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: Sorting Sorts	2	24
3: Pams vs. Ratriots	1	14
4: Costly Campaigning	1	16
5: Parkour Performance	1	16
6: Mostly-Blue	1	16
7: Zero Sum	2	32
8: Soldier Supply	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 $f(n) \in O(3^{2n})$, then $f(n) \in \Omega(2^{3n})$.

Solution: False. Counter example: $f(n) = n$ is in $O(3^{2n})$ but is not in $\Omega(2^{3n})$.

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

Solution: False. Counter example: $T(n) = n \lg n$ satisfies the recurrence since $n \lg n = n \lg(n/2) + n$ where $n \in O(n^2)$, and $n \lg n$ is not in $\Omega(n^2)$.

(c) **T** **F** Given n positive integers, where each is at most u, Radix sort sorts them by breaking each integer into $O(\log_{10} u)$ base-10 digits, and then sorts them by their digits in least-significant order using counting sort.

Solution: False. Radix sort breaks up integers base-n. If a base-10 (or any constant) were used instead, Radix sort would have a $O(n \log_{10} u)$ running time, which is $O(n \log n)$ when $u = O(n^c)$ for some constant c.

(d) T F Inserting n^2 items sequentially to the back of an empty dynamic array takes worst-case $O(n^2)$ time.

Solution: True. Inserting into the back of a dynamic array takes amortized $O(1)$ time, which by definition means doing the operation k times takes worst-case $O(k)$ time.

(e) T F Given a Set AVL tree storing n keyed items ordered by key, one can construct a key-ordered max-heap on the same n items in worst-case $O(n)$ time.

Solution: True. Perform an in-order traversal of the AVL tree in $O(n)$ time to output an array of the items in the tree, and then transform the array into a max-heap (build the heap) in $O(n)$ time.

(f) T F Given a weighted connected undirected graph $G = (V, E)$ containing exactly $|V| - 1$ edges, one can solve weighted Single-Source Shortest Paths from any $s \in V$ in $O(|V|)$ time.

Solution: True. A connected graph with exactly $|V| - 1$ edges is a tree and contains no undirected cycles. Run breadth-first search from s in $O(|V|)$ time to direct every edge away from s, and then run DAG relaxation on the resulting DAG from s to compute single-source shortest paths, also in $O(|V|)$ time.

(g) T F Given a weighted simple directed graph $G = (V, E)$, where every vertex connects to at least $|V|/2$ edges and every edge has strictly negative weight, one can find a maximum-weight directed path from vertex $s \in V$ to $t \in V$ in linear time.

Solution: True. Negate all edge weights so all weights are positive, and run Dijkstra's algorithm from s to t in time $O(|V| \log |V| + |E|)$. There are $|E| = \Omega(|V|^2)$ edges in this graph, so Dijkstra's algorithm runs in linear time.

(h) T F While running Bellman-Ford from vertex s in a simple graph, suppose distance estimate $d'[s, v]$ is assigned a new smaller value D in the k^{th} round of relaxation. Then D is the weight of a path from s to v traversing exactly k edges.

Solution: False. Many edges along a path may be relaxed within a single round of relaxation. So, D may correspond to the weight of a path from s to t that traverses more than k edges. (Note: this is a correct invariant for some dynamic programming formulations of Bellman-Ford; we gave points to students who argued correctly based on such a formulation.)

(i) T F If a decision problem A is NP-hard, then A is also NP-complete.

Solution: False. Counter example: the halting problem is NP-hard and not in NP, so it is not NP-complete.

(j) T F If a decision problem A is NP-hard and $A \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$.

Solution: True. By definition, if A is NP-hard, then every problem in NP is polynomially reducible to A, so if $A \in \mathsf{P}$, then every problem in NP can be solved in polynomial time.

Problem 2. [24 points] **Sorting Sorts**

(a) [12 points] The $\boldsymbol{x}$**-value** of a pair (a, b) is the real number $\sqrt{a}+b\sqrt{x}$. Let B be an array of n pairs of positive integers (a_i, b_i) with $a_i < n$ and $b_i < n^3$ for all $i \in \{1, \ldots, n\}$. Describe an $O(n)$-time algorithm to sort the pairs in B by their x-values for $x = n$.

