Solution: Quiz 1

- 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 120 minutes to earn a maximum of 120 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 one 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) 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: Warmup	4	20
2: Sortid Casino	3	18
3: Restaurant Lineup	1	15
4: Range Pair	2	25
5: Rainy Research	1	20
6: Left Smaller Count	1	20
Total		120

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. [20 points] **Warmup**

(a) [5 points] Given array A of n integers, the Python function below appends all integers from set $\{A[x] \mid 0 \leq i \leq x < j \leq n \text{ and } A[x] < k\}$ to the end of dynamic array B.

```
def filter_below(A, k, i, j, B):
    if (j - i) > 1:
        c = (i + j) // 2
        filter_below(A, k, i, c, B)
        filter_below(A, k, c, j, B)
    elif (j - i) == 1 and A[i] < k:
        B.append(A[i])
```

Argue the **worst-case** running time of `filter_below(A, k, 0, len(A), [])` in terms of $n =$ `len(A)`. You may assume that n is a power of two.

Solution: This function has recurrence of form $T(n) = 2T(n/2) + f$, where f is constant amortized. The number of base cases $n^{\log_2 2}$ dominates, so since a linear number of constant amortized operations are performed, this function runs in worst-case $O(n)$ time.

Common Mistakes:

- wrong recurrence, or solved correct recurrence incorrectly
- not acknowledging amortized operation

(b) [5 points] The integer array $A = [4, 3, 1, 5, 0, 2]$ is not a heap. It is possible to make A either a max or min heap by swapping two integers. State two such integers.

Solution: Swap 4 and 0 to construct a **min heap**.

Common Mistakes:

- trying for a max heap with a single swap, which is not possible

(c) [5 points] Let T be a binary search tree storing n integer keys in which the key k appears $m > 1$ times. Let p be the lowest common ancestor of all nodes in T which contain key k. Prove that p also contains key k.

Solution: Suppose for contradiction p contains some key $k^* \neq k$. Since p is the lowest common ancestor of at least two nodes containing key k, one such node exists in p's left subtree and one such node exists in p's right subtree. But then by the BST property, $k \leq k^* \leq k$, a contradiction.

Common Mistakes:

- assuming a node storing k is always parent or child of another node storing k

(d) [5 points] Given the hash family $\mathcal{H} = \{h_a(k) = a(k+a) \bmod m \mid a \in \{1, \ldots, m\}\}$ and some key $k_1 \in \{0, \ldots u-1\}$ where $2 < 2m < u$, find a key $k_2 \in \{0, \ldots, u-1\}$ with $k_2 \neq k_1$ such that $h_a(k_1) = h_a(k_2)$ for every $h_a \in \mathcal{H}$.

Solution: Choose $k_2 = k_1 + m$ if $k_1 < m$, and $k_2 = k_1 - m$ if $k_1 \geq m$. Clearly, $0 \leq k_2 < u$ by construction, with $k_1 \neq k_2$. Further, $h'(k_1) = h'(k_2)$ because:

$$h'(k_1) = (ak_1 + a^2) \bmod m = (ak_1 + a^2 \pm am) \bmod m = (a(k_1 \pm m) + a^2) \bmod m = h'(k_2).$$

Common Mistakes:

- choosing particular pair of keys instead of k_2 for any k_1
- choosing $k_2 = k_1 \bmod m$ without bounding $k_2 \in \{0, \ldots, u-1\}$
- choosing k_2 which depends on a; fixed pair must collide for every a

Problem 2. [18 points] **Sortid Casino** (3 parts)

Jane Stock is secret agent 006. She is searching for criminal mastermind Dr. Yes who is known to frequent a fancy casino. Help Jane in each of the following scenarios. Note that each scenerio can be **solved independently**.

(a) [6 points] A dealer in the casino has a deck of cards that is missing 3 cards. He will help Jane find Dr. Yes if she helps him determine which cards are missing from his deck. A full deck of cards contains kn cards, where each card has a value (an integer $i \in \{1, \ldots, n\}$) and a suit (one of k known English words), and no two cards have both the same value and the same suit. Describe an efficient[1] algorithm to determine the value and suit of each of the 3 cards missing from the deck.

