AQA Computer Science A-Level

4.4.4 Classification of algorithms

Advanced Notes

Specification:

4.4.4.1 Comparing algorithms

Understand that algorithms can be compared by expressing their complexity as a function relative to the size of the problem. Understand that the size of the problem is the key issue. Understand that some algorithms are more efficient:

- time-wise than other algorithms
- space-wise than other algorithms.

Efficiently implementing automated abstractions means designing data models and algorithms to run quickly while taking up the minimal amount of resources such as memory.

4.4.4.2 Maths for understanding Big-0 notation

Be familiar with the mathematical concept of a function as a mapping from one set of values, the domain, to another set of values, drawn from the co-domain, for example $\mathbb{N} \rightarrow \mathbb{N}$. Be familiar with the concept of:

- a linear function, for example y = 2x
- a polynomial function, for example y = 2x2
- an exponential function, for example y = 2x
- a logarithmic function, for example y = log10 x.

Be familiar with the notion of permutation of a set of objects or values, for example, the letters of a word and that the number of permutations of n distinct objects is n factorial (n!). n! is the product of all positive integers less than or equal to n

4.4.4.3 Order of complexity

Be familiar with Big-O notation to express time complexity and be able to apply it to cases where the running time requirements of the algorithm grow in:

- constant time
- logarithmic time
- linear time
- polynomial time
- exponential time.

Be able to derive the time complexity of an algorithm

4.4.4.4 Limits of computation

Be aware that algorithmic complexity and hardware impose limits on what can be computed.

4.4.4.5 Classification of algorithmic problems

Know that algorithms may be classified as being either:

- tractable – problems that have a polynomial (or less) time solution are called tractable problems.
- intractable – problems that have no polynomial (or less) time solution are called intractable problems.

Heuristic methods are often used when tackling intractable problems.

4.4.4.6 Computable and non-computable problems

Be aware that some problems cannot be solved algorithmically.

4.4.4.7 Halting problem

Describe the Halting problem (but not prove it), that is the unsolvable problem of determining whether any program will eventually stop if given particular input. Understand the significance of the Halting problem for computation. The Halting problem demonstrates that there are some problems that cannot be solved by a computer.

Comparing Algorithms

> **Note**
>
> **Efficiently implementing automated abstractions means designing data models and algorithms to run quickly while taking up the minimal amount of resources such as memory**

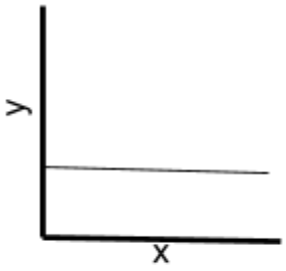
The complexity of each algorithm can be shown through a function relative to the size of the task. An algorithm can be complex in two ways - in terms of space and in terms of time. An ideal algorithm will run quickly and take up little space as possible. Often, a programmer will have to create a compromise solution, relative to the situation.

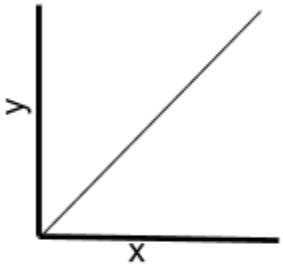
Maths for understanding Big-0 notation

Here are some graphs you will be expected to recognise.

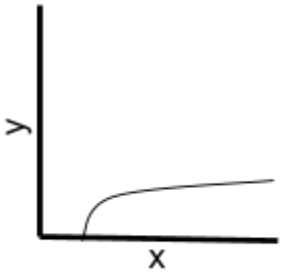
> **Synoptic Link**
>
> You should be familiar with **functions** as a **mapping** from **one set of values**, the **domain**, to **another set of values**, drawn from the **co-domain**.
>
> Domain and Co-domain are covered in **Functional Programming Paradigm** under **Fundamentals of Functional Programming Structures**.

Constant
E.g. y = 3

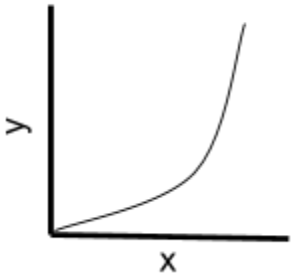




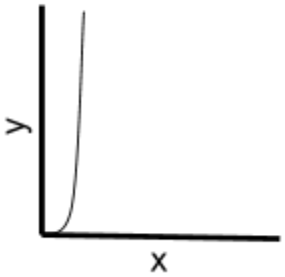
Linear
E.g. y = 3x



Logarithmic
E.g. y = log(x)



Polynomial
E.g. $y = x^2$



Exponential
E.g. $y = 3^x$

As you can see from above, the different types of graphs have different rates of increase. Apart from the constant function, they all grow as the input increases.These functions can be used to represent the complexities of different algorithms. There are two other functions you have to be aware of - y = xlog(x) and y = x!. ! stands for factorial. The factorial of a number is all the positive integer values either smaller or equal to that value multiplied, e.g. 4! = 1 x 2 x 3 x 4 = 24. Factorials are useful for working out permutations, e.g. how many ways are there to order the alphabet - there are 26 letters in the alphabet, so there are 26! different ways of ordering the alphabet (or approx 400,000,000,000,000,000,000,000,000)

Big O Notation

The complexity of an algorithm can be described by big O notation. Big O always assumes a worst case scenario. In big O is is customary to describe the input in terms of n rather than x. If we wanted to specify the complexity of an algorithm with a linear time complexity, we would say the time complexity is O(n). If an algorithm has a complexity of $n^3 + 200n^2 + 1000n + 25$, its big O is simply $O(n^3)$ as n^3 is the largest polynomial - when n is very large, the other parts of the equation are dwarfed by n^3.

From top to bottom, the table is least to most complex.

Function	Big O	How to recognise	Example
Constant	O(C)	Time is independent of input	Determining if a number is odd or even
Logarithmic	$O(\log_2(n))$	Halves the number of items each iteration	Binary search
Linear	O(n)	In a worst case scenario must go through each item once.	Linear search
Linear Logarithmic	O(nlog(n))	N/A	Merge sort
Polynomial	$O(n^2)$	A loop inside a loop	Bubble Sort
Polynomial	$O(n^3)$	A loop inside a loop inside a loop	Checking 3D coordinates
Exponential	$O(2^n)$	Intractable - cannot be solved within a useful amount of time	Recursively calculating Fibonacci numbers
Factorial	O(n!)	Intractable - cannot be solved within a useful amount of time	Travelers problem, explorers problem

Synoptic Link

Linear and **Binary** searches are used for **locating** an **item** in a **list**. **Merge** sort and **Bubble** sort **order** lists.

Linear and Binary Searches are covered in **Searching Algorithms.** Merge and Bubble sorts are covered in **Sorting Algorithms.** Both are under **Fundamentals of Algorithms**.

Limits of computation

There exists two types of algorithms - tractable and intractable. A tractable problem can be solved within a useful period of time. Tractable problems have a polynomial or less time solution. Intractable problems are theoretically solvable (i.e. there exists a corresponding algorithm) but are classified as ‘insoluble’ due to limits of computations - due to the speed of today’s technology, it may take millions of years to solve. Intractable problems do not have a polynomial (or less) time solution. They

cannot be solved within a useful period of time. For these types of problems, we may use a heuristic method. These are approximate solutions to a problem; they are not optimal, but are more useful than their intractable equivalent.

However, not every problem can be solved algorithmically, e.g. the Halting problem. The halting problem: it is impossible to write an algorithm to determine if another algorithm will finish with a given input. The halting problem demonstrates that there are some problems which cannot be solved by computers.