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 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: Modified Priorities	2	8
2: Recurrent Recurrences	2	8
3: Tree Trials	2	8
4: Missing Balls	2	20
5: Dentucky Kerby	1	20
6: Tidying Twigs	2	30
7: CartoonCon	1	24
Total		120

Name: _____

MIT Athena: _____

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. [8 points] **Modified Priorities** (2 parts)

Suppose `PQ` is a Python class that implements max priority queue operations with the following running times, where k is the number of items in the priority queue at the time of the operation:

Operation	Description	Running Time
`PQ()`	return an empty priority queue	**worst-case** $\Theta(1)$
`build(A)`	build with items from `A`	**worst-case** $\Theta(\|\texttt{A}\|)$
`insert(x)`	add item `x` to priority queue	**amortized** $\Theta(1)$
`delete_max()`	remove item with largest key	**expected** $\Theta(\log k)$, **worst-case** $\Theta(k)$

Suppose `B` is an array containing $n \gg 10$ items. For each of the following pieces of code, state both its **worst-case** and **expected** running times in terms of n, and state whether each is amortized.

(a) [4 points]

```
1 Q = PQ()
2 for x in B:
3     Q.insert(x)
4 Q.delete_max()
```

i) Worst-case:

Solution: $O(n)$, not amortized

ii) Expected:

Solution: $O(n)$, not amortized

(b) [4 points]

```
1 Q = PQ()
2 Q.build(B)
3 i = 0
4 while 10*i < len(B):
5     Q.delete_max()
6     i = i + 1
```

i) Worst-case:

Solution: $O(n^2)$, not amortized

ii) Expected:

Solution: $O(n \log n)$, not amortized

Common Mistakes:

- Thinking that doing an $\Theta(1)$-time op n times takes amortized $\Theta(n)$
- Not understanding the meaning of k (everything should be in terms of n)

Problem 2. [8 points] **Recurrent Recurrences** (2 parts)

Argue **upper** and **lower** bounds for the following recurrences. (More points will be awarded for tighter bounds.) Assume that $T(1) = \Theta(1)$, and that $T(a) < T(b)$ for all $a < b$.

(a) [4 points] $T(n) = 2\,T(\frac{n}{2}) + O(n^2)$

i) Upper bound:

Solution: $T(n) \leq 2T(\frac{n}{2}) + \Theta(n^2)$, and Case 3 of Master theorem gives upper bound $T(n) = O(n^2)$.

ii) Lower bound:

Solution: $T(n) \geq 2T(\frac{n}{2}) + \Theta(1)$, and Case 1 of Master theorem gives lower bound $T(n) = \Omega(n)$.

(b) [4 points] $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + \Theta(n)$

i) Upper bound:

Solution: Monotonicity gives $T(n) < 2T(\frac{n}{2}) + \Theta(n)$, and Case 2 of the Master theorem gives upper bound $T(n) = O(n \log n)$. We can improve this upper bound by substitution, proving $T(n) \leq 4cn = O(n)$ by induction on n.

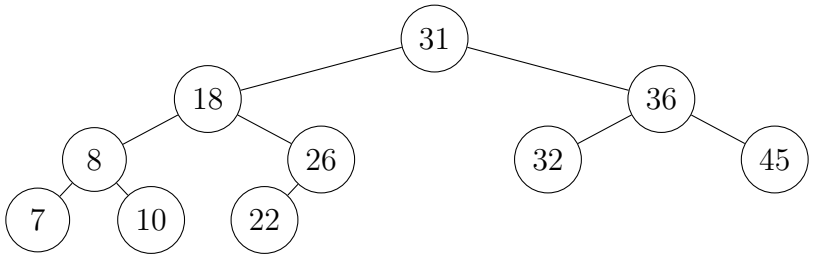
ii) Lower bound:

Solution: Monotonicity gives $T(n) > 2T(\frac{n}{4}) + \Theta(n)$ and Case 1 of Master theorem gives lower bound $T(n) = \Omega(n)$.

