Final Exam Solutions

Math 182, Fall 2021

INSTRUCTIONS: Answer each question in the space provided. If you run out of room, use the blank pages at the end. You may consult two double-sided pages of notes. You may not use a calculator, phone or computer. If you believe there is a typo in a question, you may raise your hand to ask about it.

If you cannot figure out how to solve a problem, you may receive partial credit for solving an easier version of the same problem, as long as you state clearly that that's what you are doing.

ADVICE: I recommend that early on in the exam, you skim all the questions. If you find yourself stuck on a question for a long time, try switching to a different question. For the algorithm design questions, you will receive most of the points as long as your main idea is correct, even if your pseudocode is missing or incorrect. So it may be a good idea to try to first solve and write main ideas for the algorithm design questions and only start writing pseudocode after you have solved or gotten stuck on all of them. And if you ever find yourself getting panicked or having trouble thinking productively about a question, take a moment to take a deep breath and relax.

Name: _____

UID: _____

Question	Points	Score
1	15	
2	2	
3	3	
4	10	
5	10	
6	12	
7	12	
8	12	
9	12	
10	12	
11	0	
12	0	
Total:	100	

Don't turn over this page until you are told to do so.

1 Short Answer

The following three questions are all True/False or short answer. No justification is required.

Question 1: True/False (15 points)

(a) For any function $f\colon \mathbb{N} \to \mathbb{N}$, $f(n) \in O(f(n)^2)$.

√ **True** ◯ False

Solution: Since f takes values only in the natural numbers, $f(n) \leq f(n)^2$ for any n.

Common Mistakes: Some people thought the codomain of f was the positive real numbers rather than the natural numbers and came up with counterexamples to the statement in that case.

(b) Suppose you have a priority queue implemented using a binary heap and you perform a series of n `insert` operations and n `delete-min` operations in such a way that the binary heap never contains more than $\sqrt{n}$ elements. The total running time of all $2n$ operations is $O(n \log(n))$.

√ **True** ◯ False

Solution: In a binary heap with m elements, `insert` and `delete-min` both take $O(\log(m))$ time. Thus if the priority queue never has more than $\sqrt{n}$ elements then each operation never takes more than $O(\log(\sqrt{n})) = O(\log(n))$ time. Since there are $2n$ total operations, the running time is upper bounded by $O(2n \log(n)) = O(n \log(n))$.

Comment: It actually doesn't matter that the size of the heap is never more than $\sqrt{n}$. If the heap starts out empty and we perform a series of n `insert` operations and n `delete-min` operations then its maximum possible size at any point is n (in the case where we do all the insert operations first) and that is still small enough to imply that each operation takes $O(\log(n))$ time.

(c) If a directed graph can be topologically sorted then it has no cycles.

√ **True** ◯ False

Solution: Suppose G is a directed graph and $v_1, v_2, \ldots, v_n$ is a topological sort of the vertices of G. Any cycle in G would at some point have to go from a vertex with a later position in this ordering to a lower position in this ordering, but that violates the definition of topological sort. Thus G has no cycles.

(d) Suppose you implement a hash table using external chaining. If the hash table contains n elements and no chain has length more than $O(\log(n))$ then checking whether the hash table contains a given item takes $O(\log(n))$ time in the worst case.

√ **True** ◯ False

Solution: Looking up an item in a hash table that uses external chaining consists of two steps: first compute the hash of the given item and find the corresponding bucket in the hash table's array. Then look through the chain stored in that bucket, comparing each element of the chain to the given item. The first step takes $O(1)$ time and the running time of the second step is proportional to the length of chain. So if all chains have length $O(\log(n))$ then the total running time is $O(\log(n))$.

(e) Recall that the INDEPENDENT SET problem asks whether a given graph has an independent set (a set of vertices with no edges between them) larger than some given size and the REACHABILITY problem asks whether a given graph has a path between two given vertices. True or false: a reduction of the INDEPENDENT SET problem to the REACHABILITY problem is a polynomial time algorithm A that, when given a graph G and vertices s and t in G, produces a graph G' and a number m such that there is a path from s to t in G if and only if there is an indepndent set in G' of size larger than m.

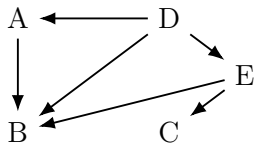
◯ True √ **False**

Solution: The algorithm described in the problem is actually a reduction of REACHABILITY to INDEPENDENT SET, not a reduction of INDEPENDENT SET to REACHABILITY.

Comment: Some people seemed to think that this question is false because $P \neq NP$. While it is true that a reduction of INDEPENDENT SET to REACHABILITY would imply that $P = NP$, the question was simply asking about the definition of "reduction." If the statement had used the correct definition of reduction then the answer would have been "True," in spite of the fact that the existence of such a reduction would imply $P = NP$.

Question 2: Topological Sort (2 points)

Find a topological sort of the following graph.