Solution: Construct an initially empty hash table in constant time. For each card in the deck, check whether its suit is in the hash table. If not, add the suit to the hash table mapping to an empty dynamic array. Then in either case, append the card's number to the end of the suit's dynamic array. At the end of this process, each suit array contains the card numbers of that suit from the deck. Processing each card in this way takes expected amortized $O(1)$ time, so doing this with all $nk - 3$ cards takes expected $O(nk)$ time. After inserting all cards, check the length of each suit array in $O(k)$ time. Any suit array with length n has all its cards. For any suit array whose length is less than n, sort the numbers in the array using counting sort in worst-case $O(n)$ time since the numbers are positive and bounded by n. Then loop through the numbers in order to find any that are missing. At most three suit arrays have length less than n. In total, this algorithm runs in expected $O(nk)$.

Common Mistakes:

- using an integer sort on suits without mapping suits to integers
- looping over suits without first generating a list of suits

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

(b) [6 points] The dealer doesn't know Dr. Yes, but he knows that Dr. Yes is one of the k best players in the casino. Jane scans the room and for each of the $p > k$ players, she transmits back to headquarters a pair (c, ℓ) representing the number of chips c and location ℓ of the player. Assuming that no player has the same number of chips, describe an efficient algorithm for headquarters to determine the locations of the k players in the casino who have the most chips.

Solution: Build a max heap out of the p player pairs keyed by chip number in worst-case $O(p)$ time. Then, remove the maximum player pair from the heap k times and store its location in a dynamic array to return. This process takes worst-case $O(k \log p)$ time, so this process runs in $O(p + k \log p)$ time in total.

Common Mistakes:

- correct solutions that were inefficient

(c) [6 points] After determining the locations of the k players with the most chips, Jane observes the game play of each of them. She watches each player play exactly $h < k$ game rounds. In any game round, a player will either win or lose chips. A player's **win ratio** is one plus the number of wins divided by one plus the number of losses during the h observed hands. Given the number of observed wins and losses from each of the k players, describe an efficient algorithm to sort the players by win ratio.

Solution: Observe that since wins plus losses equals h, one win ratio $(w_1+1)/(\ell_1+1)$ is larger than another win ratio $(w_2+1)/(\ell_2+1)$ if and only if $w_1 > w_2$, so it suffices to sort the players based on their wins. Since wins are positive and bounded by $h < k$, we can use counting sort to sort the players in worst-case $O(k)$ time.

Common Mistakes:

- directly computing win ratios with arbitrary precision
- multiplying ratios by product of denominators (numbers can be exponential)
- comparison sorting win ratios by cross-multiplication (inefficient)

Problem 3. [15 points] **Restaurant Lineup**

Popular restaurant Criminal Seafood does not take reservations, but maintains a wait list where customers who have been on the wait list longer are seated earlier. Sometimes customers decide to eat somewhere else, so the restaurant must remove them from the wait list. Assume each customer has a different name, and no two customers are added to the wait list at the exact same time. Design a database to help Criminal Seafood maintain its wait list supporting the following operations, each in $O(1)$ time. State whether each operation running time is worst-case, amortized, and/or expected.

- `add_name(`x`)`: add name x to the back of the wait list
- `remove_name(`x`)`: remove name x from the wait list
- `seat()`: remove and return the name of the customer from the front of the wait list

Solution: Maintain a doubly-linked list containing customers on the wait list in order, maintaining a pointer to the front of the linked list corresponding to the front of the wait list, and a pointer to the back of the linked list corresponding to the back of the wait list. Also maintain a hash table mapping each customer name to the linked list node containing that customer. To implement `add_name(`x`)`, create a new linked list node containing name x and add it to the back of the linked list in worst-case $O(1)$ time. Then add name x to the has table pointing to the newly created node. To implement `remove_name(`x`)`, lookup name x in the hash table in and remove the mapped node from the linked list in expected $O(1)$ time. Lastly, to implement `seat()`, remove the node from the front of the linked list containing name x, remove name x from the hash table, and then return x, also in expected $O(1)$ time.

Note, this problem can also be solved with amortized bounds without using a linked list but the amortization would need to be fully analyzed for full points.

Common Mistakes:

- using $O(n)$ delete from the middle of a double-ended dynamic array
- using direct access array on names (integer representations not bounded)
- providing a correct amortized solution without analysis
- re-inventing a linked list using other data structures

Problem 4. [25 points] **Range Pair** (2 parts)

Given array $A = [a_0, a_1, \ldots, a_{n-1}]$ containing n **distinct** integers, and a pair of integers (b_1, b_2) with $b_1 \leq b_2$, a **range pair** is a pair of indices (i, j) with $i \neq j$ such that the sum $a_i + a_j$ is within range, i.e., $b_1 \leq a_i + a_j \leq b_2$. Note that parts (a) and (b) can be **solved independently**.

