**Examples of Big-O analysis**

**Big O** time complexity notation and also learn how to compute the time complexity of any program.

There are different asymptotic notations in which the time complexities of algorithms are measured. Here, the **”O”(Big O)** notation is used to get the time complexities. Time complexity esti­mates the time to run an algo­rithm. It’s calcu­lated by counting the elemen­tary opera­tions. It is always a good practice to know the reason for execution time in a way that depends only on the algorithm and its input. This can be achieved by choosing an elementary operation, which the algorithm performs repeatedly, and define the time complexity **T(N)**as the number of such operations the algorithm performs given an array of length **N**.

**Example 1:**

**The time complexity for the loop with elementary operations:** Assuming these operations take unit time for execution. This unit time can be denoted by ***O(1)***. If the loop runs for **N times** without any comparison. Below is the illustration for the same:

|  |
| --- |
| // Java program to illustrate time  // complexity for single for-loop  class GFG  {  // Driver Code  public static void main(String[] args)  {  int a = 0, b = 0;  int N = 4, M = 4;  // This loop runs for N time  for (int i = 0; i < N; i++)  {  a = a + 10;  }  // This loop runs for M time  for (int i = 0; i < M; i++)  {  b = b + 40;  }  System.out.print(a + " " + b);  }  } |

**Output:**

40 160

**Explanation:** The Time complexity here will be ***O(N + M)***. Loop one is a single for-loop that runs **N times** and calculation inside it takes ***O(1) time***. Similarly, another loop takes **M times** by combining both the different loops takes by adding them   
is ***O( N + M + 1) = O( N + M)***.

**Example 2:**

After getting familiar with the elementary operations and the single loop. Now, to find the time complexity for nested loops, assume that two loops with a different number of iterations. It can be seen that, if the **outer**loop runs once, the inner will run **M times**, giving us a series as **M + M + M + M + M……….N times**, this can be written as **N \* M**. Below is the illustration for the same:

|  |
| --- |
| // Java program to illustrate time  // complexity for nested loop  import java.io.\*;  class GFG{  // Driver Code  public static void main (String[] args)  {  int a = 0;  int b = 0;  int N = 4;  int M = 5;  // Nested loops  for(int i = 0; i < N; i++)  {  for(int j = 0; j < M; j++)  {  a = a + j;    // Print the current  // value of a  System.out.print(a + " ");  }  System.out.println();  }  }  } |

**Output:**

0 1 3 6 10

10 11 13 16 20

20 21 23 26 30

30 31 33 36 40

**Example 3:**

After getting the above problems. Let’s have two iterators in which, outer one runs**N/2 times,**and we know that the time complexity of a loop is considered as ***O(log N)***, if the iterator is divided / multiplied by a constant amount **K** then the time complexity is considered as ***O(logKN)***. Below is the illustration of the same:

|  |
| --- |
| // Program to illustrate time  // complexity of the form O(log2 N)  import java.util.\*;  class GFG {  // Driver Code  public static void main (String[] args)  {  int N = 8, k = 0;    // First loop run N/2 times  for (int i = N / 2; i <= N; i++) {    // Inner loop run log N  // times for all i  for (int j = 2; j <= N;  j = j \* 2) {    // Print the value k  System.out.print(k + " ");    k = k + N / 2;  }  }    }  } |

**Output:**

0 4 8 12 16 20 24 28 32 36 40 44 48 52 56

**Example 4:**

Now, let’s understand the while loop and try to update the iterator as an expression. Below is the illustration for the same:

|  |
| --- |
| // Java program to illustrate time  // complexity while updating the  // iteration  import java.io.\*;  class GFG {  // Driver Code  public static void main (String[] args)  {  int N = 18;  int i = N, a = 0;  // Iterate until i is greater  // than 0  while (i > 0) {  // Print the value of a  System.out.print(a + " ");  a = a + i;  // Update i  i = i / 2;  }  }  } |

**Output:**

0 18 27 31 33

**Explanation:** The equation for above code can be given as:

|  |
| --- |
| => (N/2)K = 1 (for k iterations)  => N = 2k (taking log on both sides)  => k = log(N) base 2.  Therefore, the time complexity will be  T(N) = O(log N) |

**Example 5:** Another way of finding the time complexity is converting them into an expression and use the following to get the required result. Given an expression based on the algorithm, the task is to solve and find the time complexity. This methodology is easier as it uses a basic mathematical calculation to expand a given formula to get a particular solution. Below are the two examples to understand the method.

**Steps:**

* Find the solution for **(N – 1)th** iteration/step.
* Similarly, calculate for the next step.
* Once, you get familiar with the pattern, find a solution for the **Kth** step.
* Find the solution for **N** times, and solve for obtained expression.

Below is the illustration for the same:

|  |
| --- |
| *Let the expression be: T(N) = 3\*T(N – 1).*  *T(N) = 3\*(3T(N-2)) T(N) = 3\*3\*(3T(N – 3))*  ***For k times:*** *T(N) = (3^k – 1)\*(3T(N – k))*  ***For N times:*** *T(N) = 3^N – 1 (3T(N – N)) T(N) = 3^N – 1 \*3(T(0)) T(N) = 3^N \* 1  T(N) = 3^N* |

The third and the simplest method is to use the Master’s Theorem or calculating time complexities