Sep22 : Day 3

Kiran Waghmare

CDAC Mumbai

Algorithm:

- -Design
- -Domain knowledge
- -Language
- -Hardware/software
- -Analysis

Program:

- -Implementation
- -Programmer
- -Programming Language
- -Hardware/software
- -Testing

Prior Analysis

- -Algorithm
- -Platform Independent
- -Hardware Independent
- -Time and Space

Posterior Analysis

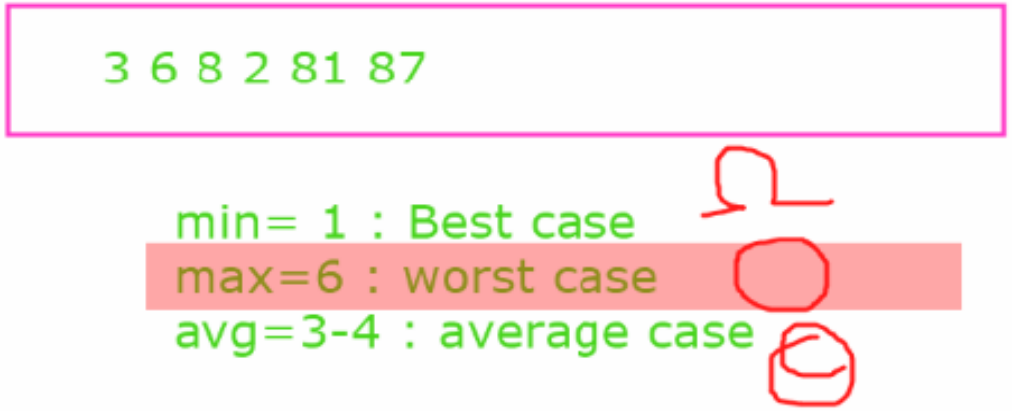
- -Program
- -Platform Dependent
- -Hardware Dependent
- -Time

Asymptotic Notations:

Asyptotic analysis of algorithm refers to defining the mathemaetical boundingof it's runtime

Types od Asymptotic notations:

1. Best case: minimum time required for program execution.
2. Average case:average time required for program execution.
3. Worst case:maximum time required for program execution.



O(n), O(n^2)

Algorithm Complexity:-

Two main factors, which decide the efficiency of X.

- **Time Factor** – Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** – Space is measured by counting the maximum memory space required by the algorithm.

Asymptotic Notations

Asymptotic analysis of an algorithm refers to defining the mathematical boundation of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Usually, the time required by an algorithm falls under three types –

- **Best Case** – Minimum time required for program execution.
- **Average Case** – Average time required for program execution.
- **Worst Case** – Maximum time required for program execution.

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- **O Notation**
- **Ω Notation**
- **θ Notation**

How to write Algorithm:

Ex 1: Algorithm: for swapping of 2 numbers

Time

Space

```
swap(a,b)
{
        temp = a;        ------------> 1
        a=b;             ------------> 1
        b=temp;          ------------> 1
}
```

a---------> 1word
b---------> 1
temp-----> 1

x= 5*a+6*b -----> 1sec
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) = 6sec ===> O(1)

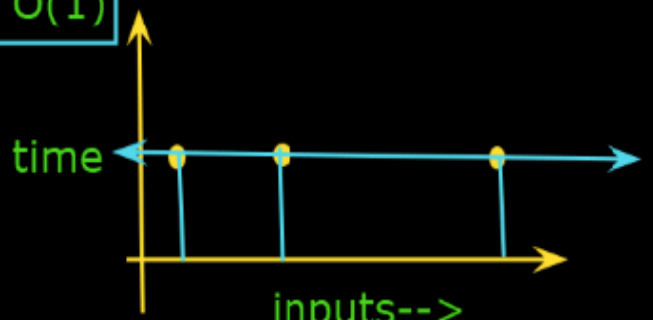
Constant complexity

f(n) = 3

O(1)

