Classification of Algorithms

Comparing Algorithms

- The time efficiency of algorithms refers how long an algorithm takes to run as a function of the size of the input.
- More than one algorithm can be used to solve the same proble.
- For instance to calculate the sum of a sequence of numbers we can use the following algorithm:

$$sum = (n + 1) * n / 2$$

where n is the number we wish to sum the values up to. Using this calculation the time remains constant regardless the value of n. In other words, regardless of how many numbers we wish to add up the time taken will always be same.

We could use an alternative algorithm to calculate the sum of a sequence of numbers

```
sum ← 0
FOR i ← 1 to n
      sum ← sum + i
ENDFOR
OUTPUT sum
```

Using this algorithm the number of operations increases in linear time with the size of the input. Therefore, the time taken for the algorithm to run will grow in linear time as in size of the input increases. Clearly this is more inefficient than the first algorithm even though it solves the same problem.

Another area where algorithms differ in their efficiency is in regard to the memory requirements of algorithms. For instance programs that read in huge data files into memory can end up taking up a large space in memory.

When developing algorithms it is important to consider the hardware constraints of the system you are developing for (eg mobile phone which has limited processing and space capability). If you have large memory then your algorithm can afford to be less space efficient. Likewise if you have access to tremendous processing power algorithm (eg supercomputer) your may not need to be time efficient although it is still desirable to make algorithms as efficient as possible.

Maths for Big-O Notation

A function allows us to map a set of input values to a set output values

$$y = f(x)$$

where x is a value from the domain and y a value from the codomain

domain -> codomain

A **linear function** takes the form $y = mx + c$, where m is the gradient and c the intercept on the y axis.

A **polynomial function** takes the form $y = ax^2 + bx + c$

An exponential function takes the form $y = a^x$

A **logarithmic function** takes the form $y = a \log_n x$

Permutations illustrate how the number of operations grows factorially when we add additional dimensions to some problems.

How many different combinations can sequence of digits have?

No. of digits	No of combinations
2	2
3	6
4	24
5	120

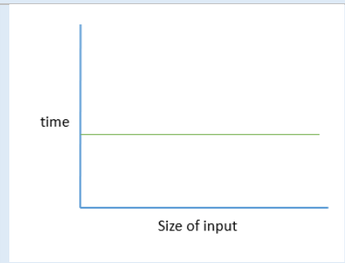
Big O notation gives us an idea of how long a program will run if we increase the size of the input. We need to consider how many operations will need to be carried out for a given size of input. This gives us the time complexity of the algorithm.

Constant Time O(1)

The time remains constant even when the number of input increases. E.g. calculating the sum of a sequence of numbers.

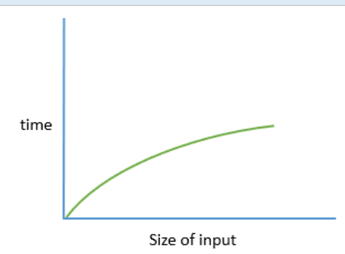
$$sum = (n + 1) * n / 2$$

Regardless of how many numbers we wish to add up the time taken will always be same.



Logarithmic Time O(log n)

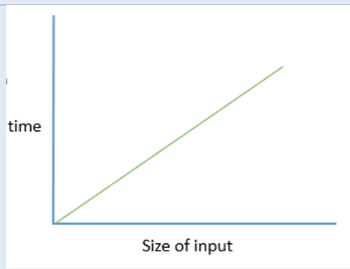
The time taken for the algorithm to run will grow slowly as in size of the input increases.



Linear Time O(n)

The time taken for the algorithm to run will grow in linear time as in size of the input increases.
Eg inefficient algorithm to calculate the sum of a sequence of numbers

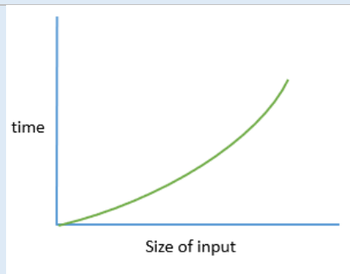
```
sum = 0
for i=0 to n
      sum = sum + i
output(sum)
```



Polynomial Time O(n^2)

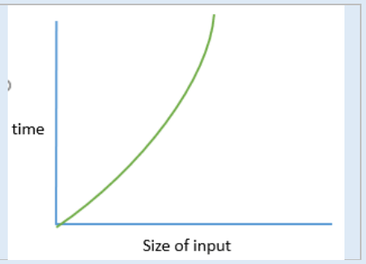
The time taken for the algorithm to run will grow proportionally to the square of the size of the data set. Normally when you have nested for loops this will have a polynomial time complexity.

```
for i=0 to n
 for j=0 to n
  Do something
```



Exponential Time O(2^n)

The time taken for the algorithm will grow as the power of the number of inputs, so the time taken for the algorithm to run will grow very quickly as more input data are added.



The time taken for an algorithm to run will depend on the hardware (eg processor clock speed, RAM size), even though the number of operations will be constant for a fixed input.

Tractable problems are problems that have a polynomial or less time solution eg O(1), O(n), O(log n), O(n^2)

Intractable problem are problems that can be theoretically solved but take longer than polynomial time e.g. O(n!), O(2^n)

Heuristic algorithms are used to provide approximate but not exact solutions to intractable problems

The travelling Salesman Problem
The idea is to find the shortest route to visit all cities. This is a permutation of the number of cities so has a factorial time complexity so quickly becomes an intractable problem with an unfeasibly huge number of permutations.

To solve this we use an heuristic algorithm. This provide an acceptable solution to the problem but it may not be the optimal or best solution. So for the travelling salesman problem we may find a short route but not necessarily the shortest route. Heuristic algorithms for the travelling salesman problem include the following:

- Greedy algorithm – take the shortest route to the next city
- Visit the cities in a circle
- Brute force algorithm – Apply to small but different subsets of cities and combine together. Apply the brute force algorithm to fewer manageable problems rather than a single intractable problem

Time Complexity of common algorithms

Linear Search	O(n)
Binary Search	O(log n)
Binary Tree Search	O(log n)
Bubble Sort	O(n^2)
Merge sort	O(n log n)
Travelling Salesman Problem	O(n!)
Brute force password cracker where n is the length of the password	O(A^n)

Unsolvable problems Some problems cannot be solved by a computer. The **Halting problem** is one such problem and show that some problems cannot be solved algorithmically.

The **halting problem** states that there is no computer program that exists that can determine whether another computer program will halt or will continue to run forever given some specified input.

The halting problem show that some problems cannot be solved by a computer