AQA GCSE

COMPUTER SCIENCE

Steve Cushing

SAMPLE CHAPTERS

DYNAMIC LEARNING

HODDER EDUCATION

LEARN MORE

Textbook subject to change based on Ofqual feedback.

Contents

The Publishers would like to thank the following for permission to reproduce copyright material:

Photo credits

p. 6 *t* © TfL from the London Transport Museum collection, *b* © TfL from the London Transport Museum collection

Every effort has been made to trace all copyright holders, but if any have been inadvertently overlooked the Publishers will be pleased to make the necessary arrangements at the first opportunity.

Although every effort has been made to ensure that website addresses are correct at time of going to press, Hodder Education cannot be held responsible for the content of any website mentioned. It is sometimes possible to find a relocated web page by typing in the address of the home page for a website in the URL window of your browser.

1 Computational Thinking

KEY POINT

Computational thinking involves applying a set of problem-solving skills and techniques that are used by computer programmers to write programs.

KEY TERMS

Problem-solving skills and techniques refer to the designing of systems, and understanding concepts such as decomposition and abstraction which are fundamental to computer science.

Output is anything that a computer produces.

Logical reasoning is the process or method of using a rational and systematic approach to solving a problem. It will often be based on mathematical assumptions and procedures.

Before you can succeed in computer science you must learn about what is called 'computational thinking'. Computational thinking involves applying a set of **problem-solving skills and techniques** that are used by computer programmers to write programs. Computational thinking is not thinking about computers or even thinking like a computer. Computers don't think for themselves. If you give ten computers the same instructions and the same input, they will give exactly the same **output**. Computers are predictable.

Computer scientists use **logical reasoning** to work out exactly what a program or computer will do. Computational thinking involves thinking about a problem in a logical way, and enabling a computer to solve it. This logical reasoning is the essential building block of computer science, so first we need to fully understand the techniques involved and how we start with a problem and end up with the programming code.

KEY TERMS

Decomposition is breaking any given task or problem into simple logical steps or parts.

Abstraction is the process of taking away or removing irrelevant characteristics. In terms of solving a problem, this would mean removing irrelevant characteristics in order to reduce it to something that is simpler to understand. In computer science, abstraction is often used for managing the complexity of computer systems.

KEY POINT

Decomposition means breaking a problem into a number of sub-problems, so that each sub-problem accomplishes an identifiable task, which might itself be further subdivided.

KEY POINT

Decomposition is a term used for the separation of a task into discernible parts, each of which is simpler than the whole.

Two important techniques used in computational thinking are:

- **Decomposition**: This is breaking any given task or problem into simple logical steps or parts.
- **Abstraction**: This is the process of taking away or removing characteristics from something in order to reduce it to something simpler to understand. In computer science, abstraction is often used for managing the complexity of computer systems.

We will explore each of these in detail later in the book but let's start with a simple example of decomposition and abstraction as it relates to problem solving.

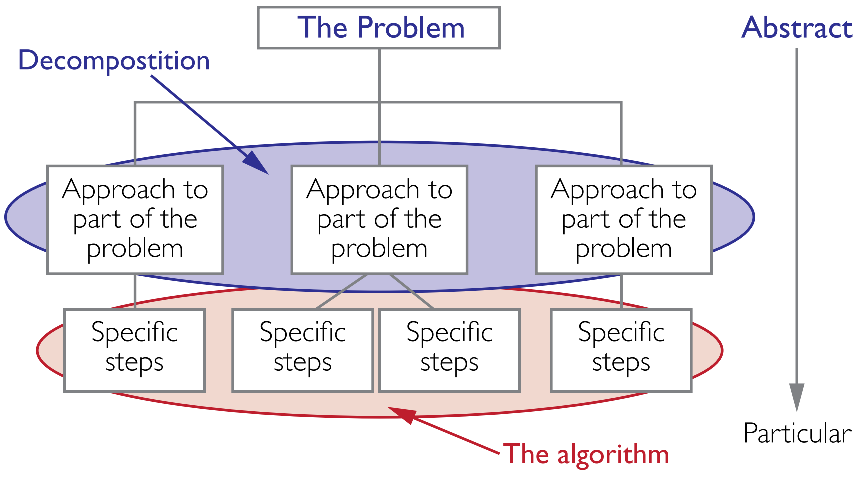


Figure 1.1 *Decomposition and Abstraction*

Decomposition

When a chef writes a recipe for a meal, that chef is creating a set of instructions that others can then follow to replicate the meal. Each part of the recipe is listed separately. The overall meal is decomposed into separate dishes, and these are often decomposed further for example, making the pastry and the filling.

Abstraction

You may have come across the term 'Abstract Art', where a painting is a set of shapes representing the scene. A good example of the use of abstraction for technical purposes is the London tube map. It is the brainchild of an electrical draughtsman named Harry Beck.

Rather than emphasising the real distances and geographical location of all the tube lines, Beck stripped away the sprawling tube network by abstracting just the information needed by travellers. He then used this to create an easy to read diagram of coloured, criss-crossing lines common in electrical diagrams.

KEY POINT

In abstraction we remove unnecessary details from a problem until the problem is represented in a way that is possible to solve.

