Solution: Final

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet.
- You have 180 minutes to earn a maximum of 180 points. Do not spend too much time on any one problem. **Read through all problems first**, then solve them in an order that allows you to make the most progress.
- **You are allowed three double-sided letter-sized sheet with your own notes**. No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. Pages will be scanned and separated for grading. If you need more space, write "Continued on S1" (or S2, S3, S4, S5, S6, S7) and continue your solution on the referenced scratch page at the end of the exam.
- Do not waste time and paper rederiving facts that we have studied in lecture, recitation, or problem sets. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. Be sure to **briefly** argue that your **algorithm is correct**, and analyze the **asymptotic running time of your algorithm**. Even if your algorithm does not meet a requested bound, you **may** receive partial credit for inefficient solutions that are correct.

Problem	Parts	Points
0: Information	2	2
1: Decision Problems	10	40
2: Network Upgrades	2	20
3: Sort by Number	2	18
4: Isomorphic Sentences	1	15
5: Sidestepping Smells	1	15
6: MonoMono Manors	1	15
7: Plant Peeping	1	20
8: JobBunny	1	15
9: DJ Ikstra Playlist	1	20
Total		180

Name: _____

School Email: _____

Problem 0. [2 points] **Information** (2 parts)

(a) [1 point] Write your name and email address on the cover page.
Solution: OK!

(b) [1 point] Write your name at the top of each page.
Solution: OK!

Problem 1. [40 points] **Decision Problems** (10 parts)

For each of the following questions, circle either **T** (True) or **F** (False), and **briefly** justify your answer in the box provided (a single sentence or picture should be sufficient). Each problem is worth 4 points: 2 points for your answer and 2 points for your justification. **If you leave both answer and justification blank, you will receive 1 point**.

(a) **T** **F** Selection sort always performs at least as many comparisons as insertion sort.

Solution: True. Selection sort always makes exactly $n(n-1)/2$ comparisons, while insertion sort makes between $n-1$ and $n(n-1)/2$ comparisons.

(b) **T** **F** If $T(n) = 4T(n/2) + O(n^2 \log^2 n)$ and $T(1) = O(1)$, then $T(n) = O(n^2 \log^2 n)$.

Solution: False. By case 2 of the Master theorem, $T(n) = O(n^2 \log^3 n)$.

(c) **T** **F** A max heap can be converted into a min heap in linear time.

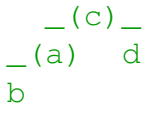
Solution: True. Building a min heap on any array takes linear time.

(d) **T** **F** Performing a single rotation on a binary search tree always results in binary tree that also satisfies the BST Property.

Solution: True. Rotations change the position and connections of nodes, but key order is maintained.

(e) **T** **F** If a node a in an AVL tree is not a leaf, then a's successor is a leaf.

Solution: False.



(f) **T** **F** When storing n integer keys in a hash table, using a hash function randomly chosen from a universal hash family, there will be no collisions in the hash table if all keys are distinct.

Solution: False. A hash function may map two distinct keys to the same slot.

(g) **T** **F** Johnson's Algorithm relaxes each edge in a graph $G = (V, E)$ at most $|V|$ times.

Solution: False. Bellman-Ford relaxes every edge at least $|V| - 1$ times, but each of the $|V|$ Dijkstras will also relax every edge. So each edge will be relaxed at least $2|V| - 1 > |V|$ times.

(h) T F Let dynamic programming subproblem $x(i)$ depend on subproblems $x(1), x(2), \ldots, x(i-1)$. In a top-down implementation, the value $x(i)$ is computed before $x(i-1)$, and in a bottom-up implementation, $x(i-1)$ is computed before $x(i)$.

Solution: False. In both implementations, $x(i-1)$ will be computed before $x(i)$ for all $i > 1$.

(i) T F Suppose an algorithm runs in $\Theta(ns)$ time for any input array A of n positive integers where $\max(A) = s$. Then the algorithm runs in polynomial time.

Solution: False. Algorithm A runs in pseudopolynomial time.

(j) T F If some NP-complete problem is EXP-hard, then P $\neq$ NP.

Solution: True. P $\neq$ EXP, so if the NP-complete problem is EXP-hard, it is not in P.

Problem 2. [20 points] **Network Upgrades**

Acrosoft Mizure is a cloud computing company that maintains a network of n computers. Mizure has connected p pairs of computers using wired links: a **wired link** is a direct two-way wired connection between two computers. The **wired connectivity** of a computer is the number of computers it can talk to via any sequence of wired links.

