Beszúrásos rendezés és bináris keresés

Csima Judit
BME SZIT
csima@cs.bme.hu

2019. szeptember 25.

Nagy ordó jelölés

Az előző előadáson bevezettük a nagy ordó jelölést, melynek segítségével algoritmusok $T(n)$ lépésszámáról tudunk állításokat megfogalmazni.

Idézzük fel újra a definíciót:

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 $n \geq n_0$.

A nagy ordó jelölés segítségével

1. $T(n)$-t, az algoritmus lépésszámát 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.

Tipikusan olyan állításokat fogunk majd tenni a félév során, hogy egy algoritmus lépésszáma $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$.

Példa a jelölés használatára
Ha egy algoritmus $T(n)$ lépésszámára fennáll, hogy $T(n) \leq 11n \log n + 7n - 18$, amennyiben $n \geq 1$, akkor az algoritmus lépésszáma $O(n \log n)$, mert:

$$T(n) \leq 11n \log n + 7n - 18 \leq 11n \log n + 7n \leq 11n \log n + n \log n = 12n \log n$$

ahol a becslés során azt használtuk, hogy a -18-as tag elhagyható, a $7n$-es tagra pedig felső becslés $n \log n$, amennyiben $n \geq 2^7$, vagyis a fenti becslés alapján a $c = 12$, $n_0 = 2^7$ pár jó választás.

Még egy példa
Ha egy algoritmus két egymás után lefuttatatandó részből áll, az első rész lépésszáma $O(n \log n)$, a másodiké pedig $O(n)$, akkor a teljes algoritmus lépésszáma $O(n \log n)$.

Ez azért van így, mert ha az első rész $O(n \log n)$, akkor van olyan c_1 konstans és n_1 küszöb, hogy az első rész lépésszáma legfeljebb $c_1 n \log n$, ha $n \geq n_1$ és ha második rész $O(n)$, akkor van olyan c_2 konstans és n_2 küszöb, hogy a második rész lépésszáma legfeljebb $c_2 n$, ha $n \geq n_2$. Így viszont

$$T(n) \leq c_1 n \log n + c_2 n \leq c_1 n \log n + c_2 n \log n = (c_1 + c_2) n \log n$$

ha $n \geq max(n_1, n_2, 2)$, mert $\log n \geq 1$ igaz minden $n \geq 2$ esetén.
Vagyis azt kaptuk, hogy a $c = (c_1 + c_2)$ és $n_0 = max(n_1, n_2, 2)$ választással teljesül, hogy $T(n)$ $O(n \log n)$.

Buborékrendezés megint

Nézzünk rá újra a buborékrendezésre és lássuk meg, hogy mennyire egyszerűsíti az életünket a nagy ordó jelölés akkor, amikor az algoritmus lépésszámáról akarunk beszélni!

Idézzük fel a jegyzet előző részében látott pszeudokódot:

```
ciklus j = n-1-től 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
```

Ahelyett, hogy pontosan megpróbálnánk kiszámolni, hogy mennyi lépés történik az egyes fázisokban (a külső ciklus egyes futásaiban, azaz akkor, amikor $j = n-1, n-2, \ldots, 2, 1$), használhatunk becsléseket: a külső ciklus legfeljebb n-szer fut le és minden lefutása $O(n)$ lépés, azaz a már ismert szabály értelmében (legfeljebb n-szer fut egy $O(n)$-es ciklusmag) a $O(n^2)$. Ennél pontosabban nem akarjuk megatározni a lépésszámot, ezt a becslést viszont különösebb számolgatás nélkül meg tudtuk kapni, a nagyo ordó jelölést használva.

Beszúrásos rendezés

Ennek a rendező algoritmusnak a következő eljárás az alapja:

Input: egy olyan $A[0 : n-1]$ tömb, melynek az első $n-1$ cellája, azaz $A[0 : n-2]$ már rendezett
Cél: Az $A[n-1]$ elemet is a helyére rakni, azaz rendezni az A tömböt.

Az eljárás szövegesen
Az eljárás úgy működik, hogy az $A[n-1]$ elemet addig cserélgetjük a tőle balra álló elemmel, amíg kisebb nála, vagyis folyamatos cserékkel $A[n-1]$-et a helyére mozgatjuk.

Példa
Ha a kiindulási tömb $1, 3, 8, 10, 12, 4$, akkor 3 cserével ($12, 10$ és 8) az $1, 3, 4, 8, 10, 12$ tömböt kapjuk.

