Lösningsförslag

för tentamen LET375, 2020–10–09

Algoritmer och datastrukturer

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 1, grundläggande: Komplexitet & korrekthet

(A) `s = [0], v = 1` gör att programmet fastnar i en oändlig loop.

(B) Inskränk startIdx/endIdx enligt följande:
```
startIdx = candidateIdx + 1;
```
respektive
```
endIdx = candidateIdx - 1;
```

(C) Vid varje steg i while-loopen så kommer antalet för nästa loopsteg att halveras [tack vare inskränkingen i endIdx/startIdx].
Detta kan ske maximalt lg(S) gånger.
Värstafallskomplexiteten är då O(log S).

Uppgift 2, grundläggande: Hashtabeller

Antag att de individualiserade värdena som ficks var:

C:14, E:11, B:10, G:12, A:15, F:12, D:16

Med insättningsordning C, E, B, G, A, F, D får vi följande tabell:

0 <u>B-0</u>

1 <u>E-0</u>

2 <u>G-0</u> F-0

3 --

4 <u>C-0</u>

5 <u>A-0</u>

6 <u>D-0</u>

7 <u>F-1</u> // 2 + 12 * 2 + 1 = 27 = 7 (mod 10)

8 --

9 --

Uppgift 3, grundläggande: Prioritetsköer

(A) Fungerande källkod för konstruktorn, `removeSecond` och `insert`.

```
public class SecondaryQueue<T extends Comparable<T>> {
   private final PriorityQueue<T> pq;

   public SecondaryQueue() {
      this.pq = new PriorityQueue<>();
   }

   public T removeSecond() {
      if(this.pq.size() < 2) {
         throw new NoSuchElementException();
      }
      T first = this.pq.remove();
      T second = this.pq.remove();
      this.pq.add(first);
      return second;
   }

   public void insert(T e) {
      this.pq.add(e);
   }
}
```

(B) Vi använder standardklassen `java.util.PriorityQueue`. Enligt dokumentationen (https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/PriorityQueue.html) så är den baserad på en priority heap.

Komplexiteten för konstruktorn är konstant, O(1), då det är den förväntade komplexiteten för att skapa ett nytt objekt och allokera plats för det.

`removeSecond` tar bort det minsta och näst minsta elementet från en heap, i värsta fall O(log n) där n är antalet elemet på kön. Sedan läggs det minsta elementet (men inte det näst minsta) elementet tillbaks vilket också är i O(log n) (amorterat, se nedan). Vi får 3 operationer som samtliga är i O(log n) vilket medför att hela `removeSecond` är i O(log n).

`insert` är också i *amorterad tid* O(log n) (det vi förväntar oss av en prioritetskö implementerad med en heap). *Amorterad tid* då heapen kan behöva växa; vid de punkterna behöver heapens värden kopieras till den nya, större, heapen för komplexitet O(n).

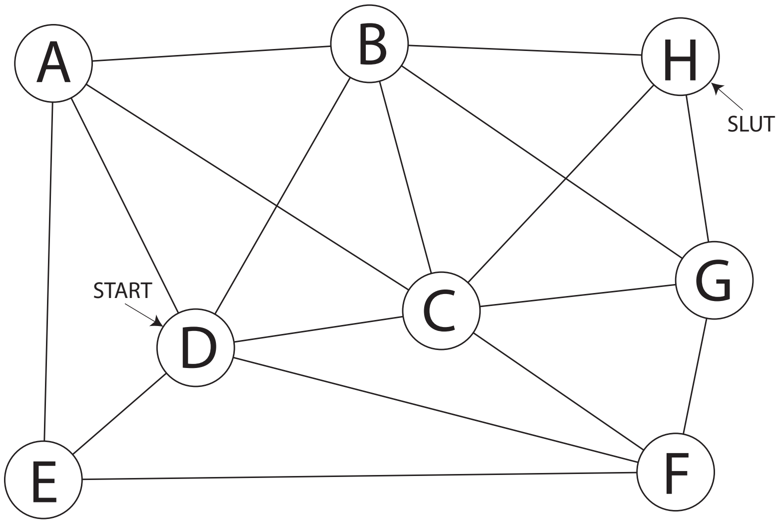
CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 4, grundläggande: Grafer