(a) [10 points] Describe an efficient[1] algorithm to find a largest subset of **existing** wired links that could be **removed** without changing any computer's wired connectivity.

Solution: Construct a graph on the computers, with an unweighted undirected edge between two computers if they share a wired link. Running Full-DFS on the graph constructs a DFS tree on each connected component of the graph. Each connected component of size k requires at least $k-1$ edges to remain connected, which each DFS tree achevies. So return each wired link not in any DFS tree, since removing it will not affect connectivity. Full-DFS takes $O(n+p)$ time, while constructing the output set takes $O(p)$ time.

(b) [10 points] Describe an efficient[1] algorithm to find a smallest set of **new** wired links that, if **added** to the network, would maximize each computer's wired connectivity.

Solution: Do the same thing as part (a) to construct a DFS tree for each of the k connected components of the graph. This graph requires the addition of $k-1$ edges to make the graph connected. There are many valid sets of $k-1$ edges one could add. One such set could be to connect the root of the first connected component's DFS tree to the root of every other connected component's DFS tree, and return the corresponding set of wired links. Again, Full-DFS takes $O(n+p)$ time, while constructing the output set takes $O(p)$ time.

Common Mistakes:

- Not defining your graph.
- Running BFS or DFS without accounting for multiple connected components (as in Full DFS).
- Counting edges instead of returning a subset.
- Running (Full) DFS but keeping forward edges instead of just tree edges (part (a)).
- Looking for a path for each component instead of a tree (part (a)).
- Connecting *every pair* of connected components using $\binom{k}{2}$ edges, when $k-1$ is enough (part (b)).

[1] By "efficient", we mean that faster correct algorithms will receive more points than slower ones.

Problem 3. [18 points] **Sort by Number**

In each part, describe an algorithm to sort n **pairs of integers** $(a_1, b_1), \ldots, (a_n, b_n)$ by increasing $f(a_i, b_i)$ value, for each function f described below. If your algorithms use division, they should only use **integer** division (Python's `//` and `%`).

(a) [8 points] If $f(a, b) = \left(1 + \frac{a}{n}\right) \cdot 2^b$, and all input pairs (a_i, b_i) satisfy $0 \leq a_i < n$ and $0 \leq b_i < n$, show how to sort pairs in $O(n)$ time.[2] Note that $f(a, b) < f(0, b + 1)$ for all valid a and b, because $1 + \frac{a}{n}$ is always less than 2.

Solution: Since $f(a, b) < f(0, b+1)$, then $f(a, b) < f(a', b')$ if and only if $bn + a < b'n + a'$. So convert each pair (a, b) into number $n < bn + a < n^2$, and then sort them in worst-case $O(n)$ time using radix sort.

(b) [10 points] If $f(a, b) = a/b$, and all input pairs (a_i, b_i) satisfy $0 \leq a_i < n$ and $0 < b_i < m < n$, then show how to sort pairs in $O(n \log m)$ time.

Solution: First run tuple sort on the n pairs by running (stable) counting sort on the a-coordinates and then (stable) counting sort on the b-coordinates. Because all $2n$ coordinates are in $[0, n)$, a size-n direct access array is sufficient, so this step takes $O(n)$ time. Tuple sort groups the tuples into $\leq m$ subarrays according to b-value, and within each subarray the a-values are sorted. (In other words, the values of $f(a, b)$ are sorted within each subarray.)

Then merge the m sorted lists into a single sorted list using a min heap as follows. Remove the smallest pair from each list and insert it into a min heap of size m in $O(m \log m)$ time, where pair (a, b) is less than (a', b') if and only if $ab' < a'b$. The minimum pair (a^*, b^*) in this heap is the minimum of the rationals remaining, so extract it and append it to an output list. Lastly, remove the smallest pair remaining from the list associated with b^* (if it exists), and insert it into the min heap. Repeating this process n times places each pair into the output list in sorted order in $O(n \log m)$ time.

Common Mistakes:

- Running radix sort on tuples, which is not defined (use tuple sort instead).
- Running tuple sort in the wrong order, or not specifying an order at all.
- Using tuple sort without specifying what stable sorting algorithm should be used on each coordinate.
- Trying to compute $f(a, b)$ directly, which falls outside the scope of the given integer operations.
- Grouping by b value and then running m different counting sorts. This uses $O(mn)$ time, because each counting sort reads the entire size-n direct access array.

