13 Algorithms for problem solving

A broader definition of numeracy extends far beyond arithmetic and the application of number. Numeracy skills which are highly valued by employers include: problem solving in a vocational context, the ability to work with information technology systems, and an ability to convert between different representations of quantitative data using numbers, diagrams or algebraic expressions as appropriate. All of these wider skills come together in the design of **algorithms**.

An algorithm is a set of instructions for carrying out a task. This might be something as familiar as a cookery recipe:

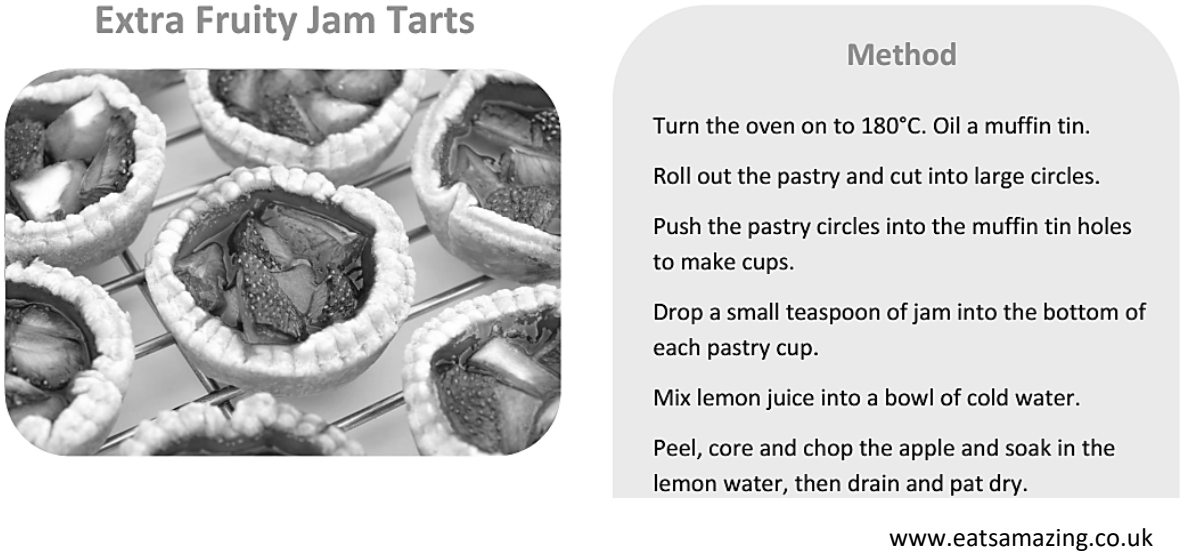


Figure 415: algorithm for making jam tarts

However, we tend to use the term ***algorithm*** most frequently when referring to sequences of instructions in computer programs. Typically, an algorithm involves a multi-step calculation, and its design can require a high level of numeracy and problem solving skills.

When presenting the design for an algorithm, the steps may be written in ordinary language, as in the recipe example above, or may be displayed in the form of a flowchart or other type of diagram.

The use of mathematical algorithms in computer programs has become increasingly important as computer systems have become more powerful. Software can now accomplish many of the complex tasks which were previously only possible through the application of human intelligence. Examples which we will examine in this chapter include journey planning, fast sorting of data, game playing, and data encryption.

We begin by investigating algorithms used in journey planning.

Dijkstra's algorithm

A frequent requirement in computer applications is to find the shortest, quickest, or cheapest route from one place to another through a network of possible paths. To demonstrate this task, we will consider a small network:

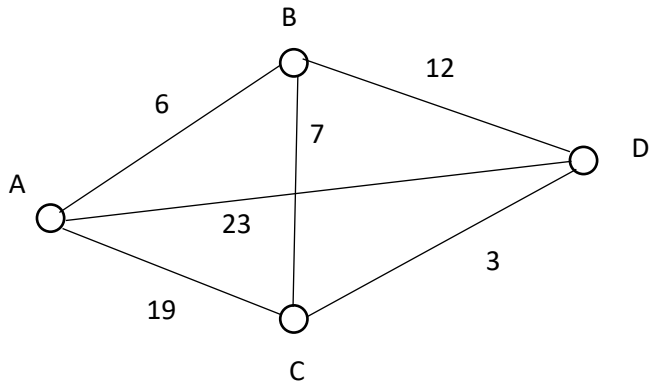


Figure 416: network of towns and journey distances

The **nodes** in the network, A – D represent four towns. The **links** between the nodes represent roads, with distances shown in km. The network is represented diagrammatically, so the links are not drawn to scale. We are simply interested in the **topology** of the network: the way in which the nodes are connected.

The objective of the task is to find the shortest route from town A to town D, out of the various routes which are possible. If we were to do this manually, we might make a few simple calculations by mental arithmetic before deciding on the solution. However, if the task is to be undertaken by a computer, then a very precise algorithm for checking all possible routes must be specified. An efficient method for carrying out this task is **Dijkstra's algorithm**. The same algorithm can be used by the computer to quickly find solutions to much more complex and challenging route finding problems, such as choosing the shortest route to travel by road from Cardiff, via the Channel Tunnel, to Budapest!

Returning to our simple network, the first step is to identify the towns which can be reached directly from the starting point. We then record the distances travelled along the links.

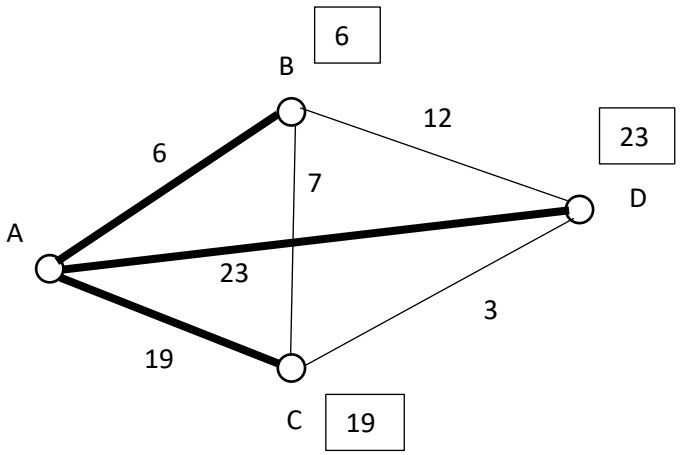


Figure 417: links from town A

Notice that the distances are only **temporary** solutions at this stage. It may be possible to reach some of the towns by a shorter route. (This is indeed the case for town C, which could be reached in only 13 km by travelling via town B.)

We now look at the temporary distances and choose the lowest of these. This is the distance of 6 km for town B. This ***must*** be the shortest distance to town B. Any other route to town B would involve travelling via one of the other nodes whose distance from the start point is already more than 6 km. We can therefore make the distance to town B a **permanent** solution. We now examine the routes from town B to any other nodes which still only have temporary distances allocated:

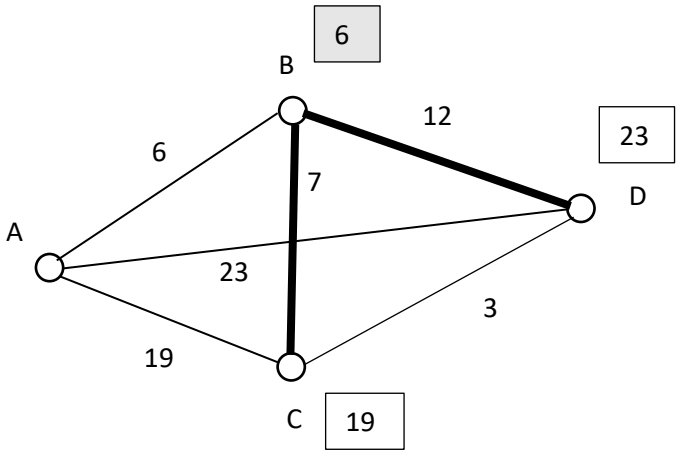


Figure 418: links from town B

If we travel to town D via town B, the total distance would be 18 km. This is an improvement on the previous value of 23 km, so we can update node D. In a similar way, it is only 13 km to reach town C via town B, which is an improvement on the previous value of 19 km. The diagram now becomes:

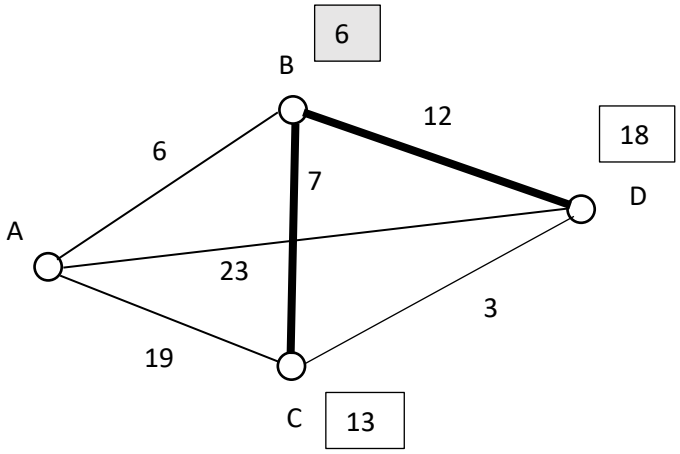


Figure 419: updated distances via town B

The algorithm can now be repeated. We look for the smallest of the temporary distances, which in this case is 13 km for town C. This ***must*** be the shortest distance from the start

point to town C. The only other option we have not yet considered is to reach town C via town D. This could not give a shorter route, as the distance to D is already greater. We can therefore record the shortest distance to town C as **permanent**. We then check for links from town C to any other nodes which still have a temporary distance allocated.

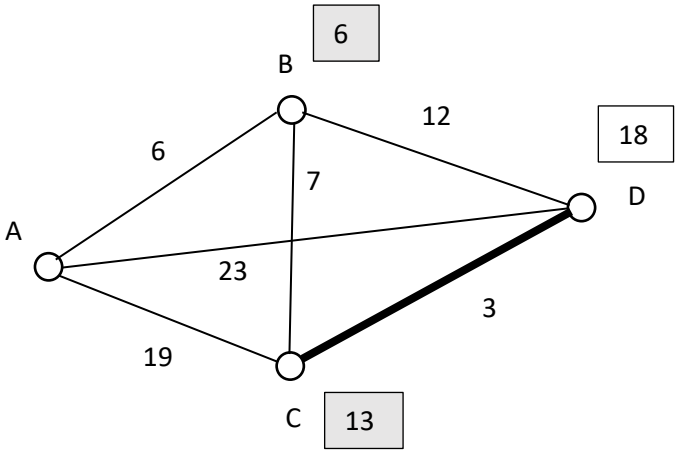


Figure 420: link from town C

Only the link to town D remains to be processed. We calculate that this node could be reached in 16 km via town C, which is an improvement on the previous distance of 18 km. The node can therefore be updated.

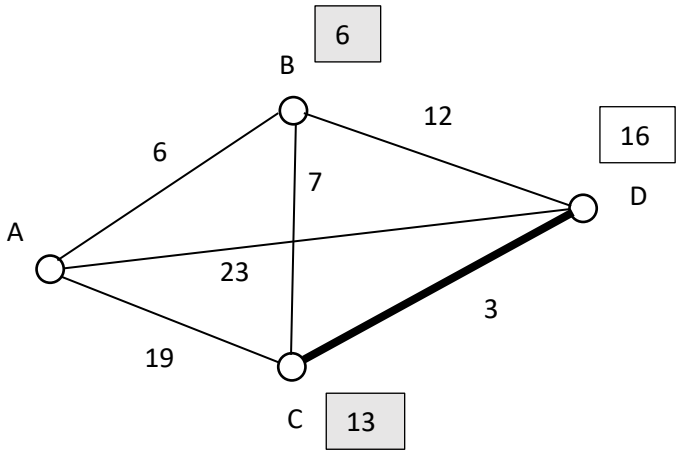


Figure 421: updated distance to town D via town C

Town D is the lowest (and only) remaining temporary node, so the distance of 16 km can be made permanent.

We have now correctly calculated the shortest possible distances from the start point to each of the other nodes in the network, but how can this help with our original task to find the shortest route from town A to town D? The answer is to keep a record of the previous node as each distance is recorded or updated:

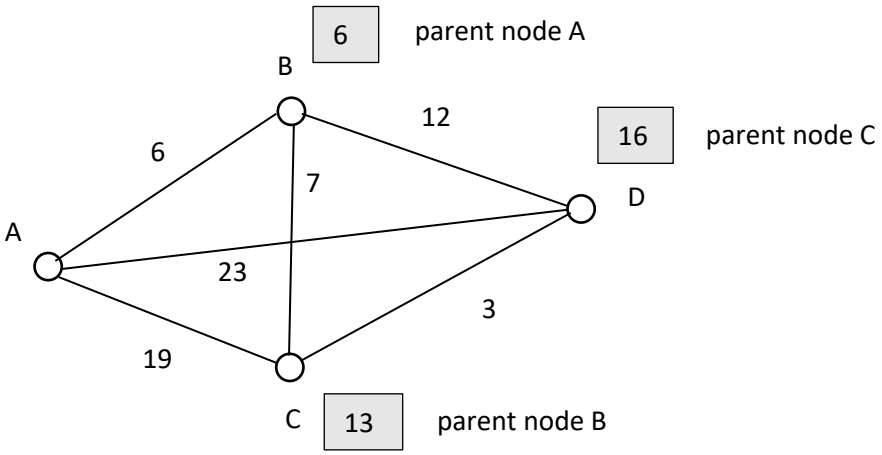


Figure 422: parent nodes for shortest travelling distances to each town

Working backwards from the destination D, we find that this was reached from town C. Town C was reached from town B, and town B was reached from the starting point A. The shortest route has therefore been found:

town A → town B → town C → town D

Let us now consider a more realistic example:

> A student wishes to travel from Aberystwyth to Dublin by the cheapest route. Possible options are:
>
> - Travel by train to Manchester airport. Fly directly to Dublin, or fly to Belfast and complete the journey by train.
> - Travel by train to London airport, then fly to Dublin.
> - Travel by train to Cardiff, then complete the journey to Dublin by train and ferry.

