Chapter 2

Elements of computational thinking

Features that make a problem solvable by computational methods

Example

Here are two closely related problems.

'How can we speed up the throughput to a set of six lifts in a tall building?'

For this, we need to gather data about usage, lift speeds, typical stopping frequencies, strategies for calling lifts, and so on. It should be solvable by fairly standard analytical and algorithmic methods.

But suppose the problem is 'How do we reduce the number of complaints about waiting for lifts in this hotel?'

We could apply the solution to the first problem and hope that satisfies the users. Another approach that has worked is to install mirrors by the lifts. That way, the users have something else to look at when waiting and are less likely to get bored and frustrated.

This is an example of an increasingly common situation where there is a mixture of human reactions and computable problems, showing that humans and computers working together can be a good way to tackle real problems. It also highlights the importance of really understanding what the problem is.

This is an area that has long been studied by computer scientists. In 1936, Alan Turing devised a theoretical computer based on an unlimited memory made from paper tape. Symbols are printed on the tape and at any given moment the machine can manipulate the symbol according to a set of rules. A Turing machine can be used to simulate a computer **algorithm**. One way of deciding if a problem is computable is to test it against the capabilities of a Turing machine.

Computability is whether or not a problem can be solved using an algorithm. It is worth noting that any problem that can be solved by a computer today can also be solved by a Turing machine. Indeed all computers ever made are capable of solving exactly the same set of problems, *given enough time and memory*.

The speed computers run at and the memory that they can access are the limiting factors to the problems we can solve with computers. We increasingly have access to exponentially larger amounts of computing power; we have the internet, data centres, supercomputers, nanocomputers, server farms and more developments are always appearing. This means the range of problems we can practically tackle using computers is increasing.

As we learn more about computers and indeed how to think, solving problems is now a more wide-ranging question than it was. We also have to realise that solving problems is now a joint enterprise between these computing agents and the humans that work with them, so a solvable problem might mean something rather more than just a computable problem.

It can be proved that there are some problems that we will never be able to solve by computer.

Problem recognition

The example given above shows that, given a situation that needs attention, it is important to determine exactly what the problem is: it may not always be what you think.

Some problems are obvious: A traffic queue at a road junction is clearly a problem – it wastes time and causes stress. By using computational and intuitive methods, it may be possible to come up with a solution, if only a partial one.

Questions

Given a regular traffic hold-up spot at a junction:

1. What data would you need to acquire?
2. What processes to solve the problem might you consider?
3. To what extent do you think the problem is intractable?

Backtracking

Backtracking is an algorithmic approach to a problem where partial solutions to a large problem are incrementally built up as a pathway to follow, and then, if the pathway fails at some point, the partial solutions are abandoned and the search begins again at the last potentially successful point. This is a well-known strategy for solving logic problems and is nicely demonstrated by looking at a set of rules in the programming language Prolog.

Question

Your mobile phone is normally fine. It doesn't work today. Explain how you could use backtracking to find what the problem is.

Example

Prolog is a logic-declarative language where rules and relationships are constructed, and from these logical inferences can be made.

Here is a set of rules:

```
give_pay_rise(X):-
works_hard(X),
is_relative(X).

works_hard(alberich).
works_hard(wotan).
works_hard(siegfried).

is_relative(tristan).
is_relative(isolde).
is_relative(siegfried).
```

This set of facts shows us that Alberich works hard and so do Wotan and Siegfried. It also tells us who is a relative.

If we now pose the query:

```
?- give_pay_rise(Who).
```

this asks Prolog to bind to the variable (Who) anyone who fits the rules for give_pay_rise.

Prolog first looks at Alberich. He works hard, but he isn't a relative. So Prolog backtracks and tries again with Wotan. That fails too. Prolog backtracks again and this time, when trying to match all the rules with Siegfried, it succeeds and will output Siegfried.

Key points

- Problem solving can be a disciplined process.
- Some problems are not solvable.
- Some problems are best solved by humans, some by computers and some by a partnership of both.
- Backtracking can be an effective way of solving sequential problems.

