Biyani's Think Tank

A concept based exclusive material

Algorithms and Application Programming

M.Sc. I.T.

Nitika Newar
MCA
Lecturer
Deptt. of Information Technology
Biyani Girls College, Jaipur

Biyani's
Group of Girls' Colleges

Published by :
Think Tanks
Biyani Group of Colleges

Concept & Copyright :
©Biyani Shikshan Samiti
Sector-3, Vidhyadhar Nagar,
Jaipur-302 023 (Rajasthan)
Ph : 0141-2338371, 2338591-95 • Fax : 0141-2338007
E-mail : acad@biyanicolleges.org
Website :www.gurukpo.com; www.biyanicolleges.org

First Edition : 2009

Leaser Type Setted by :
Biyani College Printing Department

Preface

I am glad to present this book, especially designed to serve the needs of the students. The book has been written keeping in mind the general weakness in understanding the fundamental concepts of the topics. The book is self-explanatory and adopts the "Teach Yourself" style. It is based on question-answer pattern. The language of book is quite easy and understandable based on scientific approach.

This book covers basic concepts related to the microbial understandings about diversity, structure, economic aspects, bacterial and viral reproduction etc.

Any further improvement in the contents of the book by making corrections, omission and inclusion is keen to be achieved based on suggestions from the readers for which the author shall be obliged.

I acknowledge special thanks to Mr. Rajeev Biyani, *Chairman* & Dr. Sanjay Biyani, *Director* (*Acad.*) Biyani Group of Colleges, who are the backbones and main concept provider and also have been constant source of motivation throughout this Endeavour. They played an active role in coordinating the various stages of this Endeavour and spearheaded the publishing work.

I look forward to receiving valuable suggestions from professors of various educational institutions, other faculty members and students for improvement of the quality of the book. The reader may feel free to send in their comments and suggestions to the under mentioned address.

Author

Syllabus

M.Sc.-IT (Sem.-I)

111 : ALGORITHMS AND APPLICATION PROGRAMMING

Algorithmic Methodology : Basic Concepts and Notation, Understanding the Problem, Plan the Logic, Code the Program, Pseudocode and Flowchart, Efficiency of Algorithms. Complexity Measures, Basic Time Analysis of an Algorithm, Space Complexity.

Conditionals : Control Structures and Program Writing.

Looping : Repetitions.

Abstract Data Types : Data Abstraction and Basic Data Structures, Data Types, Abstract Data Types.

Recursion : Characteristics of Recursive Functions, Mathematical Induction Propositional Logic, First Order Predicate Calculus : Introduction to Formal Logic, Propositions, Conditional Propositions, Proofs, Resolution Proofs, Rules of Inference Program Specification : Problem Solving, Variables, Selection, Modules and Repetitions.

Arrays : Storage Allocation Functions, Linked Allocation, Hashed Allocation Techniques, Sorting Searching Sequential, Binary and Hashed Searching, Internal and External Sorting Techniques, Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Radix Sort and Quick Sort Comparisons and String Conversions, Representations of Variable-Length Strings, Examples of Operations on Strings.

Data Structures and File Handling : File Organisation, Text and Binary Files, Opening and Closing Files.

Advanced Programming Concepts : (Introduction Only) : Recursion, Dynamic Memory Management and Allocation, Operating System Calls, Inner Process Communication, Advanced File Handling and Indexing.

Language : C (Examples using C wherever required).

Content

CHAPTER-1

Algorithmic Methodology

Q.1 What are the various steps to plan Algorithm ?

Ans.: Following steps must be followed to plan any algorithm :

(1) **Device Algorithm :** Creating an algorithm is an art in which may never be fully automated. When we get the problem, we should first analyse the given problem clearly and then write down some steps on the paper.

(2) **Validate Algorithm :** Once an algorithm is devised , it is necessary to show that it computes the correct answer for all possible legal inputs . This process is known as algorithm validation. The algorithm need not as yet be expressed as a program. It is sufficient to state it in any precise way. The purpose of validation is to assure us that this algorithm will work correctly independently of the issues concerning the programming language it will eventually be written in. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as program proving or program verification.

