Bubble Sort With Iteration Tracing

Anmol Singh Sethi IIT2016040
Anirudh Singh Rathore IIT2016091
Dheeraj Chouhan IIT2016078
Manish Ranjan IIT2016059
Vikas Kumar IIT2016058

***Abstract*—In this report we have devised an optimized bubble sort algorithm along with the tracing of the path of elements and contents of the array's indices.**

***Index Terms*—bubble-sort, sorting-algorithm, tracing-path, linked-lists**

I. Introduction and Literature Survey

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order.

The bubble sort is the oldest and simplest sorting method in use. Unfortunately, it's also the slowest. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list. The total number of comparisons, is (n - 1) + (n - 2)...(2) + (1) = n(n - 1)/2 or O(n^2)). The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant O(n) level of complexity. General-case is an abysmal O(n^2)

II. Algorithm Design And Explanation

We adopted a simple algorithm in order to solve the problem. We have implemented bubble sort optimized version and used various techniques to solve the tracing of elements problems:-

First we will describe the algorithm of bubble sort:-
1. We take an unsorted array and then from the beginning of the array we compare two elements at a time.
2. Then if the first element is greater then the second element we swap i.e. we interchange their positions.
3. Then we keep on doing this until the whole array is sorted.

Now we have to store the positions at which each of the element of the array is going during the above sorting process.
1. So to do this what we do is that we take another array(temp array) which stores the indexes of the corresponding values and whenever the swap function is called, with the swapping of two elements the indexes of those elements in the original array are also swapped.
2. And now to store their positions, we take a little help of the concept of linked list. What we do is that we make array of linked list where the heads are the indexes which always corresponds to a single element.
3. So when we check the two elements and swapping is done we push the indexes accordingly to corresponding heads. Then finally we can get the all the positions of every element of array to which they are going during the sorting process by just printing the array of linked list.

Now to print the elments at each indexes during the sorting process:-
1. Our approach is same as above. We simply make another array of linked list where the heads will be the indexes.
2. Then after every swap a new array is formed and we just push the data at their indexes to their corresponding heads.
3. Then we print that array of linked list.

The pseudo-code for the algorithms described above is the following:-
The pseudo-code contains two arrays of the structure type, comtaining the data and pointer to the address. The array "temparr" stores the original positions of the elements in the unsorted array.
Input: An array of size N filled with random integers.
Output: An array.
Consider the following:-

$$\begin{pmatrix} 5 & 1 & 4 & 2 & 8 & 9 \end{pmatrix}$$

The array given above is run through the sorting function and we get the following output:-

$$\begin{pmatrix} 1 & 2 & 4 & 5 & 8 & 9 \end{pmatrix}$$

III. Time Complexity Analysis and Discussion

1. *Best Case* : Consider an array of size N filled with the sorted numbers in increasing order. For such a case, the output will remain the same. This the time taken is of the order n which is the best-case. This concludes our best-case scenario. For e.g.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

2. *Worst Case*:- Consider an array of size N filled with numbers such that they are in reverse order. In such a case only maximum time will be taken. For e.g.

$$\begin{pmatrix} 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix}$$

Algorithm 1 Sorting by using bubble sort and trace the index and elements

```
procedure push(head, x)
    push element at the end of the linked list

procedure print(head)
    print linked list

procedure Swap( j )
    int temp
    temp = arr[j]
    arr[j] = arr[j+1]
    arr[j+1] = temp
    temp = temparr[j+1]
    temparr[j+1] = temparr[j]
    temparr[j] = temp

procedure Bubblesort()
    for ( i = 0 to n-1) do
        brr[temparr[i]] = push(brr[temparr[i]],i)
    for ( i = 0 to n-1) do
        for ( j = 0 to n-1-i) do
            if (arr[j] > arr[j+1]) then
                brr[temparr[j+1]] =
push(brr[temparr[j+1]],j)
                brr[temparr[j]] =
push(brr[temparr[j]],j+1)
                swap(j)
            for ( k = 0 to n-1) do
                crr[k] = pushend(crr[k], arr[k])
    print(brr[])
    print(crr[])

procedure main
    Bubblesort();
    for ( i = 0 to n-1) do
        print(array[i])
```

The time complexity will be of the order n^2.

3. *Average Case*:- Consider an array of size N that is filled with numbers that are randomly generated. The time take to compute this will lie between the best-case and the worst-case. It is difficult to compute the time for such a case in general because it will be purely case-dependent. But the limits can be found.

