Algorithms & Data Structure

Kiran Waghmare

Algorithm:

-Design
-Domain Knowledge
-Any Language
-H/w & S/w
-Analyze

Priori Analysis

-algorithm
-Independent of Platform
-Independent of H/w
-Time & Space

Program:

-Implementation
-Programmer
-Programming Language
-H/w & S/w
-Testing

Posterior Analysis

-program
-Dependent on platform
-Dependent on H/w
-Time

Ex: Swapping of 2 values:

Time

Space

```
swap(a,b)
{
    temp = a;
    a = b;
    b = temp;
}
```

1
1
1

a → 1
b → 1
temp → 1

f(n) = 3

O(1)

s(n) = 3 words

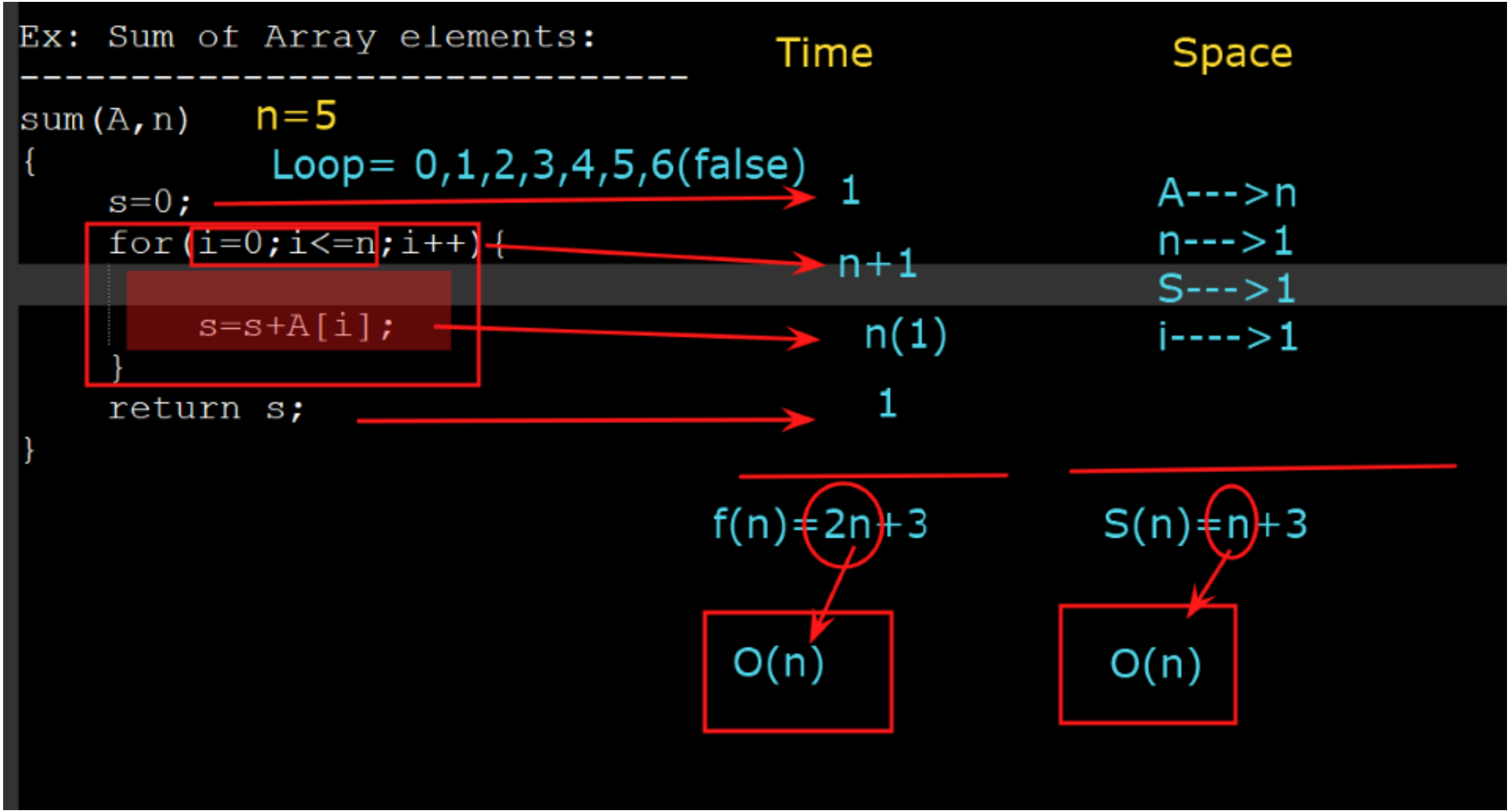
O(1)

```
x = 5*a+6*b
```

→ 1

```
x = 5*a+6*b
x = 5*a+6*b
x = 5*a+6*b
x = 5*a+6*b
x = 5*a+6*b
```

f(n) = 6-------> O(1) => constant


Ex: Sum of Array elements:
Time
Space
sum(A,n)
n=5
{
Loop= 0,1,2,3,4,5,6(false)
s=0;
1
A--->n
n--->1
for(i=0;i<=n;i++){
n+1
S--->1
s=s+A[i];
n(1)
i---->1
}
return s;
1
}
f(n)=2n+3
S(n)=n+3
O(n)
O(n)