Solution: Since $\sqrt{a_i} < \sqrt{n}$ for all $i \in \{1, \ldots, n\}$, then $\sqrt{a_i} + b_i\sqrt{n} < \sqrt{a_j} + b_j\sqrt{n}$ whenever $b_i < b_j$, independent of a_i and a_j. Thus, we can sort the pairs by their n-values by first sorting by a_i's using radix sort, and then sorting by b_i's, also with radix sort. Thus algorithm is runs in worst-case $O(n)$ time, since the largest number u is less than n^3, and is correct because the b_i's are more significant than the a_i's and counting sort is stable.

Common Mistakes:

- Computing x-values (we've not given you any algorithm to compute square roots)
- Sorting by b_i's than a_i's, rather than the other way around
- Performing some transformation on x-values that does not preserve order

(b) [12 points] Let A be an array $(a_1, \ldots, a_n)$ of n positive integers, where $\lfloor \lg n \rfloor$ adjacent pairs in A appear in sorted order[1]. Describe an $O(n \log \log n)$-time algorithm to sort A.

Solution: The adjacent pair condition implies that A decomposes into $\lfloor \lg n \rfloor + 1$ contiguous reverse-sorted subsequences which can be found in $O(n)$ time. The problem reduces to merging $\lfloor \lg n \rfloor + 1$ input sorted arrays of total length n into one sorted array. Construct a min-heap on the smallest integer from each of the input arrays in $O(\log n)$ time, also storing with each integer a pointer to the array from which it came. Repeatedly remove the minimum integer x from the heap (inserting x to the back of an output dynamic array), and if there are integers left in x's input array, insert the next smallest integer from that array into the heap, until the heap is empty. At the end of this process, the output dynamic array contains the integers from A in sorted order. Each loop takes $O(\log \log n)$ time since the heap always has $O(\log n)$ size; and since the loop repeats exactly n times, this algorithm runs in $O(n \log \log n)$ time.

Common Mistakes:

- Incorrectly trying to black-box merge sort or heap sort
- Incorrectly assuming input array is $(\log n)$-proximate
- Only sorting a subset of the input, or reverse-sorting instead of sorting
- Claiming merge sort takes $O(\log \log n)$ time

[1]i.e., $|\{a_i \mid i \in \{2, \ldots, n\} \text{ and } a_{i-1} < a_i\}| = \lfloor \lg n \rfloor$

Problem 3. [14 points] **Pams vs. Ratriots**

Bom Trady is a football fan who will be attending the Big Game$^{\text{TM}}$, where her home team, the Ratriots, will be playing rival team, the Pams. The game will be held in a neutral city, so Bom will fly there and walk to the stadium from the airport.

- Upon arriving, Bom discovers some streets are **occupied**. A street occupied by fans from team A may only be traversed while wearing a team A jersey, and possessing no other jersey.
- Bom has a map depicting the m two-way streets and n intersections in the city, where each street directly connects two intersections, and each intersection connects at most four streets. The airport and stadium intersections are both marked on the map.
- Each street is marked with its positive integer length and its occupied status: either Ratriots, Pams, or unoccupied. Some intersections on the map are marked as having shops, which sell jerseys from both teams.

Assume Bom begins wearing a Ratriots jersey, always has enough money to purchase jerseys, and may discard a jersey at any time. Given her map, describe an $O(n \log n)$-time algorithm to return a shortest occupation-respecting route from the airport to the stadium (or return no such route exists).

Solution: Construct a graph G with a vertex for each intersection in the map marked whether it contains a shop, and for each street between intersections a and b of length ℓ, add an undirected edge $\{a, b\}$ of weight ℓ, marked with the street's occupation status. G has n vertices and $m = O(n)$ edges (since each vertex connects to at most four edges), so can be constructed in $O(n)$ time.

