ALGORITHM

An algorithm is a finite set of well-defined instructions for accomplishing some tasks which, given an initial state, will terminate in a corresponding recognizable end-state. By this definition, algorithms can be viewed as specific steps for tackling and solving specific tasks.

Algorithms can be implemented by computer programs or computer languages. An error in the design of an algorithm for solving a problem can lead to failures in the implementation of the program.

A computer program can be viewed as an elaborate algorithm. In mathematics, an algorithm usually means a small procedure that solves a recurrent problem.

History of Algorithm

The word "algorithm" comes from the name of the 9th century Persian mathematician: Abu Abdullah Muhammad bin Musa al-Khwarizmi. The word "algorism" originally referred only to the rules of performing arithmetic using Arabic numerals but evolved in algorithm by the 18th century. The word has now evolved to include all definite procedures for solving problems or performing tasks.

The first case of an algorithm written for a computer was by Ada Byron's note on the *Analytical Engine* written in 1842, for which she is considered by many to be the world's first programmer. However, since Charles Babbage never completed his Analytical Engine so the algorithm was never implemented on it.

The Concepts of Algorithm:

The fact that algorithms are "well-defined procedures" for solving a problem does not mean that only one format of an algorithm can solve a particular problem. Just as there are different approaches for solving a problem, different algorithms may complete the same task with different set of instructions in more or less time, space, or effort than others.

There are simple algorithms and so many other algorithms are much more complex even with steps that repeat (iterate) or require decisions

(such as in logic or comparison) until the task is completed. Correctly performing an algorithm will not solve a problem if the algorithm is flawed or not appropriate to the problem. For example: a hypothetical algorithm for cooking yam will fail if there are no yams present.

Classes of Algorithms

One way of classifying algorithms is by their design methodology or paradigm. There is a certain number of paradigms, each different from the other. Furthermore, each of these categories will include the different types of algorithms. Some commonly found paradigms include:

- Divide and conquer
- Dynamic programming
- The greedy method
- Linear programming
- Search and enumeration
- The probabilistic and heuristic paradigm

Formalised Algorithms

The advancements in the use of algorithms in computer has made it possible to formalise some aspects of the algorithms since they have become essential to the way computers process information. This is because a computer program is essentially an algorithm that tells the computer what specific steps to perform in order to carry out a specific task, such as calculating employees' pay checks. Thus, an algorithm can be considered to be any sequence of operations which can be performed by a *Turing Compte System*.

Because an algorithm is a precise list of precise steps, the order of computation will almost always be critical to the functioning of the algorithm. Instructions are usually assumed to be listed explicitly, and are described as starting 'from the top' and going 'down to the bottom', an idea that is described more formally by *Flow Of Control*.

Implementing Algorithms

Algorithms are not only implemented as computer programs, but often also by other means such as in electric circuits or in a mechanical device.

The analysis and study of algorithms in one discipline of computer science and is often practiced abstractly, i.e. without the use of any programming language. In this sense, it resembles other mathematical disciplines in that the analysis focuses on the underlying principles of the algorithm, and not on any particular implementation. One way to codify an algorithm is the writing of *pseudocodes* (false codes, which could be user defined).

Example: Imagine you have an unsorted list of random numbers. Our goal is to find the highest number in this list. Upon first thinking about the solution, you will realise that you must look at every number in the list. Upon further thinking, you will realise that you need to look at each number only once. Taking this into account, here is a simple algorithm to accomplish this:

- When you begin, the first number is assumed to be the largest number in the list you've seen so far.
- Look at the next number, and compare it with the largest number you've seen so far.
- If this next number is larger, then make that the new largest number you've seen so far.
- Repeat steps 2 and 3 until you have gone through the whole list.

And here is a more formal coding of the algorithm in a *pseudocode* that is similar to most programming languages.

```
given: a list "List"
largest = list [1]
counter = 2
while counter <= length (List):
if List [counter] > largest; then
largest = List [counter]
counter = counter +1
print largest
```

Notes on Notation

- = as used above indicates *assignment*. That is, the value on the right-hand side of the expression is assigned to the container (or variable) on the left-hand side of the expression.
- `List [counter]` as used above indicates the counterth element of the list. For example: if the value of counter is 5, then `List [counter]` refers to the 5th element of the list.
- <= as used above indicates 'less than or equal to'

Note also that the algorithm assumes that the list contains at least one number. It will fail when presented an empty list. Most algorithms have similar assumptions on their inputs, called *pre-condition*.

As it happens, most people who implement algorithms want to know how much of a particular resource (such as time or storage) a given algorithm requires.