(a) [10 points] Assuming $b_2 - b_1 < 6006$, describe an $O(n)$-time algorithm to return a range pair of A with respect to range (b_1, b_2) if one exists. State whether your algorithm's running time is expected, worst-case, and/or amortized.

Solution: Let $c = b_2 - b_1$, a constant. Then, for each $k \in \{1, \ldots, c\}$, we can find whether any pair (i, j) satisfies $a_i + a_j = b_1 + k$. For each $k \in \{1, \ldots, c\}$, build a hash table H mapping each integer a_i to i in expected $O(n)$ time. Then for each a_i, check whether $b_1 + k - a_i$ is in the hash table. If it is, you have found a range pair, so return (i, j). Otherwise, there is no a_j for which $a_i + a_j = b_1 + k$, so we proceed to check a_{i+1} until $i = n$. Each of the n checks takes expected constant time, so checking for range pairs for k runs in expected $O(n)$ time. We repeat this procedure for all $k \in \{1, \ldots, c\}$, returning no range pair exists if the process terminates without finding a range pair. Since c is constant, this algorithm runs in expected $O(cn) = O(n)$ time.

Common Mistakes:

- assuming that a_i, b_1, b_2 polynomially bounded
- checking sums within an incorrect range (not exactly the range from b_1 and b_2)

(b) [15 points] Assuming $\log_n(\max A - \min A) < 6006$ (with no restriction on b_1 or b_2), describe an $O(n)$-time algorithm to return a range pair of A with respect to range (b_1, b_2) if one exists. State whether your algorithm's running time is expected, worst-case, and/or amortized.

Solution: Observe that since $\log_n(\max A - \min A)$ is some constant k, then $n^k = \max A - \min A$. Loop through A to find $\min A$ in worst-case $O(n)$ time and subtract $\min A$ from each a_i. Now A contains integers in the range $\{0, \ldots, n^{k+1}\}$, and we can use radix sort to sort them in worst-case $O(n)$ time. Lastly, add $\min A$ to each value, also in worst-case $O(n)$ time, yielding array A', containing the elements of A in sorted order. Now we try to find a range pair. Initialize pointers $i = 0$ and $j = n - 1$. Repeat the following procedure, maintaining the invariant that at the start of each loop, we've either returned a range pair or confirmed that a_x cannot exist in a range pair for any $x < i$ and $x > j$, which is vacuously true at the start. To process loop (i, j), if $b_1 \leq a_i + a_j \leq b_2$, then return (i, j) as a range pair. If $i = j$, then by the invariant, there is no range pair, so return that none exists. Otherwise:

- If $a_i + a_j < b_1$, then a_i cannot be a part of a range pair with any a_x for $x \leq j$, so increase i by one, maintaining the invariant.
- If $a_i + a_j > b_2$, then a_j cannot be a part of a range pair with any a_x for $x \geq i$, so decrease j by one, maintaining the invariant.

Each loop takes worst-case $O(1)$ time to execute, and with each loop, either i increase or j decreases, so $j - i$ decreases from $n - 1$ to $j - i = 0$, at which point the algorithm terminates. Thus this algorithm runs in worst-case $O(n)$ time.

Common Mistakes:

- assuming each a_i is polynomially bounded (not subtracting $\min A$)
- assuming $b_2 - b_1$ is bounded and trying to use hash table as in 4a)
- two-finger algorithm mistakes:
  - pointers move in same direction or can move back and forth
  - algorithm has no termination condition

Problem 5. [20 points] **Rainy Research**

Mether Wan is a scientist who studies global rainfall. Mether often receives data measurements from a large set of deployed sensors. Each collected data measurement is a triple of integers (r, ℓ, t), where r is a positive amount of rainfall measured at latitude ℓ at time t. The **peak rainfall** at latitude ℓ **since** time t is the maximum rainfall of any measurement at latitude ℓ measured at a time greater than or equal to t (or zero if no such measurement exists). Describe a database that can store Mether's sensor data and support the following operations, each in worst-case $O(\log n)$ time where n is the number of measurements in the database at the time of the operation.

- `record_data(`r, ℓ, t`)`: add a rainfall measurement r at latitude ℓ at time t
- `peak_rainfall(`ℓ, t`)`: return the peak rainfall at latitude ℓ **since** time t

Solution: Maintain a latitude AVL tree keyed on distinct measurement latitudes, where each latitude ℓ maps to a rainfall AVL tree containing all the measurement triples with latitude ℓ, keyed by time. We only store nodes associated with measurements, so the height of each AVL tree is bounded by $O(\log n)$. For each rainfall AVL tree, we augment each node p with the maximum rainfall $p.m$ of any measurement within p's subtree. This augmentation can be maintained in constant time at a node p by taking the maximum of the rainfall at p and the augmented maximums of p's left and right children; thus this augmentation can be maintained without effecting the asymptotic running time of standard AVL tree operations.

