



DATA STRUCTURES AND ALGORITHMS

Sep22 : Day 2

Kiran Waghmare
CDAC Mumbai

Date : 02/03/2023

Day 2 : Python ADS

Complexity:Analysis of Algorithms

-Priori analysis

- Algorithms
- Independent Programming language
- Independent Hardware dependent
- Space, Time

-Posterior analysis

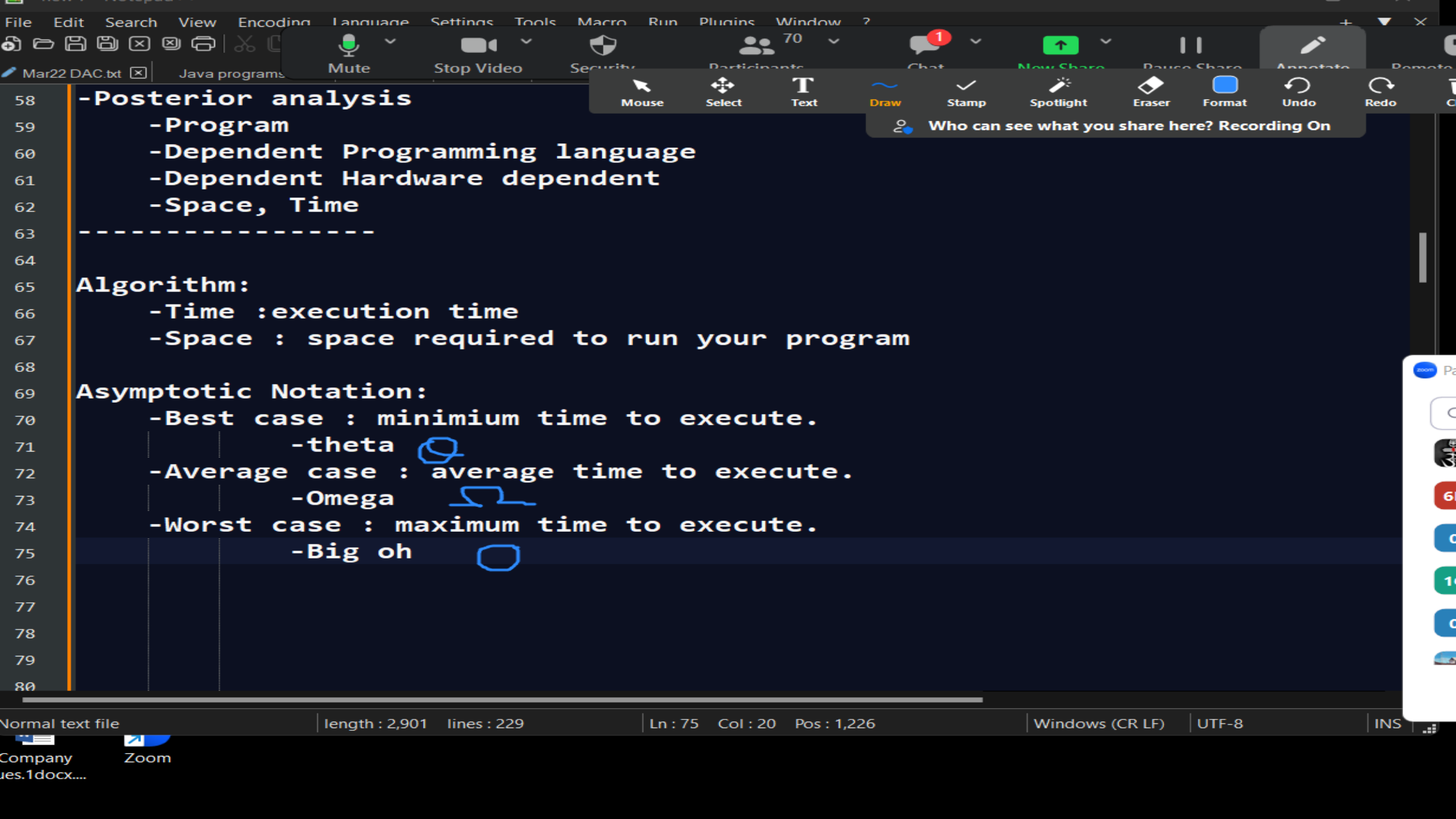
- Program
- Dependent Programming language
- Dependent Hardware dependent
- Space, Time

