Complexity and Stacks/Queues

# Introduction

Before we get started with Big O and complexity, I want to talk a little bit about how we are going to teach it. To start off with this will be a practical introduction to this topic, we are going to use complexity and big O as a tool to compare algorithms. There is a lot to this topic that we are going to simplify, firstly because we are limited in time and secondly and more importantly because we are not lecturers.

# Complexity

Complexity is the measure of resources used by an algorithm.

This resource can be pretty much anything that the algorithm uses, but we will be focusing on time and then touch on space.

Why? 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 long it takes an algorithm to run.

When I say “how long” 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

The reason we don’t use units of time is because it is far too dependent on context, such as what device we are using, what language etc.

When writing an algorithm, generally it will take an input, such as numbers in an array.

The size of this input it often given the name “n”.

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.

“val” is the value to look for, “arr” is the array and “n” is the size of the input.

We are going to calculate this algorithms worst-case complexity, this means for the input “n” what’s the most amount of operations it could do.

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.

Lets go through this algorithm step by step:

The for loop in theory does 3 things, defines “i”, compares “i” to the condition (is “i” less than “n”) and then increments “i”.

“i” is defined once, “i” is incremented “n” times then finally “i” is compared “n+1” times.

This gives a complexity for just the for loop of 2n + 2.

Then for the code inside the for loop, as this is the worst-case the value we are looking for will be at the end of the array, so this will do “n” comparisons then stops

This finally gives:

As you can probably tell that was a bit of a lengthy process, for a very simple algorithm.

Also it is very specific, and we are about to see why this can be a problem:

Here is bubble sort



Lets try and find the complexity of this algorithm now:

We are again going to find the worst-case complexity, so when the array of values to be sorted is in reverse order.

Lets start on the inner nested loop.

As this is worst-case the array of numbers will be in reverse order, meaning the if statement here will always be true.

So as well as the 1 op for the comparison there is 3 ops for the swapping of values. This gives 4.

This is then going to be run by the for loop “n – i – 1” times.

And now we run into our problem.

We have a second variable “i”, this makes things a bit more difficult to solve as the amount of iterations the inner for loop does is variable.

This is now where Big O comes into it. We’ll come back to this equation in a moment.

## Big O notation

Big O, in computer science, is a notation for showing complexity of an algorithm.

To show you this there is going to be a little bit of maths:

Given you have the equation and , then

If for

Where and are constants.

What this means is that c \* f(n) is the upper bound to g(n) when n > n0.

This can be more graphically shown by stealing an image from one of our lecturers.

Now this can be used to help us with the problem we were facing earlier with bubble sort.

We had 4 operations being done “n – i – 1” times, but using big O notation we can overestimate the complexity. So we can change “n – i – 1” into just “n” as this will give a larger complexity.

Lets carry on with the rest of the algorithm:

The inner for loop will do “2n + 2” operations and will also be run “n” times.

Then the outer for loop will also do “2n + 2” ops and will be run once.

So all in all:

Onto another important part of Big O is that we want to be able to easily compare algorithms, and like I said earlier the small specific factors we don’t really care about, so usually when calculating Big O we just keep the highest order of “n” and discard the rest , so this becomes:

Or we can say that this equation has a quadratic runtime.

Any questions?

Using Big O notation we can also simplify the root we take to get to the answer.

For the example above instead of counting each individual operation the algorithm takes we can just look at the important parts, which are in this case the for loops.

Each of these for loops we can see increases linearly in the amount of iterations it does with “n”.

So as “n” increases so does the amount of operation.

We then can just ignore the 4 ops it does in does inside the loops and how many steps a for loop takes and skip straight to the answer of

Big O will often simplify down into a few categories here are some of them:

- Constant runtime

– Linear runtime

- quadratic runtime

– Cubic

– Logarithmic

This is not all but these are some of the most common.

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 fully.

