MIDTERM EXAM: Solutions **CMSC 451 (Spring 2016)**

1. *Short answer questions.*

   For the following questions, explanations are not required, but can be given for partial credit.

   (a) For each of the following functions, indicate whether $f(n)$ is in o, ω, or Θ of $g(n)$. In this problem, log denotes the base-2 logarithm.

   i. [1 point] $f(n) = 10^{n/2}$, $g(n) = 5^n$
   Solution: $f(n) \in o(g(n))$, since $\sqrt{10} < 5$

   ii. [1 point] $f(n) = n^2 + 10^{100}\, n \log n$, $g(n) = 10^{100}\, n^2 + n \log n$
   Solution: $f(n) \in \Theta(g(n))$, since both expressions are $\Theta(n^2)$

   iii. [1 point] $f(n) = \log \sqrt{n}$, $g(n) = \sqrt{\log n}$
   Solution: $f(n) \in \omega(g(n))$, since $\log \sqrt{n} = \frac{1}{2} \log n$ is quadratically larger than $\sqrt{\log n}$

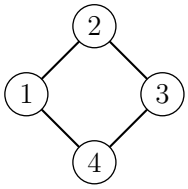
   (b) [2 points] Recall that in the Gale-Shapley algorithm as discussed in class, companies make offers to students. Which of the following describes how the sequence of students engaged to a given company changes over time? The company's preference for the student to whom it is engaged...

   i. ... strictly increases in each iteration.
   ii. ... may either increase or stay the same from one iteration to the next.
   iii. ... may either decrease or stay the same from one iteration to the next.
   iv. ... strictly decreases in each iteration.

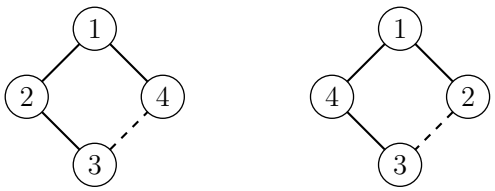
   Solution: iii. ... may either decrease or stay the same from one iteration to the next.

   (c) [3 points] Give an example of a graph for which the breadth-first search algorithm (as presented in class) can output different BFS trees starting from some particular initial vertex, depending on the order in which the neighbors of vertices are considered. Your example should involve as few vertices as possible.

   Solution: Consider a 4-vertex cycle:



   Then BFS from vertex 1 produces the tree on the left if vertex 2 is added before vertex 4, but it produces the tree on the right if vertex 4 is added before vertex 2:



(d) [2 points] Recall that in the *interval scheduling problem*, we are given a list of start times s_i and finish times f_i for all $i \in \{1, 2, \ldots, n\}$, and our goal is to find a largest possible subset of the intervals $[s_i, f_i]$ such that no two intervals overlap. Consider the following greedy algorithm for interval scheduling:

Sort the intervals by *starting* times
While there are any remaining intervals
    Add an interval with the *latest* starting time to the schedule
    Delete all conflicting intervals
Endwhile

True or false: This algorithm outputs an optimal schedule.

Solution: True. This is equivalent to the algorithm discussed in class (schedule in increasing order of finishing time) if we reverse the direction of time, and this reversal does not affect the maximum number of intervals that can be scheduled.

2. *Maximum-cost path in a weighted DAG.* [10 points]

You are given a directed acyclic graph $G = (V, E)$. Each edge $(u, v) \in E$ of this graph is associated with a weight $w(u, v)$, which can be any positive real number. The cost of a path is defined to be the sum of the weights of the edges along the path. Present an efficient algorithm that computes the maximum cost of any path in G. Justify your algorithm's correctness and establish its running time.

Solution: This problem is similar to problem 3(b) of assignment 2, except that the edges of the graph are weighted. However, essentially the same strategy can be used here.

