SORTING TECHNIQUES

☆ Criteria Used for Analysing Sorts :-

1. **Number of Comparisons** :- For Sorting the element, Sorting techniques may be Comparing the elements and Actually Time Complexity depends upon No. of Comparisons

2. **Number of Swaps** :- For re-arranging the element in increasing/Decreasing order, we will Swaps and we will analyse no. of Swaps.

3. **Adaptive** :- If a Sorting technique already takes minimum time to arrange an already Sorted list, then we call it Adaptive.

4. **Extra Memory** :- Some Algorithms require extra space.

5. **Stable** :- Suppose we have list of Students & marks

Name →	A	B	C	D	E	F	G
Marks →	5	8	6	4	6	7	10

See, C & E have same marks (6) and we want after Sorting C → 6 will always before E → 6, and for this Such algorithms are called Stable.

After Sorting :-

Name →	D	A	C	E	F	B	G
Marks →	4	5	6	6	7	8	10

⇒ **Reason** :- In Databases, when we want to sort the differe
Columns then we will perform this as the resu
first sort should be preserved in next sort, it m
if there are duplicate then their order should be preserve
"Not All Agorithms are stable, few are stable".

★ **No. of Sorting Techniques** :-

1. Bubble Sort
2. Insertion Sort
3. Selection Sort

Time Complexity :- $O(n^2)$

4. Heap Sort
5. Merge Sort
6. Quick Sort
7. Tree Sort

Time Complexity :- $O(n \log n)$

8. Shell Sort

Time Complexity :- $O(n^{3/2})$

"Comparison Based Sorts

9. Count Sort
10. Bucket / Bin Sort
11. Radix Sort

Time Complexity :- $O(n)$
but occupy more space

Index Based Sorts

Bubble Sort :-

$n = 5$ A | 8 | 5 | 7 | 3 | 2 |

it will compare consecutive pair of Elements Every time, if first element is greater than second than it will swap

1st Pass :-

8]	5	5	5	5
5]	8]	7	7	7
7	7]	8]	3	3
3	3	3]	8]	2
2	2	2	2]	(8)

⇒ • Largest Element is sorted
• 4 Comparisions.
• 4 Swaps

2nd Pass :-

5]	5	5	5
7]	7]	3	3
3	3]	7]	2
2	2	2]	(7)
8	8	8	(8)

⇒ • Two Largest Elements are sorted
• 3 Comparisons
• 3 Swaps (2 swaps are done but actually 3 swaps can be done)

3rd Pass :-

5]	3	3
3]	5]	2
2	2]	(5)
7	7	(7)
8	8	(8)

⇒ : Three elements are sorted
• 2 Comparisons
• 2 Swaps

4th pass :-

3] 2
2 3
5 5
7 7
8 8

→ . All are sorted
. 1 Comparison
. 1 swap

Analysing :-

Total no. of passes :- 4 (For 'n' elements = n-1)

no. of Comparisons :- 1+2+3+4 (For 'n' elements = 1+2+..+n-1 = $\frac{n(n-1)}{2}$ = $O(n^2)$)

Max. no. of Swaps :- 1+2+3+4 ⇒ $\frac{n(n-1)}{2}$ ⇒ $O(n^2)$

⇒ Few Interesting things :-

- The name is given as bubble Sort, bcz if we observe then we can see that the Heaviest element from the top will goes to the bottom and Lighter element will get up as if we drop something heavy thing in water and bubbles comes up.
- Now if we see that in first pass, we get one sorted element and in second we get two sorted elements and if we are in kth pass we get 'k' sorted elements, so if we want 2 sorted elements then just perform 2 passes, similarly if we want 3 sorted elements then perform 3 passes and so on.

Program :-

```
void Bubble sort (int A[], int n)
{
    int Flag;
    for (i=0 ; i<n-1 ; i++)          — O(n)
    {
        flag = 0;
        for (j=0 ; j<n-1-i ; j++)    — O(n)
        {
            if(A[j] > A[j+1])
            {
                Swap (A[j] , A[j+1]);
            } flag = 1;
        }
    } if (flag==0)
        break;
}
```

This loop is for no. of passes, as we perform 4 passes for 5 elements ∴ (i<n-1)

Nesting

This loop is for comparing the element with next element but in every pass, comparisons will be subtracted by 1, but comparisons are done in all passes, ∴ (j<n-1-i)

Time Complexity :- $O(n^2)$

⇒ Now is it Adaptive or not? Now we take help of flag variable here. So if (flag = 0) at the end of 1st pass, then this list will be sorted (already)

Now, Time taken

3	2	2	2	2	
5	3	3	3	3	⇒ No. of Comparisons = n-1 = 4
7	5	5	5	5	O(n)
8	7	7	7	7	No. of swaps = 0
	8	8	8	8	

So, max. time taken by bubble sort = $O(n^2)$

Min. time taken by bubble sort = $O(n)$ → when list is already sorted.

"So, Bubble sort is Adaptive, if list is already sorted"

Note :- Bubble Sort is not itself Adaptive, we can make it Adaptive (Like here we use flag variable), and also Bubble Sort known as Adaptive Algorithm.

⇒ Now, whether it is stable or not ?

Now,

```
*8
 8   > if we sort them,
 8     then they will not
 3     swap, bcz they are
 5     equal (i.e. * ≠ *)
 4
```

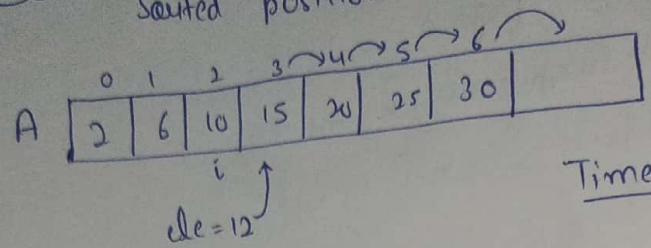
→ Hence, Bubble Sort is Stable, bcz * → 8, will retain its position as it is

Code :-

```
int main ()
{
  int i;
  int A[] = {3,7,9,10,6,5,12,4,11,2};
  int B[] = {2,3,4,5,6};
  void bubbleSort (int A[], int n);
  bubbleSort (A, 10);
  bubbleSort (B,6); // In debug area we can that flag
                       will be 0 and loop will break in
                       bubble Sort.
  for (i=0 ; i<10; i++)
  {
    printf ("%d _", A[i]);
  }
}
```

```
void bubbleSort (int A[], int n)
{
    int i, j, flag = 0;
    void swap (int *x, int *y);
    for (i=0 ; i<n-1 ; i++)
    {
        flag = 0;
        for (j = 0; j<n-1 ; j++)
        {
            if (A[j] > A[j+1])
            {
                swap (&A[j], &A[j+1]);
                flag = 1;
            }
        }
        if (flag == 0)
            break;
    }
}
void swap (int *x, int *y)
{
    int *temp = *x;
    *x = *y;
    *y = temp;
}
```

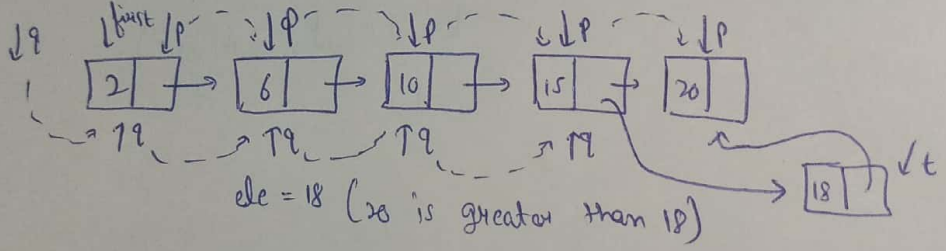
★ **Insertion Sort** :- Inserted an element in Sorted Array at Sorted position.



Time Complexity :- Min — O(1)
Max — O(n)

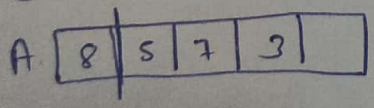
- Shift all the elements to right side which are greater than 12
- Then insert 12 at (i+1).

⇒ **Insertion in Linked List** :- We dont change the position of data here, So we will check first



Time Complexity :- Min — O(1)
Max — O(n)

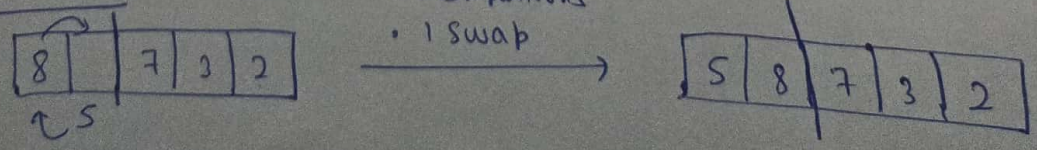
⇒ **Now, We will Study Insertion Sort;**



- In this we will assume first element will be sorted and we have sort other elements and take on this side

1st pass

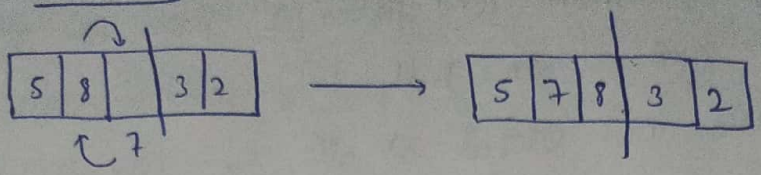
① Insert 5



- Check with (8), it is Smaller than (8), So shift (8) and insert (5) there.

