CMPSCI 311: Introduction to Algorithms
Practice Final 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 1. (*10 points*) **True or False?** Indicate whether each of the following statements is TRUE or FALSE. No justification required.

1.1 (*2 points*): $\sum_{i=1}^{n} 2^i = \Theta(2^n)$.

Solution: ~~False~~ True. (Typo corrected Friday May 5 at 8:30am.)

1.2 (*2 points*): *A dynamic program that implements the following recursive form can be used to solve the subset sum problem, which asks to find a subset S of numbers from $x_1, \ldots, x_n$ (all non-negative) with maximum weight subject to not exceeding a given number W.*

$$OPT(i, w) = \max\{OPT(i-1, w), x_i + OPT(i-1, w - x_i)\}$$

Solution: True

1.3 (*2 points*): *For any flow network, and any two vertices s, t there is always a flow of at least 1 from source s to target t.*

Solution: False

1.4 (*2 points*): *The recurrence $T(n) = 2T(n-1) + O(1)$ solves to $\Theta(n^2)$.*

Solution: False

1.5 (*2 points*): *Suppose $X \leq_P Y$ and X is solvable in polynomial time. Then $P = NP$.*

Solution: False. Any problem X that is solvable in polynomial time reduces to *every* other problem Y.

Question 2. (*20 points*) **Short Answer.** Answer each of the following questions in at most two sentences.

2.1 (*4 points*): *In a weighted graph G where all edges have weight 1, how can we use Djikstra's algorithm to find a minimum spanning tree?*

Solution: All trees are minimum spanning trees when the edges have weight 1. So as you run Djikstra's just record the parent of each node, and this is an MST.

2.2 (*4 points*): *Solve the recurrence $T(n) = 3T(n/2) + O(n)$.*

Solution: $T(n) = \sum_{i=0}^{\log_2 n - 1} 3^i \frac{n}{2^i} = O(n \times (3/2)^{\log_2 n - 1}) = O(n^{\log_2 3})$.

2.3 (*4 points*): *Suppose a dynamic programming algorithm creates an $n \times m$ table and to compute each entry of the table it takes a minimum over at most m (previously computed) other entries. What would the running time of this algorithm be, assuming there is no other computations.*

Solution: We compute $n \times m$ numbers and each computation requires $O(m)$ time, so the total is $O(nm^2)$.

2.4 (*4 points*): *Suppose a connected graph G on n vertices has exactly two cycles. How many edges does it have?*

Solution: It has $n + 1$ edges — a tree plus two additional edges.

2.5 (*4 points*): *Suppose G is a graph with distinct edge weights and T is a minimum-spanning tree. Let $e \in T$. Describe how to find a cut $(S, V - S)$ such that e is the smallest weight edge crossing the cut.*

Solution: Remove e from T and let S and $V - S$ be the two connected components. If e' crosses this cut and has lower weight than e, we could create a cheaper spanning tree by adding e' instead of e.

Question 3. (*12 points*) Consider the longest increasing subsequence problem defined as follows. Given a list of numbers $a_1, \ldots, a_n$ an increasing subsequence is a list of indices $i_1, \ldots, i_k \in \{1, \ldots, n\}$ such that $i_1 < i_2 < \ldots, i_k$ and $a_{i_1} \leq a_{i_2} \leq \ldots, \leq a_{i_k}$. The longest increasing subsequence is the longest list of indices with this property.

3.1 (*1 points*): *What is the longest increasing subsequence of the list* $5, 3, 4, 8, 7, 10$*?*

Solution: There are two, $3, 4, 8, 10$ or $3, 4, 7, 10$.

3.2 (*2 points*): *Consider the greedy algorithm that chooses the first element of the list, and then repeatedly chooses the next element that is larger. Is this a correct algorithm? Either prove its correctness or provide a counter example.*

Solution: The above input is a counter example as the greedy algorithm would choose 5, but 5 is not in either of the optimal solutions.

3.3 (*2 points*): *Consider the greedy algorithm that chooses the smallest element of the list, and then repeatedly chooses the smallest element that comes after this chosen one. Is this a correct algorithm? Either prove its correctness or provide a counterexample.*

Solution: The input $2, 3, 4, 1$ is a counter example, since this greedy algorithm would choose 1 first, but 1 is not in the optimal solution.

3.4 (*2 points*): *Consider a divide and conquer strategy that splits the list into the first half and second half, recursively computes* $L = (\ell_1, \ldots, \ell_{k_L}), R = (r_1, \ldots, r_{k_R})$ *the longest increasing subsequences in each half, and then, if the last chosen element in the first half is less than the first chosen index in the second half (i.e.* $a_{\ell_{k_L}} \leq a_{r_1}$*) returns* $L \cdot R$*, otherwise it returns the longer of* L *and* R*. Is this a correct algorithm? either prove its correctness of provide a counterexample.*

Solution: Consider the input $2, 3, 4, 5, 0, 1$. The optimal is $2, 3, 4, 5$ but if we do divide and conquer we find $2, 3, 4$ and $0, 1$ so we output $2, 3, 4$.

3.5 (*5 points*): *Design a dynamic programming algorithm for longest increasing subsequence. Prove its correctness and analyze its running time.*

Solution: Let the input have size n, we build a size n table A by,

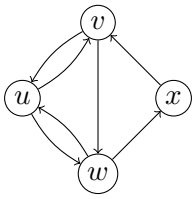
$$j^\star(i) = \text{argmax}\{A[j] \text{ such that } j < i \text{ and } a_j \leq a_i\}$$
$$A[i] = 1 + A[j^\star(i)]$$

$j^\star(i)$ is set to zero if no choice for j exists (for example on the first element of the list).