Now construct two graphs G_r and G_p, where G_r is G with all Pams-occupied edges removed, and G_p is G with all Ratriots-occupied edges removed. Then for every intersection v that contains a shop, connect v_p from G_p to v_r from G_r via a weight-zero undirected edge to construct graph G' (which also takes $O(n)$ time to construct). Let s and t correspond to the airport and stadium intersections respectively. Then any occupation-respecting route from the airport to the stadium corresponds to a path from s_r to either t_r or t_p in G'. G' has only positive edge weights, so run Dijkstra's algorithm from s_r and return the shortest route to either t_r or t_p, which runs in $O(n \log n)$ time. If the minimum weight path has weight ∞, return that no path exists.

Common Mistakes:

- Not adequately specifying starting and ending nodes for a search
- Not adequately specifying weights
- Not adequately specifying edges corresponding to unoccupied streets

Problem 4. [16 points] **Costly Campaigning**

Welizabeth Arren is campaigning in a Massachusetts (MA) election in the year 2048. Due to rising sea-levels, Massachusetts has become a line of n towns $(t_1, \dots, t_n)$, bounded by water on either side. Arren wants to run TV ads in some of the towns: MA residents love to gossip, so if Arren runs an ad in the town named t_i, the k cities on either side[2] of t_i will also hear about Arren's campaign. Each town t_i has a known positive integer cost c_i to broadcast an ad. Describe an $O(nk)$-time algorithm to determine a subset of towns in which to run TV ads, so as to minimize total broadcast costs, while ensuring that every town in MA hears about Arren's exciting plan.

Solution: Here is a dynamic programming solution:

1. **Subprojects**
   - $x(i, j)$: minimum total cost to reach all towns in prefix $(t_1, \dots, t_i)$, where the last town an ad was broadcast in was town t_{i+j}
2. **Relate**
   - An optimal ad campaign either broadcasts in town t_i or not
   - If $j = 2k + 1$, then we must broadcast in t_i
   - $x(i, j) = \min \left\{ \begin{array}{ll} c_i + x(i-1, 1) & \text{always} \\ x(i-1, j+1) & \text{if } j < 2k+1 \end{array} \right\}$
   - for $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, 2k+1\}$
   - Subproblem $x(i, j)$ only depends on subproblems with strictly smaller i, so acyclic
3. **Base**
   - $x(1, j) = 0$ for $j \leq k$ (town t_1 already reached)
   - $x(1, j) = c_1$ for $j > k$ (town t_1 not yet reached)
4. **Solution**
   - $x(n, k+1)$ is the minimum total cost to reach all towns in MA (where no houses have yet been reached)
   - Store parent pointers to reconstruct the set of towns advertised
5. **Time**
   - # subproblems: $\leq n(2k+1) = O(nk)$
   - work per subproblem: $O(1)$
   - $O(nk)$ total running time

Common Mistakes:

- Recurrence only notifies houses on one side of an advertised house
- Recurrence enforces advertising every k houses (not every $2k$)
- Recurrence skips next k houses after advertising at a house
- Incorrectly dealing with the first/last k houses

[2] Specifically, towns $(t_{\max(i-k,1)}, \dots, t_i, \dots, t_{\min(i+k,n)})$ will hear about Arren's campaign.

Problem 5. [16 points] **Parkour Performance**

Parkour athlete Diane Royal is planning a performance on top of an $n \times n$ square grid of platforms, where each platform p_{ij} has a known positive integer height $h(p_{ij})$ for $i, j \in \{1, \ldots, n\}$. Diane has a single ladder which she can place between any two adjacent[3] platforms at the start of the performance, and she may climb between them, up the ladder, **at most once** during the performance. Diane will start her performance on some platform, and then repeatedly either:

- **jump** from her current platform a to an adjacent platform b with strictly lower height[4], or
- **climb** the ladder she placed at the start, if her current platform is adjacent to the ladder.

Jumping from a platform a to a square b with $h(b) < h(a)$ will result in $(h(a) - h(b))^2$ **coolness**, but climbing the ladder yields no coolness. Given the height of each platform, describe an $O(n^2)$-time algorithm to determine: (1) where she should place her ladder and (2) where she should start, so as to maximize the total sum of coolness resulting from her performance.

Solution: Construct a graph G with a vertex for each of the n^2 platforms, with a directed edge from platform a to platform b with weight $-(h(a) - h(b))^2$ if a and b are adjacent in the grid and $h(b) < h(a)$. This graph is a DAG and has $O(n^2)$ vertices and edges, and can be constructed in $O(n^2)$ time.

