**Session -1:**

**Time Complexity**

* It is the time taken by a computer to completely execute a program.
* Program may be in any language. (C, C++, Java etc…)

Asymptotic Notations

* It is the meaningful way of representation of time complexity.

They are

i) Big Oh [ O (g(n)) ] { Upper bound }

ii) Big Omega [ Ω (g(n)) ] { Lower bound }

iii) Big Theta [ Ɵ (g(n)) ] { Average }

The Order increases in this format:

O (1), O (log n), O (n/2), O (n), O (n2), O (n3) . . .

**Analysis: Sum of n natural numbers**

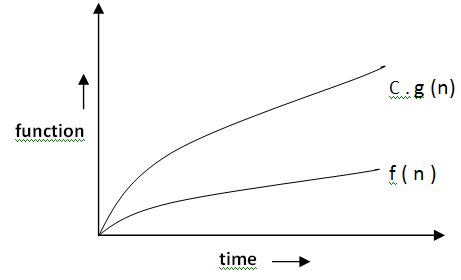
**Algorithm: Time taken**

1. Start ------ 0
2. Input: Read ‘n’ ------ 1
3. initialize: sum=0, i=1 ------ 1
4. Process: sum = sum + i ------ n
5. i = i +1 ------ n
6. if ( i < = n ) go to step 4 ------ n +1
7. Output:
8. print ‘sum’ ------ 1
9. Stop ------ 0

Total time in function f(n) = 3n + 4

**Big Oh: O() { Upper bound – Worst Case}**

The function f(n) = O(g(n)), if and only if there exists constants “c and n0 “ such that f(n) ≤ c . g(n) for all value of n, where as n>n0 .

****

**Analysis : Sum of n natural numbers**

f(n) = 3n + 4 ; f(n) ≤ C . g(n)

3n + 4 ≤ C . g(n)

3n + 4 ≤ 3n + 4n (Upper bound)

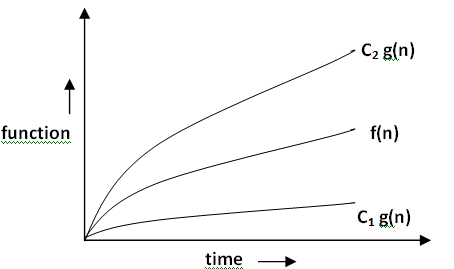
If 3n + 4 ≤ 7n then C = 7 , g(n) = n

If n=1 then 7 ≤ 7 ; If n=2 then 10 ≤ 14 ;If n=3 then 13 ≤ 21

* Therefore f(n) = O(g(n)) = O(n)
* thus f(n) = O(n) [Worst case ]

**Big Theta [ Ɵ () ] { Average }**

The function f(n) = Ɵ(g(n)), if and only if there exists constants “c1, c2 and n0“ such that c1 . g(n) ≤ f(n) ≤ c . g(n) for all value of n, where as n>n0 .

****

**Analysis : Sum of n natural numbers**

f(n) = 3n + 4 ; c1 . g(n) ≤ f(n) ≤ c . g(n)

C1 . g(n) ≤ 3n + 4 ≤ C2 . g(n)

3n ≤ 3n + 4 ≤ 7n (Lower and Upper bound)

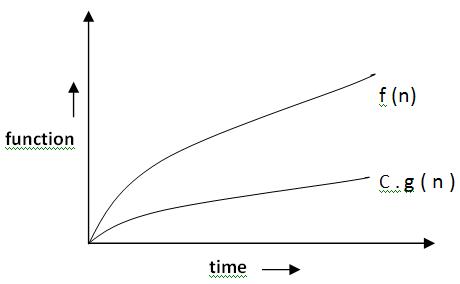
If 3n ≤ 3n + 4 ≤ 7n then C1 = 3 , C2 = 7 and g(n) = n

If n=1 then 3 ≤ 7 ≤ 7 ; If n=2 then 6 ≤ 10 ≤ 14 ;If n=3 then 9 ≤ 13 ≤ 21

* Therefore f(n) = Ɵ(g(n)) = Ɵ(n)
* thus f(n) = Ɵ(n) [Average Case]

**Big Omega [ Ω () ] { Lower bound }**

The function f(n) = Ω(g(n)), if and only if there exists constants “c and n0 “ such that f(n) ≥ c . g(n) for all value of n, where as n>n0 .

