Elective Module 2: Time complexity.

Isak Oswald
s225375329

April 7, 2025

Requirements

0.1 Request for feedback

Hi Peter, I took a different approach in this module to improve my summary. Instead of writing the summary during completion of each activity, I studied the entire module and then wrote my summary. I feel as this maybe allowed me to cover key points easier and make the summary more concise? Also, I learnt how to implement references and used the auto references which i hope you like. Thanks!

0.2 Response to feedback

N/A

1 Introduction

Big O notation is a mathematical way of representing the efficiency of algorithms by measuring their time and space complexity. This mathematical representation gives us a way to easily group these algorithms into different categories in relation to their performance as the input size grows. We use Big O notation to represent the algorithms **worst**-case scenario which allows us to evaluate its performance in extreme conditions which could include of a very large input size. [1]

2 Big O Notation

As we now know what Big O notation is, it's important to be able to understand what someone knows what someone means when they say something like "$O(n)$". Have a look the program below, and see if you can identify its time complexity.

```
#include <iostream>

int main()
{
        int array[5];

        for(int i = 0; i < 5; i++)
        {
                array[i] = i*2;
        }
        return 0;
}
```

In this simple C++ program we are creating an array of 5 integers called *array*. After this is created, we use a for loop which loops through each index of the array and adds an element. First, we have to ask ourselves ”how many operations does it do to complete” and ”what happens if we double the size of the input (**int array**[**10**]**;**)? First off, since the array is of size 5, we need to iterate through 5 elements to complete the algorithm. Secondly if we double this size to 10 elements, we will need to iterate over twice the amount of elements. Considering this, we know the algorithm is $O(n)$ as the size of the elements has a direct relationship to the amount of operations the algorithm has to complete.

2.1 Common notation

There are some key common Big O notations that you should remember as you will commonly hear others talking about these in conversation. I have complied a list of common Big O notations below. The table is in the format **BigO** — **Description** — **Example**

1. $O(1)$ — Constant (like instant) time — Accessing an array element.
2. $O(n)$ — Linear time — Traversing an array.
3. $O(n^2)$ — Quadratic time — Bubble sort or Selection sort.
4. $O(\log n)$ — Logarithmic time — Binary search.
5. $O(n \log n)$ — Log-Linear time — Merge sort or Quick sort.
6. $O(n^n)$ — Exponential time (every bad) — Computations that involve very large input sizes.

[2] Now that you know this, have a look at the code example to see why this is O(n).

3 Comparing runtime

Just measuring the time for an input may be misleading in relations to its real performance, just because something is an $O(n^2)$ operation, it does not mean that it is *always* faster then something for $O(n)$. Have a look below to see what I am talking about with a further explanation.

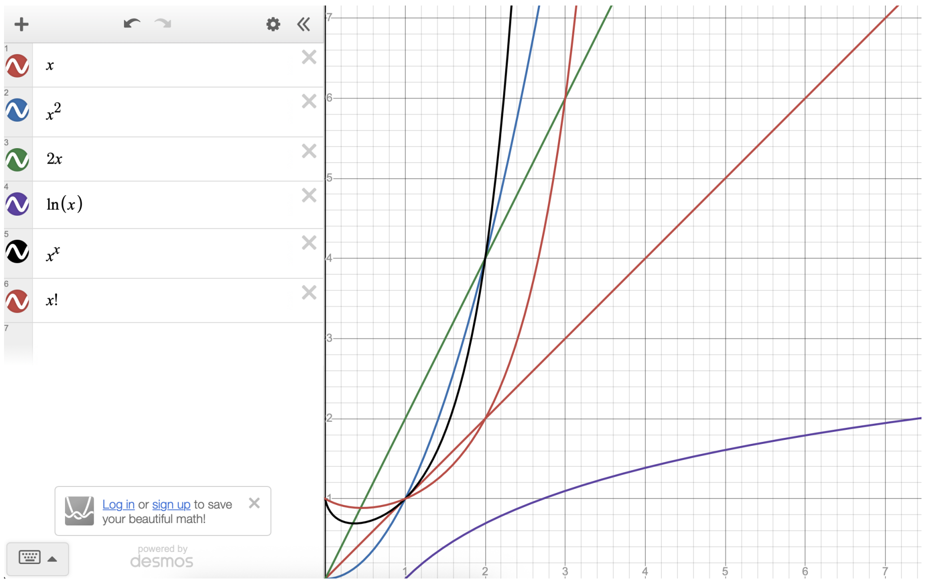


Figure 1: Desmos time complexity comparison

[3] If we ignore all of the lines except for the red x and blue x^2 (which represents $O(n)$ and $O(n^2)$ we can see something quite interesting happening, for clarity, the X axis is the sample size and the Y is the time. As you can maybe see, there is a specific input size where $O(n^2)$ is actually faster than $O(n)$. This only one example of an interesting characteristics of these linear growths, feel free to analyse the screenshot for more information. If you take away any one thing from this section, it should be that the algorithms time complexity its not the only factor which represents performance, you need to also take in-to account other factors such as sample size.

