Final

- **The exam has 6 questions, is worth 100 points, and will last 180 minutes.**
- Read the instructions and the questions carefully first.
- Begin each problem on a new page.
- Be precise and concise.
- The questions start with true/false and short answer, and end with long answer. The problems may **not** necessarily follow the order of increasing difficulty.
- Good luck!

1 True/False + Justification (3pts each)

For the following parts, state clearly whether the given statement is true or false and give a brief justification for your answer.

(a) In a zero-sum game, it's possible for Player 2 to give Player 1 an advantage by announcing her strategy prior to the game.

True. For example, in rock, paper, scissors, P2 announcing she'll always play rock gives P1 an advantage.

(b) In a zero-sum game, it's possible for Player 2 to give Player 1 an advantage by announcing **an optimal Player 2 strategy** prior to the game.

False. By LP duality of zero-sum games.

(c) Suppose the SCC algorithm on some graph returns the SCCs in the order: {G, H, I, J, K, L}, {D}, {C, F}, {B, E}, {A}. Then, it is guaranteed that in the first DFS (the one done on G^R), Postorder(H) > Postorder(D).

False. Not necessarily. For example, imagine a graph where the first SCC contains edges between G and H, and have H connect to nothing else. In the DFS, let G go to H, which then finishes, certainly before D.

(d) There exists an algorithm using $O(\log n)$ space to compute the mean of a stream of n integers, each in $[0, n]$.

True. Note that the sum of all the integers is at most n^2. Store the sum of all the integers in $\log(n^2) = 2\log n = O(\log n)$ space, and divide by n at the end of the stream.

(e) If you find a polynomial time algorithm for an NP problem, then you've proved P = NP.

False. We can solve addition (an NP problem) in polynomial time, but this does not imply P = NP.

(f) If you find a polynomial time algorithm for an NP-complete problem, then you've proved P = NP.

True. Then every problem in NP would have a polynomial time algorithm, since every NP prolem reduces to the NP-complete problem in polynomial time.

(g) Consider $h_1(x) = x$ and $h_2(x) = 1 + (x \pmod 4)$, both from $\{1, 2, \ldots, 24\}$ to itself. Then $\mathcal{H} = \{h_1, h_2\}$ is universal.

False. Collisions are frequent for multiples of 4, i.e. $\Pr_{h\in\mathcal{H}}(h(4) = h(8)) = \frac{1}{2} > \frac{1}{24}$.

(h) There is guaranteed to exist an $O(n^3)$ solution to the Vertex Cover problem if P = NP.

False. P = NP guarantees there is a polynomial time algorithm for Vertex Cover, but the leading exponent in the polynomial runtime may be greater than 3.

(i) Suppose that a Monte-Carlo algorithm A exists for a problem which always runs in time $T(n)$ returns a correct answer with probability $\frac{1}{2}$. Then there exists a Las-Vegas algorithm that solves the problem in expected time $2T(n)$. (Assume a correct answer can be verified in constant time.)

True. Run A and check if the output is correct – if not, repeat until the output is correct. The finishing time of this Las-Vegas algorithm is a geometric random variable which has expectation $2T(n)$.

2 Short Answer (5pts each)

(a) The SCC algorithm returns the SCC's one by one in a reverse topological order (sink to source). How would you make a small modification to the algorithm to return the SCC's in topological order (source to sink) instead?

Run $DFS(G)$ followed by $DFS(G^R)$ rather than $DFS(G^R)$ followed by $DFS(G)$.

(b) Consider two algorithms for the same problem:

- Algorithm A, which runs in $O(n)$ and produces a correct answer with probability 0.7, and a wrong answer with probability 0.3.
- Algorithm B, which runs in $O(n \log n)$ and produces a correct answer with probability 0.99, and a wrong answer with probability 0.01.

Which of the two algorithms should you use to build a more asymptotically (in n) efficient algorithm with probability 0.99 of producing a correct answer? Justify. Assume that you can always check if an answer is correct in constant time.

Algorithm A. For example, running algorithm A four times (and outputting a correct answer if one exists among the four runs) will yield a correct answer with probability $1 - (0.3)^4 = 1 - 0.0081 = 0.9919 > 0.99$ in $O(n)$ time.

(c) Argue (very briefly) that any algorithm to compute the minimum spanning tree of a graph $G = (V, E)$ must use at least $\Omega(|E|)$ time.

The algorithm must at least read the weight of every edge, since a priori any edge could be in the MST. This takes $\Theta(|E|)$ time.

(d) Show that it is possible to achieve negative regret in the expert's problem if the losses are chosen ahead of time. Give the losses and your choices of experts.

