CS 170 Algorithms Fall 2014 David Wagner MT1 Soln

I include explanations for many of the answers, to help you understand why they are correct. You did not need to include any of these explanations in your answer.

Problem 1. [True or false] (9 points)

Circle TRUE or FALSE. Do not justify your answers on this problem.

(a) [TRUE] or FALSE: A connected undirected graph is guaranteed to have at least $|V| - 1$ edges.

(b) [TRUE] or FALSE: A strongly connected directed graph is guaranteed to have at least $|V| - 1$ edges.

(c) TRUE or [FALSE]: In a DAG, the number of distinct paths between two vertices is at most $|V|^2$.

Explanation: there could be as many as $2^{|V|-2}$ paths.

(d) [TRUE] or FALSE: Every DAG has at least one source.

Explanation: Start at an arbitrary vertex v, and repeatedly go backwards in the graph. If there is no source, you can keep going forever. But that means some vertex will be visited twice, and in particular, you'll have found a cycle in the graph—so it couldn't have been a dag in the first place.

(e) [TRUE] or FALSE: Depth-first search on a connected undirected graph G will visit all of the vertices of G.

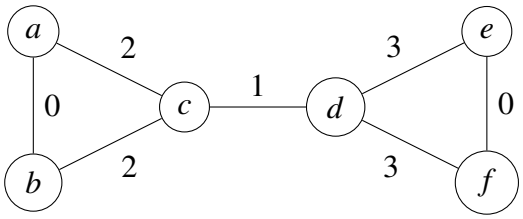
Explanation: No matter what vertex v you start at, every vertex will be reachable from v (since G is connected), so all vertices will be visited. (Also, DFS restarts the search and calls Explore on any unvisited vertices, so this would be true even for a graph that is not connected.)

(f) TRUE or [FALSE]: After running depth-first search on a directed graph, the node with the smallest post number is part of a source component.

Explanation: The vertex with *largest* post number is part of a source component.

(g) TRUE or [FALSE]: Suppose we have a graph where each edge weight value appears at most twice. Then, there are at most two minimum spanning trees in this graph.

Explanation: There are 4 MSTs in the following graph:



Every MST includes $a-b$, $e-f$, and $c-d$. Now you can choose either $a-c$ or $b-c$ (2 choices), and independently choose either $d-e$ or $d-f$ (2 choices), for $2 \times 2 = 4$ possible MSTs in all.

(h) TRUE or FALSE: If $f(n) = O(n^2)$ and $g(n) = O(n^2)$, then $f(n) = O(g(n))$.

Explanation: consider, e.g., $f(n) = n^2$ and $g(n) = n$.

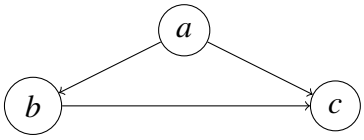
(i) TRUE or FALSE: If $f(n) = O(g(n))$ and $g(n) = O(n^2)$, then $f(n) = O(n^2)$.

(j) TRUE or FALSE: Suppose we run DFS on an undirected graph and find exactly 17 back edges. Then the graph is guaranteed to have at least one cycle.

Explanation: If the graph has a back edge, then it has a cycle.

(k) TRUE or FALSE: DFS on a directed graph with n vertices and at least n edges is guaranteed to find at least one back edge.

Explanation: DFS on the following graph starting at a does not find any back edge.



(l) TRUE or FALSE: DFS on an undirected graph with n vertices and at least n edges is guaranteed to find at least one back edge.

Explanation: A tree has $n-1$ edges. Any graph with more edges than that must have a cycle. If there's a cycle, then DFS will find a back edge.

(m) TRUE or FALSE: Suppose we run DFS on an undirected graph, and we discover a vertex v with $\text{pre}(v) = 1$ and $\text{post}(v) = 2|V|$. Then the graph must be connected.

Explanation: This vertex was the first to be visited and the last to finish, so all other vertices were visited before this vertex finished. That means all other vertices are reachable from this vertex.

(n) TRUE or FALSE: Suppose we run DFS on a directed graph, and we discover a vertex v with $\text{pre}(v) = 1$ and $\text{post}(v) = 2|V|$. Then the graph must be strongly connected.

Explanation: Consider the graph shown in part (k).

(o) TRUE or [FALSE]: If the expected running time of an algorithm is $O(n)$, then its worst-case running time is also $O(n)$.

