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. Skim them all first, and attack them in the 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 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.
- **Pay close attention to the instructions for each problem**. Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points
0: Information	2	2
1: Decision Problems	10	40
2: Alternate Sorts	2	16
3: Biscuit Brute	1	15
4: Blue Friends	1	15
5: Trek Timing	1	20
6: Boxing Books	1	16
7: Parity Proximate	1	18
8: Minion Matching	1	18
9: Cache Cost	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** If $T(n) = 2T(n/4) + \sqrt{n}\log n$ with $T(1) = \Theta(1)$, then $T(n) = \Theta(\sqrt{n}\log n)$.

Solution: False. This is case two of Master's Theorem. Work is balanced at each level, so an extra log factor is needed, leading to $\Theta(\sqrt{n}\log^2 n)$.

(b) **T** **F** A $\Theta(n^{10})$-time algorithm will always take at least as long to run as a $\Theta(\log n)$-time algorithm on a problem with input size n.

Solution: False. Only true in the limit of large n. Constant factors could make a $\Theta(n^{10})$-time algorithm faster than a $\Theta(\log n)$-time algorithm for small n.

(c) **T** **F** Given an array of n distinct comparable integers, we can identify and sort the $\frac{n}{\log n}$ smallest of them in $O(n)$ time using a heap.

Solution: True. We can build a heap on array in $O(n)$ and pop the k largest in order in $O(k\log n)$ time. For $k = n/\log n$, this running time is $O(n)$.

(d) **T** **F** Given k distinct integer keys, there exists a binary search tree containing all k of them that satisfies the max heap property.

Solution: True. A chain to the left where each subtree is rooted at its max element.

(e) **T** **F** Checking if a string of length k exists in a hash table takes worst-case $O(k)$ time.

Solution: False. Hash table operations are expected time operations.

(f) **T** **F** Using an AVL tree as Dijkstra's priority queue results in the same asymptotic running time as using a Fibonacci heap, when running Dijkstra's algorithm to compute single source shortest paths on a graph with n vertices and $O(n)$ edges.

Solution: True. An AVL tree will yield a running time of $O(|V| \log |V|)$ on sparse graphs, which is asymptotically equivalent to Dijkstra using a Fibonacci heap.

(g) **T** **F** Depth-first search solves single-source shortest paths in an unweighted, directed graph $G = (V, E)$ in $O(|V| + |E|)$-time.

Solution: False. A complete graph on three vertices is a counter example.

(h) T F Given a connected weighted directed graph having positive integer edge weights, where each edge weight is at most k, we can compute single source shortest paths in $O(k|E|)$ time.

Solution: True. Replace each edge (a, b) with weight w with a directed unweighted path from a to b, and run BFS from the source.

(i) T F If **P** = **NP**, it follows that **NP-Hard** = **EXP**.

Solution: False. For so many reasons. For example, Halting problem is in **NP-Hard** but not in **EXP**.

(j) T F Let A be a problem that cannot be solved in polynomial time, and let ALG be an algorithm with optimal running time $T(n)$ to solve a size n instance of another problem B. If there is an algorithm to solve a size m instance of problem A in time $O(m^c) + T(m^d)$ for some constants c and d by using ALG, then problem B also cannot be solved in polynomial time.

Solution: True. If $T(n)$ were polynomial in n, then $O(m^c) + T(m^d)$ would be polynomial in m, meaning problem A could be solved in polynomial time, a contradiction.

Problem 2. [16 points] **Alternate Sorts**

(a) [8 points] Given an array A containing n integers, where each integer is in the range $\{1, \ldots, n^m\}$, describe an $O(n \cdot \min\{m, \log n\})$-time algorithm to sort A.

Solution: Branch on the value of m. If $m < \log n$, sort using radix sort in $O(nm) \leq O(n \log n)$ time; otherwise, use merge sort to sort in $O(n \log n) \leq O(nm)$ time. This yields the desired running time $O(n \cdot \min\{m, \log n\})$.

Common Mistakes:

- Not comparing m and $\log n$ before running algorithms
- i.e., saying to just use the fastest, which doesn't let you branch preemptively
- Comparing m with a value besides $\log n$, i.e., checked whether m was constant

(b) [8 points] Recall that an array of **distinct** integers is ***k-proximate*** if every integer of the array is at most k places away from its place in the array after being sorted[1]. Describe an efficient[2] **in-place** algorithm to sort a k-proximate array.

