# Final Exam

Back to Week 7



**33/40** points earned (82%)

Quiz passed!



# Be Recognized for Your Achievements.

"Course Certificates give you the recognition you need to get the job, the material gives you the skills to do the job. It makes you look

more valuable because you are more valuable." - Peter B., USA, Software Developer

Showcase Your Accomplishment! Earn Your Course

Certificate! \$79 USD >



2/2 points

1.

Consider a connected undirected graph with distinct edge costs. Which of the following are true? [Check all that apply.]

The minimum spanning tree is unique.

## **Correct Response**

We proved this in the video lectures (see the correctness of Prim's algorithm).

Suppose the edge e is the cheapest edge that crosses the cut (A, B). Then e belongs to every minimum spanning tree.

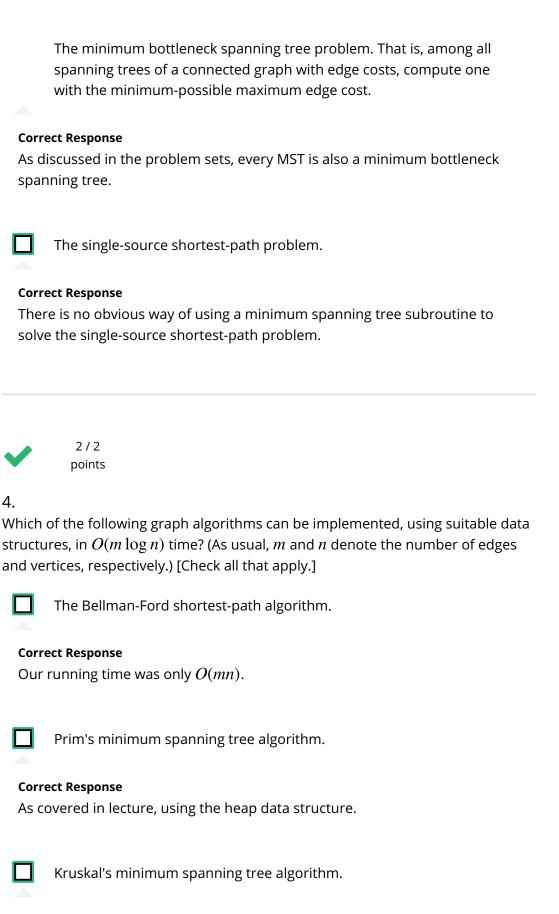
# **Correct Response**

This is the Cut Property, from the lectures.

	Suppose the edge $e$ is the most expensive edge contained in the cycle $C$ . Then $e$ does not belong to any minimum spanning tree. <b>ect Response</b> discussed this property in Problem Set #2.
	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. <b>ect Response</b> two-edge path graph provides a counterexample.
adjace spanni if we cl is not k	2/2 points e given a connected undirected graph $G$ with distinct edge costs, in ncy list representation. You are also given the edges of a minimum ing tree $T$ of $G$ . This question asks how quickly you can recompute the MST hange the cost of a single edge. Which of the following are true? [RECALL: It known how to deterministically compute an MST from scratch in $O(m)$ where $m$ is the number of edges of $G$ .] [Check all that apply.]
Let A belo cros	Suppose $e \in T$ and we increase the cost of $e$ . Then, the new MST can be recomputed in $O(m)$ deterministic time. <b>ect Response</b> $A,B$ be the two connected components of $T-\{e\}$ . Edge $e$ no longer ongs to the new MST if and only if it is no longer the cheapest edge using the cut $(A,B)$ (this can be checked in $O(m)$ time). If $f$ is the new appest edge crossing $(A,B)$ , then the new MST is $T-\{e\}\cup\{f\}$ .
	Suppose $e \not\in T$ and we decrease the cost of $e$ . Then, the new MST can be recomputed in $O(m)$ deterministic time.
Let (	ect Response $C$ be the cycle of $T \cup \{e\}$ . Edge $e$ belongs to the new MST if and only is no longer the most expensive edge of $C$ (this can be checked in

if it is no longer the most expensive edge of C (this can be checked in O(n) time). If f is the new most expensive edge of C, then the new MST is  $T \cup \{e\} - \{f\}$ .