Now make two copies of G, G_1 and G_2, which will correspond to the performance before using a ladder and after using a ladder respectively. Then connect a vertex s to every vertex v_1 in G_1 via a weight-zero directed edge (s, v_1), and add a weight-zero directed edge (v_1, u_2) from vertex v_1 in G_1 to vertex u_2 in G_2 if vertices v and u are adjacent in G, to form graph G'. Then the weight of any path in G' from s corresponds to the negative total coolness of a valid performance, and the total coolness of every valid performance corresponds to the negative weight of a path in G' from s.

G' has size $O(n^2)$ (s, G_1, G_2, plus at most five edges for each vertex of G_1) and is a DAG, so run DAG relaxation from s to compute to find a minimum weight path in G' from s. Diane's starting platform corresponds to the second vertex on this path. Further, if the path ends in G_2, whichever edge along the path transitioned from G_1 to G_2 corresponds to the optimal placement of a ladder. If the path ends in G_1, the ladder may be placed anywhere as it is not used in some optimal performance.

Common Mistakes:

- Assuming an optimal performance starts at the tallest platform
- Running DAG relaxation from all platforms instead of from an auxiliary node
- Using positive weight edges and finding a path of minimum weight
- Not recognizing their graph is a DAG with size $\Theta(n^2)$

[3] Two platforms p_{ij} and p_{xy} are adjacent if and only if $|i - x| + |j - y| = 1$.

[4] i.e., $h(b) < h(a)$

Problem 6. [16 points] **Mostly-Blue**

- A **blue-labeled** graph is a directed graph where every edge is labeled either blue or not-blue.
- A cycle in a blue-labeled graph is **mostly-blue** if more than half of its edges are blue.
- A vertex is **high-degree** if the sum of its in-degree and out-degree is strictly more than two.

Given a blue-labeled simple graph G containing n vertices and at most $O(n)$ edges, where exactly $k < n$ vertices are high-degree, describe an $O(n + k^3)$-time algorithm to determine whether G contains a mostly-blue cycle.

Solution: Observe that every connected component in G either (a) contains a high-degree vertex, (b) is a directed cycle, or (c) does not contain a directed cycle. Run a full depth-first search of G in $O(n)$ time to identify the connected components of the graph and categorize each one as above. (c) components may be safely ignored. A (b) component is a mostly-blue cycle if more than half of its edges are blue, which can be checked in $O(n)$ time.

It remains to check for mostly-blue cycles in the (a) components. Without loss of generality, assume G contains only (a) components. Let the **blue-difference** of a directed path in a blue-labeled graph be the number of non-blue edges on the path minus the number of blue ones. Construct a new directed graph G' on the k high-degree vertices, with an edge from high-degree vertex u to high-degree vertex v of weight w if v is reachable from u along a path having only low-degree vertices between them, where w is the minimum blue-difference of any such path.

To identify these edges, split every high-degree vertex in G with degree d into d vertices, one attached to each adjacent edge. This new graph G'' is a set of disjoint chains. For each chain, run depth-first search from each endpoint to identify whether the chain is directed consistently from one endpoint to the other, and if so, compute its blue-difference directly in time proportional to the length of the chain. Doing this for each chain, we can construct the edges in G' in $O(n)$ time, using a hash table to retain the minimum blue-difference of all edges having the same start and end.

Now, G contains a mostly-blue cycle if and only if G' contains a negative-weight cycle, since a path in G' of weight w corresponds to a path in G having $-w$ more blue edges than non-blue edges. G' has at most k^2 edges, so we can run Bellman-Ford on G' to detect negative-weight cycles in $O(k^3)$ time, leading to an $O(n + k^3)$-time algorithm in total.

