Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Filip Strömbäck och Magnus Nielsen

2025-10-30

Datastrukturer och algoritmer Lösningsförslag till tentamen 2025-10-30

1. (a) quicksort — rad 1

   (b) bubble sort — rad 4

   (c) insertionsort — rad 2

   (d) selectionsort — rad 5

   (e) Den sista algoritmen är merge sort. Det ses ex.vis genom att vi har sekvenser om 4 element som är inbördes sorterade.

2. (a) Traverseringen blir som följer. Siffra inom parentes anger varifrån vi kom till noden som besöks:

   1(-) (alla vägar med 0 hopp hittade)
   $\Rightarrow$ 4(1) $\Rightarrow$ 5(1) (alla vägar med 1 hopp hittade)
   $\Rightarrow$ 6(5) $\Rightarrow$ 3(5) (alla vägar med 2 hopp hittade)
   $\Rightarrow$ 2(6) (målet hittat, traverseringen bryts)

   (b) Vi kan utnyttja siffran inom parentes för att hitta vägen: $1 \Rightarrow 5 \Rightarrow 6 \Rightarrow 2$

   (c) Traverseringen blir som följer. Notationen är: nod(nuv. avstånd, föregående)

   Nod 1(0, -) besöks: uppdaterar 4(3, 1), 5(6, 1)
   Nod 4(3, 1) besöks: uppdaterar 5(inga ändringar)
   Nod 5(6, 1) besöks: uppdaterar 6(7, 5), 3(11, 5)
   Nod 6(7, 5) besöks: uppdaterar 3(9, 6), 2(13, 6)
   Nod 3(9, 6) besöks: uppdaterar 2(11, 3)
   Nod 2(11, 3) besöks: vi är klara

   (d) Enligt siffra inom parentes får vi kortaste vägen: $1 \Rightarrow 5 \Rightarrow 6 \Rightarrow 3 \Rightarrow 2$

3. (a) Loop som går från 1 till $2n$, men där i dubbleras varje gång. i kan uttryckas som: $i = 2^x$ där x räknar antal iterationer från 0. Vi får antal iterationer genom att sätta: $i = 2n$: $2n = 2^x \iff x = \log(2n) = \log(n)$. Alltså: $\mathcal{O}(\log(n))$

   (b) Rekursiv funktion där n minskar ett steg för varje rekursivt steg. Avslutas när $n = 0$. Alltså: $\mathcal{O}(n)$

   (c) Notera att $t = n \cdot m$. Loopen körs $n - 1$ gånger, och i loopen anropas `f(t)`. Enligt (a) får vi alltså: $\mathcal{O}((n-1) \cdot \log(n \cdot m)) = \mathcal{O}(n \log(nm))$

   (d) Bästa fallet är exempelvis när `arr` innehåller n stycken tal som är $\geq$ `minValue`. Då kommer loopen i `findInterval` köras en gång för varje anrop. Då blir den totala tiden för `shortestValuableInterval` $\mathcal{O}(n)$.

   Värsta fallet är exempelvis när `arr` innehåller n stycken 1:or, och `minValue` $\geq$ `arr.size()`. Då kommer loopen i `findInterval` alltid att gå till slutet av arrayen, vilket tar $\mathcal{O}(x)$ tid (x är antal kvarvarande element). Gör vi detta n gånger får vi: $\mathcal{O}\left(\sum_{x=0}^{n} x\right) = \mathcal{O}(n^2)$

4. (a) Alternativ 1: Lägga in kakan i slutet av en dynamisk array (`push_back` i `vector`) tar $\mathcal{O}(1)$ (amorterad) tid, men $\mathcal{O}(k)$ i värsta fall. Insertionsort tar vanligtvis $\mathcal{O}(k^2)$ tid, men i det här fallet kommer hela arrayen vara sorterad förutom sista kakan. Det gör att insertionsort får sitt bästa fall på $\mathcal{O}(k)$. Totalt blir det alltså: $\mathcal{O}(k)$

   Alternativ 2: Insättning i heap består av: insättning i den dynamiska arrayen som ovan (vilket är $\mathcal{O}(1)$ i de flesta fall, men $\mathcal{O}(k)$ i värsta fallet), följt av en uppbubbling i heapen, vilket tar $\mathcal{O}(\log(k))$. Totalt får vi alltså en amorterad tid på $\mathcal{O}(\log(k))$, men ett värsta fall på $\mathcal{O}(k)$.

