COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 1: Big-O Notation

Lecturer: Dr. Andrew Hines *Scribes: Kerrie Lowe, Niamh Crowley*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

1 What is the Big-O and why is it important?

Big-O Notation is an analysis tool for choosing the most efficient algorithm to solve a problem. An efficient algorithm ensures that a programme will spend as little time possible executing in the CPU. **Big O notation evaluates the worst-case time complexity of an algorithm**. We use time complexity here instead of running time because we are focusing on the behaviour of the algorithm as the **data or input increases towards infinity**. Through the use of an upper bound we ensure that we are finding the worst-case running time scenario but finding the closest upper bound means our deductions wont be too far off the real run time. The importance of Big-O notation evaluation can be seen when trying to traverse large data sets or when using real time software. An example of this would be the response time of an aeroplane when given a complex instruction.

2 What is the running time of a programme?

The running time of an algorithm is the length of time taken to run a program using that algorithm on a computer. To calculate the running time, programs can be written to test an algorithm and experiment with different inputs. **The running time of a program is proportional to the size of the input on which it is run**. Thus, different input sizes can cause fluctuations in time. The worst case running time on any input given is recorded rather than the average as this would indicate that all inputs of size n are equally likely to achieve that run time. In contrast to Big O Notation, **the running time is dependent on the computers software and hardware.** This creates a major problem as predicted running times should be consistent across different machines and software. As mentioned above, the Big O focuses on time complexity rather than running time of an algorithm.

3 So how do we find the Big-O of a programme?

3.1 Theoretical Analysis

It is important to provide an in-depth description of any given algorithm. Pseudocode is one mechanism to do this as it is more structured than the English language but less detailed than a program. Theoretical analysis takes into account all possible inputs. It determines the amount of time, storage and other resources required to execute algorithms. For theoretical analysis, the number of primitive operations in the algorithm are calculated and identified in pseudocode. This allows us to determine the maximum number of operations executed by an algorithm, as a function of the input size.

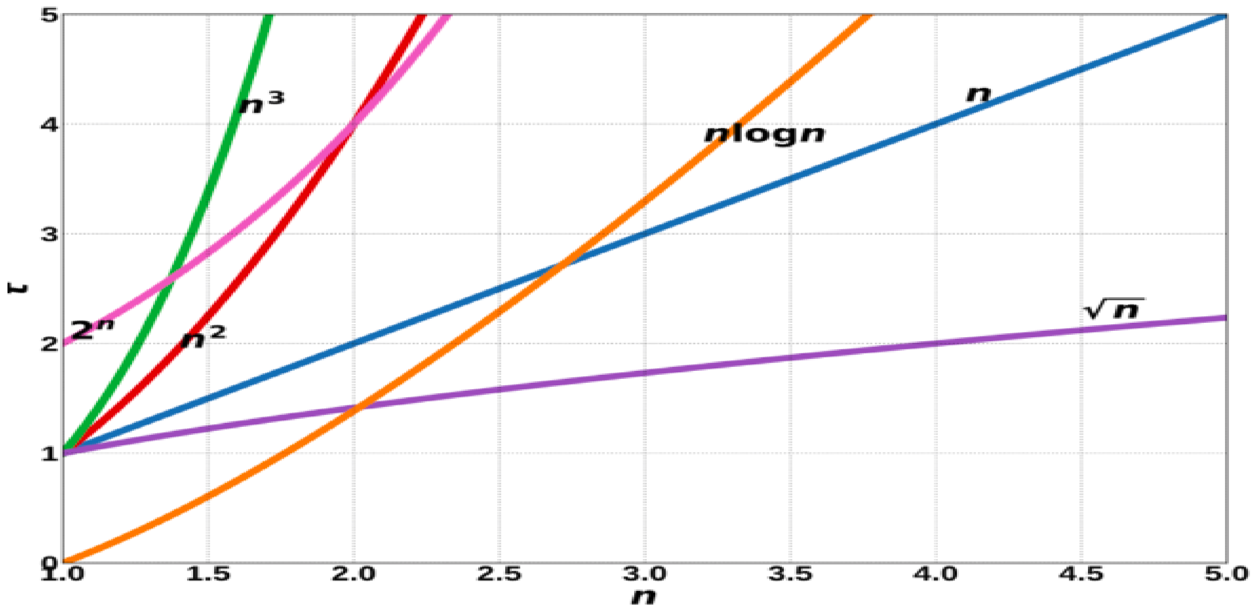
Algorithm 1: Return the index of the max value of an array

```
1 function argmax (array);
  Input  : an array, of size n
  Output: the indexof the maximum value
2 index ← 0    #1 op:assignment
3 foreach i in [1, n-1] do    #2 op per loop: index and assignment
4 if array[i] ¿ array[index] then    #4 ops per loop: index x2, comparison, check if boolean true/false
5 index ← i    #1 op per loop: assignment, sometimes
6 endif
7 endfor
8 return index    #1 op: return
```