Algorithm:

- Time : execution time
- Space : space required to run your program

Asymptotic Notation:

- Best case : minimum time to execute.
 - theta
- Average case : average time to execute.
 - Omega
- Worst case : maximum time to execute. (Very important)
 - Big oh (O)



-Posterior analysis

- Program
- Dependent Programming language
- Dependent Hardware dependent
- Space, Time

Algorithm:

- Time :execution time
- Space : space required to run your program

Asymptotic Notation:

- Best case : minimum time to execute.
 - theta Θ
- Average case : average time to execute.
 - Omega Ω
- Worst case : maximum time to execute.
 - Big oh O

How to write Algorithm:

Ex 1: Algorithm: for swapping of 2 numbers

def swap(a,b):

temp = a; → 1

a=b; → 1

b=temp; → 1

Time

Space

a → 1

b → 1

temp → 1

$f(n) = 3$

$O(1)$

$s(n) = 3 \text{ words}$

$O(1)$

$x = 5*a + b$ -----> 1sec

$x = 5*a + b$

$x = 5*a + b$

$x = 5*a + b$

Constant complexity

$f(n) = 4$ -----> $O(1)$

Ex 2: Algorithm: sum of array elements

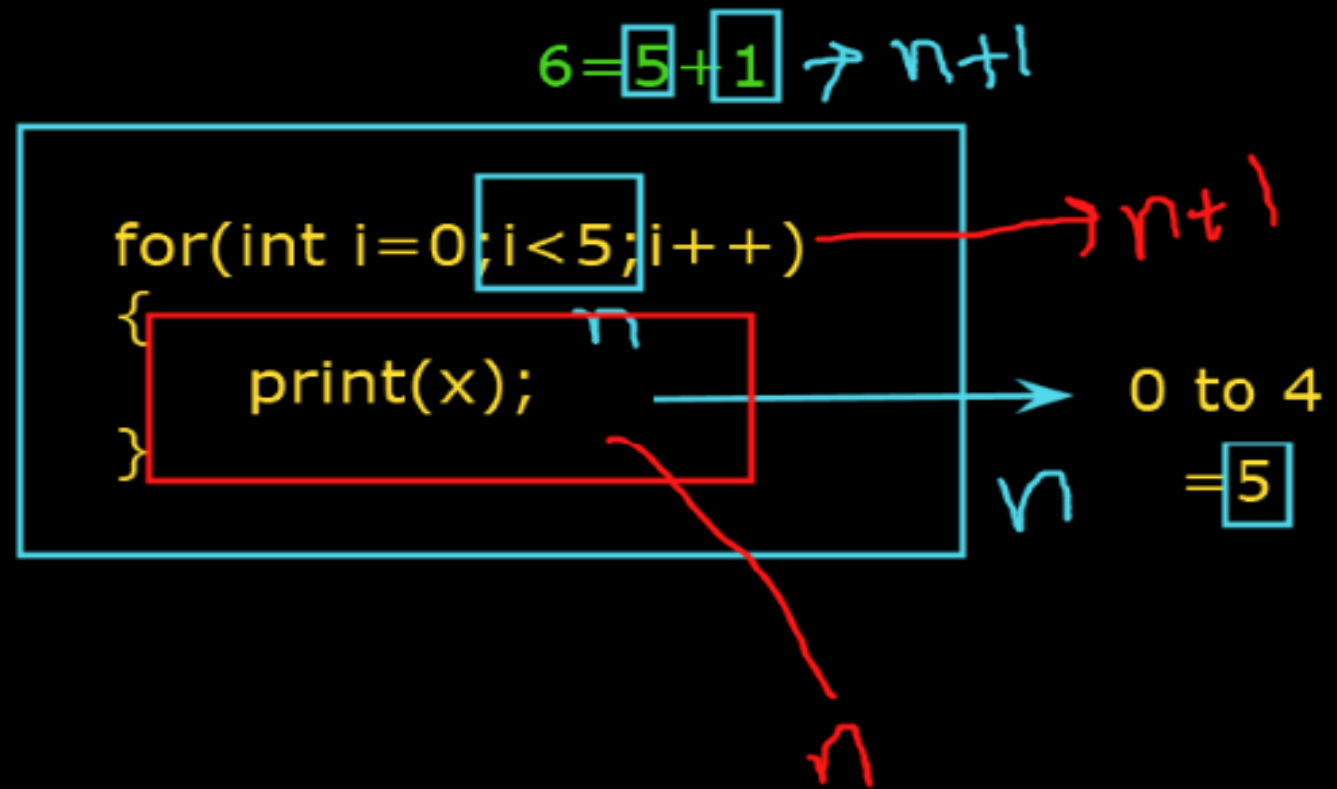
```
def sum(A,n):
```

```
    s=0
```

```
    for x in range(n):
```

```
        s=s+A[i]
```

```
    return s
```



