Midterm 1 Solutions

Instructions:

- Do not begin the quiz until directed to do so. Read all the instructions on this page.
- Write your name below. In addition, write your name at the top of **every** subsequent page.
- Please do not separate or detach pages. All pages must be available for scanning.
- **You are allowed one double-sided, letter-sized sheet with your own notes**. Calculators, cell phones, programmable devices and communication devices are prohibited.
- Do not discuss the exam until after grades have been released.
- You should be able to fit your solutions in the spaces provided. If you need more space, continue on the extra pages at the end of the exam booklet. Make sure to leave pointers in both directions (e.g. Continued on Extra Page 2 / Problem 4 continued) so that we can find your solutions.
- Solutions must be **clear and concise**. Algorithms should be described in English, not written in Python.
- Proof is always required unless otherwise specified.
- Runtime analysis of an algorithm requires both upper and lower bounds unless otherwise specified.

Advice:

- You have 120 minutes to earn a maximum of 100 points. **Do not spend too much time on any single problem.** Read them all first, and attack them in the order that allows you to make the most progress.
- Do not waste time re-deriving results from class. Simply state and cite them.

Question	Points
Multiple Choice	25
Sleepy Sliggoo	25
#4SUM	15
Heaps	12
Augmentations	23
Total:	100

Name: ______________________ MIT ID (9 digit #): ______________________

Problem 1. [25 points] **Multiple Choice** (9 parts)

FILL IN THE APPROPRIATE SQUARES COMPLETELY to mark your answers. This problem will be graded automatically, so you may lose credit if you mark your answers differently!

Proof is not required for this problem; points will be awarded solely based on your final answer. Some parts require you to select exactly one answer; others require you to select all correct answers.

(a) [4 points] Let $f, T : \mathbb{N} \to \mathbb{N}$. Suppose that $f \in O(n \log n)$, and that T satisfies the recurrence:

$$\begin{aligned} T(0) &= 1 \\ T(n) &= 3T(\lfloor n/3 \rfloor) + f(n) \end{aligned}$$

Which of the following must be true? Select all that apply.

- ☐ $T(n) \in O(n)$
- 🟩 $T(n) \in \Omega(n)$
- ☐ $T(n) \in O(n \log n)$
- ☐ $T(n) \in \Omega(n \log n)$
- 🟩 $T(n) \in O(n \log^2 n)$
- ☐ $T(n) \in \Omega(n \log^2 n)$

Rubric:

- 2pts ea: correct upper and lower bounds
- 1pt partial: $O(n \log n)$ upper bound
- 1pt partial: $\Omega(n \log^2 n)$ lower bound
- Auto zero: $O(n)$ upper bound or inconsistent answer

Common Mistakes:

- Inconsistent answer
- Assuming $f \in \Theta(n \log n)$
- Trying to use Master Theorem as stated

(b) [2 points] Suppose you have an algorithm MEAP which runs in amortized expected $\Theta(n)$ time. Which of the following is the runtime of n MEAPS? Select the one best answer.

- ☐ $\Theta(n^2)$ (worst case)
- ☐ $\Theta(n^2)$ (amortized)
- 🟩 $\Theta(n^2)$ (expected)

(c) [2 points] Eevee creates a HASH TABLE that uses AVL TREES for chaining instead of LINKED LISTS. What is the new expected runtime (under SUHA) for the INSERT, FIND, and DELETE operations? Select the one best answer.

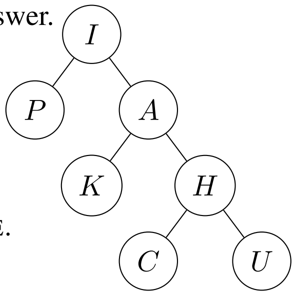
- 🟩 $\Theta(1)$
- ☐ $\Theta(\log n)$
- ☐ $\Theta(n)$

(d) [2 points] What is the new worst-case runtime (under SUHA) for the same three operations in Eevee's HASH TABLE? Select the one best answer.