Fares for different sections of the possible routes are shown in figure 423 below.

A computer would use a data table when solving the problem. We begin by listing the nodes of the network. Additional columns are provided in the table for the **cost** of the fare to each node, **status** as temporary, or permanent if we know that this is now the lowest possible fare. A column is also provided to record the **parent** node linking backwards along the route.

NODE	COST	STATUS	PARENT
Aberystwyth	0	permanent	
Belfast	-	temporary	
Cardiff	-	temporary	
Dublin	-	temporary	
London	-	temporary	
Manchester	-	temporary	

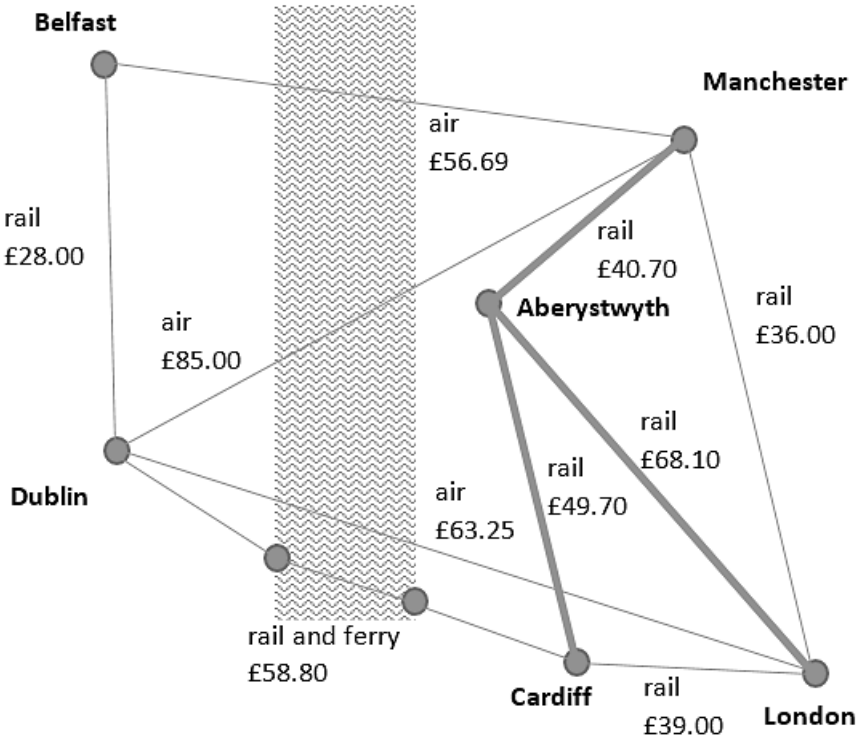


Figure 423: fare options for travel from Aberystwyth to Dublin

We begin at ***Aberystwyth***. This has links to ***Cardiff***, ***London*** and ***Manchester***. We record the fares in each case, and set the ***parent*** node for each of these locations as Aberystwyth.

NODE	COST	STATUS	PARENT
Aberystwyth *	0	permanent	
Belfast	-	temporary	
Cardiff	49.70	temporary	Aberystwyth
Dublin	-	temporary	
London	68.10	temporary	Aberystwyth
Manchester	40.70	temporary	Aberystwyth

We identify ***Manchester*** as the temporary node with the lowest cost of 40.70. This is now set to ***permanent***, and becomes the ***current node***.

NODE	COST	STATUS	PARENT
Aberystwyth	0	permanent	
Belfast	-	temporary	
Cardiff	49.70	temporary	Aberystwyth
Dublin	-	temporary	
London	68.10	temporary	Aberystwyth
Manchester *	40.70	permanent	Aberystwyth

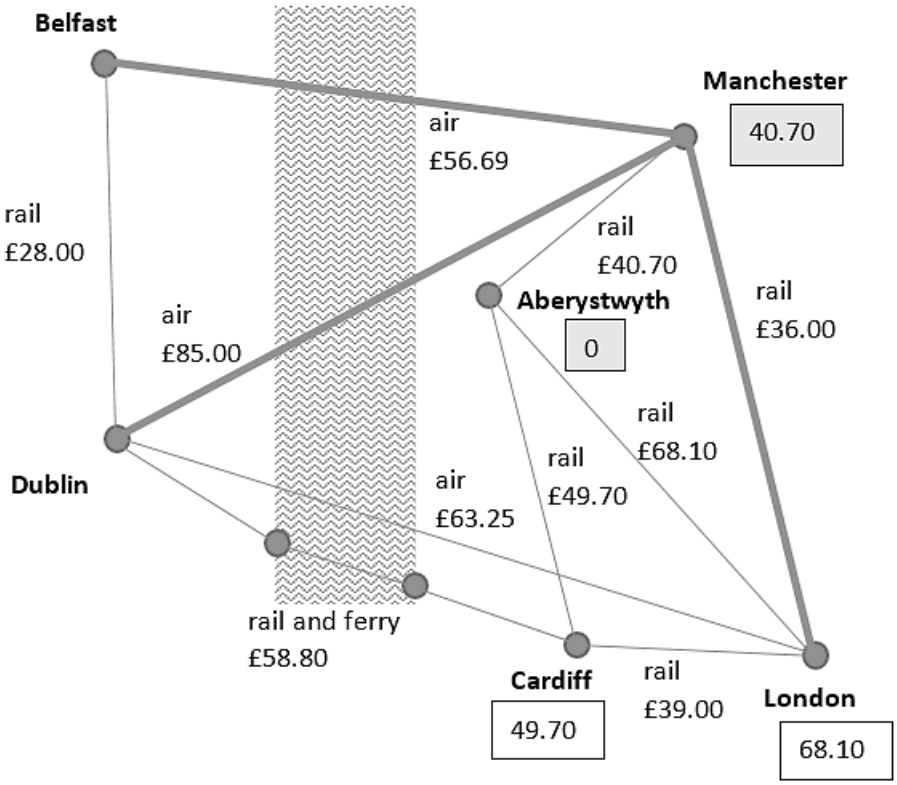


Figure 424: links from Manchester to other temporary nodes

Manchester has links to the temporary nodes ***Belfast, London*** and ***Dublin***. We can calculate the total fares to these towns via Manchester. The fares to Belfast and Dublin can be recorded and their parent nodes set to Manchester. However, the fare to London via Manchester of £76.70 would be more expensive than travelling directly from Aberystwyth.

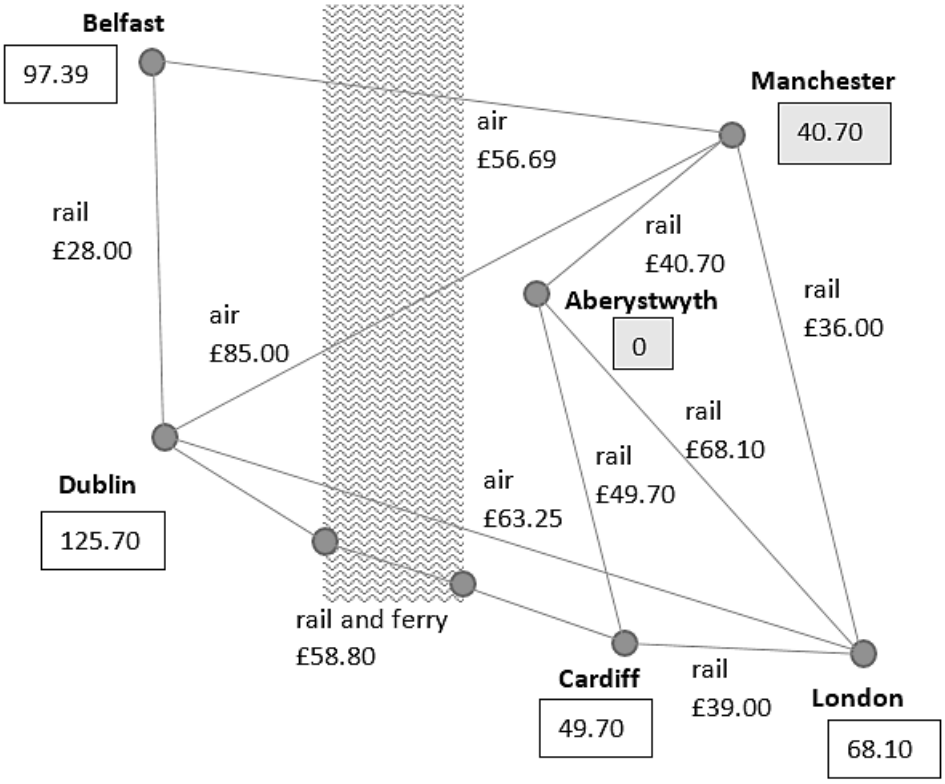


Figure 425: fares updated from Manchester

Now that the Manchester node has been processed, ***Cardiff*** becomes the temporary node with the lowest cost of 49.70. This is now set to ***permanent***, and becomes the ***current node***. Cardiff has links to the temporary nodes ***London*** and ***Dublin***. We calculate the total fares to these towns. The fare to London via Cardiff would be more expensive than the value already shown, so this is ignored. The fare to Dublin is less than the current value of 125.70, so the cost is updated and the ***parent*** is re-set to Cardiff.

NODE	COST	STATUS	PARENT
Aberystwyth	0	permanent	
Belfast	97.39	temporary	Manchester
Cardiff *	49.70	permanent	Aberystwyth
Dublin	108.50	temporary	Cardiff
London	68.10	temporary	Aberystwyth
Manchester	40.70	permanent	Aberystwyth

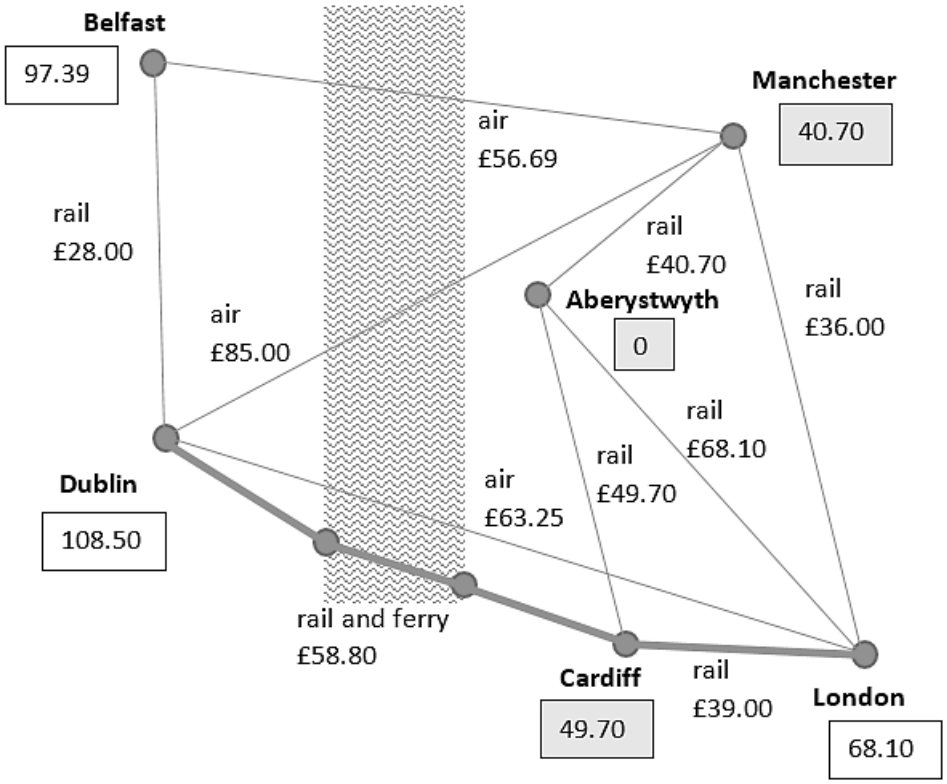


Figure 426: fares updated from Cardiff

We now identify ***London*** as the temporary node with the lowest cost of 68.10. This is set to ***permanent***, and becomes the ***current node***. London has a link to the temporary node ***Dublin***. However, the fare to Dublin via London would be more expensive than the value already shown, so this is ignored.

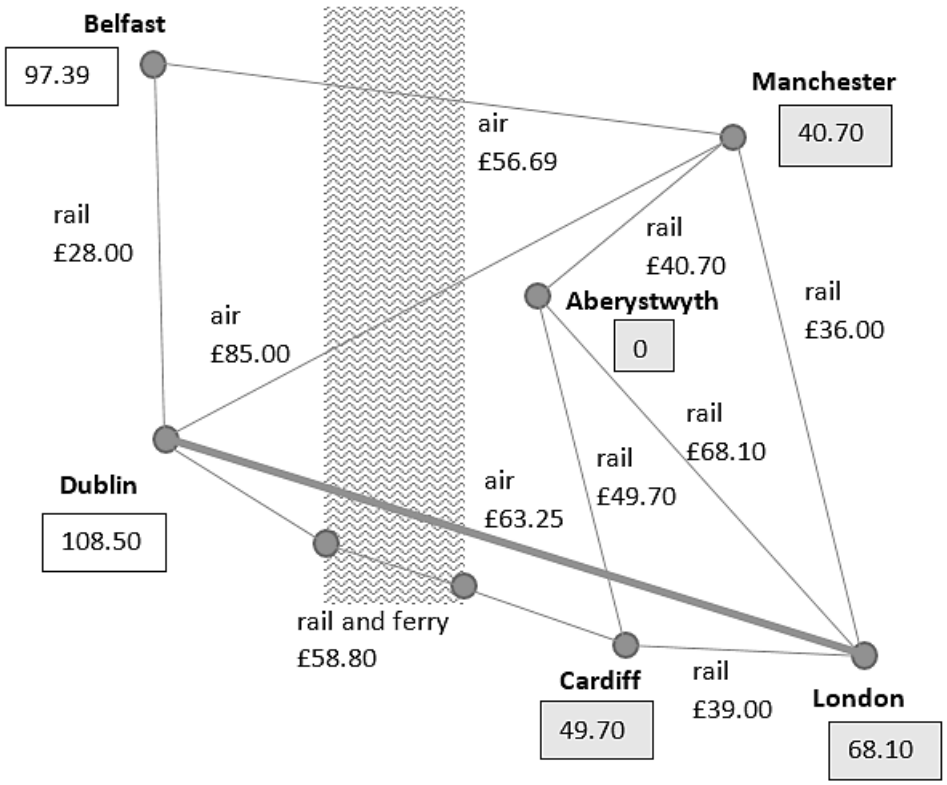


Figure 427: checking for fare updates from London

We now identify ***Belfast*** as the temporary node with the lowest cost of 97.39. This is now set to ***permanent***, and becomes the ***current node***. Belfast has a link to the final temporary node of ***Dublin***. The fare to Dublin via Belfast would be more expensive than the value already shown, so this is ignored.