Ex 2: Algorithm: sum of array elements

```
def sum(A,n):
```

```
    s=0
```

```
    for x in range(n):
```

```
        s=s+A[i]
```

```
    return s
```

Time

Space

1

n+1

n

1

A[n]---->n

x----->1

s----->1

n----->1

$$f(n) = \cancel{2n+3}$$

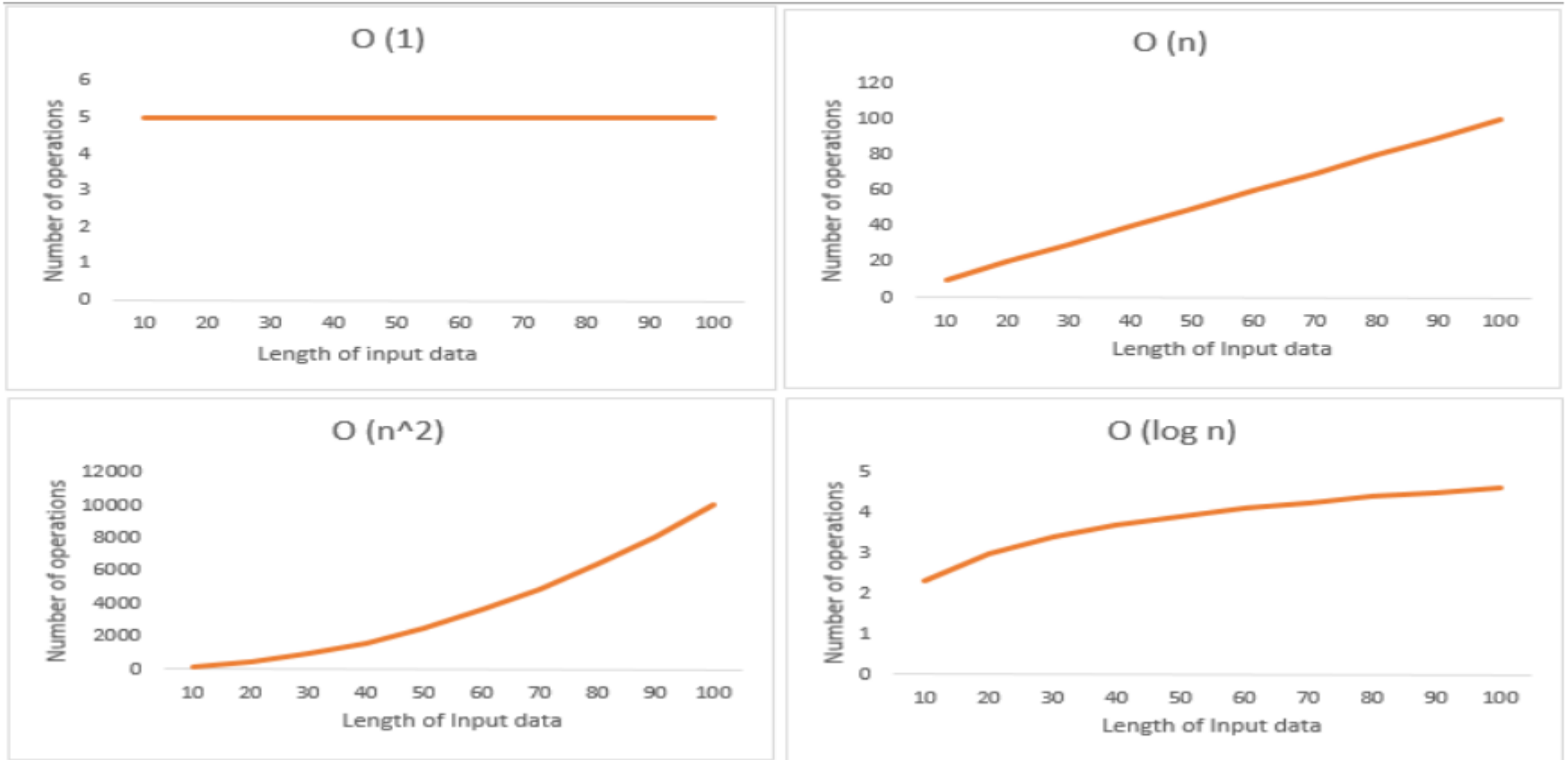
$$S(n) = \cancel{n+3}$$

$O(n)$

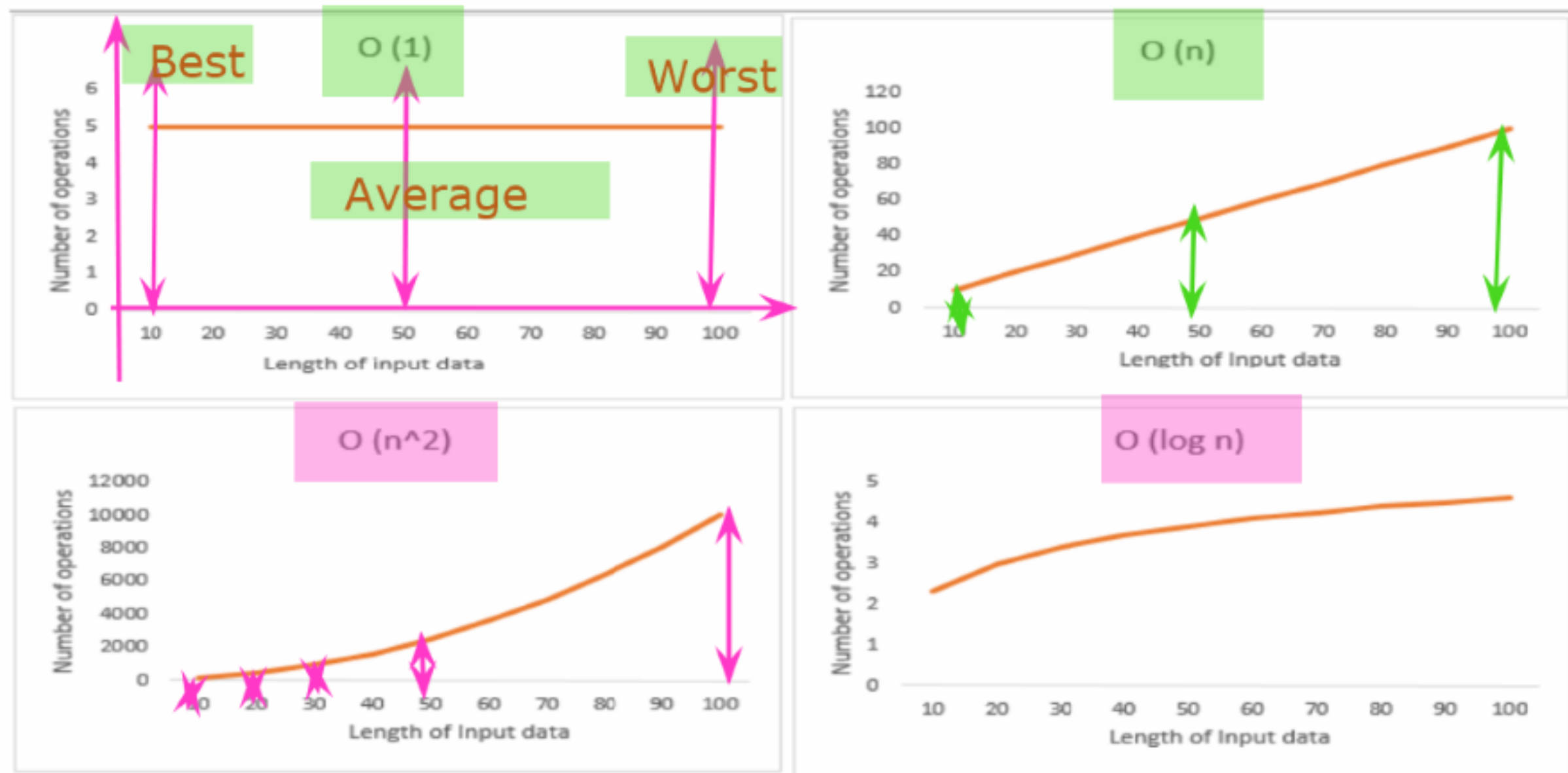
$O(n)$

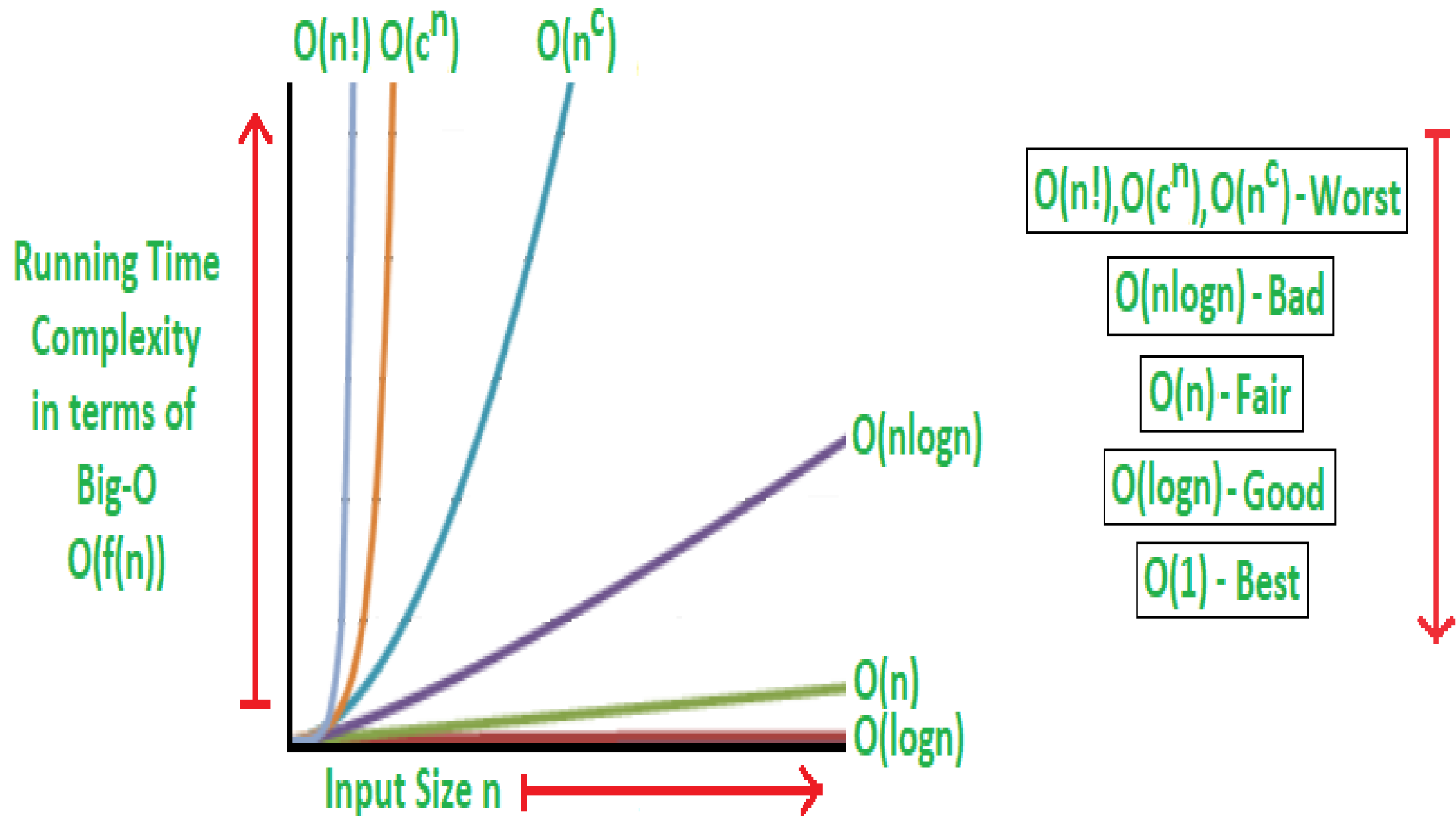
Linear Complexity

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



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





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.

```
def findDuplicates(arr, Len):
```

```
    ifPresent = False
```

```
    a1 = []
```

```
    for i in range(Len - 1):
```

```
        for j in range(i + 1, Len):
```

```
            if (arr[i] == arr[j]):  
                if arr[i] in a1:  
                    break
```

```
            else:  
                a1.append(arr[i])  
                ifPresent = True
```

```
    if (ifPresent):  
        print(a1, end = " ")
```

```
    else:
```

```
        print("No duplicates present in arrays")
```

Method 2:

=====

```
def printDuplicates(arr):
```

```
    dict = {}
```

```
    for ele in arr:
```

```
        try:
```

```
            dict[ele] += 1
```

```
        except:
```

```
            dict[ele] = 1
```

```
    for item in dict:
```

```
        if(dict[item] > 1):
```

```
            print(item, end=" ")
```

```
    print("\n")
```

Linear Search

20-->Best case $O(1)$

45-->Average case $O(n)$

67--->worst case $O(n)$

- Find 37?

