Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Filip Strömbäck och Magnus Nielsen

27 augusti 2024

TDDI16
Datastrukturer och algoritmer
Datortentamen (DAT1)
2024-08-27, 08–12

Examinator:	Filip Strömbäck
Jour:	Filip Strömbäck (telefon 013-28 27 52)
Antal uppgifter:	6
Max poäng:	40 poäng
Preliminära gränser:	Betyg 5 = 35p, 4 = 27p, 3 = 20p.
Hjälpmedel:	Inga hjälpmedel tillåtna!

VÄNLIGEN IAKTTAG FÖLJANDE

- Observera att betygsgränserna kan komma att justeras, i samtliga kurser.
- Vid frågor om tidskomplexitet, svara alltid på den form som är mest relevant.
- Skriv dina lösningar i valfri textredigerare, exempelvis LibreOffice, Emacs, eller liknande. Strukturera din text så att det enkelt går att se vad som svarar på vilka deluppgifter.
- Lösningar till olika problem skall skrivas i en egen fil. Skriv inte två lösningar i samma fil. Delproblem får dela fil.
- Om inte annat framgår ska indexering av arrayer/listor börja från 0.
- Skicka in som lösning till rätt problem i tentaklienten, och döp filen till ett passande namn (exempelvis uppg1.txt/uppg1.odt).
- **MOTIVERA DINA SVAR ORDENTLIGT:** avsaknad av, eller otillräckliga, förklaringar resulterar i poängavdrag. **Även felaktiga svar kan ge poäng** om de är korrekt motiverade.
- Om ett problem medger flera olika lösningar, t.ex. algoritmer med olika tidskomplexitet, ger endast optimala lösningar maximalt antal poäng.
- SE TILL ATT DINA LÖSNINGAR/SVAR ÄR LÄSLIGA. Oläsliga lösningar beaktas ej.

Lycka till!

1. **Sorteringsalgoritmer** (5 p)

Svaren behöver ej motiveras.

Efter **ett fåtal** iterationer av några olika sorteringsalgoritmer på den osorterade arrayen *Original* i tabellen nedan (iteration i bemärkelse fullständig körning av inre loop, alternativt rekursivt anrop) har vi resultaten i *1, 2, 3* och *4*.

Original:	16	12	87	63	47	7	72	75	61	77	31	49	42	86	43
1:	7	12	16	63	47	87	72	75	61	77	31	49	42	86	43
2:	12	16	63	87	47	7	72	75	61	77	31	49	42	86	43
3:	7	12	42	31	16	43	47	75	61	77	63	49	87	86	72
4:	75	63	72	61	47	49	43	16	42	12	31	7	77	86	87
Sorterad:	7	12	16	31	42	43	47	49	61	63	72	75	77	86	87

Matcha de delvis sorterade arrayerna mot algoritmerna nedan. En av algoritmerna nedan finns inte bland de fyra arrayerna ovan. Ett av svaren blir alltså ”finns ej ovan”. Felaktig matchning ger minuspoäng, dock kan uppgiften ej ge total minuspoäng. Utelämnat svar ger ej minuspoäng. För totalt 5 poäng krävs alltså 5 korrekta svar.

(a) Quicksort, elementet längst till höger i partitionen används som pivot (1)

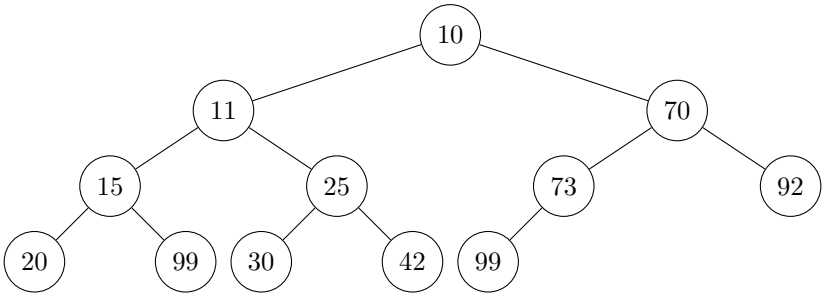
(b) Bubble sort, element bubblas från höger till vänster (1)

(c) Insertionsort (1)

(d) Selectionsort (1)

(e) Heapsort (1)

2. **Träd** (5 p)



Svara på följande frågor om trädet. Motivera dina svar.

