PREFACE

This study material deals with the concepts of c programming and algorithms. C is a general-purpose high level language that was originally developed by Dennis Ritchie for the Unix operating system. It was first implemented on the Digital Equipment Corporation PDP-11 computer in 1972. C was initially used for system development work, in particular the programs that make-up the operating system. C was adopted as a system development language because it produces code that runs nearly as fast as code written in assembly language. All the C programs are written into text files with extension ".c". When you write any program in C language to be executed then, that program need to be compiled by a C Compiler. The compiler converts the program into object code which can be read by a computer. This is called machine language (i.e. binary format). So before proceeding, make sure you have C Compiler available in your computer.

An algorithm refers to the concept of data structures in detail. Before introducing data structures we should understand that, computer is to store, retrieve, and process a large amount of data. If the data is stored in well organized way on a storage medium and in computer's memory then it can be accessed quickly for processing. This can further reduce the latency and the user is provided fast response. Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.

A data structure should be seen as a logical concept that must address two fundamental concerns. First, how the data will be stored, and second, what operations will be performed on it? As data structure is a scheme for data organization so the functional definition of a data structure should be independent of its implementation. The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation. The implementation part is left on developers who decide which technology better suits to their project needs.

Ms. Uma Shankari

HoD

Department of Computer Application

Dr. M.G.R. Janaki Arts & Science College for Women

DEGREE OF BACHELOR OF COMPUTER APPLICATIONS & LAW (B.C.A.LLB,)

Title of the Course/ Paper	Programming in C and Algorithms(HDAH)
Core - 1	I Year & Second Semester Credit: 4
Objective of the course	This course introduces the basic programming concepts and fundamentals of Programming Language C and Algorithm
Course outline	Unit 1: C fundamentals Character set – Identifier and keywords – data types – constants – Variables – Declarations – Expressions – Statements – Operators – Library functions- Data input output functions – Simple C programs – Flow of control – if, if-else, while, do-while, for loop, Nested control structures – Switch, break and continue, go to statements – Comma operator
	Unit-2: Functions -Definition – proto-types – Passing arguments – Recursions- Storage Classes – Arrays – Defining and Processing – Passing arrays to functions – Multi-dimension arrays – Arrays and String. Structures – User defined data types – Passing structures to functions – Self-referential structures – Unions – Bit wise operations.
	Unit 3: Fundamentals of algorithms - Notion of an algorithm- Pseudo-code conventions like assignment statements and basic control structures – Analysis of algorithms – Running time of an algorithm – worst and average case analysis.
	Unit-4: Sorting Algorithms - Bubble, Selection- Insertion and Merge sort- Efficiency of algorithms -Implement using C- Searching Algorithms-Linear Search and Binary Search - Graph Algorithms- BFS, DFS, shortest paths - single source and all pairs.
	Unit-5 : Pointers – Declarations – Passing pointers to Functions – Operation in Pointers – Pointer and Arrays – Arrays of Pointers – Structures and Pointers – Files- Creating, Processing, Opening and Closing a data file.

1. Recommended Texts

   (i) E.Balagursamy, 2010, Programming in ANSI C, Fifth Edition, TMH, New Delhi.

   (ii) Horowitz, S. Sahni, and S. Rajasekaran, Computer Algorithms, Galgotia Pub, Pvt. LTs., 1998.

2. Reference Books

   (i) B. W. Kernighan and D. M. Ritchie, 1990, The C Programming Language, Second Edition, PHI, New Delhi.

   (ii) J. R. Hanly and E. B. Koffman, 2005, Problem solving and program design in C, Fourth Edition, Pearson Education India.

   (iii) H.Schildt, C: The Complete Reference, 4th Edition, TMH Edition, 2000.

PROGRAMMING IN C AND ALGORITHMS

CHAPTER 1 INTRODUCTION

Computer

Basically it is a fast calculating machine which is nowadays used for variety of uses ranging from house hold works to space technology. The credit of invention of this machine goes to the English Mathematician Charles Babbage.