Data mining

Data mining is a process for trawling through lots of data that probably comes from many sources. It is a useful way to search for relationships and facts that are probably not immediately obvious to a casual observer. It is also used when the data comes from data sets that are not structured in the same way. So, for example, a supermarket may have data from its loyalty card scheme that shows a few personal details plus purchases made. This is a huge collection of data for a typical large supermarket.

If you perform searches that attempt to find patterns, some of the best algorithms will show whether certain products tend to be bought together, or by the same customer, or by the same demographic group. If you include weather data into the mining operation, you might get correlations showing up between hot weather and ice cream sales, which would be expected, but maybe not what one supermarket found out: that when hurricanes are forecast, people buy more fruit tarts.

Algorithms that help with data mining are known by such terms as 'pattern matching' and 'anomaly detection'. Data mining has become possible because of:

- big databases
- fast processing.

Data mining is useful for many purposes, such as business modelling and planning, as well as disease prediction. Certain groups can be shown to be prone to certain diseases and data mining can sometimes show links with lifestyle factors. This is an aspect of computability that would not have been foreseen in 1936.

Performance modelling

We often want to know how well a system will perform in real life before we have implemented it. It is not feasible to test all possibilities for reasons such as:

- safety
- time
- expense.

You would not test every single configuration of a car body for crash resistance by crashing a real prototype. You would not try re-routing trains on the London Underground by experimenting in the rush hour. You wouldn't try out a new computer system on live exam data in the middle of the exam season.

In all these cases, the sensible thing to do is to build models or simulations in order to best predict the outcomes. Producing computer models is one of the most important uses of computers and is a part of **computational thinking**.

Key point

Why not consider creating a computer model as your programming coursework?

Performance modelling is only as useful as the accuracy of the model and the data that will be fed into it. Various mathematical considerations will form part of a suitable model such as:

- **statistics:** if there is existing relevant data, then it should be taken into account in the model
- **randomisation:** many real-life situations are improperly understood so a random function is often the best we can do to model uncertainty.

Pipelining

Key term

Instruction set The collection of opcodes a processor is able to decode and execute.

Pipelining in computing is a situation where the output of one process is the input to another.

It is useful in RISC (reduced **instruction set**) processors where the stages of the fetch–decode–execute cycle can be separated and thus instructions can be queued up, thereby speeding up the overall process of running a program. While one instruction is being executed, another is being decoded and yet another is being fetched. This is further explained in Chapter 10. It has drawbacks though because if an instruction causes a jump, then the queued instructions will not be the correct ones and the pipelining has to be reset.

The Unix® pipe is a system that connects processes to the outside world (printers, keyboards and the like) by standard input and output streams, thereby relieving the programmer of having to write code to connect to a physical device. This is yet another useful application of abstraction – a virtual concept substitutes for a physical one.

In the Unix command line, you can use a pipe to pass the output of one program to another.

For example the ls (list) command sends a list of the contents of the current working directory to the default output device, usually the console.

Here is some example output from an ls command:

```
sean@zoostorm-ubuntu:~$ ls
Desktop                                          list
Documents                                        list.
Downloads                                        Music
Dropbox                                          pics backup
examples.desktop                                 Pictures
kompozer_0.8~b3.dfsg.1-0.1ubuntu2_amd64.deb      Public
kompozer-data_0.8~b3.dfsg.1-0.1ubuntu2_all.deb   Templates
kompozer-data_0.8~b3.dfsg.1-0.1ubuntu2_all.deb.1 Videos
```

Figure 2.1 Output from an ls command

Here is the output from ls | head -3. The ls output is piped to the 'head' program with the parameter 3. In other words, output the first three items.

```
sean@zoostorm-ubuntu:~$ ls | head -3
Desktop
Documents
Downloads
```

Figure 2.2 Output from ls | head -3

Just the first three items have been output by the head program.

Question

Itemise some of the inputs, outputs and processes involved in building a house.