Common Mistakes:

- Stating bounds without argument
- Inconsistent, e.g., $f(n) = O(g(n))$ and $f(n) = \Omega(h(n))$ when $g(n) = o(h(n))$
- (a) Not recognizing that $\Theta(1) \subset O(n^2)$
- (b) Ignoring a term in the recurrence

Problem 3. [8 points] **Tree Trials**



The above binary tree T satisfies the binary search tree property. Please solve the following problems about T.

(a) [4 points] T is **packed** so could be stored in memory as a heap array. State this array.

Solution: `[31, 18, 36, 8, 26, 32, 45, 7, 10, 22]`

Common Mistakes:

- Providing an inorder traversal rather than a heap ordering
- Imposing a max or min heap property

(b) [4 points] Perform one or more rotations to T (drawing the resulting tree after each rotation) to produce a new tree T' which is (1) height-balanced, (2) maintains the binary search tree property, and (3) contains 18 at the root. (T' will not be packed)

Solution: Perform a single right rotation at the root:

```
1          _________(31)_____                  ____(18)_________
2      _____18_______    ___36___     =>    __8___        ___31_______
3    __8___     ___26    32    45          7   10      ___26     ___36___
4    7   10     22                                     22        32    45
```

Common Mistakes:

- T' is not height-balanced or does not maintain the binary search tree property

Problem 4. [20 points] **Missing Balls** (2 parts)

Jebron Lames is a professional basketball player who has played n games in his career. After each game he plays, Jebron signs a ball from the game, labels it with its **consecutive game number** i and the **positive number of points** p_i he scored that game, and stores the ball in his trophy room. The day after playing game number n, Jebron comes home to find his balls jumbled in disarray; someone has robbed his trophy room! Help Jebron file his police report. For each of the following parts, state whether your algorithms achieve worst-case, expected, and/or amortized running times.

(a) [10 points] Given a list of the game numbers from all jumbled balls that were not stolen, describe an efficient[1] algorithm to compile a list of all game numbers from balls missing from Jebron's trophy room. (Remember to provide arguments for **correctness** and **running time** for any algorithm you describe on this quiz.)

Solution: Construct a direct access array A of size $n + 1$ fully initialized to 0 in worst-case $O(n)$ time (so that A has indices 0 to n). Then, for each game number i from balls that were not stolen (which has a unique value between 1 and n inclusive), change index i of A from 0 to 1 in worst-case $O(1)$ time. There are at most n balls that were not stolen, so this process takes at most worst-case $O(n)$ time. After this process, every ball number i from 1 to n satisfies the property that $A[i]$ is 0 if ball i was stolen and 1 otherwise. So to return a list of the missing balls, scan through A from $i = 1$ to n and append all i such that $A[i]$ is 0 to the end of a dynamic array in amortized $O(1)$ time, which is correct by the property previously stated. Because this last step also runs in worst-case linear time, so does the entire algorithm. This algorithm is efficient because every ball number must be read or returned, so any correct algorithm will take at least $\Omega(n)$ time.

Common Mistakes:

- Not justifying a linear sort
- Using hashing yielding expected instead of worst-case running time

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

(b) [10 points] Older and higher scoring balls are worth more; specifically, the value v_i of ball i is $v_i = 3^{p_i}(n-i)^2$. Given a list of game number-point pairs (i, p_i) for the missing balls sorted by game number, describe an efficient algorithm to compute the **median** value of the missing balls. Assume that: at least half the balls are missing, the number of missing balls is odd, and that the maximum number of points Jebron has scored in any game is at most $23 \lg n$.

Solution: Compute the value of each of the $n/2 \leq k \leq n$ missing balls, and sort them by value. Then return the middle value, which will correctly return the median by definition. It remains to specify how the values can be computed and sorted efficiently. Since $p_i \leq 23 \lg n$ for all i from 1 to n, then $v_i \leq 3^{23 \lg n} n^2 = O(n^c)$ for constant $c = 2 + 23 \lg 3$, so if we had an array of the values, we could use radix sort to sort them in worst-case $O(n)$ time. Performing $\Theta(\log n)$ arithmetic operations to compute each value would take $O(n \log n)$ time in total. We can do better by first precomputing the values 3^i for i from 0 to $23 \lg n$ in $O(\log n)$ time and store them sequentially in a direct access array. Then each value can be computed in $O(1)$ time by looking up the appropriate value of 3^{p_i} stored at index p_i in the direct access array and performing a $O(1)$ number of arithmetic operations. So all values can be computed in $O(n)$ time, acheiving worst-case $O(n)$ time in total. This algorithm is efficient because every ball could be the median, so any correct algorithm will take at least $\Omega(n)$ time.

