Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Magnus Nielsen och Filip Strömbäck

29 oktober 2018

TDDI16
Datastrukturer och algoritmer
Datortentamen (DAT1)
2018-10-29, 14–18
Lösningsförslag

Uppgift 2 – Sorteringsalgoritmer

1. Insertionsort (de första elementen är sorterade, resterande är på sin originalplats)
2. Heapsort (alla element är en heap förutom de sista, som är på korrekt plats)
3. Selectionsort (de första elementen är på rätt plats, resterande är ungefär på sin originalplats)
4. Quicksort (några av pivotelementen, exempelvis 25, 5 och 91, är på rätt plats)

Notera att motivering *ej* krävs för full poäng.

Uppgift 3 – Para ihop strumpor

Nedan finns två bra lösningsalternativ. Båda antar att strumporna representeras i form av heltal, där värdet på heltalet representerar strumpans färg. Exempelvis 1 = blå, 2 = grön, etc. Vi har också ett speciellt värde, -1, som inte representerar en färg, utan i stället markerar att strumpan redan är använd till något. Lösningarna kan också anpassas för att hantera andra datarepresentationer med bibehållen komplexitet om så önskas.

Alternativ 1

1. Lägg alla strumpor i en array $\mathcal{O}(n)$
2. Sortera strumporna med lämplig sorteringsalgoritm, exempelvis heapsort $\mathcal{O}(n \log(n))$
3. Iterera igenom den sorterade arrayen: n gånger
   (a) Jämför den nuvarande strumpan med nästa strumpa: $\mathcal{O}(1)$

   Om de är lika har vi hittat ett par. Räkna upp antalet hittade par, och sätt sedan båda strumporna till exempelvis -1 (vilket betyder ”ingen strumpa”) så att vi inte råkar räkna en strumpa som en del av två par.

Denna algoritm kräver totalt $\mathcal{O}(n) + \mathcal{O}(n \log(n)) + n\mathcal{O}(1) = \mathcal{O}(n \log(n))$ tid och $\mathcal{O}(n)$ minne (vi använder bara ett array med alla strumpor i).

Algoritmen kan göras effektivare om vi antar att vi har ett fåtal olika färger och använder exempelvis bucketsort i stället för quicksort. Då blir tidskomplexiteten i stället $\mathcal{O}(nk)$ där k är antalet färger som finns.

Alternativ 2

1. Skapa ett *set* (lämpligtvis en hashtabell), D, som kan innehålla färger. $\mathcal{O}(1)$
2. För varje strumpa: n gånger
   (a) Kontrollera om strumpans färg, k, finns i D: $\mathcal{O}(1)$ (amorterad)
   (b) Om den inte fanns: lägg till k i D $\mathcal{O}(1)$ (amorterad)
   (c) Annars: Räkna upp antalet hittade par, ta sedan bort k ur D $\mathcal{O}(1)$ (amorterad)

Denna algoritm kräver totalt $\mathcal{O}(1) + n\mathcal{O}(1) = \mathcal{O}(n)$ tid, och $\mathcal{O}(n)$ minne. I alla fall så länge vi antar att vi använder en vettig hashfunktion.

Ett balanserat sökträd kan också användas i stället för en hashtabell, då blir tidskomplexiteten $\mathcal{O}(n \log(n))$ i stället.

Uppgift 4 – Den rättvisa festen

Här finns återigen två exempellösningar. I båda lösningarna antar vi att rättvisekvotan i början lagras i en array R, och att de konsumerade dryckerna lagras i en array K. Antalet element i R och K noteras med r och k.

Alternativ 1

1. Sortera R med lämplig algoritm, exempelvis quicksort $\mathcal{O}(r \log(r))$
2. Sortera K med lämplig algoritm, exempelvis quicksort $\mathcal{O}(k \log(k))$
3. Initera två heltalsvariabler a och b till 0 $\mathcal{O}(1)$
4. Så länge $a < r$ och $b < k$: max. $n + k$ gånger
   (a) Om $R[a] = K[b]$: sätt $a = a + 1$ samt $b = b + 1$ $\mathcal{O}(1)$
   (b) Om $R[a] < K[b]$: ge gästen dryck $R[a]$, sätt $a = a + 1$ $\mathcal{O}(1)$
   (c) Om $R[a] > K[b]$: ska inte kunna hända, gästen har fått mer än rättvisekvotan
5. Så länge $a < r$: max. r gånger
   (a) Ge gästen dryck $R[a]$ $\mathcal{O}(1)$
   (b) Sätt $a = a + 1$ $\mathcal{O}(1)$