Pipelining is a useful technique to use in everyday problems too. Notice that some jobs may be done in parallel if you have the resources (people or processors) to do that. Consider any production line or job, such as making an iced cake:

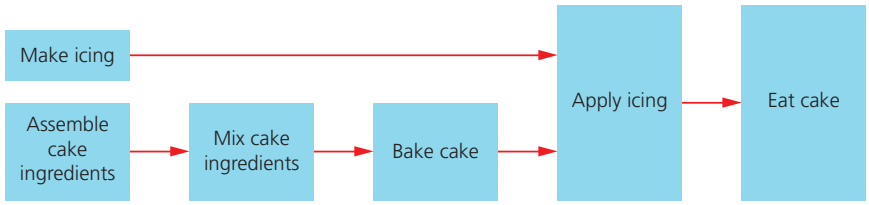

Figure 2.3 Pipeline model

Visualisation to solve problems

Visualisation is a common computing technique to present data in an easy-to-grasp form. At its simplest, it is a matter of presenting tabular data as a graph. More complex visualisations are possible using computer processes, which allow a more sophisticated view of a complex situation. Visualisations can make facts and trends apparent that were never noticed before.

Here is a visualisation of Oyster card use on the London Underground. An Oyster card is a payment card that registers a person's journey by them touching it against a reader when entering or leaving a station. On a map of London on a typical morning, the red circles show where people 'touch in' – in other words, where they board a train – and the green circles show 'touching out' – in other words, where people leave a station. The diameters of the circles show numbers involved.

Figure 2.4 Visualisation of Oyster card use on the London Underground

Here is a visualisation of some text from this chapter using software available on wordle.net:

Figure 2.5 A visualisation of the text in this chapter from wordle.net

This example is useful in showing visually the frequency of use of the words in the text. It can help to improve your writing style!

Key points

- Data mining can show patterns and relationships that are not immediately obvious. Computer systems have enabled huge data stores to be examined for pattern matching.
- Pipelining is a common computing technique that can be applied to everyday problems.
- Sometimes complex situations can be best explained by visualisations.
- Computer systems have enabled the production of strikingly effective visualisations.

Questions

1. Suggest ways to use computing techniques to visualise data about:
   (a) the age of people living in different parts of a city
   (b) the means of transport used to get from the suburbs into a city centre.
2. In each case, suggest what data would be needed, how it could be collected and whether there is existing software to do the visualisation.

Thinking abstractly

Example

Fred has lost his mobile phone. It is a Samsung Galaxy, running the latest version of the Android operating system. It is normally in a white case and has a police siren ring tone. Fred last saw it (he thinks) on the window ledge in the bathroom. He can't remember if it is charged up or even switched on. But possibly, he left it in the taxi after coming home last night. It cost a lot of money and has sentimental value because his girlfriend bought it as a birthday present.

An abstraction is a concept of reality. It commonly makes use of symbols to represent components of a problem so that the human mind or a computing agent can process the problem. Abstraction is also about teasing out what does and what does not matter in a scenario.

Questions

Read the example scenario to the left.

1. Itemise information from this description that would be of use in finding the missing phone.
2. Suggest a strategy for finding the phone.
3. Suggest a sequence of steps that would be helpful in finding the phone.

Most problems that we face in everyday life are like in the example. They are messy. All sorts of things may possibly be important in solving a problem but probably are not.

Abstraction helps us maximise our chances of solving a problem by letting us separate out the component parts and decide which are worth investigating. But don't forget, in real life, sometimes information that looks irrelevant can trigger an 'aha!' moment, which is unlikely to be the case in any current computer system.

Abstraction and real-world issues

Abstraction is extremely important in computing, to an extent that using computers to solve real-world problems would be impossible without it.

Every program worth thinking about uses variables. Variables are an abstraction. They represent real-world values or intermediate values in a calculation.

At a higher level, objects are a clear abstraction of real-world things as well as being used to represent other abstractions. We all know what a chair is. It is a real-world object that has a surface to sit on and usually four legs. It is a concept. A real chair will normally comply with these abstractions and can be regarded as one instance of the class 'chair'.

