# IDAA432C Assignment-3 (Group: 41)


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


***Question***

**Write an efficient algorithm to achieve the following. Given a set of numbers, find out the smallest and the largest numbers in the list. Swap the smallest with the first element and largest with the last element. Find the second smallest and second largest and swap them with their proper positions. Continue in this way until all the elements are sorted. Track the movement of every element in the list while sorting. Do the necessary experimentation and analysis with your algorithm.**

## I. Introduction

This problem requires us to sort a given array in the manner described in the question, 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.

## II. Algorithm Design

### A. Sorting

The algorithm needed is similar to the selection sort algorithm, which selects a single element from an array and swaps it with an element at the beginning of the array.

This algorithm, however, selects two elements from an unsorted subarray — the smallest and the largest ones — and swaps the smallest with the first element and the largest with the last element. The same is repeated for every inner subarray. This effectively sorts the array in a manner similar to selection sort.

### B. 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 1** Sort Algorithm

```
Input: arr, heads, n
for i ← 0 to n/2 − 1 do
    mini = getMinIdx(arr, n, i)
    if mini ≠ i then
        swap(arr, heads, i + 1, mini, i)
    maxi = getMaxIdx(arr, n, i)
    if maxi ≠ n − i − 1 then
        swap(arr, heads, i + 1, maxi, n − i − 1)
```

**Algorithm 2** Swap Algorithm

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

**Algorithm 3** getMinIdx Algorithm

```
Input: arr, n, i
idx ← i
min ← arr[i]
for j ← i + 1 to n − i − 1 do
    if arr[j] < min then
        min ← arr[j]
        idx ← j
return idx
```

**Algorithm 4** getMaxIdx Algorithm

Input: $arr, n, i$
$idx \leftarrow i$
$max \leftarrow arr[n-i-1]$
**for** $j \leftarrow i$ *to* $n-i-2$ **do**
  **if** $arr[j] \geq max$ **then**
    $max \leftarrow arr[j]$
    $idx \leftarrow j$
*return* $idx$

## III. Analysis and Discussion

### A. Time Complexity

Finding the smallest and the largest element in an array of size $n$ takes $O(n)$ time. The implemented sorting algorithm does this twice for every subarray from the $i^{\text{th}}$ to the $(n-i)^{\text{th}}$ element, where each subarray has $k$ elements. Hence, the total number of computations for an array of size $n$ is given by the following relation:

$$\begin{aligned} T(n) &= 2(O(n) + O(n-2) + ... + O(2)) \\ &= 2\sum_{k=0}^{\frac{n}{2}-1} O(n-2k) \\ &= \frac{n^2+2n}{2} \end{aligned}$$

This clearly has the time complexity of $O(n^2)$. This is the case for the best, average and worst cases.

### 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.

**Best Case:** When all the elements are already in a sorted fashion, no elements need to be swapped and hence the space complexity would be - $\Omega(n)$

**Worst Case:** The maximum number of swaps that can occur is $n-1$. So then the complexity would be - $O(n+2*(n-1)) = O(3*n-2) = O(n)$.

**Average Case:** As in both cases we got the same complexity, the average case complexity would be $\theta(n)$

### C. Stability

The sorting algorithm has to traverse the entire array to find the smallest and largest elements, by the $getMinIdx$ and $getMaxIdx$ functions, respectively. This would render it either stable or unstable depending on the condition by which the elements are compared within these functions.

Using the conditions $(arr[j] < min)$ in $getMinIdx$ and $(arr[j] \geq max)$ in $getMaxIdx$ ensures that the first occurrence of the smallest element and the last occurrence of the largest element are selected. This would make the implemented sorting algorithm a stable one.

## IV. Experimental Study

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

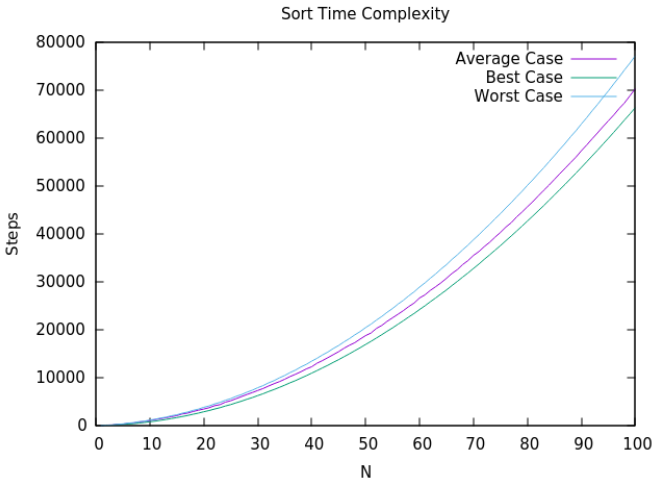


Fig. 1. Time Complexity for the Sorting Algorithm

It can be seen that the derived expression for time complexity agrees with the obtained graph which is a semi-parabola denoting the graph (Fig. 1) of $y = kx^2$ with varying values of $k$ for the best, average and worst cases.

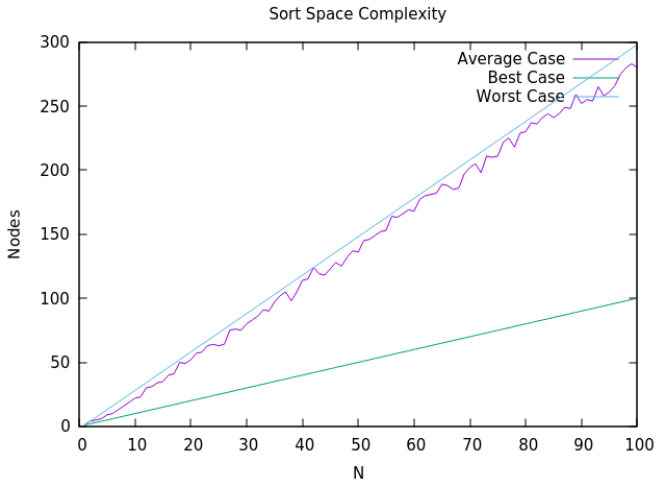


Fig. 2. Space Complexity for the Sorting Algorithm

The graph for space complexity is linear thus agreeing with the derived expression.

## V. Conclusion

In this paper we proposed an algorithm to sort an array of numbers in the given method. The time-complexity of our algorithm is $\theta(n^2)$. Moreover we have also tracked the positions of each element in the array by implementing linked lists with a space complexity of $\theta(n)$.