CMPSCI 311: Introduction to Algorithms **Spring 2018**

Homework 1 Solutions

Released 1/24/2018 Due 11:59pm 2/7/2018 in Gradescope

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

Instructions. You may work in groups, but you must individually write your solutions yourself. List your collaborators on your submission.

If you are asked to design an algorithm as part of a homework problem, please provide: (a) the pseudocode for the algorithm, (b) an explanation of the intuition for the algorithm, (c) a proof of correctness, (d) the running time of your algorithm and (e) justification for your running time analysis.

There are four questions, each worth 25 points total.

Submission instructions. This assignment is by 11:59pm on 2/7/2018 in Gradescope. Please submit a pdf file. You may submit a scanned handwritten document, but a typed submission is preferred.

1. **Stable Marriages**

   (a) **Stable Marriages: K&T Ch 1, Ex 5.**

   i. **Strong Instability.**

   Lets see if the Propose-and-reject algorithm from class works. Adapting this algorithm to the new context, when a college is deciding between proposing to two students of the same priority, the college will arbitrarily pick one of the two student and then maybe try the unchosen student in a future proposal. If a college proposes to the student, the student only accepts if the student prefers the new proposing college over their current.
   A college will be content with the student it ends with, as it will have tried all students who it preferred and been rejected or dropped each time until coming to the student it ends with. As students always upgrade, theres no chance that the student will reconsider a school after rejecting it. In other words, the colleges cannot get anything better than what they got, so there is no strong instability.
   As students only ever upgrade the college they are going to, we know that this algorithm will terminate based on the proof in class.

   ii. **Weak Instability.**

   Consider the case where $n = 2$. Both students value college 1 and college 2 equally. However, both colleges want student 1 more than student 2. If student 1 goes to college 1, college 2 will want student 1 and student 1 will be indifferent. The reverse will happen if student 1 chooses college 2. Either way, we have a weak instability, where the college who got stuck with student 2 wants a swap but student 1 is indifferent.

2. **Big-Oh and Asymptotics**

   (a) **Big-Oh**

i. $\lim_{n\to\infty} \frac{\frac{1}{2}n^2}{n^2} = \frac{1}{2}$. Thus the limit exists. Thus $c = 2$.

ii. Note that $c = 1$ is too small since $\frac{n(\log n)^3}{n}$ doesn't converge as n tends to infinity. Using L'Hospital's rule three times however ,

$$\lim_{n\to\infty} \frac{n(\log n)^3}{n^2} = \lim_{n\to\infty} \frac{(\log n)^3}{n} = \lim_{n\to\infty} \frac{3(\log n)^2}{n} = \lim_{n\to\infty} \frac{6\log n}{n} = \lim_{n\to\infty} \frac{6}{n} = 0$$

and hence $c = 2$ is sufficient.

iii. $\sum_{i=0}^{\lceil \log n\rceil} \frac{n}{2^i} = n[1 + \frac{1}{2} + \frac{1}{2^2} + ... + \frac{1}{2^{\lceil \log n\rceil}}] \geq n$. Since $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$, $f(n) \leq 2n = O(n)$ and $c = 1$.

iv. Since $\sum_{i=1}^{n} i^3 = [\frac{n(n+1)}{2}]^2$, we know that $f(n) = O(n^4)$. If you didn't know this formula, the easier proof is

$$\sum_{i=1}^{n} i^3 \leq \sum_{i=1}^{n} n^3 = n^4.$$

To prove that this is tight, note that

$$\sum_{i=1}^{n} i^3 \geq \sum_{i=n/2}^{n} i^3 \geq \sum_{i=n/2}^{n} (n/2)^3 = (n/2)^4 = \Omega(n^4).$$

v. $2^{(\log n)^2} = n^{(\log n)}$. This is not a polynomial function of n. Hence no such c value exists for which $f(n) = O(n^c)$.

(b) **Asymptotics. K&T Ch 2, Ex 6.**

i. Add up $A[i] + A[i+1] + \ldots + A[j]$ runs for O(n) time. Each of the for loops runs at most n times. Thus complexity of the algorithm is $O(n^3)$.
Speaking specifically, the addition $A[i] + A[i+1] + \ldots + A[j]$ runs for O(j-i+1) time (which is again bounded by O(n)). Consider cases when $i \leq n/4$ and $j \geq 3n/4$. Then $j - i + 1 \geq 3n/4 - n/4 + 1 > n/2$. Thus the add up loop runs for at least $n/2$ iterations. We are left to analyze the number of cases when this scenario occurs. The number of such (i, j) pairs are $(n/4)^2$. Thus the total running time is $n/2 \times (n/4)^2$, which means the algorithm is $\Omega(n^3)$.
Hence, the algorithm is $\Theta(n^3)$.

ii. Consider the following algorithm:

```
For $i = 1, 2, \ldots, n$
    For $j = i, \ldots, n$
        if $j == i$
            Store $B[i,j] = A[i,j]$
        else
            $B[i,j] = B[i,j-1] + A[i,j]$
        end if-else
        Store in $B[i,j]$
    end For
end For.
```

The whole algorithm runs in two for loops. Each of these loops runs for n times. Thus the algorithm is bound by $O(n^2)$.