Ex 5:

```
for(i=0;i<n;i++)
{
        stmt;
}
```

O(n)

Ex 6:

```
for(i=n;i>0;i--)
{
        stmt;
}
```

O(n)

Ex 7:

```
for(i=1;i<n;i+2)
{
        stmt;
}
```

n/2

O(n)

Ex 8:

```
for(i=1;i<n;i=i+20)
{
        stmt;
}
```

n/20

O(n)

Ex 9:

```
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
        stmt;
}
}
```

n(n)

O(n²)

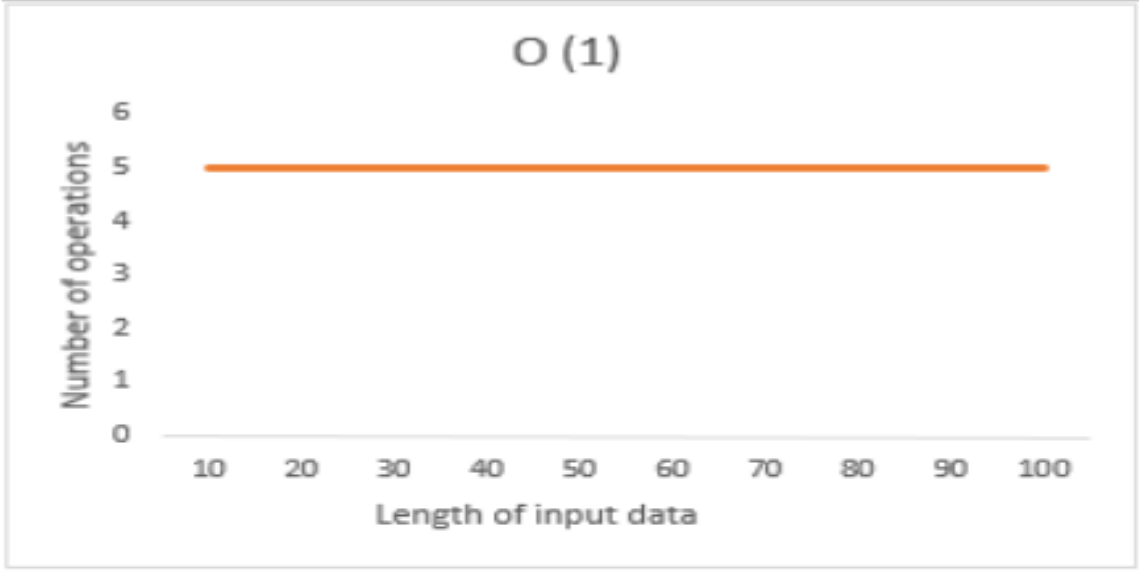
Ex 10:

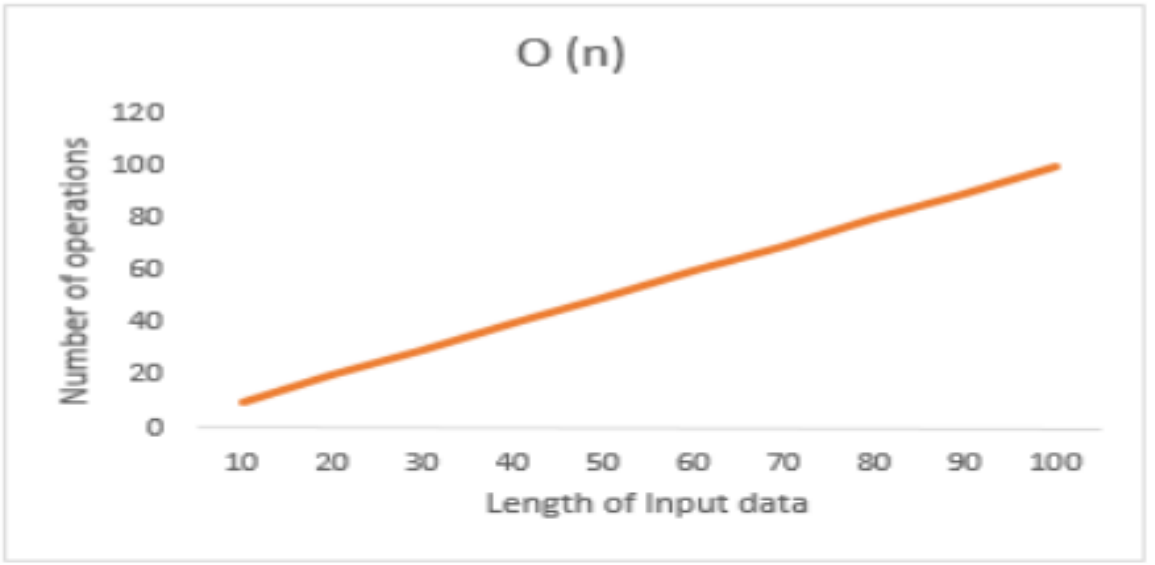
```
for(i=0;i<n;i++)
{
for(j=0;j<i;j++)
{
        stmt;
}
```

