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) and continue your solution on the referenced scratch page at the end of the exam.
- Do not spend time and paper rederiving facts that we have presented in lecture or recitation. 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
1: Information	2	2
2: Frequentest	2	8
3: Haphazard Heaps	2	10
4: Transforming Trees	2	10
5: Sorting Sock	4	20
6: Triple Sum	1	15
7: Where am i?	1	15
8: Methane Menace	1	20
9: Vapor Invite	1	20
Total		120

Name: ____________________

MIT Kerberos Username: ____________________

Problem 1. [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 2. [8 points] **Frequentest** (2 parts)

The following two Python functions correctly solve the problem: given an array `X` of n positive integers, where the maximum integer in `X` is k, return the integer that appears the most times in `X`. Assume: a Python `list` is implemented using a dynamic array; a Python `dict` is implemented using a hash table which randomly chooses hash functions from a universal hash family; and `max(X)` returns the maximum integer in array `X` in worst-case $O(|X|)$ time. For each function, state its **worst-case** and **expected** running times **in terms of n and k**.

(a) [4 points]

```
def frequentest_a(X):
    k = max(X)
    H = {}
    for x in X:
        H[x] = 0
    best = X[0]
    for x in X:
        H[x] += 1
        if H[x] > H[best]:
            best = x
    return best
```

i. Worst-case:

Solution: $O(n^2)$

ii. Expected:

Solution: $O(n)$

(b) [4 points]

```
def frequentest_b(X):
    k = max(X)
    A = []
    for i in range(k + 1):
        A.append(0)
    best = X[0]
    for x in X:
        A[x] += 1
        if A[x] > A[best]:
            best = x
    return best
```

i. Worst-case:

Solution: $O(n + k)$

ii. Expected:

Solution: $O(n + k)$

Common Mistakes: Saying doing a $O(1)$-time operation m times takes amortized $O(m)$ time.

Problem 3. [10 points] **Haphazard Heap** (3 parts)

Array `[A,B,C,D,E,F,G,H,I,J]` represents a **binary min-heap** containing 10 items, where the key of each item is a **distinct** integer. State which item(s) in the array could have the key with:

(a) the smallest integer

Solution: A

(b) the third smallest integer

Solution: B, C, D, E, F, G

Common Mistakes: Incorrectly assuming third smallest must be in either B or C.

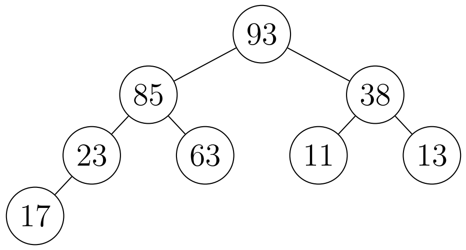
(c) the largest integer

Solution: F, G, H, I, J

Common Mistakes: Thinking the largest integer must be in the bottom level (instead of in any leaf)

Problem 4. [10 points] **Transforming Trees** (2 parts)

The tree below contains 8 items, where each stored item is an integer which is its own key.



(a) [6 points] Suppose the tree drawn above is the implicit tree of a binary max-heap `H`. State the array representation of `H`, **first before** and **then after** performing the operation `H.delete_max()`.

Solution:

Before: `[93, 85, 38, 23, 63, 11, 13, 17]`

After: `[85, 63, 38, 23, 17, 11, 13]`

Common Mistakes:

- Not building heap correctly (e.g., confusing heap order with traversal order)
- Not correctly deleting min (by swapping root with last leaf and heapifying down)

(b) [4 points] Suppose instead that the original tree drawn above is a Sequence AVL Tree `S` (note Sequence data structures are zero-indexed). The items in **the leaves** of `S` in traversal order are $(17, 63, 11, 13)$. Perform operation `S.delete_at(3)` on `S` including any rotations, and then **list the items** stored in **the leaves** of `S` in traversal order, after the operation has completed. (You do not need to draw the tree.)