Types of Computers:

Based on nature, computers are classified into Analog computers and Digital computers. The former one deals with measuring physical quantities (concerned with continuous variables) which are of late rarely used. The digital computer operates by counting and it deals with the discrete variables.

There is a combined form called Hybrid computer, which has both features. Based on application computers are classified as special purpose computers and general computers. As the name tells special computers are designed to perform certain specific tasks where as the other category is designed to cater the needs of variety of users.

Basic structure of a digital computer

The main components of a computer are

1) Input unit (IU),
2) Central Processing unit (CPU) and
3) Output unit (OU).

The information like data, programs etc are passed to the computer through input devices. The Keyboard, mouse, floppy disk, CD, DVD, joystick etc are certain input devices. The output device is to get information from a computer after processing. VDU (Visual Display Unit), Printer, Floppy disk, CD etc are output devices.

The brain of a computer is CPU. It has three components- Memory unit, Control unit and Arithmetic and Logical unit (ALU)- Memory unit also called storage device is to store information. Two types memory are there in a computer. They are RAM (random access memory) and ROM (read only memory). When a program is called, it is loaded and processed in RAM. When the computer is switched off, what ever stored in RAM will be deleted. So it is a temporary memory. Where ROM is a permanent memory, where data, program etc are stored for future use. Inside a computer there is storage device called Hard disk, where data are stored and can be accessed at any time. The control unit is for controlling the execution and interpreting of instructions stored in the memory. ALU is the unit where the arithmetic and Logical operations are performed.

The information to a computer is transformed to groups of binary digits, called bit. The length of bit varies from computer to computer, from 8 to 64. A group of 8 bits is called a Byte and a byte generally represents one alphanumeric (Alphabets and Numerals) character.

The Physical components of a computer are called hardwares. But for the machine to work it requires certain programs (A set of instructions is called a program). They are called softwares. There are two types of softwares – System software and Application software – System software includes Operating systems, Utility programs and Language processors.

ASCII Codes:

American Standard Code for Information Interchange. These are binary codes for alpha numeric data and are used for printers and terminals that are connected to a computer systems for alphabetizing and sorting.

Operating Systems

The set of instructions which resides in the computer and governs the system are called operating systems, without which the machine will never function. They are the medium of communication between a computer and the user. DOS, Windows, Linux, Unix etc are Operating Systems.

Utility Programs

These programs are developed by the manufacturer for the users to do various tasks. Word, Excel, Photoshop, Paint etc are some of them.

Languages

These programs facilitate the users to make their own programs. User's programs are converted to machine oriented and the computer does the rest of works.

Application Programs

These programs are written by users for specific purposes.

Computer Languages

They are of three types –

1 Machine Language (Low level language)

2 Assembly language (Middle level language)

3 User Oriented languages (High level language)

Machine language depends on the hardware and comprises of 0 and 1 .This is tough to write as one must know the internal structure of the computer. At the same time assembly language makes use of English like words and symbols. With the help of special programs called Assembler, assembly language is converted to machine oriented language. Here also a programmer faces practical difficulties. To overcome this hurdles user depends on high level languages, which are far easier to learn and use. To write programs in high level language, programmer need not know the characteristics of a computer. Here he uses English alphabets, numerals and some special characters.

Some of the High level languages are FORTRAN, BASIC, COBOL, PASCAL, C, C++, ADA etc. We use C to write programs. Note that High level languages cannot directly be followed by a computer. It requires the help of certain software's to convert it into machine coded instructions. This software's are called Compiler, Interpreter, and Assembler. The major difference between a compiler and an interpreter is that compiler compiles the user's program into machine coded by reading the whole program at a stretch where as Interpreter translates the program by reading it line by line. C and BASIC are an Interpreter where as FORTRAN is a

PROGRAMMING METHODOLOGY

A computer is used to solve a problem.

