Note: the Fall ‘16 version of this class covered graph algorithms before dynamic programming in two separate units. This offering covered these topics in reverse order, and in less depth. So the last questions on both tests would be considered out of scope this time.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Title</th>
<th>Points</th>
<th>Parts</th>
<th>Grade</th>
<th>Initials</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Name / GT Id on top of every page</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Knapsack</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Weighted Interval Scheduling</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Row Segmentation Game</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Dropping Mugs</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Reachability and Shortest Paths</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Minimum Spanning Trees</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Changing Words</td>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cut Property</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Coyote reloaded</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
0 (1 point) **Write your name and student id on top of every page.**

1. (5 points) Volatile Knapsack

You're trying to fill a knapsack with volatile objects. There are \( n \) types of objects, each with weight \( w_i \), and a volatility level \( p_i \). There are an infinite number of objects of each type, so think of this as a variant of knapsack with replacement.

The following dynamic program tries to minimize the maximum volatility of an object needed to make up a total capacity of \( W \).

State: for each \( 0 \leq w \leq W \), let \( OPT[w] \) be the minimum maximum volatility of a set of objects whose total weight is \( w \).

Base case: \( OPT[0] = 0 \).

Transition

\[
OPT[w] = \min \{ \max \{ OPT[w - w_i], p_i \} \},
\]

Ordering: increasing order of \( W \).

Suppose \( W = 10 \) and there are three types of objects:

(a) \( w_i = 1, p_i = 10 \),
(b) \( w_i = 3, p_i = 7 \),
(c) \( w_i = 4, p_i = 6 \)

Compute the values of \( OPT[1] \ldots OPT[10] \): each correct value is worth 1/2 points.

\[
\]
SOLUTION:

\[ OPT[0] = 0 \]
\[ OPT[1] = 10 \]
\[ OPT[2] = 10 \]
\[ OPT[3] = 7 \]
\[ OPT[4] = 6 \]
\[ OPT[5] = 10 \]
\[ OPT[6] = 7 \]
\[ OPT[7] = 7 \]
\[ OPT[8] = 6 \]
\[ OPT[9] = 7 \]
\[ OPT[10] = 7 \]

GRADING:
1/2 point per correct value.
2. (5 points) Weighted Interval Scheduling

Alfred has somehow come up with a schedule of \( n \) crimes that will occur tonight in Gotham City. Each crime happens in an interval 
\[ [\text{start}_i, \text{end}_i], \]
and can be stopped if Batman is present for the entire duration of the crime.

We will refer to these crimes as intervals from now on. For simplicity, you may assume that all the \( \text{start}_i \) and \( \text{end}_i \) values are distinct. However, the intervals may overlap.

Commissioner Gordon assigned a value to each interval if it is stopped (aka. Batman is present for its entire duration). Batman can’t be at two places at once, but can travel from one place to another in negligible time via the Batmobile. Given this list, generate a sublist of non-overlapping intervals of maximum total value.

(a) (2 points) If Alfred uses the following greedy algorithm to generate the sublist for Batman,

i. Select an interval of maximum value, add it to the ‘stopped’ list.

ii. Remove all remaining interval whose time range overlap with the one added.

iii. Repeat from step (i.) until the list of interval is exhausted.

Show an example with 4 or fewer interval where his algorithm fail

**SOLUTION:**
Any valid example suffices. For example, 1 – 10 with value 10, then 1 – 2 and 3 – 4 with values 9 each.

**GRADING:**
1 point for example and algorithm output, 1 point for the correct output
1/2 removed if values not mentioned or gave wrong answer,
1 point if only gave intervals.

(b) (3 points) Design a \( O(n^{10}) \) time (or better) algorithm to find the maximum value of a subset of crimes whose time intervals don’t overlap, and therefore can be stopped.

**SOLUTION:**
Sort the intervals in increasing order of finishing time. \( a_1, a_2...a_n \)
State \( T(i) \) is the max value that can be obtained in the sorted intervals \( a_1, a_2...a_i \)
Base case is \( T(1) = v_1 \)
Recurrence is \( T(i) = \max(v_i + T(\text{comp}(i)), T(i - 1)) \)
where \( \text{comp}(i) \) is the last interval in the list before \( a_i \) compatible (not overlapping) with \( a_i \)
Overall time complexity would be \( O(n^2) \)

**GRADING:**
2 points for correct algorithm and 1 point for complexity.
1 point off for minor errors.
3. (5 points) Row Segmentation Game

Consider the following game played on a row of \( n \) buttons. Initially the first and last buttons are ‘+’, and the other buttons are either ‘X’ or ‘?’. Also, initially the left and right neighbors of ‘?’ buttons are always ‘X’.

