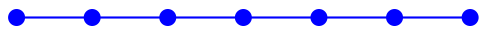
1 True or False (15 marks)

(i) Any tree with at least 7 vertices must have a vertex of degree at least 3.

False.

Consider the graph

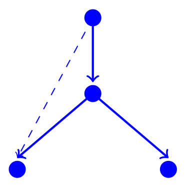


This graph is a tree, but each vertex has degree at most 2.

Each **True** or **False** answer should be accompanied by a brief justification. A single sentence should always be sufficient. These questions can cover any part of the course.

(ii) Let $G = (V, E)$ be a directed graph and let T be any DFS tree of G. Then for any edge $(u, v) \in E$ that is not in T we must have $\text{PRE}[u] > \text{PRE}[v]$.

False. We can have an edge from an ancestor to a descendent like the dashed line below.



(iii) Recall that we can modify BFS to check if a graph is *bipartite*. We can also modify BFS to check if a graph is *tripartite*. A graph $G = (V, E)$ is *tripartite* if there are disjoint sets of vertices V_1, V_2, V_3 such that $V = V_1 \cup V_2 \cup V_3$ and $\forall (u, v) \in E$, u, v are not in the same set V_i.

False. A tripartite graph is the same as a 3-Colorable graph, and 3-COLORABLE is **NP**–complete.

(iv) The complement independent set $S \subseteq V$ in a graph $G = (V, E)$ is a clique.

False. An independent set is the complement of a **vertex cover**. For example consider any subset of a graph with no edges.

(v) When constructing a verifier algorithm to prove that problem A is in **NP**, the length of the certificate that the verifier expects is important.

True. There must be some polynomial p so that certificates are of length at most $p(n)$ on inputs of size n.

2 Short answers (20 marks)

(i) Prove that if $G = (V, E)$ is a connected weighted graph with a unique minimum-weight edge e, then every minimum spanning tree of G contains the edge e.

Assume for contradiction that $T = (V, E')$ is a MST of G that does not include e. Let $H = (V, E' \cup \{e\})$. Adding the edge e creates a cycle, so we can remove any edge e' in this cycle to obtain a different spanning tree T' of G. But if we remove any $e' \neq e$ in this cycle, the resulting tree T' has smaller weight than T, contradicting the fact that T is a MST.

All answers in this section should be, as the title suggests, short. At most 3 sentences should be required to answer these questions in the exam.

(ii) For a directed graph G and a DFS tree T, explain why any edge $(u, v) \in E$ that is not in T must satisfy $\textsc{post}[u] > \textsc{post}[v]$.

If $\textsc{post}[u] < \textsc{post}[v]$ then all of neighbors of u have been explored before we finish exploring v. But v is a neighbor of u, so this is a contradiction.

(iii) Define the decision problem of LONGEST INCREASING SUBSEQUENCE (LIS): on input (x, k) return TRUE iff there is an increasing subsequence of length at least k in x. Recall the HAMILTONIAN PATH (HAMPATH) problem. Explain why LIS $\leq_{\mathbf{P}}$ HAMPATH.

LIS $\in \mathbf{P}$ so there is a polynomial-time algorithm A that solves LIS. Our polynomial-time reduction can be: run $A(x, k)$ to solve LIS, then construct a small dummy instance of HAMPATH (e.g. a YES-instance would be two connected vertices and a NO-instance would be two disconnected vertices).

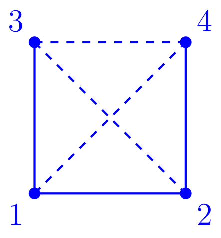
(iv) Are there graphs where ALL-PAIRS-SHORTEST-PATHS can be solved by using Dijkstra's algorithm on each vertex faster than it can be solved by Floyd–Warshall?

Yes. Dijkstra's algorithm has complexity $O((m + n)\log n)$ on graphs with n vertices and m edges, while Floyd–Warshall is $\Omega(n^3)$. We can solve APSP using Dijkstra in time $O(n^2 \log n)$ when $m = O(n)$ and the weights are non-negative.