3. **Disrupting Communication Networks.**

(a) The depth first search will only return to the root node if all the subtrees have been explored. Thus the subtrees rooted at the immediate children of the node r are disjoint. This is based on the fact that all non-tree edges in the DFS tree describe ancestor-descendant relationship, so there are no edges between the subtrees of the root. Therefore $f(r)$ is the number of children of r in the DFS tree.

(b) No descendant of v has a non-tree edge to an ancestor of v. The components formed by removing the vertex v in the DFS tree is equal to the number of its children and its parent. Thus $f(v) =$ number of children of $v + 1$.

(c) Let us perform a DFS on the tree. At each node v, we set $d(v) = d(v_p) + 1$, where v_p is the parent of v. We do this because using DFS, we already know the parent before any child node is visited and as such we can generate the values as a function of the depth of the parent node.

(d) For any node $v \neq r$, one of the connected components is the parent of v. Now for each children w_i of v if $up(w_i) == d(v)$, then w_i is not connected to the component of v's parent. Note that $up(w_i)$ is never strictly greater than $d(v)$ since w_i is a child of v. Thus, $f(v) = 1$(corresponding to the parent's component) $+ \sum_{i=1}^{k} \mathbb{1}(up(w_i) == d(v))$.

(e) Perform a DFS and store the depth information of each of the nodes in the graph G. This takes $O(|V| + |E|)$. Now perform a recursive DFS on the tree, so we process the leaves first, and more generally we process all the children of a node v before we process v. In this traversal, for each node v, we compute $x = \min_{u:(u,v)\in E} d(u)$ and $y = \min_{\text{children } w \text{ of } v} up(w)$. With these two values, we compute $up(v) = \min\{x, y\}$. For the root r, set $up(r) = \infty$.

Since in this second traversal, we process the nodes from the leaves to the root, the up values for the children are always available when we are processing a node, so this runs in $O(|V| + |E|)$ time.

(f) Traversing the DFS tree, we can calculate $f(v)$ as formulated in part(d) from the up array created in part(e). For the root, we follow part(a). The traversal can be done again as a DFS or a BFS, which takes $O(|V| + |E|)$ time.

As a reference for the future, here is the kind of pseudocode we are looking for:

```
Perform DFS starting from some node $r$ to build a DFS tree.
Set $d(r) = 0$.
Perform iterative DFS on the tree setting $d(v) = 1 + d(p)$ for every parent-child edge $(p, v)$.
// Make sure you set $d(p)$ before processing the children.
Perform recursive DFS and set $up(v) = \min\{x, y\}$ where $x, y$ defined above.
// Make sure you process the children before the parent.
Visit each node and set $f(v) = 1 + \sum_{\text{children} w_i} \mathbb{1}\{up(w_i) == d(v)\}$.
For the root set $f(r)$ to be the number of children of $r$ in the DFS tree.
```

4. **Graphs**

(a) **K&T Ch3.Ex12. Graph construction**:

For each person assign two nodes. For P_i^{th} person let the two nodes be b_i and d_i. e_i is a directed edge from b_i to d_i. If any person P_j has died before P_i then join the nodes d_j and b_i with a directed edge. If we know that P_i and P_j were alive at the same time, we add two directed edges, one from b_i to d_j and one from d_i to b_j.

We claim that that this graph has a topological ordering if and only if the facts are consistent. First, if the graph has a topological ordering, then in this ordering it must be the case that for all people P_i, b_i comes before d_i (otherwise there would be a back edge). Moreover, for any fact the topological ordering must make it so that the edges point forward, which by construction imply that the ordering satisfies the constraints imposed by those facts. As an example, uppose that P_i and P_j were alive at the same time. Then the topological ordering restricted to just the nodes corresponding to P_i and P_j must be one of $(b_i, b_j, d_i, d_j), (b_i, b_j, d_j, d_i), (b_j, b_i, d_i, d_j), (b_j, b_i, d_j, d_i)$. Otherwise one of edges in our graph would be a back edge.

Conversely, if the facts are consistent, then there must be some ordering of the birth and deaths of all the people that satisfies all the facts. Since the edges in our graph correspond exactly to these facts, this ordering will have no back edges in the graph. **Solution**:

Our job is thus to check if this graph we have constructed has a topological ordering. To do so, just run the topological ordering algorithm we saw in lecture. If that algorithm succeeds, the ordering you found is consistent with the facts, so just return it. If the algorithm fails, output inconsistent.

Time: Topological sorting can be implemented in $O(|V| + |E|)$ time for the graph. If there are n people and m facts, the graph is $2n$ vertices and $\leq n + 2m$ edges. So the runtime is $O(n + m)$.

5. **(0 points).** How long did it take you to complete this assignment?