(3) **Analyse Algorithm :** As an algorithm is executed , it uses the computers central processing unit to perform operations and its memory (both immediate and auxiliary) to hold the program and data. Analysis of algorithm or performance analysis refers to the task of determining how much computing time and storage an algorithm requires. An important result of this study is that it allows you to make quantitative judgments about the value of one algorithm over another. Another result is that it allows you to predict whether the software will meet any efficiency constraints that exist. Analysis can be made by taking into consideration.

(4) **Test A Program :** Testing a program consists of 2 phases : debugging and performance management. Debugging is the process of executing programs on sample data sets to determine whether results are incorrect if so corrects them. Performance management is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results. These timing figures are useful in that they may confirm a previously done analysis and point out logical places to perform useful optimization.

Q.2 What is Pseudocode? What are its benefits?

Ans.: An outline of a program, written in a form that can easily be converted into real programming statements.

Some examples of pseudocode are :

1.
```
If student's grade is greater than or equal to 60
    Print "passed"
else
    Print "failed"
```

2.
```
Set total to zero
Set grade counter to one
While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
Set the class average to the total divided by ten
Print the class average.
    Print 'no grades were entered'
```

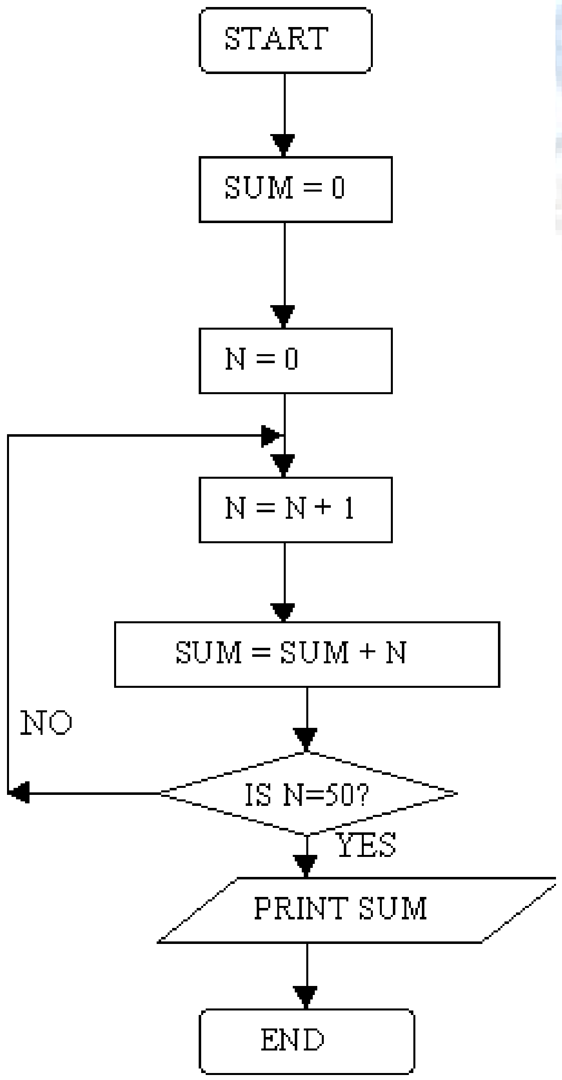
3. The pseudocode for a bubble sort might be written :
```
while not at end of list
compare adjacent elements
if second is greater than first
switch them
get next two elements
if elements were switched
repeat for entire list
```

Pseudocode cannot be compiled nor executed, and there are no real formatting or syntax rules. It is simply one step - an important one - in producing the final code. The benefit of pseudocode is that it enables the programmer to concentrate on the algorithms without worrying about all the syntactic details of a particular programming language. In fact, you can write pseudocode without even knowing what programming language you will use for the final implementation.

Q.3 What is Flowchart? What are various symbols of Flowchart?

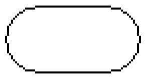
Ans.: A flowchart is a diagrammatic representation that illustrates the sequence of operations to be performed to get the solution of a problem. Flowcharts are generally drawn in the early stages of formulating computer solutions. Flowcharts facilitate communication between programmers and business people. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Often we see how flowcharts are helpful in explaining the program to others. Hence, it is correct to say that a flowchart is a must for the better documentation of a complex program.

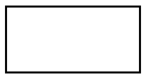
Example of a simple flowchart: a flowchart to find the sum of first 50 natural numbers.