Loopen i steg 4 körs maximalt en gång per element i de båda arrayerna eftersom vi alltid ökar minst en av a och b. Loopen i steg 5 körs maximalt r gånger i det värsta fallet då K är tom. Totalt tar algoritmen alltså $\mathcal{O}(r \log(r)) + \mathcal{O}(k \log(k)) + (r + k)\mathcal{O}(1) + r\mathcal{O}(1)$. Vi kan också observera att $k \leq r$, vi kan således förenkla uttrycket genom att sätta $r = k$ (tidskomplexiteten blir då bara sämre, vilket är okej då vi använder $\mathcal{O}$): $\mathcal{O}(r \log(r)) + \mathcal{O}(r \log(r)) + 2r\mathcal{O}(1) + r\mathcal{O}(1) = \mathcal{O}(r \log(r))$. Notera också att detta är medelfallet för algoritmen, då vi har antagit att vi inte påverkas av värsta fallet för quicksort, som är $\mathcal{O}(n^2)$, vilket skulle ge ett västafall på vår algoritm på $\mathcal{O}(r^2)$. Detta kan vi dock sannolikt bortse från eftersom värstafallet för quicksort ytterst sällan inträffar i praktiken.

Minnesanvändningen är $\mathcal{O}(r + k) = \mathcal{O}(r \log(r))$ eftersom vi bara lagrar data i de två arrayerna R och K. I värsta fall behöver quicksort $\mathcal{O}(n)$ minne för att sortera n element, så detta påverkar inte den asymptotiska minnesanvändningen ($\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$).

Om vi vill effektivisera utlämningen av drycker med en konstant faktor kan vi ytterligare optimera vår algoritm genom att sortera R en gång i början i stället för varje gång vi ska lista ut vilka drycker en specifik gäst ska få.

Alternativ 2

1. Skapa en *dictionary* (implementerad med en hashtabell), D. $\mathcal{O}(1)$

   Nyckeln är en dryck och värdet är ett heltal

2. För varje dryck i R: r gånger
   (a) Om drycken finns i D, öka värdet med 1. $\mathcal{O}(1)$
   (b) Annars, sätt in drycken med värdet 1 i D. $\mathcal{O}(1)$
3. För varje dryck i K: k gånger
   (a) Minska värdet för drycken i D med ett. $\mathcal{O}(1)$
4. För alla element i D (uttryckt som d för dryck, och n för antal): max. r gånger
   (a) Ge gästen n stycken av drycken d. $\mathcal{O}(n)$

Loopen som körs i steg 4 kräver närmare analys. Här itererar vi igenom alla element i D, det vill säga en iteration för varje sorts dryck som finns på festen. Detta är maximalt r stycken. Inuti loopen ger vi gästen ett visst antal glas av en dryck, detta tar $\mathcal{O}(n)$ tid, $\mathcal{O}(1)$ för varje glas. n kan i detta fallet vara maximalt r, så till en början kan denna loop tyckas ta totalt $\mathcal{O}(r^2)$ tid. Så är dock inte fallet, eftersom vi vet att gästen kan få maximalt r stycken glas, det vill säga att summan av alla värden som finns i D är maximalt r. Alltså blir tidskomplexiteten för den sista loopen $\mathcal{O}(r)$.

Totalt blir alltså tidskomplexiteten: $\mathcal{O}(1) + r\mathcal{O}(1) + k\mathcal{O}(1) + \mathcal{O}(r)$. Om vi som i förra alternativet sätter $k = r$ eftersom vi vet att $k \leq r$ och eftersom vi undersöker övre gränsen så får vi: $\mathcal{O}(1) + 2r\mathcal{O}(1) + \mathcal{O}(r) = \mathcal{O}(r)$.

Minnesanvändningen är även här $\mathcal{O}(r)$ eftersom en hashtabell använder $\mathcal{O}(n)$ minne för att lagra n element. Eftersom vissa element är dubbletter, kommer vi dessutom ibland använda mindre minne eftersom.

Ett balanserat sökträd kan också användas för att representera D. Då blir dock tidskomplexiteten $\mathcal{O}(r \log(r))$ i stället för $\mathcal{O}(r)$.

Upptift 5

(a) Från hashtabellen kan vi se att 0 måste sättas in innan 20 och att 6, 7 och 8 måste sättas in innan 26. Exempelvis:

0, 20, 13, 25, 6, 7, 18, 26

13, 0, 20, 25, 6, 7, 18, 26

18, 7, 25, 6, 26, 0, 20, 13

18, 7, 6, 25, 13, 0, 20, 26

(b) Alternativ 1: Tombstones – markera det borttagna elementet med ”deleted”:

0	20	null	13	null	25	6	7	del	26

Alternativ 2: Ta bort det valda elementet, iterera igenom kvarvarande element (till *null* hittas) och se om de kan flyttas tillbaka till rätt position (ex.vis genom att anropa *insert* igen):

0	20	null	13	null	25	6	7	26	null

Uppgift 6

(a) $\mathcal{O}(n)$

(b) Iteration från n till 1 där n halveras varje iteration. Om vi tittar på iterationerna baklänges ser vi att för iteration $m - x$ (m är antalet iterationer) är $i = 2^x$. Antalet iterationer ges alltså av $n = 2^x \iff x = \log_2(n)$. Alltså: $\mathcal{O}(\log(n))$

(c) Här körs funktionerna `fun` och `more_fun` $n - 3$ gånger. Alltså får vi summan:

$$\sum_{i=3}^{n} \mathcal{O}(2i) + \mathcal{O}(\log(i))$$

Den kan förenklas genom att slå samman de båda ordo-uttrycken till följande:

$$\sum_{i=3}^{n} \mathcal{O}(i)$$

Vilket kan kännas igen som summan av alla heltal från 3 till n, vilket ger: $\mathcal{O}\left(\frac{n^2}{2}\right) = \mathcal{O}(n^2)$.