****

**Analysis : Sum of n natural numbers**

f(n) = 3n + 4 ; f(n) ≥ C . g(n)

3n + 4 ≥ C . g(n)

3n + 4 ≥ 3n (Lower bound)

If 3n + 4 ≥ 3n then C = 3 , g(n) = n

If n=1 then 7 ≥ 3

If n=2 then 10 ≥ 6

If n=3 then 13 ≥ 9

* Therefore f(n) = Ω(g(n)) = Ω(n)
* thus f(n) = Ω(n) [Best Case]

**Example II : Sum of n natural numbers**

**Algorithm: Time taken**

1. Start ------ 0
2. Input: Read n ------ 1
3. Process: sum = n \* (n+1) / 2 ------ 1
4. print ‘sum’ ------ 1
5. Stop ------ 0

Total time in function f(n) = 3

f(n) = 3 . 1 = C . g(n) ; C = 3 , g(n) = 1

therefore f(n) = O(g(n) ) = O (1) = Ω(1) = Ɵ(1)

**Session -2:**

**Time complexity Analysis of Algorithms With examples**

**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.

# Time Complexity of Loop with Powers

What is the time complexity of below function?

|  |
| --- |
| void fun(int n, int k)  {      for (int i=1; i<=n; i++)      {        int p = pow(i, k);        for (int j=1; j<=p; j++)        {            // Some O(1) work        }      }  } |

Time complexity of above function can be written as 1k + 2k + 3k + … n1k.

Let us try few examples:

k=1

Sum = 1 + 2 + 3 ... n

= n(n+1)/2

= n2 + n/2

k=2

Sum = 12 + 22 + 32 + ... n12.

= n(n+1)(2n+1)/6

= n3/3 + n2/2 + n/6

k=3

Sum = 13 + 23 + 33 + ... n13.

= n2(n+1)2/4

= n4/4 + n3/2 + n2/4

In general, asymptotic value can be written as **(nk+1)/(k+1) + Θ(nk)**

Note that, in asymptotic notations like Θ we can always ignore lower order terms. So the time complexity is **Θ(nk+1 / (k+1))**

**Simple Statement**

This statement takes **O(1)** time.

int y= n + 25;

**If Statement**

The worst case O(n) if the if statement is in a loop that runs n times, best case **O(1)**

if( n> 100)

{

…

}else{

..

..

}

**For / While Loops**

The **for** loop takes n time to complete and and so it is **O(n)**.

|  |
| --- |
| for(int i=0;i<n;i++)  {  ..  ..  } |

If the **for** loop takes n time and i increases or decreases by a constant, the cost is **O(n)**

for(int i = 0; i < n; i+=5)

   sum++;

for(int i = n; i > 0; i-=5)

   sum++;

**Nested loops**

If the nested loops contain sizes n and m, the cost is **O(nm)**

|  |  |
| --- | --- |
|  | for(int i=0;i<n;i++)  {      for(int i=0;i<m;i++){      ..      ..      }  } |

If the first loop runs n2 times and the inner loop runs n times or (vice versa), the cost is **O(n3)**

|  |  |
| --- | --- |
|  | for(int j=0;j<n\*n;j++)  {      for(int i=0;i<n;i++){      ..      ..      }  } |

If the first loop runs n times and the inner second loop runs n2 times and the third loop runs n2, then **O(n5)**

for(int i = 0; i < n; i++)

    for( int j = 0; j < n \* n; j++)

        for(int k = 0; k < j; k++)

            sum++;

If the **for** loop takes n time and i increases or decreases by a multiple, the cost is **O(log(n))**

for(int i = 1; i < =n; i\*=2)

   sum++;

for(int i = n; i > 0; i/=2)

   sum++;

If the first loop runs N times and the inner loop runs log(n) times or (vice versa), the cost is **O(n\*log(n))**

|  |  |
| --- | --- |
|  | for(int i=0;i<n;i++)  {      for(int j=1;i<=n;j\*=4){      ..      ..      }  } |