Mid-term. May 4, 2015

CS180: Algorithms and Complexity
Spring 2015

Guidelines:

- The exam is closed book and closed notes. Do not open the exam until instructed to do so. You will have to stop writing at 5:50 PM.
- Write your solutions clearly and when asked to do so, provide complete proofs. You may use results we proved in class without proofs as long as you state what you are using. You may also use any algorithms we covered in class as sub-routines without giving full descriptions (or rewriting the whole algorithm again).
- Most importantly, make sure you adhere to the policies for academic honesty set out on the course webpage. The policies will be enforced strictly.

Problem	Points	Maximum
1		5
2		10
3		15
4		25
5		25
6		10
7		30
Total		120

Name:
UID:
Section:

1 Problem

For each pair (f, g) below indicate the relation between them in terms of O, Ω, Θ. For each missing entry, write-down Y (for YES) or N (for NO) to indicate whether the relation holds (no need to justify your answers here). For example, if $f = O(g)$ but not $\Omega(g)$, then you should enter Y in the first box and N in the other two boxes. Similarly, if $f = \Theta(g)$, then you should enter Y in all the boxes. [5 points].

f	g	O	Ω	Θ
$\log_2 n$	$\log_{10} n$	Y	Y	Y
$2^{(\log_2 n)^4}$	n^5	N	Y	N
$n^3 \cdot 2^n$	3^n	Y	N	N
$2^{\sqrt{\log_2 \log_2 n}}$	$\log_2 n$	Y	N	N
$n!$	n^n	Y	N	N

2 Problem

Answer true or false for the following (no need for explanations) [10 points]:

- Consider an instance of the stable matching where a doctor D_1's first choice is H_1 and H_1's first choice is D_1. Is it true that in every stable matching D_1 should be matched to H_1?

  TRUE. If not, both D_1, H_1 would be better off by dropping their partners and pairing with each other.

- Consider an instance of the stable matching problem and a candidate perfect matching M where one doctor gets her top choice and one hospital gets its top choice, while every other doctor and hospital get their second choice. Is M necessarily a stable matching?

  FALSE. Take the following example for instance $D_1 : H_1 > H_2$, $D_2 : H_1 > H_2$, $H_1 : D_1 > D_2$, $H_2 : D_1 > D_2$. Then, the matching $D_1 \leftrightarrow H_2$ and $D_2 \leftrightarrow H_1$ satisfies the condition but is not stable.

- If α is an n'th root of unity, then $\sum_{j=0}^{2n-1} \alpha^j = 2n$.

  FALSE. It is true only if $\alpha = 1$.

- Dijkstra's algorithm when run on an unweighted undirected graph starting from a vertex s gives us (by looking at the graph formed by *Parent* links) the breadth-first-search tree starting from s.

  TRUE. We stated this in class.

- Any polynomial $P : \mathbb{R} \to \mathbb{R}$ of degree d is uniquely determined by its evaluations at d distinct points $x_1, \ldots, x_d$.

  FALSE. You need at least $d + 1$ evaluations for a degree d polynomial.

3 Problem

Given a connected undirected graph $G = (V, E)$ as input (in adjacency list representation), give an algorithm to check if G is a tree. You must analyze the time-complexity of your algorithm but don't need to prove correctness. For full credit, your algorithm should be correct and run in time $O(|V| + |E|)$. [15 points]

Solution: The best way to do this is to use breadth first search but check if you visit a vertex twice. However, you need to be careful to keep track of parent vertices. Let s be an arbitrary start vertex.

```
1. Initialize $Discovered[v] = 0$ and $Parent[v] = \emptyset$ for all $v \neq s$.
2. Set $j = 0$, let $L_0 = \{s\}$ and set $Discovered[s] = 1$, $Parent[s] = s$.
3. While $L_j \neq \emptyset$:
   (a) Set $L_{j+1} = \emptyset$.
   (b) For every vertex $u \in L_j$:
   (c) For every edge $\{u, v\}$ from $u$:
       If $Discovered[v] = 1$ and $Parent[u] \neq v$, RETURN NOT-A-TREE.
       Elseif $Discovered[v] = 0$, set $Discovered[v] = 1$, $Parent[v] = u$ and add $v$ to $L_{j+1}$.
   (d) Increment $j$.
4. Check if $Discoverd[v] = 1$ for all vertices $v \in V$. If yes, RETURN TREE, else RETURN NOT-A-TREE.
```

The correctness of the algorithm follows from what we did in class. For time complexity note that each edge of the algorithm is only examined once and we spend $O(|V|)$ time in the initialization process (and the final check). Therefore, the total time is $O(|V| + |E|)$.

Remark: Many of you gave the following algorithm: Run BFS (or DFS) to get a tree T - check if G equals T if yes, output Tree, else output not a tree. However, this is incomplete as it does not specify how to check if G equals T in time $O(|V| + |E|)$. This solution got you 10 points typically.