II^nd Pass

Insert 7

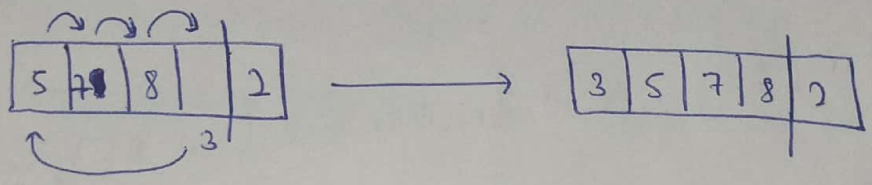


Max. Comparisons = 2

Max. Swaps = 2

III^rd Pass :-

Insert 3

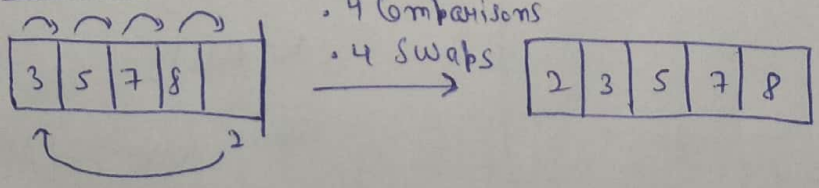


Comparisons = 3

Swaps = 3

IV^th pass :-

Insert 2



No. of Passes = 4 → (n-1) passes

No. of Comparisons = 1+2+3+4 = $\frac{n(n-1)}{2}$ → $O(n^2)$

No. of Swaps = 1+2+3+4 = $\frac{n(n-1)}{2}$ → $O(n^2)$

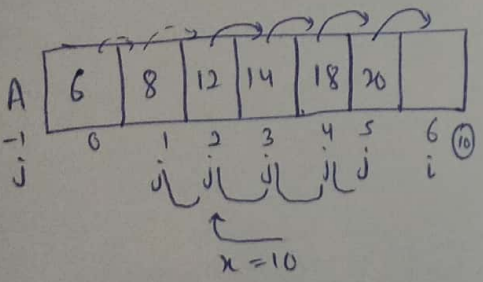
Time Complexity :- $O(n^2)$

Note :- (1) As we see in bubble sort, after 1st pass, we get largest element and after 2nd pass we get two large elements and so on. but In insertion sort only 1 pass 2 pass are not useful at all, bcz we didn't get any sm or larger element.

(2) Insertion Sort is designed for Linked Lists. As we can see in an Array, we have to perform shifting but in Linked List, we dont have to shift Anything, this is the importance.

⇒ **Program** :-

```
Void InsertionSort (int A[], int n)
{
    for (i=1 ; i<n ; i++)
    {
        j = i-1 ;
        x = A[i] ; || Copy inserted element in x
        while ((j>-1) && A[j]>x)
        {
            A[j+1] = A[j];
            j--;
        }
        A[j+1] = x;
    }
}
```



If x = 2

This loop is for no. of passes and yet we start from index = 1.

In while loop, (j > -1) stands for, if we want to insert x = 2, than, j will be (-1) but we dont want this thing thats why we apply this condition.

Now, is it Adaptive or not ?

Let, A | 2 | 5 | 8 | 10 | 12 | , already sorted, Now, while sorting it first we check 5 with 2, than 8 with 5, than 10 with 8 and than 12 with 10, so No Swapping & No Shifting and,

No. of Comparisons = $n-1 = O(n)$

No. of Swaps = $0 = O(1)$

So, Time taken by Insertion Sort is $\boxed{O(n)}$, if list is already sorted So, it is taking minimum time, Hence it is Adaptive. and it is Adaptive itself as we dont use any flag (i.e. used in bubble sort).

So, for Insertion Sort :-

	Min	Max.
Time	$O(n)$	$O(n^2)$
Swaps	$O(1)$	$O(n^2)$

⇒ Now, is it Stable or not ?

Let A | 3 | 5 | 8 | 10 | 12 | ⟶ | 3 | 5 | *5 | 8 | 10 | 12 |

x = *5, i = 5

So, it is stable as we can see * → 5 & ∘ → 5 will be occupied different positions as before * → 5 is next to ∘ → 5 and after sorting we get the same (i.e * → 5 is next to ∘ → 5).

Hence, Insertion Sort is Stable.

Code :-

```
int main()
{
  int i;
  int A[] = {3,7,9,10,6,5,12,4,11,2};
  Void InsertionSort (int A[], int n);
  InsertionSort (A, 10);
  for (i=0 ; i<10; i++)
  {
    printf ("%d _", A[i]);
  }
}
Void InsertionSort (int A[], int n)
{
  int i, j, x;
  for (i = 1 ; i<n ; i++)
  {
    j = i-1;
    x = A[i];
    while (j>-1 && A[j]<x)
    {
      A[j+1] = A[j];
      j--;
    }
    A[j+1] = x;
  }
}
```

Comparison of Insertion Sort and Bubble Sort :-

		Bubble Sort	Insertion Sort
Already in Ascend.	Minimum Comp.	$O(n)$	$O(n)$
Already in Descen.	Maximum Comp.	$O(n^2)$	$O(n^2)$
Already in Ascend.	Minimum Swap	$O(1)$	$O(1)$
Already in Desc.	Maximum Swap	$O(n^2)$	$O(n^2)$
	Adaptive	✓	✓
	Stable	✓	✓
	Linked List	Not Suitable	Yes Suitable
	k passes	Useful → we get k Sorted elements	Not useful → we Cant get any result.

Imp

Note :- ① There are only two Adaptive Algorithms for Sorting.

- Bubble Sort
- Insertion Sort

② There are only three Stable Algorithms for Sorting.

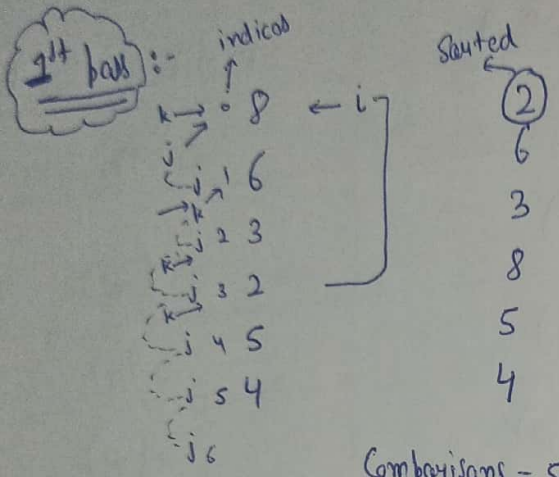
- Bubble Sort
- Insertion Sort
- Merge Sort.

☆ Selection Sort :-

It is also $O(n^2)$ Algorithm and Comparison based Sorting and in each pass get a sorted element. and $(n-1)$ passes required.

A	8	6	3	2	5	4

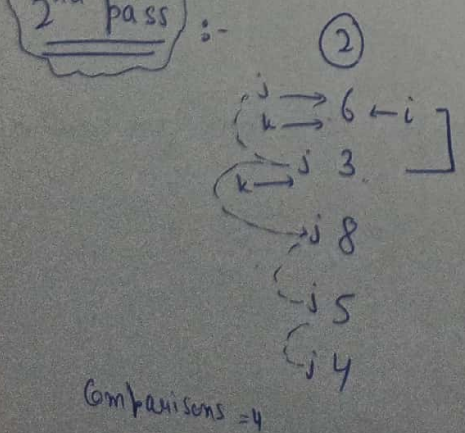
1st pass :-



Comparisons = 5
Swaps = 1

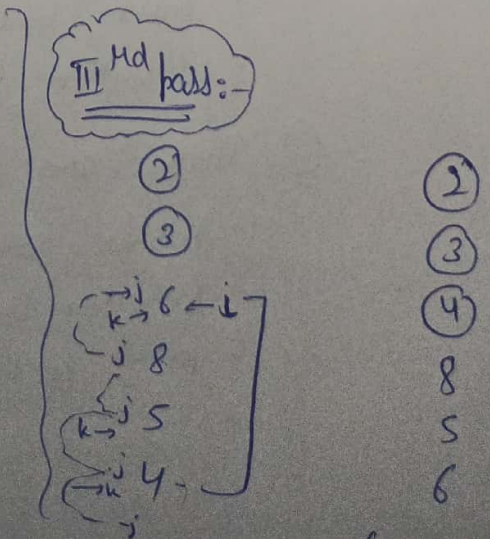
Now, In 1st pass, we will Select first position, to find out an element for that position, so Let us take (i) on 0th index, So who will come at this place, minimum element will take this place, so find out which is smallest, for this we will take two pointers k & j, where (i) points, Now move j to next element and check whether it is smaller than where k is pointing, So bring 'k' here and move j and do this until (j < k) → value, when j will be larger, don't move k, only move j to next location. and when j will be out of the list, and we get smallest element where k is pointing, Now interchange the element from (i) with (k) and we get one/first sorted element and this will be smallest element in the list.

