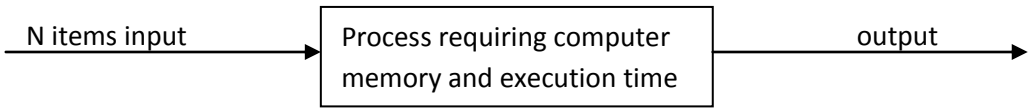
Big O Notation in Computing

In computer science it is important to have a measure of the computer memory requirements and the processing time of computer codes. However, the means of storage may be altered and different computers run at different speeds and so O notation provides a means of abstracting the performance of a computer programming technique from its environment.

Big O notation is also used in mathematics in a similar way, where it can model the asymptotic (or limiting) behaviour of a function either at a point (often zero) or at infinity[1]. In computing big O notation is normally used to give a concise representation of the space (memory) or processing time requirements of an algorithm or computer code, as the amount of data that is handled gets large or (theoretically) approaches infinity. Computer codes can be analysed so that often a functional representation of its space and time requirements can be determined[2].

Computers are a means or storing and manipulating data. Data is stored in data structures[3] and data is manipulated by algorithms[4]. Data structures require computer memory for their storage (eg RAM (random access memory) or hard drive) and algorithms take time to execute. For many computing tasks we do not need to concern ourselves with the storage requirements or the execution time since much everyday personal computing can be easily stored and the execution times are not noticeable. However, there are many occasions in computing where we need to be aware of the storage requirements – since we may exceed the (finite) memory capacity of the computer – and aware of the computing time – since the user may need to know how long to wait for the solution to a problem. In this document we analyse space and time requirements of algorithms from samples of code. From this a computer scientist can select or develop algorithms and data structures with a view to solving a problem within the constraints of the computer system, or select a computer system that can sustain the developed code.

If we think of a computer program that handles *n* items, that is most representative of the storage or execution time requirements of the software. For example a banking program may have *n* accounts, a university database may have *n* students. The following diagram gives a simple illustration of a computational process.



Modern computers have RAM measured in megabytes (MB) or gigabytes (GB), hard drives measured in gigabytes (GB) or terabytes (TB) and the central processing unit's speed is measured in megahertz (MHz) or gigahertz (GHz), meaning that they can compute millions or billions of basic instructions per second. For small values of *n* the storage requirements and the computer processing time is likely to be insignificant on modern computers. However, this may not be the case for large values of *n*. For example a typical bank could have around ten million accounts: how would a computer system cope with such demands on it? Every int, float or "object" etc declared in a program requires memory for storage (RAM)

[1] Big O Notation in Mathematics
[2] Space and Time
[3] Data Structures
[4] Algorithms

Where does the demand for 'space' or computer memory arise and why does the computer require 'time' to complete an operation? When a computer program is run the data structures (eg ints, floats, arrays) require memory for storage (usually RAM). Every basic computer operation such as addition, multiplication and comparison takes a fragment of time to complete. Computer memory or space is allocated when a computer program declares a data structure and time is required when an algorithm is implemented.

Those responsible or either software design or computer specification will normally show an interest in the storage and execution time requirements of computer programs. However, they will not be particularly interested in an exact specifications. For example if we knew that a process required 3.261785 GB and it would require 5.362945 seconds to run then it is of little interest to include the information after the decimal points. Rather it is sufficient to obtain an estimate or simple formula which approximates the storage requirements and execution time.

The following table shows the memory required to store data types. The examples given are from c or Java, but they are very similar across computer programming languages.

Data type	**Storage (bytes)**
boolean	1 (1 bit usually stored in a byte)
char	1 or 2
int	4
float	4
double	8

A byte is made up of 8 bits and a bit is a variable that can only take one of two values '0' or '1'. A kilobyte is 1024 (ie 2^{10}) bytes, a magabyte is 1024 kilobytes, a megabyte is 1024 kilobytes and a terabyte is 1024 gigabytes.

The operations of a computer take place in the central processing unit (CPU) or 'chip'. All computer operations, such as multiplying integers or adding floating point numbers are broken down into simpler operations that can be directly executed by the CPU. These operations are measured in 'cycles' such as the 'fetch and execute cycle'. The speed of the CPU is measured by the number of cycles that it can execute in each second or hertz (Hz). A kilohertz (KHz) is 1000Hz, a megahertz (MHz) is 1000KHz and a gigahertz (GHz) is 1000 MHz. Computers can include a number of CPUs, so that the computer is able to carry out a number of computations simultaneously, sometimes referred to as parallel processing.

The measurement of the space or time requirements can vary from being very simple to be rather complicated. To get things started, we will begin with a simple example program for finding the average (arithmetic mean)[5] of a set of data written in pseudo-code.