The steps involved are :

1. Analyze the problem
2. Identify the variables involved
3. Design the solution
4. Write the program
5. Enter it into a computer
6. Compile the program and correct errors
7. Correct the logical errors if any
8. Test the program with data
9. Document the program

Algorithms

Step by step procedure for solving a problem is called algorithm.

Example

To make a coffee

Step1: Take proper quantity of water in a cooking pan

Step2: Place the pan on a gas stove and light it

Step3: Add Coffee powder when it boils

Step4: Put out the light and add sufficient quantity of sugar and milk

Step5: Pour into cup and have it.

To add two numbers

Step1: Input the numbers as x, y

Step2: sum=x + y

Step3: print sum

For a better understanding of an algorithm, it is represented pictorially. The pictorial representation of an algorithm is called a Flow Chart. For this certain pictures are used.

Consider a problem of multiplying two numbers

Algorithm

Step1: Input the numbers as a and b

Step2: find the product a x b

Step3: Print the result

Consider the following problem to find the highest of three numbers

Algorithm

```
Step 1: read the numbers as x ,y and z
Step 2: compare x and y
Step 3: if x > y then compare x with z and find the greater
Step 4: Otherwise compare y with z and find the greater
```

Review Questions

1. Define computers?
2. Enlist the features of C.
3. How will you classify computer systems?
4. What are the basic operations of Computer?
5. What are the characteristics of computers?
6. What are the steps to solve the problem in a computer system?

Exercise:

Write Algorithm and flow chart for the solution to the problems :

1. To find the sum of n, say 10, numbers.
2. To find the factorial of n , say 10.
3. To find the sum of the series 1+x+x2+x3+.......... + xn
4. To find the sum of two matrices.
5. To find the scalar product of two vectors
6. To find the Fibonacci series up to n
7. To find gcd of two numbers

CHAPTER 2 C FUNDAMENTALS

A BRIEF HISTORY OF C

C evolved from a language called B, written by Ken Thompson at Bell Labs in1970. Ken used B to write one of the first implementations of UNIX. B in turn was a descendant of the language BCPL (developed at Cambridge (UK) in 1967), with most of its instructions removed.

So many instructions were removed in going from BCPL to B, that Dennis Ritchie of Bell Labs put some back in (1972), and called the language C. The famous book *The C Programming Language* was written by Kernighan and Ritchie in 1978, and was the definitive reference book on C for almost a decade.

The original C was still too limiting, and not standardized, and so in 1983 an ANSI committee was established to formalise the language definition.

It has taken until now (ten years later) for the ANSI (American National Standard Institute) standard to become well accepted and almost universally supported by compilers

STRUCTURE OF A PROGRAM

A C program basically consists of the following parts:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World":

```
#include<stdio.h>

Int main()

{

/*my first program in C */

Printf("Hello, World! \n");

Return 0;

}
```

Let us look various parts of the above program:

1. The first line of the program *#include <stdio.h>* is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation
2. The next line *int main()* is the main function where program execution begins.
3. The next line /*...*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
4. The next line *printf(...)* is another function available in C which causes the message "Hello, World!" to be displayed on the screen.
5. The next line **return 0;** terminates main()function and returns the value 0.

THE CHARACTER SET

The character set is the fundamental raw material of any language and they are used to represent information. Like natural languages, computer language will also have well defined character set, which is useful to build the programs.

Types	Character Set
Lowercase Letters	a-z
Uppercase Letters	A to Z
Digits	0-9
Special Characters	!@#$%^&*
White Spaces	Tab Or New line Or Space

IDENTIFIER AND KEYWORDS

Identifier are names given to various program elements like variables, arrays and functions.

(i) The name should begin with a letter and other characters can be letters and digits and also can contain underscore character (_)Example: area, average, x12 , name_of_place etc.........

