**What is Logarithmic Time Complexity? A Complete Tutorial**

**Logarithmic time complexity**is denoted as **O(log n).**It is a measure of how the runtime of an algorithm scales as the input size increases. In this comprehensive tutorial. In this article, we will look in-depth into the Logarithmic Complexity. We will also do various comparisons between different logarithmic complexities, when and where such logarithmic complexities are used, several examples of logarithmic complexities, and much more. So let’s get started.

*What is Logarithmic Time Complexity*

**Table of Content**

* [What is a Logarithm?](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#what-is-a-logarithm)
* [What is Complexity Analysis?](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#what-is-complexity-analysis)
* [What is Space Complexity?](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#what-is-space-complexity)
* [What is Time Complexity?](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#what-is-time-complexity)
  + [How to measure complexities?](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#how-to-measure-complexities)
* [What is a Logarithm?](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#what-is-a-logarithm-1)
* [Different Types of Logarithmic Complexities](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#different-types-of-logarithmic-complexities)
  + [Simple Log Complexity (Log a)](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#simple-log-complexity-loga-b)
  + [Double Logarithm (log log N)](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#double-logarithm-log-log-n)
  + [N logarithm N (N \* log N)](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#n-logarithm-n-n-log-n)
  + [logarithm^2 N (log^2 N)](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#logarithm2-n-log2-n)
  + [N^2 logarithm N (N^2 \* log N)](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#n2-logarithm-n-n2-log-n)
  + [N^3 logarithm N (N^3 log N)](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#n3-logarithm-n-n3-log-n)
  + [logarithm √N (log √N)](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#logarithm-n-log-n)
* [Examples To Demonstrate Logarithmic Time Complexity](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#examples-to-demonstrate-logarithmic-time-complexity)
* [Practice Problems for Logarithmic Time Complexity](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#practice-problems-for-logarithmic-time-complexity)
* [Comparison of various Logarithmic Time Complexities](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#comparison-of-various-logarithmic-time-complexities)
* [Frequently Asked Questions(FAQ’s) on Logarithmic Time Complexity](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#frequently-asked-questionsfaqs-on-logarithmic-time-complexity)
* [Conclusion](https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/#conclusion)

**What is a Logarithm?**

The power to which a **base**needs to be raised to reach a given**number**is called the [logarithm](https://www.geeksforgeeks.org/logarithm/) of that **number**for the respective **base.**  
For finding logarithmic two necessary factors that need to be known are **base**and **number.**

**What is Complexity Analysis?**

The primary motive to use DSA is to solve a problem effectively and efficiently. How can you decide if a program written by you is efficient or not? This is measured by complexities. Complexity is of two types:

[**What is Space Complexity?**](https://www.geeksforgeeks.org/g-fact-86/)

The space Complexity of an algorithm is the space taken by an algorithm to run the program for a given input size. The program has some space requirements necessary for its execution these include auxiliary space and input space. The important standard for comparison of algorithms is the space taken by the algorithm to run for a given input size Hence it needs to be optimized.

[**What is Time Complexity?**](https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/)

In Computer science, there are various problems and several ways to solve each of these problems using different algorithms. These algorithms may have varied approaches, some might be too complex to Implement while some may solve the problem in a lot simpler way than others. It is hard to select a suitable and efficient algorithm out of all that are available. To make the selection of the best algorithm easy, calculation of complexity and time consumption of an algorithm is important this is why **time complexity analysis**is important, for this [**asymptotic analysis**](https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/)of the algorithm is done.

There are three cases denoted by three different notations of analysis:

* [**Big-oh(O) Notation**](https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/)**:**Denotes the upper bound of any algorithm’s runtime i.e. time is taken by the algorithm in the worst case.
* [**Big-omega(Ω) Notation**](https://www.geeksforgeeks.org/analysis-of-algorithms-big-%CF%89-big-omega-notation/)**:**Denotes the best runtime of an algorithm.
* [**Big-Theta(Θ) notation:**](https://www.geeksforgeeks.org/analysis-of-algorithms-big-%CE%B8-big-theta-notation/#:~:text=In%20simple%20language%2C%20Big%20%E2%80%93%20Theta,the%20program%20into%20smaller%20segments.) Denotes average case time complexity.