Explanation: Consider, for example, the randomized algorithm for finding the median in an unsorted list. Its expected running time is $O(n)$, but its worst-case running time is $O(n^2)$.

(p) [TRUE] or FALSE: There is an algorithm to multiply two n-bit numbers in $O(n^{\log_2 3})$ time.

Explanation: Gauss's trick (the divide-and-conquer algorithm).

(q) [TRUE] or FALSE: There is an algorithm to square a n-bit number in $O(n^{\log_2 3})$ time.

(r) [TRUE] or FALSE: If we had an algorithm to square a n-bit number in $O(n)$ time, we could multiply two n-bit numbers in $O(n)$ time.

Explanation: $xy = [(x+y)^2 - x^2 - y^2]/2$. The right-hand side can be computed using three squarings ($O(n)$ time), two subtractions ($O(n)$ time), and one right-shift ($O(n)$ time), for a total of $O(n)$ time.

Alternatively, $xy = [(x+y)^2 - (x-y)^2]/4$. Or, square $z = 2^{2n+1}x + y$; you can find the value of xy in the middle bits of z^2.

Problem 2. [Solving recurrences] (8 points)

You don't need to justify your answer or show your work on this problem. Express your answer using $\Theta(\cdot)$ notation.

(a) What's the solution to the recurrence $T(n) = 2T(n/2) + n$?

Solution: $T(n) = \Theta(n \lg n)$.

Explanation: use the Master theorem. Or, recognize this as the recurrence for MergeSort.

(b) What's the solution to the recurrence $U(n) = 2U(n/2) + n \lg n$?

Solution: $T(n) = \Theta(n(\lg n)^2)$.

Explanation: Use the recursion tree method. The recursion tree has depth $\lg n$ and $2^{\lg n} = n$ leaves. The extra stuff at level ℓ is $n \lg(n/2^\ell) = n \lg n - \ell n$. For an upper bound, the extra stuff at each level is $\leq n \lg n$, so the sum of all the extra stuff is $\leq n(\lg n)^2$. For a lower bound, sum the extra stuff at levels $\ell = 0, 1, 2, \ldots, 0.5 \lg n$; you find that the extra stuff at each such level is $\geq 0.5 n \lg n$, so the sum of the extra stuff at all levels is $\geq (0.5 \lg n) \times (0.5 n \lg n) = 0.25 n (\lg n)^2$. Therefore the sum of the extra stuff is $\Theta(n(\lg n)^2)$, and the total running time is $n + \Theta(n(\lg n)^2) = \Theta(n(\lg n)^2)$.

Alternatively, you can sum the extra stuff directly:

$$\sum_{\ell=0}^{\lg n} n \lg n - \ell n = n(\lg n)^2 - n \sum_{\ell=0}^{\lg n} \ell = n(\lg n)^2 - n(\lg n)(1 + \lg n)/2 = 0.5 n (\lg n)^2 - 0.5 n \lg n,$$

so the sum of the extra stuff is $\Theta(n(\lg n)^2)$.

Or, recognize this recurrence as the recurrence for the divide-and-conquer algorithm presented in lecture for finding the closest two points in a 2D plane, and recall the solution from lecture.

Or, in a pinch, look at the recurrences and guess that $U(n) \approx T(n) \lg n$. (This is not mathematically valid in general, but if you're at a total loss, there are worse ways to make an informed guess.)

(c) What's the solution to the recurrence $F(n) = F(n/2) + \Theta(n)$?

Solution: $T(n) = \Theta(n)$.

Explanation: This is a geometric sum. $n + n/2 + n/4 + n/8 + \cdots = \Theta(n)$. Or, use the Master theorem.

(d) What's the solution to the recurrence $G(n) = 0.5G(n-2) + \Theta(1)$?

Solution: $T(n) = \Theta(1)$.

Explanation: This is a geometric sum. $1 + 0.5 + 0.25 + 0.125 + \cdots = \Theta(1)$

Problem 3. [Fast Fourier Transform] (4 points)

Suppose we would like to evaluate a polynomial $p(x)$ on n distinct values $x_1, x_2, \ldots, x_n$, where n is a power of two. What values of $x_1, \ldots, x_n$ should we choose, so that we can use the FFT for this purpose? Does it matter which values we pick?