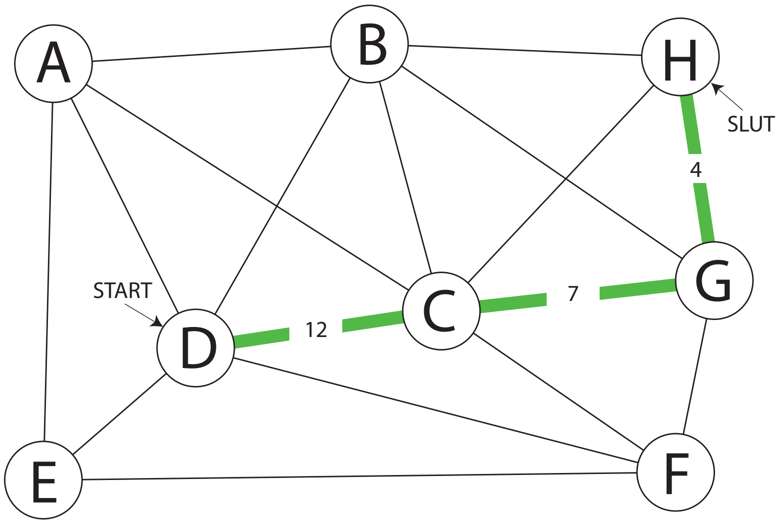
Antag R = 23, START = D, SLUT = H.

Kortaste vägen från D till H skall alltså ha kostnaden 23 och det minimala uppspännande trädet ska ha kostnad 32.

Grafen med start- och slutnod markerade



Vi fixerar en kortaste väg från D till H med kostnad 23, vägen markerad i grönt.

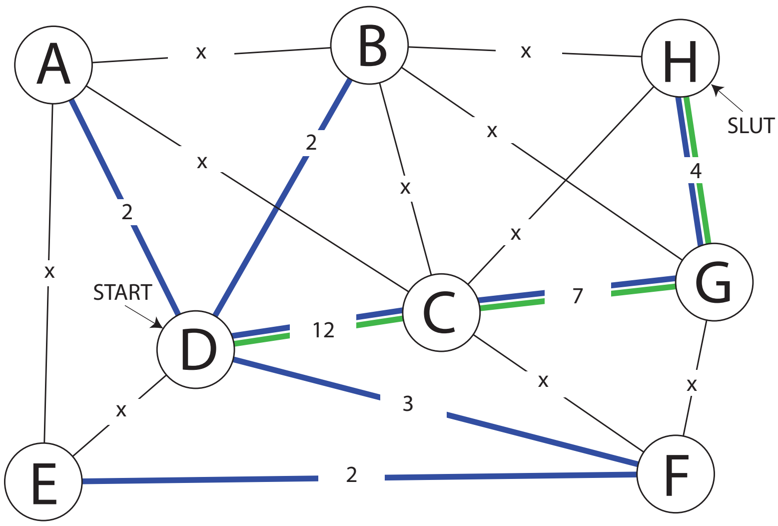


Nu kan vi låta kortaste vägen utgöra en del av vårt MST. Vi lägger till vikter på kanterna som krävs för att alla noder ska vara nåbara. Dessa kanter summerar till 9 (kravet var att MST ska ha kostnad R + 9).

Då det inte fanns någon övre gräns för vikter så kan vi sätta alla resterande vikter till godtyckligt högt, x, sådant att x > R.

(x kan sättas lägre men att sätta x > R är garanterat tillräckligt stort.)

MST markerat i blått.



Uppgift 5, grundläggande: Binära sökträd

(A)

```
// To Node.java
public int getBalance( ) {
   return getNumberOfDescendants(this.right) -
         getNumberOfDescendants(this.left);
}

private int getNumberOfDescendants(Node<T> ancestor) {
   if(ancestor == null) {
      return 0;
   }

   // Add 1 to include this ancestor as part of the count.
   return 1 + getNumberOfDescendants(ancestor.left) +
               getNumberOfDescendants(ancestor.right);
}
```

(B)

Varje nod kan referera till maximalt två barn som har en unik position i trädet. Om samma värde skulle läggas in i trädet flera gånger så kommer trädegenskapen att bibehållas (max 2 barn, inga cykler).

