# Big O Notation

Big O notation is an assessment of an algorithm’s efficiency. It helps to gauge the amount of work that is taking place. The Big O notation is written like O(N).

The speed of Big O from fastest to slowest is:

O(1) - constant

O(log N) - logarithmic

O(N) - linear

O(N log N) - linearithmic

O(N2) - quadratic

O(2N) – exponential

## The Constant Big O

The Constant Big O is O(1).

**int** x = 9;

This has O(1). However another efficiency with O(1) is this:

**int** x = 9;

x += 12;

System.***out***.println(x);

x -=4;

System.***out***.println(x);

This is 5\*O(1) = O(5) = O(1).

## The Logarithmic Big O

The binary search is a O(log N) function.

**public** **static** **int** search(**int**[] list, **int** target)

{

**int** left = 0, right = list.length-1;

**int** mid = (left+right)/2;

**while**(left<=right)

{

**if**(list[mid] == target)

**return** mid;

**else** **if**(list[mid] < target)

{

left = mid+1;

mid = (left+right)/2;

}

**else**

{

right = mid-1;

mid = (left+right)/2;

}

}

**return** -1;

}

The actual look is O(1)+O(1)+O(log N)+O(1). The slowest term controls the speed, so it is O(log N).

## The Linear Big O

The linear Big O, or O(N), is a normal, everyday loop that is iterative in nature.

**for**(**int** i=1; i<10; i++)

System.***out***.println(i);

This code runs 9 lines of code, but if it was instead i < 100000, it would run 100000 times. Relatively speaking, we say it is based on i < N. N being any value we want. That is why it is consider O(N).

Furthermore, it doesn’t have to grow by a value of one, but rather by a constant value. So, another O(N) algorithm could be

**for**(**int** i=1; i<10000; i+=5)

System.***out***.println(i);

## The Linearithmic Big O

The linearithmic Big O, which is O(N log N), is a nested for loop that has a linear loop and a logarithmic loop in it. The two examples of loops that are Linearithmic is the merge sort and the quick sort.

## The Quadratic Big O

The quadratic Big O is O(N2). This is a nested loop which has a linear loop both as its outer loop and its inner loop.

**for**(**int** r = 0; r < 1000; r+=3)

**for**(**int** c = 0; c <100; c++)

System.***out***.println(r+" "+c);

A selection sort and an insertion sort are considered Quadratic Big O.

## The Exponential Big O

The exponential Big O is O(2n). Never write code that does this! The best example of an exponential Big O is a stooge sort where the programmer randomizes the array and then checks to see if it is sorted. If it isn’t, then the programmer randomizes the array again until it is sorted.

## Why use Big O

Big O is not really a concern in the high school academic world, because the programs being written are so small that it doesn’t really matter. However, in the real world, storage and speed are a very real concern to the programmer and Big O allows the programmer to assess the quality of the code and make decisions about which algorithm to use for a program. If the programmer is writing code that uses a huge amount of data, then a quadratic sort may not be the best choice compared to using a linearithmic sort. Big O is a good way to determine if the algorithm the user envisions is a good algorithm for the problem. College courses will go into more depth of Big O, so it is good for the programmer to become familiar with the basics of the notation.

## Determining Big O Notation

The most restrictive BigO term determines the Big O. In other words, if a section of code is determined to be O(7) + O(N log N) + O(N) + O(N2) + O(log N) + O(13), then the entire section of code is O(N2) because it is the slowest of all terms. This helps the programmer identify where the algorithm could be improved.

## Array Big O runtimes

traversing an array – O(N)

search for an item – O(N) or O(log N)

remove any item location unknown –O(N)

get any item location unknown – O(1)

add item at the end – O(1)

add item at the front –O(N)

## Linked List Big O runtimes

traversing an array – O(N)

search for an item – O(N)

remove any item location unknown –O(N)

get any item location unknown – O(N)

add item at the end – O(N)

add item at the front –O(1)

## Double Linked List Big O runtimes

same as a Linked List except

add item at the end – O(1)

## Binary Tree Big O runtimes

traversing an array – O(N)

search for an item – O(log N)

remove any item location unknown –O(log N)

get any item location unknown – O(log N)

add item at the end – O(log N)

add item at the front –O(1)

## ArrayList Big O runtimes

traversing an array – O(N)

search for an item – O(log N) or O(N)

remove any item location unknown –O(N)

get any item location unknown – O(1)

add item at the end – O(1)

add item at the front –O(N)

## Tree Set Big O runtimes Hash Set Big O runtimes

add – O(log N) add – O(1)

remove – O(log N) remove – O(1)

contains -- O(log N) contains -- O(1)

## Tree Map Big O runtimes Hash Map Big O runtimes

put – O(log N) put – O(1)

get – O(log N) get – O(1)

containsKey -- O(log N) containsKey -- O(1)

## Best Case/Average Case/Worst Case

Linear Search O(1) / O(N) / O(N)

Binary Search O(1) / O(log N) / O(log N)

Selection Sort O(N2) / O(N2) / O(N2)

Bubble Sort O(N2) / O(N2) / O(N2)

Insertion Sort O(N) / O(N2) / O(N2)

Merge Sort O(N log N) / O(N log N) / O(N log N)

Quick Sort O(N log N) / O(N log N) / O(N2)

Heap Sort O(N log N) / O(N log N) / O(N log N)