First we obtain a topological ordering of G in time $O(m+n)$, where $n = |V|$ and $m = |E|$. Denote the vertices of G, ordered according to the topological ordering, as $v_1, v_2, \ldots, v_n$. Let c_i denote the highest cost of any path that ends at vertex v_i. Since all edges in the DAG point from vertices with smaller index to vertices with larger index in the topological ordering, the values c_i satisfy the following recurrence relation:

$$c_i = \begin{cases} 0 & \text{if } v_i \text{ has no incoming edges} \\ \max\limits_{\substack{j<i \\ (v_j, v_i) \in E}} c_j + w(v_j, v_i) & \text{otherwise.} \end{cases} \tag{1}$$

We can compute c_i for every i in time $O(m + n)$ since for every vertex we consider all of its neighbors, and do a constant amount of additional work. Once we have computed c_i for all i, the final answer is simply $\max_i c_i$, which can be computed in time $O(n)$. Overall, the running time of the algorithm is $O(m + n)$.

3. *Borůvka's algorithm.*

In 1926, Otakar Borůvka published a method for constructing an efficient electrical network for the country of Moravia. This minimum spanning tree algorithm predates both the Kruskal and Prim algorithms. Given a connected, undirected input graph G with positive edge weights w such that no two edges have the same weight, it works as follows:

```
Borůvka(G, w):
    Let F be a forest with the same vertices as G, but no edges
    While F has more than one component
        Let S be an empty set
        For each component C of F
            Let {u, v} be the lowest-weight edge with u in C and v not in C
            Add edge {u, v} to S
        Endfor
        Add the edges in S to the forest F
    Endwhile
```

In both parts of this problem, you should assume that G is a connected, undirected graph and that no two edges of G have the same weight.

(a) [9 points] Prove that Borůvka's algorithm outputs a minimum spanning tree of G. (You are *not* asked to analyze its running time.)

Solution: First we show that every edge added by the algorithm must be part of any minimum spanning tree. Recall the lemma we used to show this for the Kruskal and Prim algorithms: the lowest-weight edge crossing any nontrivial cut in G must be part of any minimum spanning tree of G. The edge $\{u, v\}$ added by the algorithm is the lowest-weight edge crossing the cut $(C, V \setminus C)$, so it satisfies the conditions of the lemma, and hence is part of any minimum spanning tree.

It remains to show that the algorithm outputs a spanning tree. Adding a single edge between two components of a forest does not create a cycle. It is possible that when we consider component C, we add some edge joining it to another component C', and when we consider C', we add some edge joining it to C. However, in this case the two edges must be the same, since in both cases we choose the lowest-weight edge. Thus the algorithm never creates a cycle. Finally, the algorithm does not terminate until it has a single component, so the final output must be a spanning tree.

(b) [1 point] True or false: Borůvka's algorithm and Prim's algorithm always output the same minimum spanning tree.

Solution: True: A graph with distinct edge weights has a unique minimum spanning tree.

4. *Scheduling to maximize profit.* [10 points]

Suppose you are given n jobs, where job i has deadline d_i (a positive integer) and profit p_i (a positive real number) for all $i \in \{1, 2, \ldots, n\}$. Each job takes unit time, so it may be scheduled in any interval $[s_i, s_i + 1]$ for some nonnegative integer s_i provided $s_i + 1 \leq d_i$ (i.e., the job is finished by its deadline) and no other job is scheduled during the same interval. Your goal is to schedule a subset of the jobs to maximize the total profit, which is the sum of p_i over all scheduled jobs i. Design a greedy algorithm for this problem. Prove that your algorithm finds an optimal schedule, and analyze its running time.

Solution: *Algorithm:* Sort the jobs in order of decreasing profit and go through the jobs in this order, scheduling each job at the latest possible time (if any such time is available).

Running time: This algorithm can be implemented in time $O(n^2)$ since sorting takes time $O(n \log n)$, and then we run through all n jobs, performing $O(n)$ operations to check the possible times at which they might be scheduled. (In fact, you can get running time $O(n \log n)$ with a more careful implementation, but you are not asked to optimize the algorithm.)

