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.

Correct Response

We 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.

Correct Response

The two-edge path graph provides a counterexample.

---

2 / 2
points

2.

You are given a connected undirected graph G with distinct edge costs, in adjacency list representation. You are also given the edges of a minimum spanning tree T of G. This question asks how quickly you can recompute the MST if we change the cost of a single edge. Which of the following are true? [RECALL: It is not known how to deterministically compute an MST from scratch in $O(m)$ time, where m is the number of edges of G.] [Check all that apply.]

- [ ] Suppose $e \in T$ and we increase the cost of e. Then, the new MST can be recomputed in $O(m)$ deterministic time.

Correct Response

Let A, B be the two connected components of $T - \{e\}$. Edge e no longer belongs to the new MST if and only if it is no longer the cheapest edge crossing the cut (A, B) (this can be checked in $O(m)$ time). If f is the new cheapest edge crossing (A, B), then the new MST is $T - \{e\} \cup \{f\}$.

- [ ] Suppose $e \notin T$ and we decrease the cost of e. Then, the new MST can be recomputed in $O(m)$ deterministic time.

Correct Response

Let C be the cycle of $T \cup \{e\}$. Edge e belongs to the new MST if and only 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.

Correct Response

The MST does not change (by the Cut Property), so no re-computation is needed.

Suppose $e \notin T$ and we increase the cost of e. Then, the new MST can be recomputed in $O(m)$ deterministic time.

Correct Response

The MST does not change (by the Cycle Property of the previous problem), so no re-computation is needed.

---

2 / 2
points

3.
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.

Correct Response

Just negate 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.

Correct Response

The optimal such set is the complement of a maximum-cost spanning tree.

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.

Correct Response

As discussed in the problem sets, every MST is also a minimum bottleneck spanning tree.

- [ ] The single-source shortest-path problem.

Correct Response

There is no obvious way of using a minimum spanning tree subroutine to solve the single-source shortest-path problem.

---

2 / 2
points

4.
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.

Correct Response

Our running time was only $O(mn)$.

- [ ] Prim's minimum spanning tree algorithm.

Correct Response

As covered in lecture, using the heap data structure.

- [ ] Kruskal's minimum spanning tree algorithm.

Correct Response

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

- [ ] Johnson's all-pairs shortest-path algorithm.

Correct Response

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

---

2 / 2

points

5.

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_j

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

- Schedule the requests in increasing order of deadline d_j
- Schedule the requests in increasing order of the product $d_j \cdot p_j$
- None of the other options are correct

---

0 / 2
points

7.

Consider an alphabet with five letters, $\{a, b, c, d, e\}$, and suppose we know the frequencies $f_a = 0.28, f_b = 0.27, f_c = 0.2, f_d = 0.15$, and $f_e = 0.1$. What is the expected number of bits used by Huffman's coding scheme to encode a 1000-letter document?

- 2250
- 2520

Incorrect Response

For example, $a = 00, b = 01, c = 10, d = 110, e = 111$.

- 2230
- 2450

---

2 / 2 points

8.

Of the following dynamic programming algorithms covered in lecture, which ones always perform $O(1)$ work per subproblem? [Check all that apply.]

- The dynamic programming algorithm for the sequence alignment problem.

Correct Response

$O(1)$ work for each of the $\Theta(mn)$ subproblems.

- The dynamic programming algorithm for the optimal binary search tree problem.

Correct Response

You have to try all of the candidate roots, which in general takes $\Theta(n)$ work.

- The Floyd-Warshall all-pairs shortest-paths algorithm.

Correct Response

$O(1)$ 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 $\Theta(n^2 v_{max})$ 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.

Correct Response

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 $k \leq n$ 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.

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 Σ. 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 α_{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.

- [ ] Compute the length of a longest common subsequence of X and Y. (Recall a subsequence need not be consecutive. For example, the longest common subsequence of "abcdef" and "afebcd" is "abcd".)

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.

- [ ] Assume that X and Y have the same length n. Does there exist a permutation f, mapping each $i \in \{1, 2, \ldots, n\}$ to a distinct $f(i) \in \{1, 2, \ldots, n\}$, such that $X_i = Y_{f(i)}$ for every $i = 1, 2, \ldots, n$?

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.

- [ ] Compute the length of a longest common substring of X and Y. (A substring is a consecutive subsequence of a string. So "bcd" is a substring of "abcdef", whereas "bdf" is not.)

Correct Response

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

1 / 2
points

12.
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.

- [ ] Run Dijkstra's algorithm n times, once for each choice of a source vertex.

Correct Response
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.

---

0 / 2
points

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 Π 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 Π 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 Π.

Correct Response

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 Π, you can legitimately claim that thousands of other scientists (including Turing Award winners, etc.) have likewise tried and failed to solve Π.

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 Π); none have yet succeeded.

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

Correct Response
Not true; perhaps the instances of Π 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.

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).

- [ ] 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 $k = O(1)$.

- [ ] 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?

Correct Response

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

- [ ] 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?

Correct Response

There is a reduction between the vertex cover and independent set problems where you take the complement of one to get the other (see PSet #5). 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.

- [ ] 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?

Correct Response

The Vertex Cover algorithm covered in lecture has running time $O(2^k m)$ and hence runs in polynomial time in this case.

---

2 / 2 points

18.

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.

Correct Response

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.

Correct Response

Precisely.

- [ ] No. A polynomial-time algorithm for the traveling salesman problem does not necessarily imply that P=NP.

Correct Response

Since (the decision version of) the traveling salesman problem is NP-complete, a polynomial-time algorithm for TSP would indeed 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.

Correct 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

19.

Consider the Knapsack problem and the following greedy algorithm: (1) sort the items in nonincreasing order of value over size (i.e., the ratio 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).

Correct Response
Easy exchange argument.

- [ ] If all items have the same value/size ratio, then this algorithm always outputs an optimal solution (no matter how ties are broken).

Correct Response
Suppose $W = 4, v_1 = w_1 = 3, v_2 = w_2 = v_3 = w_3 = 2$.

- [ ] If the size of every item is at most 20% of the Knapsack capacity (i.e., $w_i \leq W/5$ 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.

Correct Response
As discussed in lecture, this is true even without the "Step 3" optimization that compares the greedy solution to the max-value item.

- [ ] If all items have the same size, then this algorithm always outputs an optimal solution (no matter how ties are broken).

Correct Response
Easy exchange argument (all relevant feasible solutions pack the same number of items).

- [ ] This algorithm always outputs a feasible solution with total value at least 50% times that of an optimal solution.

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.