Common Mistakes:

- Not justifying a linear sort
- Assuming exponentiation is a $O(1)$-time operation
- Tuple sorting (i, p_i) pairs directly with p_i more significant (not always true)
- Trying to use a Set AVL without augmenting by size to find median

Problem 5. [20 points] **Dentucky Kerby**

The Dentucky Kerby is one of the most selective horse races in America. To qualify, a horse must participate in multiple qualifying races during the season. The **results** of a given qualifying race is a list R containing the names of the $|R|$ horses that participated in that race in the rank order in which they finished[2]. At the end of the season, the k horses with the lowest average rank over all qualifying races will be invited to race in the Dentucky Kerby (ties may be broken arbitrarily). Note that k is chosen at the start of the season, and does not change during the season. Describe a database supporting the following operations, where n is the number of horses stored in the database at the time of the operation. For each operation, state whether your running time is worst-case, expected, and/or amortized.

`record_race(R)`	record results `R` from a new qualifying race in $O(\|\texttt{R}\| \log n)$ time
`top_k()`	return names of k horses with lowest average rank in $O(k)$ time

Solution: This problem is similar to PS4-4. For each horse h, maintain its name $h.name$, the number $h.n$ of qualifying races in which it has participated, and the sum total $h.s$ of its ranks in all races. Maintain two priority queues storing horses keyed on rank average, which can be compared in $O(1)$ time via cross multiplication (i.e., $\frac{s_i}{n_i} < \frac{s_j}{n_j}$ if and only if $s_i n_j < s_j n_i$). The first P_1 is a max priority queue that maintains the up to k horses with lowest average rank, and the second P_2 is a min priority queue that maintains all horses who have raced that are not in P_1. We can acheive the desired time bounds by implementing these priority queues with either binary heaps or Set AVL trees; we will assume Set AVL trees. In addition, maintain a dictionary D mapping each horse's name to its location in either P_1 or P_2. We can acheive the desired time bounds by implementing the dictionary using either a hash table or an Set AVL tree; we will assume a

To implement `record_race(R)`, for the i^{th} horse h_i in R, find h_i in its priority queue using D in $O(\log n)$ time. If h_i is not in either priority queue, construct a new horse object with $h_i.n = 0$ and $h_i.s = 0$ in $O(1)$ time. In either case, increase the horse object properties $h_i.n$ by one, and $h_i.s$ by i in $O(1)$ time, restoring the invariant the each horse object maintains their properties. If the number of horses currently being considered is at most k, then insert h_i into P_1 in $O(\log n)$ time. Otherwise remove the max h_1 from P_1 and min h_2 from P_2, each in $O(\log n)$ time, compare them in $O(1)$ time, and reinsert all three into either P_1 and P_2 in $O(\log n)$ time to restore the invariant the P_1 contains k lowest average rank horses. Lastly, update D so that each of h_i, h_1, and h_2 point to their correct places in P_1 and P_2 in $O(\log n)$ time. Doing this for each horse in R can then be done in $O(|R| \log n)$ time. This algorithm correctly restores the invariants of P_1, P_2, and D.

To implement `top_k()`, perform an in-order traversal of P_1 in $O(k)$ time and return each horse's name in sequence. This is correct because we maintain the invariant that P_1 contains k horses with lowest average rank.

Common Mistakes:

- Not indicating how averages should be stored/compared
- Maintaining k with lowest average rank without maintaining others
- Using a single AVL tree to acheive $O(k + \log n)$ running time for `top_k`

[2]For example, the name of the second-place horse in a qualifying race would appear as the second name in the results listing, with that horse achieving rank 2 in that qualifying race.