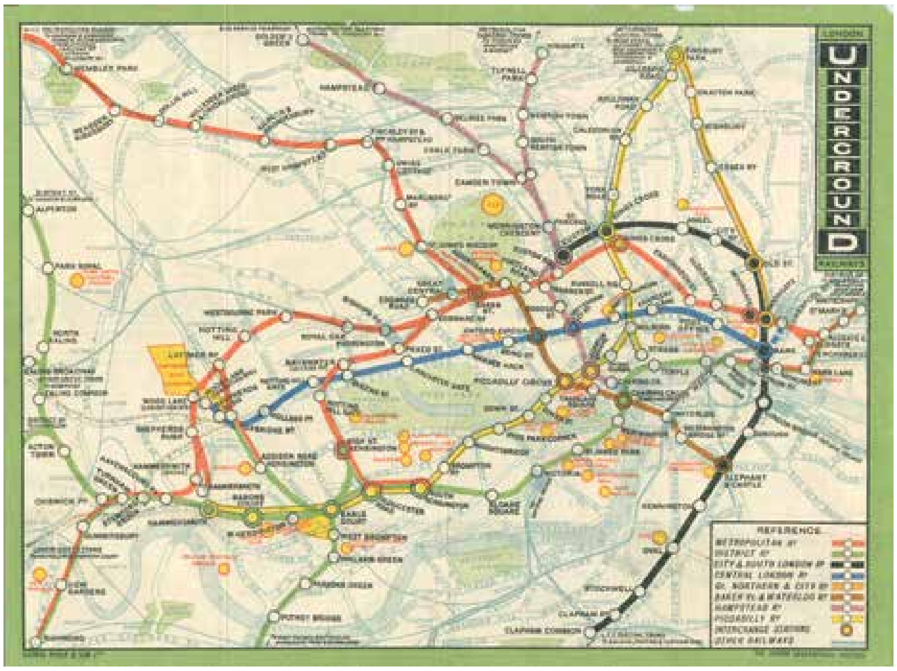


Figure 1.2 *A geographical map of the London underground stations*

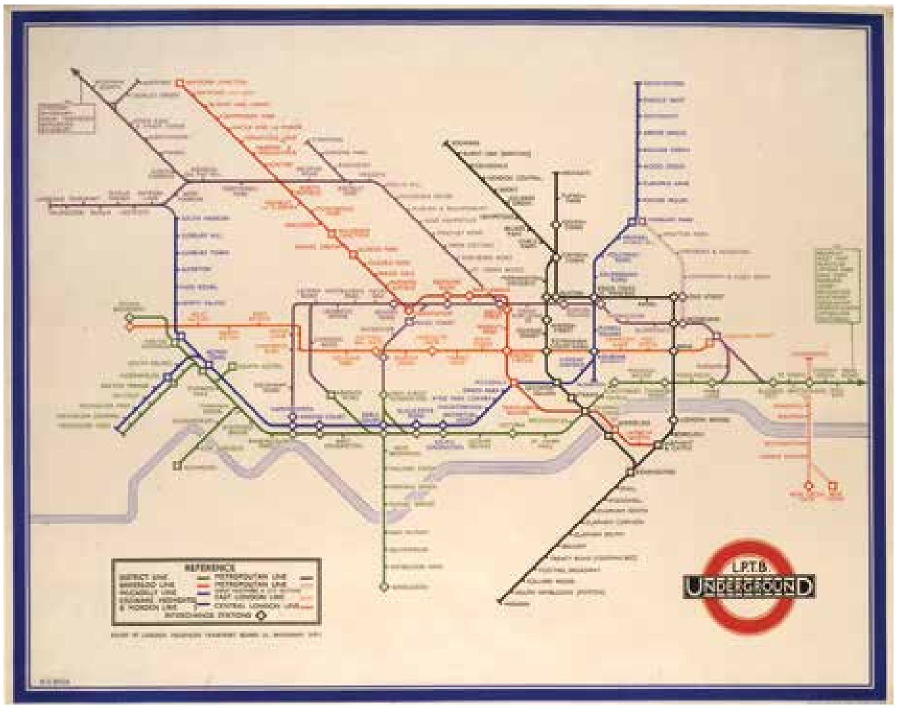


Figure 1.3 *An original abstracted map of the London underground*

Let us look at a simple example of abstraction.

EXAMPLE

Figure 1.4 *Fred wants to cross a river*

A man called Fred wishes to cross a 10 metre wide river with a wolf, a white goat and a bail of newly cut hay. He has a small blue boat and oars, but unfortunately he can only take one thing across at a time. The problem is, if he leaves the wolf and the goat alone together, the wolf will eat the goat, and if he leaves the goat with the hay, the goat will eat the hay. They are currently all together on one side of the river, which we will call bank B, and they want to get to the other side, called bank A.

How does he do it?

There is a simple computational approach for solving this problem.

Of course, you could simply try all possible combinations of items that may be rowed back and forth across the river. Trying all possible solutions to a given problem is referred to in computer science as a **brute force approach**. But logical thinking will bring about a better solution.

Only the relevant aspects of the problem need to be represented, all the **irrelevant** details can be ignored. A representation that leaves out details of what is being represented is a form of abstraction. So what can we leave out?

```
Is the man's name relevant?
Is the width of the river
relevant?
Is the colour of the boat
relevant?
```

KEY POINT

A representation that leaves out unnecessary details of what is being represented is a form of abstraction.

```
We can start with the following
bits of information:
```

- `River banks are A and B`
- `Goat = G`
- `Hay = H`
- `Wolf = W`
- `Man = M`

So to start with we have:

```
A          B
           G H W M
```

But we need to end up with:

```
A          B
G H W M
```

Each step we show needs to correspond to the man rowing a particular object across the river (or the man rowing alone).

Let's look at the first step:

```
A          B
G M        H W
```

The man (M) has taken the goat (G) to the other side of the river.

TASK

Solve the rest of the river-crossing problem.

KEY POINTS

- Abstraction is the process of removing unnecessary detail from a problem.
- Abstraction draws out the essence of a problem. By solving it we can also see what other problems can be solved using the same techniques.

More than one solution to any problem

There will often be more than one solution to the same problem, but you always need to create **ordered steps** to achieve any of these solutions. Let's look at another simple problem.

Imagine a map; you are given a starting point and the point you wish to arrive at. The map contains a grid to help navigation. The map grid has numbers in the vertical axis, and letters in the horizontal axis. Let's say we start at 10 C and want to arrive at 15 L.

Figure 1.5 shows four possible pathways. There are of course many more. We could take a very complicated route, but we want to be efficient and take as few moves as possible.

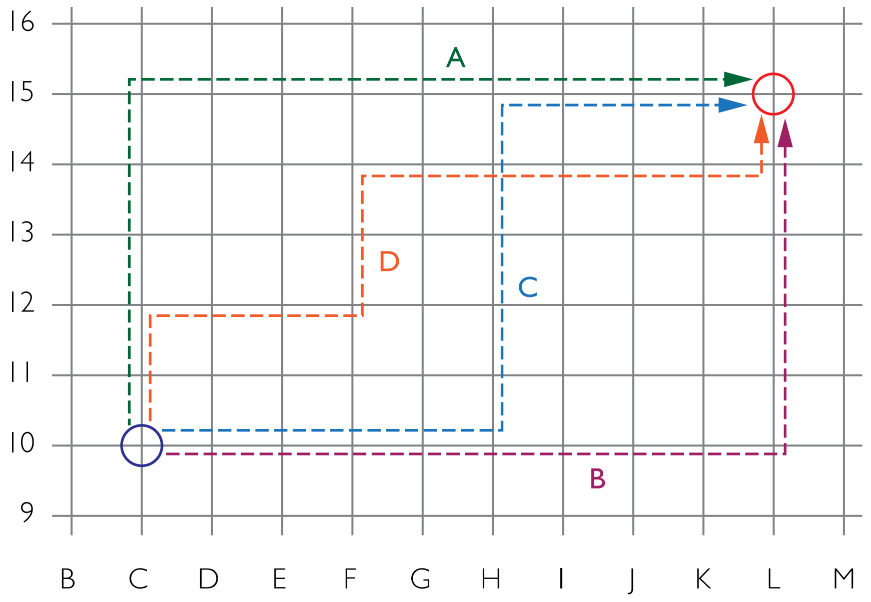


Figure 1.5 *Four possible pathways*

We could describe each of these pathways using words. Pathway 'A' for example could say move north until you reach map reference 15, then turn right 90°, now move forward to map reference 'L'.

We can also describe the path using distances rather than the grid positions. For example, move forward five, turn right 90°, move forward nine.

Of course both of these directions will only work if people follow them exactly. We have abstracted the problem as an example. We could make the directions better by **refining the instructions** and adding more detail, perhaps by informing

the user what to do if they go wrong. We could add a **position check**. If you can understand these concepts, you are well on the way to being able to write computer programs.

Choosing the best solution

So we know there can often be many answers to the same problem, but we need to determine what makes the best solution and would lead to the best algorithm.

The first set of criteria we need to consider are:

- does the solution work?
- does the solution complete its task in a finite amount of time (within set boundaries)?

We have lots of solutions to our problem and each, whilst very different, satisfies these two criteria. Therefore, the next step is to determine which of our solutions is 'best'.

There are generally two criteria used to determine whether one computer algorithm is 'better' than another. These are:

- the **space requirements** (i.e. how much memory is needed to complete the task)
- the **time requirements** (i.e. how much time will it take to complete the task).

