Några svar till
TDDI16 Datastrukturer och algoritmer

2014-01-08

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. 1 (D), linjär.

   2 (G). Varje iteration av den inre loopen är kvadratisk i den yttre loop-variabeln. Ett enkelt sätt att komma fram till resultatet är att inse att vi vill ha summan $\sum_{i=0}^{n-1} i^2$.

   3 (H). Varje anrop ger upphov till två nya anrop. När vi kommer till basfallen (där $n = 1$) har vi genererat 2^n anrop till `f3`

   4 (D). Det här fallet är likt Mergesort och Quicksort, förutom att varje rekursivt anrop bara utför en konstant mängd arbete i stället för en linjär mängd arbete. Mönstret är samma som det för att bygga en heap från botten och upp. Längst upp gör vi 1 enhet arbete, på andra nivån gör vi 2 enheter arbete, på tredje nivån 4 enheter, etc. Den totala mängden arbete ges därför av $1+2+4+8+\ldots+n$. Denna summa är linjär i n.

   5 (E). Det här är exakt samma mönster som i Mergsort och Quicksort. Om man vill tänka på det som en summa så blir det $n+n+\ldots+n$ där det finns $\log_2 n$ summander.

   6 (B). Efter första iterationen blir $i = 2$. Efter andra blir $i = 2^2$. Efter tredje blir $i = 2^{2^2}$, etc. Den här proceduren använder $\log^* n$ steg för att nå n. Det går bra att hänvisa till att $\log^* n$ var det enda möjliga svaret med en tillväxthastighet mellan konstant och logaritmisk.

2. (a) Trädets struktur är felaktig och så finns det element som gör att heapegenskapen inte är uppfylld.

   (b)
   ```
            A
         A      E
       C   S  R   T
      U D T U T R
   ```

   (c)
   ```
            A
         A      E
       C   S  R   R
      U D T U T T
   ```

3. (a) $O(n^2)$ resp. $O(n \log n)$.

   (b) Använd ett balanserat sökträd istället.

   (c)
   ```
                  25
               /      \
             10        50
           /   \     / \
          5     15  30 60
        /          /   / \
       1          27  55 75
   ```

4. (a) En hashtabell `originals` som avbildar strängar på mängder av strängar.

   Huvudidén är att undvika att generera alla permutationer av varje ord (vilket kräver $L!$ tid).

   - Läs varje ord `word` från indata, sortera det med insertion sort för att få `dorw` och lägg till `word` i mängden `originals[dorw]` (skapa mängden om den inte finns). Håll reda på största mängdstorleken. $O(NL^2)$.
   - Iterera över `originals`. Stanna när en mängd av maximal storlek hittas. Skriv ut orden i denna mängd. $O(N)$.

   Man kan tänka sig att sortera varje ord med räknesortering, vilket skulle ge tidskomplexitet $O(NLR)$, där R är storleken på alfabetet, men insertion sort ($O(NL^2)$) är antagligen snabbare i praktiken för ord från något naturligt språk.

   (b) Vi behöver undvika en exponentiell sökning efter alla möjliga ordstegar. Här är en sådan lösning.

   Vi behöver en metod `neighbours(dorw)` som antar att `originals` redan finns. Givet ett ord w av längd k genererar metoden de k ord av längd $k-1$ som w ger upphov till och returnerar de som är giltiga (dvs de som är nycklar i `originals`). $O(L^2)$.

   - Initialisera en hashtabell `rung_height` från sorterade ord till `int`:ar med alla värden satta till 0. `rung_height` för varje `dorw` är maximala stegpinnehöjden för `dorw` över alla anagramstegar `dorw` kan finnas med i. Lägsta stegpinnehöjden räknas som 0.
   - Skapa en sorterad array av sorterade ord i ökande längdordning. $O(N)$ tid med räknesortering.
   - För varje sorterat ord `dorw` i arrayen:

     ```
     rung_height[dorw] = max(rung_height[nbr] for nbr in neighbours[dorw]) + 1
     // Om neighbours[dorw] är tom, gör ingenting, eftersom dorw måste vara
     // lägsta stegpinnen i alla möjliga stegar. Notera att grannarna redan
     // behandlats pga sorteringen.
     ```

     N anrop till `neighbours`, $O(NL^2)$.
   - Hitta det sorterade ordet med störst `rung_height` och sök iterativt efter en sekvens av grannar med `rung_height` ett mindre än föregående. $O(N + \text{poly}(L))$.
   - Slå upp de sorterade orden i denna stege (baklänges) i `originals` och skriv ut en sekvens av ord i språket.

5. (a) ja, ja, ja, nej, nej

   (b)
   - A. Om vi vill sortera en mängd av slumpvis ordnade element så att vi får bäst prestanda (utan att bry oss om stabilitet) bör vi använda quicksort.
   - A eller C. Även i det här fallet vill vi ha hög prestanda, men bryr oss inte om stabilitet. Om observationerna är slumpmässigt ordnade vinner quicksort. Ett rimligt antagande är att den osorterade `Observation`-arrayen fyllts i efter någorlunda stigande tidsstämpel. I så fall vill vi använda insertionsort för att dra nytta av att arrayen innehåller partiellt ordnat data.
   - B. I det fallet vill vi ha effektivitet och stabilitet och objekten är ordnade slumpmässigt efter `importance`. Den vinnande algoritmen blir mergesort.
   - C. Här har vi en array som är nästan helt ordnad, så vi vill använda insertionsort.

6. 0:
   1: 28 -> 19 -> 10
   2: 20
   3: 12
   4:
   5: 5
   6: 15 -> 33
   7:
   8: 17