Lösningsförslag till tentamen

Datastrukturer, DAT037, 2018-01-10

1. Båda looparna upprepas `n` gånger.

   - `s.pop()` tar $O(1)$, eventuellt amorterat.
   - `t.add()` tar $O(\log i)$ för i:te iterationen av första loopen.
   - `t.deleteMax()` tar $O(\log(\mathtt{n}-i))$ för i:te iterationen av andra loopen.
   - `l.addFirst()` tar $O(1)$.

   Övrigt tar konstant tid.

   Tidskomplexiteten är

   $$O(\sum_{i=1}^{\mathtt{n}}(1+\log i)+\sum_{i=1}^{\mathtt{n}}(\log(\mathtt{n}-i)+1)) = O(\mathtt{n}\log\mathtt{n}+\mathtt{n}\log\mathtt{n}) = O(\mathtt{n}\log\mathtt{n})$$

2.

5	8	8	10	14	13	16	12	13	15	20	20

3. Uppgiften säger inte vad `higher(x)` ska returnera om det inte finns tal större än `x` men `x` själv finns i mängden. Lösningen får därför returnera vad som helst i detta fall. I förslaget nedan returneras `null`.

   För betyg tre kan man t.ex. lagra heltalen sorterat i en lista. `add()` tar då $O(n)$, `higher()` tar $O(\log n)$ med binärsökning och `longestInterval()` tar $O(n)$ med linjärsökning efter längsta intervall mellan två efterföljande tal.

   För högre betyg kan man använda två balanserade sökträd, ett med heltal och ett med intervall sorterade på längd. För intervallen kan man ha en hjälpklass och en komparator definierad för denna. Denna komparator tittar på intervallens längd i första hand och på t.ex. vänstergränsen i andra hand.

```
class IntSet {
    private class Interval {
        int leftLimit, rightLimit;

        Interval(int leftLimit, int rightLimit) {
            this.leftLimit = leftLimit;
            this.rightLimit = rightLimit;
        }
    }

    private class IntervalComparator implements Comparator<Interval> {
        @Override
        public int compare(Interval o1, Interval o2) {
            int lengthDiff = (o2.rightLimit - o2.leftLimit) -
                             (o1.rightLimit - o1.leftLimit);
            if (lengthDiff == 0) return o1.leftLimit - o2.leftLimit;
            return lengthDiff;
        }

    }

    private TreeSet<Integer> s = new TreeSet<>();
    private TreeSet<Interval> intervals =
        new TreeSet<>(new IntervalComparator());

    public void add(int x) {
        if (s.contains(x)) return;
        s.add(x);
        Integer prev = s.lower(x);
        Integer next = s.higher(x);
        if (prev != null && next != null) {
            intervals.remove(new Interval(prev, next));
```

```
        }
        if (prev != null) {
            intervals.add(new Interval(prev, x));
        }
        if (next != null) {
            intervals.add(new Interval(x, next));
        }
    }
    public Integer higher(int x) {
        return s.higher(x);
    }
    public Integer longestInterval() {
        if (intervals.isEmpty()) return null;
        return intervals.first().leftLimit;
    }
}
```

`lower()` och `higher()` är definierade i Java collections framework så de får räknas som standardoperationer. `lower()`/`higher()` utför en vanlig sökning och när den når värdet eller ett tomt delträd returnerar den närmaste förfader för vilken noden eller det tomma delträdet befinner sig i höger/vänster delträd. Dessa operationer har logaritmisk komplexitet. `first()` är också definierad i Java collections framework så den får räknas som en standardoperation. Denna traverserar trädet genom att hela tiden gå till vänster. Denna operation har logaritmisk tidskomplexitet.

`s` innehåller n element och `intervals` innehåller $n-1$ stycken. Trädoperationerna tar alltså $O(\log n)$.

Konstruktorn/`empty()` tar $O(1)$. `add()` innehåller inga loopar och alla operationer tar max $O(\log n)$. `longestInterval()` innehåller också operationer som tar max $O(\log n)$.

