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

Notice how square root and exponentiate aren’t there, these are technically not elementary operations (along with sin, cos, tan, etc) but to not over complicate things now assume they are.

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.

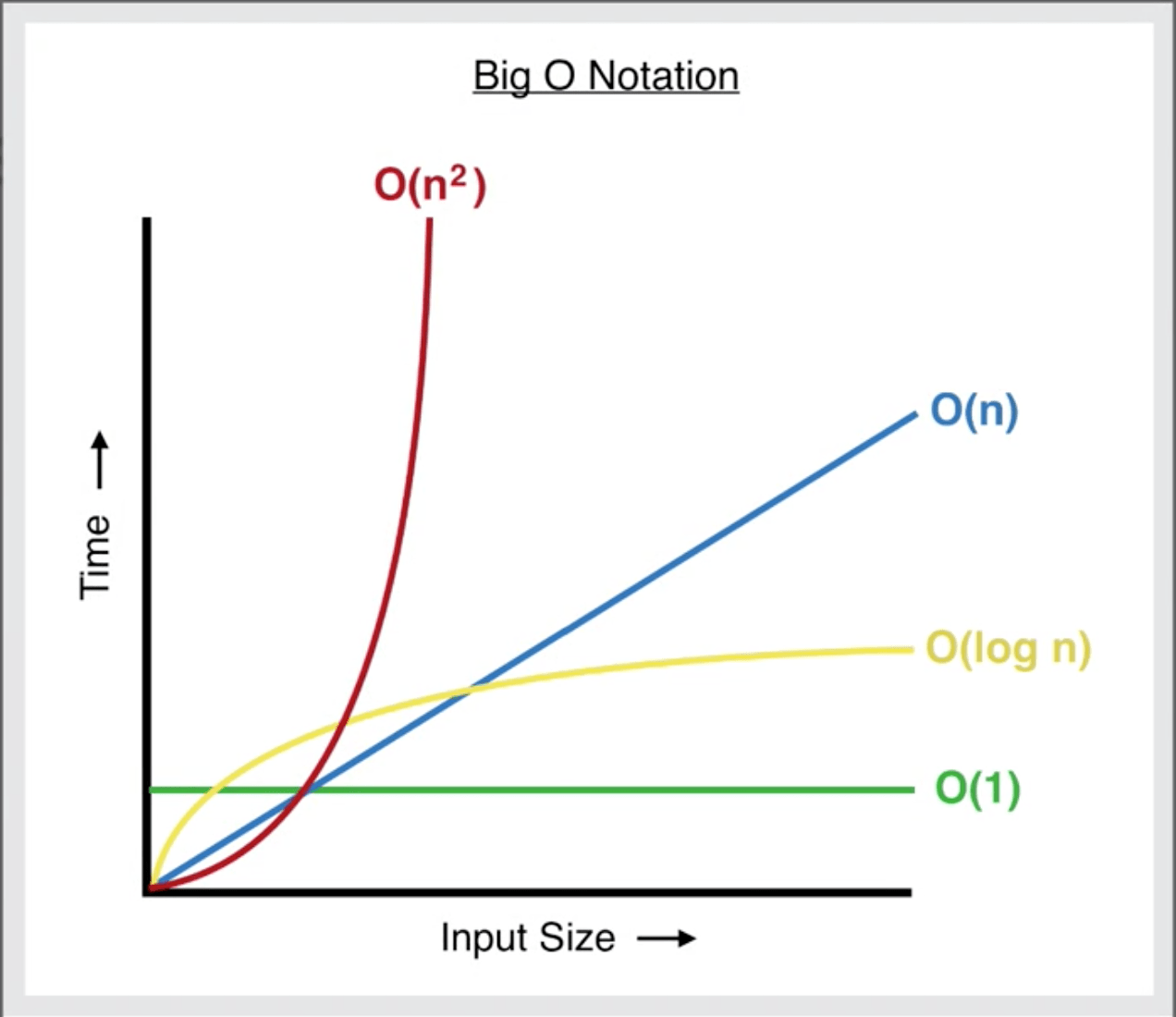


Image from: <https://levelup.gitconnected.com/big-o-time-complexity-what-it-is-and-why-it-matters-for-your-code-6c08dd97ad59>

This graph here shows how a few of the complexities scale with “n”.

In future sessions we will show the usefulness of knowing these complexities with some actual programming when we dive into some more advanced algorithms.

# 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.

## Implementations

Now we want you guys to go and try and implement this in anyway you feel works, there isn’t going to be a test in a minute scoring how you did or anything, we just want to see how you may do this.

### 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.

# If there is time, then do this stuff

# Trees and Heaps

A heap is an efficient way to implement a priority queue but before that we need to learn a bit about trees.