	Suppose $e\in T$ and we decrease the cost of $e$ . Then, the new MST can be recomputed in $O(m)$ deterministic time.			
The M	Correct Response The MST does not change (by the Cut Property), so no re-computation is needed.			
	Suppose $e \not\in T$ and we increase the cost of $e$ . Then, the new MST can be recomputed in $O(m)$ deterministic time.			
The M	Is Response IST does not change (by the Cycle Property of the previous problem), re-computation is needed.			
<b>~</b>	2/2 points			
	f the following problems reduce, in a straightforward way, to the m 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.			
	egate all edge costs and run an MST algorithm.			
_	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.			
Correc	rt Response			
	ptimal such set is the complement of a maximum-cost spanning			



4.

As covered in lecture, using the union-find data structure.

Johnson's all-pairs shortest-path algorithm.

Our running time was only  $O(mn \log n)$ .



2/2 points

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 the distance between every pair of points chosen by the greedy algorithm (one pair per iteration) is at most S.

# **Correct Response**

This was a lemma we used in the proof of correctness for the greedy clustering algorithm.

If the greedy algorithm produces a k-clustering with spacing S, then every other k-clustering has spacing at most S.

# **Correct Response**

This is precisely the correctness theorem we discussed for the greedy clustering algorithm.

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.

#### **Correct Response**

It can be more, as we discussed in the proof of correctness of the greedy clustering algorithm.

This greedy clustering algorithm can be viewed as Prim's minimum spanning tree algorithm, stopped early.

#### **Correct Response**

No, it can be viewed as *Kruskal's* minimum spanning tree algorithm, stopped early.



0/2 points

6.

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).

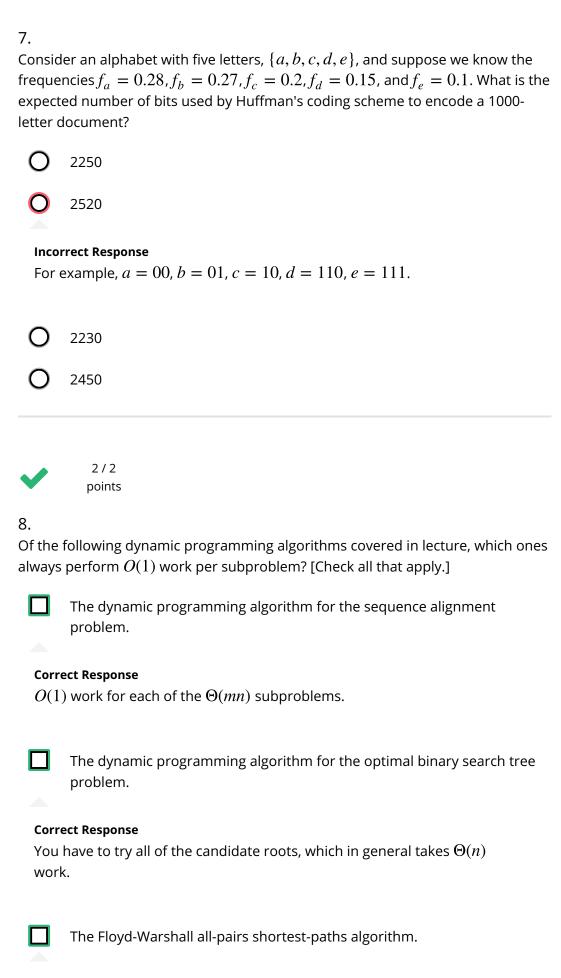


Schedule the requests in increasing order of processing time  $p_i$ 

#### **Incorrect Response**

What if one job has a small deadline, and all the others have huge deadlines?

0	Schedule the requests in increasing order of deadline $d_{\it j}$
0	Schedule the requests in increasing order of the product $d_j \cdot p_j$
0	None of the other options are correct



$O(1)$ work for each of the $\Theta(n^3)$ subproblems.				
The dynamic programming algorithm for the knapsack problem.				
Correct Response $O(1)$ work for each of the $\Theta(nW)$ or $\Theta(n^2v_{max})$ subproblems (depending on which Knapsack dynamic programming algorithm you're talking about).				
The Bellman-Ford shortest-path algorithm.				
Correct Response The work per subproblem is proportional to a node's in-degree, which can be as large as $\Theta(n)$ .				
2/2 points				
9. 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.				
Correct Response Our first dynamic programming algorithm for the Knapsack problem proves this. (Note one can assume that the capacity $W$ is less than the sum of the item sizes, otherwise the instance is trivial.)				
If there is a polynomial-time algorithm for the Knapsack problem in general, then P=NP.				
Correct Response Yes, the (decision version of) the Knapsack problem is NP-complete.				

