6. Play it Again Sam (Iteration and Recursion)

"Oh No Not Again"
The Bowl of Petunias (Douglas Adams),
The Hitch Hiker's Guide to the Galaxy (1979).

General Repetition
Repetition (often called **Iteration**) is important to Computer Scientists because one of the things that computers are very good at is doing the same thing over and over and over again. They increasingly are taking over all the mundane, boring, repetitive tasks that require little intelligence – just an ability to follow instructions blindly. Examples include spraying cars with paint and ensuring we all get paid at the end of the month (or start of the term for students). For computers to just follow instructions blindly, we need a way of saying to do something repeatedly. That is what a **loop** is for.

Let us look at a simple task to examine what makes a loop. At School as a punishment you might have to write lines: writing the same thing over and over again. "Write out 100 times, 'I must not throw chewing gum at the teacher' ", or "Write out 30 times, 'I must not break up my desk and pass the bits out of the window behind the teacher's back' ", for example). The teacher will have told you the sentence you have to write out repeatedly. They will also have told you when you can stop – probably by telling you the number of times you had to write it (perhaps 100 times – or maybe 500 times if your aim with the chewing gum had been accurate). There are thus three basic things we must specify if we are to give an instruction that something must be done over and over again:

- we must make it clear that we do want something to be repeated and not just done once,
- we must clearly say what it is that must be repeated (this is called the **body** of the loop), and
- we must indicate in what circumstances the repetition is to continue and when to stop (this is called the **termination condition** of the loop).

Without knowing the termination condition the body would be repeated forever. Any repetitive behaviour can be written just with those two pieces of information. In computer programming, one of the basic kinds of loop contains just this information and is known as a **while loop**.

Suppose I wish to read all the books on my bookshelf. This is a situation where I am doing a similar thing over and over again. It should therefore be possible to describe it by giving the instructions to be repeated and the instructions on when to stop. The repetitive task (the **body**) is:

1. Pick a book that you have not read.
2. Read the book.
3. Put the book back.

How do we describe the circumstances when I should continue and when I should stop? We most commonly phrase this question the other way round in computing: we describe the situations in which we should keep going. For example, here the **termination condition** is:

Continue provided there are books on the shelf you have not read

Putting these together with an indication that we want something to be repeated:

While there are books on the shelf you have not read **do the following repeatedly**
1. Pick a book that you have not read.
2. Read the book.
3. Put the book back.

Suppose the task is to search for something. For example, suppose I wish to find an article I remember reading about “Sabre-tooth Wombats”. I think the article was in the magazine *Dinosaurs Today*, but am not sure. If it was, then it will be in the pile of *Dinosaurs Today* in my loft (I have every one). The task to be repeated is to
1. Take the next *Dinosaurs Today* from the Pile
2. Check whether the article is in it
3. Add it to a discard pile

What is the condition for continuing to search (i.e. do the repetitive task)?
I am not at the bottom of the pile **and** I have not found it yet
When either of the above are not true I would stop. Notice that this is just one of our true/false boolean questions again. Putting them together:

While I am not at the bottom of the pile **and** I have not found it yet **do the following repeatedly**
1. Take the next *Dinosaurs Today* from the Pile
2. Check whether the article is in it
3. Add it to a discard pile

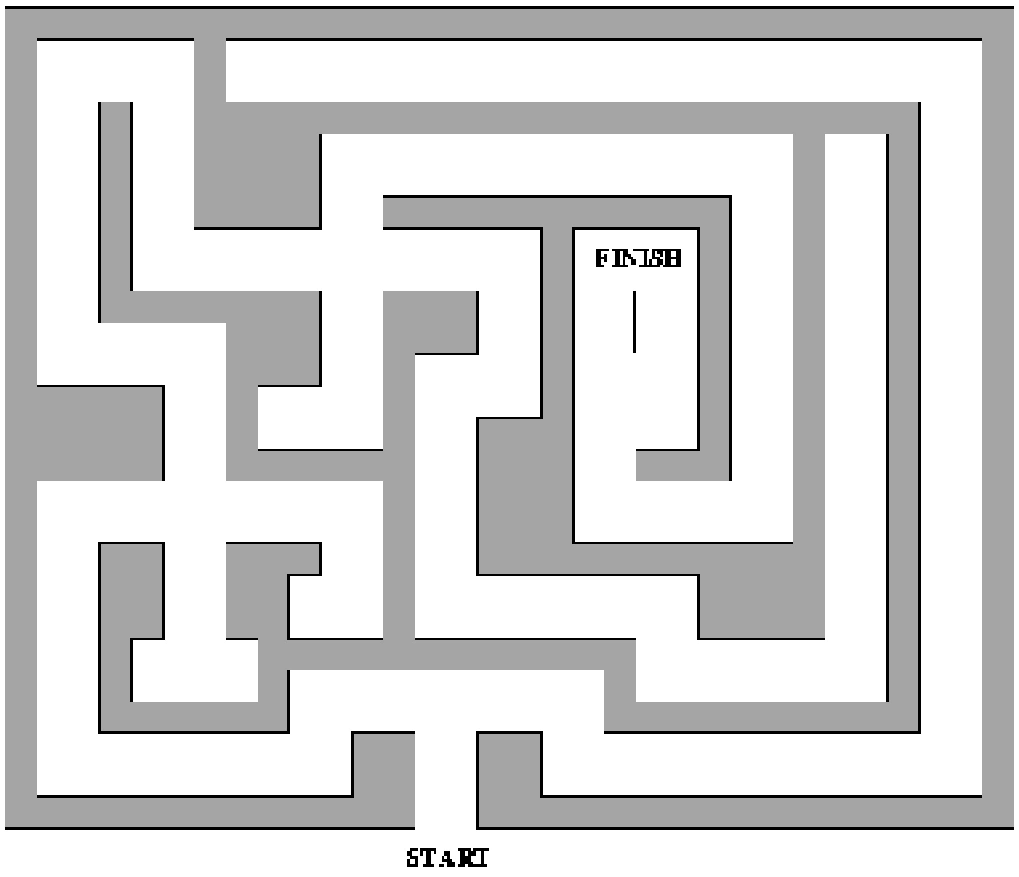
The **termination condition** that tells us when to continue and when to stop can thus be thought of as a question meaning “Do I continue”. In the first example above, the question is
“Are there still books on the shelf I have not read?”
If the answer to this question is YES then I must continue. If the answer is NO then I stop. Similarly in the second situation we are asking the question
“Am I not yet at the bottom of the pile **and** have not yet found it?”
We ask this question once every time we have done the repetitive task to see if we need to do it again.

Remember the maze puzzle we looked at when discussing doing things one after another (sequencing). We came up with the following algorithm to solve it:
To get to the centre of the maze:
1. Enter the maze by the gate following the path ahead.
2. Turn left following the path ahead.
3. Go straight across the junction following the path ahead.
4. Turn left following the path ahead.
5. Go straight across the junction following the path ahead.
6. Stop at the cross.

We can actually simplify this algorithm using a loop. It contains some repetition.
Problem
Which instructions are being repeated in this algorithm?