Another criterion that we can consider is the cost of **human coding time**. This is the time it will take us to develop and maintain the program. A clever **coding** system may improve the space and/or time requirements but result in a loss of program **readability** and an increase in the human cost to maintain the program.

KEY TERMS

Space requirements are how much memory is needed to complete the task.

Time requirements are how much time it will take to complete the task.

Human coding time is the time it will take us to develop and maintain the program.

Coding is a process that turns computing problems into executable computer code.

Readability is making the code easy for another person to read and easier for you to fix all of the bugs.

KEY POINTS

- An algorithm is simply a set of steps that defines how a task is performed.
- A program is a sequence of instructions to perform as task.
- An algorithm is not a computer program; a computer program is the implementation of an algorithm.
- Programs must be carefully designed before they are written. During the design stage, programmers use tools such as pseudo-code and flowcharts to create models of programs.

What is an algorithm?

The word 'algorithm' comes from the ninth-century Arab mathematician, Al-Khwarizmi, who worked on 'written processes to achieve some goal.' The term 'algebra' also comes from the term 'al-jabr,' which he introduced.

Algorithms are at the very heart of computer science. An algorithm is simply a set of steps that defines how a task is performed. For example, there are algorithms for cooking (called recipes), algorithms for finding your way through a strange city (directions), and algorithms for operating washing machines (manuals). There are even algorithms for playing music (sheet music).

A scientific description of an algorithm would be:

'a series of unambiguous steps to complete a given task in a finite amount of time.'

TASK

What is an algorithm?

An algorithm has input data, and is expected to produce output data after carrying out a process which is the actions taken to achieve the required outcome. You will need to understand this input–process–output model as much of what you will learn in computer science is founded upon this model.

The input–process–output model

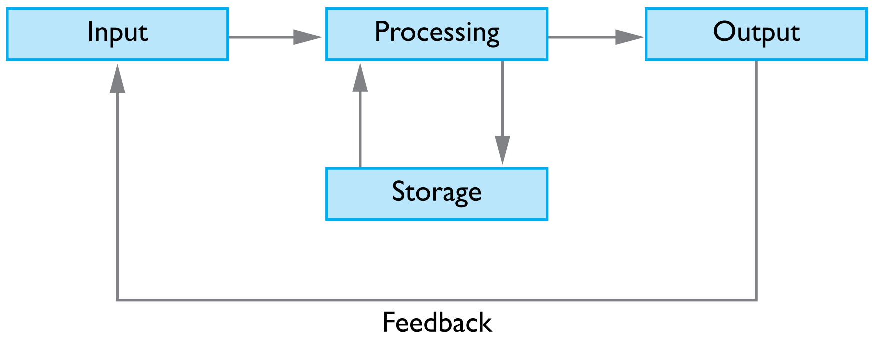


Figure 1.6 *The input–process–output model*

A computer can be described using a simple model, as shown in Figure 1.6.

KEY POINT

You must be able to identify where inputs, processing and outputs are taking place within an algorithm.

The INPUT stage represents the flow of data into the process from outside the system.

The PROCESSING stage includes all the tasks required to affect a transformation of the inputs.

The OUTPUT stage is where the data and the information flow out of the transformation process.

You will notice that we have added two new parts to the model: storage and feedback.

The STORAGE stage keeps the data until it is needed.

In solving any problem, you must also follow this model. First you define the problem, then you define what the solution must be, and finally you work on the transformation (process)

to achieve the desired solution. FEEDBACK occurs when outputs of a system are fed back as inputs that form a circuit or loop.

Sometimes programmers even plan out their code using these headings.

Sequences

To solve a problem there must be a **sequence**. In computer science, a sequential algorithm is an algorithm that is executed sequentially, one step at a time from start to finish. It does this without any other processes executing. Most standard computer algorithms are sequential.

EXAMPLE

Say we want to input two numbers, add them together and show the answer on the screen.

Table 1.1

Input	Process	Output
Two numbers	Add the first number to the second	The new number

We could write this as the following sequence:

```
INPUT first number                      # input stage
INPUT second number                     # input stage
ADD first and second number together and
STORE as total                          # process stage
OUTPUT total                            # output stage
```

Saving an INPUT with a name – for example in this instance we input a number and called it 'first number', then we input another number and called it 'second number' – is called **assignment**. It is called this because we 'assign' a value to the variable. In computer programming, an assignment statement sets and/or re-sets the value stored in a storage location. We will explore this in more detail later in the book.

Why are sequences so important?

In computer programming you have to first work out the correct **sequence** of the commands. This may sound simple but let's look at an example to show how careful you need to be. If you write down your friend's address it may look like this:

John Smith
22 Holly Road
Hempton
London
AB12 3CD

You know that this is the order that you should write an address, but this is the exact opposite of the way that the postal system works. There are millions of John Smiths. There may be hundreds of Holly Roads, many of them with a number 22. The postal system needs to know the address in the logical task order, meaning the order that you carry out finding the address. In this case that is London first, followed by the area, then the road and finally the number.

KEY POINT

Many computer programmers label their files using the date format year, month, day as this is the logical way to automatically list them, the year being the first piece of data required, the month the next and the day last. This is because there can be 12 files with the same day number in a single year.

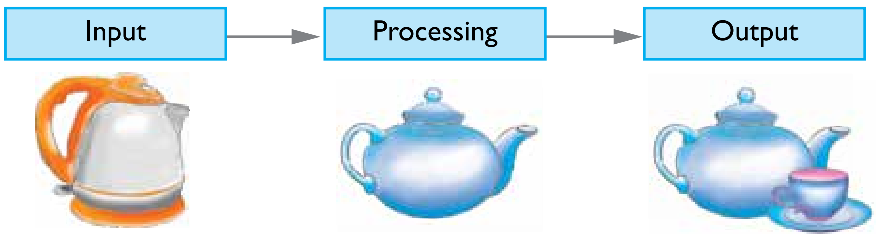
In some countries the conventional order follows the logical task order. In Russia, letters are addressed in exactly the opposite order to the UK, with the city first.

For your programming to work correctly, all the commands have to be there *and* they need to be in the correct sequence. Sequencing is extremely important in programming.

Decomposition and sequences

Let's look at a simple problem in terms of input, process and output.

Sample System Diagram



- Water
- Tea bags
- Boiling water
- Milk
- Sugar

- An electric kettle is filled with water
- Tea bags are placed in the tea pot
- The boiling water is poured into the tea pot and the tea brews
- Milk is added to the cup
- Sugar is added to the cup
- The 'brewed' tea is poured into a cup

- Tea

Figure 1.7 *A sample system diagram*

Let's say we want to make a cup of tea using tea bags and a kettle.

The sequence could look like this:

Table 1.2

Input	Process	Output
• Water • Tea bags • Boiling water • Milk • Sugar	• An electric kettle is filled with water • Tea bags are placed in the tea pot • The boiling water is poured into the tea pot and the tea brews • Milk is added to the cup • Sugar is added to the cup • The 'brewed' tea is poured into a cup	• Tea

We would do the same to make more than one cup of tea but we would only fill the kettle and boil it once. We could divide the sequence into two parts.