4 Formal definition and asymptotic

Now that we have learned that function complexity can be based upon how they behave (their exponents, multipliers, etc), you hopefully realised that when the input (the x axis) of the function is massive, the values like +5 and $2x^2$ became almost insignificant. Because of this extra complexity, it can become difficult to understand the relationship between how the function is growing as there are multiple variables contributing to the growth. Lucky, for us, using the formal definition 1 is a great way to simplify the function to see the overall way that it grows. Have a look below.

$$\exists C, k \in \mathbb{R} \text{ such that } |f(x)| \leq C|g(x)| \text{ for all } x \geq k. \tag{1}$$

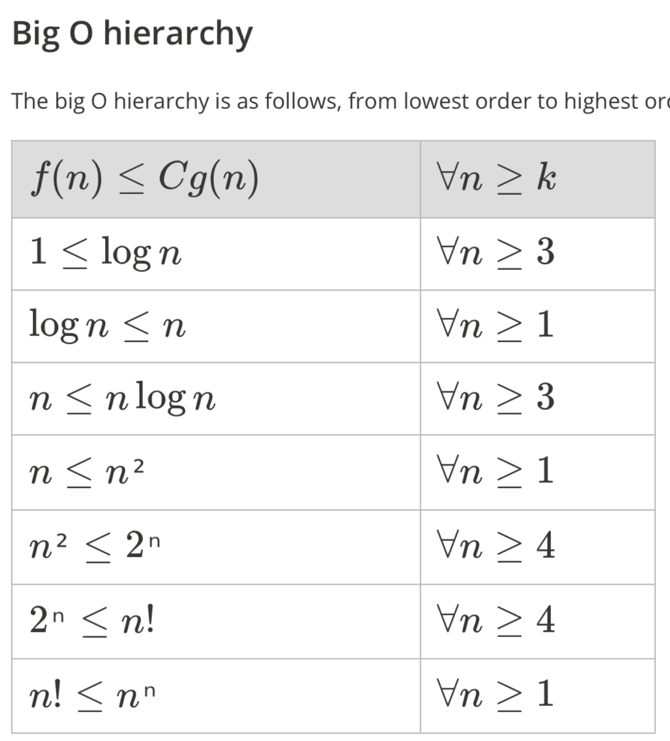
This absolutely confused me to bits in the beginning, however I recommend playing with Desmos for a bit to understand why $Cg(x)$ is very close to being that same at very large input sizes. Briefly, the reason for this is because when we look at a function, when we substitute in increasing input sizes the term with the highest multiplier (the dominating term) will contribute to most of the growth at a large size. One question I had during this task was "why do we want to do this?" which is a valid question however after you understand you will this that this allows us to simply determine the growth with the new simple function. To do this, we need to find constants k and c which are witnesses to the function growing. Have a look in the next section for an example.

4.1 A quick example.

Since we now know why and how to do this, let's do it so you can see the process. Just say we have the question "$3x^3 + 2x^2 + 4x - 5 = O(x^3)$", initially, we know that representing this as $C(x^3)$ is good because the x^3 is the dominate term which will cause the most amount of grown, especially at large values. If we set our k constant to one we have the function $f(1) = 4$, therefor for any constant C it must be greater than or equal to 4 for all values after 4. Therefor if we set our C to 4 we know that $4(x^3)$ for all values of x when $k = 1$ will be greater than or equal to our original function. Therefore, we have found the witness pairs $k = 1$ and $C = 4$. As you can now see, $4(x^3)$ is a much easier to see representation of growth compared to our original function.

4.2 Other functions

Now that we know how to calculate some witness pairs for some basic functions, what about some other functions which include logarithms, powers of x and factorials?. This may seem difficult initially however its actually quite similar to what we have already been doing, however we just need to make sure to follow the hierarchy list so we can find the element which is contributing most to the growth of a function as x increase towards infinity, then find our witness pairs. Have a look at the image below for the hierarchy table below.