Solution: Use the nth roots of unity. Yes, it matters.

Explanation: we need the x-values to be paired at each level of the recursion, so you can't use just any x-values. The roots of unity satisfy this requirement.

Problem 4. [Short answer] (6 points)

Answer each question with "Yes" or "No". If you answer Yes, give a brief justification (one sentence). If you answer No, draw a small counterexample.

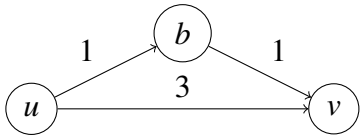
(a) Suppose G is a connected, undirected graph whose edges all have positive weight. Let M be a minimum spanning tree of this graph. Now, we modify the graph by adding 7 to the weight of each edge. Is M guaranteed to be a minimum spanning tree of the modified graph?

Solution: Yes. This modification adds $7(|V| - 1)$ to the total weight of all spanning trees, so it doesn't change which one is minimal.

Alternative solution: Yes. Kruskal's algorithm only cares about the order of the edge weights, not their absolute values, and adding 7 to each will not change the sorted order. (Technically, this is not a valid justification, as we have not proved that every possible minimum spanning tree could be output by Kruskal's algorithm. However, it is not a bad way to get intuition, if you couldn't find a more rigorous justification.)

(b) Suppose G is an undirected graph whose edges all have positive length. Let P be a shortest path from u to v. Now, we modify the graph by adding 7 to the weight of each edge. Is P guaranteed to be a shortest path from u to v in the modified graph?

Solution: No.



Explanation: $u \to b \to v$ is the shortest path in the graph above. However, if we add 7 to each edge, the shortest path changes to $u \to v$.

Problem 5. [Running time analysis] (~~12~~ 13 points)

You don't need to justify your answers or show your work on this problem.

Given two 64-bit integers a, n, here is an algorithm to compute a^n:

Power(a,n):
1. If $n = 0$: return 1.
2. Return $a \times$ Power$(a, n-1)$.

Assume throughout this problem that we don't need to worry about overflow (a^n fits into a 64-bit integer variable) and that each operation on a 64-bit integer takes $O(1)$ time.

(a) Let $T(n)$ denote the running time of Power(a,n). Write a recurrence relation for $T(n)$.

Solution: $T(n) = T(n-1) + \Theta(1)$.

(b) What is the solution to your recurrence from part (a)? Use $\Theta(\cdot)$ notation.

Solution: $T(n) = \Theta(n)$.

You are now given another algorithm for the same problem:

AltPower(a,n):
1. If $n = 0$: return 1.
2. If $n = 1$: return a.
3. If n is even:
4. Return AltPower$(a \times a, n/2)$.
5. else:
6. Return $a \times$ AltPower$(a \times a, (n-1)/2)$.

(c) Let $A(n)$ denote the running time of AltPower(a,n). Write a recurrence relation for $A(n)$.

Solution: $T(n) = T(n/2) + \Theta(1)$.

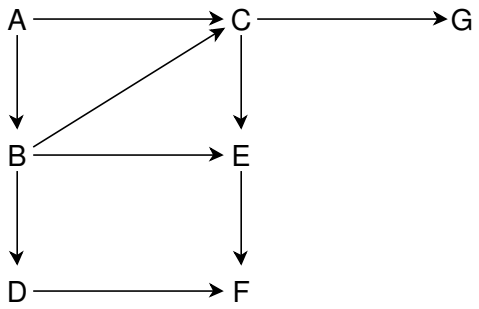
(d) What is the solution to your recurrence from part (c)? Use $\Theta(\cdot)$ notation.

Solution: $T(n) = \Theta(\lg n)$.

(e) Which would you expect to be faster, AltPower or Power?

Solution: AltPower.

Problem 6. [DFS] (8 points)



Perform a depth-first search on the graph above, starting from vertex A. Whenever there's a choice of vertices, pick the one that is alphabetically first.

(a) Fill in the table below with the pre and post number of each vertex.

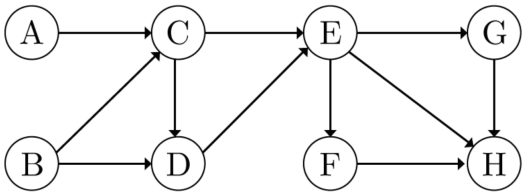
	A	B	C	D	E	F	G
pre	1	2	3	11	4	5	8
post	14	13	10	12	7	6	9

(b) Next, label each edge in the graph above as a tree, back, forward, or cross edge.

