Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Filip Strömbäck och Magnus Nielsen

28 oktober 2022

Datastrukturer och algoritmer Lösningsförslag till tentamen 2022-10-28

1. (a) Exempelvis:
   16, 28, 19, 0, 8, 2, 22, 13
   2, 22, 13, 19, 28, 0, 16, 8
   0, 2, 22, 13, 19, 28, 8, 16
   2, 19, 28, 22, 13, 0, 16, 8

   (b) Alternativ 1: Ersätt 28 med en tombstone (efter vanlig uppslagning för att hitta det). Detta element fungerar som ett null-element i beaktningen att det beskriver att inget element finns på den givna platsen. Däremot skiljer den sig från null i och med att uppslagningen inte kan avslutas när en tombstone hittas, precis som när ett ”vanligt” element hittas.

   Detta ger: 0, 8, 2, 22, 13, Null, 16, Null, Tombstone, 19

   Alternativ 2: Ta bort elementet (efter vanlig uppslagning för att hitta det). Fortsätt att iterera igenom arrayen tills ett null-element hittas. För varje element, ta bort elementet från arrayen och sätt in det genom att anropa ”insert”. Detta gör så att element som behöver flyttas hamnar på rätt plats.

   Detta ger: 0, Null, 2, 22, 13, Null, 16, Null, 8, 19

2. 1-d, 2-a, 3-c, 4-b

3. (a) 1: Insättning tar amorterad konstant tid i en dynamisk array. Alltså: $\mathcal{O}(1)$

   2: Insättning i heap består av insättning sist i en dynamisk array (amorterad konstant tid), sedan återställande av heap-egenskapen, vilket tar $\mathcal{O}(\log n)$ tid. Totalt: $\mathcal{O}(\log n)$.

   3: Insättning i en kö tar $\mathcal{O}(1)$ tid. Om kön är implementerad som en cirkulär dynamisk array är detta den amorterade tiden. Är den implementerad med en länkad lista är det $\mathcal{O}(1)$ (ej amorterad).

   (b) 1: Borttagning kräver sortering, vilket tar $\mathcal{O}(n \log n)$ i det väntade fallet (vi antar en bra implementation av quicksort där vi sällan ser $\mathcal{O}(n^2)$-beteendet). Borttagning tar sedan $\mathcal{O}(n)$ tid. Totalt: $\mathcal{O}(n \log n)$.

   2: Det största elementet finns i toppen av heapen, och kan därmed kommas åt på $\mathcal{O}(1)$ tid. Borttagning kräver dock att heapegenskapen återställs, vilket tar $\mathcal{O}(\log n)$ tid. Totalt: $\mathcal{O}(\log n)$.

   3: Att undersöka de tre köerna tar konstant tid. Borttagningen ur en kö tar sedan $\mathcal{O}(1)$ tid, oavsett hur kön implementeras.

   (c) Nej. Alternativ 1 och 2 fungerar inte korrekt. Båda har problemet att sorteringen (quicksort eller sorteringen som görs av heapen) inte är stabila. Detta gör att alternativ 1 och 2 endast garanterar att de med högst prioritet kommer ut först, men det är inte garanterat att den inbördes ordningen mellan användarna bevaras. Beroende på implementationen av alternativ 1 kan det också vara så att prioriteringen av användarna blir omvänd (om sorteringen sker i stigande ordning, och inte i fallande ordning).

4. (a) Loopen körs från $i = 3$ till $i = n$, och i dubbleras varje gång. Loopen körs alltså tills $3\dot{2}^x \geq n$ där x är antalet iterationer. Detta ger att $x \in \mathcal{O}(\log n)$, och tidskomplexiteten blir därmed $\mathcal{O}(\log n)$.

   (b) Här finns två nästlade loopar. Den yttre kör n^2 gånger (i går från 1 till n^2 och ökas med 1 varje gång). Den inre kör n gånger av samma anledning. Detta ger totalt: $\mathcal{O}(n^2\dot{n}) = \mathcal{O}(n^3)$.

   (c) Detta är en rekursiv funktion som anropar sig själv tills n blir 0. Den går att skriva om som följande:

```
int funnieast(int n) {
  int result = 0;
  while (n > 0) {
    n = n / 2;
    result++;
  }
  return result;
}
```

   Från denna omskrivning kan vi se att loopvariabeln går från n till 0 och halveras varje gång. Likt i (a) ger detta en tidskomplexitet på $\mathcal{O}(\log n)$.

   (d) Här finns två extremfall. Antingen körs koden i if-satsen en gång per iteration (om elementen är i fallande ordning), eller så körs den aldrig (om elementen är i stigande ordning, eller lika).