Why write the same instructions out several times if we can avoid it? This maze is small so the amount of repetition in the instructions is not great – but imagine if the maze was much bigger with more junctions, then if there was repetition it would be very worthwhile avoiding repeatedly writing the same instructions. Also if we can make the instructions simpler they will be easier to learn (remember you were learning them for a race). See if you can rewrite the above algorithm using a while loop. We will go through the answer to the above maze later, but if you find it difficult read on.

Before we look at the above maze together, lets look at a simpler version that is easier to see as a loop.



Solve this maze by drawing a line, then try and write an algorithm just like for the last one, before reading on.

Here is my solution (without a loop).

1. Enter the maze by the gate following the path ahead.
2. Turn Left following the path ahead.
3. Turn Left following the path ahead.
4. Turn Left following the path ahead.
5. Stop at the cross.

If you were trying to memorise this I would hope you would notice the pattern and memorise a simpler version once you have entered the maze: “At every junction I must turn left and then follow the path ahead”. Can we write it as an algorithm in this way without writing out “Turn Left following the path ahead” over and over again? We must ask ourselves two questions to devise the loop. The first question is “How do we know when to continue?” We continue as long as we are *not at the cross*.

The second question we must ask is: “what in the above is being repeated?” It is the instruction:

1. Turn Left following the path ahead.

We can write the algorithm as:

```
1. Enter the maze by the gate following the path ahead.
2. While not at the cross do the following repeatedly
        Turn left following the path ahead.
```

This says that what we should do is first enter the maze and follow the path ahead. We then must follow the loop instruction. If we follow this we expect to repeatedly turn left and follow the path ahead. However the first thing it tells us to do is to check if we are at the cross because if so we have finished. If not then we turn left and follow the path ahead. We have not finished there however. Unlike sequencing, with a loop we do not just follow instructions down the page, stopping when we get to the last one. With a loop, we want to repeatedly do the instruction that is inside the loop “Turn left following the path ahead” in this case. On getting to the bottom of the loop we go back to the top and ask the question again. Perhaps we are at the cross in which case we could stop. If not we will need to continue and follow the instructions in the loop again. We therefore check if we are at the cross to decide whether we have finished – we go back to the top of the loop. Only if we are at the cross do we carry on with the rest of the algorithm.

This algorithm is slightly more general than the original. It wont get us to the centre of any maze, but it would get us to the centre of any maze where you must turn left at any junction, irrespective of the number of junctions.

Let us now return to the original maze. In it you sometimes had to turn left and sometimes go straight on. Here it is again.

To get to the centre of the maze:

1. Enter the maze by the gate following the path ahead.
2. Turn left following the path ahead.
3. Go straight across the junction following the path ahead.
4. Turn left following the path ahead.
5. Go straight across the junction following the path ahead.
6. Stop at the cross.

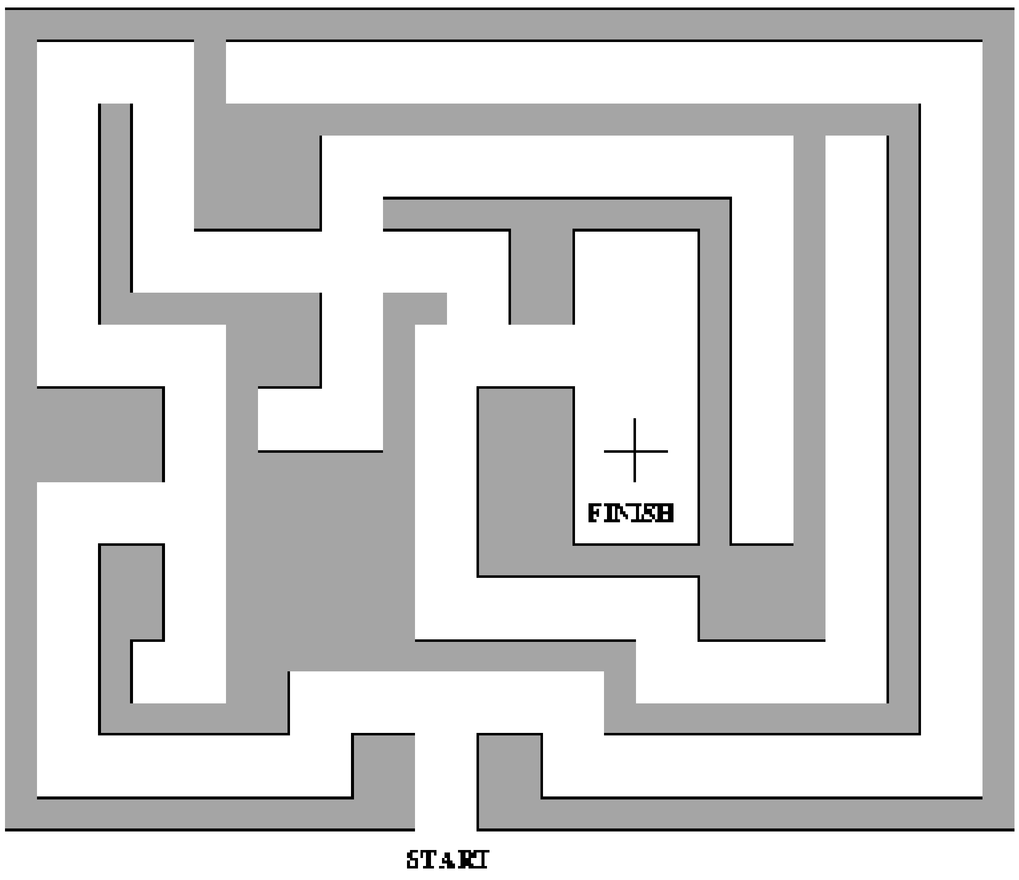
What is the thing that we want to do repeatedly. This time it is two instructions: turning left then going straight on at the next junction. We stop, as before, if we are at the cross. Using a loop, the algorithm is:

To get to the centre of the maze:

```
1. Enter the maze by the gate following the path ahead.
2. While not at the cross do the following repeatedly
```

1. Turn left following the path ahead.
2. Go straight across the junction following the path ahead.

What we have here is a loop where its body is not just a single instruction but is a sequence of instructions. We can put any combination of control structures inside a loop. For example, we could put an if statement inside a loop. Here is a maze where that might be useful. Write an algorithm that solves it.



Writing out a solution to this in full we get:

To get to the centre of the maze:

1. Enter the maze by the gate following the path ahead.
2. Turn left following the path ahead.
3. Turn left following the path ahead.
4. Go straight across the junction following the path ahead.
5. Turn left following the path ahead.
6. Stop at the cross.

The secret of this maze is that whenever you get to a T-junction you turn left and whenever you get to a crossroads you go straight on. In other words, if you are at a T-junction turn left else go straight on. We can use this observation to rewrite the instructions so that there is repetition. The above algorithm can be replaced by the slightly more general one below.

To get to the centre of the maze:

```
1. Enter the maze by the gate following the path ahead.
2. If at a T-junction
   then turn left following the path ahead
```

```
        else go straight across the junction following the path ahead.
    3.  If at a T-junction
        then turn left following the path ahead
        else go straight across the junction following the path ahead.
    4.  If at a T-junction
        then turn left following the path ahead
        else go straight across the junction following the path ahead.
    5.  If at a T-junction
        then turn left following the path ahead
        else go straight across the junction following the path ahead.
    6.  Stop at the cross.
```

Convince yourself that this still works by using it to get to the centre of the maze. Now it is obvious what the repetition is. We can put the if statement in a loop to avoid repeatedly writing it out.