[2] This function essentially corresponds to how computers represent floating-point numbers.

Problem 4. [15 points] **Isomorphic Sentences**

In this problem, a **string** is a sequence of lowercase English letters, and a **sentence** is a sequence of strings. Assume that each string is storable in a **constant number** of machine words. Two sentences $A = (a_1, \ldots, a_k)$ and $B = (b_1, \ldots, b_k)$ are **isomorphic** if they have the same ordered pattern of string repetitions, i.e., $a_i = a_j$ if and only if $b_i = b_j$, for all i, j. For example, sentence A below is isomorphic to sentence B, but A is not isomorphic to sentence C.

$$\begin{aligned} A = (&\texttt{ i}, &&\texttt{ like}, &&\texttt{ that}, &&\texttt{ you}, &&\texttt{ like}, &&\texttt{ algorithms})\\ B = (&\texttt{ my}, &&\texttt{ dog}, &&\texttt{ has}, &&\texttt{ many}, &&\texttt{ dog}, &&\texttt{ friends})\\ C = (&\texttt{ that}, &&\texttt{ example}, &&\texttt{ is}, &&\texttt{ a}, &&\texttt{ silly}, &&\texttt{ example}) \end{aligned}$$

Given a list of n sentences, each containing exactly k strings, describe an $O(nk)$-time algorithm to partition them into groups such that two sentences are in the same group if and only if they are isomorphic. Specify whether your running time is expected, amortized, or worst-case.

Solution: We map each sentence to a target tuple such that two sentences are isomorphic if and only if their target tuples are the same. For each sentence S, insert each string $w_i \in S$ for $1 \leq i \leq |S|$ in order into a hash table mapping strings to ranks. While inserting, if w_i is already in the hash table, skip it; otherwise if w_i is not in the hash table, map w_i to an integer j starting at 1, and then increase j by one. This procedure maps the jth unique string appearing in S to an integer rank $r(w_i) = j$, and does so in expected $O(k)$ time (hashing each string w_i takes expected $O(1)$ time). Then construct S's target tuple $(r(w_1), r(w_2), \ldots, r(w_{|S|}))$ by mapping each string w_i to its integer rank $r(w_i)$, also in expected $O(k)$ time. Constructing a target tuple for each sentence then takes expected $O(nk)$ time. If two sentences S_1 and S_2 are isomorphic, they will have the same target tuple. Alternatively, if two sentences S_1 and S_2 correspond to the same target tuple, then the mapping between the unique strings of S_1 and unique strings of S_2 through their ranks corresponds to a bijection between them, so they are isomorphic. So, construct a new hash table mapping each unique target tuple to an empty list (dynamic array). Then for each sentence S, append S to the list associated with its target tuple. Finally, return the list associated with each unique target tuple as a group, doing so in expected $O(nk)$ time.

Common Mistakes:

- Many students misunderstood how hash tables work, and lost points by incorrectly describing their internals instead of treating hash tables as a black box. Unless you're modifying the hash table data structure (which this problem does not call for), you shouldn't need to discuss collisions or chaining.
- Universal hash families do not guarantee the absence of collisions (see also Problem 1(f)).
- Hash tables cannot include more than one element with the same key. Inserting the same key a second time instead overwrites the original item.
- Misunderstanding the meaning of "isomorphic", e.g., assuming that only one word can appear more than once, or that each word appears at most twice.
- After constructing a hash table for each sentence, trying to hash or sort these hash tables directly instead of turning them into target tuples (or similar).

Problem 5. [15 points] **Sidestepping Smells**

Cinnamon City consists of n locations where pairs of locations are connected by two-way roads, and each location is incident to at most 5 roads. Some locations contain a trash dumpster. Each dumpster d_i has a different **smell radius** r_i. Someone at a location can **smell** a dumpster d_i if and only if they can walk from their location at most r_i feet along roads to d_i. Orcas the Grouch wants to walk across town while avoiding the smell of trash from dumpsters. Given a map of Cinnamon City marked with the length of each road, and the location and smell radius of each dumpster, describe an efficient[3] algorithm to find the shortest route from 600 Rosemary Rd. to 6 Allspice Ave., along which Orcas will never smell trash, if such a route exists.

