Tushar Upadhayay

11911050

CSE

1) INSERTION SORT

Let there be an array of size 'n'.

⇒ Main Code

```
for (int i=1; i<n; i++)
{
    temp = a[i]
    j = i-1;
    while (j>=0 && a[j] > temp)
    {
        a[j+1] = a[j];
        j--;
    }
    a[j+1] = temp;
}
```

for the worst case scenario inner loop will be executed 'i' times for every 'i' in outer loop.

If i=n

Worst time is = $1+2+3+\cdots+(n-1)$

$\Rightarrow \frac{n(n-1)}{2} = \frac{n^2-n}{2}$

$\Rightarrow \frac{O(n^2-n)}{2} = O(n^2)$

11911050

For best case the array will be in ascending order
∴ The inner loop will be executed only once.
for every 'i' inner loop will be executed once.

⇒ 1+1+ - - - - -(n-1) times
⇒ O(n-1)
⇒ O(n) time complexity.

Now,
Can we optimise it further?

(i) We can optimize the searching by using binary search

It will improve the searching complexity from O(n) ⟶ O(logn) for 1 element
So for n element ⟶ O(nlogn).

But since it will take O(n) for one element to be placed at correct position, n elements will take n*O(n) ⟶ O(n²)

So the complexity remains same O(n²)

(ii) We can optimize the swapping by using linked list

It will improve the complexity of swapping from O(n) to O(1) due to use of pointers. But the complexity of search remains O(n²). So overall complexity remains same O(n²)

* Therefore we conclude that we cannot reduce the worst case complexity of Insertion sort from $O(n^2)$.

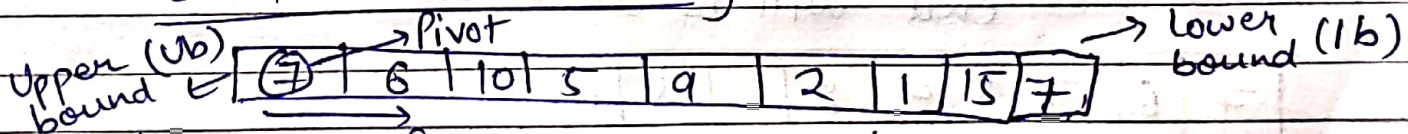
(2) Quick Sort: (In place)

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

* The key process in quick sort is partition. We have to put pivot element at its correct position in sorted array and put all smaller element (smaller than pivot) before it and rest to the right of pivot element.

Example:

let there be an array:



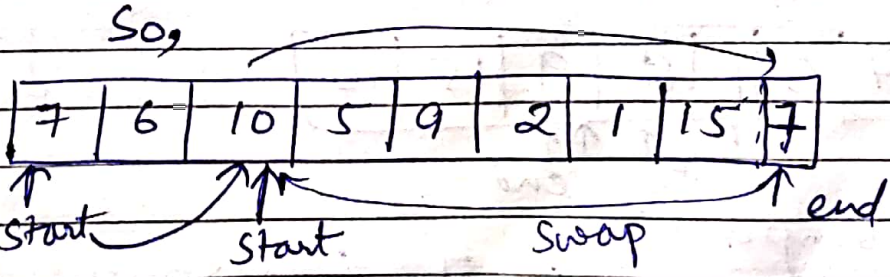
We take first element as pivot and compare it with 6.

$6 < 7$ (True)

then compare with 10

$10 < 7$ (False)

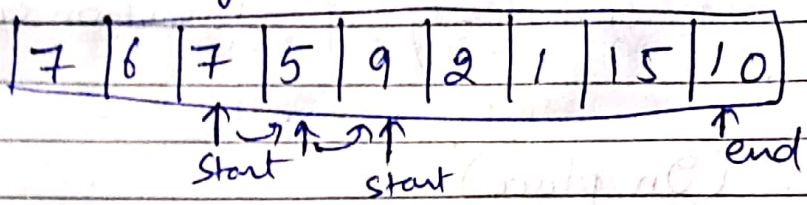
So,



Now we compare from the end any element which is less than Eq to 7 and swap with '10'.

