By: **Sethu Senthil**

HW 3

1) a) Constructed Heap: [0, 5, 1, 7, 6, 4, 3, 10, 8, 12]

0: Heap: [1, 5, 3, 7, 6, 4, 12, 10, 8]
0, 1: Heap: [3, 5, 4, 7, 6, 8, 12, 10]
[0, 1, 3]: Heap: [4, 5, 8, 7, 6, 10, 12]

b) In a well-formed heap within a specific tree, every level is fully occupied, except for the last level, which may not contain all the nodes. In other words, the heap structure ensures that each parent node has at least two children. Knowing this, we can deduce that the last half of the elements in the array are all leaf nodes. Given a node of index i, the left child of the node can be found at 2i + 1 and the right child can be found at 2i + 2.

If we look at the elements from the list after n / 2 (AKA the second/last half of the list) they all must be leaves in order for it to fulfill the heap structure, otherwise, it won't be a valid heap.

c) We normally decrease i to from (n / 2) - 1 to 0 because in some cases doing the inverse, increasing i to (n / 2) - 1 from 0, does not produce a valid heap. When we decrease i in this fashion, we are basically starting with the lowest level (a node with no children AKA non-leaf node), and working our way up. This is the only proper way to construct a heap as it ensures that lower-level nodes have already been formatted into valid heap forms.

Instead, if we increase i, we are going from top to bottom (the opposite of the previous approach). This does not always work as future nodes may violate the heap validity of their children.

Here is an example of any array: [5,3,8,1,4,6].

In this example, if we try to use the increase method, we would start with 5 and attempt to heapify it downward. However, this may lead to a situation where the heap property is not maintained in every step.

If we attempt to build a min heap by increasing i, we could start with the first element, 5, and try to heapify it downward. However, this may lead to a

situation where the heap property is not maintained at every step, resulting in an invalid min heap. The correct approach is to start from the last non-leaf node (⌊n / 2⌋-1) and move towards the root, ensuring that each subtree is a valid heap along the way. Like , when we swap 5 and 3 to try to satisfy the min-heap property it does not produce a proper min heap.

We get the following after the first step (through the incorrect approach): [3,5,8,1,4,6]

However, this is not valid. The sub-list below 8 is not a valid min heap as the parent node is larger than the child node.

2)

- We can arrange an array of integers in O(n + M) time by using the Counting Sort algorithm. Count Sort has a time complexity of O(n + k) time, where k = the range of numbers. For this example, M represents the range of numbers. In other words k = M. Additionally, M is defined to be linear, maintaining the linear time complexity of the statement we need to prove. Through substitution (k = M), we get: O(n + M) which is the statement we need to prove.

- The lower bound AKA $\Omega(n\ log\ n)$ does not apply here because M would be too small as M is calculating the difference between the maximum and minimum values.

3)

a) Binary

i) Recurrence Relation: $T(n) = T(\frac{n}{2}) + \theta(1)$

- $T(\frac{n}{2}) \rightarrow$ Represents how the list is being cut in half each n number of comparison
- $\theta(1) \rightarrow$ Represents the constant time computations done in every recursion step

ii) Solving the Recurrence (Using Master Therom)

- $T(n) = aT\frac{n}{b} + f(n) \rightarrow$ Masters Theorm
- We can easily identify the pattern from the general formula and the given recurrence
  - a = 1 → given from the found recurrence, subproblem per recursion
  - b = 2 → given from the found recurrence, represents the size of each subproblem
  - f(n) = $\theta(1) \rightarrow$ the constant time
- There are three cases defined in the Masters Therom
  - c = 0 and $\log_b\{a\} = \log_2\{1\} = 0$
    - Transitive Property: c = $\log_b\{a\}$, hence we fall in case 2
  - **We fall into case #2** : If $f(n) = \theta(n^{logb\{a\}} log^k n)$ for some $k \geq 0$, $then$, $T(n) = \theta(n^{\log_b\{a\}} log^{k+1} n)$
  - Solution: $T(n) = \theta(log\, n)$

b) Ternary

i) Recurrence Relation: $T(n) = T(\frac{n}{3}) + \theta(1)$

- a = 1 → # of subproblems
- b = 3 → size of each subproblem
- f(n) = $\theta(1) \rightarrow$ the constant time

ii) c = 0 and $\log_b\{a\} = \log_3\{1\} = 0$

- Transitive Property: c = $\log_b\{a\}$, hence we fall in case 2 (again)
- Solution: $T(n) = \theta(log\, n)$

