Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Magnus Nielsen

23 oktober 2017

TDDC91
Datastrukturer och algoritmer
Datortentamen (DAT1)
2017-10-23, 14–18
Lösningsförslag

1. **Uppgift 2**

   (a) Vi vet att det är stora mängder data som ska sorteras, och vi vet det är numeriska nycklar vi ska sortera. Med detta i åtanke vill vi se till att vi har en algoritm som fungerar väl för syftet. QuickSort eller MergeSort är bra val, men de är jämförelsebaserade och sålunda kan de inte bli snabbare än $O(n \log n)$. Vi har de räknebaserade algoritmerna RadixSort och BucketSort som båda är bra till stora datamängder och har tidskomplexitet $O(n)$, men de använder mycket mer minne än de jämförelsebaserade algoritmerna.

   Ex 1) Då företaget har god ekonomi och många (bara?) nöjda kunder gör jag antagandet att de har (eller har råd med) riktigt bra datorer, och gör valet att hastigheten är viktigare än vara försiktig med minnet, och väljer RadixSort. RadixSort partitionerar data likt QuickSort (fast med en annan metod), men den är inte lika snabb på små datamängder, i synnerhet inte om det är stora tal, så jag väljer att använda mig av CountingSort för tillräckligt små datamängder. Brytpunkten avgör jag med testning. (Jag har tagit hänsyn till tidskomplexitet, minne, samt både stora och små datamängder. Full poäng).

   Ex 2) Jag anser att de räknebaserade algoritmerna skulle kräva för mycket minne, och tar beslutet att den högre tidskomplexiteten hos jämförelsebaserade algoritmer vägs upp av den lägre minnesanvändningen. Både QuickSort och MergeSort hade fungerat bra, men QuickSort har jag implementerat i labbserien, så jag är bekväm med, och förstår, den och valet faller därför på QuickSort. Då jag vet att QuickSort blir betydligt mindre effektiv på små datamängder väljer jag att köra tillräckligt små partitioner genom SelectionSort (eller InsertionSort eller någon annan algoritm som är speciellt effektiv på små datamängder). (Jag har tagit hänsyn till tidskomplexitet, minne, samt både stora och små datamängder. Full poäng).

   (b) **Ex 1)** RadixSort och CountingSort har båda tidskomplexiteten $O(n)$
   Ex 2) QuickSort har tidskomplexiteten $O(n^2)$ i värsta fallet, men förväntade fallet är $O(n \log n)$. Selection-/Insertion/etc-sort har tidskomplexitet $O(n^2)$, men då vi använder dem som en optimering på riktigt små datamängder för QuickSort som redan har en förväntad tidskomplexitet på $O(n \log n)$ kan vi med gott samvete säga att vår lösning har en tidskomplexitet $O(n \log n)$ i det förväntade fallet.

   (c) Vi har nu en datumklass istället för numeriska värden, och är sålunda tvungna att använda oss av jämförelsebaserade sorteringsalgoritmer om vi inte väljer att göra typkonverteringar.
   Ex 1) Vi väljer att typkonvertera datumen till heltal på formen: YYYYMMDD. På så vis kan vi använda oss av en Radix-/CountingSort lösning liknande den innan. Att

konvertera ett datum till heltal förutsätter vi tar konstant tid ($O(1)$), så att konvertera alla n datum borde rimligen ha tidskomplexiteten $O(n)$. Vi bedömer att det är värt även denna overhead då listan är såpass stor att vi fortfarande tjänar på det. Tidskomplexiteten för lösningen kommer fortfarande vara $O(n)$.
Ex 2) Vi har redan beslutat oss för jämförelsebaserade algoritmer, så vi bortser från möjligheten att typkonvererta till heltal. Vi kan förutsätta att datumen är jämförbara, vilket gör att QuickSort fortfarande ett passande alternativ, i synnerhet kombinerat med Insertion-/Selection-/etc Sort. Eventuellt kan jämförelsen i sig vara något dyrare än för heltal, men baserat på vår motivation ovan anser vi fortfarande att QuickSort med SelectionSort för mindre partitioner är ett bättre alternativ. Vi bibehåller tidskomplexiteten $O(n \log n)$.
Ex 3) Vi anser att, även om vi kan, är det för omstädigt och dyrt att göra typkonvererertingar följt av räknesortering trots att vi använde RadixSort och CountingSort i a. Vi väljer istället att använda MergeSort som är speciellt bra på riktigt stora datamängder. När de individuella partitionerna är tillräckligt små övergår vi till SelectionSort då den är mer effektiv på små datamängder. MergeSort/SelectionSort kombinationen har en förväntad tidskomplexitet $O(n \log n)$.
(Vi har i samtliga fall gjort grundliga nya utvärderingar baserat på de nya förutsättningarna och antingen gjort nya val, med motivering, eller inte gjort nya val baserat på motivering. Full poäng).