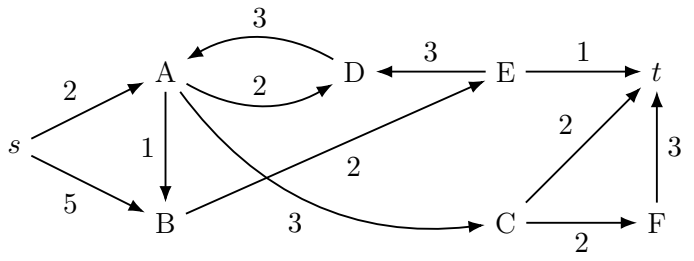
Solution: There are several valid topological sorts of this graph:

1. DECAB
2. DEACB
3. DAECB
4. DEABC
5. DAEBC

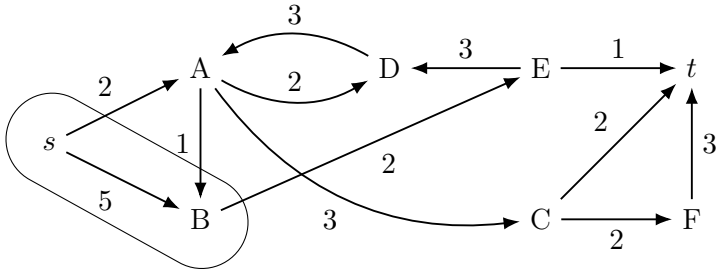
Common Mistakes: There were no mistakes on this question. It is the only question on the exam that everyone got completely correct.

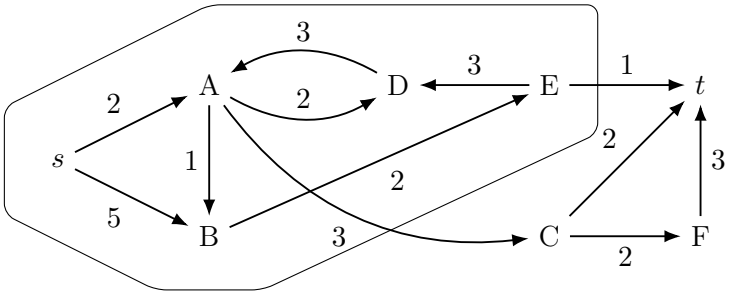
Question 3: Minimum Cut (3 points)

Find a minimum s-t cut of the following graph and state its weight. Feel free to indicate the cut by simply circling the vertices which are on the same side of the cut as s.



Solution: The minimum cut has weight $\boxed{4}$. There are two distinct minimum cuts.





Common Mistakes: Some people found a correct max flow, but did not correctly convert it into a minimum cut. Some people thought that the weight of a cut was the sum of the weights of the edges going out of the cut *minus* the sum of the weights of the edges coming into the cut. However, the weight of a cut is defined as just the sum of the weights of the edges going out of the cut.

2 Correct or Not?

Both of the following two questions contain a description of a problem and an algorithm that is supposed to solve that problem. For each question you should determine whether or not the algorithm is correct. If it is correct, prove it. If it is not correct, provide an example of an input on which it fails to find the correct answer.

Question 4: Find a MAST (10 points)

Problem. If $G = (V, E)$ is a weighted, connected, undirected graph, a set $E' \subseteq E$ is called an *almost spanning tree* of G if (V, E') is connected and $|E'| = |V|$. The *weight* of an almost spanning tree is defined in the same way as the weight of a spanning tree—as the sum of the weights of the edges in E'.

Input. A connected graph $G = (V, E)$ such that $|E| \geq |V|$.
Output. An almost spanning tree of G whose weight is as small as possible.

Algorithm. First use Prim's algorithm to find a minimum spanning tree of G and then add to it the lightest edge in G that it does not yet contain.

This algorithm is:

√ **Correct.** ◯ Not correct.

Proof of correctness or counterexample.

Solution: There are technically two things to show here: first, that the algorithm returns an almost spanning tree and second, that the weight of the almost spanning tree it returns is no larger than that of any other almost spanning tree of G. The first of these is straightforward. The second is a bit trickier; one way to prove it is to show that if T is a spanning tree and E' is an almost spanning tree then E' can always be broken into a spanning tree of G plus an edge that is not in T. The complete proof is below.

Proof. First note that every spanning tree of G is connected and has exactly $|V| - 1$ edges, so if we add one edge to a spanning tree we get a connected set of exactly $|V|$ edges. Thus the algorithm does return an almost spanning tree. It remains to prove that the almost spanning tree it returns has minimum weight.

Let T be the MST found by Prim's algorithm and let e be the lightest edge not in T. Let E' be any almost spanning tree of G. Since $|E'| > |V| - 1$, E' must contain a cycle. Since T does not contain a cycle, there must be at least one edge on this cycle that is not in T. Let e' be such an edge.

Now let's make a couple of observations. First, since e' is not in T and e is the lightest edge not in T, $w(e') \geq w(e)$. Second, since removing one edge from a cycle does not disconnect a connected graph, $E' - \{e'\}$ is connected. Since $|E' - e'| = |V| - 1$, E' must actually be a spanning tree of G and therefore $w(E' - \{e'\}) \geq w(T)$. Combining these two observations we have that

$$w(E') = w(E' - e') + w(e') \geq w(T) + w(e).$$

In other words, the weight of E' is at least the weight of the almost spanning tree returned by the algorithm.

Comment: Another way to prove that the set returned by the algorithm is correct is to first show by induction that on each step of Prim's algorithm, the current set of edges is contained in

some minimum almost spanning tree and then show that the final edge added by the algorithm above maintains this property.

Common Mistakes: Many people stated that the algorithm is correct, but there were very few correct proofs. Relatively few of these proofs contained statements that were actually false, however most contained at least one major unproven assumption.

