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: Binary Array	1	10
2: Arboreal Attributes	3	15
3: Slice Merge	3	18
4: Nick Sorts	3	18
5: Sum-Chums	1	15
6: Alien Arrival	1	20
7: Local Dynamic Seq.	1	22
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!

Common Mistakes: Not writing your name.

Problem 1. [10 points] **Binary Array**

A **binary array** is an array of integers that contains only 0's and 1's. Given a length-n binary array that is also **sorted increasing**, describe an $O(\log n)$-time algorithm to compute the sum of the array. Remember to argue the correctness and running time of your algorithm.

Solution: We describe an algorithm analogous to binary search. Let $M(i, j)$ be the sum of subarray `A[i:j]`. To compute $M(i, j)$, check the middle element at array index `c = (i + j)//2`. If `A[c]` is 0, the sum of subarray `A[i:c]` equals 0, so $M(i, j) = M(c, j)$. Otherwise, if `A[c]` is 1, the sum of subarray `A[c:j]` equals $j - c$, so $M(i, j) = M(i, c) + j - c$. In either case, computing $M(i, j)$ makes a recursive call on a subarray of roughly half the size, until reaching a subarray containing only one element where $j = i + 1$, and the sum is simply the value at `A[i]`. Evaluating $M(0, n)$ then computes the sum of the array. The recurrence for this algorithm is the same as binary search, $T(n) = T(n/2) + O(1)$, leading to an $O(\log n)$ running time.

Common Mistakes:

- A regular binary search for 1 isn't guaranteed to find the **earliest** 1 in the array; it stops when it finds **any** instance of the requested key.
- This is not a data structure design question, so any data structures other than the provided array need to be built explicitly. An AVL tree would be nice, but we'd have to construct it ourselves in $O(n)$ time.
- If you find that your runtime doesn't match our request, it's best to report the correct runtime **for your algorithm**. Similarly, when using a recurrence, be sure to solve the recurrence instead of assuming/hoping it matches the desired bounds. You can get partial credit for runtime analysis, but not if your reported runtime doesn't match your algorithm.

Problem 2. [15 points] **Arboreal Attributes** (3 parts) In this class, we've defined three properties of binary trees that store keys at their nodes:

- **Max-Heap Property:** for any node `v`, the key stored at `v` is **at least as large** as every key stored in the subtree rooted at `v`.
- **BST Property:** for any node `v`, the key stored at `v` is **at least as large** as every key in the subtree rooted at `v.left`, and **less or equal to** every key in the subtree rooted at `v.right`.
- **AVL Property:** for any node `v`, the **absolute difference** between the heights of the subtrees rooted at `v.left` and `v.right` is at most 1.

An AVL Tree satisfies both the AVL and BST Properties, while a Max-Heap satisfies both the AVL and Max-Heap Properties. Answer the following questions about binary trees and these properties.

(a) [5 points] Draw a binary tree on the keys $\{1, 2, 3, 4, 5, 6, 7\}$ that satisfies the Max-Heap Property, **but not** the BST or AVL Property (tree should **not** be height balanced).

Solution: There are many possible solutions. Here are three:

```
7___                    ______7___                          ______7___
   6___                 2___     6___                 ______6___     4
      5___                 1        5___           ___3___     5
         4___                          4___        2     1
            3___                          3
               2___
                  1
```

(b) [5 points] Draw a binary tree on the keys $\{1, 2, 3, 4, 5, 6, 7\}$ that satisfies both the Max-Heap Property **and** the BST Property, **but not** the AVL Property.

Solution: There is only one solution:

```
               ___7
            ___6
         ___5
      ___4
   ___3
___2
1
```

(c) [5 points] Draw a binary tree on seven nodes containing any integer keys, which satisfies all three: the Max-Heap Property, the BST Property, **and** the AVL Property.