Big O hierarchy

The big O hierarchy is as follows, from lowest order to highest order:

$f(n) \leq Cg(n)$	$\forall n \geq k$
$1 \leq \log n$	$\forall n \geq 3$
$\log n \leq n$	$\forall n \geq 1$
$n \leq n \log n$	$\forall n \geq 3$
$n \leq n^2$	$\forall n \geq 1$
$n^2 \leq 2^n$	$\forall n \geq 4$
$2^n \leq n!$	$\forall n \geq 4$
$n! \leq n^n$	$\forall n \geq 1$

Figure 2: Desmos time complexity comparison

[4] Now having this table handy, let's use the table to solve a problem.

4.3 Solving a problem using a table

Question $f(x) = x^2 + 2log(x) + 4$ First, we extract the parts of the table that a reverent to our problem, we use this table to compare growth rates. As you can see, $logx \leq x$ for $x \geq 1$ so now lets try to find our k, it is always a good idea to try $k = 1$ first so lets do that. We end up having an equality thant looks something like this: $x^2 + 2xlogx + 4 \leq x^2 + 2x^2 + 4$. Now after we collect the inequalities on the left side we have $7x^2$. As you may of seen, I didn't use the absolute value as all the numbers were positive. Therefore, we end up with $C = 7$ and $k = 1$ as a solution.

5 An example of an algorithm

It is usually pretty simple to determine an algorithms time complexity, let's consider the following sorting algorithm.

- Input: A list of numbers
- Output: The list of numbers sorted in increasing order
- Step 1: Find the smallest number and swap it with the number in the first position.
- Step 2: Find the ith smallest number. Swap it with the number in the ith position.
- Repeat step 2 until i = n.

As you can see, this algorithm involves iterating over and array of numbers n times (to get the elements) and the $n-1$ times to check the element to other elements in the list. The reason that this second for loop is defined as $n-1$ is beaches it doesn't need to check to see if it is less than itself. Now, since we are executing the first for loop n times, and the second for loop $n-1$ times, we are using two operations which means its time complexity is $O(n^2)$. Even as this algorithm is a little more efficient than $O(n^2)$ as it iterates over the inner loop $n-1$ times, because its dominated by quadratic growth, its defined as $O(n^2)$ One thing to note here is that we pick our solution based on the chat, we know that $2n^2$ as its has a faster growth.

6 Reflection

6.1 What is the most important thing I learnt in this module?

I think that the most important thing I learned within this module was how to use Big O notations to analyse the time complexity and therefore efficiency of an algorithm. Due to my learning through the module unit site and Desmos, I was able to understand how different algorithms grew in terms of time based on the input size and how some algorithms are better than others. I also think that learning about the formal definition was important as it allowed me to prove that something was of a certain time complexity by finding constants c and k. Doing this allowed me to effectively compare different algorithms simply.

6.2 How do this relate to what I already know?

This program related heavily to what I have been learning in SIT102 (Programming) as I'm up to only three tasks left have completed a fair chunk of the module and dabbled in dome basic algorithms and data structures like linked lists. For my HD project in Programming, I am planning on creating a breadth first search algorithm visualiser and what I have learnt here will hopefully be a feature that I add to the visualisations. Before learning this, I knew that some programs were faster then others, however I was unable to explain why as in the early points in my programming journey I thought that if a code was short it was efficient. However, now that I have learned about time complexity, I will be able to create more efficient programs in relation to time complexity.

6.3 Why do I think the course team wants me to learn this content for my degree?

I think that the course team wanted me to learn this content as it is crucial for creating effective solutions to real world problems. I think that my programs that I develop in the future will be better in the future as I am able to provide not only a solution, but an effective one. I think that the course team wants me to learn this content as it provides a foundation for working with larger data sets or performance-critical systems potentially in an employed environment.

7 FOLLOW UP QUESTION

$$7log(x) + 6 + 2^x + 3 + x^x \tag{2}$$

First we were asked "What is the order", to solve this we have to find what contributes to the most growth. Since x^x grows faster than our constants 3 and 6, as well as our 7 $\log(x)$ and 2^x the order of the equation is $O(x^x)$ since it contributes to the most growth, especially at the high inputs (x's). Secondly, we need to find the witness pairs which will be in the form $7\log(x)+6+2^x+3+x^x \leq c \times x^x$ Now we if we calculate $f(1)$ (for our k value) we have a result of 12. This means that at any $k \geq 1$ we must have our constant c which is 12, this looks like $12 \times x^x$. This means that our witness pair is $k = 1, c = 12$

Other

References

[1] freeCodeCamp. (202, Jan 16). *What is Big O Notation Explained: Space and Time complexity.* freeCodeCamp.org

[2] Olawanle, J. (2024, Novemeber 7). *Big o cheat sheet - Time Complexity Chart.* freeCodeCamp.org

[3] Desmos — Graphing Calculator. (n.d). Desmos. Desmos.com

[4] Deakin University (n.d). *Time Complexity Hierarchy*