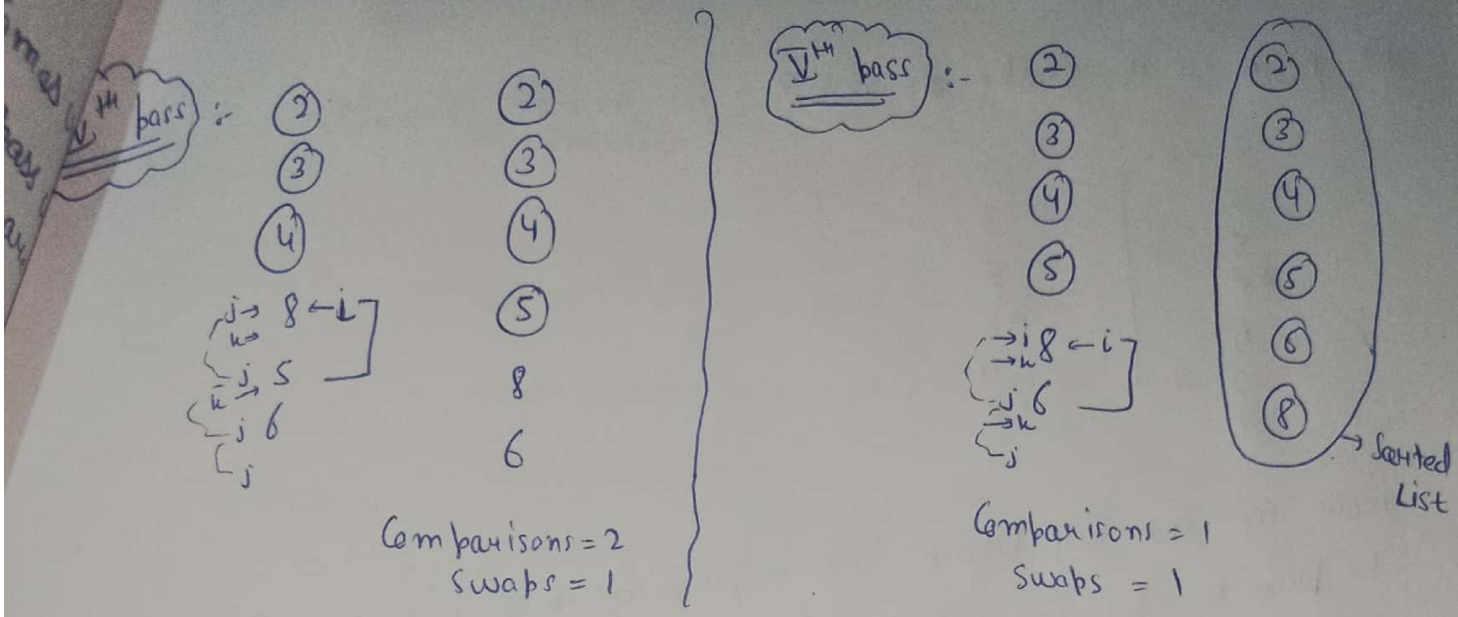
2nd pass :-



Comparisons = 4
Swaps = 1

III Rd pass :-





So, In this procedure, we select a position and found an element for that position thats why it is known as Selection Sort.

No. of Comparisons :- $1+2+3+\cdots+(n-1) = O(n^2)$

No. of Swaps :- For n passes :- $n-1 \Rightarrow O(n)$ Swaps

Note :- ① This is the only Algorithm which has $O(n)$ Swaps as there we are not swapping each and every time, so Selection Sort is good for minimum no. of Swaps.

② If we perform 'k' passes, we will get 'k' smallest elements as in bubble Sort, we get 'k' largest elements, Hence intermediate results of Selection Sort are useful.

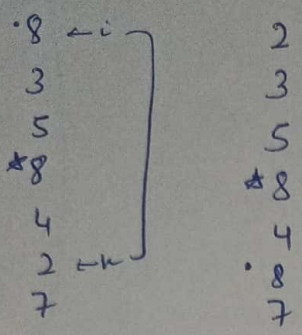
⇒ Now is it Adaptive or not? ⇒ Now if a list is already sorted

0 2 ← i
1 4 ← k
2 8
3 10
4 12
Suppose ← (5)

Now, i will move till the end and it will swap with itself. So if an element is swaping with itself than list is already sorted, so suppose if it is not sorted, than it will swap with itself, than how we can say that all elements are sorted so, it will always take $O(n^2)$ time, So it is not Adaptive.

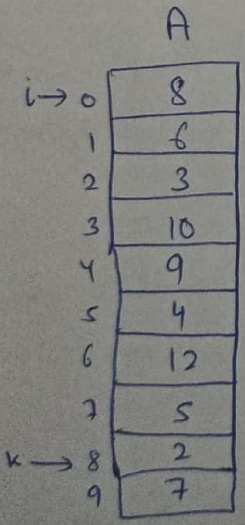
⇒ Now is it stable or not? Let us take example

Code



Now, we can say before ·8 is prior to *8, but after sorting *8 will be prior to ·8, so order of elements are changed, so we can't say / trust Selection Sort for stability, Hence it is not stable. It will not preserve order.

⇒ Program :-




```
void SelectionSort (int A[], int n)
{
    int i, j, k;
    for (i = 0; i < n-1; i++)
    {
        for (j=k=i; j<n; j++)
        {
            if (A[j] < A[k])
                k=j;
        }
        Swap (&A[i], &A[k]);
    }
}
```

Code :-

```
int main()
{
  int i;
  int A[] = {8, 6, 3, 2, 5, 4};
  Void SelectionSort (int A[], int N);
  SelectionSort (A, 6);
  for (i = 0; i < 6; i++)
    {
      printf ("%d_", A[i]);
    }
}
```

```
Void SelectionSort (int A[], int N)
{
  void Swap (int * x, int * y);
  int i, j, k;
  for (i=0; i<n-1; i++)
   { j=k=i;
     while (j<n)
     { j++;
       if (A[j] < A[k])
         { k=j;
           j++;
         }
       else
           j++;
     }
     Swap (&A[i], &A[k]);
   }
}
```

→ By me
(By Sir, is in Program only)

```
void Swap (int * x, int * y)
{
  int temp = *x;
  *x = *y;
  *y = temp;
}
```

★ **Count Sort** :- It is index based Sorting, it is faster but lots of memory.

A	6	3	9	10	15	6	8	12	3	6
	0	1	2	3	4	5	6	7	8	9

So, we require an extra Array with size = largest element in the array to be Sort

Count

0	0	0	2	0	0	1	0	1	1	0	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

→ initialize i with 0

Process :- Scan through all the elements of an Array A, and whatever the element, increment the value at that index in Count, it means no. 8 is present/element is present.
We can say that in Count Array, we get no. of occurance of element in array A

Now, Let Array A be empty, Now Scan through Array Count and Copy element to array A, and Copy as many time as they are occured. (i.e. 3 → 2 times, 6 → 3 time - -).

Time Complexity :- Time for Scanning through A + Time for Count

⇒ O(n) + O(n)

⇒ O(n)

- This is Space Consuming Algorithm.
- This is Linear Time taking Algorithm.

Program :-

```
Void CountSort (int A[], int N)
{
    int max, i, j;
    int *C; // Count Array
    max = findmax(A,N); // To find maximum element in A[]
    C = new int [max+1];
    for (i=0 ; i< max+1 ; i++)        — O(n)
        C[i] = 0
    for (i=0 ; i<n ; i++)             — O(n)
    {
        C[A[i]]++;
    }
    i=0 ; j=0;
    while (i< max+1)                  — O(n)
    {
        if (C[i] > 0)
        { A[j++] = i;
          C[i]--;
        }
        else
            i++;
    }
}
```

⇒ O(n)

n = 10

A: | 3 | 3 | 6 | 6 | 6 | 8 | 9 | 10 | 12 | 15 |
   0   1   2   3   4   5   6   7   8   9

j

C: | 0 | 0 | 0 | 2 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
   0   1   2   3   4   5   6   7   8   9   10 11 12 13 14 15

★ Code :-

```
int main()
{
    int i;
    int A[] = {6,3,9,10,15,6,8,12,3,6};
    Void CountSort (int A[], int n);
    CountSort (A, 10);
    for (i=0; i<10; i++)
    {
        printf ("%d_", A[i]);
    }
}
```

```
Void CountSort (int A[], int n)
{
    int max, i, j;
    int *C;
    for (i=0 ; i<n ; i++)
    {
      if (max < A[i])
          max = A[i];
    }
    C = (int *) malloc (sizeof (int) * (max+1));
    for (i=0 ; i< max+1 ; i++)
     {
        c[i] = 0;
     }
    for (i=0; i<n; i++)
    {
       c[a[i]]++;
     }
   i=0 , j=0;
   while (i<max+1)
   {
      if (c[i]>0)
      {
        A[j++] = i;
        c[i]--;
      }
     else
        i++;
   }
}
```

Quick Sort :-

⇒ **Idea behind Quick Sort** :- Quick Sort works on the idea that an element is in a sorted position, if all the elements before that elements are smaller and all the elements after that element are greater, than we say element is in a sorted position.

→ (10) 30 20 70 40 80 100 90 ⇒ 10 is smallest

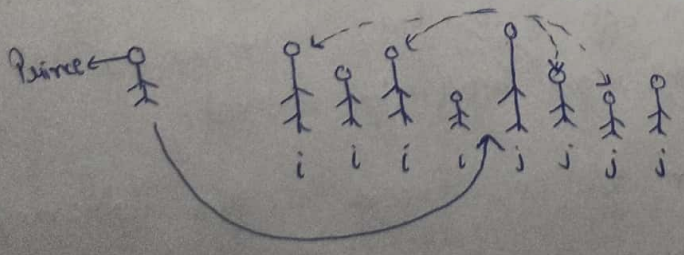
→ 80 70 40 30 20 10 (90) ⇒ 90 is greatest

→ (40 30 20) (50) (90 70 80) ⇒ Quick Sort

Smaller than 50 — Sorted element — greater than 50

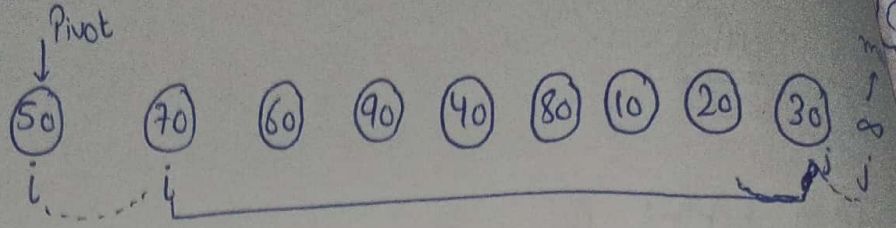
Example :- Let there be group of students and teacher wants to arrange them in increasing order of their heights, So Teacher has two ways :- one is ask every student and select their position, Second is teacher asks students to form a line on their own., so second one is Quick, this is the idea of Quick Sort.