Levels of abstraction

Computer systems make considerable use of another abstraction idea – levels of abstraction. In a complex system, it is often useful to construct an abstraction to represent a large problem and to create lower-level abstractions to deal with component parts.

The power of this approach is that the details in each layer of abstraction can be hidden from the others. This frees up the solution process to concentrate on just one issue at a time, or maybe send the different sub-problems to different staff or different companies to work on.

This idea of levels of abstraction is easily seen in the idea of layering. Layering is found widely, such as in the construction of operating systems, database systems (see Chapter 15), networks (see Chapter 16) and indeed any large system.

Layers are a way of dividing the functionality of a big system into separate areas of interest; for example an operating system will not normally contain code for communicating with any number of peripherals – it will devolve that responsibility to drivers, retaining to itself only the necessary interfaces that connect to the drivers.

The same principle applies to a physical item such as a car. A car designer might be interested in the combustion properties of a new fuel, but that issue is treated separately from the design of the dashboard. Real progress can sometimes be made when creativity is applied across layers, but this is the exception rather than the rule. Specialisation leads to reliability and cost benefits.

Questions

1. Explain how a map is an example of an abstraction.
2. Identify examples of levels of abstraction on a map of your choice.
3. Explain how levels of abstraction assist the map-maker.
4. Explain how levels of abstraction assist the map user.

Thinking ahead

Thinking ahead has always been standard good advice for all sorts of aspects of life. The better you anticipate what needs to be done in any situation, the easier it is to do the job when it happens.

For example, if you plan to decorate your house, you don't get on a ladder and get to work, you first determine how much paint you need, what colour you want, what type of paint you want for a given location, what you need to do to prepare the surface, and then you need to calculate how much paint you need to buy. Once you have all the data you need, you can go to the DIY superstore and buy all the things you need. If you get this wrong, you may find yourself making multiple extra trips only to discover that your colour has now sold out.

Of course, the same disciplines apply to producing computer solutions, but analysts have long formalised how best to do this. Awareness of how the professionals plan ahead can help us with everyday problems.

Inputs and outputs

Key point

Follow the advice to the right when planning your programming project.

When planning a computer system, one of the first things an analyst needs to do is to determine what outputs are needed. After all, that is why we have computer systems: to produce outputs.

Suppose an online vendor wants to produce a picking list for customers. This is the list that is sent to a warehouse where the staff use it to collect the items that the customers want when fulfilling the order. The list might look like this:

Picking List			
Order Number	2001	**Date**	25/01/15
Ordered by	3846		
Item Code	**Item Quantity**	**Location**	**Quantity**
564	10	Shelf A1.1	
755	15	Shelf B3.2	

To get an output like this, the designer of the system needs to ensure that at some stage there are inputs for all the data items on the list. Of course this is part of a larger system, but a similar design process needs to be used.

Caching

Caching is a good example of how 'thinking ahead' can be related to computing processes. In caching, data that is input might be stored in RAM 'in case' it is needed again before the process is shut down. If it is required, it does not need to be read in again from disk, thereby giving a faster response time.

Prefetching is another related computer operation, where an instruction is requested from memory by the CPU before it is required, to speed up instruction throughput. There are algorithms that can predict likely future instructions needed so that they are ready in the cache as soon as they are in fact needed.

In real life, this can be compared with getting your Oyster card (used for payment on public transport) out when you arrive in London and having it in your pocket ready to use instead of having to fish it out of your wallet each time you take a bus or tube.

Question

Explain in detail how prefetching is useful when:

(a) baking a cake

(b) cleaning a car.

Caching brings various other advantages to a computer system, such as reducing the load on a web server because data required by an application can be anticipated, thereby reducing the number of separate access actions.

Caching isn't all good news. It can be very complicated to implement effectively. Also, if the wrong data is cached, then it can be difficult to re-establish the correct sequence of data items or instructions.

Preconditions and reusability

We have already seen that by dividing up a planned system into various component parts, it makes it a lot easier to devise solutions. An added advantage is that separate program modules of any other items such as data stores can be reused in future projects.