**How to measure complexities?**

Below are some major order of Complexities are:

* **Constant:**If the algorithm runs for the same amount of time every time irrespective of the input size. It is said to exhibit constant time complexity.
* **Linear:**If the algorithm runtime is linearly proportional to the input size then the algorithm is said to exhibit linear time complexity.
* **Exponential:**If the algorithm runtime depends on the input value raised to an exponent then it is said to exhibit exponential time complexity.
* **Logarithmic:**When the algorithm runtime increases very slowly compared to an increase in input size i.e. logarithm of input size then the algorithm is said to exhibit logarithmic time complexity.

| **Notation** | **Complexity** |
| --- | --- |
| **O(1)** | Constant |
| **O(log N)** | Logarithmic |
| **O(N)** | Linear time |
| **O(N \* log N)** | Log linear |
| **O(N^2)** | Quadratic |
| **O(N^3)** | Cubic |
| **O(2^N)** | Exponential |
| **O(N!)** | Factorial |

*Measurement of Complexity analysis*

**What is a Logarithm?**

*The power to which a****base****needs to be raised to reach a given****number****is called the*[*logarithm*](https://www.geeksforgeeks.org/logarithm/)*of that****number****for the respective****base.*** *For finding logarithmic two necessary factors that need to be known are****base****and****number.***

**Examples:**

***logarithm of 8 for base 2****= log2(8) = 3,****Explanation:****23 = 8 Since 2 needs to be raised to a power of 3 to give 8, Thus logarithm of 8 base 2 is 3.*

***logarithm of 81 for base 9****= log9(81) = 2,****Explanation:****92 = 81 Since 9 needs to be raised to a power of 2 to give 81, Thus logarithm of 81 base 9 is 2.*

**Note:** An **exponential function** is the exact opposite of a**logarithmic function**. When a value is being multiplied repeatedly it is said to grow exponentially whereas when the value is being divided repeatedly it is said to grow logarithmically.

**Different Types of Logarithmic Complexities**

Now that we know what is a logarithm, let’s deep dive into different types of logarithmic complexities that exists, such as:

**Simple Log Complexity (Loga b)**

Simple logarithmic complexity refers to **log of b to the base a**. As mentioned, it refers to the time complexity in terms of base a. In [design and analysis of algorithms](https://www.geeksforgeeks.org/design-and-analysis-of-algorithms/), we generally use 2 as the base for log time complexities. The below graph shows how the simple log complexity behaves.

*Simple Log Complexity (Log(a) b)*

 There are several standard algorithms that have **logarithmic**time complexity:

* [Merge sort](http://www.geeksforgeeks.org/merge-sort/)
* [Binary search](https://www.geeksforgeeks.org/binary-search/)
* [Heap sort](https://www.geeksforgeeks.org/heap-sort/)

**Double Logarithm (log log N)**

Double logarithm is the power to which a **base** must be raised to reach a value ‘**x’**such that when the base is raised to a power ‘**x’**it reaches a value equal to given **number.**

*Double Logarithm (log log N)*

**Example:**

***logarithm (logarithm (256)) for base 2****=**log2(log2(256))**= log2(8) = 3.*

***Explanation:****28 = 256, Since 2 needs to be raised to a power of 8 to give 256, Thus logarithm of 256 base 2 is 8. Now 2 needs to be raised to a power of 3 to give 8 so log2(8) = 3.*

**N logarithm N (N \* log N)**

**N\*logN** complexity refers to **product of N and log of N to the base 2**. **N \* log N** time complexity is generally seen in sorting algorithms like [Quick sort](https://www.geeksforgeeks.org/quick-sort/), [Merge Sort](https://www.geeksforgeeks.org/merge-sort/), [Heap sort](https://www.geeksforgeeks.org/heap-sort/). Here N is the size of data structure (array) to be sorted and log N is the average number of comparisons needed to place a value at its right place in the sorted array.