[5] Measures of Average, Centre or Central Tendency

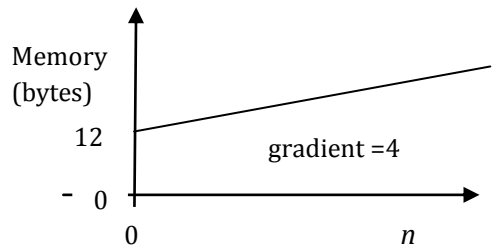
Pseudocode for computing the average of 'n' pieces of data

```
data [n]
total=0.0
for i=1..n
 {
  total=total+ data[i]
 }
average=total/n
```

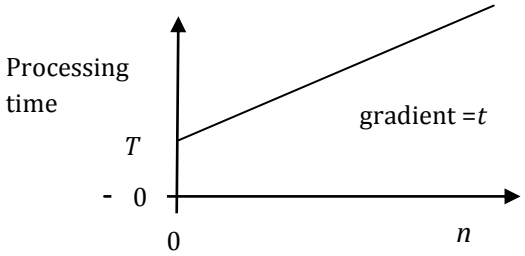
Let us first look at the data storage requirements. There is an array of data of length n and variables called total, i, and average. Let us assume that data is a float array and hence its size is 4n bytes[6]. If the variables total and average are also of the float data type and i is an int then this requires another 12 bytes. Hence the total memory requirements for the data structures in this code can be estimated at $4n+12$ bytes.

Let us now consider the execution time for this code. Time will be required for the general housekeeping, setting up the data structures and initialisation. Looking outside of the for loop, there are assignments and a division. Let the work outside of the for loop take time T and this will be largely independent of n. If we now consider the loop then clearly the piece of code within the loop needs to be executed n times. Let t be the time taken for each iteration of the loop and hence the completion of the execution of the loop will takes a time of nt. Hence the total time for the execution of this piece of code is $nt+T$.

The following sketches of graphs[7] of the memory requirements and the processing time of the code above are given below.



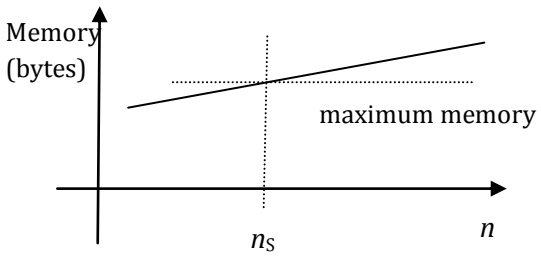
Space



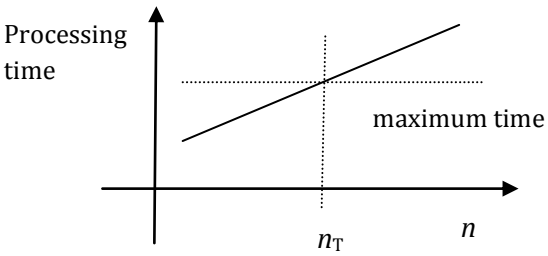
Time

From a computing point of view, these graphs can help estimate memory requirements and computer time for any value of n. In the case of the code for finding the average of a set of n pieces of data the graph of memory requirements and computer processing time are both straight line graphs. Both graphs show a 'growth' with n, which indicates that for 'large' values of n that some threshold may be reached. For example we may not have sufficient memory to store the data structures or the computer time may be too long to wait for a computer to process some data, as illustrated in the following extensions of the above graphs.

[6] Arrays
[7] Plotting a Graph



Space



Time

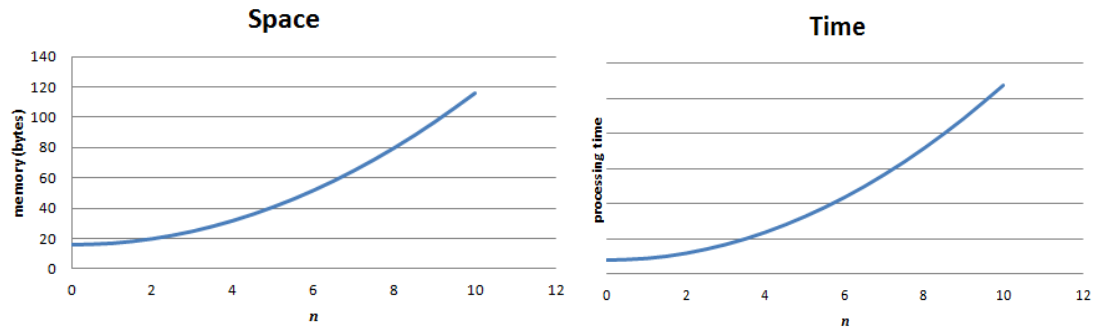
In the practical circumstances of running the code on a computer there are limits on n the number of items of data. As illustrated in the graphs above, $n \leq n_S$ in order to fit in the finite space and $n \leq n_T$ in order that the time taken for the process is acceptable. This is the case with all computer codes.

