<u>Problem Set 7</u>

Daniel Wang (S01435533)

1. When building a heap using binary tree, we need to insert nodes n times. Each time we need to heapify the tree after insertion, and heapify requires $O(\lg n)$ for a length of n array. Thus, the complexity is:
   $O(\lg 1) + O(\lg 2) + \cdots + O(\lg n) = O(\lg n!)$

   Now we want to show $\lg n!$ is upper bounded by $n \lg n$. We know $n! \geq n^n$ as $n! = 1 \cdot 2 \cdot \ldots \cdot n \leq n \cdot n \cdot \ldots n = n^n$. Also, we know logarithm is a monotonic function. Thus:
   $O(\lg n!) \leq O(\lg n^n) = O(n \lg n)$

   From above, we know that building the heap has $O(n \lg n)$ upper bound.

2. Building the heap from scratch, we have these intermediate steps (the right arrow symbol represents a step of swimming up when some child values are greater than the parent one):
   (1) Insert 4: [4]
   (2) Insert 2: [4 2]
   (3) Insert 7: [4 2 7] $\rightarrow$ [7 2 4]
   (4) Insert 9: [7 2 4 9] $\rightarrow$ [7 9 4 2] $\rightarrow$ [9 7 4 2]
   (5) Insert 12: [9 7 4 2 12] $\rightarrow$ [9 12 4 2 7] $\rightarrow$ [12 9 4 2 7]
   (6) Insert 1: [12 9 4 2 7 1]
   (7) Insert 3: [12 9 4 2 7 1 3]
   (8) Insert 0: [12 9 4 2 7 1 3 0]
   (9) Insert 10: [12 9 4 2 7 1 3 0 10] $\rightarrow$ [12 9 4 10 7 1 3 0 2]
   $\rightarrow$ [12 10 4 9 7 1 3 0 2]
   (10) Insert 11: [12 10 4 9 7 1 3 0 2 11] $\rightarrow$ [12 10 4 9 11 1 3 0 2 7]
   $\rightarrow$ [12 11 4 9 10 1 3 0 2 7]

   After heapify the array, it becomes [12 11 4 9 10 1 3 0 2 7]

3. Here I represent red boldface characters are numbers I am now considering. After building the complete array, I heapify the array from bottom to top:
   4 2 7 9 12 1 3 0 10 11 (original array)
   4 2 7 9 <u>**12**</u> 1 3 0 10 <u>**11**</u> (no swap is needed)
   4 2 7 <u>**10**</u> 12 1 3 <u>**0**</u> <u>**9**</u> 11 (9 is swapped with 10)
   4 2 <u>**7**</u> 10 12 <u>**1**</u> <u>**3**</u> 0 9 11 (no swap is needed)

4 **12** 7 **10** **2** 1 3 0 9 11 (2 is swapped with 12)

4 12 7 10 **11** 1 3 0 9 **2** (2 is swapped with 11)

12 **4** **7** 10 11 1 3 0 9 2 (4 is swapped with 12)

12 **11** 7 **10** **4** 1 3 0 9 2 (4 is swapped with 11)

12 11 7 10 **4** 1 3 0 9 **2** (no swap is needed)

After heapify the array, it becomes [12 11 7 10 4 1 3 0 9 2]

4. Given the heapified array [12 11 4 9 10 1 3 0 2 7], the steps are:
   (1) Swap the root with last element and pop the last one:
       [12 11 4 9 10 1 3 0 2 7] → [7 11 4 9 10 1 3 0 2 12] → [7 11 4 9 10 1 3 0 2]
   (2) Heapify the array from top to bottom:
       [7 11 4 9 10 1 3 0 2] → [11 7 4 9 10 1 3 0 2]
       [11 7 4 9 10 1 3 0 2] → [11 10 4 9 7 1 3 0 2]

   After these steps, the array becomes [11 10 4 9 7 1 3 0 2]