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.

#### **Correct Response**

No, only when either the item values or the item sizes are polynomially bounded.

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.

# **Correct Response**

Our second dynamic programming algorithm for the Knapsack problem proves this.



2/2 points

10.

Assume that  $P \neq NP$ . 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.

#### **Correct Response**

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.

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.

Requires only minor modifications to the standard Knapsack dynamic programming algorithm.

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.

# **Correct Response**

Every straightforward dynamic programming approach has running time exponential in m. More generally, this problem is NP-hard even if all of the numbers are polynomially bounded (non-trivial exercise: can you prove this?).

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  $k \le n$  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.

#### **Correct Response**

Add another dimension to the array to keep track of how many items you've used so far, this increases the running time by a factor of at most n.



2/2 points

11.

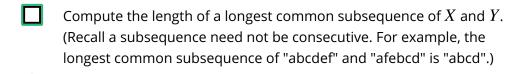
Assume that  $P \neq NP$ . The following problems all take as input two strings X and Y, of length m and n, over some alphabet  $\Sigma$ . Which of them can be solved in O(mn) time? [Check all that apply.]

Consider the following variation of sequence alignment. Instead of a single gap penalty  $\alpha_{gap}$ , you're given two numbers a and b. The penalty of inserting k gaps in a row is now defined as ak + b, rather than

 $k\alpha_{gap}$ . Other penalties (for matching two non-gaps) are defined as before. The goal is to compute the minimum-possible penalty of an alignment under this new cost model.

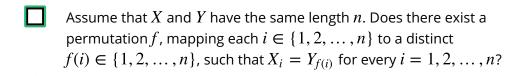
## **Correct Response**

Variation on the original sequence alignment dynamic program. With each subproblem, you need to keep track of what gaps you insert, since the costs you incur in the current position depend on whether or not the previous subproblems inserted gaps. Blows up the number of subproblems and running time by a constant factor.



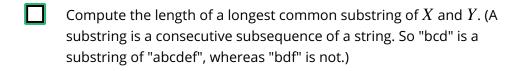
# **Correct Response**

Similar dynamic programming to sequence alignment, with one subproblem for each  $X_i$  and  $Y_j$ . Alternatively, this reduces to sequence alignment by setting the gap penalty to 1 and making the penalty of matching two different characters to be very large.



#### **Correct Response**

This problem can be solved in O(n) time, without dynamic programming. Just count the frequency of each symbol in each string. The permutation f exists if and only if every symbol occurs exactly the same number of times in each string.



#### **Correct Response**

Similar dynamic programming to sequence alignment, with one subproblem for each  $X_i$  and  $Y_j$ .

1	2.
Α	ssu

Assume that  $P \neq NP$ . Which of the following problems can be solved in polynomial time? [Check all that apply.]

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.,  $\max_{u,v \in V} \max_{P \in \mathcal{P}_{uv}} \sum_{e \in P} c_e$ , where  $\mathcal{P}_{uv}$  denotes the set of cycle-free u-v paths).

# **Incorrect Response**

The NP-complete Hamiltonian Path problem (recall PSet #6) reduces easily to this problem, so it cannot be solved in polynomial time assuming  $P \neq NP$ .

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).

# **Correct Response**

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

Given a directed acyclic graph with real-valued edge lengths, compute the length of a longest path between any pair of vertices.

#### **Incorrect Response**

By multiplying all edge lengths by -1, the problem reduces to computing the shortest path between any pair of vertices. Since the graph is acyclic, there are no negative-cost cycles, and this problem can be solved in polynomial time (e.g., via Floyd-Warshall or Johnson).

Given a directed graph with nonnegative edge lengths, compute the length of a longest cycle-free path between any pair of vertices (i.e.,  $\max_{u,v\in V}\max_{P\in\mathcal{P}_{uv}}\sum_{e\in P}c_e$ , where  $\mathcal{P}_{uv}$  denotes the set of cycle-free u-v paths).

# **Correct Response**

The NP-complete Hamiltonian Path problem (recall PSet #6) reduces easily to this problem, so it cannot be solved in polynomial time assuming  $P \neq NP$ .