Problem 6. [30 points] **Tidying Twigs** (2 parts)

Gryan Briffin is a dog who has n favorite fetching sticks. Each stick has a **unique** positive integer length $\ell_i \in L$ where L is a list of all stick lengths, $(\ell_0, \ell_1, \ldots, \ell_{n-1})$. Gryan is a tidy dog and wants to store the sticks neatly in the corner of the room, using exactly two of the sticks to demarcate a rectangular storage area on the ground (in which to store the remaining $n-2$ sticks). There will be enough room to store the sticks if Gryan can find two border sticks ℓ_a and ℓ_b such that the rectangular area $\ell_a \times \ell_b$ formed by the border is **at least as large** as the total length $T = \sum_{\ell_i \in L} \ell_i$ of all the sticks[3]. For each of the following parts, state whether your algorithms achieve worst-case, expected, and/or amortized running times.

(a) [15 points] Given L, describe an $O(n \log n)$-time algorithm to find the lengths of two sticks ℓ_a and ℓ_b such that $\ell_a \times \ell_b - T$ is minimized, subject to $\ell_a \times \ell_b \geq T$ (or return that no pair ℓ_a and ℓ_b exist such that $\ell_a \times \ell_b \geq T$).

Solution: This problem is similar to PS3-4b. First, compute T by summing up each length in worst-case $O(n)$ time. Sort the lengths in L using a worst-case $O(n \log n)$ comparison sorting algorithm, like merge sort or heap sort, into an array B. Then we can sweep the sorted array using a two-finger algorithm. Initialize $i = 0$ and $j = n-1$, and repeat the following procedure, maintiaining two indices a and b initially undefined. If $B[i]B[j] < T$, then $B[i]B[k] \leq B[i]B[j] < T$ for all $k < j$: such pairs (i, k) can never satisfy the condition, so increase i by one. Otherwise, $T \leq B[i]B[j]$. If $B[i]B[j] < B[a]B[b]$ (or if a and b are undefined), then we've found a better pair of sticks, so set $(a, b) = (i, j)$; but either way, $B[a]B[b] < B[k]B[j]$ for any $k > i$: such pairs (k, j) cannot yield better than our current estimate, so decrease j by one. If $j < i$, return $B[i]$ and $B[j]$ if they are defined, or else return that no pair exists. This algorithm maintains the invariant that at the start of each loop, $\ell_a \times \ell_b \geq T$ is smallest for all a', b' with $0 \leq a' < i$ and $j < b' < n$, or a and b are undefined if no such a', b' exist, so the algorithm is correct. Since each iteration of the loop takes $O(1)$ time, decreases $j - i$ by one, and $j - i$ starts at $n - 1$, this procedure takes at most $O(n)$ time in the worst case.

Common Mistakes:

- Trying to sort in linear time
- Recomputing T for each check (extra linear factor)
- Returning wrong pair (largest, smallest, closest to $\sqrt{T}$, closest to median, etc.)

[3] Gryan doesn't know that comparing areas to lengths may not be a good idea...

(b) [15 points] Gryan, too impatient for the previous algorithm, now just wants to check whether any pair can form a rectangle with area **equal** to the total length. Given L, describe an $O(n)$-time algorithm to determine whether there exist two sticks ℓ_a and ℓ_b such that $\ell_a \times \ell_b = T$.

Solution: This problem is similar to PS3-4a. Again, compute T in $O(n)$ time as in (a). It suffices to check for each ℓ that evenly divides T whether $T/\ell \in L$, which would be correct by exhaustive search. To perfom this check efficiently, insert each length into a hash table H in expected $O(n)$ time. Then for each $\ell \in L$, check whether $T \equiv 0 \bmod \ell$ in $O(1)$ time (e.g., using `%`). If so compute $\ell' = T/\ell$ (an integer) in $O(1)$ time and check whether ℓ' is in H in expected $O(1)$ time. If yes and $\ell \neq \ell'$, then return ℓ and ℓ'. Otherwise ℓ and ℓ' do not satisfy the condition so continue with the next length to check. This algorithm runs in expected $O(n)$ in total because processing each length takes expected $O(1)$ time.

