Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Filip Strömbäck och Magnus Nielsen

31 oktober 2024

Datastrukturer och algoritmer Lösningsförslag till tentamen 2024-10-31

1. (a) quicksort — rad 4

   (b) bubble sort — rad 5

   (c) insertionsort — rad 1

   (d) selectionsort — rad 3

   (e) heapsort — rad 2

2. (a) $8 \Rightarrow 1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4$ (går tillbaka till nod 8) $\Rightarrow 7 \Rightarrow 5$ (1 redan besökt, tillbaka till nod 7) $\Rightarrow$ 6 (1 och 4 redan besökta)

   (b) Vägen som hittas är första vägen som besöks: $8 \Rightarrow 1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4$

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

   8(-) (alla vägar av längd 1 hittade)
   $\Rightarrow$ 1(8) $\Rightarrow$ 7(8) (alla vägar av längd 2 hittade)
   $\Rightarrow$ 2(1) $\Rightarrow$ 5(7) $\Rightarrow$ 6(7) (alla vägar av längd 3 hittade, notera att nod 1 redan är besökt och kommer inte att besökas igen)
   $\Rightarrow$ 3(2) $\Rightarrow$ 4(5) (alla vägar av längd 4 hittade, nod 4 är besökt, så traversering av nod 3 kommer ej lägga till den i kön igen)

   (d) Vägen som vi kom til när vi hittade nod 4 första gången. Vi hittar den genom att följa siffrorna i parentes ovan: $8 \Rightarrow 7 \Rightarrow 5 \Rightarrow 4$

3. (a) Loop som ökas med 2 varje gång, till $n/2$: $\mathcal{O}((n/2)/2) = \mathcal{O}(n)$

   (b) Rekursiv funktion, n halveras för varje rekursivt anrop tills n når 0. Ger $\mathcal{O}(\log_2(n)) = \mathcal{O}(\log(n))$

   (c) Loop som körs n gånger. Inuti anropas `f(10)` (tar $\mathcal{O}(1)$ tid), samt `fn(m)` (tar $\mathcal{O}(\log(m))$ tid). Totalt: $\mathcal{O}(n \log(m))$

   (d) Funktionen `findPosition` är en binärsökning. Den tar $\mathcal{O}(\log(n))$ tid. `insert` tar (amorterad) konstant tid i bästa fall, då element sätts in i slutet. Värsta fallet är att sätta in först, eftersom alla element i `arr` då måste flyttas fram ett steg.

   Detta ger:

   Bästa fall: $\mathcal{O}(\log(n))$

   Värsta fall: $\mathcal{O}(n)$ (insättningen dominerar binärsökningen)

4. (a) Insertion sort har ett bästa fall av $\mathcal{O}(n)$ om alla element är sorterade, eller om endast ett (eller ett fåtal) element är på fel plats. Värsta- och medelfall är $\mathcal{O}(n^2)$.

   Detta gör att vi får bästa fallet ifall listan `x` redan är sorterad. Då tar steg 0 $\mathcal{O}(n)$ tid, steg 1 $\mathcal{O}(n)$ tid, steg 2 $\mathcal{O}(1)$ tid (man krymper typiskt sett inte den dynamiska arrayen), steg 3 $\mathcal{O}(1)$ tid (vi behöver aldrig omallokera arrayen på grund av steg 2). Totalt $\mathcal{O}(n)$ för en iteration. Detta görs $n-1$ gånger eftersom arrayen blir ett steg kortare varje gång. När vi går tillbaka till steg 1 så kommer maximalt ett element vara på fel plats. Insertion sort kommer därför först att spendera $\mathcal{O}(n)$ tid med att konstatera att alla element är på rätt plats, följt av mellan $\Omega(1)$ och $\mathcal{O}(n)$ tid att flytta elementet till rätt plats. Detta tar $\mathcal{O}(n)$ tid oavsett.

   Totalt blir bästa fall alltså $\mathcal{O}(n^2)$.

   Värsta fall ges om x inte är sorterad till att börja med. Då tar insertion sort $\mathcal{O}(n^2)$ tid. Resten av analysen är densamma. Intressant nog tar resterande sorteringar med insertion sort $\mathcal{O}(n)$ tid av samma anledning som ovan. Totalt får vi alltså ett värsta fall som är:
   $\mathcal{O}(n^2$ (1:a sorteringen) $+ n \cdot n$ (resten av iterationerna)) $= \mathcal{O}(n^2)$

(b) Steg 0 tar $\mathcal{O}(n)$ tid, steg 1 och 2 tar $\mathcal{O}(\log(n))$ tid, steg 3 tar också $\mathcal{O}(\log(n))$ tid. Steg 1–3 repeteras sedan n gånger i och med att heapen minskar med 1 i storlek varje iteration.

Värt att notera är att steg 1–2 har ett bästa fall på $\mathcal{O}(1)$, vilket inträffar om det element som hamnar överst i heapen efter borttagning inte behöver flyttas. Liknande kan ske i steg 3, om det nya elementet är tillräckligt stort för att inte behöva flyttas. Det är dock mycket svårt (om ens möjligt) att konstruera indata så att bästa fallet inträffar varje gång. Vi antar därmed att bästa och värsta fallen sammanfaller.

