Några svar till
TDDI16 Datastrukturer och algoritmer

2015-01-08

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. (a) Sant. Enligt definitionen finns konstanter n_1, c_1, n_2, c_2 sådana att $f(n) \leq c_1 h(n)$ för alla $n \geq n_1$ och $g(n) \leq c_2 h(n)$ för alla $n \geq n_2$. Låt $n_0 = \max(n_1, n_2)$. För alla $n \geq n_0$ gäller att $f(n) \leq c_1 h(n)$ och $g(n) \leq c_2 h(n)$. Låt vidare $c_0 = c_1 + c_2$. För alla $n \geq n_0$ gäller att $f(n) + g(n) \leq (c_1 + c_2)h(n) = c_0 h(n)$ vilket medför att $f(n) + g(n) \in \mathcal{O}(h(n))$.

   (b) Falskt. $n^3 \in \Theta(n^3)$ men $n^3 \notin \Theta(n^2) \Rightarrow \Theta(n^2) \not\subset \Theta(n^3)$.

2. (a) I värsta fallet är p pch q ungefär lika stora så vi behöver testa olika p från 2 till $\lfloor \sqrt{r} \rfloor$. Vi antar att $r = p \cdot q$, $q > p$, så p kan inte vara större än $\lfloor \sqrt{r} \rfloor$.

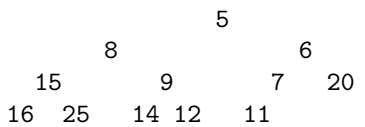
   $$n = (\log_2 r)/8 \Rightarrow \log_2 r = 8n \Rightarrow r = 2^{8n} \Rightarrow \sqrt{r} = 2^{4n} = 16^n$$

   Eftersom varje division tar $\mathcal{O}(n)$ tid och vi behöver testa $\sqrt{r} - 1$ heltal i värsta fallet är tidskomplexiteten $(16^n - 1) \cdot \mathcal{O}(n) \in \mathcal{O}(n \cdot 16^n)$.

   (b) En uppskattning av tidsåtgången är $\sqrt{r} - 1$ mikrosekunder. Eftersom r har 100 bitar kan r vara så stort som $2^{100} - 1$. Tiden som går åt är alltså ungefär 2^{50} mikrosekunder. $2^{50} = 2^{10 \cdot 5} = 1024^5 \approx 10^{15}$. 10^{15} mikrosekunder $= 10^9$ sekunder $\approx 31,7$ år.

3. Om vi söker på vanligt sätt efter nyckeln k i ett AVL-träd som har minst en förekomst av denna nyckel hittar sökningen den nod i AVL-trädet med lägst djup som innehåller nyckeln. Låt oss kalla denna nod för v. Alla övriga förekomster av nyckeln k finns alltså i vänster och/eller höger delträd till v. I synnerhet kan vi söka oss nedåt till höger i vänster delträd till v till vi stöter på en nod med nyckel k och spara undan den noden tillsammans med alla noder i dess högra delträd. Samma typ av sökning behöver sedan fortsätta i den funna nodens vänstra delträd. Samma typ av strategi också användas i v:s högra delträd.

4. `removeMin()` följt av `insert(3)`. Mellanheapen:



5.
```
 0: \
 1: \
 2: -> 10\
 3: \
 4: -> 40\
 5: 22 -> 11\
 6: -> 15\
 7:
 8:
 9: -> 16\
10: -> 9 -> 20\
```

6. Låt S' och S'' vara elementen i första respektive andra halvan av S och låt s'_i och s''_i vara elementen med rank i i S' respektive S''. Då gäller att antalet inversioner i S = antalet inversioner i S' + antalet inversioner i S'' + antalet inversioner med ett element i S' och ett i S''. Antalet inversioner i S' och S'' kan bestämmas rekursivt. Återstår att effektivt hitta antalet inversioner med ett element från varje sekvens. Eftersom S' består av element före dem i S'' uppstår en inversion bara om ett element i S' är större än ett element i S''. Dessutom, om S' och S'' är *sorterade* och s'_i och s''_j utgör en inversion utgår alla element i S' med rank större än i också inversioner med s''_j.

   Vi får en modifierad merge-sort — i merge-steget när ett element väljs i stället för ett element i S' ska anatalet inversioner ökas med anatalet ej mergeade element kvar i S'. Vi behöver bara ett pass genom S' och S'' för att hitta alla inversioner med element från båda sekvenserna.

   (a)
```
countInversions(S)
      // triviala fall
      if S.size() < 2
        return 0

      // dela upp
      låt S' och S'' vara två nya sekvenser
      for 1 <= i < n/2
         S'.insertLast(S.removeFirst())
      for n/2 < i <= n
         S''.insertLast(S.removeFirst())

      // rekrusivt steg
      inv1 <- countInversions(S')
      inv2 <- countInversions(S'')

      // merge
      inv3 <- 0
      while S' och S'' är icketomma do
         s'  <- S'.first()
         s'' <- S''.first()
         if s' <= s'' then      // rätt ordning
            S.insertLast(s')
            S'.removeFirst()
         else                   // en inversion
            S.insertLast(s'')
            inv3 <- inv3 + S'.size()
            S''.removeFirst()

      return inv1 + inv2 + inv3
```

(b) Att dela upp och göra merge tar tid $\mathcal{O}(m)$ för sekvenser av storlek m och rekursionssteget delar problemet i två delproblem av storlek $m/2$. På djup i finns 2^i delproblem av storlek $n/2^i$ vardera, så total tid för uppdelning och merge av dessa problem är $\mathcal{O}(2^i \cdot n/2^i) \in \mathcal{O}(n)$. Det maximala djupet för ett delproblem är $\log n \Rightarrow$ tidskomplexiteten är $\mathcal{O}(n \log n)$.