To verify the theoretical claim, we plotted the graph between T VS N, to find out that the worst-case if proportional to n^2 and the best-case is proportional to n and all other cases lie in this range.

WORST CASE = O(n^2)

BEST CASE = $\Omega(n)$

IV. Experimental Study

In this section of the repost the actual data of the time calculation has been shown. The time plots have been constructed using the unit time equations at every step of the computation. Further the data was written in a file and GNU plots were constructed to verify our claims in the previous sections. All the values have been obtained considering unit time.

TABLE I
TIME COMPLEXITY TABLE- PART(B)

n	t_{best}	$t_{average}$	t_{worst}
10	82	82	532
20	172	358	2167
30	262	855	4902
40	352	1147	8737
50	442	442	13672
60	532	1158	19707
70	622	1352	26842
80	712	712	35077
90	802	1758	44412
100	892	892	54847

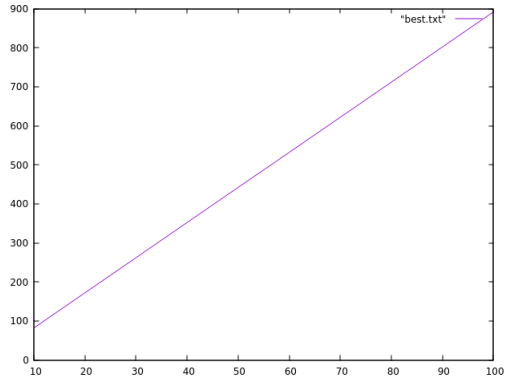

Fig. 1. Best Case T(y-axis) VS N(x-axis)

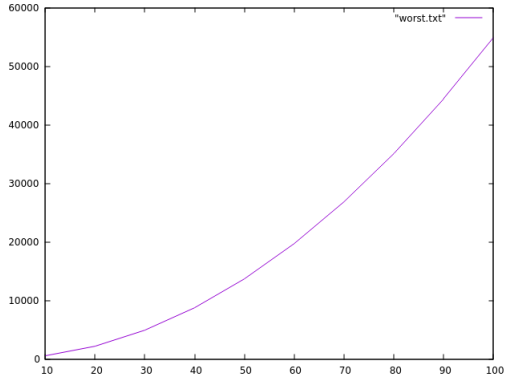

Fig. 2. Worst Case T(y-axis) VS N(x-axis)

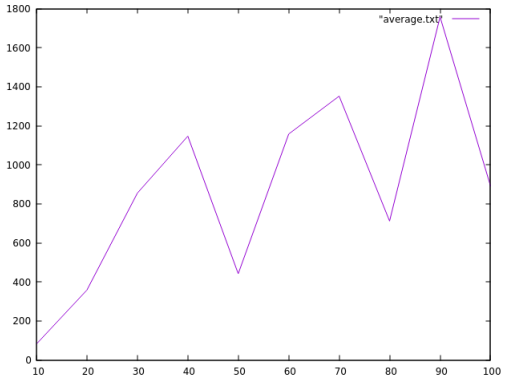

Fig. 3. Average Case T(y-axis) VS N(x-axis)

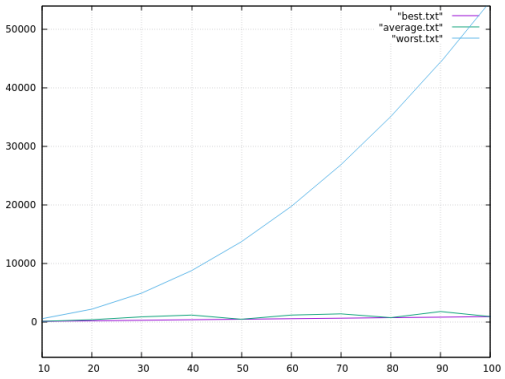

Fig. 4. Relative Cases T(y-axis) VS N(x-axis)

The average and best case nearly overlap each other while worst case can be seen evidently.

V. Conclusion

Here, we have tried to design an efficient algorithm that sorts the given set of numbers using bubble sort and also finds the path of each elements i.e the indices of the number that it visits and also we found what the contents of each of the locations are throughout the iterations.

References

[1] http://ocw.utm.my/file.php/31/Module/ocwChp5BubbleSort.pdf

[2] https://www.geeksforgeeks.org/bubble-sort/

[3] http://bvicam.ac.in/news/INDIACom202011/86.pdf