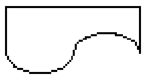


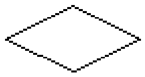
Various symbols used in flowchart are :

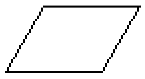
Flowcharts use special shapes to represent different types of actions or steps in a process. Lines and arrows show the sequence of the steps, and the relationships among them.

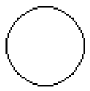

Start/End : The terminator symbol marks the starting or ending point of the system. It usually contains the word "Start" or "End."

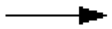

Action or Process : A box can represent a single step ("add two cups of flour"), or and entire sub-process ("make bread") within a larger process.

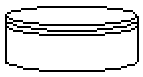

Document : A printed document or report.

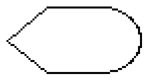

Decision : A decision or branching point. Lines representing different decisions emerge from different points of the diamond.


Input/Output : Represents material or information entering or leaving the system, such as customer order (input) or a product (output).


Connector : Indicates that the flow continues where a matching symbol (containing the same letter) has been placed.


Flow Line : Lines indicate the sequence of steps and the direction of flow.


Database : Indicates a list of information with a standard structure that allows for searching and sorting.


Display : Indicates a step that displays information.

Q.4 What are the advantages and disadvantages of using Flowcharts?

Ans.: Advantages of using Flowcharts : The benefits of flowcharts are as follows :

(1) **Communication :** Flowcharts are better way of communicating the logic of a system to all concerned.

(2) **Effective Analysis :** With the help of flowchart, problem can be analysed in more effective way.

(3) **Proper Documentation :** Program flowcharts serve as a good program documentation, which is needed for various purposes.

(4) **Efficient Coding :** The flowcharts act as a guide or blueprint during the systems analysis and program development phase.

(5) **Proper Debugging :** The flowchart helps in debugging process.

(6) **Efficient Program Maintenance :** The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part

Limitations of using Flowcharts :

(1) **Complex Logic :** Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.

(2) **Alterations and Modifications :** If alterations are required the flowchart may require re-drawing completely.

(3) **Reproduction :** As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.

(4) The essentials of what is done can easily be lost in the technical details of how it is done.

Q.5 What is Time Complexity & Space Complexity measures of Algorithm?

Ans.: The **time complexity** of a problem is the number of steps that it takes to solve an instance of the problem as a function of the size of the input (usually measured in bits), using the most efficient algorithm. To understand this intuitively, consider the example of an instance that is n bits long that can be solved in n^2 steps. In this example we say the problem has a time complexity of n^2. Of course, the exact number of steps will depend on exactly what machine or language is being used. To avoid that problem, the Big O notation is generally used (sometimes described as the "order" of the calculation, as in "on the order of"). If a problem has time complexity $O(n^2)$ on one typical computer, then it will also have complexity $O(n^2)$ on most other computers, so this notation allows us to generalize away from the details of a particular computer.

Example : Mowing grass has linear time complexity because it takes double the time to mow double the area. However, looking up something in a dictionary has only logarithmic time complexity because a double sized dictionary only has to be opened one time more (i.e. exactly in the middle, then the problem size is reduced by half).

The **space complexity** of a problem is a related concept, that measures the amount of space, or memory required by the algorithm. An informal analogy would be the amount of scratch paper needed while working out a problem with pen and paper. Space complexity is also measured with Big O notation.

□ □ □

CHAPTER-2

Conditionals & Looping

Q.1 What is Control Structures?

Ans.: A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the *compound-statement* or *block*. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }:

```
{ statement1; statement2; statement3; }
```

Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces ({}). But in the case that we want the statement to be a compound statement it must be enclosed between braces ({}), forming a block.

Conditional structure : if and else

The if keyword is used to execute a statement or block only if a condition is fulfilled. Its form is :

```
if (condition) statement
```

Where condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

For example, the following code fragment prints x is 100 only if the value stored in the x variable is indeed 100:

```
if (x == 100)
```

```
    cout << "x is 100";
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces { }:

```
if (x == 100)
{
  cout << "x is ";
  cout << x;
}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword else. Its form used in conjunction with if is:

if (condition) statement1 else statement2

For example:

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

prints on the screen x is 100 if indeed x has a value of 100, but if it has not -and only if not- it prints out x is not 100.