11911050

Now array (After first swap)



Now compare

7 <= 7 (Increment Start)

5 <= 7 (Increment Start)

9 <= 7 (false)

Start will be at 9

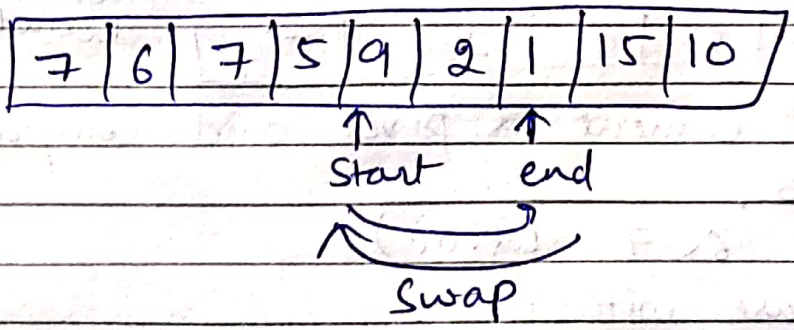
Now compare end:-

7 <= 10 (decrement ~~increment~~ end)

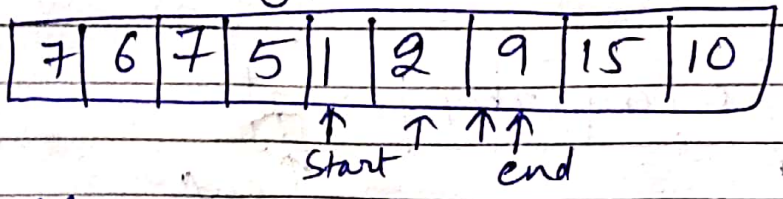
7 <= 15 (decrement ~~increment~~ end)

7 <= 1 (false)

end will be at 1



Now Array (After second swap)



Now compare

1 <= 7 (Increment Start)

2 <= 7 (Increment Start)

9 <= 7 (false)

Start is at '9'

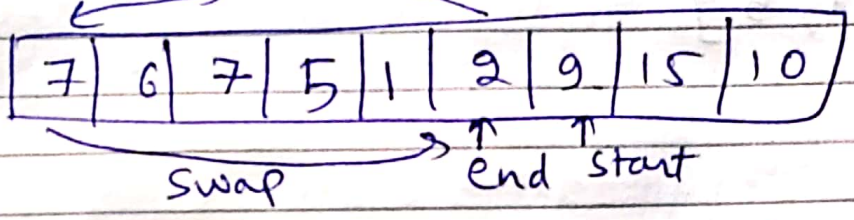
Now compare

7 <= 9 (decrement end)

7 <= 2 (False)

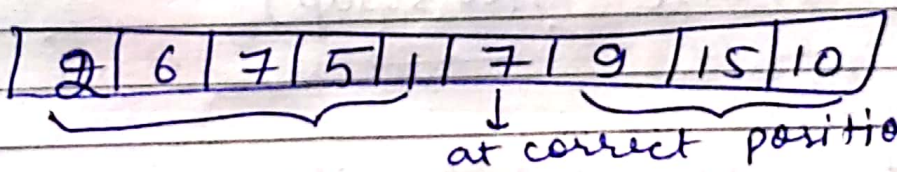
So end at '2'

Now array



Now start has crossed end so we will not swap instead we will swap end with pivot element i.e. 7 at a[0].

Now '7' is at correct position. The same procedure will be followed by taking pivot between 2 and 1 & 9 and 10.



In this way Quick sort works.

Bubble Sort: (In place)

It is the simplest sorting algorithm that works by repeatedly swapping adjacent elements if they are in wrong order.

Example:-

Let we have an array

A = [5, 4, 1]

11911050

First Loop :-

5 > 4 (swap)

→ A = [4, 5, 1]

5 > 1 (swap)

→ A = [4, 1, 5]

II Loop :-

4 > 1 (swap)

→ A = [1, 4, 5]

Now the array is sorted but the program will still run.

