NAME→ RACHIT SONTHALIA

ROLL NO.→ 11911079

Page No.:
Date: / /

1.) Insertion Sort

for an array of size n

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

∴ in the worst case inner loop executes i times for every i in outer loop.

∴ if i = n

The no. of time loop executes in worst time is $1+2+3+\cdots+(n-1)$

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

∴ in worst case $\approx O(n^2)$

for best case scenario the array should be in ascending order

NAME → RACHIT SONTHALIA

ROLL NO. → 11911079

Page No.
Date: / /

if the array is in the ascending order. The inner loop will execute only once.

for every i the inner loop will execute 1 time

∴ Time complexity ⇒ 1+1+ ---- upto (n-1) time

⇒ n-1

Time complexity ≈ O(n)

Example:

for array size n = 5

```
for (i=1 ; i<5; i++)
{
    for (j=i ; j>=0 ; j--)
    {   if ( )
        ---
    }
```

if the code is in ascending order the inner loop will terminate for every i in the if condition.

Time complexity ⇒ O(5)

In the best case scenario.

NAME → RACHIT SONTHALIA

ROLL NO. → 11911079

In the worst case scenario the array is in descending order and we have to arrange it in ascending order.

To reduce complexity of insertion sort in worst case we have to increase its space complexity

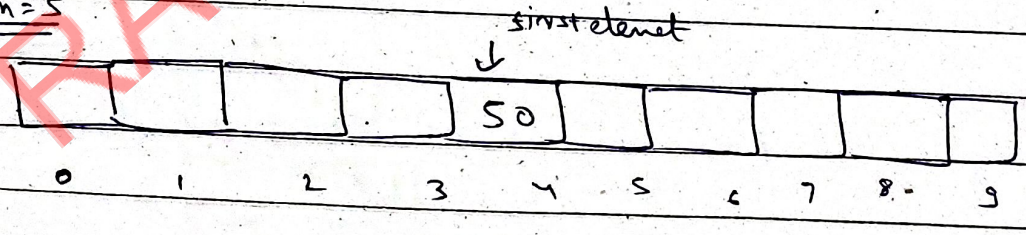
for example array is arr = {50, 40, 30, 20, 10}

using insertion sort to arrange ascending order

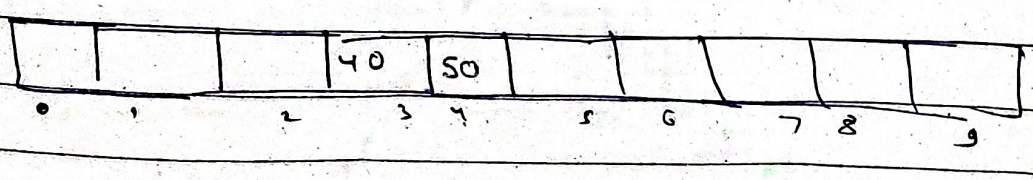
Take array of size 2n if their are n elements to be sorted

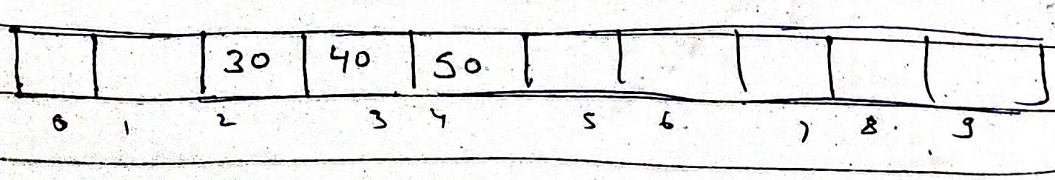
and start filling the elements from n-1 postion.

for n=5



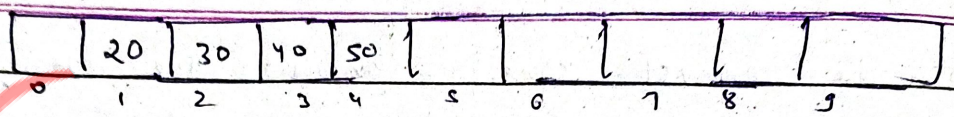
array of size = 10

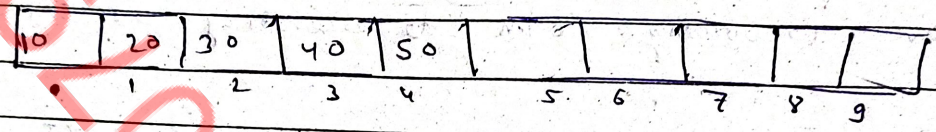




NAME → RACHIT SOMTHALIA

ROLL NO. → 1191079





So here the time complexity ~~for~~ is $O(n)$ and it was worst case scenario.

So by this time complexity in best and worst case scenario is $O(n)$ ~~but for average~~ case it is still $O(n^2)$

1191079

Page No.:
Date: / /

Quick Sort (Inplace)

pivot pt is choosen in quick sort

all the values left of the pivot pt are less than the value of pivot pt and all the values greater than pivot pt are on the right of it.