Varje anrop till `getBalance( )` kan resultera i att man som värst tittar på alla vänster- och alla högerättlingar exakt en gång. Dessa uppgår till (som värst) $n - 1$; den nuvarande noden räknas bort. Att uppdatera balansräknaren (”`balance`”) tar konstant tid. Vi får $O(n)\ O(1) = O(n)$.

(C)

Antag ett maximal obalanserat träd. I vår implementation fanns ingen information i varje nod om hur många ättlingar en viss nod har. Då följer att vi kan behöva gå ner maximalt till höger eller maximalt till vänster, vilket ger $n - 1 \in O(n)$ steg.

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 6, grundläggande: Sortering

(A)

```
S1: {18, 23, 19, 24, 23, 13, 28}
S2: {11, 21, 28, 28, 28, 28, 28}
```

(B)

```
static int[] findLRSS(int[] arr) {
    if (arr.length == 0) {
        return new int[] { 0, 0 };
    }
    int bestRisingLength = 0;
    int risingLength = 1;
    int beginIdx = 0;
    int bestBeginIdx = 0;
    int bestEndIdx = 1;
    for (int i = 1; i < arr.length; i++) {
        // Is the current element part of a rising sequence?
        if (arr[i - 1] <= arr[i]) {
            risingLength++;
            // Is the length of the current RS greater than the best
            // length seen so far?
            if (risingLength > bestRisingLength) {
                bestRisingLength = risingLength;
                bestBeginIdx = beginIdx;
                bestEndIdx = i + 1;
            }
        } else {
            // Not part of a rising sequence. Reset.
            risingLength = 1;
            beginIdx = i;
        }
    }
    return new int[] { bestBeginIdx, bestEndIdx };
}
```

Koden ovan kommer löpa i tid $O(n)$ där n = `arr.length`. Detta inses genom att observera att det enda loopvillkor som finns är `i` $< n$ aoch att `i` alltid ökar med 1, oavsett vad som händer i loopkroppen.

(C)

Applicerad på S1: (0, 2)

Applicerad på S2: (0, 8)

Uppgift 7, avancerad: Komplexitet

```
public static Long[] sortNoDuplicates(final Long[] array) {
   final List<Long> output = new ArrayList<>(); // 1. O(1)
   final Set<Long> set = new HashSet<>();       // 2. O(1)

   for(final Long s : array) {                  // 3. O(n)
      if(!set.contains(s)) {                    // 4-1. O(m)      4-2. average O(1)
         set.add(s);                            // 5-1. O(m)      5-2. average O(1)
         output.add(s);                         // 6. O(1), amortized.
      }
   }
   // You may assume sort is in O(n log n).
   Collections.sort(output);                    // 7. O(m log m)
   return output.toArray(new Long[0]);          // 8. O(m)
}
```

Antaganden:

Att skapa instanser av `ArrayList` och `HashSet` är i konstant tid.

Vi kan iterera över elementen i en array och varje iterationssteg (motsvarande `next()`) är i konstant tid.

Att se om en hashtabell innehåller ett element är i *värsta fall* linjärt i tabellens storlek (i nuvarande Java-implementation faktiskt logaritmiskt...), *genomsnittligt* konstant tid.

Att lägga till ett element till en hashtabell är i värsta fall linjärt (logaritmiskt i sena Java-versioner) och genomsnittligt konstant.

Att lägga till ett element till slutet av en `ArrayList` är amorterat konstant.

Att kopiera från en `ArrayList` till en array är linjärt i `ArrayList`ens storlek.

Värstafallskomplexitet

(1) + (2) + (3) * ((4-1) + (5-1) + (6)) + (7) + (8) =

$O(1) + O(1) + O(n)\ (O(m) + O(m) + O(1)) + O(m \log m) + O(m) =$

$O(nm) + O(m \log m) = O(nm + m \log m)$.

