TASK

Computer Science Fundamentals and Big O Notation

Visit our website

Introduction

WELCOME TO THE COMPUTER SCIENCE FUNDAMENTALS AND BIG O NOTATION TASK!

In this task, you will be introduced to Big O notation, which is used to describe the performance or complexity of an algorithm.

WHAT IS COMPUTER SCIENCE?

As the name suggests, computer science is a science that was born through the invention of computers. Computer Scientists are mostly concerned with software and software systems. This includes their theory, design, development, and application.

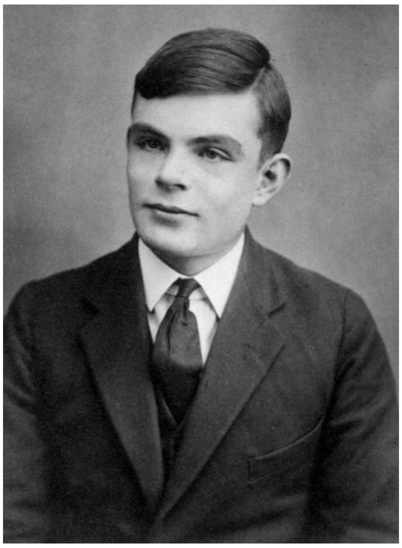
Although it is important to know how to program, programming is only a small element of computer science. A Computer Scientist should understand the "why" behind computer programs. Computer Scientists design and analyse algorithms to invent new ways to solve problems and study the performance of computer hardware and software.

Computer science can be seen on a higher level as a science of problem-solving and computational thinking. Computer Scientists must be adept at modelling and analysing problems. They must also be able to design solutions and verify that they are correct.

Computer science can be used in various other disciplines like engineering, healthcare, and business. Finding solutions to problems these disciplines face requires both computer science expertise and knowledge of that particular industry. Thus, Computer Scientists often become proficient in many other fields.

THE FATHER OF MODERN DAY COMPUTING

Alan Turing (1912 – 1954) was a British Mathematician, Logician, and Cryptographer. He is considered by many to be the father of modern computer science. He contributed to the design of some of the earliest electronic, programmable, digital computers.



*Alan Turing in 1927, age 16 (**http://turingarchive.org/**)*

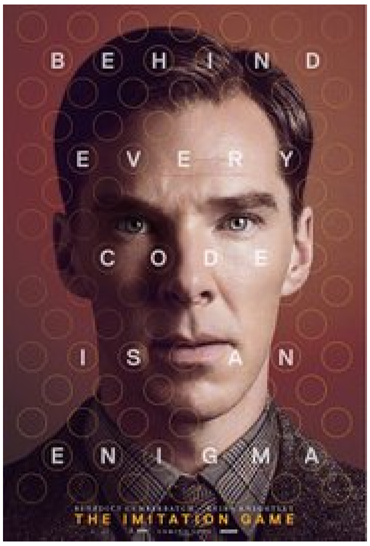
During the Second World War, Turing was recruited by the military to head a classified mission at Bletchley Park. This mission was to crack the Nazi's Enigma machine code which was used to send secret military messages. Many historians believe that breaking the Enigma code was key to bringing an end to the war in Europe.

In 1936, Turing published a paper called "On Computable Numbers" that is now recognised as the foundation of computer science. He proved in this paper that there cannot exist any universal algorithmic method of determining truth in mathematics and that mathematics will always contain undecidable propositions. The "Universal Turing Machine" was also introduced in this paper. This machine is capable of computing anything that is computable. The central concept of the modern computer was based on Turing's paper.

Alan Turing is famous for being a homosexual in a time when it was illegal. This led to him being convicted of acts of gross indecency in 1952. He was given a choice between prison or a course of hormone therapy. He chose hormone therapy, which eventually rendered him impotent.

Turing died on June 7, 1954. The post-mortem exam concluded that the cause of death was asphyxia due to cyanide poisoning and his death was ruled a suicide. However, many people suggest foul play and that he was actually assassinated due to his sexuality.

British Prime Minister Gordon Brown made an official public apology in 2009 on behalf of the British government. Queen Elizabeth II granted him a posthumous pardon in 2013.



