DAA ASSIGNMENT-3
GROUP-3
Modified Selection Sort

Sai Charan IIT2016039 | Mantek Singh IIT2016007 | Shruti Reddy IIT2016019 | Rakesh Lakra IIT2016052

***Abstract*—Here, we have to sort a given array using different form of selection sort algorithm and also track the positions of the elements.An $O(n^2)$, $\Omega(n^2)$ and $\theta(n^2)$ algorithm has been designed to solve this problem.**

***Index Terms*—Sorting, storing positions, time complexity, graphs**

I. Introduction and Literature Survey

Sorting is any process of arranging items in a sequence ordered by some criterion. Here we are sorting the alphabets in order of their ASCII values.

Here we have to traverse across the list of n unsorted numbers where n is the size of the list, finding the maximum and minimum element in remaining array and swapping it up with the first and last element of the minimum array and storing their positions each time they swapped. So we tracked the position of elements by maintaining another array of pointers to the linked lists which stores the positions of that particular element. While swapping the elements we also swap their corresponding pointers in the array so that data of that particular elements can be accessed easily. Additionally we are also tracking each position by storing which elements came to that particular position while sorting process is going on. so every time we swap an element four nodes are created two for each element and two for each position.

II. Algorithm Design

The algorithm has a simple $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$ approach.
1. The array a stores the initial unsorted list.
2. The ele array of struct element type as shown in pseudo code stores the changes made in position of that element and is used for tracking that element.
3. The posarr array of element type shows the element stored in that position after a certain number of iterations as shown in pseudo code.
4. In the function selectionsort, n/2 iterations are totally made to calculate the maximum and minimum element left in the remaining array, and that minimum element is swapped with the first while the maximum element is swapped with the last element of the remaining array.
5. The struct node datatype is used to store the position of the element after a certain number of iterations it.
The pseudo-code for the above explained algorithm is as follows:

Algorithm 1 Longest-sorted-partition

```
Struct node {
it,pos;
struct node *link;                ▷ the struct node datatype is used to store the position of the element after a certain number of iterations it.
};    ▷ The ele array of struct element type as shown in pseudo code stores the changes made in position of that element and is used for tracking that element.
struct element {
struct node *head, *tail;
};
procedure INSERTNODE(arr, index, value, iteration) ▷ array contains the list of numbers
    struct node *temp
    temp.it ← iteration
    temp.pos ← value
    arr[index].tail.link ← temp
    arr[index].tail ← temp
procedure SWAP(posarr, ele, a, x, y, iteration)
    z ← a[x]
    a[x] ← a[y]
    a[y] ← z
    struct element temp
    temp ← ele[x]
    ele[x] ← ele[y]
    ele[y] ← temp
    INSERTNODE(ele, x, x, iteration)
    INSERTNODE(ele, y, y, iteration)
    INSERTNODE(pos_arr, x, a[x], iteration)
    INSERTNODE(posarr, y, a[y], iteration)

procedure SELECTIONSORT(posarr, ele, a, n)
    for i = 0 to (n/2)-1 do
        min ← a[i]
        mindex ← i
        for j = i+1 to n-2 do
            if min > a[j] then
                min ← a[j]
                mindex ← j
```

34: **if**
35: $mindex \neq i$ **then**
36: SWAP($posarr, ele, a, i, mindex, i+1$)
37: $max \leftarrow a[n-i-1]$
38: $maxdex \leftarrow n-i-1$
39: **for** j = i+1 to n-2 **do**
40: **if** max $<$ a[j] **then**
41: $max \leftarrow a[j]$
42: $maxdex \leftarrow j$
43: **if**
44: $maxdex \neq n-i-1$ **then**
45: SWAP($posarr, ele, a, n-i-1, maxdex, i+1$)

III. Time Complexity Analysis

The time complexity for the above mentioned algorithm for an array of size n is proportional to $O(n^2)$ because for every time we are traversing through the remaining row to find the minimum or maximum element.

Worst Case :In worst case the minimum or maximum element gets updated more number times which will result in swapping the elements in the array. Here we are also maintaining the positions of every element at different iterations so even that pointer which points to the linked-lists of those elements should be swapped. In swapping we insert two nodes one for each element which contains its position and iteration number. we are also tracking every element that enters a particular position, so two more nodes one for each position should also be inserted.

concretely the expression obtained for time complexity is $2*n^2+122*n-63$ i.e $O(n^2)$which serves as the upper bound i.e the worst case time complexity.

Best Case :In the best case the given array is already in sorted order so the the minimum and maximum element will not get updated in the inner loop and so no swapping of elements and updating of positions will take place.

concretely the expression obtained for time complexity is $2*n^2+47*n+12$ i.e $\Omega(n^2)$ which serves as the lower bound i.e the best case time complexity.