(v) Give a counterexample showing that the following Divide & Conquer algorithm for computing the Minimum Spanning Tree of a complete graph[1] does not work (assume that $|V|$ is a power of 2):

```
Algorithm 1: MST($G = (V = [v_1, \ldots, v_n], E)$)
    if $n = 2$ then
        return $(v_1, v_2)$
    $V_1 \leftarrow [v_1, \ldots, v_{n/2}]$;
    $V_2 \leftarrow [v_{n/2+1}, \ldots, v_n]$;
    $e \leftarrow$ minimum-weight edge $(u, v)$ such that $u \in V_1, v \in V_2$;
    return MST($(V_1, E)$) $\cup$ MST($(V_2, E)$) $\cup \{e\}$ ;
```

Put the solid lines with weight 1 and the dashed lines with weight 100 in the below graph. Then the algorithm will select either $(1, 3)$ or $(2, 4)$ but not both.



These questions can cover any part of the course. These questions can ask for proofs (as above) or counter-examples, ask about specific examples, ask for definitions, ask for algorithms, ask about the analysis of specific algorithms, etc.

[1] A complete graph is one where there is an edge between every pair of vertices

3 Greedy algorithms (15 marks)

In the IndianaJones problem, the input is a set of n pieces of precious treasure with non-negative weights $w_1, \dots, w_n$ and non-negative values $v_1, \dots, v_n$, and a number t of items that you can put in a backpack. You goal is to find the set $S \subseteq \{1, 2, \dots, n\}$ of $|S| = t$ pieces of treasure with maximum value $\sum_{i \in S} v_i$.

For each of the following greedy algorithms, provide either a **proof** that the algorithm solves the IndianaJones problem or a **counter-example** showing that it does not.

(i) **Algorithm A:** Return the set $S \subseteq \{1, 2, \dots, n\}$ of the t pieces of treasure that have the largest values v_i.

Proof of correctness. We do an exchange argument. Assume that $T \subseteq \{1, 2, \dots, n\}$ is a valid solution to the problem, so that $\sum_{i \in T} v_i$ is maximized.

If $S = T$, then the solution returned by the algorithm also has maximum total value.

Otherwise, choose any $i \in T \setminus S$ and $j \in S \setminus T$. Then $v_i \leq v_j$, otherwise i would have been included in S instead of j, so the set $T' = (T \setminus \{i\}) \cup \{j\}$ also has maximum total value. We can continue this exchange process until we obtain a set $T' = S$, so S has maximum total value as well.

Note that a more formal version of the exchange argument above proceeds by induction on the size of the symmetric difference between the optimal set T and the set S returned by the algorithm. But it's not necessary to have those formal details in your solution: what is necessary, however, in the exchange argument, is a clear argument that we can indeed exchange some parts of the optimal solution to get closer to the solution returned by the algorithm.

We have seen in this class other proof techniques to establish the correctness of greedy algorithms; all of those are also perfectly acceptable.

(ii) **Algorithm B:** Return the set $S \subseteq \{1, 2, \ldots, n\}$ of t pieces of treasure that have the smallest weights w_i.

Counter-example. Take $t = 1$ and two pieces of treasure with values $v_1 = 1000$ and $v_2 = 1$ and weights $w_1 = 10$ and $w_2 = 5$. Algorithm B will select treasure 2, which has total value 1, instead of the treasure 1 that has value 1000.

Counter-examples need a justification to explain why they disprove the correctness of the algorithm, but that justification—as in the model solution above—does not need to be very long or formal.

(iii) **Algorithm C:** Return the set $S \subseteq \{1, 2, \ldots, n\}$ of the t pieces of treasure that have the largest ratios v_i/w_i.

Counter-example. Take $t = 1$ and two pieces of treasure, the first with value $v_1 = 1000$ and weight $w_1 = 1000$ and the second with value $v_2 = 10$ and weight $w_2 = 1$. Then the algorithm selects treasure 2 (since $v_2/w_2 = 10 > 1 = v_1/w_1$) which has value 10 instead of treasure 1 that has value 1000.

4 Dynamic programming (20 marks)

In the ConsecSum problem, we are given n integers $a_1, a_2, \ldots, a_n$, at least one of which is positive, and we must find a pair of indices $1 \le \ell \le r \le n$ such that the sum of consecutive integers $\sum_{i=\ell}^{r} a_i$ is maximized.

(i) Give an input showing that the sum $\sum_{i=1}^{n} a_i$ does not always have the maximum value among all sums of consecutive integers.

Take $a_1 = -1$, $a_2 = 5$, $a_3 = 4$, and $a_4 = -1$.

