Algorithms

An algorithm is a finite list of ordered instructions that solves a particular problem.

In other words, when you are using an algorithm, you are following instructions. Understanding algorithms requires just patience and accuracy. You have to do as you are "told" at each step of the process.

Algorithms occur particularly within the domain of computing, and we shall introduce them by looking at some common processes used in computing.

One example of an important computer process is that of data processing. Data processing often requires data to be searched or sorted into an order. In order to specify a sorting or searching algorithm we require a language to describe elements in a list.

For example, let n_r denote the r th element in a list. Then, in the following list

14	29	6	51	99	37	63	18

$n_1 = 14$
$n_2 = 29$
$n_3 = 6$
$n_4 = 51$
$n_5 = 99$
$n_6 = 37$
$n_7 = 63$
$n_8 = 18$

Recall that a set is a collection of objects. For example, the set

$$\{14, 29, 6, 51, 99, 37, 63, 18\}$$

is the collection of the numbers, 14, 29, 6, 51, 99, 37, 63, 18. But this is not an ordered collection. In other words, any arrangement of these numbers is regarded as making up the same set. Thus, for example

$$\{14, 29, 6, 51, 99, 37, 63, 18\} = \{29, 14, 6, 51, 99, 37, 63, 18\}$$

These are the same sets, even though the first two numbers in them have been swapped over.

So we would need another notation to represent the ordered set of numbers

14, 29, 6, 51, 99, 37, 63, 18

For your information, this ordered set in set notation this is written

$X = \langle 14,\ 29, 6, 51, 99, 37, 63, 18 \rangle$

which means,

X is the ordered set of elements 14, 29, 6, 51, 99, 37, 63, 18 in that order

Partitions

It is not absolutely necessary to be familiar with the idea of a partition in order to understand the topic of algorithms, but it can help.

To partition a set, means to divide it into two sets.

A list is an ordered set of elements. A partition of a list will be a division of the list into two sets, one comprising all elements below a certain element of the list and one comprising all elements above that certain element. The first set will be the set of lower elements of the list & the second set the set of upper elements of the list. The element where the partition is made must be included in either the lower list or the upper list.

For example,

S = the set of all elements from the list below and not including n_5

T = the set of all elements from the list above and including n_5

Then

$S = \langle 14,\ 26, 6, 51 \rangle$
$T = \langle 99,\ 37, 63, 18 \rangle$

We will begin by describing two algorithms for searching a list of numbers.

Recursive algorithm

A recursive algorithm is one that can call itself. This means that in a sequence of steps in the algorithm, one step instructs the algorithm to return to an earlier step in the algorithm. For example, the following algorithm is recursive:

STEP 1 LET $N_1 = 1$, $N_2 = 1$, LET $K = 3$

STEP 2 LET $N_K = N_{K-1} + N_{K-2}$

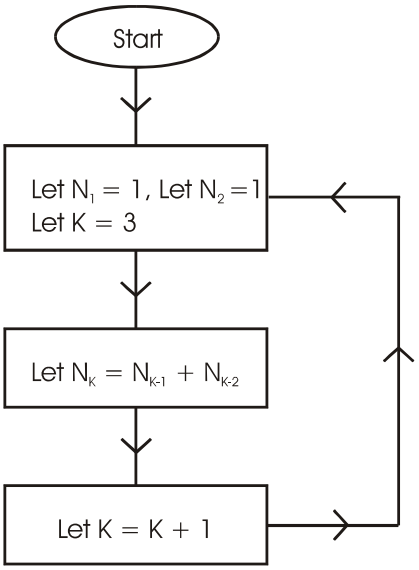
STEP 3 LET $K = K + 1$

STEP 4 GOTO STEP 2

It computes the Fibonnaci sequence

$N_1 = 1$
$N_2 = 1$
$N_3 = 2$
$N_4 = 3$
$N_5 = 5$ …

Step 4 returns the algorithm to an earlier stage of the computation. A flow diagram for this algorithm would be



The presence of the recursion process is shown in the flow diagram by the line returning the process to an earlier stage of the algorithm. This creates a loop in the program.

Note that there is no criterion in this programme for deciding when to stop the computation. This programme will work indefinitely – infinitely – computing the Fibonnaci numbers.

The algorithm for the Linear search is also a recursive algorithm – notice the way the linear search algorithm calls its own STEP 2 at STEP 4.