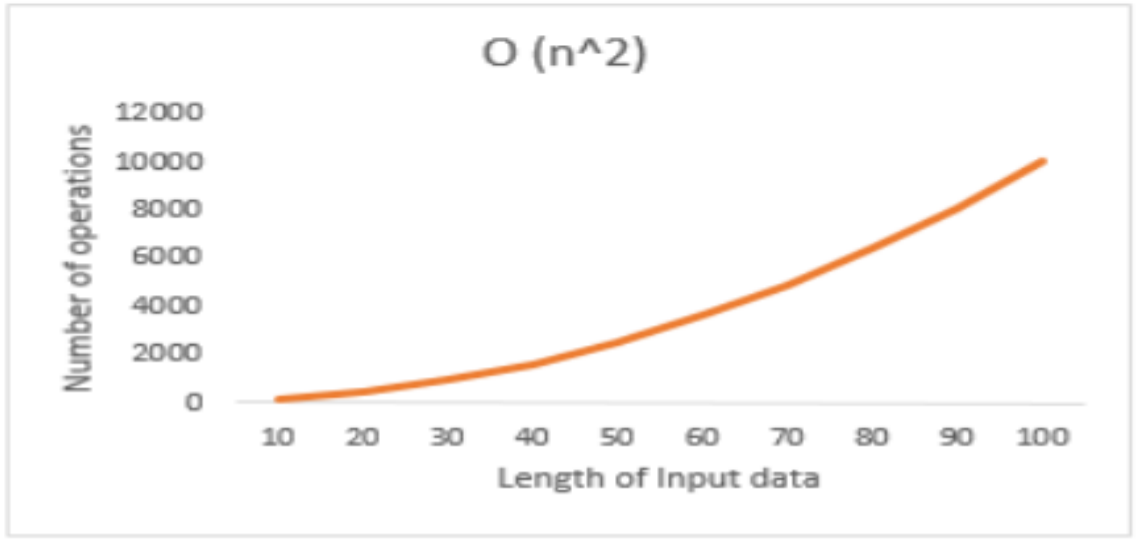
O(n²)

