Algorithms and programming

Algorithm Constructs

An **algorithm** is a sequence of instructions used to solve a problem.

Note the word **sequence** in this definition. It means there is an **order** to the instructions.

Algorithm Constructs

Pseudocode is a method of writing down an algorithm.

Pseudocode does not use the syntax of any particular programming language but it does show the **structure** of a solution.

It is important that '**blocks**' and **loops** are **indented** to show the **structure** clearly.

There are not really any conventions used in pseudocode but I would suggest using the following:

- For any input/output, use...

```
input (mark)
output (totalmark)
```

- use appropriate variable names,...
- use **=** to assign values...

```
counter = 0
```

The basic '**constructs**' of an algorithm are :

Sequence : A number of instructions is processed **one after the other**.

eg

```
statement 1;
statement 2;
statement 3;
```

Pseudocode example

```
input(name)
input(age)
output(message)
```

Algorithms and programming

Selection : The next instruction to be executed depends on a '**condition**' :

```
if (condition is true) then
                statement 1
else
             statement 2;
 end if
```

Pseudocode example

```
input(name)
input(age)
if age < 21 then
    output(young message)
else
    output(old message)
end if
```

Iteration : A number of instructions is repeated....

(a)...a fixed number of times:

```
for 20 times do
    statement 1;
    statement 2;
end do
```

Pseudocode example

```
for 20 times do
    input(name)
    output(name)
end do
```

[Note in the program that a loop is used to draw the line. This is an example of a 'nested loop' - a loop inside another loop]

(b)....until a condition is met:

```
repeat
   statement 1;
   statement 2;
until (condition is met)
```

or

Algorithms and programming

```
while (condition is true) do
   statement 1;
   statement 2;
end while
```

Pseudocode example

```
repeat
   input (mark)
   total = total + mark
until mark = 0
```

Algorithm : Counts

A **variable** (integer) can be used to count values that satisfy a certain condition.

Remember counting always starts from 0 so the value of the variable must be **initialised** to 0.

Example

20 scores are to be input. How many of them are over 50?

We will use a variable called '**counter**' to store the number of scores over 50.
The pseudocode algorithm is..

```
counter = 0
for 20 times do
        input (score)
        if the score is over 50 then
                add 1 to counter
        end if
end do
output (counter)
```

Algorithm : Rogue Values

A sequence of inputs may continue until a specific value is input. This value is called a **rogue value** and must be a value that would not normally arise.

A rogue value lets the computer know that a sequence of input values has come to an end.

Example

**A number of scores is to be input (terminated by a rogue value of -1).
How many of them are over 50?**

Note that -1 is a value which would not arise.

Algorithms and programming

```
counter = 0
repeat
        input (score)
        if score > 50 then
                add 1 to counter
        end if
until score = -1
output(counter)
```

You will need to be careful that you do not 'process' the rogue value!
The general 'shape' of an algorithm which uses a rogue value will be...

```
counter = 0
Repeat
        input (item)
        if item is not the rogue value then
                process item
        end if
until item = rogue value
output(counter)
```

Linear search

A **search** is a method for finding an item of data.
The simplest type of search is a **linear** search (**serial** or **sequential** search), which basically starts at the beginning of a file, reading each record until the required record is found.
Not an efficient method of searching but if the data is **not sorted**, this is the **only** method of searching.
Pseudocode:
Assume an array of strings **Key[n]**. The string to be searched for and processed is **SearchKey**

```
input(SearchKey)

n = 0

repeat

        if key[n] = SearchKey then

           process SearchKey

        end if

        n = n + 1

Until key[n-1] = SearchKey
```

Algorithms and programming

Binary search

If the data is **sorted** then a fast search which can be used is a **binary search**.

This involves looking at the '**middle**' record. If the required record is before this one, then **discard** the second half of the file; if it is after this one, then **discard** the first half of the file.

Repeat the process, discarding half the file each time, until the required record is found.

Pseudocode:

Assume the data keys are stored in an array **Key**[1..n] and we are searching for the data whose key is **WantedKey**

```
first = 1;
last = n;
repeat
  m = (first+last) div 2;
  if WantedKey < Key[m] then
             last = m-1
  else
             first = m+1;
until Key[m] = WantedKey;
```

Note that this algorithm assumes that the data key is not missing. A better algorithm would be:

```
first = 1;
last = n;
found = false
repeat
  m = (first+last) div 2;
  if WantedKey = Key[m] then found := true;
  if WantedKey < Key[m] then
                   last = m-1
  else
                   first = m+1;
until found or (last < first);
```

Note that '**until found**' is short for '**until found = true**'.

