A lépésszám becslése, a nagy Ordó jelölés

Csima Judit
BME SZIT
csima@cs.bme.hu

2019. szeptember 18.

Kiválasztásos rendezés

Láttuk, hogy az első előadás 6. algoritmusa megtalálja egy tömb legkisebb elemét. Ha nem csak magára a legkisebb értékre vagyunk kíváncsiak, hanem arra is, hogy ez az érték hol, melyik cellában, azaz hányas indexnél található, akkor a következőképpen kell a pszeudokódot módosítani:

```
min  := A[0]
min_hely: = 0
ciklus i = 1-tól n-1-ig:
    ha A[i] < min:
        min := A[i]
        min_hely:= i
ciklus vége
return min és return min_hely
```

Helyesség Ez az algoritmus azért helyes, mert amint elérjük a tömb legkisebb elemét, akkor a min változóba ez a legkisebb érték bekerül, a *min_hely* változó pedig azt az indexet fogja felvenni, ahol éppen tartunk, vagyis a legkisebb elem cellájának indexét, később pedig soha nem írjuk már át ezeket a változókat, hiszen soha nem fogunk a legkisebb értéknél kisebbel találkozni.

A kiválasztásos rendezés leírása

A fenti algoritmuson alapul egy ismert rendező algoritmus, melynek neve kiválasztásos rendezés.

Itt a feladat a következő: Az input egy $n > 0$ hosszú, csupa különböző egész számot tartalmazó A tömb, a feladat pedig az, hogy módosítsuk az A tömböt úgy, hogy az az elemeket rendezetten tartalmazza, azaz a kimeneti A tömbre igaz, hogy $A[i] < A[i+1]$ minden $0 \leq i \leq n-2$ esetén.

Kiválasztásos rendezés szövegesen

Tegyük fel, hogy $n > 1$, azaz legalább két elemű a tömb (az $n = 0$ és $n = 1$ esetekben nem kell tennünk semmit, az elvárt kimenet megegyezik a bemenettel).

1. fázis A fenti algoritmussal keressük meg az $A[0 : n-1]$ tömb legkisebb elemét (és ennek helyét), majd a megtalált legkisebb elemet cseréljük fel $A[0]$-lal (azaz egy cserével vigyük a

legkisebb elemet a tömb elejére).

2. fázis A fenti algoritmussal keressük meg az 1. fázisban kapott tömb $A[1 : n-1]$ résztömbjének legkisebb elemét (és ennek helyét), majd a megtalált legkisebb elemet cseréljük fel $A[1]$-gyel (azaz egy cserével vigyük a legkisebb elemet a tömb 1-es indexű cellájába, azaz a második helyre a tömbben).

3. fázis A fenti algoritmussal keressük meg a 2. fázisban kapott tömb $A[2 : n-1]$ résztömbjének legkisebb elemét (és ennek helyét), majd a megtalált legkisebb elemet cseréljük fel $A[2]$-gyel (azaz egy cserével vigyük a legkisebb elemet a tömb 2-es indexű cellájába, azaz a harmadik helyre a tömbben).

$\vdots$

(n-1). fázis A fenti algoritmussal keressük meg az előző fázisban kapott tömb $A[n-2 : n-1]$ résztömbjének legkisebb elemét (és ennek helyét), majd a megtalált legkisebb elemet cseréljük fel $A[n-2]$-vel (azaz egy cserével vigyük a legkisebb elemet a tömb utolsó előtti cellájába).

Példa

Input tömb: $8, 5, -4, 0, 10$

Az 1. fázis során a megtalált minimum a -4, ennek indexe 2, az 1. fázis végén a kapott tömb: -4, 5, 8, 0, 10.

A 2. fázis során a megtalált minimum a 0, ennek indexe 3, a 2. fázis végén a kapott tömb: -4, 0, 8, 5, 10.

A 3. fázis során a megtalált minimum a 5, ennek indexe 3, a 3. fázis végén a kapott tömb: -4, 0, 5, 8, 10.