So, suppose, you have to stand in a position such smaller students will be on left side of you and larger students will be on right side, so you have to find your position in that sorted way.



Using i → find anybody taller
Using j → find anybody shorter
ask them to interchange their position.

⇒ Algorithm :-

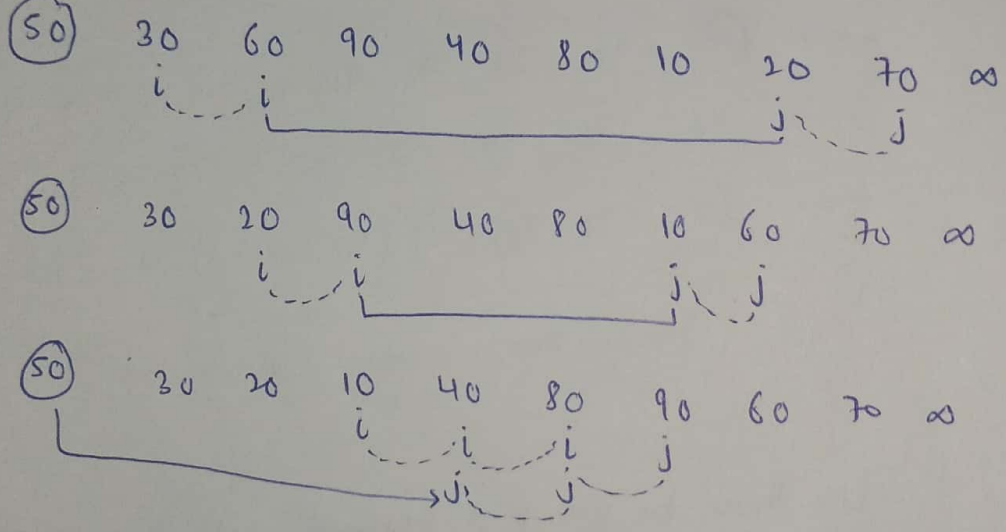


i → will be looking for greater element than Pivot

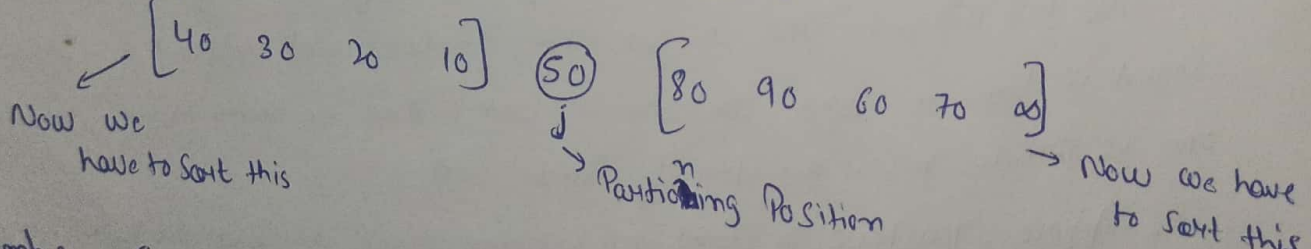
j → will be looking for smaller element than Pivot

So, i → will be stop when i reaches greater than 50

j → will be stop when reach smaller than Pivot.



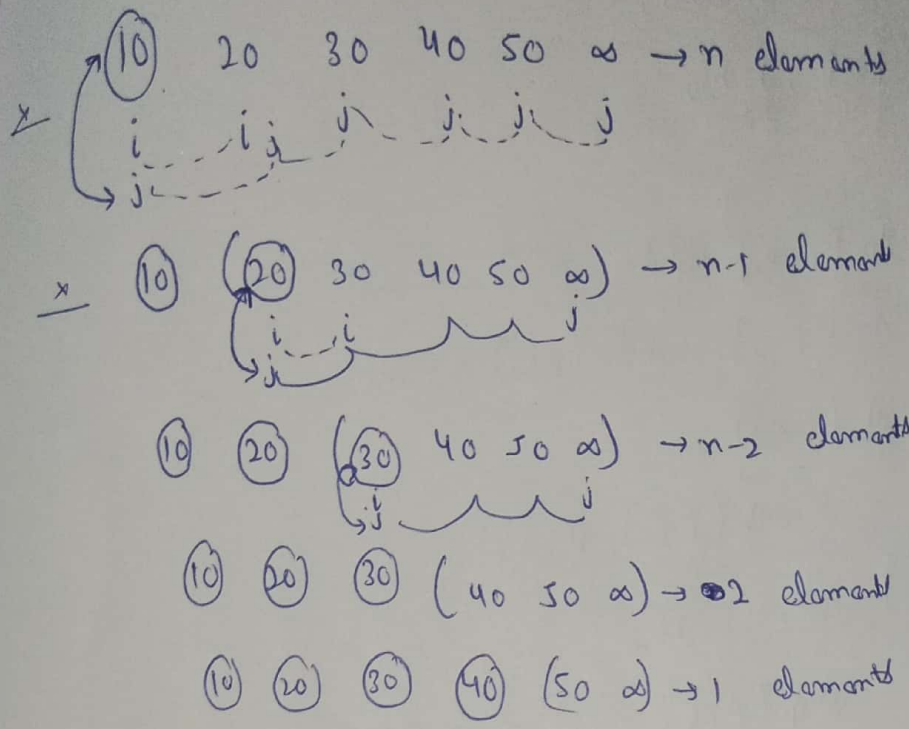
Imp :- When j > i, interchange Pivot with j

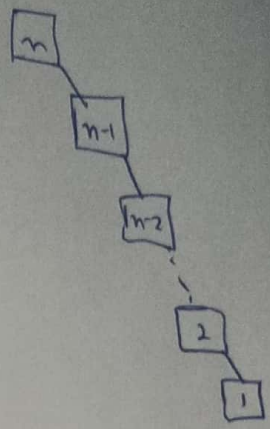


Imp :- So we can see this 50 is sorted (Partitioning Position) and then we have to also sort Left & right list, so we can say that it is recursive for left and right list (ie. for sorting).

Hence, Upto this Partitioning Position, Process is called Partitioning Process and it is recursive.

Program :-

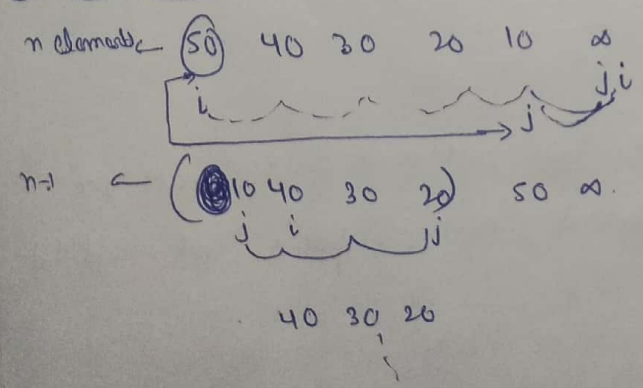


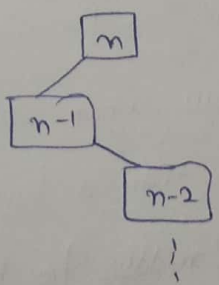


(As j is moving till Pivot) Total no. of Comparisons :- $1+2+3+\cdots+n = O(n^2)$

Actually, this for Sorted list → Worst Case (Time Complexity)

So, if a list Sorted in Ascending order than it takes $O(n^2)$ time.





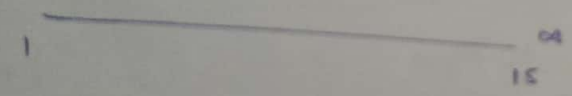
if we Continue, than we know it will also b $O(n^2)$ for Sorted list in Descending order.

⇒ Conclusion :- Whether the elements in Ascending order / Descending order, Quick Sort will always take $O(n^2)$ time.