The expression for time complexity in the average case is bounded below by the expression for time complexity obtained in the best case and bounded above by the expression for time complexity obtained in the worst case.

g(x) $<$ f(x) $<$ h(x) where g(x) is the tightest lower bound of the function and h(x) is the least upper bound of the function. Since the order of polynomials is n^2, the time complexity is $\theta(n^2)$. Note, we have used $\theta(n^2)$ because the time complexity in all the case is the same. This problem does not have worst case and best case time complexities of differing orders as both are of the the order of n^2.

IV. Space Complexity Analysis

For Selection Sort no extra space is required as all the elements gets swapped in the same array. Here as we tracking the path of each element and also each element in a particular position we require some more space.

As an element at max can go through the whole array and for a position at max every element may come once so if want to store them in arrays then we need an array of size n (n is size of input array) for each element and each position, so we require two matrices of size nxn but at max there are $2*n$ swaps and so 2^n positions are updated so $2*(n^2-2*n)$ space will be unnecessarily wasted if we use arrays. So here we have implemented it using linked lists to reduce the space complexity.

if we consider space taken by one element is 1 then each node in linked list occupies 3 spaces. Initially two array one element array and the other position array of linked lists are created, which are filled with initial positions and initial elements respectively so space occupied will be $2*n*3$

Worst Case :As mentioned above in worst case there will be $n-1$ swaps, For each swaps two nodes in element array and two nodes in position array are to be added so space occupied will be $2*(2*(n-1)*3)$ i.e in worst case total space occupied will be $18*n-12$ which implies that for $n>18$ space taken to implement in linked lists will be less than that of arrays in the worst case.

Best Case :In the best case the given array is already sorted so that no swapping will take place. Hence the space is required only to store initial elements and position i.e $2*n*3$ which is very much less than the space required when we implement it in arrays.

The expression for space complexity in the average case is bounded below by the expression for space complexity obtained in the best case and bounded above by the expression for space complexity obtained in the worst case.

V. Experimental study

We have found the time complexity of the algorithm by calculating it in each step. Now to will verify it with the experimental results.

We have plotted T VS N graph where is T is the total number of computations performed for a given N. The T and N values for calculating in rows graphs for them are plotted accordingly.

Table I
N vs T graph

n	t_{best}	$t_{average}$	t_{worst}
10	783	1034	1158
20	2863	3565	3798
30	6243	7183	7888
40	10923	12373	13428
50	16903	18549	20418
60	24183	26390	28858
70	32763	35453	38748
80	42643	45689	50088
90	53823	57392	62878
100	66303	70097	77118

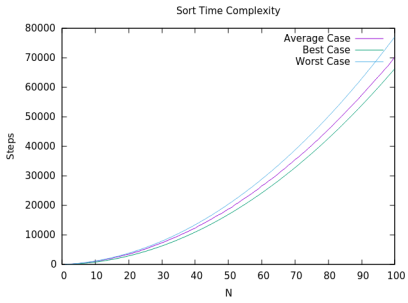

Figure 1. Rows T(y-axis) VS N(x-axis)

Now to took each node creation uses one unit of space then the values and graph for space complexity is as follows:

Table II
N VS SPACE GRAPH

n	$\mathbf{t_{best}}$	$\mathbf{t_{average}}$	$\mathbf{t_{worst}}$
10	10	22	28
20	20	52	58
30	30	80	88
40	40	114	118
50	50	136	148
60	60	168	178
70	70	202	208
80	80	230	238
90	90	252	268
100	100	280	298

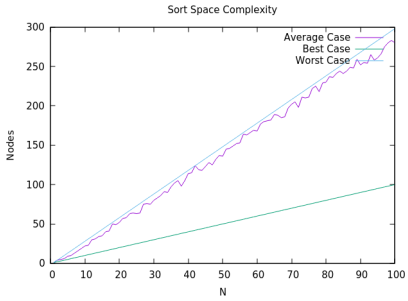

Figure 2. Diagonals T(y-axis) VS N(x-axis)

VI. Discussions

Thus from time complexity analysis and experimental study we can see the overall complexity of the algorithm is always proportional to n^2 i.e always $O(n^2)$.

So we can see that the curve is a semi parabola with axis of symmetry as Y-axis , but the co-efficient of the n^2 term varies in best and the worst cases.

The space occupied in any case is proportional to n. we can see that the graph of space complexity is proportional to n. Thus the experimental study proves that our time complexity and space complexity analysis is correct.

VII. Conclusion

we have tried to design an efficient algorithm that sorts a given list by finding the largest and smallest element along every iteration and swapping it up with the first and last indexed element of the remaining list. A simple approach of traversing across the array (n/2) times and decreasing size of remaining array by 2 with each iteration has been used, the details of implementation and pseudo code along with relevant calculations of time complexities and experimental details is done above to the best of our abilities. Hence, it can be concluded that the given question can be solved in $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$. Moreover, all the possible cases have been discussed for the formation of as efficient algorithm as possible.