The if + else structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in x is positive, negative or none of them (i.e. zero):

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
```

```
cout << "x is 0";
```

Q.2 What are the different types of Loops?

Ans.: Iteration Structures (Loops) : Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is :

```
while (expression) statement
```

and its functionality is simply to repeat statement while the condition set in expression is true. For example, we are going to make a program to countdown using a while-loop:

The do-while loop

Its format is :

```
do statement while (condition);
```

The for loop

Its format is :

```
for (initialization; condition; increase) statement;
```

□ □ □

CHAPTER-3

Abstract Data Types & Recursion

Q.1 What is Abstract Data Structure ? Give some examples.

Ans.: An **abstract data structure** is an abstract storage for data defined in terms of the set of operations to be performed on data and computational complexity for performing these operations, regardless of the implementation in a concrete data structure.

Selection of an abstract data structure is crucial in the design of efficient algorithms and in estimating their computational complexity, while selection of concrete data structures is important for efficient implementation of algorithms.

Abstract data types (ADT) typically implemented in programming languages (or their libraries) include :

Complex Number

List

Priority Queue

Queue

Stack

String

Tree

Q.2 What is Recursion? Explain with example?

Ans.: **Recursion**, in mathematics and computer science, is a method of defining functions in which the function being defined is applied within its own definition. The term is also used more generally to describe a process of repeating objects in a self-similar way. For instance, when the surfaces of two mirrors are almost parallel with each other the nested images that occur are a form of recursion.

It is a way of thinking about and solving problems. It is, in fact, one of the central ideas of computer science. [1] Solving a problem using recursion means the solution depends on solutions to smaller instances of the same problem. [2]

Factorial : A classic example of a recursive procedure is the function used to calculate the factorial of an integer.

Function Definition : A recurrence relation is an equation that relates later terms in the sequence to earlier terms[6].

Recurrence relation for factorial:

$b_n = n * b_{n-1}$

$b_0 = 1$

Computing the recurrence relation for n = 4:
$b_4 = 4 * b_3$ $= 4 * 3 * b_2$ $= 4 * 3 * 2 * b_1$ $= 4 * 3 * 2 * 1 * b_0$ $= 4 * 3 * 2 * 1 * 1$ $= 4 * 3 * 2 * 1$ $= 4 * 3 * 2$ $= 4 * 6$ $= 4 * 6$ $= 24$

□ □ □

CHAPTER-4

Searching & Sorting Techniques

Q.1 Explain the concept of Bubble Sort along with Algorithm.

Ans.: **Bubble sort** is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort.

Step-by-Step Example : Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements written in **bold** are being compared.

First Pass :

(**5 1** 4 2 8) (**1 5** 4 2 8) Here, algorithm compares the first two elements, and swaps them.

(1 **5 4** 2 8) (1 **4 5** 2 8)

(1 4 **5 2** 8) (1 4 **2 5** 8)

(1 4 2 **5 8**) (1 4 2 **5 8**) Now, since these elements are already in order, algorithm does not swap them.

Second Pass :

(**1 4** 2 5 8) (**1 4** 2 5 8)

(1 **4 2** 5 8) (1 **2 4** 5 8)

(1 2 **4 5** 8) (1 2 **4 5** 8)

(1 2 4 **5 8**) (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed.

Algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass :

(**1 2** 4 5 8) (**1 2** 4 5 8)

(1 **2 4** 5 8) (1 **2 4** 5 8)

(1 2 **4 5** 8) (1 2 **4 5** 8)

(1 2 4 **5 8**) (1 2 4 **5 8**)

Finally, the array is sorted, and the algorithm can terminate.

