

In this chapter you will learn to:

- recognise the logic in a standard approach (such as a sort or search)
- apply standard approaches as part of the solution to complex problems
- document the logic required to solve problems, including:
 - file handling and management
 - random number generators
 - multi-dimensional arrays
 - nesting of control structures
- develop a suitable set of test data and desk check algorithms that include complex logic
- select an appropriate data structure to solve a given problem
- develop a standard module and document its use
- correctly incorporate a standard module into a more complex solution, passing parameters effectively
- evaluate the effectiveness of using commercially developed software
- represent a software solution in diagrammatic form
- identify the parts of the system that require software to be custom-designed and -developed
- select and use appropriate CASE software to assist in the development of a software solution.

Which will make you more able to:

- explain the interrelationship between hardware and software
- describe how the major components of a computer system store and manipulate data
- identify and evaluate legal, social and ethical issues in a number of contexts
- construct software solutions that address legal, social and ethical issues
- identify needs for which software solutions are appropriate
- apply appropriate development methods to solve software problems
- apply a modular approach to implement well-structured software solutions and evaluate their effectiveness
- apply project management techniques to maximise the productivity of the software development
- create and justify the need for the various types of documentation required for a software solution
- select and apply appropriate software to facilitate the design and development of software solutions
- communicate the processes involved in a software solution to an inexperienced user
- use a collaborative approach during the software development cycle.

In this chapter you will learn about:

Standard algorithms for searching and sorting

- standard logic used in software solutions, namely:
 - finding maximum and minimum values in arrays
 - processing strings (extracting, inserting, deleting)
 - file processing, including sentinel value
 - linear search
 - binary search
 - bubble sort
 - insertion sort
 - selection sort

Custom-designed logic used in software solutions

- requirements to generate these include:
 - identification of inputs, processes and outputs
 - representation as an algorithm
 - definition of required data structures
 - use of data structures, including multi-dimensional arrays, arrays of records, files (sequential and relative/random)
 - use of random numbers
 - thorough testing

Standard modules (library routines) used in software solutions

- requirements for generation or subsequent use include:
 - identification of appropriate modules
 - consideration of local and global variables
 - appropriate use of parameters (arguments)
 - appropriate testing using drivers
 - thorough documentation

Customisation of existing software solutions

- identification of relevant products
- customisation
- cost effectiveness

Documentation of the overall software solution

- tools for representing a complex software solution include:
 - algorithm descriptions
 - system flowcharts
 - structure diagrams
 - data flow diagrams
 - data dictionary

Selection of language to be used

- event-driven software
 - driven by the user
 - program logic
- sequential approach
 - defined by the programmer
- relevant language features
- hardware ramifications
- Graphical User Interface (GUI).

PLANNING AND DESIGN OF SOFTWARE SOLUTIONS

Planning and designing software solutions is the second stage of the software development cycle. This is the stage where data structures are designed and algorithms for the final solution are created. Often modules from previous solutions or from libraries of commonly used modules will be incorporated into the solution.

Models of the system developed during the first stage of the software development cycle are used to give an overall view of the system’s design. Individual team members will be allocated particular modules to complete. Data flow diagrams, structure diagrams, IPO charts and screen designs provide the information necessary for team members to develop the data structures and logic necessary to fully develop each module.

In this chapter, we will examine a number of standard algorithms, introduce some new data structures, examine file processing and then consider some problems requiring custom-designed logic. We conclude by considering issues in regard to selecting a programming language in which to implement the solution.

Before we continue let us review some important concepts from the preliminary course.

What is an algorithm?

An algorithm is a method of solution for a problem. Algorithms describe the steps taken to transform inputs into the required outputs. Each algorithm will have a distinct start and end, and will be composed of the three control structures: sequence, decision and repetition. Algorithms are described using an algorithm description language; in this course we use either pseudocode or flowcharts. Pseudocode and flowcharts provide a language in which we can express algorithms.

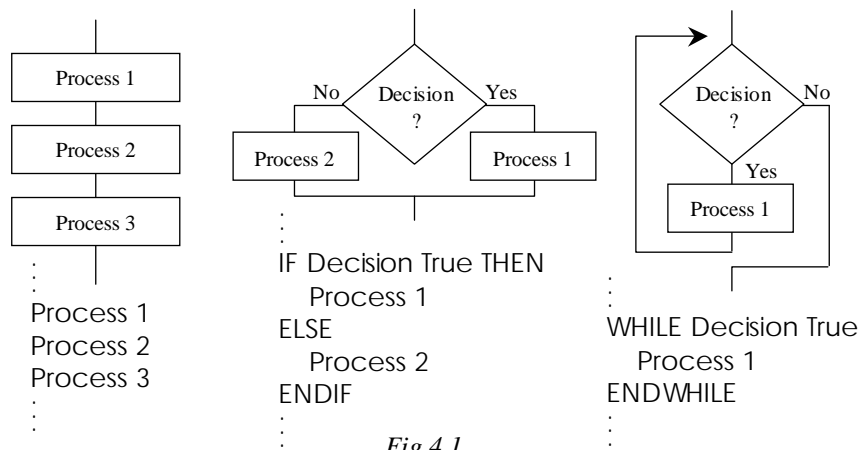


Fig 4.1 The three control structures: sequence, decision and repetition.

What is top-down design?

Top-down design is the process of breaking down a problem into its component parts. Each component part or module is further broken down until a level is reached where the parts can be implemented in a programming language. This process is known as stepwise refinement: each level or step is a refinement of the previous level. Structure diagrams provide a modelling technique for describing the top-down design of a solution.

An algorithm is written for each component or module of the top-down design. The syntax used in pseudocode and flowcharts is illustrated below.

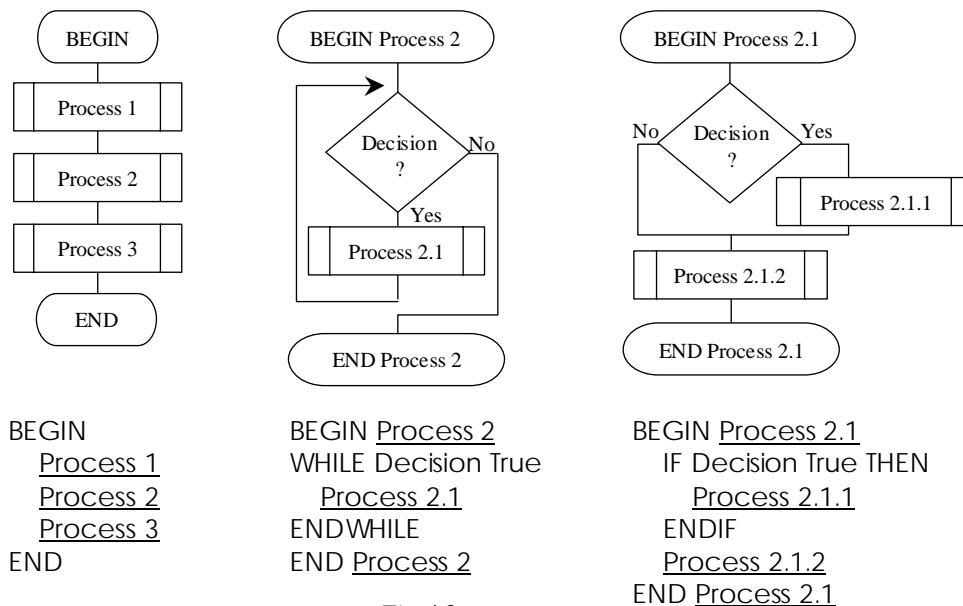


Fig 4.2

The top-down design of a solution expressed using flowcharts and pseudocode.

What are functions and procedures?

The algorithm for each module of the solution will be implemented in a programming language as either a function or a procedure. Functions and procedures are called by higher level modules. Procedures return their results via parameters. Functions return one result by setting the value of the function itself. In essence, the function's name is a variable with a data type just like any other variable. Functions are used as parts of expressions; procedure calls are statements on their own. If you complete *Option 1 Evolution of Programming Languages*, you will learn about functional languages that are entirely based on functions.

For example, if a function called Average is written, which calculates the average of an array of numbers, then `Result = Average(Marks)` would set the variable Result to the Average of the elements contained in the array called Marks. A procedure written to achieve the same result would require two parameters: one for the input and a second parameter to return the result. `Average(Marks, Result)` would return the average via the second parameter, and the variable Result would be set to the average of the elements contained in the array Marks. In this example, a function would be a better choice than a procedure, however both will achieve the same result.

STANDARD ALGORITHMS FOR SEARCHING AND SORTING

Many problems encountered by software developers use similar techniques as part of their solution. It makes sense to develop standard methods of approach rather than continuing to reinvent the wheel. These standard modules can be included in a library of routines for future use. In this section, we examine a number of standard algorithms that perform tasks associated with searching and sorting data. It is more important to understand the concept behind each of these algorithms than to memorise the algorithms themselves.

We will examine standard algorithms for each of the following:

- Linear search
- Finding maximum and minimum values
- Binary search
- Insertion sort
- Selection sort
- Bubble sort
- Processing strings (extracting, deleting and inserting).

LINEAR SEARCH

The word linear means straight line. A linear search involves examining items one at a time from the start of the list to the end of the list. Linear searches do not require the data to be in any particular order. A list containing 100,000 items will require between 1 and 100,000 comparisons to find the required item. On average 50,000 comparisons, or half the number of items in the list, will be required to find a particular item.



The concept

Each item in the list is examined in turn until the required item is found or the end of the list is encountered.

```

BEGIN LinearSearch
  Set Count to 0
  Get ItemToFind
  WHILE more items in list
    IF Item(Count) = ItemToFind THEN
      Display "Found"
    ENDIF
    Add 1 to Count
  ENDWHILE
END LinearSearch
    
```

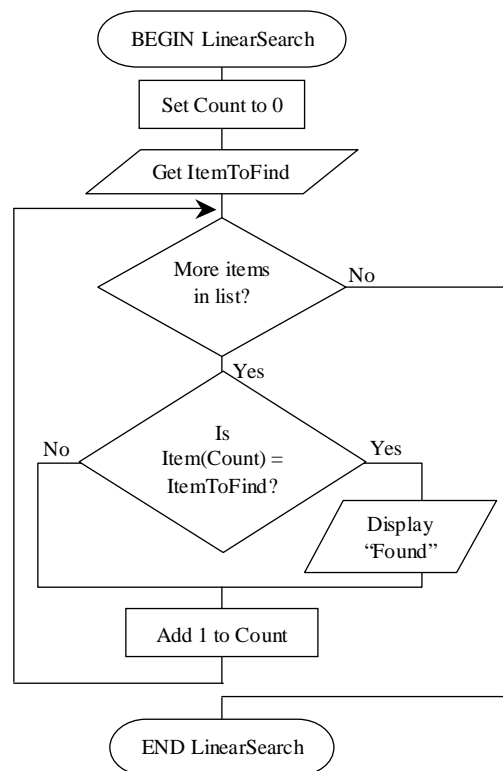


Fig 4.3
Pseudocode and flowchart for a linear search

FINDING MAXIMUM AND MINIMUM VALUES

Finding maximum and minimum values in an unsorted list of items is essentially performed using a linear search where a record is maintained of the largest or smallest value as the list is traversed. For sorted lists, finding maximum and minimum values is trivial: they will be either the last or the first values in the list.



The concept

The first item in the list is initially set as the maximum (or minimum) value. Each item in the list is compared in turn with the maximum (or minimum) value. If an item is found to be larger (or smaller) then it becomes the maximum (or minimum) value. This process continues until the end of the list is reached.

```

BEGIN FindMaximum
  Set Count to 1
  Set Maximum to 0
  WHILE more items in list
    IF Item(Count) > Item(Maximum) THEN
      Set Maximum to Count
    ENDIF
    Add 1 to Count
  ENDWHILE
  Display Item(Maximum)
END FindMaximum
  
```

```

BEGIN FindMinimum
  Set Count to 1
  Set Minimum to 0
  WHILE more items in list
    IF Item(Count) < Item(Minimum) THEN
      Set Minimum to Count
    ENDIF
    Add 1 to Count
  ENDWHILE
  Display Item(Minimum)
END FindMinimum
  
```

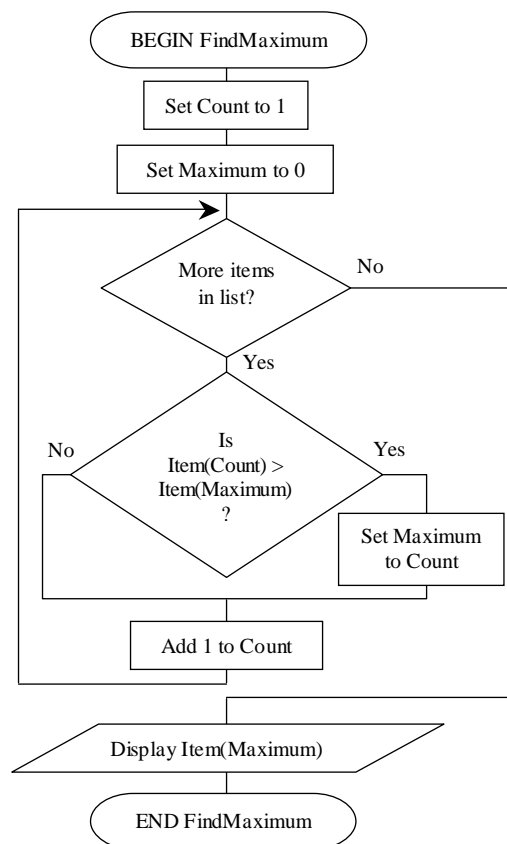


Fig 4.4

Pseudocode for finding maximum and minimum values and flowchart for finding maximum value.



GROUP TASK Activity

Alter the linear search algorithm on the previous page so that it displays the total number of times the search item occurs in the list.



GROUP TASK Activity

Using a flowchart, write an algorithm to find the minimum value by searching from the end of the list to the beginning of the list.



Consider the following:

A list of 30 unsorted integers between 1 and 5 is stored in an array. As part of the solution to a problem it is necessary to search this array and find the number of times each integer occurs in the array.

An algorithm has been written to perform this task and is reproduced below:

```

BEGIN TallyIntegers
  Set Count to 1
  WHILE Count <= 30
    Total(Item(Count)) = Total(Item(Count)) + 1
    Add 1 to Count
  ENDWHILE
END TallyIntegers
    
```

The following set of data is used to perform a desk check of the above algorithm:

1, 2, 2, 5, 5, 4, 3, 4, 5, 3, 2, 2, 3, 1, 4, 4, 3, 3, 1, 5, 3, 2, 4, 1, 3, 4, 3, 5, 1, 3

Following is the desk check:

Count	Item(Count)	Total(1)	Total(2)	Total(3)	Total(4)	Total(5)
1	1	1				
2	2		1			
3	2		2			
4	5					1
5	5					2
6	4				1	
7	3			1		
8	4				2	
9	5					3
10	3			2		
11	2		3			
12	2		4			
13	3			3		
14	1	2				
15	4				3	
16	4				4	
17	3			4		
18	3			5		
19	1	3				
20	5					4
21	3			6		
22	2		5			
23	4				5	
24	1	4				
25	3			7		
26	4				6	
27	3			8		
28	5					5
29	1	5				
30	3			9		
31						



GROUP TASK Discussion

Describe the purpose and data type of Count, Item and Total in regard to this algorithm.



