Några svar till
TDDC70/91 Datastrukturer och algoritmer

2011-10-18

Följande är lösningsskisser och svar till uppgifterna på tentan. Lösningarna som ges här ska bara ses som vägledning och är oftast inte tillräckliga som svar på tentan.

1. (a) **Sant**. $\log(n!) < \log(n^n) = n\log(n)$ som växer polynomiskt.

   (b) **Sant**. Enligt definitionen av O behöver vi två konstanter $c > 0$ och $n_0 \geq 1$ sådana att $\max\{f(n), g(n)\} \leq c(f(n) + g(n))$ för varje heltal $n \geq n_0$. Eftersom funktionerna $f(n)$ och $g(n)$ är icke-negativa är maximum av de två funktionerna alltid mindre än summan av funktionerna. Detta gäller för $c = 1$ och $n_0 = 1$.

2. Den grundläggande idén är att om två prov är olika så kan de förkastas eftersom ett av dem inte tillhör majoriteten och majoriteten bevaras om vi gör detta.

   Algoritmen behandlar proverna ett och ett och håller reda på en kandidat c till att vara ett majoritetsprov och dess ”multiplicitet” k. Invarianten vi använder är att om vi lägger till k kopior av prov c till de obehandlade proverna så är majoritetsprovet i ursprungsuppsättningen av prover och denna nya uppsättning prover samma.

   Algorithm 0.1: FindMajority(S)

   $k \leftarrow 0$
   for $i \leftarrow 0$ **to** $n - 2$
   do { **if** $k = 0$
   then { $k \leftarrow 1$; $c \leftarrow S[i+1]$ }
   else { **if** $S[i+1] = c$ **then** $k \leftarrow k + 1$ **else** $k \leftarrow k - 1$ }
   return (c)

   Alternativa lösningar skulle kunna innebära att gå genom arrayen med prover och jämföra varje konsekutivt par. Om ett par innehåller distinkta prov, kasta bort dem. Om ett par innehåller två ”likadana” prov, behåll ett av dem. Utför sedan samma procedur på resterande prover. I varje sådant svep över alla kvarvarande prover kastas åtminstone hälften av proverna bort. Låt oss, för enkelhets skull, anta att $n = 2^k$. Då är antalet jämförelser som behöver utföras inte fler än $n/2 + n/4 + n/8 + \ldots + 1 = 2^{k-1} + \ldots + 1 = 2^k - 1 = n - 1$.

3. (a) `5`
   `5 3`
   `5`
   `5 2`
   `5 2 8`
   `5 2`

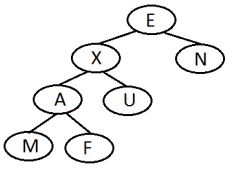
```
5
5 9
5 9 1
5 9
5 9 7
5 9 7 6
5 9 7
5 9
5 9 4
5 9
5
```

(b)
```
5
5 3
3
3 2
3 2 8
2 8
8
8 9
8 9 1
9 1
9 1 7
9 1 7 6
1 7 6
7 6
7 6 4
6 4
4
```

Algorithm 0.2: REVERSE(Q)

(c)
```
S ← en tom stack
while ! Q.isEmpty()
  do S.push(Q.dequeue())
while !S.isEmpty()
  do Q.enqueue(S.pop())
```

4. (a)



(b)

Djupet av en nod v är lika med djupet av föräldern plus ett. Därför kommer vår algoritm att härma en preordertraversering (varje förälder måste behandlas före sina barn). För att beräkna djupet av varje nod i T anropas följande algoritm med T och T.root() som indata.

Algorithm 0.3: ComputeDepth(T, v)

```
if T.isRoot(v)
  then setDepth(v, 0)
  else setDepth(v, 1 + getDepth(T.parent(v)))
children ← T.children(v)
while children.hasNext()
  do { child ← children.next()
     { ComputeDepth(T, child)
```

Vi antar att children(v) går i tid $O(c_v)$ i värsta fallet, där c_v är antalet barn v har. Då tar varje rad i **if** -satsen tid $O(1)$. Tilldelningen till *children* tar $O(c_v)$ tid och **while** -loopen exekveras också c_v gånger för att rekursivt beräkna djupen av alla noder i delträdet rotat i v. Om vi exkluderar de rekursiva anropen tar ComputeDepthT $O(c_v)$ tid.