Solution: There are many possible solutions, but main point is that the tree must contain repeated keys:

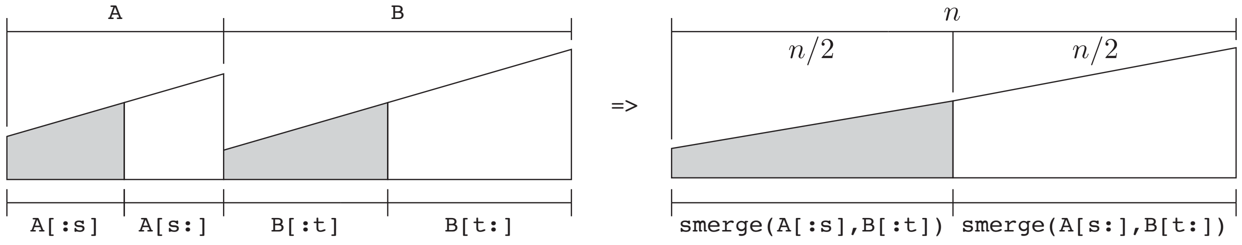
```
   ______1______                  ______3___
___1___     ___1___                ___2___     3___
1     1     1     1             ___1     2        3
                                0
```

Common Mistakes: (for all 3 parts)

- Nodes in a binary tree are allowed to have just one child.
- The Max-Heap Property can apply to binary trees of any shape, but a Max Heap by definition must have the Max-Heap Property **and** be a complete binary tree. So there are many trees that have the Max-Heap Property without being a Max Heap.

Problem 3. [18 points] **Slice Merge** (3 parts)

In this problem, we will analyze an alternative to the merge step of merge sort. Suppose `A` and `B` are sorted arrays with possibly **different** lengths, and let $n =$ `len(A)+len(B)`. You may assume n is a power of two and all n items have **distinct** keys. The **slice merge** algorithm, `smerge(A,B)`, merges `A` and `B` into a single sorted array as follows:



(1) Find indices s and t such that $s+t = n/2$ and the prefix subarrays `A[:s]` and `B[:t]` together contain the smallest $n/2$ keys from `A` and `B` combined.

(2) Recursively compute `X = smerge(A[:s], B[:t])` and `Y = smerge(A[s:], B[t:])`, and return their concatenation `X + Y`, a sorted array consisting of all items from `A` and `B`.

For example, if `A = [1,3,4,6,8]` and `B = [2,5,7]`, we find $s = 3$ and $t = 1$ and then recursively compute:

```
smerge([1,3,4], [2]) + smerge([6,8], [5,7]) = [1,2,3,4] + [5,6,7,8].
```

(a) [10 points] Describe an algorithm to find indices s and t satisfying step (1) in $O(n)$ time, using only $O(1)$ additional space beyond arrays `A` and `B` themselves. Remember to argue the correctness and running time of your algorithm.

Solution: To find indices s and t, begin with $s = 0$ and $t = 0$. If `A[s] < B[t]`, add one to s, otherwise, add one to t, and repeat until $s + t = n/2$. Claim: at the end of each iteration, `A[:s]` and `B[:t]` contains the $s+t$ smallest keys from `A` and `B` combined. Proof by induction. At that start, $s + t = 0$ so the claim is vacuously true. Now assume the claim holds at end of iteration $k = s + t$ so that `A[:s]` and `B[:t]` contain the smallest $s + t$ elements from `A` and `B`. Arrays `A` and `B` are sorted, so the minimum of elements from `A[s:]` and `B[t:]` is either `A[s]` or `B[t]`. The procedure adds one to whichever index contains the smaller of the two, reinstating the inductive hypothesis. This algorithm does constant work in each of $n/2$ iterations, using only constant external storage to keep track of s and t, so this algorithm takes $O(n)$ time.