Solution: Insertion sort can sort a k-proximate array in-place in $O(nk)$ time, but we can do better!

Apply heap sort to sort the first $2k$ elements in place in $O(k \log k)$ time. Since the array is k-proximate, the first k elements are now guaranteed to be smallest k elements in sorted order. Now repeatedly use heap sort to sort items $k \cdot i$ to $k(i+2)$ in place, which maintains the invariant that before iteration $i > 0$, items 0 to $k \cdot i - 1$ are sorted and items $k \cdot i$ to n is k-proximate, until $k(i+2) > n$, or when $i = \lceil n/k - 2 \rceil$. Each of the $O(n/k)$ loops takes $O(k \log k)$ time, leading to $O(n \log k)$ time in total.

Common Mistakes:

- Correct in-place $O(nk)$ algorithms got lots of partial credit here
- $O(n \log k)$ algorithms that were not in-place received less partial credit

[1] If the ith integer of the unsorted input array is the jth largest integer contained in the array, then $|i - j| \leq k$.

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

Problem 3. [15 points] **Biscuit Brute**

The Biscuit Brute is a sleepy monster who lives in the mountains and loves to eat biscuits. Fortuitously, the mountains are home to a community of n bakers who bake lots of biscuits. If the Brute ever visits the home of a baker, the baker will give the Brute exactly k biscuits to eat. Not wanting to take advantage of their generosity, after receiving biscuits from one baker, the Brute will always travel to another baker to beg for biscuits before returning.

The Brute has an integer **sleepiness**. Sleepiness increases by 1 each minute the Brute is awake, and decreases by 10 whenever the Brute eats a biscuit. For every ordered pair of bakers (b_i, b_j), the Brute knows how many minutes m_{ij} it will take to travel from baker b_i to baker b_j (assume it takes no time to receive or eat biscuits). Describe an efficient **polynomial-time** algorithm to determine the minimum sleepiness the Brute can have to reach the house of baker b_n, starting from the house of baker b_1 with sleepiness -10.

Solution: Construct a directed graph with bakers as vertices, and a directed edge (b_i, b_j) from b_i to b_j weighted by $m_{ij} - 10k$. This graph has n vertices and $O(n^2)$ edges. The weight of any path in this graph corresponds to the minimum increase in Brute sleepiness by traversing that path, so we want to find the minimum weight path from b_1 to b_n in this graph. Since edge weights can be negative, run Bellman-Ford from b_1 in $O(n^3)$ time and return 10 less than the weight of the minimum path found to b_n (if the minimum weight path is $-\infty$, then the Brute can acheive arbitrarily low sleepiness).

Common Mistakes:

- Assuming $m_{ij} = m_{ji}$ which is a different problem. Pairs are **ordered**!
- Not including the -10 sleepiness at start
- Not interpreting what a negative cycle means for Brute sleepiness

Problem 4. [15 points] **Blue Friends**

A **blue-red** graph is a **simple**[3] undirected graph where every edge is colored either red or blue. A blue-red graph is **blue friendly** if every pair of vertices is reachable from each other via a path of only blue edges. Given a (simple) blue-red graph $G = (V, E)$ that is not blue friendly, describe an $O(|V|^2)$-time algorithm to determine whether you can add blue edges to G to make a new (simple) blue-red graph G' that is blue friendly.

Solution: Adding blue edges only improves connectivity along blue edges, so it suffices to check whether adding all blue edges leads to a blue friendly graph. Construct a new graph G' which is the complete graph on all the vertices, but remove any undirected edge $\{a, b\}$ if it appears as a red edge in G. Graph G' has $|V|$ vertices and at most $O(|V|^2)$ edges, and can be constructed naively in $O(|V|^2)$ time. We can add edges to G to make all vertices reachable from each other along blue edges if and only if G' is connected, so run BFS or DFS (Full- is also fine here) to determine whether G' is connected in $O(|V| + |V|^2) = O(|V|^2)$ time.

Common Mistakes:

- Adding blue edges where red edges already exist
- Only connecting each component to at least one other component
- Missing details on how to ensure a single blue friendly component at end

[3] Recall, a graph is simple if every edge in the graph connects two distinct vertices, and there is at most one edge connecting any vertex pair.