Då m inte är oberoende av n, vi har relationen $m \leq n$, så kan vi ersätta m med n och får $O(n^2 + n \log n) = O(n^2)$.

Genomsnittskomplexitet

(1) + (2) + (3) * ((4-2) + (5-2) + (6)) + (7) + (8) =

$O(1) + O(1) + O(n)\ (O(1) + O(1) + O(1)) + O(m \log m) + O(m) =$

$O(n) + O(m \log m) = O(n + m \log m)$.

Här kan och bör vi inte utnyttja att m som störst är n; $O(n)$ kan vara större än $O(m \log m)$ och omvänt.

I det föregående fallet så använde vi oss av att $O(n \log n) \subset O(n^2)$ men $O(n^2) \not\subset O(n \log n)$.

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 8, avancerad: Annorlunda traversering

Förberedelse:

Vi behöver ett sätt att hålla reda på hur mycket till vänster eller till höger vi är.

Utifrån en vänster/höger-angivelse behöver vi en funktion för att ge oss delmängdens prioritet.

Höger/vänster håller vi ordning på genom en balans-räknare. Är vi i mitten så är balansen 0, ett steg till vänster –1, två steg till höger 2, o.s.v.

Utifrån balansräknaren kan vi enkelt skapa en funktion, $p(b)$, som ger delmängdens prioritet utifrån balansen, b:

$$p(b) = \begin{cases} -2b & \mid \text{om } b < 0 \\ 2b + 1 & \mid \text{om } b \geq 0 \end{cases}$$

Lösning i O(n log n):

```
private static class PrioNode<T> implements Comparable<PrioNode<T>> {
    public final int prio;
    public final Node<T> node;

    public PrioNode(int prio, Node<T> node) {
        this.prio = prio;
        this.node = node;
    }

    @Override
    public int compareTo(PrioNode<T> o) {    return this.prio - o.prio; }
}

void printNodesFromMiddle() {
    final PriorityQueue<PrioNode<T>> pq = new PriorityQueue<>();
    traverseAndAddPQ(pq, this.root, 0);
    StringBuilder b = new StringBuilder();
    while(!pq.isEmpty()) {
        PrioNode<T> n = pq.remove();
        b.append(n.node.value + " ");
    }
    System.out.println(b.toString());
}

private void traverseAndAddPQ(final PriorityQueue<PrioNode<T>> pq,
        final Node<T> n, final int b) {
    final int prio = b < 0 ? -2 * b : 2 * b + 1;
    pq.add(new PrioNode<>(prio, n));
    if (n.left != null) {        traverseAndAddPQ(pq, n.left, b - 1);         }
    if (n.right != null) {       traverseAndAddPQ(pq, n.right, b + 1);        }
}
```

Motivering

Om vi tar hjälp av en prioritetskö så uppnår vi målet. Elementen som ligger i mitten har högst prioritet, följt av de element som ligger ett steg till vänster, ett steg till höger, o.s.v.

För att kunna använda Javas standardklass `PriorityQueue` så måste vi se till att elementen är jämförbara med avseende på prioritet, vilket är syftet med innerklassen `PrioNode`.

Om vi antar att `PriorityQueue` är implementerad med en binär heap så kommer insättning för varje element gå i tid $O(\log n)$ där n är köns storlek. Varje element sätts in maximalt en gång vilket ger $O(n)\ O(\log n) = O(n \log n)$.

När kön är klar övergår vi till utskrift. En detalj i sammanhanget är att vi använder `StringBuilder`, man kunde också tänkt sig att bara skriva ut direkt via `System.out.println()`.

Givet tidigare antagande om `PriorityQueue`s implementation så är `remove()` i $O(\log n)$.

Antalet gånger `remove()` anropas är n, för total $O(n)\ O(\log n) = O(n \log n)$ för utskriften.

Summa summarum får vi $2\ O(n \log n) = O(n \log n)$.