- ☐ $\Theta(1)$
- 🟩 $\Theta(\log n)$
- ☐ $\Theta(n)$

(e) [3 points] Which of the following rotations or sequences of rotations transforms the following TREE T into a TREE that satisfies AVL Property? Select the one best answer.

- ☐ RIGHT-ROTATE(A)
- 🟩 LEFT-ROTATE(I)
- ☐ RIGHT-ROTATE(A), then LEFT-ROTATE(I)
- ☐ T already satisfies the AVL Property.
- ☐ No sequence of rotations can transform T into an AVL TREE.



(f) [2 points] How many rotations are required to transform an AVL TREE with n nodes into a BINARY SEARCH TREE? Select the one best answer.

- ☐ 0
- ☐ $o(\log n)$
- ☐ $\Theta(\log n)$
- ☐ $\omega(\log n)$
- 🟩 It may not be possible.

(g) [4 points] Which of the following are efficiently maintainable augmentations for a BINARY TREE node T (according to the definition presented in class)? Select all that apply.

- 🟩 The maximum of all items in T's subtree
- ☐ T's successor in traversal order
- ☐ T's depth
- 🟩 42

Rubric:

- -2pts per error (min 0)

(h) [2 points] This is an asymptotically optimal algorithm to BUILD a HEAP H from a LINKED LIST L: For x in L: INSERT x into H.

- ☐ TRUE
- 🟩 FALSE

(i) [4 points] Which of the following ARRAYS are valid MAX-HEAPS? Select all that apply.

- ☐ $[1, 2, 3, 4, 5, 6, 7]$
- ☐ $[4, 6, 3, 7, 2, 5, 1]$
- 🟩 $[7, 3, 6, 1, 2, 4, 5]$
- 🟩 $[7, 6, 5, 1, 2, 3, 4]$

Rubric:

- -2pts per error (min 0)

Problem 2. [25 points] **Sleepy Sliggoo** (2 parts)
Sleepy Sliggoo will play Pokémon Sleep with n friends tomorrow and wants to know when to plan on being asleep. Each friend i has sent a closed interval $I_i = [a_i, b_i]$ of time during which they will be playing, where $0 \leq a_i < b_i \leq 24$ are expressed as floating-point numbers of hours after midnight. Help Sleepy Sliggoo compute the union of all of the I_i, expressed as a union of *disjoint* closed intervals. You may assume that each a_i and b_i can be represented in a single machine word.

For example, if Sleepy Sliggoo's four friends will be playing during the intervals $[1, 3]$, $[6, 12]$, $[5.5, 8]$, $[3, 4]$, then the output of your algorithm should be $[1, 4]$, $[5.5, 12]$.

(a) [15 points] Suppose that every friend will start playing on the hour, i.e. every a_i is an integer. Give an $O(n)$ time algorithm to solve the problem. Prove your algorithm correct and analyze its runtime. You must qualify this runtime as worst-case, amortized, and/or expected. Partial credit may be given for less efficient solutions.

Solution: DAA-SORT

- DAA-SORT the intervals by a_i.
- If two intervals collide when sorting, discard the smaller one.
- While any two intervals in the resulting set S overlap, replace them with a single merged interval.
- Return S.

Correctness: The union of all intervals is preserved by all discards, so S has the same union as the input. By the loop guard of the last step, returned intervals are disjoint.

Runtime: There are 24 possible values for a_i, so DAA-SORT takes worst-case $\Theta(n)$ time. This results in at most 24 intervals, so the last step takes constant time. Total time is worst-case $\Theta(n)$.

Solution: COUNTING-SORT

- COUNTING-SORT the input intervals by a_i and put them in a DOUBLY-LINKED LIST S.
- Iterate over S, merging each interval with the previous interval if they overlap.
- Return S.

Correctness: We maintain the invariants:

- Intervals before iteration index are sorted by start time
- Intervals before iteration index are disjoint
- $\bigcup S = \bigcup I_i$

First follows from fact that S is sorted by start time. Second follows from first and same fact. Third is by construction. Upon termination, they imply that S is the correct output.

Runtime: There are 24 possible values for a_i, so COUNTING-SORT takes worst-case $\Theta(n)$ time. Loop takes constant time per iteration, so $\Theta(n)$ time total. Total time is worst-case $\Theta(n)$.