Solution: Construct a graph on the n locations in Cinnamon City, with an undirected edge between two locations with weight w if the locations are connected by a two-weight road of length w. Next, we identify all locations within the smell radius of any dumpster. Let R be the maximum smell radius of any dumpster. Construct a new vertex x, and to each dumpster location v having smell radius r, add edge (x, v) with weight $R - r$. Then run Dijkstra from x to find all locations having distance R from x. Such locations are within the smell radius of some dumpster, so remove them from the graph. Then run Dijkstra on the modified graph from 600 Rosemary Rd. to find a shortest path to 6 Allspice Ave. Two runs of Dijkstra takes $O(n \log n)$ time, since vertices have constant-bounded degree.

Common Mistakes: Many incorrect or slow approaches to finding nodes to remove, such as:

- "Branching out" dumpsters: if dumpster d_i has a weight w edge, then put a new dumpster of smell radius $r_i - w$ at the other end, and repeat. This can work, but it needs more care to be fast, and then it just turns into Dijkstra.
- Supernode with edge-weights of zero, which doesn't account for differing smell radii.
- Making the graph unweighted with edge duplication; this only works if edge weights are known to be small.
- Interpreting smell radius *geometrically* (i.e., easily checked with coordinates of some kind) instead of a condition along the graph.
- Run Dijkstra from a dumpster, stop once its smell radius is reached, delete all discovered nodes from the graph, and then iterate for all other dumpsters. This does not work, because deleting vertices earlier may separate other nodes from dumpsters they can smell. If all deletion is saved for the end, then n rounds of Dijkstra is correct but inefficient.

[3] By "efficient", we mean that faster correct algorithms will receive more points than slower ones.

Problem 6. [15 points] **MonoMono Manors**

The new **single-player** board game *MonoMono* consists of n **manors** $M = \{m_0, m_1, \ldots, m_{n-1}\}$ arranged in a directed cycle. The player begins at manor m_0 with d dollars (where d is a positive integer), and then repeatedly takes a turn. Each turn, the player moves between 2 and 12 spaces forward along the directed cycle, based on a roll of the dice. For example, if $n = 10$ and the first two rolls are 6 and 8, then the player will first move to m_6 and then move to m_4. The player owns a known subset $M_p \subset M$ of manors, and manor ownership never changes. Each manor m_i has an integer **transaction cost** c_i: when the player moves to manor m_i, the player gains c_i dollars if the player owns the property, and loses c_i dollars otherwise. Describe an $O(nd)$-time algorithm to determine the minimum number of turns the player could possibly take to at least **double their starting money, without ever going bankrupt** (having a negative balance).

Solution: Construct a graph G on $n(2d) + 1$ vertices: a vertex t, and a vertex $v(i, j)$ for each $0 \le i < n$ and $0 \le j < 2d$ corresponds to being at manor i with j dollars. For each vertex $v(i, j)$ add a directed unweighted edge from $v(i, j)$ to $v(k, j + c_k)$ for $k \in \{(i + r) \bmod n \mid 2 \le r \le 12\}$ if $0 \le j + c_k < 2d$, and from $v(i, j)$ to t if $j + c_k \ge 2d$. Then any path from $v(0, d)$ to t corresponds to a sequence of turns that at least doubles starting money without ever going bankrupt. So run a breadth-first search from $v(0, d)$ to find the length of a shortest such path, i.e., the minimum number of turns needed to satisfy the goal. This graph has $O(nd)$ vertices and at most $11 = O(1)$ outgoing edges per vertex, so BFS runs in $O(nd)$ time.

Common Mistakes:

- Using Bellman-Ford to detect negative-weight / positive-profit cycles doesn't work for two reasons:
  - Negative-weight cycles aren't necessary to reach $2d$.
  - Bellman-Ford cannot detect bankruptcy: Bellman-Ford only measures net monetary gain, so it doesn't know whether the player's wealth *temporarily* goes negative.
- Straightforward dynamic programming attempts lead to cyclic dependencies, so cannot lead to a finite algorithm.
- Smaller graph duplication mistakes:
  - Not describing the duplicated graph clearly enough.
  - End condition should be $\ge 2d$, not $= 2d$.
  - Not omitting edges leading to negative-balance nodes.

