Final Exam

> *Consider a connected undirected graph with distinct edge costs. Which of the following are true? [Check all that apply.]*
>
> - *Suppose the edge e is not the cheapest edge that crosses the cut (A, B). Then e does not belong to any minimum spanning tree.*
> - *Suppose the edge e is the most expensive edge contained in the cycle C. Then e does not belong to any minimum spanning tree.*
> - *The minimum spanning tree is unique.*
> - *Suppose the edge e is the cheapest edge that crosses the cut (A, B). Then e belongs to every minimum spanning tree.*

ANSWER: Option 3 is correct because the edge weights are distinct. Options 2 and 4 follow from the Cut Property based on the correctness of option 3 and are correct as well.

Consider the following graph, where cost of edge $A - B$ is greater than the cost of edge $C - D$. Edges $A - B$ and $C - D$ are both included in the MST, even though $A - B$ isn't the cheapest edge crossing the cut shown.

Thus, option 1 is incorrect.

> *Which of the following graph algorithms can be sped up using the heap data structure?*
>
> - *Dijkstra's single-source shortest-path algorithm (from Part 2).*
> - *Prim's minimum-spanning tree algorithm.*
> - *Our dynamic programming algorithm for computing a maximum-weight independent set of a path graph.*
> - *Kruskal's minimum-spanning tree algorithm.*

ANSWER: Options 1 and 2 are correct. MWIS doesn't consider the edges in sorted order, so a heap won't help. Kruskal's algorithm requires the edges to be sorted, which can be done by using a min-heap, but it won't provide any speed up over a fast comparison sort.

> *Which of the following problems reduce, in a straightforward way, to the minimum spanning tree problem? [Check all that apply.]*
>
> - *The maximum-cost spanning tree problem. That is, among all spanning trees of a connected graph with edge costs, compute one with the maximum-possible sum of edge costs.*
> - *The minimum bottleneck spanning tree problem. That is, among all spanning trees of a connected graph with edge costs, compute one with the minimum-possible maximum edge cost.*
> - *The single-source shortest-path problem.*
> - *Given a connected undirected graph $G = (V, E)$ with positive edge costs, compute a minimum-cost set $F \subseteq E$ such that the graph $(V, E - F)$ is acyclic.*

ANSWER: If we negate the weights of all edges, a minimum-cost spanning tree algorithm yields a maximum-cost spanning tree. Thus, option 1 is correct.

A minimum bottleneck spanning tree is also a MST. Thus, option 2 is correct.

Option 4 is also correct since a MST, by definition, doesn't have a cycle. However, it's not clear to me what prevents is to take only a single edge. There's nothing in the question that indicates $(V, E - F)$ needs to be connected.

Option 3 is incorrect; this problem doesn't consider edge costs.

> *Recall the greedy clustering algorithm from lecture and the max-spacing objective function. Which of the following are true? [Check all that apply.]*
>
> - *If the greedy algorithm produces a k-clustering with spacing S, then every other k-clustering has spacing at most S.*
> - *Suppose the greedy algorithm produces a k-clustering with spacing S. Then, if x, y are two points in a common cluster of this k-clustering, the distance between x and y is at most S.*
> - *If the greedy algorithm produces a k-clustering with spacing S, then the distance between every pair of points chosen by the greedy algorithm (one pair per iteration) is at most S.*
> - *This greedy clustering algorithm can be viewed as Prim's minimum spanning tree algorithm, stopped early.*

ANSWER: Option 1 is correct; the greedy algorithm is optimal.

Option 2 is incorrect. The greedy algorithm merges closest points as long as the number of clusters is less than or equal to k. If the distance between x, y was at most S, they would've been merged.

Option 3 is correct.

Option 4 is incorrect. The greedy algorithm can be viewed as Kruskal's algorithm stopped early, not Prim's.

> *We are given as input a set of n jobs, where job j has a processing time p_j and a deadline d_j. Recall the definition of completion times C_j from the video lectures. Given a schedule (i.e., an ordering of the jobs), we define the lateness l_j of job j as the amount of time $C_j - d_j$ after its deadline that the job completes, or as 0 if $C_j \leq d_j$. Our goal is to minimize the total lateness, $\sum_j l_j$.*
>
> *Which of the following greedy rules produces an ordering that minimizes the total lateness? You can assume that all processing times and deadlines are distinct.*
>
> *WARNING: This is similar to but not identical to a problem from Problem Set #1 (the objective function is different).*
>
> - *Schedule the requests in increasing order of processing time p_j*
> - *None of the other options are correct*
> - *Schedule the requests in increasing order of deadline d_j*
> - *Schedule the requests in increasing order of the product $d_j \cdot p_j$*

ANSWER: Potts and Van Wassenhove have shown this to be a NP-Hard problem Single Machine Scheduling to Minimize Total Late Work (https://pubsonline.informs.org/doi/abs/10.1287/opre.40.3.586). Option 2 is correct.