Solution: Recursion - solves (b) without modification

- If $n \leq 1$, return input.
- Otherwise, divide input in half and recurse on both halves; let S be the resulting set of intervals.
- While any two intervals in S overlap, replace them with a single merged interval.
- Return S.

Correctness: by (strong) induction

Runtime: As above, S contains $\Theta(1)$ intervals, so merge step takes constant time. By Master Theorem, runtime is worst-case $\Theta(n)$.

Solution: Iteration - solves (b) without modification

- Initialize $S = \emptyset$.
- For each i:
  - Add I_i to S
  - While any two intervals in S overlap, replace them with a single merged interval.
- Return S

Correctness: We maintain the invariants:

- S is a set of disjoint intervals
- $\bigcup S = \bigcup_{j \leq i} I_i$

First is maintained by inner loop. Second is by construction. Upon termination, they imply that S is the correct output.

Runtime: Inner loop takes constant time by first invariant, the fact that every interval contains one of 24 integers, and the Pidgeyhole Principle. Outer loop executes $\Theta(n)$ times, for total worst-case $\Theta(n)$ runtime.

Common Mistakes:

- Assuming b_i are integers
- Neglecting lower bound on runtime
- When merging intervals, always using newer interval instead of larger interval
- Trying to counting/radix sort b_i
- Inappropriate use of Hash Tables

(b) [10 points] Suppose instead that every friend will play for more than one hour, i.e. $b_i - a_i > 1$. (You may no longer assume that a_i is an integer.) Give an $O(n)$ time algorithm to solve the problem. You need not analyze runtime or prove correctness. You may assume a correct solution to part (a), even if you did not solve it yourself.

Solution: Reduction to (a)

- For every i, there is an integer c_i such that $a_i < c_i < b_i$.
- Compute $\bigcup [24 - c_i, 24 - a_i]$ using part (a).
- Replace each interval $[x, y]$ in the union with $[24 - y, 24 - x]$.
- Compute $\bigcup [c_i, b_i]$ using part (a).
- While any two remaining intervals overlap, merge them.
- Return the resulting set of intervals.

Solution: Adaptation of (a)

Use the Sorting algorithm for (a) except:

- Use sort key $\lceil a_i \rceil$ instead of a_i.
- when an interval collides with an interval already stored in the DAA, replace them with a single merged interval instead of discarding the smaller one.

Solution: Iterative and recursive solutions for (a) work as described.

Common Mistakes:

- Asserting that all solutions to (a) also solve (b)
- In iterative solution: replacing inner loop with a single merge
- Inappropriate use of radix sort
- Not knowing what a floating point number is
  - A floating point number has the form $a \times 2^b$, where $a, b \in \mathbb{Z}$
  - Its length is $\Theta(\log(1 + |ab|))$
  - "Represented in a single machine word" means ab is polynomial in n, not that b is constant

Problem 3. [15 points] **#4Sum** (1 part)
The #4Sum problem is defined as follows:
Given an Array A of n integers, count how many 4-tuples (w, x, y, z) of indices (not necessarily distinct) satisfy $A[w] + A[x] + A[y] + A[z] = 0$.

For example, if $A = [1, 2, -3]$, then $\#4\text{Sum}(A) = 4$; the four solutions are $(0, 0, 0, 2)$, $(0, 0, 2, 0)$, $(0, 2, 0, 0)$, and $(2, 0, 0, 0)$. If instead $A = [1, -2, -4]$, then $\#4\text{Sum}(A) = 0$.

Give an $O(n^2)$-time algorithm to solve #4Sum. Analyze its runtime. You must qualify this runtime as worst-case, amortized, and/or expected. You need not prove correctness. Partial credit may be given for less efficient solutions.

Solution:

- Create a Hash Set H of pairs (k, v), keyed by k.
  (For ease of exposition, we think of H as dynamically representing a function, where $H(k) = v$ if $(k, v) \in H$, and $H(k) = 0$ if no such v exists.)
- For each pair of indices (w, x): increment $H(A[w] + A[x])$.
- Return the sum over all keys k in H of $H(k) \times H(-k)$.