Linear Search

The idea behind a linear search is that we examine each element of the list in order and stop when the desired element is found.

The linear search algorithm locates the position of a certain value, *X*, within a sequence $N_1, N_2)\ldots. N_M$ which is in some kind of order, but not necessarily the order of lowest to highest. An algorithm for it is

STEP 1 LET $I = 1$

STEP 2 IF $I > M$ THEN PRINT "*X* HAS NOT BEEN FOUND" AND STOP

STEP 3 IF N(I) = *X* PRINT "THE POSITION OF *X* IS AT " PRINT *I* AND STOP

STEP 4 LET $I = I + 1$ AND GOTO STEP 2

A more efficient method of searching data is the binary search that can be illustrated by the following sample algorithm. The binary search algorithm requires that sequence be already sorted into order.

Binary Search

This algorithm locates the position of a certain value, *X*, within a sequence *N(1), N(2)…. N(M)* that has already been ordered.

We can illustrate this process on the following sequence: 2, 4, 13, 19, 25, 47, 51, 63 when $X = 47$.

Here $N(1) = 2,\ N(2) = 4$ and so on...

Since $N(6) = 47,$ and $X = 47,$ then the answer to the problem is $R = 6,$ where R is the position of X in the list.

The idea of the binary search algorithm is that we first compare our search number, X, with a number from the middle of the list. If that is the number we require then we have found the position of X; if not, then if the desired number is less than the currently searched number then we search the left hand list, and if it is greater, then we search the right hand list. This process of dividing the list into two is repeated recursively until either the item is found or a message is displayed indicating that the search has failed.

The binary search algorithm that we will describe will make use of the following function:

$$\text{int}\left(\frac{I+J}{2}\right)$$

This finds the largest integer less than or equal to the average of two integers I & J.

For example, if $I = 6$ and $J = 9$ then

$$\text{int}\left(\frac{I+J}{2}\right) = \text{int}\left(\frac{6+9}{2}\right) = \text{int}\left(\frac{15}{2}\right) = \text{int}\left(7.5\right) = 7$$

Here is a version of the algorithm for the binary search:

STEP 1: LET $I = 1$ AND $J = M$

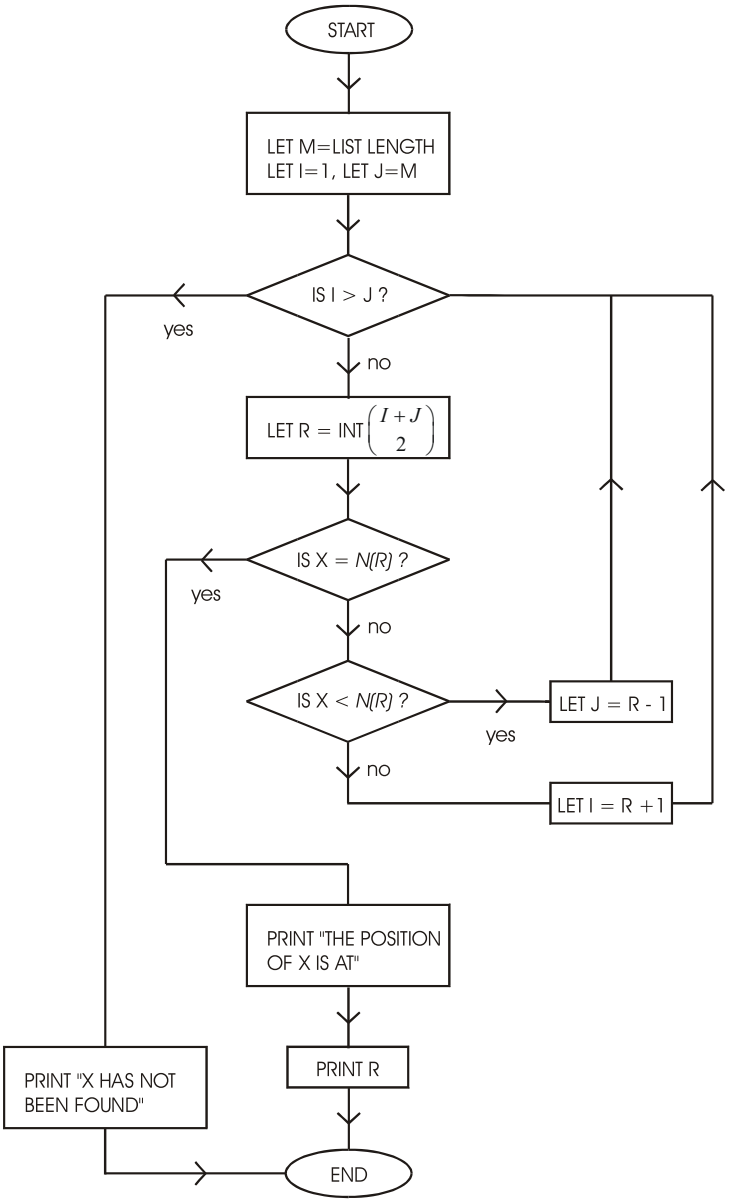
STEP 2: IF $I > J$ PRINT "X HAS NOT BEEN FOUND" AND STOP