(ii) Keywords are reserved words in C language. They have predicted meanings and are used for the intended purpose. Standard keywords are auto,break, case, char, const, continue, default, do, double, else enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while. (Note that these words should not be used as identities.)

DATA TYPE

- The variables and arrays are classified based on two aspects first is the data type it stores and the second is the type of storage.
- The basic datatypes in C language are int, char, float and double.
- They are respectively concerned with integer quantity, single character, numbers, with decimal point or exponent number and double precision floating point numbers (ie; of larger magnitude).
- These basic data types can be augmented by using quantities like short, long, signed and unsigned. (ie; long int, short int, long double etc.....).

CONSTANTS

There are 4 basic types of constants. They are integer constants, floating-point constants, character constants and string constants.

(a) **Integer constants**: It is an integer valued numbers, written in three different number system, decimal (base 10) , octal(base8), and hexadecimal(base 16).

A decimal integer constant consists of 0,1,.....,9..

Example : 75 6,0,32, etc.....

5,784, 39,98, 2-5, 09 etc are **not** integer constants.

An octal integer constant consists of digits 0,1,...,7. with 1st digit 0 to indicate that it is an octal integer.

Example : 0, 01, 0756, 032, etc.....

32, 083, 07.6 etc..... are **not** valid octal integers.

A hexadecimal integer constant consists of 0,1, ...,9,A, B, C, D, E, F. It begins with 0x.

Example: 0x7AA2, 0xAB, etc......

0x8.3, 0AF2, 0xG etc are **not** valid hexadecimal constants.

Usually negative integer constant begin with (-) sign. An unsigned integer constant is identified by appending U to the end of the constant like 673U, 098U, 0xACLFUetc. Note that 1234560789LU is an unsigned integer constant.

(b) floating point constants : It is a decimal number (ie: base 10) with a decimal point or an exponent or both. Ex; 32.65, 0.654, 0.2E-3, 2.65E10 etc. These numbers have greater range than integer constants.

(c) Character constants : It is a single character enclosed in single quotes like 'a'. '3', '?', 'A' etc. each character has an ASCII to identify. For example 'A' has the ASCII code 65, '3' has the code 51 and so on.

(d) escape sequences: An escape sequence is used to express non printing character like a new line, tab etc. it begin with the backslash (\ ) followed by letter like a, n, b, t, v, r, etc. the commonly used escape sequence are

\a : for alert	\n : new line	\0 : null
\b : backspace	\f : form feed	\? : question mark
\f : horizontal tab	\r : carriage return	\' : single quote
\v : vertical tab	\" : quotation mark	

(e) string constants : it consists of any number of consecutive characters enclosed in double quotes .Ex : " C program" , "mathematics" etc......

VARIABLES AND ARRAYS

A variable is an identifier that is used to represent some specified type of information. Only a single data can be stored in a variable. The data stored in the variable is accessed by its name. before using a variable in a program, the data type it has to store is to be declared.

Example : int a, b, c,

a=3; b=4;

c=a+b;

Note: A statement to declare the data types of the identifier is called declaration statement. An array is an identifier which is used to store a collection of data of the same type with the same name. the data stored is an array are distinguished by the subscript. The maximum size of the array represented by the identifier must be mentioned.

Example : int mark[100] .

With this declaration n, mark is an array of size 100, they are identified by mark[0],mark[1],.........,mark[99].

Note : along with the declaration of variable, it can be initialized too. For example int x=10; with this the integer variable x is assigned the value 10, before it is used. Also note that C is a case sensitive language. i.e. the variables d and D are different.

DECLARATIONS

This is for specifying data type. All the variables, functions etc must be declared before they are used. A *declaration* tells the compiler the name and type of a variable you'll be using in your program.

In its simplest form, a declaration consists of the type, the name of the variable, and a terminating semicolon:

Example :

```
int a,b,c;

Float mark, x[100], average;

char name[30];

char c;

int i;

float f;
```

You may wonder *why* variables must be declared before use. There are two reasons:

1. It makes things somewhat easier on the compiler; it knows right away what kind of storage to allocate and what code to emit to store and manipulate each variable; it doesn't have to try to intuit the programmer's intentions.