Runtime: With a constant load factor, it takes amortized expected $\Theta(1)$ time to Insert, and expected $\Theta(1)$ time to Find. The first loop performs each $\Theta(n^2)$ times, for a total of expected $\Theta(n^2)$ time. It takes $\Theta(n^2)$ time to iterate over the $\Theta(n^2)$ summands in the last step, and each requires an additional Find, for a total of expected $\Theta(n^2)$ time. Total runtime for the algorithm is expected $\Theta(n^2)$.

Correctness (not needed): $H(k)$ counts the number of ordered pairs (w, x) such that $A[w] + A[x] = k$. We want to compute $|S|$, where $S = \{(w, x, y, z) : A[w] + A[x] + A[y] + A[z] = 0\}$.

$$\begin{aligned}
|S| &= \left|\bigcup_{k\in\mathbb{Z}} \{(w, x, y, z) : A[w] + A[x] = k, A[y] + A[z] = -k\}\right| \\
&= \left|\bigcup_{k\in\mathbb{Z}} \{(w, x) : A[w] + A[x] = k\} \times \{(y, z) : A[y] + A[z] = -k\}\right| \\
&= \sum_{k\in\mathbb{Z}} |\{(w, x) : A[w] + A[x] = k\}| \times |\{(y, z) : A[y] + A[z] = -k\}| \\
&= \sum_{k\in\mathbb{Z}} H(k) \times H(-k) \\
&= \sum_{H(k)\neq 0} H(k) \times H(-k)
\end{aligned}$$

Common Mistakes:

- Using a Hash Table as a multi-set

Problem 4. [12 points] **Heaps** (1 part)
Prove or disprove: For every ARRAY $A = [a_1, a_2, \ldots, a_n]$, if A satisfies the MIN-HEAP Property and the reverse of A (i.e. $[a_n, a_{n-1}, \ldots, a_1]$) satisfies the MAX-HEAP Property, then A is sorted.

Solution: The claim is false; take as a counterexample $A = [1, 3, 2, 4]$. MAX-HEAP property requires $A[4] \geq A[3] \geq A[1]$ and $A[4] \geq A[2]$, which are both satisfied. MIN-HEAP property requires $A[1] \leq A[2] \leq A[4]$ and $A[1] \leq A[3]$, which are also both satisfied. However, A is not sorted; $A[2] = 3 > 2 = A[3]$.

Common Mistakes:

- Switching Min-Heap and Max-Heap Properties
- (Attempting to prove statement)

Problem 5. [23 points] **Augmentations** (3 parts)
Falkner's n birds live in an AVL TREE SET keyed by a unique ID number. Falkner's TREE currently supports the operations `insert(bird)`, `find(ID)`, and `delete(ID)`, implemented as in class, but he would like to give it additional capabilities. Describe how to change Falkner's TREE to implement the following operations. You may not change the asymptotic runtimes of any of the SET operations given above. You need not prove correctness or analyze runtime. However, partial credit may be awarded for justification of partially correct answers.

(a) [5 points] `recent()`: In $O(1)$ time, find and output the bird whom Falkner accessed (via `insert` or `find`) most recently.

Solution: (Augmentation)
Keep a counter `timer`. When inserting, `timestamp` the inserted bird with `timer` and increment `timer`. When finding, overwrite the found bird's `timestamp` with `timer` and increment `timer`. Augment every node with the bird of maximum `timestamp` value in its subtree. To implement `recent`, output the augmentation value at the root.

Note: augmentation can be computed by comparing root item's timestamp to augmentations of children.

Solution: (Cross-linking)
Keep a cross-linked DOUBLY-LINKED LIST with the same birds sorted by access time, i.e. every tree node is also a list node, with all attributes needed for both data structures. All SET operations are modified to have the node in question `delete` itself from the LIST, and then (if appropriate) `insert` itself at the head of the LIST. Now `recent()` outputs the head of the list.