A 4. fázis (egyben utolsó fázis) során a megtalált minimum a 8, ennek indexe 3, a 4. fázis végén a kapott tömb: -4, 0, 5, 8, 10 (itt igazából nem történt valódi csere, mert a 8-as szám jó helyen volt már).

A kiválasztásos rendezés megtekinthető animált formában itt: https://visualgo.net/bn/sorting (A felső sorban levő menüben a SEL (selection sort) lehetőséget kell választani.)

Helyesség

Ez az eljárás helyes, mert az 1. fázis után az input tömb legkisebb eleme az első cellába kerül és később innen soha el nem mozdítjuk. A 2. fázis után a maradék elemek közül a legkisebb, vagyis a tömb 2. legkisebb eleme a 2. cellába kerül és innen soha el nem mozdítjuk később. Általában is igaz, hogy a k. fázis után a tömb legkisebb k eleme már a helyén lesz és soha nem mozdul el onnan később, így az eljárás végére mindenki a helyére fog kerülni.

Megjegyzés a csere megvalósításáról

Amikor azt mondjuk, hogy az i és j indexű cellák elemeit felcseréljük, akkor igazából az alábbi történik:

```
temp := A[i]
A[i] := A[j]
A[j] : = temp
```

Azaz az i indexű cella értékét egy *temp* (mert temporary) nevű változóban eltároljuk és innen írjuk majd vissza ezt az értéket a j indexű cellába, miután az i indexű cellábe beleraktuk a j indexű cella értékét.

Gondoljuk meg, hogy ez az eljárás akkor is megfelelő, ha $i = j$.

Kiválasztásos rendezés pszeudokóddal

Nézzük meg, hogy mit jelentenek az egyes fázisok pszeudokóddal leírva:

1. fázis

```
min  := A[0]
min_hely: = 0
ciklus i = 1-tól n-1-ig:
    ha A[i] < min:
        min := A[i]
        min_hely:= i
ciklus vége
csere A[0] és A[min_hely]
```

2. fázis

```
min  := A[1]
min_hely: = 1
ciklus i = 2-tól n-1-ig:
    ha A[i] < min:
        min := A[i]
        min_hely:= i
ciklus vége
csere A[1] és A[min_hely]
```

3. fázis

```
min  := A[2]
min_hely: = 2
ciklus i = 3-tól n-1-ig:
    ha A[i] < min:
        min := A[i]
        min_hely:= i
ciklus vége
csere A[2] és A[min_hely]
```

⋮

(n-1). fázis

```
min  := A[n-2]
min_hely: = n-2
ciklus i = n-1-tól n-1-ig:
    ha A[i] < min:
        min := A[i]
        min_hely:= i
ciklus vége
csere A[n-2] és A[min_hely]
```

Ez így még nem egy jól megírt pszeudokód, hiszen a leírásának a hossza attól függ, hogy mekkora n értéke, de ha két egymásba ágyazott ciklust használunk, akkor már megkapjuk az előbbi szöveges leíráshoz tartozó pszeudokódot. A külső ciklus fogja szabályozni azt, hogy melyik résztömbben dolgozunk (hányas cellától kezdve keressük a minimumot), a belső ciklus pedig az így meghatározott résztömbben keresi meg és mozgatja a résztömb elejére a legkisebb értéket.

```
ciklus j = 0-tól (n-2)-ig:
    min:= A[j]
    min_hely := j
    ciklus i = j+1-tól (n-1)-ig:
        ha A[i] < min:
             min:= A[i]
             min_hely:= i
    ciklus vége
    csere A[j] és A[min_hely]
ciklus vége
```

Helyesség

Mivel ez a pszeudokód az előző szöveges leírást valósítja meg (amiről már láttuk, hogy jó), ezért helyes eredményt ad.