4. För trea kan man göra en dfs som inte håller reda på om en nod redan besökts och räkna antal vägar man kommer till slutnoden. För högre betyg kan man göra en dfs som för varje nod håller reda på om den är besökt och, om den är besökt, hur många väger det finns från denna nod till slutnoden.

```
public int countPaths(int start, int dest) {
    int[] visCount = new int[adj.length];
    for (int i = 0; i < visCount.length; i++) {
        visCount[i] = -1;
    }
    visCount[dest] = 1;
    dfs(visCount, start);
    return visCount[start];
}

private void dfs(int[] visCount, int cur) {
    if (visCount[cur] >= 0) return;
    int n = 0;
    for (int i = 0; i < adj[cur].length; i++) {
        dfs(visCount, adj[cur][i]);
        n += visCount[adj[cur][i]];
    }
    visCount[cur] = n;
}
```

Talet `-1` i `visCount` representerar att noden inte besökts/antalet vägar till slutnoden är okänt.

Loopen i `countPaths` tar $O(v)$. Hjälpmetoden `dfs` anropas max $e + 1$ gånger därför att rekursiva anrop sker max en gång per kant eftersom loopen i `dfs` bara utgörs max en gång per nod eftersom if-satsen på första raden avslutar metoden om noden redan besökts. `visCount[cur]` sätts inte förrän i slutet av metoden men detta gör inget eftersom grafen är acyklisk vilket innebär att noden `cur` inte kommer besökas i de rekursiva anropen. Koden efter första raden i `dfs` utförs max v gånger och loopen itereras som sagt totalt e gånger. Tidskomplexiteten är alltså $O(v + e)$.

5.
```
public boolean contains(A e) {
    Node c = first;
    while (c != null) {
        if (c.contents.equals(e)) return true;
        c = c.next;
    }
    return false;
}

public void reverse() {
    Node c = first;
    Node p = null;
    while (c != null) {
        Node n = c.next;
        c.next = p;
        p = c;
        c = n;
    }
    first = p;
}
```

Loopen i `contains` upprepas i värsta fall n gånger. Varje iteration samt övrig kod tar konstant tid. Alltså $O(n)$.

Loopen i `reverse` upprepas n gånger. Varje iteration samt övrig kod tar konstant tid. Alltså $O(n)$.

6. För trea kan man använda en länkad lista av strängar. `add` lägger till element till listan. Det tar $O(1)$. `contains` gör linjärsökning. För varje element i listan tar jämförelsen $O(\ell)$ i värsta fall. Totalt tar det $O(\ell n)$. `restrictToPrefix` går igenom alla element och tar bort alla som inte börjar på s. Det tar $O(\ell n)$.

   För högre betyg kan man använda ett prefixträd. Varje nod har tio barn, ett för varje siffra. `add` och `contains` utförs som standardoperationerna för prefixträd. Dessa tar $O(\ell)$. `restrictToPrefix` utför en sökning efter s och på vägen raderas alla delträd förutom det barn man går vidare till.

```
class Set {
    private class Node {
        boolean member = false;
        Node[] children = new Node[10];
    }

    private Node root = null;

    public Set() {
    }

    public void add(String s) { ... }

    public boolean contains(String s) { ... }

    public void restrictToPrefix(String s) {
        Node n = root;
        int i = 0;
        while (n != null && i < s.length()) {
            int d = Character.digit(s.charAt(i), 10);
            for (int j = 0; j < 10; j++) {
                if (j != d) n.children[j] = null;
            }
            n.member = false;
            n = n.children[d];
            i++;
        }
        if (n == null) {
            root = null;
        }
    }
}
```

   Loopen i `restrictToPrefix` upprepas max ℓ gånger. Den inre loopen upprepas ett konstant antal gånger. If-satsen på slutet och alla atomära satser tar konstant tid. Alltså är tidskomplexiteten $O(\ell)$.