# IDAA432C Assignment-4 (Group: 41)

Aditya Vallabh IIT2016517
Kiran Velumuri ITM2016009
Neil Syiemlieh IIT2016125
Tauhid Alam BIM2015003

***Question***

**Given a set of numbers, write an efficient Heap Sort algorithm to report the required (ascending/descending) sorted sequence. Trace the movement of every element of the Heap during the execution of your algorithm.**

## I. Introduction

This problem requires us to sort a given array using heap sort, while keeping track of the locations every element obtains as it moves around the array. The sort is a basic sorting scheme. Tracking the movement of the elements simply involves storing every new location that the elements obtain after they are swapped. Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

## II. Algorithm Design

### A. Binary Heap

A Binary Heap is a Binary Tree with following properties.

1) Its a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible).
2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Similarly, in a Max Binary Heap, the key at the root must be the maximum among all the keys present in the heap.

### B. Array Traversal

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index $I$ (0-indexed)

1) The left child will be at $2 * I + 1$
2) The right child will be at $2 * I + 2$

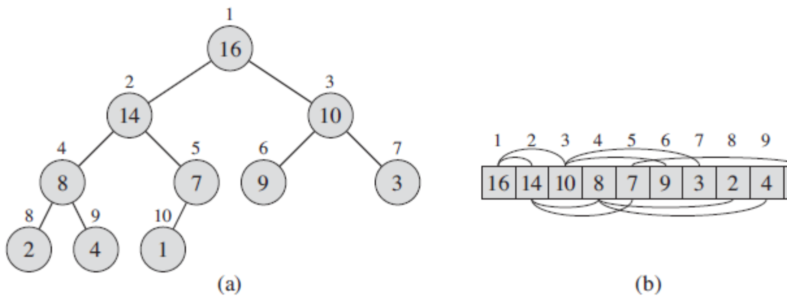

Fig. 1. Array representation for a heap (1-based indexing)

### C. Heapify

Heapify will be used to maintain the heap, we can give any index i and from index i it will go forward to its children and balance the heap and so on. The implementation of Heapify given below:

**Algorithm 1** Heapify Algorithm

```
Input: $arr, heads, n, idx$
$largest \leftarrow idx$
$left \leftarrow 2 * idx + 1$
$right \leftarrow 2 * idx + 2$
if $left < n$ and $arr[left] > arr[largest]$ then
    $largest \leftarrow left$
if $right < n$ and $arr[right] > arr[largest]$ then
    $largest \leftarrow right$
if $largest \neq idx$ then
    $swap(arr, heads, idx, largest)$
    $heapify(arr, heads, n, largest)$
```

The above method has the following steps:

1) It finds if there any bigger value is available in children nodes
2) Then it swaps the root with the child
3) Calls Heapify at the child to maintain the heap property recursively

We can use this algorithm to build the heap too. We call Heapify from the last but one level of the tree till the root. This will build the heap which is required for sorting.

### D. Sorting

Finally we are ready to sort the array, because our top element is the biggest in max-heap, we will start replacing these elements with bottom elements and will call Heapify in every iteration reducing the size of heap, to sort in increasing order, just like that:

**Algorithm 2** Heap Sort Algorithm

```
Input: $arr, heads, n, idx$
**for** $i \leftarrow \frac{n}{2} - 1$ *to* $0$ **do**
    $heapify(arr, heads, n, i)$
**for** $i \leftarrow n - 1$ *to* $0$ **do**
    $swap(arr, heads, 0, i)$
    $heapify(arr, heads, i, 0)$
```

### E. Tracking Movement

To keep track of the movement of the elements as they are swapped around in the array, we have associated a linked list with every array element. This linked list stores the indices that the corresponding element obtains as it moves through different locations of the array.

The head nodes of all the linked lists are stored in another array that runs parallel to the given input array.

After two elements are swapped, the corresponding head nodes in the second array are also swapped and a new node is appended to both their corresponding linked lists. This node contains the new index of that element and the iteration at which the swap happened.

**Algorithm 3** Swap Algorithm

```
Input: $arr, heads, it, a, b$
$heads[a] \leftarrow insertNode(heads[a], b)$
$heads[b] \leftarrow insertNode(heads[a], a)$
$tmp \leftarrow arr[a]$
$arr[a] \leftarrow arr[b]$
$arr[b] \leftarrow tmp$
$temp \leftarrow heads[a]$
$heads[a] \leftarrow heads[b]$
$heads[b] \leftarrow temp$
```

## III. Analysis and Discussion

### A. Time Complexity

*1) Heapify:* The method Heapify this will run from root to children and from the children to their children and so on, so the complexity of this method will be $O(\log n)$ or $O(h)$ where $h$ is height of node in tree.