GROUP TASK Discussion

Discuss the similarities between this algorithm and a standard linear search algorithm.

BINARY SEARCH

The word binary means two. A binary search involves splitting the list into two parts each time a comparison is made. As the list must have been previously sorted, one half of the list can be discarded after each comparison is made. Eventually the required item will be found. A list of items containing 100,000 items will after 1 comparison be reduced to 50,000 items, after two comparisons to 25,000 items, 3 comparisons 12,500 items, and so on. For 100,000 items it will take a maximum of only 17 comparisons to locate a specific item in the list. As $2^{16} = 65,536$ and $2^{17} = 131,072$, and there are 100,000 items, and 100,000 lies between 2^{16} and 2^{17} , the maximum number of comparisons is 17. In general, a list containing X items will take $\ln X / \ln 2$ (rounded up to the nearest integer) comparisons to find any particular item.



The concept

The list of data items must be sorted. The middle item in the list is compared to the required item. If the required item lies in the first half of the list then the second half is discarded. If the required item lies in the second half of the list then the first half is discarded. This process is repeated with the remaining list of items. Eventually the required item will be found or the list of possible items will be empty.

```

BEGIN BinarySearch
  Set Low to 1
  Set High to number of items in list
  Set Found to False
  Get ItemToFind
  WHILE High > Low AND Found = False
    Set Middle to INT((Low + High)/2)
    IF ItemToFind < Item(Middle) THEN
      Set High to Middle - 1
    ELSEIF ItemToFind = Item(Middle) THEN
      Set Found to True
    ELSE
      Set Low to Middle + 1
    ENDIF
  ENDWHILE
  IF Found = True THEN
    Display "Found"
  ELSE
    Display "Not found"
  ENDIF
END BinarySearch
    
```

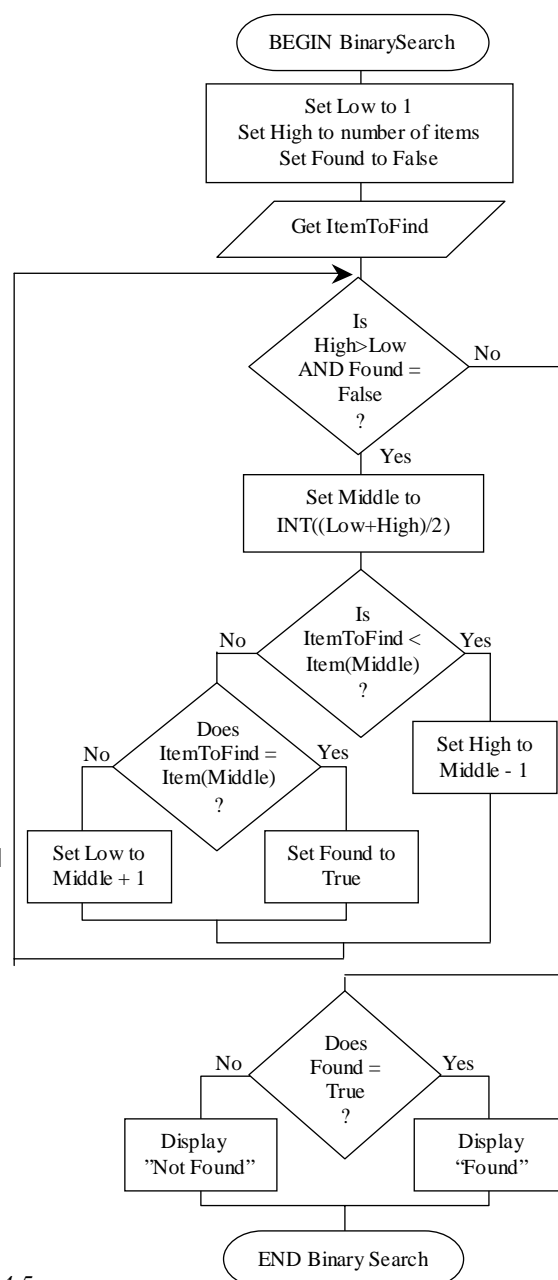


Fig 4.5
Pseudocode and flowchart for a binary search.



Consider the following:

A list of animals has been sorted into alphabetical order. The list is stored in an array called Animals. A binary search is required to find if a particular animal already exists within the array Animals.

Let us consider the following sample Animals array. Using the binary search algorithm from the previous page, with the Items array replaced by the Animals array, we would first set Low to 1 and High to 10. Found is also set to False.

1	2	3	4	5	6	7	8	9	10	
Ant	Beetle	Cow	Dog	Frog	Goose	Hen	Moose	Rat	Yak	
Low										High

We now get an ItemToFind, say Cow. As High is greater than Low and Found is False we progress into the body of the loop and set Middle to $\text{INT}((1 + 10)/2)$, which is 5.

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Cow	Dog	Frog	Goose	Hen	Moose	Rat	Yak
Low		Middle				High			

We now compare the required item Cow with the data item associated with 5, namely Frog. As Cow is less than Frog, in an alphabetical sense, High is set to 4, the current value of Middle - 1. In essence, we have discarded the top half of the array.

1	2	3	4
Ant	Beetle	Cow	Dog
Low		High	



Flag
 A variable used to indicate that a certain condition has been met. Usually a Flag is set to either True or False. In the binary search algorithm, Found is a Flag.

We now repeat the process. Middle is set to $\text{INT}((1 + 4)/2)$, which is 2.

1	2	3	4
Ant	Beetle	Cow	Dog
Low	Middle	High	

Cow is compared to item Animals(2). It is not less than Beetle and it is not equal to Beetle, so Low is set to 3, the current value of Middle + 1. The lower half of the array is discarded.

3	4
Cow	Dog
Low	High

The process repeats again. Middle is set to $\text{INT}((3 + 4)/2)$, which is 3. We compare the required item with Animals(3) and find we have a match. Found is set to True and we then drop out of the loop. As Found is True the message Found is displayed.



GROUP TASK Activity

Choose an animal that is not in the above array. Work through the binary search to ensure the algorithm operates correctly and the output is Not found.



GROUP TASK Activity

Write down a list of 30 numbers in order from lowest to highest. Select one of the numbers and perform a desk check of the binary search algorithm.

SET 4A

1. A search that involves comparing each item in turn with the required item is called a:
 - (A) binary search.
 - (B) descending search.
 - (C) quadratic search.
 - (D) linear search.
2. A sorted list is one of the requirements of a:
 - (A) binary search.
 - (B) descending search.
 - (C) quadratic search.
 - (D) linear search.
3. Finding the maximum data item in a list is relevant for:
 - (A) numeric data.
 - (B) string or text data.
 - (C) a sorted list.
 - (D) both A and B.
4. A search of a data list containing 20,000 items takes on average 10,000 comparisons to find a particular item. The type of search being used is probably a:
 - (A) binary search.
 - (B) minimum search.
 - (C) maximum search.
 - (D) linear search.
5. Functions and procedures are the methods used to implement algorithms. Which of the following statements is true?
 - (A) Procedures always return one value; functions can return any number of values.
 - (B) Functions always return one value; procedures can return any number of values.
 - (C) Functions are only able to process numeric data.
 - (D) All functions must include at least one input parameter.
6. The process of progressively breaking a problem down into its component parts is known as:
 - (A) algorithm development.
 - (B) top-down design.
 - (C) system modelling.
 - (D) stepwise refinement.
7. A library of routines is:
 - (A) used to store standard algorithms for reuse.
 - (B) used to store standard modules of source code for reuse.
 - (C) of use to end users who wish to modify the original product.
 - (D) a common way of storing the top-down design of a program.
8. An algorithm description language:
 - (A) provides excellent user documentation.
 - (B) is either a flowchart or pseudocode.
 - (C) provides a method of expressing algorithms.
 - (D) is a type of programming language.
9. Any particular item is always found in a list of 1000 items within 10 comparisons. What can be said about the 1000 items?
 - (A) There must be only 10 unique items in the list.
 - (B) The list of items must be sorted.
 - (C) Most of the list must be empty.
 - (D) There is insufficient information to answer this question.
10. The largest value in a list of 10,000 items is found to be in position 234. Which of the following is True?
 - (A) The list must be sorted.
 - (B) Larger items may exist in positions 235 to 10,000.
 - (C) The list must be unsorted.
 - (D) The list only contains numeric data.
11. Often, with numeric decimal values, a problem will involve finding a value within a specific range. Design a linear search algorithm that will search for and display all the values in a list between a lower and upper value.
12. Create an algorithm to find and output both the maximum and the minimum values stored in an array of unsorted data items.
13. Perform a desk check of both the linear search algorithm and the binary search algorithm using the following test data: The Item array contains the 6 elements CPU, Keyboard, Monitor, Mouse, Printer and Scanner indexed from 1 to 6. The input to the algorithms is Printer.
14. An array contains a sorted list of 2000 people's surnames. Design an algorithm using a binary search strategy that will find the number of occurrences of a particular surname.
15. Your English teacher wishes you to develop a program to tally marks for them. The marks are to be tallied into deciles. For example, a mark of 72 is in the 8th decile, 36 is in the 4th decile and a mark of 6 is in the 1st decile. Design an algorithm to accomplish this task. Assume the marks are stored in an unsorted array called Marks, which is indexed from 0 to 99.

INSERTION SORT

Insertion sorts are used to arrange data into order. They are particularly useful when a small number of items need to be added to an already sorted list. Insertion sorts involve looking through a sorted list for the correct position for a new item and inserting the item into that position.

For example, imagine we are playing a card game. At this stage of the game we pick up two cards and add or insert them into our existing hand of sorted cards. We are using an insertion sort to accomplish this task.

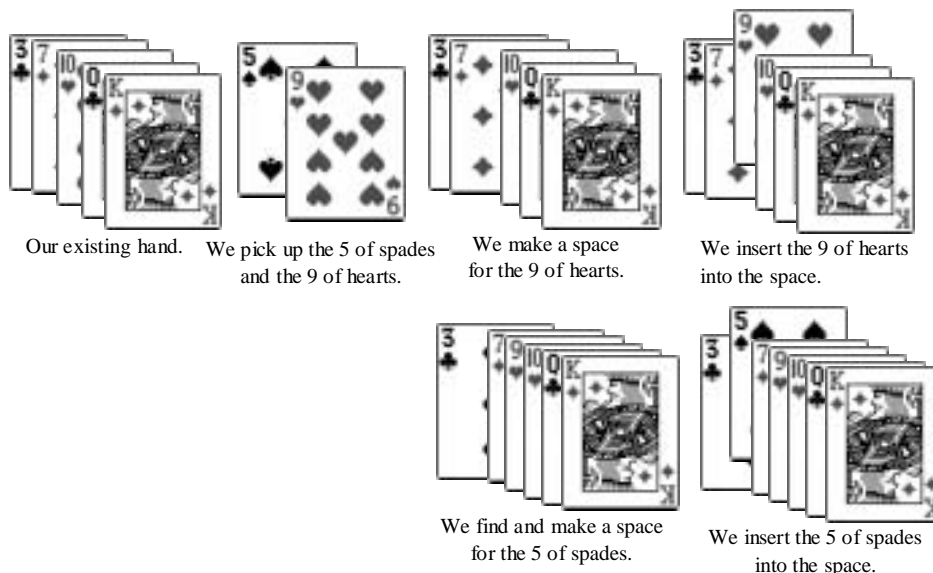


Fig 4.6

Cards inserted using an insertion sort strategy.



The simplified concept

An unsorted and a sorted list are maintained. Each item in the unsorted list is considered in turn. Its correct place in the sorted list is found. A space is made by moving (or shuffling) items up one place. Finally, the item is placed in position. This process continues until no items remain in the unsorted list.

```
BEGIN
  WHILE more items to insert
    Find correct position for item
    Make space for new item
    Insert item
  ENDWHILE
END
```

Fig 4.7

A simplified insertion sort algorithm.



The enhanced concept

The following algorithm expands on the simplified version above. In this version, a complete array called Item is sorted. We start by considering the first item to be the sorted part of the array. We then consider where the second item should be placed in relation to the first item, and insert it into position. The third item is then considered. Its correct place in the sorted list is found, a space is made and the item is inserted. This process continues until no items remain in the unsorted list.

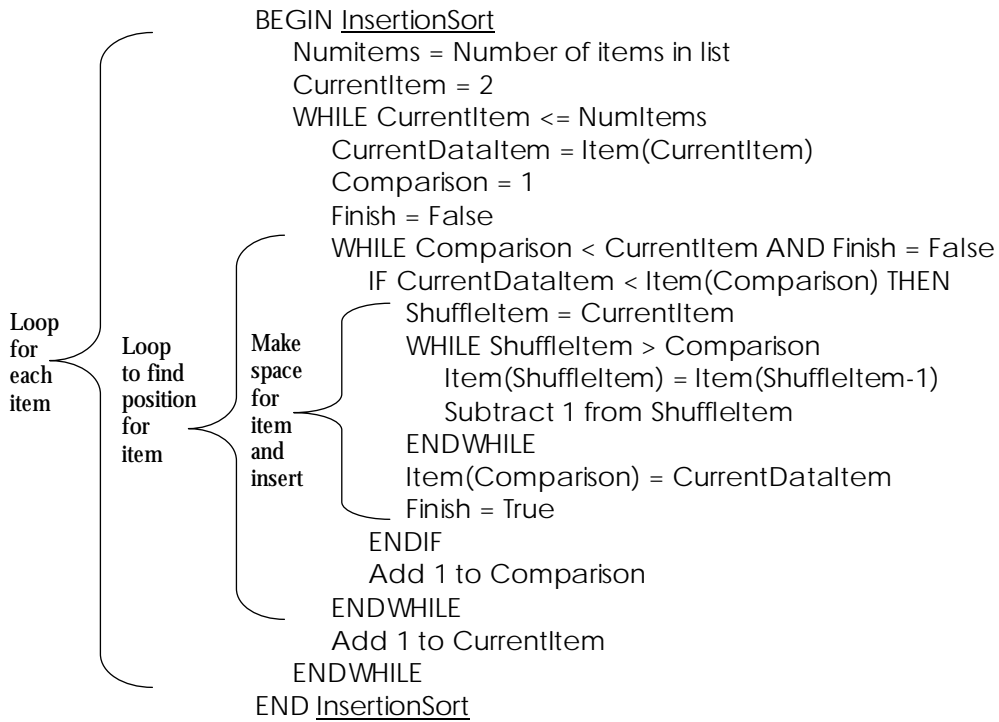


Fig 4.8 Expanded version of an insertion sort algorithm.