Then $a_1 + a_2 + a_3 + a_4 = 7$ but $a_2 + a_3 = 9 > 7$. So the sum of all integers has smaller sum than the sum of the two consecutive integer a_2 and a_3.

(ii) A possible greedy algorithm for the problem finds the largest integer a_{m^*}, then returns the smallest integer $1 \le \ell \le m^*$ and the largest integer $m^* \le r \le n$ for which $a_\ell, a_{\ell+1}, \ldots, a_{r-1}, a_r$ are all non-negative. Give an example showing that this algorithm does not always find the optimal solution.

Take $a_1 = 10$, $a_2 = -1$, $a_3 = 1$, $a_4 = 15$, $a_5 = 1$, $a_6 = -1$, and $a_7 = 10$.

The greedy algorithm identifies $m^* = 3$ and returns $\ell = 3$ and $r = 5$. The resulting consecutive sum has value $a_3 + a_4 + a_5 = 1 + 15 + 1 = 17$ but the maximum consecutive sum is $\sum_{i=1}^{7} a_i = 35$.

(iii) Design and analyze a dynamic programming algorithm that solves the ConsecSum problem. Your answer must include the algorithm description, pseudocode, a proof of correctness, and the time complexity analysis. (Your answer should include the subproblems being solved and how they are solved.)

Subproblems. For $k = 1, 2, \ldots, n$, define

$$B[k] = \max_{1 \le j \le k} \sum_{i=j}^{k} a_i$$

to be the maximum value of a consecutive sum that ends at k.

Base case. $B[1] = a_1$.

Recurrence. $B[k] = \begin{cases} B[k-1] + a_k & \text{if } B[k-1] \ge 0 \\ a_k & \text{otherwise.} \end{cases}$

Pseudocode.

```
B[1] = $a_1$;
for k=2,3,...,n
   B[k] = max(B[k-1] + $a_k$, $a_k$);
return max(B[1], B[2], ..., B[n]);
```

Proof of correctness. We first show $B[k] = \max_{1 \le j \le k} \sum_{i=j}^{k} a_i$, as claimed. $B[1] = a_1$ by the base case.

For $k > 1$, there are two possibilities: either the maximum sum that ends at a_k includes some elements a_i with $i < k$ (in which case $B[k] = B[k-1] + a_k$ since it equals the largest sum that ends at a_{k-1} plus the value a_k) or it does not (in which case $B[k] = a_k$). So the recurrence correctly computes $B[2], B[3], \ldots, B[n]$.

Finally, the largest consecutive sum in the input ends at a_k for *some* $1 \le k \le n$ so the value $\max\{B[1], \ldots, B[n]\}$ correctly returns the consecutive sum with maximum value.

Time complexity. The initialization takes constant time. Each iteration of the loop takes constant time for a total time complexity $\Theta(n)$. The final line examines n values to return the maximum one so takes time $\Theta(n)$.

So the overall time complexity of the algorithm is $\Theta(1) + \Theta(n) + \Theta(n) = \Theta(n)$.

We highly recommend writing solutions for dynamic programming algorithm design questions in the format above, with

- Subproblems,
- Base case(s),
- Recurrence,
- Pseudocode,
- Proof of correctness, and
- Time complexity analysis

sections. With this format, the proof of correctness should be quite short and should only have to justify that the recurrence is correct and that the subproblems really do yield a correct final solution.

5 Graph algorithms (20 marks)

There are n cities $c_1, \ldots, c_n$ that you are interested in visiting with your car. Some of these cities have roads between them, and you know what the cost of fuel $f_{i,j} \geq 0$ would be to drive between each pair of cities c_i, c_j that are connected by a road (roads can be one-way, and the cost of fuel is not necessarily the same in both directions because of hills etc.).

Also, between each pair of cities c_i, c_j with a road between them, there may be carpoolers willing to pay you some amount of money $p_{i,j} \geq 0$ to travel from c_i to c_j. (Assume that there will always be carpoolers willing to pay the price.) For example, we might have the following situation:

Road	Fuel Cost	Carpool Price
$c_1 \to c_2$	$f_{1,2} = 50$	$p_{1,2} = 0$
$c_2 \to c_1$	$f_{2,1} = 20$	$p_{2,1} = 10$
$c_2 \to c_3$	$f_{2,3} = 40$	$p_{2,3} = 30$
$c_3 \to c_4$	$f_{3,4} = 10$	$p_{3,4} = 10$
$c_4 \to c_2$	$f_{4,2} = 70$	$p_{4,2} = 90$
$c_4 \to c_1$	$f_{4,1} = 60$	$p_{4,1} = 20$

(i) Describe how to encode the travel prices in a graph, and draw the resulting graph for the example above.