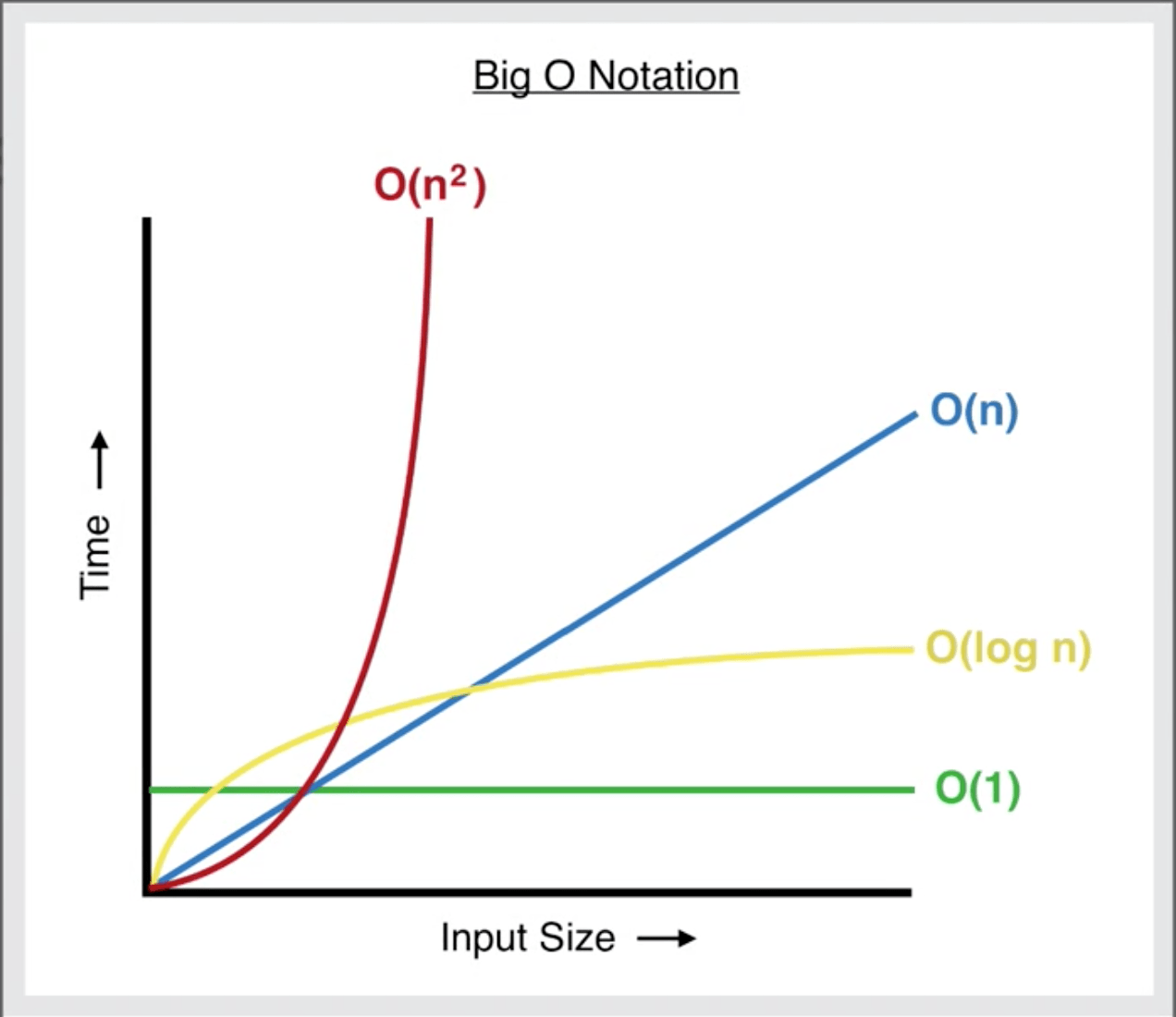


Image from: <https://levelup.gitconnected.com/big-o-time-complexity-what-it-is-and-why-it-matters-for-your-code-6c08dd97ad59>

## 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.

The link:

<https://www.hackerearth.com/practice/data-structures/trees/heapspriority-queues/practice-problems/algorithm/monk-and-champions-league/>

### 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.