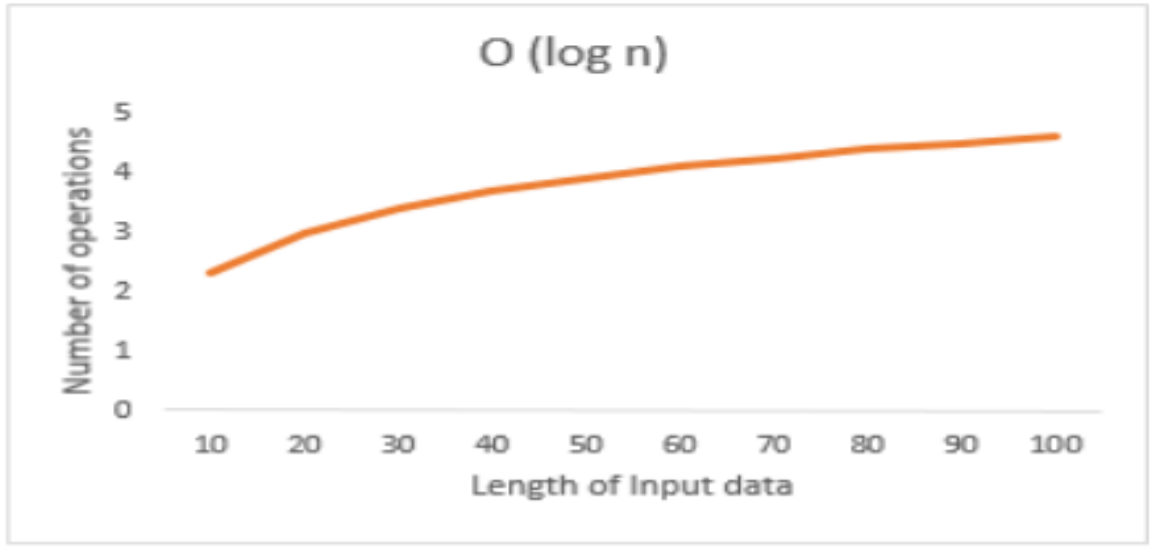
CDAC Mumbai:Kiran Waghmare

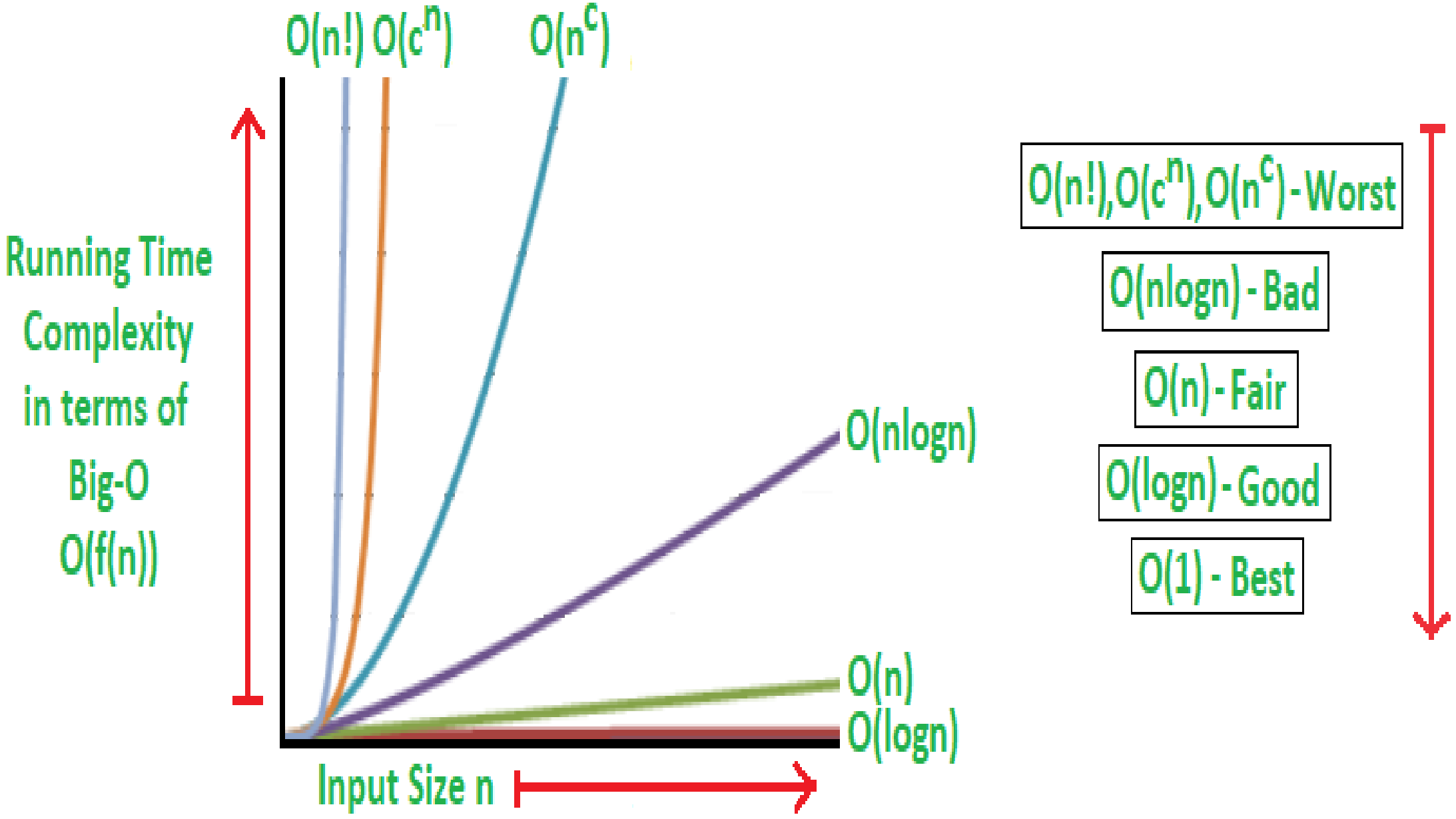
The order of growth for all time complexities are indicated in the graph below:










O(n!) O(c^n)
O(n^c)
Running Time Complexity in terms of Big-O O(f(n))
O(nlogn)
O(n)
O(logn)
Input Size n
O(n!),O(c^n),O(n^c) - Worst
O(nlogn) - Bad
O(n) - Fair
O(logn) - Good
O(1) - Best

Complexities of an Algorithm

The complexity of an algorithm computes the amount of time and spaces required by an algorithm for an input of size (n).