STEP 3: LET $R = \text{int}\left(\frac{I+J}{2}\right)$

STEP 4: IF $X = N(R)$ THEN PRINT "THE POSITION OF X IS AT" AND PRINT R

STEP 5: IF $X < N(R)$ THEN LET $J = R - 1$ ELSE LET $I = R + 1$
GOTO STEP 2

A flow diagram for the binary search algorithm is



In this diagram the rectangles represent processes – the computation of functions, or, if you like, the following of instructions. The diagonal boxes represent decisions. Here the decisions are always based on precise mathematical criteria – there is no room for "freewill" in an algorithm. The outcome of the decision is always determined.

We can illustrate the binary search on the following ordered sequence: 2, 4, 13, 19, 25, 47, 51, 63 when $X = 47$ – that is we are searching for the position of the number 47 in the list. The answer, is, of course, 6. The process by which the binary search algorithm comes to the same answer is as follows

STEP 1: $I = 1,\ J = 8$

STEP 2: $I < J$

STEP 3: $R = \text{int}\left(\frac{1+8}{2}\right) = \text{int}\left(\frac{9}{2}\right) = 5$

STEP 4: $N(5) = 25,\ X = 47,\ X \neq \square N(5)$

STEP 5: $X > N(5),\ I = R + 1 = 5 + 1 = 6$

STEP 2: $I < J$

STEP 3: $R = \text{int}\left(\frac{6+8}{2}\right) = 7$

STEP 4: $N(7) = 51,\ X \neq \square N(7)$

STEP 5: $X < 51,\ J = 7 - 1 = 6$

STEP 2: $I < J$

STEP 3: $R = \text{int}\left(\frac{6+6}{2}\right) = 6$

STEP 4: $X = N(6) = 47$

THE POSITION OF X IS AT 6

For another illustration, take $X = 16$

STEP 1: $I = 1,\ J = 8$

STEP 2: $I < J$

STEP 3: $R = 5$

STEP 4: $N(5) = 25,\ \square X = 16,\ X \neq N(5)$

STEP 5: $X < N(5),\ J = 5 - 1 = 4$

STEP 2: $I < J$

STEP 3: $R = 3$

STEP 4: $N(3) = 13$, $X = 16$, $X \neq N(3)$

STEP 5: $X > N(3)$, $I = 3+1 = 4$

STEP 2: $I = J$

STEP 3: $R = 4$

STEP 4: $N(4) = 19$, $X = 16$, $X \neq N(4)$

STEP 5: $X < N(4)$, $J = 4 - 1 = 3$

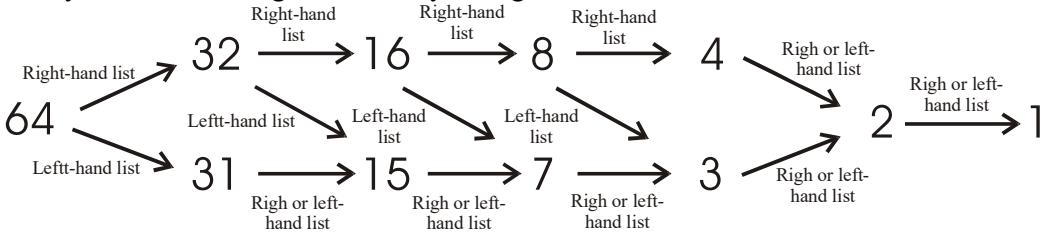
STEP 2: $I > J$

X HAS NOT BEEN FOUND

Efficiency of an algorithm

There are different ways of measuring the efficiency of an algorithm. One measure of efficiency is the number of runs through the repeat loop in the worst case for different sizes of data. We will compute the efficiency of this algorithm in the case that the size of data (i) M = 64, (ii) M = 32, (iii) M = 16, (iv) M = 8. The worst case means either the item is not found, or it is found at the last step.