To get to the centre of the maze:

```
    1.  Enter the maze by the gate following the path ahead.
    2.  while not at the cross do the following repeatedly
            if at a T-junction
            then turn left following the path ahead
            else go straight across the junction following the path ahead.
```

Again convince yourself that this still works – you check if you are the cross and if not check if you are at a T-junction taken the specified action depending on the answer. Then you do it all again, stopping if you are at the centre of the maze.

Problem
Does this algorithm work for all three mazes that we have looked at?

We described sequencing and selection in terms of relay races with a baton being passed. What sort of relay race is iteration? It is a race round a circular track where at the start a termination question is asked. If the answer to the question is false then the race (the execution of the loop) ends. If the answer is true the runners must do another lap. When the baton gets back to the start line again the question is asked again. The runners keep doing laps until the answer to the question tells them they can stop. Our earlier relay races were always the same fixed length. A race containing a loop could keep going for any length – it all depends on the question. Notice the question must be something whose answer could change as the race progresses. If it always has the same "true" answer, the runners would keep going for ever (or at least until they collapsed – the equivalent of the computer crashing as it runs out of resources to continue).

The Imps would deal with the loop in a very similar way to branch statements. The instruction Imp responsible for the question either passes the baton to the instruction Imp responsible for the instruction immediately after the whole while loop, or to the Imp whose instruction is the first of those to be repeated. The Instruction Imp responsible for the last instruction to be repeated always passes the baton straight back to the Imp responsible for the question. It is only that Imp that has the power to terminate the repetition.

Here is an algorithm for treating heatstroke that saves lives, adapted from BMA (1990). Can you identify the loop in it and say what the repeated action is and what is the termination condition?

1. Move the victim to a cool place.
2. Remove clothing.
3. Place the victim in the sitting position.
4. Support the head and shoulders using pillows.
5. Cover the victim with a wet sheet.
6. Fan the victim with a magazine until the temperature drops to 38C.
7. Seek medical help immediately.

The loop is in line 6. What are we told to do repeatedly? "Fan the victim with a magazine". What is the condition we have to keep checking to determine if we can stop? "The temperature drops to 38C". Rewriting line 6 we get

```
While the temperature is above 38C do the following repeatedly
    Fan the victim with a magazine.
```

All loops can be characterised as above with a termination condition and body. However, certain kinds of termination question come up over and over again and so we can characterise two more specific kinds of loop. In particular, **counter-controlled loops** and **sentinel-controlled loops** are common.

Counter-controlled Loops

One of the most common forms of loop occurs where you know in advance how many times the repetition must be done. The example of the teacher giving a pupil a 100 lines is an example. The pupil knows exactly how many times they must write the sentence before they start. If they have any sense they will keep a count of how many times they have written the sentence: perhaps keeping a tally (making marks in groups of five each time they write a line). When the tally gets to a 100 they stop. Without a counter of some kind they would have no way of knowing when to stop. Doing a task a fixed number of times whilst keeping a count is known as a **counter-controlled loop**.

What do we do repeatedly? We write the punishment line, but we also make another tally mark. What is the termination condition? We continue while the tally is not 100.

```
while the tally is not showing 100 do the following repeatedly
    Write "I must not pick my nose in class".
    Add one to the count by making another tally mark.
```

Here is a dice game. Each player throws the dice 5 times. The player who gets the highest score when all their throws are added together, wins the round. This is an example of a counter-controlled repetition. On each repetition the player will throw the dice, and add their score to the total but they will also add one to their count (perhaps by putting up an extra finger). The termination condition that tells them when to stop is when their count has got to five (or they have run out of fingers).

Problem

Write out the instructions to this dice game in the form of a while loop.

The simplest counter-controlled loop is one where the whole point is just to count. The counting is the repeated task. The game of hide and seek is an example. The child doing the seeking must cover their eyes then count to 10 then shout "Coming, ready or not" then look for the other children. The counting part involves adding one to the last number you thought of.

```
while the number you are thinking is not 10 do the following repeatedly
        add one to the number you are thinking of.
```

What number are you thinking of to start with? One presumably. We ought to say that in our instructions as otherwise a devious child might count "9, 10. Coming ready or not" and could justifiably say that they were not cheating if those were the instructions given. We need to **initialise** the counter: say what its first value is. In fact strictly we should have done this in each of the above examples too – make sure our tally was zero when we started writing lines and making sure that our fingers were showing zero before we started rolling dice.

```
1. Think of the number 1.
2. while the number you are thinking is not 10 do the following repeatedly
        add one to the number you are thinking of.
```

The full instructions for hide and seek are:

```
1. Think of the number 1.
2. while the number you are thinking is not 10 do the following repeatedly
        Add one to the number you are thinking of.
3. Shout "Coming, ready or not".
4. Look for the other children.
```

Notice that these instructions do not tell you to shout "Coming, ready or not" repeatedly. That instruction is not part of the loop. It comes after the loop and is not in the body. It is the instruction you follow after the loop has finished: after the termination condition allowed you out of the loop. You will only shout when the number you are thinking of gets to 10.

All counter controlled loops have the basic elements seen in the above examples:

```
Set the counter to its initial value.
while the counter is not the final value do the following repeatedly
        Do the tasks that are to be repeated the known number of times.
        Change the counter.
```

For example,

```
Set the counter to zero.
while the counter is not 100 do the following repeatedly
        Write "I must not point my gun at Jimmy in class".
        Add 1 to the counter.
```

In the above examples we always added one to the counter when we changed it. That makes us count upwards. We could equally well count down too. Then we would set the counter to the number we wanted to count down from. We would subtract one from our count each time and the number we would stop at would be 0.

```
Set the counter to 10
**while** the counter is not 0 **do the following repeatedly**
    Write "I must not set fire to Sally's hair in class"
    Subtract one from the counter.
```

There is a whole genre of children's nursery rhymes that are loops like this. The most obvious is

There were ten in the bed and the little one said, "Roll over, Roll over"
So they all rolled over and one fell out.
There were nine in the bed and the little one said, "Roll over, Roll over"
So they all rolled over and one fell out.
etc
There were none in the bed and the little one said "Good Night".

This rhyme is just describing a series of actions done in a given order. Re-interpreting it as an algorithm and writing it as a loop we get:

```
1. Ten get into bed.
2. **while** there are any in the bed **do the following repeatedly**
    1. The little one says "Roll over".
    2. They all roll over.
    3. One falls out.
3. The little one says "Good night".
```

This has the same form as the previous example. We initialise the number in the bed. Our termination condition is that this number is not zero. Some actions occur repeatedly and after each time we reduce the number (in the bed) by one. It is a counter controlled loop that counts down.

Problem
Think of another nursery rhyme that is a counter-controlled loop, and rewrite it as one.

When playing darts, each person has three darts. For each of their turns, they have to get the highest score they can with just three darts. This is a repeated task where you know in advance that you will do the task three times. It is a counter-controlled loop where the darts themselves act as a counter that is counting down – start with three darts and when you have no darts stop! What is being repeated? Throw a dart, then subtract the score from your total.

Problem
Write the instructions for throwing darts as a counter-controlled loop.