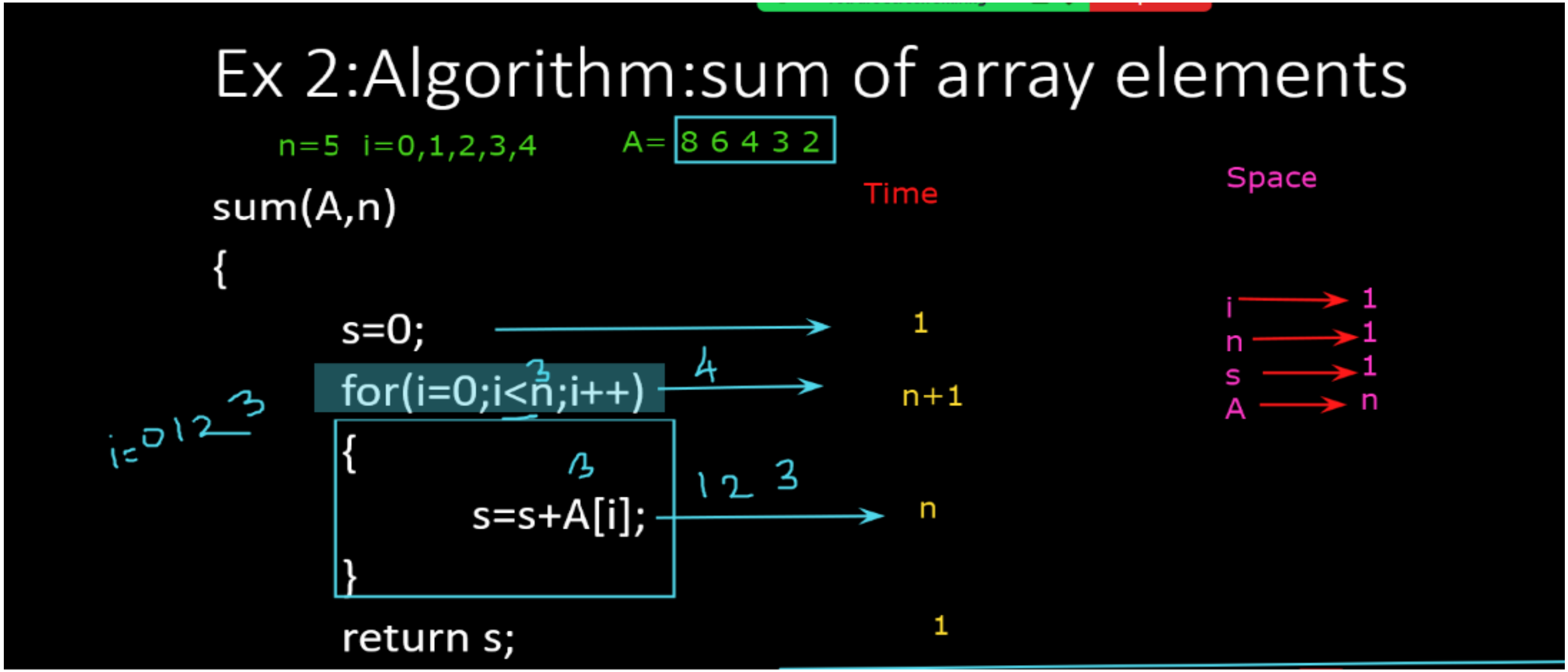
s(n) = 3 words

O(1)



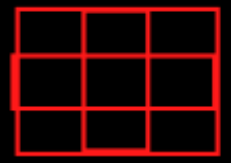
CDAC Mumbai:Kiran Waghmare

Ex 2:Algorithm:sum of array elements



Who can see what you share here? Recording On

Ex 3:Algorithm to add 2D array elements



nXn
3X3

```
Add(A,B,n)
{
    for(i=0;i<n;i++)
    {
    for(j=0;j<n;j++)
    {
    C[i,j]=A[i,j]+B[i,j];
    }
    }
}
```

Time

for(i=0;i<n;i++) → n+1

for(j=0;j<n;j++) → n (n+1)

C[i,j]=A[i,j]+B[i,j]; → n n

f(n)=2n^2+2n+1

O(n^2)

