We have discussed [Asymptotic Analysis](https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/),  [Worst, Average and Best Cases](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/) and [Asymptotic Notations](https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/) in previous posts. In this post, analysis of iterative programs with simple examples is discussed.

**1) O(1):**Time complexity of a function (or set of statements) is considered as O(1) if it doesn’t contain loop, recursion and call to any other non-constant time function.

// set of non-recursive and non-loop statements

For example [swap() function](http://geeksquiz.com/c-program-swap-two-numbers/) has O(1) time complexity.  
A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

// Here c is a constant

for (int i = 1; i <= c; i++) {

// some O(1) expressions

}

**2) O(n):** Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example following functions have O(n) time complexity.

// Here c is a positive integer constant

for (int i = 1; i <= n; i += c) {

// some O(1) expressions

}

for (int i = n; i > 0; i -= c) {

// some O(1) expressions

}

**3) O(nc)**: Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example, the following sample loops have O(n2) time complexity

for (int i = 1; i <=n; i += c) {

for (int j = 1; j <=n; j += c) {

// some O(1) expressions

}

}

for (int i = n; i > 0; i -= c) {

for (int j = i+1; j <=n; j += c) {

// some O(1) expressions

}

For example [Selection sort](http://geeksquiz.com/selection-sort/) and [Insertion Sort](http://geeksquiz.com/insertion-sort/) have O(n2) time complexity.

**4) O(Logn)** Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.

for (int i = 1; i <=n; i \*= c) {

// some O(1) expressions

}

for (int i = n; i > 0; i /= c) {

// some O(1) expressions

}

For example [Binary Search(refer iterative implementation)](http://geeksquiz.com/binary-search/) has O(Logn) time complexity. Let us see mathematically how it is O(Log n). The series that we get in first loop is 1, c, c2, c3, … ck. If we put k equals to Logcn, we get cLogcn which is n.

**5) O(LogLogn)** Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.

// Here c is a constant greater than 1

for (int i = 2; i <=n; i = pow(i, c)) {

// some O(1) expressions

}

//Here fun is sqrt or cuberoot or any other constant root

for (int i = n; i > 1; i = fun(i)) {

// some O(1) expressions

}

**How to combine time complexities of consecutive loops?**  
When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

for (int i = 1; i <=m; i += c) {

// some O(1) expressions

}

for (int i = 1; i <=n; i += c) {

// some O(1) expressions

}

Time complexity of above code is O(m) + O(n) which is O(m+n)

If m == n, the time complexity becomes O(2n) which is O(n).

**How to calculate time complexity when there are many if, else statements inside loops?**  
As discussed [here](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/), worst case time complexity is the most useful among best, average and worst. Therefore we need to consider worst case. We evaluate the situation when values in if-else conditions cause maximum number of statements to be executed.

s  
For example consider the [linear search function](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/) where we consider the case when element is present at the end or not present at all.  
When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.  
**How to calculate time complexity of recursive functions?**  
Time complexity of a recursive function can be written as a mathematical recurrence relation. To calculate time complexity, we must know how to solve recurrences.

Complexity Analysis of Binary Search

Complexities like **O(1)** and **O(n)** are simple to understand. O(1) means it requires constant time to perform operations like to reach an element in constant time as in case of dictionary and O(n) means, it depends on the value of n to perform operations such as searching an element in an array of n elements.

But for **O(Log n)**, it is not that simple. Let us discuss this with the help of Binary Search Algorithm whose complexity is **O(log n)**.

**Binary Search:** Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

**Example:**  
[](https://www.geeksforgeeks.org/binary-search/)

Sorted Array of 10 elements: 2, 5, 8, 12, 16, 23, 38, 56, 72, 91

Let us say we want to search for 23.