Sentinel-controlled Loops
Consider the game of Dice Cricket. A special Dice has on its sides the numbers 0, 1,2,4 and 6 and on the last side is written "OUT". One player acting as the batsman repeatedly rolls the dice, adding up the scores obtained which correspond to the number of runs scored on that bowl. If they roll "OUT" then that batsman is out and it is the end of their turn. Here the termination question is

"Is the dice roll OUT"

The sentinel value is OUT showing on the dice.

To give an algorithm with sentinel controlled repetition we need to indicate that some instructions need to be repeated, what the question is and which instructions exactly we want to be repeated while the answer to the question means carry on. We can write it out similarly to above. For example the above game could be written out as follows:

```
while the dice is not showing OUT do the following repeatedly
    1. Add whatever is showing on the dice to the current score.
    2. Roll the dice.
```

We roll the dice. If the dice shows out we stop. If it does not, then we add the dice value to the score and roll again. We then repeat this all again and keep doing so until the dice does show stop, at which point we have finished the while instruction and carry on with any subsequent instructions. If we tried to play the game following the above rules alone, they would not quite work. Can you see what is wrong? If not get a dice and do exactly what the instructions tell you and then try to correct the instructions above before reading on.

There are two problems. The first is that we have not said what the score is at the start. Do I start with a score of 100 or of 0? Understanding the game I know that the score starts at 0, but the algorithm does not say that, so somebody who was trying to learn the rules from my instructions would be confused. In fact if we were playing a series of rounds of the game then we might actually want the score to be its previous value. For now we will assume we are only playing one game so zero is the correct initial score. We need to add an extra instruction “Set the score to 0”. Notice this is something that we only want to happen once so we need to make it clear that it is not part of the repetition, but comes first.

```
1. Set the score to zero.
2. while the dice is not showing OUT do the following repeatedly
    1. Add whatever is showing on the dice to the current score.
    2. Roll the dice.
```

This still has a problem however. We set the score to zero, then go into the loop, asking if we should do the instructions to be repeated or stop straight away: is the dice showing OUT? However, we have not rolled the dice yet! We have not got to the instruction that tells us to do that yet! We need to roll the dice once first before we do the repetitive part, to get us started.

```
1. Set the score to zero.
2. Roll the dice.
3. while the dice is not showing OUT do the following repeatedly
    1. Add whatever is showing on the dice to the current score.
    2. Roll the dice.
```

Both these extra instructions are **initialisation** instructions. They are needed to set initial values for the things the loop manipulates. Most loops need something initialising to work and they are easy to forget when writing instructions, so it is an important thing to double check.

The game of Pass-The-Pigs is similar. This game involves tossing a pair of special Pig shaped dice. They can land on their feet, on their back, on either side or even occasionally on their snouts. Different positions score different amounts, but if a particular position of both pigs on the same side is rolled, the players turn ends. Here the termination question is

"Are the pigs both on their sides"

(In fact the termination question is slightly more complicated as the turn could also end by the player deciding to give up and so keep their score).

Darts has a similar termination condition. There each player takes turns throwing 3 darts and subtracting the score from their total. The game (and so the repetition) ends when the score hits exactly 0. The termination condition is thus

"Is the score of the current player 0"

These termination conditions have the same form: they all ask

"Has a particular value arisen".

In motor racing, drivers repeatedly lap a circuit. On each circuit the pit crew hold up boards to give information to the driver. One of the situations in which a driver can be made to stop is if the information on the board tells him to come into the pits for a tyre change. Here the termination condition is

"Does the board on this lap say Change Tyres"

The particular value being checked for differs completely in each case but the form of the question is the same. Because this kind of termination condition arises frequently the corresponding kind of loop is given a special name: a **sentinel-controlled loop**. The special value that if it arises causes the repetition to stop is called the **sentinel value**.

Sentinel values are normally values that are arising from outside the system (rather than in the Darts case where it is generated by a calculation done inside the loop). This means that there must be some value that arises in the same way as the others but that is not needed to hold an actual value. For example, in dice cricket, if we only had a six-sided dice and wanted batsmen to be able to score 0,1,2,3,4 or 6, then there is no side of the dice left to be the sentinel. If we pick any of the values to be a sentinel, it can no longer be used as a score as then we would not be able to tell whether on a given roll it should mean the score or out. We must always "waste" one value that is not treated in the same way as the others.

Non-terminating Loops

The following appeared on the grave stone of Catherine Alsopp who hanged herself:

"Don't mourn for me now, don't grieve for me never,

For I'm going to do nothing for ever and ever" (quoted in Lovric 2000)

Suppose we wanted to give precise instructions to do nothing for ever and ever. How would we do it? Is it possible using one of our loop constructs? it clearly is a repetitive thing we want to be done. What is it that we wish to be done over and over? Nothing! Under what circumstances do we keep doing it? Always! Our loop will be something like:

```
While true do the following repeatedly
    Do nothing.
```

We needed a test to mean “always” and used “true” for this. How does that work? Well we want to always keep going and the loop keeps going when the test is true. So if we always want it to keep going we just use “true” as the test. We are basically seeing “While true is true do the following repeatedly”. Since true is always true the answer to the question is always yes. This kind of loop with a test that is always true is known as a **non-terminating loop**. Usually non-terminating loops are bugs – run-time errors in the program. However sometimes you do never want a program to stop (e.g. it might be useful if a program controlling a heart pace maker would keep going forever).

A Harder problem: Loops inside loops

Back in the first chapter we looked at a puzzle to exchange the positions of two sets of pieces on a board of seven squares, either by sliding or jumping pieces. Here is the algorithm we came up with:

A 15-step algorithm for solving the puzzle is as follows:

1. Move the piece in square 2 to square 3.
2. Jump the piece in square 4 to square 2.
3. Move the piece in square 5 to square 4.
4. Jump the piece in square 3 to square 5.
5. Jump the piece in square 1 to square 3.
6. Move the piece in square 0 to square 1.
7. Jump the piece in square 2 to square 0.
8. Jump the piece in square 4 to square 2.
9. Jump the piece in square 6 to square 4.
10. Move the piece in square 5 to square 6.
11. Jump the piece in square 3 to square 5.
12. Jump the piece in square 1 to square 3.
13. Move the piece in square 2 to square 1.
14. Jump the piece in square 4 to square 2.
15. Move the piece in square 3 to square 4.

This was the answer for the puzzle where there are three pieces of each colour. However, this is just one of a whole family of similar puzzles with different sized boards. For example, while on holiday in Dorset I came across a wooden version called “Puffin Round Up” being sold by Puffin Wooden Games (Portland, Dorset DT5 2LN). It is identical to the one we looked at, but has 5 pieces of each colour on a board of size 11. It adds the extra rule that pieces cannot be moved backwards. Our algorithm above never moves a piece backwards anyway so that rule makes no difference. We can similarly add a rule a piece cannot jump over another piece of the same colour without it being a problem as none of our moves do that either. The numbers 3 and 5 are not special. You could also have a version with 20 pieces of each colour or even a 100. In fact for each number there is a version of the puzzle with that number of pieces of each colour. Do we need to devise a new algorithm for each version? Or can we write one set of rules that can be followed and would work whatever the version of the puzzle we are trying to solve? It turns out that the algorithm for each version is very similar. Using loop instructions it is possible. See if you can do this before reading on (its quite hard). HINT: A good problem solving approach to general problems like this is to try some specific cases first, looking for patterns. Only once you understand the answers for the individual cases is it worth trying to solve the general case: try and write algorithms for the versions of the game with 4 and 5 pieces and see if you can spot the pattern common to all.

