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	Value	Points Earned
1	10	
2	20	
3	30	
4	10	
5	20	
6	10	
Total	100	

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

1.1 (*2 points*): 3-COLORING *can be solved by breadth first search and therefore is in* P.
False

1.2 (*2 points*): $\sum_{i=1}^{n} 2^i = \Theta(2^n)$.
True

1.3 (*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.

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

True

1.4 (*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.
False

1.5 (*2 points*): *The recurrence* $T(n) = 2T(n-1) + O(1)$ *solves to* $\Theta(n^2)$.
False

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

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

2.4 (*4 points*): *Why is* MaxFlow *in* $\mathcal{NP} \bigcap$ *co-*$\mathcal{NP}$*?*
We can verify both positive and negative instances via the Max-Flow Min-Cut theorem.

2.5 (*4 points*): *Suppose* $\mathcal{A}$ *is a randomized algorithm that finds the optimal solution to some minimization problem with probability at least* $p \in (0, 1)$*. More precisely, if we run* $\mathcal{A}$ *o some input, it returns a candidate solution* O *along with the cost of* O*, and with probability at least* p*, we are guaranteed that* O *minimizes the cost function. For another parameter* $\delta > 0$*, how can we use* $\mathcal{A}$ *to find an optimal solution with probability at least* $1 - \delta$ *and what is the running time of this new algorithm?*
If we run the algorithm m times, then the probability that all m runs fail to find the optimum is at most $(1-p)^m = [(1-p)^{1/p}]^{pm} \leq= [(1 - \frac{1}{1/p})^{1/p}]^{pm} \leq e^{-pm}$. We want this to be at most δ, which requires we set $m = p^{-1} \log(1/\delta)$ so our running time is $O(p^{-1} \log(1/\delta) \times$ runtime for $\mathcal{A})$.

Question 3. (*20 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 (*2 points*): *What is the longest increasing subsequence of the list* $5, 3, 4, 8, 7, 10$*?*
There are two, $3, 4, 8, 10$ or $3, 4, 7, 10$.

3.2 (*4 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.*
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 (*4 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.*
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 (*5 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.*
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 (*15 points*): *Design a dynamic programming algorithm for longest increasing subsequence. Prove its correctness and analyze its running time.*
Let the input have size n, we build a size n table A by,

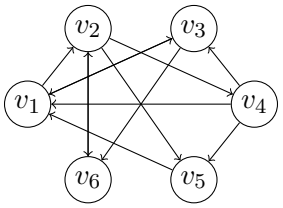
$$j^\star(i) = \text{argmax}\{A[j] \text{ such that } j < i \text{ and } a_j \leq a_i\}$$
$$A[i] = 1 + A[j^\star(i)]$$

$j^\star(i)$ 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 arc-set problem which generalizes the topological ordering. Given a directed graph (which may contain cycles), the goal in feedback arc-set is to find an ordering of the vertices that minimizes the number of back edges. More precisely, if $G = (V, E)$ is a directed graph, and $(a_1, \ldots, a_n) \in V$ with $a_i \neq a_j$ is an ordering of the vertices, we define the cost as

$$\text{cost}(a_1, \ldots, a_n) = \sum_{i<j} \mathbf{1}[(a_j, a_i) \in E]$$

Here $\mathbf{1}[\cdot]$ is a function that is 1 if the argument is true and zero otherwise. This is the number of edges going from right to left (backward) if we ordered the vertices with a_1 on the left and a_n on the right.



4.1 (*1 points*): *In the above graph, what is the cost of* $(v_1, v_2, v_3, v_4, v_5, v_6)$*?*
$3 + 1 + 1 + 0 + 0 + 0 = 5$

4.2 (*1 points*): *In the above graph, what is the cost of* $(v_6, v_5, v_4, v_3, v_2, v_1)$*?*
$2 + 2 + 1 + 1 + 1 + 0 = 7$

4.3 (*2 points*): *True or False. a directed acyclic graph always has an ordering* O *with cost*$(O) = 0$*.*
True, any topological ordering has no feedback arcs.

