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 $O(|V|^2)$.

   Solution: True. In a dense graph, $|E| = O(|V|^2)$. We know from class that the runtime of DFS is $O(|V| + |E|)$ so for dense graphs, the runtime is $O(|V|^2)$.

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

   Solution: False. Here is a counterexample: Consider the graph on three vertices with edges (a, b) and (b, c) each of weight 1 and (a, c) of weight 2 then this graph has two edges of length one but it only has one MST, namely $(a, b), (b, c)$.

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

   Solution: False. P is the set of all search problems for which there is a polynomial time algorithm to solve it. We have encountered many problems in P and we know that $\mathsf{P} \subseteq \mathsf{NP}$ therefore there are problems in NP for which there is a 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.

   Solution: False. If there are two sources, then there is no way to get from one to the other therefore there will be a vertex that was not 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.

   Solution: Create a new graph G' from G so that for each vertex v in G there are three copies v_1, v_2, v_3 in G'. For every edge (u, v) in G, G' will have edges $(u_1, v_2), (u_2, v_3)$ and (u_3, v_1). Then you run DFS from s_1 and if t_1 is visited then there is a path with length a multiple of 3 from s to t in G.

   Justification of correctness:
   If there is a path from s_1 to t_1 in G', then the sequence of vertices in the path must be of the form $(s_1, -2, -3, -1, -2, -3, \ldots, -1, -2, -3, t_1)$, in other words you must go through vertices in the order 1,2,3,1,2,3...1,2,3,1 and so if you start with 1 and end with 1, then the number of edges will be a multiple of 3.

   Runtime Analysis: Creating G' will take $O(|V| + |E|)$ because for each vertex, you must create

an adjacency list for G' that has 3 times as many edges then loop through the original adjacency list and for each edge, create 3 new edges in G' so you have to do 3 constant time operations for each vertex and each edge so the runtime is $O(|V| + |E|)$. Then DFS on this new graph is $O(3|V| + 3|E|) = O(|E| + |V|)$.

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 ← max(0, MaxWtSubtree(r.leftchild))
  - B ← max(0, MaxWtSubtree(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 $n = 2^h - 1$.

   Solution: The recurrence relation that we obtain for a complete and full binary tree is:

$$T(n) = 2 \cdot T\left(\frac{n-1}{2}\right) + O(1); \quad T(1) = O(1)$$

   We can use Master theorem with $a = 2; b = 2$; and $d = 0$. This gives $T(n) \in O(n^{\log_b a})$. So, $T(n) \in O(n)$.

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

   Solution: For the general case, we write the recurrence in terms of the number of nodes L in the left subtree and number of nodes R in the right subtree:

$$T(n) = T(L) + T(R) + O(1); \quad T(1) = O(1)$$

   We cannot apply the master method on this recurrence. We use the guess-and-check method. Let us guess $T(n) \in O(n)$. More specifically, we will show using induction that $T(n) \leq cn$ for a constant c that is larger than the constant for $T(1)$ and the constant involved in the $O(1)$ term in the recurrence.
   Base case: $T(1)$ is indeed at most $c \cdot 1$ because of the choice of our constant c.
   Induction step: We assume that $T(k) \leq c \cdot k$ for all $1 \leq k \leq n$ and will show that $T(n+1) \leq$

$c \cdot (n+1)$. For any tree with $n+1$ nodes and L and R nodes in the left and right subtrees respectively (this means that $L + R = n$), we have:

$$\begin{aligned} T(n+1) &\leq T(L) + T(R) + c \quad \text{(using recurrence relation)} \\ &\leq c \cdot L + c \cdot R + c \quad \text{(this follows from Induction Hypothesis)} \\ &\leq c \cdot (L + R + 1) = c \cdot (n+1). \end{aligned}$$

So, by the principle of Mathematical Induction, we get that for all $n \geq 1, T(n) \leq c \cdot n$. This implies $T(n) \in O(n)$.

(**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.*)

   Solution: Consider the bipartite graph $L = \{u_1, u_2, u_3\}$; $R = \{v_1, v_2, v_3\}$ and $E = \{(u_1, v_3), (u_2, v_2), (u_3, v_3)\}$.
   The greedy strategy A applied on this example will return $\{(u_1, v_3)\}$ whereas the largest monotone matching for this example is $\{(u_2, v_2), (u_3, v_3)\}$.

   (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 $E = \{(u_1, v_2), (u_1, v_3), (u_1, v_4), (u_2, v_1), (u_2, v_2), (u_2, v_3), (u_2, v_5)(u_3, v_1), (u_3, v_5)\}$.

   Solution: As per the greedy strategy A, the first edge to be chosen will be (u_1, v_2). After this, the greedy strategy is applied on the graph $L = \{u_2, u_3\}$; $R = \{v_3, v_4, v_5\}$; and $E = \{(u_2, v_3), (u_2, v_5), (u_3, v_5)\}$. The edge that is chosen next is (u_2, v_3). The graph after this is $L = \{u_3\}$; $R = \{v_4, v_5\}$; and $E = \{(u_3, v_5)\}$. At this point edge (u_3, v_5) is chosen. So, the greedy algorithm returns the monotone matching $\{(u_1, v_2), (u_2, v_3), (u_3, v_5)\}$.