Problem 5. [20 points] **Trek Timing**

Steryl Chrayed wants to visit Old Loyal, the geyser in Bluerock National Park which **erupts** exactly on the hour every 60 minutes. Steryl has a map showing the m two-way trails connecting n hiking junctions in the park. Each trail t_i is marked with its positive integer **length** ℓ_i measured in hundreds of meters, and its positive integer **difficulty** d_i. Steryl will leave at precisely 6:00 a.m. from the parking lot at junction p, and hike to Old Loyal at junction g, but wants to do so by hiking at a constant pace of exactly one hundred meters per minute along trails without stopping[4], so that Old Loyal erupts precisely when she first arrives at g. Given Steryl's map, describe an efficient **polynomial-time** algorithm to determine whether it is possible for Sheryl to do this, and if so, return such a hiking path that minimizes the sum of trail difficulties hiked.

Solution: Construct a graph G having $O(n)$ vertices and $O(m)$ edges: for each hiking junction v, add 60 vertices v_i for $i \in \{0, \ldots, 59\}$, and for each trial connecting junctions a and b having length $\ell_{(a,b)}$ and difficulty $d_{(a,b)}$, add two directed edges for each $i \in \{0, \ldots, 59\}$, from vertex a_i to vertex b_j and from vertex b_i to vertex a_j weighted by $d_{(a,b)}$, where $j = \mathrm{mod}(i + \ell_{(a,b)}, 60)$ (except do not include any outgoing edges from any vertex associated with junction g as we do not want Steryl to be able to arrive more than once at g). Each vertex v_i corresponds to arriving at a junction v at i minutes past the hour, and an edge connects vertices a_i and b_j if Steryl can leave junction a at i minutes past the hour along a path of difficulty $d_{(a,b)}$ to arrive at junction b at j minutes past the hour at her constant pace, so any path in G from vertex p_0 to vertex g_0 corresponds to a hiking path to reach Old Loyal precisely at eruption time starting from the parking lot on the hour, where the path's weight corresponds to the sum of trail difficulties hiked along the path. Graph G may have cycles but has only positive edges weights, so run Dijkstra from p_0 and return the shortest path weight to g_0 in $O(m + n \log n)$ time.

Common Mistakes:

- Not removing outgoing edges out of g to ensure only visit g once
- Not looping to allow paths longer than one hour
- Assuming some constraint such as max 24 hours
- Tried to use dynamic programming by assuming a DAG

[4]Steryl will always continue at the same pace in the same direction along a trail until reaching a junction.

Problem 6. [16 points] **Boxing Books**

The semester has ended, and Tim the Beaver needs to pack up the books in his dorm room. All of Tim's books have the same height and width, but each book has a different thickness proportional to the positive number of pages in the book. Tim has **three** identical boxes that can each fit at most m pages of books. Given a list $P = (p_1, \ldots, p_n)$ of the page length of each of his n books, describe an $O(nm^2)$-time dynamic programming algorithm to determine whether Tim can fit all of his books into his boxes. (Each book must fit into one of the boxes; do not break Tim's books!)

Solution:

1. Subproblems
   - Label the boxes as box 1, box 2, and box 3
   - $x(i, j, k)$: a Boolean which is True if two disjoint subsets of the book pages from p_1 to p_i can sum to j and k respectively (to be placed in boxes 1 and 2), and False otherwise.
2. Relate
   - We make a choice of which of boxes 1, 2, or 3 to place book pages p_i
   - $x(i, j, k) = \text{OR} \left\{ \begin{array}{ll} x(i-1, j-p_i, k) & \text{if } j \geq p_i \\ x(i-1, j, k-p_i) & \text{if } k \geq p_i \\ x(i-1, j, k) & \text{always} \end{array} \right\}$
   - Subproblems $x(i, j, k)$ depends only on smaller i so acyclic
3. Base
   - $x(0, 0, 0) =$ True (no more items and no more pages to fill)
   - $x(0, j, k) =$ False for $j > 0$ or $k > 0$ (no more items, cannot fill $j + k$ pages)
4. Solution
   - Solve subproblems for $i \in \{0, \ldots, n\}$ and $j, k \in \{0, \ldots, m\}$
   - Can pack books into boxes if there is a True $x(n, j, k)$ such that $\left(\sum_{p \in P} p\right) - j - k \leq m$
   - So loop through all subproblems and evaluate this condition
5. Time
   - $O(nm^2)$ subproblems
   - $O(1)$ work per subproblem
   - $O(nm^2)$ work total (including $O(nm^2)$ work to evaluate condition in solution step)

