There are 5 questions.

---

1. State True or False and give a one or two sentence explanation.
   (a) DFS on a dense graph has runtime $O(|V|^2)$.

   (b) If a connected undirected graph has two edges that both have the minimum weight then the graph has 2 distinct MSTs.

   (c) If a search problem is in NP then there is no known polynomial time algorithm to solve it.

   (d) If you perform explore on a DAG starting at a source vertex, then all vertices will be marked as visited.

2. (**Graph path algorithms**) Give an $O(|E| + |V|)$ algorithm, possibly using known algorithms from class as sub-routines, to tell, given a directed graph G and two vertices s and t, whether there is a not-necessarily simple path from s to t whose length is a multiple of 3. You can use without proof the correctness and time analysis of algorithms covered in class, but need to relate the above problem to the algorithms in question.

3. (**Divide and Conquer**) The maximum weight sub-tree problem is as follows. You are given a balanced binary tree T of size n, where each node $i \in T$ has a (not necessarily positive) weight $w(i)$ for each node $i \in T$. (Every node in T has pointers to its left-child, right-child, and parent, and you are given a pointer to the root of the tree. A NIL field for the children means the node is a leaf, and for the parent, means the node is the root. You are given a pointer to the root r of T.) A rooted sub-tree of T is a connected sub-graph of T containing the root r. (So a sub-tree is not necessarily the entire sub-tree rooted at a node. However, it cannot contain the children of a node without containing the node.) You wish to find the maximum possible value of the sum of weights of nodes in a rooted sub-tree S of T, $\sum_{i \in S} w(i)$.

   Here is a recursive algorithm that solves this problem, given a pointer to the root of T:

```
MaxWtSubtree($r$)
    - IF ($r = NIL$) return 0
    - $A \leftarrow \max(0, \texttt{MaxWtSubtree}(\texttt{r.leftchild}))$
    - $B \leftarrow \max(0, \texttt{MaxWtSubtree}(\texttt{r.rightchild}))$
    - Return $(w[r] + A + B)$
```

   (a) Give a recurrence and a time analysis for this algorithm in the case when T is a complete binary tree of height h and size $n = 2^h - 1$.

   (b) Prove that the same worst-case bound holds if T is *any* tree of size n.

(**Monotone Matchings**) All the remaining questions concern variations of the following problem.

Let G be a bipartite graph, with $L = \{u_1, ..u_l\}$ the set of nodes on the left, $R = \{v_1, ..v_r\}$ the set of nodes on the right, E the set of edges, each with one endpoint in L and the other in R, and $m = |E|$ the number of edges.

A *matching* in G is a set of edges $M \subseteq E$ so that no two edges in M share an endpoint (neither the one in L nor the one in R). A matching M is *monotone* if for every two edges (u_{i_1}, v_{j_1}) and (u_{i_2}, v_{j_2}) in M, if $i_1 < i_2$ then $j_1 < j_2$. That is, one could draw all the edges in the matching without crossing, if the nodes are put in order on the two sides.

The problem is, given a bipartite graph G, find the largest monotone matching in G.

Assume $l \leq r$. Then a monotone matching M is *perfect* if it has size l, i.e., $|M| = l$.

4. (**Greedy Algorithms and data structures**) Below is a greedy strategy for the largest monotone matching problem.

   Candidate Strategy A: For each $i = 1$ to l, if u_i has at least one undeleted neighbor v_j, match it to the unmatched neighbor with smallest value of j. Then delete u_i and $v_1, ...v_j$, and repeat.

   (a) Give a counter-example where the above greedy strategy fails to produce an optimal solution. (*Hint: Since below you will show that the algorithm works when the maximum monotone matching is perfect, your example shouldn't have a perfect monotone matching.*)

   (b) Illustrate the above strategy on the following graph with a perfect matching: $L = \{u_1, u_2, u_3\}$,$R = \{v_1, v_2, v_3, v_4, v_5\}$, and $E = \{(u_1, v_2), (u_1, v_3), (u_1, v_4), (u_2, v_1), (u_2, v_2), (u_2, v_3), (u_2, v_5)(u_3, v_1), (u_3, v_5)\}$.

   (c) Prove that, if G has a perfect matching, then Candidate Strategy A finds one.

   Hint: Let strategy A match u_i with v_{j_i} (unless it can't be matched). Let OPT be a perfect matching that matches each u_i with v_{k_i}. (Note: all left nodes will be matched by OPT, since it is perfect.) Use one of the following two methods.

   - *Exchange method: Prove by induction on T that there is a left-perfect matching OPT_T that matches each u_i with v_{j_i} for $1 \leq i \leq T$.*
   - *Greedy-stays ahead: Prove by induction on T that v_{j_T} exists and that $j_T \leq k_T$.*

   (d) Describe an efficient algorithm that carries out the strategy. Your description should specify which data structures you use, and any pre-processing steps. Assume the graph is given in adjacency list format. Give a time analysis, in terms of l, r and m.

5. (**Back-tracking and Dynamic Programming**) The following recursive algorithm for the maximal monotone matching problem finds the maximum matching whether or not it is perfect. It branches on whether a node on the left is matched or unmatched. By the analysis of greedy algorithm A in the previous problem, we can see that when a node is matched, it should always be matched to its smallest neighbor. The backtracking algorithm just returns the size of the maximum monotone matching, not the actual matching.

```
BTMMM($G = (L = \{u_1, ..u_l\}, R = \{v_1, ..v_r\}, E)$)) // $G$ is bipartite graph
    - IF $|L| = 0$ return 0.
    - $Unmatched \leftarrow$ BTMMM($G - \{u_1\}$).
    - IF $|N(u_1)| = 0$ return $Unmatched$
    - Let $J$ be the first neighbor of $u_1$, i.e., the smallest value so that $v_J \in N(u_1)$
    - $Matched \leftarrow 1+$ BTMMM($G - \{u_1, v_1, ..v_J\}$)
    - Return $Max(Matched, Unmatched)$
```

(a) Illustrate the above algorithm on your counter-example graph for the greedy strategy, (as a tree of recursive calls and answers.)

(b) Give an upper bound on the number of recursive calls the above algorithm makes, in the worst-case. (Be sure to expain your answer.)

(c) Give a dynamic programming version of the recurrence.

(d) Give a time analysis of this dynamic programming algorithm.

(e) Show the array or matrix that your dynamic programming algorithm produces on the example graph from Part (b) of the greedy algorithm.