Here is the algorithm for the puzzle with 4 black and 4 white pieces. It looks similar to the original. What is the common pattern they both share?

1. Slide the white piece.
2. Jump the black piece.
3. Slide the black piece.
4. Jump the white piece.
5. Jump the white piece.
6. Slide the white piece.
7. Jump the black piece.
8. Jump the black piece.
9. Jump the black piece.
10. Slide the black piece.
11. Jump the white piece.
12. Jump the white piece.
13. Jump the white piece.
14. Jump the white piece.
15. Slide the black piece.
16. Jump the black piece.
17. Jump the black piece.
18. Jump the black piece.
19. Slide the white piece.
20. Jump the white piece.
21. Jump the white piece.
22. Slide the black piece.
23. Jump the black piece.
24. Slide the white piece.

To make the pattern more obvious we will change the way we write the algorithm to get rid of all the numbers of board positions: it will be the same algorithm just a less precisely written version. We will concentrate for now on the three piece version. Notice that given we are not allowing a piece to move backwards towards its original position, at any time there are only four possible moves: Jump a black piece over one of the other colour, Jump a white piece over one of the other colour, Slide a white piece forward, and slide a black piece forwards. Rewriting our solution algorithm using those instructions instead of one giving board positions (each instruction will still be referring to a unique unambiguous move, but it we are now leaving the player to work out where the piece to move actually is). The algorithm becomes:

1. Slide the white piece.
2. Jump the black piece.
3. Slide the black piece.
4. Jump the white piece.
5. Jump the white piece.
6. Slide the white piece.
7. Jump the black piece.
8. Jump the black piece.
9. Jump the black piece.
10. Slide the white piece.
11. Jump the white piece.
12. Jump the white piece.

13. Slide the black piece.
14. Jump the black piece.
15. Slide the white piece.

Follow this algorithm to check you understand it and it does work. Then think about its repetitive pattern. What is the pattern?

The pattern is that it alternates a series of jump moves with a single slide move of the same colour. The number of jumps increases by one each time in the first part of the algorithm and then decreases by one in the second part. The colour also alternates- first you slide/jump white pieces, then you slide/jump black pieces, then go back to white pieces and so on. There is a pattern there but it is all really complicated, so if we are going to get to grips with it we will need to make things simpler for ourselves. One way to do this is to write it out in a way that makes the pattern even more obvious. Another good problem solving technique is to split the problem into smaller parts and tackle each in turn. First let us make the pattern more obvious. We decided that the first part of the pattern is that it repeats a series of jumps and a slide move. Let us make that more obvious by writing each series of jumps as a single instruction.

1. Slide the white piece.
2. Jump 1 black piece.
3. Slide the black piece.
4. Jump 2 white pieces.
5. Slide the white piece.
6. Jump 3 black pieces.
7. Slide the white piece.
8. Jump 2 white pieces.
9. Slide the black piece.
10. Jump 1 black piece.
11. Slide the white piece.

Here when I give an instruction like “Jump 2 white pieces”. I mean do jumps in a row (though they will be different pieces jumping. Each of those instructions is actually a counter controlled loop: we are doing something (a jump) a known number of times. We will return to that problem later – it is a smaller problem we can solve separately. First let us see if we can replace the above algorithm as it is written into one or more loops. There is obviously something being repeated – jump-slide pairs so we ought to be able to replace the above by something a bit like:

```
while .... do the following repeatedly
        Jump ...
        Slide ...
```

That is repeatedly do a jump followed by a slide move (or maybe it is the other way round). But writing out the detail is tricky – and what is the continuation condition? One problem is that the number of slides first increases and then decreases again. We have seen counter controlled loops where we have a counter going up and others where we have a counter going down. But how do you do both? The answer is to use that problem solving strategy again:

If the problem is difficult break it into smaller bits that you can solve.

Here, we can break it into two parts: the count up and the count down part. The count up part involves doing the same puzzle but where the aim is to get the pieces half way so that they alternate: black piece-white piece (see the diagram). Let us suppose we were set the puzzle just to get to that position, the algorithm would be just the first half of ours above:

1. Slide the white piece.
2. Jump 1 black piece.
3. Slide the black piece.
4. Jump 2 white pieces.
5. Slide the white piece.
6. Jump 3 black pieces.

Now what is the best way to think of the pairings: a slide followed by a jump or a jump followed by a slide? You could do it either way. Given how we have written it, the most obvious is as a slide followed by a jump. (Surprisingly it turns out it is easier to do it if you think of it as a jump followed by a slide so we will look at that in a moment). Now it is looking more like something we can sort out with a counter controlled loop. However we still have a problem: the colours alternate. Here is another problem solving tip.

If the problem is too difficult solve an easier version of it first.
Only then go to the original harder version with you improved understanding.

So let us ignore the colours for now – if we cannot solve it without the colours we will not solve it with them. Our algorithm becomes:

1. Slide the piece.
2. Jump 1 piece.
3. Slide the piece.
4. Jump 2 pieces.
5. Slide the piece.
6. Jump 3 pieces.

Now it is much easier to see how we turn this into a loop: it is just a counter controlled loop! Write it out yourself before reading on.

We stop when the counter gets to four (there is no "Jump 4 pieces"), and what we do repeatedly is a slide followed by a jump.

```
Set the counter to 1.
While the counter is not four do the following repeatedly
    Slide the piece.
    Jump counter pieces.
    Add 1 to the counter.
```

However, that is not quite the algorithm we want as it does not say which coloured piece to slide or jump each time. Looking back to the earlier version, the first time we wanted to slide a white piece, then jump black pieces. However we cannot just put that in the algorithm:

```
Set the counter to 1.
```

```
    **While** the *counter* is not four **do the following repeatedly**
        Slide the white piece.
        Jump *counter* black pieces.
        Add 1 to the *counter*.
```

This does not work as the second time round the loop we want to slide black, then jump white! If something varies, then we need a variable to represent it! Here the easiest thing is to have two that we will call: *Slide Colour* and *Jump Colour*. As we saw above, to start with we want the Slide Colour to be white and the Jump Colour to be black. We initialise them as such.

```
    Set the *Slide Colour* to white.
    Set the *Jump Colour* to black.
    Set the *counter* to 1.
    **While** the *counter* is not four **do the following repeatedly**
        Slide the piece with colour *Slide Colour*.
        Jump *counter* pieces of colour *Jump Colour*.
        Add 1 to the *counter*.
```

That only partially solves the problem though: the colours still do not alternate. We need to add extra instructions into the loop, to make them alternate. If the Slide Colour is white then we want to make it black for the next time round the loop and vice versa. We are doing one of two different things depending on the current colour: so we need an if statement.

```
    **If** the *Slide Colour* is currently white
    **then** Set the *Slide Colour* to black.
    **else** Set the *Slide Colour* to white.
```

Notice how whichever colour the Slide Colour is currently this instruction switches it to being the other one. The Jump colour needs to be changed in exactly the same way. We need to add these lines to the end of our loop, ready for the next time round the loop.

```
    Set the *Slide Colour* to white.
    Set the *Jump Colour* to black.
    Set the *counter* to 1.
    **While** the *counter* is not four **do the following repeatedly**
        Slide the piece with colour *Slide Colour*.
        Jump *counter* pieces of colour *Jump Colour*.
        Add 1 to the *counter*.
        **If** the *Slide Colour* is currently white
        **then** Set the *Slide Colour* to black.
        **else** Set the *Slide Colour* to white.
        **If** the *Jump Colour* is currently white
        **then** Set the *Jump Colour* to black.
        **else** Set the *Jump Colour* to white.
```

That is now our algorithm for doing the first part of the puzzle – just getting the pieces to halfway. The second part is similar though.