Quadratic complexity

Space

i → 1
j → 1
n → 1
A → n*n=n^2
B → n*n=n^2
C → n*n=n^2

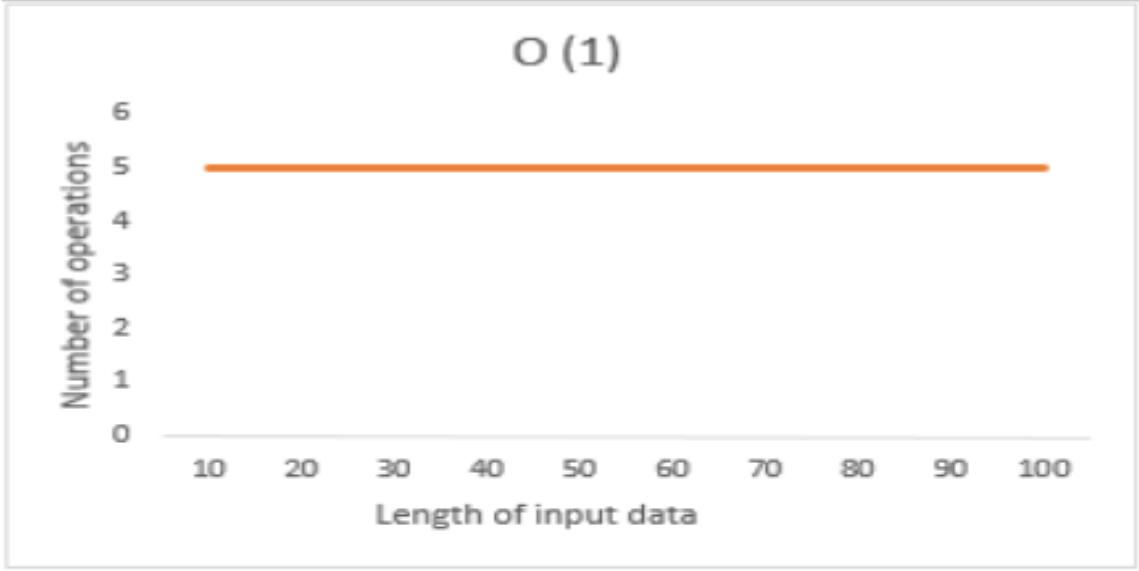
s(n)=3n^2 + 3

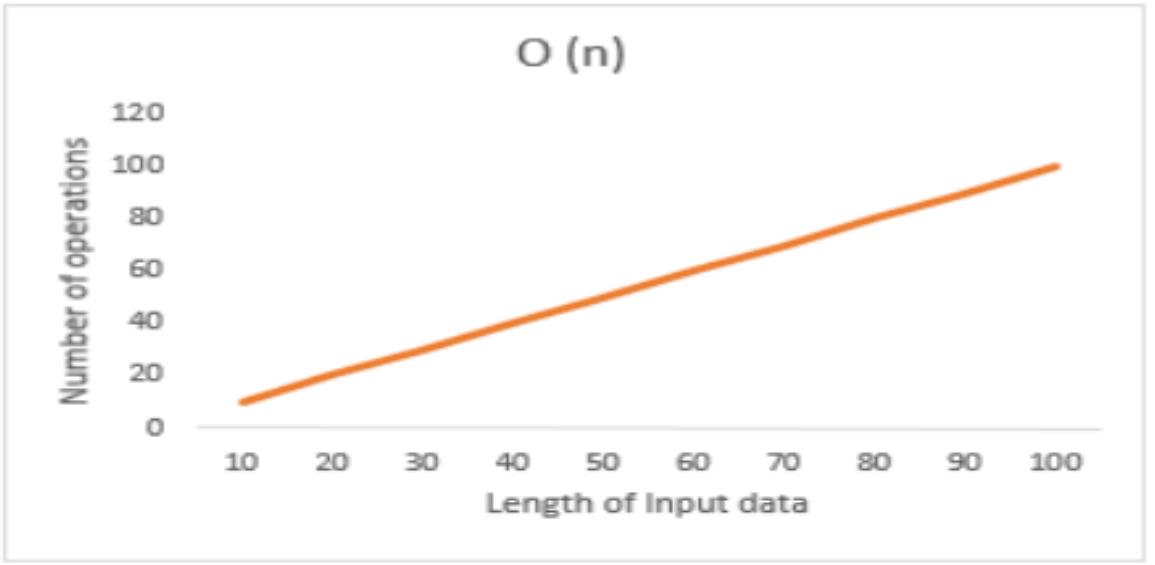
O(n^2)