The pivot pt divides array into 2 ~~bad~~ parts

worst case → when pivot element is smallest or largest or all elements are same.

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

```
partion (l, m, array)

    int start = l
    int end = m
    pivot = a[l]

    while (start < end)
    {
        while ( a[start] <= pivot) && start <= m)
        {
            start ++;
        }
        while (a[end] > pivot) && (end >= l)
        {
            end --;
        }
        if (start < end)
```

1911079

Page No.:
Date: / /

C

swap (array [start], array [end]);

}

}

swap (array [1], array [end])

return end;

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

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

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

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

$$\rightarrow O(n^2)$$

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

11911079

Page No.:
Date: / /

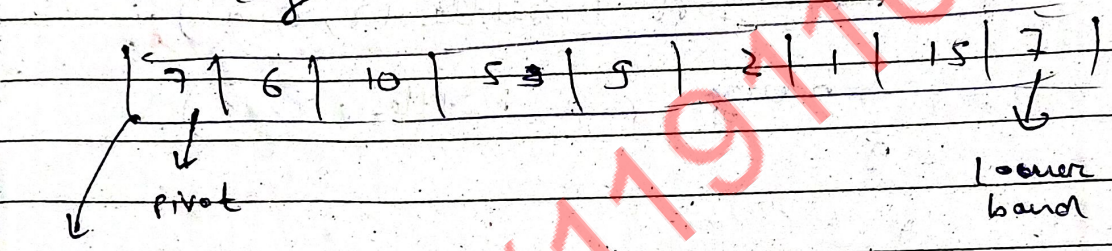
each level of calls needs only ~~to~~ O(n) time all together.

$O(n \times \log_2 n)$

$\rightarrow O(n \log n)$

Example.

An array



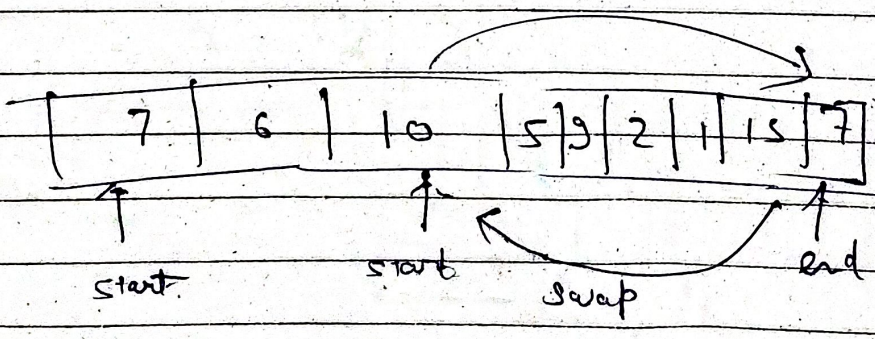
We take first element as pivot and compare it with 6.

$6 < 7$ (True)

Then compare 10

$10 < 7$ ~~(True)~~ (False)

So.

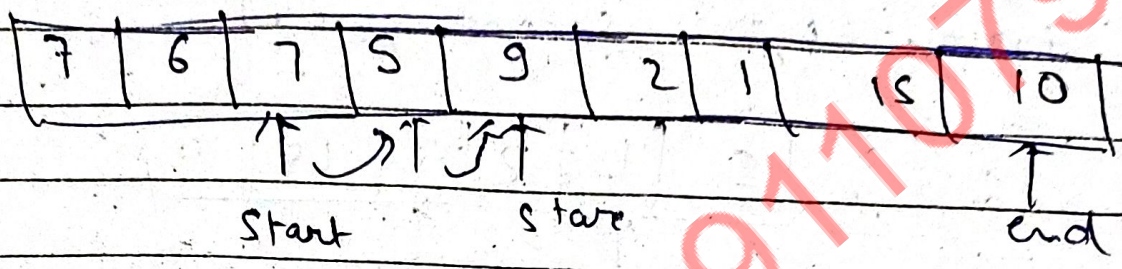


11 911079

Page No.:
Date: / /

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

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 :

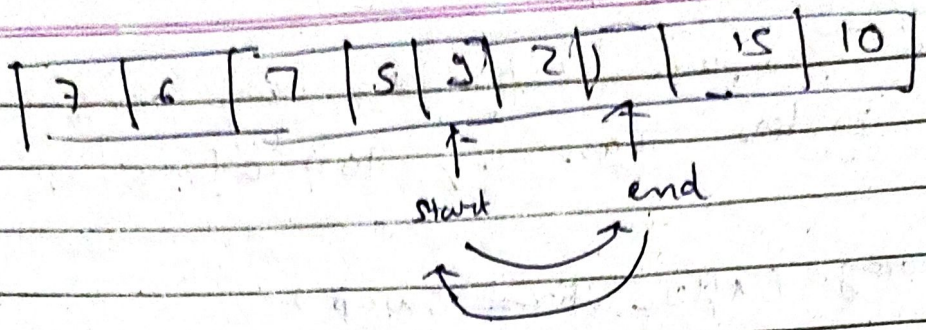
7 <= 10 (decrement end

7 <= 15 (decremed ed)