Bättre genomsnittskomplexitet (men sämre värstafallsdito)

En intressant hybrid som ger *genomsnittskomplexitet* $O(n + m \log m)$, där m är antalet olika prioriteter, kan uppnås m.h.a. en hashtabell.

Följande Map används:

```
Map<Integer, ArrayList<T>> elements = new HashMap<>();
```

Nya element läggs till med

```
ArrayList<T> pl = elements.get(prio); // Expected constant time.
if(pl == null) {
   pl = new ArrayList<>();elements.put(prio, pl);
}
pl.add(n);
```

För att skriva ut elementen i rätt ordning sorterar vi nycklarna i elements och itererar över dem för att få de element som hörde ihop med respektive prioritet. Det är alltså här $O(m \log m)$ kommer in.

Den genomsnittliga komplexiteten för den här lösningen blir då $O(n) + O(m \log m) = O(\mathrm{n} + m \log m)$.

Övning

Varför kan vi inte uppnå värstafallskomplexitet $O(n + m \log m)$ genom att istället för en hashtabell använda en dynamisk array av listor, där prioriteten indexerar in i arrayen? Indexering i en dynamisk array är ju en konstanttidsoperation.

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 9, avancerad: Riktade acykliska grafer

Körning

Låt ordningen hos `graphNodes` vara {D, A, C, F, E, B}. Låt kolumnen n visa värdet i isAcyclic och "n int" visa n i internalIsAcyclic. Om vi returnernar från rekursionen i flera steg skriver vi inte ut det.

För graf 1 får vi:

n	n int	visited (in)	visited (out)	tmpVisit (in)	tmpVisit (out)
D	D	-	D	-	D
	E	D	D E	D	-
A	A	D E	D E A	-	A
	D	D E A	D E A	A	-
C	C	D E A	D E A C	-	C
F	F	D E A C	D E A C F	-	F
	E	D E A C F	D E A C F	F	-
E	E	D E A C F	D E A C F	-	-
B	B	D E A C F	D E A C F B	-	B
	A	D E A C F B	D E A C F B	B	B
	E	D E A C F B	D E A C F B	B	B
	C	D E A C F B	D E A C F B	B	-

Då det inte i något fall skett att `origin` finns med i `tmpVisit` så är graf acyklisk.

För graf 2 får vi:

n	n int	visited (in)	visited (out)	tmpVisit (in)	tmpVisit (out)
D	D	-	D	-	D
	E	D	D E	D	D E
	F	D E	D E F	D E	D E F
	C	D E F	D E F C	D E F	D E F C
	B	D E F C	D E F C B	D E F C	D E F C B
	E	D E F C B	D E F C B	D E F C B	D E F C B

Värt att notera är att vi inte specificerat vilken ordning `origin.edgesTo` gås igenom.

När vi i graf 2 går från B till E så kunde vi lika gärna gått from B till A (till D).

I det sista steget (E) så finns E redan med i tmpVisit och grafen har alltså minst en cykel.

Komplexitet

Vi gör följande antaganden:

Låt mängderna `visited` och `tmpVisit` implementeras m.h.a. ett binärt balanserat sökträd. Komplexiteten för `add()`, `contains()` och `remove()` är då logaritmisk i antalet element. Det maximala antalet element `visited` kan innehålla är |V|, samma sak för `tmpVisit`.

I `isAcyclic` sker som flest |V| iterationer.

I `internalIsAcyclic` sker som flest |E| iterationer. *Vi är dock garanterade att en kant bara kan gås över en gång.*

Då *summan* av rekursiva anrop i `internalIsAcyclic` uppgår till (som mest) |E| och varje anrop utför (som mest) 2 `contains()` och 2 add() (alla fyra med värstafallskomplexitet O(|V|)) så kommer komplexiteten hos `internalIsAcyclic` vara O(E log V). Om origin inte har några utgående kanter så får vi O(log V).

Tack vare att varje kan bara kan besökas en gång så får vi en summa istället för en produkt i form av:

Den första termen nedan är fallet då vi inte har några utgående kanter. Den andra termen är fallet då en specifik nod har samtliga utgående kanter i grafen.

Sammantaget: O(V log V) + O(E log V) = O((E + V) log V).