NODE	COST	STATUS	PARENT
Aberystwyth	0	permanent	
Belfast *	97.39	permanent	Manchester
Cardiff	49.70	permanent	Aberystwyth
Dublin	108.50	temporary	Cardiff
London	68.10	permanent	Aberystwyth
Manchester	40.70	permanent	Aberystwyth

We identify ***Dublin*** as the only remaining temporary node. This is set to ***permanent***, and the algorithm ends. We have now solved the network and have obtained the lowest fares from Aberystwyth to each of the other nodes. Returning to our original task of finding the cheapest route, we can work back from Dublin, using the entries in the parent column to determine each previous node along the route:

Dublin is reached from Cardiff
Cardiff is reached from Aberystwyth.

The student should travel by train to Cardiff, then take the train and ferry connections to Dublin.

Travelling salesman problem

Another common task for route planning software is to find the shortest or quickest way in which a journey can be made around series of points, visiting each point just once and then returning to the start. This is known as the **travelling salesman problem**:

> A business in Dolgellau in North Wales produces craft items. Orders are to be delivered to shops in: Aberystwyth, Caernarfon, Holyhead and Rhyl. Find the best route for the delivery van to take.

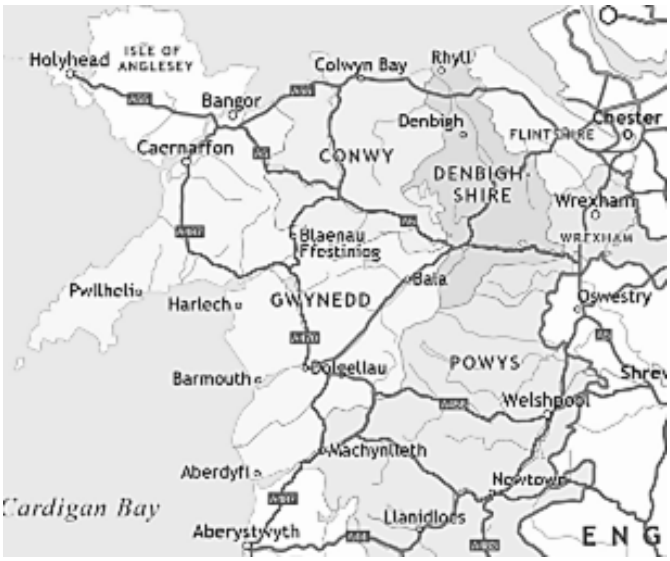


Figure 428:

location map of North Wales

We begin by compiling a table of the distances in km between the delivery locations:

	Dolgellau	Aberystwyth	Caernarfon	Holyhead	Rhyl
Dolgellau		54	70	115	89
Aberystwyth			125	170	139
Caernarfon				46	63
Holyhead					87

A possible route can be found using the **nearest neighbour algorithm**:

> We choose a starting point for the delivery journey. The nearest town is selected, and set as the next point on the route. From the town we have now reached, we then select the nearest of the remaining towns which have not yet been visited and add this to the route. The process continues until every town has been visited, then we return directly to the starting point to complete the circular route.

We choose Dolgellau as the starting point for the route.

From Dolgellau, the nearest node is Aberystwyth at 54 km
From Aberystwyth, the nearest unvisited node is Caernarfon at 125 km
From Caernarfon, the nearest unvisited node is Holyhead at 46 km
From Holyhead, the last unvisited node is Rhyl at 87 km
The return link to Dolgellau is 89 km

This makes a total journey distance of **401 km**. The route is illustrated in figure 429 below:

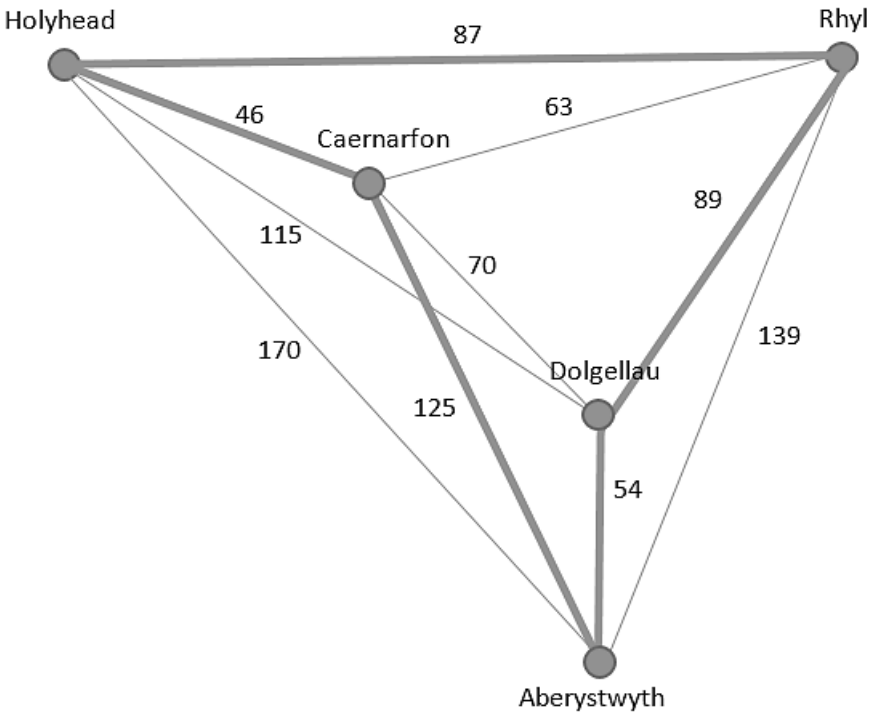


Figure 429: network diagram for the delivery locations

A problem with the nearest neighbour algorithm, in contrast to Dijkstra's algorithm, is that it does not guarantee to find the ***best*** solution to the problem. We can explore this by trying an experiment:

Since every town is visited just once, the shortest route should not depend on where in the loop we start.

If we choose instead to make Aberystwyth the starting point:

From Aberystwyth, the nearest node is Dolgellau at 54 km
From Dolgellau, the nearest unvisited node is Caernarfon at 70 km
From Caernarfon, the nearest unvisited node is Holyhead at 46 km
From Holyhead, the last unvisited node is Rhyl at 87 km
The return link to Aberystwyth is 139 km

This gives a slightly shorter total journey distance of **396 km**. The route is illustrated in figure 430 below.

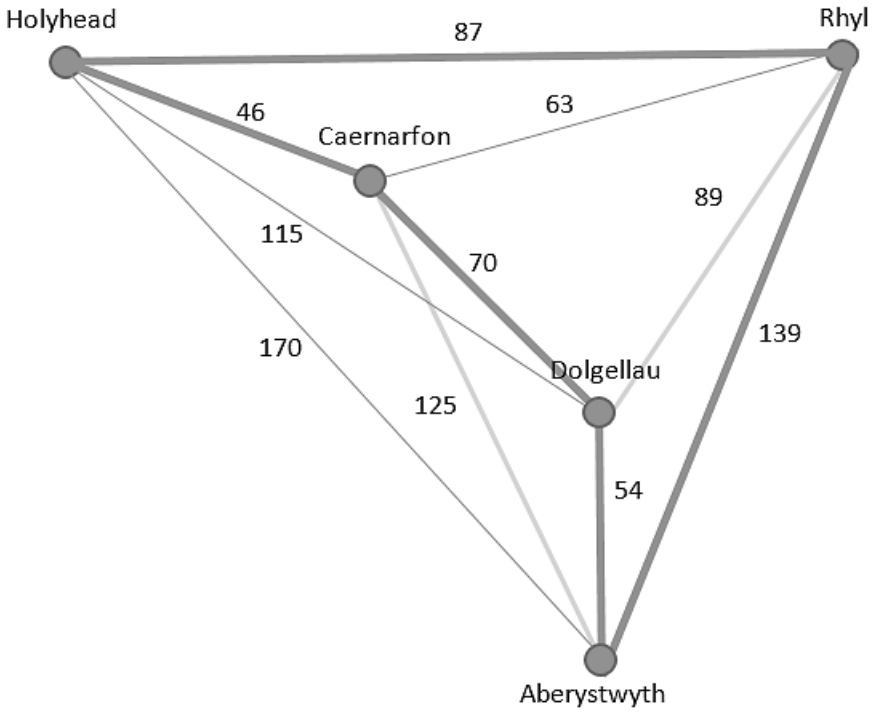


Figure 430: route generated by the nearest neighbour algorithm, starting from Aberystwyth

The uncertainty in the solution can present difficulties, particularly for large and complex networks where many different starting points for a route are possible. We do, however, have a way of determining a lower limit, below which the solution cannot lie. To do this:

> Temporarily remove one of the nodes from the network, for example **Aberystwyth**. We then find the shortest way of linking together the remaining four nodes. This can be done using link distances of 46 km, 63 km and 70 km as shown in figure 431.
>
> Re-attach the missing node for Aberystwyth, and connect it to the network by the two shortest possible links of 54 km and 125 km. as shown in figure 432.

In the first step, we connected all but one of the nodes using the shortest possible distances. A continuous loop around these nodes would have at least this length, and probably be longer. The missing node was then reattached using the two shortest distances possible. This node would have to have two links of at least these lengths in the continuous circuit. We can therefore say with certainty that no true circuit could have a shorter total distance than this set of links. The total length we have obtained is **358 km.**

We are able to say that the shortest possible route around all the points, returning to the start, must be at least **358 km**, and is less than or equal to the distance of **396 km** which we found earlier.

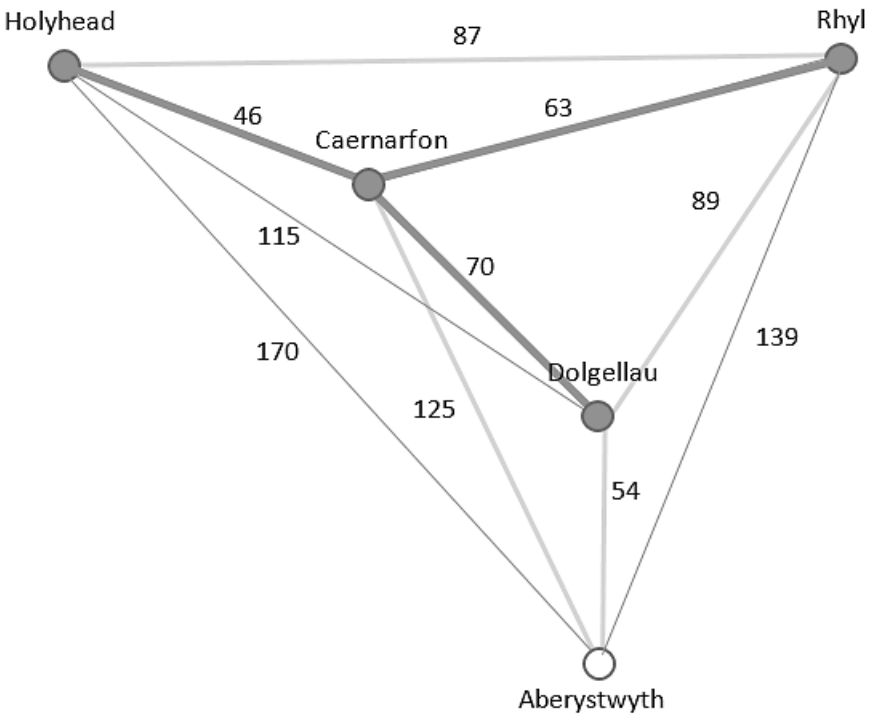


Figure 431: network after removing Aberystwyth and linking the remaining nodes by the shortest distances

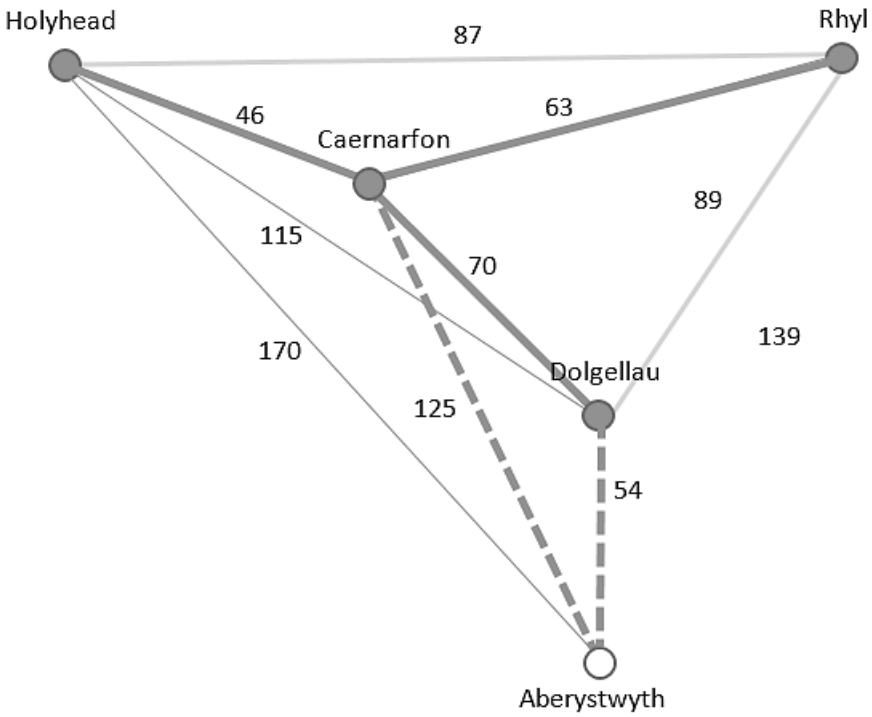


Figure 432: network after replacing and linking the node

Although the nearest neighbour algorithm has given a reasonable solution to this simple problem, difficulties can occur with larger networks. It is not unusual for figure-eight loops to be created, as in figure 433, leading to a greater total journey distance. One solution is to try a variety of different starting points for the circuit, as we did earlier, in the hope of obtaining the shortest route. However, this can be very time consuming. A better option is to use an optimisation algorithm to improve the initial result. A shorter route through the same set of points is shown in figure 434.