Sequence one

```
Count the number of people wanting tea    #input
Fill the kettle with enough water         #input and process
Boil the water                            #process
```

Sequence two

```
For each person wanting tea    #requires input from first sequence
  Put tea bag in pot           #input and process
  Pour on water                #input and process
  Add sugar                    #input and process
  Add milk                     #input and process
Allow tea to brew              #process
Serve tea                      #output and process
```

Indents have been used to show the parts of the sequence we would repeat.

KEY TERMS

A **Variable** is a value that can be changed.

We would need to carry the number of people wanting tea from the first sequence to the next sequence so we can repeat the second sequence of the task in order to make a cup of tea for each person, but only boil the water once. We could do this by creating what is called a **variable**.

```
Count people wanting tea
Store answer in a variable called teaCount
Fill the kettle
Boil water
Repeat the following steps for the number stored
in the variable called teaCount
   Put teabag in cup
   Pour on water
   Add sugar
   Add milk
End Repeat
Serve
```

We could of course extend this by asking who wants sugar or milk. This is called an IF statement construct.

```
Do you require sugar?
If answer is yes
   Add sugar
End If construct
Do you require milk?
If answer is yes
   Add milk
End If construct
```

We could also run parts of the task in parallel. For example, whilst the kettle is boiling we could add the teabags to the cups, and we would almost certainly add all the teabags to the cups before adding the water. We would not add one teabag, then add the water to that cup before adding the next teabag.

Explore all these possibilities and represent them as simple English sentences. If you do this you have just decomposed a problem and have started to create a program. We will explore all of the concepts you need such as ‘loops’ and ‘if’ statements in more detail later in the book. For now, you just need to

understand how to break a problem down into simple steps and how to group these steps into separate parts of the task.

Modularity

There are several advantages to designing solutions in a structured manner. One is that it reduces the complexity, as each set of steps can act as a **separate module**. Modularity allows the programmer to tackle problems in a logical fashion. Modules can also be reused.

To develop modules the programmer needs to carry out what is called decomposition. This is to break down a problem into easy-to-understand steps.

Because modules can be re-used many times, it saves time and reduces complexity, as well as increasing reliability as the modules will have already been tested in another program. It also offers an easier method to update or fix the program by replacing individual modules rather than larger amounts of code.

Structured programming makes extensive use of subroutines, block structures and 'for' and 'while' loops. We will explore all of these later in the book.

KEY POINTS

- A subroutine is a sequence of instructions that is set up to perform a frequently performed task
- A procedure is a subroutine that does not return values.
- In pseudo-code a computer can repeat a group of actions using REPEAT-UNTIL.

Writing Algorithms and Code

Each instruction should be carried out in a finite amount of time. An algorithm, given the same input, will always produce the same output. During the processing the underlying code will always pass through the same sequence of states.

Since we can only input, store, process and output data on a computer, the instructions in our algorithms will always be limited to these functions.

First, we must not only fully understand the problem but give each item a name before solving it:

- Identify and name each Input/Given
- Identify and name each Output/Result
- Assign a name to our algorithm (Name)
- Combine the previous three pieces of information into a formal statement (Definition)
- Results = Name (Givens)

KEY POINT

The term 'Call' to subroutine means the code inside the subroutine should be executed.

Recording your ideas

Once we have abstracted the necessary data and understood the sequences involved, rather than writing long text explaining the problem and its solution, we need to find a way to record our thinking and the method we will use to solve the problem. The most effective way is to use either a **flowchart** or **pseudo-code**.

KEY TERMS

Flowcharts show a sequence of events or movements involved in a complex activity or process.

Pseudo-code is an easy-to read language to help with the development of coded solutions.

CHAPTER REVIEW

In this chapter we have explored computational thinking including decomposition and abstraction. We also looked at the input, process and output model and explored the importance of sequences.

Remember before tackling any computer science task or examination question on this topic you must:

- be able to take a complex problem and break it down into smaller problems
- be able to work out the sequences needed
- understand and explain the terms algorithm and decomposition
- understand and explain the term abstraction and manage the complexity of the task by abstracting the key details.

2 Using Flowcharts

KEY POINTS

- A flowchart is a diagram representing an algorithm.
- Flowcharts are a graphical method of designing programs.
- A well-drawn flowchart is easy to read.

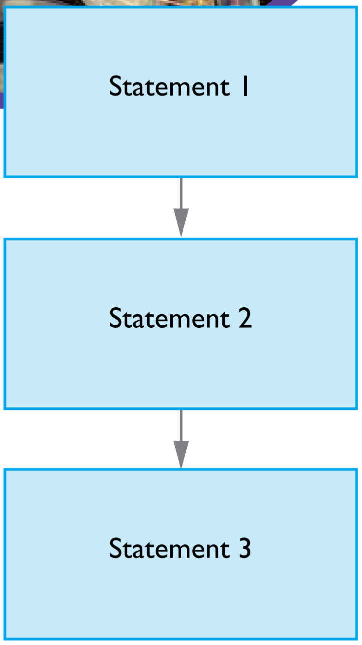


Figure 2.1 *A flowchart*

There are a lot of different design procedures and techniques for building large software projects. The technique discussed in this chapter, however, is for smaller coding projects and is referred to by the term 'top down, structured flowchart methodology'. We will explore how to take a task and represent it using a flowchart. A flowchart puts the sentences from a sequence into shaped boxes. The shapes indicate the action.

You will know from the last chapter that a sequence is where a set of instructions or actions are ordered, meaning that each action follows the previous action.

Flowchart advantages

- Flowcharts are a graphical way of writing an algorithm.
- They are standardised: they all agree on the symbols and their meaning.
- They are very visual.

Flowchart disadvantages

- They are hard to modify and can be time consuming.
- They need special software for symbols, although some software has these built in.

General rules for flowcharts

- All symbols of the flowchart are connected by flow lines (these must have arrows, not lines, to show direction).
- Flow lines enter the top of the symbol and exit out of the bottom, except for the Decision symbol, which can have flow lines exiting from the bottom or the sides.
- Flowcharts are drawn so that flow generally goes from top to bottom of the page.
- The beginning and the end of the flowchart is indicated using the Terminal symbol.

KEY POINTS

- An algorithm is a sequence of steps that can be followed to complete a task.
- A sequence is where a set of instructions or actions are ordered, meaning that each action follows the previous action.
- Flowcharts must have flow lines with arrows to show the direction.

Let's look at a simple sequence. Say we want to add A to B, where A = 200 and B = 400.

```
Start
A = 200 B = 400
Add = 200 + 400
Output = 600
End
```

We could create a simple flowchart, like the one shown in Figure 2.2.

Let's look at another sequence, for example the sequence you carry out each morning in the bathroom could be:

- Brush your teeth.
- Wash your face.
- Comb your hair.

TASK

Produce a sequence to show how to brush your teeth.

QUESTION

What is a Sequence?

As you can see, sequences are a useful tool for showing what happens, and in what logical order each step happens. But each step, for example 'brush teeth', needs to be defined in more detail to be carried out.