Consider the following:

The array of animals below is to be sorted using the insertion sort described in the algorithm in Fig 4.8. Let us examine the array after each item has been inserted. The sorted list initially contains only the first item, Goose, and we consider the second item, Yak.

1	2	3	4	5	6	7	8	9	10
Goose	Yak	Ant	Dog	Moose	Cow	Hen	Rat	Frog	Beetle

Yak is now in the sorted part of the list and we consider the third item, Ant:

1	2	3	4	5	6	7	8	9	10
Goose	Yak	Ant	Dog	Moose	Cow	Hen	Rat	Frog	Beetle

Ant belongs to the left of Goose, so Goose and Yak have been moved up one and Ant inserted into the space. We now consider the fourth item, Dog:

1	2	3	4	5	6	7	8	9	10
Ant	Goose	Yak	Dog	Moose	Cow	Hen	Rat	Frog	Beetle

Dog belongs between Ant and Goose, so Goose and Yak are moved to the right and Dog is inserted. We now consider the fifth item, Moose:

1	2	3	4	5	6	7	8	9	10
Ant	Dog	Goose	Yak	Moose	Cow	Hen	Rat	Frog	Beetle

Moose has been inserted into position. We now consider Cow:

1	2	3	4	5	6	7	8	9	10
Ant	Dog	Goose	Moose	Yak	Cow	Hen	Rat	Frog	Beetle

Cow is inserted between Ant and Dog. We now consider the seventh item, Hen:

1	2	3	4	5	6	7	8	9	10
Ant	Cow	Dog	Goose	Moose	Yak	Hen	Rat	Frog	Beetle

Space is made for Hen in position five and Hen is inserted. We now consider Rat:

1	2	3	4	5	6	7	8	9	10
Ant	Cow	Dog	Goose	Hen	Moose	Yak	Rat	Frog	Beetle

Rat comes before Yak, so Yak is shuffled up and Rat inserted. We now examine Frog:

1	2	3	4	5	6	7	8	9	10
Ant	Cow	Dog	Goose	Hen	Moose	Rat	Yak	Frog	Beetle

Frog is inserted into position four by shuffling up items. Finally, Beetle is examined:

1	2	3	4	5	6	7	8	9	10
Ant	Cow	Dog	Frog	Goose	Hen	Moose	Rat	Yak	Beetle

Space is made and Beetle is inserted. As there are no more items in the unsorted part of the list, the sort is complete:

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Cow	Dog	Frog	Goose	Hen	Moose	Rat	Yak



GROUP TASK Discussion

The insertion sort described in *Fig 4.8* is used to sort an entire array of data. Is this the most appropriate sort for this purpose? Discuss.



GROUP TASK Discussion

Perform an insertion sort showing the state of the array after each item has been inserted, using the following data:

34, 44, 23, 35, 34, 55, 42, 12, 52, 16, 16, 17, 56, 78, 26, 23, 41, 29, 30, 42.

This time perform the sort from right to left. That is, initially the sorted part of the array will contain 42, and the first item to be inserted will be 30.

SELECTION SORT

Selection sorts essentially are a repetition of a linear search to find the smallest (or largest) item in a list. Each time the search is complete, the item found is moved to the end of the sorted list. This type of sort is rather more intuitive for humans as it is a strategy often employed when sorting things by hand.

For example, when playing card games we often arrange the cards in our hand using this selection sort method.

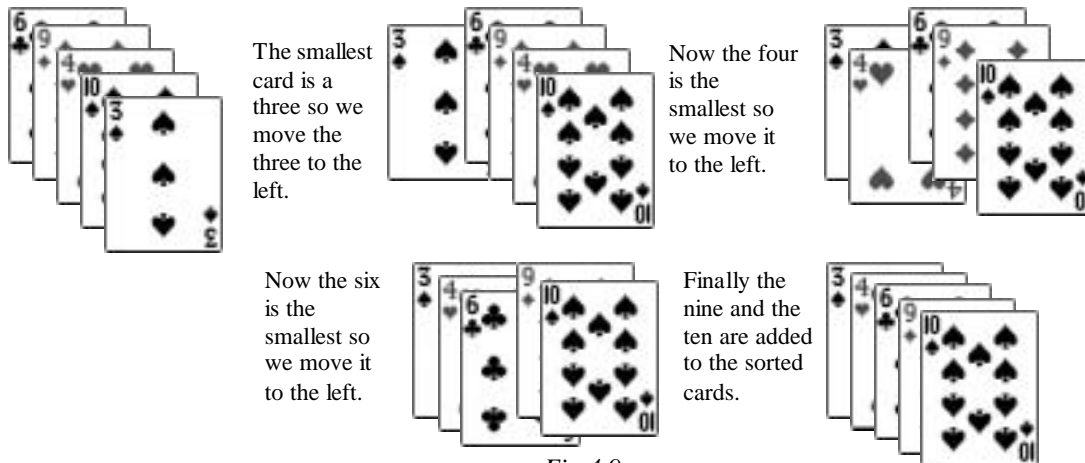


Fig 4.9
Cards sorted using a selection sort strategy.



The simplified concept

A sorted and an unsorted list are maintained. The smallest item in the unsorted list is swapped with the first item in the unsorted list. The sorted list is increased in length to include the new item. This process is repeated while there are items remaining in the unsorted list.

```
BEGIN SimpleSelectionSort
  WHILE unsorted list has items
    Find smallest item
    Swap with first item in unsorted list
    Increase sorted list size by 1
  ENDWHILE
END SimpleSelectionSort
```

Fig 4.10
A simplified selection sort algorithm.



Swap

A swap is used to exchange the content of one variable with that of another. A temporary variable and three steps are required.

```
Temp = X
X = Y
Y = Temp
```



The enhanced concept

The list is examined one item at a time for the smallest item. This item is swapped with the first item in the list. This first item is now sorted and becomes the sorted list. The remaining unsorted list is examined for the smallest item. This item is swapped with the first item in the unsorted list. The length of the sorted list is increased by one. There are now two items in the sorted list. This process is repeated while there are still items remaining in the unsorted list.

```

BEGIN SelectionSort
  Pass = 1
  WHILE Pass < Number of items
    Count = Pass + 1
    Minimum = Pass
    WHILE Count <= Number of Items
      IF Item(Count) < Item(Minimum) THEN
        Minimum = Count
      ENDIF
      Count = Count + 1
    ENDWHILE
    Swap Item(Minimum) with Item(Pass)
    Pass = Pass + 1
  ENDWHILE
END SelectionSort
    
```

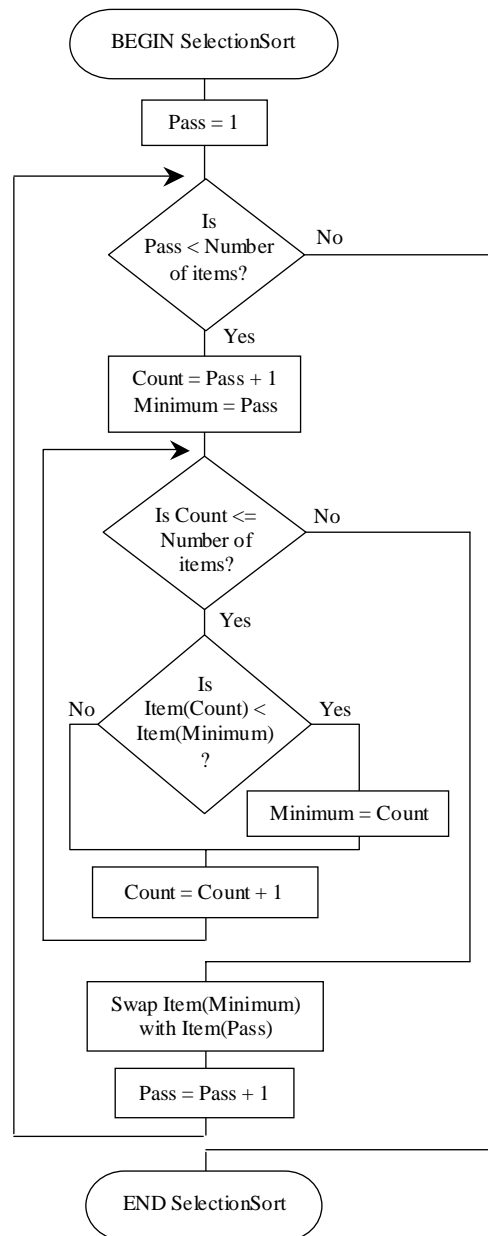


Fig 4.11 Pseudocode and flowchart for a selection sort.



Consider the following:

The array of animals below is to be sorted using the selection sort described in the algorithm in Fig 4.11. Firstly, we search for the smallest item in the list, namely Ant:

1	2	3	4	5	6	7	8	9	10
Goose	Yak	Ant	Dog	Moose	Cow	Hen	Rat	Frog	Beetle

Ant is swapped with the first item, Goose. The new first item, Ant, now forms the sorted part of the array. The unsorted list is searched for the smallest item, namely Beetle:

1	2	3	4	5	6	7	8	9	10
Ant	Yak	Goose	Dog	Moose	Cow	Hen	Rat	Frog	Beetle

Beetle is swapped with the first item in the unsorted list, namely Yak. The unsorted list is again scanned for the smallest item, and Cow is found:

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Goose	Dog	Moose	Cow	Hen	Rat	Frog	Yak

Cow and Goose are swapped. The smallest item is now Dog:

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Cow	Dog	Moose	Goose	Hen	Rat	Frog	Yak

Dog is swapped with itself. The smallest item is now Frog:

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Cow	Dog	Moose	Goose	Hen	Rat	Frog	Yak

Frog and Moose are swapped. Goose is now the smallest item:

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Cow	Dog	Frog	Goose	Hen	Rat	Moose	Yak

Goose is swapped with itself. Hen is now the smallest item:

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Cow	Dog	Frog	Goose	Hen	Rat	Moose	Yak

Hen is swapped with itself. Moose is now the smallest item:

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Cow	Dog	Frog	Goose	Hen	Rat	Moose	Yak

Moose is swapped with Rat. Rat is now the smallest item:

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Cow	Dog	Frog	Goose	Hen	Moose	Rat	Yak

Rat is swapped with itself. As only one item remains, the list is now sorted.

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Cow	Dog	Frog	Goose	Hen	Moose	Rat	Yak



GROUP TASK Discussion

Examine the selection sort algorithm and describe how the variables Pass and Count combine to change the length of the sorted and unsorted sections of the array Item.



GROUP TASK Activity

Perform a selection sort showing the state of the array after a new item has been placed into the sorted part of the array, using the following data: 34, 44, 23, 35, 34, 55, 42, 12, 52, 16, 16, 17, 56, 78, 26, 23, 41, 29, 30, 42. This time perform the sort from right to left. That is, search for the largest item each time and swap it with the right-hand item in the unsorted part of the array.

BUBBLE SORT

Bubble sorts, just like the insertion and selection sorts, are used to arrange data into either numerical or alphabetical order. Data items tend to bubble, or float to the end of the list when a bubble sort is undertaken. The list is sorted by considering the order of individual item pairs and swapping them if they are out of order. This results in items moving one position at a time towards their final position.



The simplified concept

Pairs of adjacent items are compared in turn. If they are out of order, they are swapped. Once the end of the list is reached, we move back to the start and repeat the process. Eventually no swaps will be required, indicating that the list is sorted.

```

BEGIN SimpleBubbleSort
  Swapped = True
  WHILE Swapped = True
    Swapped = False
    Comparison = 1
    WHILE Comparison < Number of items
      IF Item(Comparison) < Item(Comparison + 1)
        Swap Items
        Swapped = True
      ENDIF
      Add 1 to Comparison
    ENDWHILE
  ENDWHILE
END SimpleBubbleSort

```

*Fig 4.12
Pseudocode for a simple bubble sort.*

The Boolean variable `Swapped` is used as a flag to determine if a swap has occurred during a pass through the array `Items`. If no swap has occurred, indicating that the array is sorted, then `Swapped` will have a value of `False` and the algorithm will end. The variable `Comparison` is used to traverse the array. For example, when `Comparison` is equal to 5, items 5 and 6 are being compared; when `Comparison` equals 6, items 6 and 7 are being compared.

**Boolean Variable**

A variable that can hold one of two values: `True` or `False`. Boolean variables are often used as flags to indicate that a certain condition has been met.



Consider the following:

The simple bubble sort described in the algorithms in Fig 4.12 and Fig 4.13 will correctly sort an array of items into ascending order.



GROUP TASK Activity

Rewrite the simple bubble sort algorithm so that the array is sorted into descending order.



GROUP TASK Activity

Using the following list of data items, complete a full desk check of the simple bubble sort algorithm:

34, 44, 23, 35, 34, 55, 42, 12, 52, 16,
16, 17, 56, 78, 26, 23, 41, 29, 30, 42.

Perform a desk check of your descending bubble sort algorithm using the above data list.

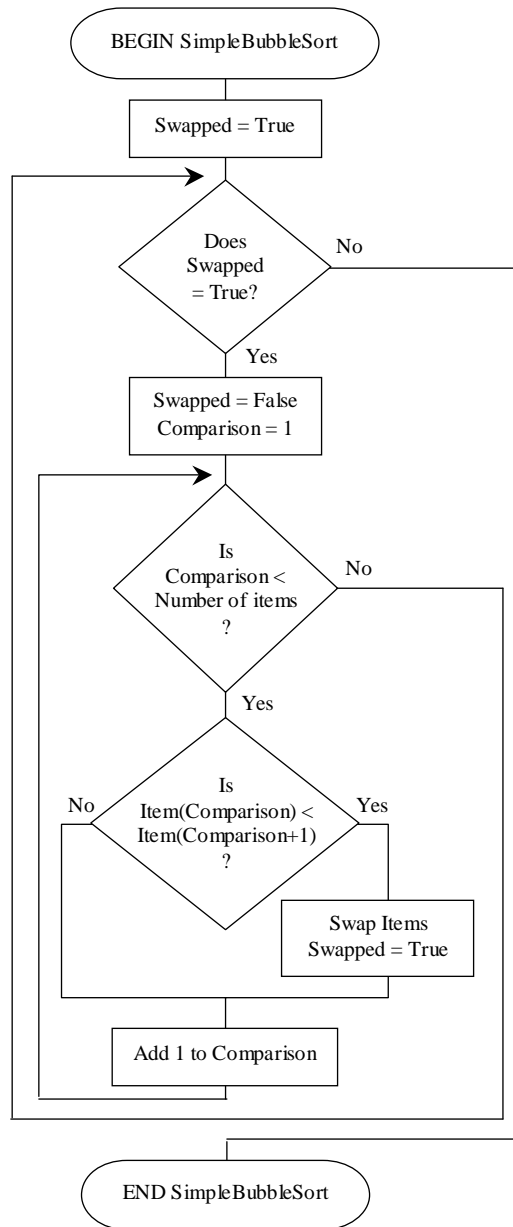


Fig 4.13
Flowchart for a simple bubble sort.