The complexity of an algorithm can be divided into two types.

The time complexity and the space complexity.

Time Complexity of an Algorithm

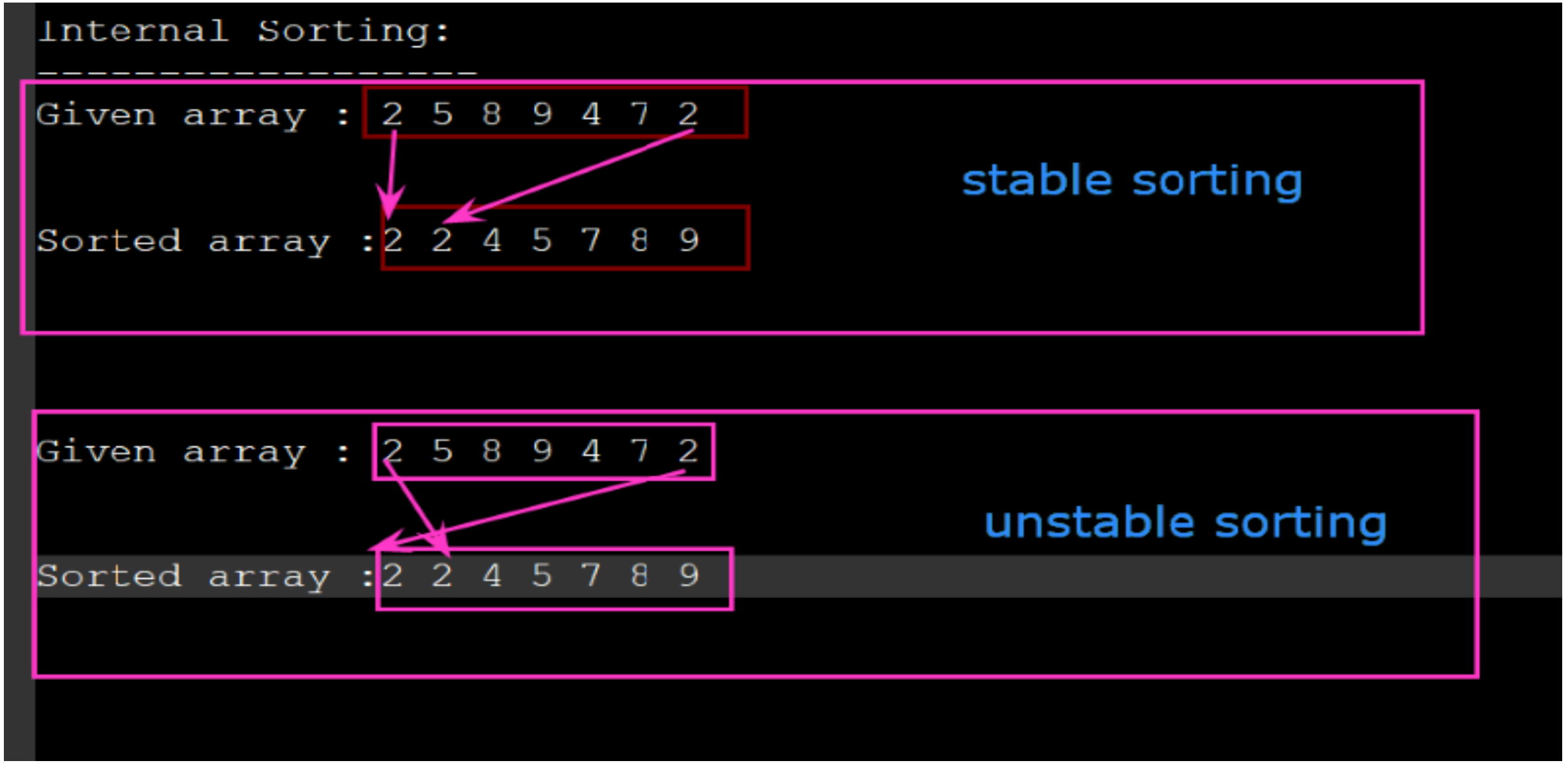
The time complexity is defined as the process of determining a formula for total time required towards the execution of that algorithm.