The most common of these was to assume that there is some minimum almost spanning tree that contains a minimum spanning tree. However, this assumption is not at all obvious and is, in fact, most of what needs to be proved. A slightly more extreme form of this mistake was to assume that for every minimum spanning tree T, there is some minimum almost spanning tree which consists of T plus one extra edge. Again, this assumption is true, but is not obvious (and making it renders the proof pretty much trivial).

There were also several proofs that claimed to use induction but never actually used the inductive hypothesis.

Question 5: Triple Matching (10 points)

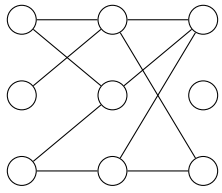
Problem. An undirected graph $G = (V, E)$ is called *strongly tripartite* if V can be partitioned into three sets V_1, V_2, V_3 such that every edge in E either has one endpoint in V_1 and one endpoint in V_2 or one endpoint in V_2 and one endpoint in V_3. The partition (V_1, V_2, V_3) is called a *tripartition* of G.

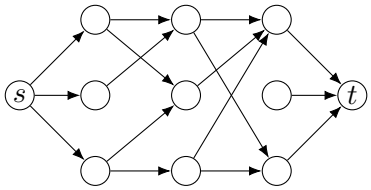
Suppose $G = (V, E)$ is a strongly tripartite graph with tripartition (V_1, V_2, V_3). A *triple matching* of G is a set $M \subseteq V^3$ of triples of vertices (i.e. lists of three vertices) in G such that no vertex occurs more than once in M and for each $(u, v, w) \in M$, $u \in V_1$, $v \in V_2$, $w \in V_3$ and $(u, v), (v, w) \in E$.

Input. A strongly tripartite graph $G = (V, E)$ and a tripartition (V_1, V_2, V_3) of G.
Output. The size of the largest possible triple matching of G.

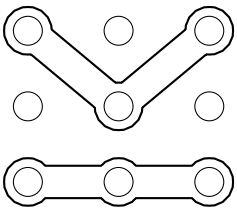
Algorithm. Create a network G' as follows. Start with V and add two new vertices, s and t. For each edge (u, v) in E such that $u \in V_1$ and $v \in V_2$, add a directed edge (u, v) to G' with weight 1. Similarly, for each edge (u, v) in E such that $u \in V_2$ and $v \in V_3$, add a directed edge (u, v) to G' with weight 1. Also add a directed edge of weight 1 from s to each vertex in V_1 and from each vertex in V_3 to t. Then use the Ford-Fulkerson algorithm to find a maximum flow on G'. Return the size of this flow as the size of the largest possible triple matching in G.

Example. Shown below is a graph G, the network G' formed from it as described above, and a triple matching of G.





Assume every edge has weight 1.



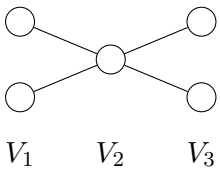
The largest triple matching has size 2

This algorithm is:

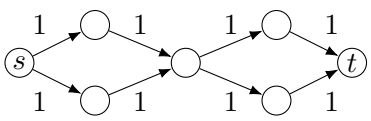
◯ Correct. √ **Not correct.**

Proof of correctness or counterexample.

Solution: Consider the following strongly tripartite graph, G.



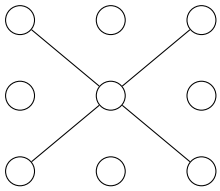
If we convert this to a network G' as described above then we get the following graph.



The maximum size of any tripartite matching in G is 1, but the maximum flow in G' has size 2.

Comment: Even though the algorithm described above does not solve the triple matching problem correctly, there is a modification of it which does work. Namely, create the network G' as above, but instead of merely giving a weight of 1 to every edge in G', also give every vertex a capacity of 1. See the homework 7 solutions for a description of how to efficiently solve the max flow problem when there are vertex capacities.

Comment: Instead of the counterexample in the solution above, many people gave a more complicated counterexample. For instance, several people drew the following graph:



This is a valid counterexample, but the 4 isolated nodes are not necessary.

Common Mistakes: A number of people claimed the algorithm is correct and tried to provide a proof. The most common mistake in these proofs was claiming that in any flow, at most one unit of flow goes through each vertex. However, this is not correct: it is perfectly valid to have a flow where every edge has weight 1 but some vertices have more than one unit of flow entering. The reason this is not an issue in the case of bipartite graphs is that each vertex has either at most one edge going to it or at most one edge leaving it, and this *does* force every vertex (other than the sink) to have at most one unit of flow going through it.

Common Mistakes: A few people stated that the algorithm was not correct, but had incorrect counterexamples. The most common mistake was to draw a graph G and claim that the maximum flow in G' is smaller than the size of the largest triple matching in G. However this is always false: for any strongly tripartite graph G, the size of the largest triple matching of G is less than or equal to the size of the maximum flow in G'.

3 Algorithm Design

Each of the next 5 questions asks you to design an efficient algorithm to solve some problem. For each problem, you should describe the main idea of your algorithm, give pseudocode for it and state its running time. You should give a brief justification of the running time (1-2 sentences should usually suffice). You do not need to provide a proof of correctness.

Partial credit. For each problem, you will receive at least 5 points (and sometimes more) if you give a correct algorithm which runs in polynomial time. Incorrect algorithms may or may not receive partial credit, depending on the details of the algorithm. Incorrect algorithms are more likely to receive partial credit if you state that you know the algorithm is incorrect and give an example of an input on which it fails.