> *Which of the following extensions of the Knapsack problem can be solved in time polynomial in n, the number of items, and M, the largest number that appears in the input? [Check all that apply.]*
>
> - *You are given n items with positive integer values and sizes, as usual, and two positive integer capacities, W_1 and W_2. The problem is to pack items into these two knapsacks (of capacities W_1 and W_2) to maximize the total value of the packed items. You are not allowed to split a single item between the two knapsacks.*
> - *You are given n items with positive integer values and sizes, and a positive integer capacity W, as usual. The problem is to compute the max-value set of items with total size exactly W. If no such set exists, the algorithm should correctly detect that fact.*
> - *You are given n items with positive integer values and sizes, and a positive integer capacity W, as usual. You are also given a budget $k \leq n$ on the number of items that you can use in a feasible solution. The problem is to compute the max-value set of at most k items with total size at most W.*
> - *You are given n items with positive integer values and sizes, as usual, and m positive integer capacities, $W_1, W_2, \ldots, W_m$. These denote the capacities of m different Knapsacks, where m could be as large as $\Theta(n)$. The problem is to pack items into these knapsacks to maximize the total value of the packed items. You are not allowed to split a single item between two of the knapsacks.*

ANSWER: Problem 1 is the **0-1 Knapsack Problem** with 2 knapsacks, and is solvable in polynomial time. Add another dimension to the array to keep track of the residual capacity of the second knapsack, this increases the running time by a factor of at most W. See Problem Set 3 for a solution.

Problem 2 is the **Subset-Sum Problem**, and is solvable in polynomial time.

Problem 3 is the **Bounded Knapsack Problem**, and is solvable in polynomial time. Add another dimension to the array to keep track of number of items used so far, this increases the running time by a factor of at most n.

Problem 4 is NP-Complete, and is not solvable in polynomial time.

Thus, options 1, 2 and 3 are correct.

> *Of the following dynamic programming algorithms covered in lecture, which ones always perform $O(1)$ work per subproblem? [Check all that apply.]*
>
> - *The Floyd-Warshall all-pairs shortest-paths algorithm.*
> - *The dynamic programming algorithm for the optimal binary search tree problem.*
> - *The dynamic programming algorithm for the knapsack problem.*
> - *The dynamic programming algorithm for the sequence alignment problem.*
> - *The Bellman-Ford shortest-path algorithm.*

ANSWER: Each of Floyd-Warshall, DP algorithms for the Knapsack Problem and the Sequence Alignment Problem do $O(1)$ work per subproblem.

Bellman-Ford does linear work per subproblem, so does the DP algorithm for the Optimal BST Problem.

Thus, options 1, 3, and 4 are correct.

> *Assume that $P \neq NP$. Which of the following problems can be solved in polynomial time? [Check all that apply.]*
>
> - *Given a directed acyclic graph with real-valued edge lengths, compute the length of a longest path between any pair of vertices.*
> - *Given a directed graph with real-valued edge lengths, compute the length of a longest cycle-free path between any pair of vertices (i.e., $\max_{u,v \in V} \max_{P \in \mathrm{P}_{uv}} \sum_{e \in P} c_e$, where P_{uv} denotes the set of cycle-free $u - v$ paths).*
> - *Given a directed graph with nonnegative edge lengths, compute the length of a longest cycle-free path between any pair of vertices (i.e., $\max_{u,v \in V} \max_{P \in \mathrm{P}_{uv}} \sum_{e \in P} c_e$, where P_{uv} denotes the set of cycle-free $u - v$ paths).*
> - *Given a directed graph with nonnegative edge lengths, compute the length of a maximum-length shortest path between any pair of vertices (i.e., $\max_{u,v \in V} d(u, v)$, where $d(u, v)$ is the shortest-path distance between u and v).*

ANSWER: Option 1: Longest path in a DAG can be computed in polynomial (linear) time by simply negating the edge weights. also see this paper.