Common Mistakes:

- Not assigning a source vertex when running a single-source shortest paths algorithm
- Running Bellman-Ford or Johnson's on a graph of size $\Theta(n)$
- Trying to enumerate all cycles (there may be an exponential number of cycles)
- Assuming DAG or trying to only detect unweighted cycles

Problem 7. [32 points] **Zero Sum**

(a) [16 points] Given $2k$ arrays, where each array contains exactly n integers, describe an $O(n^k)$-time algorithm to determine whether there exists $2k$ integers, exactly one from each array, that sum to zero. State whether your running time is worst-case, expected, and/or amortized. Significant partial credit will be given for $O(kn^k)$-time algorithms. (Hint: First try solving for small k.)

Solution: Let $S(i, j)$ be the set of sums of all $O(n^{j-i+1})$ subsets that could be formed by taking one integer from each of the arrays from array i to array j. If we could compute $S(1, k)$ and $S(k + 1, 2k)$ in $O(n^k)$ time, we could insert each sum from $S(1, k)$ into a hash table in expected $O(1)$ time, and then look up the negative of each sum in $S(k+1, 2k)$ to certify whether $S(1, 2k)$ contains zero in expected $O(n^k)$ time.

Naively, we could compute $S(1, k)$ in $O(kn^k)$ time by computing each subset sum of k integers directly. Alternatively, notice that we can compute the $O(n^i)$ sums in $S(1, i)$ from $S(1, i - 1)$ by adding a single integer from array i to one of the $O(n^{i-1})$ sums from $S(i-1)$ in $O(1)$ time, and $O(n^i)$ time in total. Then computing $S(1, k)$ can be done by computing $S(i)$ in order for $i \in \{1, \dots, k\}$ in time $O(\sum_{i=1}^{k} n^i) = O(n^k)$ as desired, with $S(k + 1, 2k)$ computed in a similar fashion.

Common Mistakes:

- Not computing sums efficiently
- (or not recognizing that naive sums take $O(k)$ time)

(b) [16 points] Now suppose the absolute value of any integer in the $2k$ arrays is less than u. Describe an $O(nuk^2)$-time algorithm to determine whether there exists $2k$ integers, exactly one integer from each array, whose sum is zero.
(Hint: use dynamic programming!)

Solution:

1. **Subproblems**
   - Let $A_i = (a_{i,1}, \dots, a_{i,n})$ be array i for $i \in \{1, \dots, 2k\}$
   - $x(i, j)$: True if there exists a subset formed by taking one integer from each of the first i arrays having sum j
2. **Relate**
   - $x(i, j) = \text{OR}\{x(i-1, j - a_{i,t}) \mid t \in \{1, \dots, n\}\}$
   - Subproblem $x(i, j)$ only depends on subproblems with strictly smaller i, so acyclic
3. **Base**
   - $x(0, 0) =$ True
   - $x(0, j) =$ False for $j \neq 0$
4. **Solution**
   - $x(2k, 0)$ is whether there exists a zero-sum subset formed by taking one integer from every array
5. **Time**
   - # subproblems: $\leq (2k+1)(2uk+1) = O(uk^2)$
   - work per subproblem: $O(n)$
   - $O(nuk^2)$ total running time
   - Note that this problem can be solved in $O(nuk \log k)$-time by repeatedly computing subset sums between adjacent arrays, instead of sequentially.

Common Mistakes:

- Defining subproblems in terms of fixed sum, rather than a generalized sum
- Missing index subproblem parameter
- Missing one of the base cases

Problem 8. [20 points] **Soldier Supply**

Savos Deaworth is a general who commands n soldiers.

- Savos has assigned each soldier i a pair of positive integers: a **troop number** t_i and a unique **soldier ID** d_i, and has stored all n soldier pairs (t_i, d_i) in a length-n array S.
- At the start of battle, Deaworth sends soldiers to the front, to fight side-by-side in a line. The initial **front line** is provided in an array L: $|L| < n$ soldier IDs listed in a **specified** order.
- Sometimes, a soldier may be temporarily **wounded** and must be removed from the front.
- Other times, Savos may **send** a new soldier to the front line, and will need to tell them where to stand: specifically, how many soldiers should be on their left when they go to the front.
- A soldier is more effective if they fight **next to** another soldier having the same troop number[5].