Algorithms and programming

Bubble sort

Searching for Data

If you had to find a name in a very long list...is it quicker if the list is jumbled up or sorted into alphabetical order? Well the answer is obvious...and gives us the reason why sorting data is useful.

To **sort** data is to place the data **in order** (numerical or alphabetic).

If the amount of data is small enough to be held in memory then an **internal sort** may be used.

Bubble sort

Example :

23	12	34	17	21	18

Start from the left and compare each pair of numbers. If the smallest is on the left, then leave them. If not...swap them over.

Move on to compare the next pair of numbers.

C = Number of comparisons
E = number of exchanges (swaps)

After First Pass :	12	23	17	21	18	**34**	C=5	E=4

(after the first pass the largest number should be at the right...so on the next pass, we need one less comparison)

After 2nd Pass :	12	17	21	18	**23**	**34**	C=4	E=3

(the last two numbers are in the correct position)

After 3rd Pass :	12	17	18	**21**	**23**	**34**	C=3	E=1
After 4th Pass :	12	17	**18**	**21**	**23**	**34**	C=2	E=0

Once a pass produces **no exchanges** then the file is sorted and the algorithm can stop.

Notes :

Algorithms and programming

The bubble sort is quick for '**nearly sorted**' files. For example, files which have been recently sorted but may have had a few more records added at the end.

Pseudocode :

```
repeat
set a flag to False
for each pair of keys
      if the keys are in the wrong order then
          swap the keys
          set the flag to True
      end if
next pair
until flag is not set.
```

A more detailed (and more helpful) pseudo-code :
Assume records are held in an array **key**[1..n]

```
Set n to number of records to be sorted
repeat
   flag = false;
   for counter = 1 to n-1 do
            if key[counter] > key[counter+1] then
                    swap the records;
                    set flag = true;
            end if
   end do
   n = n-1;
until (flag = false) or (n=1)
```

Problem Analysis

Top-down Design

When confronted by a large task.......... break it down into a number of smaller tasks. (If you have to move a mountain...... move one boulder at a time!).

Top-down design is the technique of breaking down a problem into a number of smaller problems. Each of these is broken down into even smaller problems and so on..... until each problem is sufficiently simple to be solved easily.

These 'small' problems are referred to as **modules** and should be as **self-contained** as possible....this means that each module can be solved on its own without depending on the solution of other modules.

Top-down design is sometimes called **stepwise refinement**.

Non-programming example:

The problem :	Put on a musical show

Algorithms and programming

Break it down	Choose cast Rehearse Perform show

The first steps might be broken down again....

Choose cast	Choose singers Choose speaking parts Choose dancers
Rehearse	Select times to rehearse Develop a rehearsal schedule Arrange use of a hall Rehearse
Perform show	Hire costumes Hire lighting and sound Sell tickets Performances

and these could be broken down again.....

Standard modules, subroutines and functions

All programming languages have **standard functions**. These are already-defined instructions that perform some sort of calculation on values (**parameters**) that are passed to it.

For **example** a function may calculate the length of a text string or the square root of an integer.

We have learned that it is better to break down large and complex programming tasks into smaller ones that are easier to manage.

A programmer may define their own small sub-programs that can be '**called**' (run) from anywhere else in the program using a single instruction...and as many times as necessary.

There are two types of user-defined **sub-program** ...

1. **Subroutine** - which performs a specific task...eg draws a box on screen
2. **Function** - which returns an **answer**...eg a mathematical calculation

Every subroutine has a **name**. The subroutine can be called by using its name.

Another good programming practice is to put subroutines and functions into a '**standard module**'.

Standard modules are really **libraries** of subroutines and functions which may be used in any program. The advantages of this are ...

- **Time** is saved as the programmer does not have to write the same modules again.
- The subroutines have already been **tested** so they will work and there is no need to test them again.
- Programs will end up being **shorter** and therefore easier to modify.

Algorithm Testing

How do you know if your algorithm (program) works?

Well..you **test** it...Try it out with a variety of test data and see if any problems arise.

One technique is to **dry run** the program.