Option 2: The NP-Complete Hamiltonian Path problem reduces easily to this problem, so it cannot be solved in polynomial time assuming $P \neq NP$.

Option 3: The NP-Complete Hamiltonian Path problem reduces easily to this problem, so it cannot be solved in polynomial time assuming $P \neq NP$.

Option 4: Since edge lengths are nonnegative, there are no negative cycles. Thus, this problem reduces to all-pairs shortest paths.

> *Recall the all-pairs shortest-paths problem. Which of the following algorithms are guaranteed to be correct on instances with negative edge lengths that don't have any negative-cost cycles? [Check all that apply.]*
>
> - *Run the Bellman-Ford algorithm n times, once for each choice of a source vertex.*
> - *Run Dijkstra's algorithm n times, once for each choice of a source vertex.*
> - *Johnson's reweighting algorithm.*
> - *The Floyd-Warshall algorithm.*

ANSWER: Options 1,2, and 4 are correct. Follows from the lectures.

> *Suppose a computational problem Π that you care about is NP-complete. Which of the following are true? [Check all that apply.]*
>
> - *Since the dynamic programming algorithm design paradigm is only useful for designing exact algorithms, there's no point in trying to apply it to the problem Π.*
> - *You should not try to design an algorithm that is guaranteed to solve Π correctly and in polynomial time for every possible instance (unless you're explicitly trying to prove that $P = NP$).*
> - *NP-completeness is a "death sentence"; you should not even try to solve the instances of Π that are relevant for your application.*
> - *If your boss criticizes you for failing to find a polynomial-time algorithm for Π, you can legitimately claim that thousands of other scientists (including Turing Award winners, etc.) have likewise tried and failed to solve Π.*

ANSWER: Option 1: False; dynamic programming can potentially be used to design faster (but still exponential-time) exact algorithms (as with TSP), to design heuristics with provable performance guarantees (as with Knapsack), and to design exact algorithms for special caes (as with Knapsack).

Option 2: True; remember, in trying to solve one NP-complete problem, we are trying to solve them all. Countless brilliant minds have tried to devise polynomial-time algorithms for NP-complete problems (and thus, indirectly, for this NP-complete problem); none have yet succeeded.

Option 3: False; perhaps the instances of Π arising in our domain are special enough to be solved efficiently (in theory and/or in practice).

Option 4: True.

> *Of the following problems, which can be solved in polynomial time by directly applying algorithmic ideas that were discussed in lecture and/or the homeworks? [Check all that apply.]*
>
> - *Given an undirected graph G and a value for k such that $k = \Theta(\log n)$, where n is the number of vertices of G, does G have a vertex cover of size at most k?*
> - *Given an undirected graph G and a constant value for k (i.e., $k = O(1)$, independent of the size of G), does G have an independent set of size at least k?*
> - *Given an undirected graph G and a value for k such that $k = \Theta(\log n)$, where n is the number of vertices of G, does G have an independent set of size at least k?*
> - *Given an undirected graph G and a constant value for k (i.e., $k = O(1)$, independent of the size of G), does G have a vertex cover of size at most k?*

ANSWER: Option 1: True; the Vertex Cover algorithm covered in lecture has running time $O(2^k m)$ and hence runs in polynomial time in this case.

Option 2: True; brute-force search (checking each subset of k vertices) runs in time $O(n^k)$, which is polynomial when $k = O(1)$.

Option 3: False; there is a reduction between the vertex cover and independent set problems where one can take the complement of one to get the other. Unfortunately, this transforms vertex covers of size k to independent sets of size $n - k$ and thus is not useful here. Also, it is not clear how to adapt the Vertex Cover algorithm from lecture to the Independent Set problem. In fact, it is conjectured that this problem cannot be solved in polynomial time at all.

Option 4: True; brute-force search (checking each subset of k vertices) runs in time $O(n^k)$, which is polynomial when $k = O(1)$.

> *In lecture we gave a dynamic programming algorithm for the traveling salesman problem. Does this algorithm imply that $P = NP$? [Check all that apply.]*
>
> - *Yes, it does.*
> - *No. A polynomial-time algorithm for the traveling salesman problem does not necessarily imply that $P = NP$.*
> - *No. Since we sometimes perform a super-polynomial amount of work computing the solution of a single subprolem, the algorithm does not run in polynomial time.*
> - *No. Since we perform a super-polynomial amount of work extracting the final TSP solution from the solutions of all of the subproblems, the algorithm does not run in polynomial time.*
> - *No. Since there are an exponential number of subproblems in our dynamic programming formulation, the algorithm does not run in polynomial time.*

ANSWER: Option 1: False; if it did, we'd be a million bucks richer!