4 > 5 (False → No swap)

III Loop

1 > 4 (False → No swap)

1 > 5 (False → No swap)

A = [1, 4, 5] (Sorted)

11911050

Bubble sort algorithm:

```
for (i=1 ; i<m ; ++i)
  {
    for (j=0 ; j<(m-i) ; ++j)
    {
      if (a[j] > a[j+1])
        {
          k = a[j];
          a[j] = a[j+1];      } Swapping
          a[j+1] = k;
        }
    }
  }
```

Quick Sort Algo:

```
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low-1);

for( int j = low; j<=high-1 ; j++)
  {
    if (arr[j]<pivot)
    {
      i++;
      swap(&arr[i], &arr[j]);
    }
swap (&arr[i+1], &arr[high]);
return (i+1); }
```

11911050

```
Void quickSort (int arr[], int low, int high)
{
    if (low < high)
    { int p = partition (arr, low, high);

        quickSort (arr, low, p-1);
        quickSort (arr, p+1, high);
    }
}
```

11911050

In place algorithms are those algorithms that does not need an extra space and produces an ouput in the same memory that contains the data by transforming the input 'in place'.

* So Quick Sort and Bubble sort are Inplace. whereas Merge sort is not because it requires an extra O(n) space.

⇒ Merge Sort is Out of place algo.

Time Complexity of Bubble Sort

In bubble Sort worst case (n-1) comparisons will be done in 1st pass, (n-2) in 2nd pass and so on

⇒ $(n-1)+(n-2)+\cdots+3+2+1$

⇒ $\frac{n(n-1)}{2}$ ⇒ $O\left(\frac{n^2-n}{2}\right)$ = $\boxed{O(n^2)}$

Best case :- $1+1+1+\cdots$ (n-1) times

⇒ $O(n-1) = \boxed{O(n)}$

Time complexity of Quick Sort

Worst Case :- When pivot element is smallest or largest or all elements are same.

The returned sublist will be of size (n-1).

If this happens in every partition then the ith call will do $O(n-i)$ work to do the partition

$$\Rightarrow \Theta \sum_{i=0}^{n} \Theta(n-i) = n+(n-1)+(n-2)+\cdots 3+2+1$$

$$\Rightarrow \left(\frac{n^2+n}{2}\right)$$

$$= O\left(\frac{n^2+n}{2}\right)$$

$$\Rightarrow \boxed{O(n^2)}$$

Best Case :-

In most balanced case, each time we perform a partition we divide the list in two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only $\boxed{\log_2 n}$ nested calls before we reach a list of size 1.

Each level of calls needs only $O(n)$ time all together.

$$\Rightarrow O(n \times \log_2 n)$$

$$\Rightarrow \boxed{O(n \log n)}$$

11911050

Comparison of Time Complexities

	Best Case	Worst Case
Insertion Sort	$O(n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$

Time Complexity of Merge Sort

When we divide a no in half in each step $\longrightarrow \log n$

no of steps $\longrightarrow \log(n+1) \longrightarrow$ atmost

to find middle of any array $\rightarrow O(1)$

And to merge $\longrightarrow O(n)$

$\Rightarrow$ Total time $\Rightarrow n \log(n+1)$

$O(n \log n)$

11911050

(3) In my program the merge() fn is used for merging 2 halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1, r] are sorted and merges the two sorted sub-arrays into one.

MergeSort(arr[], l, r)
If r > l

1. Find the middle point to divide the array in 2 half
   middle = m = $\frac{(l+r)}{2}$
2. MergeSort for 1st half:
   MergeSort(arr, l, m)
3. MergeSort for 2nd half:
   MergeSort(arr, l+m, r)
4. 2 half are then merged
   Merge(arr, l, m, r)

11911050

(4) Approach used by me :-

I created different functions for different functions namely :-

(i) height → to calculate height (left and right)
(ii) ~~Right~~ rotate right → to Rotate Right
(iii) rotate left → to Rotate left
(iv) RR → for Right Right Rotation.
(v) LL → for Left Left Rotation.
(vi) LR → for Left Right Rotation
(vii) RL → for Right Left Rotation
(viii) Bf → to calculate the binary factor and rotate according to it.
(ix) inorder → to print in inorder
(x) insert → to insert a new node.

In the main fn I take value from user for the no of elements and store in 'n'. and using for loop take entry using insert function.
And using inorder fn print the ~~value of~~ list in order form after suitable rotations.