For example, here is a possible arrangement with 11 buttons:

\[ + \; X \; X \; ? \; X \; ? \; X \; X \; ? \; X \; + \]

To play, you press the buttons labelled with ‘?’ one at a time. Each time you do:

- You are charged a cost that’s

\[
\frac{1}{\text{number of ‘X’s between the nearest ‘+’s to the left and right of the ‘?’ pushed}}
\]

- and the ‘?’ becomes a ‘+’.

Your goal is press all the ‘?’s with the minimum total cost.

**You may assume that all arithmetic are exact for this problem.**

For example, the following would be the intermediate results from pressing the second ‘?’; then the first ‘?’; then the third ‘?’; (all indexed according to the initial row) and the respective costs:

\[
\begin{align*}
+ \; X \; X \; ? \; X \; ? \; X \; X \; ? \; X \; + & \quad \text{initialization} \\
+ \; X \; X \; ? \; X \; + \; X \; X \; ? \; X \; + & \quad \text{cost} = 1/6 \\
+ \; X \; X \; + \; X \; + \; X \; X \; ? \; X \; + & \quad \text{cost} = 1/3 \\
+ \; X \; X \; + \; X \; + \; X \; X \; + \; X \; + & \quad \text{cost} = 1/3
\end{align*}
\]

(a) (2 points) Consider the greedy algorithm that always presses the ‘?’ button with least cost and, in the case of ties, selects the leftmost min-cost button.

Using the same initialization

\[ + \; X \; X \; ? \; X \; ? \; X \; X \; ? \; X \; + \]

show that this algorithm will not play the min-cost game.

**SOLUTION:**

This algorithm will push buttons from left to right, resulting in:

+XX?X?XX?X+ - initialization  
+XX+X?XX?X+ - cost = 1/6  
+XX+X+XX?X+ - cost = 1/4  
+XX+X+XX+X+ - cost = 1/3  

However, pressing buttons from right to left is even cheaper:

+XX?X?XX?X+ - initialization
+XX?X?XX+X+ - cost = 1/6
+XX?X+XX+X+ - cost = 1/5
+XX+X+XX+X+ - cost = 1/3

**GRADING:**

−0.5 for any minor error. Pressing the third, then first, and then second green button also gives a total cost of \((1/6 + 1/5 + 1/3)\), but all other answers are at least as costly as pressing from left to right.

(b) (6/3 points) Give an \(O(n^{10})\) time (or better) dynamic programming algorithm that finds the min-cost way to press all the '?' buttons.

**SOLUTION:**

Assign the indices 1 . . . n to the ‘X’ buttons from left to right. Assign each of the ‘?’ buttons the index of its left ‘X’ neighbor. Let \(G\) store the indices of the ‘?’ buttons. Let \(C(i, j)\) be the min-cost of pressing all ‘?’ buttons between the two ‘X’ buttons with indices \(i, j\) where \(i < j\) and buttons \(i, i + 1, \ldots, j\) are bordered by ‘+’ buttons.

Base case: \(i, j\) are between some \(G[h] + 1, G[h + 1]\), meaning that there are no ‘?’ buttons between the ‘X’ buttons at indices \(i, j\). In this case, return \(C(i, j) = 0\) since no buttons to push means zero cost.

Transition function: \(C(i, j) = \min_{h: i < G[h] < j} (C(i, G[h]) + C(G[h] + 1, j) + 1/(j - i + 1)),\) where, again, \(i, j\) are the indices of ‘X’ buttons bordered by ‘+’ buttons.

Runtime: We compute \(C(1,n)\). For efficiency, we can store a cost table that has \(O(n^2)\) entries, one for each pair \(i, j\), where \(i \leq j\). For any entry in the table, we have to look up at most \(O(m)\) positions of our ‘?’ buttons. Since \(m \leq (n - 1)\) by design, each lookup is \(O(n)\), so our total runtime is \(O(n^3)\).

**GRADING:**

2 points for a correct \(O(n^{10})\). 1 point for a nearly correct algorithm.
1 point for reasonable explanation of correct runtime.
4. (5 points) Mug Dropping

There are \( k \) identical mugs and a \( n \) story building. There is some value \( x \) between 1 and \( n \) such that if a mug is dropped from a floor \( \geq x \), it breaks. (aka. it’s guaranteed that the mug breaks when dropped from floor \( x \) or higher.) You want to find this value of \( x \) by dropping the fewest number of mugs.