Solution: $A \to C$ and $B \to E$ are forward edges. $D \to F$ is a cross edge. All other edges are tree edges. There are no back edges.

Problem 7. [Topological sorting] (~~6~~ 7 points)

Consider the following graph:



(a) Which of the following orderings is not a valid topological sort of the graph?

(i) ABCDEFGH

(ii) ABCDEGFH

(iii) BACDEFGH

(iv) BACEDFGH

Explanation: there's an edge $D \to E$, so D has to come before E in any valid topological sort.

(b) If we run DFS on this graph, which of the following statements must be true? Circle all that must be true.

(i) $\boxed{\text{post(B)} > \text{post(D)}}$

(ii) post(G) > post(F)

(iii) post(A) > post(B)

(iv) $\boxed{\text{post(D)} > \text{post(F)}}$

Problem 8. [Shortest paths] (8 points)

Let $G = (V, E)$ be a directed graph, with non-negative edge lengths; $\ell(e)$ denotes the length of edge e.

(a) Suppose we run Dijkstra's algorithm starting from s. After it finishes, is it guaranteed that dist(t) will hold the length of the shortest path from s to t? Write "yes" or "no". Do not justify your answer.

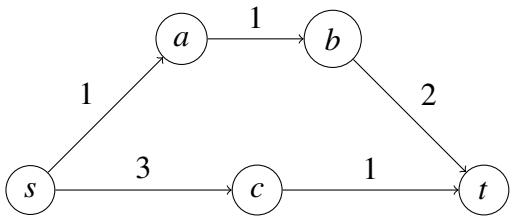
Solution: Yes.

(b) If we run Dijkstra's algorithm starting from s, then after it finishes we can use the prev$(\cdot)$ pointers to find a shortest path from s to t. Concisely explain how.

Solution: $s \to \cdots \to \text{prev}(\text{prev}(\text{prev}(t))) \to \text{prev}(\text{prev}(t)) \to \text{prev}(t) \to t$

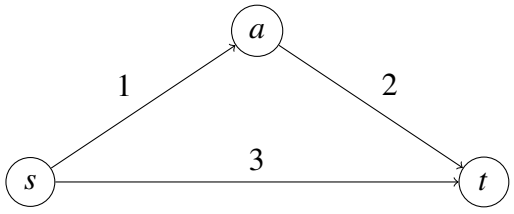
(c) Is it guaranteed that the path you constructed in part (b) will have the fewest number of edges, out of all paths from s to t of length dist(t)? If yes, write "yes" and explain why briefly. If no, write "no" and show a small counterexample.

Solution: No.



Explanation: This counterexample was taken directly from the solution to HW6 Q3 (Travel planning). Dijkstra's algorithm will return the path $s \to a \to b \to t$, but the one with the smallest number of edges is just $s \to c \to t$.

Comment: Be careful. The following graph is not a valid counterexample, at least not for the version of Dijkstra's algorithm shown in the book:

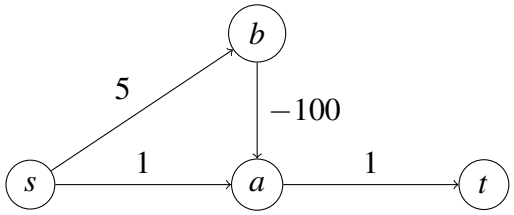


The version of Dijkstra's algorithm shown in the book returns the path $s \to t$, which is the one with the fewest number of edges. It visits the vertices in the order s, a, t. When visiting s, it sets $d(t)$ to 3 and prev(t) to s. When visiting a, it computes $d(a) + \ell(a,t) = 1 + 2 = 3$, notices that this is not smaller than $d(t)$, and does not update $d(t)$ or prev(t). Thus it terminates with prev$(t) = s$, corresponding to the path $s \to t$.

This would be a valid counterexample for a variant of Dijkstra's algorithm where we change the condition $d(u) + \ell(u,v) < d(v)$ to $d(u) + \ell(u,v) \leq d(v)$, but that feels a bit artificial (there's no clear reason to make this change), and it doesn't match what is in the book.