Common Mistakes:

- Trying to sort in linear time
- Trying to compute on real numbers without bounding precision
- Using same stick length twice when T is a perfect square

Problem 7. [24 points] **CartoonCon**

Mundall Ranroe is organizing CartoonCon, a national convention for cartoonists like herself. Event scheduling for the convention is crowd-sourced. If an attendee would like to host an event during the convention, they will submit to the convention website: the event's name, and two integers denoting the starting and ending times for the event. Two time ranges **overlap** if there exists a time that is in both ranges that is not an endpoint of either range[4]. While events may be scheduled to overlap, Mundall would like to help attendees determine whether a particular time range is currently **available**, i.e., no existing event overlaps the range. Describe a database supporting the following operations, each in worst-case $O(\log n)$ time, where n is the number of events scheduled at the time of the operation.

`schedule(e, s, t)`	add event named `e` starting at time `s` and ending at time `t`
`unschedule(e)`	remove event named `e` from the schedule (if it exists)
`check_avail(s, t)`	return whether an existing event overlaps the time range from `s` to `t`

Hint: Try storing events in a modified Set data structure keyed on **starting times**.

Solution: Maintain a Set AVL tree T storing event start and end times keyed on start time. Let node $v \in T$ have start time $v.s$ and end time $v.t$. Augment each node v with the maximum end time $v.m$ of any event in v's subtree. This augmentation can be computed in $O(1)$ time from the augmentations of its children by taking the maximum between the node's ending time and the augmentations of the node's children (if they exist). In addition, maintain a dictionary D mapping each event name to the node in T storing that event. Implement D using a Set AVL tree. All running times quoted in this solution are worst-case.

To implement `schedule(e, s, t)`, add (s, t) to T in $O(\log n)$ time (maintaining augmentations as described above), and add e to D mapping to the node in T storing (s, t), aslo in $O(\log n)$ time. This algorithm maintains the invariants of T and D. To implement `unschedule(e)`, look up e in D in $O(\log n)$ time and remove the node in T associated with e, also in $O(\log n)$ time. This algorithm maintains the invariants of T and D.

To implement `check_avail(s, t)`, we implement recursive function $f(v, s, t)$ which returns whether any event in node v's subtree overlaps range (s, t). Note that `check_avail(s, t)` $= f(r, s, t)$ where r is the root of T. To implement $f(v, s, t)$, we first note that if $s < v.s < t$, then we can return True since the event at v overlaps the range. Otherwise, either $t \leq v.s$ or $v.s \leq s$. If $t \leq v.s$, then the event at v and the events in v's right subtree cannot overlap the range, since t is less than all of their starting times, so we can recursively return $f(v.left, s, t)$ (or False if $v.left$ does not exist). Alternatively, if $v.s \leq s$, we must check of overlap with the event at v, and events in the subtrees of v's children. If $s < v.t$ then we can return True since the event at v overlaps the range. Otherwise, if v has a left subtree and $s < v.left.m$, then the range overlaps some event in v's left subtree, since the event starts before s and ends strictly after s. Otherwise, the range does not overlap any event in v's left subtree, and we can recursively return $f(v.right, s, t)$ (or False if v's right subtree does not exist). This algorithm is correct by induction on the nodes in level order from lowest leaf to the root and runs in $O(\log n)$ time because each recursive step takes $O(1)$ time and makes at most one recursive call down a tree of height $O(\log n)$.

[4]Specifically, given two time ranges (a, b) and (c, d) where $a < b$ and $c < d$, the ranges would not overlap if $b \leq c$ or $d \leq a$, but would overlap in all other cases (for example, if $c < b < d$ or $a \leq c < d \leq b$).

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: (Problem 7)

- Using a hash table (good expected bounds, not worst-case)
- Trying to search by name in a tree keyed by times
- Augmenting by a property that cannot be maintained
- Not bounding branching of recursive query leading to linear time

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.