Implementing Bubble Sort Algorithm

① Starting with the first element (Index = 0), compare the current element with the next element of the array.

② If the current element is greater than the next element of the array, swap them.

③ If the current element is less than the next element, Repeat Step 1.

Lets consider array with values {5, 1, 6, 2, 4, 3}

5>1, so interchange	5	1	6	2	4	3
5>6, No Swaping	5	1	6	2	4	3
6>2, so interchange	1	5	6	2	4	3
6>4, so interchange	1	5	2	6	4	3
6>3, so interchange	1	5	2	4	6	3
	1	5	2	4	3	6

This is first Insertation

Similarly after all the insertation the array get stored

So as we can see representation above,
after the first iteration, 6 is placed at the last index, which is the correct position of it.

Similarly after the second iteration, 5 will be at the second last index. and so on.

Optimized The Bubble Sort Algorithm:

we can indicate flag to monitor wheather elements are getting swapped inside the inner for loop:
Hence, in the inner for loop, we check wheather swaping of elements is taking place ~~and~~ not, everytime..
If for a particular iteration, no swaping took place. It means the array has been sorted and we can jump out of the for loop, insted the executing all the iterations.

Lets conside any array with values

{11, 17, 18, 16, 23}

11 > 17 (No Swaping)

11	17	18	26	23

flag = 0 (flag remains 0)

17 > 18 (No swaping)

11	17	18	26	23

flag = 0

18 > 26 (NO)

11	17	18	26	23

flag = 0

26 < 23 (Swap them)

11	17	18	26	23

flag = 1

Complexity:

If bubble sort, $n-1$ Comparisons will be done in the 1st pass, $n-2$ in 2nd pass, $n-3$ in 3rd pass and so on. So the total number of comparisons will be,

output: $(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$

$$Sum = n(n-1)/2$$

i.e $O(n^2)$

2020

Hence Time Complexcity of bubble Short is $O(n^2)$

-> Worst time Complexicity [Big-O] : $O(n^2)$

-> Best time Complexity [Big-omeg] : $O(n)$

-> Avarage time complexity [Big-theta] : $O(n^2)$

-> space Complexity : $O(1)$

Linear Search

Worst Case:

For the linear search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() function compares it with all the element of a[] one by one.

So the worst case time complexity of linear search would be $O(n)$.

Average Case:

For the linear search problem, assume that the all cases are uniformly distributed. So we sum all the cases and devided the sum by $(n+1)$.

$$\text{Average Case time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{O((n+1)*(n+2)/2)}{(n+1)}$$

$$= O(n)$$

Best Case: merge sort does O(nLogn) operations for all cases.

— o —