Question 6: Painted Penguin Prefix (12 points)

You are the proud owner of n penguins, which are arranged in a line. You hired someone to paint all the penguins red, starting from penguin 1 and continuing in order until penguin n. However, the painter you hired was interrupted in the middle of their job and only painted the first few penguins. You want to find out how many of the penguins have already been painted. Design an efficient algorithm to solve this problem. For full credit, your algorithm should run in $O(\log(n))$ time.

Input. The number of penguins, n, and access to a function, `red`, which works as follows. For any $i \leq n$, `red(`i`)` returns `True` if and only if penguin i is painted red. You may assume this function runs in $O(1)$ time and that if `red(`i`)` returns `False` then so does `red(`j`)` for any $j \geq i$.

Output. The largest number i such that `red(`i`)` returns `True`.

Main idea.

Solution: Use binary search. Look at the middle penguin in the list. If it's red, recurse on the right half of the list. If it's not red then recurse on the left half of the list.

Pseudocode.

Solution: The usual pseudocode for binary search works with almost no changes.

```
FindPenguin(n):
 i = 1, j = n
   while i ≤ j:
     mid = floor((i + j)/2)
     if red(mid):
       i = mid + 1
     else:
       j = mid - 1
 return j
```

Running time.

Solution: $\Theta(\log(n))$. On every iteration of the loop we cut the length of the current range in half. Thus there will be $\Theta(\log(n))$ iterations of the loop and since we do $O(1)$ work per iteration, the total running time is $\Theta(\log(n))$.

Question 7: Filling in an Array (12 points)

Suppose you play the following game with your friend. Your friend thinks of an array A of n numbers, which you will try to guess. To help you, your friend gives you a series of hints of the form "$A[i] < A[j]$" or "$A[i] > A[j]$" (where $1 \leq i < j \leq n$). After getting k such hints, you want to find an array B of real numbers (not necessarily integers) such that each hint you received about A is also true of B. Design an efficient algorithm to solve this problem.

Example. Suppose that $n = 5$, $k = 4$ and the hints you receive are $A[1] > A[2]$, $A[1] < A[3]$, $A[2] < A[3]$, and $A[4] < A[5]$. Then one valid way to fill in B is `[5.5, 5, 6, 7.2, 7.3]`.

Input. A number n and arrays `L` and `G` such that for each $i \leq n$, `L[`i`]` contains a list of all hints of the form "$A[i] < A[j]$" and `G[`i`]` contains a list of all hints of the form "$A[i] > A[j]$."

Output. A length n array B of real numbers such that for each hint of the form "$A[i] < A[j]$," B satisfies $B[i] < B[j]$ and for each hint of the form "$A[i] > A[j]$," B satisfies $B[i] > B[j]$.

Main idea.

Solution: Create a directed graph whose vertex set is $\{1, 2, \ldots, n\}$ and such that there is an edge from i to j if you have been given a hint of the form $A[i] < A[j]$ or $A[j] > A[i]$. Because the inequalities are strict, this graph will always be a DAG (a loop in the graph would correspond to a situation where you have been given hints of the form $A[i_1] < A[i_2], A[i_2] < A[i_3], \ldots, A[i_n < A[i_1]$, which taken together are contradictory). Find a topological sort of this graph and for each i, set $B[i]$ to be the position of i in the topological sort.

Pseudocode.

Solution:

```
FillArray(L, G, n):
  H = new directed graph with vertex set {1, 2, ..., n} and no edges
  for i = 1, 2, ..., n:
    for j in L[i]:
      add edge (i, j) to H
    for j in G[i]:
      add edge (j, i) to H
  C = TopologicalSort(H)
  B = new empty array of length n
  for i = 1, 2, ..., n:
    B[C[i]] = i
```

Running time (in terms of both n and k).

Solution: $\Theta(n + k)$. The graph H has n vertices and k edges, so creating H takes $\Theta(n + k)$ time, as does running the topological sort algorithm from class on H. Filling in the array B takes $O(n)$ time. So the total running time is $\Theta(n + k + n + k + n) = \Theta(n + k)$.

Comment: I originally intended this to be a problem about greedy algorithms, but it turns out that writing down a correct and efficient greedy algorithm for this problem is pretty tricky.

Comment: It is natural to ask whether there is still an efficient algorithm to solve this problem if the hints can also have the form $A[i] \leq A[j]$ or $A[i] \geq A[j]$ (i.e. if the inequalities are not necessarily strict). You can no longer use topological sort because the resulting graph will not always be a DAG (e.g. it is valid to have both $A[1] \leq A[2]$ and $A[1] \geq A[2]$ as hints, which corresponds to having edges $(1, 2)$ and $(2, 1)$ in the graph). It turns out there is also a $\Theta(n + k)$ time solution to this problem, though it relies on ideas we did not learn about in class.

In directed graphs, the relation "there is a path from u to v and also a path from v to u" is an equivalence relation on vertices. The equivalence classes of this equivalence relation are called *strongly connected components* of the graph (in other words, a strongly connected component of a graph is a maximal set of vertices such that for any ordered pair of vertices in the set, there is a path from the first vertex to the second vertex).

Now suppose G is a directed graph. Consider the directed graph G' whose vertices consist of the strongly connected components of G, with an edge from one strongly connected component to another any time there is an edge from some vertex in the first component to some vertex in the second component. It is possible to show that this graph is always a DAG. In other words, every directed graph can be decomposed into strongly connected components connected by a DAG.