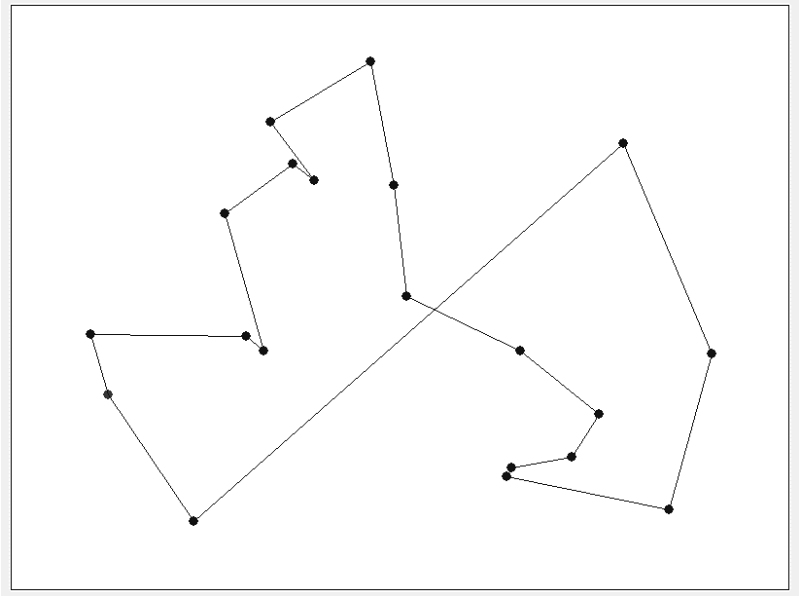


Figure 433:

initial route through a set of points produced by the nearest neighbour algorithm

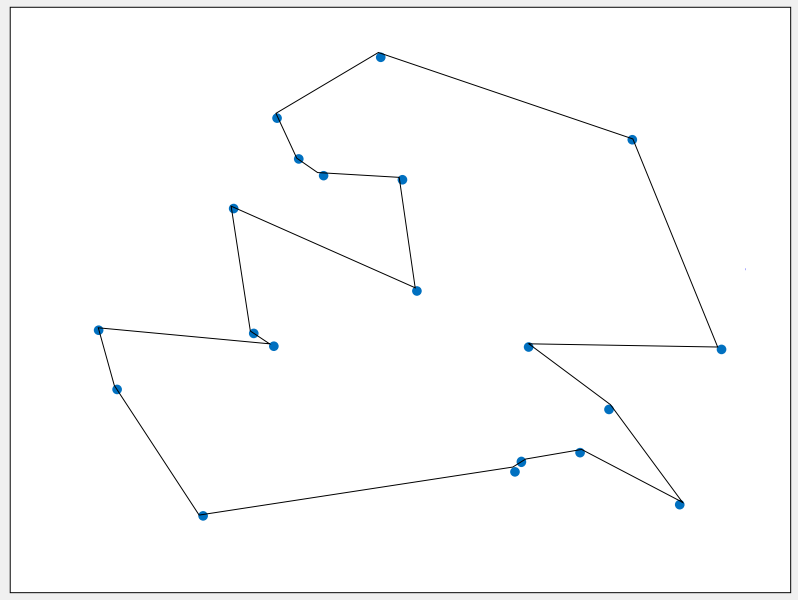


Figure 434:

improved route through the set of points after optimisation

A simple optimisation method is to take each group of three adjacent points along the route, then check the possible set of connections between the points. The shortest set of links is selected.

If three points are labelled A, B and C, then a total of six different sequences are possible:

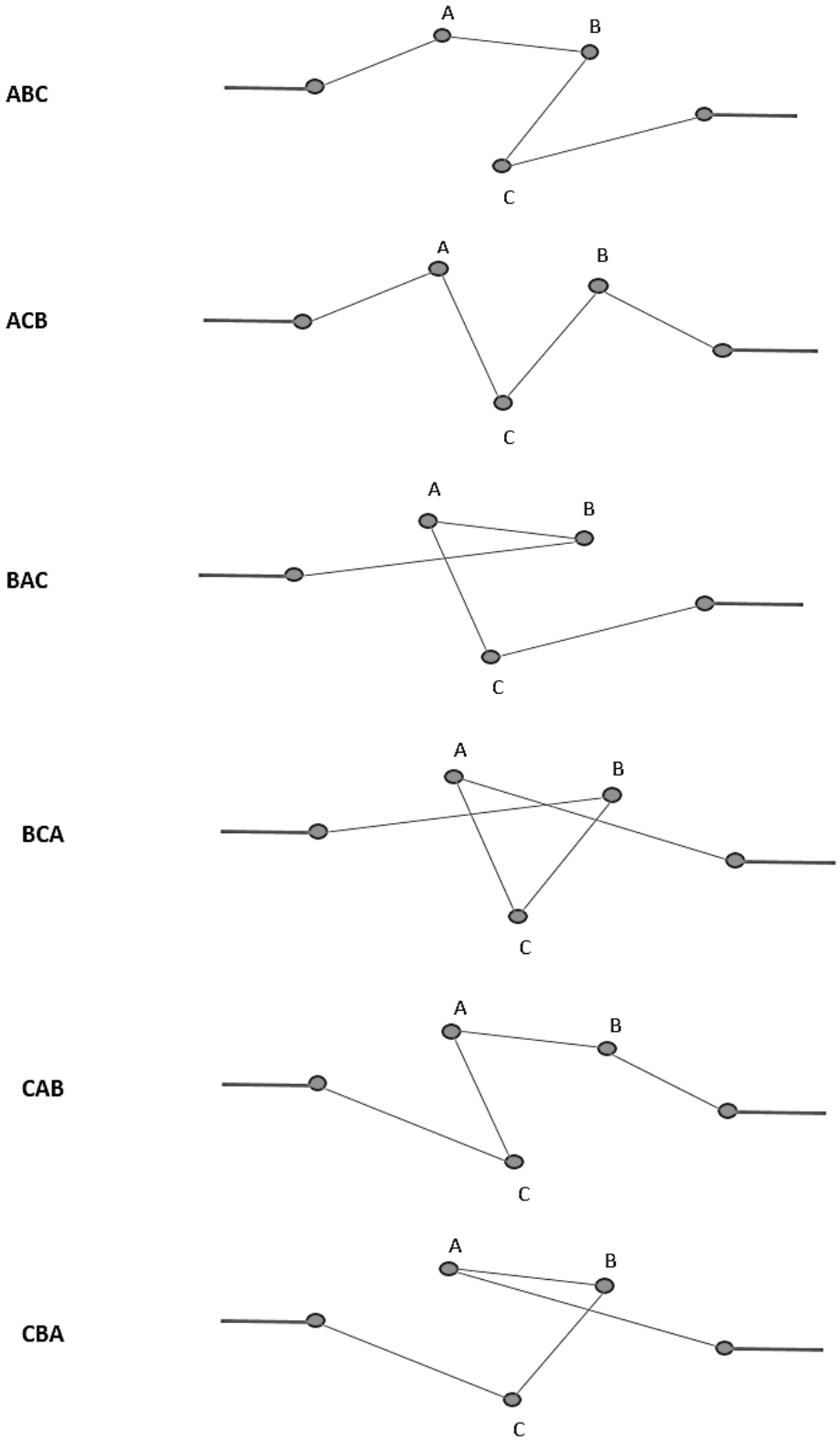


Figure 435: alternative connection sequences for three points A, B and C

To carry out the optimisation procedure, we begin at the first point of the route. We then select the next three points, calling these A, B and C. The different sequences shown in figure 435 above are then tested, and the shortest selected. The sequence of the points in the original route can then be rearranged if necessary.

We then move forward one node, and carry out optimisation on the next three points A, B and C. This procedure is repeated until the route is completed and we have returned to the starting point.

The route may still not be the best we can find. The complete optimisation sequence can be carried out on the route as many times as necessary, until no further improvement in the total distance can be achieved.

Sorting

A common requirement in many types of computer program is to sort data into either alphabetical, numerical or date order. For example: a database of customers might be sorted alphabetically by surname, or a spreadsheet of students' examination results might be sorted into numerical order of the marks awarded.

Whilst general purpose computer software is available for carrying out most routine administrative tasks, it may be necessary for a business or organisation to develop its own specialist software for particular purposes. Programs are very likely to have a requirement to sort data at some stage. The numeracy skills of pattern recognition and problem solving will be important to the programmers who develop the sorting algorithms for this software.

A simple sorting method is the **bubble sort**. To demonstrate how this works, we will take the series of words:

goat rat pig fox cow owl

and attempt to sort them into alphabetical order. We move down the list, comparing each pair of words in turn. If the order of any word pair is incorrect, the words are exchanged.

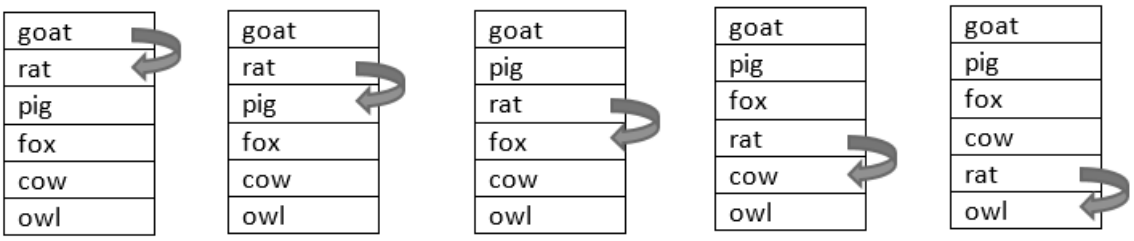


Figure 436: first pass through a list of words during a bubble sort

Examining the sequence in figure 436 above, we see that:

goat and **rat** are already in correct alphabetical order
rat and **pig** are in incorrect order so they are exchanged
rat and **fox** are in incorrect order so they are exchanged…

At the end of the first pass through all the data, we have the sequence:

goat
pig
fox
cow
owl
rat

This is not yet the correct alphabetical sequence, but words nearer the start of the sequence have 'bubbled up' towards the top of the list – hence the name **bubble sort**.

We use the modified sequence as the starting point for another pass through the data:

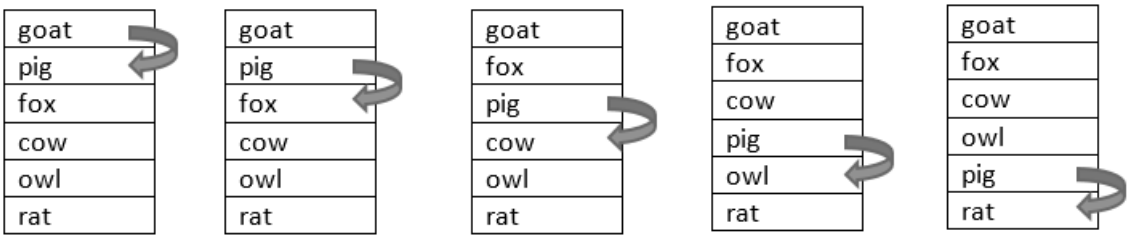


Figure 437: second pass through the list of words during the bubble sort

We now have the sequence:

goat
fox
cow
owl
pig
rat

This is still not in correct alphabetical order, so another pass is carried out:

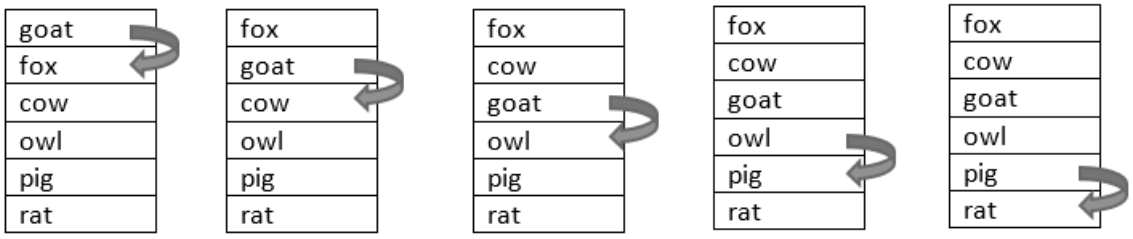


Figure 438: third pass through the list of words during the bubble sort

The list is still not quite in the correct order. One further pass through the data will be needed to bring **cow** to the top of the list. A large number of comparisons have been necessary and the algorithm is not particularly efficient. If many items have to be sorted by this method, the program would be slow. Better sorting methods are needed for large data sets.