(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 $j_T \leq k_T$.*

Solution: We prove using an exchange argument. We first show the following exchange lemma.
Exchange lemma: Let M be any perfect monotone matching such that $(u_1, v_{j_1}) \notin M$. Then there is another perfect monotone matching M' such that $(u_1, v_{j_1}) \in M'$.
Proof. Let $(u_1, v_{k_1}) \in M$. Due to our choice of j_1, we have $k_1 > j_1$. This means that $M' = M - \{(u_1, v_{k_1}) \cup \{(u_1, v_{j_1})\}\}$ is also a perfect monotone matching. □
We now show using induction (on the number of left vertices) that for any bipartite graph with perfect matching the greedy algorithm returns a perfect matching.
Base case: $l = 1$. For any such case the greedy algorithm indeed returns a matching of size 1.
Inductive step: Assume that the greedy algorithm returns a perfect matching for any bipartite graph with perfect matching where $l = 1, 2, ..., k$. We will show that the greedy algorithm also works for any graph with perfect matching with $l = k + 1$. Consider any such graph with $l = k + 1$. The greedy algorithm returns $(u_1, v_{j_1}) \cup$ greedy algorithm applied on subgraph with vertices $\{u_2, ..., u_l\}$ and $\{v_{j_1+1}, ..., v_r\}$. From our exchange lemma, we know that this remainder graph has a perfect matching and by the Induction hypothesis, we know that our greedy algorithm will return a matching of size $l-1 = k$ on this graph. So, the greedy algorithm returns a matching of size $k + 1$. This completes the inductive argument.

(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.

Solution: Let the bipartite graph be specified in an adjacency list format. This means that for every vertex on the left, its neighbors in the right are given (in no particular order). The pseudocode for the greedy algorithm can be written as follows:

```
GreedyA(L, R, E)
  - Use L, R, E to build an adjacency list for left vertices
  - M ← ∅; currentR = 0
  - for i = 1 to |L|:
      - min ← |R| + 1
      - For every neighbor v_j of u_i:
          - If (j ≥ currentR and j < min) min ← j
      - M ← M ∪ {(u_i, v_min)}
      - currentR ← min + 1
```

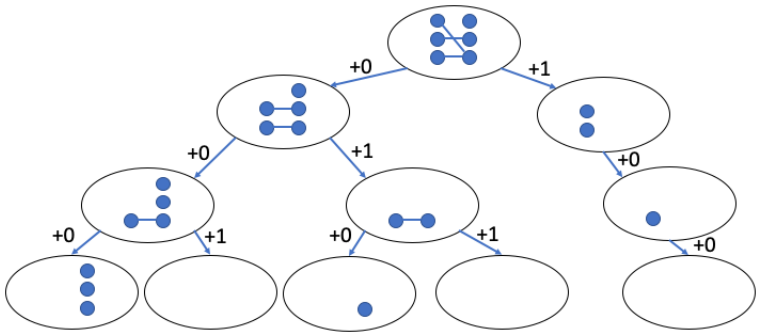
We note that the running time is proportional to going over the adjacency list of the left vertices which is $O(l + m)$.

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 ← 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 ∈ N(u_1)
  - Matched ← 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.)

Solution:



(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.)

Solution: There are two possibilities to consider for every vertex on the left and in the worst case all such combinations will be attempted. This will result in the number of recursive calls being $O(2^n)$. Another way of analysing the number of recursive calls is to write a recurrence relation. Let $R(n)$ denote the bound on the number of recursive calls for graphs where the number of vertices on the left is n. Then the recurrence relation will be $R(n) = 2 \cdot R(n-1)$. This gives $R(n) = O(2^n)$.

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

Solution: We note that the subproblems in the backtracking solution is of the general form: find the largest monotone matching in the subgraph with left vertices $u_i, ..., u_l$ and right vertices $v_j, ..., v_r$. So, we define the following quantity/matrix:
$MM[i, j]$: The size of the largest monotone matching in the subgraph with left vertices $u_i, ..., u_l$ and right vertices $v_j, ..., v_r$.
Then we can write the following base case for this quantity. $l+1$ corresponds to the left vertex set being empty:

$$MM[l+1, j] = 0 \quad \text{for any } j = 1, ..., r$$

We can write the following recursive relationship between these subproblems:

$MM[i, j] = \max(MM[i+1, j], 1 + MM[i+1, k+1])$, where $k \geq j$ is the least index such that $(u_i, v_k) \in E$

There are two possibilities to consider. First the case where u_i is not matched in which case the size of maximum matching is the same as size of maximum matching for subgraph $u_{i+1}, ..., u_l$ and $v_j, ..., v_r$. The other case is when u_i is part of the maximum monotone matching in which case, we match it to the smallest index vertex on the right to which it has an edge. This will maximise the matching size as per the exchange lemma that we prove in the previous problem. This explains the above recursion.

In the top-down recursion, i increases. So, for bottom-up dynamic programming computation, we need to consider decreasing i. This gives us the following DP algorithm:

```
DPMM(L, R, E)
  - l ← |L|; r ← |R|
  - for j = 1 to n: MM[l+1, j] ← 0
  - for i = l downto 1:
      - for j = 1 to r:
          - MM[i, j] ← MM[i+1, j]
          - If ((u_i, v_k) ∈ E for some k ≥ j)
              - Let k ≥ j be the least index such that (u_i, v_k) ∈ E
              - If (1 + MM[i+1, k+1] > MM[i, j]) MM[i, j] ← 1 + MM[i+1, k+1]
  - return(MM[1, 1])
```

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

Solution: For any j, the time required to fill the table entry $MM[i, j]$ is proportional to $deg(u_i)$ since it involves examining the neighbors of u_i. So, the time to fill the i^{th} row is $r \cdot (1 + deg(u_i))$. The overall time for filling the entire table is $\sum_i (r \cdot (1 + deg(u_i))) = O(r \cdot (l + m))$.

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

Solution:

	1	2	3	4	5
1	3	3	2	2	1
2	2	2	2	1	1
3	1	1	1	1	1
4	0	0	0	0	0