Solution: $(17, 85, 11, 13)$

Common Mistakes:

- Indexing by heap order instead of traversal order (or using 1-indexing)
- Listing full traversal order (rather than **just the leaves**)
- Deleting the item but not rebalancing the tree to satisfy the AVL Property

Problem 5. [20 points] **Sorting Sock** (4 parts)

At Wog Hearts School of Wizcraft and Witcherdry, n incoming students are sorted into four houses by an ancient magical artifact called the Sorting Sock. The Sorting Sock first sorts the n students by each of the four houses' attributes and then uses the results to make its determinations. For each of the following parts, state and justify what type of sort would be most efficient. (By "efficient", we mean that faster correct algorithms will receive more points than slower ones.)

(a) [5 points] For House Puffle Huff, students must be sorted by ***friend number***, i.e., how many of the other $n - 1$ incoming students they are friends with, which can be determined in $O(1)$ time.

Solution: Friend numbers are non-negative integers less than n, so we can use counting sort to sort the students in worst-case $O(n)$ time. (Radix sort also works with the same running time.) Since we have to compute friend number for each student, any algorithm will take at least $\Omega(n)$ time, so this is optimal.

Common Mistakes: Using a sort that is not $O(n)$

(b) [5 points] For House Craven Law, students must be sorted by the weight of their books. Book weights cannot be measured precisely, but the Sorting Sock has a ***scale*** that can determine in $O(1)$ time whether one set of books has total weight greater than, less than, or equal to another set of books.

Solution: A scale weighing is a comparison with a constant number of outcomes, so the comparison sort $\Omega(n \log n)$ lower bound applies. So we cannot do better than by using an worst-case $O(n \log n)$ sorting algorithm, e.g., merge sort, using the scale to compare one student's books against another's.

Common Mistakes: Insufficient justification for why $O(n \log n)$ is optimal

(c) [5 points] For House Driven Gore, students must be sorted by ***bravery***, which can't be directly measured or quantified, but for any set of students, the Sorting Sock can determine the bravest among them in $O(1)$ time, e.g., by presenting the students with a scary situation.

Solution: We can't quantify bravery, so we can't hope to use any integer-based algorithms. However, the Sorting Sock can find a student of maximum bravery in $O(1)$ time, so we repeatedly find and select a bravest student among all previously unselected students in worst-case $O(n)$ time, which is again optimal. (This is priority queue sort, using the Sorting Sock as the priority queue to find the maximum.)

Common Mistakes: Arguing a $\Omega(n \log n)$ lower bound

(d) [5 points] For House Leather Skin, students must be sorted by their ***magical lineage***: how many of a student's ancestors within the previous $3\lceil \log n \rceil + 4$ generations were magical. Recall that humans, magical or not, always have two parents in the previous generation, unlike binary tree nodes which have at most one. Assume the Sorting Sock can compute the magical lineage of a student in $O(1)$ time.

Solution: Each student has at most 2^k ancestors in the kth generation preceding. Thus the number of wizard ancestors will be a non-negative number bounded above by $\sum_{i=1}^{3\lceil \log n \rceil + 4} 2^k < 2^{3(\log n + 1)+5} = 2^8 2^{3 \log n} = 2^8 n^{3 \log 2} = O(n^c)$ for any $c > 3 \log 2$. Thus we can use radix sort to sort the students by their magical lineage in worst-case $O(n)$ time, which is again optimal.

Common Mistakes:

- Claiming $3\lceil \log n \rceil + 4$ ancestors instead of $O(2^{3\lceil \log n \rceil + 4})$
- Saying that $2^{3\lceil \log n \rceil + 4} = O(n)$ and using counting sort

Problem 6. [15 points] **Triple Sum**

Given three arrays A, B, C, each containing n integers, give an $O(n^2)$-time algorithm to find whether some $a \in A$, some $b \in B$, and some $c \in C$ have zero sum, i.e., $a + b + c = 0$. State whether your running time is worst-case, expected, and/or amortized.