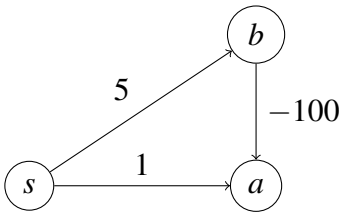
(d) Now suppose that exactly *one* edge of G has a negative length, and all other edges have non-negative length, and there is no negative cycle in the graph. Suppose we run Dijkstra's algorithm starting from s. After it finishes, is it guaranteed that dist(t) will hold the length of the shortest path from s to t? If yes, write "yes" and explain why. If no, write "no" and show a small counterexample.

Solution: No.



Explanation: Dijkstra's algorithm will visit the vertices in the order s, a, t, b. After it has visited t, it has set $d(s) = 0$, $d(a) = 1$, $d(b) = 5$, $d(t) = 2$. When it visits b, it updates $d(a)$ to -95, which is lower than before—but having already visited a once, it does not schedule a to be re-visited again. Consequently, Dijkstra's algorithm outputs $d(t) = 2$, when the correct distance is -94.

Comment: Be careful. Smaller graphs probably won't form a valid counterexample. For instance, Dijkstra's algorithm does happen to compute the correct distances for the following graph, even though it has a negative edge:



Dijkstra's algorithm visits the vertices in the order s, a, b. After visiting a, the distances are $d(s) = 0$, $d(a) = 1$, $d(b) = 5$. When visiting b, Dijkstra's algorithm updates $d(a)$ to -95, and stops, so all vertices receive the correct distance for this graph. That said, this *would* be a valid counterexample for a variant of Dijkstra's algorithm that only considers edges (u,v) where $v \notin R$ (i.e., when visiting u, we iterate over all edges (u,v) out of u, but skip over it if v has been previously visited).

Problem 9. [All paths go through...] (8 points)

Google Maps wants to add a new feature: include a stop to Disneyworld in your route. In particular, they are looking for an algorithm for the following problem:

Input: a directed graph $G = (V, E)$, with a positive length $\ell(e)$ on each edge e; vertices s, w, t

Output: the length of the shortest path from s to t that goes through w.

They propose the following algorithm:

1. Call Dijkstra(G, ℓ, w), to get $d(w, v)$ for each $v \in V$.
2. Reverse the direction of all the edges; call the result G^r.
3. Call Dijkstra(G^r, ℓ, w), to get $d(v, w)$ for each $v \in V$.
4. Return $d(s, w) + d(w, t)$.

(a) Is their algorithm correct? If yes, write "yes" and explain why in a sentence or two. If no, write "no" and show a small counterexample.

Solution: Yes, a shortest path from s to t going through w is the concatenation of a shortest path from s to w and a shortest path from w to t.

Alternative solution: Yes, this is basically the same as HW6 Q4 (Road network design), but here we have a single new road of length 0.

(b) What is the asymptotic running time of their algorithm? Use $\Theta(\cdot)$ notation.

Solution: $\Theta((|V| + |E|) \lg |V|)$.

(c) Suppose G is a dag. Is G^r guaranteed to be a dag? Yes or no. Don't justify your answer.

Solution: Yes.

Explanation: Any cycle in G^r would correspond to a cycle in G (just going in the opposite direction). But G is a dag and does not have any cycles—so neither does G^r.

(d) Suppose G is a dag. If we replace both calls to Dijkstra's algorithm with calls to the algorithm for computing shortest paths in a dag, will the modified algorithm be correct? Yes or no. Don't justify your answer.

Solution: Yes.

(e) What is the asymptotic running time of the modified algorithm from part (d)? Use $\Theta(\cdot)$ notation.

Solution: $\Theta(|V| + |E|)$.

Problem 10. [Graph subsets] (~~7~~ 8 points)

Let $G = (V, E)$ be a connected, undirected graph, with edge weights $w(e)$ on each edge e. Some edge weights *might be negative*. We want to find a subset of edges $E' \subseteq E$ such that $G' = (V, E')$ is connected, and the sum of the weights of the edges in E' is as small as possible subject to the requirement that $G' = (V, E')$ be connected.

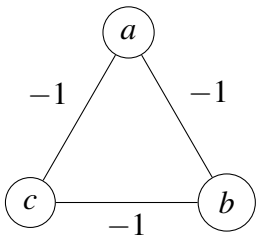
(a) Is it guaranteed that the optimal solution E' to this problem will always form a tree? Write yes or no. Don't justify your answer.