One good example of reusing modules in action is the Windows® DLL libraries. A DLL is a Dynamic Link Library. This is a package of program code that can be called at runtime to provide certain functionality to a program. Particularly useful modules are accessed again and again by many programs, for example if you write Windows-based programs, you do not need to write code to make a dialogue box. A DLL can be linked to your code to produce a familiar and standard dialogue box format.

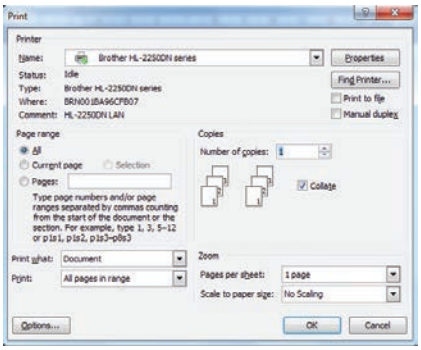

Figure 2.6 Dialogue box

Note that some DLLs are provided with Windows but you can easily write your own if you think that you might need to reuse code. Adding new ones can lead to various difficult problems, as you can see in the section on DLL Hell in Chapter 8.

Code libraries are widespread. Many programming languages have extra collections of commands for use in certain situations. We have already seen how Python has a Logo library and indeed it has many others. They all are examples of reusing code modules, such as the incorporation of the Logo library as mentioned on page 12.

Python uses the command 'import' to bring in these libraries. C and C++ have the preprocessor directive '#include' to bring in 'header files', for example #include <stdio.h> inserts the header file stdio.h into the code being written. This header file is necessary to provide standard input and output functions.

Thinking procedurally

When producing a complete computer system or a single program, we have seen how useful it is to decompose the problem. This makes its solution more manageable. Once a problem has been decomposed, it usually lends itself to the production of program modules that correspond with each sub-problem.

Question

Outline some problems and sub-problems that would form a plan for producing a multi-player online game.

For example, an online ordering system will have sub-problems and hence program modules that deal with customer records, order processing, invoice production, bank account access and stock control at the least. Trying to create a single system to deal with all these separate issues would be highly unlikely to succeed. Also, it is likely that modules to do these jobs already exist and can be customised to fit in with the scenario.

Order order

When planning solutions to a problem, the order may or may not be important. In the case of event-driven solutions, the order of events may

Questions

In each of these scenarios, is the order of solution important? For each case, list some of the main sub-problems in a sensible order. Are there any steps where the order does not matter?

1. Building a house.
2. Buying a train ticket online.
3. Buying a drink in a coffee shop.

be unpredictable. You cannot anticipate whether a customer on your website will browse books, kitchen equipment or anything else in some predetermined order. Also, the placing of orders can be unpredictable. Therefore, the modules dealing with display, searching and purchase need to be accessible in any order.

However, a system that processes exam results cannot produce grades until the marks are recorded. It cannot produce certificates until after that. Order can be important. Establishing whether it is important and if so what the order should be is something that is part of computational thinking and can usefully be applied to real life as well.

Thinking logically

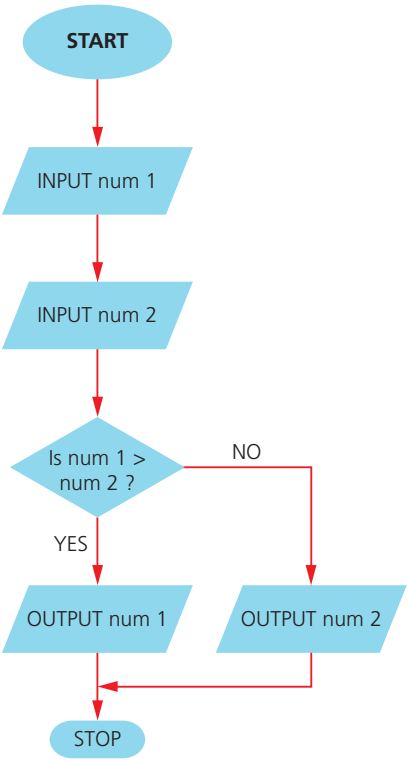

Figure 2.7 Decision flowchart

We have seen (page 7) that in any non-trivial program, there will be points at which decisions need to be made. These will either lead to a branching point (if..then) or a repetition in a loop (for example repeat.. until or do..while).

