Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Filip Strömbäck

25 oktober 2019

TDDI16
Datastrukturer och algoritmer
Datortentamen (DAT1)
2019-10-25, 14–18 - Lösningsförslag

Uppgift 1 – Tidskomplexitet

(a) $\mathcal{O}(n)$. Förklaring: En loop med konstanttidsoperationer som körs n gånger.

(b) $\mathcal{O}(n)$. Förklaring: En loop med konstanttidsoperationer som körs maximalt n gånger ($\mathcal{O}(1)$ i bästa fallet).

(c) $\mathcal{O}(1)$. Förklaring: Värsta fallet som tar $\mathcal{O}(n)$ händer tillräckligt sällan för att tiden ska kunna fördelas mellan körningarna.

(d) $\mathcal{O}(n)$. Förklaring: Anropet till remove ser till att vi alltid kommer att exekvera bästa fallet för remove, tidskompexitet $\Omega(1)$. Notera att motivering *ej* krävs för full poäng.

Uppgift 2 – Träd

(a) Nej. Trädet är ett binärt träd då alla inre noder har 0-2 barn, men det är inte ordnat. I de flesta fall uppfyller det att en godtycklig nod har mindre nycklar till vänster, och större nycklar till höger, men inte alla. Noden med nyckeln 50 bryter detta då 55 och 52 ligger till vänster.

(b) Nej. Om 32 hade ett vänsterbarn hade trädet varit *komplett*. Ett *komplett* träd får inte ha några "luckor" mellan lövnoder / luckor i en levelordertraversering.

(c) 40-25-12-8-20-32-39-50-55-52-58 (behöver ej motiveras).

Uppgift 3 – Hashtabell

(a) Vi vet att 23 3 13 måste ha stoppats in i den ordningen, och 28, 8, 18 i den ordningen samt att 0 måste ha stoppats in innan 18..

- 6, 23, 3, 13, 28, 8, 0, 18
- 23, 3, 13, 6, 28, 8, 0, 18
- 23, 3, 13, 28, 8, 0, 18, 6
- 23, 6, 3, 13, 28, 8, 0, 18

(b) Vi har två alternativ: Omhashning och tombstone.

- Omhashning: Vi hashar om probingkedjan fram till det första *null* vi stöter på.

18	null	null	23	3	13	6	null	28	8

- Tombstone: Vi stoppar in en tombstone på den borttagna positionen, vilken tillåter oss att fortsätta söka förbi den positionen, eller skriva över den vid insättning.

X	18	null	23	3	13	6	null	28	8

Upptift 4 – Sorteringsalgoritmer

(a) 3

(b) 2

(c) 4

(d) 1

Notera att motivering *ej* krävs för full poäng.

Uppgift 5 – Falafelleverans

(a) Vi har en viktad graf så jag väljer att använda mig av Dijkstra's algoritm för att leta efter kortaste vägar, då Dijkstra's inte tar hänsyn till antalet noder för en kortaste väg, utan bara den totala vikten på bågarna vi passerar. Dijkstra's liknar en bredden-först-sökning. Vi har en prioritetskö för att hålla reda på noderna vi ska besöka. I varje iteration av algoritmen plockar vi den första noden från kön, vilken kommer vara den som är billigast hittills, och tittar på varje granne. För varje granne undersöker vi om vi har hittat en kortare väg dit, och lägger i så fall till den i prioritetskön. Eftersom vi har en sammanhängande graf (kan vi inte leverera till en plats är den inte intressant) vet vi att vi har tidskomplexitet $\mathcal{O}(E \log(N))$ där E är antalet bågar och N är antalet noder i grafen.

Jag vill hitta ett sätt att snabba upp sökningen och väljer därför att köra graftraverseringen från Falafelhuset till samtliga andra noder i förväg, och sedan lagra resultaten i en hashtabell. Varje nyckel är en nod i grafen, och värdet som lagras är snabbaste / bästa vägen till den noden. Denna lösning kommer att använda ganska mycket minne i förhållande till antalet noder, men vi bryr oss i detta fall bara om söktiden.

(b) Jag kan förutsätta perfekt hashning (unika namn, unika nodindex, etc i noderna) vad jag än hashar på (inom rimliga gränser). Därmed vet jag vet att jag garanterat kommer ha konstant tidskomplexitet vid uppslagning. $\mathcal{O}(1)$.

