Några svar till
TDDI16 Datastrukturer och algoritmer

2013-10-29

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**. A representerar en giltig max-heap enligt definitionen.

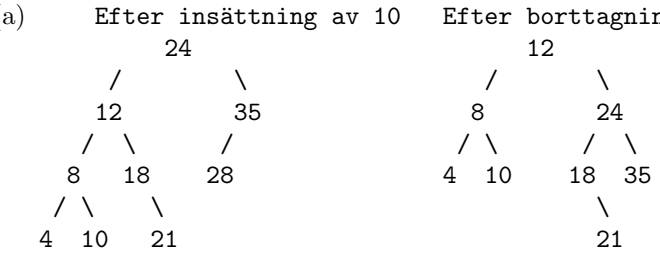
   (b) **Falskt**. Counting Sort använder tid $O(n+k)$ för att sortera n heltal från $\{1,\ldots,k\}$. Om t.ex. $k \in \Omega(n^2)$ blir inte tidskomplexiteten linjär. Dessutom behöver inte indatasekvensen bestå av heltal utan kanske av element som vi bara kan jämföra.

   (c) **Sant**. Vi ger ett induktionsbevis. Två basfall behövs. Notera att $f_3 = 2$ är två gånger så stort som $f_2 = 1$ och $f_4 = 3$ är 1.5 gånger så stort som f_3. Om vi antar att $f_{k-1} \geq 1.5 \cdot f_{k-2}$ och $f_{k-2} \geq 1.5 \cdot f_{k-3}$, så gäller $f_{k-1} + f_{k-2} \geq 1.5 \cdot (f_{k-2} + f_{k-3})$ eller $f_k \geq 1.5 \cdot f_{k-1}$. Genom induktion gäller alltså, för alla $n > 2$, att

   $$f_n \geq 1.5 \cdot f_{n-1} \geq (1.5)^2 \cdot f_{n-2} \geq \ldots \geq (1.5)^{n-2} \cdot f_2 = (1.5)^{n-2}.$$

   Med andra ord växer f_n snabbare än funktionen $(1.5)^{n-2}$.

2. (a)



   (b) Denna uppgift skulle inte ha varit med på tentan. Alla tilldelades därför maxpoäng på just den här uppgiften.

3. Det spelar egentligen ingen roll vilken variant man väljer. I varje fall kommer man att få minst två operationer som kräver linjär tid, medan övriga operationer bara använder konstant tid.

4. Den enklaste lösningen testar bara varje element i arrayen. Detta kräver uppenbarligen $O(n)$ tid, men det finns en snabbare lösning.

   Följande algoritm använder $O(\log n)$ tid eftersom det väsentligen är en binärsökning. Tricket använder två faktum: arrayen innehåller heltal och dessa är ordnade i strikt ökande ordning (det finns inga dubletter). Tänk på vad vi vet ifall $A[i] < i$ för något i. I så fall har vi att $A[i-1] \leq A[i] - 1 < i - 1$, så $A[i-1] < i - 1$. På samma sätt kan vi visa att $A[i-2] < i - 2$, och så vidare, induktivt, kan vi visa att $A[j] < j$ för j med $0 \leq j \leq i$. Med ett liknande argument får vi att om $A[i] > i$ så gäller $A[j] > j$ för j med $i \leq j \leq n-1$.

```
int FindEqual( int A[], int n, int & loc )
{
  int low = 0, high = n-1, mid;

  while ( low <= high ) {
    mid = (low + high) / 2;
    if ( A[mid] == mid ) {
      loc = mid;  return true;
    }
    else if ( A[mid] < mid )
      low = mid+1;
    else
      high = mid-1;
  }
  return false;  // at this point low > high and there is
                 // nothing left to search
}
```

Eftersom sökrymden halveras i varje oteration krävs som mest $O(\log n)$ iterationer. Eftersom varje iteration kräver $O(1)$ tid får vi tidskomplexitet $O(\log n)$.

5. (a) 2 eller 3. Det måste vara ett av barnen till rotnoden.
   (b) $\lfloor n/2 \rfloor + 1$ till n. Detta är alla löv.
   (c) $\Theta(n)$ eftersom alla $n - \lfloor n/2 \rfloor$ platser i frågan ovan måste besökas.

6. `0 |-|->28`
   `1 |-|->15`
   `2 | |`
   `3 |-|->10-->17`
   `4 | |`
   `5 |-|->12-->33-->19-->5`
   `6 |-|->20`

7. (a) Här är arrayen:

0	1	2	3	4	5	6	7	8
8	12	14	16	32	71	26	25	24

   Det rekursiva anropet är `Q_Select_One( A, 3, 3, 8)`. Notera att detta anrop görs trots att värdet 16 på index 3 är rätt värde.

   (b) Det värsta fallet inträffar när `A` redan är sorterad. I det fallet gör det första anropet $n-1$ jämförelser och betraktar rekursivt intervallet från 1 till $n-1$. Vart och ett av de efterföljande k anropen innefattar en jämförelse mindre och, förutom när `Left == k`, görs rekursiva anrop på intervall som är en plats mindre. Detta ger $k+1$ anrop och $O(n)$ jämförelser per anrop — totalt $O(nk)$ jämförelser. Mer precist är antalet jämförelser

$$\sum_{i=0}^{k}(n-i-1) = (k+1)n - \sum_{i=0}^{k} i - (k+1) =$$

$$= (k+1)n - k(k+1)/2 - (k+1) \in O(kn), \text{ eftersom } k \leq n.$$