2. It forces a bit of useful discipline on the programmer: you cannot introduce variables willy-nilly; you must think about them enough to pick appropriate types for them. (The compiler's error messages to you, telling you that you apparently forgot to declare a variable, are as often helpful as they are a nuisance: they're helpful when they tell you that you misspelled a variable, or forgot to think about exactly how you were going to use it.)

EXPRESSION

This consists of a single entity like a constant, a variable, an array or a function name. it also consists of some combinations of such entities interconnected by operators.

Example : a, a+b, x=y, c=a+b, x<=y etc........

STATEMENTS

Statements are the "steps" of a program. Most statements compute and assign values or call functions, but we will eventually meet several other kinds of statements as well. By default, statements are executed in sequence, one after another

A statement causes the compiler to carry out some action. There are 3 different types of statements – expression statements compound statements and control statements. Every statement ends with a semicolon.

Example:

```
(1) c=a + b;
(2) {
a=3;
b=4;
c=a+b;
}
(3) if (a<b)
{
printf("\n a is less than b");
}
```

Statement may be single or compound (a set of statements). Most of the statements in a C program are *expression statements.* An expression statement is simply an expression followed by a semicolon.

The lines

```
i = 0;
i = i + 1;
```

and printf("Hello, world!\n"); are all expression statements

SYMBOLIC CONSTANTS

A symbolic constant is a name that substitutes for a sequence of characters, which represent a numeric, character or string constant. A symbolic constant is defined in the beginning of a program by using #define, without ";" at the end.

Example :

```
#define pi 3.1459
#define INTEREST P*N*R/100
```

With this definition it is a program the values of p, n ,r are assigned the value of INTEREST is computed.

Note : symbolic constants are not necessary in a C program.

Review Questions

1. Explain features of preprocessors
2. How is the #include directive is used?
3. What is a pre-processor explain #include ,#define
4. Write a short note on C preprocessors

5. Define Constants in C. Mention the types.
6. Define variable & constant?
7. Discuss the basic structure of a 'C' program.
8. Explain escape sequence character in C?
9. Explain primary data types used in C?
10. Explain symbolic constants used in C?
11. Explain type identifiers in C?
12. Is C a low-level or high-level language? Explain your answer.
13. Name and describe the various datatypes available in C.
14. What are identifier and keywords? Explain it with suitable example.
15. What are the three constants used in C?
16. What do you mean by variables in 'C'?
17. What do you understand by constant, variable and keywords? Discuss the scope of a variable.
18. What is character constant? How character constant is differ from integer constant?
19. What is data type explain the any four data types used in C language?
20. What is purpose of keyword void?
21. What is string constant? How string is constant is differ from character constant?
22. What is the purpose of main( ) function? Can we have a program without main( ).
23. What is variable? What are the rules for defining variables?
24. Why is go to not necessary for the structured programming language like C?
25. Write a rule for declaring character constant.
26. Write a rule for declaring numeric constant.
27. Write a rule for declaring string constant.

CHAPTER 3 OPERATORS AND EXPRESSIONS

The basic operators for performing arithmetic are the same in many computer languages:

- + addition
- - subtraction
- * multiplication
- / division
- % modulus (remainder)

For exponentiations we use the library function **pow**. The order of precedence of these operators is % / * + - . it can be overruled by parenthesis.

Integer division:

Division of an integer quantity by another is referred to integer division. This operation results in truncation. i.e. When applied to integers, the division operator /discards any remainder, so 1 / 2 is 0 and 7 / 4 is 1. But when either operand is a floating-point quantity (type float or double), the division operator yields a floating-point result, with a potentially nonzero fractional part. So 1 / 2.0 is 0.5, and 7.0 / 4.0 is 1.75.