Alternativt kan man tänka att vi försöker beräkna arean av en triangel, vilket ger samma svar.

(d) En kvadrat av storlek k innehåller $2k^2 + 2k + 4k + 4 = 2k^2 + 6k + 4 \in \mathcal{O}(k^2)$ tecken (exklusive nyradstecken). Om vi skriver ut de n första storlekarna på kvadrater får vi följande komplexitet:

$$\sum_{k=1}^{n} \mathcal{O}(k^2)$$

Detta liknar uttrycket i förra uppgiften, men är svårare att lösa analytiskt. Här kan vi tänka oss att varje utskriven kvadrat bygger upp ett lager i en pyramid, och att vi försöker beräkna volymen av pyramiden. Resultatet blir alltså: $\mathcal{O}(n^3)$.

Summan kan beräknas exakt enligt följande (men det är inget krav):

$$\sum_{k=1}^{n} \mathcal{O}(k^2) = \mathcal{O}\left(\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}\right) = \mathcal{O}(n^3)$$

Uppgift 7

(a) Nej. I ett binärt sökträd måste varje nods vänsterbarn vara mindre än noden själv och högerbarnet vara större än noden själv. Detta uppfylls exempelvis inte i rotnoden.

(b) Ja. Höjden av vänster och höger delträd till varje nod skiljer sig med maximalt 1.

(c) Ingetdera. För att trädet ska vara en heap måste det vara komplett. Nod 13 måste vara ett vänsterbarn till 67 för att trädet ska vara komplett, men det är ett högerbarn.

Uppgift 8

(a) Nod 2 är startnoden, den läggs på kön först:

Kö: 2

Nod 2 besöks först. Därifrån hittar vi grannarna (i någon ordning): 1, 3 och 4

Kö: 1, 3, 4

Nod 1 besöks. Nod 2 är redan besökt. Inget mer händer.

Kö: 3, 4

Nod 3 besöks. Nod 2 är redan besökt, men inte nod 8.

Nod 8 är vårt mål, så sökningen avslutas. Vägen som hittats är: $2 \to 3 \to 8$

Detta ger inte alltid den kortaste vägen eftersom nodvikterna ignoreras. Vi får dock alltid vägen som besöker minst antal noder, men detta behöver inte vara den kortaste vägen.

(b) Här skriver jag avståndet till noden i parentes efter nodens nummer. Om jag kan komma till nod 8 med avstånd 5 skriver jag alltså 8(5).

Nod 2 är startnoden. Vi kan komma dit genom att gå 0 längdenheter:

Kö: 2(0)

Nod 2(0) besöks först. Därifrån hittar vi grannarna (i någon ordning): 1(2), 3(5) och 4(3).

Kö: 1(2), 4(3), 3(5)

Nod 1(2) besöks sedan. Därifrån hittar vi 2(4), men det är en längre väg än 2(0), så vi gör inget.

Kö: 4(3), 3(5)

Nod 4(3) besöks sedan. Därifrån hittar vi 2(6), 5(7) och 6(8). 2(6) läggs inte till, den har vi en bättre väg till.

Kö: 3(5), 5(7), 6(8)

Nod 3(5) besöks sedan. Här hittar vi 2(10) och 8(11). 2(10) läggs inte till.

Kö: 5(7), 6(8), 8(11)

Nod 5(7) besöks sedan. Här hittar vi 4(11), 7(9) och 8(11). 4(11) och 8(11) läggs inte till, det finns redan bättre (eller lika bra) vägar till dem.

Kö: 6(8), 7(9), 8(11)

Nod 6(8) besöks sedan. Nod 4(13) hittas, men läggs inte till.

Kö: 7(9), 8(11)

Nod 7(9) besöks sedan. Nod 5(11) hittas, men läggs inte till.

Nod 8(11) besöks. Detta är målet, så vi drar slutsatsen att den kortaste vägen är 11 enheter lång, och att den kortaste vägen här $2 \to 3 \to 8$.

Eftersom vi hittade en annan väg som var lika lång hade vi också kunnat svara $2 \to 4 \to 5 \to 8$ beroende på hur algoritmen är implementerad.

(c) För att hitta kortaste vägen $2 \to 7 \to 8$ så kan vi helt enkelt köra Dijkstras algoritm två gånger. Först hittar vi kortaste vägen mellan nod 2 och 7, sedan kortaste vägen mellan 7 och 8.