   Om if-satsen körs varje gång kommer loopen att köras $n/2$ gånger (eftersom ungefär hälften av elementen tas bort). Remove-operationen tar $\mathcal{O}(x-i)$ tid där i är det element som tas bort, och x är antalet element kvar i arrayen. Första gången kommer alltså $n-1$ element behöva flyttas, nästa gång $n-3$, sedan $n-5$, och så vidare. Totalt får vi alltså:

$$\mathcal{O}\left(\sum_{i=0}^{n/2} n - 2i\right) = \mathcal{O}(n^2) \tag{1}$$

   Geometriskt kan man tolka detta som arean av en triangel med ena sidan $n/2$ och andra sidan n, vilket ger en area av $n^2/4 \in \mathcal{O}(n^2)$.

   Om if-satsen aldrig körs så kommer bara jämförelsen i yttre loopen att köras n gånger.

   Totalt får vi alltså att bästa fallet är: $\mathcal{O}(n)$ och värsta fallet är $\mathcal{O}(n^2)$.

5. (a) Ett lämpligt sätt att lagra datan som ska skickas till skärmen är exempelvis följande:

```
struct Transaction {
  int id;
  Date date;
  int price;
  string seller;
  string description;
};

struct Seller_Info {
  int total_price = 0;
  vector<Transaction> transactions;
};

unordered_map<string, Seller_Info> info;
```

Nyckeln i hashtabellen (`info`) är namnet på säljaren, vilket vi antar är unikt (det finns inget annat sätt att identifiera säljare).

Vi antar att vi till vårt program får in data i en `vector<Transaction>` (eller liknande). Då kan vi fylla i datastrukturen `info` enligt följande;

```
unordered_map<string, Seller_Info> data_to_screen(const vector<Transaction> &table) {
  unordered_map<string, Seller_Info> info;
  for (const Transaction &t : table) {
    info[t.seller].total_price += t.price;
    info[t.seller].transactions.push_back(t);
  }
  return info;
}
```

(Notera: vi kan undvika dubbel uppslagning i hashtabellen med exempelvis en iterator eller en referens till elementet. För tydlighet har jag valt att inte göra detta här.)

(b) Givet att hashtabellen använder en bra hashfunktion tar uppslagning och insättning i en hashtabell amorterad konstant tid. Insättning i `transactions` inuti `Seller_Info` tar också amorterad konstant tid. Eftersom vi uppdaterar hashtabellen en gång för varje rad i tabellen vi får som indata får vi totalt: $\mathcal{O}(n)$

(c) Datastrukturen lagrar alla transationer, alltså får vi $\mathcal{O}(n)$ minne. Eftersom vi skapar en `Seller_Info` för varje säljare kan vi indikera detta genom att skriva $\mathcal{O}(n + k)$ där k är antal säljare. Dock vet vi att $k \leq n$ (vi känner bara till de säljare som finns i tabellen), så detta är samma sak som $\mathcal{O}(n)$

(d) Detta kan göras med en min-heap enligt följande:

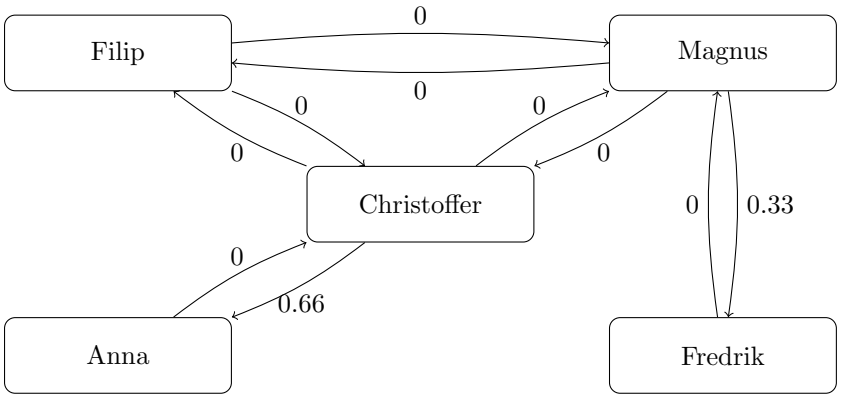
- Skapa en min-heap, h. Heapen lagrar transaktioner (`Transaction`), och sorterar på pris.
- För varje element, t, i tabellen:
  - Om h innehåller fler än k element, ta bort min-elementet från h.
  - Lägg till transaktionen t i heapen.
- Ta sedan bort min-elementet i heapen tills heapen är tom. Lagra dessa element i exempelvis en array.
- Returnera arrayen, den innehåller nu en sorterad lista av de topp k dyraste transaktionerna. Beroende på hur arrayen fylls kan den vara sorterad bakvänt, men detta går att lösa i $\mathcal{O}(k)$ tid.