Correctness: We prove optimality of the greedy schedule with an exchange argument. Let $(g_1, \ldots, g_\ell)$ be the jobs scheduled by the greedy algorithm in order of decreasing profit, where job g_i is scheduled at time t_i. Let $(b_1, \ldots, b_m)$ be the jobs in some optimal schedule, again in order of decreasing profit, where job b_i is scheduled at time u_i. Let j be the first index at which the two schedules differ, i.e., the smallest index so that either $g_j \neq b_j$ or ($g_j = b_j$ with $t_j \neq u_j$). We consider these two cases separately:

If $g_j = b_j$ with $t_j \neq u_j$, then $u_j < t_j$, since the greedy algorithm schedules jobs at the latest possible time. Now adjust the optimal schedule by changing the time of b_j from u_j to t_j, and if $u_k = t_j$ for some k, changing the time of b_k from u_k to u_j. The resulting schedule is still valid because b_j is moved to a time that is still before the deadline (namely, the time used in the greedy schedule) and b_k is moved to an earlier time. It includes the same jobs, so it has the same total profit and hence is still optimal.

If $g_j \neq b_j$, then the profit of g_j is at least that of b_j (otherwise the greedy algorithm would have added b_j next), and job g_j does not appear in the optimal schedule (since the jobs are sorted in decreasing order of profit, and we can assume that ties are broken in some consistent way). Then we add g_j to the optimal schedule at time t_j, deleting any job previously scheduled at time t_j if such a job exists in the optimal schedule. This schedule is still valid because t_j is before the deadline for g_j (by the definition of the greedy algorithm). It is still optimal because, if there was a job previously in the optimal schedule at time t_j, then it must have been one of the jobs $b_j, \ldots, b_m$, and these jobs all have profit no more than that of g_j.

In either case, we adjusted the schedule so that $g_j = b_j$ and $t_j = u_j$. Continuing in this way, we ensure that the first ℓ jobs of the optimal schedule are identical to those of the greedy schedule. If there were any remaining jobs in the optimal schedule (i.e., if $m > \ell$) then there would be at least one more job that the greedy algorithm could add to increase its profit. Since this is not the case, we must have $m = \ell$, and we have transformed the optimal schedule into the greedy one without decreasing its profit. Thus the greedy schedule is optimal.

5. *Maximum ordered ratio.* [10 points]

Suppose you are given as input a sequence of numbers $a_1, a_2, \ldots, a_n$ with $n \geq 2$. Your goal is to find the largest ratio between two of these numbers where the numerator occurs after the denominator in the sequence. In other words, you would like to compute

$$\max\left\{\frac{a_i}{a_j} : i, j \in \{1, 2, \ldots, n\} \text{ with } i > j\right\}.$$

Describe a divide-and-conquer algorithm to solve this problem. Prove its correctness and analyze its running time. For full credit, your algorithm should run in time at most $O(n \log n)$.

Solution: A natural divide-and-conquer algorithm proceeds as follows:

MOR$(a_1, \ldots, a_n)$:
  If $n = 2$, return a_2/a_1
  If $n = 3$, return $\max\{a_2/a_1, a_3/a_2, a_3/a_1\}$
  Let $\ell = \text{MOR}(a_1, \ldots, a_{\lfloor n/2 \rfloor})$
  Let $r = \text{MOR}(a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n)$
  Let $u = \min\{a_1, \ldots, a_{\lfloor n/2 \rfloor}\}$
  Let $v = \max\{a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n\}$
  Return $\max\{\ell, r, v/u\}$

(Note that either $n = 2$ or $n = 3$ may occur as a base case when we recursively split the list in half, although this is a relatively minor detail.)

Clearly the maximum ordered ratio of the input sequence is either the largest such ratio in the first half, the largest such ratio in the second half, or the ratio of the largest number in the second half to the smallest number in the first half. This is precisely what the algorithm computes, so it is correct.

To determine the running time, observe that the min and max can both be computed in linear time, so we have the recurrence $T(n) = 2T(n/2) + O(n)$ for the running time $T(n)$, and we know that this has solution $T(n) = O(n \log n)$.

In fact, it is possible to reduce the running time to $O(n)$ by a slightly more clever approach (say, using dynamic programming).