The proof is based on the fact that $A[j]$ holds the length of the longest increasing subsequence that ends on a_j. This is proved by induction, where the base case is easy. For the inductive step, consider some index i, we set $A[i] = 1 + A[j^\star(i)]$, and we are guaranteed that $j^\star(i)$ maximizes $A[j]$ among all possible positions j that could come before i in an increasing subsequence. Thus we are ensured that $A[i]$ holds the proper value.

The running time is $O(n^2)$.

Question 4. (*10 points*) In this problem we investigate the feedback vertex-set problem. Given a directed graph (which may contain cycles), a feedback vertex set S is a set of nodes such that removing the nodes S from G leaves a graph with no cycles. Another way to put this is that every cycle in G contains at least one node from S.



4.1 (*2 points*): *In the above graph, what is the size of the minimum feedback vertex set?*

Solution: 2. No single node is contained in every cycle, but there are multiple different sets of two nodes that break every cycle. For example, $\{u, x\}$ is a feedback vertex set.

4.2 (*2 points*): *True or False. A directed graph with a topological ordering always has a feedback vertex-set of size zero.*

Solution: True, a directed graph with a topological ordering is a DAG, so it has no cycles. Thus, the empty set is a feedback vertex-set.

4.3 (*6 points*): *Prove that the decision version of feedback vertex-set is NP-complete. That is given a directed graph and an integer k, decide whether the graph has a feedback vertex-set with cost at most k.* ***Hint:*** *reduce from vertex cover.*

Solution: (Proof by Ran Libeskind-Hadas). We claim that minimum feedback vertex-set (MFVS) is NP-complete. MFVS is in the class NP because a certificate would be the set of k vertices to remove. Wed remove these vertices from the graph G to build a new graph G. Wed then run depth-first search from each vertex to test for cycles. This clearly takes polynomial time.

Next, we show MFVS is NP-hard by a reduction from Vertex Cover (VC). An instance of VC is an undirected graph G and a positive integer k. In our reduction, we build a new graph G with the same vertices as G but with each edge u, v of G replaced by a pair of oppositely directed edges (u, v) and (v, u). Call this new directed graph G. Our instance of MFVS is G, k (we use the same k as in the instance of VC).

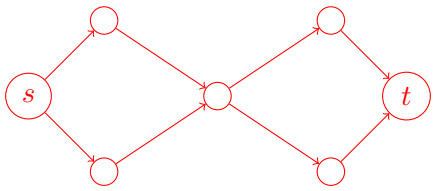
Now, we show that there is a vertex cover of size k in G if and only if there is a subset of k vertices in G whose removal breaks all cycles. First, assume that there is a vertex cover of size k in G. Now, remove these k vertices from G along with the edges incident upon them. Notice that for any directed edge $(u, v) \in G'$, at least one of u or v must have been removed, because one of u or v must have been in our vertex cover. Thus, after removing these k vertices and their incident edges from G, no two vertices have an edge between them, and consequently there can be no cycles.

Conversely, assume that there exists a set S of k vertices in G whose removal breaks all cycles. By construction every pair of vertices u, v that had an edge between them in G have a cycle (u, v), (v, u) in G. Since the removal of set S broke all cycles in G, for each edge $\{u, v\} \in G$, the set S must contain either u or v (or both). Thus, S is a vertex cover of G.

Question 5. (*10 points*) In this problem we investigate vertex-capacitated flow networks. We are given a directed graph $G = (V, E)$ with source s and sink t and a capacity c_v for each $v \in V$. We want an $s-t$ flow f that satisfies the usual conservation of flow constraints, but instead of satisfying edge-capacity constraints, satisfies the vertex capacity constraints $f(v) \leq c_v$. Here $f(v) = \sum_{(u,v)\in E} f_{u,v}$ is the total flow entering the node v. The goal is to design an algorithm for computing a maximum $s-t$ flow in a vertex-capacitated network.

5.1 (*3 points*): *Draw a directed graph G with clearly labeled source s and sink t, where if we consider the usual edge-capacitated version of the problem (with edge capacities $c_e = 1$) we get a maximum flow with a different value than if we consider the vertex capacitated version of the problem (with vertex capacities $c_v = 1$).*

Solution:



In this graph the maximum flow in the edge-capacitated version has value 2, while the vertex capacitated version has value 1.

5.2 (*7 points*): *Design a polynomial time algorithm for computing the maximum flow in a node-capacitated network. Prove that the algorithm is correct and analyze its running time.*

Solution: In the flow network, replace each vertex v with two vertices $v_{(1)}$ and $v^{(2)}$ with an edge $e_v = (v^{(1)} \to v^{(2)})$ of capacity $c(e_v) = c_v$. All incoming edges to v in the original network point to $v^{(1)}$ and have capacity ∞ (no constraint). All outgoing edges from v in the original network now start from $v^{(2)}$ and have capacity ∞.

The edge capacitated maximum flow in this network is precisely the vertex capacitated maximum flow in the original network. The only capacity constraints are on the e_v edges, and these precisely encode the fact that we must satisfy the vertex capacities. The running time is the same as the max flow problem $O((|V| + |E|)F)$ where F is the value of the maximum flow and $|E|$ is the number of edges in the original network and $|V|$ is the number of vertices.