4 Problem

Given the coefficients of a polynomial P of degree d and an integer k as input, give an algorithm to compute the coefficients of the polynomial $P(x)^k$. For example, if your input is $(1, 1)$ (to denote the polynomial $1 + x$) and $k = 3$, your output should be $(1, 3, 3, 1)$ to denote the polynomial $(1 + x)^3 = 1 + 3x + 3x^2 + x^3$. Similarly, if the input is $(1, -3)$ (to denote $P = 1 - 3x$), $k = 3$, your output should be $(1, -9, 27, -27)$.

To get full credit, your algorithm should be correct, run in time $O((k \cdot d) \log(k \cdot d))$ and you must analyze the time-complexity of your algorithm (no need to prove correctness). [25 points]

Remark: Here we measure time-complexity as in the fast-polynomial multiplication algorithm, where we count complex additions and multiplications as unit-cost.

Solution: This is very similar to problem two from assignment two. The one difference is that instead of using Karatsuba's algorithm we will use the fast-polynomial-multiplication algorithm we did in class. Let FMULT(P,Q) be the algorithm we discussed in class for multiplying two polynomials P, Q of degree at most n that takes time $O(n \log n)$.

POWERP(k): (Input - polynomial P, positive integer k; Output - P^k).

```
1. If $k = 0$, RETURN 1.

2. If $k$ is even:
   Compute $Q = $ POWERP($k/2$).
   RETURN FMULT($Q, Q$).

3. If $k$ is odd:
   Compute $Q = $ POWERP($(k-1)/2$).
   RETURN FMULT($P$, FMULT($Q, Q$)).
```

Analysis. It is easy to show the correctness by strong induction. The base-case, $k = 0$, is clearly correct as it is hardcoded into the algorithm. If it is true for all integers $j < k$, then the value of Q as defined in the algorithm would be what it should be and hence $P(x)^k$ would be computed exactly.

To compute the time-complexity, note that the polynomial Q computed in the algorithm has degree at most kd. Therefore, if we denote the time-complexity for input k as $T(k)$, then we have the recurrence

$$\begin{aligned} T(k) &\leq T(\lfloor k/2 \rfloor) + (\text{cost of } \textsc{Fmult}(Q,Q)) + (\text{cost of } \textsc{Fmult}(P, Q^2)) \\ &= T(\lfloor k/2 \rfloor) + O((kd)\log(kd)) \leq T(\lfloor k/4 \rfloor) + O((kd/2)\log(kd)) + O(kd\log(kd)) \leq \dots \\ &\leq O(1)(\log kd)\,(kd + (kd/2) + (kd/4) + (kd/8) + \cdots) = O((kd)\log(kd)). \end{aligned}$$

Remark: A reasonably direct solution for the problem is to have a simple for loop and repeatedly multiply P k times using FMULT. This takes time $\sum_{i=1}^{k} O(id\log(id)) \approx O(k^2 d \log(kd))$. This solution was typically given 12 points if the time-complexity analysis was wrong and 15 if the time-complexity was analyzed correctly.

An unfortunate mistake that many people made was to make **two** recursive calls to the powering function for $k/2$. This leads to the recurrence, $T(k) \leq 2T(\lfloor k/2 \rfloor) + O(kd(\log kd))$. The solution to this recurrence is not $T(k) = O((kd)(\log kd))$, but $T(k) = O(kd(\log d)(\log k))$ - you lose another logarithmic factor. This solution was typically given 19 points if the time-complexity analysis was wrong and 21 if the time-complexity was analyzed correctly.

5 Problem

Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that G is an *ordered graph* if it has the following properties.

1. Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

2. Each node except v_n has at least one edge leaving it. That is, for every node $v_i, i = 1, 2, \ldots, n-1$, there is at least one edge of the form (v_i, v_j).

Given an ordered graph $G = (V, E)$ (in adjacency list representation), give an algorithm to compute the number of paths that begin at v_1 and end at v_n.

You must analyze the time-complexity of your algorithm (no need to prove correctness). To get full-credit your algorithm must be correct and run in time $O(|V| + |E|)$. [25 points]

Remark: You can assume that adding two numbers takes constant time in your time-complexity calculations.

Solution: The algorithm is quite similar (but simpler) than the problems from problem set 3. Let $Path[i]$ denote the number of paths from v_i to v_n.

1. Set $Path[n-1] = 1$. This is correct by property (2) of the graph.
2. Initialize $Path[i] = 0$ for $i \neq n-1$.
3. For $i = n-2$ to 1:
   For each edge $(v_i, v_j) \in E$, set $Path[i] = Path[i] + Path[j]$.
4. RETURN $Path[1]$.

To analyze the time-complexity, note that the initialization takes $O(n)$ time. After, that each edge of the graph results in us performing one addition. Therefore, the total time-complexity is $O(|V| + |E|)$.