Hur lång tid tar ComputeDepth$(T, T.\text{root}())$? Notera att för varje nod v i T anropar vi ComputeDepth(T, v) exakt en gång (eftersom varje nod har som mest en förälder och vi börjar med roten i T). Därför är exekveringstiden för ComputeDepth$(T, T.\text{root}())$ lika med den sammanlagda tiden det tar att exekvera den icke-rekursiva delen av ComputeDepth för varje nod i T. Det blir totalt $\sum_{v \in T} O(c_v) = O(\sum_{v \in T} c_v)$, vilket är $O(n)$.

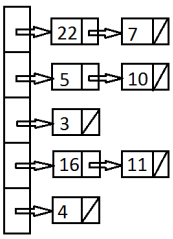
En alternativ lösning till det här problemet behöver en extra datastruktur som kan göra insättningar och borttagningar i $O(1)$ tid (t.ex. stackar och köer). Följande är en algoritm som använder en stack för att beräkna djupet. För att få till en algoritm som använder en kö får vi starta med en tom kö och ersätta alla push- och pop-operationer med anrop till enqueue respektive dequeue.

Algorithm 0.4: ComputeDepth(T)

```
current ← T.root()
S ← en tom Stack
S.push(current)
while !S.isEmpty()
  do { current ← S.pop()
     { if T.isRoot(current))
     {   then setDepth(current, 0)
     {   else setDepth(v, 1 + getDepth(T.parent(v)))
     { children ← T.children(v)
     { while children.hasNext())
     {   do S.push(children.next())
```

Anledningen till att vi kan använda både stack och kö här är att det inte spelar någon roll i vilken ordning vi behandlar barnen till varje nod, så länge vi behandlar noden själv före alla dess barn. Vi säkerställer detta genom att ta ut en nod och *sedan* sätta in dess barn i vilken extra datastruktur det nu är vi använder. Observera att varje nod sätts in och tas ut exakt en gång. Antalet insättningar är en övre gräns för hur många gånger den yttre **while** -loopen exekveras och antalet uttagningar begränsar antalet iterationer i den inre **while** -loopen. Allting annat är enkla operationer och metoder som tar $O(1)$ tid, så den här algoritmen går också i $O(n)$ tid.

5.



6. (a) För att illustrera lösningsprincipen ger vi först en algoritm baserad på antagandet att elementen har en av två färger: röd eller blå. Vi håller reda på två index i arrayen $front$ och *end*. $front$ och *end* rör sig mot varandra till de möts coh då terminerar algoritmen. Medan de traverserar sekvensen gör vi följande: så fort $front$ ser ett rött element och *end* ser ett blått element byter vi plats på dessa element. Det betyder att alla platser i arrayen $front$ traverserar kommer att innehålla blåa element, medan alla element *end* traverserar kommer att innehålla röda element.

Algorithm 0.5: REDBLUESORT(S)

$n \leftarrow S$.size
$front \leftarrow 0$
$end \leftarrow n - 1$
while $front < end$
do
- **if** $S[front]$.color() $=$ *red* **and** $S[end].color() = blue$
  - **then**
    - **comment:** swap the two elements
    - $tmp \leftarrow S[front]$
    - $S[front] \leftarrow S[end]$
    - $S[end] \leftarrow tmp$
- **comment:** find the leftmost red element if such exists
- **while** $S[front]$.color() $=$ *blue* **and** $front < end$
  do $front \leftarrow front + 1$
- **comment:** find the rightmost blue element if such exists
- **while** $S[front]$.color() $=$ *red* **and** $front < end$
  do $end \leftarrow end - 1$

Algoritmen är in-place och eftersom $front$ och *end* rör sig mot varandra besöks varje element i sekvensen av någon av dem exakt en gång. Varje gång vi inkrementerar $front$ eller dekrementerar *end* utförs som mest en swap-operation och som mest tre kontroller (färgkontroll och möjligtvis två kontroller av $front < end$). Alla dessa tar konstant tid att utföra. Därför är körtiden för REDBLUESORT $O(n)$.

Algoritmen för fallet med k färger är baserad på ovanstående algoritm. k gånger gör vi följande:

- Vi håller reda på $right\_boundary$ för S (initialiserad till n).
- Vi väljer en av k färger, $color_i$ som vi inte valt förut.
- Vi sorterar arrayen S precis som ovan förutom att vi använder $color_i$ som den röda färgen och alla färger som inte är $color_i$ som den blåa färgen. Dessutom sorterar vi endast element i S som finns mellan 0 och (inte inklusive) $right\_boundary$.
- Efter att vi är klara sätter vi $right\_boundary$ till det värde indexen $front$ och *end* konvergerat till.