We create a graph with one vertex for each city c_i, and with an edge with weight $f_{i,j} - p_{i,j}$ for every road from city c_i to c_j.

(Note that a complete solution should also include a drawing of the directed weighted graph on 4 nodes for the example above.)

For the rest of this question, you will describe how to solve **Problems A–D** using graph algorithms. For each problem, do the following two steps.

1. Identify the corresponding graph problem. Your answer should be one of the graph problems we have seen in class:

Graph problems
Connectedness (also known as Reachability)
Testing bipartiteness
Computing the spanning tree
Finding cut vertices
Testing acyclicity of directed graphs (also known as Testing DAGs)
Linearization of DAGs (also known as Topological Sorting of DAGs)
Strong connectivity of directed graphs
Minimum Spanning Tree (MST) of weighted graphs
Single-Source Shortest Paths (SSSP)
All-Pairs Shortest Paths (APSP)

2. Describe the algorithm that you will use to solve the problem. **Only a high-level description of the algorithm is required**. You do **not** need to provide detailed pseudocode, a proof of correctness, or a formal time complexity analysis of your algorithm. You may use DFS and BFS in your solution; any other algorithm that we have seen in class for the graph problems above needs to be described in your solution.

(ii) **Problem A:** What is the minimum number of legs of the journey $c_1 \to c_n$? (A *leg* of a journey is a trip between adjacent cities.)

Graph problem: SSSP (in unweighted graphs)

Algorithm: Run BFS from the vertex for c_1 with the variant where when we discover a new vertex, we mark its distance from c_1 to be 1 + the distance of the vertex from which it was discovered.

(Note that we ignore the edge weights in this solution.)

Note that the description of the algorithm is only the high-level description. No need to specify the details of the implementation or the formal proof of correctness, as the instructions for the question point out.

(iii) **Problem B:** What is the smallest net cost (carpooler revenue − fuel cost) of a trip from c_1 to c_n?

Graph problem: SSSP in weighted graphs.

Algorithm: Bellman–Ford algorithm (because the weights can be negative). Go through each edge $(u, v) \in E$ and update the distance to v if taking this edge from u gives a shorter path to v than any found so far. Repeat this process $n - 1$ times so that we consider all paths of length at most $n - 1$, which will include the shortest path.

Just saying "Bellman–Ford algorithm" is not sufficient, but you should be able to describe this algorithm and all the other graph algorithms we saw in class in a similar brief way using just a few sentences. And as in the solution above, it's not necessary to have all the precise details of the algorithm (for instance, the description above does not describe how to initialize the distances) but the main idea should be present.

(iv) **Problem C:** You have a friend Alice at city c_i wanting to get to city $c_{i'}$ and another friend Bob at $c_j \neq c_i$ wanting to get to city $c_{j'}$. Is it possible to get both friends where they want to go and make a profit at the same time? (You start at city c_1 and you may end up anywhere.)

Graph problem: SSSP in weighted graphs.

Algorithm: There are 6 possible kinds of paths:

$$\begin{gathered} c_1 \to c_i \to c_{i'} \to c_j \to c_{j'} \\ c_1 \to c_i \to c_j \to c_{i'} \to c_{j'} \\ c_1 \to c_i \to c_j \to c_{j'} \to c_{i'} \\ c_1 \to c_j \to c_{j'} \to c_i \to c_{i'} \\ c_1 \to c_j \to c_i \to c_{j'} \to c_{i'} \\ c_1 \to c_j \to c_i \to c_{i'} \to c_{j'} \, . \end{gathered}$$

The most profitable (i.e. cheapest) paths for each of these options can be found using 5 calls to Bellman–Ford, with starting points $c_1, c_i, c_j, c_{i'}, c_{j'}$.

If you do use the same graph algorithm in many questions, no need to repeat its description.

(v) **Problem D:** You are lonely and you want to make some friends, so you want to take only roads where carpoolers will keep you company. Starting from c_1, can you visit any city you want, if you only take roads with carpoolers? (Assume that if $p_{i,j} = 0$ there are no carpoolers wanting to go from city c_i to c_j.)