*The Imitation Game (**imdb.com**)*

In 2015 Alan Turing's life was brought to the big screen in the movie The Imitation Game, starring Benedict Cumberbatch and Keira Knightley.

ALGORITHMS

According to *Math Vault* glossary (2020), "an algorithm is a finite sequence of **well-defined**, computer-implementable instructions, typically to solve a class of problems or to perform a computation." One of the characteristics of algorithms is that they do not require any intelligence to carry out; they are mechanical processes in which each step follows from the last according to a simple set of rules. In other words, an algorithm is like a recipe. It is a step-by-step method for solving a problem. You have probably already encountered many non-technical algorithms in your everyday life, such as a recipe to bake a cake or instructions on how to assemble a piece of furniture.

Algorithms can be written in any commonly understood language, but are most often expressed as programs in a programming language or as pseudocode in computer science. Ideally, they should be easy to understand and easy to turn into a working program.

BIG O NOTATION

In order to start taking advantage of algorithms, you need to know how fast they are so you can choose the most efficient one. In computer science, Big O notation is used to describe the performance or complexity of an algorithm. Specifically, Big O describes the worst-case scenario and can be used to describe the execution time required or the space used by an algorithm.

Big O doesn't use measurements of time, such as seconds or minutes, to determine how fast an algorithm is. Instead, it uses the number of operations, and how these operations grow over time.

In order to use Big O notation, you will need to know some mathematics. Most notably the concept of logarithms. There is no need to panic if you are uncomfortable with mathematics, however. You don't need to know that much about it. We will explain the basics of logarithms next.

LOGARITHMS

A logarithm is another way of thinking about exponents. For example, 2 raised to the 3rd power (2^3) is 8. We can express this as $2^3 = 8$. Now, suppose I asked, "2 raised to which power equals 8?" The answer would obviously be 3. We can express this by the logometric equation $\log_2(8) = 3$ which is read as "log base two of eight is three".

The general definition of a logarithm is as follows:

$$\text{Log}_n(x) = y$$

Where:

- **n** is the base
- **y** is the exponent
- **x** is the argument

For example, let's look at the following Logarithm:

$$\text{Log}_5(100)$$

This expression is asking "how many times do I need to multiply 5 by itself in order to reach the number 100?"

Well, we know that 5*5*5 = 125 (5^3=125) so we need to raise 5 to the power of 3 in order to get to a number that is equal or greater than 100. The result of this logarithm will, therefore, be 3.

```
Log₅(100) = 3
```

As you probably noticed, the calculation was not exactly equal; we went past 100 and ended up at 125. We think of 3 as the worst-case number of steps involved when calculating that particular item. In this case, the number 100 represents the number of items we have to execute our algorithm against. Therefore, if you were using an algorithm that runs over a collection of items, the length of the items would be 100.

BACK TO BIG O NOTATION

As previously stated, Big O notation tells you how fast an algorithm is. It expresses the runtime of the algorithm, not in terms of a measurement of time, but how quickly it grows relative to the input, as the input gets larger. To break that down:

- **How quickly the runtime grows** — The runtime depends on many things, such as the speed of the processor and other programs the computer is running, so we use Big O notation to talk about how quickly the runtime grows and do not measure the actual runtime.
- **Relative to the input** — We represent our speed in terms of the size of the input, which we call "n". We can, therefore, say that the runtime grows "depending on the size of the input" ($O(n)$) or "depending on the square of the size of the input" ($O(n^2)$).
- **As the input gets larger** — Our algorithm may have steps that seem expensive when n is small, but as n gets larger these steps become insignificant. For Big O analysis, we care most about the steps that grow fastest as the input grows, because everything else quickly becomes insignificant as n gets very large

As the name suggests, Big O notation is written with a big “O”. The “O” is then followed by some function of *n* between brackets. Let's now try to understand how Big O notation works using some examples. (Remember that *n* represents the size of the input.)

O(1)

O(1) describes an algorithm that will always execute at the same time, regardless of how large or small the input is. For example:

```
public static void printItems(int[] arrayOfItems) {
    System.out.println(arrayOfItems[0]);
}
```

In the example above we can see that it doesn't matter whether the input array has 1 item or 1000 items, the method would still just require one step.