*N \* log N*

**logarithm2 N (log2 N)**

log2 N complexity refers to **square of log of N to the base 2**.

*log2 N*

**N2 logarithm N (N2 \* log N)**

**N2\*log N** complexity refers to **product of square of N and log of N to the base 2**. This Order of time complexity can be seen in case where an N \* N \* N 3D matrix needs to be sorted along the rows. The complexity of sorting each row would be N log N and for N rows it will be N \* (N \* log N). Thus the complexity will be N2 log N,

*N2 \* log N*

**N3 logarithm N (N3 log N)**

N3\*log N complexity refers to **product of cube of N and log of N to the base 2**. This Order of time complexity can be seen in cases where an N \* N matrix needs to be sorted along the rows. The complexity of sorting each row would be N log N and for N rows it will be N \* (N \* log N) and for N width it will be N \* N \* (N log N). Thus the complexity will be N3 log N,

*N3 log N*

**logarithm √N (log √N)**

log √N complexity refers to **log of square root of N to the base 2**.

*log √N*

**Examples To Demonstrate Logarithmic Time Complexity**

**Example 1: loga b**

**Task:** We have a number**N**which has an initial value of **16**and the task is to reduce the given number to 1 by repeated division of 2.   
**Approach:**

* Initialize a variable number\_of\_operation with a value 0 .
* Run a for loop from N till 1.
  + In each iteration reduce the value of N to half.
  + Increment the number\_of\_operation variable by one.
* Return the number\_of\_operation variable.

**Implementation:**

Recommended Problem

[Power of Four](https://www.geeksforgeeks.org/problems/power-of-four/1/?itm_source=geeksforgeeks&itm_medium=article&itm_campaign=practice_card)

Topics:

[Mathematical](https://www.geeksforgeeks.org/explore?page=1&category%5b%5d=Mathematical&sortBy=submissions)[Algorithms](https://www.geeksforgeeks.org/explore?page=1&category%5b%5d=Algorithms&sortBy=submissions)

[Solve Problem](https://www.geeksforgeeks.org/problems/power-of-four/1/?itm_source=geeksforgeeks&itm_medium=article&itm_campaign=practice_card)

Basic

40.48%

14.3K

C++JavaPythonC#JavaScript

*// C++ code for reducing a number to its logarithm*

#include *<bits/stdc++.h>*

**using** **namespace** **std**;

int main()

{

int N = 16;

int number\_of\_operations = 0;

cout << "Logarithmic reduction of N: ";

**for** (int i = N; i > 1; i = i / 2) {

cout << i << " ";

number\_of\_operations++;

}

cout << '\n'

<< "Algorithm Runtime for reducing N to 1: "

<< number\_of\_operations;

}

**Output**

Logarithmic reduction of N: 16 8 4 2

Algorithm Runtime for reducing N to 1: 4

**Explanation:**

It is clear from the above algorithm that in each iteration the value is divided by a factor of 2 starting from 16 till it reaches 1, it takes 4 operations.

As the input value gets reduced by a factor of 2, In mathematical terms the number of operations required in this case is **log2(N),** i.e. **log2(16) = 4.So,**in terms of time complexity, the above algorithm takes logarithmic runtime to complete i.e.  **log2(N)**.

**Example 2:**[**Binary search algorithm**](https://www.geeksforgeeks.org/what-is-binary-search-algorithm/)**(log N)**

Linearly Searching a value in an array of size **N**can be very hectic, even when the array is sorted but using binary search this can be done in a lot easier way and in lesser time as the algorithm reduces the search space by half in each operation thus gives a complexity of **log2(N),**Here **base** is 2 because process repeatedly reduces to half.