Totalt får vi alltså att varje iteration i loopen tar $\mathcal{O}(\log(n))$ tid. Detta görs n gånger efter att vi har gjort steg 0 en gång. Vi får då: $\mathcal{O}(n + n + n\log(n)) = \mathcal{O}(n\log(n))$

(c) Vi kan utnyttja observationen från (a) att arrayen kommer vara näst intill sorterad i alla iterationer förutom den första. Vi kan därför göra följande omskrivning:

Första gången vi kör steg 1 så sorterar vi arrayen med ex.vis Quicksort eller Heapsort ($\mathcal{O}(n\log(n))$)

Resterande gånger så ersätter vi sorteringen med följande loop:

```
for (int i = x.size() - 1; i > 1; i--) {
  if (x[i - 1] > x[i])
    swap(x[i - 1], x[i]);
  else
    break;
}
```

I majoriteten av fallen behöver vi endast köra ett fåtal iterationer av loopen, därav $\mathcal{O}(1)$ bästa fall. Dessa omskrivningar gör att vi får en förväntad körtid av $\mathcal{O}(n\log(n))$, vilken domineras av sorteringen första gången steg 1 körs.

5. (a) Exempelvis enligt koden nedan:

```
struct Pun {
  double fun;
  string pun;
  bool used = false;

  // Här har vi också en <-operator som jämför 'fun'.
};

vector<int> select_puns(vector<Pun> puns, vector<double> &segments) {
  std::sort(puns.begin(), puns.end()); // Sortering mha quicksort, O(n log n)

  vector<int> selected;
  for (double segment : segments) {
    // Binärsökning, O(log n)
    auto found = std::lower_bound(puns.begin(), puns.end(), segment);

    // Hitta nästa roligare pun:
    while (found != puns.end()
        && found->fun <= segment
        && found->used)
      ++found;

    if (found == puns.end())
      throw "impossible";

    selected.push_back(found - puns.begin());
  }

  return selected;
}
```

Idén är alltså att vi sorterar `puns`, och sedan kan vi enkelt binärsöka fram det pun som är närmast roligare än vad vi behöver.

(b) Detta tar tid $\mathcal{O}(p\log(p)$ (sortering) $+ n\log(p)$ (hitta sketch)) $= \mathcal{O}(p\log(p) + n\log(p))$

(c) Det enda extra minne vi behöver är en kopia av `puns` och en lista av segment. Quicksort använder också $\mathcal{O}(\log(p))$ extra minne för sorteringen, men det domineras av vår kopia av `puns`. Alltså: $\mathcal{O}(n + p)$.

6. Problemet går att se som en riktad och oviktad graf. Pekarna som finns i lokala och globala variabler kan ses som en startnod (vi behöver inte bry oss om variabelnamnen). Resterande objekt är individuella noder där varje pekare är en båge till ett annat objekt.

   (a) Vi kan identifiera alla nåbara objekt genom att göra en BFS från startnoden. På så vis markerar vi alla objekt som är nåbara. Efter det så kan vi helt enkelt hitta alla noder som inte är markerade. De är minnesläckor.

   Det kan implementeras på följande sätt:

```
struct Object {
  bool visited = false;
  vector<int> pointers;
};

// 'start' är pekarna från lokala och globala variabler.
// 'objects' är alla objekt, index i vectorn motsvarar objekt-id.
vector<int> find_leaks(vector<int> start, vector<Object> objects) {
  queue<int> q;
  for (int s : start) {
    objects[s].visited = true;
    q.push(s);
  }

  while (!q.empty()) {
    int current = q.top(); q.pop();
    for (int p : objects[current].pointers) {
      if (objects[p].visited)
        continue;

      objects[p].visited = true;
      q.push(p);
    }
  }

  vector<int> leaks;
  for (int i = 0; i < objects.size(); i++)
    if (!objects[i].visited)
      leaks.push_back(i);

  return leaks;
}
```

   (b) Detta tar $\mathcal{O}(o + p)$ tid (värsta fall är att alla noder och alla bågar besöks).

(fortsättning på nästa sida)

(c) Vi kan göra detta genom att utgå från resultatet från (a). Vi väljer helt enkelt en av de noder som `find_leaks` hittade, lägger till en variabel från godtycklig besökt nod som pekar på det objektet. Sedan kör vi en BFS (likt i `find_leaks`) med start i den noden. Vi återställer *inte* visited när vi gör en ny BFS, vilket gör att vi inte besöker noder som inte är läckor igen. Denna BFS kommer att ge oss en ny lista med noder som inte är nåbara (förhoppningsvis har den nya variabeln vi lade till gjort att fler objekt blivit nåbara). Vi repeterar sedan denna process (dvs. lägg till båge + kör BFS) tills alla objekt blivit nåbara.

Detta tar totalt $\mathcal{O}(o+p)$ tid om vi har en någorlunda sammanhängande graf. Kör vi algoritmen ovan på en graf som inte innehåller några bågar alls så får vi dock $\mathcal{O}(o^2)$ tid eftersom vi måste hitta alla objekt som är läckor genom att söka fram dem linjärt. Det går att lösa genom att lägga icke-besökta noder i en `std::unordered_set`, exempelvis.

Algoritmen ovan är inte optimal, men OK eftersom optimalitet inte efterfrågas. Eftersom vi bara väljer en nod på måfå och lägger till en båge kan det vara så att det finns en annan icke-nåbar nod som skulle vara resultera i att fler noder blir nåbara.