(a) Är trädet ovan komplett? (1)

(b) Är trädet ovan fullt? (1)

(c) Är trädet ovan ett binärt sökträd? (1)

(d) Är trädet ovan en heap? Om ja, ange ifall det är en min- eller en max-heap. (1)

(e) Noden 11 sätts in. Vilka noder behöver traverseras för att återställa trädet enligt ditt svar i (c) och (d)? (1)

3. **Algoritmer och tidskomplexitet** (6 p)

Visa kort hur du kommer fram till tidskomplexiteten i följande uppgifter.

(a) Beräkna tidskomplexiten med avseende på parametern n för följande funktion: (1)

```
int f(int n) {
  int result = 0;
  for (int k = 0; k < 4; k++) {
    for (int i = 1; i < n; i *= 2) {
      result += i * k;
    }
  }
  return result;
}
```

(b) Beräkna tidskomplexiten med avseende på parametern n för följande funktion: (1)

```
int fn(int n) {
  int result = 0;
  for (int i = 0; i < n; i++) {
    result += f(10);
  }
  return result;
}
```

(c) Beräkna tidskomplexiten med avseende på parametrarna n och m för följande funktion: (2)

```
int fun(int n, int m) {
  int sum = 0;
  for (int i = 1; i < n; i = i * 2) {
    for (int j = 0; j < m; j++) {
      sum += i * j;
    }
  }
  return sum;
}
```

(d) Beräkna tidskomplexiteten med avseende på parametern n. Tänk på bästa och värsta fall in din analys. (2)

```
int loop(int n) {
  int times = 0;
  while (n > 0) {
    if (n % 2 == 0) { // n är jämnt
      n /= 2;
    } else {          // n är udda
      n -= 2;
    }
    times++;
  }
  return times;
}
```

4. **Subscribe for more DALG** (10 p)

Du driver en webbsida som publicerar DALG-uppgifter dagligen som prenumeranterna kan lösa för att tävla och hamna på highscore-listor. En viktig del i systemet är då hanteringen av prenumeranter på sidan.

Varje prenumerant får ett kundnummer. Kundnumren börjar på 1 och ökar med ett för varje nyregistrerad prenumerant. Systemet behöver lagra kundinformation så att det snabbt kan kontrollera om en kund har en aktiv prenumeration. Varje kund representeras av följande datatyp:

```
class Customer {
    String name;
    String email;
    int score;
    bool isSubscribed;
}
```

Dessa lagras sedan i en datastruktur som har följande operationer:

`create()` Skapa en ny, tom datastruktur.

`new_customer(customer)` Anropas när en ny prenumerant har registrerat sig på sidan. Genererar ett nytt kundnummer och sparar `customer` tillsammans med det genererade kundnumret. Returnerar sedan kundnumret.

`find_customer(id)` Hittar en `Customer`-instans som tidigare blivit insatt med `new_customer`.

Du har inte tid att implementera detta själv, så du har delegerat det till en anställd i företaget. Den anställda har kommit med följande två förslag om hur detta kan implementeras:

1: Lagra data som en hashtabell med kundnummer (heltal) som nyckel och `Customer` som värde. Hashtabellen använder länkad indexering, och har alltid 50 element i det underliggande arrayen (dvs. 50 buckets). Datastrukturen innehåller också ett separat heltal, *next*, som lagrar nästa lediga heltal.

   `create` skapar en tom hashtabell och sätter *next* till 0.

   `new_customer(customer)` ökar *next* med 1. Sätter sedan in `customer` med nyckeln *next* i hashtabellen, och returnerar *next*.

   `find_customer(id)` letar reda på kunden med `id` i hashtabellen och returnerar den.

2: Lagra data i en dynamisk array. Arrayen startar med plats för 8 element. Vid omallokering dubbleras platsen i arrayen.

   `create` skapar en tom dynamisk array. Som anges ovan så har den plats för 8 element.

   `new_customer(customer)` lägg in `customer` sist i arrayen. Returnera storleken av arrayen.

   `find_customer(id)` returnera elementet på plats `id - 1` i arrayen.