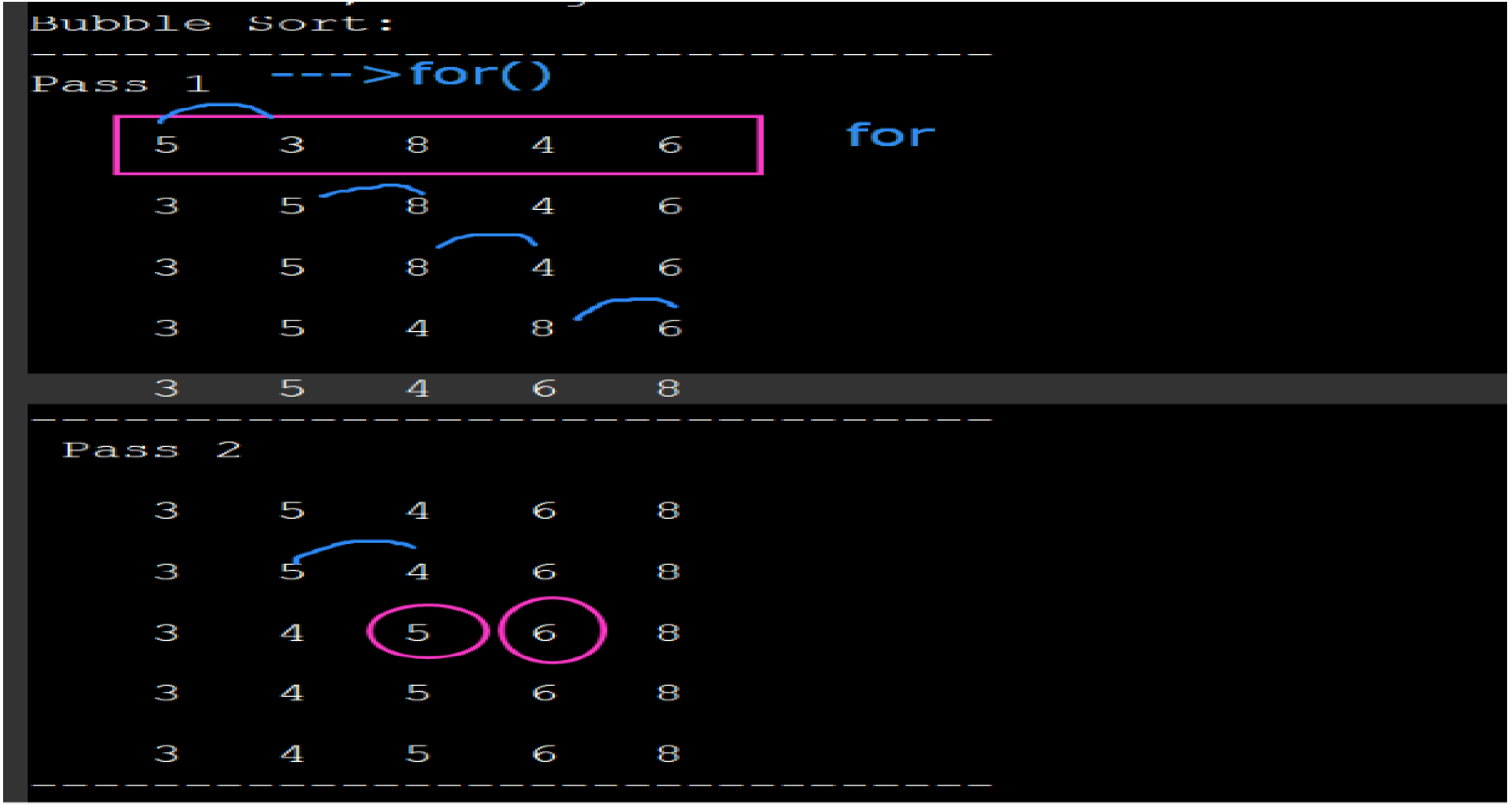
This calculation is totally independent of implementation and programming language.

Space Complexity of an Algorithm

Space complexity is defining as the process of defining a formula for prediction of how much memory space is required for the successful execution of the algorithm.

The memory space is generally considered as the primary memory.


