UPPSALA
UNIVERSITET

Tentamen Daniel Westerlund

2020-03-11

Fråga 1 (3p)

Koden visar Insertion Sort. Den har tidenkomplexiten O(n^2) för en lista som är sorterad med fallande ordning t.ex. {5,4,3,2,1}.

Fråga 2 (3p)

Valet av datastruktur kommer att påverka algoritmen, kan man t.ex. välja en array får man konstant tid för access av data, medans en länkad lista kan ge O(n) i värsta fall för access av av data.

Valet av sökalgoritmer kommer även det vara avgörande för hur snabb algoritmen kommer att vara, har man t.ex. valt att lagra sin data i ett BST kan man göra sökningar på element med en tidskomplexitet på log(n), men har man valt att t.ex lagra dom i en osorterad array kan man behöva söka igenom hela array med en tidskomplexitet av O(n).

Fråga 3 a/ (4p)

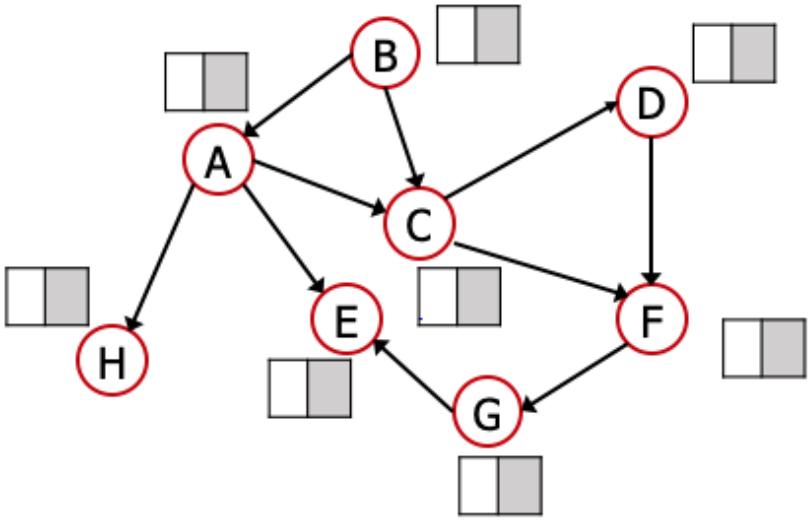
Används bland annat för att schema lägga saker, där saker är beroende av varandra. T.ex. att man måste ha gjort B för C. För att lösa en topologisk ordning gäller det att det finns minst en nod som inte har några krav på sig, t.ex. B i detta fall, som kan bli start noden. För att lösa en topologisk sortering använder man djupet-först sökning och således en stack för som datastruktur.
Om man väljer B som startnod, så lägger man dom noderna som B kan nå på stacken [B,A,C] Därefter kommer man att lägga de noder som A och C kan nå, [B,A,C,H,E,D,F] Så fyller man på med noderna på stacken.
Allt eftersom man besöker noderna så markerar man att man besökt dom.
När man besökt alla noder så kan man skriva ut stacken och som då är sorterad i topologisk ordning eftersom en stack är LIFO.

Fråga 3 b/ (2p)

Fyll i ingångsvärden FÖRE en topologisk sortering startar.

A → C, E,H
B → A,C

C → D,F
D → F
E →
F → G
G → E
H →



Fråga 4 (10p)

Med en HashMap kan man spara orden som nyckel, värdet är frekvensen av ordet som ökar för varje gång man hittar det i en text. HashMap har tidskomplexiten O(1) så man får access till objekten snabbt. HashMap använder en array i med länkade listor som datastruktur, vilket ger en minneskomplexitet på O(N). Dock om man vill få ut dom sorterade i fallande ordning ger det en tidskomplexitet på O(n) (Ev $O(n^2)$, men har för mig att det går på O(n)

Ett annat alternativ är att använda två arrayer där man i den ena sparar ordet, och på samma index i den andra arrayen sparar frekvensen av ordet. Nackdel är att det kan bli onödigt mycket kopierande av data fall man måste göra större array ofta, kopiering utav data tar mycket tid. Denna lösning kommer att ge O(1) förr access av element. Om man vill ha ut dom sorterade och använder shell short får man en tidskomplexitet på O(n log(n)). minneskomplexiteten blir O(N):

Jämför man dessa två algoritmer så ser man att HashMap är bättre på alla punkter som gäller hantering lagring/sökning av data och de har samma minneskomplexitet. Vilket gör att valet blir HashMap.

Fråga 5 (1p)

Sant. MergeSort är stabil.

Fråga 6 (3p)

methodA har tidskomplexitet O(n)

methodB har tidskomplexitet $O(n^2)$

methodC har tidskomplexitet O(n)

Fråga 7 (1p)

Sant.