Option 2: False; since (the decision version of) the traveling salesman problem is NP-complete, a polynomial-time algorithm for TSP would indeed imply that $P = NP$.

Option 3: False; we only do $O(n)$ work computing the answer to a single subproblem; the issue is that there are exponentially many such subproblems.

Option 4: False; We only do $O(n)$ work computing the final answer, given the solutions of all of the (exponentially many) subproblems.

Option 5: True.

> *Consider the Knapsack problem and the following greedy algorithm: (1) sort the items in nonincreasing order of value over size (i.e., the ratio $\frac{v_i}{w_i}$); (2) return the maximal prefix of items that fits in the Knapsack (i.e., the k items with the largest ratios, where k is as large as possible subject to the sum of the item sizes being at most the knapsack capacity W). Which of the following are true? [Check all that apply.]*
>
> - *If all items have the same value, then this algorithm always outputs an optimal solution (no matter how ties are broken).*
> - *This algorithm always outputs a feasible solution with total value at least 50% times that of an optimal solution.*
> - *If all items have the same size, then this algorithm always outputs an optimal solution (no matter how ties are broken).*
> - *If the size of every item is at most 20% of the Knapsack capacity (i.e., $w_i \leq \frac{W}{5}$ for every i), then this algorithm is guaranteed to output a feasible solution with total value at least 80% times that of an optimal solution.*
> - *If all items have the same value/size ratio, then this algorithm always outputs an optimal solution (no matter how ties are broken).*

ANSWER: Remember that in the Knapsack Problem, we prefer items of higher value and smaller size.

Option 1: True; clearly, the more number of items we pick, greater the value. We can pick more items we we prefer smaller ones than larger ones.

Option 2: False; this is only true if you add a third step, which takes the better of this solution and the max-value item (as discussed in lecture).

Option 3: True; all feasible solutions pack the same number of items, thus, clearly, the optimal solution should choose items with higher values first.

Option 4: True; as discussed in lecture, this is true even without the “step 3” optimization that compares the greedy solution to the max-value item.

Option 5: False; suppose $W = 4, \{(v_i, w_i)\} = \{(3, 3), (2, 2), (2, 2)\}$. If ties are broken arbitrarily, v_1 might be chosen, whereas the optimal solution would choose v_2 and v_3.

> *Which of the following statements are true about the tractability of the Knapsack problem? [Check all that apply.]*
>
> - *Assume that $P \neq NP$. The special case of the Knapsack problem in which all item sizes are positive integers less than or equal to n^5, where n is the number of items, can be solved in polynomial time.*
> - *If there is a polynomial-time algorithm for the Knapsack problem in general, then $P \neq NP$.*
> - *Assume that $P \neq NP$. The special case of the Knapsack problem in which all item values are positive integers less than or equal to n^5, where n is the number of items, can be solved in polynomial time.*
> - *Assume that $P \neq NP$. The special case of the Knapsack problem in which all item values, item sizes, and the knapsack capacity are positive integers, can be solved in polynomial time.*

ANSWER: Option 1: True; lecture video 17.4.

Option 2: True; the (decision version of) the Knapsack problem is NP-Complete.

Option 3: True; lecture video 17.5.

Option 4: False; if the w_is and W are integers, can solve via DP in $O(nW)$ time. If v_is are integers, can solve in $O(n^2 v_{max})$ time. Can't solve in polynomial time for arbitrarily large item values, item sizes, and the knapsack capacity.

> *Which of the following graph algorithms can be implemented, using suitable data structures, in $O(m \log n)$ time? (As usual, m and n denote the number of edges and vertices, respectively.) [Check all that apply.]*
>
> - *The Bellman-Ford shortest-path algorithm.*
> - *Prim's minimum spanning tree algorithm.*
> - *Johnson's all-pairs shortest-path algorithm.*
> - *Kruskal's minimum spanning tree algorithm.*

ANSWER: Option 1: False; $O(mn)$, which in sparse graphs, is $O(n^2)$, in dense graphs, $O(n^3)$.

Option 2: True; as covered in lecture, using the heap data structure.

Option 3: False; $O(mn \log n)$. Better than n invocations of Bellman-Ford in sparse graphs.

Option 4: True; as covered in lecture, using the union-find data structure.