Problem 7. [20 points] **Plant Peeping**

Nevergreen Terrace is a straight flat street that starts at the dangerous Fallfield Nuclear Power Plant located at address 0. Tall buildings are frequently built and demolished along Nevergreen Terrace, and a building's property value is higher when another building shields it from view of the plant. Specifically, a building b can **see** the power plant if b is **strictly taller** than every building strictly between b and the power plant. Describe a database to maintain the set of buildings along Nevergreen Terrace, supporting the following three operations each in worst-case $O(\log n)$ time, where n is the number of buildings on Nevergreen Terrace at the time of the operation.

- `construct(b)`: record a new building b with height $b.h$ (an integer number of meters) at address $b.a$ (an integer number of meters from the power plant).
- `demolish(a)`: remove the building at address a from the database.
- `can_see_plant(a)`: return `True` if the building at address a is **strictly taller** than every building having positive address less than a, and `False` otherwise.

Solution: Store each building b with height $b.h$ and address $b.a$ in an AVL tree keyed on address. Augment each node v with $v.m$, the maximum height of any building in its subtree. These values can be computed in constant time from v's children using $v.m = \max\{v.left.m, v.right.m, v.b.h\}$, where missing children should be omitted; so this augmentation can be maintained without affecting asymptotic runtimes of AVL tree operations. The `construct` and `demolish` operations are the usual AVL insert and delete operations (while maintaining the augmentation), which each require worst-case $O(\log n)$ time.

Define a helper query `max_before`(v, a) which finds, in v's subtree, the maximum height of any building with address strictly less than a. If $v.b.a \geq a$ then all nodes with valid addresses are in v's left subtree, so `max_before`(v, a) equals `max_before`$(v.left, a)$ (or $-\infty$ if v has no left child). Otherwise, when $v.b.a < a$,

$$\texttt{max\_before}(v, a) = \max\{v.left.m,\ v.h,\ \texttt{max\_before}(v.right, a)\},$$

because v and all nodes in $v.left$ are guaranteed to have address less than a. (The recursive call can be omitted if v has no right child.) In either case, `max_before`(v, a) makes at most one recursive call to a child of v and otherwise does $O(1)$ work, so this query takes $O(\log n)$ time overall.

Finally, to compute `can_see_plant(a)`, first AVL find to locate building b with address a, and then call `max_before`(r, a), where r is the root of the AVL tree, which returns the maximum height m of all buildings between the power plant and b. Then v can see the power plant if and only if $b.h > m$. This requires worst-case $O(\log n)$ time.

Common Mistakes:

- Augmenting each AVL node by storing the tallest building to its left. This is not efficiently updateable: for example, if a very tall building is built at address 1, then all n buildings must update their augmentation.
- Assuming that the only nodes with smaller key than some building b are in b's left subtree. This is false for most nodes.

- Comparing $b.h$ to the max height in the left subtree of *every* ancestor of b, instead of only those ancestors with key less than $b.h$. Testing against every node erroneously includes nodes to the right of b.
- Augmenting by the max height in each node's *left* subtree only. This cannot be updated in $O(1)$ directly from the node's children—it requires $O(\log n)$ time per update (you must walk down-right as far as possible), meaning each AVL insertion/deletion takes $O(\log^2 n)$ time. If you also augment by the max height in each node's *right* subtree, it is efficiently updateable again.
- Not explaining how to update your augmentation (see previous bullet).
- Using hash tables with or instead of an AVL tree, which eliminates the possibility of worst-case bounds.

Problem 8. [15 points] **JobBunny**

Rager Robbit is a freelancer who works independent jobs posted on the new job website, JobBunny. There are n jobs currently listed on the site. Each job has a start date, an end date, and the number of dollars that Rager would earn by working on that job every day from its start date up to and including its end date. Multiple jobs may have the same start or end date. Rager is bad at multitasking, so he cannot work multiple jobs on the same day. Describe an $O(n \log n)$-time **dynamic-programming** algorithm to determine which JobBunny jobs Rager should work on in order to maximize his earnings. Slower correct algorithms will receive partial credit.