Describe a database supporting the following **worst-case** operations:

`init(S, L)`	initialize with soldier array `S` and initial front line `L` in $O(n \log n)$ time
`wound(d)`	remove the soldier with ID `d` from the front line in $O(\log n)$ time (assume `d` is in the front line before the operation)
`send(d)`	send and place the soldier with ID `d` into the front line in $O(\log n)$ time place `d` next to a soldier in the front line having the same troop number if possible if no soldier in front line with the same troop number, place `d` in left-most position **return** the number of soldiers to the left of where `d` is placed (assume `d` is not in the front line before the operation)

Solution: We implement the database by maintaining the following data structures:

- A Sequence AVL tree F to maintain the current extrinsic order of the front line
- A Set AVL tree D storing all n soldier pairs (keyed by soldier ID), mapping to their location in the Sequence AVL tree if they are on the front line. (A static sorted array may also be used for D.)
- A Set AVL tree $\mathcal{T}$ keyed by troop number storing a Set AVL tree keyed by soldier ID for each troop number, where the Set AVL tree $T(t_i)$ associated with troup t_i contains the soldier IDs of all soldiers currently on the front line who have troop number t_i. (A static sorted array may also be used for $\mathcal{T}$.)

Each of these data structure will always contain at most n time, so all respective tree operations will each take $O(\log n)$ time in the worst-case. For brevity, we will use (*) to end a passage describing a computation that takes worst-case $O(\log n)$ time. Let $L = (s_1, \ldots, s_{|L|})$.

To implement `init(S, L)`, build trees F, D, and $\mathcal{T}$ directly:

- Initialize D by inserting each soldier pair from S (*), in $O(n \log n)$ time total.
- Initialize $\mathcal{T}$ by inserting an empty Set AVL tree corresponding to each unique troop number in S (*), in $O(n \log n)$ time total.

(Continued on S1)

[5]Two soldiers fight next to each other if they are adjacent in the front line.

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.

Solution: (Continuation of Problem 8)

- Initialize F by doing the following for each s_i in order for i from 1 to $|L|$: look up s_i in D to find its troop number t (*); add s_i to $T(t)$ (*); take s_i and `insert_last` into F (*); and add a pointer to s_i's location in F to the record of s_i in D.

This process takes $O(\log n)$ time per s_i, so the whole process finishes in worst-case $O(n \log n)$ time, correctly constructs the data structures described above.

To implement `wound(d)`: look up d in D to find its troop number t and its location in F (*); remove d from F using the stored pointer (*); and find tree $T(t)$ in $\mathcal{T}$ and remove d from $T(t)$ (*). This process takes $O(\log n)$ time worst-case, and maintains the invariants of D, F, and $\mathcal{T}$.

To implement `send(d)`: look up d in D to find its troop number t and its location in F (*), and find tree $T(t)$ in $\mathcal{T}$ (*). If $T(t)$ is empty, `insert_first` d into F (*) and return 0. Otherwise, arbitrarily select a soldier number d' from $T(t)$ (*), and look up d' in D to find its location in F (*) in node v'. Let x denote the index of node v in the in-order traversal of F. To compute x, initialize to zero and repeatedly walk up the tree to the root. When at node v:

- If v is v' or the previous node is a right child of v, add the size of the left subtree of v to x, since everything in that subtree comes before v' in the in-order traversal.
- Otherwise, the previous node is a left child of v, so continue.

This process takes $O(\log n)$ time to walk up the tree to the root, spending at most $O(1)$ time at each node. Then `insert_at` d into F at index $x + 1$ and return $x + 1$, the number of soldiers to the left of d in F after insertion. In either case, once d is inserted into F, insert d to $T(t)$ and add a pointer to d's location in F to the record of s_i in D (*). This process takes $O(\log n)$ time worst-case in total, and maintains the invariants of D, F, and $\mathcal{T}$.

Common Mistakes:

- Missing set data structures to perform intrinsic lookups by soldier ID and/or troop number
- Missing a sequence data structure to keep track of extrinsic front-line order
- Modifying soldier ID set data structure after initialization
- Using a hash table when worst-case bounds are requested
- Storing index for each soldier in front line (not maintainable in $O(\log n)$ time)
- Neglecting to return an index in `send(d)`
- Assuming all soldiers with same troop number appear adjacent in the initial front-line

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.