An improved sorting method, the **quicksort algorithm**, makes use of the technique of **recursion**.

Recursion involves carrying out a version of a task *within the task itself*. A simple example is to produce a geometric pattern by recursion.

> We begin by drawing a square.
> Four smaller squares are then drawn with their centres at each of the corners of the larger square.
> Further smaller squares can then be added at the corners of each of these small squares, producing a pattern of any chosen depth.

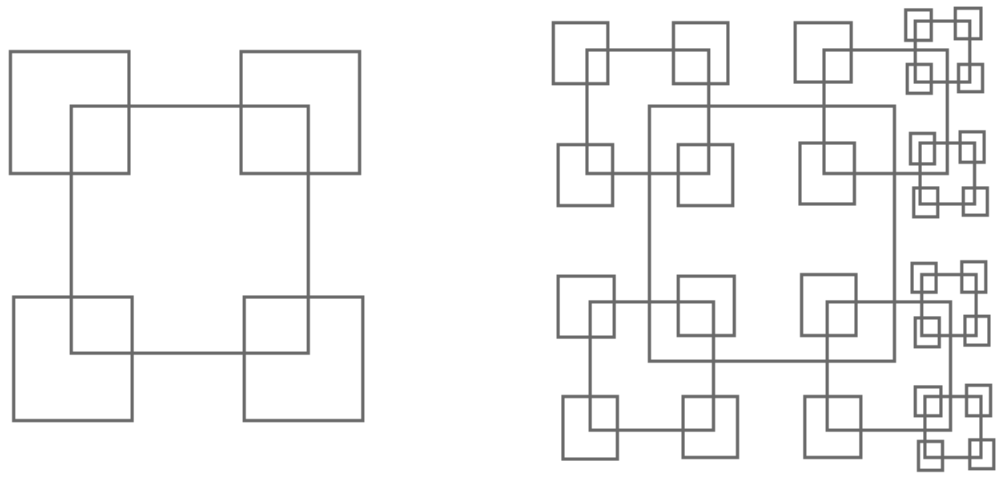


Figure 439: developing a recursive geometric pattern

The **quicksort algorithm** begins with an unsorted set of data. One data item, often the first in the sequence, is selected:

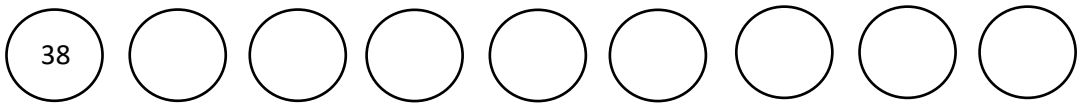


Figure 440: pivot selected as the first item in the unsorted list

The program then compares the reference item, known as the **pivot**, to each of the other items in the data set. All items with lower values are moved in front of the pivot, whilst all items with higher values remain after the pivot.

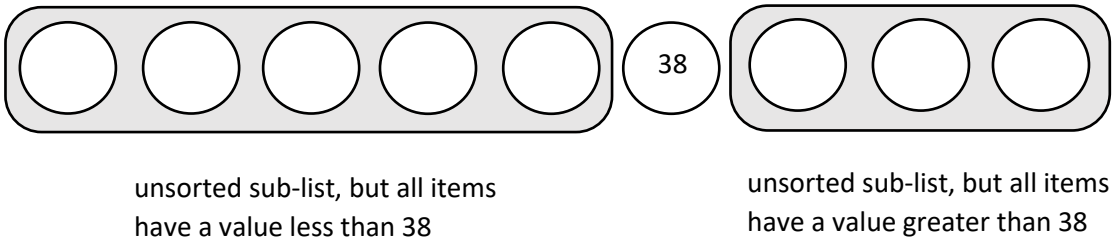


Figure 441: division of the original list into two sub-lists

We have now divided the original list into two unsorted sub-lists, separated by the pivot item. We know that the pivot is now in its correct position, as any rearrangement which is necessary in the sub-lists will not affect the pivot.

The sorting procedure is now repeated **recursively** on each of the sub-lists. New pivot values are used to make comparisons to create further sub-lists:

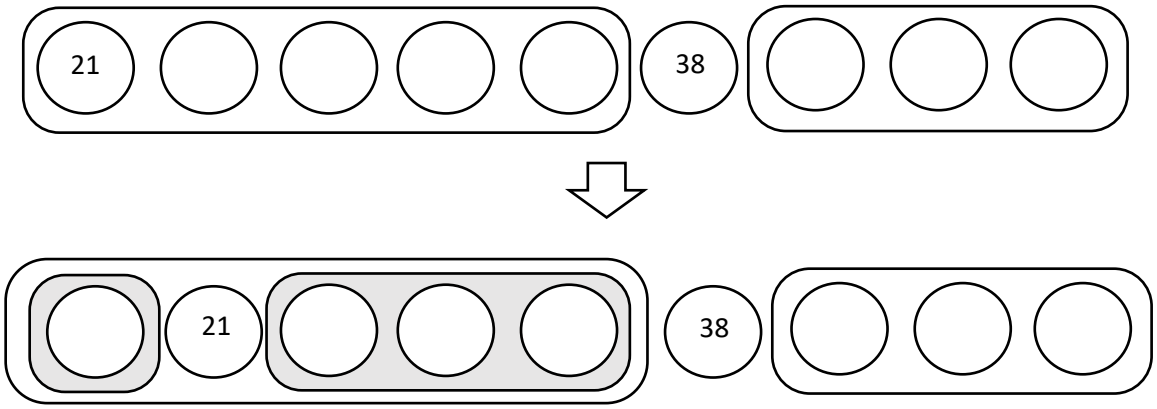


Figure 442: further sorting of a sub-list by recursion

Recursion continues until all sub-lists have been reduced to single items. At this point, the set of data will be fully sorted.

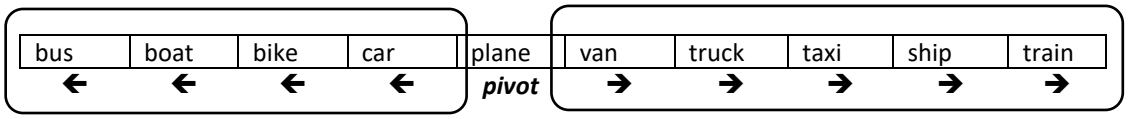
To demonstrate the quicksort algorithm in action, consider the series of random words for methods of transport. These words need to be sorted into alphabetical order.

plane	van	truck	bus	boat	taxi	ship	bike	car	train

We will select the word '***plane***' at the start of the sequence as the ***pivot***. We then decide whether each of the other words would come before (🡐) or after (🡒) the pivot value in alphabetical order.

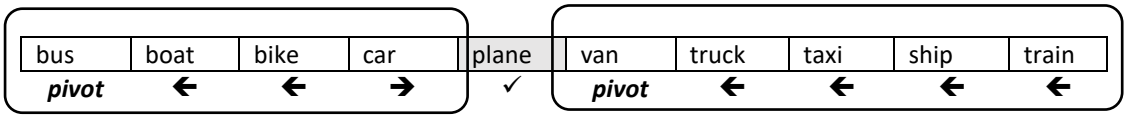
plane	van	truck	bus	boat	taxi	ship	bike	car	train
pivot	🡒	🡒	🡐	🡐	🡒	🡒	🡐	🡐	🡒

We make a new copy of the data, putting all the items before the pivot value on the left, followed by the pivot itself, then all the items after the pivot on the right.



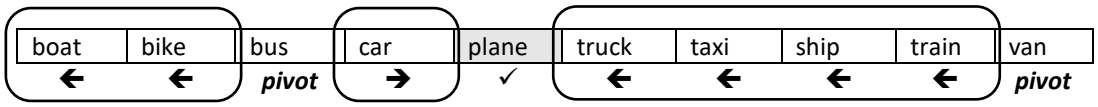
Although the groups of data to the left and right of the pivot are not yet sorted, the pivot itself must be in the correct position in the sequence. Any further sorting cannot alter its position.

The process is now repeated for the groups of unsorted data to the left and right of the pivot. The new comparison values will be '***bus***' and '***van***'.



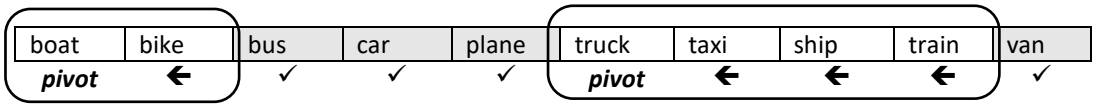
Within each unsorted group, we decide whether each word comes before or after the pivot value.

The words in each unsorted group are again rearranged, so that words before the pivot are listed first, then the pivot itself, then the words which come after the pivot value in alphabetical order.

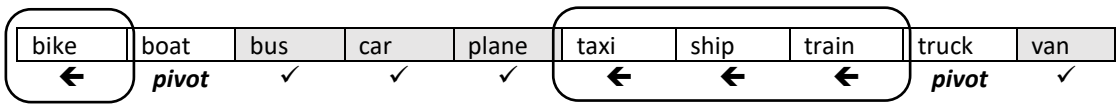


The pivot values '***bus***' and '***van***' are now in correct positions in the sequence. However, '***car***' must also be in a correct position as it is a single item which cannot be exchanged with any other word.

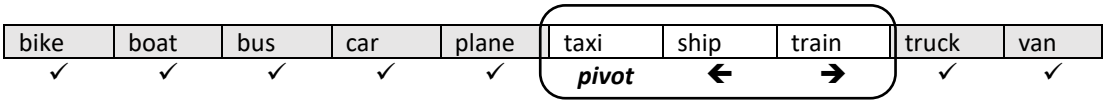
The sort continues by creating pivot comparison values for the remaining unsorted groups.



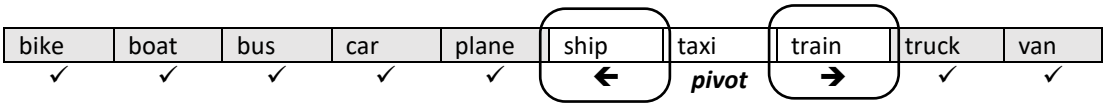
Rearranging the remaining data items gives:



The pivot items '***boat***' and '***truck***' are now in correct positions. The single item '***bike'*** must also be correct. Sorting will continue with one remaining group of items.



The final rearrangement completes the sorting.



We can see that versions of the sorting method are taking place ***within itself***, so the problem can be solved by recursion.

It was stated earlier that the quicksort algorithm is a faster and more efficient sorting method than the bubble sort algorithm. Let us consider why this is the case.

The efficiency, or ***order***, of algorithms is often specified by means of **big-O notation**. This is a way of describing the effect on the sorting time if the amount of data is increased.

Let us first examine the **bubble sort** algorithm. Without worrying too much about the exact sorting mechanism, we can identify two general steps which are required to put the data items into their correct position in a list:

- each item of data must be selected
- the item of data must be compared to each of the other data items

Each of these processes will take a certain amount of processing time, and each will depend on the number of data items **n** which are present. If the number of data items is doubled, then both processes will take twice as long. The overall effect will be to make the sorting four times slower. We say that the order of the sorting method is:

$$O(n^2)$$

which means that the overall time for the algorithm is proportional to the square of the number of data items.

Turning now to the **quicksort**, we can again say that two processes are involved:

- each item of data must be selected
- the item of data must be compared to each of the other items in the sub-list

However, increasing the number of data items makes only a small change to the average size of the sub–lists. Doubling the number of data items only adds one larger list at the top level of the recursion, with all the sub-lists below this remaining the same size. Increasing

the amount of data by 256 times would only add eight extra levels to the recursion tree, so there would be little change to the average length of the sub-lists which are being sorted.

It is found that the order of the quicksort algorithm is:

$$O(n \log n)$$

The sorting time will be proportional to the number of data items **n** multiplied by the logarithm of this number.

We can see why a quicksort is faster than a bubble sort by plotting graphs of the two functions n^2 and n log n:

N	N squared	N ln N
2	4	1.38629436
3	9	3.29583687
4	16	5.54517744
5	25	8.04718956
6	36	10.7505568
7	49	13.621371
8	64	16.6355323
9	81	19.7750212
10	100	23.0258509
11	121	26.376848
12	144	29.8188798
13	169	33.3443416
14	196	36.9468026
15	225	40.620753
16	256	44.3614196
17	289	48.1646268
18	324	52.0266916
19	361	55.9443406
20	400	59.9146455
21	441	63.9349712
22	484	68.002934
23	529	72.116367

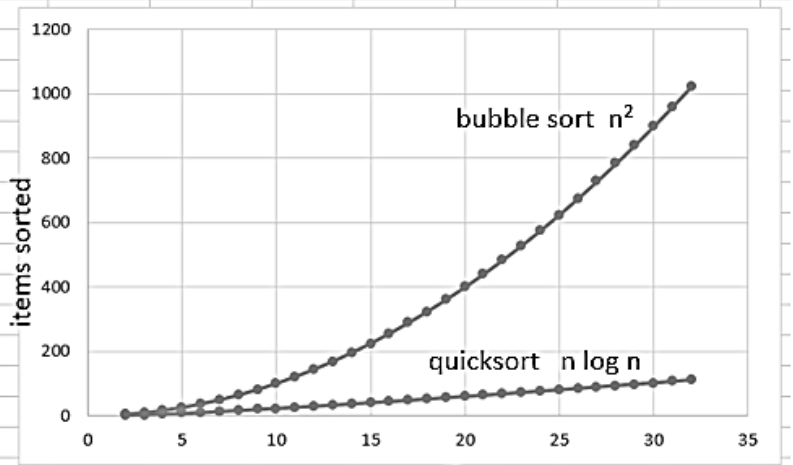


Figure 443: graphs of n^2 and n log n

Whilst the sorting time for ***quicksort*** increases in an almost linear manner as the number of data items is increased, the sort time for the ***bubble sort*** increases as a steepening upwards curve. For large amounts of data, there will be a major advantage in using the quicksort method. However, with small amounts of data there is little difference in speed, and programmers may prefer to use the bubble sort algorithm which is simpler to program.