Example :

```
int a, b, c;

a=5;

b=2;

c=a/b;
```

Here the value of c will be 2

Actual value will be resulted only if a or b or a and b are declared floating type. The value of an arithmetic expression can be converted to different data type by the statement (data type) expression.

Example :

```
int a, b;

float c;a=5;b=2;

c=(float) a/b
```

Here c=2.5

Order of Precedence

- Multiplication, division, and modulus all have higher *precedence* than addition and subtraction. The term "precedence" refers to how "tightly" operators bind to their operands (that is, to the things they operate on).
- In mathematics, multiplication has higher precedence than addition, so 1 + 2 * 3 is 7, not 9.
- In other words, 1 + 2 * 3 is equivalent to 1 + (2 * 3). C is the same way.

UNARY OPERATORS

A operator acts up on a single operand to produce a new value is called a unary operator.

(1) the **decrement and increment** operators - ++ and — are unary operators. They increase and decrease the value by 1. if x=3 ++x produces 4 and –x produces 2.

Note : in the place of ++x , x++ can be used, but there is a slight variation. In both case x is incremented by 1, but in the latter case x is considered before increment.

(2) **sizeof** is another unary operator

int x, y;

y=sizeof(x);

The value of y is 2 . the *sizeof* an integer type data is 2 that of float is 4, that of double is 8, that of char is 1.

RELATIONAL AND LOGICAL OPERATORS

< (less than),

<= (less than or equal to),

> (greater than),

>= (greater than or equal to),

= = (equal to) and

!= (not equal to) are relational operators.

A logical expression is expression connected with a relational operator.

For example 'b*b – 4*a*c< 0 is a logical expression. Its value is either true or false.

int i, j, k ;

i=2;

j=3 ;

k=i+j ;

k>4 has the value true k<=3 has the value false.

LOGICAL OERATORS

The relational operators work with arbitrary numbers and generate true/false values. You can also combine true/false values by using the *Boolean operators*, which take true/false values as operands and compute new true/false values

The three Boolean operators are:

- && and
- || or
- ! not (takes one operand; "*unary*")

The && ("and") operator takes two true/false values and produces a true (1) result if both operands are true (that is, if the lefthand side is true **and** the right-hand side is true).

The || ("or")operator takes two true/false values and produces a true (1) result if either operand is true. The ! ("not") operator takes a single true/false value and negates it, turning false to true and true to false (0 to 1 and nonzero to 0).

&& (and) and || (or) are logical operators which are used to connect logical expressions.

Where as ! (not) is unary operator, acts on a single logical expression.

For example, 1. (a<5) && (a>-2)

2. (a<=3) || (b>2)

In the first example if a= -3 or a=6 the logical expression returns true.

ASSIGNMENT OPERATORS

These operators are used for assigning a value of expression to another identifier.

=, + =, - = , * =, /= and %= are assignment operators.

a = b+c results in storing the value of b+c in 'a'.

a += 5 results in increasing the value of a by 5

a /= 3 results in storing the value a/3 in a and it is equivalent a= a/3

Note : 1. if a floating point number is assigned to a integer type data variable, the value will be truncated.

Example : float a=5.36;

int b;

b=a

It results in storing 5 to b.

Similarly if an integer value is a assigned to a float type like float x=3 the value of x stored is 3.0.

CONDITIONAL OPERATOR

The operator ?: is the conditional operator. It is used as

variable 1 = expression 1 ? expression 2 : expression 3.

Here expression 1 is a logical expression and expression 2 and expression 3 are expressions having numerical values. If expression 1 is true, value of expression 2 is assigned to variable 1 and otherwise expression3 is assigned.

Example :

int a,b,c,d,e

a=3;b=5;c=8;

d=(a<b)? a : b;

e=(b>c) ? b : c;

Then d=3 and e=8

LIBRARY FUNCTIONS

They are built in programs readily available with the C compiler. These functions perform certain operations or calculations. Some of these functions return values when they are accessed and some carry out certain operations like input, output. A library function accessed in a used written program by referring its name with values assigned to necessary arguments.