4) Python Code

```
def merge_and_count(arr, l, m, r): # TOTAL: O(n log n)
```

```
    # Left subarray
    left = arr[l:m + 1]  # O(m - l + 1)
    # Right subarray
    right = arr[m + 1:r + 1]  # O(r - m)

    i = j = 0
    k = l
    swaps = 0

    # Merge the left and right subarrays while counting inversions
    while i < len(left) and j < len(right):  # O(m - l + 1) + O(r - m) iterations in
the worst case
        if left[i] <= right[j]:  # O(1)
            arr[k] = left[i]  # O(1)
            i += 1  # O(1)
        else:
            arr[k] = right[j]  # O(1)
            j += 1  # O(1)
            # Count inversions when an element from the right subarray is chosen
            swaps += (m + 1) - (l + i)  # O(1)
        k += 1  # O(1)

    # Copy any remaining elements from the left subarray
    while i < len(left):  # At most O(m - l + 1) iterations
        arr[k] = left[i]  # O(1)
        i += 1  # O(1)
        k += 1  # O(1)

    # Copy any remaining elements from the right subarray
    while j < len(right):  # At most O(r - m) iterations
        arr[k] = right[j]  # O(1)
        j += 1  # O(1)
        k += 1  # O(1)

    return swaps  # O(1)


def merge_sort_and_count(arr, l, r): # TOTAL: O(n log^2 n)
    count = 0  # O(1)
    if l < r:  # O(1)
        m = (l + r) // 2  # O(1)
        # Recursively count inversions in the left subarray
```

```
        count += merge_sort_and_count(arr, l, m)  # T(n/2) time complexity (each
recursive call halves the array size)
        # Recursively count inversions in the right subarray
        count += merge_sort_and_count(arr, m + 1, r)  # T(n/2) time complexity (each
recursive call halves the array size)
        # Merge and count inversions in the current subarray
        count += merge_and_count(arr, l, m, r)  # O(n log n) time complexity (merging
step)
    return count  # O(1)
```

Using the Master Theorem (since this algorithm is also divide and conquer based): The recurrence relation is $T(n) = 2T(\frac{n}{2}) + O(n \log n)$, where the divide and conquer step splits the array in half, and the merging step has a time complexity of O(n log n). The Master Theorem is applicable here with

- a = 2,
- b = 2,
- f(n) = O(n log n).

$\log_b \{a\} = \log_2\{2\} = 1 \rightarrow$ First Case of Master Theorem

The recurrence relation falls into first case of the Master Theorem, yielding a time complexity of O(n log n). Therefore, the algorithm efficiently counts inversions in an array using a modified merge sort with a time complexity of O(n log n).

Question 5: (Python-esk Sudo Code) - Uses stack like data structure

```
Algorithm medianTracker(array):
```

```
maxHeap ← empty max heap (to store smaller elements)
minHeap ← empty min heap (to store larger elements)

# Procedure to add a number to the heaps and balance them
Procedure addNumber(n):
    if maxHeap is empty:
        # If maxHeap is empty, add the first element
        insert n into maxHeap  # O(log(1)) = O(1)
    else if size(maxHeap) == size(minHeap):
        # If both heaps have the same size, choose where to insert based on the next number
        if n < minHeap.peek():
            # If the number is smaller than the smallest in minHeap, insert into maxHeap
            insert n into maxHeap  # O(log(n))
        else:
            # If the number is larger, insert into minHeap and balance by moving the smallest to maxHeap
            insert n into minHeap  # O(log(n))
            insert -minHeap.pop() into maxHeap  # O(log(n))
    else if size(maxHeap) > size(minHeap):
        # If maxHeap has more elements, choose where to insert them based on the next number
        if n > -maxHeap.peek():
            # If the number is larger than the largest in maxHeap, insert into minHeap
            insert -n into minHeap  # O(log(n))
        else:
            # If the number is smaller, insert into maxHeap and balance by moving the largest to minHeap
            insert -n into maxHeap  # O(log(n))
            insert -maxHeap.pop() into minHeap  # O(log(n))

# Procedure to calculate and return the current median
Procedure getMedian():
    if maxHeap is empty:
        # If maxHeap is empty, return 0 as there is no median
        return 0  # O(1)
    else if size(maxHeap) == size(minHeap):
        # If both heaps have the same size, return the average of their tops as the median
        return (-maxHeap.peek() + minHeap.peek()) / 2.0  # O(1)
```

```
    else:
        # If maxHeap has more elements, return the top of maxHeap as the median
        return -maxHeap.peek()  # O(1)
```

This pseudocode outlines an algorithm for tracking the median of a array / list. The algorithm uses the technique of two heaps as discussed in office hours, a max heap to store smaller elements and a min heap for larger ones. The addNumber procedure ensures balanced insertion by tracking the current sizes of the heaps. The getMedian procedure, as named, returns the median, accounting for scenarios where the heaps have equal or different sizes (as depicted in the hint of the question). The time complexity for adding a number is logarithmic, mainly determined by heap operations, while obtaining the median is achieved in constant time (O(1)).

6)

- I worked in a group: Eric He
- I did not consult with anyone other than my group members (besides TAs in office hours)

- Resources Used:
    - https://www.digitalocean.com/community/tutorials/min-heap-binary-tree
    - https://www.geeksforgeeks.org/heap-sort/
    - https://www.youtube.com/watch?v=VmogG01IjYc
    - https://www.youtube.com/watch?v=4VqmGXwpLqc&t=3s
    - https://www.geeksforgeeks.org/stack-in-python/
    - https://www.geeksforgeeks.org/difference-between-min-heap-and-max-heap/
    - https://www.geeksforgeeks.org/python-program-for-count-inversions-in-an-array-set-1-using-merge-sort/
    - https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture3.pdf
    - https://www.programiz.com/dsa/master-theorem
    - https://www.geeksforgeeks.org/counting-sort/
    - https://www.geeksforgeeks.org/inversion-count-in-array-using-merge-sort/
    -