The simple bubble sort can be further enhanced to optimise its operation. We will look at an enhanced bubble sort algorithm in the next section. Before considering the enhanced bubble sort it is important that you be confident in the operation of the simplified version. Complete a number of desk checks of the algorithm to become confident in regard to its operation.



The enhanced concept

The enhanced bubble sort recognises that after each pass through the list the largest item will be in its correct position at the right-hand end of the list. It is therefore not necessary to consider this item in future passes through the list. In other words, the length of the unsorted list always decreases by one after each pass. If we keep track of the number of passes completed, we can then just complete comparisons up to the number of items in the list minus the number of completed passes.

Let us consider the saving in terms of comparisons. For a list of 100 items, using the simplified version could take 99 passes through the list, and each pass involves 99 comparisons and possible swaps. Ninety-nine times 99 results in 9,801 comparisons necessary to be sure the list is sorted. The enhanced bubble sort could still take 99 passes to complete, however the first pass will involve 99 comparisons, the second 98 comparisons, the third 97 and so on. The last pass will involve only one comparison

```
BEGIN EnhancedBubbleSort
  Swapped = True
  Pass = 0
  WHILE Swapped = True
    Swapped = False
    Comparison = 1
    WHILE Comparison < Number of Items - Pass
      IF Item(Comparison) < Item(Comparison + 1)
        Swap Items
        Swapped = True
      ENDIF
      Add 1 to Comparison
    ENDWHILE
    Add 1 to Pass
  ENDWHILE
END EnhancedBubbleSort
```

Fig 4.14

Pseudocode for the enhanced bubble sort.

as there will only be 2 items remaining in the list. The average of 99, 98, 97...3, 2, 1 is 50 - that is, the average number of comparisons per pass is 50. Ninety-nine passes times 50 comparisons per pass results in a total of 4,950 comparisons. This is approximately half the number required by the simplified bubble sort. The larger the list of data items, the larger the saving in comparisons. In general, the number of comparisons is halved using the enhanced version rather than the simplified version.



Consider the following:

The array of animals below is to be sorted using the enhanced bubble sort described in the algorithm in Fig 4.14. Let us examine the array after each pass is completed.

1	2	3	4	5	6	7	8	9	10
Goose	Yak	Ant	Dog	Moose	Cow	Hen	Rat	Frog	Beetle

After the first pass, with Yak now in its correct position:

1	2	3	4	5	6	7	8	9	10
Goose	Ant	Dog	Moose	Cow	Hen	Rat	Frog	Beetle	Yak

After the second pass. Notice that Goose and Moose are bubbling up:

1	2	3	4	5	6	7	8	9	10
Ant	Dog	Goose	Cow	Hen	Moose	Frog	Beetle	Rat	Yak

After the third pass, Goose has moved up and Moose is now correct:

1	2	3	4	5	6	7	8	9	10
Ant	Dog	Cow	Goose	Hen	Frog	Beetle	Moose	Rat	Yak

After the fourth pass, Hen is now correct:

1	2	3	4	5	6	7	8	9	10
Ant	Cow	Dog	Goose	Frog	Beetle	Hen	Moose	Rat	Yak

After the fifth pass, five items are now in their correct positions:

1	2	3	4	5	6	7	8	9	10
Ant	Cow	Dog	Frog	Beetle	Goose	Hen	Moose	Rat	Yak

After the sixth pass:

1	2	3	4	5	6	7	8	9	10
Ant	Cow	Dog	Beetle	Frog	Goose	Hen	Moose	Rat	Yak

After the seventh pass:

1	2	3	4	5	6	7	8	9	10
Ant	Cow	Beetle	Dog	Frog	Goose	Hen	Moose	Rat	Yak

After the eighth pass:

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Cow	Dog	Frog	Goose	Hen	Moose	Rat	Yak

The ninth pass completes the process:

1	2	3	4	5	6	7	8	9	10
Ant	Beetle	Cow	Dog	Frog	Goose	Hen	Moose	Rat	Yak



GROUP TASK Discussion

Examine each pass in the above sort. How many swaps have occurred during each pass?



GROUP TASK Discussion

In this example, no swap occurred on the ninth pass. If a swap had occurred on the ninth pass, explain how the sort would have ended. Examine the value of any relevant variables as part of your discussion.

PROCESSING STRINGS

Strings are data types used to hold characters or text. Many applications require the manipulation of string data as part of their solution. A string is essentially an array of characters; in some programming languages strings must be declared as arrays of characters. Some useful routines for processing strings include ones for extracting, deleting or inserting strings out of or into other strings. Most programming languages include functions for performing these activities. In this section, we examine the algorithms behind these standard string manipulation functions.

Joining two strings is known as concatenation. Most programming languages use either an ampersand (&) or an addition sign (+) as the symbol for concatenation. In this text we will use the addition sign to indicate concatenation. For example, "The" + "Cat" would result in the string "TheCat". If the string variable Word contains "Dogs" and the variable Phrase contains the string "chase balls", then Word + " " + Phrase results in the concatenated string "Dogs chase balls".

Let us now consider algorithms for extracting, deleting and inserting strings out of and into other strings. These algorithms have been written as functions with inputs via parameters, as this is generally the way they are implemented in most programming languages.



Parameter

A value or variable passed to or returned from a function or procedure. Parameters are often called arguments.

Extracting

Extracting data from a string is essentially making a copy of a section of the original string; the original string remains intact. Three pieces of information are required to achieve the extraction: the original or initial string; the starting point for the extraction; and either the length of the extraction string or the finish point of the extraction string.

The algorithm in *Fig 4.15* describes a function that receives the start and finish positions of the required string, together with the actual initial string, as input via the three parameters Start, Finish and ExtractString. The characters within the initial string from position Start to position Finish are returned by the function.

The function call:

Extract(6, 8, "Mathematics") returns the string "mat".

```
BEGIN Extract(Start,Finish,ExtractString)
  Temp = Null string
  Position = Start
  WHILE Position <= Finish
    Temp = Temp + ExtractString(Position)
    Add 1 to Position
  ENDWHILE
  Extract = Temp
END Extract
```

Fig 4.15

Extract algorithm for strings.



GROUP TASK Activity

Design an algorithm for an extract function using as a parameter the length of the required string, rather than the finish position.

Deleting

Deleting data from a string is essentially cutting a portion from the string. The original string is reduced in length by the number of characters that have been removed. As for the extraction algorithm, three identical parameters are required: the original or initial string; the starting point for the deletion; and either the length of the string to be deleted or the finish point of the string to be deleted.

The deletion algorithm can be created using the extract function (see *Fig 4.16*). This algorithm takes the first part of the array up to the characters to be deleted and concatenates this string with the characters that follow the string to be deleted.

A more efficient algorithm, such as the one in *Fig 4.17*, moves each character occurring after the characters to be deleted forward to join the existing front of the original array. The length of the string is then reduced by the length of the deleted string.

```
BEGIN Delete(Start,Finish,DeleteString)
  Temp = Extract(1,Start-1,DeleteString)
  Length = Length of DeleteString
  Temp = Temp + Extract(Finish+1,Length,DeleteString)
  Delete = Temp
END Delete
```

Fig 4.16

Delete algorithm using the Extract function.

```
BEGIN Delete(Start,Finish,DeleteString)
  Temp = DeleteString
  Position = Start
  Length = Finish - Start + 1
  StrLength = length of DeleteString - Length
  WHILE Position <= StrLength
    Temp(Position) = Temp(Position + Length)
    Add 1 to Position
  ENDWHILE
  Reduce size of Temp by Length
  Delete = Temp
END Delete
```

Fig 4.17

Delete algorithm for strings.



GROUP TASK Activity

Design an algorithm for a delete function using as a parameter the length of the required string, rather than the finish position.

Inserting

Inserting a string of characters into another string of characters involves splitting the initial string into two parts and then concatenating the first part of the string with the insertion string and the final part of the initial string. Parameters required to complete this process are the start position for the insertion, the insertion string, and the main or initial string.

This process can be accomplished by making use of the extract function algorithm, as shown in *Fig 4.18*. This algorithm first sets Temp to the first part of MainString up to position Start. To this is added the InsertString, and then the remainder of MainString is concatenated with the variable Temp. As this is a function,

```
BEGIN Insert(Start,InsertString,MainString)
  Temp = Extract(1,Start-1,MainString)
  Temp = Temp + InsertString
  Length = Length of MainString
  Temp = Temp + Extract(Start,Length,MainString)
  Insert = Temp
END Insert
```

Fig 4.18

Insert algorithm using the Extract function.

the function name Insert is finally set to the string contained in Temp. For example, `Insert(7,"lovely ","Hello people")` returns "Hello lovely people".

SET 4B

1. Adding a small number of items to an existing sorted list is best accomplished using:
 - (A) a bubble sort.
 - (B) a selection sort.
 - (C) an insertion sort.
 - (D) a pack of cards.
2. Continuously finding the smallest or largest item in a list is the major concept in:
 - (A) a bubble sort.
 - (B) a selection sort.
 - (C) an insertion sort.
 - (D) a binary search.
3. Swapping pairs of out of order items occurs during:
 - (A) a bubble sort.
 - (B) a selection sort.
 - (C) an insertion sort.
 - (D) a quick sort.
4. A parameter can be best described as:
 - (A) a value set by one module that can be accessed by other modules.
 - (B) a value or variable that provides the input to another module.
 - (C) a variable that provides a location for the output from a module.
 - (D) A value or variable that is sent and returned from a module.
5. Making a copy of a section of a string is called:
 - (A) inserting.
 - (B) deleting.
 - (C) extracting.
 - (D) concatenation.
6. The original string will be smaller after:
 - (A) inserting.
 - (B) deleting.
 - (C) extracting.
 - (D) concatenation.
7. Joining 2 or more strings involves the process of:
 - (A) inserting.
 - (B) deleting.
 - (C) extracting.
 - (D) concatenation.
8. The two assignment statements $X=Y$ and $Y=X$ would result in:
 - (A) the contents of X and Y being swapped.
 - (B) both X and Y containing the original value of X.
 - (C) both X and Y containing the original value of Y.
 - (D) no change to the values held in X and Y.
9. Variables that can hold either the value True or the value False are known as:
 - (A) flags.
 - (B) arrays.
 - (C) numeric variables.
 - (D) Boolean variables.
10. An array is being sorted. After examining each item in the array the largest item is known to be in its correct position. This sort could be a(n):
 - (A) insertion or bubble sort.
 - (B) insertion or selection sort.
 - (C) selection or bubble sort.
 - (D) insertion, selection or bubble sort.
11. Make a list of all the students in your roll call class. Cut up the list so that each person's name is on a separate slip of paper. Shuffle the slips of paper. Sort these slips using each of the three sorting strategies studied in this chapter: that is, first sort using an insertion sort, then shuffle again and sort using a selection sort, and finally repeat the process using a bubble sort.
12. What changes need to be made to the insertion sort in *Fig 4.8* so that the array will be sorted into descending order?
13. In your own words describe the sorting strategy employed when using an insertion sort, a selection sort, and a bubble sort.
14. 'Extracting, deleting and inserting can be likened to the copy, cut and paste commands in many software applications'. Explain this statement. Use examples to justify your explanation.
15. An existing array contains a sorted list of all of a company's clients' names. Every day or so, a new client needs to be added to this array in the correct position. Develop an algorithm that will accomplish this task.

CUSTOM-DESIGNED LOGIC USED IN SOFTWARE SOLUTIONS

Before commencing work on an algorithm, it is necessary to know the inputs and the desired outputs, together with an idea of the processing required to transform the inputs into the outputs. This information is often obtained from models of the system, in particular dataflow diagrams and IPO charts. Once the inputs, processing and outputs are determined, work commences on designing data structures and assigning the appropriate data types. This information is documented using a data dictionary. Finally, an algorithm that describes the logic of the solution is formulated.

DATA STRUCTURES AND FILES

Efficient and appropriate data structures greatly assist in the subsequent development of an algorithm. In this section, we examine some common data structures - multi-dimensional arrays, and arrays of records. We also examine sequential and random access files.

Multi-Dimensional Arrays

In the Preliminary course, we examined single-dimensional arrays. Remember that all items in an array must be of the same data type. An example of this would be an array that is used to store the different names of people, up to 1000 items. That is: Name(1), Name(2), Name(3)... Each of these items must hold the same type of data, and in this example the type would be a string. Multi-dimensional arrays are also of one data type. Rather than one dimension or index, multi-dimensional arrays can have a number of indexes.



Index

An integer value used to denote a particular data item held in an array. Also called a dimension or subscript.

Often the dimensions or indexes for an array are called subscripts, because of their similarity to subscripts commonly used in mathematics. The dimensions of an array are usually given a name that describes their purpose. An example of this would be an array that is used to store the assessment task results of your class.

AssessResults(StudentNum, TaskNum)

Let us examine some possible data that could be held in this array. AssessResults(11,4)=72. This means that student number 11 on the 4th assessment task scored 72 marks. The names given to the indexes are only for documentation purposes. The following code fragments are functionally identical:

Table = 5

Chair = 4

AssessResult(Table,Chair) = 85

Dog = 5

Cat = 4

AssessResult(Dog,Cat) = 85

Two-dimensional arrays can be visualised as tables where each cell in the table holds data of the same type. Usually the indexes are used as a way of grouping the data. In the AssessResults array, StudentNum is the first index and groups the data according to individual students. TaskNum, the second index or dimension, groups the data by the task number.

		Assessment Task Number				
		1	2	3	4	5
Student Number	1	60	65	67	68	69
	2	65	73	70	62	71
	3	72	60	58	71	69
	4	85	86	80	82	66
	5	68	74	85	90	88
	6	58	55	52	59	62
	7	70	75	70	71	69

Fig 4.19

Two-dimensional arrays can be represented as tables.

Three-dimensional arrays can be thought of as a cube or a rectangular prism of data where each block contains a data item. Let us extend the AssessResults example so we can store data about all the classes in Year 12.

AssessResults(ClassNum, StudentNum, TaskNum)

If your SDD class is class number 6, your student number is 14, and you have just completed the 2nd assessment task then AssessResults(6,14,2) would be used to store your result in this assessment task.

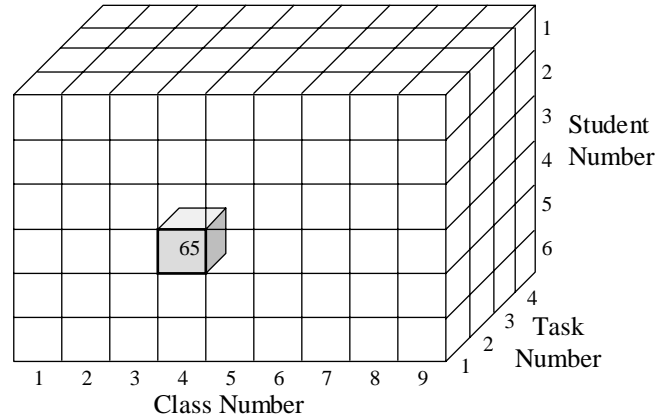


Fig 4.20

Three-dimensional arrays can be represented as rectangular prisms.