Fråga 8 (3p)

in-place update - additional memory:
Om sorterings algoritmen behöver extra minne för att kunna sortera, som te.x Merge Sort som behöver O(n) extra minne, vilket gör den till inte ”in-place”. Medans t.ex. Insertion sort är in-place för den inte behöver något extra minne för att klara av att sortera, då den byter plats på element istället. Klassiskt fall för ”trade-off”, extra minne kan ge snabbare algoritm.

internal sorting - external sorting
Handlar om var man datan man ska sortera befinner sig. I internal sorting så är hela datan inläst i ram-minnet och man kan ha all data i en array för att få access till den.
I external sorting så har man så mycket data att man inte kan få plats med allt i minnet och man får använda hårddisken för att lagra en del data på och kommer göra att det går extremt långsamt eller sortera sin data i omgångar. Helst vill man ha alla data i ram-minnet (eller ännu bättre i cachen).

stable – unstable

Om sorteringen tar hänsyn till den interna ordningen bland elementen. Om mängden {A,B,C_1, D, C_2} resulterar i {A,B,C_1, C_2, D,}. Insertion sort är stabil då den respekterar den interna ordningen av element med samma värde. Selection sort är inte stabil efter som den byter plats på element på ett sätt som gör att i en större mängd skulle t.ex. C_2 kunna kommer före C_1.

Fråga 9 (3p)

En kö är fifo, dvs som en helt vanlig bankomat kö.
[A][B][B][C][D][D][E]
Metoder för en kö, är enqueu som lägger till ett element innan [A].
är dequeue som tar bort ett element ur kön , skulle ta bort [E].
Peek låter en titta på elementet som är först, [E].
Datastrukturen för en queue är oftast en länkad lista, vilket gör att om man lägger till ett element behöver den bara peka på det senaste elementet som lags till i listan.

Fråga 10 (1p)

Ange bokstaven för rätt utskrift!
Utskriften av kodsatsen blir enligt kolumn: A

Fråga 11 (5p)

```
public void doSomethingWithWord(String prefix, String word) {

   if(word.length() <= 1) {  1
      System.out.println(prefix + word);
   }
   else {
      for(int i = 0; i < word.length(); i++) {  2
         String current = word.substring(i, i + 1); 3
         String before = word.substring(0, i); 4
         String after = word.substring(i + 1); 5
         doSomethingWithWord(prefix + current, before + after);  6
      }
   }
}
```

1: Base case för rekursiva metoden. Kommer skriva ut
2: For-loop som kommer gå igenom alla bokstäver i word
3: Kommer välja ut bokstaven på plats i + 1.
4: Kommer välja ut den första bokstaven och bokstaven på plats i.
5: Kommer välja ut bokstaven efter platsen i.

6: Rekursiva anropet:
Iteration 1: Rekursiva anrop.
doSomethingWithWord(“a”, “lgoritmer”);
doSomethingWithWord(“al”, “goritmer”);
doSomethingWithWord(“alg”, “oritmer”);
doSomethingWithWord(“algo”, “ritmer”);
doSomethingWithWord(“algor”, “itmer”);
doSomethingWithWord(“algori”, “tmer”);
doSomethingWithWord(“algorit”, “mer”);
doSomethingWithWord(“algoritm”, “er”);
doSomethingWithWord(“algoritme”, “r”);

Skriver ut: algoritmer

Nya rekursiva anrop:
doSomethingWithWord(“algoritm”, “algoritm”);

```
doSomethingWithWord("al", "algoritm");
 doSomethingWithWord("alg", "oritmre");
  doSomethingWithWord("algo", "ritmre");
   doSomethingWithWord("algor", "itmre");
    doSomethingWithWord("algori", "tmre");
    doSomethingWithWord("algorit", "mre");
     doSomethingWithWord("algoritm", "re");
      doSomethingWithWord("algoritmr", "e");
```

Skriver ut: algoritmre

Kommer fortsätta så med väldigt många rekursiva anrop och skriva ut alla andra ord som går att bilda utav bokstäverna: algoritmer och sluta med att skiva ut det baklänges.

Fråga 12 (3p)

tid
tdi
itd
idt
dti
dit

Fråga 13 (3p)

(76,9)
(31,9)
(12,9)
(29,9)
(55,9)
(31,55)
(12,29)
(12,55)
(29,55)

Vilket ger totalt 8 inversioner.

Fråga 14 (4p)

DFS:
En djupet-först sökning kommer att börja söka så djupt ner i trädet den kan, vilket gör den bra för om man det man säker är långt ifrån root-elementet.
Är bra på att hitta en väg från A → B, men den behöver inte vara den bästa vägen.
Använder en stack som datastruktur.

BFS:
En bredden-först sökning kommer att börja söka av grannarna först för att sedan jobba sig nedåt i trädet, vilket gör den bra om man vet att det man söker är i närheten. Eftersom den söker på det sättet kommer den att hitta den bästa vägen mellan A → B. Använder Queue för datastruktur.

Båda två har samma tidkomplexitet, O(Noder + Kanter),

Fråga 15 (4p)

A: n^2
B: o(n)
C: n log(n)
D: 2^n