1. Slide the white piece.
2. Jump 2 white pieces.
3. Slide the black piece.
4. Jump 1 black piece.

5. Slide the white piece.

It is just a counter controlled loop counting down doing slide-jump pairs repeatedly, then finishing with an extra slide move after the loop.

The pattern is that it alternates a series of jump moves with a single slide move of the same colour. The number of jumps increases by one each time in the first part of the algorithm and then decreases by one in the second part. The colour also alternates- first you slide/jump white pieces, then you slide/jump black pieces, then go back to white pieces. With more pieces this

Recursion
Fanny Robin, one of the characters in Thomas Hardy's novel "Far from the Madding Crowd" (Hardy, 1874), suffering from exhaustion, is trying to walk to Casterbridge and safety. Eventually her exhaustion overcomes her and she collapses. After lying in the road for 10 minutes or more, she struggles up again. Seeing the lights of Casterbridge she calculates how far she must still go and it seems desperately far:

> "*Five or six steps to a yard, ... I have to go seventeen hundred yards. A hundred times six, six hundred. Seventeen times that. O pity me, Lord*"

Rather than give up she comes up with a way of beating this seemingly impossible problem.

> " *'I'll believe that the end lies five posts forward and no further and so get strength to pass them.'*
> *She passed five posts.*
> *'I'll pass five more by believing my longed-for spot is at the next fifth'*
> *She passed five more.*
> *'It lies five further.'*
> *She passed five more.*
> *'But it lies five further.'*
> *She passed them.*"

In this way she drags herself towards her destination. Her approach to problem solving is one that is widely useful. Rather than trying to solve a seemingly impossible problem in one go, she devises a way of breaking it down into a series of identical but increasingly smaller problems that are easily achievable: walking five posts further down the fence. After each smaller problem is solved she is left in the situation of solving an identical looking problem: her destination is still an impossibly long way away. However, it can be attacked in the same way as the original problem was. Gradually her problem is solved.

Recursive problem solving uses this trick. Solve a problem by finding a simpler version of the same problem together with a way of converting your problem into that simpler one. **Recursion** is about having a series of things that are the same except for some property that gradually gets simpler due to small identical steps being taken. For Fanny Robins, dragging herself along the highway, the property that was getting smaller was the distance to her destination. Otherwise the problem looked the same: an impossibly long line of fence posts. The individual step that made the property smaller was the passing of five posts.

The following poem based on a quote of Swift (the author of *Gulliver's travels*) gives a feel for the idea of recursion.

Great fleas have little fleas
Upon their backs to bite 'em
And little fleas have lesser fleas
And so ad infinitum

Augustus De Morgan (quoted in Gardner, 1977)

Toy robots are all the rage at the moment. I have therefore just invented a robot, called Robbie, that can climb down a flight of stairs, or at least so far I have got it to go down a single stair. He can also do recursive problem solving to solve big problems by breaking them down into smaller but similar problems. The basic things I originally taught him to do was to go down a single step. Robbie also knows how to stop and knows how to check when at the base of a flight of stairs. I stand him at the top of the stairs tell him to get to the bottom of the stairs...and hold my breath. Does he know enough to recursively solve the problem of getting down a whole flight of stairs? Robbie thinks of his state at the top of the steps and his problem. How does he see the problem ahead? He sees a long flight of stairs ahead of him. He does not know immediately how to descend a whole flight of stairs. If he is to come up with a recursive solution he needs to find a way of converting the problem into a similar problem that looks the same but is a little bit simpler. What does he know how to do that will leave it with a similar situation? Well he can take a step. What will his problem be then? He will see a long flight of stairs ahead just as before, but the new problem is simpler – the flight of stairs are not quite as long. How can he get down the stairs? He can take a step and then tackle the remaining flight of stairs in the same way. A part of a recursive solution that involves turning the problem into a similar problem is called a **step case**. For Robbie the step case is taking an actual step, but in general it is anything that converts the problem into a simpler but similar problem. Based on the above reasoning Robbie formulates the recursive algorithm:

```
To descend a flight of stairs:
1. Take a step.
2. Descend the remaining flight of stairs (using this same algorithm)
```

He takes the plunge and starts to follow the algorithm he has devised. By following this algorithm as expected he takes step after step and eventually gets to the bottom but then it all goes wrong - he keeps trying to take steps even though he is already at the bottom and once he starts to follow an algorithm he does not stop until an explicit instruction in the algorithm tells him to. Luckily, eventually his batteries run out and he falls over exhausted. The problem is that the algorithm never terminates. Robbie did not include an instruction to stop. The algorithm needs a way of checking when to stop: it needs a **base case**. For Robbie, the base case is when he gets to the base of the stairs. In general the base case is just the case when the problem is so simple it can be solved in some trivial way (like just stopping). With new batteries Robbie learns his lesson and adds a base case to his algorithm:

```
To descend a flight of stairs:
    if at the bottom of the stairs
    then (the base case)
        STOP
    else (the step case)
      Take a step.
```

```
        Descend the remaining flight of stairs (using this same algorithm)
```

Now once at the bottom it just stops and does not try to carry on descending stairs.

A physical thing that has this kind of **recursive** property is a set of Russian Dolls: wooden dolls that break in half and fit inside one another. Each wooden doll is identical except for the property of being smaller than the last. However, the dolls do not go on getting smaller forever. Eventually as you take the dolls apart you get to the smallest doll, which does not split in half and does not have a smaller doll inside. The smallest doll is equivalent to the **base case** of the recursion. The dolls that do break apart to reveal yet more dolls are the **step cases** of the recursion. If there was no base case in a set of Russian dolls it would mean every single doll had another one inside it. You would open one up and there would always another one inside that could be opened itself to reveal another which....just like the fleas.

The Children's story "The Cat in the Hat Comes Back" (Dr. Seuss, 1997) has recursion as a central part of its plot. The Cat in the Hat has left a stain in the bath that he can move from one place to another (from the bath to a dress to some shoes, etc) but not remove. Eventually he realises the problem is too much for him to solve alone, so instead he lifts off his hat to reveal a smaller but otherwise identical cat in a hat ("Cat A") standing on his head. This cat brakes up the spot and moves the stain to a place where the cat in his own hat ("Cat B") can deal with it. He splits it into smaller spots and moves it to a place where the cat in his hat ("Cat C") can deal with it.