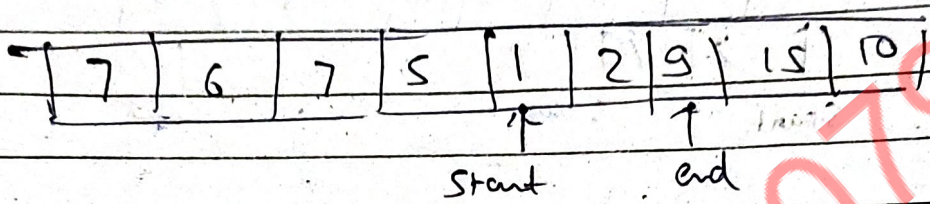
7 <= 1 (false)

end will be at 1

11911079



Now Array (After second swap)



Now compare

1 <= 7 (Increment start)

2 <= 7 (Incremented start)

9 <= 7 (False

Start is at '9'

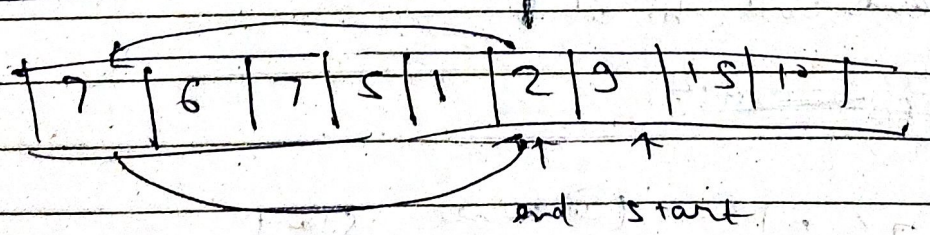
Now compares

7 <= 9 (decrement end)

7 <= 2 (False)

So end at 2

Now array

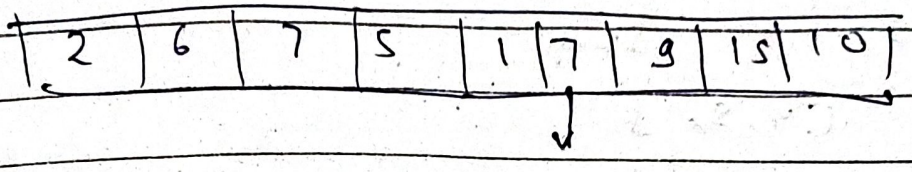


Now start has crossed end so we will not swap instead we will swap end with pivot element i.e. 7 at a[0]

11911079

Page No.:
Date: / /

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



at correct postion

In this way Quick sort works.

Bubble sort (Inplace)

It works by repeatedly swapping adjacent elements if they are in wrong order.

example:-

let we have an array

A = [5, 1, 1]

```
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];
        a[j+1] = k;
```

swapping

```
} } }
```

1911079

A = [5, 4, 1]

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 progra
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)

Timple complexity:

$T(n) = O(n-1) \times O(n-1)$

$[O(n-1)]^2 \approx O(n^2)$

1191079

Page No.:
Date: / /

Loop -1 operates (n-1) times

loop-2 operates (n-2) times

$T(n) \approx O(n^2)$

Insertion sort

Let there be an array of size 'n'

main code

```
for (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

11911079

Page No.:
Date: / /

Worst case

$1+2+3+\cdots\cdots(n-1)$

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

$T(n) = O\left(\frac{n^2-n}{2}\right) \approx O(n^2)$

In place algo are those algo that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'inplace'. So quick sort and bubble sort are inplace where as merge sort is not, because it requires an extra $O(n)$ space.

Merge sort is out of place algo

Comparison of Time Complexity

	Best case	worst case
Insertion sort	$O(n)$	$O(n^2)$
Quick sort	$O(n \log n)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$

11911079

Page No.:
Date: / /

Merge sort analysis

```
void merge_sort ( int l, int r, int a )
{
    if (l<r)
    {
        int middle = (l+r)/2;

        merge_sort ( l, middle, a)
        merge_sort (middle+1, r, a);
        merge ( l, middle, r, a)
    }
}
void merge (int l, int m, int
{
    int n1 = m-l+1;
    int n2 = r-m;
```

1191079

Page No.:
Date: / /

4) The value is entered by the user in the program. n stores the no. of elements and for loop take entry using insert fn. And using inorder fn print the list in order form after suitable rotation. Their are different fn for diff operations. Height is used to calculate height (left and right). rotate right to rotate right same will be rotate left. RR for Right right rotation. LR for left right rotation, LL for left left rotation. BF to calculate the binary factor and rotate according to it. inorder to print in inorder. insert to insert a new node

3) The size of array is entered by the user by which I allocate arr size. string is entered by the user. Then merge sort fuction is used to do a partion of the array and then merge ~~fuction~~ fuction is used to merge the array after sorting. Merge fn check the elements of ~~the~~ both array and combines them in ascending order. The array is partitioned till ~~the~~ very last element and each ~~last~~ contains of single element.