4. All about common Complexities

Big(1) - Constant Time

Big(1) can also be called as O(1) or O(c)

- No matter size of array/input , accessing an array/input same amount of time.
- Example : Accessing a specific element in an array by index.

```java
package com.dsa;

public class Complex {

    static void constantTime() {  1 usage
        int array[] = {
                1, 2, 4, 5, 6
        };

        /**
         * here the statement which is accessing our array will take constant time no matter we increase the input size.
         * i.e. Complexity is O(1)
         */
        System.out.println(array[0]);
    }

    public static void main(String[] args) {
        constantTime();
    }
}
```

Equation for time : f(n) = C

O(n) - Linear Time

- Time grows directly proportional to the size of input.
- Example : Traversing an array of size.

```java
package com.dsa;

public class Complex_LinearTime {

    static void linearTime() {
        int array[] = {
                1, 2, 4, 5, 6
        };

        for(int i = 0; i <= array.length - 1; i++) {

            /**
             * here the statement which is accessing our array will take more time when input is increased.
             * i.e. time will grow proportionally when input is grown
             * i.e. Complexity is O(n)
             */
            System.out.println((i+1) + " time \t");
            System.out.println(array[i] + " ");
        }
    }

    public static void main(String[] args) {
        linearTime();
    }
}
```

Equation for time : $f(n) = a(n) + C$ {Linear Equation -> $y = mx=c$}

$O(n^2)$ - Quadratic Time

- As the input grows, time taken increases quadratically.
- Example : Bubble Sort or checking all pairs in an array.

```java
package com.dsa;

public class Complex_QuadtraticTime_BubbleSort {
    static void bubbleSort() {  1 usage
        int array[] = {
                4, 2, 1, 3
        };

        System.out.println("Before bubble sort: ");
        for(int a : array){
            System.out.print(a + ",");
        }

        int i,j,temp;
        boolean flag;

        for(i = 0; i < array.length-1; i++){
            flag = false;
            for(j = 0; j < array.length-i-1; j++){
                if(array[j] > array[j+1]){
                    temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;
                    flag = true;
                }
            }
            if(!flag){
                break;
            }
        }
        System.out.println("\nAfter bubble sort: ");
        for(int s : array){
            System.out.print(s + ",");
        }
    }

    public static void main(String[] args) {
        bubbleSort();
    }

}
```

Equation for time :

Here A, B and C are constants

$f(n) = An^2+Bn+C$

As compared to An^2, $Bn + C$ is far smaller so ignoring constants we get :

$f(n) = An^2$

Again ignoring constant A we get :

$f(n) = n^2$

O log(n) - Logarithmic Time

- In this the algorithm cuts problem in half with each step, so the time grows logarithmically.
- Example : Binary Search

Equation for time : $\log(n) = K$

O(n log n) - Linearithmic Time

- The algorithm divides the input into subproblems and processes each subproblem linearly.
- Example : Merge Sort, Quick Sort in average cases.

$O(2^n)$ Exponential Time Complexity

- The time grows exponentially with the size of input, meaning it doubles with each additional input.
- Recursive algorithm that solves a problem by breaking into multiple smaller sub problems.
- Example : Fibonacci Series using recursion.

O(n!) Factorial Time Complexity

- Factorial Time Complexity occurs in algorithm that involve generating all possible permutations of a set.
- Example : Brute-Force solution for the travelling salesman problem.

Overall Comparison of all Time Complexities

$n(c) < O(\log \log n) < O(\log n) < O(\log n^{1/2}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(n^k) < O(2^n)$