2/2 points

## 13.

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. **Correct Response** As discussed in lecture. Johnson's reweighting algorithm. **Correct Response** As discussed in lecture. The Floyd-Warshall algorithm. **Correct Response** As discussed in lecture.

# **Correct Response**

vertex.

As discussed in lecture, Dijkstra's algorithm need not be correct when there are negative edge lengths, even when there is no negative-cost cycle.

Run Dijkstra's algorithm n times, once for each choice of a source

# 14.

Consider an instance of the optimal binary search tree problem with 7 keys (say 1,2,3,4,5,6,7 in sorted order) and frequencies

 $w_1 = .2, w_2 = .05, w_3 = .17, w_4 = .1, w_5 = .2, w_6 = .03, w_7 = .25$ . What is the minimum-possible average search time of a binary search tree with these keys?



2.33

#### **Incorrect Response**

The root is 5, with children 3 and 7, and grandchildren 1, 4, 6, and NULL respectively (2 is a child of 1).

- 2.29
- 2.23
- 2.18



2/2 points

#### 15.

Suppose a computational problem  $\Pi$  that you care about is NP-complete. Which of the following are true? [Check all that apply.]

You should not try to design an algorithm that is guaranteed to solve  $\Pi$  correctly and in polynomial time for every possible instance (unless you're explicitly trying to prove that P=NP).

#### **Correct Response**

And, moreover, I don't recommend spending too much time trying to prove that P=NP!

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  $\Pi$ .

Not true; 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).

If your boss criticizes you for failing to find a polynomial-time algorithm for  $\Pi$ , you can legitimately claim that thousands of other scientists (including Turing Award winners, etc.) have likewise tried and failed to solve  $\Pi$ .

# **Correct Response**

Remember, in trying to solve one NP-complete problem, you're trying to solve them all. Countless brilliant minds have tried to devise polynomial-time algorithms for NP-complete problems (and thus, indirectly, for your own NP-complete problem  $\Pi$ ); none have yet succeeded.

NP-completeness is a "death sentence"; you should not even try to solve the instances of  $\Pi$  that are relevant for your application.

# **Correct Response**

Not true; perhaps the instances of  $\Pi$  arising in your domain are special enough to be solved efficiently (in theory and/or in practice).



2/2 points

16.

Which of the following statements are logically consistent with our current state of knowledge (i.e., with the mathematical statements that have been formally proved)? [Check all that apply.]

There is an NP-complete problem that is polynomial-time solvable.

## **Correct Response**

Given what has been proved up to this point in time, P=NP remains a logical possibility.

There is an NP-complete problem that can be solved in  $O(n^{\log n})$  time, where n is the size of the input.

Given what has been proved up to this point in time, the running time required to solve NP-complete problems could be anywhere between polynomial and exponential (note that  $n^{\log n}$  is more than polynomial but less than exponential).

There is no NP-complete problem that can be solved in  $O(n^{\log n})$  time, where n is the size of the input.

# **Correct Response**

Given what has been proved up to this point in time, the running time required to solve NP-complete problems could be anywhere between polynomial and exponential (note that  $n^{\log n}$  is more than polynomial but less than exponential).

Some NP-complete problems are polynomial-time solvable, and some NP-complete problems are not polynomial-time solvable.

#### **Correct Response**

A polynomial-time algorithm for a single NP-complete automatically yields polynomial-time algorithms for all NP-complete algorithms (i.e., implies that P=NP).



2/2 points

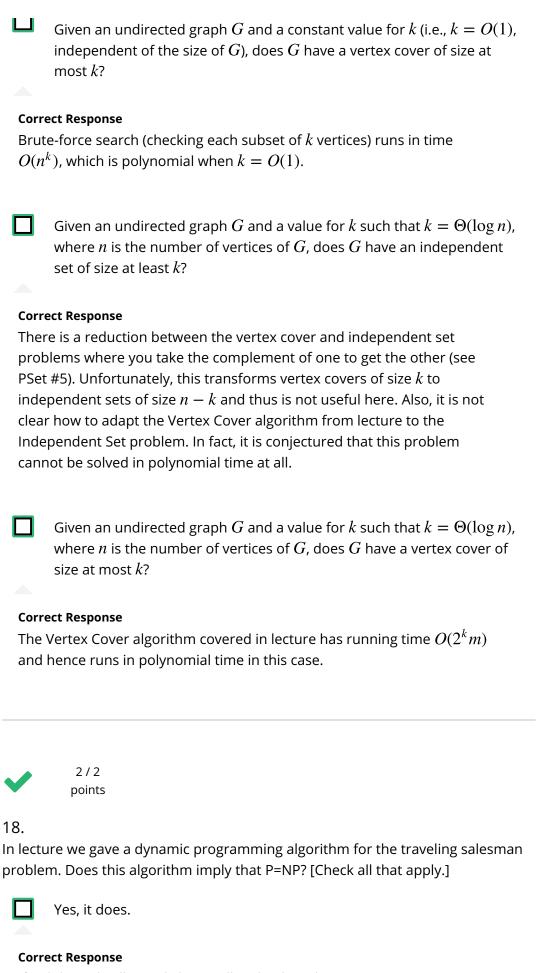
17.