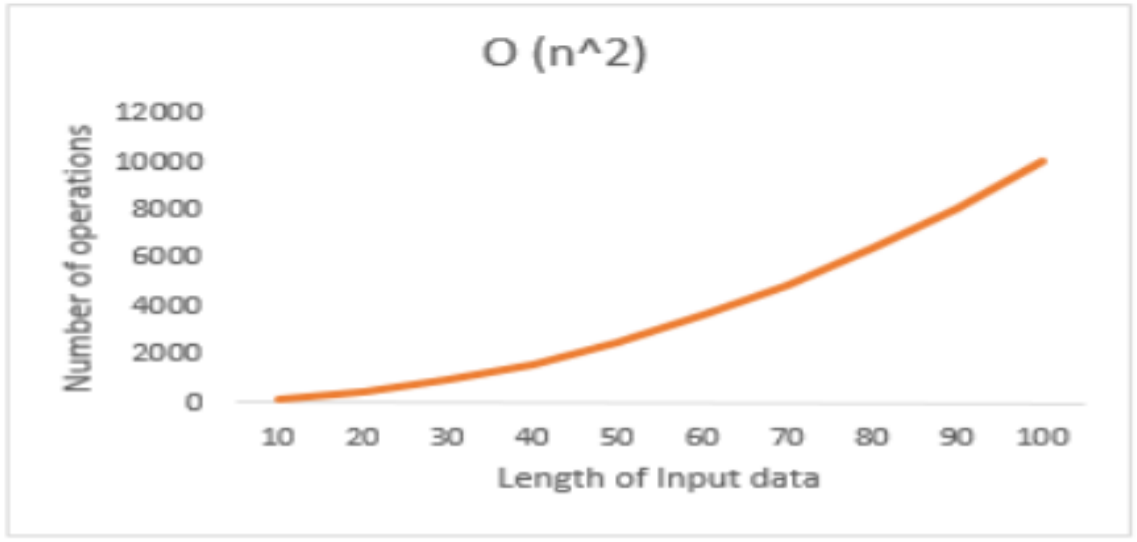
CDAC Mumbai:Kiran Waghmare

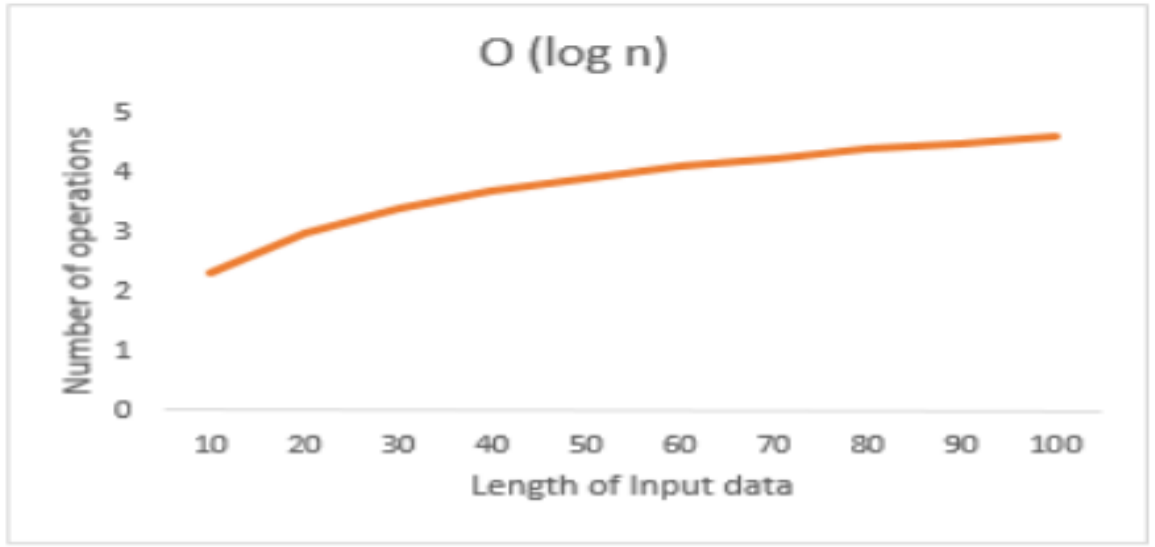
CDAC Mumbai:Kiran Waghmare

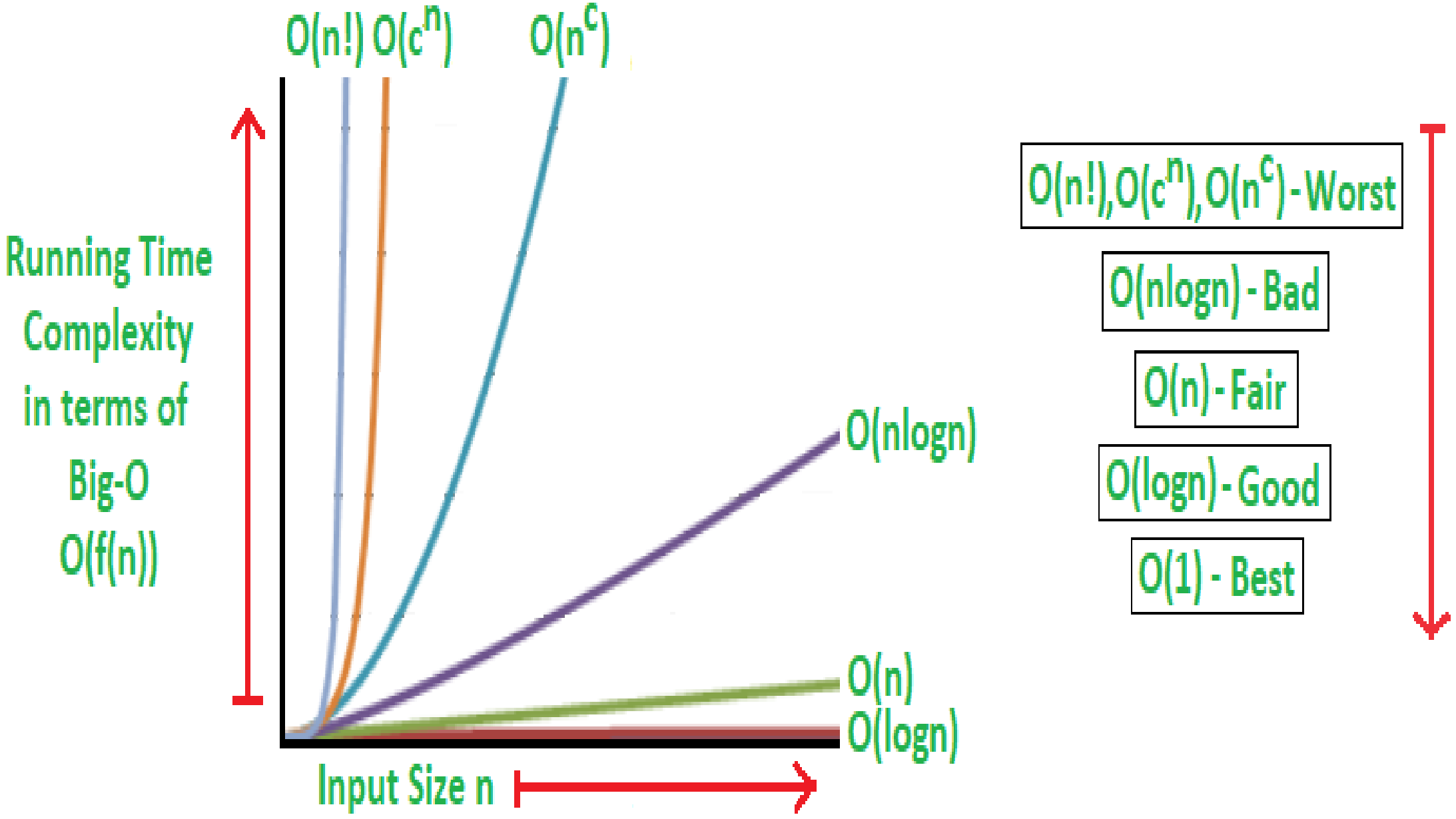
The order of growth for all time complexities are indicated in the graph below:










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

Commonly Used Functions and Their Comparison

1. **Constant Functions** - $f(n) = 1$ - Whatever is the input size n, these functions take a constant amount of time.
2. **Linear Functions** - $f(n) = n$ - These functions grow linearly with the input size n.
3. **Quadratic Functions** - $f(n) = n^2$ - These functions grow faster than the superlinear functions i.e., $n \log(n)$.
4. **Cubic Functions** - $f(n) = n^3$ - Faster growing than quadratic but slower than exponential.
5. **Logarithmic Functions** - $f(n) = \log(n)$ - These are slower growing than even linear functions.
6. **Superlinear Functions** - $f(n) = n \log(n)$ - Faster growing than linear but slower than quadratic.
7. **Exponential Functions** - $f(n) = c^n$ - Faster than all of the functions mentioned here except the factorial functions.
8. **Factorial Functions** - $f(n) = n!$ - Fastest growing than all these functions mentioned here.

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.

Searching in Arrays

- **Searching:** It is used to find out the location of the data item if it exists in the given collection of data items.
  E.g. We have linear array A as below:

1	2	3	4	5
15	50	35	20	25

Suppose item to be searched is 20. We will start from beginning and will compare 20 with each element. This process will continue until element is found or array is finished. Here:

1) Compare 20 with 15
   20 # 15, go to next element.
2) Compare 20 with 50
   20 # 50, go to next element.