   (b) Alternativ 1: Ta bort kakan från slutet av arrayen (`pop_back`) tar $\mathcal{O}(1)$ tid (man brukar inte krympa allokeringen vid borttagning).

Alternativ 2: Borttagning ur heap gör vi genom att byta plats på första och sista elementet ($\mathcal{O}(1)$), sedan borttagning ($\mathcal{O}(1)$ enl. ovan), och till sist bubblar vi ned det nya rotelementet i heapen ($\mathcal{O}(\log(k))$). Totalt: $\mathcal{O}(\log(k))$

(c) Om vi har ett stort antal kakor är alternativ 2 bäst, då alternativ 2 helt undviker $\mathcal{O}(k)$-tiden både för insättning och för borttagning (förutom i de fåtal fall då den dynamiska arrayen behöver omallokeras, men samma problem finns i alternativ 1), och kräver inte heller mer minne. $\log(k)$ växer också *mycket* långsammare än n.

(d) Ett enkelt sätt att göra detta på är att inte sortera arrayen vid insättning. I stället håller vi reda på om arrayen är sorterad (en variabel `sorted`). Vid insättning så sätter vi helt enkelt in den nya kakan sist ($\mathcal{O}(1)$ amorterad tid) och sätter `sorted` till `false`. `find_cookie` tittar först på `sorted`, om den är `false` så sorterar den arrayen med heapsort och sätter `sorted` till true. Sedan fortsätter den som i uppgiften. Detta tar $\mathcal{O}(1)$ om arrayen inte behövde sorteras, och $\mathcal{O}(k \log(k))$ om arrayen behövde sorteras.

Om alla kakor sätts in först, och sedan tas alla bort, så kommer vi bara behöva sortera arrayen en gång. Alltså får vi totalt $\mathcal{O}(k + k\log(k) + (k-1)) = \mathcal{O}(k\log(k))$. Utan förbättringen skulle detta ta $\mathcal{O}(k^2)$ tid.

Detta gör dock att datastrukturen presterar sämre om man växelvis sätter in och tar bort kakor. Att sätta in k kakor och sedan lägga till + ta bort kakor k gånger tar då: $\mathcal{O}(k^2 \log(k))$ tid eftersom vi behöver sortera arrayen vid varje borttagning.

5. (a) Vi kan exempelvis implementera detta som nedan:

```
// Lagring av tabellen:
struct Cookie {
  string type;       // kaksort
  vector<int> sizes; // storlekar
};

vector<pair<string, int>>  // utdata: namn + diameter
 cookie_pyramid(vector<Cookie> cookies, vector<int> wish) {
  // Lagra alla kakor i ett balanserat sökträd. Nyckel är
  //  kakans diameter, värdet är kaktypen.
  multiset<int, string> tree;
  for (const Cookie &t : cookie) {
    for (int size : t.sizes) {
      tree.insert(make_pair(size, t.type));
    }
  }

  // Gå igenom önskemålen och hitta lämpliga kakor.
  vector<pair<string, int>> result;
  int last_size = 0;
  for (int w : wish) {
    // Notera: lower_bound hittar nyckeln närmast mindre än eller lika med 'w'.
    // I ett sökträd går detta att göra i log(n) tid.
    auto found = tree.lower_bound(w);

    // Titta på nuvarande och nästa kaka för att se vilken som är bäst.
    if (found != tree.end()) {
      auto next = found; ++next;
      if (found->first >= last_size && w - found->first <= next->first - w) {
        result.push_back(make_pair(found->second, found->first));
        tree.erase(found);
      } else {
        result.push_back(make_pair(next->second, next->first));
        tree.erase(next);
      }
    } else if (found->first >= last_size) {
      result.push_back(make_pair(found->second, found->first));
      tree.erase(found);
    } else {
      throw "Impossible"; // should not happen.
    }
  }

  return result;
}
```

   Nyckelidén är alltså att vi använder ett balanserat sökträd för att snabbt kunna hitta närmsta mindre element *och* ta bort det. Om vi inte behöver ta bort element varje gång så hade en sorterad array + binärsökning räckt.