```
procedure bubbleSort( A : list of sortable items ) defined as:
  for each i in 1 to length(A) do:
    for each j in length(A) downto i + 1 do:
      if A[ j -1  ] > A[ j ] then
        swap( A[ j - 1],  A[ j ] )
      end if
    end for
  end for
end procedure
```

Q.2 Explain the concept of Selection Sort along with Algorithm.

Ans.: The algorithm works as follows :

(1) Find the minimum value in the list.

(2) Swap it with the value in the first position.

(3) Repeat the steps above for remainder of the list (starting at the second position).

Effectively, we divide the list into two parts: the sublist of items already sorted, which we build up from left to right and is found at the beginning, and the sublist of items remaining to be sorted, occupying the remainder of the array.

Here is an example of this sort algorithm sorting five elements :

64 25 12 22 11

11 25 12 22 64

11 12 25 22 64

11 12 22 25 64

Selection sort can also be used on list structures that make add and remove efficient, such as a linked list. In this case it's more common to *remove* the minimum element from the remainder of the list, and then *insert* it at the end of the values sorted so far. For example :

64 25 12 22 11

11 64 25 12 22

11 12 64 25 22

11 12 22 64 25

11 12 22 25 64

Pseudo-code :

A is the set of elements to sort, **n** is the number of elements in **A** (the array starts at index 0)

```
for i ← 0 to n-2 do
   min ← i
   for j ← (i + 1) to n-1 do
      if A[j] < A[min]
         min ← j
   swap A[i] and A[min]
```

Q.3 Explain the Algorithm of Insertion Sort?

Ans.: **Insertion sort** is a simple sorting algorithm, a comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such as quick sort, heap sort, or merge sort, but it has various advantages :

(1) Simple to implement.

(2) Efficient on (quite) small data sets.

(3) Efficient on data sets which are already substantially sorted: it runs in O(n + d) time, where d is the number of inversions.

(4) stable(does not change the relative order of elements with equal keys)

(5) In- place (only requires a constant amount O(1) of extra memory space)

(6) It is an online algorithm , in that it can sort a list as it receives it.

A simple **procedure for Insertion Sort** is :

```
insertionSort(array A)
   for i = 1 to length[A]-1 do
   begin
```

```
    value = A[i]
    j = i-1
    while j >= 0 and A[j] > value do
    begin
       A[j + 1] = A[j]
       j = j-1
    end
    A[j+1] = value
 end
```

Q.4 Explain the Merge Sort?

Ans.: Conceptually, a Merge Sort works as follows :

- If the list is of length 0 or 1, then it is sorted. Otherwise;
- Divide the unsorted list into two sublists of about half the size;
- Sort each sublist recursively by re-applying merge sort;
- Merge the two sublists back into one sorted list.

In a simple pseudocode form, the algorithm could look something like this:

```
function mergesort(m)
   var list left, right, result
   if length(m) ≤ 1
      return m
   var middle = length(m) / 2
   for each x in m up to middle
      add x to left
   for each x in m after middle
      add x to right
   left = mergesort(left)
   right = mergesort(right)
   result = merge(left, right)
   return result
```

Q.5 Explain the Radix Sort?

Ans.: In computer science, **radix sort** is a sorting algorithm that sorts integers by processing individual digits. Because integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers, radix sort is not limited to integers.

Most digital computers internally represent all of their data as electronic representations of binary numbers, so processing the digits of integer representations by groups of binary digit representations is most convenient. Two classifications of radix sorts are least significant digit (LSD) radix sorts and most significant digit (MSD) radix sorts. LSD radix sorts process the integer representations starting from the least significant digit and move towards the most significant digit. MSD radix sorts work the other way around.

The integer representations that are processed by sorting algorithms are often called "keys," which can exist all by themselves or be associated with other data. LSD radix sorts typically use the following sorting order: short keys come before longer keys, and keys of the same length are sorted lexicographically. This coincides with the normal order of integer representations, such as the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. MSD radix sorts use lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations. A sequence such as "b, c, d, e, f, g, h, i, j, ba" would be lexicographically sorted as "b, ba, c, d, e, f, g, h, i, j". If lexicographic ordering is used to sort variable-length integer representations, then the representations of the numbers from 1 to 10 would be output as 1, 10, 2, 3, 4, 5, 6, 7, 8, 9, as if the shorter keys were left-justified and padded on the right with blank characters to make the shorter keys as long as the longest key for the purpose of determining sorted order.

Q.6 Explain the Quick Sorting?

Ans.: Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are :

- Pick an element, called a *pivot*, from the list.
- Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
- Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base case of the recursion are lists of size zero or one, which are always sorted.

In simple pseudocode, the algorithm might be expressed as:

```
function quicksort(array)
    var list less, greater
    if length(array) ≤ 1
        return array
    select a pivot value pivot from array
    for each x in array
        if x < pivot then append x to less
        if x > pivot then append x to greater
    return concatenate(quicksort(less), pivot, quicksort(greater))
```

Q.7 Explain the concept of Binary Search?

Ans.: A **binary search algorithm** (or **binary chop**) is a technique for finding a particular value in a sorted list. It makes progressively better guesses, and closes in on the sought value by selecting the median element in a list, comparing its value to the target value, and determining if the selected value is greater than, less than, or equal to the target value. A guess that turns out to be too high becomes the new top of the list, and a guess that is too low becomes the new bottom of the list. Pursuing this strategy iteratively, it narrows the search by a factor of two each time, and finds the target value.

The algorithm : The most common application of binary search is to find a specific value in a sorted list. To cast this in the frame of the guessing game (see Example below), realize that we are now guessing the *index*, or numbered place, of the value in the list. This is useful because, given the index, other data structures will contain associated information. Suppose a data structure containing the classic collection of name, address, telephone number and so forth has been accumulated, and an array is prepared containing the names, numbered from one to *N*. A query might be: what is the telephone number for a given name *X*. To answer this the array would be searched and the index (if any) corresponding to that name determined, whereupon it would be used to report the associated telephone number and so forth. Appropriate provision must be made for the name not being in the list (typically by returning an *index* value of zero), indeed the question of interest might be only whether *X* is in the list or not.

```
low = 0
high = N
```

```
while (low < high) {
   mid = (low + high)/2;
   if (A[mid] < value)
      low = mid + 1;
   else
       //can't be high = mid-1: here A[mid] >= value,
       //so high can't be < mid if A[mid] == value
       high = mid;
}

if (low < N) and (A[low] == value)
   return low
else
   return not_found
```

This algorithm has two other advantages. At the end of the loop, *low* points to the first entry greater than or equal to *value*, so a new entry can be inserted if no match is found. Moreover, it only requires one comparison; which could be significant for complex keys in languages which do not allow the result of a comparison to be saved.

Q.8 What is Internal & External Sorting techniques?

Ans.: An **internal sort** is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory. For sorting larger datasets, it may be necessary to hold only a chunk of data in memory at a time, since it wont all fit. The rest of the data is normally held on some larger, but slower medium, like a hard-disk. Any reading or writing of data to and from this slower media can slow the sortation process considerably. This issue has implications for different sort algorithms.

Consider a Bubblesort, where adjacent records are swapped in order to get them into the right order, so that records appear to 'bubble' up and down through the dataspace. If this has to be done in chunks, then when we have sorted all the records in chunk 1, we move on to chunk 2, but we find that some of the records in chunk 1 need to 'bubble through' chunk 2, and vice versa (i.e., there are records in chunk 2 that belong in chunk 1, and records in chunk 1 that belong in chunk 2 or later chunks). This will cause the chunks to be read and written back to disk many times as records cross over the boundaries between them, resulting in a considerable degradation of performance. If the data can all be held in memory as one large chunk, then this performance hit is avoided.

On the other hand, some algorithms handle **external sorting** rather better. A Merge sort breaks the data up into chunks, sorts the chunks by some other algorithm (maybe bubblesort or Quick sort) and then recombines the chunks two by two so that each recombined chunk is in order. This approach minimises the number or reads and writes of data-chunks from disk, and is a popular external sort method.

□ □ □

CHAPTER-5

Arrays

Q.1 What are the different functions used for Allocation of Memory?

Ans.: Sizeof & Malloc : The sizeof operator returns the size in bytes of its operand. Whether the result of sizeof is unsigned int or unsigned long is implementation defined—which is why the declaration of malloc above ducked the issue by omitting any parameter information; normally you would use the stdlib.h header file to declare malloc correctly. Here is the last example done portably :

```
#include <stdlib.h>     /* declares malloc() */

float *fp;

fp = (float *)malloc(sizeof(float));
```

The operand of sizeof only has to be parenthesized if it's a type name, as it was in the example. If you are using the name of a data object instead, then the parentheses can be omitted, but they rarely are.