4.4 (*6 points*): *Prove that the decision version of feedback arc set is NP-complete. That is given a directed graph and an integer* k*, decide whether the graph has an ordering with at most* k *back-edges.*

The proof is by reduction from vertex cover. Given an instance $G = (V, E)$ of vertex cover, we will define a instance of feedback arc set. Let $G' = (V', E')$ be a new directed graph where for each $v \in V$ we have two vertices $v^{(1)}, v^{(2)} \in V'$ connected with the directed edge $(v^{(1)} \to v^{(2)})$. For every undirected edge $(u, v) \in E$, define two directed edges $(u^{(2)} \to v^{(1)})$ and $(v^{(2)} \to u^{(1)})$. If G has n vertices and m edges, this new graph has $2n$ vertices and $n + 2m$ edges. We claim that G has a vertex cover of size at most k if and only if G' has an ordering with at most k back edges.

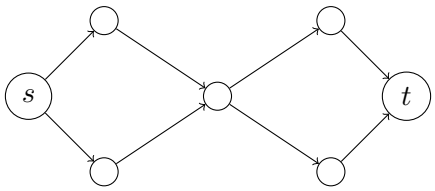
The idea for the proof is that each edge $(u, v) \in E$ forms a 4-cycle in G'. If S is a vertex cover, we can remove the edges $(v^{(1)} \to v^{(2)})$ for each $v \in S$ and break all of the 4-cycles, since we remove an edge from each cycle.

More precisely, if S is a vertex cover, we construct an ordering as follows: the first $|S|$ nodes are $v^{(2)}$ for each $v \in S$, followed by each $u^{(1)}, u^{(2)}$ pair for $u \notin S$, and finally followed by $v^{(1)}$ for each $v \in S$. The order in each group does not matter. Since S is a vertex cover, all edges from into $u^{(1)}$ for $u \notin S$ originate from some $v^{(2)}$ for $v \in S$ and similarly all edges from $u^{(2)}$ point to $v^{(1)}$ for some $v \in S$. Thus the only back edges in this ordering are from $v^{(1)} \to v^{(2)}$ for $v \in S$, which is k back edges.

The other direction is by a counting argument. If no subset of k vertices covers all edges, then there is no way to break all m cycles in G'. Observe that if you break and edge of the form $(v^{(2)} \to u^{(1)})$ you can break at most one cycle, so it is always better to break edges of the form $(v^{(1)} \to v^{(2)})$. However, since there is no vertex cover of size at most k, choosing k of these edges cannot break all the cycles.

Question 5. (*20 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 (*5 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 $c_v = 1$).*



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

5.2 (*15 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.*

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.

Additional space.

Question 6. (*10 points*) Consider a variant of the subset sum problem where we are given a set of numbers $x_1, \ldots, x_n$ and need to partition them numbers into sets $S_1, \ldots, S_K$ such that for each $k \in \{1, \ldots, K\}$, $\sum_{i \in S_k} x_i \leq W$ for some target W. The goal is to minimize K, the number of sets in the partition. We will study a simple approximation algorithm for this problem. The algorithm considers the items in order, and forms the first set S_1 by repeatedly adding the numbers $x_1, x_2, \ldots$ until the next number would exceed the target W. Then it proceed to construct the next set.

6.1 (*2 points*): *Give an example input where this algorithm does not use the minimum number of sets.*

Let $W = 5$ and consider the sequence $4, 2, 3, 1$. The optimal solution uses two sets $\{4, 1\}$ and $\{2, 3\}$ but our approximation algorithm tries to group 2 with 4 and fails, so it forms $\{4\}, \{2, 3\}, \{1\}$.

6.2 (*2 points*): *Derive a lower bound on $K^\star$ the smallest possible number of sets in the partition in terms of the target W and the total weight $X = \sum_{i=1}^{n} x_i$.*

$K^\star \geq X/W$ since all the weight must be included and at most W can be included in each set.

6.3 (*6 points*): *Use this lower bound to prove that this greedy algorithm always produces a number of sets K that is at most $2K^\star$.*

Consider sets S_1, S_2. We claim $\sum_{i \in S_1 \cup S_2} x_i \geq W$. This is clear since S_2 was only formed because adding the next element to S_1 would exceed the weight constraint, and this item was added to S_2. Similarly, for every adjacent pair of sets $s, s+1$, we have $\sum_{i \in S_s \cup S_{s+1}} x_i \geq W$ by the same argument.

Thus, letting K be the number of sets produced by the algorithm,

$$K^\star \geq X/W = \frac{1}{W} \sum_{s=1}^{K} \sum_{i \in S_s} x_i \geq \frac{1}{W} \sum_{s=1}^{K/2} \sum_{i \in S_{2s-1} \cup S_{2s}} x_i \geq \frac{1}{W} \left(K/2 \times W\right) = K/2$$

This proves that $K \leq 2K^\star$.