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

↑
≠

↑
≠

↑
=

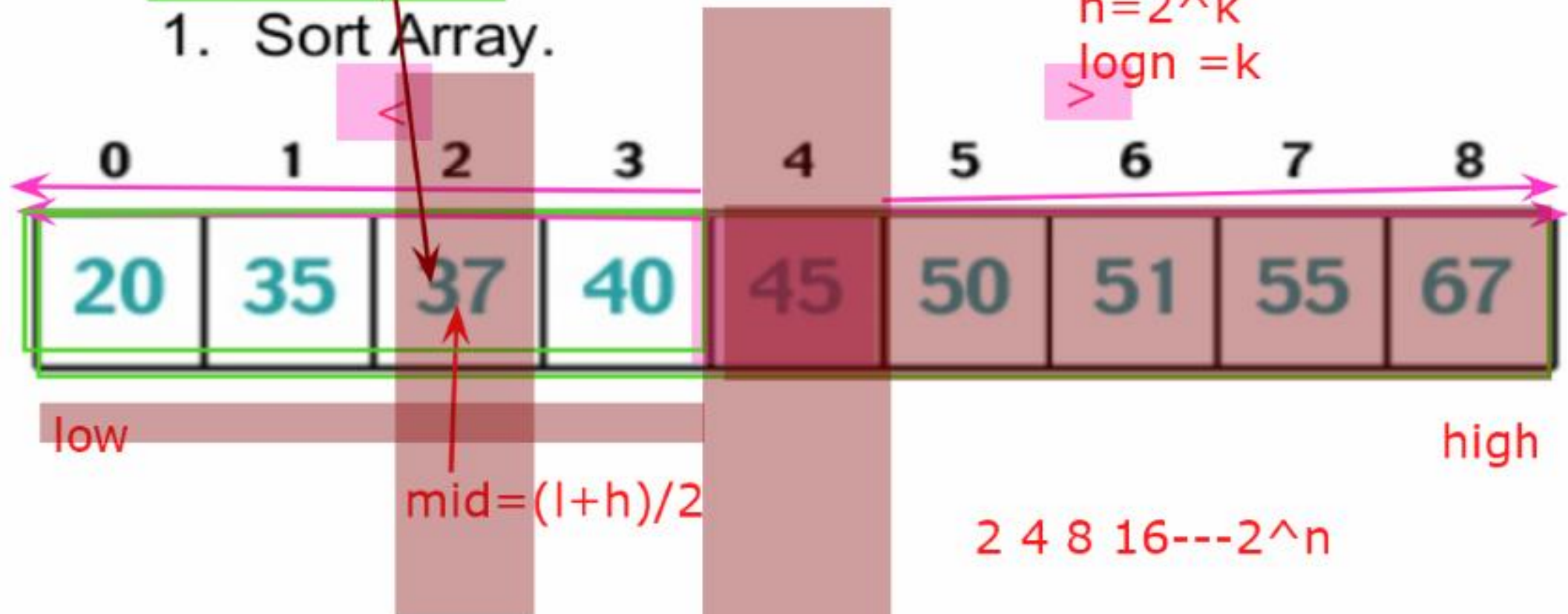
Return 2

Linear search

```
In [14]: def Lsearch(array, key):  
          n = len(array)  
          for i in range(0, n):  
              if array[i] == key:  
                  return i  
          return -1  
  
# Driver Code  
array = [2, 3, 4, 10, 40]  
x = 10  
result = Lsearch(array, x)  
if result == -1:  
    print("Element is not present in array")  
else:
```


Binary Search

- Find 37?
1. Sort Array.



$$n = 2^k$$
$$\log n = k$$

$$2 \ 4 \ 8 \ 16 \dots 2^n$$

Find duplicates in a given array when elements are not limited to a range

- Given an array of n integers. The task is to print the duplicates in the given array. If there are no duplicates then print -1.
- Examples:
- Input: {2, 10,10, 100, 2, 10, 11,2,11,2}
- Output: 2 10 11
- Input: {5, 40, 1, 40, 100000, 1, 5, 1}
- Output: 5 40 1

Problem statement: Find duplicates in an array

- Given an array `a1[]` of size `N` which contains elements from 0 to `N-1`, you need to find all the elements occurring more than once in the given array.
- **Example 1:**
 - Input:
 - `N = 4`
 - `a[] = {0,3,1,2}`
 - Output: -1
 - Explanation: `N=4` and all elements from 0 to (`N-1 = 3`) are present in the given array. Therefore output is -1.
- **Example 2:**
 - Input:
 - `N = 5`
 - `a[] = {2,3,1,2,3}`
 - Output: 2 3
 - Explanation: 2 and 3 occur more than once in the given array.

Problem statement: Program to find the initials of a name.

- Given a string name, we have to find the initials of the name
- Examples 1:
 - Input : Kabhi Haa Kabhi Naa
 - Output : K H K N
 - We take the first letter of all
 - words and print in capital letter.
- Example 2:
 - Input : Mahatma Gandhi
 - Output : M G
- Example 3:
 - Input : Shah Rukh Khan
 - Output : S R K
- Example 4: your own name