*2) BuildHeap:* The heapsort algorithm first builds a heap from a given array by calling $heapify$ on the $(\frac{n}{2} - 1)$th element to the root (first) element of the array. Since $heapify$ is $O(\log n)$ and is called $\frac{n}{2}$ times, it may seem that $BuildHeap$ is of $O(n \log n)$ time. But we can derive a tighter upper-bound by observing that heapify takes $O(h)$ for each node where $h$ is the height. At a height $h$ there are$\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes. So the time-complexity would be:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \\ &= n \sum_{h=0}^{\log n} \left\lceil \frac{h}{2^{h+1}} \right\rceil = n \sum_{h=0}^{\infty} \left\lceil \frac{h}{2^{h+1}} \right\rceil \\ &= n * \frac{\frac{1}{2}}{(1 - \frac{1}{2})^2} = n * 2 \\ &= O(n) \end{aligned}$$

*3) HeapSort:* The loop in $heapSort$ is first building the heap and then just swapping elements and running from 1 to $size$ (total n-1 times which will be $O(n)$ ) with $Heapify$ method inside which complexity is $O(\log n)$ so the total complexity will be:

$$\begin{aligned} T(n) &= O(n) + O(\log n) * O(n) \\ &= O(n \log n) \end{aligned}$$

There is no early exit or latest exit in heap sort so the worst and the best case complexities would be equal. Thus $T(n) = \theta(n \log n)$

### B. Space Complexity

The algorithm as such doesn't require any additional space as it is an in-place sorting algorithm. However we are supposed to track the positions of each element while sorting. In order to store this additional information we require extra space.

One naive method would be to store all intermediate arrays but this would unnecessarily waste a lot of space. So we have implemented linked lists where a node gets pushed to the respective list only when the position of the corresponding element in the array changes.

Hence the space complexity is directly proportional to the number of swaps which is of the order $O(n \log n)$. However the best case would be when all the elements in the array equal leading to the least number of swaps giving $\Omega(n)$

## IV. Experimental Study

Following are the experimental findings after profiling the data and plotting the relevant graphs using *gnuplot*.

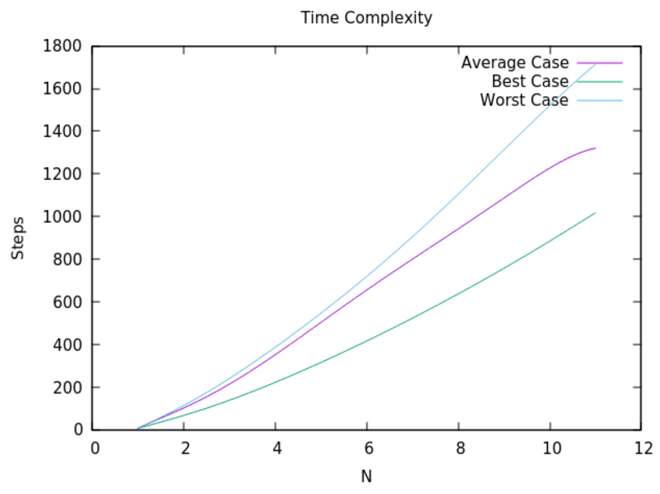

Fig. 2. Time Complexity for Heap Sort

As proved theoretically, it can be seen that the best and the worst case time-complexities are of the same order and differ only by a constant.

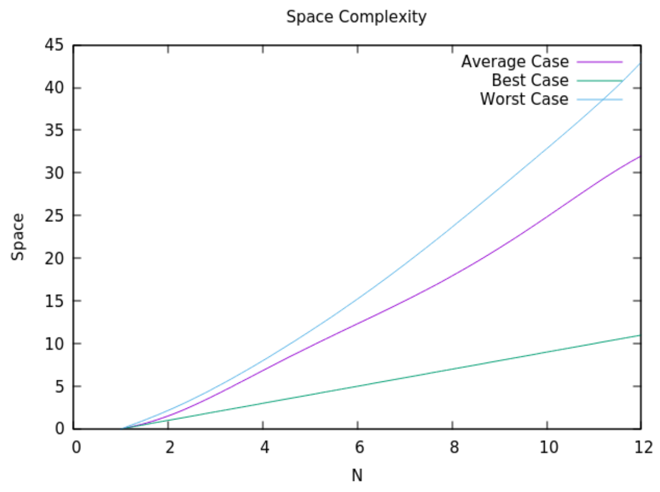

Fig. 3. Space Complexity for Tracking

It can be seen that the best case space-complexity is linear unlike the worst case complexity.

## V. Conclusion

Through this paper we proposed the algorithm to sort elements in an array using Heap Sort and provided a way to track the elements. Heap Sort in general has multiple applications like sorting a nearly sorted (or K sorted) array, finding k largest(or smallest) elements in an array, implementation of a Priority Queue, sorting numbers stored on different machines etc. It is the best sorting algorithm in terms of time and space complexity.

## References

[1] Introduction to Algorithms Book by Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen

[2] GeeksforGeeks: geeksforgeeks.org/heap-sort/

[3] DotNetLovers: dotnetlovers.com/Article/131/implementation-and-analysis-of-heap-sort

[4] GeeksforGeeks: geeksforgeeks.org/time-complexity-of-building-a-heap/