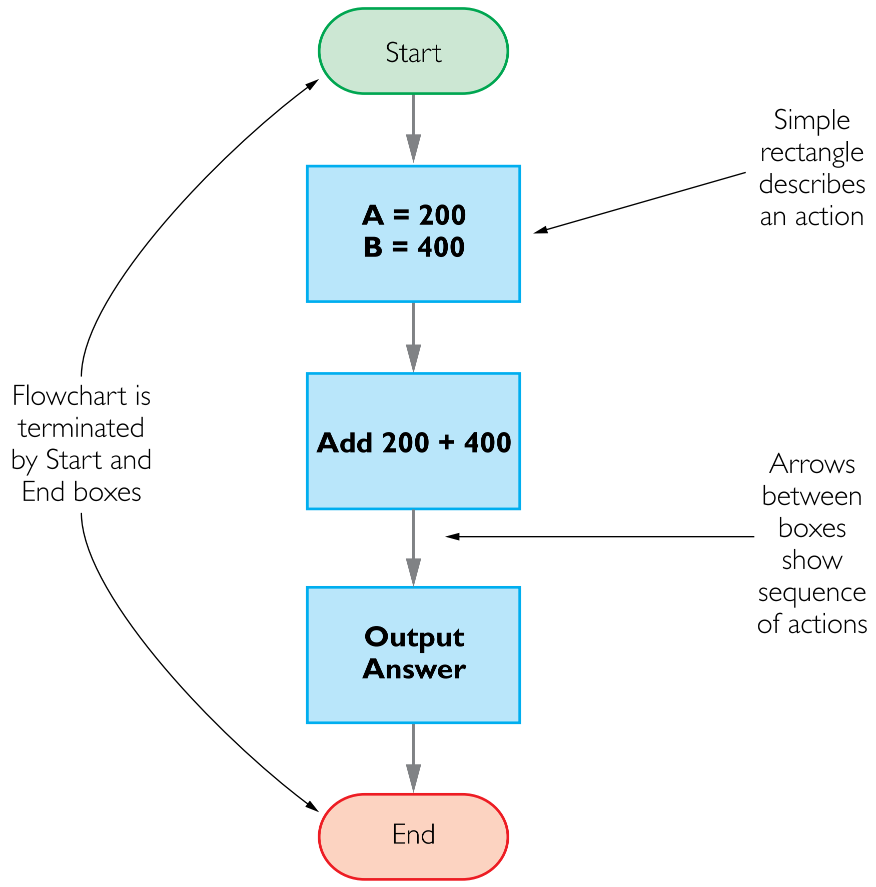

Start
A = 200
B = 400
Add 200 + 400
Output Answer
End
Simple rectangle describes an action
Flowchart is terminated by Start and End boxes
Arrows between boxes show sequence of actions

Figure 2.2 *A flowchart showing a simple sequence*

Once we have picked up our brush, turned on the tap and added the toothpaste we can put the brush in our mouth and brush. The act of actually brushing your teeth could be recorded in a linear way: press, brush up, brush down, brush up, brush down etc. But it would be much simpler to explain the brushing once and then tell the user to repeat the same action x amount of times. We will explore this later when we consider looping, but for now let us explore how we can use a flowchart to represent simple sequences. First we need a few more elements.

QUESTION

What is an input/output?

KEY POINT

Cleaning your teeth is called a procedure in coding. You perform the same action every day, for example, pick up brush, put toothpaste on brush, brush teeth for two minutes, spit out, clean brush. These actions could be given a procedure name: 'Brushing Teeth'.

Basic elements of flowcharts

The flowchart symbols denoting the basic building blocks of programming are shown in Figure 2.3. Text inside a symbol is called a label.

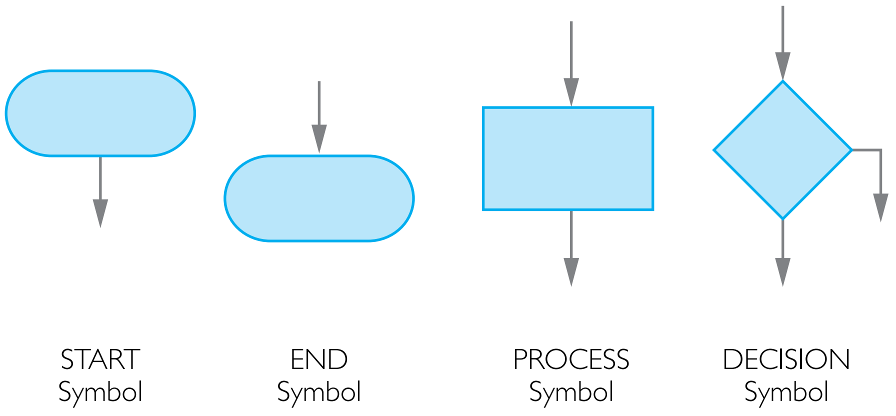


Figure 2.3 *Basics elements of a flowchart*

The START symbol represents the start of a process. The PROCESS symbol is labelled with a brief description of the process carried out by the flowchart. The END symbol represents the end of a process. It contains either END or RETURN depending on its function in the overall process of the flowchart.

KEY POINTS !

- You use a PROCESS symbol for an OPERATION or ACTION STEP.
- You use a TERMINATOR symbol for a START or END in a PROCESS.
- You use a DECISION symbol for a QUESTION or BRANCH of a process.
- Flowchart symbols contain text called labels.

Representing a process

A ‘Process’ symbol is representative of some operation that is carried out on an element of data. It usually contains a brief description of the process being carried out on the data. It is possible that the process could even be further broken down into simpler steps by another complete flowchart representing that process. If this is the case, the flowchart that represents the process will have the same label in the ‘start’ symbol as the description in the process symbol at the higher level. A Process box always has exactly one line going into it and one line going out.

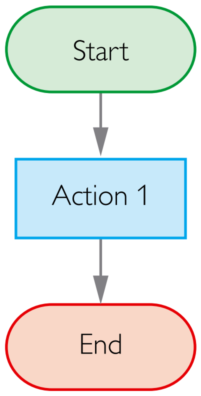


Figure 2.4 *A flowchart showing a process*

KEY POINTS !

- All flowcharts must have a start and an end symbol, unless the process continues forever.
- The ‘Decision’ symbol will have exactly one input and two outputs.

In practice, sequences are not a simple line. Often the next action depends on the last decision. This is called selection. In a selection, one statement within a set of program statements is executed depending on the state of the program at that instance. We ask a question and choose one of two possible actions based upon that decision.

Representing a decision

A 'Decision/selection' symbol always makes a Boolean choice. We will explore Boolean logic in more detail later in the book. But the label in a 'decision' symbol should be a question that clearly has only two possible answers to select from.

The 'decision' symbol will have exactly one line going into it, and two lines coming out of it. The two lines coming out of it will be labelled with the two answers to the question in order to show the direction of the logic flow depending upon the selection made.

Selections are usually expressed as 'decision' key words such as IF, THEN, ELSE, ENDIF, SWITCH or CASE. They are at the heart of all programming.

KEY POINTS

- One of the most confusing things in a flowchart is telling the loops apart from the selections. This is because both use the diamond shape as their control symbol. Mark them clearly.
- A 'Decision' symbol always makes a Boolean choice.