Common Mistakes:

- The proof of correctness needs a loop invariant, a.k.a. inductive hypothesis—something that is true at **every** iteration, not just at the end. For example, "on each iteration, `A[:s]` and `B[:t]` together contain the **smallest $s + t$ elements** from A and B", or "each step chooses the **smallest unselected element** from A and B, because the arrays are sorted". Contrast this with "on each iteration we pick the smaller of $A[s]$ or $B[t]$ so we end up with the smallest $n/2$ elements," which simply restates the steps of the algorithm and the desired result without explaining how we got there.
- `A[:s]` has s elements, not $s - 1$. Even if $s = 0$.
- A new dynamic array holding the $n/2$ elements uses $O(n)$ space, not $O(1)$, even though it only took $O(1)$ space when it was empty at the beginning.

(b) [4 points] **Write** and **solve** a recurrence for $T(n)$, the running time of `smerge(A,B)` when `A` and `B` contain a total of n items (please show your work). How does this running time compare to the `merge` step of `merge_sort`?

Solution: $T(n) = 2T(n/2) + \Theta(n)$, so $T(n) = \Theta(n \log n)$ by Case II of the master theorem, since $f(n) = \Theta(n) = \Theta(n^{\log_2 2})$.

(c) [4 points] Let `smerge_sort(A)` be a variant of `merge_sort(A)` that uses `smerge` in place of `merge`. **Write** and **solve** a recurrence for the running time of `smerge_sort(A)`.

Solution: $T(n) = 2T(n/2) + \Theta(n \log n)$, so $T(n) = \Theta(n \log^2 n)$ by Case II of the master theorem, since $f(n) = \Theta(n \log n) = \Theta(n^{\log_2 2} \log^1 n)$.

Common Mistakes: (for parts (b) and (c))

- Using O and Θ interchangeably. Use Θ when it applies, since it is more precise.

Problem 4. [18 points] **Nick Sorts** (3 parts)

Solve each of the following sorting problems by choosing an algorithm or data structure that best fits the application, and justify your choice. **Don't forget this! Your justification will be worth more points than your choice.** You may choose any algorithm or data structure we have covered in this class, or you may modify them as necessary to fit the scenario. If you find that multiple solutions could be appropriate for a scenario, identify their pros and cons, and choose the one that best suits the application. State and justify any assumptions you make. "Best" should be evaluated primarily by asymptotic running time, with other factors secondary.

(a) [6 points] At her birthday party, Pelica Angickles receives one gift from each of her n friends. After the party, she searches online to find out how much each friend spent on their gift. The most expensive gift she received was from her cousin, Pommy, who spent \$60.06. Pelica decides to rank her friends according to how much they spent. Help Pelica rank her friends.

Solution: We assume that the price of any gift is an integer number of cents between 0 and 6006. Then, we can use counting sort to sort her friends in linear time, which is faster than other algorithms. Radix sort can also be accepted, depending on how n compares to 6006.

Common Mistakes: We can use counting and/or radix sort even when we don't have bounds on n—this only requires bounds on how big the keys can get (relative to n), and here the keys are bounded by $6006 = O(1)$.

(b) [6 points] The memory from Neummy Jitron's spaceship computer has malfunctioned, leaving only a **constant amount** of onboard memory uncorrupted and accessible. Luckily, Neummy has an external disk containing all his timestamped travel logs, but in a random order. The computer can **quickly read and compute** on a small amount of data from the disk at a time, but **writing** to the disk is **extremely slow** (i.e. all that matters is the number of external writes to disk). Help Neummy rewrite the disk to store travel logs chronologically.

Solution: This problem asks us for a sorting algorithm which minimizes the number of writes. We will need at least $\Omega(n)$ writes, for example in the case where no travel log is in correct position. Thus, we would like an algorithm that only uses $O(n)$ writes. Counting sort only uses $O(n)$ writes, but we do not have a bound on the representation of timestamps, so we may have to allocate $u > n$ additional space. Selection sort on the other hand makes $O(n^2)$ reads and comparisons, but only makes $O(n)$ swaps, and thus only $O(n)$ writes, so we choose selection sort.