2. **Uppgift 3**
   A: - QuickSort
   B: - InsertionSort
   C: - BubbleSort
   D: - SelectionSort

3. **Uppgift 4**

   (a) Exempel på lösningar:
   5 30 23 26 27 9 2 13
   30 5 23 26 27 9 2 13
   23 5 30 26 27 9 2 13
   2 23 5 39 26 27 9 13

   (b) **Alternativ 1:**
   Vi tar bort 23 och hashar om probing kedjan från borttaget index till första nullvärde:
   [30, null, 2, 13, null, 5, 26, 27, null, 9]
   Alternativ 2:
   Vi tar bort 23 och sätter en tombstone / delete token på den nyligen tömda platsen:
   [30, null, 2, X, 13, 5, 26, 27, null, 9]
   En poäng per korrekt svar.

4. **Uppgift 5**

   (a) Nej. Trädet är ett binärt träd (0-2 barn per nod) men ej ett binärt sökträd. För att det ska vara ett binärt sökträd krävs att alla noder i vänstra barnträdet från roten är mindre än roten, och alla noder i högra barnträdet är större än roten. Det samma gäller för samtliga noder i trädet utom de som inte har några barn.
   Rätt svar och stark motivering: 2p. Rätt svar och svag motivering: 1p. Fel / ingen motivering: 0p

   (b) Ja, då skillnaden i djup mellan lövnoder är maximalt 1/-1 (i fallet med detta träd är skillnaden i djup 0).
   Rätt svar och stark motivering: 2p. Rätt svar och svag motivering: 1p. Fel / ingen motivering: 0p

   (c) Ingetdera. Trädet är nästan en max-heap då den uppfyller att roten är större än båda sina barn, och det samma kan sägas för varje nod i trädet utom löv. Dock måste en heap vara komplett, och sålunda måste M vara ett vänsterbarn till T för att uppfylla kravet.
   Rätt svar och stark motivering: 2p. Rätt svar och svag motivering: 1p. Fel / ingen motivering: 0p

5. **Uppgift 6**

   (a) Exempel på lösning:
   A Y G F H T S M W X

   (b) Exempel på lösning:
   A Y H X F M G S W T

6. **Uppgift 7**

   (a) **Alternativ 1:** Först skapar vi en heap av elementen, förslagsvis i en array (max- eller min-heap, endera fungerar) där första elementet är roten följt av sina barn, osv. Vi sorterar sedan genom att ta bort max elementet (i fallet då vi använder en max-heap) ur heapen och stoppa in det bakifrån i resultat-arrayen och återställer heap egenskapen (bubblar ned den nyligen uppflyttade rotelementet tills vi åter har en korrekt max-heap). Om vi använder en min-heap gör vi i princip samma sak, förutom att vi stoppar in elementen framifrån i resultat-arrayen.
   Alternativ 2: Vi kör heapify på arrayen (sorterar den så att vi har en korrekt arrayimplementation av en max-heap). Sedan swap'ar vi max elementet (roten) till den sista positionen, och återställer heapegenskaperna.. Vi upprepar detta tills arrayen är sorterad för näst sista index, osv. tills arrayen är sorterad..
   (Endera förklaring anses vara korrekt och kommer att ge full poäng om den är väl genomtänkt / uttryckt).

   (b) $\Theta(n^2)$

   (c) $\Theta(\log n)$

   (d) $\Theta(n \log n)$