We have seen that these decisions are based on Boolean expressions. For example in this shell script, an output is produced that depends on the Boolean expression "$character" = "1".

```
echo –n "Enter a number between 1 and 3 inclusive > "
read character
if [ "$character" = "1" ]; then
     echo "You entered one."
```

When planning a program, identifying the decision points is a crucial part of the program design. We can plan these using pseudocode, structured statements or flowcharts; for example the flowchart to the left indicates where a decision will be made about outputting the larger of two different numbers.

The Boolean expression that controls this is 'num1>num2', which of course is either true or false.

A similar process using flowcharts has long been used to plan human activity, for example a disaster recovery plan could be based on the following decision-making process:

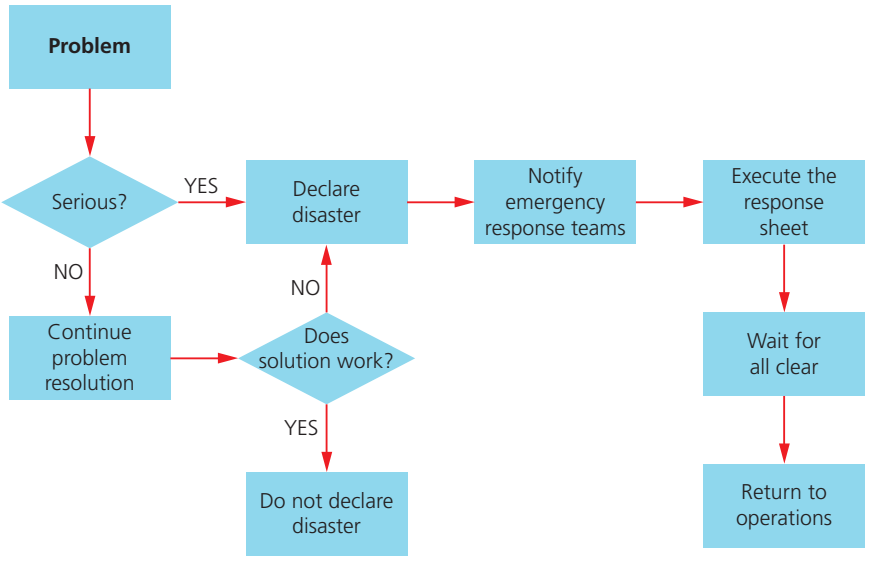

Figure 2.8 Disaster recovery plan

Thinking concurrently

A Level only

Often, as we have seen, it is possible for different parts of a problem to be tackled at the same time. This is beneficial because it saves time, although it might mean that mistakes are fed into later stages of a project.

Parallel processors enable different parts of a program to be executed simultaneously. Multi-core processors are now common, which have more than one processor mounted on a chip. There are potentially great advantages to having multiple processors. Not only are programs executed faster, but savings are also made on energy and computers can run cooler.

Programs have to be written specially to take advantage of parallel processing and this can make them longer and more complex. Also, the savings in a given program may not be that great if a substantial part of the program must be executed in sequence.

Planning human activities can also benefit from parallel processing. Projects such as building a house or creating a computer system can be planned out using a variety of tools to achieve the greatest efficiency.

Gantt charts are commonly used to plan who does what and make other plans for a project and the bars are used as a visual representation of when tasks occur. Tasks planned to be concurrent are easily shown.

Figure 2.9 A Gantt chart

Key point

There are standard computing practices that have stood the test of time and can be applied to many non-computing scenarios.

Practice questions

1. Devise a visual representation of how your computing project could be planned over a designated time period.
2. You have lost your wallet on the way to school or college. Explain how backtracking can help you find it.
3. Draw a flowchart to show how an email address could be validated as being in the correct format.
4. (a) Explain what pipelining is.
   (b) Show how pipelining can be used to improve the efficiency of a self-service cafeteria.