**Finding the given element:**  
Now to find 23, there will be many iterations with each having steps as mentioned in the figure above:

* **Iteration 1:**

Array: 2, 5, 8, 12, 16, 23, 38, 56, 72, 91

* + Select the middle element. (**here 16**)
  + Since 23 is greater than 16, so we divide the array into two halves and consider the sub-array after element 16.
  + Now this subarray with the elements after 16 will be taken into next iteration.
* **Iteration 2:**

Array: 23, 38, 56, 72, 91

* + Select the middle element. (**now 56**)
  + Since 23 is smaller than 56, so we divide the array into two halves and consider the sub-array before element 56.
  + Now this subarray with the elements before 56 will be taken into next iteration.
* **Iteration 3:**

Array: 23, 38

* + Select the middle element. (**now 23**)
  + Since 23 is the middle element. So the iterations will now stop.

**Calculating Time complexity:**

* + Let say the iteration in Binary Search terminates after **k** iterations. In the above example, it terminates after 3 iterations, so **here k = 3**
  + At each iteration, the array is divided by half. So let’s say the length of array at any iteration is **n**
  + At **Iteration 1**,

Length of array = **n**

* + At **Iteration 2**,

Length of array = **n⁄2**

* + At **Iteration 3**,

Length of array = **(n⁄2)⁄2** = **n⁄22**

* + Therefore, after **Iteration k**,

Length of array = **n⁄2k**

* + Also, we know that after

After k divisions, the **length of array becomes 1**

* + Therefore
  + Length of array = **n⁄2k = 1**
  + => **n = 2k**
  + Applying log function on both sides:
  + => **log2 (n) = log2 (2k)**
  + => **log2 (n) = k log2 (2)**
  + As **(loga (a) = 1)**  
    Therefore,
  + => **k = log2 (n)**

**Hence, the time complexity of Binary Search is**

***log2 (n)***

**Properties of Asymptotic Notations:**  
As we have gone through the definition of these three notations let’s now discuss some important properties of those notations.

1. **General Properties:**

If f(n) is O(g(n)) then a\*f(n) is also O(g(n)) ; where a is a constant.

Example: f(n) = 2n²+5 is O(n²)  
then 7\*f(n) = 7(2n²+5)  
= 14n²+35 is also O(n²)

Similarly this property satisfies for both Θ and Ω notation.  
We can say  
If f(n) is Θ(g(n)) then a\*f(n) is also Θ(g(n)) ; where a is a constant.  
If f(n) is Ω (g(n)) then a\*f(n) is also Ω (g(n)) ; where a is a constant.

1. **Reflexive Properties:**

If f(n) is given then f(n) is O(f(n)).

Example: f(n) = n² ; O(n²) i.e O(f(n))

Similarly this property satisfies for both Θ and Ω notation.  
We can say  
If f(n) is given then f(n) is Θ(f(n)).  
If f(n) is given then f(n) is Ω (f(n)).

1. **Transitive Properties:**

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)) .

Example: if f(n) = n , g(n) = n² and h(n)=n³  
n is O(n²) and n² is O(n³)  
then n is O(n³)

Similarly this property satisfies for both Θ and Ω notation.  
We can say  
If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .  
If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

1. **Symmetric Properties:**

If f(n) is Θ(g(n)) then g(n) is Θ(f(n)) .

Example: f(n) = n² and g(n) = n²  
then f(n) = Θ(n²) and g(n) = Θ(n²)

**This property only satisfies for Θ notation.**

1. **Transpose Symmetric Properties:**

If f(n) is O(g(n)) then g(n) is Ω (f(n)).

Example: f(n) = n , g(n) = n²  
then n is O(n²) and n² is Ω (n)

**This property only satisfies for O and Ω notations**.

1. **Some More Properties:**
   1. If f(n) = O(g(n)) and f(n) = Ω(g(n)) then f(n) = Θ(g(n))
   2. If f(n) = O(g(n)) and d(n)=O(e(n))  
      then f(n) + d(n) = O( max( g(n), e(n) ))  
      Example: f(n) = n i.e O(n)  
      d(n) = n² i.e O(n²)  
      then f(n) + d(n) = n + n² i.e O(n²)
   3. If f(n)=O(g(n)) and d(n)=O(e(n))  
      then f(n) \* d(n) = O( g(n) \* e(n) )  
      Example: f(n) = n i.e O(n)  
      d(n) = n² i.e O(n²)  
      then f(n) \* d(n) = n \* n² = n³ i.e O(n³)