Let's return to the problem of filling in the array B when the hints do not have to be strict inequalities. We mentioned above that this case seems tricky because the graph corresponding to the hints may not be a DAG. However, note that if i and j are part of the same strongly connected component of the graph then this implies $A[i] = A[j]$. Thus the problem of filling in the array B can be reduced to the problem of picking a number to assign to each strongly connected component. But we are then back to the case of a DAG, so we can use topological sort.

There is just one possible problem with this algorithm: how do we find the strongly connected components of the graph? Finding strongly connected components in directed graphs is trickier than finding connected components in undirected graphs, but it turns out that there is an algorithm (using DFS and postvisit numbers) which does it in $\Theta(|V| + |E|)$ time (see either of the course textbooks for more details on how this works).

Question 8: Road Trip (12 points)

You have decided to drive from Los Angeles to Boston for winter break and you want to figure out which hotels you should stay at along the way. Assume that there is only one highway from LA to Boston, that it is exactly n miles long and that there is exactly one hotel at each mile along this highway. Each day, you can drive at most k miles and then, if you are not yet at Boston, you must stop at a hotel for the night. Given the price of staying at each hotel for one night, you want to find the minimum possible total amount of money that you can spend on hotels during your trip.

Example. Suppose $n = 10$, $k = 3$ and the costs of the hotels, in order, are $3, 1, 4, 5, 2, 3, 7, 8, 1$ (note that there are only 9 hotels because after you have driven 10 miles, you have reached Boston and do not need a hotel). Then the minimum total cost is 7, which can be obtained by staying at the hotels at miles 2, 5, 6 and 9.

Input. An array A such that for each $i < n$, `A[i]` contains the cost to stay for one night at the hotel at mile i along the highway.

Output. The least number c such that there are indices $i_1 < i_2 < \ldots < i_l$ (indicating which hotels to stay at) which satisfy the two following constraints.

- $i_1 - 0$, $i_2 - i_1$, $i_3 - i_2$, …, $i_l - i_{l-1}$, $n - i_l$ are all less than or equal to k (i.e. you don't drive more than k miles per day, including the first and last days).
- $A[i_1] + A[i_2] + \ldots + A[i_l] = c$ (i.e. the total cost to stay at these hotels is c).

Note that you only need to return the minimum cost c and not the indices $i_1, \ldots, i_l$.

Hint. There is a dynamic programming solution with subproblems $C(i)$ defined as the minimum cost to travel the first i miles. You are free to use a different solution, however.

Main idea.

Solution: There were two main approaches to this problem. Both used dynamic programming.

Solution 1. For each $i \leq n$, define $C(i)$ to be the cost to reach mile i on the highway. Then $C(0) = 0$ and for all $i > 0$ we have

$$C(i) = \min\{C(j) + A[j] \mid i - k \leq j < i \text{ and } j \geq 0\}.$$

We can calculate all $C(i)$ using dynamic programming and return $C(n)$ as the final answer.

Solution 2. For each $i < n$, define $C(i)$ to be the cost to reach mile i *and stay at the hotel at mile* i. Then $C(0) = 0$ and for all $i > 0$ we have

$$C(i) = A[i] + \min\{C(j) \mid i - k \leq j < i \text{ and } j \geq 0\}.$$

We can calculate all $C(i)$ using dynamic programming and then return $\min\{C(i) \mid n - k \leq i < n\}$ as the final answer.

Pseudocode.

Solution:

Solution 1.

```
RoadTripCost(A, n, k):
```

```
  C = new array of length n + 1
  C[0] = 0
  for i = 1, 2, ..., n:
    C[i] = min(C[j] + A[j] for j between max(0, i - k) and i - 1)
  return C[n]
```

Solution 2.

```
RoadTripCost(A, n, k):
  C = new array of length n + 1
  C[0] = 0
  for i = 1, 2, ..., n:
    C[i] = A[i] + min(C[j] for j between max(0, i - k) and i - 1)
  return min(C[i] for i between max(0, n - k, n - 1)
```

Running time (in terms of both n and k).

Solution: $\Theta(nk)$ (for both approaches). There are n subproblems and each one takes $O(k)$ time to solve.

Comment: There were a number of minor variations of the above algorithms. For example, in the first solution you can immediately set $C(i) = 0$ for any $i \leq k$ and for the second solution, you can immediately set $C(i) = A[i]$ for any $i \leq k$.

For the second solution, you can also first extend the length of A by one and set $A[n] = 0$ and then just return $C[n]$ at the end of the algorithm instead of having to take the minimum over the last k elements.

Common Mistakes: One common mistake was to use an approach similar to the second solution above, but fail to take the minimum over the last k miles at the end, and instead just return $C[n]$. There are two problems with this. First, $A[n]$ is not defined so calculating $C[n]$ does not make sense. Second, even if $A[n]$ were defined, this would not be the correct answer because the last hotel you stay at can be in any of the last k miles; it does not need to be at mile n itself.

Common Mistakes: A few people tried a greedy approach to this problem. For example, it is tempting to think that the following greedy approach might work: set i_1 to be the location of the cheapest hotel in the first k miles. Then set i_2 to be the location of the cheapest hotel in the next k miles after i_1, and so on until you reach n. However, this doesn't work. To see why, consider the following example. Suppose $n = 4$, $k = 2$ and the prices of the hotels, in order, are $1, 2, 100$. In this case, the greedy algorithm will choose the hotels at miles 1 and 3, for a total cost of 101, whereas the optimal solution is to just pick the hotel at mile 2, for a total cost of 2.

