Complexity and Stacks/Queues

# Complexity

Complexity is the measure of resources used by an algorithm. These resources that we measure can be almost anything but the most common resources to measure is time and space. It is important to measure these properties of an algorithm as it gives a concrete way of comparing algorithms.

## Time Complexity

When we measure time complexity, we are looking at how fast an algorithm runs. When I say “how fast” I don’t mean in seconds or minutes, I mean how many elementary operations does this algorithm carry out. Elementary operations are the core basic operations of a programming language, for example:

* Addition
* Subtraction
* Division
* Multiplication
* Comparison

When writing an algorithm, generally it will take an input, such as numbers an array etc. When calculating the complexity of an algorithm we want to see how the amount of elementary operations scales with the size of an input, this input is often called “n” and we denote it by

So for the example below:



As you can see this function searches through an array and finds the value and returns its index. “arr” is the array and “n” is the size of the input. For the worst case we find the maximum amount of operations this function will do, so this would be when the value we are looking for is in the last index of the array. This means the algorithm will do “n” amount of comparisons then finish. This gives:

Another quick example of this is bubble sort:



I have purposely implemented this badly, but this makes it a bit easier to calculate its complexity. If you don’t like maths look away now. So the worst case for this is if the array is in descending order, this means its in the opposite order to what it should be.

The part inside the if statement takes 3 operations and the if statement takes 1. As the array is in reverse order this if statement will always be true, so the inner for loop will do have a complexity of . There is another variable “i” in the equation which will have to be eliminated. The outer for loop will call the inner one “n” times and “i” will range from 0 to . So if we write this as a summation series:

Then using the summation equations:

As fun as it is to do all this maths, it’s a long task to do this for each algorithm, bearing in mind bubble sort is a relatively simple algorithm, it also isn’t very useful for comparing algorithms as comparing it includes lots of unnecessary information. This is where Big O notation comes into it.

## Big O

The formal definition of Big O is that it is a notation used to describe the limiting factor of an equation. The limiting factor of an equation is just the most significant part of the equation as the argument gets large. So, for the example above:

As gets large the term will be insignificant compared to , so we remove it leaving us with:

In Big O notation, we want to classify each equation into a category here are some:

= Constant runtime

– Linear runtime

- quadratic runtime

– Cubic

– Logarithmic

This is not all but these are some of the most common. Can you guess which one our equation fits into?

This is a lot more useful for comparing algorithms and a lot easier to do. As you can look at the two examples above and see what category they fit into without needing to calculate its complexity. So for the basic search, you can see that the for loop grows with “n” so it’s safe to assume that this is an algorithm, and then with bubble sort you can see that there is two for loops nested which both increase with “n” so it is safe to say that it is an algorithm.