Encryption

It is often important to encrypt computer data, for example when it is sent over a network and might be intercepted by an unauthorised person. An important area of mathematics focuses on the development of secure and efficient methods of data encryption, and often involves the development of complex algorithms.

The data transmitted over a network or stored on digital media is usually a series of small numbers representing letters of the alphabet, digits or other keyboard characters. A system in common use is the American Standard Code for Information Interchange (**ASCII**) which represents the characters on a standard English language keyboard by numbers in the range from 0 to 255. In this system, character 'A' has the value 65, 'B' is 66, 'C' is 67, and so on…

Cyphers work on the principle of inverse functions. We are familiar with the idea of some mathematical operations being the reverse of others. For example, we can reverse an addition by means of a subtraction:

$$65 + 6 = 71 \qquad\qquad 71 - 6 = 65$$

An ASCII code 65 representing the letter A could be encrypted by adding 6, so that the code transmitted is 71, representing the letter G. All other characters in the message could be encrypted in the same way. For example, the message:

HELLO

becomes

NKRRU

If the encryption **key** value of 6 is known, the message can be decrypted by subtracting this from the ASCII code values received. This method of moving letters of the alphabet forward by a set number of places was used in Roman times, and is known as the **Caesar Cypher**. However, the cypher can easily be broken by analysing the frequency of letters in the message: the characters most frequently occurring in the encrypted message are likely to be common letters such as E, T or A.

More complex methods of encryption have been developed, such as the Enigma Cypher used during World War 2 which relied on machines to encrypt the code with a complex mathematical function, then decrypt by means of the inverse function. Although much more secure, a difficulty still existed with this system; details of the key settings had to be transferred in advance from the sender to the receiver so that the necessary information was available to decrypt the message. This could present a security risk if the key fell into the wrong hands.

A major breakthrough occurred when methods were discovered for **double key encryption**. In this system, one key is used to encrypt the message, then a different key is used for decryption. Even if the encryption key is known, the decryption key cannot be calculated and the message remains secure.

As an example, consider a customer buying goods from an on-line store, then paying by credit card. It is important that the customer's card details are secure when transmitted over the Internet to the store. Single key encryption would be unsuitable, as sending the key to the customer would pose a security risk. If the key was intercepted by a criminal, this could subsequently be used to decrypt the credit card details.

In the double key system, a **pair** of keys is created. One, termed the **public key**, is made freely available as part of the web page of the store. The web page uses this to encrypt the bank details when the customer sends their order. Knowledge of the public key does not make it possible to decrypt the message. This can only be carried out using the other member of the key pair, the **private key**, which is held securely in the store.

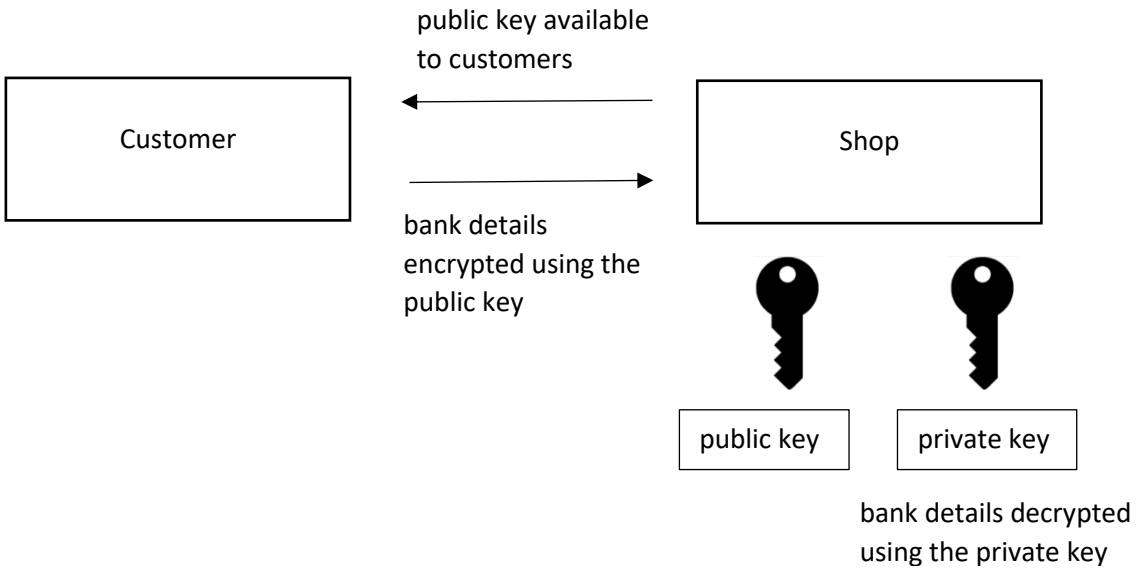


Figure 444: use of a double key encryption system

The success of the double key method depends on finding a mathematical function whose inverse cannot be determined from the available information in the public key. The function and inverse chosen were rather unusual ones. Encryption is carried out using a **modulus** function. To do this:

The unencrypted value, such as the ASCII code 65 for 'A', is given the name **m**

The encrypted value, which will be sent securely over the Internet, is called **c**

Two numbers, called **e** and **n**, are used during the encryption process.

The value **m** is then encrypted to produce **c** using the equation:

$$(m^e)\ MOD\ n = c$$

The value **m** is raised to the power **e**, then the remainder found when the number **n** is subtracted as many times as possible.

For example, suppose that the ASCII code 65 is to be transmitted. Any suitable values could be selected for the numbers **e** and **n** which will encrypt the data.

We will choose e = 3 and n = 143.

Carrying out the steps of the encryption:

$$(65^3)\ MOD\ n = c$$

$$493039\ MOD\ n = c$$

$$493039\ MOD\ 143 = 118$$

65 cubed is 493039. Subtracting 143 repeatedly from 493039 leaves a remainder of 118. This is the encrypted value which will be transmitted.

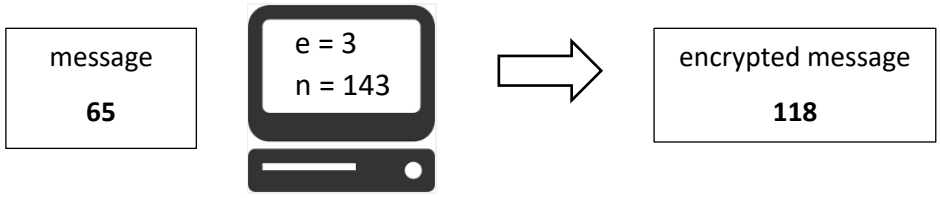


Figure 445: encryption using key values **e** and **n**

We now need an inverse function which can convert 118 back to the original value. This is:

$$\left(c^d\right)\ MOD\ n = m$$

where **c** is the encrypted value and **m** is the original message. **n** is the same modulus value which was used during the encryption.

We also need another value **d**. For reasons which will be explained shortly, a suitable value for d is 221.

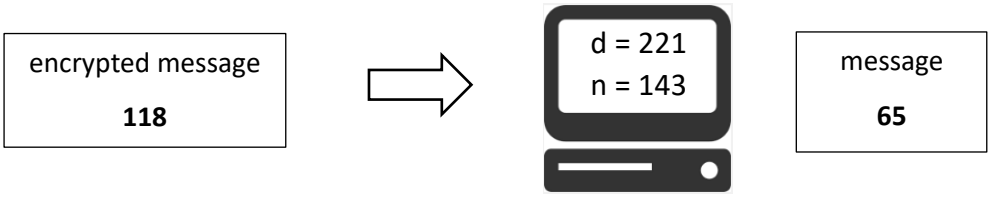


Figure 446: decryption using key values **d** and **n**

To operate the double key encryption system, the customer in the shop example would need to know the encryption values **e** and **n**. These can be made publicly available. However, the message cannot be decrypted unless the value of **d** is also known. This would be held securely on the shop's computer system and not revealed to unauthorised persons.

The security of the double key encryption method depends on it being impossible, or very difficult within a reasonable time scale, to find the value of **d** when only **e** and **n** are known.

To operate double key encryption, we need a way of generating a set of values **e**, **n** and **d** which can be used in the public and private keys. In practice, **n** is chosen to be a very large number which is produced by multiplying together two large prime numbers which we will call **p** and **q**. For example, we might choose:

$$p = 1889, \quad q = 3547$$

$$n = 1889 \times 3547 = 6{,}700{,}283$$

Only knowing the product **n** makes it very difficult to find **p** and **q**. Real systems use even larger values so the task becomes effectively impossible. For example the value produced after multiplying two large prime numbers might be:

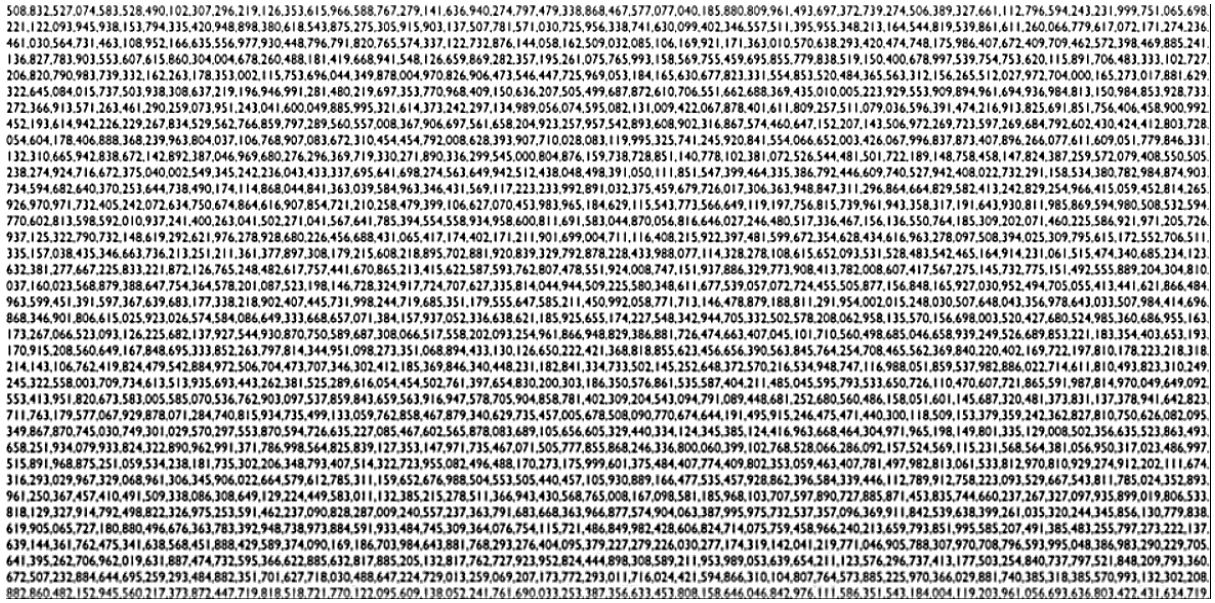


Figure 447: example of a secure large encryption value produced by multiplying prime numbers (Vance, 2014)

It would take a long time to find the two prime numbers which multiply to give this result! The significance is that the decryption value **d** can only be found if we know the two prime numbers **p** and **q** which were used to calculate **n**.

The method used to find **d** from **p** and **q** is the **Extended Euclidean Algorithm**. The procedure is quite complex, but can be illustrated by working through an example:

- We begin by choosing two very large prime numbers p and q. To keep things simple in this example, we will set:

  p = 5, q = 11

- We now calculate a value **φ** known as the **Euler totient function**. This is done by subtracting 1 from each of p and q, then multiplying the result:

$$\varphi = (5-1)(11-1) = 40$$

- We now choose a value for the encryption key **e**. It is important that this does not share a common factor with **φ.** A value of 7 would be suitable for **e**, since 7 does not divide into 40.
- The other encryption value **n** is the product of p and q:

$$n = 5 \times 11 = 55$$

It now just remains to find the decryption key **d** using the Extended Euclidean Algorithm. Although none of the steps of the algorithm involves difficult mathematics, the sequence can be difficult to remember. In practice, the key values would be generated by a computer program which has been set up to carry out the correct sequence of steps.

We begin by writing the value of **φ** at the top of two columns. Beneath this in the first column is the value of **e**, and in the second column is written 1:

40	40
7	1

We now divide **φ** by **e**, writing just the whole number result and ignoring any remainder.

40	40
7	1
5	

We multiply this result by each of the second row values:

40	40
7	1
5 * 7 = 35	5 * 1 = 5

These totals are then subtracted from the values on the top row:

40	40
7	1
(40 – 35) = 5	(40 – 5) = 35

These results now form the third row of the table:

40	40
7	1
5	35

The first row of the table can now be discarded:

7	1
5	35

The steps carried out above are now repeated:

We divide the left column value of the first row by the left column value of the second row, writing the whole number result and ignoring any remainder.

7	1
5	35
1	

We multiply this result by each of the second row values:

7	1
5	35
1 * 5 = 5	1 * 35 = 35

These totals are then subtracted from the values on the top row:

7	1
5	35
(7 – 5) = 2	(1-35) = -34

Only positive results can be accepted. The negative value of -34 is converted to a positive value by adding **φ**:

-34 + 40 = 6

These results now form the third row of the table:

7	1
5	35
2	6

The first row of the table can again be discarded:

5	35
2	6

We need to carry out the sequence of steps once more:

We divide the left column value of the first row by the left column value of the second row , writing the whole number result and ignoring any remainder.