This involves writing down a table of the instructions which are executed and noting the values of all the constants and variables as each instruction is executed. Such a table is called a **trace table**.

Example

This is a procedure which does nothing sensible (except illustrate how a dry run should be done!).

Suppose we wanted to test a procedure called '**Frenzy**' which uses a parameter '**Num**'.

```
procedure Frenzy( Num : integer);
var i ,fred : integer;
begin
        set fred = 5;
        for i = 1 to Num do
        begin
                set fred = fred - i;
                output(fred);
        end;
        output('Done');
end;
```

To dry-run this procedure we need to create a **trace table** with **column** headings:

- the **instruction** being executed
- the values of each **variable** after execution (**Num**, **i** and **fred**)
- any **output** done

The **trace table** for the procedure **Frenzy(3)** would be...

Instruction	Num	i	fred	Output
Start up..	3			
set fred = 5	3		5	
for i = 1 to Num do	3	1	5	
set fred = fred - i	3	1	4	

Algorithms and programming

output(fred)	3	1	4	4
for i = 1 to Num do	3	2	4	
set fred = fred - i	3	2	2	
output(fred)	3	2	2	2
for i =1 to Num do	3	3	2	
set fred = fred - i	3	3	-1	
output(fred)	3	3	-1	-1
output('Done')	3	3	-1	Done

Notes about the above...

- **begins** and **ends** are ignored - they do not affect any values
- each time the **loop** instruction (for i := 1 to Num) is executed the value of i is incremented.
- the **parameter** value is immediately entered at the beginning.

Even if this procedure does nothing sensible, it is a good procedure because it does not use values or results from other procedures. ie we can test it on its own.

Okay..so you need a lot of paper...but it is a useful technique if you have a program which does not work properly....and it really is useful when doing Assembly Language programming...but that's another story....

Selection of Test Data

When testing an algorithm, you need to run a **sequence** of tests. If one test works, that does not mean the program will work every time!

Select...

- data which is '**normal'**
- data which is **extreme**
- data which is **incompatible** ie data of the wrong type
- data which is **non-existent** ie zero or null data

Example

If you are testing a procedure which calculates the VAT on the price of a purchased item, choose

- sensible prices eg £12.50
- very large or very small prices eg £0.01 or £12,000,000,000,000,000.00
- try giving it a string! eg 'Hello'
- try £0.00 or even a null input.

Constants and Variables

Algorithms and programming

All elements of data used in a program are given **identifiers** (names) to distinguish between them.

These elements of data have **values**, and may be one of two types:

- **constants** - the value remains the same every time the program is run, and does not change throughout the running of the program.
- **variables** - the value may change.

Each programming language has its own rules about the naming of identifiers.

Eg. It must start with a Letter and have no spaces.

In programming, all variables and constants must be **declared**. When a declaration is made, the computer **reserves space in memory** for storing its **value**, and labels this space with the identifier.

The **scope** of a variable (or constant) is the section of program where a variable's value may be accessed and used.

Global variables exist throughout the run of a program. They are declared at the start of a program and values may be accessed at any time during the run of the program.

Local variables are declared in a **subroutine** or **function**. They are created when the subroutine is started and only exist for the time the subroutine is run. When the subroutine is completed the memory space reserved for local variables is released for other use.

Note : It is important that variables and constants should be declared locally if possible. This saves memory space when programs are run.

Logical Operators

There are 4 main **logical operators** which may be used in algorithms (programs)...

- **NOT**
- **AND**
- **OR**
- **XOR**

Examples :

If NOT (x = 100) then output(message)
means that the message will be output if the value of x is **not** equal to 100.

If (x > 19) AND (x < 50) then output(message)
means that the message will be output if both conditions (x > 19) **and** (x < 50) are true...ie if x lies in the range 20..49 (inclusive).

Truth Table for **AND** :

AND	True	False

True	True	False
False	False	False

If (x < 20) OR (y < 30) then output(message)
means that the message will be output if one **or** other (**or both**) of the conditions (x < 20) or (y < 30) are true.

Truth Table for **OR** :

OR	True	False
True	True	True
False	True	False

If (x < 20) XOR (y < 30) then output(message)
would output the message if one **or** other (but **not both**) of the conditions (x < 20) or (y < 30) are true.

Truth Table for **XOR** :

XOR	True	False
True	False	True
False	True	False

These logical operators are useful in **SQL (Structured Query Language)** statements which are used for searching databases.