5. The method I will use is to maintain a max heap with a size of N, and for each step I will pop one element from the max heap, the corresponded new heap will have a length of N-1, where the remaining part of the array will be sorted. The Python code is shown below:

```python
def heapify(arr, start, end):
    """ max-heapify the array within a range [start, end] """
    for i in range(end//2-1, start-1, -1):
        l, r = 2*i+1, 2*i+2
        max_id = i
        if arr[max_id] < arr[l]:
            max_id = l
        if r < end and arr[max_id] < arr[r]:
            max_id = r

        arr[i], arr[max_id] = arr[max_id], arr[i]
        if max_id != i:
            heapify(arr, max_id, end)

def heap_sort(arr):
    """ main loop """
    n = len(arr)

    for i in range(n-1, -1, -1):
        heapify(arr, 0, i+1)
        arr[i], arr[0] = arr[0], arr[i]
```

6. The runtime complexity is influenced by the loop `for i in range(n-1, -1, -1)` where

each loop execute heapify of length i subarray. The worst case happens when the heapify process requires sinking through the depth of the binary tree, which yields a complexity of $O(\lg i)$ where i is the length. Thus, the overall complexity is:

$$\sum_{i=1}^{n} O(\lg i) = O\left(\frac{n \cdot \lg n}{2}\right) = O(n \lg n)$$

7. The strategy I will do is to maintain a max-heap and a min-heap without overlap where elements in min-heap will be greater or equal to elements in max-heap. To make heaps be balanced, I will re-balance the heap so that the number of elements in max-heap will be greater or equal to the one in min-heap, and the difference between is at most 1. The Python code is shown below:

```
import operator

def heapify(arr, start, end, op=operator.lt):
    """
    heapify the array within a range [start, end]
    default (operator.lt) is max-heap
    change operator.gt to be min-heap
    """

    for i in range(end//2-1, start-1, -1):
        l, r = 2*i+1, 2*i+2
        target_id = i
        if op(arr[target_id], arr[l]):
            target_id = l
        if r < end and op(arr[target_id], arr[r]):
            target_id = r

        arr[i], arr[target_id] = arr[target_id], arr[i]
        if target_id != i:
            heapify(arr, target_id, end)


class LMedianHeap:
    def __init__(self):
        self.max_heap = [] # < min in min_heap
        self.min_heap = [] # > max in max_heap

    def insert(self, val):
        if not self.max_heap or val <= self.max_heap[0]:
            self.max_heap.append(val)
            heapify(self.max_heap, 0, len(self.max_heap), op=operator.lt)
        else:
            self.min_heap.append(val)
            heapify(self.min_heap, 0, len(self.min_heap), op=operator.gt)

        # balance len(max_heap) - len(min_heap) = 0 or 1
        n1, n2 = len(self.max_heap), len(self.min_heap)
        if n1 - n2 > 1:
```

```
            self.max_heap[0], self.max_heap[-1] = self.max_heap[-1], 
self.max_heap[0]
            val = self.max_heap.pop()
            heapify(self.max_heap)

            self.min_heap.append(val)
            heapify(self.min_heap)
        elif n1 - n2 < 0:
            self.min_heap[0], self.min_heap[-1] = self.min_heap[-1], 
self.min_heap[0]
            val = self.min_heap.pop()
            heapify(self.min_heap, 0, len(self.min_heap), op=operator.gt)

            self.max_heap.append(val)
            heapify(self.max_heap, 0, len(self.max_heap), op=operator.lt)

    def show_lmedian(self):
        print(self.max_heap[0] if self.max_heap else -1)
```

Here I analyze the time complexity of the two operations. For the insert function, I will heapify the min-heap and max-heap at most twice. Given that the complexity of heapify is $O(\lg n)$, the overall complexity of this operation is still $O(\lg n)$. For the show_lmedian function, it is $O(1)$ since I only get the top of the max-heap.