Az eljárás pszeudokóddal
Az input tömb legyen $A[0 : n-1]$, ahol $A[0 : n-2]$ már rendezett.

```
i: = n-1
ciklus amíg (A[i] < A[i-1] és i > 0):
    csere A[i] és A[i-1]
    i: = i-1
ciklus vége
```

A fenti kódban ciklusmagja addig fut le, amíg a vizsgált elem az i indexű cellában kisebb, mint előtte álló ($i > 0$ biztosítja, hogy legyen előtte álló elem, vagyis hogy még nem a tömb elején járunk). Amikor először teljesül, hogy az elem nem kisebb az előtte állónál vagy amikor a tömb elejére érünk, akkor a ciklus véget ér.

Az eljárás helyes
A ciklusból akkor szállunk ki (és fejezzük be az algoritmust), amikor az elem elérte a tömb elejét (vagyis kisebb volt mindenkinél, azaz ekkor a helyén van) vagy amikor egy nála kisebb elem áll előtte, tehát már nem kell tovább mozgatnunk.

Az eljárás lépésszáma
Van 1 darab értékadás, utána pedig egy ciklus, aminek a magja legfeljebb n-szer fut le (hiszen i minden lépésben csökken), egy lefutása a ciklusmagnak pedig konstans sok lépés, vagyis a ciklus $O(n)$-es, az egész algoritmus lépésszáma így $1 + O(n)$, ami $O(n)$ a következő feladat szerint.

Feladat
Lássa be, hogy ha egy algoritmus két rész egymás utáni futásából áll, ahol az első rész konstans lépés, a második pedig $O(n)$, akkor az egész algoritmus $O(n)$.

A beszúrásos rendezés

A beszúrásos rendezés során az input egy (rendezetlen) $A[0 : n-1]$ tömb (feltehetjük, hogy $n \geq 2$, mert különben az output megegyezik az inputtal.)
A beszúrásos rendezés több fázisból áll:

1. fázis A fenti algoritmust futtatjuk az $A[0 : 1]$ résztömbre, úgy tekintve, hogy ennek $A[0 : 0]$ része már rendezett, ebbe szúrjuk bele $A[1]$-t.

2. fázis A fenti cserélgetős algoritmust futtatjuk az előző fázisban kapott $A[0 : 2]$ résztömbre, hiszen itt $A[0 : 1]$ már rendezett, ebbe szúrjuk bele $A[2]$-t.

3. fázis A fenti cserélgetős algoritmust futtatjuk az előző fázisban kapott $A[0 : 3]$ résztömbre, hiszen itt $A[0 : 2]$ már rendezett, ebbe szúrjuk bele $A[3]$-t.

⋮

(n-1). fázis A fenti cserélgetős algoritmust futtatjuk az előző fázisban kapott $A[0 : n-1]$ résztömbre, hiszen itt $A[0 : n-2]$ már rendezett, ebbe szúrjuk bele $A[n-1]$-t.

Példa
Input tömb: $2, 8, 3, 10, 1, 7$

Az 1. fázis során nem történik csere, a kapott tömb megegyezik az eredetivel: $2, 8, 3, 10, 1, 7$.

A 2. fázis során a 3-as helyet cserél a 8-assal és it megáll, ezt kapjuk: $2, 3, 8, 10, 1, 7$.

A 3. fázis során nincs csere, a tömb marad a $2, 3, 8, 10, 1, 7$.

A 4. fázisban az 1-es egészen a tömb elejéig gyalogol a cserékkel, a kapott tömb $1, 2, 3, 8, 10, 7$

Az 5. fázisban a 7-es cserél a 10 és 8-as értékkel, itt megáll, a végeredmény a rendezett $1, 2, 3, 7, 8, 10$ tömb.

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

Helyesség
Ez az eljárás helyes, mert minden fázisra igaz, hogy az a résztömb, amin az algoritmus abban a fázisban fut már majdnem rendezett, azaz az utolsó elem kivételével rendezett, az ilyen (rész)tömböket pedig jól rendezi a cserélgetős eljárás. Az algoritmus végén az egész tömb lesz az a résztömb, amin a cserélgetős eljárás fut, vagyis ekkor az egész tömb lesz rendezett.