5	35
2	6
2	

We multiply this result by each of the second row values:

5	35
2	6
2 * 2 = 4	2 * 6 = 12

These totals are then subtracted from the values on the top row:

5	35
2	6
(5 – 4) = 1	(35-12) = 23

These results now form the third row of the table:

7	1
5	35
1	23

The left hand column of the final row now contains a value of 1, which means that a solution has been found. The required value for the decryption code d is the corresponding value in the right hand column. We therefore have:

$$d = 23$$

All the values required to operate the double key encryption system are now available for use:

$$n = 55,\ e = 7, \text{ and } d = 23$$

n and e together form the public key, whilst n and d together form the private key.

Game strategy

Algorithms are important in planning strategies for success in games. Apart from a role in games developed for recreation, algorithms can be important in modelling complex real world scenarios such as economic crises or military conflicts. For many of these applications, pattern recognition and the estimation of probabilities are important.

Noughts and crosses

We begin with the simple and familiar game of Noughts and Crosses. This is played on a grid of three rows of three squares. Two players, designated as nought 'O' or cross 'X', take turns to add their symbols to the board. The objective is to achieve a line of three noughts or crosses, either horizontally, vertically or diagonally.

As an interesting challenge, computing students can produce a version of the game where one player competes against the computer. To achieve this, the computer must be programmed with a game strategy.

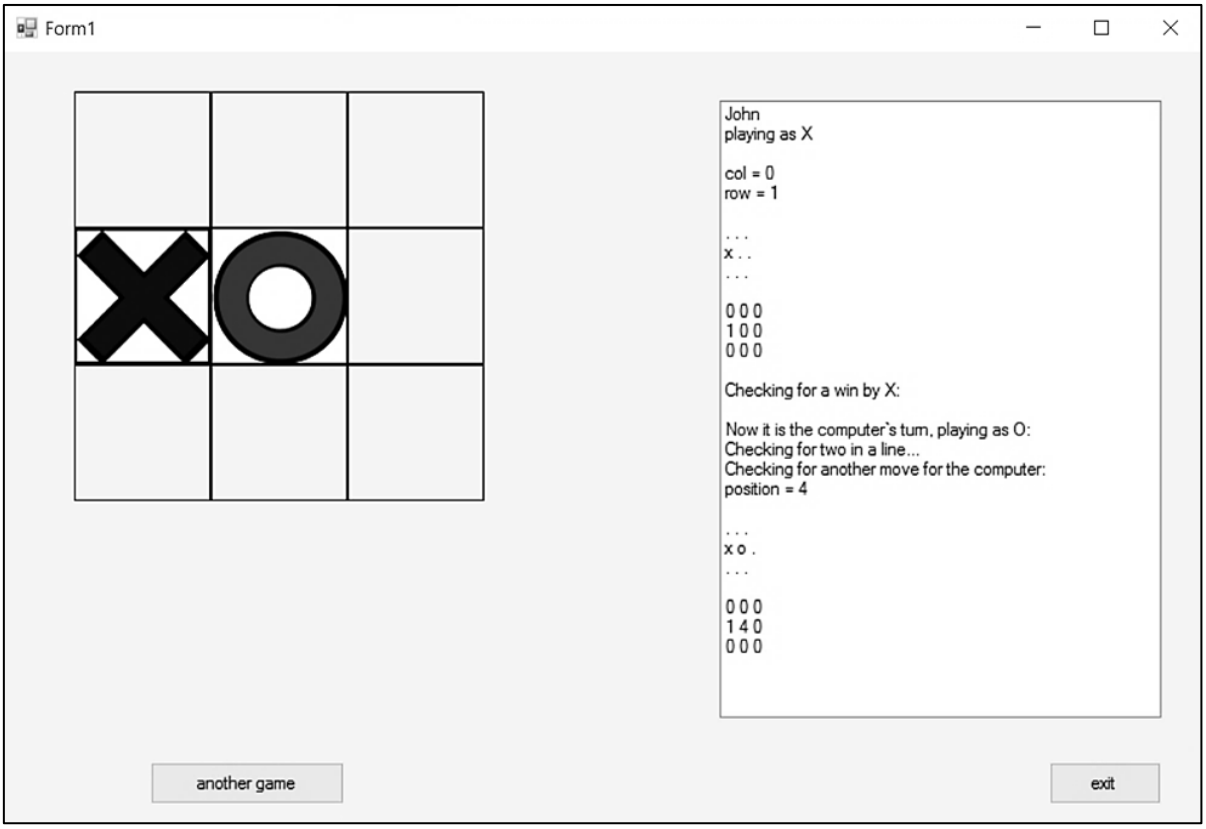


Figure 448: noughts and crosses game, including comments to explain the computer strategy

The player is designated as X, and the computer as O.

After each move by the player, the computer takes over control and can make a move. It must, however, first check whether the player has just completed a line of three Xs and has won the game.

If the game is continuing, the computer next checks whether there is a line with two Os and a blank space. If so, the computer can complete the final O and win the game.

If the game is still continuing, the computer must try to make a move. We can see that not all the squares on the board are of equal importance in terms of strategy. The centre square can become part of the largest number of winning lines:

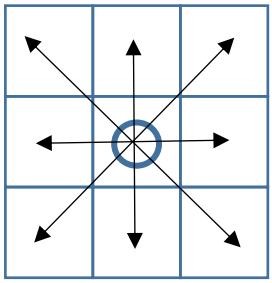


Figure 449:

winning lines passing through the centre square

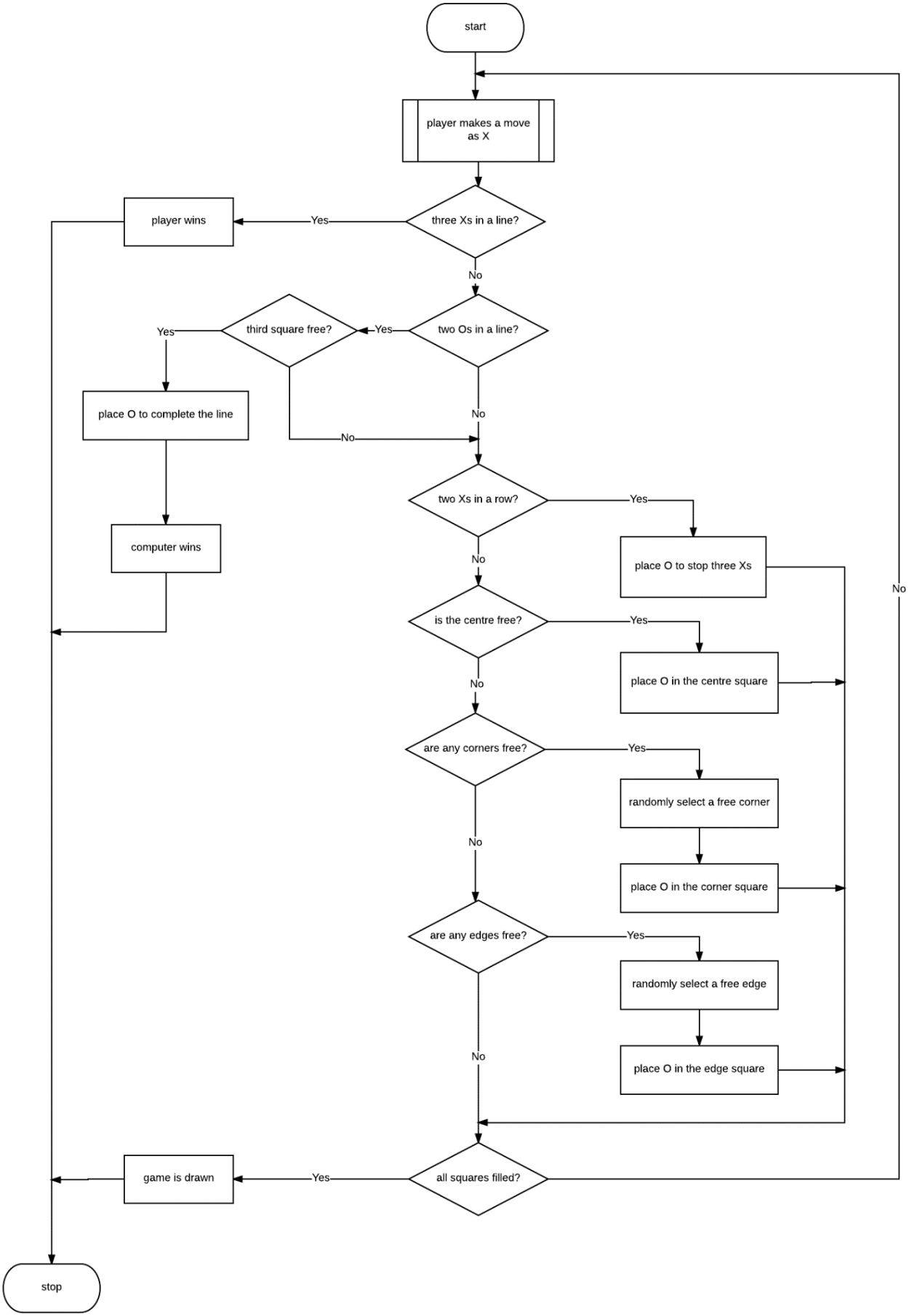


Figure 450: flowchart for the noughts and crosses computer game

There is an advantage to the computer to select the centre square if this is still available.

If the centre square is already occupied, then the next best strategy is to select a corner square if available.

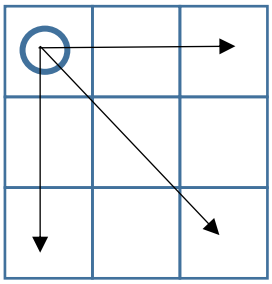


Figure 451:

winning lines passing through a corner square

If more than one corner square is empty, the computer chooses the corner randomly. This introduces an element of chance and allows the player some opportunity to beat the computer.

If all corner squares are already filled, then the final option is to select an edge square if available.

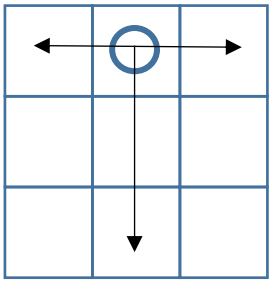


Figure 452:

winning lines passing through an edge square

If more than one edge square is empty, the computer again chooses randomly.

Once the computer has made its move, control returns to the player. However, if there are no squares still free on the board then the game will have ended in a draw.

Solitaire

Our next example is the game of solitaire. This is played on a board containing a cross-shaped pattern of holes into which pegs can be inserted. At the start of the game, all the holes, with the exception of the central hole, contain pegs (figure 453).

Solitaire is played by a single player. The objective is to remove all but one of the pegs from the board, leaving the last peg in the central hole. A peg can be removed by another peg jumping over it to reach an empty hole. Moves can take place horizontally or vertically, but not diagonally.

In this section, we give an algorithm for completing the solitaire puzzle. Computing students can be set the challenge of producing an automated solution sequence. An animation can be created using graphics drawn by the program.

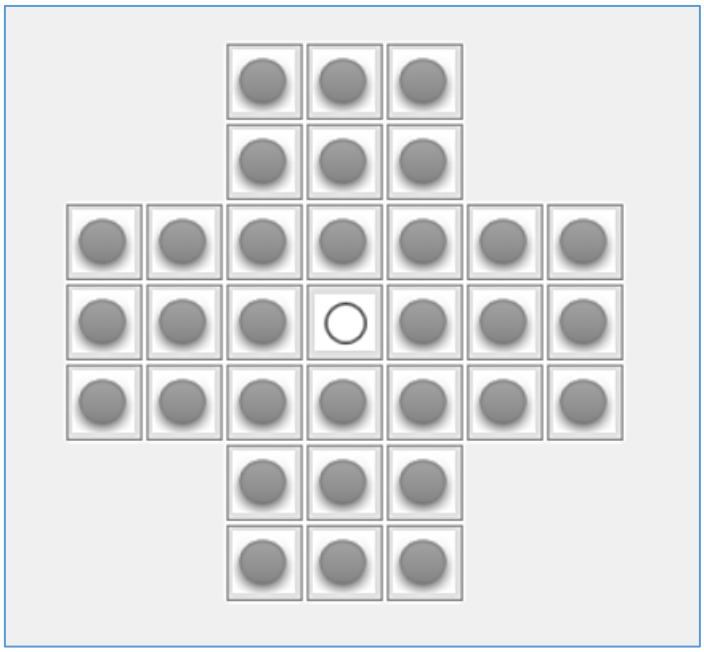


Figure 453:

Solitaire board at the start of the game

To write the game algorithm, a system will be required for specifying the moves. We begin by creating a coordinate system for the holes. The board is divided into five areas called: North, South, East, West and Centre, as shown in figure 454. Within each of these areas, individual hole positions are identified by a horizontal coordinate, followed by a vertical coordinate.