Solution: We present both expected and worst-case solutions (both were accepted for full points).

Expected Time

For each pair of numbers $(a, b) \in A \times B$, store $a + b$ in a hash table H. Then return Yes if $-c$ appears in H for any $c \in C$, and return No otherwise.

Correctness: If any $-c$ appears in H for any $c \in C$, then $-c = a' + b'$ for some $(a', b') \in A \times B$ so $a' + b' + c = 0$. Otherwise, there is no $c \in C$ for which $-c = a' + b'$ for any $(a', b') \in A \times B$.

Running Time: There are $O(n^2)$ pairs in $A \times B$, so inserting them into H takes expected $O(n^2)$ time. Then checking whether each $-c$ appears in H takes expected $O(1)$ time each, and expected $O(n)$ in total. So this algorithm runs in expected $O(n^2)$ time.

Continued on scratch paper S1 for worst-case solution...

Common Mistakes:

- Using counting/radix sort or creating a direct access array (no bound on u so not efficient)
- Saying n insertions into a hash table gives an amortized bound
- Checking all triples in $\Omega(n^3)$ time

Problem 7. [15 points] **Where Am i?**

Given a Sequence AVL Tree T containing n nodes, and a pointer to a node v from T , describe an $O(\log n)$-time algorithm to return the (zero-indexed) index i of node v in the traversal order of T. (Recall that every node `u` in a Sequence AVL Tree T stores an item `u.item`, parent `u.parent`, left child `u.left`, right child `u.right`, subtree height `u.height`, and subtree size `u.size`.)

Solution: Our algorithm will be to walk up the tree from v to the root r of the Sequence AVL Tree, counting the nodes preceding v in the traversal order along the way, since the number of nodes preceding v in the tree is equivalent to v's (zero-indexed) index.

Let $\#_v(u)$ be the number of vertices preceding v in a vertex u's subtree, where v is in the subtree of u. Then $\#_v(v) = \texttt{v.left.size}$ if v has a left child and zero otherwise; and can be computed in $O(1)$ time. Then, for every ancestor of u starting from v, we compute $\#_v(\texttt{u.parent})$ from $\#_v(\texttt{u})$. There are two cases:

- Case 1, `u` is the left child of `u.parent`: then all the nodes preceding v in the subtree of `u.parent` are in the subtree of `u`, so set $\#_v(\texttt{u.parent}) = \#_v(\texttt{u})$.
- Case 2, `u` is the right child of `u.parent`: then all nodes in the left subtree of `u.parent` precede v (as does u), so set $\#_v(\texttt{u.parent}) = 1 + \texttt{u.parent.left.size} + \#_v(\texttt{u})$.

Then return $\#_v(r)$, since this is the number of nodes preceding v in r's subtree (i.e., the entire tree). Correctness is argued within the algorithm description. This algorithm spends worst-case $O(1)$ work for each ancestor of v, so since the Sequence AVL Tree is balanced, the number of ancestors is bounded by $O(\log n)$, and the algorithm runs in worst-case $O(\log n)$ time.

Common Mistakes:

- Finding v given i instead of finding i given v
- Breaking early (e.g., as soon as node is a left child, instead of continuing up tree)
- Walking down the tree from root (assuming the way to go to find v)

Problem 8. [20 points] **Methane Menace**

FearBird is a supervillian who has been making small holes in the methane gas pipe network of **mahtoG City**. The network consists of n pipes, each labeled with a distinct positive integer. A hole i is designated by a pair of positive integers (p_i, d_i), where p_i denotes the label of the pipe containing the hole, and d_i is a positive integer representing the ***distance*** of the hole from the ***front*** of pipe p_i. Assume any two holes in the same pipe p_i will be at different distances from the front of p_i. When a new hole (p_i, d_i) is spotted, the city receives a ***report*** of the hole to keep track of. The city will periodically patch holes using the following priority scheme:

- if each pipe contains at most one hole, patch any hole (if one exists);
- otherwise, among pairs of holes (p_i, d_i) and (p_j, d_j) appearing on the **same pipe**, i.e., $p_i = p_j$, identify any pair with smallest distance $|d_i - d_j|$ between them, and patch one of them.