Consider an array **Arr[]**= {2, 4, 6, 8, 10, 12, 14, 16, 18}, If it is required to find the index of 8 then the algorithm will work as following:

C++JavaPythonC#JavaScript

*// C++ program for finding the index of 8*

#include *<iostream>*

**using** **namespace** **std**;

int find\_position(int val, int Arr[], int n, int& steps)

{

int l = 0, r = n - 1;

**while** (l <= r) {

steps++;

int m = l + (r - l) / 2;

**if** (Arr[m] == val)

**return** m;

**else** **if** (Arr[m] < val)

l = m + 1;

**else**

r = m - 1;

}

**return** -1;

}

*// Driver Code*

int main()

{

int Arr[8] = { 2, 4, 6, 8, 10, 12, 14, 16 };

int steps = 0;

*// Function Call*

int idx = find\_position(8, Arr, 8, steps);

cout << "8 was present on index: "<<idx << endl;

*// Since the worst case runtime of Binary search is*

*// log(N) so the count of steps must be less than log(N)*

cout << "Algorithm Runtime: " << steps << endl;

**return** 0;

}

**Output**

8 was present on index: 3

Algorithm Runtime: 1

**Explanation:**

Binary search works on **Divide and conquer** approach, In above example In worst case 3 comparisons will be needed to find any value in array. Also the value of log (N) where N is input size i.e. 8 for above example will be 3. Hence the algorithm can be said to exhibit logarithmic time complexity.

**Example 3: Binary search algorithm (log log N)**

An example where the time complexity of algorithm is **Double logarithmic**along with a length factor **N**is when prime numbers from 1 to **N**need to be found.

C++JavaPythonC#JavaScript

#include *<bits/stdc++.h>*

**using** **namespace** **std**;

**const** long long MAX\_SIZE = 1000001;

*// isPrime[] : isPrime[i] is true if number is prime*

*// prime[] : stores all prime number less than N*

*// SPF[] that store smallest prime factor of number*

*// [for Exp : smallest prime factor of '8' and '16'*

*// is '2' so we put SPF[8] = 2 , SPF[16] = 2 ]*

vector<long long> isprime(MAX\_SIZE, true);

vector<long long> prime;

vector<long long> SPF(MAX\_SIZE);

*// Function generate all prime number less than N in O(n)*

void manipulated\_seive(int N)

{

*// 0 and 1 are not prime*

isprime[0] = isprime[1] = false;

*// Fill rest of the entries*

**for** (long long int i = 2; i < N; i++) {

*// If isPrime[i] == True then i is*

*// prime number*

**if** (isprime[i]) {

*// put i into prime[] vector*

prime.push\_back(i);

*// A prime number is its own smallest*

*// prime factor*

SPF[i] = i;

}

*// Remove all multiples of i\*prime[j] which are*

*// not prime by making isPrime[i\*prime[j]] = false*

*// and put smallest prime factor of i\*Prime[j] as*

*// prime[j] [ for exp :let i = 5 , j = 0 , prime[j]*

*// = 2 [ i\*prime[j] = 10 ] so smallest prime factor*

*// of '10' is '2' that is prime[j] ] this loop run*

*// only one time for number which are not prime*

**for** (long long int j = 0;

j < (int)prime.size() && i \* prime[j] < N

&& prime[j] <= SPF[i];

j++) {

isprime[i \* prime[j]] = false;

*// put smallest prime factor of i\*prime[j]*

SPF[i \* prime[j]] = prime[j];

}

}

}

*// Driver program to test above function*

int main()

{

int N = 13; *// Must be less than MAX\_SIZE*

manipulated\_seive(N);

*// Print all prime number less than N*

**for** (int i = 0; i < prime.size() && prime[i] <= N; i++)

cout << prime[i] << " ";

**return** 0;

}

**Output**

2 3 5 7 11

In above example the complexity of finding prime numbers in a range of 0 to **N**is O(N \* log (log (N))).