Common Mistakes:

- Heapsort is good at minimizing extra memory allocation but it can still sometimes make $\Omega(n \log n)$ swaps and therefore $\Omega(n \log n)$ writes.
- Confusing selection sort with insertion sort. Insertion sort swaps an item leftward repeatedly until it finds its place, sometimes making $\Omega(n^2)$ swaps. By contrast, selection sort makes a single swap per element.

(c) [6 points] The Bum Chucket is an underwater restaurant, owned by Pleldon Shankton and managed by an underwater computer named Plaren. During the day, Plaren records the name of each customer, and the amount they spent on food. Assume that no customer purchases food more than once per day. Periodically, Pleldon wants to know the names of the k highest spenders so far that day. Help Plaren store information to quickly maintain and generate this list for Pleldon upon request.

Solution: Have Plaren maintain a binary min-heap or AVL tree storing the names of customers and keyed by amount spent, maintaining the k highest spenders since the beginning of the day. To maintain, insert the first k customers into the database, and then for each subsequent customer c, remove the minimum spending customer m from the database, and reinsert either c or m, whichever spent more. This storage method takes $O(n \log k)$, with $O(\log k)$ time spent per customer.

Common Mistakes:

- The value k is **unchanging**, meaning you know it in advance and can design your algorithm using it. But it is **not $O(1)$**, i.e., **not constant sized**, meaning big-O notation can't ignore it like it ignores constant factors.
- Storing all n items, when we only need to store k items.
- Finding the k largest keys in an AVL tree takes $O(k + \log n)$ time, not $O(k)$.

Problem 5. [15 points] **Sum-Chums**

Given an array A of n **positive** integers, a pair of array indices (i, j) are **sum-chums** if

$$(A[i] - i) + (A[j] - j) = 0 \qquad \text{or equivalently,} \qquad A[i] + A[j] = i + j.$$

Note that $A[i] + A[j] < 2n$ for any sum-chum pair (i, j), since $i, j \in \{0, \ldots, n-1\}$. Describe a **worst-case** $O(n)$-time algorithm to determine whether an array A contains a sum-chum pair. Maximum 10 points for an expected $O(n)$-time solution. Remember to argue the correctness and running time of your algorithm.

Solution: The given problem is equivalent to computing the value of $A[i] - i$ for each index i, and determining if there are two distinct indices for which the values sum to 0. We're assuming in this solution that i and j have to be distinct to be a sum-chum pair, but as this was not specified, either interpretation was accepted.

Expected Linear: Create a hash table. For each value of i from 0 to $n - 1$, we compute the value $k_i = A[i] - i$. We can check if key $-k_i$ exists in the hash table: if it does, then the corresponding value j is an index such that $(A[i] - i) + (A[j] - j) = 0$; otherwise it does not exist, so insert key k_i into the hash table with corresponding value i. If no match in the hash table is found, then there does not exist a pair of indices such that the corresponding values sum to 0. Each insert and find in the hash table takes expected $O(1)$ time, so the algorithm runs in expected $O(n)$ time.

Worst-case Linear 1: We use the same strategy as the hash table solutions, but use a direct access array instead, noting that for any sum-chum pair (i, j), $0 \le A[i] - i + n \le 3n$. So we initialize a direct access array with length $4n$, inserting on keys $n + k_i$ and finding collisions at keys $n - k_i$, for any key within range of the direct access array.

Worst-case Linear 2: Use counting or radix sort. Recall that $A[i] + A[j] < 2n$, and since the array contains only positive integers, any satisfying $A[i]$ must be $< 2n$. We again compute the value of $k_i = A[i] - i + n$ for each i, and eliminate any value for which $A[i] > 2n$. These integer values are bounded above by $3n$ and below by 0, so we can perform radix sort on the values, also storing the associated indices. We can then start a pointer at the first and last items, and either increment the left pointer or decrement the right pointer if the sum is greater than or less than $2n$ respectively. If a sum of $2n$ is identified, then the corresponding indices i and j can be returned, otherwise no such pair exists. Computing the values and sorting using counting or radix sort takes $O(n)$ time, and using the two pointers to identify a sum of $2n$ also takes $O(n)$ time, so the algorithm runs in worst-case $O(n)$ time.