Solution: No.

Explanation: The optimal solution for the graph shown in part (b) is to select all of the edges. That has a cycle, so it isn't a tree.

(b) Does Kruskal's algorithm solve this problem? If yes, explain why in a sentence or two; if no, give a small counterexample.

Solution: No.



(c) Describe an efficient algorithm for this problem. Be concise. You should be able to describe your algorithm in one or two sentences. (You don't need to prove your algorithm correct, justify it, show pseudocode, or analyze its running time.)

Algorithm #1: Run Kruskal's algorithm, then add all the negative edges.

Algorithm #2: Select all the negative edges (and call Union for each to form connected components), then run Kruskal's algorithm from there.

Either answer is valid.

Explanation: The optimal solution surely includes all negative edges (you can always add any missing negative edge to any solution, and it will remain a valid solution). The negative edges create a set of connected components. Any valid solution needs to connect all of them. If you shrink each connected component to a single meta-node, then this is exactly the minimum spanning tree problem, so Kruskal's algorithm finds the cheapest way to connect them. This proves that Algorithm #2 is correct.

We can prove that Algorithm 1 outputs the same set of edges as Algorithm 2, so it is correct, too. In particular, if Algorithm 1 includes an edge $e = (u,v)$ with non-negative weight, then Algorithm 2 will include it too. If $e = (u,v)$ is selected by Kruskal's algorithm, then the set of edges that are lighter than e must not be enough to connect u to v; but that set includes all of the negative-weight edges, which are all considered before Kruskal's algorithm considers e, so u won't be connected to v when e is considered in Algorithm 2, and thus Algorithm 2 will select e too. This means that the set of non-negative edges selected by Algorithm 1 is a subset of the set of non-negative edges selected by Algorithm 2, so the total weight of the solution produced by Algorithm 1 must be at most the total weight of the solution produced by Algorithm 2. Since we know Algorithm 2 outputs an optimal solution, so too must Algorithm 1: in fact they select exactly the same set of edges.

You did not need to provide any of this explanation, just the algorithm.

Problem 11. [Graphs and Reductions] (~~6~~ 8 points)

If S is a set of vertices in an undirected graph $G = (V, E)$, define $f(S)$ to be the length of the shortest edge between a vertex in S and a vertex not in S, i.e.,

$$f(S) = \min\{\ell(v,w) : v \in S, w \notin S, \{v,w\} \in E\}.$$

We'd like an algorithm for the following problem, with running time $O((|V|+|E|)\log|V|)$ or less:

Input: a connected, undirected graph $G = (V, E)$, with a non-negative length $\ell(e)$ on each edge e.

Output: a non-empty set S that makes $f(S)$ as large as possible, subject to the requirement that $S \neq V$.

We can solve this problem by making a small change to one of the graph algorithms we've seen in this class. Which algorithm?

Solution: Kruskal's algorithm. (Or: Prim's algorithm.)

What's the small change? Answer concisely (one sentence).

Solution: Stop it just before Kruskal's algorithm selects the last edge; this partitions the graph into two components, so let S be one of those components.

Alternative solution: Delete the last edge Kruskal's selected, and let S be one of the two components that remains.

Alternative solution: Run Prim's algorithm to get a MST, then delete the largest edge in that tree, dividing the graph into two components, and let S be one of those two components.

Explanation: Let M be a minimum spanning tree for G. Consider any cut $(S, V-S)$. Since M is a spanning tree, there is at least one edge from M that crosses the cut $(S, V-S)$; let e_S be the shortest such, i.e., the shortest edge in M that crosses the cut $(S, V-S)$. Then $f(S) = \ell(e_S)$. (Why? Clearly $f(S) \leq \ell(e_S)$. If $f(S) < \ell(e_S)$, then there is some other edge e' that crosses the cut and is shorter than e, but isn't in M; but the cut property says this is impossible, as $M - e + e'$ would be a spanning tree whose total weight is even less than M.)

This means that if we want to maximize $f(S)$, we need to choose a cut $(S, V-S)$ that maximizes $\ell(e_S)$. One way to do this is to pick the longest edge e in M, and use it to select a cut $(S, V-S)$ such that e is the only edge crossing this cut. This is exactly what the solutions above do.

You didn't need to provide any explanation.