Question 9: Graph with Unreliable Edges (12 points)

Suppose $G = (V, E)$ is a directed graph and $s, t \in V$. Some of the edges in G have been marked as *unreliable*. You want to know if there is a path from s to t in G that uses at most 2 unreliable edges. Design an efficient algorithm to solve this problem.

Input. A directed graph $G = (V, E)$, vertices $s, t \in V$ and access to a function `unreliable`, such that for any edge $e \in E$, `unreliable(`e`)` returns `True` if and only if e is an unreliable edge. You may assume that this function runs in $O(1)$ time.

Output. `True` if there is a path from s to t in G that uses at most 2 unreliable edges and `False` otherwise.

Main idea.

Solution: There are several valid solutions to this problem.

Solution 1. Assign a weight of 0 to every reliable edge and 1 to every unreliable edge and run Dijkstra's algorithm on the resulting weighted graph. Then check whether the distance from s to t is less than or equal to 2.

If you want to avoid using edges with weight 0, you can also set each reliable edge to have weight 1 and each unreliable edge to have weight $|E| + 1$ and check whether the distance from s to t is less than or equal to $3|E| + 2$.

Solution 2. Make three copies of G and delete the unreliable edge from each copy then for each unreliable edge, add an edge from the first copy to the second copy and the second copy to the third copy. Use DFS or BFS to check if the any copy of t is reachable from the first copy of s.

More precisely, create a new graph G' as follows. For every vertex v in V, add three vertices to G', v^0, v^1, v^2. For each reliable edge (u, v) in G, add edges (u^0, v^0), (u^1, v^1), (u^2, v^2) to G' and for each unreliable edge (u, v), add edges (u^0, v^1) and (u^1, v^2) to G'. Then use DFS or BFS to check if any of t^0, t^1, t^2 are reachable from s^0. The motivating idea of this solution, by the way, is that a vertex v^i in G' corresponds to a vertex v in G which is reachable using exactly i unreliable edges.

Solution 3. See the solution to the second extra credit problem.

Pseudocode.

Solution:

Solution 1.

```
Weight(u, v):
  if unreliable(u, v): return 1
  else: return 0

UnreliableEdges(G = (V, E), s, t):
  dist = Dijkstra(G, Weight, s, t)
  if dist ≤ 2: return True
  else: return False
```

Solution 2. There are (at least) two ways to turn the idea of the second solution into pseudocode. One is to directly produce the modified graph G' and then run DFS/BFS on it. The other is to modify DFS or BFS so that running the modified algorithm on G is equivalent to running DFS on

G'. Both approaches ar shown below.

```
UnreliableEdges(G = (V, E), s, t):
  G' = new graph with vertex set Vx{0, 1, 2} and no edges
  for edge (u, v) in E:
    if unreliable(u, v):
      add edges ((u, 0), (v, 1)), ((u, 1), (v, 2)) to G'
    else:
      add edges ((u, 0), (v, 0)), ((u, 1), (v, 1)), ((u, 2), (v, 2)) to G'
  if Reachable(G', (s, 0), (t, 0)) or
      Reachable(G', (s, 0), (t, 1)) or Reachable(G', (s, 0), (t, 2)):
    return True
  else:
    return False
```

```
Explore(G, v, visited, visit_number):
  if visit_number ≤ 2:
    visited[v, visit_number] = True
    for u in neighbors(v):
      if unreliable(v, u):
        if not visited[u, visit_number + 1]:
          Explore(G, u, visited, visit_number + 1)
      else:
        if not visited[u, visit_number]:
          Explore(G, u, visited, visit_number)

UnreliableEdges(G = (V, E), s, t):
  visited = new nx4 array of Booleans, initially all set to False
  Explore(G, s, visited, 0)
  if visited[t, 0] or visited[t, 1] or visited[t, 2]: return True
  else: return False
```

Running time.

Solution:

Solution 1. $\Theta(|V| \log |V| + |E|)$. Essentially just the cost of running Dijkstra's algorithm on G (plus the time it takes to compute edge weights, but computing the weight of a single edge takes constant time and Dijkstra's algorithm will check the weight of each edge exactly once).

Solution 2. $\Theta(|V| + |E|)$. The new graph G' has $3|V|$ vertices and at most $3|E|$ edges, so running DFS/BFS on it will take $\Theta(3|V| + 3|E|) = \Theta(|V| + |E|)$ time.

Common Mistakes: A lot of people attempted to modify DFS or BFS to solve this problem, but did so incorrectly. The most common strategy was to run DFS and keep a counter for each vertex indicating how many unreliable edges were used to reach that vertex. Unfortunately, this just tells you whether the path from s to t in the DFS tree uses at most 2 unreliable edges, rather

than whether *some* path uses at most 2 unreliable edges.

Some people realized this problem and attempted to fix it, but did so incorrectly. Other people made a mistake when implementing this strategy, giving algorithms that did not even correctly report how many unreliable edges there are in the path from s to t in the DFS tree. One common mistake of this form was to keep a single global counter of unreliable edges rather than having a separate counter for each vertex. But this just counts the total number of unreliable edges in the DFS tree, which has little to do with the number of unreliable edges on paths between s and t.

