ISE315 – Analysis of Algorithms
Fall 2021
Homework-2 Report

Aral ŞEN
150170217

a) (20 points) Write down the corresponding heap operations for the **extract, decrease and insert** operations in the implementation. Then, give the asymptotic upper bounds for each operation and explain these bounds with your own words.

insert -> This function takes an input value and inserts it into its correct position in the heap and in order to maintain the heap property then rebuilds the heap with heapifhy() since the order is broken now.

```
def insert(self, element):
    if self.size >= self.maxsize:
        return
    self.size += 1
    self.storage[self.size] = element
    self.heapify()
```

Insert a new node to the end of the heap. Now to maintain the heap property, we traverse from the last node, and swapped if needed, to maintain the property of heap which could have been violated.

The time complexity here is O(log n) as we only need to traverse the height of the subtree.

So the cost will be: $\Theta(\log \quad n)$

extract -> This function is used when we need the top element of the heap.

```
def extract_min(self):
    popped = self.storage[self.ROOT]
    self.storage[self.ROOT] = self.storage[self.size]
    self.size -= 1
    self.min_heapify(self.ROOT)
    return popped
```

Since we can't just take the top element and leave the heap as it is, we need to rebuild our heap. What extract does is, it extract the top element and then replaces the top element with the last element in the list, then heapify's the heap. This way, we maintain our heap property when extracting. This is really similar to stack, if you pop but do not decrement the stack pointer, you will lose your stack. min_heapify costs O(logn) and since there is no loop, it will cost O(logn).

The cost will be: $\Theta(\log \quad n)$

decrease -> This function is called where there is an update to a node.

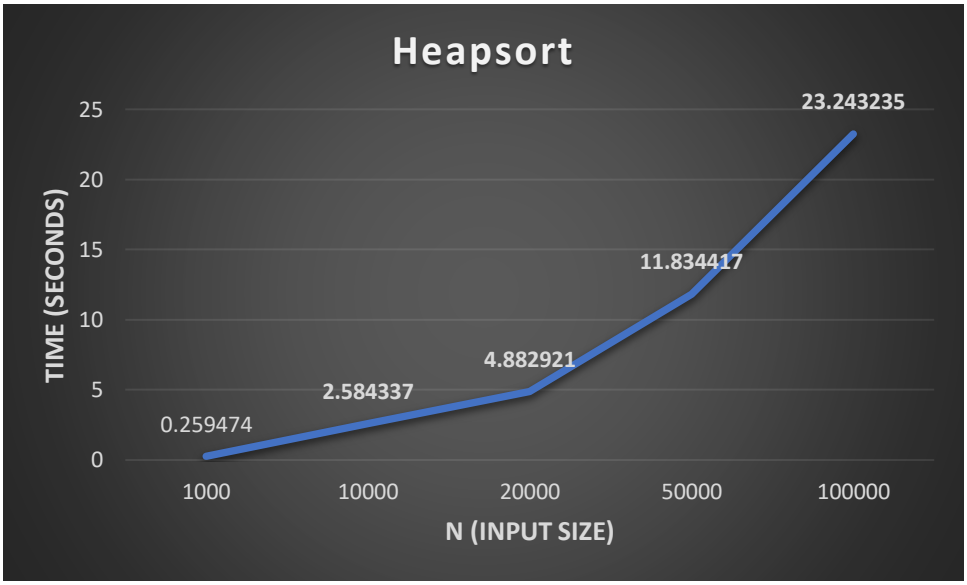
```
def decrease_key(self, i, new_val):
    self.storage[i].distance = new_val
    self.heapify()
```

Decrease function changes the value of the node and then looks if its parents are bigger than it, if they are, it is carried upwards to maintain min heap property. The worst case for this function is increasing the furthest leaf's value. Since binary tree's height is logn, this function will cost O(logn).

The cost will be: $\Theta(\log \quad n)$

b) (20 points) For different values of N (1000, 10K, 20K, 50K, 100K) calculate the average time of multiple executions (at least 5 times). Report the average execution times in a table and prepare an Excel plot which shows the N – runtime relation of the algorithm. Comment on the results in detail considering the asymptotic bounds that you gave in Question 1.

Input Size (N)	Elapsed Time
1000	**0.259474 seconds**
10K	**2.584337 seconds**
20K	**4.882921 seconds**
50K	**11.834417 seconds**
100K	**23.243235 seconds**



Heapsort has O(nlogn) time complexity for all the cases. Asymptotic upper-bound on running time of Heap Sort is O(nlogn). Like the Merge-sort, Heapsort also has divide process which generates binary tree, since height of tree is logn. However, Heapsort is much more closer to the linear-time because after generating binary tree, in terms of sorting in Heapsort, we have less complex operations. As we consider the operations that we found asymptotic bounds in Question 1, that means operations happen in a constant time and we keep doing them N times, which yields linearity. According to the table above, for different N values, how much time the Heapsort operations takes, as we see, whatever the operation numbers Heapsort does the operations in seconds. Even if the operation number is 100K, with Heapsort it takes approximately 20-25 seconds to do.

We could say the run time of the algorithm is $T(n)$ = $\Theta(n * \log \quad n)$