The only allowed operation is to drop a mug from some floor: if the mug breaks when you drop it, you cannot reuse it again, otherwise it can be reused.

Your goal is to devise a dynamic programming algorithm to determine the minimum number of drops needed to determine \( x \) in the worst case.

(a) (1 point) Suppose you are given 1 mug for a 5 story building \((k = 1, n = 5)\). What is the minimum number of drops you have to make from which you are guaranteed to find the lowest floor where the mug breaks?

**SOLUTION:**
Minimum 4 drops required.

**GRADING:**
\( n - 1 \) accepted as a correct answer.

(global for problem 4:) \(-1/2\) if off-by-one anywhere, but only once total.

(b) (1 point) We will give a dynamic programming solution to this problem based on the following state:

\( T[i][j] \) denotes the minimum number of tests that guarantees to return \( x \) when the range of possible \( x \) is an interval of \( i \) floors and there are \( j \) mugs available.

Based on your answer to part (a), fill out the following entries in the table:

\[
\begin{align*}
T[1][1] &= \\
T[2][1] &= \\
T[3][1] &= \\
\end{align*}
\]
SOLUTION:

\[ T[1][1] = 0 \]
\[ T[2][1] = 1 \]
\[ T[3][1] = 2 \]

GRADING:
(global for problem 4:) \(-1/2\) if off-by-one anywhere, but only once total.

(c) (\(\frac{3}{2}\) points) Suppose you now have 2 mugs for the same 5 story building, and you dropped your first mug from floor 3. What’s the entry of \(T\) that gives the optimum number of drops in the case where:

i. (\(\frac{1}{2}\) point) The mug breaks (note that in this case, floor 3 is still a candidate for \(x\)):

SOLUTION:
\[ T[3][1]. \]

ii. (\(\frac{1}{2}\) point) The mug doesn’t break:

SOLUTION:
\[ T[2][2]. \]

GRADING:
(global for problem 4:) \(-1/2\) if off-by-one anywhere, but only once total.

(d) (\(\frac{1}{2}\) point) State the transition of a dynamic programming algorithm using the states \(T[i][j]\) specified in part (b). It should enumerate over the floor \(a\) where the first mug is dropped from.

SOLUTION:

\[ T[i, j] = \min_{1 \leq a \leq i} \{ \max \{T[a, j - 1], T[i - a, j]\} \} + 1 \]

GRADING:

(global for problem 4:) \(-1/2\) if off-by-one anywhere, but only once total.
5. (4 points) Reachability and Shortest Paths

For the following directed graph on 6 vertices, 

\[ r, s, t, u, v, w, \]

with 9 edges given in order as:

\[
\begin{align*}
 r &\rightarrow s, \quad l_{r\rightarrow s} = -3 \\
 s &\rightarrow t, \quad l_{s\rightarrow t} = 3 \\
 v &\rightarrow t, \quad l_{v\rightarrow t} = 1 \\
 t &\rightarrow s, \quad l_{t\rightarrow s} = 1 \\
 v &\rightarrow r, \quad l_{v\rightarrow r} = 5 \\
 s &\rightarrow v, \quad l_{s\rightarrow v} = -2 \\
 u &\rightarrow s, \quad l_{u\rightarrow s} = 1 \\
 u &\rightarrow v, \quad l_{u\rightarrow v} = 1 \\
 w &\rightarrow u, \quad l_{w\rightarrow u} = 1 
\end{align*}
\]

(a) (2 points) In this graph, indicate which vertices are reachable from \( s \). The answer in part b) may be helpful for this.

**SOLUTION:**

\( s, t, v, r. \)

**GRADING:**

1/2 point per vertex (including \( s \)), \(-1/2\) per extra vertex.
(b) ( /2 points)

Consider running the Bellman-Ford algorithm starting from $s$, with edges checked in EXACTLY this ordering per iteration of the outer loop.

**Due to discrepancy with the text book, you may assume that the updates are done with either the $d[u]$ values from the previous outer iteration, or the most up-to-date ones.**

However, the choice needs to be clearly stated, and consistent throughout the steps.