> *" 'With some help we can do it!'*
> *Said Little Cat C.*
> *Then POP! On his head*
> *We saw Little Cat D!"*

The stain is split and moved around by a whole chain of identical but increasingly smaller cats. Eventually "Cat Z" takes off his hat to reveal not another Cat but something totally different called "Voom" that can clean up all the small spots on its own.

> *"Then the Voom...*
> *It went Voom!"*

Voom is a base case if ever I saw one! Base cases solve problems in a different way to the step cases (the cats in the hats) but can only deal with the problem once they have been broken down by the step cases.

The idea of recursion can be used as a way of solving problems and thus of structuring an algorithm: giving **recursive algorithms**. To solve a problem using the recursive approach you must do the following.

1) First work out a series of simpler versions of the same problem that are otherwise identical.
2) Next, find a way of transforming a version of the problem to the next simplest version from part 1 (the step case).
3) Finally find a version of the problem that can be solved directly (the base case)

The recursive algorithm will then be of the form:

```
    To solve the problem:
        if the problem is the simplest version (the base case)
        then solve it directly
```

```
else
  transform the problem into the next simplest version (the step case)
  solve the simpler problem in the same way
```

The following is a recursive algorithm for completely taking apart a Russian Doll.

```
To take a series of Russian dolls apart:
    if you are holding the smallest doll
    then place it on the table
    else
        Take the halves of the largest doll apart
        Take out the doll inside it and place it on the table.
        Take the smaller Russian dolls apart in the same way
```

Notice that the problem has been transformed into a repetitive task: doing something over and over again. It therefore should consist of the same basic elements as the loops we saw earlier and it does. There is a test to see if the repetition has finished (the test to see if we have arrived at the base case: "Are you holding the smallest doll") and a series of instructions to be repeated each time (the step case: "Take the halves ..."). The base case appears to be extra, though in many respects it is similar to the initialisation stage with a loop. In fact any solution that can be written recursively could instead be written in the form of a loop. They are just alternative ways of thinking of the same thing. Some problems are more obviously described using recursion and some using loops, but any repetitive problem could be described either way.

Recursion does not have to involve doing the transformation task first and then doing the recursion to finish the problem. Sometimes you do the recursion first to get a result to work with, then do the **step** part of the task. This is the situation when putting a Russian doll back together.

```
To put a series of Russian dolls together:
    if you are holding the smallest doll
    then just hold on to it
    else
        Put the smaller Russian dolls together in the same way.
        Put the next smallest doll (which already has all the others
        inside it) inside the base of this doll.
        Put the top on this doll
```

The feature of a recursive algorithm is that it is defined in terms of itself – there is always a line that says something like "solve the simpler version using this algorithm in the same way". The problem is reduced to just transforming the problem into a slightly simpler version, rather than solving the whole thing in one go. At first sight recursive algorithms look paradoxical in that you appear to be saying something like "to solve this problem, you solve it in the same way" which does not appear to get you anywhere. For example, if you were given the instructions below they would be no help at all.

```
To get to Liverpool Street Station:
    First go to Liverpool Street Station
```

Although this looks like a recursive solution it is not. The secret of recursion is that on each step you have made the problem smaller. Even that is not enough on its own as you would then go on forever always needing to solve a simpler problem. However the base case stops this happening as you will eventually hit it and so no longer need to solve a simpler version of the problem.

Not all problems can easily be solved using recursive problem solving, but ones that can, can often be solved very easily using recursion. The secret is

The following algorithmic puzzle for which a recursive solution can be given is adapted from Kordemsky, 1975.

> *A company of soldiers arrive at a river that they must cross. However, it is too deep to wade, and due to the cold weather it is important that they do not get wet, so they cannot swim. Luckily two children are playing in a small rowing boat nearby. Unfortunately, the boat is large enough only for the two children or a single soldier to be in it at once, though one child can row it on their own. The soldiers do not have any rope or anything else that would be of use. The sergeant gives your group the task of working out a way to get the whole company across the river. How do you do it?*

Try to solve this problem before reading on, writing out a recursive solution. Rather than trying to do it in your head, you may find it easier if you use coins to represent soldiers and children.

The recursive part of the solution is given below. We need to make the observation that if we can move one soldier across the river leaving the children back in the boat, then we will be back in an identical situation, but with one less soldier to get across the river. The base case is when no soldiers remain to cross – then there is nothing more to do.

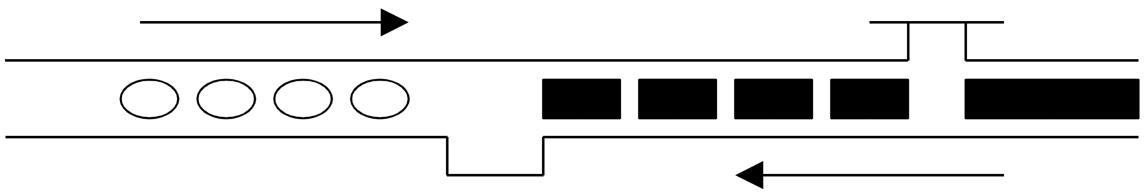
```
To get any remaining soldiers across the river:
    if all soldiers are across
    then stop
    else
      Move one soldier across ending up with the children back in the boat
      Get any remaining soldiers across in the same way
```

We still need an algorithm for getting a single soldier across. For the recursive solution to work the children must be back in the boat too, as otherwise we will not be back in the same situation as when we started.

```
To move one soldier across ending up with the children back in the boat:
    Both children go across in the boat.
    One child comes back leaving the other behind.
    A soldier goes across, leaving the child behind.
    The child on the far bank comes back and picks up the other child.
```

The following puzzle also can be solved recursively (an equivalent puzzle concerning balls is given by Kordemsky, 1975)

A long narrow road contains a single passing point. A single car or van can pull into the passing point allowing cars and vans then to pass it along the road. A side road to the right of the passing point contains a low bridge that cars can get under but that lorries and vans cannot. 4 cars are trying to move from the left and wish to go down the side road. 4 vans followed by a lorry are moving from right to left along the main road (see diagram). The lorry driver has stopped just before the junction and is refusing to reverse as she can see how the jam can be solved without the lorry moving.



Find a way for solving the traffic jam, without the lorry reversing or the cars having to reverse back the several miles to the point where the road widens.

Before we get to the recursive part of the solution we need to do an additional step, to give us a situation that we can get back to. This first step is to reverse the back 3 cars far enough to leave room for the 4 vans to fit between them and the passing place. A recursive solution without this first step would require moving the cars forward and then immediately backwards unnecessarily. This kind of initial manipulation of the problem is often needed for a recursive solution.
The recursive solution is then:

```
To get the next car down the side road:
        if all cars are out
        then the road is now free to move the vans and lorry on down the road
        else
                Move one car out leaving all the other vans in their original
                positions.
                Move the remaining cars out in the same way
```

Now the step case must free one car leaving the other cars and vans in the same positions at the end.

```
To move one car out leaving the vans in their original positions:
        Move the first car into the passing place
        Move all the vans along the road past the passing place.
        Move the car in the passing place off down the side road.
        Reverse the vans back to their original position.
```

The Tower of Hanoi puzzle lends itself well to a recursive solution. It consists of three poles. On the first pole are a series of rings of increasing size (see diagram). At no time can a ring be placed on top of a smaller ring. The aim is to move all the rings to the last pole. Give a recursive algorithm to do this.