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 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?

## **Correct Response**

Brute-force search (checking each subset of k vertices) runs in time  $O(n^k)$ , which is polynomial when k = O(1).

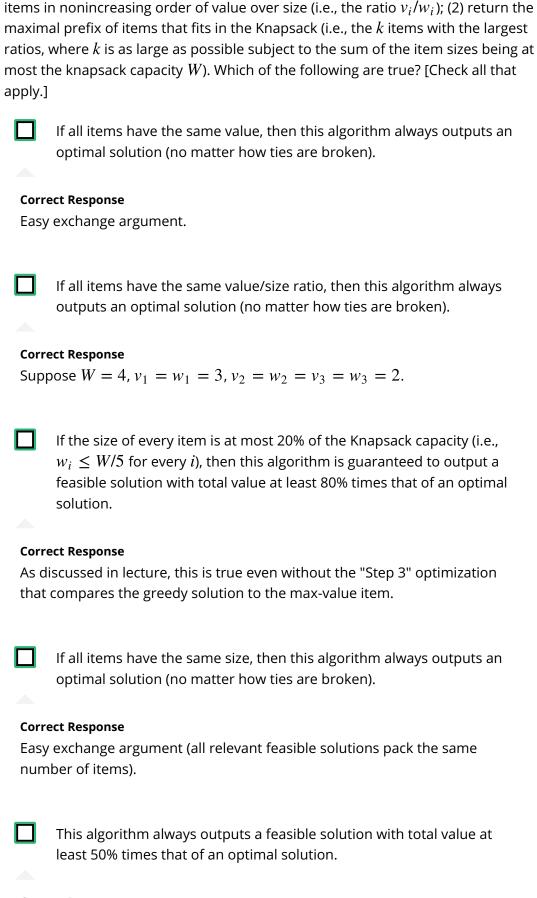


If it did, we'd collectively be a million bucks richer!

	No. Since there are an exponential number of subproblems in our dynamic programming formulation, the algorithm does not run in polynomial time.		
<b>Corre</b> Prec	ect Response isely.		
	No. A polynomial-time algorithm for the traveling salesman problem does not necessarily imply that P=NP.		
Corre	ect Response		
	e (the decision version of) the traveling salesman problem is NP- plete, a polynomial-time algorithm for TSP would indeed imply that P.		
	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.		
Corre	ect Response		
We only do $O(n)$ work computing the answer to a single subproblem; the issue is that there are exponentially many such subproblems.			
	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.		
Correct Response			
We only do $O(n)$ work computing the final answer, given the solutions of all of the (exponentially many) subproblems.			



2/2 points



Consider the Knapsack problem and the following greedy algorithm: (1) sort the

#### **Correct Response**

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).



2/2 points

## 20.

Which of the following statements are true about the generic local search algorithm? [Check all that apply.]

The generic local search algorithm is guaranteed to eventually converge to an optimal solution.

## **Correct Response**

No, only a *locally* optimal solution.

The output of the generic local search algorithm generally depends on the choice of the starting point.

## **Correct Response**

Yes, different initial choices can lead to different locally optimal solutions.

The generic local search algorithm is guaranteed to terminate in a polynomial number of iterations.

## **Correct Response**

No, in general it can require an exponential number of iterations.

The output of the generic local search algorithm generally depends on the choice of the superior neighboring solution to move to next (in an iteration where there are multiple such solutions).

# **Correct Response**

Yes, different choices for which neighboring solution to move to can lead to different locally optimal solutions.