Beszúrásos rendezés pszeudokóddal

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

1. fázis

```
i: = 1
ciklus amíg (A[i] < A[i-1] és i > 0):
    csere A[i] és A[i-1]
    i: = i-1
ciklus vége
```

2. fázis

```
i: = 2
ciklus amíg (A[i] < A[i-1] és i > 0):
    csere A[i] és A[i-1]
    i: = i-1
ciklus vége
```

3. fázis

```
i: = 3
ciklus amíg (A[i] < A[i-1] és i > 0):
    csere A[i] és A[i-1]
    i: = i-1
ciklus vége
```

⋮

(n-1). fázis

```
i: = n-1
ciklus amíg (A[i] < A[i-1] és i > 0):
    csere A[i] és A[i-1]
    i: = i-1
ciklus vége
```

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 indexű az a cella, amiben szereplő értéket be akarom szúrni az ez előtt álló rendezett résztömbbe), a belső ciklus pedig az így meghatározott résztömbben keresi meg az aktuális elem helyét a cserélgetéssel.

```
ciklus j = 1-tól (n-1)-ig:
    i := j
    ciklus amíg (A[i] < A[i-1] és i > 0):
        csere A[i] és A[i-1]
        i:= i-1
    ciklus vége
ciklus vége
```

A beszúrásos rendezés lépésszáma

A külső ciklus legfeljebb n-szer fut le, a ciklusmag $O(n)$-es, azaz az egész algoritmus $O(n^2)$-es.

Bináris keresés

Most egy kis kitérő következik, nem egy rendező algoritmust nézünk, hanem a keresési feladatot próbáljuk megoldani ügyesebben, mint ahogy ezt az első előadáson tettük. Ebben az ügyesebb algoritmusban, melynek neve bináris keresés, egy olyan gondolat jelenik meg, amit majd jól tudunk használni a következő órán, amikor visszatérünk a rendezésekhez.

A keresési feladat a következő: input egy n hosszú A tömb és egy s szám, az elvárt kimenet "Nincs!", ha a tömbben nincs benne s, ha pedig benne van, akkor annak a cellának az indexét akarjuk megkapni, ahol s van.
Ezt a feladatot az első órán már megoldott $s = 7$ esetben, az általános eset nagyon hasonló:

```
keresett_index := ‘‘Nincs!’’
ciklus i = 0-tól n-1-ig:
    ha A[i] == s:
       keresett_index := i
ciklus vége
return keresett_index
```

Helyesség Az eljárás helyes, mert minden értéket végignézünk és akkor állítjuk át a keresett_index változót *Nincs*!-ről, amikor s-et megtaláljuk (és pont arra az indexre állítjuk, ahol

megtaláltuk).

Ennek az eljárásnak a neve lineáris keresés és lépésszáma $O(n)$ hiszen $1 + O(n)$ lépésből, azaz $O(n)$ lépésből áll.

A bináris keresés elve

Ennél gyorsabban is tudunk keresni, amennyiben az input A tömb rendezett. Ennek a gyorsabb kereső algoritmusnak a neve bináris keresés és így működik:

Egyszerű eset (base case) Ha $n = 0$, akkor a válasz $''Nincs!''$.

Általános eset (ha $n \geq 1$): Vegyük a tömb középső elemét és hasonlítsuk össze az itt levő elemet s-sel. Három eset van:

1. A középső elem s: ekkor kész vagyunk és visszadjuk a középső elem cellájának indexét.
2. A középső elem nagyobb, mint s: ekkor a keresést (ugyanezzel a módszerrel) a középső elem előtti részben folytatjuk.
3. A középső elem kisebb, mint s: ekkor a keresést (ugyanezzel a módszerrel) a középső elem utáni részben folytatjuk.

Ez az eljárás azért hasznos, mert egyre kisebb és kisebb tömbökön dolgozik (minden körben feleződik a tömb mérete, amit nézünk), ezért gyorsan végetér pontosan is meg fogjuk nézni, hogy milyen gyorsan).

Helyesség

Azért helyes az eljárás, mert $n = 0$ esetben üres a tömb és nincs benne semmi, s sem, $n > 0$ esetén pedig vagy megtaláljuk az elemet (1. eset) vagy a 2. esetben biztosak lehetünk benne, hogy ha szerepel a tömbben, akkor csak a középső elem előtt lehet (hiszen a tömb rendezett), a 3. esetben pedig hasonlóan biztosak lehetünk benne, hogy ha szerepel a tömbben, akkor csak a középső elem után lehet.

Példa

Ha az $1, 3, 4, 5, 6, 8, 10$ tömbben keressük a 3-t, akkor először az 5-tel hasonlítjuk a 3-t (mert az 5 a középső elem), mivel ennél kisebb, ezért az $1, 3, 4$ tömbben dolgozunk tovább, aminek középső elemével hasonlítva meg is találjuk a 3-t.

Ha az $1, 3, 4, 5, 6, 8, 10$ tömbben a 2-t keressük, akkor először megint az 5-tel hasonlítjuk a 2-t (mert az 5 a középső elem), mivel ennél kisebb, ezért ismét az $1, 3, 4$ tömbben dolgozunk tovább, aminek középső elemével hasonlítva látjuk, hogy ennél is kisebb a 2, tehát az 1 elemű 1 tömbben dolgozunk tovább, ennek középső eleme az 1, ennél nagyobb a keresett elem, de az a tömb, amiben ezután tovább dolgoznánk már üres, vagyis azt adjuk vissza, hogy $''Nincs!''$.