Totalt tar detta $\mathcal{O}(k)$ minne, då vi maximalt lagrar k element i heapen, och ingen av heap-operationerna kräver extra minne.

Detta tar $\mathcal{O}(n \log k)$ tid, då heap-operationerna alla tar $\mathcal{O}(\log k)$ tid, och vi gör åtminstone en heap-operation för alla n element i tabellen. Att plocka ut elementen ur heapen tar sedan $\mathcal{O}(k \log k)$ tid, vilket är mindre än $\mathcal{O}(n \log k)$.

6. (a) Detta kan ses som en riktad och viktad graf. Noderna motsvarar personer. Bågarna motsvarar kontakter. Bågar finns mellan alla personer som har kontakt med varandra. Vikten på bågarna ges av formeln i uppgiften, och beror på kompetenserna hos noden som bågen går till. Vikten på bågen från *Filip* till *Christoffer* ges alltså av *Christoffers* kompetenser, men bågen åt andra hållet skulle ges av *Filips* komptenser.

Grafen i exemplet ser alltså ut som följer:



Att beräkna TrustMultiplier™mellan två personer är alltså ekvivalent med att hitta kostnaden av den kortaste vägen i grafen. Då grafen är viktad och inte har negativa cykler kan vi göra detta med Dijkstras algoritm. Till detta kan man använda följande datastrukturer:

```
struct Node {
  double multiplier;
  vector<string> edges;
  double best_weight;
};

unordered_map<string, Node> graph;
```

Grafen fylls sedan i enligt följande:

- Lägg alla efterfrågade kompetenser i en hashtabell (`unordered_set`) ($\mathcal{O}(k)$, k är antalet kontakter)
- För varje person i listan över personer: (p gånger)
  - Skapa en `Node`-instans och lägg till i hashtabellen `graph`. ($\mathcal{O}(1)$)
  - Beräkna `multiplier` för noden genom att iterera igenom personens kompetenser och se hur många som inns i hashtabellen med efterfrågade kompetenser. Detta ger k i formeln, och vi kan sedan beräkna s för personen. ($\mathcal{O}(k)$, k är antalet kompetenser)
  - Sätt `best_weight` till ∞.
- För varje rad i kontaktgrafen: (e gånger)
  - Slå upp båda personerna i `graph` ($\mathcal{O}(1)$)
  - Lägg till bågar åt båda hållen ($\mathcal{O}(1)$)

Efter detta kan vi köra Dijkstra's algoritm:

- Skapa en prioritetskö pq som innehåller par: (vikt, nodnamn). Kön sorteras på minskande vikt och implementeras med en heap.
- Lägg till examinatorn i pq. Sätt `visited` i examinatorns nod till `true`.
- Så länge det finns element i pq:
  - Ta bort första elementet, lagra vikten som w och noden som n.
  - Om `best_weight` för noden n inte är samma som w, hoppa över resten av loopen.
  - Om n är noden för den efterfrågade personen, returnera w.
  - För alla bågar från n till m:
    - Låt w' vara summan av w och `multiplier` i noden m.
    - Om w' är mindre än `best_weight` i m: sätt `best_weight` till w' och sätt in (w', m) i pq.
- Om pq blivit tom finns ingen väg från examinatorn till kandidaten. Returnera i så fall exempelvis ∞ eller rapportera ett fel.

(b) Att bygga grafen tar totalt $\mathcal{O}(p + e)$ tid (hur antal kompetenser påverkas efterfrågas ej, tar man hänsyn till den får vi $\mathcal{O}(kp + e)$, där k är medeltalet av efterfrågade kompetenser).

Att köra Dijkstra's algoritm tar sedan: $\mathcal{O}((p+e)\log p)$ tid, vilket är den dominerande termen.

(c) Minnesanvändningen är $\mathcal{O}(p+e)$ för grafen. Dijkstra's algoritm tar sedan ytterligare $\mathcal{O}(p+e)$ minne i värsta fall som den är skriven ovan (det går att lösa på $\mathcal{O}(p)$ minne, men kräver en bättre prioritetskö). Oavsett får vi totalt: $\mathcal{O}(p + e)$

(d) Om vi har en lista med n kandidater skulle det ta $\mathcal{O}(n(p + e)\log p)$ tid att köra n anrop till funktionen. Det går dock att lösa på $\mathcal{O}((p + e)\log p)$ tid genom att ta bort villkoret som avslutar algoritmen och i stället låta den köra tills pq blivit tom. Då finns TrustMultiplier™för alla kandidater i `best_weight` i noden för varje person.

(Tekniskt sett tar detta $\mathcal{O}(n + ((p + e)\log p))$ men eftersom $n \leq p$ så kan vi förenkla enligt ovan)