Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Filip Strömbäck och Magnus Nielsen

2025-10-30

TDDI16
Datastrukturer och algoritmer
Datortentamen (DAT1)
2025-10-30, 14–18

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.
- Lämna in en fil per numrerad uppgift.
- 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, 4* och *5*.

Original:	78	35	15	69	7	27	43	26	66	34	21	22	58	91	77	36
1:	22	7	15	21	26	27	34	35	36	43	58	66	69	91	77	78
2:	15	35	69	78	7	27	43	26	66	34	21	22	58	91	77	36
3:	15	35	69	78	7	26	27	43	21	22	34	66	36	58	77	91
4:	7	15	21	78	35	22	69	26	27	43	34	66	36	58	77	91
5:	7	15	21	69	78	27	43	26	66	34	35	22	58	91	77	36
Sorterad:	7	15	21	22	26	27	34	35	36	43	58	66	69	77	78	91

   Matcha de delvis sorterade arrayerna mot algoritmerna nedan. Felaktig matchning ger minuspoäng, dock kan uppgiften ej ge total minuspoäng. Utelämnat svar ger ej minuspoäng. För totalt 4 poäng krävs alltså 4 korrekta svar.

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

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

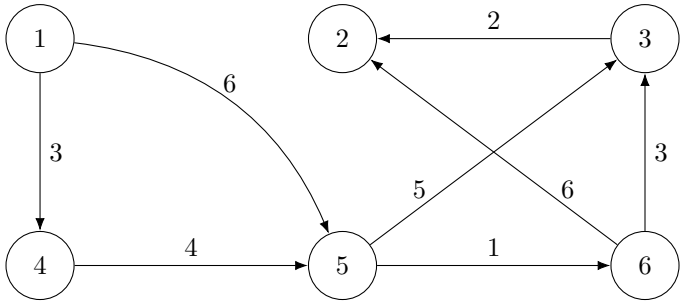
   (c) Insertionsort (1)

   (d) Selectionsort (1)

   För den femte och sista poängen i frågan (här ger felaktigt svar inte avdrag):

   - Vilken algoritm användes i den array som inte matchades ovan? Ge en kort motivering. (1)

2. Betrakta grafen nedanför när du svarar på frågorna. Antag att grafen lagras på ett sådant sätt att när man itererar över bågarna från en nod så får vi dem i stigande ordning baserat på bågens vikt. Exempelvis: om algoritmen itererar igenom bågarna från nod 5 så kommer den först att få bågen $5 \to 6$ (vikt 1) och sedan bågen $5 \to 3$ (vikt 5). (4 p)



   (a) I vilken ordning besöks noderna om BFS används för att hitta en väg mellan nod 1 och nod 2? (1)

   (b) Vilken är den väg som BFS hittar i grafen? (1)

   (c) I vilken ordning besöks noderna om Dijkstras algoritm används för att hitta en väg mellan nod 1 och nod 2? (1)

   (d) Vilken är den väg som Dijkstras algoritm hittar i grafen? (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 i = 1; i < n * 2; i *= 2) {
    result += i;
  }
  return result;
}
```

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

```
int fn(int n) {
  if (n > 0) {
     return fn(n - 1) * n;
  } else {
     return 1;
  }
}
```

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

```
int fun(int n, int m) {
  int t = m * n;
  int result = 0;
  for (int i = 1; i < n; i++) {
    result += f(t);
  }
  return result;
}
```

(d) Beräkna tidskomplexiteten på `shortestValuableInterval`[1] med avseende på storleken av `arr` ($= n$). Tänk på bästa och värsta fall in din analys. (2)