Common Mistakes:

- Worst-case runtime is stronger than expected runtime. For example, a worst-case $O(1)$ operation **is** an expected $O(1)$ operation, but the reverse is not always true. If a single step in your algorithm has expected runtime (e.g. hash table lookup), your algorithm's runtime **overall** is expected, not worst-case.
- Hash tables will likely have collisions even when the data have distinct keys, so hash table lookups are always **expected** $O(1)$, not worst-case.
- Sorting array A changes the index of each item. The problem asks about the **original** indices, not the sorted ones, so these need to be accounted for prior to sorting.
- Many students stored the positive and negative values of $A[i] - i$, forgetting that this value might also be 0.
- Ignoring sign and looking for duplicate $|A[i] - i|$ values does not work.

Problem 6. [20 points] **Alien Arrival**

Aliens have landed on earth, but humans are having a difficult time communicating with them. The aliens have allowed linguistics professor Bouise Lanks onto their ship, which houses a library containing $\boldsymbol{B}$ **engraved blocks**, each of which is covered in many symbols. There are $\boldsymbol{S}$ **symbols** in the alien language: each symbol is made up of line segments, and no two symbols have the exact same number of segments. Using carbon dating technology, Dr. Lanks can measure how many earth days have passed since each block was engraved. Help Dr. Lanks design a database to store alien symbol information, supporting the following two operations each in $O(\log B)$ time. Remember to argue the correctness and running time of your operations. State whether the running time of each operation is a worst-case, amortized, and/or expected bound.

1. RECORD-SYMBOL(s, d): record that the symbol having s line segments was engraved once on some block d days ago.
2. COUNT-SYMBOL(s, d_1, d_2): return the number of times the symbol having s line segments was engraved by the aliens during the time between d_1 and $d_2 < d_1$ days ago.

Solution: Store hash table keyed by symbol segment number, mapping each symbol s to an (initially empty) AVL tree keyed by engraving date, where each node in symbol s's AVL tree with key d stores the number of times symbol s was engraved exactly d Earth days ago. The hash table has size $O(S)$, while each AVL tree has size $O(B)$. Augment each AVL node to store subtree min date, max date, and total number of engravings stored in the node's subtree, as in PS4-3. As was shown in in PS4-3, we can maintain such augmentations during dynamic operations, and can support querying the total number of engravings between two dates in $O(\log B)$ time.

To support RECORD-SYMBOL(s, d), search in the symbol hash table to find the AVL tree associated with symbol s in expected $O(1)$ time, and then find d in worst-case $O(\log B)$ time. If d is in the AVL tree, add 1 to its count; or if d is not yet present, add a new node with key d with count 1. Along with normal AVL maintenance, maintain subtree augmentations. This operation runs in expected $O(\log B)$ time.

To support COUNT-SYMBOL(s, d_1, d_2), search in the symbol hash table to find the AVL tree associated with symbol s in expected $O(1)$ time, and then query the range (d_1, d_2) as in PS4-3 to compute the total number of times symbol s was engraved in worst-case $O(\log B)$ time. This operation runs in expected $O(\log B)$ time.