Consider the following 2-day, 2-expert instance:

- Expert A has a loss of 0 on Day 1 and a loss of 1 on Day 2.
- Expert B has a loss of 1 on Day 1 and a loss of 0 on Day 2.

By choosing A on Day 1 and B on Day 2, we achieve a total loss of 0. Moreover, the total loss of the best expert is 1 (achieved by both A and B). Hence the regret of our choices is $(0 + 0) - 1 = -1$.

(e) Show that if there is no unique minimum weight edge in a graph G, but exactly two minimum-weight edges e_1 and e_2, then both e_1 and e_2 must be included in any MST of G.

Let T be an MST of G and suppose WLOG e_1 is not in T. Note that there exists a cut which e_1 crosses and e_2 does not cross. Then since e_1 is strictly lighter than all edges except e_2, e_1 is lighter than the edge in T across this cut, which is a contradiction. Hence e_1 and e_2 are both in T.

Common misconception: Some students argued that Kruskal's algorithm would always pick e_1 and e_2 when constructing the MST. However, this argument only shows that e_1 and e_2 are in *some* MST of G, while we want to know whether e_1 and e_2 are in *any* MST of G. It would be necessary to also show that Kruskal's algorithm can generate any MST of G (even though this is in fact true).

3 Party Planning Committee (12pts)

Dunder Mifflin is having a corporate party, and you (head of the Party Planning Committee) want to make the party as cool as possible! Here's what you know:

- Every employee in Dunder Mifflin (except the CEO) has one and only one direct boss – that is, the corporate hierarchy can be represented as a rooted tree.
- Due to the awkward nature of the corporate culture, an employee will not show up to the party if their direct boss (i.e. parent node) will also be present at the party.
- Each employee e has a coolness $C(e)$. The coolness of a party is the total sum of the coolnesses of all its participants.

Your task is to design an efficient dynamic programming algorithm which, given a corporate hierarchy tree, outputs the maximum possible coolness of the party.

(a) Describe your subproblems, the corresponding recurrence relation / base cases, and in what order to compute them. Also state how to compute the maximum possible coolness of the party from your subproblems.

(b) Give a runtime analysis for your algorithm in terms of N (the number of Dunder Mifflin employees).

(a) For each employee, we compute $f(e)$, the maximum possible coolness of the party including only e and their descendants. We first compute $f(e) = C(e)$ for "leaf" employees, and then we recursively compute

$$f(e) = \max \left\{ C(e) + \sum_{g \in \text{Grandchildren}(e)} f(g), \sum_{c \in \text{Children}(e)} f(c) \right\}$$

in order of decreasing depth for "non-leaf" employees. The maximum possible coolness of the party is $f(\text{CEO})$.

(b) Each employee e is involved in a constant amount of work: computing $f(e)$, computing $f(\text{Parent}(e))$, and computing $f(\text{Grandparent}(e))$. Hence the overall runtime of the algorithm is $O(N)$.

4 Rudrata / Hamiltonian Redux (12pts)

Recall from lecture that the Rudrata / Hamiltonian Cycle (RHC) problem is: given an undirected graph G, does G have a cycle (i.e. a closed loop) which visits every vertex exactly once? Consider now the Rudrata / Hamiltonian Path (RHP) problem: given an undirected graph G, does G have a path (not closed) which visits every vertex exactly once?

(a) Give a reduction from RHP to RHC which has runtime polynomial in $|V|$ and $|E|$. Justify correctness.

Hint: Given an RHP instance G, consider adding a dummy vertex d.

(b) Give a reduction from RHC to RHP which has runtime polynomial in $|V|$ and $|E|$. Justify correctness.

Hint: Given an RHC instance G, consider duplicating a vertex in G and also adding dummy vertices s, t.

(c) Show that RHP is NP-Complete. Clearly identify whether you used the reduction in (a) or the reduction in (b) for your proof – only one of these reductions is relevant for this part.

Note: You may use the fact that RHC is NP-Hard without proof.

(a) Given an RHP instance G, create a new graph G' by adding a dummy vertex d along with edges from d to each vertex in G. Run RHC on G'.

Justification: Given a Rudrata / Hamiltonian path from s to t in G, there is a Rudrata / Hamiltonian cycle in G' by including the edges (d, s) and (d, t) with the edges of the Rudrata / Hamiltonian path in G. Conversely, given a Rudrata / Hamiltonian cycle in G', there is a Rudrata / Hamiltonian path in G by removing the edges adjacent to d.

(b) Given an RHC instance G, create a new graph G' as follows: Pick any vertex v in G and add a new vertex v' which has edges to all the neighbors of v (but not v itself). Moreover, include dummy vertices s and t with edges (s, v) and (t, v'). Then run RHP on G'.