Some of these library functions are :

abs(i),ceil(d), cos(d), cosh(d), exp(d), fabs(d),floor(d), getchar( ), log(d),pow(d,d'),printf( ), putchar(c), rand( ), sin(d), sqrt(d), scanf( ), tan(d), toascii(c),toupper(c), tolower(c).

Note : the arguments i, c, d are respectively integer, char and double type.

Example:

```
#include<math.h>

#include<stdio.h>

#include<conio.h>

main( )

{

float x, s;

printf(" \n input the values of x :");

scanf("%f ", &x);

s=sqrt(x);

printf("\n the square root is %f ",s);

}
```

Review Questions

1. Distinguish between binary minus and unary minus.
2. List out the different operators involve for comparision and logical decision making in C.
3. List out the five arithmetic operators in C.
4. What are the rules to use period(.) operator.
5. What is an expression? How is an expression different from the variables?
6. What is mean by conditional expression?
7. What is mean by the comparision and logical operator? How are they different from the arithmetic and assignment operator?
8. What is mean by the equality operator? How do these differ from an assignment operator.

9. What is modulus operator and how does it operate in C.
10. What is the associtivity rules involve in this operator.
11. What is unary operator? List out the different operator involve in the unary operator.
12. Can multiple assignments be written in C. In what order will the assignment be carried out.
13. Describe arithmetic operator?
14. Differentiate between relational and logical operators used in C?
15. Discuss Precedence order and associativity of operators.
16. Discuss the conditional operator with the help of a program.
17. Explain * operator and & operator with example.
18. Explain conditional operator with example.
19. Explain logical operators and expressions used in C?
20. Explain sizeof operator with example.
21. Explain the difference between '=' and '==' operator explain with example?
22. Explain with example ++i and i++.
23. What are Operators? Mention their types in C.
24. What do you understand by operators? Explain the use of the following operators :
25. What is an expression? How is an expression differing from variables?
26. What is the modulus operator and how does it works explain it with example
27. What is the purpose of comma operator within which statement does the comma operator usually appear.
28. What is unary operator?
29. Write a short note precedence & order of evaluation?

CHAPTER 4 DATA INPUT & OUTPUT

For inputting and outputting data we use library function. The important of these functions are getch( ), putchar( ), scanf( ), printf( ), gets( ), puts( ). For using these functions in a C-program there should be a preprocessor statement

#include<stdio.h>.

[A preprocessor statement is a statement before the main program, which begins with # symbol.] **stdio.h** is a header file that contains the built in program of these standard input output function.

getchar function - It is used to read a single character (char type) from keyboard.

The syntax is **char variable name = getchar( );**

Example:

```
char c,
c = getchar( );
```

For reading an array of characters or a string we can use getchar( ) function.

Example:

```
#include<stdio.h>
main( )
{
char place[80];
int i;
for(i = 0;( place [i] = getchar( ))! = '\n', ++i);
}
```

This program reads a line of text.

putchar function - It is used to display single character.

The syntax is **putchar(char c);**

Example:

```
char c;
c = 'a';
putchar(c);
```

Using these two functions, we can write a very basic program to copy the input, a character at a time, to the output:

```
#include <stdio.h>
/* copy input to output */
main()
{
int c;
c = getchar();
while(c != EOF)
{
putchar(c);
c = getchar();
}
return 0;
}
```

scanf function - This function is generally used to read any data type- int, char, double, float, string. The syntax is **scanf (control string, list of arguments);**

The control string consists of group of characters, each group beginning % sign and a conversion character indicating the data type of the data item. The conversion characters are c,d,e,f,o,s,u,x indicating the type resp. char decimal integer, floating point value in exponent form, floating point value with decimal point, octal integer, string, unsigned integer, hexadecimal integer. ie, "%s", "%d" etc are such group of characters.