To implement `record_data(`r, ℓ, t`)`, search the latitude AVL tree for latitude ℓ in worst-case $O(\log n)$ time. If ℓ does not exist in the latitude AVL tree, add a new node corresponding to ℓ mapping to a new empty rainfall AVL tree, also in $O(\log n)$ time. In either case, add the measurement triple to ℓ's rainfall AVL tree, for a total running time of worst-case $O(\log n)$.

To implement `peak_rainfall(`ℓ, t`)`, search the latitude AVL tree for latitude ℓ in worst-case $O(\log n)$ time. If ℓ does not exist, return zero. Otherwise, we perform a one-sided range query on the rainfall AVL tree associated with ℓ to find the peak rainfall at latitude ℓ since time t. Specifically, let $\text{peak}(p, t)$ be the maximum rainfall of any measurement in node p's subtree measured at time $\geq t$ (or zero if p is not a node):

$$\text{peak}(p, t) = \begin{cases} \max\{p.r, p.right.m, \text{peak}(p.left, t)\} & \text{if } p.t \geq t \\ \text{peak}(p.right, t) & \text{if } p.t < t \end{cases}.$$

Then peak rainfall is simply $\text{peak}(p, t)$ with p being the root of the tree, which can be computed using at most $O(\log n)$ recursive calls. So this operation runs in worst-case $O(\log n)$ time.

Note, this problem can also be solved where each latitude AVL tree is keyed by rainfall, augmenting nodes with maximum time in subtree. We leave this as an exercise to the reader.

Rubric and common mistakes continued on S1.

Problem 6. [20 points] **Left Smaller Count**

Given array $A = [a_0, a_1, \ldots, a_{n-1}]$ containing n **distinct** integers, the **left smaller count array** of A is an array $S = [s_0, s_1, \ldots, s_{n-1}]$ where s_i is the number of integers in A to the left of index i with value less than a_i, specifically:

$$s_i = |\{j \mid 0 \leq j < i \text{ and } a_j < a_i\}|.$$

For example, the left smaller count array of $A = [10, 5, 12, 1, 11]$ is $S = [0, 0, 2, 0, 3]$. Describe an $O(n \log n)$-time algorithm to compute the left smaller count array of an array of n distinct integers. State whether your algorithm's running time is worst-case, amortized, and/or expected.

Solution: We compute values s_i increasing from $i = 0$ to $i = n - 1$ by maintaining at all times an AVL tree T_i on integer keys $a_0, \ldots, a_{i-1}$, where each AVL node p is augmented with the number of nodes $p.s$ in the subtree rooted at p. This augmentation can be maintained in constant time at a node p by adding 1 to the augmented subtree sizes of p's left and right children; thus this augmentation can be maintained without effecting the asymptotic running time of standard AVL tree operations.

To compute s_i, perform a one-sided range query on T. Specifically, let $\text{count}(p, k)$ be the number of nodes in the subtree rooted at p having key strictly less than k (or zero if p is not a node):

$$\text{count}(p, k) = \begin{cases} 1 + p.left.s + \text{count}(p.right, k) & \text{if } p.k < k \\ \text{count}(p.left, k) & \text{if } p.k \geq k \end{cases}.$$

Then s_i is simply $\text{count}(p, a_i)$ where p is the root of T_i, which can be computed using at most $O(\log n)$ recursive calls. So computing s_i takes worst-case $O(\log n)$ time. We then maintain the invariant by inserting a_i into T_i to form T_{i+1}. Repeating this procedure n times then computes all $s_i \in S$ in worst-case $O(n \log n)$ time.

Note, this problem can also be solved by modifying merge sort. A rigorous solution based on this approach is **continued on S2**.

Common Mistakes:

- attempting to use an augmented heap
- attempting an unmaintainable AVL augmentation, e.g., # stored integers $\leq self.k$
- building a size i data structure from scratch for each $i \in \{1, \ldots, n\}$
- assuming each a_i is polynomially bounded
- ignoring $0 \leq j < i$ restriction, instead returning $s_i = |\{j \mid a_j < a_i\}|$

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.

Continued from Problem 5)

Common Mistakes:

- using a hash table (gives expected running times, not worst-case)
- assuming latitudes are polynomially bounded
- problems with AVL range query:
  - attempting an unmaintainable AVL augmentation
  - augmentation that depends on query t
  - not considering the measurement stored at the root of each recursive call
  - considering too many or two few subtrees in 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.