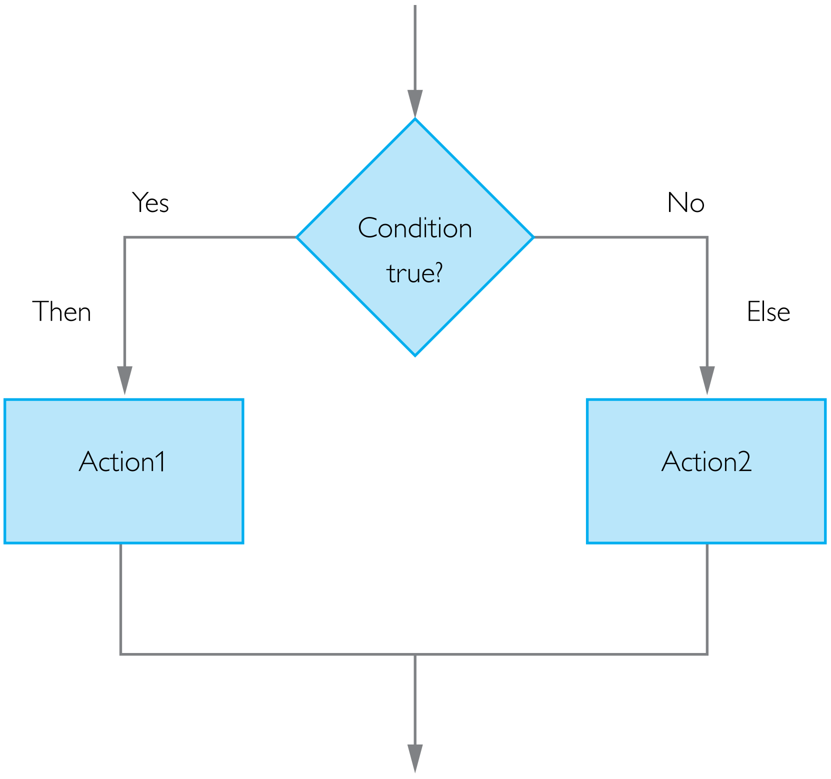
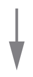
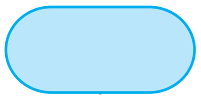
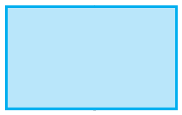
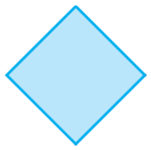


Figure 2.5 *A flowchart representing selection*

QUESTION

What is a selection?

In flowcharts you use the following symbols:

Symbol	Purpose	Use
	Flow line	The lines show the sequence of operations
	Terminal (Start/Stop)	Denotes the start and end of an algorithm
	Processing	Denotes a process to be carried out.
	Decision	Used to represent the operation in which there are two alternatives, true and false.

We can use a decision to create a flowchart of what happens in the morning on school days.

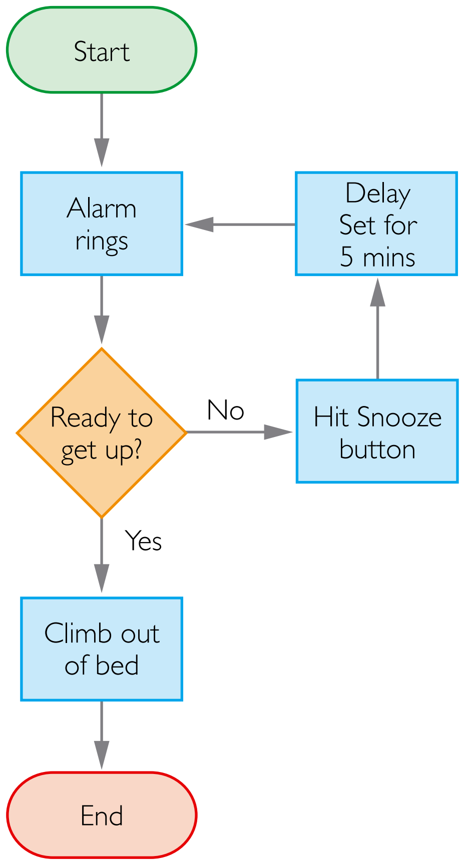


Figure 2.6 *A flowchart for what happens in the morning in a school day.*

We also explored selections a little when we looked at the sequence of making tea. We explored using IF someone wants sugar and IF someone wants milk. The process of making the tea differed according to their answer to these questions.

The flowchart below shows a different process for making tea and adds two decision boxes.

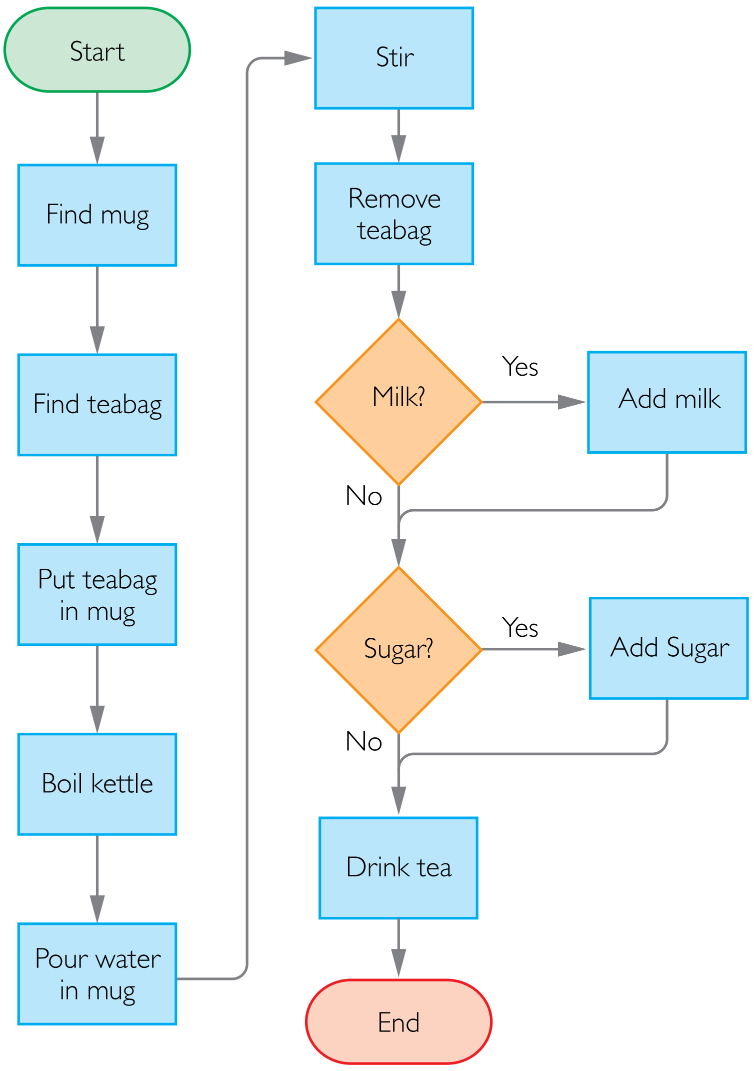


Figure 2.7 *A flowchart for making a cup of tea*

If we wanted to show how to play the game of snakes and ladders we could explain how to play the game in English as follows:

```
Start game
Throw the dice: the number indicated by dice is x.
Move your counter: x squares on the board and check:
  Have you landed on snakes head?: no/yes
    If yes slide down snake to its tail.
    If no check next statement
    Have you landed on the bottom of the ladder?: no/yes
      If yes move up the ladder.
      If no check next statement
    Have you reached the last block of the game?: no/yes
      If yes
        Output "you are the winner"
      If no
        Give the dice to the next player
Repeat until someone reaches the last block of the game.
End
```

We have more decisions in this example and could show the game with the following flowchart.