A kiválasztásos rendezés lépésszáma, motiváció egy fontos, új definícióra

Adjunk felső becslést a kiválasztásos rendezés lépésszámára az input méretének függvényében! Az input mérete most a tömb hossza, azaz n, lépésnek pedig az összehasonlítás, a csere és az értékadás számít.
Összehasonlításból $n-1+n-2+n-3+\ldots+2+1$ van (az 1. fázisban $n-1$, a 2. fázisban $n-2$, stb.), ez összesen $\frac{n(n-1)}{2}$ összehasonlítás. Cserékből minden fázisban legfeljebb 1 van, azaz összesen legfeljebb $n-1$, értékadásból pedig legfeljebb $2(n-1)+2\cdot\frac{n(n-1)}{2}$ van (mert a külső ciklus magának minden lefutása 2 értékadást tartalmaz, utána pedig legfeljebb kétszer annyi értékadás van, mint összehasonlítás). Ezek alapján a kiválasztásos rendezés lépésszáma legfeljebb $3\cdot\frac{n(n-1)}{2}+3(n-1)$.

1. megjegyzés
Vegyük észre, hogy a lépésszámra csak felső becslésünk van, pontosan nem tudjuk meghatározni, hiszen a lépések száma nem csak az n értékétől függ, hanem az adott inputtól is. Az viszont biztosan igaz, hogy a kapott $3\cdot\frac{n(n-1)}{2}+3(n-1)$ becslés minden egyes n méretű input esetén felső korlát a lépések számára.

2. megjegyzés
A lépésszám vizsgálatakor azok az érdekes esetek, amikor n értéke nagy, mert a lépésszámbecslést azért csináljuk, mert azt szeretnénk tudni, hogy mennyire lesz lassú az algoritmus akkor, ha nagy inputokon kezdjük használni (mennyire fog lelassulni, milyen gyorsan fog lelassulni, ha n nő).
Ha viszont n nagy, akkor az $3\cdot\frac{n(n-1)}{2}+3(n-1)=3\cdot\frac{n^2}{2}-1.5n+3(n-1)$ összegben a $3\cdot\frac{n^2}{2}$ tényező fog dominálni, a másik két tagnak szinte semmi szerepe nem lesz. Ezt jól láthatjuk, ha ábrázoljuk a $3\cdot\frac{x^2}{2}-1.5x+3(x-1)$ és $3\cdot\frac{x^2}{2}$ függvényeket (például a fooplot.com online

függvényábrázoló programmal $0 \leq x \leq 1000$ és $0 \leq y \leq 1500000$ beállításokkal), szinte alig látni a különbséget a két függvény között.

3. megjegyzés
Amikor lépésszámokról beszélünk, akkor általában (szinte mindig) beérjük annak meghatározásával, hogy a felső becslés n-nel, $n \log n$-nel, n^2-tel, n^3-bel arányos-e és nem törődünk azzal, hogy a felső becslés n, $3n$ vagy $100n$ alakú-e, vagyis hogy milyen, mekkora c konstans van az n-es tag előtt. Ez azért van így, mert azt szeretnénk a becsléssel leírni, hogy milyen tempóban fog az algoritmus futása lassulni, ebből a szempontból pedig az n, $3n$ és $100n$-es algoritmusok ugyanúgy viselkednek, míg például az n^2-es algoritmus gyökeresen máshogy.
Mit jelent az, hogy ugyanolyan tempóban lassul az n-es, $3n$-es és $100n$-es algoritmus? Ha egy n lépésszámú algoritmust egy S méretű input helyett $2S$ méretű inputon futtatunk, akkor a lépésszám S-ről $2S$-re változik, azaz megduplázódik. Ha a $3n$ lépésszámú algoritmust futtatjuk S méretű input után $2S$ méretű inputon, akkor a lépésszám $3S$ helyett $3(2S) = 6S$ lesz, azaz megduplázódik és ugyanez történik a $100n$-es lépésszámú algoritmussal: $100S$ helyett $100(2S) = 200S$ lesz, azaz megduplázódik a lépésszám, ha kétszer akkor inputon fut az algoritmus.
Mi a helyzet egy n^2-es algoritmussal? Itt ha S helyett kétszeres méretű, azaz $2S$ méretű inputon futtatjuk az algoritmust, akkor a lépésszám S^2 helyett $(2S)^2 = 4S^2$ lesz, azaz négyszeres lesz a futási idő és ez igaz minden $c \cdot n^2$ lépésszámú algoritmus esetén tetszőleges c pozitív szám esetén.
Ha pedig $c \cdot n^3$ lépést tesz egy algoritmus (ahol c tetszőleges pozitív szám), akkor a lépésszám nyolcszorosra fog nőni (bármi is a c értéke), miközben az input mérete megduplázódik.

