálhatjuk a majom tevékenységét: bejárunk minden kalitkát ***n***-szer, kinyitjuk/bezárjuk a kalitkát a bejárás során, majd a nyitva maradt kalitkákat megszámláljuk.

|  |
| --- |
| kalitkakV3( n){  **const** MAX = 10001  **bool** ajtok[MAX]  **Minden** ( i = 1 i <= n i++)  ajtok[i] = **igaz**  **Minden** ( k = 2 k <= n k++){  **Minden** ( i = k i <= n i = i + k){  **Ha** (ajtok[i])  ajtok[i] = **hamis**  **különben**  ajtok[i] = **igaz**  }  }  nyitvaSz = 0  **Minden** ( i = 1 i <= n i++)  **Ha** (ajtok[i])  nyitvaSz++  **térítsd** nyitvaSz  } |

**7. Egészítsétek ki**

Adott az ***n*** elemű (3 ≤ ***n*** ≤ 100), növekvően rendezett ***x*** sorozat, amely 30 000-nél kisebb különböző természetes számokat tartalmaz. A legközelebbi(x, bal, jobb, ér) algoritmus meghatározza az ***x*** sorozat legnagyobb értékű elemének pozícióját, amely a ***bal*** és ***jobb*** pozíciók között helyezkedik el (1 ≤ ***bal*** < ***jobb*** ≤ ***n***) és, amelynek az értéke kisebb, vagy egyenlő ***ér***-rel. Ha nem létezik ilyen elem, a legközelebbi(x, bal, jobb, ér) algoritmus 0-t térít vissza.

A modulusz(a) algoritmus az ***a*** egész szám abszolút-értékét téríti vissza.

A számol(n, x, 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.

Legyen ***n*** = 5, ***x*** = (5, 9, 11, 15, 99) és ***adottSz*** = 12. Állapítsátok meg melyik kHaejezéssel helyettesíthető a „…” a legközelebbi(x, bal, jobb, ér) algoritmusban, ahhoz, hogy a számol(n, x, adottSz) algoritmus 11-et térítsen vissza.

|  |
| --- |
| **Algoritmus** legközelebbi(x, bal, jobb, ér)  **Ha** ér > x[jobb] **akkor**  **térítsd** jobb  **vége(ha)**  **Ha** ér < x[bal] **akkor**  **térítsd** bal - 1  **vége(ha)**  közép ← (bal + jobb) **DIV** 2  **Ha** ... **akkor**  **térítsd** közép - 1  **különben**  **Ha** ér < x[közép] **akkor**  **térítsd** legközelebbi(x, bal, közép - 1, ér)  **különben**  **térítsd** legközelebbi(x, közép + 1, jobb, ér)  **vége(ha)**  **vége(ha)**  **Vége(algoritmus)** |
| **Algoritmus** számol(n, x, adottSz)  i ← legközelebbi(x, 1, n, adottSz)  **Ha** i = 0 **akkor**  **térítsd** x[i + 1]  **különben**  **Ha** modulusz(x[i]- adottSz) < modulusz(x[i + 1] - adottSz) **akkor**  **térítsd** x[i]  **különben**  **térítsd** x[i + 1]  **vége(ha)**  **vége(ha)**  **Vége(algoritmus)** |

1. x[közép - 1] <= ér **és** ér < x[közép]
2. x[közép - 1] <= ér **vagy** ér < x[közép]
3. x[közép - 1] < ér **és** ér <= x[közép]
4. x[közép] <= ér **és** ér < x[közép - 1]

**Megoldás**

Az algoritmusban felismerjük a bináris keresést, azzal a különbséggel, hogy itt nem egy adott értékű elemet keresünk, hanem egy másikat, amelynek az értéke a legközelebb áll az adott számhoz. A számol(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, m az adott szám, miközben jobbra egy nagyobb.

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 a **Ha**-nak az **akkor** ága hiányozna. Lássuk, hogyan hajtódik végre az algoritmus a fenti példa esetében (***n*** = 5, ***x*** = (5, 9, 11, 15, 99) és ***adottSz*** = 12):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***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 |
|  |  |  |  |  |

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.

**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.