Give the state of the algorithm after each loop through all the edges. The table below has more rows than the number of algorithm iterations.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\infty$</td>
<td>0</td>
<td>3</td>
<td>$\infty$</td>
<td>$-2$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>$-1$</td>
<td>$\infty$</td>
<td>$-2$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>$-1$</td>
<td>$\infty$</td>
<td>$-2$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SOLUTION:**

Result is same if the updates are done with the lastest $d[u]$ values, or with the $d[u]$ values from the previous iteration:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\infty$</td>
<td>0</td>
<td>3</td>
<td>$\infty$</td>
<td>$-2$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>$-1$</td>
<td>$\infty$</td>
<td>$-2$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>$-1$</td>
<td>$\infty$</td>
<td>$-2$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**GRADING:**

1 point for correct final result, 1 point for all intermediate steps. $-1/2$ for any minor issues.

Flexible on:

- initial state (either all $\infty$, a 0 and rest $\infty$, or the state after the first iteration through edges).

- Termination: either terminate when no changes are made in an outer iteration, or after all $n$ steps.
1 point if started from $r$ instead of $s$. 
6. (4/4 points) Minimum Spanning Tree

Consider an undirected graph on 8 vertices, \(a, b, c, d, e, f, g, h\) with 12 edges given in order as:

\[
\begin{align*}
    yx & : 1 \\
    tx & : 2 \\
    xw & : 3 \\
    tw & : 4 \\
    rs & : 5 \\
    tu & : 6 \\
    ry & : 7 \\
    st & : 8 \\
    uw & : 9 \\
    wv & : 10 \\
    sy & : 11 \\
    uw & : 12
\end{align*}
\]
(a) ( 2 points) Give the result of running Kruskal’s algorithm on this edge sequence (specify the order in which the edges are selected).

SOLUTION:
The edges picked are $yx, tx, xw, rs, tu, ry, uv$.

GRADING:
1 point if only the final state is given. $-1/2$ for every difference / order swap.

(b) ( 2 points) For the same graph, exhibit a cut that certifies that the edge $ry$ is in the minimum spanning tree. Your answer should be in the form of $E(S, V \setminus S)$ for some vertex set $S$. Specifically, you should find $S$.

SOLUTION:
Consider the cut $S = \{r, s\}$.

GRADING:
only this cut works. $-1/2$ if cut is given as set of edges.
7. ( /5 points) Changing Words
You have a dictionary of $n$ words, each with up to 10 characters (note that $10 = O(1)$).
Given two words $s$ and $t$, you need to find a way to change the word $s$ into the word $t$, while
changing only one letter at a time such that every intermediate word belongs to $D$.
For example, if we have
$$D = ['hit', 'cog', 'hot', 'dot', 'dog', 'lot', 'log'],$$
one way to change ‘hit’ to ‘cog’ is:
$$'hit' \rightarrow 'hot' \rightarrow 'dot' \rightarrow 'dog' \rightarrow 'cog'.$$

(a) ( /2 points)
Model this as a graph problem. What would be the vertices and edges in the graph? How
can the original problem of changing the word $s$ to the word $t$ be stated in terms of this
graph?
**SOLUTION:**
Each word in $D$ is represented as a vertex. There is an edge between two vertices if the
words are different by just one letter. In this graph model, the problem becomes finding a
path from $s$ to $t$.
**GRADING:**
1 point for correct graph construction. 1 point for correct interpretation of the original
problem.

(b) ( /2 points)
Show that this graph can be constructed in $O(n^2)$ time, and its size is up to $O(n^2)$.
**SOLUTION:**
There are $n$ words in the dictionary, so it takes $O(n)$ time to generate the vertices. For
every word in the dictionary, we check all the other words to see if there is one character
difference. If so, we create an edge between the two vertices. This takes $O(n^2)$ time. Overall, it takes $O(n^2)$ time to construct the graph.

Size: using adjacency matrix, it would be $O(n^2)$. using adjacency list, it would be $O(m) = O(n^2)$

**GRADING:**
1 point for correct explanation of time complexity. 1 point for correct explanation of space complexity.

---

(c) (1 point) Based on your answers to parts (a) and (b), give an algorithm that finds one valid transformation from the word $s$ to word $t$ in $O(n^3)$ time. Your algorithm should also be able to report if conversion is not possible. Note that any conversion suffices: you’re not required to find the shortest one.

**SOLUTION:**
the graph can be constructed by following the steps shown in (b). This takes $O(n^2)$ time.

sourceNode = getNode($s$) targetNode = getNode($t$)

if sourceNode is NULL or targetNode is NULL: not possible to transform else: BFS($s$) if $t.dist$ is infinity, $t$ is not reachable

otherwise, report that transform is possible

**GRADING:**
1 point for correct algorithm
8. ( /3 points) Cut Property

(a) ( /2 points) Consider some vertex $u$ in a weighted, undirected graph $G$. Using the Cut Property, prove that the minimum-weight edge from $u$ to one of its neighbors, aka

$$uv = \arg \min_{uv' \in E} w_{uv'},$$

is in the minimum spanning tree of $G$.