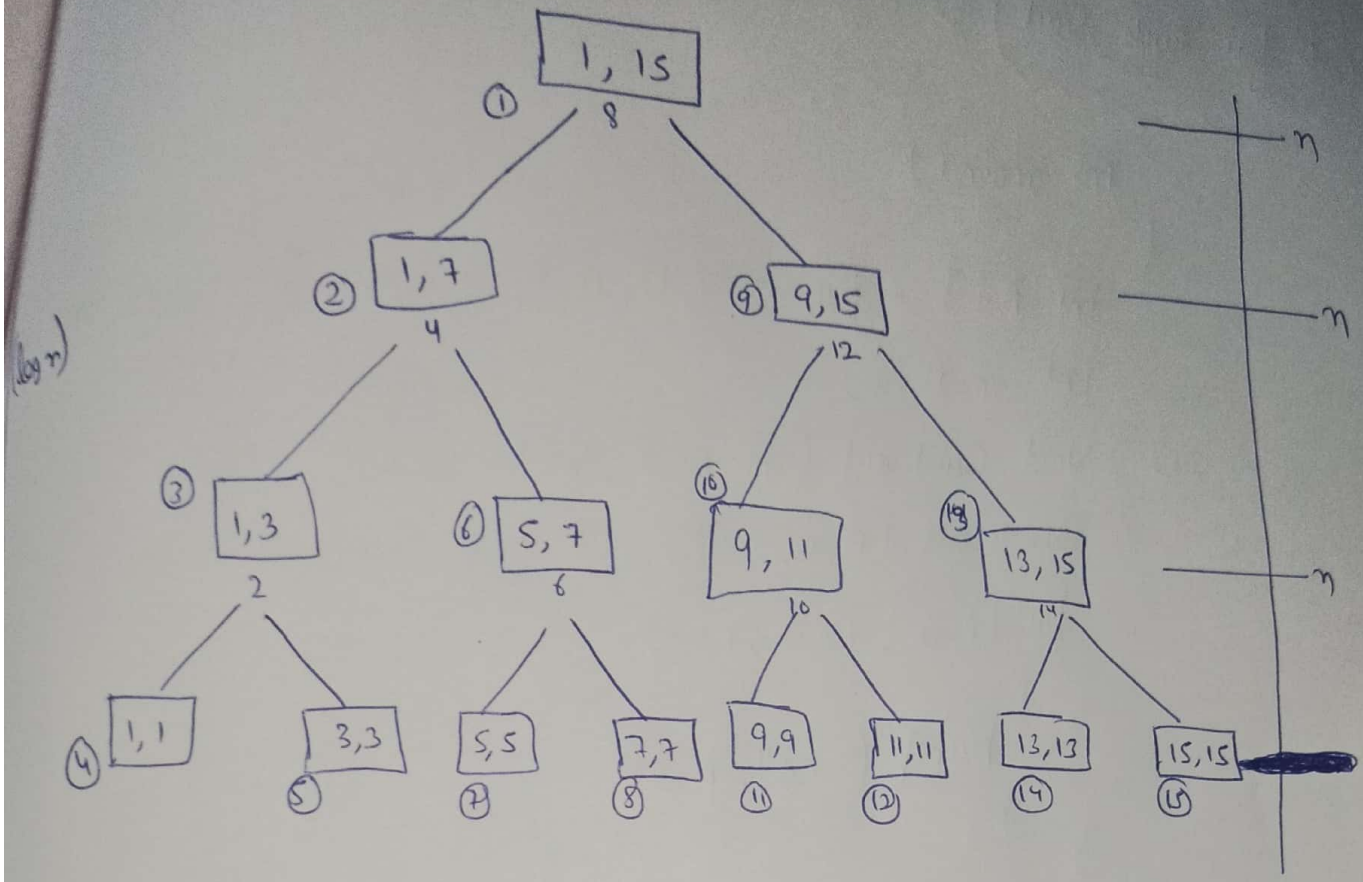
```
int Partition (int A[], int l, int h)
{
    int Pivot = A[l];
    int i=l, j=h;
    do
    {
        do {i++;} while (A[i] ≤ Pivot);
        do {j--;} while (A[j] > Pivot);
        if (i<j)
            Swap(A[i], A[j]);
    } while (i<j);
    Swap(A[l], A[j]);
    return j;
}
```

(log n)

⇒ Now we will understant for best Case, Let Array Starts from index 1 for understanding.



We know, when we perform Quick Sort, it will find a partition position, then a list will be splitted, Now suppose list is splitted in the middle for best Case.



We can see that Quick Sort is a recursive process.

So, we can see that At every level, work is done on n elements and depending upon levels also, which is $O(\log n)$ [bcz it follows succesive division of elements]

So, Total Time Complexity $= O(n \log n)$ → Best Case Analysis

⇒ Conclusion :- ① Best Case → if partitioning is in middle
Time $= O(n \log n)$

② Worst Case → if partitioning is on any end
Time $= O(n^2)$ [Already sorted]

③ Average Case → Time $= O(n \log n)$

⇒ Special Note :- • We take first element as pivot always, but if we take the middle element as pivot and bring it to first, so it will best Case (for sorted list) and Time will be $O(n \log n)$.

- For Worst Case → $O(n^2)$ → Partitioning done at any end → any list
- If we are selecting any element randomly as pivot then it is known as Randomised Quick Sort.

Code for Quick Sort :-

```
int main()
{
  int A[] = {11, 13, 7, 12, 16, 9, 24, 5, 10, 3, --INT.32_MAX--};
  int n=11, i;
  Void QuickSort (int A[], int l, int h);
  QuickSort (A, 0, n-1);
  for (i=0; i<11; i++)
    {
      printf ("%d", A[i]);
    }
}
Void Swap (int *x, int *y)
  {
    int temp = *x;
     *x = *y;
     *y = temp;
  }
int Partition (int A[], int l, int h)
  {
    int Pivot = A[l];
    int i = l;
    int j = h;
    do
     {
       do { i++;
       } while (A[i] <= Pivot);
      do
       { j--;
       } while (A[j] > Pivot);
```

```
        if(i<j)
        Swap(&A[i], &A[j]);
    } while(i<j);
    Swap(&A[l], &A[j]);
    Return j;
}
void QuickSort(int A[], int l, int h)
{
    int j;
    if(l<h)
    {
        j = Partition(A, l, h);
        QuickSort(A, l, j);
        QuickSort(A, j+1, h);
    }
}
```

Note :- "Now, we will do Comparison b/w Selection Sort and Quick Sort."

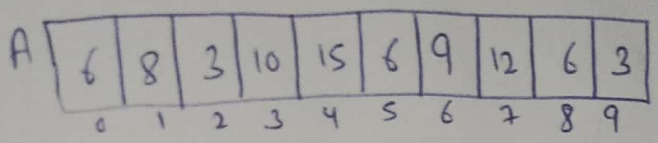
⇒ In Selection Sort → We select a position and find out an Element for that Position.

⇒ In Quick Sort → We select an element and found the position for that element where it should be in the list.

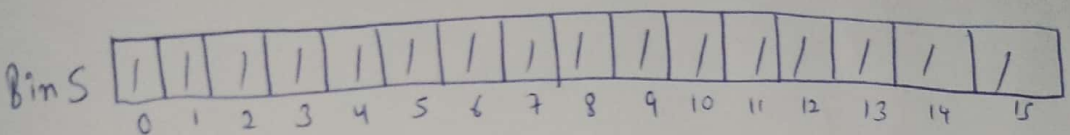
⇒ So, the difference b/w both is Selection (i.e. in Insertion, we select position whereas in Quick, we select an element).

⇒ Quick Sort is also known as :- ① Selection Exchange Sort.
② Partition Exchange Sort.
③ Quick Sort

Bin / Bucket Sort :- It is Similar to Count Sort.

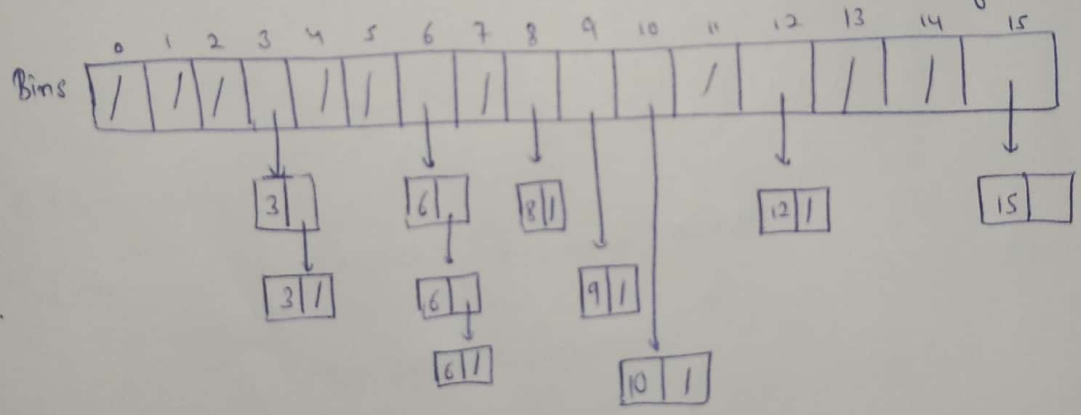


→ Array of Elements



Bins → Array of Pointers named as Bins and initially NULL

Now, we have to do that, we have to Scan through that Array A and Search the index in Bins for that element and drop in th Bin / Bucket or at that index. So we can Say Bins is an Array of Linked list.



Now we have to Scan through Array of Bins and Now delete element in FIFO fashion, in Such a way that Copy elements until the list is empty in Array A and if index is NULL, then just move.

Time Complexity :- $O(n)$

Space Complexity :- As the size of Bins depends upon Largest element in an Array Hence it occupies large space.

i.e. $O(m+n)$ → where m → max. no. element
n → no. of elements

but we can say it is linear $O(n)$.