In the above diagram AssessResults(4,4,1) holds a value of 65.

Notice that each block is determined by a unique combination of ClassNum, StudentNum and TaskNum, and that the order of the indexes is very important. A particular ClassNum allows access to all the data about that class. For example, if ClassNum is held constant at 6, then altering the student and task numbers will access all the assessment results for Class 6. Similarly, holding a StudentNum constant provides all the assessment results for that particular student.

Let us now extend our array to 4 dimensions. In our example the 4th dimension will represent the year level, which can take values from 7 through to 12.

AssessResults(YearLevel, ClassNum, StudentNum, TaskNum)

Remember, each item stored in this array is a mark from 0 to 100: that is, each data item is of the same type. An example data item could be:

AssessResults(9, 4, 16, 2) = 68

This element of the array contains a mark of 68 obtained in the 2nd task by student number 16 who is in the 4th year 9 class. Note that in this structure there can be a student number 16 in each of Years 7, 8, 9, 10, 11 and 12. The important concept is that each combination of indexes *must* be unique.

Let us now look at some algorithms that use our AssessResults array. We will assume there is a maximum of thirty students in a class and that a mark of -1 denotes a result not to be included.

Following is an algorithm to find the class average for a particular task.

```

BEGIN ClassTaskAverage
  Get YearLevel,ClassNum,TaskNum
  StudentNum=1
  Total=0
  NumStudents=0
  WHILE StudentNum <=30
    IF AssessResults(YearLevel,ClassNum,StudentNum,TaskNum) >=0 THEN
      Add 1 to NumStudents
      Total=Total+AssessResults(YearLevel,ClassNum,StudentNum,TaskNum)
    ENDIF
    Add 1 to StudentNum
  ENDWHILE
  IF NumStudents > 0 THEN
    Average=Total/NumStudents
    Display Average
  ENDIF
END ClassTaskAverage
    
```

Fig 4.21

Algorithm to calculate the class average for a particular assessment task.

Notice that a particular task for a particular class, is determined by the three indexes YearLevel, ClassNum and TaskNum. These values are obtained whilst the StudentNum is variable. Varying the student number index, StudentNum, allows access to all the marks achieved on this particular task.



GROUP TASK Discussion

The above algorithm ignores array elements that contain -1. How is this achieved?



GROUP TASK Discussion

It is possible for more than one class to have the same class number. Explain why this is reasonable. Use examples to explain your answer.

The algorithm that follows finds the average on all tasks completed by a particular year level, assuming a maximum of 100 classes per year level, 30 students per class and 10 tasks per class.

```

BEGIN YearLevelAvg
  Get YearLevel
  Total=0
  NumStudents=0
  ClassNum=1
  WHILE ClassNum<=100
    StudentNum=1
    WHILE StudentNum<=30
      TaskNum=1
      WHILE TaskNum<=10
        IF AssessResults(YearLevel,ClassNum,TaskNum,StudentNum)>=0
          Add 1 to NumStudents
          Add AssessResults(YearLevel,ClassNum,TaskNum,StudentNum) to Total
        ENDIF
        Add 1 to TaskNum
      ENDWHILE
      Add 1 to StudentNum
    ENDWHILE
    Add 1 to ClassNum
  ENDWHILE
  IF NumStudents >=0 THEN
    Average=Total/NumStudents
    Display Average
  ENDIF
END YearLevelAvg

```

Fig 4.22

Algorithm to find the average mark for all results for a particular year level.



GROUP TASK Discussion

There are three loops in the above algorithm. What is the purpose of these loops?



GROUP TASK Discussion

Array elements that do not contain marks have a value of -1. Design an algorithm that will initialise all the array elements to -1.

Arrays of Records

In the Preliminary course, we examined records and determined that a record is a data structure that contains a number of fields. Each of the fields contained in a record can be of a different data type. Once a record is declared, it becomes a data type in the same way as any other predefined, simple data type.

For example, a record used to store Name, Address and Date of Birth details could have the following fields:

Field Name	Data Type
Surname	String
Cname	String
Title	String
Street	String
Suburb	String
Postcode	String
DOBDay	Integer
DOBMonth	Integer
DOBYear	Integer

If a record called MyDetails is created then data could be assigned as follows:

```
MyDetails.Surname = "Nerk"
MyDetails.Cname = "Fred"
MyDetails.Title = "Mr"
MyDetails.Street = "12 Foogle Road"
MyDetails.Suburb = "Fendalton"
MyDetails.Postcode = "1234"
MyDetails.DOBDay = 25
MyDetails.DOBMonth = 6
MyDetails.DOBYear = 1988
```

If a second record of the same type called OtherDetails was created, and it was also to contain all of Fred Nerk's details, then the following assignment statement would achieve this:

```
OtherDetails = MyDetails
```

Records can be treated as single data types, or each of the fields contained within a record may be accessed independently. Let us use an example of the creation of an address book that contains the details of 100 people. Each record could be created individually, however the process would be tedious and is unnecessary. It would be far more efficient to create an array of these records instead. The array could be called MyContacts. If the 40th person's address details needed to be accessed, then the following items would be examined:

```
MyContacts(40).Street
MyContacts(40).Suburb
MyContacts(40).Postcode
```

If the 3rd contact needed to be swapped with the 4th contact, then the following would achieve this:

```
Temp = MyContacts(3)
MyContacts(3) = MyContacts(4)
MyContacts(4) = Temp
```

An array of records can be likened to a table (or relation) in a database. If the MyContacts array of records were viewed as a table, it would look like the following:

Index	Surname	CName	Title	Street	Suburb	Postcode	DOBDay	DOBMonth	DOBYear
1	Davis	Sam	Mr	4 Data St	Romany	1234	14	5	1988
2	Hart	Rob	Mr	1 Cap Ln	Billings	9876	29	10	1951
3	Jackson	Jack	Mr	7a Pot Rd	Potfield	8765	13	9	1983
4	Unders	Wilma	Mrs	18 May St	Kingsley	1005	30	1	1977
5	Milson	Margaret	Miss	17 Jax Cr	Watson	5633	23	6	1966



Consider the following:

An address book such as the one described above is being created in Visual Basic. The following fragment of code is used to declare the array of records MyContacts:

```
Type Address
  Surname As String * 30
  Cname As String * 30
  Title As String * 4
  Street As String * 40
  Suburb As String * 30
  Postcode As String * 4
  DOBDay As Integer
  DOBMonth As Integer
  DOBYear As Integer
End Type

Dim MyContacts(1-100) As Address
```

Firstly, a new data type called Address is defined using the Type statement. An array called MyContacts is then declared with index values from 1-100 and of data type Address.



GROUP TASK Research

Examine at least two other programming languages and create declaration statements that achieve the same affect as the Visual Basic code above.



GROUP TASK Activity

Declare the above array of records in a programming environment with which you are familiar. Create and implement an algorithm to enter data into this structure.

An array of records can also be multi-dimensional in the same way as a simple data type array. In addition, records may also contain fields that themselves are arrays.



Consider the following:

A data structure is required to store qualifying data on cars entered for an upcoming motor racing event. Some sample data is given below:

Number	Model	Driver	Lap_1	Lap_2	Lap_3	Lap_4	Lap_5
25	Holden	Davis	185.2	180.3	181.1	179.9	179.7
05	Holden	Walker	183.3	181.7	179.5	184.9	180.2
17	Ford	Harrison	179.4	181.8	183.5	178.6	182.7

An individual record could contain the following fields:

Number String
 Model String
 Driver String
 Lap(LapNum) Array of real numbers

An array of records of this data type would be used to store the qualifying details for all the cars entered. The array could be called *Qualifying*. Let us examine the contents of some items in this array of records:

Qualifying(1).Number = "25"
 Qualifying(1).Driver = "Davis"
 Qualifying(1).Lap(3) = 181.1
 Qualifying(2).Lap(5) = 180.2

The data structure chosen for a particular problem must be designed with consideration given to the processing required in the algorithm. Selecting an appropriate data structure will often greatly reduce the complexity of the final algorithm. In the above *Qualifying* data structure, calculating the fastest lap time for each car is a much simpler task because the lap times are in an array which is part of each car's record.

```
BEGIN CalculateFastestLaps
    Count = 1
    WHILE Count <= Total cars
        Num = 2
        Fastest = 1
        WHILE Num <= 5
            IF Qualifying(Count).Lap(Num) < Qualifying(Count).Lap(Fastest) THEN
                Fastest = Num
            ENDIF
            Num = Num + 1
        ENDWHILE
        Display Qualifying(Count).Driver, Qualifying(Count).Lap(Fastest)
        Count = Count + 1
    ENDWHILE
END CalculateFastestLaps
```

Fig 4.23

Algorithm to calculate and display the fastest lap time for each car.

**GROUP TASK Discussion**

Describe in words the operation of the above algorithm. In particular describe how the lap times are accessed from the Qualifying array.

**GROUP TASK Activity**

Desk check the above algorithm using the sample data in the associated table.

Files

A file, in terms of software development, means a collection of data that is stored in a logical manner. Normally, files are stored on secondary storage devices, usually a hard disk that is separate from the application program itself. There are essentially two methods of storing data in a file:

- Sequential
- Random or Relative

Sequential Files

Data is stored in sequential files in a continuous stream. The data must be accessed from beginning to end. An audiocassette is a sequential storage medium for audio data. To play the third track on an audiocassette requires fast forwarding through tracks one and two. Sequential files operate in the same manner. To access data stored in the middle of a sequential file requires reading all the preceding data.

The data stored in a sequential file may have some structure, however the structure is not stored as part of the file. Applications that access sequential files must know about the structure of the file. Text files are sequential files; the data within a text file is merely a collection of ASCII characters.

When using sequential files it is necessary to structure the data yourself. Sentinel values are used to indicate logical breaks in the data. For example, tab characters may be used between fields and carriage return characters between records. Often a particular string, such as "ZZZ", may be used as a sentinel to indicate the end of a sequential file.

**Sentinel value**

A dummy value or character used to indicate the end of a logical data stream within a file. Sentinel is related to the word sentry, a sentry being a guard who prevents passage of unauthorised persons.



*Fig 4.24
Audiocassettes store music tracks sequentially.*



Consider the following:

A sequential file contains a list of animal names separated by commas and ending with the sentinel value "ZZZ". To access a particular animal name requires reading each preceding animal name. A particular animal name cannot be directly accessed.

A possible animal name file follows:

Ant, Hen, Monkey, Pig, Rat, Cow, Tiger, Warthog, Dog, Elephant, Frog, Goat, ZZZ

If we wish to read this file into an array called Animals, we need to first open the file, then read and examine the value of each character one at a time. Commas tell us when we have reached the end of a word, and ZZZ is the sentinel value that indicates the end of the file.

```

BEGIN ReadSequentialFile
  Open the file
  Set Index to 1
  WHILE Animals(Index) does not equal "ZZZ"
    Read a character from file
    IF Character is a comma THEN
      Add 1 to Index
    ENDIF
    Animals(Index) = Animals(Index) + Character
  ENDWHILE
END ReadSequentialFile
    
```

Fig 4.25
Algorithm to read the animal name sequential file.



GROUP TASK Activity

There is a minor error in the above algorithm. Perform a desk check of the algorithm using the sample file and determine the error. Suggest a possible way of correcting this error.



GROUP TASK Activity

The above algorithm reads the sentinel value "ZZZ" into the array Animals. Redesign the algorithm so that the sentinel value is not read into the array.



GROUP TASK Activity

Most programming languages now allow you to read an entire line of text from a sequential file, a line being a string of characters that ends with a carriage return. Design an algorithm that would read the above file if each comma in the file was a carriage return.

Random or relative files

The word random, in this context, means in any order. Data stored in a random access file can be accessed in any order. Relative refers to the fact that there is a structure to these files. Each record within the file possesses characteristics common to all records in the file. These characteristics form part of the file: that is, the structure of the file is an integral part of the file.

Random access or relative files can be likened to audio compact discs. With a music CD, individual tracks can be played in any order. Many CD players include a random function whereby tracks are played in a random order. The structure of an audio CD is stored on the CD. If you wish to play track five, the laser head jumps directly to the start of track five. Random access files allow individual data items to be accessed directly without the need to read any of the preceding data. In fact, random access files are often known as direct access files.



Fig 4.26
Audio compact discs
provide random access
to individual tracks.

Random access files are used to store records. Each record of a particular data type must be of precisely the same length. Strings that are of differing lengths are usually padded out using the blank character (ASCII code 32). Individual fields within records can be identified precisely because the exact length and structure of each record is known.



Consider the following:

A random access file is used to store the details of products sold by a particular business. The size of the fields within each record, are described in Fig 4.27. The number of characters in the four string fields total $18 + 30 + 5 + 5 = 58$ characters. Each character requires one byte of storage space, plus another 2 bytes to store the cost, giving a total of 60 bytes. Every record in this file is therefore 60 bytes long.

Name	String of length 18
Description	String of length 30
SupplierID	String of length 5
ProductID	String of length 5
Cost	16 bit Integer

Fig 4.27
Record definition for the products file.

Suppose we wish to access the Description of the ninth record. The system calculates the byte number where this data is stored: eight records times 60 bytes gives 480 bytes, that is, record number 9 begins in position 481. These calculations are transparent from the point of view of the programmer. The programmer refers to the record by its index, nine in this case. Once the record is read into memory, the description field can be accessed. Most high-level languages require retrieval of complete records. The keyword Get is commonly used to retrieve records, and Put is used to store records.



GROUP TASK Investigation

Examine the documentation of a programming language with which you are familiar. Create a list of commands available for accessing random access files. Describe the function of each command you find.