A fenti gondolatmenet mutatja, hogy ha minket az érdekel (márpedig ez érdekel minket), hogy input méretének növekedésével milyen tempóban lassul az algoritmusunk, akkor teljesen jogos az összes $c \cdot n$ lépésszámú eljárást egy kalap alá vennünk (és az összes $c \cdot n^2$-est, stb.).

A nagy Ordó jelölés

A félév során általában $T(n)$-nel jelöljük azt a függvényt, ami leírja egy algoritmus lépésszámát. Ez a függvény mindegyik n érték esetén azt adja meg, hogy mennyi lépést tesz az algoritmus a legrosszabb n méretű inputon. A nehézség abban áll, hogy ezt a függvényt nem tudjuk pontosan kiszámolni (nem tudjuk általában, hogy melyik a legrosszabb input és nem tudjuk pontosan megszámolni a lépéseket, ha az algoritmus bonyolult) így jobb híján becsülni próbáljuk majd ezt a $T(n)$ függvényt.

Az előző rész megjegyzései így foglalhatók össze:

1. Felső becslést szeretnénk adni az algoritmus $T(n)$ lépésszámára az input méretének, n-nek a függvényében
2. Az az érdekes, hogy mi történik, ha az n értéke nagy (vagyis elég, ha a becslés nagy n-ekre igaz)
3. Nem számít, hogy a becslésben az n-es (n^2-es, $n \log n$-es, stb.) tag előtt milyen konstans áll.

Ezeket a pontokat fogalmazza meg a következő definíció:

Definíció
Egy algoritmus $T(n)$ lépésszáma $O(f(n))$ (úgy mondjuk, hogy $T(n)$ nagy ordó $f(n)$), ha van olyan c pozitív konstans és n_0 pozitív egész küszöbérték, hogy

$$T(n) \leq c \cdot f(n)$$

becslés igaz, ha $n \geq n_0$.

Figyeljük meg, hogy a fenti definícióban mindhárom korábbi szempont megjelenik:

1. $T(n)$-t felülről becsüljük
2. egy olyan becsléssel, ami nagy értékekre (az n_0 küszöbértéknél nagyobb n-ekre) igaz, úgy hogy
3. nem számít, hogy milyen konstans áll az $f(n)$ tag előtt.

Az új definíciót használva adjunk felső becslést a kiválasztásos rendezés lépésszámára!
Láttuk már a korábbiakban, hogy a kiválasztásos rendezés $T(n)$-nel jelölt lépésszámára igaz ($n \geq 2$ esetén), hogy

$$T(n) \leq \frac{n(n-1)}{2} + n - 1 \leq n(n-1)$$

Mivel $n(n-1) \leq n^2$ mindig igaz ezért a kiválasztásos rendezés $T(n)$ lépésszáma $O(n^2)$, a definícióban $c = 1$, $n_0 = 2$ értékeket használva.

Példa Ha egy algoritmus $T(n)$ lépésszámára fennáll, hogy $T(n) \leq 17n^3 - 2n^2 + 25$, amennyiben $n \geq 1$, akkor az algoritmus lépésszáma $O(n^3)$, mert:

$$T(n) \leq 17n^3 - 2n^2 + 25 \leq 17n^3 + 25 \leq 17n^3 + 25n^3 = 42n^3$$

Vagyis a $c = 42$, $n_0 = 1$ (mert a becslések mindig igazak voltak) választás megfelelő a definícióba.

Még egy példa Ha egy algoritmus $T(n)$ lépésszámára fennáll, hogy $T(n) \leq 100n^2 + 42 \log n + 7$, amennyiben $n \geq 10$, akkor az algoritmus lépésszáma $O(n^2)$, mert:

$$T(n) \leq 100n^2 + 42 \log n + 7 \leq 100n^2 + 42n^2 + 7n^2 = 149n^2$$

(itt használtuk, hogy $\log n \leq n^2$ és $7 \leq 7n^2$).
A fentiek alapján a $c = 149$, $n_0 = 10$ (mert a kiindulási becslés 10-től volt igaz) választás megfelelő a definícióba.