The algorithm has runtime: f(n)=7(n)+2

3.2 Asymtotic Analysis

While running time analysis concerns the length of time it takes a function to complete, **asymptotic analysis refers to the growth rate of the algorithms runtime as the input size n goes to infinity.** Functions are simplified to extract the most important part and remove the less important parts. For example; an algorithm that has 10n2̂ + 2n + 2 operations, extracts 10n2̂ and removes the remaining operations as it is larger than the other terms. Thus, we can say the running time of this algorithm grows as n2̂ after removing the coefficient of 10. By using asymptotic notation **(removing the less significant terms and the constant coefficients)**, we gain the ability to focus on the growth rate of the algorithms running time. Alongside this, asymptotic analysis enables us to estimate the growth rate of an algorithms running time and compare to others (Diagram). The slower the asymptotic growth rate, the better the algorithm. The reason for this is that it is easier to estimate the running time of an algorithm that has a slower growth rate. If the growth rate of an algorithms running time is n, then the running time is at *most* *k**f(n) for some constant k. Thus, we can say that an exponential algorithm is worse than a constant algorithm as it has a faster asymptotic growth rate.



4 Classifying and Analysing Functions

Below are the **most common function classes**, which are evaluated by their efficiency. Alongside this, we have provided some examples of how they look like in terms of computer code.

The Flash

The first and most efficient class is O(1) or O(c). This is the ideal case as the algorithm will complete in the same amount of time regardless of input(n). Examples of this in code would be returning the head or first element of a list or popping a stack.

The next class we look for is O(log(n)). This function never has to look at all the input. Often this algorithm checks half the input of a sorted array and can determine from this whether the value it is looking for is in the first half or second. An example of this in code would be a binary search. It is a very efficient algorithm when dealing with large data sets.

Next is the **linear algorithm O(n)**. The time complexity of this increases directly proportional to the input(n) given. It must check all the inputs once. Examples of this in code would be finding the min/max of a sequence.

Clark Kent

From here the time complexity of algorithms begin to worsen. **O(n log(n))** is often the time complexity when sorting elements which are first halved and then each compared. MergeSort is an example of this. An (clear) explanation can be found at the following links: GeeksforGeeks. Python.

Wonder Woman

An algorithm with O(n^2 or $n^{power>2}$) often is found in code where there are **nested loops**(with the power being reflective of the number of loops). A simple example of this would be code to produce a multiplication table.

Algorithms with O(2^n) are often **recursive** algorithms that solve a problem of size n by using n-1. An example of this in code would be calculating the Fibonacci series.

An algorithm with complexity 0(n!) iterates over all the possible combinations of an input similar to the truth tables we produced in Computer Arch + Org. **These algorithms are best avoided for large data sets.**

Black Panther

5 Bibliography

www.codeburst.io/the-ultimate-beginners-guide-to-analysis-of-algorithm-b8d32aa909c5

www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation

www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation

Bell, R., 2008-2018. Rob-Bell.net. www.rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

Kautz, S., 2010. Iowa State University: Computer Science. web.cs.iastate.edu/ smkautz/cs228s11/examples/algorithms/notes_on

Lobo, C., 2017. topcoder. www.topcoder.com/blog/learning-understanding-big-o-notation/

Murphy, L., 2018. What is the Big-O and why is it important?. Dublin: s.n

Rail, P., 2018. FreeCodeCamp. medium.freecodecamp.org/all-you-need-to-know-about-big-o-notation-to-

crack-your-next-coding-interview-9d575e7eec4