(c) Min lösning vore enkel att expandera för att lösa flera leveranser genom utöka min hashtabell så att jag lagrar alla noder som nycklar, och som värde en hashtabell med målnoder (alla andra noder i grafen) / vägen dit likt i a), och sedan köra en fullständig Dijkstra's från varje nod i grafen till alla andra noder i grafen vid uppstart och lagra resultaten. Denna lösning är dock mycket minnesintensiv, så jag beslutar mig för att det rimligare att köra Dijkstra's från Falafelhuset till första platsen de ska leverera till, och sedan en ny sökning från den platsen till nästa. Vi får då 2 sökningar, vardera med tidskomplexitet $\mathcal{O}(E \log(N))$. Den totala tidskomplexiteten blir $\mathcal{O}(E \log(N))$ eftersom vi inte tar hänsyn till konstanter.

Vill jag effektivisera lösningen ytterligare kan jag använda båda lösningsidéerna: Jag använder tabellen från a) för att hitta kortaste vägen till och från falafelhuset, men jag gör en grafsökning för att hitta kortaste vägen mellan det första och det andra leveransstället. På så vis får jag fördelarna av tabellen i a) utan att behöva använda $\mathcal{O}(n^2)$ minne. Denna lösning har dock samma tidskomplexitet som tidigare, $\mathcal{O}(E \log(N))$.

Uppgift 6 – Reseplaneraren

(a) Jag inser att jag kan behandla det som ett viktat grafproblem. För att representera grafen väljer jag att använda en hashtabell, eftersom de enkelt kan representera en graf och har snabb uppslagning, med strängar (namnen på samtliga startnoder) som nycklar mot en (array)lista av tupler/par (int, String). Jag väljer en dynamisk lista då jag inte vet hur många platser jag kan nå från en given startpunkt, och tupeln representerar priset för en resa samt namnet på destinationen.

Eftersom det är en viktad graf kan vi enkelt köra en Dijkstra's för att hitta den billigaste möjliga resan då den billigaste resan är samma sak som kortaste väg i en viktad graf. Dijkstra's bryr sig inte om hur många noder vi passerar, utan bara den totala vikten / kostnaden. Dijkstra's liknar en bredden-först-sökning. Vi har en prioritetskö för att hålla reda på noderna vi ska besöka. I varje iteration av algoritmen plockar vi den första noden från kön, vilken kommer vara den som är billigast hittills, och tittar på varje granne. För varje granne undersöker vi om vi har hittat en kortare väg dit, och lägger i så fall till den i prioritetskön.

(b) Vi kommer att göra $\mathcal{O}(N)$ insättningar, tidskomplexitet $\mathcal{O}(N)$ (amorterat). Insättning i dynamiska listtyper kan ta $\mathcal{O}(N)$ i värsta fallet men amorterad tidskomplexitet för operationen är konstant.

För Dijkstra's har vi tidskomplexiteten $\mathcal{O}(N \log(P))$.

Sammanlagd tidskomplexitet för lösningen: $\mathcal{O}(N \log(P))$.

(c) Jag väljer att ändra heltalet (int) som motsvarade bågvikten ovan till tupel/par <int, int>. Det första värdet är en "faktor" (ex. 1 för all utom buss, 2 för buss), det andra är priset. Vi modifierar Dijkstra's så att den vid traverseringen jämför lexikografiskt: i första hand på "faktorn", i andra hand på pris. Vi får som resultat att bussresor undviks under förutsättning att det alls går. Tidskomplexiteten är oförändrad.

Uppgift 7

(a) Alla måltider stoppas in i en array av längd n som jag sedan sorterar för att underlätta sökning. Jag sorterar sedan arrayen mha QuickSort / SelectionSort på priset ($\mathcal{O}(n\log(n))$.

Man vill ju inte söka linjärt i arrayen. Binärsökning börjar i mitten, om vi söker en billigare måltid upprepar vi samma sak för index [0, n/2-1], om vi söker en dyrare gör vi samma sak för index [n/2+1, n]. Binärsökningen fortsätter halvera sökområdet tills vi har rätten vi söker. Det verkar mycket bättre än linjär sökning, så jag väljer det. Om det exakta priset som budgetterats saknas tar jag det närmast billigare alternativet.