Problem Statement : Find the Missing Number

You are given a list of $n-1$ integers and these integers are in the range of 1 to n . There are no duplicates in the list. One of the integers is missing in the list. Write an efficient code to find the missing integer.

Example:

Input: `arr[] = {1, 2, 4, 6, 3, 7, 8}`

Output: 5

Explanation: The missing number from 1 to 8 is 5

Input: `arr[] = {1, 2, 3, 5}`

Output: 4

Explanation: The missing number from 1 to 5 is 4

Problem statement : Program for array rotation

Write a function rotate(ar[], d, n) that rotates arr[] of size n by d elements.

Array

1	2	3	4	5	6	7
---	---	---	---	---	---	---

Rotation of the above array by 2 will make array

ArrayRotation1

3	4	5	6	7	1	2
---	---	---	---	---	---	---

Linear Search

- Find 37?

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67



≠



≠



=

Return 2

Linear Search

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that $K \leq N$. 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

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 \neq 15, go to next element.
- 2) Compare 20 with 50
20 \neq 50, go to next element.
- 3) Compare 20 with 35
20 \neq 35, go to next element.
- 4) Compare 20 with 20
20 = 20, so 20 is found and its location is 4.

Program 3

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

- Find 37?
 1. Sort Array.

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

Binary Search

2. Calculate $\text{middle} = (\text{low} + \text{high}) / 2$.
 $= (0 + 8) / 2 = 4$.

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67
↑ first				↑ middle				↑ last

If $37 == \text{array}[\text{middle}] \rightarrow \text{return middle}$

Else if $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$

Binary Search

Repeat 2. Calculate $\text{middle} = (\text{low} + \text{high}) / 2$.
 $= (0 + 3) / 2 = 1$.