Comment: By far the most common correct solution was to use Dijkstra's algorithm as described above. There were also several (correct) variations of solution 2, some of them fairly complicated. One person came fairly close to the algorithm described in the solution to the extra credit problem.

Question 10: Small Diameter Partition (12 points)

The *diameter* of an array of numbers is the difference between the largest element and the smallest element in the array. Note that if the array is sorted in increasing order then this is just the difference between the last and first elements.

Suppose A is an array of numbers, and $k > 0$. The *diameter* of a partition of A into k subarrays is the maximum diameter of any of those subarrays. Note that a partition of A into k subarrays is equivalent to a choice of k numbers $1 = i_1 < i_2 < \ldots < i_k$ indicating the beginning of each subarray (so the first subarray consists of $A[1], A[2], \ldots, A[i_2 - 1]$, the second consists of $A[i_2], A[i_2 + 1], \ldots, A[i_3 - 1]$ and so on). Furthermore, note that if A is sorted in increasing order then the diameter of the first subarray is $A[i_2 - 1] - A[i_1]$, the diameter of the second subarray is $A[i_3 - 1] - A[i_2]$, etc.

You want to design an efficient algorithm to solve the following problem: given a length n array of integers, A, sorted in increasing order and a number $k > 0$, find the minimum diameter of any partition of A into exactly k subarrays.

Example. Suppose $A =$ `[1, 5, 6, 8, 13, 15]` and $k = 3$. The minimum possible diameter of any partition of A into k subarrays is 3 and one partition with this diameter is `[1], [5, 6, 8], [13, 15]` (in other words, $i_1 = 1$, $i_2 = 2$, $i_3 = 5$).

Input. A length n array A of integers, sorted in increasing order, and a positive integer, k.

Output. The minimum possible diameter of any partition of A into k subarrays. In other words, the minimum possible value of $\max(A[n] - A[i_k], A[i_k - 1] - A[i_{k-1}], \ldots, A[i_2 - 1] - A[i_1])$ over all choices of indices $1 = i_1 < i_2 < \ldots < i_k \leq n$.

Main idea.

Solution: Use dynamic programming. For any $0 \leq i \leq n$ and $0 \leq l \leq k$, define $D(i, l)$ to be the smallest diameter of any partition of the first i entries of A into exactly l subarrays. Then we have the recursive formula

$$D(i, l) = \min\{\max(D(j, l-1), A[i] - A[j+1]) \mid 0 \leq j < i\}$$

and base cases

$$D(i, 0) = \begin{cases} 0 & \text{if } i = 0 \\ \infty & \text{if } i > 0 \end{cases} \qquad \text{and} \qquad D(0, l) = \infty \quad \text{for all } l > 0$$

and the answer to the original problem is $D(n, k)$.

Pseudocode.

Solution:

```
MinDiameter(A, n, k):
  D = new (n + 1)x(k + 1) array
  D[0, 0] = 0
  for i = 1, 2, ..., n:
    D[i, 0] = ∞
  for l = 1, 2, ..., k:
    D[0, l] = ∞
  for l = 1, 2, ..., k:
```

```
    for i = 0, 1, 2, ..., n:
      D[i, l] = min(max(D[j, l - 1], A[i] - A[j + 1]) for 0 ≤ j < i)
  return D[n, k]
```

Running time (in terms of both n and k).

Solution: $\Theta(n^2k)$. There are $(n+1)(k+1)$ total subproblems and they each take $O(n)$ time to solve (because to find $D(i,l)$ we need to take the minimum of i values). If you want to be really careful, filling in the base cases takes $\Theta(n+k)$ time and the running time of the rest of the algorithm is proportional to

$$\sum_{l=1}^{k}\sum_{i=1}^{n} i = \sum_{i=1}^{n}\sum_{l=1}^{k} i = \sum_{i=1}^{n} ki = k\sum_{i=1}^{n} i = k\cdot\Theta(n^2) = \Theta(n^2k).$$

Common Mistakes: Several people tried to give a greedy algorithm for this problem. It is possible that there is a correct (and efficient) greedy algorithm for this problem, but nobody found one successfully on this exam. Also, I considered several different greedy approaches to this problem and did not find one that works.

Common Mistakes: A few people said that dynamic programming could be used to solve this problem but did not describe a correct dynamic programming algorithm.

Comment: It may seem odd that we set $D(0,l) = \infty$ for all $l > 0$. This is because the problem says we have to break A into k subarrays and (implicitly) says that the subarrays all have to be nonempty (e.g. the problem stipulates that $i_1 < i_2$ rather than $i_1 \le i_2$). But there is no way to break an empty array into $l > 0$ nonempty subarrays, so $D(0,l)$ must be ∞. Moreover, for any $i < l$, we must have $D(i,l) = \infty$ but this does not need to be set separately (it is handled correctly by the recursive formula when $i > 0$).

Comment: There is a way to speed up the computation of $D(\cdot,\cdot)$ in the above algorithm so that the algorithm runs in time $\Theta(nk)$ rather than $\Theta(n^2k)$.

First, note that for $j \ge l-1$, $D(j, l-1)$ is a (not necessarily strictly) increasing function of j (i.e. as we make j bigger, $D(j,l-1)$ either stays the same or increases). Second, note that for a fixed $i \ge l$, $A[i] - A[j+1]$ is a (not necessarily strictly) decreasing function of j.