Lets suppose we have a way to move all but the bottom ring from one peg to another (this will be complicated to do, but it must be simpler than the current problem as it involves one less peg, so lets not worry how for now). If we could do that we could solve the puzzle. How? We move all but the bottom ring to the middle peg, out of the way, then move the bottom ring to its correct place on the empty end peg (we can do this because it no longer has any pegs on top of it). We then move all the other pegs over to sit on top of the bottom ring on the last peg. Done! Ah, but how do we move that smaller pile of rings? We do it in the same way – move all but one (using the same method again) to the peg other than the one we really want them to move to, move the bottom one of them, then move the rest back (again using the same technique. We have a recursive algorithm.

```
To move a series of rings from one pole to a target pole using a spare pole:
        if there is only one ring
        then move it directly to the target pole
        else
                Move all but the bottom ring to the spare pole in the same way.
                Move the bottom ring to the target pole.
                Move all the rings currently on the spare pole to the target pole in the
                same way (the original pole is treated as the spare pole to do this step).
```

This algorithm has two recursive calls in each step. To solve the problem, you have to solve two identical but simpler versions of the problem. Both recursive calls are simpler than the full puzzle as they involve moving only one less ring. In both cases they involve moving all but the bottom ring from one peg to another. How do you do that? Not by moving them all together in one go, but by moving them one at a time (over and over again) – in exactly the same recursive way.

I once worked as a floor mopper and toilet cleaner for a company called Loadsa Electronics Gizmos Ltd in Cambridge. The main workshops were full of wonderful gizmos at various stages of development. The management were always worried about their competitors sending in spies to steal the best ideas. They therefore had a special lock on every door in the building that you could only get through if you knew the special code (with a special code for each door). The locks were the push button ones – a series of 4 buttons, numbered from 1 to 4 that you had to push in or out just the right order to get in. The buttons stayed in or out as you pushed them. If you pushed an in button it would pop out, and if you pushed an out button it would go in. There was also an extra button marked “clear”. As the main workshop was really important, the sequence on its lock was really long, judging by the length of time people stood there pushing buttons.

I was not, as a lowly toilet cleaner, allowed in to that room, though I knew the codes to every other room so that I could mop the floors. I was not a spy, but I was curious about what was in the room. I spent a great deal of time mopping the corridor outside the room, so had ample time to watch the engineers as they laboured pressing the correct sequence. It was clear since they could all remember it, despite its length, that it was some easy to remember algorithm that was being followed. Counting the number of presses the engineers made, it was clear that the correct sequence was 15 presses long plus the clear button that was sometimes pressed first, if the buttons had been left in random positions) or if an engineer made a mistake in the sequence. It had the effect of making all the other buttons pop in. Also the door always opened only when all but the last button was pressed in.

After several more days watching it was clear that every button push involved either pushing button 1, or pushing the button immediately after the lowest numbered one that was currently pushed out (if any were pushed out). Thus if buttons 1 and 2 were in and button 3 out (so it was the lowest numbered button out) then button 4 could be pressed. After more watching I realised that engineers that managed to unlock the door in only 15 button pushes also never pushed the same button twice in a row.

With this information I knew the sequence. Can you write down the sequence of button pushes that gives the algorithm? Assume it starts with all buttons in so that clear does not need to be pressed and that “push 1” means for example, push button 1.

The algorithm is as below (where we add comments to the instructions in bold that give the position of all the buttons that results after that move – 0 means pushed in, 1 means pushed out):

	0000
Push 1.	**1000**
Push 2.	**1100**
Push 1.	**0100**
Push 3.	**0110**
Push 1.	**1110**
Push 2.	**1010**
Push 1.	**0010**
Push 4.	**0011**
Push 1.	**1011**
Push 2.	**1111**
Push 1.	**0111**
Push 3.	**0101**
Push 1.	**1101**
Push 2.	**1001**
Push 1.	**0001**

There are some interesting things about this sequence. The first thing is that it includes every possible combination of 1s and 0s (though not in the normal binary sequence of counting). In fact the sequence is simpler than normal binary counting as you move on from one sequence to the next, just by flipping one of the 1s to a 0 or vice versa. It also has a recursive property, but we will get to that in a moment.

I wrote it down so I would not make a mistake. Then early one morning, before anyone else was around and the security guard was having his morning cuppa, I typed in the sequence and went into the room. As I said, I’m not a spy so I wont say what was in there, but it was full of many fascinating gizmos. Unfortunately, I left the piece of paper with the code on, on one of the work benches and forgot it when I left just before the guard was due to finish his cuppa. Later that afternoon the Manager noticed the note and all hell broke out, as it was Company policy that no one should write down the code of any door. The next morning the lock had been replaced by a new lock specially built in the workshop, with 5 numbered buttons instead of 4 for extra security. Now the engineers had to push 31 buttons instead of just 15! However, with careful observation I realised that the rules were the same and that it was possible to come up with a recursive algorithm that would work whatever size they made the lock. In fact there was also a safe inside the workshop, that had 6 numbered buttons, and the same algorithm worked for it too. Can you work out what the recursive algorithm is?

Look at the sequence of numbers pushed (1,2,1,3,1,2,1, **4**,1,2,1,3,1,2,1). Button 4 is pushed once in the middle. Before and after it, exactly the same sequence of pushes is followed: (1,2,1,**3**,1,2,1). This sequence has the same property again. The middle number is the largest, with the same sequence on either side. How do you generate the sequence for 4 buttons? You first follow the sequence for three buttons, then press button 4, then follow the sequence for three buttons again. But that begs the question, what is the sequence for three buttons? Well it is worked out in the same way: follow the sequence for two buttons (1,**2**,1), press button 3, then follow the sequence for two buttons again. How do we follow the sequence for two buttons: follow the sequence for one button (1), press button 2, then follow the sequence for one button. The sequence for one button is trivial: just press it!

We can write this down recursively, by replacing the number of buttons by a variable n.

```
To go through the sequence of pushes for n buttons:
    if n is 1
    then   Push button 1
    else
           Go through the sequence of pushes for (n-1) buttons in the same way.
           Press button n.
           Go through the sequence of pushes for (n-1) buttons in the same way.
```

This has virtually the same recursive pattern as that for the tower of Hanoi puzzle! Just replace pushing numbered buttons by moving numbered discs.

Summary

Loops are ways of repeating something over and over again. To describe a loop you need to specify what is to be repeated and when to stop. There are several different kinds of loop that involve giving different kinds of termination condition. Counter-controlled loops involve keeping a counter and stopping when it gets to a previously known number. In such a counter-controlled loop you know in advance how many repetitions are needed. An alternative is to stop when a given distinguished value arises. This is a sentinel controlled loop. Unlike a counter-controlled loop, in a sentinel-controlled loop you do not know how many repetitions will be required before you stop.

Recursive algorithms can be used to solve a wide range of problems. They involve transforming a problem into a similar but simpler problem that can then be transformed in the same way. As each transformation is done, the problem becomes simpler and simpler until it is so simple that it can be solved directly without transforming it further. Recursion is just an alternative to giving a loop. Any algorithm that can be expressed recursively could be described without using recursion.

However, it is often simpler to give a recursive algorithm than a non-recursive solution. Many of the algorithms we have seen in the earlier chapters such as binary search can very easily be described using recursion.