To prove correctness, note that the $Path$ variables satisfy the recurrence

$$Path[i] = \sum_{j:(v_i,v_j)\in E} Path[j].$$

This is because to get to v_n from v_i we can take any of the edges, say $(v_i, v_j) \in E$, out of v_i and then take any path leading from v_j to v_n. The desired output is $Path[1]$.

The correctness now follows from downward induction on i. It is correctly computed for $i = n, n-1$. Now, suppose $Path[j]$ is computed correctly for $j > i$. Then, $Path[i]$ will be computed correctly as we are using the right recurrence. The correctness now follows by induction.

Remark: Several people gave a recursive solution without memorization - which is highly expensive (as bad as brute-force). This solution was typically given 16 points of 25 (if the recurrence relation was identified correctly).

6 Problem

Decide whether the following statement is true or false. If it is true, give a short explanation (no need for a formal proof - a high-level description is enough). If it is false, give a counter-example.

Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G, with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? T must still be a minimum spanning tree for this new instance. [10 points]

Solution: The statement is TRUE. The proof follows from any of the three algorithms for computing a minimum spanning tree that we discussed in class. In all three algorithms the order in which the edges are considered only depends on the relative ordering of the weights and not on the actual weights themselves. Squaring all the weights does not change the relative ordering of the edges and hence all the algorithms will proceed in the same way as with the original weights.

7 Problem

Given an undirected graph $G = (V, E)$, a subset of vertices $I \subseteq V$ is an independent set in G if no two vertices in I are adjacent to each other. Let $\alpha(G) = \max\{|I| : I \text{ an independent set in } G\}$. The goal of the following questions is to give an efficient algorithm for computing an independent set of maximum size in a tree. Recall that a *leaf* in a graph is a vertex of degree at most 1 and that every acyclic graph (graph without any cycles) has at least one leaf.

Let $T = (V, E)$ be an acyclic graph on n vertices.

1. Prove that if u is a leaf in T, then there is a maximum-size independent set in T which contains u. That is, for every leaf u, there is an independent set I such that $u \in I$ and $|I| = \alpha(T)$. [15 points]

2. Give the graph T as input (in adjacency edge representation), give an algorithm to compute an independent-set of maximum size, $\alpha(T)$, in T. To get full credit your algorithm should run in time $O(|V| \cdot |E|)$ (or better) and you must prove correctness of your algorithm. You don't need to analyze the time-complexity of your algorithm and it is sufficient to solve this problem assuming part (1) (if you want) even if you don't solve it. [15 points]

Solution: Let us first show part 1. Consider a leaf $u \in T$ and let I be a maximum size independent set I. Note that if $degree(u) = 0$, then u must be in I (as else we can add u to get a larger independent set) so we are done.

Now suppose $degree(u) = 1$ with v being the sole neighbor of u in T. If $v \notin I$, we can again add u to I to get a larger independent set so v must be in I. In this case, form a new set $I' = I \setminus \{v\} \cup \{u\}$ by removing v and adding u. Clearly I' is an independent set as we only added u but removed the only other possibly conflicting vertex v. We are now done as $|I'| = |I| = \alpha(T)$.

The idea behind part 2 is to use the above property to build the independent set greedily by making "safe" choices as we did for spanning trees:

```
1. Let $I = \emptyset$. Initialize a new graph $G = T$
2. While $G \neq \emptyset$:
    (a) Find a leaf $u \in G$ and add $u$ to $I$.
    (b) Remove $u$ and any neighbor of $u$ (if exists) from $G$.
3. Return $I$.
```

Analysis: Call a subset of vertices A *feasible* if there is a maximal-size independent set in T which contains A. Call a vertex u safe for A if $A \cup \{u\}$ is feasible. The correctness will follow if we show the following invariant: The set I in our algorithm is always feasible.

To prove the claim, suppose it is true at the start of an iteration. Then, note that as G contains all those vertices which are not adjacent to elements of I and I is feasible, for any maximal independent set I' in G, $I \cup I'$ is a maximal independent set in T. Hence, by part (1) a leaf $u \in G$ is safe for I as we can find a maximal-size independent set $I' \in G$ which contains u.

Remark: For both parts, many people gave an argument based on doing BFS (or DFS) and taking all "odd" level vertices or "even" level vertices to be the independent set (depending on whichever is larger). However this doesn't quite work - while each of these is an independent set, they may be far from being the maximum. This was given 5 points typically (for each part).

Another argument was to try a dynamic programming approach based on the recurrence: for any vertex $v \in T$

$$\alpha(T) = \max \begin{cases} \alpha(T \setminus \{v\}) \\ 1 + \alpha(T \setminus (\{v\} \cup \{\text{Neighbors of } v\})) \end{cases} .$$

The recurrence follows by considering two cases of whether v is in a maximal-independent set or not. The problem with this approach is that one has to memorize and memoizing this recurrence is not so easy. This solution typically got you 10 points for part (2).