**SOLUTION:**
Take the cut defined by the single vertex $u$. This cut includes every edge from $u$ to a neighbor of $u$, and no other edges. By the Cut Property, the minimum-weight edge in this set is in the minimum spanning tree of $G$.

**GRADING:**
1 point for correct cut selection. 1 point for correct application of the Cut Property.

(b) ( /1 points) Recall that the cut property doesn’t deal with the case of ties, aka. when a cut contains two edges of minimum weight, neither of these two edges must be on the minimum spanning tree.

Give an example of a weighted, undirected graph with at most 3 vertices where $e$ is a minimum-weight edge, but there exists a minimum spanning tree that does not use $e$.

Note that there can be edges with the same weight.

**SOLUTION:**
Consider a graph where every edge has weight 1. Then every spanning tree has weight $n - 1$, and a tree that doesn’t use $e$ suffices.

**GRADING:**
1 point for correct example. Need to grade on a case-by-case basis.
9. (4 points) Coyote reloaded

Wile E. Coyote and The Road Runner are at it again! Coyote, starting at node $s$ of a directed graph, notices The Road Runner (which contrary to its name, is stationary) at node $t$. Each edge is directed, and takes $l_{u\rightarrow v}$ time to traverse.

This time, Wile E. Coyote has purchased a single rocket from the ACME cooperation (the equivalent of Amazon.com for coyotes). The rocket can be used at most once, and doubles the coyote’s speed between two nodes. That is, the Coyote can traverse an arc $u \rightarrow v$, in time $l_{u\rightarrow v}$ without using the rocket, and $l_{u\rightarrow v}/2$ with the rocket (after which it can no longer be used).

Your goal is to devise an $O(m \log n)$ time algorithm to find the quickest way to go from $s$ to $t$ with one rocket.

(a) (1 point) Suppose the best route calls for using the rocket along some edge $u \rightarrow v$. Show that the path taken by the coyote on this route before reaching $u$ must be a shortest $s \rightarrow u$ path.

**SOLUTION:**
The proof is by contradiction. Suppose not, then replacing the portion of the path from $s$ to $u$ by the shortest path would give a better solution.

**GRADING:**
1 point for any correct proof.

(b) (1 point) Recall that Dijkstra’s algorithm gives the lengths of the shortest $s \rightarrow u$ paths for all vertices $u$ in $O(m \log n)$ time. Show that we can modify the input graph and run Dijkstra’s algorithm on it once to find the lengths of the shortest $u \rightarrow t$ paths for all vertices $u$ in $O(m \log n)$ time.

**SOLUTION:**
Run Dijkstra’s algorithm with $t$ as the source vertex and all edge directions flipped.

**GRADING:**
1 point for correct modification of graph and application of Dijkstra’s algorithm.
(c) (2 points) **You may assume results of both parts a) and b) for this.** Give an \(O(m \log n)\) time algorithm for computing the shortest time it takes for the coyote to go from \(s\) to \(t\) using the rocket speed boost at most once.

**SOLUTION:**

**Layered graph approach:**

Split each vertex, labeling one as 'used' and the other as 'unused'. For any \(u \rightarrow v\) edge in the original graph, connect an edge from 'unused' \(u\) to both 'unused' \(v\) and 'used' \(v\), with weights \(l_{u \rightarrow v}\) and \(l_{u \rightarrow v}/2\) respectively. Connect 'used' \(u\) to 'used' \(v\) only with weight \(l_{u \rightarrow v}\). Run Dijkstra's algorithm with 'unused' \(s\) as the source, and return the shortest path to 'used' \(t\).

**Approach that follows from parts (a) and (b):**

Perform Dijkstra's algorithm with \(s\) as a source. Then, perform Dijkstra's algorithm again with all edges flipped with \(t\) as a source. For each edge, \(u \rightarrow v\) where \(u\) is reachable from \(s\), assume the coyote uses the rocket, and compute \(s \rightarrow t\) path length by adding shortest path length from \(s\) to \(u\), \(l_{u \rightarrow v}/2\), and shortest path length from \(v\) to \(t\). Choose the shortest \(s \rightarrow t\) path found. Total runtime is \(O(m \log n) + O(m \log n) + O(m) \cdot O(1)\), which is \(O(m \log n)\).

**GRADING:**

1 point for nearly correct algorithm.
2 points for correct \(O(m \log n)\) algorithm.