CMPSCI 311: Introduction to Algorithms
First Midterm Exam: Practice Exam

Name: _____ ID: _____

Instructions:

- Answer the questions directly on the exam pages.
- Show all your work for each question. Providing more detail including comments and explanations can help with assignment of partial credit.
- If the answer to a question is a number, *unless the problem says otherwise*, you may give your answer using arithmetic operations, such as addition, multiplication, "choose" notation and factorials (e.g., "$9 \times 35! + 2$" or "$0.5 \times 0.3/(0.2 \times 0.5 + 0.9 \times 0.1)$" is fine).
- If you need extra space, use the back of a page.
- No books, notes, calculators or other electronic devices are allowed. Any cheating will result in a grade of 0.
- If you have questions during the exam, raise your hand.

Question	Value	Points Earned
1	10	
2	10	
3	10	
4	10	
5	10	
Total	50	

Question 1. (*10 points*) Indicate whether each of the following statements is TRUE or FALSE. No justification required.

1.1 (*2 points*): $\sum_{i=1}^{n} 1/i^2 = \Theta(n^4)$.
Solution. False,

$$\sum_{i=1}^{n} \frac{1}{i^2} \leq \sum_{i=1}^{\infty} \frac{1}{i^2} = \pi^2/6 = \Theta(1)$$

1.2 (*2 points*): *A graph with n vertices and $n-1$ edges is either disconnected or a tree.*
Solution. True. All trees on n vertices has $n-1$ edges. The way to see this is to orient the edges away from any fixed vertex r and observe that every vertex has exactly one incoming edge.

1.3 (*2 points*): *For every n there exists a directed graph on n vertices with $\Omega(n^2)$ edges that has a topological ordering.*
Solution. True. Consider ordering the vertices $1, \ldots, n$ with edges (i, j) for all $i < j$. This graph clearly has a topological ordering and has $\sum_{i=1}^{n-1} i = \Theta(n^2)$ edges.

1.4 (*2 points*): *In a connected weighted graph, the edge with maximum weight is* never *in the minimum spanning tree.*
Solution. False. If the graph itself is a tree, then all edges must be in the MST.

1.5 (*2 points*): *The recurrence $T(n) = 4T(n/2) + O(n)$ solves to $T(n) = \Theta(n^3)$.*
Solution. False. It's $\Theta(n^2)$ according to the formula in class. For a recurrence of the form $T(n) = kT(n/2) + O(n)$, we saw that if $k > 2$ then $T(n) = O(n^{\log_2 k})$. Plugging in $k = 4$ gives $O(n^2)$, which is not $\Omega(n^3)$.

Question 2. (*10 points*)

2.1 (*5 points*): *Recall the scheduling problem where we have several task with lengths $t(i)$ and deadlines $d(i)$ and we want to order the tasks to minimize* lateness *where, if task i is completed at time $f(i)$, then lateness is defined as $L = \max_i \max(0, f(i) - d(i))$. Prove that ordering the intervals by their slack time, i.e., $d(i) - t(i)$ fails to find an optimal solution.*

Solution. (From K&T) Consider a two-job instance where the first job has $t_1 = 1$ and $d_1 = 2$, while the second job has $t_2 = 10$ and $d_2 = 10$. Sorting by increasing slack would place the second job first in the schedule, and the first job would incur a lateness of 9. (It finishes at time 11, nine units beyond its deadline.) On the other hand, if we schedule the first job first, then it finishe on time and the second job incurs a lateness of only 1.

2.2 (*5 points*): *On a stable matching instance, prove that if we run the Gale-Shapley algorithm twice, once with schools proposing and once with students proposing and we obtain the same matching, then the instance has a unique stable solution.*

Solution. The proof is based on the fact that when we run the algorithm with the schools proposing, we find a stable matching that is best for the schools and worst for the students. Specifically, if s is a student and c is a college, we say that (s, c) is a valid pairing if there is a stable matching where s is paired with c. Then define $\text{best}(c)$ as the highest ranked student (according to c's ranking) such that (s, c) is a valid pair. Similarly define $\text{worst}(s)$ as the lowest ranked college (according to s's ranking) such that (s, c) is a valid pair.

The textbook proves that when we run the algorithm with the schools proposing, we always find the matching with pairs $(c, \text{best}(c))$ for all colleges c. This matching turns out to be equivalent to the matching $(\text{worst}(s), s)$ for each student s. On the other hand, when we run the algorithm with the students proposing, we find the matching $(\text{best}(s), s)$ which is equivalent to $(c, \text{worst}(c))$.

If these two are the same, then the $\text{best}(c) = \text{worst}(c)$ for each college, so each college can only be paired off with a single student. Thus the matching is unique.

Refer to Facts 1.7 and 1.8 in the textbook for additional details

Question 3. (*10 points*) Alice is planning her course schedule for her time at UMass. There are n courses she must take and each course c_i can have pre-requisites P_i, which is a possibly empty set of courses. However, the department allows students to take a course and its prerequisites in the same semester. In other words, a course c_i can be taken in semester t if for all $c_j \in P_i$ the semester in which Alice takes c_j is at most t.