O(n)

An algorithm whose performance grows linearly and is directly proportional to the size of the input is described by *O(n)*. For example:

```
public static void printItems(int[] arrayOfItems) {
    for (int item : arrayOfItems) {
        System.out.println(item);
    }
}
```

In the example above, if the array has 10 items, we have to print 10 times, whereas if it has 1 000 items, we have to print 1 000 times.

O(n^2)

An algorithm whose performance is directly proportional to the square of the size of the input is represented by $O(n^2)$. This is common with algorithms that involve nested loops. For example:

```
public static void printOrderedPairs(int[] arrayOfItems) {
    for (int firstItem : arrayOfItems) {
        for (int secondItem : arrayOfItems) {
            System.out.println(firstItem + ", " + secondItem);
        }
    }
}
```

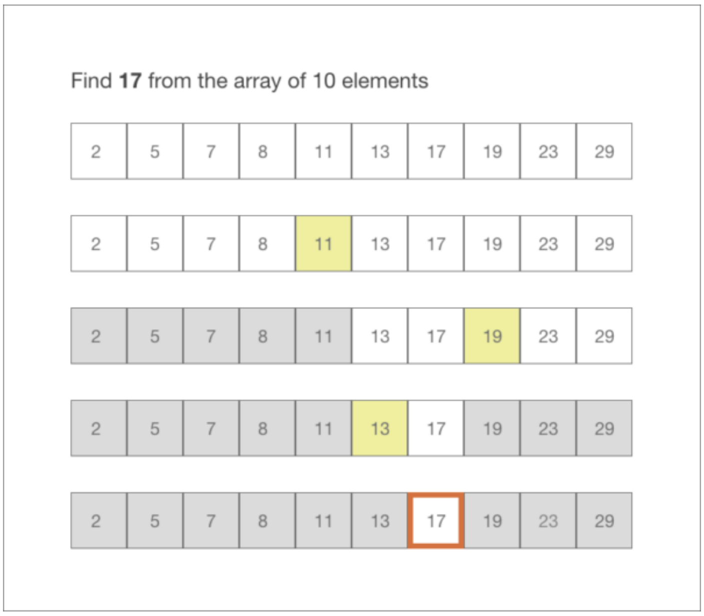
In the example above we have two nested loops. If the array contains n items, the outer loop runs n times and the inner loop runs n times for each iteration of the outer loop. This gives us n^2 total print statements. If the array has 10 items, for example, we have to print 100 times.

Note: You might not recognise the syntax of the for loop in the code above. This for loop is known as an enhanced for loop. We will be using this notation for the rest of this task so you should become familiar with it. You can read more about them **here**.

O(log n)

According to Weisstein (2020), **binary search** is "a searching algorithm which works on a sorted table by testing the middle of an interval, eliminating the half of the table in which the key cannot lie, and then repeating the procedure iteratively". If the value you are searching for is the same as the middle element, the search will return successfully. If the value you are searching for is higher than the middle element, it will take the upper half of the data set and perform the same operation against it. If the value you are searching for is lower than the middle element, it will take the lower half of the data set. The binary search will continue to halve the data set until the value has been found, or until the data set can no longer be halved.

The below diagram shows how this work. You will see that for each iteration the yellow element indicates roughly the middle point of the algorithm thereby continuously halving the array that it is searching until there are no options left. There is no need to search through every element in the array because we can tell whether the middle point is greater than or less than the number we are searching for which eliminates half of the remaining array on each iteration.



This algorithm is described by O(log n). The iterative halving of data sets in a binary search produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increases. Doubling the size of the input does not have a large effect on its growth. This is because, after a single iteration of the algorithm, the data set will be halved and, therefore, equal to an input data set half the size. Therefore, this algorithm is efficient when dealing with large inputs.

n Can be the Actual Input or the Size of the Input

Look at the methods below:

```
public static void sayHi(int n) {
    for (int j = 0; j < n; j++) {
        System.out.println("hi");
    }
}

public static void printItemsInArray(int[] theArray) {
    for (int item : theArray) {
        System.out.println(item);
    }
}
```

The runtime for both of these methods is *O(n)*, even though one takes an integer as its input and the other takes an array. Sometimes, *n* is an actual number that's an input to our method, and other times it is the number of items in an input object.