És még egy példa Ha egy algoritmus abból áll, hogy n-szer futtatunk le (egy ciklussal) egy olyan ciklusmagot, ami $O(n)$ lépésszámú, akkor ez az algoritmus $O(n^2)$-es.
Ez azért van így, mert ha a ciklusmag $O(n)$-es, akkor a definíció szerint van olyan c konstans és n_0 küszöb, hogy a ciklusmag minden egyes lefutása legfeljebb cn lépést igényel, amennyiben $n \geq n_0$. Ekkor $T(n)$-re, az egész algoritmus lépésszámára érvényes a

$$T(n) \leq n \cdot cn = c \cdot n^2$$

becslés, $n \geq n_0$ esetén, vagyis ezzel a c és n_0 választással teljesül az $O(n^2)$-es definíció.

Buborékrendezés

Az első feladatsor 1. feladatához hasonlóan (ahol minimumot kerestünk ezzel a stratégiával) a következő pszeudokód egy $n \geq 2$ hosszú tömbben a legnagyobb elemet a tömb végére mozgatja:

```
ciklus i = 0-tól n-2-ig:
    ha A[i] > A[i+1]:
        cseréljük meg A[i]-t és A[i+1]-et
ciklus vége
```

Ezen eljárás helyes, mert a legnagyobb elemet onnantól kezdve, hogy elérjük, végig jobbra mozgatjuk, így a végén a tömb utolsó cellájába kerül.
Az eljárás lépésszáma $O(n)$, mert az utolsó elem kivételével a tömb minden eleménél legfeljebb két lépést teszünk (egy összehasonlítás és legfeljebb egy csere), azaz $T(n) \leq 2(n-1) \leq 2n$, vagyis teljesül a definíció $c = 2$, $n_0 = 2$ választással.

Ezen eljárás felhasználásával készíthetünk egy rendezőalgoritmust is, melynek neve buborékrendezés.

Buborékrendezés szövegesen
Tegyük fel ismét, hogy $n > 1$, azaz legalább két elemű a tömb (az $n = 0$ és $n = 1$ esetekben nem kell tennünk semmit, az elvárt kimenet megegyezik a bemenettel).
1. fázis A fenti algoritmust lefuttatjuk az $A[0 : n-1]$ tömbön.

2. fázis A fenti algoritmust lefuttatjuk az 1. fázisban kapott tömb $A[0 : n-2]$ résztömbjén.

3. fázis A fenti algoritmust lefuttatjuk a 2. fázisban kapott tömb $A[0 : n-3]$ résztömbjén.

$\vdots$

(n-1). fázis A fenti algoritmust lefuttatjuk az előző fázisban kapott tömb $A[0 : 1]$ résztömbjén.

Példa
Input tömb: $8, 5, -4, 0, 10$

Az 1. fázis után (amikor a fenti algoritmus a teljes tömbön fut) a kapott tömb: $5, -4, 0, 8, 10$

A 2. fázis után (ahol a fenti algoritmus az első 4 cellából álló résztömbön fut) kapott tömb: $-4, 0, 5, 8, 10$

A 3. fázis után (ahol a fenti algoritmus az első 3 cellából álló résztömbön fut) kapott tömb: $-4, 0, 5, 8, 10$ (itt csere nem történt, csak összehasonlítások)

A 4. fázis után (ez egyben az utolsó fázis, ahol a fenti algoritmus az első két cellából álló résztömbön fut) a kapott tömb: -4, 0, 5, 8, 10 (itt csere nem történt, csak összehasonlítások).

A buborékrendezés megtekinthető animált formában itt: https://visualgo.net/bn/sorting (A felső sorban levő menüben a BUBBLE SORT lehetőséget kell választani.)