3) Compare 20 with 35
   20 #35, go to next element.

4) Compare 20 with 20
   20 = 20, so 20 is found and its location is 4.

Linear Search

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Problem statement

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

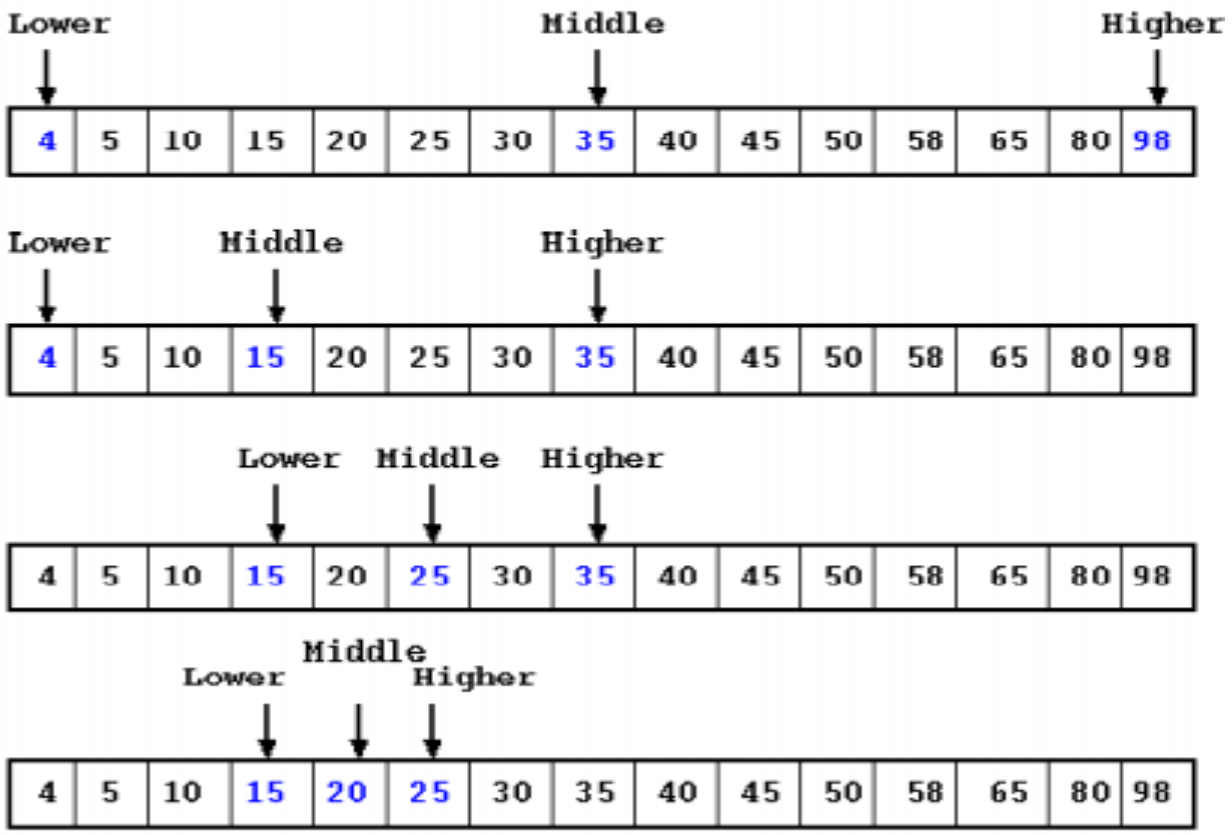
Examples :

Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 110;
Output : 6
Element x is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50,
110, 100, 130, 170}
x = 175;
Output : -1
Element x is not present in arr[].