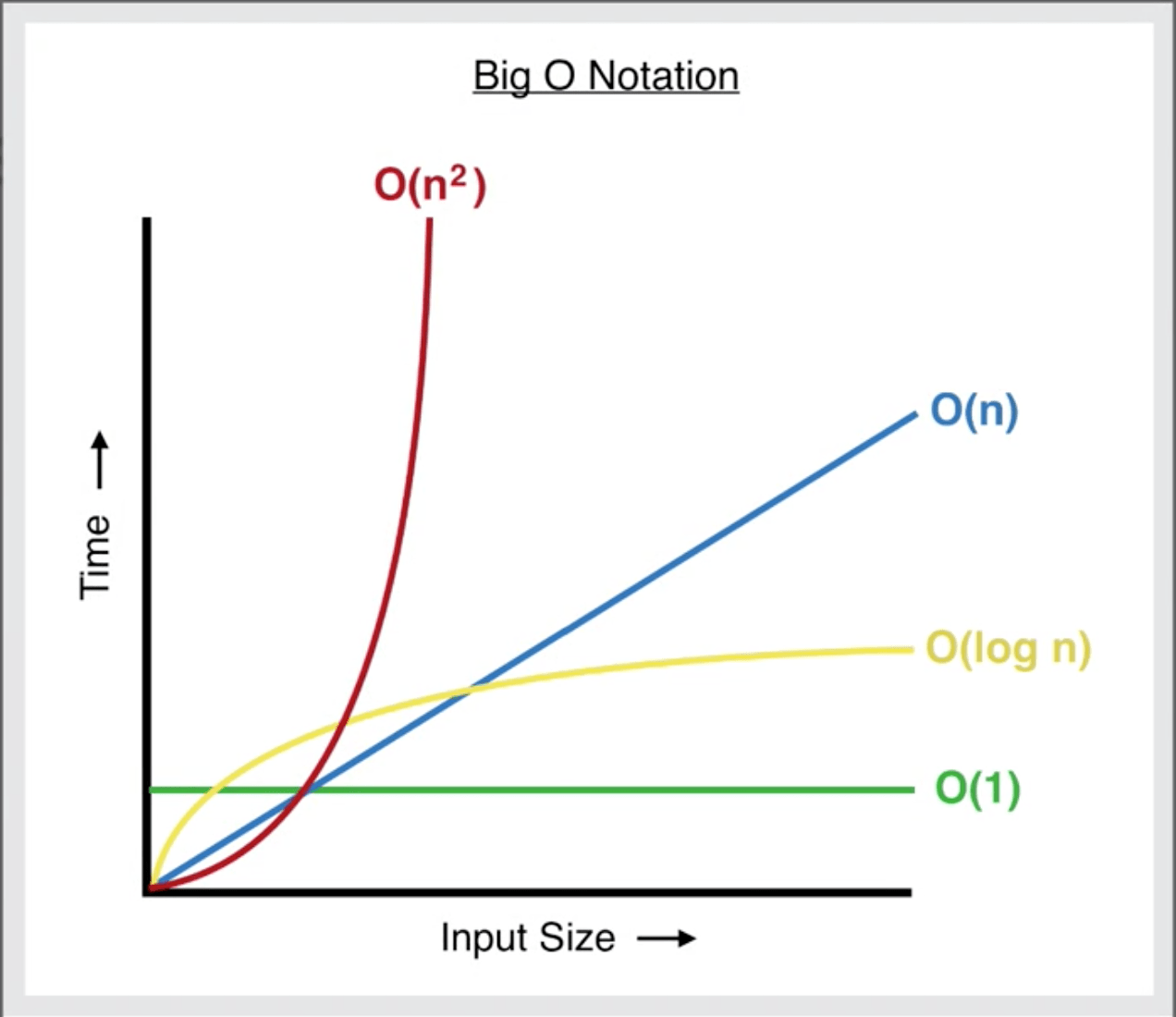


Image from: <https://levelup.gitconnected.com/big-o-time-complexity-what-it-is-and-why-it-matters-for-your-code-6c08dd97ad59>

## Space complexity

This is very similar to time complexity so I will not go over this today but allow you to do your own research into it.

## Data Structures and Big O

Data structures store data in different ways (obviously), but the characteristics of how they store them is what we are going to explore now. The operations we are going to focus on here are Access, Search, insert and deletion. These are the worst-case complexities for these operations in different data structures

### Array

Access: O(1) this is because given the index of a value in array the computer can access it without searching

Search: O(n) because there is no order to the values in an array so you must search through each index until its found

Insertion and deletion: O(n) as when you insert a value into an array all subsequent values must be moved along and the opposite for deletion

### Stack

Access: O(n) As this is worst case the value you would be looking for is at the bottom of the stack meaning n values will need to be popped

Search: O(n) As each value will need to be popped until the value is found

Insertion and Deletion: O(1) As the only value that can be deleted is at the end of the stack and the only place a value can be inserted is at the end, so not necessary to move any other data around

### Queue

This is the same as a stack but for deletion the next value will be removed from the front of the stack meaning no data needs to be moved around.

# Priority Queues

For this next section I’m going to assume you know about stacks and queues already as it’s they’re common data structures to learn about at A Level (or the equivalent).

So now we will hopefully be introducing to you something a bit more “advanced”.

Priority Queues are Queues that have order for which each value should be popped. This means that every index in the array is ordered by an attribute generally; in order from lowest to highest. For example:



If every value was then popped it would be in this order.



It is important that no matter the order that the numbers were added in, when they are popped the lowest (or whatever order they are in) always comes out first.

# Challenge time

Now you know how a Priority queue is supposed to work, we want you to try and implement it. Go to the link posted in the chat to see the challenge. For this don’t worry about actually submitting your code to the website just read the problem and see if you can implement it yourself and we’ll discuss the solutions together.

### My (basic) implementation

A basic implementation of this would look like:



This works by just storing the values in the order they come in, then when popped it searches the list for the lowest value, removes it and returns it – not very efficient.