Together, these two observations imply that for a fixed l and $i \ge l$, as we increment j from $l-1$ to i, $\max(D(j,l-1), A[i] - A[j+1])$ first decreases (as long as $A[i] - A[j+1]$ is the dominant term in the maximum) and then increases (after $D(j,l-1)$ becomes the dominant term). The minimum is the point at which it switches from decreasing to increasing. Furthermore, since $A[i+1] - A[j+1]$ is always at least as large as $A[i] - A[j+1]$, this minimum value of j can only increase as i increases (the $D(j,l-1)$ term is independent of i and the $A[i] - A[j+1]$ term increases with i, so the point where the former term becomes dominant can only increase).

This has two important consequences for our algorithm. First, to find the optimal value of

j when computing $D(i,l)$ we can start from $j = l-1$ and keep increasing j until we see $\max(D(j,l-1), A[i]-A[j+1])$ start to increase (which means the $D(j,l-1)$ term has started to dominate). The point just before it started to increase is the optimal value of j. Second, when computing $D(i+1,l)$, instead of starting again from $j = h-1$, we can start from the value of j that was optimal for $D(i,l)$. Thus for each l, we can keep track of a single variable j and for each i we increment j until we find the minimum value of $\max(D(j,l-1), A[i]-A[j+1])$. Then when computing $D(i+1,l)$ we can start from the value we left off with when looking at $D(i,l)$ rather than starting over from $j = l-1$.

Here's some pseudocode to illustrate this.

```
MinDiameter(A, n, k):
  D = new (n + 1)x(k + 1) array
  D[0, 0] = 0
  for i = 1, 2, ..., n:
    D[i, 0] = ∞
  for l = 1, 2, ..., k:
    for i = 0, 1, ..., l - 1:
      D[i, l] = ∞
    j = l - 1
    for i = l, l + 1, ..., n:
      while j + 1 < i and max(D[j + 1, l - 1], A[i] - A[j + 2])
                          ≤ max(D[j, l - 1], A[i] - A[j + 1]):
        j = j + 1
      D[i, l] = max(D[j, l - 1], A[i] - A[j + 1])
  return D[n, k]
```

Note that in the main for loop of the above algorithm, the amount of work we do is proportional to the number of times we increment i and j. But for each value of l, we can increment j at most n times and i at most n times. Thus for each value of l we do $O(2n) = O(n)$ work. Since there are k possible values of l, this implies that the total running time is $O(nk)$.

4 Extra Credit

The following two questions are extra credit. They are more challenging than the rest of the problems and are worth relatively few points, so I recommend that you only attempt them if you have finished the rest of the exam. For these questions, you do not need to provide pseudocode or an analysis of the running time, just the main idea of your algorithm.

Question 11: Faster Partitions (2 points (bonus))

Suppose that in question 10 the entries in A are all positive integers which are less than n^2. Show how to solve the problem in $O(n\log(n))$ time in this case.

Solution: First note that if we are given a number C, there is a greedy algorithm which runs in $O(n)$ time and checks whether there is a partition of A into k subarrays with diameter at most C. This algorithm works as follows: first it sets $i_1 = 1$ and finds the largest $i_2 > i_1$ such that $A[i_2 - 1] - A[i_1] \leq C$. Then it finds the largest $i_3 > i_2$ such that $A[i_3 - 1] - A[i_2] \leq C$. And so on. At the end of this process, check whether $A[n] - A[i_k] \leq C$. If so then the algorithm has found a partition with k subarrays and diameter at most C. If not, then there is no such partition.

Now note that if all entries in A are integers between 1 and n^2 then the diameter of any subarray of A is an integer between 0 and n^2. Thus the only possible values of the diameter of a partition of A into k subarrays are the integers between 0 and n^2. And for any given integer, we can check in $O(n)$ time whether there is a partition with diameter less than or equal to that integer. So we can use binary search to find the least integer C such that there is a partition of A into k subarrays with diameter at most C (pretty much identical to the penguins problem) and this C is the answer to the original problem.

The binary search needs to run for $O(\log(n^2)) = O(\log(n))$ iterations and on each iteration we run the greedy algorithm, which takes $O(n)$ time. Thus the total running time is $O(n\log(n))$.

Comment: Nobody solved this problem correctly.

Question 12: More Unreliable Edges (2 points (bonus))

Suppose that in the question 9, instead of being allowed to use at most 2 unreliable edges, you can use at most k unreliable edges. Describe an algorithm that runs in time $\Theta(|V| + |E|)$ no matter what k is. You may assume that the graph is undirected rather than directed.

Solution: First remove all unreliable edges from G and run DFS (or BFS) to find the connected components. Then create a new graph G' whose vertices are the connected components of G and where there is an edge (A, B) whenever there is an unreliable edge from some vertex in A to some vertex in B. Run BFS on this graph to check whether the connected component containing t is reachable from the connected component containing s using at most k edges.

Each step of this algorithm (removing the unreliable edges, finding the connected components, creating G', and running BFS on G') all take $O(|V| + |E|)$ time, so the entire algorithm runs in $O(|V| + |E|)$ time (no matter what k is).

Comment: Nobody solved this problem correctly either ☹. Some people claimed that their solution to problem 9 also solved this problem, but in all such cases the solution to problem 9 was either wrong or had running time slower than $\Theta(|V| + |E|)$.