Example

Suppose you wanted to find all the people called Smith who live in Cardiff but do not have either a dog or a cat...your query may include something like...

(name='Smith') and (town='Cardiff') and not((pet='dog') or (pet='cat'))

Levels of Computer Language

Computers can only execute programs in binary **machine code**.

In the beginning....programmers had to write programs in machine code...
A machine code program would look like..

```
00101000
10110011
10010101
01001011
etc...
```

Algorithms and programming

Programming was difficult and tracking down errors (**debugging**) must have been a nightmare!

Assembly Language was developed which gave **mnemonics** (meaningful abbreviations) to the machine code instructions.

Assembly language programs look more readable...

```
MOV  AL,5
CLC
MOV AH,2
INT 21H
etc...
```

..but only just!

Programming languages are classified as **Low level** (near to the language a computer uses - ie machine code) or **High level** (near to the language a human uses).

Low level (Assembly) languages are

- Computer orientated
- Difficult for programmers to develop code and to test and modify it. Simple tasks need a lot of instructions.
- Used to develop programs that need to run very fast.

High level languages started to be developed in the mid 1950s - to make it easier for programmers to develop complex programs. They are...

- Problem orientated
- Easier for humans to understand and use.
- Used to develop programs that can run on different computers.

Examples of high-level languages are :

- **FORTRAN** (FORmula TRANslator); mainly used for engineering/scientific computing.
- **COBOL**(COmmon Business Orientated Language). Still one of the main commercial data processing languages.
- **ALGOL** (ALGOrithmic Language) was also developed in the 1950s and many of the languages such as **C** and **C++** are developed from it.
- **BASIC** (Beginners All-purpose Symbolic Instruction Code) was developed in the 1960s as a simple programming language.
- **PASCAL** was developed in the 1970s as a well-structured teaching language.
- **JAVA** - a recent language developed from C++; an object-oriented language useful for developing programs which work over the Internet.
- Other high-level languages include : **LISP**, **PROLOG**, **PL/1** , **ADA**

Algorithms and programming

Scientific languages (like FORTRAN and ALGOL) would have high-precision mathematical functions, whereas **commercial** languages (like COBOL) would have powerful file handling facilities.

Scientific languages would be needed for eg. weather forecasting, real-time control systems, etc

Commercial languages would be needed for eg. information retrieval systems.

Types of High Level Language

Procedural Languages

A **sequence** of instructions is run. There is a starting point and a logical order to the instructions to be executed, until the end point is reached.

Procedural languages use **program control** constructs (If..Then, Loops, Subroutines and Functions).

Examples : PASCAL, BASIC, FORTRAN, COBOL.

Event-driven languages

A program that waits for **events** such as the clicking of the mouse or the press of a key on a keyboard.

When an event occurs, it is processed using a defined sequence of instructions called an **event handler**.

Useful for control programs where events such as readings from sensors are used to control devices.

Examples : Visual Basic, C++, Javascript

Visual Languages

Allow the programmer to manipulate objects visually on a form, setting their layout and properties. The underlying program code is automatically generated.

Used for creating Windows (GUI) applications.

Examples : Visual Basic, Visual C++, Delphi

OOP (Object-Orientated Language)

A programming language where objects are defined.(See CG3)

Objects have **Properties** and **Methods**.

Properties can be set initially or changed at run-time.

Examples : Visual Basic, C#, JAVA, PYTHON

Algorithms and programming

Methods are things that the object can do.

Mark-up Languages

Special coding instructions are used to indicate style and layout of text and other elements.

Widely used for creating web pages on the Internet.

Examples: HTML. XML, XHTML, ASP

Programming methods

Much of the work of **software engineers** (programmers) is involved with altering programs which some other programmer has written. This can be made easier if the program is a **good** program....but what do we mean by that?

The characteristics of a '**good**' program....

- the program is organised into **small sections** ...ie subroutines (*small rocks are easier to lift than large rocks*)
- the program is self-**documenting**
    - There are plenty of **comments** in the program identifying what sections of the program do etc...
      (*you know you have picked up the right rock if it has a label on it*)
    - **Meaningful names** are used for constants and variables, subroutine names etc
      E.g. It is better to call a variable **Tax_Level_Indicator** rather than just **TL** even if it does mean more typing when coding!
- **Structure** is clearly shown - using indentation of blocks.