Justification: Given a Rudrata / Hamilitonian cycle in G, there is a Rudrata / Hamiltonian path in G' starting from s, going to v, back around to v' via the edges of the Rudrata / Hamiltonian cycle in G, and ending at t. Conversely, given a Rudrata / Hamiltonian path in G', its endpoints must be s and t (since they each have only one adjacent edge) – removing s and t and merging v with v' yields a Rudrata / Hamiltonian cycle in G.

(c) To show that RHP is NP-Complete, it suffices to show that RHP is NP and NP-Hard. A Rudrata / Hamiltonian path can be verified in polynomial time, since it takes linear time to verify the condition that each vertex is visited exactly once. Hence RHP is in NP. Moreover, RHP is NP-Hard since RHC (which is NP-Hard) reduces to it via the reduction in (b).

5 GME to the... Forest Hideouts? (12pts)

Robin Hood and his merry band are on the run from the Sheriff! Let $G = (V, E)$ be an undirected, unweighted graph where vertices represent towns and edges represent roads between towns. Robin's band has n members (including Robin), and each member starts in a different town s_i. The objective for Robin's band is to find paths for each member which end at one of the hideouts, where the set of hideouts is given by $H \subseteq V$.

However, there's a catch: No two members of Robin's band can visit the same town, since two band members passing through the same town (even at different times), would draw too much attention from the Sheriff (e.g. if Robin's path includes town t, then Little John's path cannot include town t).

Note: The above restriction applies to hideouts as well, i.e. two band members cannot end at the same hideout.

(a) Given G, H and $s_1, \ldots, s_n$, reduce the problem of whether non-intersecting paths to hideouts exist for Robin's band to the max-flow problem from class.

(b) Prove the correctness of your reduction.

Note: You may use without proof the fact that an integer max-flow exists if all capacities are integers.

(c) Give a runtime analysis of the pre-processing step in your reduction, in terms of $|V|$ and $|E|$.

(a) We are being asked for an algorithm to determine whether n vertex-disjoint paths exist from a "source set" of n vertices $S = s_1, \ldots, s_n$ to a "target set" $H = h_1, \ldots, h_k$.

In order to solve this, we will first reduce to the problem of finding n edge-disjoint paths from S to H. This will then reduce more easily to max-flow. Construct a new graph G', which has two nodes v_1 and v_2, with a directed edge (v_1, v_2) for every vertex v in G. For every undirected edge (u, v) in G, we will create directed edges (u_2, v_1) and (v_2, u_1) in G'.

We will solve the edge-disjoint paths problem from $S_1 = (s_1)_1, \ldots, (s_n)_1$ to $H_2 = (h_1)_2, \ldots, (h_n)_2$ in G'. In order to do this, we will create a new graph G'' which is identical to G' but with a single source vertex s^* and a single target vertex t^*. G'' also adds edges from s^* to every vertex in S_1, and from every vertex in H_2 to t^*. Running a max-flow algorithm from s^* to t^* in G'' (with all edge capacities set to 1) will give us our answer: if the max-flow is equal to n, there is a feasible routing; if it is less than n, there is none.

(b) Firstly we will note that sets of edge-disjoint paths from s^* to t^* in G'' are in one-to-one correspondence with sets of vertex-disjoint paths from S to H in G. The correspondence is straightforward, and can be performed simply by adding or removing the subscripts on vertices in G'' and adding or removing the connections to s^* and t^* as necessary. This correspondence associates edge-disjoint paths in G'' with all and only the vertex-disjoint paths in G, because every s^*-t^* path in G'' must use (v_1, v_2) edge for all nodes v in G that the corresponding path visits (this edge is the only edge into v_2, and the only edge out of v_1). Therefore two paths in G both use a vertex v if and only if their corresponding paths in G'' both use the edge (v_1, v_2).

Suppose there are n edge-disjoint s^*-t^* paths in G''. We can route one unit of flow along each of these to achieve a total flow of n. Conversely, suppose that there is a feasible flow of n units in G'', and consider an integer flow which achieves this. Since every edge has capacity 1, all edges which are used for this flow have exactly 1 unit of flow routed along them. Since every vertex has exactly the same amount of flow into it as it has out of it, the edges used by this flow must form n edge-disjoint s^*-t^* paths. Note that a flow of size more than n is impossible, since a cut with s^* on one side and the rest of the graph on the other has size n.

Combining the above, we see that G'' has a value-n s^*-t^* flow if and only if G has n vertex-disjoint paths from S to H.