```
#include <stdlib.h>

int *ip, ar[100];

ip = (int *)malloc(sizeof ar);
```

In the last example, the array ar is an array of 100 ints; after the call to malloc (assuming that it was successful), ip will point to a region of store that can also be treated as an array of 100 ints.

The fundamental unit of storage in C is the char, and by definition

sizeof(char)

is equal to 1, so you could allocate space for an array of ten chars with

malloc(10)

while to allocate room for an array of ten ints, you would have to use

malloc(sizeof(int[10]))

If malloc can't find enough free space to satisfy a request it returns a null pointer to indicate failure. For historical reasons, the stdio.h header file contains a defined constant called NULL which is traditionally used to check the return value from malloc and some other library functions. An explicit 0 or (void *)0 could equally well be used.

Calloc and Realloc : There are two additional memory allocation functions, Calloc() and Realloc(). Their prototypes are given below:

void *calloc(size_t num_elements, size_t element_size};

void *realloc(void *ptr, size_t new_size);

Malloc does not initialise memory (to ***zero***) in any way. If you wish to initialise memory then use calloc. Calloc there is slightly more computationally expensive but, occasionally, more convenient than malloc. Also note the different syntax between calloc and malloc in that calloc takes the number of desired elements, num_elements, and element_size, element_size, as two individual arguments.

Thus to assign 100 integer elements that are all initially zero you would do:

int *ip;

ip = (int *) calloc(100, sizeof(int));

Realloc is a function which attempts to change the size of a previous allocated block of memory. The new size can be larger or smaller. If the block is made larger then the old contents remain unchanged and memory is added to the end of the block. If the size is made smaller then the remaining contents are unchanged.

If the original block size cannot be resized then realloc will attempt to assign a new block of memory and will copy the old block contents. Note a new pointer (of different value) will consequently be returned. You **must** use this new value. If new memory cannot be reallocated then realloc returns NULL.

Thus to change the size of memory allocated to the *ip pointer above to an array block of 50 integers instead of 100, simply do:

ip = (int *) calloc(ip, 50);

Q.2 What is Hashing? How is it used for searching?

Ans.: Hashing has a *worst-case* behavior that is linear for finding a target, but with some care, hashing can be dramatically fast in the *average-case*. Hashing also makes it easy to add and delete elements from the collection that is being searched.

Suppose we want to store information about each student in a database, so that we can later retrieve information about any student simply using his/her student

ID. To be specific, suppose the information about each student is an object of the following form, with the student ID stored in the key field:

```
struct Student
{
  int key;         // the student ID
  long phone;      // phone number
  string address;  // student address
};
```

We call each of these objects a record. Of course, there might be other information in each student record.

If student IDs are all in the range 0..99, we could store the records in an array of the following type, placing student ID *k* in location data[k]:

```
Student data[100];  // array of 100 records
```

The record for student ID *k* can be retrieved immediately since we know it is in data[k].

What, however, if the student IDs do not form a neat range like 0..99. Suppose that we only know that there will be a hundred or fewer and that they will be distributed in the range 0..9999. We could then use an array with 10,000 components, but that seems wasteful since only a small fraction of the array will be used. It appears that we have to store the records in an array with 100 elements and to use a serial search through this array whenever we wish to find a particular student ID. If we are clever, we can store the records in a relatively small array and still retrieve students by ID much faster than we could by serial search.

To illustrate this, suppose that we know that the student IDs will be:

0, 100, 200, ... , 9800, 9900

In this case, we can store the records in an array called data with only 100 components. We'll store the record with student ID *k* at location:

data[k/100]

If we want to retrieve information for student ID 700, we compute 700/100 and obtain the index 7. The record for student ID 700 is stored in array component data[7].

This general technique is called hashing. Each record requires a unique value called its key. In our example the student ID is the key, but other, more complex keys are sometimes used. A function called the ***hash function***, maps keys to array indices.

Suppose we name our hash function hash. If a record has a key of *k*, then we will try to store that record at location data[hash(k)]. Using the hash function to compute the correct array index is called ***hashing*** the key to an array index. The hash function must be chosen so that its return value is always a valid index for the array. In our example:

hash(k) = k / 100

Given this hash function and keys that are multiples of 100, every key produces a different index when it was hashed. Thus, hash is a *perfect hash function*. Unfortunately, a perfect hash function cannot always be found.

□ □ □