Dropping Constants

When you're calculating the Big O of something, you can just drop the constants. For example:

```
public static void printAllItemsTwice(int[] theArray) {
    for (int item : theArray) {
        System.out.println(item);
    }

    for (int item : theArray) {
        System.out.println(item);
    }
}
```

The above example is *O(n + n)*, which simplifies to *O(2n)*, which we can in turn just call *O(n)*.

Another example:

```
// This method prints the first item in the array, then prints out hi 100
times
public static void printHi(int[] myArray) {

    System.out.println(myArray[0]);

    int middleIndex = myArray.length / 2;
    int index = 0;

    while (index < middleIndex) {
        System.out.println(myArray[index]);
        index++;
    }

    for (int i = 0; i < 100; i++) {
        System.out.println("hi");
    }
}
```

The method above is *O(1+n/2+100)*, which we can further simplify to *O(n)*.

You might be asking, "How can we just drop these constants?". Remember that we are looking at what happens as *n* gets arbitrarily large. As *n* gets infinitely bigger, adding 100 or dividing by 2 has little overall effect.

Dropping Less Significant Terms

You can also drop less significant terms. For example:

```
 // This method prints out all numbers and then prints out all the sums of
pairs

 public static void printNumbersAndPairSums(int[] numArray) {

    System.out.println("these are all the numbers:");
    for (int number : numArray) {
        System.out.println(number);
    }

    System.out.println("and these are all their sums:");
    for (int firstNumber : numArray) {
        for (int secondNumber : numArray) {
```

```
            System.out.println(firstNumber + secondNumber);
        }
    }
}
```

The runtime for the above example is *$O(n + n^2)$*, however, we can just call it *$O(n^2)$*. You can simply drop all terms with a lower exponent and only keep the term with the highest exponent.

Other examples are:

- $O(n^2/2 + 100n)$ is $O(n^2)$
- $O(n^3 + 76n^2 + 7282)$ is $O(n^3)$
- $O((n + 100) * (n + 10))$ is $O(n^2)$

Remember that we can do this because the less significant terms have an insignificant effect as n gets larger.

Worst-Case Scenario

Big O describes the worst-case scenario and sometimes the worst-case runtime is much worse than the best case runtime. For example:

```
public static boolean contains(int[] haystack, int needle) {

    // determines whether the haystack contains the needle
    for (int n : haystack) {
        if (n == needle) {
            return true;
        }
    }

    return false;
}
```

Let's say that there are 100 items in the haystack. The needle might be the first item in the haystack, in which case the method should return true in just a single iteration. However, the runtime for the example above is *$O(n)$*.

Formal Definition of Big O

Now, let us develop a formal definition of Big O

Consider the following functions:

- $g(n) = n^2$
- $f(n) = n^2 + 4n + 20$

As you can see, the function g does not contain a linear term. In other words, it doesn't contain a term with n. Now look at the table below:

n	$g(n) = n^2$	$f(n) = n^2 + 4n + 20$
10	100	160
50	2500	2720
100	10 000	10 420
1000	1 000 000	1 004 020
10000	100 000 000	100 040 020

As you can see, as n becomes larger, the term $4n + 20$ in $f(n)$ becomes insignificant, while the term n^2 becomes the dominant term. Therefore, we can predict the behaviour of $f(n)$ for large values of n by looking at the behaviour of $g(n)$. If the complexity of a function can be described by the complexity of a quadratic function without the linear term, we say that the function is $O(n^2)$.

Now, let f and g be real-valued functions. Assume that f and g are nonnegative for all real numbers n, $f(n) >= 0$ and $g(n) >= 0$.

Definition: We say that $f(n)$ is Big O of $g(n)$, written $f(n) = O(g(n))$, if there exists positive constants c and n_0 such that $f(n) <= cg(n)$ for all $n >= n_0$.

In general, the following theorem exists:

Theorem: Let $f(n)$ be a nonnegative real-valued function such that
$f(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$,
where a_i shows real numbers, $a_m \neq 0$, $n >= 0$, and m is a nonnegative integer. Then $f(n) = O(n^m)$.