Describe a database supporting the following operations, where k is the number of recorded but unpatched holes in the network at the time of the operation. State whether your running times are worst-case, expected, and/or amortized.

`initialize(`H`)`	Initialize the database with n holes $H = \{(p_0, d_0), \ldots, (p_{n-1}, d_{n-1})\}$, with one hole on each pipe, in $O(n)$ time
`report(`p_i`,`d_i`)`	Record existence of a hole in pipe p_i at distance d_i in $O(\log k)$ time
`patch()`	Patch any hole that follows the priority scheme above in $O(\log k)$ time

Solution: To implement the database, maintain the following data structures:

- A Set AVL tree T_p for each pipe p containing all the unpatched holes in p keyed by hole distance
- A Hash Table D mapping each pipe p to its tree T_p
- A Binary Min Heap Q containing each consecutive pair of holes (p, d_1, d_2) appearing on the same pipe p with key being the distance $|d_2 - d_1|$ between them, and any lonely holes (p, d) (holes that are alone on their pipes) with key ∞ (when multiple stored items have the same key, we store them in a Hash Table keyed by (p, d_1, d_2) or (p, d))
- A Hash Table C mapping each consecutive hole pair (p, d_1, d_2) or lonely hole (p, d), to their location in Q.

Some parenthetical notes on this solution:

- A Set AVL Tree can be used for C or Q to achieve identical bounds.
- A solution without augmentation was intended, so our solution does not use it. But it is also possible to use augmentation:
  - e.g., to combine C and Q into a single Set AVL Tree keyed the same as C but storing a pointer to the min distance in subtree; or,
  - e.g., on each T_p to maintain the min distance within the pipe (to compute this augmentation efficiently, one would either need to maintain the distance to each hole's successor/predecessor (not a subtree property), or augment by min/max distance in subtree to be correct)

Operation descriptions: Continued on scratch paper S2...

Problem 9. [20 points] **Vapor Invite**

Vapor is an online gaming platform with n users. Each user has a unique positive integer ***ID*** d_i and an updatable ***status***, which can be either active or inactive. Every day, Vapor will post online an ***active range***: a pair of positive integers (a, b) with the property that **every user** having an ID d_i contained in the range (i.e., with $a \leq d_i \leq b$) **must be active**. Vapor wants to post an active range containing as many active users as possible, and invite them to play in a special tournament. Describe a database supporting the following **worst-case** operations:

`build(`D`)`	Initialize the database with user IDs $D = \{d_0, \dots, d_{n-1}\}$, setting all user statuses initially to active, in $O(n \log n)$ time
`toggle_status(`d_i`)`	Toggle the status of the user with ID d_i, e.g., from active to inactive or vice versa, in $O(\log n)$ time
`big_active_range()`	Return an active range (a, b) containing the largest number of active users possible in $O(1)$ time

Solution: To implement the database, maintain a single Set AVL Tree T containing each user ID and their status, keyed by ID. In additional, augment each node `x` in T with four subtree properties:

- `x.size`: the number of IDs in the subtree (as discussed in lecture).
- `x.suffix` $= (d, m)$: the smallest ID d in the subtree for which each of the m IDs $d' \geq d$ in the subtree is active, or None. Computable in $O(1)$ time as either the suffix (d_R, m_R) of the right subtree, or if m_R is equal to the size of the right subtree and `x` is active, return the suffix (d_L, m_L) of the left subtree but add $m_R + 1$ to m_L (or $(x.key, m_R + 1)$ if left suffix is None).
- `x.prefix` $= (d, m)$: the largest ID d in the subtree for which each of the m IDs $d' \leq d$ in the subtree is active, or None. Computable in $O(1)$ time as either the prefix (d_L, m_L) of the right subtree, or if m_L is equal to the size of the left subtree and `x` is active, return the prefix (d_R, m_R) of the right subtree but add m_L+1 to m_R (or $(x.key, m_L+1)$ if right prefix is None).
- `x.substring` $= (a, b, m)$: a, b are IDs from the subtree where each of the m IDs d in the subtree with $a \leq d \leq b$ is active and m is maximized. Computable in $O(1)$ time by taking the max of the substring within either left or right subtree, or the substring spanning the two subtrees if `x` is active. Specifically, consider the substrings of left and right subtrees, (a_L, b_L, m_L) and (a_R, b_R, m_R) respectively, and then if `x` is active, consider the suffix of the left subtree (d_L, m'_L) and the prefix of the right subtree (d_R, m'_R). Then depending on which of $(m_L, m'_L + 1 + m'_R, m_R)$ is the largest, return (a_L, b_L, m_L), $(d_L, d_R, m'_L + 1 + m'_R)$, or (a_R, b_R, m_R) respectively.

To implement `build(D)`, build the Set AVL Tree T in worst-case $O(n \log n)$ time, maintaining the custom augmentations during each insertion.

To implement `toggle_states(di)`, remove d_i from T in worst-case $O(\log n)$ time, toggle its status in $O(1)$ time, and then re-insert d_i into T in worst-case $O(\log n)$ time (again, maintaining augmentations).

To implement `big_active_range()`, simply return the substring augmentation at the root in worst-case $O(1)$ time, which is correct by the definition of our augmentation.

Common Mistakes: See scratch paper S3...

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: (Problem 6 continued...)

Worst-case Solution

Sort A and B increasing using merge sort. We give a two-finger algorithm to determine whether any $a \in A$ and $b \in B$ sum to a given $-c$. Doing this for each $c \in C$ directly determines whether a, b, c sum to 0.

Start with an index $i = 0$ into A and an index $j = n - 1$ into B and repeat the following procedure until either $i = n$ or $j = -1$:

- If $A[i] + B[j] > -c$, increase i.
- If $A[i] + B[j] < -c$, decrease j.
- Otherwise, $A[i] + B[j] = -c$, so return Yes.

If this procedure terminates without returning Yes, return No.

Correctness: We first prove the claim that the loop maintains the invariant that at the start of a loop, that no $A[i'] + B[j'] = -c$ for any $0 \leq i' < i$ or any $n - 1 \geq j' > j$. Proof by induction on $k = i - j$. This invariant is trivially true at the start when $k = i - j = -n + 1$. Assume the claim is true for all $k < k^* = i^* - j^*$. If we are at the start of a loop with $i = i^*$ and $j = j^*$, we cannot have yet returned, so at the start of the previous loop, there are two cases:

- Case 1, $(i, j) = (i^* - 1, j)$: $i^* - 1 - j = k^* - 1 < k^*$, so by induction, no $A[i'] + B[j'] = -c$ for any $i' < i^* - 1$ or $j' > j^*$. But we increased i when $A[i^* - 1] + B[j^*] > -c$, so since $B[j^*] \geq B[j']$ for all $j' > j^*$, then $A[i^* - 1] + B[j'] > -c$ for all $j' > j^*$, restoring the invariant.
- Case 2, $(i, j) = (i^*, j + 1)$: $i^* - (j + 1) = k^* - 1 < k^*$, so by induction, no $A[i'] + B[j'] = -c$ for any $i' < i^*$ or $j' > j^* + 1$. But we decreased j when $A[i^*] + B[j^* + 1] < -c$, so since $A[i^*] \leq A[i']$ for all $i' < i^*$, then $A[i'] + B[j' + 1] < -c$ for all $i' < i^*$, restoring the invariant.