Bináris keresés pszeudokóddal

A pszeudokód írásához két kérdést kell először megválaszolnunk:

- Hogyan fogjuk nyilvántartani a résztömböt, amiben dolgozunk?
  Ezt két változóval fogjuk megoldani, az *eleje* változó tárolja az aktuális résztömb első, a *vege* változó meg az aktuális résztömb utolsó cellájának indexét.
- Ha az aktuális résztömb kezdetének indexe i, vége pedig j, akkor mi a középső elem indexe?
  Ez egyszerű, ha $i+j$ páros, mert ekkor a középső elem indexe $\frac{i+j}{2}$, ha azonban $i+j$ páratlan (vagyis a résztömb páros hosszú), akkor el kell döntenünk, hogy az első fél utolsó vagy a második fél első elemét tekintjük középsőnek. Én most úgy döntök, hogy legyen a középső ebben az esetben az első fél utolsó eleme, aminek indexe ilyenkor $\lfloor \frac{i+j}{2} \rfloor$ (Itt a $\lfloor x \rfloor$ az alsó

egészrész függvény, ami azt a legnagyobb egész számot jelöli, ami nem nagyobb, mint x. Mivel ez a függvény egész értékű x esetén x maga, ezért mondhatjuk, a két esetet egyben kezelve, hogy a középső elem az $A[i:j]$ tömbben $\lfloor \frac{i+j}{2} \rfloor$.)

A bináris keresés pszeudokódja

A fenti két gondolatmenetet felhasználva az alábbi kódot kapjuk, az input tömb $A[0:n-1]$:

```
eleje:= 0
vége:= n-1
megvan:= ‘‘Nincs!’’
ciklus amíg (megvan == ‘‘Nincs!’’ és eleje <= vége):
    közép := alsó egész része (eleje + vége)/2-nek
    ha s == A[közép]:
        megvan := közép
    egyébként ha s < A[közép]:
        vége :=   közép -1
    egyébként:
        eleje := közép + 1
    elágazás vége
ciklus vége
return megvan
```

Ez a kód pont az, amit korábban szövegesen néztünk:

- addig fut a ciklus, amíg az elem még nincs meg, de még van esély rá, hogy megtaláljuk (a résztömb nemüres)
- ha megtaláljuk az elemet, akkor átállítjuk a *megvan* változót a megtalált elem indeére
- ha a keresett elem kisebb/nagyobb a középsőnél, akkor a vége/eleje változót kell módosítani, a középső előtti/utáni értékre.

A bináris keresés lépésszáma

Az elején van 3 lépés, utána pedig néhányszor lefut a ciklusmag, minden egyes futás konstans sok lépés (két összehasonlítás, hogy elinduljon-e a ciklumag, aztán egy összehasonlítás és két értékadás). Az csupán a kérdés, hogy hányszor fut le a ciklusmag? Ehhez vegyük észre, hogy a ciklusmag 1. futása előtt a tömb hossza (amiben dolgozunk) n és ez a hossz minden futással feleződik, pontosabban a következő résztömb mérete legfeljebb akkora, mint az előző résztömb méretének fele.
Azaz a 2. futás előtt legfeljebb $n/2$, a 3. futás előtt legfeljebb $n/4$, a 4. futás előtt legfeljebb $n/8$, a k. futás előtt legfeljebb $n/2^{k-1}$ hosszú a tömb. Ez azt is jelenti, hogy ha a k. az utolsó futás, amikor még a tömb hossza legalább 1, akkor $1 \leq$ a tömb hossza a k. futás előtt $< n/2^{k-1}$, amiből $1 < n/2^{k-1}$, amiből $2^{k-1} < n$, amiből pedig $k - 1 \leq \log n$ vagyis $k \leq \log n + 1$ adódik. Ez azt is jelenti, hogy a bináris keresés lépésszámára fennáll

$$T(n) \leq 3 + 5(\log n + 1) \leq 8 + 5 \log n \leq 6 \log n$$

amennyiben $\log n \geq 8$, azaz $n \geq 2^8$, vagyis $c = 4$ és $n_0 = 2^8$ választással teljesül az, hogy $T(n)$ $O(\log n)$.

Azt is mondhattuk volna, hogy a lépésszám konstans + $O(\log n)$, azaz $O(\log n)$, használva az alábbi feladat állítását:

Feladat

Ha egy algoritmus konstans sok lépésen kívül $O(\log n)$ lépésst tesz, akkor a lépésszáma $O(\log n)$.