Program :-

```
Void BinSort (int A[], int n)
{
    int max, i, j;
    Node ** Bins;
    max = findmax (A, n);
    Bins = new Node [max+1];
    for (i=0 ; i< max+1 ; i++)          ——— O(n)
        Bins [i] = NULL;
    for (i=0; i< n; i++)                ——— O(n)
    {
        Insert (Bins [A[i]], A[i]);
    }
    i=0, j=0
    while (i< max+1)
    {
        while (Bins [i] != NULL)        ——— O(n²) X
        {                               ——— O(n)
            A [j++] = Delete (Bin[i]);
        }
        i++;
    }
}
```

As if we can Seen from the Code the time Complexity is $O(n^2)$ but "No", the main operation done is deleting, so this nested while loop is also taking $O(n)$ time, Hence Time Complexity is $O(n)$.

Now, we can see that Size of Bins Array is equal to Largest element to be stored so, it occupies large space, so to Avoid this large space Consumption we will do Radix Sort.

Code for Bin Sort :-

```
Struct Node
{
    int data;
    Struct Node *next;
} *first = NULL;

int main()
{
    int i, n;
    printf("Enter no. of elements : \n");
    scanf("%d", &n);
    int A[n];
    for(i=0; i<n; i++)
    {
        scanf("%d", &A[i]);
    }
    void binsort(int A[], int n);
    binsort(A, n);
    for(i=0; i<n; i++)
    {
        printf("%d", A[i]);
    }
}

int Delete(struct Node **P)
{
    struct Node *q = *P;
    int x = q->data;
    (*P) = (*P)->next;
    free(q);
    return x;
}
```

```
Void insert (Struct Node **P, int x)
{
   Struct Node *t, *q = NULL, *h = *P;
   t = (Struct Node *) malloc (Size of (Struct Node));
   t -> data = x;
   t -> next = NULL;
   if (*P == NULL)  || if list is empty
      *P = t;
   else
     { q = h;
       while (q -> next != NULL)
         { q = q -> next;
         }
       q -> next = t;
         q = t;
     }
}

Void binsort (int A[], int n)
{
   int max, i, j;
   Struct Node **bins;
   max = 0;
   for (i = 0 ; i < n ; i++)
     { if (max < A[i])
         max = A[i];
     }
```

```
bins = (struct Node **)malloc((max+1) * sizeof(struct Node*));
for (i=0 ; i<max+1 ; i++)
  {
    bins[i] = NULL;
  }
for (i=0 ; i <n ; i++)
  {
    insert (& bins [A[i]] , A[i]);
  }
int m=0 , w=0;
while (m< max+1)
    {
        while (bins [m] != NULL)
            {
              A[w++] = delete (& bins [m]);
            }
        m++;
    }
}
```

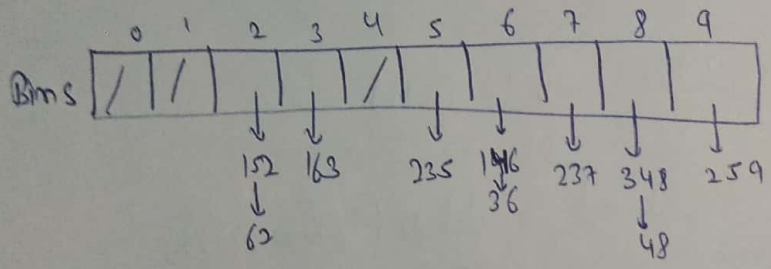
☆ Radix Sort :- It is Similar to Bin Sort.

A	237	146	259	348	152	163	235	48	36	62

But here we will not take Array of 348 bins, we have take only Array upto 10, bcz these are integer values based upon decimal no. System, if there will Binary no. System, than we will take only 2 bins if octal, than only 8 bins are enough.

Now, we will scan through Array A, now we have to drop it in Bin (based upon last digit) (ie. 237 → 7th bin) [A[i]/1 % 10]

① pass

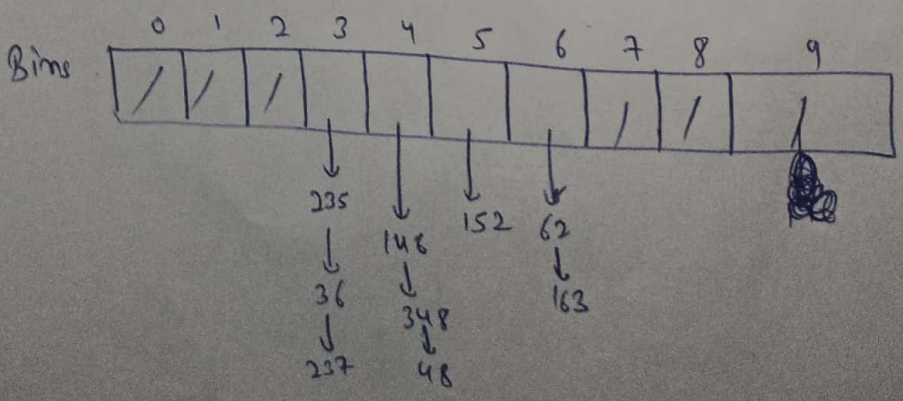


Now, takeout elements and check whether they are sorted or not.

152, 62, 163, 235, 196, 36, 237, 348, 48, 259

but these are not sorted, than again drop them in bins, but this time based upon second digit [ie. 237 → 3rd bin] [(A[i]/10) % 10]

② pass

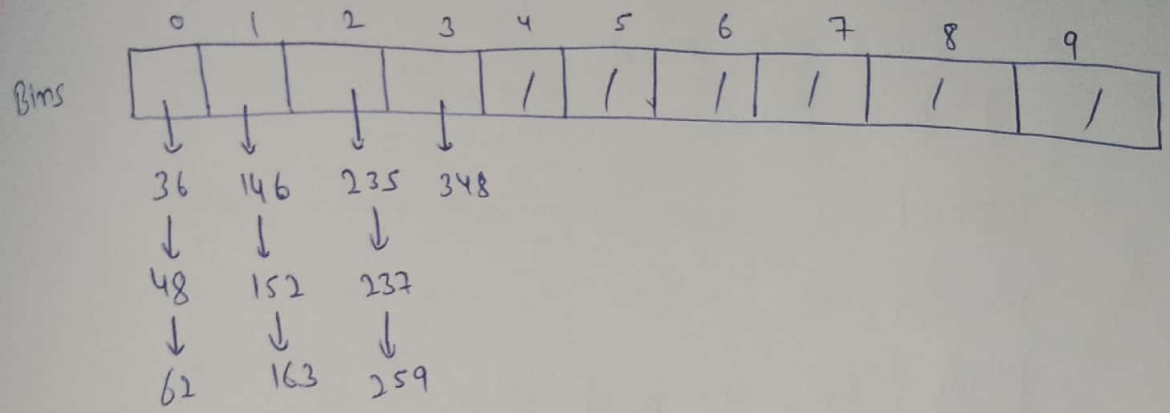


2nd pass Result :- 235, 36, 237, 146, 348, 48, 152, 62, 163

Now Again check them in bins based upon leftmost digit,

235, 036, 237, 146, 348, 048, 152, 259, 062, 163

$\Rightarrow (A[i]/100)\ \%10$



36 146 235 348

↓ ↓ ↓

48 152 237

↓ ↓ ↓

62 163 259

Take out Again, 36, 48, 62, 146, 152, 163, 235, 237, 259, 348

Now, they are sorted, After 3rd pass, So in this space will occupied less, but time is there depending upon no. of digits

Now, Time Complexity :- $O(dn)$ → Here 'n' is we are Copying elements and put them back again depending upon no. of elements (i.e. denoted by d), so TimeComplexity will be $O(n)$ [bcz d is Constant (i.e. max no. of digits)]

Space Complexity :- No. of Nodes Created in bins depends upon no. of elements, so space is limited (i.e. $O(n)$)

Code for Radix Sort :-

```
Struct Node
{ int data;
  Struct Node * next;
} * first = NULL;

int main()
{
    int i, n;
    printf ("Enter the no. of Elements : \n");
    Scanf ("%d", &n);
    int A[n];
    for (i=0 ; i<n ; i++)
     { Scanf ("%d", &A[i]);
     }
    int Count = 0 ;
    int max = -1 ;
    for (i=0 ; i<n ; i++)
     { if (max < A[i])
          max = A[i];
     }
    while (max ! = 0) || checking no. of digits
     {
        max = max / 10 ;
     } Count ++ ;
    int g = 1 ;
   Void binSort (int A[], int n, int g) ;
   for (i=0 ; i< Count ; i++)
     { binsort (a, n, g) ;
     } g = g * 10 ;
```

```
for (i=0 ; i<n ; i++)
  {
    printf ("%d\n", A[i]);
  }
}
Void insert (struct Node **P, int x)
{
    Struct Node *t, *q = NULL, *h = *P;
    t = (struct Node *) malloc (size of (struct Node));
    t -> data = x;
    t -> next = NULL;
     if (*P == NULL) || if list is empty
         *P = t;
      else
        {  q=h;
           while (q -> next != NULL)
             { q = q -> next;
             }
           q -> next = t;
           q = t;
        }
}
int delete (struct Node **P)
  {
     struct Node *q = *P;
       int x = q -> data;
       (*P) = (*P) -> next;
        free (q);
        return x;
  }
```

```
Void binSort (int A[], int n, int g)
{
    int i, j;
    Struct Node **bins;
    bins = (struct Node **) malloc (10 * sizeof (struct Node *
    for (i=0 ; i<10 ; i++)
    {
        bins [i] = NULL;
    }
    for (i=0 ; i<n ; i++)
    {
        insert (&bins [(a[i]/g) %10] , A[i]);
    }
    int w=0 , m=0;
    while (m<10)
    {
        while (bins [m] != NULL)
        {
            A[w++] = delete (&bins [m]);
        }
    m++;
    }
}
```

Shell Sort :- This Sort is used to Sort a large no. of elements in a list and the name is upon the person, who made this Sort.