Internal Sorting:
Given array : 2 5 8 9 4 7 2
Sorted array :2 2 4 5 7 8 9
stable sorting
Given array : 2 5 8 9 4 7 2
Sorted array :2 2 4 5 7 8 9
unstable sorting


Bubble Sort:
Pass 1 --->for()
5 3 8 4 6
for
3 5 8 4 6
3 5 8 4 6
3 5 4 8 6
3 5 4 6 8
Pass 2
3 5 4 6 8
3 5 4 6 8
3 4 5 6 8
3 4 5 6 8
3 4 5 6 8

```java
class Sorting
{
    void bubbleSort(int a1[])
{
    int n = a1.length;
    for(int i=0;i<n-1;i++)
    {
        for(int j=0;j<n-i-1;j++)
        {
            if(a1[j] > a1[j+1])
            {
                int temp = a1[j];
                a1[j]=a1[j+1];
                a1[j+1]=temp;
            }
        }

    }
}
    void display(int a1[])
    {
        int n = a1.length;
        for(int i=0;i<n;i++)
        {
            System.out.print(a1[i]+" ");
        }
    }
```

Best case:
Worst case: → O(n^2)
Average case:

No of comparision: n-1

Space Complexity:O(n)

```java
void selectionsort(int a1[])
{
    int n = a1.length;
    for(int i=0;i<=n-1;i++)
    {
        int min=i;
        for(int j=i+1;j<n;j++)
        {
            if(a1[j] < a1[min])
                min = j;
        }
        //swapping
                int temp = a1[min];
                a1[min] = a1[i];
                a1[i]=temp;



    }

}
    void display(int a1[])
    {
        int n = a1.length;
        for(int i=0;i<n;i++)
        {
```

Best case:
Worst case:
Average case:
O(n^2)

Space Complexity:O(n)

O(1)

```
void insertionsort(int a1[])
{
    int n = a1.length;
    for(int i=1;i<n;i++)
    {
        int k = a1[i];
        int j = i-1;

        while(j>=0 && a[j]>k)
        {
            a1[j+1]=a1[j];
            j=j-1;

        }
        a1[j+1]=k;
    }
}

//Best case
|1 2 3 4 5
1| 2 3 4 5
1 2 | 3 4 5
1 2 3 | 4 5
```

Best case: → O(n)

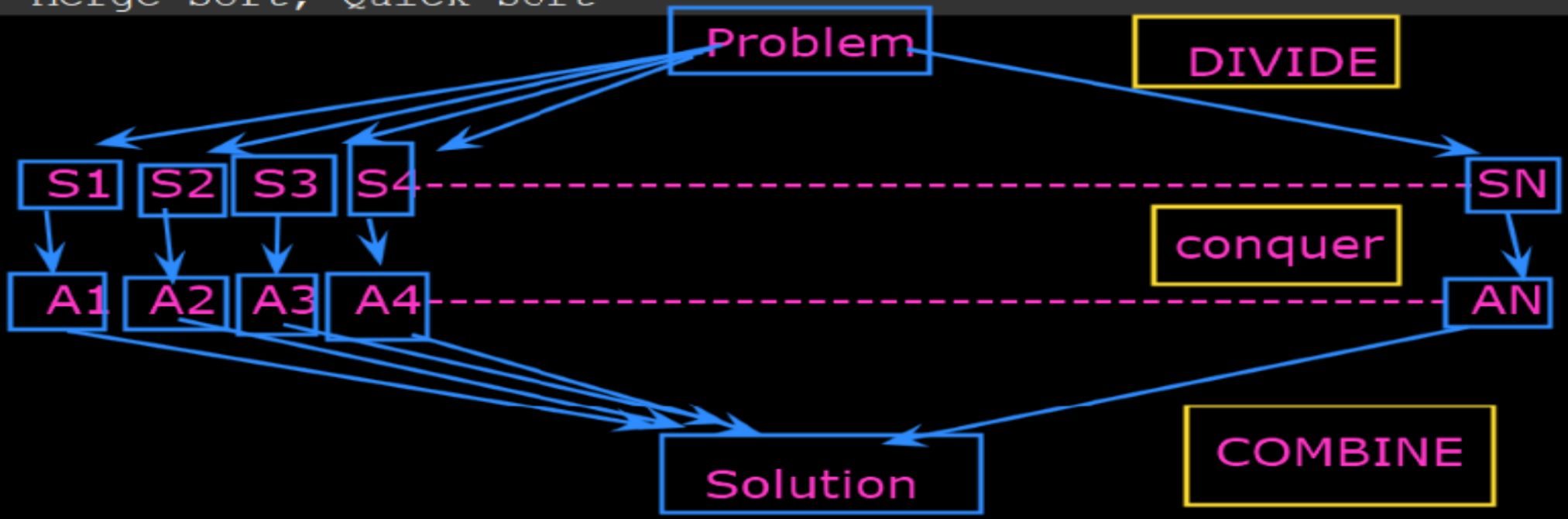
Worst case: → O(n^2)