Comment: Running Prim's algorithm and stopping it just before it selects the last edge does not work. Prim's algorithm does not necessarily select edges in order of increasing weight; the last edge isn't necessarily the heaviest edge in the MST.

Comment: This problem shows how to separate the vertices into two clusters that are "as far apart from each other" as possible, for an appropriate definition of "far apart." This algorithm is sometimes used in practice for clustering: we let $\ell(u,v)$ be some measure of dissimilarity between items u, v, form a graph, and run the algorithm above to separate the vertices into two well-separated clusters.

Problem 12. [Algorithm design] (~~11~~ 13 points)

We are given an array $A[0..n-1]$, where $n > 1$ and all array elements are non-negative integers. Our goal is to find the maximum value of $A[i] + A[j]^2$, where the indices i, j range over all values such that $0 \le i < j < n$.

Fill in the blanks below to produce an efficient algorithm that correctly solves this problem.

FindMax($A[0..n-1]$):
1. If $n \le 1$, return $-\infty$.
2. Let $k := \lfloor n/2 \rfloor$.
3. Set $x :=$ FindMax($\boxed{A[0..k-1]}$).
4. Set $y :=$ $\boxed{\text{FindMax}(A[k..n-1])}$.
5. Set $z := \max(A[0], \ldots, A[k-1]) +$ $\boxed{\max(A[k]^2, \ldots, A[n-1]^2)}$.
6. Return $\max(x, y, z)$.

(a) Write a recurrence relation for the running time of your algorithm.

Solution: $T(n) = 2T(n/2) + \Theta(n)$.

(b) What is the asymptotic running time of your algorithm? Use $\Theta(\cdot)$ notation. You don't need to justify your answer.

Solution: $T(n) = \Theta(n \lg n)$.

(You do not need to prove your algorithm correct.)

(c) Describe a $O(n)$ time algorithm for this problem. (No proof of correctness or justification of running time needed.)

Algorithm #1:

Main idea: Modify FindMax to also return the maximum of all of the elements provided to it; then step 5 can be implemented in $\Theta(1)$ time.

Pseudocode:

FindMax($A[0..n-1]$):
1. If $n \le 0$, return $(-\infty, -\infty)$. If $n = 1$, return $(-\infty, A[0])$.
2. Let $k := \lfloor n/2 \rfloor$.
3. Set $(x, m_1) :=$ FindMax($A[0..k-1]$).
4. Set $(y, m_2) :=$ FindMax($A[k..n-1]$).
5. Set $z := m_1 + m_2^2$.
6. Return $(\max(x, y, z), \max(m_1, m_2))$.

Algorithm #2:

Main idea: Use a linear scan over j. For each j, compute $M[j] = \max(A[0], \ldots, A[j-1])$ (which can be computed easily from $M[j-1]$). The answer is $\max\{M[j] + A[j]^2\}$.

Pseudocode:

FindMax($A[0..n-1]$):
1. If $n \leq 1$, return $-\infty$.
2. Set $M[1] := A[0]$.
3. For $j := 2, 3, \ldots, n-1$: set $M[j] := \max(M[j-1], A[j])$.
4. Return $\max\{M[j] + A[j]^2 : 1 \leq j < n\}$.

Algorithm #3:

Main idea: Greedily choose j that maximizes $A[j]^2$ (resolving ties in favor of larger values of j), then choose i to maximize $A[i]$ subject to the requirement $i < j$.

Pseudocode:

FindMax($A[0..n-1]$):
1. If $n \leq 1$, return $-\infty$.
2. Set $j := 1$.
3. For $k := 2, 3, \ldots, n-1$: If $A[k] \geq A[j]$, set $j := k$.
4. Return $\max(A[0], A[1], \ldots, A[j-1]) + A[j]^2$.