On the other hand, Alice can take at most 3 courses in a semester.

1. Prove that if the pre-requisite graph has a cycle of length 4, then there is no way for Alice to find a schedule satisfying all the pre-requisites.

   Solution. If there is a cycle in the prerequisite graph, then all courses in the cycle must be taken in the same semester. Suppose that S is such a cycle. If there aren't then there must be some subset of courses $S' \subset S$ that is taken in an earlier semester than the remaining courses $S \setminus S'$. But since S is a cycle, at least one course in S' has a prerequisite in $S \setminus S'$, which is a contradiction.

   Thus a cycle of length 4 cannot be scheduled.

2. Prove that if every course is involved in at most one cycle of length at most 3, then a valid schedule must exist.

   Solution. For any course c, consider the courses C involved in a cycle containing c. We will assign these all to the same semester, but we must assign all other pre-requisites in an earlier semester. This is always possible, since while pre-requisites can be in cycles of their own, they cannot be in cycles containing C. In particular, it cannot be the case that there is a cycle between some pre-requisite p, a course $c' \in C$, and a further course r that has some $c'' \in C$ as a pre-requisite. This implies that c' (and c'') are in multiple cycles.

   Thus there are no reverse edges between the pre-requisites of C and the courses that require C (e.g., edges from a course that depends on C to a course that C is dependent on), which means that we can schedule all of the pre-requisites before C, and then proceed to schedule everything else recursively.

Question 4. (*10 points*) Given two lists, L_1 of length n and L_2 of length m. We say that L_2 is a *subsequence* of L_1 if we can remove elements from L_1 to produce L_2. This means that there exists indices $i_1 < \ldots < i_m$ such that $L_1[i_j] = L_2[j]$ for each j. Design an algorithm that detects if L_2 is a subsequence of L_1 and outputs the indices $i_1, \ldots, i_m$ if L_2 is a subsequence of L_1.

Solution. The algorithm is a greedy algorithm that makes one pass over both lists. It starts with pointers at the first element of each list, repeatedly incrementing the pointer on L_1 until it finds an element i such that $L_1[i] = L_2[1]$. This value i is the output i_1, and at this point the algorithm increments the pointer on L_2 to point to the second element, while also incrementing the pointer on the first list, to point to the $i+1$st element. The process repeats until both lists have been traversed.

Here is the pseudocode.

j = 1, k=1
for $k = 1, \ldots, m$ **do**
  while $L_1[j] \neq L_2[k]$ **do**
    $j \leftarrow j + 1$
  end while
  $i_k = j$ (in the solution)
  $j \leftarrow j + 1, k \leftarrow k + 1$
end for

This algorithm stays ahead of any other possible solution, in the sense that among all possible subsequences, it outputs the one with lowest indices $i_1 < \ldots < i_m$. More formally, for any other partial subsequence I' (consisting of indices $i'_1 < \ldots < i'_m$, where some could be say larger than n if elements were not matched) if we define $\Phi(j; I')$ to be the number of elements of L_2 that have been matched in I using the first j indices of L_1, then we'll soon prove that our algorithm satisfies $\Phi(j; I) \geq \Phi(j; I')$ for all j where I is the subsequence produced by our algorithm. Assuming this claim is true momentarily, we see that $\Phi(n; I) \geq \Phi(n; I')$ for all other potential partial subsequences, which means that if L_2 is a subsequence of L_1, then I must be one of them.

The proof of the claim involving Φ is based on the fact that our algorithm always chooses the first match that it finds. Since when $j = 0$, all subsequences have zero matches, and since we always choose the first valid match, our algorithm always stays ahead.

The running time is clearly $O(n + m)$. We just make one pass through both lists.

Question 5. (*10 points*) Suppose we have a complete k-ary tree with n leaves (suppose $n = k^d$ for some integer d). Each leaf v is associated with a weight $w(v)$. The weight of an internal node is defined to be the sum of the weights of all leaves that are descendants of this node. So the weight of the root r is $w(r) = \sum_{\text{leaves } v} w(v)$. Design and analyze an algorithm to compute the weight of every internal node.

Solution. This problem can be solved by a divide-and-conquer. The idea is that at each node, we first recursively compute w on each direct child and then we simply add up the values at the children. This is correct because we are working with a tree, which implies that the leaves are disjoint. Therefore, we have,

$$w(v) = \sum_{u, u \text{ child of } v} w(u).$$

The running time of the algorithm is given by a recurrence of the form,

$$T(n) = kT(n/k) + O(k)$$

The most straightforward way to see that this recurrence is $O(n)$ is to observe that we do a constant amount of work for each edge in the tree. For each edge, we first make a recursive call (which will process other edges) and then pass one number back up this edge. In total this is $O(1)$ work per edge.

The number of edges is at most $2n - 1$. This can be found using a partial geometric series, but it's easier to just observe that the maximize the number of internal nodes, you want $k = 2$, which gives a balanced binary tree. A balanced binary tree with n leaves has $n - 1$ internal nodes.