Binary Search

- The binary search algorithm can be used with only sorted list of elements.
- Binary Search first divides a large array into two smaller sub-arrays and then recursively operate the sub-arrays.
- Binary Search basically reduces the search space to half at each step

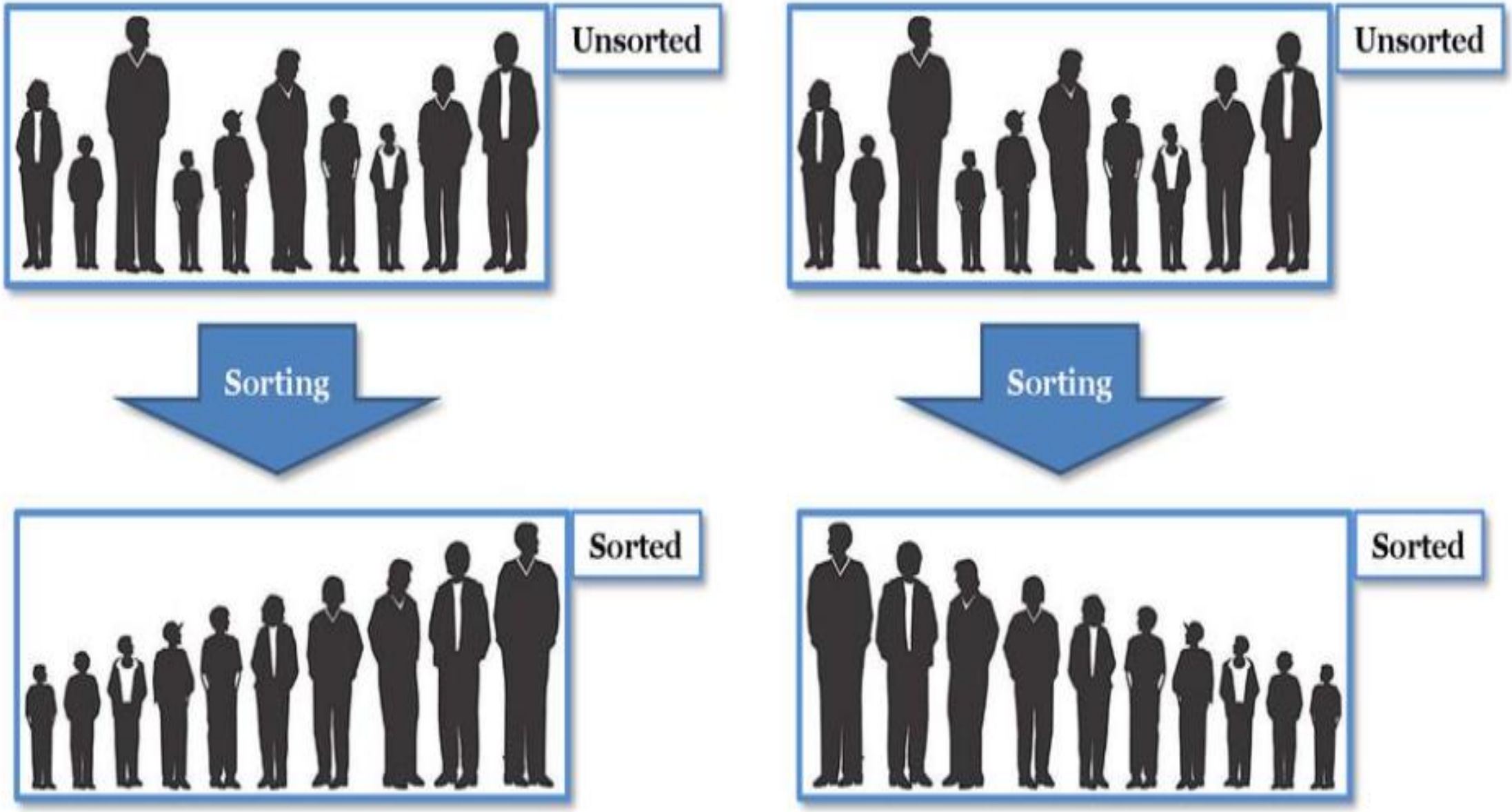


Problem statement

Move all negative numbers to beginning and positive to end with constant extra space

An array contains both positive and negative numbers in random order. Rearrange the array elements so that all negative numbers appear before all positive numbers.
Examples :