So the claim is true. Then if the algorithm terminates without returning Yes, either $i = n$ or $j = -1$, so the claim implies that no integer from either A or B respectively can be in a triple with c that sums to zero.

Running Time: Sorting A and B takes worst-case $O(n \log n)$ time. We perform the two-finger algorithm n times. A single two-finger algorithm takes worst-case $O(n)$: every loop does $O(1)$ work and either increases i or decreases j, so since the loop terminates when either $i = n$ or $j = -1$, the loop executes at most $2n = O(n)$ times. So the algorithm runs in worst-case $O(n^2)$ time in total.

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.

Solution: (Problem 8 continued...)

To implement `initialize(H)`, initialize empty D and C, and then for each $(p, d) \in H$, construct an empty Set AVL Tree T_p, insert d into T_p, insert p into D mapping to T_p. Then build a hash table D_∞ on every $(p, d) \in H$, store it in Q with key ∞, and then insert each (p, d) into C mapping to D_∞. Building D_∞ takes expected $O(n)$ time, and for each (p, d), this procedure takes $O(1)$ time (expected in the case of inserting into D). So it takes expected $O(n)$ time in total, and maintains the invariants of the database directly. (Note that we could not use a Set AVL Tree for D, as it could take $\Omega(n \log n)$ time to construct.)

To implement `report(p, d)`, lookup p in D to find T_p, and then insert d into T_p.

- If d has no predecessor or successor in T_p, then insert (p, d) into Q with key ∞ and insert (p, d) into C mapping to its location in Q.
- Otherwise d has a predecessor or successor in T_p.
  - If d has a predecessor d_1 and successor d_2, lookup (p, d_1, d_2) in C to find it in Q, and then remove (p, d_1, d_2) from both C and Q.
  - Otherwise, it has one of them d', so lookup (p, d') in C to find it in Q and then remove (p, d') from both C and Q.
  - In either case, if d has a predecessor d_1, add (p, d_1, d) to Q with key $|d - d_1|$ and add (p, d_1, d) to C pointing to its location in Q;
  - and if d has a successor d_2, add (p, d, d_2) to Q with key $|d_2 - d|$ and add (p, d, d_2) to C pointing to its location in Q.

This procedure does a constant number of worst-case $O(\log k)$ time or amortized expected $O(1)$ time operations, so this operation runs in amortized expected $O(\log k)$ time, and maintains the invariants of the database by removing any consecutive pairs or lonely holes if they are no longer consecutive or lonely, and adding any new consecutive pairs or lonely holes that may have been introduced.

To implement `patch()`, delete the minimum item from Q containing one or two holes on pipe p, remove it from C, lookup p in D to find T_p, and remove the relevant holes from T_p. This procedure does a constant number of worst-case $O(\log k)$ time or amortized expected $O(1)$ time operations, so this operation runs in amortized expected $O(\log k)$ time, and maintains the invariants of the database directly. It is correct because Q exactly implements the requested priority scheme: consecutive pairs with smaller key have higher priority, and will only remove a lonely hole if there are no consecutive pairs having finite key contained in Q.

Common Mistakes: See scratch page 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.

Common Mistakes: (For Problem 8)

- Taking a minimum in some pipe rather than a minimum over all pipes
- Not prioritizing pipes having more than one hole over those with only one
- Storing an AVL or heap on all pipes which cannot be maintained within the time bounds
- Attempting an augmentation-based solution incorrectly
- Claiming $O(\log n)$-time insertion into a sorted array
- Initializing a direct access array of non-polynomially bounded size
- Using Sequence AVL Trees instead of Set AVL Trees

Common Mistakes: (For Problem 9)

- Assuming the max range always goes through the root
- Claiming a substring augmentation without showing how to maintain it in $O(1)$ time
- Using 2 AVL Trees, one of active and one of inactive (doesn't help to find largest range).
- Substring augmentation maintenance doesn't consider substrings containing the root