Average case: → O(n^2)

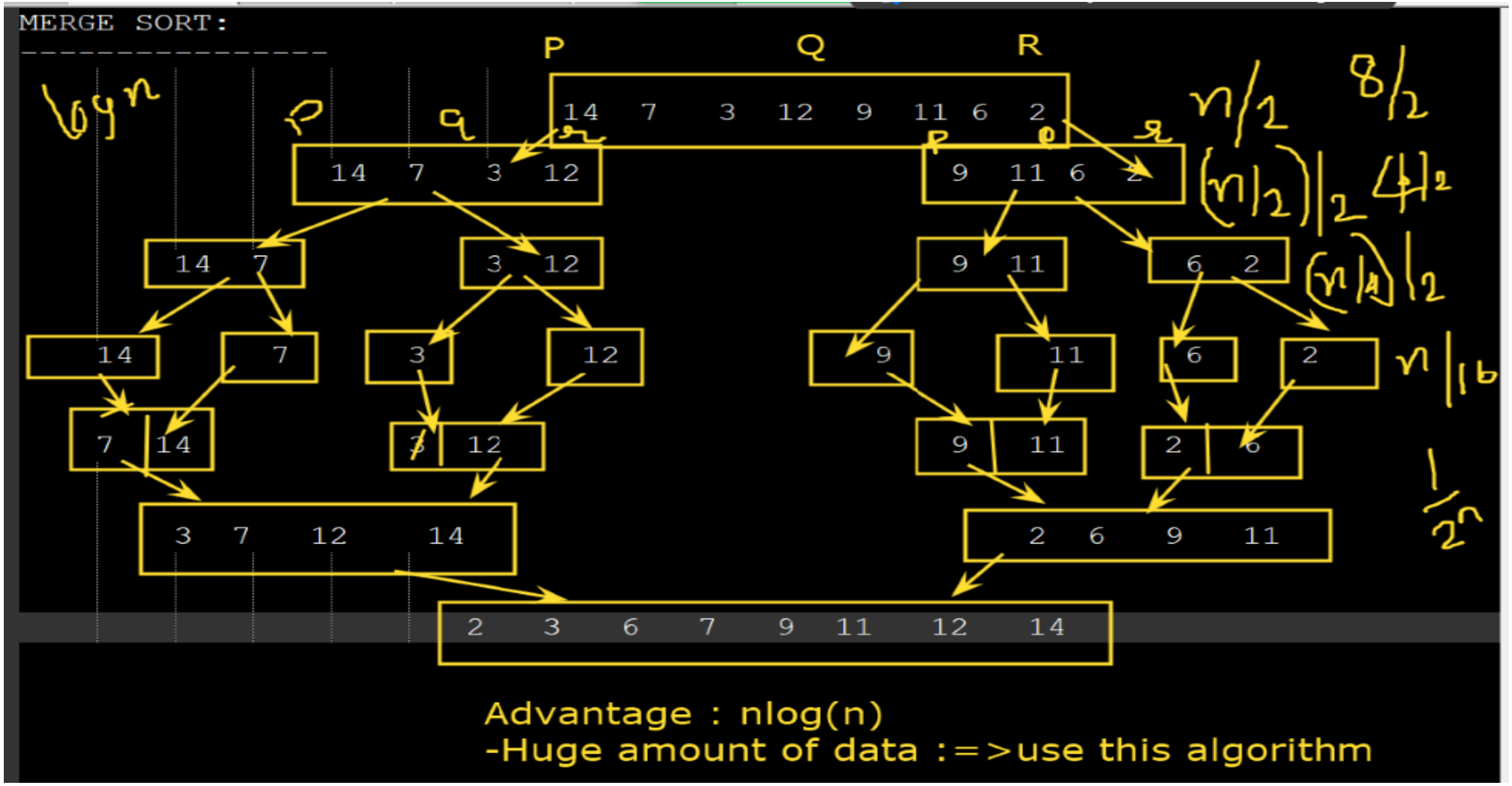
-purpose: tohandle the worst case running
-Ex: Merge Sort, Quick Sort

Mouse Select Text Draw Stamp Spotlight Eraser Format Undo Redo

Who can see what you share here? Recording On



DIVIDE AND CONQUER ALGORITHMS


MERGE SORT:
P
Q
R
14 7 3 12 9 11 6 2
14 7 3 12
9 11 6 2
14 7
3 12
9 11
6 2
14
7
3
12
9
11
6
2
7 14
3 12
9 11
2 6
3 7 12 14
2 3 6 7 9 11 12 14
Advantage : nlog(n)
-Huge amount of data :=>use this algorithm

Thanks