```
int findInterval(const vector<int> &arr, int from, int minValue) {
  int sum = 0;
  for (int i = from; i < arr.size(); i++) {
    sum = sum + arr[i];
    if (sum >= minValue) {
      return i - from;
    }
  }
  return arr.size();
}

int shortestValuableInterval(const vector<int> &arr, int minValue) {
  int shortestInterval = arr.size();
  for (int i = 0; i < arr.size(); i++) {
    int currInterval = findInterval(arr, i, minValue);
    shortestInterval = min(shortestInterval, currInterval);
  }
  return shortestInterval;
}
```

[1] Funktionen hittar den kortaste sammanhängande sekvensen av tal i `arr` vars summa är $\geq$ `minValue`.

4. **Optimal Kakkonsumtion** (8 p)

I skrivande stund är det 2 månader kvar till julafton. Då kommer som bekant jultomten dela ut presenter till alla snälla barn. För att orka med detta så behöver tomten regelbundet fylla på med energi. Som tur är så följer många traditionen att ställa fram kakor till tomten. På så vis har tomten tillgång till extra energi att fylla på med. Problemet är dock att tomten under sin färd kommer att samla på sig en stor mängd kakor, vilket gör det svårt att hitta den godaste kakan när tomten är hungrig (tomten vill så klart äta den godaste kakan först, när den är så färsk som möjligt!).

För att hålla koll på kakorna behöver tomten en datastruktur med följande operationer:

`add_cookie(x)` Anropas när tomten har plockat upp en kaka. `x` är ett par: (g, n), där g är ett heltal som beskriver hur god kakan är (högre tal, godare kaka), och n är en sträng som beskriver vilken kaka det är.

`find_cookie()` Anropas när tomten är hungrig. Ska ta bort och returnera den godaste gakan som lagts till i datastrukturen (som en tupel (g, n) enligt ovan).

Tomten har följande två möjliga implementationer av datastrukturen:

Alternativ 1: Lagra data i en dynamisk array `a`, som är tom till en början.

`add_cookie(x)` Lägger in elementet `x` i slutet av `a`. Sorterar sedan `a` med hjälp av insertionsort, i stigande ordning på g i varje tupel.

`find_cookie()` Tar bort och returnerar det sista elementet i `a`.

Alternativ 2: Lagra data i en maxheap, `h`, som lagras i en dynamisk array. Heapen använder g i varje tupel för att jämföra kakor, och är tom till en början.

`add_cookie(x)` Lägger in elementet `x` på heapen med hjälp av standardinsättning för en heap.

`find_cookie()` Tar bort det största elementet ur heapen `h` med hjälp av standardborttagning för en heap, och returnerar det borttagna elementet.

(a) Vilken tidskomplexitet har `add_cookie` för alternativ 1 respektive alternativ 2 ovan? Uttryck tidskomplexiteten i antal kakor i datastrukturen, k. (2)

(b) Vilken tidskomplexitet har `find_cookie` för alternativ 1 respektive alternativ 2 ovan? Uttryck tidskomplexiteten i antal kakor i datastrukturen, k. (2)

(c) Vilket av alternativen är bäst, givet att tomten i de flesta fall har ett stort antal kakor att välja mellan? Motivera ditt svar. (1)

(d) Föreslå hur vi kan förbättra alternativ 1 i fallet då vi sätter in nästan alla k kakor på en gång (dvs. vi sätter in k kakor först, sedan tar vi bort alla k kakor). Ditt förslag ska fortfarande använda arrayen (och ex.vis inte använda den som en heap). Du får lägga till variabler som kräver tar $\mathcal{O}(1)$ minne till datastrukturen om du vill. Vad blir tidskomplexiteten för detta fall för både originalimplementationen och din förbättrade version? (3)

Hur påverkar din föreslagna optimering fallet då tomten lägger till och äter kakor om vartannat (exempelvis: tomten lägger först till k kakor, sedan äter tomten 1, lägger till 1, och så vidare, k gånger). Din lösning behöver inte förbättra detta fallet, men du ska analysera tiden din lösning tar i även detta fall.

5. **Kakpyramider** (7 p)