Explanation: The proof of correctness for Algorithm 3 is tricky. Clearly if all array elements are zero, this algorithm is correct, so we can assume the maximum element of the array is at least 1. Let j_0 be the value computed in lines 2–3 of this algorithm (i.e., it maximizes $A[j_0]$, resolving ties in favor of larger indices), and i_0 be the index of the largest of $A[0], A[1], \ldots, A[j_0 - 1]$. Could $A[i] + A[j]^2$ be larger than $A[i_0] + A[j_0]^2$? We argue next that it cannot. There are three cases:

- If $j < j_0$, then $A[i] \leq A[i_0]$ (since $i < j < j_0$, so $A[i]$ is one of $A[0], A[1], \ldots, A[j_0 - 1]$, and $A[i_0]$ is the maximum of those values). Also $A[j] \leq A[j_0]$ (since $A[j_0]$ is the maximum), so we find $A[i] + A[j]^2 \leq A[i_0] + A[j_0]^2$.
- If $j = j_0$, then $A[i] \leq A[i_0]$ (since $i < j = j_0$, so $A[i]$ is one of $A[0], A[1], \ldots, A[j_0 - 1]$) and $A[j] = A[j_0]$, so we find $A[i] + A[j]^2 \leq A[i_0] + A[j_0]^2$.
- If $j > j_0$, then $A[i] \leq A[j_0]$ (since $A[j_0]$ is the maximum) and $A[j] < A[j_0]$ (since ties were resolved by choosing the largest possible value of j_0, so all of the elements $A[j_0 + 1], A[j_0 + 2], \ldots, A[n-1]$ are strictly smaller than $A[j_0]$). In particular, $A[j] \leq A[j_0] - 1$ (since the array elements are integers). Now when $x \geq 1$, $x + (x-1)^2 \leq 0 + x^2$. Plugging in $x = A[j_0]$ and using the facts inferred above, we find

$$A[i] + A[j] \leq A[j_0] + (A[j_0] - 1)^2 \leq x + (x-1)^2 \leq 0 + x^2 \leq A[i_0] + A[j_0]^2.$$

In no case can $A[i] + A[j]^2$ be larger than $A[i_0] + A[j_0]^2$. Since i, j were arbitrary, this means that $A[i_0] + A[j_0]^2$ is indeed the maximum possible value for $A[i] + A[j]^2$. Therefore Algorithm 3 is correct.

You might notice that when greedily choosing j, it is essential to resolve ties in favor of larger values of j. If we don't, we might get the wrong answer: for instance, on array $[0, 2, 1, 2]$, the optimal answer is $2 + 2^2$, but if we resolve the tie by choosing $j = 1$, we produce a non-optimal answer $0 + 2^2$.

You might also notice that the proof of correctness breaks down when the array elements might include negative numbers, or non-integers. That's because Algorithm 3 isn't correct for those kinds of inputs. For instance, Algorithm 3 outputs the wrong answer on the array $[-100, 2, 1]$: it outputs $-100 + 2^2$, but the correct answer is $2 + 1^2$. As another example, Algorithm 3 outputs the wrong answer on the array $[0, 2, 1.9]$: it outputs $0 + 2^2 = 4$, but the correct answer is $2 + 1.9^2 = 5.61$. Therefore, the restriction to non-negative integers is critical to the correctness of Algorithm 3.

In contrast, Algorithms 1 and 2 do work correctly when the array elements are arbitrary real numbers. (If some array elements might be negative, Algorithm 1 needs a small tweak: each recursive call should return not only the maximum of the elements provided to it, but also the maximum of the square of the elements.)

Of course, you did not need to show any justification or explanation of why your algorithm is correct.

We were strict about grading of 12(c). Your algorithm needed to be correct to get credit. We did not give partial credit for algorithms that were less than fully correct. The two exceptions where we gave partial credit were where your algorithm was correct apart from one of two specific minor errors: failure to handle the special case where $A[0]$ is the largest array element, or failure to handle ties correctly. Answers with more serious problems—e.g., did not work correctly for some inputs, did not run in $O(n)$ time—generally received no credit.

If you are not sure what was wrong with your algorithm, I recommend that you implement it in Python and implement a known-good algorithm (e.g., one of the algorithms above). Then, run it on a million randomly generated test cases: generate random arrays of length 4 or 5, with each element being a randomly chosen number from some range (say, 0..1000), and compare the output of your algorithm to that of the known-good algorithm. If you think your algorithm is correct but you received no credit, before filing a regrade request we encourage you to try this simple test to confirm your impression.

A common error was to write an algorithm that does not respect the requirement that $i < j$. For instance, one common approach was to let j be the index of the largest element in A and i be the index of the second-largest element. This does not work, because it does not ensure that $i < j$. Consider, for example, the input array $[1, 10, 5]$; this algorithm would output $5 + 10^2$, but the correct answer is $1 + 10^2$. No credit was provided for this sort of answer.