It is the extended version of insertion Sort. it follows idea of insert Sort.

Like in insertion Sort, elements are inserted one after the another but in Shell Sort, they are inserted at gaps, Now look at this:

$n = 11$

$gap = \lfloor \frac{n}{2} \rfloor$

$gap = \lfloor \frac{11}{2} \rfloor = 5$

A

9	5	16	8	13	6	12	10	4	2	3
0	1	2	3	4	5	6	7	8	9	10

Now, we will Sort the elements with a gap of (5) but again this gap will reduced until gap becomes 1, we will perform insertion

A

9	5	16	8	13	6	12	10	4	2	3
0	1	2	3	4	5	6	7	8	9	10

↑ ← gap (5) → ↑

→ just look at 2 elements and Compare them

A

6	5	16	8	13	9	12	10	4	2	3

A

6	5	10	8	13	9	12	16	4	2	3

A

6	5	10	4	13	9	12	16	8	2	3

A

6	5	10	4	2	9	13	16	8	13	3

Now at gap of (5) there are 3 elements So again Compare

A

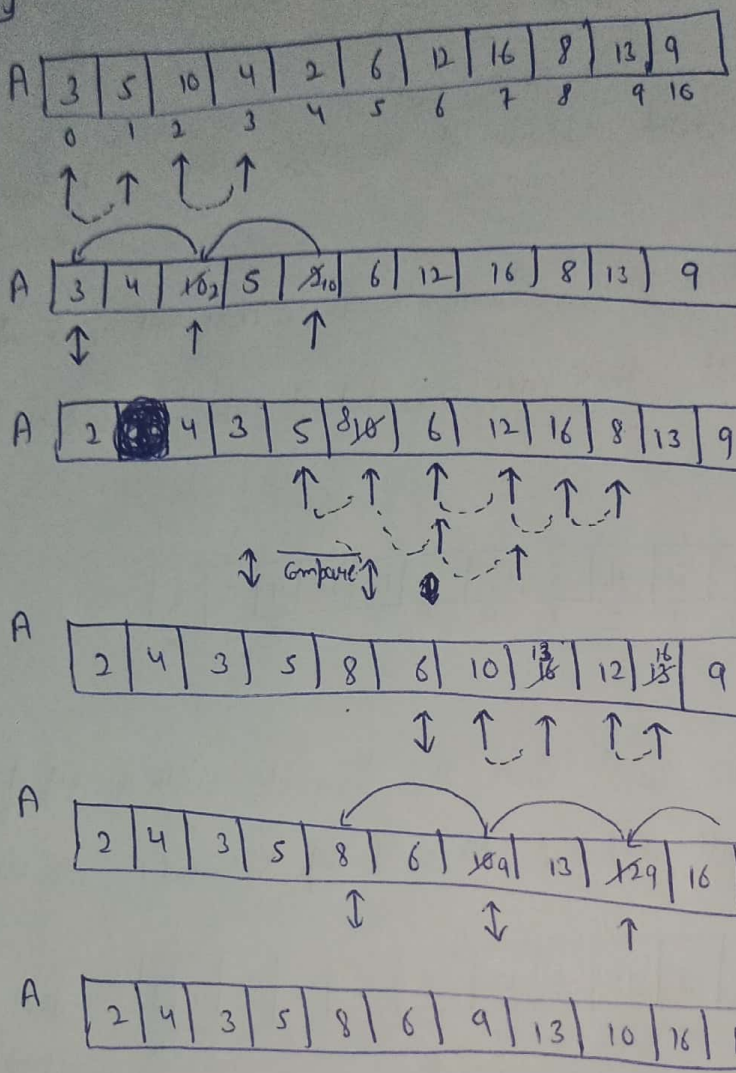
6	5	10	4	2	3	13	16	8	13	9

↑ ← gap (5) → ↑ ← gap (5) → ↑

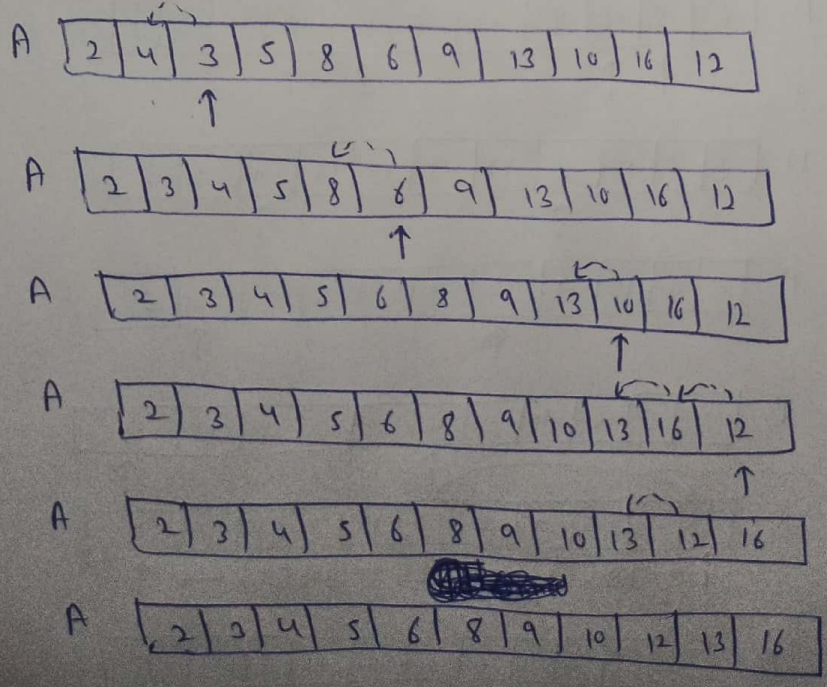
A

3	5	10	4	2	6	13	16	8	13	9

Now, gap = $\lfloor \frac{5}{2} \rfloor = 2$



Now, gap = $\lfloor \frac{2}{2} \rfloor = 1$ [Comparing with previous element, like in insertion sort



ysis :- First we are scanning 'n' elements and we perform $\lfloor \frac{n}{2} \rfloor$ passes (i.e. Successive division By 2)

So, Time Complexity :- $O(n \log_2(n))$

Note :- As we are taking gap $\lfloor \frac{n}{2} \rfloor$, we can also take gap equals to nearest prime number of total no. of elements (i.e. n). and Time Complexity for this is $O(n^{3/2})$. Also Analysed as $O(n^{5/3})$.

Code for Shell Sort :-

```
int main()
{
int A[] = {11, 13, 7, 12, 16, 9, 24, 5, 10, 3}, n=10, i;
Void ShellSort (int A[], int n);
Shell Sort (A, n);
for (i=0 ; i<10; i++)
 {
 printf ("%d", A[i]);
 }
}
```

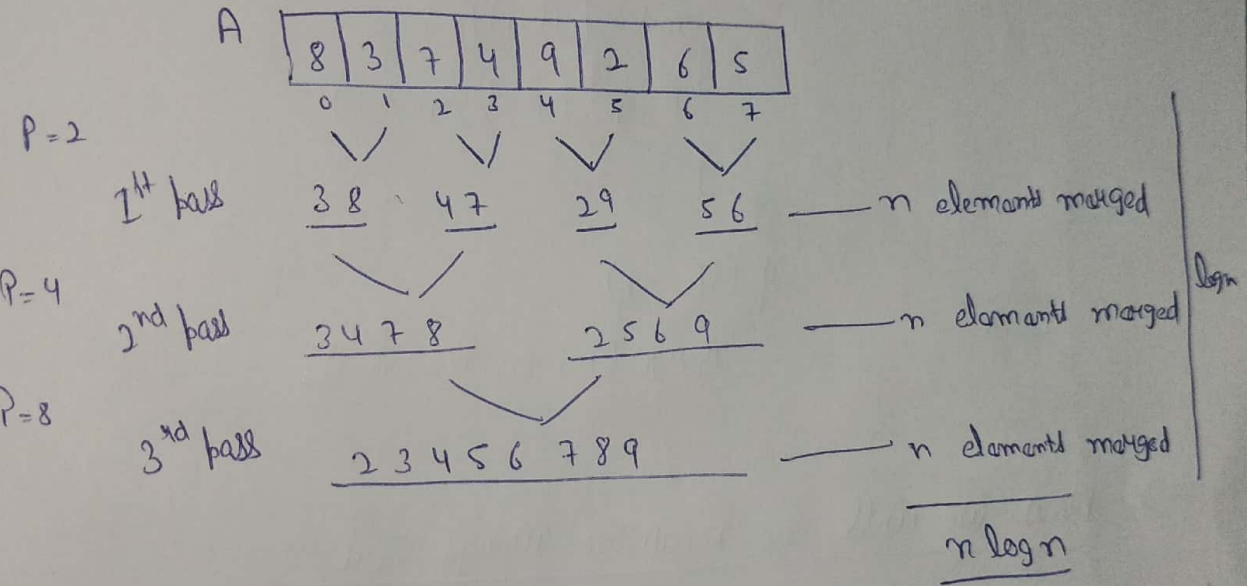
```
Void Shell Sort (int A[], int n)
{
    int gap, i, j;
    for (gap = n/2; gap >= 1; gap /= 2)
    {
        for (i = gap ; i < n ; i++)
        {
            int temp = A[i];
            j = i - gap;
            While (j >= 0 && A[j] > temp)
            {
                A[j+gap] = A[j]
            }     j = j - gap;
            A[j + gap] = temp;
        }
    }
}
```

Merging :- It is a two way Merge Sort.