SET 4C

1. Multi-dimensional arrays are useful for storing:
 - (A) a number of data items of different types that need to be accessed together.
 - (B) data of the same type that possesses a number of characteristics by which it can be grouped.
 - (C) X and Y style coordinates that can later be graphed.
 - (D) documentation information that will later assist with upgrades and program maintenance.
2. A value used to indicate the end of a data stream is called:
 - (A) a sentinel value.
 - (B) an end of file (EOF) character.
 - (C) a flag.
 - (D) a driver.
3. In the expression Products(Count), Count would most likely be:
 - (A) a variable of type string.
 - (B) the index for the array.
 - (C) a variable of type integer.
 - (D) an array of strings.
4. The ability to retrieve a particular record from a file is a feature of:
 - (A) text files.
 - (B) sequential files.
 - (C) random access files.
 - (D) binary files.
5. The names given to the indexes in a multi-dimensional array:
 - (A) are used to determine the data type of the array.
 - (B) can also be called dimensions or subscripts.
 - (C) define the names of the variables that must be used to access data items held in the array.
 - (D) are used for documentation purposes.
6. Fields within a record:
 - (A) must hold data of the same type.
 - (B) can be arrays.
 - (C) can themselves be records.
 - (D) All of the above.
7. Before work can commence on the development of algorithms it is vital to:
 - (A) make sure the system model is logically correct in every detail. It must explain every data structure the algorithm will require.
 - (B) know the details of the original problem in its entirety.
 - (C) examine the hardware on which the final product will operate.
 - (D) know the inputs and outputs, have an idea of the processing required, and have designed the data structures.
8. A sequential file uses a comma to separate fields. This is a poor choice of separator when:
 - (A) the length of each data item varies.
 - (B) the data itself may contain commas.
 - (C) numeric data is stored in the file.
 - (D) each data item is the same length.
9. An array of records:
 - (A) can only have one index.
 - (B) contains records that are all of the same type.
 - (C) can only be accessed sequentially.
 - (D) is used to store a single list of data of a particular type.
10. The expression Products(5).Price can be best described as accessing:
 - (A) the fifth record in the array Products.
 - (B) the field called Price in the Products record.
 - (C) the fifth Price in the array called Products.
 - (D) the Price of the fifth record stored in the array Products.
11. If A=5, B=3, C=1 and D=6, what can you say about the data items accessed by each of the following expressions? Item(A,B,2), Item(D-C,3,A-B), and Item(2B-C,3C,D/B).
12. A teacher's class roll is used to store the personal details of each student in the class together with absence data on each student for each school day. Design a data structure that could be used to hold this information.
13. The class roll data from question 12 needs to be stored in a file. Would you use a sequential or a random access file for this purpose? Explain the advantages and disadvantages of each method of file storage in regard to this scenario.
14. Your local council is collecting statistics on the number of vehicles using your street each day. The data is collected using a device that automatically increments when a car passes. The council wishes to analyse the data in terms of the total traffic per day, per week and per month over a 12-month period. Design a data structure to store this data in a way that allows for easy calculation of the required totals.
15. Using your data structure developed in question 14, design an algorithm that will output, in table form, the daily, weekly and monthly traffic flow.

CREATING CUSTOM-DESIGNED ALGORITHMS

Once a model of the system has been created and data structures have been designed, the work of creating the algorithms can commence. Many solutions will involve the use of existing, standard modules such as the searches and sorts examined earlier in this chapter. Often the most difficult task for software developers is the creation of custom algorithms. It is difficult to learn how to write custom algorithms, as every problem presents new difficulties and challenges. Often many attempts will be necessary before a successful algorithm is developed. Some consolation can be had in the knowledge that the more algorithms you write, the easier it will become.

Let us now go through the process of creating some custom algorithms:

The card game 'snap'

In this section, we will create a series of algorithms to simulate the card game snap. Snap utilises a standard pack of playing cards. There are 4 suits: hearts, diamonds, spades and clubs. Each suit contains 13 cards: Ace, 2, 3, 4...10, Jack, Queen and King - a total of 52 cards. For this simulation, the game is played with two players. Each player in turn takes a card from their half of the pack and places it on the stack in the middle of the table. If the card placed has the same value as the previous card, the first player to put their hand on the stack wins the stack. The stack of cards is placed under the player's existing cards and the game continues. The winner is the player who eventually wins all the cards.

The following structure diagram models this game:

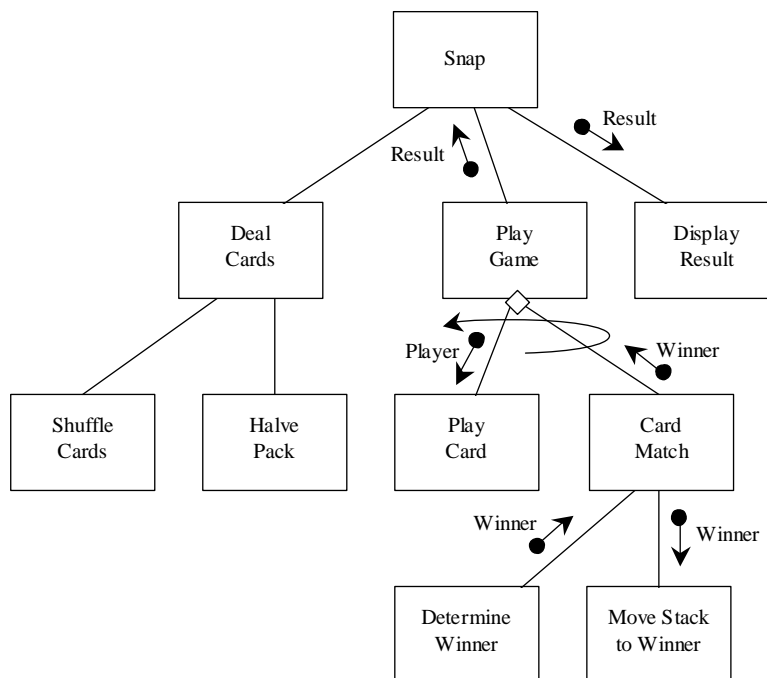


Fig 4.28

Structure diagram for the 'snap' simulation.

Data structures for the card game ‘snap’

We now need to think about and develop the data structures to be used. There are a total of 52 cards and each card will always be in one of three places throughout the game: either with one of the two players or in the centre stack.

Initial thoughts are that we could create three separate arrays, one for each stack. Each card would be represented by a data item in one of the three stacks. For example, $\text{Player1}(5) = 3H$ would mean that the fifth card in player 1’s stack is the 3 of hearts. Each stack can contain a maximum of 52 items, as there are 52 cards in the pack. To win, a player’s stack or array needs to be full.

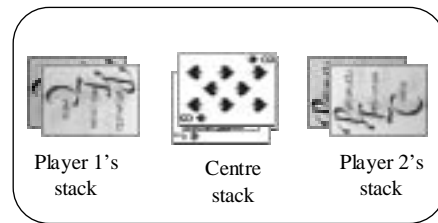


Fig 4.29
Three stacks that together contain all 52 cards in the pack.

A second approach would be to create a single array of 52 items, with the 13 hearts in order from index 1 to 13, followed by the diamonds, spades and clubs. The index for the array is used to represent each card in the pack and the data stored in the array is used to assign each card to a particular stack. For example $\text{Card}(7) = 0$ could mean the 7 of hearts is in the centre stack. $\text{Card}(24) = 1$ could mean the 24th card, which is the 11th card in the diamonds suit, the Jack of diamonds, is in player 1’s stack. For player 1 to win requires that all items in the array contain a value of 1.

After considering the processing that will be required, a better structure is discovered that combines aspects of each of the above structures. We use one array containing 52 items, called, say, *Cards*. The data held in this array is a number that corresponds to each card in the pack. The index for the array indicates the order in which the cards are held in each stack. For example, $\text{Cards}(18) = 30$ means that the 18th card in the stacks is card 30. As $30 - 26 = 4$, this card is the 4 of spades. Variables are maintained to store the start position of each of the three stacks within the array *Cards*. For example, if $\text{Stack0} = 1$, $\text{Stack1} = 12$ and $\text{Stack3} = 40$, then the centre stack is *Cards*(1) to *Cards*(11), player 1’s stack is *Cards*(12) to *Cards*(39), and player 2’s stack is *Cards*(40) to *Cards*(52). For player 1 to win, Stack1 must have a value of 1 and Stack0 and Stack2 must have values of zero. A small refinement to improve the efficiency is to create an array called *Stack* that contains 3 items indexed from 0 to 2. Our previous Stack0 , Stack1 and Stack2 become $\text{Stack}(0)$, $\text{Stack}(1)$ and $\text{Stack}(2)$.

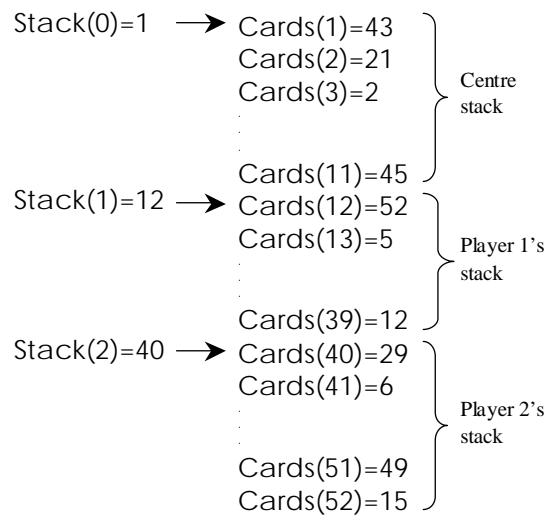


Fig 4.30
Example of the final data structure for ‘snap’.

These 2 arrays, *Cards* and *Stack*, will be required by most of the modules in the final solution, therefore we declare them as global variables. Individual variables within modules should be local to the particular module.

Algorithms for the card game 'snap'

Using the structure diagram in *Fig 4.28*, we can commence designing the algorithms. The main program is quite straightforward as no processing occurs, merely the calling of lower-level modules. Notice that both `PlayGame` and `DisplayResult` include the passing of the parameter `Result`. The module `DealCards` is equally simple: it merely calls the modules `ShuffleCards` and `HalvePack`.

Let us now work on the `ShuffleCards` module. In the real game, this involves manually manipulating the cards into a random order. In our simulation we need to assign a unique card number to each element of our array `Cards`.

The numbers randomly assigned must be in the range 1 to 52, and no number can be repeated. If a 5 were to exist in the array twice, this would mean we had two 5 of hearts, which would be unacceptable.

Let us ignore for the moment the fact that the numbers need to be unique, and concentrate on generating them in the range 1 to 52. We can simulate the random effect using a random number generator. In most programming environments the `RND` function generates a random number between 0 and 1. For example, the statement `Number = RND` may set `Number` to 0.345419688 the first time it is executed, and 0.932156944 the next time. Each subsequent call to the function `RND` generates a new random number.

How can we generate numbers from 1 to 52 using this function? If we multiply the random number generated by 100 we will get a result within the range 0 to 100. For example, 0.345419688 times 100 equals 34.5419688, and 0.932156944 times 100 equals 93.2156944. If we chop off the fractional part of these values we now have integer results from 0 to 99. For example, 34.5419688 becomes 34 and 93.2156944 becomes 93. Most programming languages contain an integer function to perform this process, usually called `INT` e.g. `INT(93.2156944) = 93`. In general, we have performed `INT(RND * 100)` which results in a whole number, or integer, in the range 0 to 99. The range 0 to 99 contains 100 different values. We want only 52 different values so we multiply `RND` by 52 rather than 100. As before, we then take just the whole number or integer part: `INT(RND * 52)`. If the random number generated is 0.000001, then 0 results from the functional expression `INT(RND * 52)`; if 0.9999999 is generated, then 51 is the result. The range is 0 to 51. If we add one each time, the range is now correctly 1 to 52. The final statement will be `INT(RND * 52) + 1`.



Random number generators

Used to generate a number in the range 0 to 1 e.g. 0.562314. If `RND` is the function's name then the expression `INT(RND*NumItems)+Start` will generate integer values from `Start` to `NumItems+Start-1`.

```
BEGIN MAINPROGRAM
  DealCards
  PlayGame(Result)
  DisplayResult(Result)
END MAINPROGRAM

BEGIN DealCards
  ShuffleCards
  HalvePack
END DealCards
```

Fig 4.31

Main program and DealCards module for the game "snap".

We need to generate the card values uniquely. Therefore, before storing each value in the array Cards we must check it does not already exist in the array. The algorithm that follows includes the function CardExists, which checks the array for a particular card and returns the value True if the card is found. The card is only added to the array Cards if CardExists returns the value False.

```

BEGIN ShuffleCards
  Index = 1
  REPEAT
    Temp = INT(RND*52)+1
    IF CardExists(Temp) is False THEN
      Cards(Index) = Temp
      Increment Index
    ENDIF
  UNTIL Index > 52
END ShuffleCards

BEGIN CardExists(CardNum)
  CardExists = False
  Index = 1
  REPEAT
    IF Cards(Index) = CardNum THEN
      CardExists = True
      Index = 52
    ENDIF
    Increment Index
  UNTIL Index > 52
END CardExists
    
```

Fig 4.32 Algorithms for the ShuffleCards and CardExists modules of the game 'snap'.

To maintain the integrity of the documentation of the original structure diagram in Fig 4.28, a branch leading from ShuffleCards containing the CardExists function needs to be added. Maintaining the integrity of your documentation is vital at all stages of the software development cycle.

Remember that variables within a module are local to that module, so the identifier Index can be used in both the ShuffleCards module and then again in the CardExists function. These two occurrences of the identifier Index are completely different; one does not affect the other in any way.



GROUP TASK Discussion
 A parameter is used as part of the call to the CardExists function above. With reference to the two modules above, discuss the term 'parameter passing'.



GROUP TASK Activity
 Design a driver algorithm that could be used to test the ShuffleCards module. The algorithm should output the values in the array Cards.

The HalvePack routine involves setting the values in the Stack array. This effectively gives player 1 the first 26 cards and player 2 the last 26 cards.



Driver
 In terms of software development, a driver is a module written to test the operation of another module. Commonly, drivers set any necessary values, the module to be tested is called, and the results of the call to the module are output.

```

BEGIN HalvePack
  Stack(0) = 0
  Stack(1) = 1
  Stack(2) = 27
END HalvePack
    
```

Fig 4.33 The HalvePack module for the game 'snap'.

We now consider the PlayGame module. The original structure diagram (Fig 4.28) provides us with rich information in regard to the logic of this module. It contains a loop that calls the PlayCard module or the CardMatch module. In other words, either a player plays a card or a match is detected and dealt with. The parameter Result is used to send the overall winner back to the main program. A preliminary design based on this information is shown in Fig 4.34. This design is not quite complete but does include all the information from the structure diagram.

A number of questions arise:

- How do we detect matching cards?
- How do we determine when a player has won the game?
- How do we know whose turn it is?

These questions must be solved within the PlayGame module.

Detecting matching cards requires that we examine the previous card on the centre stack. If the previous card has the same value then we have a match. For example, if player 1 places the 5 of diamonds on the centre stack and then player 2 places the 5 of clubs on the centre stack, we have a match.

The 5 of diamonds has a value of 13 plus 5, which is 18 in our array Cards, and the 5 of clubs has a value of 13 times 3 plus 5, which equals 44. Eighteen divided by 13 leaves a remainder of 5, and 44 divided by 13 also leaves a remainder of 5. Modular arithmetic provides a simpler way of calculating these remainders, as the MOD function is included in most programming environments: $18 \text{ MOD } 13 = 5$, and $44 \text{ MOD } 13 = 5$. In general, $\text{CardNumber MOD } 13$ gives the face value of the card. There is one exception: the Kings, where values 13, 26, 39 and 52 will generate values of 0. This will not affect our game as all we need is for them to be the same.