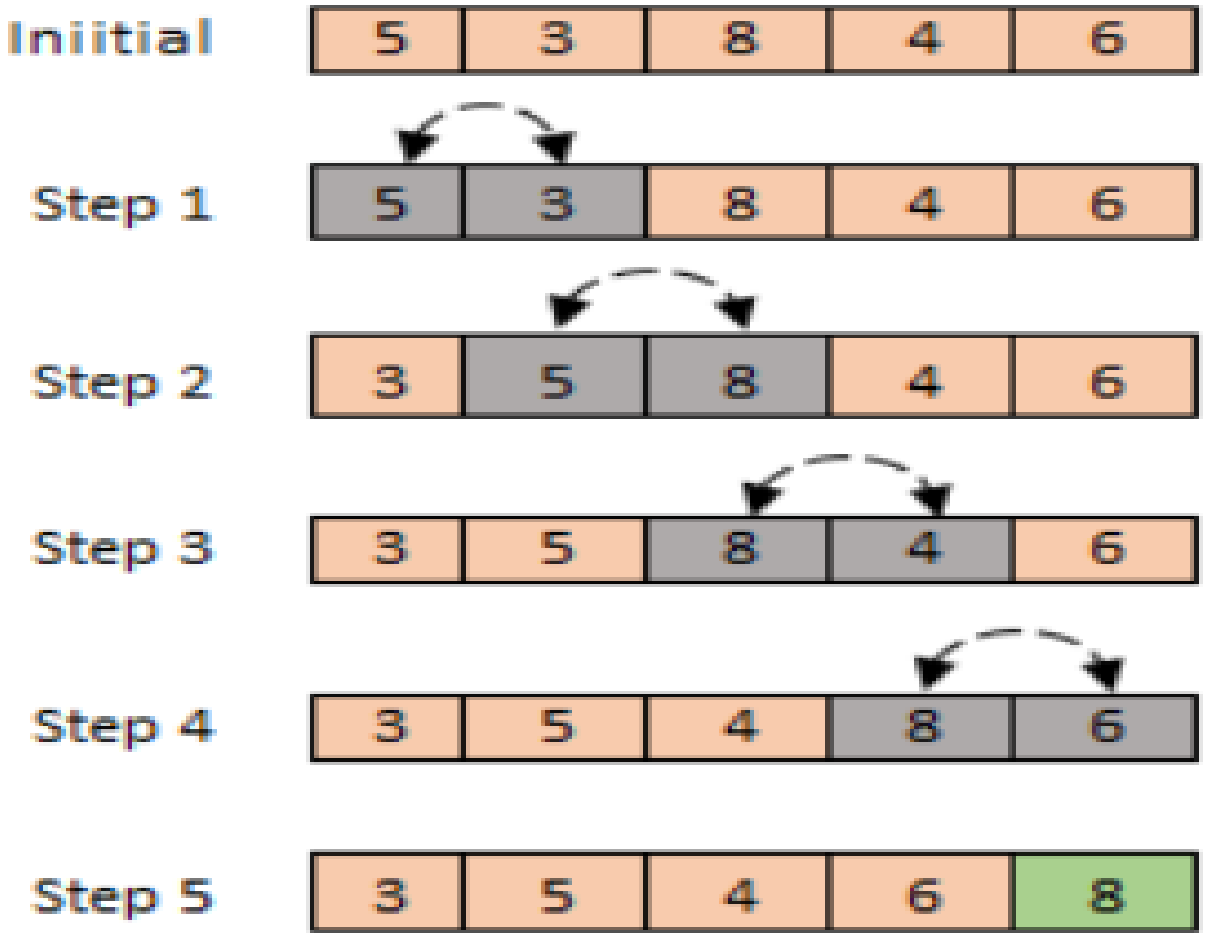
Input: -12, 11, -13, -5, 6, -7, 5, -3, -6
Output: -12 -13 -5 -7 -3 -6 11 6 5


Unsorted
Unsorted
Sorting
Sorting
Sorted
Sorted

BUBBLE SORT

- In bubble sort, each element is compared with its adjacent element.
- We begin with the 0^{th} element and compare it with the 1^{st} element.
- If it is found to be greater than the 1^{st} element, then they are interchanged.
- In this way all the elements are compared (excluding last) with their next element and are interchanged if required
- On completing the first iteration, largest element gets placed at the last position. Similarly in second iteration second largest element gets placed at the second last position and so on.

Bubble sort example



Algorithm 1: Bubble sort

```
Data: Input array A[]
Result: Sorted A[]
int i, j, k;
N = length(A);
for j = 1 to N do
    for i = 0 to N-1 do
        if A[i] > A[i+1] then
            temp = A[i];
            A[i] = A[i+1];
            A[i+1] = temp;
        end
    end
end
```

TIME COMPLEXITY

- The time complexity for bubble sort is calculated in terms of the number of comparisons f(n) (or of number of loops)

- Here two loops(outer loop and inner loop) iterates(or repeated)the comparison.

- The inner loop is iterated one less than the number of elements in the list (i.e., n-1 times) and is reiterated upon every iteration of the outer loop