- Done in Iterative Version
- Done in Recursive Version

Iterative Merge Sort :-

Let us Suppose that Every Element in an Array is itself a list, Now we will perform merging on two lists a time so, it will be like this



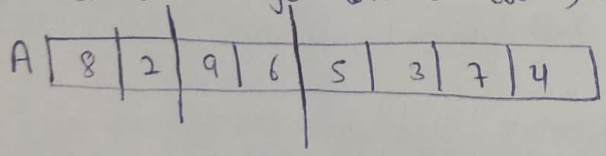
Now, if we see it is just forming a tree, so it will be (log n) passes performed and 'n' elements will be merged in any pass. So the Time Complexity will O(n log n) for this iterative merge sort.

Program :-

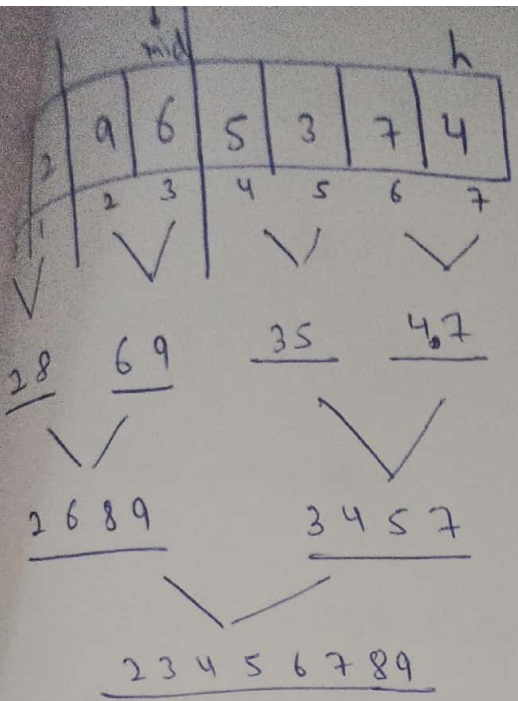
```
Void IMergeSort (int A[], int n)
{
    int P, i, l, mid, h;
    for (P=2; P<=n; P=P*2)
    {
        for (i=0; i+P-1<n; i=i+P)
        {
            l=i;
            h=i+P-1;
            mid = ⌊(l+h)/2⌋;
            Merge (A, l, mid, h);
        }
    }
    if (P/2<n)
        Merge (A, 0, P/2, n-1);
}
```

⇒ Now we will see Recursive Merge Sort,

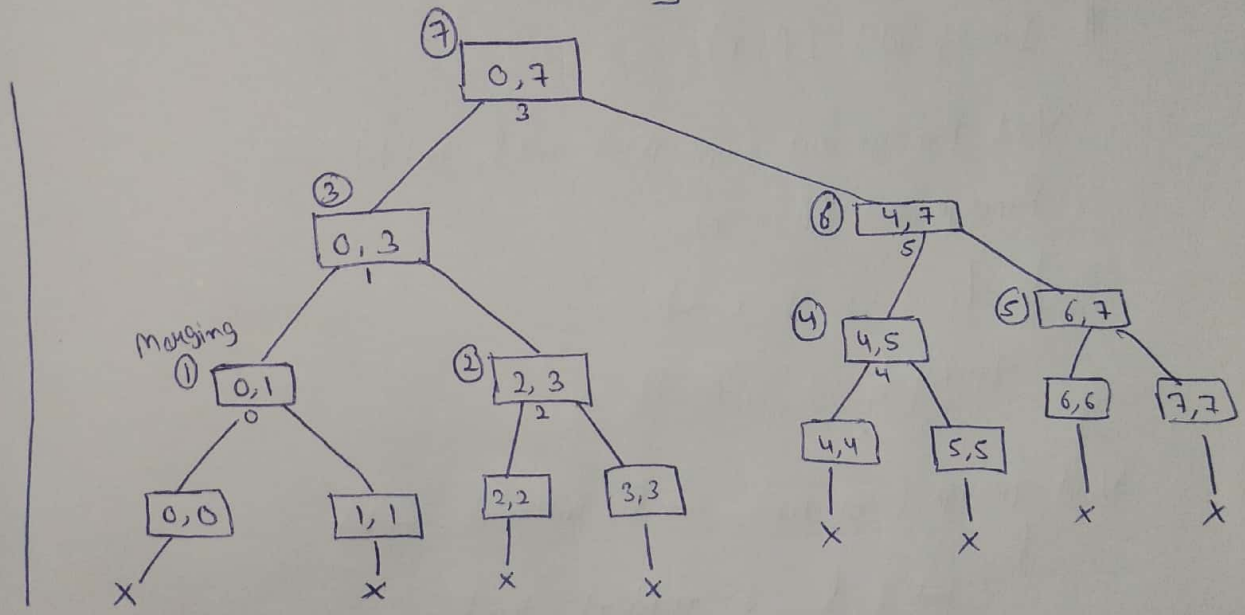
Idea Behind Recursive Merge sort is that, suppose we have list of elements



Now, Divide it from half and it is not guaranteed that either left or right list is sorted or not. then again half and continue until we reach one-one element, bcz one-one element is itself sorted, so this is the idea behind merge sort. and Merging of two lists will be done in returning phase.




```
Void R MergeSort (int A[], int l, int h)
{
    if (l<h)
    {
      1.) mid = ⌊(l+h)/2⌋;
      2.) R Merge sort (A, l, mid);
      3.) R Merge Sort (A, mid+1, h);
      4.) Merge (A, l, mid, h);
    }
}
```



See, we are doing Merging, and we are perform Merging on 8 elements at all levels and doing it for (n+n+n) times depending upon the height and it is just like a full Binary tree / Completely tree (log n)

So, Time Complexity :- O(n log n)

Note :- "We can see that the Merging done in left to right then to top, it is just like Postorder, so yes Merging is done in Postorder."

Space Complexity :- A Array take → O(n)
Auxilary Array used in Merge - O(n)
And it is recursive it uses stack
So, for it O(log n)

→ Total = 2n + log n
So, Merge Sort is only algorithm in Comparison based which returns extra space.

Code for Merge Sort (Iterative and Recursive) :-

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int A[] = {11, 13, 7, 12, 16, 9, 24, 5, 10, 3, 45}, n=11, i;
  // Void Imerge Sort (int A[], int n);
  // Imerge Sort (A, n);
   Void Rmerge Sort (int A[], int l, int h);
   Rmerge Sort (A, 0, n);
   for (i=0 ; i<11 ; i++)
     printf ("%d", A[i]);
}
Void Merge (int A[], int l, int mid, int h)
   {
      int i=l , j = mid+1 , k=l ;
      int b[100];
      while ( i<=mid  && j<=h)
       { if (A[i] < A[j])
           b[k++] = A[i++];
         else
       }  b[k++] = A[j++];
        for ( ; i<=mid ; i++)
           b[k++] = A[i];
```

```
        for (; j<=h ; j++)
            b[k++] = A[j];
        for (i=l ; i<=h ; i++)
    }       A[i] = b[i];

Void IMergeSort (int A[], int n)   // Iterative Procedure
{
    int P, l, mid, i, h;
    for (P=2 ; P<=n ; P=P*2)
    {
        for (i=0 ; i+P-1 < n ; i = i+P)
        {
            l=i;
            h= i + P-1;
            mid= floor((l+h)/2);
            Merge (A, l, mid, h);
        }
    }
    if (P/2 < n)
        Merge (A, 0, P/2-1, n);
}

Void RmergeSort (int A[], int l, int h)   // Recursive Procedure.
{
    if (l<h)
    {
        int mid = floor((l+h)/2);
        Rmerge Sort (A, l, mid);
        Rmerge Sort (A, mid+1, h);
        Merge (A, l, mid, h);
    }
}
```

Asymptotic Notations :-

See we have seen various Algorithms on data structures and we got Time Complexities as follows :

We have Time Complexities in increasing order, here,

$1 < \log n < n < n \log n < n^2 < n^3 \text{ --- } < 2^n < 3^n < n^n$

- $1 < \log n < n < n \log n < n^2 < n^3$ — Polynomial Time Complexities.
- $2^n < 3^n < n^n$ — exponential Time Complexities

Now, Sometimes from function, we can analyse time Complexity

$f(n) = n \longrightarrow O(n)$

$f(n) = n^2 \longrightarrow O(n^2)$

$f(n) = \sum_{i=1}^{n} i \longrightarrow 1+2+3+\cdots+n \longrightarrow \frac{n(n+1)}{2} \longrightarrow O(n^2)$

Now, Suppose time function is like this $f(n) = \sum_{i=1}^{n} i * 2^i$

We can't get exact formula by expanding, We can get approx. formula

So, when we can't get exact formula, We will never predict time Complexity We can get predict whether Lower value or higher value;

So, if we are taking Lower value $\longrightarrow \Omega(2^n)$ [omega → Lower bound]

if we are taking higher value $\longrightarrow O(n2^n)$ [Big O → Upper bound]

if we are taking exact value $\longrightarrow \Theta(n2^n)$ [theta → Tight bound]

Hence, when we use exact function, (Say Θ), otherwise (Say O) bcz many of times we will take atmost value (i.e. upper bound).

Conclusion :- We always said order of / theta in our full Course bcz We get Time Complexities as exact values of n (i.e. Tight bound)

"COURSE"

"FINISHED"

"CONGRATS PRINCE"