Alternative solution to Problem 6)

Solution: We modify MergeSort($A[1:n]$) to design a recursive function Count($A[1:n]$, $S[1:n]$) as follows. When passing a subarray, assume its indices are passed (or one can create a new array of smaller size and copy the contents, as it can be done in linear time anyway; or consider both A and S being global variables). We will not grade based on implementation details, as long as it is clear how it can be done in linear time (in size of a given array).

Count() is going to perform Merge Sort on A and fills in S while doing so. If $n = 1$, we set $S[1] = 0$ and return (this is the base case). If $n > 1$, we first split A into equal-size halves, A_L and A_R, and recursively call Count on A_L and A_R; we also refer to the returned array $S[]$ from each call by S_L and S_R. That is, let $m = \lfloor n/2 \rfloor$, and call Count($A_L = A[1:m], S_L = S[1:m]$) and Count($A_R = A[m+1:n], S_R = S[m+1:n]$).

Suppose the two recursive calls correctly sorted A_L and A_R as well as correctly computed S_L and S_R (this is our inductive hypothesis in proof of correctness). To sort A, we perform Merge Sort by keeping two indices l and r each running from 1 to m and $m+1$ to n, respectively on A_L and A_R. We fill in $A[1:n]$ and $S[1:n]$ by keeping an index i that runs from 1 to n. Let c be the number of elements from A_L that have been added to A (if l runs from 1 to m, c should be equal to $l-1$). We initialize $c = 0$ to begin with.

- ($l > m$ or $r > n$, then we treat $A_L[l] = \infty$ or $A_R[r] = \infty$ in the following if-statements.)
- If $A_L[l] < A_R[r]$, we set $A[i] \leftarrow A_L[l]$, $S[i] \leftarrow S_L[l]$, and increment c, l, i by one.
- If $A_L[l] > A_R[r]$, we set $A[i] \leftarrow A_R[r]$, $S[i] \leftarrow S_R[r] + c$, and increment r, i by one.

Once this is completed, we will have A that is sorted and S that contains the correct values (but sorted); to recover the output we want to (according to the original ordering of A), one can iterate over the original input and search for the key (via binary search) in the sorted list, and output S accordingly, which can be done in $O(n \log n)$ time. Or, one can also associate the original index of each $A[i]$ in the algorithm or use a hash-map; as long as you clearly state how you can re-construct the final output in the correct ordering in $O(n \log n)$ time you'd get a credit for this part.

This algorithm runs in $O(n \log n)$ time as the recurrence of runtime is $T(n) = 2T(n/2) + O(n)$; the indices i, l, r only increase in the (implicit) while-loop above. (Continued on S3)

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.

Continued alternative solution to Problem 6)

Solution: To prove correctness, we prove by induction. The base case is trivial: On a sub-array of size 1, $S[i]$ should be equal to 0. Now suppose our recursive calls correctly sorted A_L, A_R and correctly computed S_L, S_R respectively on A_L and A_R. We show that A will be correctly sorted (on A_L plus A_R) and S will be correctly computed on A. Since we know from Merge Sort A is correctly sorted, we do not need to prove this part. To show that $S[i]$ is computed correctly (with respect to A), consider the first case when some element $A_L[l]$ on the left-half of A is being added to $A[i]$. All elements in A_R appear after $A_L[l]$ in the original array A, and thus if we computed $S_L[l]$ correctly from the recursive call, then $S[i]$ is correctly computed – which is true by inductive hypothesis. Now consider the second case where some element $A_R[r]$ on the right-half of A is being added to $A[i]$. Within the right-half, our recursive call correctly computed the number of elements smaller than $A_R[r]$ which is stored in $S_R[r]$. The number of elements in the left-half (A_L) that are smaller than $A_R[r]$ is exactly equal to c, which counts the number of elements from A_L that have been added to $A[]$ so far. Since we increment c by one only when we add an element from A_L to A, c is computed correctly. This proves that both A and S are correctly computed at the end of the merge step.

(Common errors include: Not using the counter (in our solution, that is c) correctly. Sorting A and compute differences between original indices and new indices. Decreasing/increasing S values during the merge step by iterating over L or R – this results in $O(n^2)$ during the merge step.)

Since the problem asks for $O(n \log n)$ solution and there exists a trivial $O(n^2)$ solution (by computing $S[i]$ by iterating over $j \in [1, i-1]$), there is no partial credit for any answer with worse runtime than $O(n \log n)$.

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.