If you are interested in a career in software development, [http://www.programmerinterview.com/](http://www.programmerinterview.com/) is a site you may want to explore more fully. Here is what they say about the goal of this site:

“The goal of this site is to help programmers successfully prepare for technical interviews. We do this by presenting a variety of questions, grouped by subject matter. This site is a great resource for any level of programmer – whether you are an entry level “fresher” just starting out your career, or an experienced senior level programmer who is looking for more advanced questions, we present a wide variety of real interview questions. These questions have been pulled from our personal interview experiences with big software companies such as Google, Amazon, Accenture, Apple, Intuit, Adobe, NCR, and Oracle, just to name a few.

Rather than simply provide a question and an answer, we provide you with the steps that we took to actually solve the problem. That way, you can see how you would need to structure your thinking when approaching a problem in an interview. We wouldn’t advise you to just remember the answers to the questions on this site. You’re never guaranteed to face the same questions in an interview. So, if you really want to get something out of this site, you should first try solving the problem on your own. If you’re stuck on a certain part of the problem, then read the answer up to that point, and then try to work out the rest of it yourself.”

---


**How does Big-O Notation work, and can you provide an example?**

First and foremost, do not even walk into a software interview without knowing what Big O Analysis is all about – you will embarrass yourself. Big O Notation is simply something that you must know if you expect to get a job in this industry. Here we present a tutorial on Big O Notation, along with some simple examples to really help you understand it. You can consider this article to be sort of a big O notation for dummies tutorial, because we really try to make it easy to understand.

**What is Big O Analysis in computer science – a tutorial:**

When solving a computer science problem there will usually be more than just one solution. These solutions will often be in the form of different algorithms, and you will generally want to compare the algorithms to see which one is more efficient.

This is where Big O analysis helps – it gives us some basis for measuring the efficiency of an algorithm. A more detailed explanation and definition of Big O analysis would be this: it measures the efficiency of an algorithm based on the **time** it takes for the algorithm to run as a function of the input size. Think of the input simply as what goes into a function – whether it be an array of numbers, a linked list, etc.

Sounds quite boring, right?
It’s really not that bad at all – and it is something best illustrated by an example with actual code samples.

**Big O Notation Practice Problems**

Even if you already know what Big O Notation is, you can still check out the example algorithms below and try to figure out the Big O Notation of each algorithm on your own without reading our answers first. This will give you some good practice finding the Big O Notation on your own using the problems below.

**Big O Notation Examples in Java**

Now it’s really time to pay attention – let’s start our explanation of Big O Notation with an actual problem. Here is the problem we are trying to solve:

*Let’s suppose that we want to create a function that, when given an array of integers greater than 0, will return the integer that is the smallest in that array.*

In order to best illustrate the way Big-O analysis works, we will come up with **two** different solutions to this problem, each with a different Big-O efficiency.

Here’s our first function that will simply return the integer that is the smallest in the array. The algorithm will just iterate through all of the values in the array and keep track of the smallest integer in the array in the variable called curMin.

Let’s assume that the array being passed to our function contains 10 elements – this number is something we arbitrarily chose. We could have said it contains 100, or 100000 elements – either way it would have made no difference for our purposes here.

**The CompareSmallestNumber Java function**

```java
int CompareSmallestNumber (int array[ ])  
{
    int x, curMin;

    // set smallest value to first item in array
    curMin = array[0];

    /* iterate through array to find smallest value
    and also assume there are only 10 elements */
    for (x = 1; x < 10; x++)
    {
        if( array[x] < curMin) {
            curMin = array[x];
        }
    }

    return curMin;
}
```
As promised, we want to show you another solution to the problem. In this solution, we will use a different algorithm - we will soon compare the big O Notation of the two different solutions below. What we do for our second solution to the problem is compare each value in the array to all of the other numbers in the array, and if that value is less than or equal to all of the other numbers in the array then we know that it is the smallest number in the array.

### The `CompareToAllNumbers` Java function

```java
int CompareToAllNumbers (int array[]) {
    bool isMin;

    int x, y;

    /* iterate through each element in array, assuming there are only 10 elements: */
    for (int x = 0; x < 10; x++) {
        isMin = true;

        for (int y = 0; y < 10; y++) {
            /* compare the value in array[x] to the other values if we find that array[x] is greater than any of the values in array[y] then we know that the value in array[x] is not the minimum remember that the 2 arrays are exactly the same, we are just taking out one value with index 'x' and comparing to the other values in the array with index 'y' */
            if (array[x] > array[y])
                isMin = false;
        }

        if(isMin)
            break;
    }
    return curMin;
}
```
Now, you've seen 2 functions that solve the same problem - but each one uses a different algorithm. We want to be able to say which algorithm is more efficient using mathematical terms, and Big-O analysis allows us to do exactly that.