**Practice Problems for Logarithmic Time Complexity**

| **Articles** | **Practice** | **Time Complexity** |
| --- | --- | --- |
| [Search an element in a sorted and rotated Array](https://www.geeksforgeeks.org/search-an-element-in-a-sorted-and-pivoted-array/) | [Solve](https://www.geeksforgeeks.org/problems/search-in-a-rotated-array4618/1) | O(log N) |
| [Sieve of Eratosthenes – GeeksforGeeks](https://www.geeksforgeeks.org/sieve-of-eratosthenes/) | [Solve](https://www.geeksforgeeks.org/problems/sieve-of-eratosthenes5242/1) | O(n\*log(log(n))) |
| [Count Inversions in an array](https://www.geeksforgeeks.org/counting-inversions/) | [Solve](https://www.geeksforgeeks.org/problems/inversion-of-array-1587115620/1) | O(n\*log n) |
| [QuickSort](https://www.geeksforgeeks.org/quick-sort/) | [Solve](https://www.geeksforgeeks.org/problems/quick-sort/1) | O(n\*log n) |
| [Prim’s Minimum Spanning Tree](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/) | [Solve](https://www.geeksforgeeks.org/problems/minimum-spanning-tree/1) | O(E log V) |

**Comparison of various Logarithmic Time Complexities**

Below is a graph to show the comparison between different logarithmic time complexities that have been discussed above:

*Comparison between various Logarithmic Time Complexities*

**Frequently Asked Questions(FAQ’s) on Logarithmic Time Complexity:**

**1) Why does logarithmic complexity need no base?**

Logarithms from any base i.e. 2, 10, e can be transformed to any other base with an addition of a constant, So the base of log doesn’t matter.

**2) How are logarithms used in real life?**

In Real Life scenario like measuring the acidic, basic or neutral behavior of a substance that describes a chemical property in terms of pH value logarithm is used.

**3) Is logarithm repeated division?**

Logarithm is repeated division by the base **b** until 1 is reached. The logarithm is the number of divisions by b. Repeated division doesn’t always result in exactly 1.

**4) What is the difference between logarithm and algorithm?**

Algorithm is a step by step process to solve a certain problem whereas logarithm is an exponent.

**5) Why is binary search logarithmic?**

Binary search is a Divide and Conquer method of searching, its key idea is to reduce the search space to half after each comparison to find the key. Thus the search space repeatedly drops by half and the complexity is logarithmic.

**6) What is faster N or log N?**

log N is faster than N as the value of log N is smaller than N.

**7) What is faster O(1) or O(log N)?**

O(1) is faster than O(log N), as O(1) constant time complexity and fastest possible .

**8) What is best case time complexity?**

In the best case constant number of operations need to be performed irrespective of value of N. So time complexity in the best case would be O(1) i.e. Most optimal time complexity.

**Conclusion**

From the above discussion, we conclude that the analysis of an algorithm is very important for choosing an appropriate algorithm and**the**Logarithm order of complexities is one of the most optimal order of time complexities.

**import** **java.io.\***;

**class** **GFG** {

**public** **static** void main (String[] args) {

int N = 16;

int number\_of\_operations = 0;

System.out.print("Logarithmic reduction of N: ");

**for** (int i = N; i > 1; i = i / 2) {

System.out.print(i + " ");

number\_of\_operations++;

}

System.out.println();

System.out.print("Algorithm Runtime for reducing N to 1: " + number\_of\_operations);

}

}

*// Java program for finding the index of 8*

**import** **java.io.\***;

**class** **GFG** {

**static** int steps = 0;

**static** int find\_position(int val, int Arr[], int n)

{

int l = 0, r = n - 1;

**while** (l <= r) {

steps++;

int m = l + (r - l) / 2;

**if** (Arr[m] == val)

**return** m;

**else** **if** (Arr[m] < val)

l = m + 1;

**else**

r = m - 1;

}

**return** -1;

}

*// Driver Code*

**public** **static** void main (String[] args)

{

int Arr[] = { 2, 4, 6, 8, 10, 12, 14, 16 };

steps = 0;

*// Function Call*

int idx = find\_position(8, Arr, 8);

System.out.println("8 was present on index: "+idx);

*// Since the worst case runtime of Binary search is*

*// log(N) so the count of steps must be less than log(N)*

System.out.println("Algorithm Runtime: " + steps);

}

}

*// This code is contributed by Aman Kumar*