Detta fungerar lika väl för många hungriga turister som för bara mig.

(b) Sorteringen tar $\mathcal{O}(n\log(n)$.

Jag gör antagandet att vi inte vill, eller för den delen kan (vi är många, $k \approx n!$) äta på samma restaurang, så vi struntar i vilken restaurang var och en får.

Även binärsökningen har tidskomplexitet $\mathcal{O}(\log(n))$, och vi har $k \approx n$ turister, vilket innebär att den totala lösningen har tidskomplexiteten $k\log(n), k \approx n \implies \mathcal{O}(n\log(n))$ (sorteringen görs bara en gång).

(c) Jag gör ett antagande:

- Jag vill inte äta samma rätt på samma restaurang (jag kan tänka mig att äta samma rätt på en annan restaurang).

Insättning och sortering görs på samma sätt som i del a.

Steg 0: Deklaration av två variabler: int $bestA = 0$ och int $bestB = 0$.

Steg 1: Jag itererar fram till den dyraste rätten jag har råd med, a. Därefter söker jag med binärsökning fram den dyrate rätten b jag kan få som uppfyller $a \neq b$ och priset för $b <$ budget - priset för a.

- Om det sammalagda priset inte är exakt min budget: dessa sparas undan som $bestA$ och $bestB$ förutsatt att priset för $a + b >= bestA + bestB$,
- Om priset för båda rätterna är exakt min budget har jag en pefekt matching.

Steg 2-n: Upprepa steg 1 för den dyraste rätten vi ännu inte undersökt (1 steg i loopen).

(d) Insättning och sortering tar, som tidigare, $\mathcal{O}(n\log(n))$.

Jag kommer maximalt göra maximalt n stycken binärsökningar (totalt $\mathcal{O}(n\log(n))$).

Total tidskomplexitet: $\mathcal{O}(n\log(n))$

Uppgift 8

(a) Jag väljer att använda en hashtabell för att lösa problemet, med datum som nyckel och en int som värde.

För att lösa problemet itererar jag igenom datamängden ($\mathcal{O}(n)$). För varje transaktion:

- Datumet finns ej i hashtabellen ännu: Jag stoppar in datumet i transaktionen som nyckel med priset som värde.
- Datumet finns i hashtabellen: Jag adderar priset på transaktionen till datumets plats i hashtabellen.

Väl instoppade i hashtabellen kan jag enkelt plocka ut summan för varje dag.

(b) Insättningen i hashtabellen tar $\mathcal{O}(1)$ tid. Vi har n stycken nycklar vilket ger totalt $\mathcal{O}(n)$ för insättning och summering.

Totalt: $\mathcal{O}(n)$

(c) Jag väljer att utöka min hashtabell från del a, men istället för att summera varje datum har jag på varje datum ytterligare en hashtabell där jag gör insättning baserat på försäljarens namn mot en int (precis som tidigare). Nu har jag en sortering / partitionering baserat först på datum, sedan på försäljare. Jag kan nu upprepa algoritmen från del a, förutom att jag stoppar in / summerar på både datum och försäljare.

Även här har vi $\mathcal{O}(n)$ tid då vi fortfarande bara har n stycken transaktioner att summera, och uppslagning / insättning i hashtabell tar $\mathcal{O}(1)$ tid.

Jag skapar en hashtabell *result* som mappar försäljare mot int (vinsträknare).

För varje datum i hashtabellen itererar jag igenom varje försäljare på det datumet och hittar den mest högst summa för dagen:

- Om fösäljaren ej finns i *result*: Stoppa in försäljaren tillsammans med värdet 1 i *result* då vi har hittat den första vinstdagen.
- Om försäljaren finns i *result*: Räkna upp vinsträknaren med 1.

Total tidskomplexitet i detta steg: $\mathcal{O}(n)$ (vi har n transaktioner och kan omöjligtvis behöva titta på fler par av datum-försäljare än vi har transaktioner).

Jag iterar sedan igenom *result* för att hitta den försäljare som har högst vinsträknare.

Total tidskomplexitet: $\mathcal{O}(n)$