To work out the efficiency we must work first the length of the list that has to be searched at each stage of the algorithm in the worst case. Starting with M = 64, either we search the left hand list with 31 elements or the right hand list with 32 elements. After searching a list of 31 elements both left and right hand lists will have 15 elements. After searching a list of 32 elements either we search the left hand list of 15 elements or the right hand list of 16 elements. After searching a list of 15 elements we search a list of 7 elements; after searching a list of 16 elements we search either a list of 7 elements or 8 elements. Searching lists of 8 or 7 elements leads to searches of lists of 3 or 4 elements; at the next iteration we search a list of 2 elements and finally 1 element. Diagrammatically, this gives:



Therefore, the relationship between number of iterations (efficiency) and sequence (list) length is:

List length	Efficiency
1	1
2	2
4	3
8	4
16	5
32	6
64	7

In the linear search efficiency increases proportionally to the length of the list, but for the binary search, efficiency increases as the square root of the list length.

Linear search efficiency $\propto n$

Binary search efficiency $\propto \sqrt{n}$

Where n = list (sequence) length

This shows how the binary search is more efficient. However, as already noted, it does require the data to be already sorted.

Arithmetic Processes

Arithmetic processes are also algorithms. For example, the evaluation of a polynomial function is an algorithm.

For example, suppose $f(x) = 2x^3 - 5x^2 + 2x - 9$ and we wish to evaluate $f(x)$ when $x = 3$.

Direct substitution results in

$f(3) = 2\times(3)^3 - 5\times(3)^2 + 2\times 3 - 9 = 6$

This requires 4 + 3 + 2 + 1 = 10 operations (separate multiplications and one final addition). A polynomial of degree $(n-1)$ would require

$n + (n - 1) + (n - 2) + \ldots + 2 + 1$ operations.

i.e. $\sum_{r=1}^{n} r = \frac{n(n-1)}{2}$ operations

A more efficient method, in the sense of requiring fewer multiplications, is Horner's method, also called the method of synthetic division. In this method, x is successively

factored out of the polynomial on the right.

$$\begin{aligned} f(x) &= 2x^3 - 5x^2 + 2x + 9 \\ &= \left(2x^2 - 5x + 2\right)x - 9 \\ &= \left(\left(2x - 5\right) + 2\right)x - 9 \end{aligned}$$

Then

$$f(3) = \left(\left(2 \times 3 - 5\right) \times 3 + 2\right) \times 3 - 9 = 6$$

In this case the algorithm requires 3 multiplications and 3 additions or 6 operations in all. Horner's method in general requires n multiplications and n additions in all. Thus for polynomial evaluations we can compare the efficiency of the two algorithms as follows:

Direct substitution efficiency = $\dfrac{n(n-1)}{2}$

Horner's method efficiency = $2n$

Where n = degree of polynomial.

The Big O Notation

To compare the efficiency of algorithms it is useful to employ the big O notation.

The big O notation is a measure of the rate of increase of a function. For example, consider the function:

$f(n) = 2n^3 - 5n^2 + 2n - 9$

When n is very small, the constant term (here – 9) dominates the others; as n grows larger the linear ($2n$) then the quadratic ($-5n^2$) terms dominate; but eventually for large n, it is the cubic term that determines the overall size of the value of the function. We say that $f(n)$ is of the order of n^3. This is written using a capital O as follows:

$f(n) = O(n^3)$

In general, a polynomial function of degree n has order n.

A formal definition of the big O notation is

Suppose $f(x)$ and $g(x)$ are functions defined on a set or sub-set of the real numbers $\mathbb{R}$. Then "$f(x)$ is the order of $g(x)$", which is written:

$f(x) = O(g(x))$

If there exists a real number K and a positive constant C such that for all $x > K$

$|f(x)| < C\,|g(x)|$

Using these definitions we can compare the complexity (efficiency) of the algorithms so far examined

Polynomial evaluation

Direct substitution	$O(n^2)$
Horner's method	$O(n)$

Search algorithms

Linear search	$O(n)$
Binary search	$O\left(\sqrt{n}\right)$