Common Mistakes:

- Making a DP with capacities for all three boxes (inefficient)
- Not covering when capacities were exceeded
- Running subset sum twice to try and break the set into three subsets
- Filling one box at a time in the DP

Problem 7. [18 points] **Parity Proximate**

A sequence of integers is **parity proximate** if every even integer in the sequence is adjacent to another even integer, and every odd integer in the sequence is adjacent to another odd integer. Given a sequence of integers $A = (a_1, \ldots, a_n)$, describe an $O(n)$-time dynamic programming algorithm to determine the maximum sum of any parity proximate subsequence of A. For example, if $A = (4, 3, 6, 5, 7, 2, 1, 4)$, then the subsequence $(4, 6, 5, 7, 2, 4)$ is parity proximate and has sum 28, which is maximum over all parity proximate subsequences of A.

Solution:

1. Subproblems
   - $x(i, -1)$: max sum of any parity proximate subsequence of $(a_1, \ldots, a_i)$
   - $x(i, j)$: max sum of any parity proximate subsequence of $(a_1, \ldots, a_i)$, ending in a number which is equal to j modulo 2, which may be alone, for $j \in \{0, 1\}$
2. Relate
   - If $j = -1$, unrestricted in choice; may take or leave element a_i
   - If $j \neq -1$, restricted in choice; still may take or leave element a_i
   - $x(i, -1) = \max\{x(i-1, -1), a_i + x(i-1, \text{mod}(a_i, 2))\}$
   - $x(i, j \in \{0, 1\}) = \max \left\{ \begin{array}{l} x(i-1, j), \\ a_i + \max\{x(i-1, -1), x(i-1, j)\} \quad \text{if } \text{mod}(a_i, 2) = j \end{array} \right\}$
   - Subproblems $x(i, j)$ only depend on smaller i so acyclic
3. Base
   - $x(0, -1) = 0$ (no more items, so best is no more sum)
   - $x(0, j) = -\infty$ for $j \in \{0, 1\}$ (no more items, so can't choose one!)
4. Solution
   - $x(n, -1)$
5. Time
   - $O(n)$ subproblems
   - $O(1)$ work per subproblem
   - $O(n)$ work total

Common Mistakes:

- Subproblems do not track parity at all
- Subproblems are contradictory (defines a parity proximate subsequence, but allows ending in a single even or odd)
- Relation does not guarantee parity proximate
- Initializing all base cases to the same output (when different outputs are needed)
- Outputting wrong solution relative to defined subproblems

Problem 8. [18 points] **Minion Matching**

Supervillain Shrunk leads a team of social minions. Each minion has a unique **name**: a string of at most k lowercase English letters. Each pair of minions has at most two **buddy names**, formed by the concatenations of their names in either order (note it is possible for a pair of minions to have only one buddy name if both concatenations are the same). Four minions are **buddy buddies** if a buddy name of two of them is identical to a buddy name of the other two. For example, minions `'alex'`, `'isabel'`, `'alexis'`, and `'abel'` are buddy buddies.

Given a list $A = (a_1, \ldots, a_n)$ containing the names of Shrunk's n minions, describe an $O(kn^2)$-time algorithm to determine whether any four of them are buddy buddies. State whether your algorithm's running time is worst-case, amortized, and/or expected.

Solution: For each of the $O(n^2)$ pairs of minions, compute their two buddy names via string concatenation each in $O(k)$ time. Then for each buddy name m formed by concatenating the name of minion a to minion b, insert m into a hash table, mapping each buddy name to a linked list or dynamic array storing every (a, b) pair of minions who combined to form that buddy name. Since each of the $O(n^2)$ insertions takes amortized expected $O(k)$ time, all these operations can be performed in expected $O(kn^2)$ time.

Note that two minion pairs hashing to the same buddy name does not guarantee the existence of buddy buddies, so we must do some work to find them. For each buddy name m stored in the hash table linking to a list of minion pairs P_m, we claim that if m is the common buddy name of some buddy buddies, then some pair of the first three minion pairs in P_m are buddy buddies. Suppose for contradiction no pair of the first three minion pairs in P_m are buddy buddies. If no pair of them are buddy buddies, then every pair must share a minion in common. Let (a, b) be the first minion pair. Then either the other two pairs are (c, a) and (b, d) for some $c, d \notin \{a, b\}$, or the other two pairs are (b, a) and (c, d) for some $c, d \notin \{a, b\}$. In the first case, (c, a) and (b, d) are distinct, so are buddy buddies, and in the second case, (a, b) and (c, d) are distinct, so are buddy buddies, a contradiction. Thus, for each list P_m, it suffices to check whether any pair of the first three pairs of minions form buddy buddies in $O(k)$ time, so this algorithm runs in expected $O(kn^2)$.