**Big O analysis of algorithms**

For our purposes, we assumed an input size of 10 for the array. But when doing Big O analysis, we don't want to use specific numbers for the input size - so we say that the input is of size n.

Remember that Big-O analysis is used to measure the efficiency of an algorithm based on the time it takes for the algorithm to run as a function of the input size.

When doing Big-O analysis, "input" can mean a lot of different things depending on the problem being solved. In our examples above, the input is the array that is passed into the different functions. But, input could also be the number of elements of a linked list, the nodes in a tree, or whatever data structure you are dealing with.

Since input is of size n, and in our example the input is an array - we will say that the array is of size n. We will use the 'n' to denote input size in our Big-O analysis.

So, the real question is how Big-O analysis measures efficiency. Basically, Big-O will want to express how many times the 'n' input items are 'touched'. The word 'touched' can mean different things in different algorithms - in some algorithms it may mean the number of times a constant is multiplied by an input item, the number of times an input is added to a data structure, etc.

But in our functions CompareSmallestNumber and CompareToAllNumbers, it just means the number of times an array value is compared to another value.

**Big O notation time complexity**

In the function CompareSmallestNumber, the n (we used 10 items, but lets just use the variable 'n' for now) input items are each 'touched' only once when each one is compared to the minimum value. In Big O notation, this would be written as O(n) - which is also known as linear time. Linear time means that the time taken to run the algorithm increases in direct proportion to the number of input items. So, 80 items would take longer to run than 79 items or any quantity less than 79. Another way to phrase this is to say that the algorithm being used in the CompareSmallestNumber function has order of n time complexity.

Subscribe to our newsletter for more free interview questions.
You might also see that in the CompareSmallestNumber function, we initialize the curMin variable to the first value of the input array. And that does count as 1 ‘touch’ of the input. So, you might think that our Big O notation should be O(n + 1). But actually, Big O is concerned with the running time as the number of inputs - which is 'n' in this case - approaches infinity. And as 'n' approaches infinity the constant '1' becomes very insignificant - so we actually drop the constant. Thus, we can say that the CompareSmallestNumber function has O(n) and not O(n + 1).

Also, if we have n^3 + n, then as n approaches infinity it's clear that the "+ n" becomes very insignificant - so we will drop the "+ n", and instead of having O(n^3 + n), we will have O(n^3), or order of n^3 time complexity.

**What is Big O notation worst case?**

Now, let's do the Big O analysis of the CompareToAllNumbers function. The **worst case** of Big O notation in our example basically means that we want to find the scenario which will take the longest for the CompareToAllNumbers function to run. When does that scenario occur?

Well, let's think about what the worst case running time for the CompareToAllNumbers function is and use that as the basis for the Big O notation. So, for this function, let's assume that the smallest integer is in the very last element of the array - because that is the exact scenario which will take the longest to run since it will have to get to the very last element to find the smallest element. Since we are taking each element in the array and comparing it to every other element in the array, that means we will be doing 100 comparisons - assuming, of course, that our input size is 10 (10 * 10 = 100). Or, if we use a variable "n" to represent the input size, that will be n^2 'touches' of the input. Thus, this function uses a O(n^2 ) algorithm.

**Big O analysis measures efficiency**

Now, let's compare the 2 functions: CompareToAllNumbers is O(n^2) and CompareSmallestNumber is O(n). So, let's say that we have 10,000 input elements, then CompareSmallestNumber will 'touch' on the order of 10,000 elements, whereas CompareToAllNumbers will 'touch' 10,000 squared or 100,000,000 elements. That's a huge difference, and you can imagine how much faster CompareSmallestNumber must run when compared to CompareToAllNumbers - especially when given a very large number of inputs. Efficiency is something that can make a huge difference and it's important to be aware of how to create efficient solutions.

In an interview, you may be asked what the Big-O of an algorithm that you've come up with is. And even if not directly asked, you should provide that information in order to show that you are well aware of the need to come up with an efficient solution whenever possible.