Solution:

1. **Subproblems**
   - First, sort the jobs by start date in $O(n \log n)$ time, e.g. via merge sort
   - Store the sorted jobs in an array A from job $A[0]$ to $A[n-1]$
   - Let the start, end, and earnings of job $A[i]$ be $A[i].s$, $A[i].e$, and $A[i].v$ respectively
   - $x(i)$: maximum earnings possible working only jobs from suffix $A[i:]$
2. **Relate**
   - Either maximum earnings uses job $A[i]$ or not
   - $x(i) = \max\{A[i].v + x(k), x(i+1)\}$ where $k = \min\{j \mid i < j < n \text{ and } A[i].e < A[j].s\}$
   - $x(i) \leq x(j)$ iff $i > j$, so minimizing k will find a largest $x(k)$
   - Subproblems $x(i)$ only depend on strictly larger i so acyclic
3. **Base**
   - $x(n) = 0$ (no jobs, no earnings)
4. **Solution**
   - Solution is $x(0)$, the maximum earnings considering all jobs
5. **Time**
   - # subproblems: n.
   - Work per subproblem naïvely is $O(n)$, leading to $O(n^2)$ running time
   - Since A is sorted by start time, you can find k using binary search in $O(\log n)$ time
   - This optimization leads to $O(n \log n)$ total running time

Common Mistakes:

- Not following parent pointers to reconstruct the selected jobs.
- Iterating over all days in range (which leads to a pseudopolynomial algorithm) instead of only the n given start dates.
- Not sorting by start date but still only considering the jobs in order.
- Storing an explicit set of jobs as a subproblem value or parameter, leading to inefficient copying or manipulating, or exponentially many subproblems.

Problem 9. [20 points] **DJ Ikstra Playlist**

DJ Ikstra is preparing a playlist for a music festival. She has a list of n songs in her library, and has tagged each song with its duration in integer seconds, its genre (either house, techno, or dubstep), and its **awesomeness**: a positive integer indicating how awesome she thinks it is (no two songs are equally awesome). The festival organizers have allocated only r seconds for her playlist, which will consist of a sequence of songs played back to back. DJ Ikstra wants her playlist to be both elevating and diverse. A playlist is **elevating** if the songs in playlist order never decrease in awesomeness; and is **diverse** if no two consecutive songs belong to the same genre. Describe an $O(n \log n + nr)$-time **dynamic-programming** algorithm to find an elevating and diverse playlist lasting at most r seconds that maximizes the total awesomeness of all songs played.

Solution:

1. **Subproblems**
   - First, sort songs by awesomeness into array A, from $A[0]$ to $A[n-1]$ in $O(n \log n)$ time
   - Let time, genre, and awesomeness of song $A[i]$ be $A[i].t$, $A[i].g$ and $A[i].a$ respectively
   - Every elevating playlist will be a subsequence of A
   - $x(i, j, g)$: maximum awesomeness of any elevating diverse playlist from suffix $A[i:]$ using at most j seconds, where first song does not have genre g
2. **Relate**
   - Either playlist uses song $A[i]$ or not
   - $x(i, j, g) = \max \left\{ \begin{array}{ll} A[i].a + x(i+1, j - A[i].d, A[i].g) & \text{if } A[i].g \neq g \\ x(i+1, j, g) & \text{always} \end{array} \right\}$
   - Subproblems $x(i, j, g)$ only depend on strictly larger i so acyclic
3. **Base**
   - $x(n, j, g) = 0$ for $j \geq 0$ (no songs, no awesomeness)
   - $x(i, j, g) = -\infty$ for $j < 0$ (can't fill negative time)
4. **Solution**
   - Solution is $x(0, r, None)$, the maximum awesomeness considering all songs
5. **Time**
   - # subproblems: $O(nr)$.
   - Work per subproblem is $O(1)$
   - $O(n \log n) + O(nr) = O(nr)$ total running time

Common Mistakes:

- Referencing information not present in the subproblems or relation, such as which songs have been used.
- Storing an explicit set of songs as a subproblem value or parameter, leading to inefficient copying or manipulating, or exponentially many subproblems.

- Not tracking the latest genre and/or awesomeness in subproblems, meaning the required conditions cannot be enforced.
- Not sorting songs by awesomeness but still only reading songs in linear order.
- Making greedy choices such as "break awesomeness ties with the song that has shortest duration", which is not always correct.
- Trying to use Dijkstra, possibly because the runtime looks like a Dijkstra runtime.

SCRATCH PAPER 1. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S1" on the problem statement's page.

SCRATCH PAPER 2. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S2" on the problem statement's page.

SCRATCH PAPER 3. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S3" on the problem statement's page.

SCRATCH PAPER 4. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S4" on the problem statement's page.

SCRATCH PAPER 5. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S5" on the problem statement's page.

SCRATCH PAPER 6. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S6" on the problem statement's page.

SCRATCH PAPER 7. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S7" on the problem statement's page.