   (b) Detta tar $\mathcal{O}(n \log(n)$ (insättning i sökträdet)$+k \log(n)$ (uppslagning)$) = \mathcal{O}(n \log(n)+k \log(n))$

   (c) För att förbättra lösningen från (a) så låter vi helt enkelt sökträdet `tree` ligga i en klass eller dylikt så att vi inte behöver bygga upp den från grunden varje gång. Insättning och uppslagning i datastrukturen sker då som ovan, men i separata medlemsfunktioner.

   Detta gör att steg (1) tar $\mathcal{O}(\log(n))$ tid, och att steg (2) tar $\mathcal{O}(k \log(n))$ tid. Detta gör att steg (2) är billigare än att köra koden i (a).

6. Problemet går att formulera som en riktad och viktad graf. Som tur är så har vi aldrig negativa bågvikter, så vi kan inte få negativa cykler i grafen. Vi behöver alltså inte använda en algoritm som

klarar negativa bågvikter, utan vi klarar oss med Dijkstras algoritm. Bågar med oändlig restid kan vi helt enkelt ignorera, eftersom vi letar efter den kortaste vägen.

(a) Vi kan lösa detta problem genom att köra Dijkstras algoritm mellan den givna start- och slutnoden. Notera att alla platser kan helt enkelt representeras som strängar, oavsett om de har koordinater eller inte.

```
// En rad i tabellen i uppgiften:
struct Table_Row {
  string from;
  string to;
  int weight;
};

// En nod i grafen:
struct Node {
  bool visited = false;
  int distance = 0;
  string previous = "";
  vector<pair<int, string>> edges;
};

using Graph = unordered_map<Node>;

// Bygg upp grafen. Görs i ett separat steg för att kunna göra
// steget separat senare.
Graph build_graph(vector<Table_Row> table) {
  Graph graph;
  for (const Table_Row &row : table) {
    Node &node = graph[row.from];
    if (row.weight != INFINITY)
      node.edges.push_back(make_pair(row.weight, row.to));
  }
  return graph;
}

// Hitta en väg mellan givna noder.
vector<string> find_path(Graph &graph, string from, string to) {
  for (Node &n : graph) {
    n.visited = false;
    n.distance = INT_MAX;
    n.previous = "";
  }

  priority_queue<pair<int, string>, std::greater> pq; // ger oss minsta elementet varje gång
  pq.push(make_pair(0, from));

  while (!pq.empty()) {
    auto current = pq.top(); pq.pop();
    if (current.second == to)
      break;

    Node &node = graph[current.second];
    if (node.visited)
      continue;
    node.visited = true;

    for (auto &&x : node.edges) {
      int new_dist = node.distance + x.first;
      Node &to = graph[x.second];
      if (!to.visited && to.distance > new_dist) {
        to.distance = new_dist;
```

```
        to.previous = current.second;
        pq.push(make_pair(new_dist, x.second);
      }
    }
  }

  if (!graph[to].visited)
    return {};

  vector<string> path;
  for (string at = to; at != ""; at = graph[at].previous)
    path.push_back(at);
  std::reverse(path.begin(), path.end());
  return path;
}
```

(b) Detta tar $\mathcal{O}((p+v)\log(p))$

(c) För att lösa problemet optimalt krävs att vi löser en variant av *traveling salesman*-problemet. Detta är ett välkänt svårt problem som inte har en polynomiell lösning.

Vi kan dock göra en approximation enligt följande:

1. Lagra noderna vi har kvar att besöka i en array `to_visit`.
2. Kom ihåg vår startpunkt som `current`.
3. Kör `find_path(current, )`. Som `find_path` är skriven så kör den tills alla noder är besökta, men den kommer inte lyckas att hitta en väg. Däremot så är `distance` uppdaterad i alla noder.
4. Undersök noderna i `to_visit`, välj den med lägst `distance`, och kalla den `next`. Det är dit vi vill gå härnäst.
5. Extrahera vägen till noden på samma sätt som i slutet av `find_path`, och skriv ut den.
6. Sätt `current` till `next`, och ta bort `next` från `to_visit`.
7. Om `to_visit` är tom så är vi klara. Annars, gå till steg 3.

(d) Här kör vi Dijkstras algoritm maximalt p gånger. Resterande steg är billigare än Dijkstras (så även om vi hade kunnat ta bort element billigare i `to_visit` med ex.vis en länkad lista, så spelar det mindre roll). Alltså: $\mathcal{O}(p \cdot (p+v)\log(p))$