Remember that our array Cards contains first the centre stack, then player 1's stack, and finally player 2's stack. The previous two cards played, in our example the 5 of clubs and the 5 of diamonds, must be in the first two positions within the Cards array. It is possible that no cards or only one card is/are in the centre stack, so we need to ensure Stack(1) has a value at least 2 more than Stack(0). Fig 4.36 shows an amended algorithm that incorporates these features.

```
BEGIN PlayGame(Result)
  WHILE No Result
    IF Match detected THEN
      CardMatch(Winner)
    ELSE
      PlayCard(Player)
    ENDIF
  ENDWHILE
END PlayGame
```

Fig 4.34

Preliminary pseudocode for the PlayGame module based on the structure diagram.

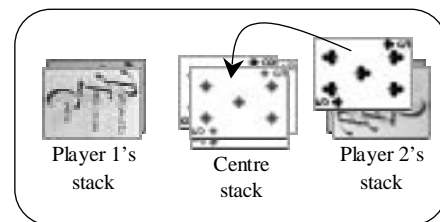


Fig 4.35

A match results when player 2 plays the five of clubs.

```
BEGIN PlayGame(Result)
  WHILE No Result
    IF Cards(1) MOD 13 = Cards(2) MOD 13
      AND Stack(1) >= Stack(0) + 2 THEN
      CardMatch(Winner)
    ELSE
      PlayCard(Player)
    ENDIF
  ENDWHILE
END PlayGame
```

Fig 4.36

The revised PlayGame algorithm to detect matching cards.

Determining when a player has won the game involves examining the array Stack. If player 1 has won, then Stack(0) = 0, Stack(1) = 1 and Stack(2) = 0. If player 2 has won then Stack(0) = 0, Stack(1) = 0 and Stack(2) = 1.

In fact we need only consider Stack(1) and Stack(2), as the centre stack, Stack(0), must be empty if either of the players has cards in position 1 of the Cards array. We need an efficient decision to do this. If we add the values in Stack(1) and Stack(2) we must get a sum of 1 for there to be a winner. If Stack(1) also contains a value of 1 then player 1 wins, otherwise player 2 wins. Let us develop this as a function that returns 0 meaning no winner, 1 meaning player 1 has won, and 2 meaning player 2 has won. The algorithm FinalWinner in Fig 4.37 performs this function.

```
BEGIN FinalWinner
  IF Stack(1) + Stack(2) = 1 THEN
    IF Stack(1) = 1 THEN
      FinalWinner = 1
    ELSE
      FinalWinner = 2
    ENDIF
  ELSE
    FinalWinner = 0
  ENDIF
END FinalWinner
```

Fig 4.37

Function to determine the final winner.

We need to remember to add this FinalWinner function to our structure chart to maintain the accuracy of our documentation. If we were using an integrated CASE tool, the module could automatically be added to the system model when created. An updated version of the PlayGame module is shown in Fig 4.38. The FinalWinner function becomes the decision to determine if the game should continue. Once a winner has been determined, the parameter Result is set to the winner's player number.

```
BEGIN PlayGame(Result)
  WHILE FinalWinner = 0
    IF Cards(1) MOD 13 = Cards(2) MOD 13
      AND Stack(1) >= Stack(0) + 2 THEN
      CardMatch(Winner)
    ELSE
      PlayCard(Player)
    ENDIF
  ENDWHILE
  Result = FinalWinner
END PlayGame
```

Fig 4.38

The revised PlayGame algorithm including calls to the FinalWinner function.

Finally, let us consider whose turn it is to play a card. Player 1 has the first turn, followed by player 2, then player 1 again. Turns continue to alternate until two cards match and a player puts his hand on the centre pile. The player who does this first wins the cards in the centre pile.

This winner is returned in the Winner parameter from the CardMatch module. This player has the next turn and then the alternating sequence continues. We need a function that turns a one into a two and a two into a one, so we can easily alternate players. 2 divided by 1 is 2, and 2 divided by 2 is 1, which is precisely what we require. The final PlayGame algorithm is given in Fig 4.39.

```
BEGIN PlayGame(Result)
  Player = 1
  WHILE FinalWinner = 0
    IF Cards(1) MOD 13 = Cards(2) MOD 13
      AND Stack(1) >= Stack(0) + 2 THEN
      CardMatch(Player)
    ELSE
      PlayCard(Player)
    ENDIF
    Player = 2 / Player
  ENDWHILE
  Result = FinalWinner
END PlayGame
```

Fig 4.39

The final PlayGame algorithm.



GROUP TASK Discussion

The parameter returned from the CardMatch module has been changed in the final PlayGame algorithm. What effect does this change have on the operation of the PlayGame algorithm?



GROUP TASK Discussion

If a player runs out of cards before a match is encountered, the other player continues to play their cards. Does the PlayGame algorithm deal with this possibility? If not, design a modification to deal with this scenario.

Finally, we will develop an algorithm for the Playcard module. We will leave the remaining algorithms for you to develop.

The PlayCard module receives the player number from the PlayGame module. The top card on the player’s stack is removed and added to the top of the centre stack. Changing the values in the Stack array is straightforward; rearranging the cards in the Cards array requires some manipulation.

Let us consider the sample data in Fig 4.40. If the PlayCard module is called with the statement PlayCard(2) then it is player 2’s turn. As player 2’s stack commences at position 32, then card 39 is played. Card 39 is the King of spades. In terms of the array Cards, the data held in Cards(32) is stored in a temporary location, all the items in the array above position 32 move down one place, and finally Cards(1) is assigned the value held in the temporary location, in this example, 39. We then adjust the values of Stack(1) and Stack(2) to reflect our changes. Fig 4.41 shows our algorithm.

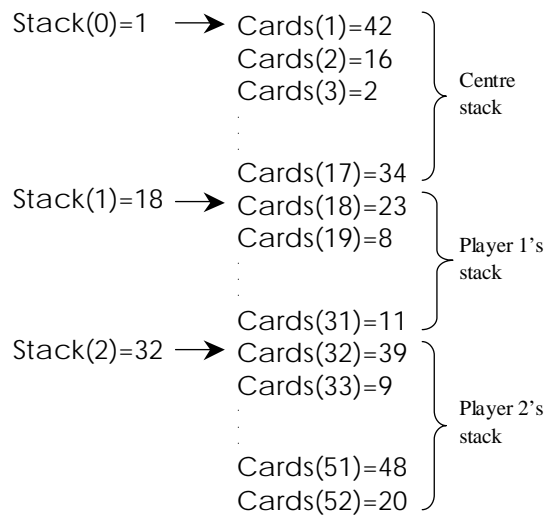


Fig 4.40
Sample data for “snap”.

```

BEGIN PlayCard(Player)
  TempCard = Cards(Stack(Player))
  Display TempCard
  MoveItemsDown(Stack(Player))
  Cards(1) = TempCard
  AdjustStacks(Player)
END PlayCard
    
```

Fig 4.41
The PlayCard algorithm.

We need to create the two modules MoveItemsDown and AdjustStacks called by this algorithm. Rather than go into detail, these two modules are described in Fig 4.42. Remember these modules need to be added to the structure diagram to ensure the integrity of the documentation.

```

BEGIN MoveItemsDown(Index)
  Count = Index - 1
  WHILE Count <= 1
    Cards(Count+1) = Cards(Count)
    Subtract 1 from Count
  ENDWHILE
END MoveItemsDown
    
```

```

BEGIN AdjustStacks(Player)
  Add 1 to Stack(1)
  IF Player = 2 THEN
    Add 1 to Stack(2)
  ENDIF
END AdjustStacks
    
```

Fig 4.42
Algorithms for the MoveItemsDown and AdjustStacks modules.

Developing Test Data

Developing suitable test data to ensure the correct operation of algorithms is an important aspect of algorithm development. Test data should test every possible route through the algorithm as well as testing each boundary condition. Testing each route makes sure that all statements are correct and that each statement works correctly in combination with every other statement. Testing boundary conditions ensures that each decision is correct. Commonly, a decision will be out by 1 as a result of an incorrect relational operator, for example, $Result < 10$ rather than $Result \leq 10$. Making sure your test data includes values where $Result = 10$ will lead to the discovery of the error.



GROUP TASK Discussion

Perform a desk check of the MoveItemsDown module using suitable test data of your own choice. What similarities can you see between this module and aspects of the insertion sort?

Thorough Documentation

Maintaining the integrity of all documentation is vital during the development of algorithms. This is particularly true when a team of developers are working on a software project. Later, when upgrades are under development, the documentation will provide important information to future developers. The revised structure diagram for the game 'snap' is shown in Fig 4.43 to assist in answering the group task that follows.

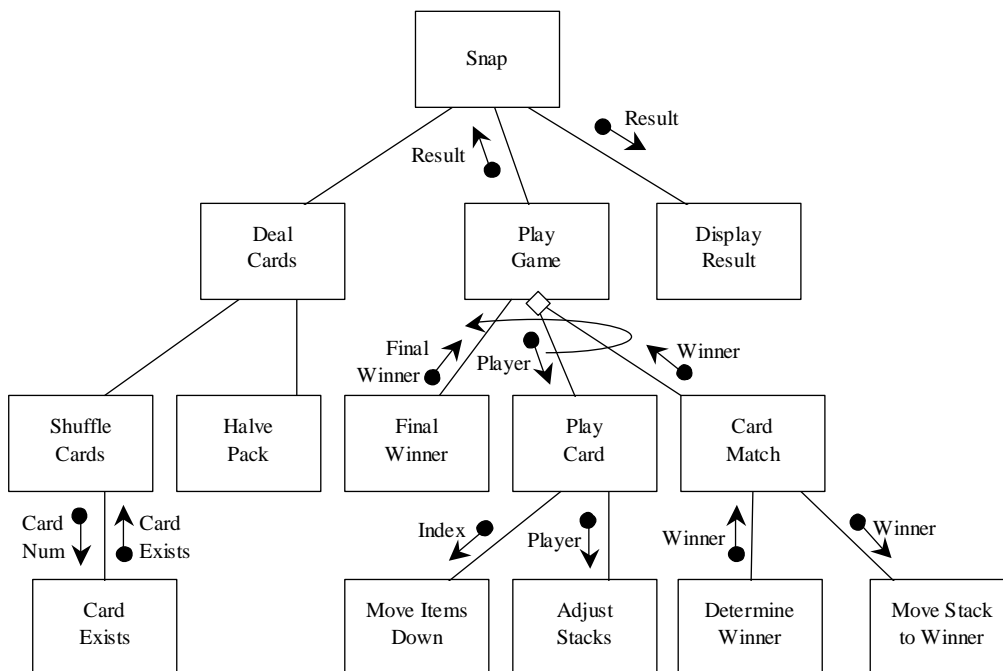


Fig 4.43 Revised Structure diagram for the 'snap' simulation.



GROUP TASK Activity

Develop algorithms for the remaining modules of the 'snap' card game i.e. CardMatch, DetermineWinner, MoveStackToWinner and DisplayResult. Revise the 'snap' structure diagram to reflect any new modules you include in your solution.

SET 4D

1. If RND generates a number in the range 0 to 1 then $\text{INT}(\text{RND} \times 16) + 2$ will generate integers in the range:
 - (A) 2 to 16.
 - (B) 0 to 15.
 - (C) 2 to 17.
 - (D) 3 to 19.
 2. Test data for algorithms needs to be designed to test:
 - (A) all paths through the algorithm and each boundary condition.
 - (B) for unexpected inputs.
 - (C) for syntax errors in the source code.
 - (D) All of the above.
 3. A variable used as a parameter in a call to a sub-program:
 - (A) must always have the same name.
 - (B) must be of the same type as the parameter.
 - (C) would normally be a global variable.
 - (D) All of the above.
 4. The identifiers or names used to declare local variables:
 - (A) must be unique within the entire program.
 - (B) cause errors if used in more than one module.
 - (C) can be used in multiple modules without affecting each other.
 - (D) cannot be used as parameters in procedure and function calls.
 5. Models of the system, such as dataflow diagrams and structure diagrams:
 - (A) should be updated as algorithms are developed.
 - (B) remain static once they have been created.
 - (C) are used to assist in the training of users.
 - (D) need to be revised to describe the logic of each algorithm.
 6. 'Algorithm creation is an exact science.' In what sense is this statement correct?
 - (A) There is only one solution to each problem.
 - (B) There are strict rules that, if followed, will allow the development of a correct solution.
 - (C) Correct algorithms are structured in a precise and logical manner.
 - (D) All of the above.
 7. A data structure needs to be available for access by all modules within a system. This data structure can be described as:
 - (A) a file.
 - (B) local.
 - (C) static.
 - (D) global.
 8. A module written specifically to test the operation of another module is called:
 - (A) a function.
 - (B) a stub.
 - (C) a driver.
 - (D) a parameter.
- Use the following pseudocode to answer questions 9 and 10.
- ```

BEGIN
 Count = 1
 WHILE Count < 6
 Get Value
 IF Value < Count THEN
 Value = Count
 ELSE
 Count = Count + 1
 ENDIF
 Display Value
 ENDWHILE
END

```
9. Perform a desk check of the above algorithm with the data items 7, 2, 1, 5, 3, 8, 9. What is the output from the desk check?
    - (A) 7, 2, 1, 5, 3, 8, 9
    - (B) 1, 2, 3, 4, 5, 6
    - (C) 7, 2, 3, 5, 4, 8, 9
    - (D) 7, 2, 1, 3, 5, 3, 4, 8, 9
  10. The best explanation of this algorithm is:
    - (A) A pre-test loop goes around 6 times. Each time a value is input, and the smaller of the value and the loop counter is output each time.
    - (B) A value is input within a loop that counts from 1 to 6. If the value is smaller than the loop counter then the value of the loop counter is output, otherwise the loop counter is incremented and the value input is output.
    - (C) A pre-test loop goes around 6 times. Each time a value is input, the loop counter is incremented and the larger of the value and the loop counter is output.
    - (D) A value is input within a loop that counts from 1 to 5. If the value is smaller than the loop counter then the value of the loop counter is output, otherwise the loop counter is incremented and the value input is output.

11. 'The development of a well-structured algorithm is a much more difficult task than implementing the logic in a particular programming language.' Do you agree with this statement? Justify your answer.

Questions 12 to 15 relate to the following scenario:

The board game 'chess' uses an 8 by 8 square board. Two players compete against each other using white and black pieces. Without going into the intricacies of the game, each player takes a turn at moving one of his or her pieces. Different types of piece can move in different ways. For example, bishops can move any number of places but only along diagonals. Rooks can move any number of places but only in a straight line. The queen can move both as a bishop and as a rook. Rooks, bishops and queens cannot jump over other pieces when moving. Pieces take the opposing player's pieces by moving onto their squares. You cannot move onto a square containing one of your own pieces.

