Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Filip Strömbäck och Magnus Nielsen

28 oktober 2022

Datastrukturer och algoritmer Lösningsförslag till tentamen 2023-10-27

1. Rad 1: a
   Rad 2: b
   Rad 3: finns ej i listan
   Rad 4: d
   Rad 5: c

   Den sista algoritmen är heapsort. Det kan man exempelvis se genom att de sista 3 elementen är på rätt plats, och att resten av arrayen är i form av en heap (77 är det stösta kvarvarande elementet, det är först).

2. (a) Nej, det är inte ett binärt sökträd. Trädet är ett binärt träd. Det är dock inte ett sökträd i och med att villkoret att för alla delträd ska alla vänsternoder vara mindre än roten och alla högerträd vara större än roten. Det uppfylls exempelvis inte i och med att 7 är större än 3.

   (b) Ja. Det är komplett i och med att det inte finns några ”hål” när vi gör en level-order traversering av trädet.

   (c) Ja. Trädet är komplett. Desutom gäller villkoret att för alla noder är båda nodens barn mindre än noden själv.

   (d) 3, 7, 8, 15, 20, 12, 28, 42, 21, 38, 42, 22

3. (a) För alternativ 1:

   Vi måste traversera hela sökträdet eftersom namnet inte är nyckeln: $\mathcal{O}(n)$. Sedan kan vi sätta in elementet genom vanlig insättning följt av balansering: $\mathcal{O}(\log n)$. Totalt: $\mathcal{O}(n)$

   För alternativ 2:

   Först slår vi upp namnet i hashtabellen. Detta tar $\mathcal{O}(1)$ (amorterad) tid, givet en bra hashfunktion. Vi lägger sedan in elementet i heapen, vilket tar $\mathcal{O}(\log n)$ tid att ”bubbla upp” elementet. Totalt: $\mathcal{O}(\log n)$

   (b) För alternativ 1:

   Vi hittar det minsta elementet genom att traversera trädet från rotnoden längs vänsterbarnen så långt det går. Detta tar $\mathcal{O}(\log n)$ tid då trädet är balanserat.

   För alternativ 2:

   Det minsta elementet finns i toppen av min-heapen. Vi kan därmed plocka ut det direkt i $\mathcal{O}(1)$ tid.

   (c) För alternativ 1:

   Likt i uppslagning följer vi `left`-pekarna så långt vi kan, tar bort noden, och balanserar om trädet på vägen uppåt. Detta ger $\mathcal{O}(\log n)$ tid då trädet är balanserat.

   För alternativ 2:

   Vi plockar bort elementet från min-heapen genom att byta det med det sista elementet. Sedan återställer vi heap-egenskapen genom att ”bubbla ner” toppelementet. Detta tar $\mathcal{O}(\log n)$ tid. Vi tar också bort namnet från hashtabellen, vilket tar $\mathcal{O}(1)$ tid givet en bra hashfunktion. Totalt: $\mathcal{O}(\log n + 1) = \mathcal{O}(\log n)$

   (d) Alternativ 2 är bäst. Sätter vi in n element i alternativ 1 så tar det $\mathcal{O}(n \cdot n) = \mathcal{O}(n^2)$ tid, medan i alternativ 2 tar det bara $\mathcal{O}(n \log n)$ tid.

   Utöver detta är alternativ 2 bättre på att hitta toppelementet. Det har dock den lilla nackdelen att det använder lite mer minne (inte asymptotiskt, dock) i och med hashtabellen.

(e) Vi kan lösa detta genom att modifiera hashtabellen så att den i stället för att lagra bara namn, så lagrar den *namn* → *positioniheap*. Detta gör så att vi vid insättning kan slå upp studenten i hashtabellen och hitta var den ligger i heapen i $\mathcal{O}(1)$ tid. Vi kan sedan bara "bubbla ner"' elementet i heapen från dess nuvarande position för att återställa heapegenskapen. Detta tar $\mathcal{O}(\log n)$ tid totalt.

Vi måste då också modifiera våra heapoperationer så att de uppdaterar hashtabellen. Detta går att göra: heapen lagrar redan namn, så att vi kan med hjälp av namnet där enkelt hitta och uppdatera rätt plats i hashtabellen.

4. (a) Två nästlade loopar. Den yttre kör n gånger, den inre kör $n - i$ gånger.
Detta ger: $\mathcal{O}\left(\sum_{i=0}^{n} i\right) = \mathcal{O}\left(\frac{n(n+1)}{2}\right) = \mathcal{O}(n^2)$

(b) Loopen går från n till 0, och i halveras varje gång. Det ger $\mathcal{O}(\log n)$. Sedan anropas $f(n)$, vilket ger $\mathcal{O}(n^2)$ enligt ovan.
Totalt: $\mathcal{O}(\log n + n^2) = \mathcal{O}(n^2)$

(c) Vi har en loop från 0 till n. Detta kör $f(m)$ n gånger.
Alltså: $\mathcal{O}(n \cdot m^2)$

(d) I bästa fall:

Vi hittar elementet tidigt (ex. det första elementet) och returnerar direkt: $\mathcal{O}(1)$.