(fortsättning på nästa sida)

Svara på följande frågor med avseende på de två alternativen ovan:

(a) Vilken tidskomplexitet har vart och ett av alternativen ovan för insättning (dvs. `new_customer`) i **medelfallet**? Uttryck tidskomplexiteten i termer av antalet element i datastrukturen (n). Beskriv kort ditt resonemang. (2)

(b) Vilken tidskomplexitet har vart och ett av alternativen ovan för insättning (dvs. `new_customer`) i **värsta fall**? Uttryck tidskomplexiteten i termer av antalet element i datastrukturen (n). Beskriv kort ditt resonemang. (2)

(c) Vilken tidskomplexitet har vart och ett av alternativen ovan för uppslagning (dvs. `find_customer`) i **medelfallet**? Uttryck tidskomplexiteten i termer av antalet element i datastrukturen (n). Beskriv kort ditt resonemang. (2)

(d) Vilket av alternativen 1–2 är bäst i det här scenariot? Motivera ditt svar. (1)

(e) Du har insett att datastrukturen ovan kräver $\mathcal{O}(n)$ tid för att sammanställa highscore-listan. Ge ett förslag på en lämplig datastruktur för att lagra användares poäng på ett sätt som är billig att uppdatera, och som går att använda för highscore-listan. (3)

5. **Internationella Badplatser (IBP)** (6 p)

Du har fått ett konsultuppdrag från Svengelska turistbyrån. Deras sales-pitch är att specialisera sig på just internationella badplatser (eller *International Bathing Places (IBP)* på svengelska). I och med detta vill de ha ett verktyg på sin sida som låter användaren hitta de bästa badplatserna.

Internationell badplats, Japan, CC BY-SA 3.0 `https://commons.wikimedia.org/wiki/File:Onsen_in_Nachikatsuura,_Japan.jpg`

Sidan som turistbyrån vill ha låter användaren först specifiera hur mycket de värderar *pris* mot *ranking*. Detta anges i termer av en procentsats, w: hur viktigt användaren tycker att priset är. Användaren anger sedan hur många badplatser denne är intresserad av, n. Systemet letar sedan i sin (stora) databs för att hitta de n bästa badplatserna utifrån den angivna vikten, $0 \leq w \leq 100$.

Utifrån viktningen så beräknar systemet en total *score*, s för varje badplats enligt:

$$score = w \cdot pris + (100 - w) \cdot ranking$$

Systemet väljer sedan de n badplatser med lägst *score*. Ett stort problem är att det finns *många* badplatser i världen, fler än vad som får plats i RAM på den server som turistbyrån har. Algoritmen måste alltså hitta de $n < 500$ badplatserna med lägst *score* utan att lagra alla element från databasen i RAM. Tabellen innehåller information likt nedan (men innehåller fler än 10^{12} rader):

Badplats	**Pris**	**Ranking**
Natchikatsuura onsen	50 kr	3
Piteå havsbad	5 kr	30
Bestorps badpats	0 kr	100
...	...	...

(a) Beskriv en algoritm som kan hitta de n badplatserna med lägst *score*, givet n och w från användaren. Kom ihåg att du inte kan lagra alla badplatser i databasen i minnet. (4)

Beskriv din algoritm, gärna i punktform eller med pseudokod. Tänk på att ha med hur du lagrar data och vilka datastrukturer din algoritm använder. Du behöver inte detaljbeskriva sådant som finns i standardbiblioteket.

(b) Vilken tidskomplexitet har din implementation uttryckt i antal badplatser i databasen (m)? Motivera ditt svar. (1)

(c) Hur mycket minne använder din implementation uttryckt i antalet badplatser i databasen (m), och antalet efterfrågade badplatser (n). Svara med ett $\mathcal{O}$-uttryck, samt visa hur du kom fram till svaret. (1)

6. **Bostadsberoenden** (8 p)

Du jobbar som egenföretagare, och har insett att det finns ett problem i bostadsmarknaden som du kan ta betalt för att lösa åt folk! Bostäder är i allmänhet så dyra att en person inte har råd att äga mer än en bostad i taget. Det är ett problem när man flyttar, eftersom man då behöver äga två bostäder samtidigt under själva flytten. Som tur var har de flesta banker lån som täcker just detta. Dock kräver bankerna att man har ett påskrivet kontrakt att den gamla bostaden är såld. Notera att kontrakten anger en tidpunkt i framtid när flytten faktiskt sker.