Observera att en iteration av den här algoritmen har exekveringstid $O(n)$ eftersom det, i princip, är samma algoritm som för fallet med två färger och den körs k gånger. Därför blir den totala exekveringstiden $O(nk)$.

Algorithm 0.6: SINGLECOLORSORT($S, right\_boundary, currcolor$)

$front \leftarrow 0$
$end \leftarrow right\_boundary - 1$
while $front < end$
do
- **if** $S[front]$.color() $=$ $currcolor$ **and** $S[end].color() \neq currcolor$
  - **then**
    - **comment:** swap the two elements
    - $tmp \leftarrow S[front]$
    - $S[front] \leftarrow S[end]$
    - $S[end] \leftarrow tmp$
- **comment:** find the leftmost $currcolor$ element if such exists
- **while** $S[front]$.color() $\neq currcolor$ **and** $front < end$
  - **do** $front \leftarrow front + 1$
- **comment:** find the rightmost non-$currcolor$ element if such exists
- **while** $S[front]$.color() $= red$ **and** $front < end$
  - **do** $end \leftarrow end - 1$

if $S[front]$.color() $= currcolor$
then return $(front)$
else return (end)

Algorithm 0.7: COLORSORT(S, C)

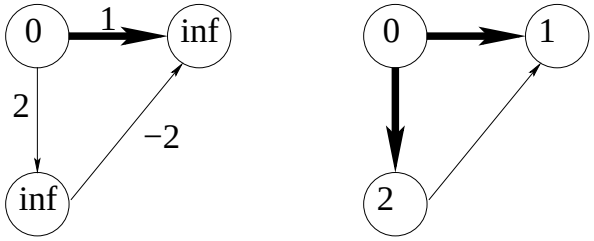
$right\_boundary \leftarrow S$.size()
for $colorind \leftarrow 0$ **to** C.size() $- 1$
do
- $currcolor \leftarrow C[colorind]$
- $right\_boundary \leftarrow$ SINGLECOLORSORT($S, right\_boundary, currcolor$)

(b) Låt k vara ett heltal i intervallet $[0, n^2 - 1]$. Om vi representerar talet i bas n skulle det bestå av två tal (l, f), sådana att $k = l \cdot n + f$, där l och f båda är från intervallet $[0, n-1]$. Lämpligt nog är den lexikografiska ordningen på bas n-representationen precis samma som ordningen för heltalen vi vill sortera. Därför kan vi använda Radix-sort för att sortera sekvensen i linjär tid.

Algorithm 0.8: LIMITEDSORT(S, n)

comment: change the representation
for $i \leftarrow 0$ **to** $n - 1$
do $A[i] \leftarrow ((S[i] - S[i] \bmod n)/n, S[i] \bmod n)$
RADIXSORT(A)
comment: restore the representation
for $i \leftarrow 0$ **to** $n - 1$
do $S[i] \leftarrow A[i][0] \cdot n + A[i][1]$

7. (a) Problemet är att uppdateringen av avståndsmärkningen bara sker för noder som är grannar till den nod som behandlas och som inte är ”i molnet” än.



(b) Det här kan misslyckas av två skäl. För det första är det inte säkert att strategin hittar någon stig alls, eftersom den inte har någon mekanism för att backa om den hamnar i situationen att alla utgående bågar från noden den befinner sig i leder till noder som redan besökts. I figuren nedan till vänster besöks noderna i ordningen *start*, *a* och *b*. Väl i *b* har alla grannar till *b* redan besökts, så algoritmen fallerar vid steg iv.

För det andra kan strategin misslyckas om det finns tre noder v_i, v_j, v_k där bågvikterna mellan dem uppfyller följande egenskaper:

- $w((v_i, v_j)) < w((v_i, v_k))$ (så stigen algoritmen väljer mellan v_i och v_k går genom v_j)
- $w((v_i, v_j)) + w((v_j, v_k)) \geq w((v_i, v_k))$ (triangelolikheten)

Figuren nedan till höger visar ett exempel på detta — noderna besöks i ordningen *start*, *a*, *b*, *goal* med 70 som total väglängd. Den kortaste vägen, däremot, är *start* → *c* → *goal*, med längd 15. Att bågvikterna uppfyller triangelolikheten är rätt så vanligt, särskilt om de representerar ett faktiskt fysiskt avstånd mellan noderna.