# Complexity and Stacks/Queues

## Big O and Complexity

Knowing how to quantify the speed of an algorithm is important. As in my mind coming to uni I had it in my mind that programming language was a significant part of how fast my program run and while this is true in part its not the whole story. The more significant part of how fast your program runs is the algorithm you write for it. For example:

Here are two function, both search through an ordered list of numbers looking to find the index of a value





Then using this code, you can test the speed, this works by generating an array of length “n” which has integers in increasing order. Then picks a random number in each of them and calls these functions to find them and then records how long it takes to do this 100 times:



When this code is run it gave these results:



And as you can see the binary search outperformed the linear searched by many orders of magnitude.

This was all down to the algorithm. I will not explain how binary search works here as it’s a bit long, but the point is it an O(log(n)) algorithm whereas the basic search is an O(n) algorithm. Meaning that as the input “n” increases the amount of time taken to produce an output will scale by about log(n) for binary search and n for linear search.

### So how do you actually do it?

This can be easily done for most algorithms that you will do. So the goal is to try and find how the time taken to run the algorithm scales as “n” increases (n being the size of the input, generally the length of an array). So for the linear search above, assuming that the value you are looking for is in a random place in the array, on average it will be half way through the array. This means that on average the program will need to perform operations until it finds the given value.

So we know have a mathematical function:

Now we want to remove all terms of the equation apart from the “most significant” term. This means as “n” becomes large which term is the largest, our function only has one term so that’s easy but for example if you had

The most significant term of this is the .

Back to the original equation.

Although this may seem counter intuitive, we know remove the coefficient in front of the term. This is because we want to measure how much the time taken will scale with “n” and as the coefficient doesn’t affect this, we remove it.

This leaves us with “n”. You would write this as , this algorithm is said to have Linear Runtime.

Just as another quick example:



Analysing the complexity:

Inner for loop is called times.

Inner for loop does comparison.

(if you aren’t a big fan of maths look away now)

Now as we strip the unimportant parts to give us:

or

Generally you will calculate the complexity this way, as you can generally guess from just looking at the algorithm, for the example above you can see that there is two nested for loops that both increase in the amount of operations as “n” increases, so you could assume that this algorithm is .

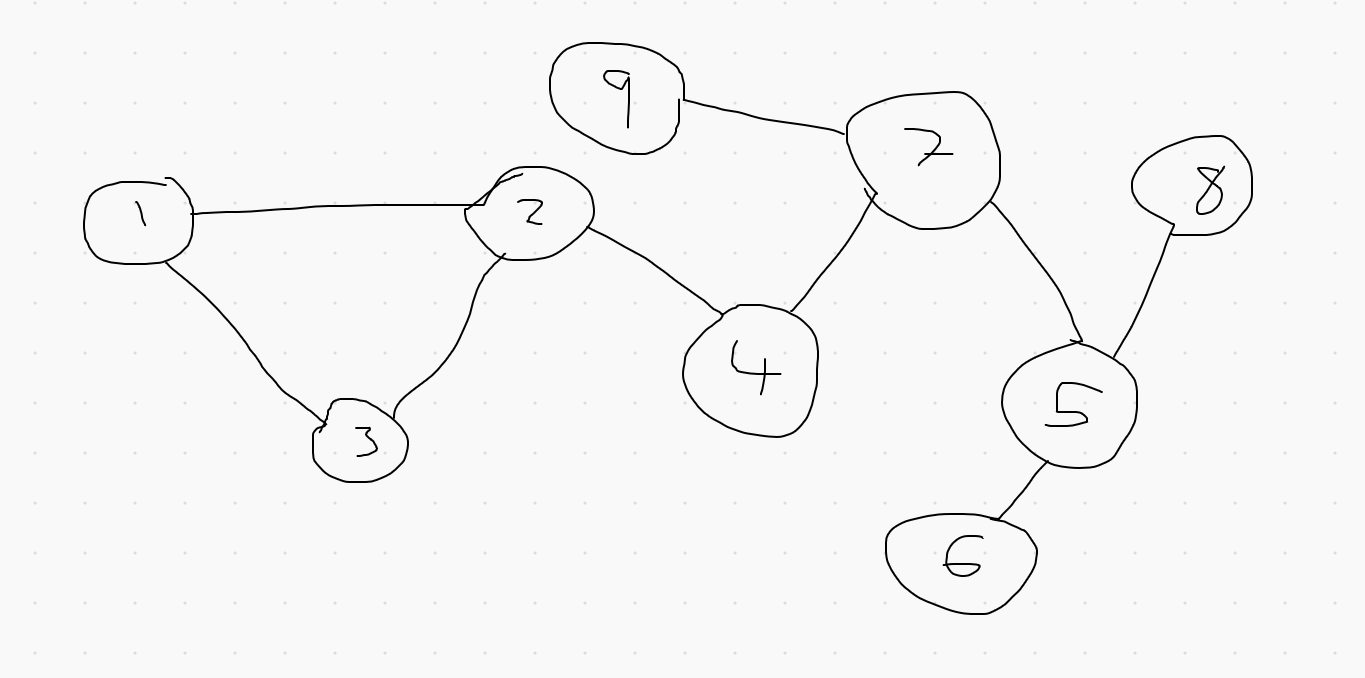
## Stacks and Queues

Now this is time for you guys to do things after we briefly go over stacks and queues we want you guys to give it a go.

Lets start off with the easier of the two:

### Stacks

If you don’t know already a Stack is an array of value for which you can only do two things with: pop and push. Pop means to remove the index that is at the end of the array and return it, push means add the given value onto the end of the array. This is a first in last out system. These are useful for things such as backtracking as when you go down a path you push the locations of where you have been onto the stack then on your way back just pop each location of one at a time for example:



Lets say you wanted to get from node 6 to node 3. You would walk push 6 on the stack then 5, 7, 4 then finally 2. Now you are at 3 the stack would look like:

[ 5, 7, 4, 2 ]

Now for the way back you would pop each pop each one off one by one.

The implementation of this is very simple, this is just a class with an array and a variable keeping track of the next free index, when pop is called take 1 from the index and return the value there and when push is called set the index to the given variable then add 1 to the index.

### Queues

Queues are similar to stacks, they both are data structures that have 2 methods, pop and push (there could be more but these are the main 2). But in this case pop now removes the value at the front, so the first value in is the first value out.

This complicates things as now we have data being moved around at either end of the array. There are a couple different ways to implement this but we want you guys to have a go at something instead of just listening to us for all this time, so try implement this yourself.

### Priority Queues

Now things are about to get a bit more spicy. Priority Queues are Queues that have order to them. This means that every index in the array is ordered by an attribute generally in order from lowest to highest. For example:



This would print:



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

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.

There are more efficient implementations than this but they involve trees so we will come back to this at a later time.