Exempelvis: Kim vill flytta från Hans Meijers väg 29 till Mäster Mattias väg 1. För att kunna flytta sina möbler så behöver Kim tillfälligt äga båda boendena. För att få lånet måste då Kim först hitta en köpare till Hans Meijers väg 29. Filip är intresserad av att köpa Hans Meijers väg 29, men måste då först sälja Mårdtorpsgatan 27, och måste då i sin tur hitta en köpare.

Ett ytterligare krav från mäklaren är att en person som köper en bostad måste ha skrivit kontrakt för att sälja sin bostad innan personen får skriva på ett köpekontrakt. I exemplet ovan måste alltså Magnus sälja sin bostad innan han kan köpa Kims bostad.

Tjänsten som du vill erbjuda är att du vill (mot en avgift) hjälpa personer som vill flytta att hitta köpare åt dem. Det som är unikt för dig, är att du också hittar köpare åt köparen och så vidare, tills du hittat någon som vill köpa bostad men inte sälja något (ex.vis någon som flyttar från sin korridor i Ryd till första ”riktiga” bostaden).

Till din hjälp vill du skriva ett program som löser problemet åt dig. Programmet ska alltså, givet din klient, hitta den kortaste sekvensen av personer som behöver sälja sina bostäder för att din klient ska kunna flytta. Denna kedja måste alltså sluta i en person som inte vill sälja någon bostad.

Till din hjälp har du en lista över bostäder (vänster) och en lista över önskade affärer (höger):

ID	**Bostad**
0	Mårdtorpsgatan 27
1	Västanågatan 22
2	Vallavägen 4c
3	Hans Meijers väg 29
4	Mäster Mattias väg 1
...	...

Person	**Vill sälja**	**Vill köpa**
Filip	0	3, 4
Magnus	1	2
Kim	3	4
Emma	68	0
Maria	-	68
...	...	...

Notera: Om det står fler än en siffra i *vill köpa*-kolumnen, så kan personen tänka sig att köpa vilken som av bostäderna. Filip vill alltså **inte** köpa både bostad 3 och bostad 4, utan endast en av dem.

Exempel: Kim har bett dig om hjälp att hitta köpare. Ditt program bör då föreslå att *Maria* måste köpa *Emmas* lägenhet. *Emma* ska i sin tur köpa *Filips* lägenhet, och *Filip* kan då köpa *Kims* lägenhet. Detta är den kortaste kedjan i exemplet.

(a) Beskriv hur du kan implementera programmet som hittar den kortaste kedjan av affärer som måste ske för att din klient ska få sin bostad såld. Programmet får personen som är din klient och listan över önskade affärer som indata. (4)

Beskriv din algoritm, gärna i punktform eller med pseudokod. Tänk på att ha med vilka datastrukturer du använder, vilken data som lagras i dem, och hur de initieras. Du behöver inte detaljbeskriva sådant som finns i standardbiblioteket.

(b) Vilken tidskomplexitet har din implementation i (a), uttryckt i antal personer i tabellen (p) och antal bostäder (b)? Motivera ditt svar. (1)

(fortsättning på nästa sida)

(c) Efter ett tag inser du att det ibland krävs långa kedjor av affärer för att se till att din klient kan sälja sin bostad. Du kom därmed på den briljanta idén att erbjuda att tillfälligt köpa klientens bostad. Detta gör att om exempelvis Kim vill köpa Emmas bostad och vice versa, så kan du först köpa Kims bostad. Detta gör att Kim nu kan köpa Emmas bostad, och Emma kan köpa Kims gamla bostad från dig. (2)

Beskriv hur du kan modifiera ditt program så att det också tar hänsyn till möjligheten att du tillfälligt köper bostaden av din klient. Beskriv din algoritm, gärna i punktform eller med pseudokod. Tänk på att ha med vilka datastrukturer du använder, vilken data som lagras i dem, och hur de initieras. Du behöver inte detaljbeskriva sådant som finns i standardbiblioteket.

(d) Vilken tidskomplexitet har din implementation i (c), uttryckt i antal personer i tabellen (p) och antal bostäder (b)? Motivera ditt svar. (1)