Common Mistakes:

- Failing to handle deletion
- Gratuitous use of Hash Tables
- Attempting the cross-linking solution without cross-linking

(b) [8 points] `recent-up-to(max)`: In $O(\log n)$ time, find and output the bird whom Falkner accessed most recently from among all birds whose IDs are $\leq$ `max`. You may assume that Falkner's TREE and all of its subtrees have a correct implementation of `recent()`.

Solution: (Recursive)

- If `T` is empty, output $\perp$.
- If `T.item` has an ID $>$ `max`, output `recent-up-to(T.left, max)`.
- Otherwise, the desired output is whichever one of the three birds `recent(T.left)`, `T.item`, or `recent-up-to(T.right, max)` (exists and) has the maximum `timestamp`.

Solution: (Iterative)

- FIND the bird b in the tree whose ID is largest among those $\leq$ max.
- Compare the timestamps of b, all of its ancestors a with smaller IDs, and `recent(`a`.left)` for each of these ancestors.
- Output the one with the most recent timestamp.

Solution: (Dual)
Keep a second AVL Tree Set with the same birds, but keyed by `timestamp` and augmented with the minimum ID `minID` in each subtree. Define `recent-up-to(T, max)` recursively on this second tree:

- If `T` is empty or `minID(T) > max`: output $\perp$.
- If `recent-up-to(T.right, max)` is not $\perp$, output it.
- If `T.item` has an ID $\leq$ `max`, output `T.item`.
- Output `recent-up-to(T.left, max).`

Common Mistakes:

- Everything from 5(a)
- Traversing the tree
- Assuming (implicitly or explicitly) that ID `max` is always at the root
- Thinking that `recent-up-to(max)` is always either `recent()` or $\perp$
- Using recursion instead of `recent()`
- Using `recent()` instead of root ID
- Comparing against root timestamp in both recursive cases or neither
- Destroying the tree

(c) [10 points] `find-closest()`: In $O(1)$ time, find and output the pair of birds whose IDs have the smallest (positive) absolute difference.

Solution: (Augmentation)
Augment every node with three values: the bird with `min` ID, the bird with `max` ID, and the pair `closest` of birds with most similar IDs in the subtree. To implement `find-closest`, output the `closest` augmentation value at the root.

Note: `min` and `max` augmentations can be computed as in part (a), and `closest(T)` is one of the four pairs `closest(T.left)`, `closest(T.right)`, `(max(T.left), T.item)`, or `(T.item, min(T.right))`.

Solution: (Auxiliary tree)
Keep a second AVL SET S of pairs of birds (adjacent in traversal order of Falkner's tree), keyed lexicographically by the function $(x, y) \mapsto (y.\text{ID} - x.\text{ID}, x.\text{ID})$. Insertion into Falkner's tree must be modified to find the inserted bird b's predecessor a and successor c, delete (a, c) from S if applicable, and insert (a, b) and (b, c) into S if applicable. Similarly, deletion must be modified to find the deleted bird b's predecessor a and successor c, delete (a, b) and (b, c) from S if applicable, and insert (a, c) into S if applicable. Finally, we augment S with the minimum element of every subtree, and `find-closest` outputs this augmentation at the root.

Solution: (Cross-linking)
Keep a cross-linked DOUBLY-LINKED LIST with the in-order traversal of Falkner's TREE. SET `deletion` is modified as in (a), and SET `insertion` leverages the facts that only leaves are inserted and that a leaf's parent is either its predecessor or successor in traversal order. Every node can now be augmented with the `closest` augmentation defined above, using the fact that `max(T.left)` and `min(T.right)` are the predecessor and successor of `T` in traversal order. Note that while this is not technically a *tree* augmentation anymore, it can be maintained in constant time using *list* operations and so functions identically to an augmentation in the cross-linked data structure. To implement `find-closest`, output the `closest` augmentation at the root.