Ett bra sätt att ställa fram kakor till tomten[2] är att bygga en kakpyramid av dem. En kakpyramid med höjd k består av k stycken runda kakor staplade på varandra. Kakorna har olika diameter. För att konstruktionen ska vara stabil så placeras kakan med störst diameter längst ner, kakan med näst störst diameter näst längst ner, och så vidare. Detta gör pyramiden mycket stabil, så att den inte faller ihop när tomten landar på taket.

Ett litet bageri har specialiserat sig på att producera just kakpyramider. Bageriet låter kunderna välja dels hur många kakor som pyramiden ska bestå av, och dels hur stor diameter varje nivå ska vara. När bageriet sedan bygger pyramiden så väljer bageriet lämpliga kakor från de s olika sorters kakor de har för att hitta kakor som är av lämplig storlek. Det finns totalt n kakor av alla sorter, fördelade så att det finns ungefär lika många av varje sort. Eftersom kakorna är bakade för hand så varierar diametern av kakorna något, även om de är av samma sort. Exempelvis finns det den 13:e december följande kakor på bageriet:

Sort	**Kakdiametrar** (mm)
Havreflarn	150, 153, 148, 120, 181, 179, ...
Hallongrotta	52, 48, 50, 43, 57, 60, ...
Chocolate chip cookie	101, 199, 203, 145, 132, ...
...	...

I och med att det är svårt att uppfylla kundernas önskemål har de bett dig att skriva ett program som hjälper till. Programmet får alla existerande kakor (dvs. datan i tabellen ovan) och kundens önskemål som indata. Kundens önskemål är en lista med den önskade diametern (i mm) för varje lager (ex.vis $(178, 100, 55, 44)$ för en 4-lagerspyramid). Önskemålet är alltid sorterat i fallande ordning. Programmet ska sedan välja en kaka för varje lager som är så nära önskemålet som möjligt (den kaka med minst skillnad i storlek jämfört med önskemålet). För att det ska bli en giltig pyramid måste alla utom den första kakan som väljs vara mindre än den förra kakan.[3] För exempelönskemålet skulle utdata kunna vara: *Havreflarn 179 mm*, *Chocolate chip cookie 101 mm*, *Hallongrotta 57 mm*, *Hallongrotta 43 mm*.

(a) Beskriv hur du kan implementera programmet som beskrivs ovan. (4)

Beskriv vilka **datastrukturer** du använder, samt den **algoritm du använder** för att lösa problemet. Beskriv gärna algoritmen kortfattat **i punktform** eller med **pseudokod**. Du behöver inte detaljerat beskriva sådant som finns i standardbiblioteket (men nämn kort vilken algoritm/datastruktur som du anser används av standardbiblioteket).

(b) Vilken tidskomplexitet har ditt program uttryckt i antal kakor i butiken (n), och antal kakor som kunden vill ha i sin pyramid (k). Du kan också använda s för antal sorter kakor om det är relevant för din lösning. (1)

(c) Efter att ha sett att ditt program fungerar bra ber de dig uppdatera programmet för att bättre hantera en hel dag på bageriet. De vill kunna starta programmet i början av dagen, sedan (1) lägga till nya kakor vartefter de bakas, och (2) kunna producera kakpyramider enligt kunders önskningar (vilket också tar bort kakorna från programmets datastrukturer). Givet detta nya användningssätt, behöver du revidera din lösning från (a) för att detta ska gå att göra effektivt? Beskriv i så fall hur. Oavsett: ge tidskomplexiteten för steg (1) och (2) separat (med bästa- och värsta fall om det är relevant). Använd k, n, och eventuellt s som i (b). (2)

[2] Se uppgift 4.

[3] Exempelvis: det finns kakor med diameter 20, 100 och 120 mm. En kund vill ha pyramiden $(100, 90)$. Programmet ska då välja 100 mm-kakan (bästa alternativet). Som kaka nummer två finns inget exakt alternativ. 120 mm-kakan är 30 mm ifrån, och 20 mm-kakan är 70 mm ifrån, så 120 mm-kakan är bäst. Vi kan däremot inte välja den eftersom den är större än första kakan, så programmet ska välja 20 mm-kakan.

6. **Professor of Cruel and Unusual Geography** (10 p)

Som nybliven Professor of Cruel and Unusual Geography vid Unseen University i Discworld är det viktigt för dig att göra ett gott intryck tidigt. Du har därför gjort ett program (till *Hex*, den magidrivna datorn) som förutspår var intressanta geografiska formationer finns (uttryckt som 2-dimensionella punkter (x, y) — världen är platt som vi alla vet), tillsammans med restid till närmaste välkända landmärke. Universitetet läcker dock magi ut i världen, och magin gör att restiden på två vägar som vid första anblick ser lika långa ut kan ha väldigt olika restid. Det är till och med så att samma väg kan ha olika restid beroende på i vilken riktning du färdas på den.