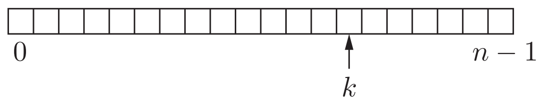
Common Mistakes:

- Using an AVL tree where each node stores hash tables of $\Theta(S)$ symbols is not maintainable in $O(\log B)$ time, as rotations may require updates that take $\Theta(S)$ time.
- Many students misunderstood or misused the AVL range-query augmentation technique from pset problem 4-3. This is worth reviewing.
- Augmentations must be accompanied by their update method, because many attempted augmentations cannot be updated cleanly. For example, "the number of nodes in my subtree with the same symbol as me" is not easily propagated up the tree. Another common wrong augmentation was prefix or suffix sums, which are not subtree properties.
- A common strategy was to make a size-B AVL tree where each node is a size-S AVL tree, and to forget that inserting into the inner tree takes $O(\log S)$ time.

Problem 7. [22 points] **Local Dynamic Sequence**

Design a data structure that maintains a sequence $x_0, x_1, \ldots, x_{n-1}$ of n items, maintains an index k satisfying $0 \le k < n$, and supports the following operations, each in $O(1)$ time. Remember to argue the correctness and running time of your operations. State whether the running time of each operation is a worst-case, amortized, and/or expected bound.

1. AT(i): return item x_i given **any** i where $0 \le i < n$. (Same as in the sequence interface.)
2. INDEX$()$: return the current index k.
3. MOVE-LEFT$()$: decrement the index: $k \leftarrow k - 1$.
4. MOVE-RIGHT$()$: increment the index: $k \leftarrow k + 1$.



5. INSERT-HERE(x): shift-right all items to the right of the index ($x_k \to x_{k+1} \to \cdots \to x_{n-1} \to x_n$), set $x_k = x$, and increment n. (Same as sequence-interface operation INSERT-AT(k, x).)
6. DELETE-HERE$()$: shift-left all items to the right of the index ($x_k \leftarrow x_{k+1} \leftarrow \cdots \leftarrow x_{n-1}$), and decrement n. (Same as sequence-interface operation DELETE-AT(k).)

Maximum 5 points if any operation takes $\Omega(n)$ amortized expected time.

Solution: Maintain two stacks L and R, implemented using dynamic arrays, where L represents the sequence of items $x_0, \ldots, x_k$ and R represent the sequence of items $x_{n-1}, \ldots, x_{k+1}$ (in reversed order).

1. AT(i): If `i < len(L)`, return item `L[i]`. Otherwise, `i` references an item in `R`, so return item `R[len(L) + len(R) - 1 - i]`. This operation takes worst-case $O(1)$ time.
2. INDEX$()$: Simply return `len(L)` in worst-case $O(1)$ time.
3. MOVE-LEFT$()$: if `len(L) > 0`, remove the right-most item from `L` with `L.delete_right()`, and append as the right-most item of `R` with `R.insert_right()` in amortized $O(1)$ time.
4. MOVE-RIGHT$()$: if `len(R) > 0` remove the right-most item from `R` with `R.delete_right()`, and append as the right-most item of `L` with `L.insert_right()` in amortized $O(1)$ time.
5. INSERT-HERE(x): `R.insert_right(x)` x as the right-most item of `R` in amortized $O(1)$ time.
6. DELETE-HERE$()$: `R.delete_right()` the right-most item of `R` in amortized $O(1)$ time.

Common Mistakes:

- Many students used hash tables with keys $\{0, 1, \ldots, n-1\}$ instead of direct access arrays, which is not wrong but unnecessarily introduces expected time bounds instead worst-case time bounds.
- Confusing different types of arrays: static vs dynamic vs direct access.
- If maintaining two cross-linked data structures, you must explain how to maintain the cross references through each operation. Many solutions failed by not noticing linear-time updates to these cross references.
- Putting $O(1)$ or $\Theta(n)$ empty space between every pair of items doesn't work—it requires too frequent maintenance to amortize to $O(1)$ per operation. In the latter case, just allocating the requisite space takes $\Theta(n)$ amortized work per insert.
- We can't make assumptions about how the user will query this data structure. Amortized bounds that rely on the user calling `at(i)` much more frequently than `insert-here(x)` (for example) are not properly using the meaning of "amortized."

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.

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.