I värsta fall:

Vi hittar inte elementet och kör sedan `push_back`. Detta tar $\mathcal{O}(n)$ för att söka, och $\mathcal{O}(n)$ för `push_back` (värsta fallet). Totalt: $\mathcal{O}(n + n) = \mathcal{O}(n)$

5. (a) Exempelvis kan vi använda en hashtabell för att ta bort dubbletter från arrayen för varje klipp:

```
vector<int> compute_laughs(const vector<vector<int>> &input, int p) {
  // Initiera utdata till 0. O(p) tid
  vector<int> output(0, p);

  // För varje klipp:
  for (const vector<int> &clip : input) {
    // Kopiera skratten till en hashtabell: O(s) tid, totalt
    unordered_set<int> laughs(clip.begin(), clip.end());

    // Spara antalet: O(s) tid, totalt.
    for (int laughed : laughs) {
      output[laughed]++;
    }
  }

  return laughed;
}
```

(b) Givet att vi har en bra hashfunktion tar operationerna på hashtabellen $\mathcal{O}(1)$ tid (dvs. $\mathcal{O}(s)$ tid att sätta in s element). Totalt kommer vi att sätta in s element i hashtabellen, och iterera igenom maximalt s element i loopen `Spara antalet` (lite färre på grund av dubbletter). Detta ger oss totalt: $\mathcal{O}(s + p)$ (i och med att vi måste initiera utdataarrayen också).

(c) Det enda extra minnet vi behöver är `laughs`. Den innehåller maximalt s element (ofta mycket mindre). Alltså $\mathcal{O}(s)$. Räknar vi också med utdatan får vi $\mathcal{O}(s + p)$.

6. (a) Uppgiften beskriver en topologisk sortering av tabellen. Vi behöver bara tänka på kolumnerna *ID* och *Måste ätas efter*.

   Vi antar att grafen lagras i följande struktur:

```
struct Node {
  // Namn, för utskrift senare.
  string name;
  // Restaurang, för att hitta rätt senare.
  string restaurant;

  // Beroenden.
  vector<int> dependencies;

  // För algoritmen, initieras till false.
  bool visited = false;
}
```

   Här kan vi alltså se att vi representerar varje rätt som en nod i en graf. Bågarna representerar beroenden. Hela grafen lagras som en array av `Node`. Till skillnad från ”klassiska” topologiska sorteringar så går bågarna i omvänd riktning. Vi vill alltså hitta en sortering där a kommer före b om det finns en båge från b till a. Detta gör bara lösningen enklare dock.

   Vi kan lösa problemet med en DFS ungefär som nedan:

```
vector<Node> find_order(vector<Node> &graph) {
  vector<Node> result;

  for (size_t i = 0; i < graph.size(); i++) {
    find_recursive(result, graph, i);
  }

  return result;
}

// Rekursiv hjälpfunktion.
void find_recursive(vector<Node> &result, vector<Node> &graph, size_t node) {
  if (graph[node].visited)
    return;
  graph[node].visited = true;

  // Traversera beroenden, lägg till dem först.
  for (int dep : graph[node].dependencies) {
    find_recursive(result, graph, dep);
  }

  // Lägg till noden själv.
  result.push_back(graph[node]);
}
```

   (b) Detta tar $\mathcal{O}(n+m)$ tid: vi traverserar alla bågar och alla noder ett konstant antal gånger.

(c) Uppgiften beskriver ett kortaste-vägen-problem. Vi kan därmed lösa det med en BFS. Vi använder samma grafrepresentation som i uppgiften innan. I och med att vi inte vet var vi ska, så börjar vi i noden k och slutar så snart vi hittar en nod utan beroenden.

```
vector<Node> find_order(vector<Node> &graph, int k) {
  queue<int> to_visit;
  unordered_map<int, int> previous;
  previous[k] = -1;
  to_visit.push(k);

  while (!to_visit.empty()) {
    int current = to_visit.front();
    to_visit.pop();

    // Om noden är tom, är detta den närmsta rätten utan beroenden.
    if (current.dependencies.empty()) {
      vector<Node> result;
      for (int i = current; i > 0; i = previous[i]) {
        result.push_back(graph[i]);
      }
      return result;
    }

    for (int child : current.children) {
      if (previous.count(child))
        continue;

      previous[child] = current;
      to_visit.push(child);
    }
  }

  // No solution.
  return vector<Node>();
}
```

(d) Tidskomplexiteten blir $\mathcal{O}(n + m)$, vi behandlar varje nod och varje båge ett konstant antal gånger.