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67
↑	↑		↑					
first	middle		last					

If $37 == \text{array}[\text{middle}] \rightarrow \text{return middle}$


Else if $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$

Binary Search

Repeat 2. Calculate $\text{middle} = (\text{low} + \text{high}) / 2$.
 $= (2 + 3) / 2 = 2$.

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67


middle first last

If $37 == \text{array}[\text{middle}] \rightarrow \text{return middle}$

Else if $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$

Binary Search

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

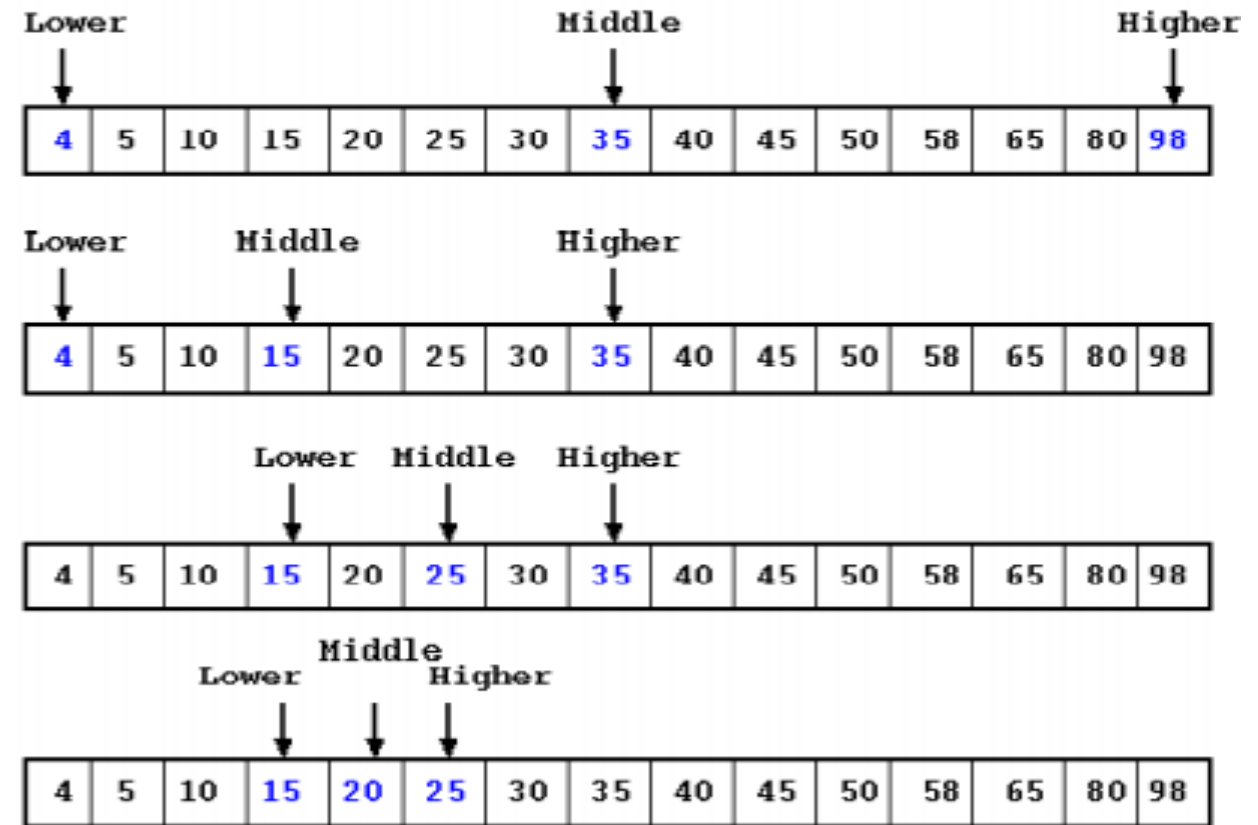
↑
middle

Binary Search

- If not found → stop when $low > high$.

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



Binary Search

```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.

    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while
end procedure
```