KEY POINTS

Repetition is used when the same bit of code is needed several times. Instead of writing it over and over again, you can use the repeat command. Repetition can also be called looping.

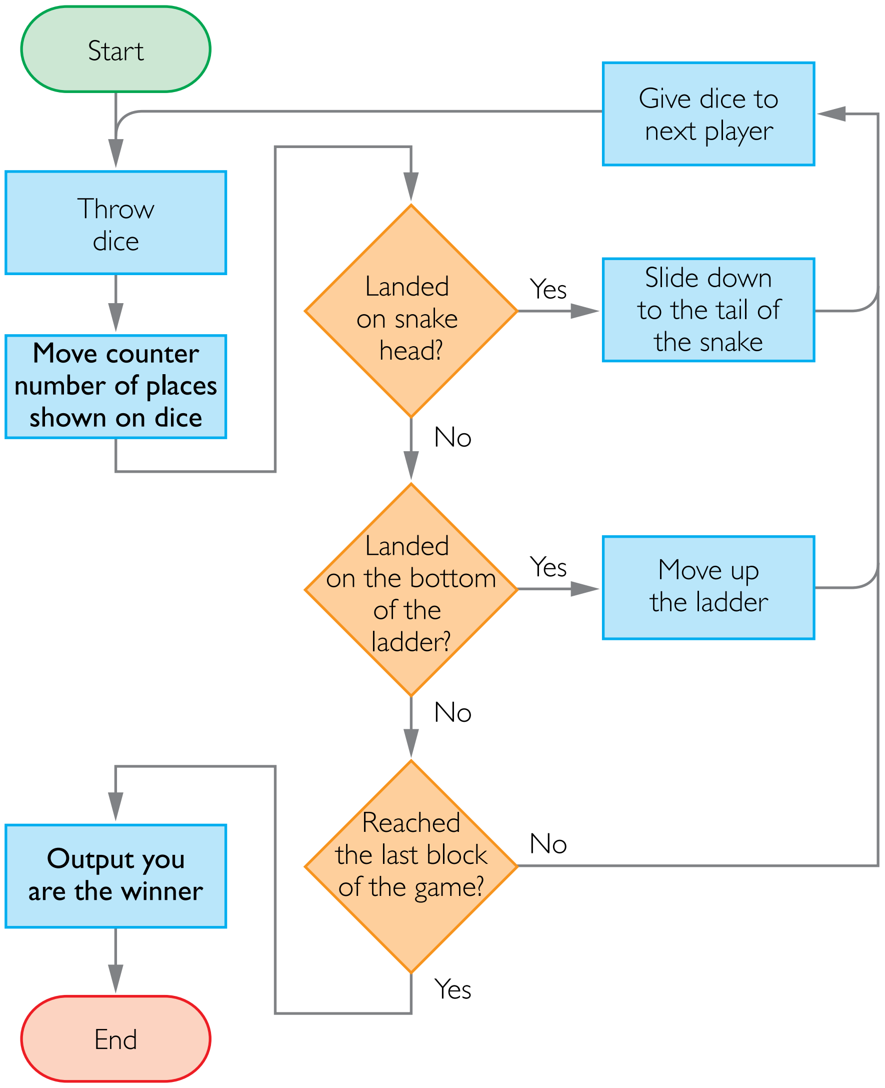

Start
Throw dice
Move counter number of places shown on dice
Landed on snake head?
Yes
Slide down to the tail of the snake
No
Landed on the bottom of the ladder?
Yes
Move up the ladder
No
Reached the last block of the game?
No
Give dice to next player
Yes
Output you are the winner
End

Figure 2.8 *A flowchart for playing snakes and ladders*

Other structures we will use are shown in Figure 2.9.

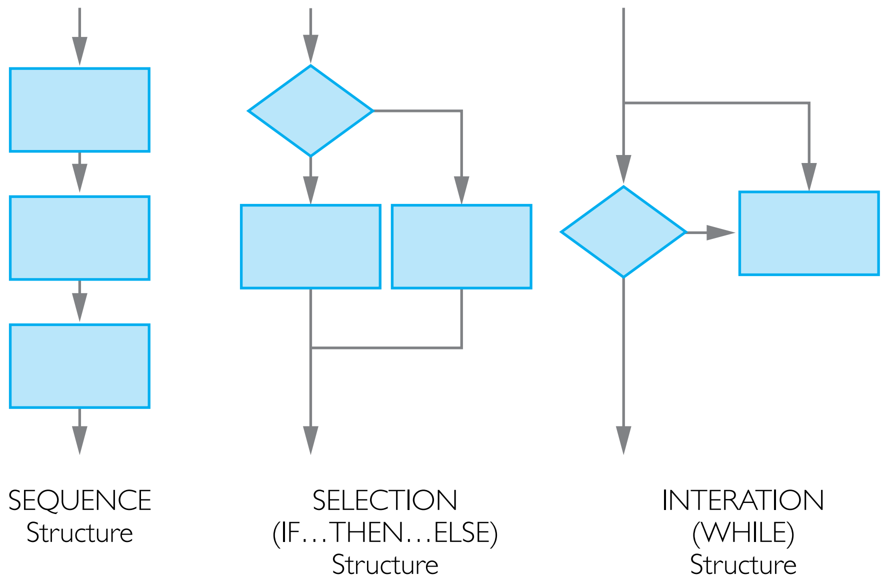


Figure 2.9 *Other structures we will use in this book.*

On-page and off-page 'Connectors' may also appear in some flowcharts. This occurs when a flowchart goes over more than one page. For the purpose of this chapter we will only explore flowcharts that can be represented on a single page. If a flowchart is so big it needs to go on to another page, you should split in to subprocesses.

Subprocesses

We can also use subprocesses in flowcharts using the symbol shown in Figure 2.10.

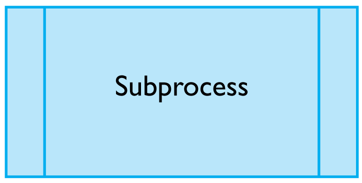


Figure 2.10 *The subprocess symbol*

Subprocesses are useful because:

- they help with the modularisation of complex programs;
- they provide a way of simplifying programs by making common processes available to a wide number of programs;
- they lead to more reliable programs, since once a process is tested and works it can be made into a subprocess and need not be tested again.

In flowcharts subprocesses are also useful in dealing with the flowcharting rule that a flowchart should fit on a single page.

Figure 2.11 shows an example of the main page of a flowchart. It contains two subprocess symbols. Each subprocess contains text which describes briefly what the subprocess does.

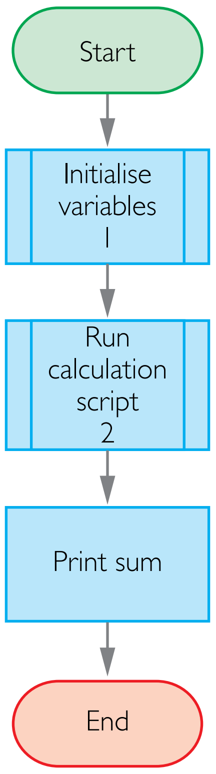


Figure 2.11 *The main page of a flowchart*

Each subprocess symbol also contains a page reference where the flowchart for the subprocess will exist.

