# Time Complexity

## Step 01 Time Complexity

Time complexity in Big O notation measures how an algorithm's running time increases with the size of its input.

## Step 02 Statements Run Time

Each line of your code takes a certain time to run, and we can calculate the time for each statement by defining the type of the statement first.

### 01 Assignment, operations (+,-,*,etc...)

```
int x = 1;
// 1 assignment
x = 1 + 2;
// 1 assignment + 1 addition(+)
x = y;
// 1 assignment
```

CODE 1

The time complexity for the above code is O(1)

### 02 Comparison

```
if (condition){}
// 1 comparison
```

CODE 2

The time complexity for the above code is O(1)

### 03 Return

```
return result;
// 1
```

CODE 3

The time complexity for the above code is O(1)

### 04 Loop

Its time depends on how many iterations the loop will do. The following code will take (n) time complexity.

```
for (i = 1; i <= n; i++){}
// 1 assignment, (1 comparison + 1 increment) done n times
```

CODE 4

The loop will iterate till the i value reaches the n value, as the comparison is (i <= n).
(i = 1) 1 assignment.
( i<=n ) 1 comparison, done n times as the comparison will be performed in each iteration.
( i++ ) 1 increment, done n times as the incrementation will be performed in each iteration.

- Time calculation

-> T(1 + (1 + 1) * n )
-> T(1 + 2n)
-> Ignore constant (1, 2)
-> T(n)
-> O(n)

**NOTE**
Ignore constant is one of Big O rules and it will be explained in the following section

## Step 03 Big O Rules and Examples

Big O follows the rules listed below in time calculations

### 01 Ignore constant

```
int x = 0;
for ( i = 1; i<=n; i++){
    x = i + 1;
}
```

CODE 5

- Time analysis

Line #1 1 assignment
Line #2 1 assignment, (1 comparison + 1 increment) done n times ( 1+(1+1)*n )
Line #3 ( 1 assignment + 1 addition(+) ) done n times ( (1+1)*n )

- Time calculation

-> T(1 + (1 + 1)*n + (1 + 1)*n)
-> T(1 + 2n + 2n)
-> T(1 + 4n)

-> Ignore constant (1, 4)
-> T(n)
-> O(n)

## 02 Drop lower values

```
01  for (i = 1; i <= n; i++){
02      for (j = 1; j <= n; j++){
03          x = y + z
04       }
05  }
```

CODE 6 **nested loop**

- **Time analysis**

Line #1 1 assignment, (1 comparison + 1 increment) done n times ( 1+(1+1)*n )
Line #2 1 assignment done n times, (1 comparison + 1 increment) done n * n times (1*n + (1+1)n*n )
Line #3 (1 assignment + 1 addition(+)) done n * n times ( (1+1)n*n )

- **Time calculation**

-> T(1 + 2n + n + 2n^2 + 2n^2)
-> T(1 + 3n + 4n^2)
-> Drop lower values (3n), Ignore constant (1, 4)
-> T(n)
-> O(n^2)

## 03 Variables are combined only if they refer to the same input

```
01  int a = 0;
02  int b = 0;
03  for (i = 1; i <= n; i++){
04      a = a + 1;
05  }
06  for (i = 1; i <= m; i++){
07      b = b + 1;
08  }
```

CODE 7 **multiple loops**

- **Time analysis**

Line #3 1 assignment, (1 comparison + 1 increment) done n times ( 1+(1+1)*n )
Line #4 (1 assignment + 1 addition(+)) done n times ( 2n )
Line #6 1 assignment, (1 comparison + 1 increment) done m times ( 1+(1+1)*m )
Line #7 (1 assignment + 1 addition(+)) done m times ( 2m )

- **Time calculation**

-> T(1 + 2n + 2n + 1 + 2m + 2m)
-> Variables are combined only if they refer to the same input (2n, 2n), (2m, 2m)
-> T(2 + 4n + 4m)

-> Ignore constant (2, 4, 4)
-> T(n + m)
-> O(n + m)

## 04 Always take the worst-case time complexity

```
01  if (items.length > 0){
02    for (i = 1, i <= n, i++){
03  System.out.println(items[i])
04  }
05  } else {
06  System.out.println("Items is empty")
07  }
```

CODE 8

- Time analysis

Line #2 1 assignment, (1 comparison + 1 increment) done n times ( 1+(1+1)*n ) -> O(n)
Line #6 1 assignment (1)-> O(1)
The time complexity for the above code is O(n) as it is the worst-case.
O(n) > O(1).

## Step 04 Example of Time Complexity Enhancement

In this section, we will take a piece of code and enhance its time complexity to be faster.
The below example finds the maximum value in an array of integers.

```
public static int findMax(int[] arr) {
    int max = 0;                                          // {( 1 )}
    for (int i = 0; i < arr.length; i++) {                // {( n )}
        if (arr[i] > max) {                               // {( n )}
            max = arr[i];                                 // {( n )}
        }
    }
    return max;                                           // {( 1 )}
}
```

CODE 9

- Time calculation

-> T(1 + 3n + 1)
-> T(2 + 3n)
-> T(n)
-> O(n)

We can use a divide-and-conquer approach to enhance the time complexity to find the maximum element. This approach splits the array into smaller subarrays, recursively finds the maximum in each subarray, and then compares the maximum values obtained. This approach has a time complexity of O(log n):

```java
public static int findMax(int[] arr, int low, int high) {
   if (low == high) {
     return arr[low];
   }
   int mid = low + (high - low) / 2;
   }
   int leftMax = findMax(arr, low, mid);
   int rightMax = findMax(arr, mid + 1, high);
return Math.max(leftMax, rightMax);
}
```

CODE 10

- Time calculation

  -> T(1 + 1 + 1 + log n + log n + 1)

  -> T(4 + 2 log n)

  -> T(log n)

  -> O(log n)

In this enhanced version, the `findMax` method takes the array, the low index, and the high index as parameters. It checks if the low index is equal to the high index, which indicates a single element, and returns that element as the maximum If the low index is not equal to the high index, the method calculates the midpoint and recursively finds the maximum in the left and right subarrays. Finally, it compares the maximum values obtained from the subarrays using the `Math.max` function and returns the overall maximum.

By implementing this enhancement, we reduce the time complexity from O(n) to O(log n) and achieve a more efficient algorithm for finding the maximum element in an array.