Image by: user Di, Creative Commons: `https://commons.wikimedia.org/wiki/File:Turtle_World.png`

I värsta fall är restiden så lång att man aldrig kommer fram (dvs. restiden är oändlig), men väljer man rätt väg så tar resan ingen tid alls (dvs. man är framme direkt när man börjar). Du kommer dock aldrig fram *innan* du börjar resa (dvs. restiden är alltid ≥ 0). Det är därför väldigt viktigt att planera sin resa noggrant!

Du börjar med att kombinera informationen från ditt program med en lista av vägar till välkända landmärken. Du får då tabellen nedan. De platser som börjar med en koordinat är förutspådda formationer som du vill utforska. Resten är välkända platser.

Från	**Till**	**Restid (minuter)**
Unseen University	$(10050, 650)$ Inverted mountain	42
Unseen University	Fifth Elephant Shaped Rock	0
Fifth Elephant Shaped Rock	Unseen University	40
Fifth Elephant Shaped Rock	$(8053, 928)$ Inside out cave	10
Fifth Elephant Shaped Rock	$(10050, 650)$ Inverted mountain	8
$(8053, 928)$ Inside out cave	Hogfather shaped stalactites	∞
Hogfather shaped stalactites	$(8053, 928)$ Inside out cave	500
...	...	...

Notera: Du kan anta att om det finns en väg från a till b så finns det en väg från b till a. Som du ser i tabellen så är det dock inte nödvändigtvis så att de har samma restid. Du kan också anta att det inte går att ”fastna” någonstans på grund av oändliga restider.

(a) Du vill ta dig till de förutspådda platserna för att utforska dem och dokumentera dem (om du är riktigt duktig kanske de blir döpta efter dig). För att maximera antalet platser du kan utforska vill du så klart hitta en väg med kortast möjliga restid. Du skriver därför ett program (till *Hex* återigen) för att hjälpa dig med detta. Programmets indata är tabellen ovan, tillsammans med var du är nu (namn på en plats), och vilken plats du vill besöka. Programmet ska sedan producera en beskrivning av den snabbaste vägen till platsen du vill besöka. (4)

Beskriv vilka **datastrukturer** du använder, samt den **algoritm du använder** för att lösa problemet. Beskriv gärna algoritmen kortfattat **i punktform** eller med **pseudokod**. Du behöver inte detaljerat beskriva sådant som finns i standardbiblioteket (men nämn kort vilken algoritm/datastruktur som du anser används av standardbiblioteket).

(b) Vilken tidskomplexitet har din lösning i (a), uttryckt i det totala antalet platser p och antalet vägar mellan dem v? (1)

(fortsättning på nästa sida)

(c) En bit in i din expedition inser du att även om man tar den snabbaste vägen mellan platser så kan det ta väldigt lång tid om man besöker dem i fel ordning. Du inser därför att det är värt din tid att modifiera programmet från (a) så att du kan mata in både vart du är nu och alla platser du vill besöka. Programmet ska då beräkna i vilken ordning det är bäst att besöka platserna. (4)

Notera: I och med att detta är dyrt att lösa optimalt räcker det med en approximation av den optimala lösningen. Din approximation ska ge en bättre lösning än att bara slumpa fram en ordning du besöker platserna i. Det är dock okej att din lösning inte hittar den globalt bästa ordningen. Detta går att göra snabbare än $\mathcal{O}(v^4)$.

Beskriv vilka **datastrukturer** du använder, samt den **algoritm du använder** för att lösa problemet. Beskriv gärna algoritmen kortfattat **i punktform** eller med **pseudokod**. Du behöver inte detaljerat beskriva sådant som finns i standardbiblioteket (men nämn kort vilken algoritm/datastruktur som du anser används av standardbiblioteket).

(d) Vilken tidskomplexitet har din lösning i (c), uttryckt i det totala antalet platser p och antalet vägar mellan dem v? (1)