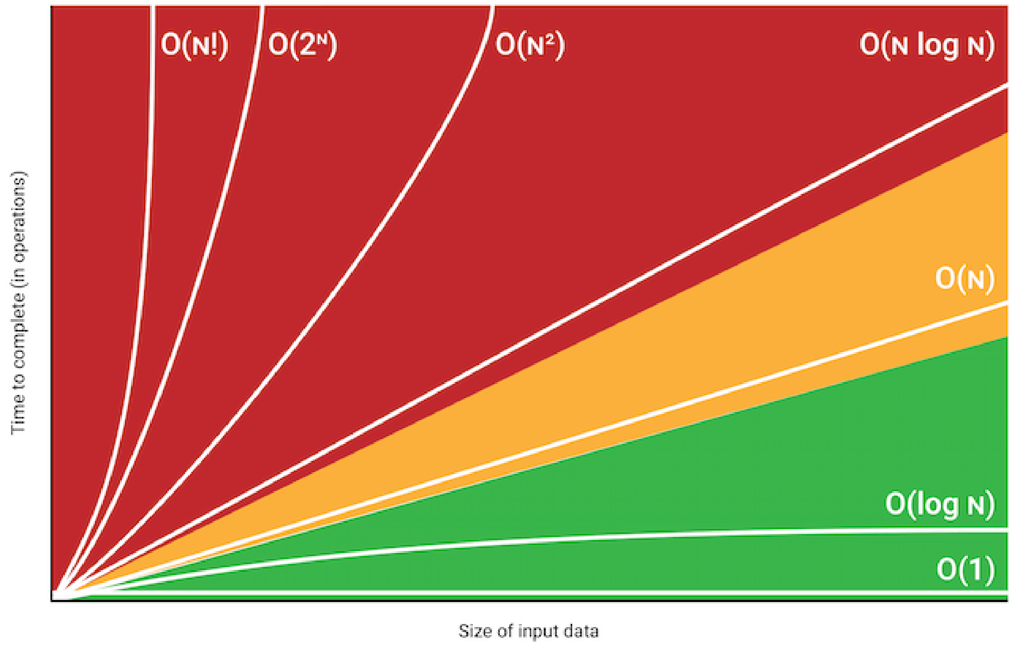
In the examples below we use the theorem to establish the Big O of certain functions:

Function	Big O
$f(n) = 2n + 1$	$f(n) = O(n)$
$f(n) = n^2 + 5n + 1$	$f(n) = O(n^2)$
$f(n) = 23n^8 + 30$	$f(n) = O(n^8)$
$f(n) = 89n^{123}$	$f(n) = O(n^{123})$

Summary of Some Common Big O Runtimes

Big O	Growth Rate
$O(1)$	The growth rate is constant and so does not depend on *n*.
$O(\log n)$	The growth rate is a function of *log n*. Because a logarithm function grows slowly, the growth rate is slow.
$O(n)$	The growth rate is linear. The growth rate is directly proportional to the size of the input.
$O(n\log n)$	The growth rate is faster than a linear algorithm
$O(n^2)$	The growth rate increases rapidly with the size of the input. The growth rate is quadrupled when the input size is doubled.

You can see the comparison of the runtime of each of these in the diagram below.



Compulsory Task

Answer the following questions in a file called **bigo.java**:

- Write an O(n) algorithm that sequentially inserts an element into a list
- Write an $O(n^2)$ algorithm that iterates over a 2D array of integers.
- Write an O(log n) algorithm that sequentially inserts an element into a sorted list.

Optional Bonus Task

Answer the following questions:

- Create a text file called **runtime.txt**. Inside, outline an algorithm that checks all possible combinations of characters in a string.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

References:

No Author. (2020). The Definitive Glossary of Higher Mathematical Jargon. Retrieved 7 February 2020, from https://mathvault.ca/math-glossary/#algo

Weisstein, Eric W. (2020). "Binary Search." From *MathWorld* -- A Wolfram Web Resource. Retrieved 24 February 2020, from http://mathworld.wolfram.com/BinarySearch.html