Note that this problem can also be solved in worst-case $O(kn^2)$ by computing all buddy names and sorting them in worst-case $O(kn^2)$ time via a tuple sort on the strings with counting sort as the stable subroutine.

Common Mistakes:

- Using a histogram or frequency tables of letter counts (order matters here!)
- Not accounting for a buddy pairs generatable by non-disjoint pairs:
- i.e., 'a' and 'aa' both make 'aaa', and 'ab', 'aba', 'ba' can create 'ababa' using (1,2) or (2,3), so checking a single pair forming the same string forward and backwards is insufficient
- Claimed hashing a length k string is $O(1)$ time
- Tried to use dynamic programming (incorrectly)

Problem 9. [20 points] **Cache Cost**

A computer program accesses n variables $V = (v_1, \ldots, v_n)$ over time. When a program accesses a variable v, we would like to know the **cache cost** of the access: specifically, the number of **distinct** variables accessed since variable v was last accessed (if v has not been accessed before, its access cost is zero). For example, if a program accesses variables $V = (a, b, c, d)$ in the order (a, b, c, d, b, a, b), then the cache cost of the accesses are:

access #	1	2	3	4	5	6	7
variable	a	b	c	d	b	a	b
cache cost	0	0	0	0	2	3	1

Accesses 1-4 are first accesses so each have cost 0; access 5 has cost 2 because c and d are accessed between the first two accesses of b; access 6 has cost 3 because b, c, and d are accessed between accesses of a; and access 7 has cost 1 because a is accessed between the last two accesses of b.

Describe a dynamic data structure to store variable accesses over time from the start of a program, supporting one operation, `access(`v`)`, which accesses variable v and returns the cost of the access. The operation should run in $O(\log n)$ time where n is the number of variables accessed by the program. State whether your operation running time is worst-case, amortized, and/or expected.

Solution: Maintain a counter keeping track of the next access number which will be $n = 0$ at the start of a program. Maintain variable accesses using a variable dictionary (e.g., a hash table or AVL tree) mapping already accessed variables to their most recent access number (we assume that variable names are hashable or comparable in constant time), and an access AVL tree storing keys of the most recent access number for each variable already accessed, where each node x in the access AVL tree having key $x.k$ is augmented with the size $x.s$ if its subtree. $x.s$ can be maintained in constant time from the augmentations of its children via the relation $x.s = 1 + x.left.s + x.right.s$ (if the left and right children exits).

To implement the `access(`v`)` operation, first increment the access counter by adding 1 to n, and check whether v is already in the variable dictionary. If it is not, add v to the dictionary mapping to the current access number n and add n to the access AVL, returning 0 since this is variable v's first access. Alternatively, v has been accessed previously, so look up variable v's last access time n' in the variable dictionary, and then replace the last access value with n. We need to return the number of distinct variables accessed since variable v's last access time n', so perform a one-sided range query on the access AVL. Specifically, recursively compute $p(x, n')$, the number of nodes in x's with access time above n' as follows:

$$p(x, n') = \begin{cases} p(x.right, n') & \text{if } n' > x.k \\ 1 + x.right.s + p(x.left, n') & \text{if } x.k \leq n' \end{cases}$$

Return $p(r, n')$ where r is the root of the access AVL tree, which can be computed in $O(\log n)$ time as at most one recursive call is made down the tree. Lastly, remove n' from the access AVL tree and insert n. This algorithm runs in worst-case $O(\log n)$ time if an AVL tree is used for the variable dictionary, and in expected $O(\log n)$ time if a hash table is used.

Common Mistakes: See S1

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.

Common Mistakes:

- Counting all accesses instead of distinct accesses
- Not mapping variables to access number to find a variable's node in access number AVL
- Forgetting to delete nodes from AVL
- Making augmentations that can't be maintained in constant time per node
- Forgetting to return 0 on first access
- Attempting range query on access number using AVL keyed by variable
- Underspecified range query

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.