Computer scientists are therefore very interested in the nature of the graph of memory requirements and computer processing time with respect to the size of the data that is to be managed. In order to illustrate this further we will consider the following fragment of code for computing the average of a set of data stored in a square array.

Pseudocode for computing the average of 'n^2' pieces of data

```
data [n][n]
total=0.0
for i=1..n
 {
 for j=1..n
  {
   total=total+ data[i][j]
  }
)
average=total/n/n
```

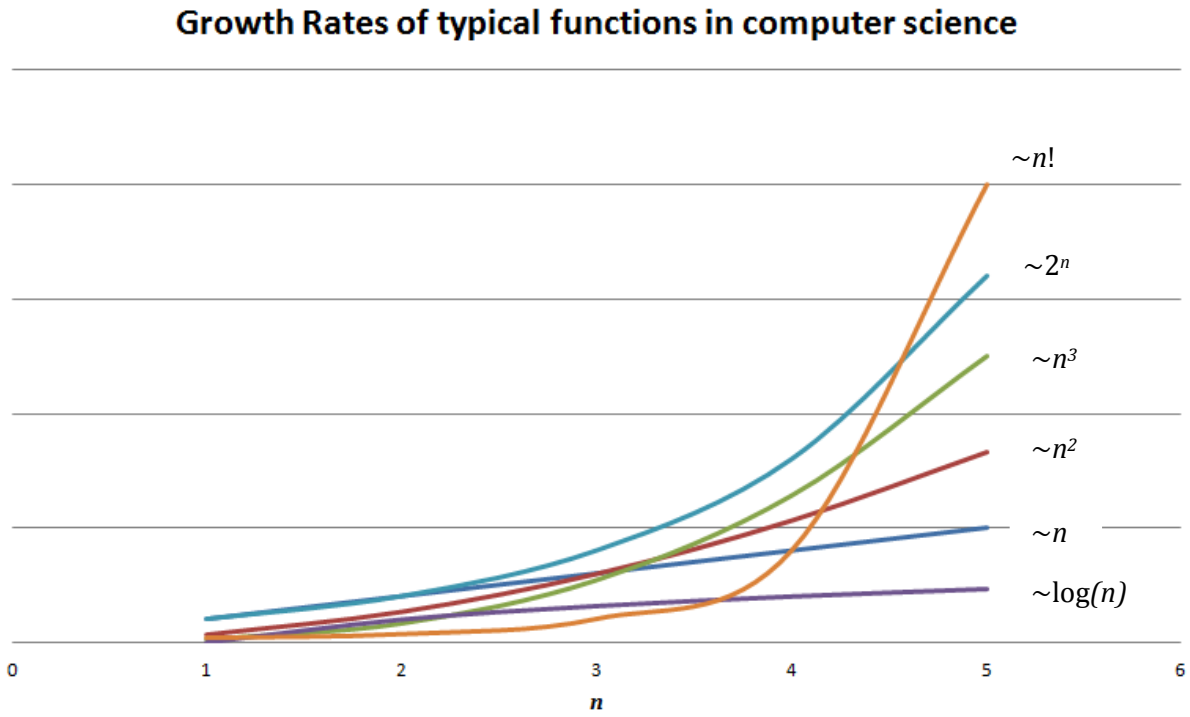
If we analyse the data requirements in this code then is an array of data with n^2 components and variables called total, i, j, and average. Using the same assumptions about types as in the previous code then the data structures in this code can be estimated at $4n^2+16$ bytes. If we analyse the data requirements then let us assume that the code within the innermost loop takes a time u and the code outside the loops and the set-up time is U, then the total time for the code can be estimated as n^2u+U. The following graphs illustrate how space and time requirements o the code vary with n.



The graphs show that in the case of this latest code that the growth in both space and time with respect to *n* is stronger than in the first code. In general the stronger the growth in either spae or time the more concerned that computer scientists will be about its viability.

In the two examples the graphs of space and time have similar shapes. However, this is not normally the case and space and time should be analysed independently.

In te following graphs, typical function that crop up in algorithm and data structure analysis are graphed. The graphs have been scaled for illustrative purposes. However, the most important area to focus on in these graphs is their growth, for it is the growth in the curve that will have the significant impact when *n* is large.



In the graphs we consider the factorial function (*n!*)[8], logarithmic and exponential functions[9] as well as powers of *n*. For these functions, in terms of 'growth' we note that the factorial function is the most powerful, followed by the exponential and then the descending powers of *n*. The logarithm of *n* has the weakest growth.

[8] Factorial
[9] Logarithm and Exponential Functions