Common Mistakes:

- Mistaking left and right children for in-order predecessor and successor
- Entirely forgetting the case where the root node is one of the closest pair
- Using only the distance from the root node to its closest neighbor as an augmentation
- Not handling duplicates in the auxiliary tree solution
- Attempting the cross-linking solution - not an error per sé, but very difficult to do correctly

Recurring Misunderstandings

1. **Recursion**
   If something has already been defined, do not attempt to redefine it. At best, you gain nothing. You will, however, lose points based on clarity, concision, ambiguity, or for any mistakes you make. This is especially important when defining recursive algorithms. Many attempts at

```
def f(x):
        Base case: return g(x)
        If b: return f(y)
        Else: return f(z)
```

   were instead phrased as

```
def f(x):
        If b: "Consider" y.
                (vague, underspecified attempt at repetition)
                return g(y)
        Else: "Consider" z.
                (vague, underspecified attempt at repetition)
                return g(z)
```

   You have `f`, its signature, and its intended behavior; use them. Just as with induction, you are allowed to assume that `f` works on smaller inputs when writing its definition.
   Many students similarly rewrote sorting algorithms, the two-finger algorithm, FIND / binary search, the augmentation paradigm, solutions to 2a when answering 2b, or solutions to 5a when answering 5b (this last is even *worse* than redundant, as 5b explicitly allowed a *stronger* assumption than a correct solution to 5a).

2. **Preconditions**
   Some of the algorithms from class (most notably sorting algorithms) have preconditions which must be satisfied in order for the algorithm to work correctly.

   - COUNTING-SORT: Sort keys must be natural numbers for the algorithm to work at all. They must be $O(n)$ for runtime to be linear.
   - DAA-SORT: In addition to the above, keys must be distinct; otherwise duplicate keys are erased.
   - TUPLE-SORT: Auxiliary sorting algorithms must be specified (otherwise a fully general comparison sort is assumed), and all preconditions of auxiliary sorting algorithms must be satisfied.
   - RADIX-SORT: Sort keys must be natural numbers for the algorithm to work at all. They must be $n^{O(1)}$ for runtime to be linear.

3. **Keys and SETS**
   If S is a SET of X keyed by k, this means that:

   - S mathematically represents a set of items of type X, i.e. all $x \in S$ are distinct.
   - The key k is an intrinsic property of items of type X.
   - The keys of all $x \in S$ are also distinct.
   - If the key k is not specified, it is assumed to be the identity function by default, i.e. the key of x is x itself.

"S is a SET that maps X to Y" is abuse of notation for "S is a SET of (X, Y) pairs, keyed by X", just as mathematically, a (partial) function S is a set of ordered pairs. If S represents a total function $x \mapsto x.y$, then S does nothing *by definition* except waste time and space. By contrast, a SET that represents the function $x.y \mapsto x$ may be useful.

4. **Multi-sets**
Some SET implementations can be adapted naturally to function as multi-sets, where duplicate keys are allowed, and where `find` may output *any* item with the given key. This does *not* work with HASH TABLES, which fundamentally rely on distinct keys in a way that other SET implementations we have seen do not. For e.g. the given solution to #4SUM, if A contains many 0s, it forces H to perform many repeated FINDS. If instead, H were a multi-set that stored each pair (w, x) separately instead of counting them, these repeated FINDS would instead be forced *collisions*, and the expected runtime analysis of the HASH TABLE no longer works.

5. **Binary tree properties**
    - Binary Search Property: in-order traversal yields the keys in sorted order
    - AVL Property: every node's children have heights that differ by at most 1
    - Max-(resp. Min-)Heap Property: every node's key is no smaller (larger) than those of its children
    - An AVL SET satisfies the Binary Search Property and the AVL Property, but (generally) not the Heap Properties.
    - An AVL SEQUENCE satisfies the AVL Property but has no keys, so it doesn't make sense to talk about the Binary Search Property or the Heap Properties.
    - A HEAP satisfies one of the Heap Properties and the AVL Property (because it is compact), but (generally) not the Binary Search Property.
    - A binary tree could in principle satisfy none of the above properties, e.g. other efficient SEQUENCE implementations that use weaker balancing properties.
    - A binary tree could in principle satisfy one of the Heap Properties but not the AVL Property. However, the point of the Heap Properties is to enable the use of compact binary trees and arrays.
    - A binary tree is degenerate in some sense if it satisfies both Heap Properties or one Heap Property and the Binary Search Property.