(c) The preprocessing for this algorithm involves only creating a new graph which is a constant factor larger than G, and doing a constant amount of work for each edge and vertex in G to build this graph; so the runtime is $\Theta(|V| + |E|)$.

6 K-Paths (12pts)

Suppose T is a rooted tree (not necessarily binary) and k is an integer. Your first task is to devise an algorithm to compute **the largest possible set of non-intersecting directed k-paths**.

- A directed k-path is a path in the tree of exactly k edges that goes strictly upwards (i.e. always from child to parent) in the tree, i.e. no path contains two children of the same node. Directed k-paths do not have to start at leaves.
- By *non-intersecting*, we mean that distinct k-paths do not share any vertices.

(a) Describe a greedy algorithm which, given T and k as input, computes the largest possible set of non-intersecting directed k-paths. Show that your algorithm is correct and analyze its runtime.

(b) Now, suppose that each vertex in T has a reward associated to it and you want to find the set of non-intersecting directed k-paths which maximizes the sum of the vertex rewards it covers. Draw or describe a counterexample to show that your greedy algorithm does not work for this problem.

(c) Now suppose each reward is independently drawn uniformly at random from $\{0, 1, 2, \ldots, 10\}$. Show that for any given rooted tree, your greedy algorithm achieves an average approximation factor of at least $\frac{1}{2}$ for the problem in (b). That is, show that

$$\mathbb{E}_r\left[\frac{G(r)}{M(r)}\right] \geq \frac{1}{2},$$

where $G(r)$ and $M(r)$ are respectively the greedy algorithm's total reward and maximum total reward for r, a random assignment of rewards for the given tree.

Hint: Let N be the maximum number of non-intersecting k-paths for the given tree. Start by showing that

$$\mathbb{E}_r\left[\frac{G(r)}{M(r)}\right] \geq \frac{1}{10N(k+1)} \cdot \mathbb{E}_r[G(r)].$$

(a) We will first use BFS to compute the depth of every node in the tree, and order the nodes in decreasing order of depth. Then:

1. Select the k-path which which starts at the lowest node and continues through k edges upwards
2. Remove this path and all the descendants of its nodes from the tree
3. Repeat steps 1 and 2 until step 1 does not find a k-path

For the proof of correctness, we will note that the algorithm always finds at least one k-path when possible, and then show that the first path chosen by the algorithm is always present in some optimal choice of paths. This is sufficient because the algorithm removes a subtree after every step – meaning that every path chosen by the algorithm is its first choice in some tree.

Suppose towards a contradiction that there is some tree for which no optimal solution includes the path a chosen by the algorithm in its first step. Consider one of these optimal solutions. Since the subtree rooted at the highest node in a has depth $k+1$, at most one path in the optimal solution 'conflicts' with a (i.e. contains nodes in a). So, the path intersecting a from the optimal solution (if there is any) can be removed, and replaced with a. This will produce a solution of the same size (therefore optimal) which uses the first path picked by the algorithm; this is a contradiction.

In the argument above, we used the fact that a subtree of depth $k+1$ contains at most one non-intersecting k-path. This is true because any k-paths starting in a subtree of depth at most $k+1$ always must contain the root of the subtree, so there can never be more than one non-intersecting k-path using the nodes of such a subtree.

This algorithm takes $O(|V| + |E|)$ time to perform BFS, and then considers every vertex at most twice (once in the list of all nodes ordered by depth, and once when deleting it), taking $O(|V|)$ time. Since our graph is a tree, $|E| = |V| - 1$, and so our overall runtime is $O(|V|)$.

(b) Consider a path graph of 3 nodes and 2 edges, where the first node has weight 10 and the second and third nodes have weight 1. If the first node is the root of the tree, then in the $k = 1$ case the greedy algorithm will choose only the lower (second) edge for a total node weight of 2, whereas a better option would have been to choose only the upper (first) edge for a total node weight of 11.

(c) Take N as given in the hint. The maximum total reward is achieved by at most $N(k+1)$ vertices (since by correctness of the greedy algorithm, any selection of non-intersecting directed k-paths has at most $N(k+1)$ vertices), so combined with the fact that each vertex achieves reward at most 10, we have $M(r) \leq 10N(k+1)$. Then

$$\mathbb{E}_r\left[\frac{G(r)}{M(r)}\right] \geq \mathbb{E}_r\left[\frac{G(r)}{10N(k+1)}\right] = \frac{1}{10N(k+1)} \cdot \mathbb{E}_r[G(r)] = \frac{5N(k+1)}{10N(k+1)} = \frac{1}{2},$$

where the second-to-last equality is due to linearity of expectation, combined with the facts that exactly $N(k+1)$ vertices are covered by the greedy selection of k-paths and each vertex has expected reward 5.