A software simulation is under construction for this game. The main data structure used is a two dimensional array called Board. This array contains details of the position of all pieces on the playing board. For example, Board(2,4) = BR1 means that row 2 column 4 on the board contains black rook number 1. Board(5,6) = WQ means the white queen is currently in row 5 of column 6.

You, as a developer working for the company developing this product, have been assigned the task of developing a series of routines that ensure legal moves.

12. Create an algorithm for the function called RookCheck that checks that a move performed by a rook is legal. The function has four parameters and returns True or False as its output.

The format of a function call is as follows:

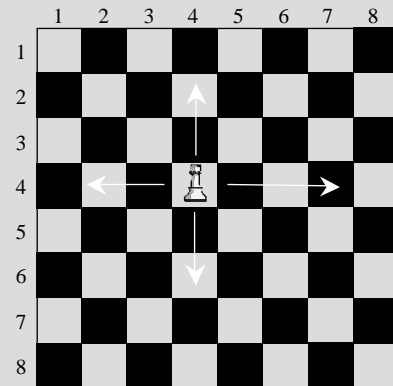
RookCheck(OldRow,OldCol,NewRow,NewCol)

13. Create an algorithm for the function BishopCheck to check that a move performed using a bishop is legal. The parameters for the function are to be the same as the RookCheck function, developed in question 12.

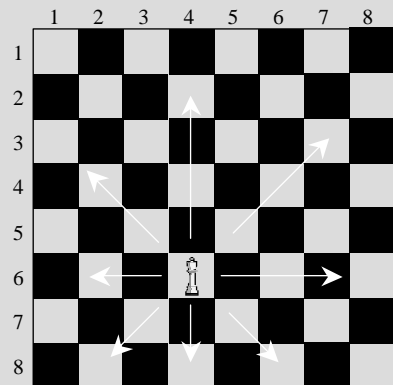
14. Create an algorithm for the function QueenCheck to check that a move performed using a queen is legal. Use calls to the RookCheck and BishopCheck functions as part of your solution.

15. Knights are able to move in an L-shaped pattern: one space horizontally then two spaces vertically, or two spaces horizontally then one space vertically. The knight is the only piece that may jump other pieces to reach its destination. The diagram at right shows all the possible moves for a knight in row 4 column 6.

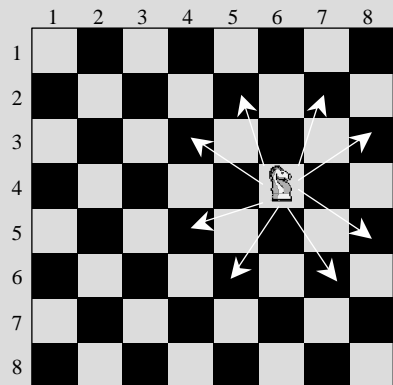
Create an algorithm for the KnightCheck function.



The rook can move any number of spaces but only in a straight line. This rook is in position Board(4,4).



The queen can move any number of spaces in any direction. This queen is in position Board(6,4).



The knight can move from Board(4,6) into any of the squares indicated by the arrows. Knights can jump other pieces.

## CUSTOMISATION OF EXISTING SOFTWARE SOLUTIONS

Many custom off-the-shelf packages allow customisation using a programming language included as part of the COTS package. These languages allow developers to include features not included in the original package.



Consider the following:

A school is developing a reports package to computerise the creation of reports to parents. The school wishes the package to automatically display the text *Above average*, *Average* or *Below average* based on the student's mark compared with the class' average. Marks within 5 marks of the average will be assigned the word *Average*.

The report package is being written in Microsoft Access. This COTS package does not provide the facility to complete this task directly, however it does provide a programming language: Visual Basic for Applications.

An example of the relevant data held in the database tables 'Reports' and 'Classes' is shown in Fig 4.44.

| Reports  |           |         |      |
|----------|-----------|---------|------|
| ReportID | StudentID | ClassID | Mark |
| 3158     | 00001     | 7DT     | 83   |
| 3162     | 00001     | 7EN1    | 62   |
| 3157     | 00001     | 7MA2    | 62   |
| 3156     | 00002     | 7DT     | 69   |
| 3161     | 00002     | 7EN1    | 83   |
| 3163     | 00002     | 7MA2    | 72   |
| 3159     | 00003     | 7EN2    | 44   |
| 3160     | 00003     | 7DT     |      |
| 3164     | 00003     | 7MA2    | 70   |

| Classes |          |         |
|---------|----------|---------|
| ClassID | CourseID | Average |
| 7MA1    | MA       | 65      |
| 7DT     | DT       | 67      |
| 7EN1    | EN       | 66      |
| 7MA2    | MA       | 60      |
| 7EN2    | EN       | 63      |

Fig 4.44

Sample data from a school's reports package.

An algorithm is developed to perform this task. This algorithm is then implemented as a function using Visual Basic for Applications. The function `AverageWord` then becomes part of the Reports package and can be used to generate the required text on the final reports.

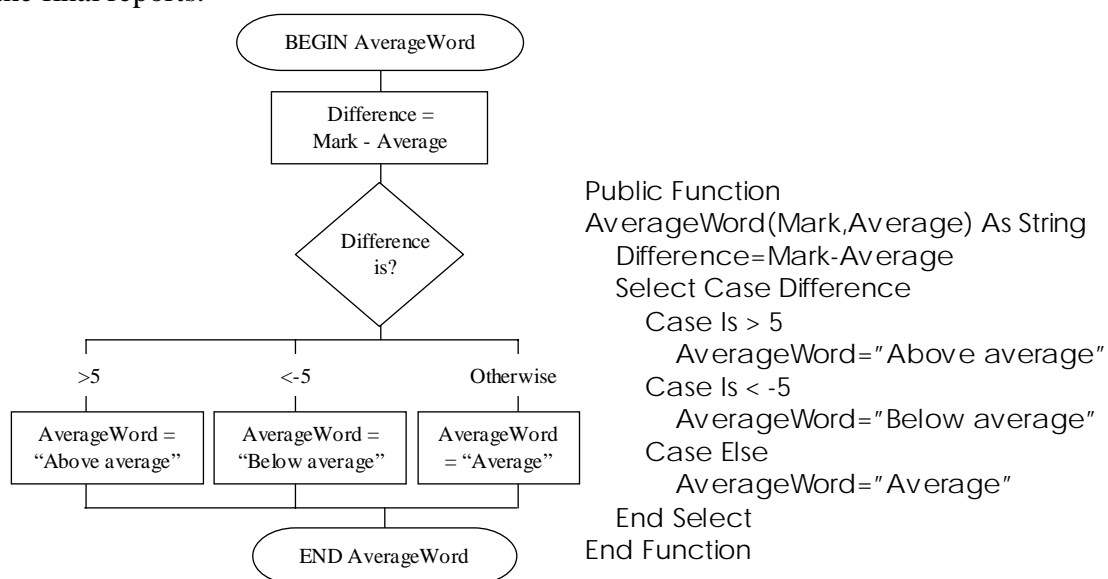


Fig 4.45

Algorithm and Visual Basic for Applications code for the `AverageWord` function.

## SELECTION OF LANGUAGE TO BE USED

Algorithms should be written in such a way that they could be implemented in a variety of languages. The most appropriate programming language should be selected after considering each of the following criteria:

- Is the programming logic driven by the user or by the programmer?
- Does the language provide for all the required features of the solution?
- What hardware ramifications are there?
- Is a graphical user interface (GUI) required?
- What is the experience of the developers?

All these questions need to be considered carefully.

### IS THE PROGRAMMING LOGIC DRIVEN BY THE USER OR BY THE PROGRAMMER?

The answer to this question will determine whether the language selected should use an event-driven approach or a sequential approach. In the Preliminary course, we studied the major differences between these two approaches.

Event-driven languages allow the user to choose the order in which processing occurs. For example, a Word Processor is clearly an event-driven piece of software. The user decides in which order tasks are accomplished. They can enter all the text and then choose to edit and format the text, or they can edit as they go and format at the end. A Word Processor allows the user to alter the order of processing, as each process is initiated by the user. In modern GUI environments, events are usually initiated by selecting an item from a drop-down menu. Once the menu item is selected, the particular module of code associated with that event is executed.

Sequential languages ensure that the software executes in a distinct order. The programmer determines the order of processing when writing the software. The user must follow a distinct route through the software. For example, the game of 'snap' examined in the previous section, must be played according to a set of rules that determine a strict sequence of processing. Players cannot play out of turn; the game follows a strict sequence of events.

Many problems involve a combination of approaches. For the game of 'snap' we may wish either player to be able to terminate the game at any time. If this is the case, we would provide a menu item to end the game that is always available. A player initiating the end game option is effectively initiating an event. The program responds to this event by running a module that halts the game. Once an event, say to start the game of 'snap', is initiated, then a distinct sequence of events must be completed before another event can be begun. In effect, event-driven software combines a number of smaller, sequential modules.

## **DOES THE LANGUAGE PROVIDE FOR ALL THE REQUIRED FEATURES OF THE SOLUTION?**

The reason so many programming languages exist is that solutions to different problems require different features within languages. This has given rise to the development of a vast assortment of languages. Solutions requiring many low-level operations will best be implemented in a quite different language to that used for the development of a new web page. Financial software would be implemented in a different language than for a product based on the processing of text.

Careful consideration of features required to implement a solution, and an intelligent selection of an appropriate language, will result in reduced development times and more efficient final products.

## **WHAT HARDWARE RAMIFICATIONS ARE THERE?**

Many programming languages are designed for particular operating systems. As operating systems are specific to particular types of hardware, the programming language selected will most likely have hardware ramifications. This problem is particularly prevalent when using COTS (Customised off-the-shelf) packages as development tools. Most COTS packages are designed to operate under a specific operating system. A product developed using a Unix-designed programming environment will normally require significant changes to operate in a Macintosh environment.

If a language is used that does not support the system settings provided in modern operating systems, then drivers for different peripheral devices may need to be written. For custom products written for a particular known client, this situation may be acceptable, but for products developed for a large and unknown customer base this situation is anything but acceptable. For example, it would not be economically feasible to write drivers for every available printer on the market. It is far wiser to use a language environment that can utilise the operating system's printer drivers and settings.

## **IS A GRAPHICAL USER INTERFACE (GUI) REQUIRED?**

For personal computer-based applications the use of a standard GUI is almost obligatory. The majority of users will be accustomed to the familiar features of their installed, GUI-style operating system. Custom solutions developed for particular clients are another matter. A software solution developed for a new model of mobile phone will use a proprietary user interface designed specifically for the particular phone. Software products that operate with minimal user interaction are often best developed without the extra overhead of a GUI interface. For example, a software encryption product for the military runs in the background, with only very limited user interaction required. The users, who have a high-level of expertise, will only interact with the product to alter configuration settings. The use of a GUI would reduce the efficiency of this product.

## **WHAT IS THE EXPERIENCE OF THE DEVELOPERS?**

The first programming language learnt by a developer is always the most difficult. As developers become proficient in further languages, the learning curve flattens. Nevertheless, learning a new programming language is a difficult and time-consuming task. Before deciding on a language for implementation, the experience and expertise available within the software development team need to be examined. If a new language is used, then development times will need to be extended.

**CHAPTER 4 REVIEW**

1. Which of the following lists is in the correct order of creation?
  - (A) System models, algorithms, data structures.
  - (B) Data structures, system models, algorithms.
  - (C) System models, data structures, algorithms.
  - (D) Algorithms, system models, data structures.
2. Which of the following statements is true?
  - (A) Linear searches are generally faster than binary searches.
  - (B) The data items must be sorted before a linear search.
  - (C) Binary searches discard half the list after each comparison.
  - (D) A binary search will examine every data item in the list.
3. At least one item must be in its correct final position after a single pass through the data items when using:
  - (A) an insertion or bubble sort.
  - (B) a selection or insertion sort.
  - (C) a selection or bubble sort.
  - (D) a selection sort only.
4. Adding a new string into the middle of an existing string could be called:
  - (A) extracting.
  - (B) deleting.
  - (C) inserting.
  - (D) copying.
5. Which sort does not involve swapping data items?
  - (A) Insertion sort.
  - (B) Selection sort.
  - (C) Bubble sort.
  - (D) Both insertion and selection sorts.
6. Finding the largest value in an unsorted list of items is:
  - (A) similar to one pass of a selection sort.
  - (B) the main concept behind an insertion sort.
  - (C) accomplished using a binary search.
  - (D) possible only when the data is numeric.
7. Adding one string to one or more other strings is called:
  - (A) concatenation.
  - (B) inserting.
  - (C) combining.
  - (D) ampersand.
8. A data structure where each individual data item may have a different data type is:
  - (A) an array of records.
  - (B) a record.
  - (C) an array.
  - (D) a multi-dimensional array.
9. Records that need to be accessed directly would be stored in a:
  - (A) sequential file.
  - (B) random access file.
  - (C) text file.
  - (D) binary file.
10. Languages that use an event-driven approach contain all the following features EXCEPT:
  - (A) Programs written in them have a distinct start and finish.
  - (B) The user controls the order of processing.
  - (C) They often use a GUI.
  - (D) Modules are executed as a result of some user action.
11. Often a series of records need to be sorted by two fields. For example, it is common to sort names by surname, and then within each surname by first name. Can this be accomplished by applying one of the sorts studied, twice? Explain your answer in terms of the insertion, selection and bubble sorts.
12. Read the following scenario and create an algorithm to complete the task:
 

Throwing two dice always results in a total between two and twelve, however each total is not equally likely. For instance, there is only one way of throwing a total of two: you must throw a double one. However, a total of four can be obtained in three ways: a double two, a one and a three or a three and a one. We could work out the theoretical probability for each total, however what if we threw the two dice a million times? How many times would each total come up?

Create an algorithm to simulate throwing two dice a million times. Your algorithm is to output the number of times each total is obtained.



13. Internet search engines allow us to enter a list of words separated by spaces. The search engine then looks for web sites that include any combination of the words entered. The search engine examines the total string you have entered and splits it into individual words based on the spaces separating the words.

Create an algorithm that receives a string of words separated by spaces. The algorithm is to split this string into its component words and store the words in an array.



Examine the algorithms at right to answer questions 14 and 15.

14. Describe the nature and data type of each of the following: IsMatch, ClientName, ClientID, ProductID, FindClientID, Index, Client, Clients, OrderExists, Order.
15. What do you think is the purpose of these algorithms? Explain.

```
BEGIN IsMatch(ClientName,ProductID)
 ClientID=FindClientID(ClientName)
 IsMatch=OrderExists(ClientID,ProductID)
END IsMatch
```

```
BEGIN FindClientID(Client)
 Set Index to 1
 WHILE More records
 IF Clients(Index).Name=Client THEN
 FindClientID=Index
 ENDIF
 Add 1 to Index
 ENDWHILE
END FindClientID
```

```
BEGIN OrderExists(ClientID,ProductID)
 IF Order(ClientID,ProductID) is blank THEN
 OrderExists=False
 ELSE
 OrderExists=True
 ENDIF
END OrderExists
```