Helyesség
Ez az eljárás helyes, mert az 1. fázis után az input tömb legnagyobb eleme a tömb végére kerül, vagyis éppen oda, ahol a rendezett sorban állnia kell és később innen soha el nem mozdítjuk. A 2. fázis után a maradék elemek közül a legnagyobb, vagyis a tömb 2. legnagyobb eleme az utolsó előtti cellába kerül és innen soha el nem mozdítjuk később. Általában is igaz, hogy a k. fázis után a tömb k legnagyobb eleme már a helyén lesz és soha nem mozdul el onnan később, így az eljárás végére mindenki a helyére fog kerülni.

Buborékrendezés pszeudokóddal

Hasonlóan a kiválasztásos rendezéshez, itt is két, egymásba ágyazott ciklusra lesz szükségünk a pzseudokódban. A külső ciklus fogja szabályozni azt, hogy mekkora annak a résztömbnek a hossza, amiben dolgozunk (amin belül a legnagyobb elemet a végére mozgatjuk), a belső ciklus pedig azt fogja megvalósítani, hogy a külső ciklus által definiált résztömbön végigmenve a legnagyobb elemet a résztömb végére mozgatja.

```
ciklus j = n-1-tól 1-ig:   // az A[0:j] tömbben dolgozunk
    ciklus i = 0-tól (j-1)-ig:
        ha A[i]> A[i+1]:
            csere A[i] és A[i+1]
    ciklus vége
ciklus vége
```

A fenti kódban a // jel utáni rész a komment, amit azért írunk ide, hogy a j ciklusváltozó szerepét elmagyarázzuk.

A buborékrendezés lépésszáma

Ha pontosan akarunk számolni, akkor mondhatjuk, hogy összehasonlításból $n-1+n-2+\ldots+2+1=\frac{n(n-1)}{2}$ van (mert az első fázisban $n-1$, aztán $n-2$, stb.), csere pedig legfeljebb annyi van, mint összehasonlítás (sose cserélünk úgy, hogy nem volt előbb összehasonlítás) vagyis a lépésszámra igaz, hogy

$$T(n) \leq \frac{n(n-1)}{2} + \frac{n(n-1)}{2} = n(n-1) \leq n^2$$

vagyis a buborékrendezés lépésszáma $O(n^2)$, mert $c=1$, $n_0=2$ választással teljesül a definíció.

De becsülhettünk volna nagyvonalúbban is úgy, hogy a külső ciklus legfeljebb n-szer fut le (igazából $n-1$-szer fut le pontosan) és minden lefutása $O(n)$ lépés, mert egy legfeljebb n hosszú tömbben futattjuk azt az algoritmust, aminek lépésszáma ℓ méretű tömb esetén $O(\ell)$. Ez azt jelenti, hogy (a korábban látottak alapján, miszerint $n \cdot O(n)$ az $O(n^2)$) a buborékrendezés $O(n^2)$-es algoritmus.

Megjegyzés A buborékrendezés nem $O(n)$-es algoritmus. Ezt úgy mutathatjuk meg, hogy észrevesszük, hogy összehasonlításból mindig van $\frac{n(n-1)}{2}$ darab, vagyis $T(n) \geq \frac{n(n-1)}{2}$ és ha $T(n)$ $O(n)$ lenne, akkor lenne olyan c konstans és n_0 küszöbérték, hogy $T(n) \leq cn$ teljesülne, ha $n \geq n_0$.
A $T(n) \geq \frac{n(n-1)}{2}$ és $T(n) \leq cn$ egyenlőtlenségeket összekapcsolva azt kapnánk ekkor, hogy $\frac{n(n-1)}{2} \leq T(n) \leq cn$, azaz $\frac{n(n-1)}{2} \leq cn$, amennyiben $n \geq n_0$.
Az egyenlőtlenséget n-el elosztva azonban azt kapjuk, hogy $\frac{(n-1)}{2} \leq c$ ha $n \geq n_0$, ami ellentmondás, vagyis a kezdeti feltevésünk (hogy $T(n)$ $O(n)$-es) biztosan hibás volt, hiszen minden más lépés helyes volt a levezetésünkben.