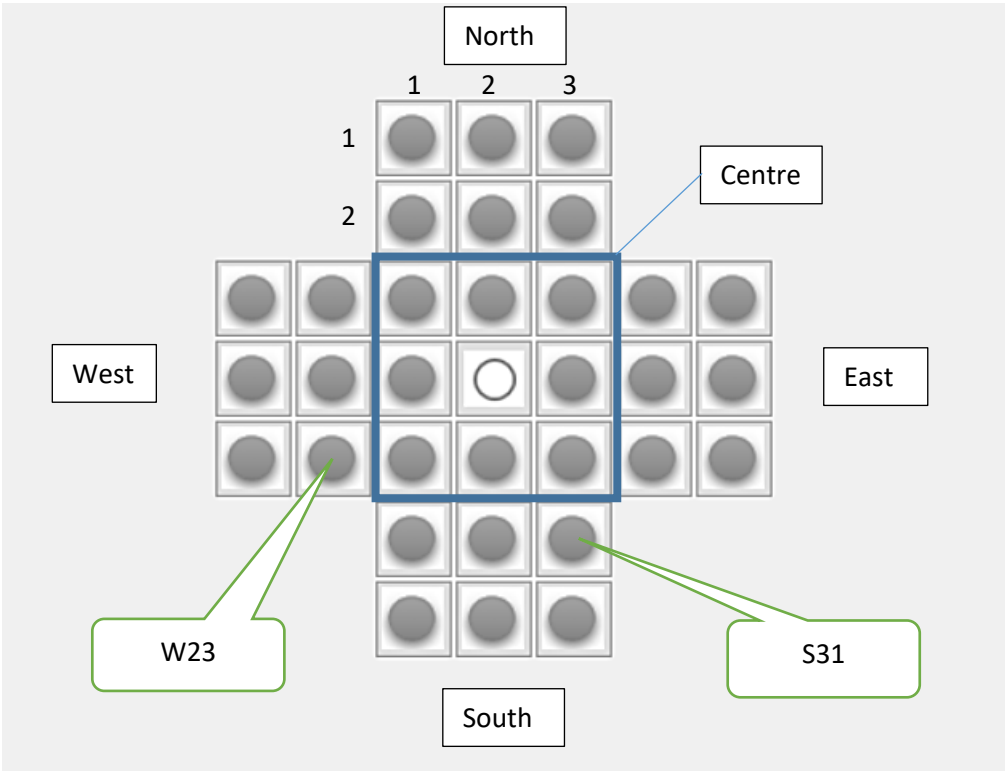
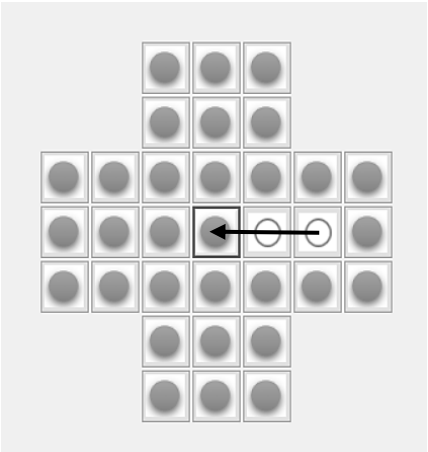
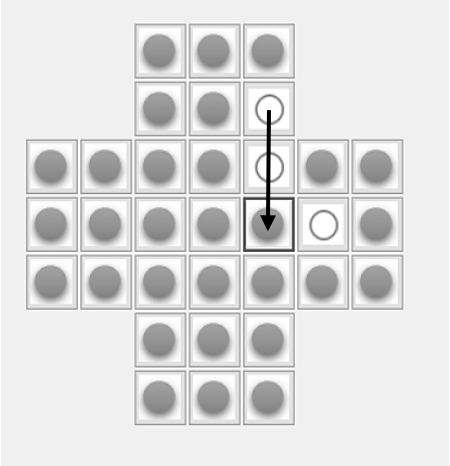


Figure 454: coordinate system for the Solitaire board

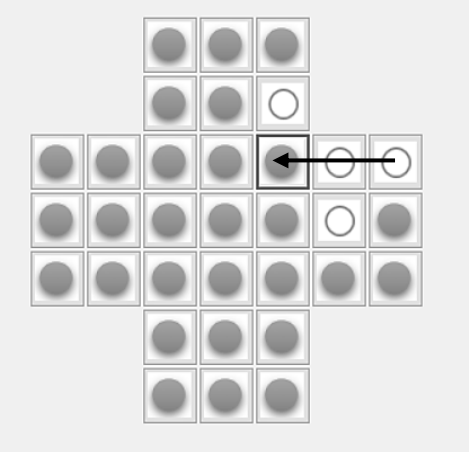
The first objective is to clear most of the pegs from the East section of the board. This is achieved in a series of moves:



E12 – C22

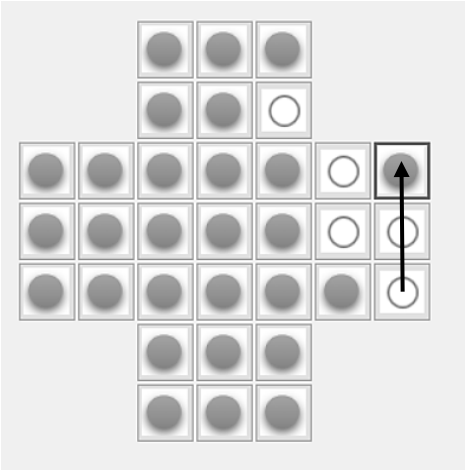


N32 – C32

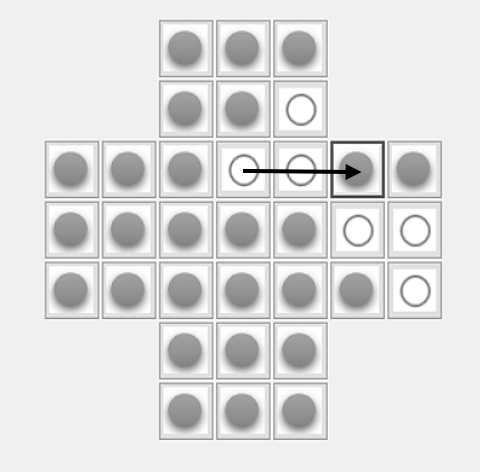


E21 – C31

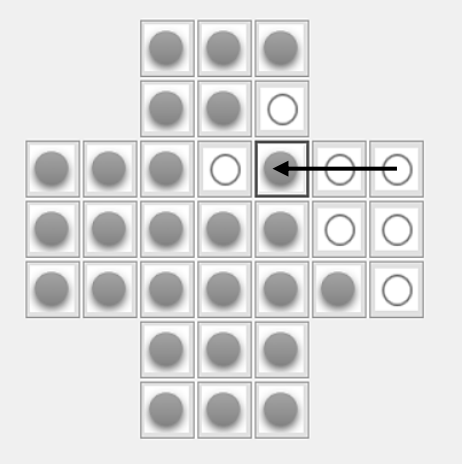
Figure 455: sequence of moves to clear the East section of the board



E23 – E21



C21 – E11



E21 – C31

Figure 455(cont.): sequence of moves to clear the East section of the board

This completes the clearance of the East area of the board. The North area can now be cleared in a similar way, using the sequence of moves listed in figure 456.

W21-C21

N11-C11

N31-N11

C12-N12

N11-C11

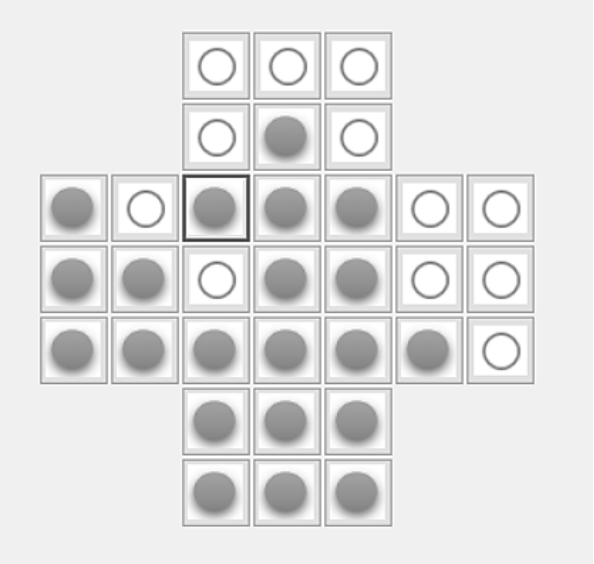


Figure 456: sequence of moves to clear the North section of the board

The West area can be cleared in a similar way, using the sequence of moves in figure 457.

S11-C12

W13-C13

W11-W13

C23-W23

W13-C13

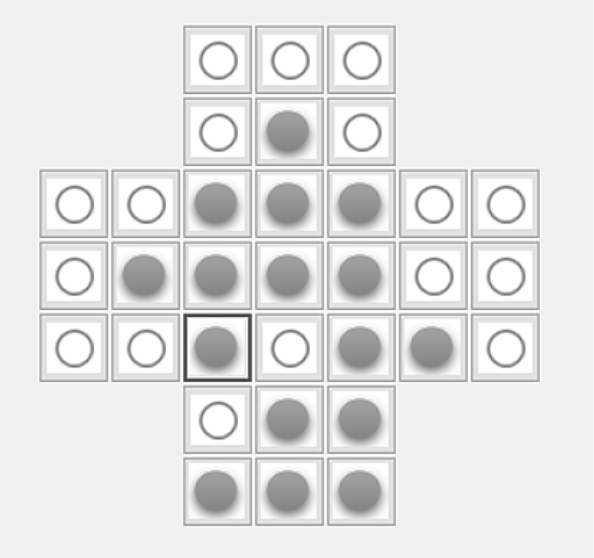


Figure 457: sequence of moves to clear the West section of the board

We finally clear the South section of the board, leaving an arrow-shaped pattern in the centre of the board, as shown in figure 458 below.

E13-C23

S32-C33

S12-S32

C32-S31

S32-C33

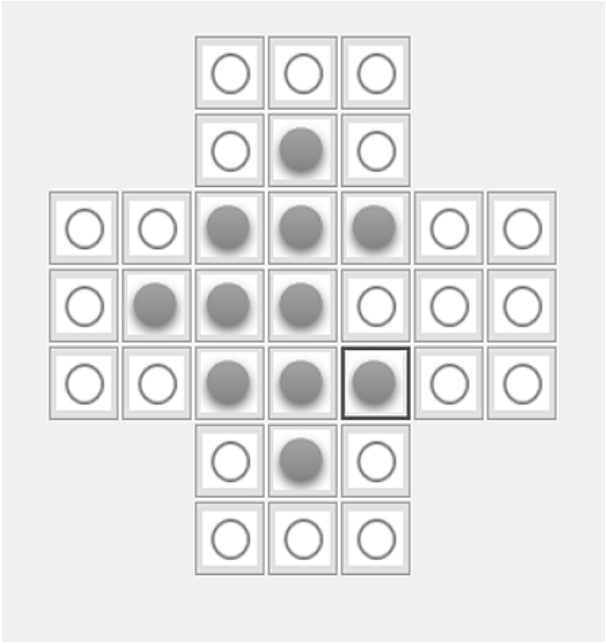


Figure 458: sequence of moves to clear the South section of the board

It is now possible to move one peg around the board in a series of jumps, removing pegs as it goes and leaving a simple T-shape.

C12-N12

N12-N32

N32-C32

C32-S31

S31-S11

S11-C12

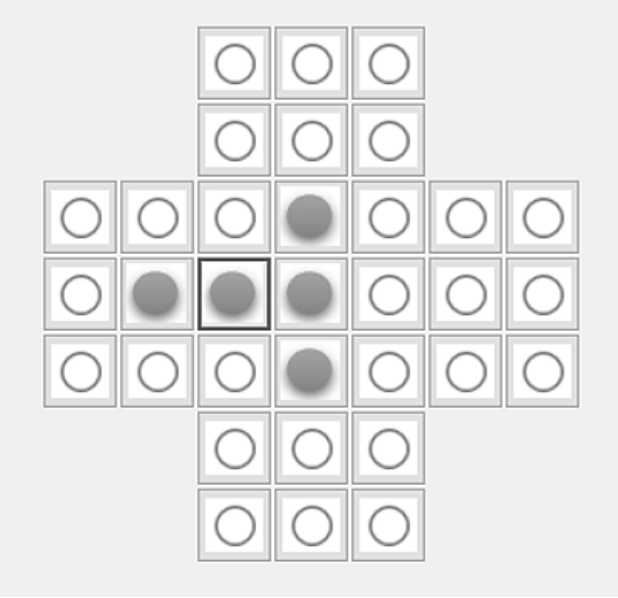


Figure 459: sequence of moves to create the final T-configuration

The game can be completed with the final sequence of moves shown in figure 460.

C22-N22

W22-C22

C23-C21

N22-C22

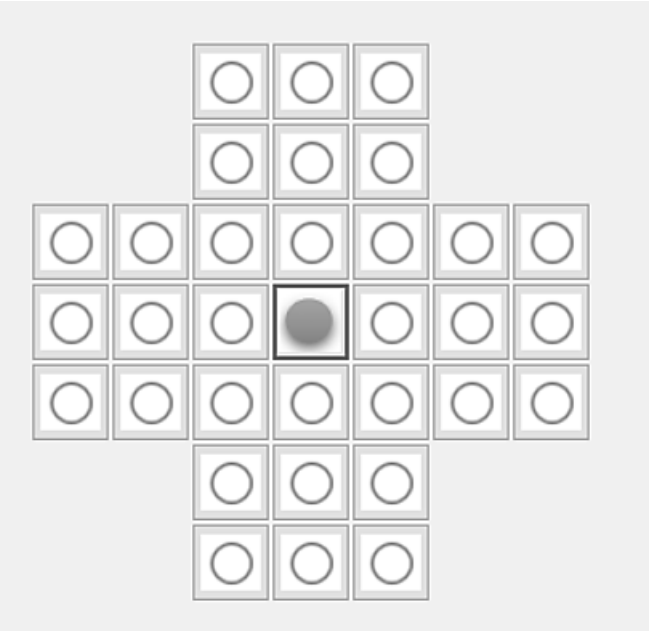


Figure 460: final set of moves to complete the solution

Summary

Algorithms have several features in common:

- During each cycle of the algorithm, a check may be carried out (e.g. whether words are in the correct order, or whether a town can be reached by a shorter distance), then some change may be made to the data (e.g. word order is changed, or a distance total is updated).
- The algorithm steps may be repeated a number of times until some objective has been achieved (e.g. a set of words are now in correct alphabetical order, or the shortest distance for a journey has been found).

An algorithm can provide a sequence of instructions to guide a person through a complex task (e.g. solving the solitaire puzzle) or provide a sequence of commands for a computer program to carry out the task (e.g. generating sets of keys for double key encryption).

Design of efficient algorithms can require high levels of numeracy skills in problem solving, pattern recognition, and perhaps knowledge of techniques in application of number, geometry or algebra. Algorithms are often implemented by computer programs, so an understanding of information technology systems may also be important.