CHAPTER REVIEW

In this chapter we built upon the last chapter to explore sequences in more detail and how we can show these using flowcharts.

We looked at the basic elements of flowcharts and introduced the concept of decisions and how these can be represented.

Remember before tackling any computer science task or examination question on this topic you must:

- work out the steps or rules for getting things done;
- use a systematic approach to problem solving and algorithm creation representing those algorithms using flowcharts;
- be able to explain simple algorithms in terms of their inputs, processing and outputs;
- understand the concept of selection and the concept of subprocesses;
- be able to record your ideas using flow diagrams;
- be able to describe the structured approach to programming;
- be able to explain the advantages of the structured approach.

3 Using Pseudo-code

Pseudo-code

Pseudo-code is another way to develop an algorithm. It consists of natural language-like statements that precisely describe the steps required.

Pseudo-code must:

- contain statements which describe actions
- focus on the logic of the algorithm or program
- avoid language-specific elements
- be written at a level so that the desired programming code can be generated almost automatically from each statement
- contain steps. Subordinate numbers and/or indentation are used for dependent statements in selection and repetition structures.

KEY POINTS !

- Pseudo-code is a language designed to express algorithms in an easy-to-follow form.
- Pseudo-code is an easy-to-read language to help with the development of coded solutions.

KEY POINT

When writing in pseudo-code, resist the urge to write in whatever language you are most comfortable with.

QUESTION

What is pseudo-code?

Pseudo-code advantages

- Pseudo-code is similar to everyday English.
- It helps programmers to plan an algorithm.
- It can be done easily on a word processor.
- It is easily modified.
- It implements structured concepts well.

Pseudo-code disadvantages

- Pseudo-code is not visual like flowcharts.
- There is no accepted standard, so it varies widely.
- It is not an actual programming language.
- It is an artificial and informal language.

The importance of syntax

Syntax is the set of rules, principles, and processes that enable us to understand a programming language. The syntax rules of a programming language define the spelling and grammar and, as with natural human languages, each language has its own rules. Computers are very inflexible and understand what you write only if you state what you want in the exact syntax that the computer expects and understands.

Each programming language has its own rules and specialist syntax including the words the computer understands, which combinations of words are meaningful, and what punctuation is necessary to be correctly structured. Whilst pseudo-code does not have a fixed syntax, you will need to understand the syntax used in AQA pseudo-code. Understanding the importance of syntax is also vital when you start using a programming language.

Symbols

When we write code in English we also use symbols in the form of punctuation and special characters. For example, – is a symbol, and so is #. We will explore these later in the book. Symbols are used as they are human-readable, but they are important as they also have an effect in your code.

KEY POINT

The symbols ←, <<, <- are often used to represent the assignment operator in programming languages.

Symbols can also be used as what are called identifiers. In some programming languages, they are also called atoms rather than symbols.

The symbols ←, <<, <- are often used as what are called 'assignment operators'.

Examples

```
Name ← USERINPUT
or
LengthOfJourney ← USERINPUT
or
YesNo ← USERINPUT
```

Common action keywords

Several keywords are often used to indicate common input, output and processing operations.

- Input: READ, OBTAIN, GET, USERINPUT
- Output: PRINT, DISPLAY, SHOW, RETURN, OUTPUT
- Process/compute: COMPUTE, CALCULATE, DETERMINE
- Initialise: SET, INIT
- Add one: INCREMENT, BUMP

KEY POINTS

- In pseudo-code a computer can receive information using READ or OBTAIN (from storage) or USERINPUT or GET (from keyboard).
- In pseudo-code a computer can display information using OUTPUT, DISPLAY or SHOW (on monitor) or PRINT.

In AQA Pseudo-code you use the following syntax to send output to the screen. The red brackets < > are only to show where you add something; you don't need to put them in your code

Syntax

```
OUTPUT <add expression here>
```

Example

```
OUTPUT 'Have a good day.'
```

Whilst there is no common way of writing pseudo-code, in this book we have written the commands in capital letters to differentiate them from the examples in Python and to help you understand what the command words are.

Some of the pseudo-code words used by AQA involving code are:

```
ELSE            INPUT
ENDFOR          OUTPUT
ENDIF           REPEAT
ENDWHILE        RETURN
FOR             THEN
IF              WHILE
```

Commenting on your code

Good code is not only well written but should also be well annotated. There are programmers who argue that comments are not necessary if the code is written well, but remember that you are telling a third party what your code does and why.

You will find many examples of commented code in this book. Comments are shown using either // or #. Different programming languages have different ways to tell the computer that this is a comment *not* the code. You can make all the code you write in pseudo-code a comment when you write the actual code using your chosen language. This is considered good practice when learning to code. You can also comment out bits of code to find errors but we will explore this later.

The following pseudo-code syntax may be used in code for comments:

```
#some text
```

Multiple comments will show the hash # for each separate comment line.

```
#some text
#some more text on a new line
```

KEY POINT

Good code is well written and well annotated.

Comment tags remind you and the examiner why you included certain functions. They also make maintenance easier for you later.

Have you ever tried to work with someone else's complex spreadsheet or database? It's not easy. Imagine how difficult it is if you're looking at someone else's programming code.

When you fully document your code with comment tags, you're answering two questions (at least):

1 Why did I do that?
2 What does this code do?

No matter how simple, concise, and clear your code may end up being, it's impossible for code to be completely self-documenting. Even with very good code it can only tell the viewer how the program works; comments can also say why it works.

KEY POINT

A comment is explanatory text for the human reader.

QUESTION

What is a comment?

TASK

Describe the main reasons why a programmer would wish to annotate or add comments to their code.

Adding selection

As we discovered in the last chapter on flowcharts, another important aspect of programming is selection. If we want to write pseudo-code that tells a user to enter a number to a variable and then we want the code to see if the number they entered is a 3 or a 4 we could write a selection algorithm in pseudo-code that could look like this:

```
inputNumber ← USERINPUT     #Input
IF inputNumber = 3 THEN     #Selection (Process)
  OUTPUT "your number is 3" #Output
ELSE
  IF inputNumber = 4 THEN
    OUTPUT "your number is 4"
  ELSE
    OUTPUT "your number is not 3 or 4"
  ENDIF
ENDIF
```

If we remove the comments, then the code would look like this:

```
inputNumber ← USERINPUT
IF inputNumber = 3 THEN
  OUTPUT “Your number is a 3”
ELSE
  IF inputNumber = 4 THEN
    OUTPUT “Your number is a 4”
  ELSE
    OUTPUT “Your number is not a 3 or a 4”
  ENDIF
ENDIF
```

CHAPTER REVIEW

In this chapter we have explored pseudo-code and how to use it to show program flow and decision making.

We also explored the importance of syntax and how to comment on your code.

Remember before tackling any computer science task or examination question on this topic you must:

- be able to work out the steps or rules for getting things done;
- manage the complexity of the task by focusing on the key details;
- use a systematic approach to problem solving and algorithm creation representing those algorithms using pseudo-code;
- be able to explain simple algorithms in terms of their inputs, processing and outputs;
- be able to determine the purpose of simple algorithms;
- record your ideas using pseudo-code;
- think about the correct syntax needed;
- understand that more than one algorithm can be used to solve the same problem;
- be able to obtain user input from the keyboard;
- be able to output data and information from a program to the computer display.