Graph problem: Connectivity (on the graph containing only edges (i, j) with $p_{i,j} > 0$).

Algorithm: Run DFS or BFS and check that every vertex is visited.

6 NP-completeness (15 marks)

Consider the following two decision problems.

Problem A. CliqueAndUnconnected: Given a graph $G = (V, E)$ and a parameter k, determine if G has a clique of size at least k *and* is not a connected graph.

Problem B. LongPath: Given a graph $G = (V, E)$ and a parameter k, determine if there is a simple path in G of length at least k.

(i) Prove that CliqueAndUnconnected $\in$ **NP**.

Let A be a verifier that requests a clique $S \subseteq V$ of size $|S| = k$ as its certificate. In polynomial time, A can verify that for every $u \neq v \in S$, the edge (u, v) is in the graph. If any edge is missing, the verifier outputs `No`. Then by running a BFS or DFS from any start node, the verifier A can check in polynomial time whether the graph is connected; if it is, A outputs `No`. If the set S does form a clique and the BFS or DFS showed that G is not connected, A outputs `Yes`.

When G is a `Yes` instance of CliqueAndUnconnected, there is a certificate that causes A to output `Yes` (namely: a set S of k vertices in the clique of size at least k in G).

And when G is a `No` instance of CliqueAndUnconnected, A outputs `No` regardless of the provided certificate: if G is connected, A always outputs `No` after the connectedness test, and if G does not have a clique of size k then any possible set S of size k will be missing at least one edge.

Your answer should describe the verifier algorithm, and justify its correctness by explaining why there is a certificate that causes it to accept all `Yes` instances and why it always rejects (no matter what claimed certificate is provided) all `No` instances.

(ii) Prove that LongPath $\in$ **NP**.

Let A be a verifier that asks for the list of vertices $v_0, v_1, v_2, \ldots, v_k$ followed in a simple path of length k in G as a certificate. It then checks that the k vertices are all distinct, and that $(v_{i-1}, v_i) \in E$ for each $1 \leq i \leq k$. It outputs `Yes` iff both of the checks pass. This algorithm runs in polynomial time.

When G does contain a simple path of length k, we can provide the vertices visited along the path to the verifier and it will accept.

When G does not contain a simple path of length k, then any claimed certificate will either contain a duplicated vertex or it will contain a pair of adjacent vertices v_{i-1} and v_i such that $(v_{i-1}, v_i) \notin E$. Therefore, A always outputs `No`.

(iii) Prove **<u>ONE</u>** of the following two statements:

- Clique $\leq_\mathbf{P}$ CliqueAndUnconnected.[2]

  OR

- HamPath $\leq_\mathbf{P}$ LongPath.[3]

(You may prove either one; the choice is up to you.)

We prove Clique $\leq_\mathbf{P}$ CliqueAndUnconnected.

Let $G = (V, E)$ and k be an input to Clique.

Construct $G' = (V \cup \{v'\}, E)$ be the graph obtained by adding an extra isolated vertex v to G. Consider the input G', k to the CliqueAndUnconnected problem. The transformation is completed in polynomial time.

If G, k is a `Yes` instance to Clique, then G contains a clique of size k. So G' also contains a clique of size k, and it is not a connected graph because of v. Therefore G', k is a `Yes` instance to CliqueAndUnconnected.

If G, k is a `No` instance to Clique, then G does not contain a clique of size k and neither does G' (since we didn't add any new edges). So G', k is a `No` instance to CliqueAndUnconnected.

The answer must describe the reduction itself, and it must include a justification that `Yes` instances do map to `Yes` instances, and that `No` instances map to `No` instances.

No detailed justification of the polynomial-time complexity of the reduction is necessary unless it's not obvious from the construction.

[2] Clique: Given a graph $G = (V, E)$ and a positive integer k, determine if there is a set $S \subseteq V$ of size $|S| \leq k$ such that for every distinct $u, v \in S$, $(u, v) \in E$.

[3] HamPath: Given a graph $G = (V, E)$, determine if there is an ordering $v_1, v_2, \ldots, v_n$ of the vertices in V such that for every $2 \leq i \leq n$, $(v_{i-1}, v_i) \in E$.

USE THIS PAGE IF ADDITIONAL SPACE IS REQUIRED

Clearly state the question number being answered and refer the marker to this page.

This page may be torn off and used as scrap paper.

This page may be torn off and used as scrap paper.