An example of reading a data:

```
#include<stdio.h>
main( )
{
char name[30], line;
int x;
float y;
.........
..........
scanf("%s%d%f", name, &x, &y);
scanf("%c",line);
}
```

NOTE:

1) In the list of arguments, every argument is followed by & (ampersand symbol) except string variable.

2) s-type conversion applied to a string is terminated by a blank space character. So string having blank space like "Govt. Victoria College" cannot be read in this manner. For reading such a string constant we use the conversion string as "%[^\n]" in place of "%s".

Example:

```
char place[80];
................
scanf("%[^\n]", place);
.................
```

with these statements a line of text (until carriage return) can be input the variable 'place'.

printf function

This is the most commonly used function for outputting a data of any type.

The syntax is **printf(control string, list of arguments)**

Here also control string consists of group of characters, each group having % symbol and conversion characters like c, d, o, f, x etc.

Example:

```
#include<stdio.h>
' main()
{
int x;
scanf("%d",&x);
x*=x;
printf("The square of the number is %d",x);
}
```

Note that in this list of arguments the variable names are without &symbol unlike in the case of scanf( ) function. In the conversion string one can include the message to be displayed. In the above example "The square of the number is" is displayed and is followed by the value of x. For writing a line of text (which include blank spaces)the conversion string is "%s" unlike in scanf function. (There it is "[^\n]").

More about printf statement

There are quite a number of format specifiers for printf. Here are the basic ones :

%d	print an int argument in decimal
%ld	print a long int argument in decimal
%c	print a character
%s	print a string
%f	print a float or double argument
%e	same as %f, but use exponential notation
%g	use %e or %f, whichever is better
%o	print an int argument in octal (base 8)
%x	print an int argument in hexadecimal (base 16)
%%	print a single %

To illustrate with a few more examples:

the call

```
printf("%c %d %f %e %s %d%%\n", '1', 2, 3.14, 56000000.,"eight", 9);
```

would print 1 2 3.140000 5.600000e+07 eight 9%

The call

```
printf("%d %o %x\n", 100, 100, 100);
```

would print 100 144 64

Successive calls to printf just build up the output a piece at a time, so the calls

printf("Hello, ");

printf("world!\n");

would also print Hello, world! (on one line of output).

While inputting or outputting data field width can also be specified. This is included in the conversion string.(if we want to display a floating point number convert to 3 decimal places the conversion string is "%.3f").For assigning field width, width is placed before the conversion character like"%10f","%8d","%12e" and so on...Also we can display data making correct to a fixed no of decimal places.

For example if we want to display x=30.2356 as 30.24 specification may be "%5.2f" or simply "%.2f".

Review Questions

1. Distinguish between i)fprintf() ii)fscanf()
2. Distinguish between getch and getc
3. Distinguish between putch and putc
4. Explain i)fprintf() ii)fscanf()
5. Explain putc and getc in brief
6. What are the format codes used along with the scanf(). Display the various data types in C.
7. What are the salient features of standard input and output file
8. What is the printf() and compare with putchar().
9. What is the scanf() and how does it differ from the getchar().
10. Difference between formatted & unformatted statement ?
11. Explain - a) getc() b) putc()
12. Explain Getw() & Putw() function
13. Explain printf() function with an example
14. Explain the following functions i)getch() and ii) clrscr()

CHAPTER 5 CONTROL STATEMENTS

When we run a program, the statements are executed in the order in which they appear in the program. Also each statement is executed only once. But in many cases we may need a statement or a set of statements to be executed a fixed no of times or until a condition is satisfied. Also we may want to skip some statements based on testing a condition. For all these we use control statements.

Control statements are of two types – branching and looping.

BRANCHING

It is to execute one of several possible options depending on the outcome of a logical test, which is carried at some particular point within a program.

LOOPING

It is to execute a group of instructions repeatedly, a fixed number of times or until a specified condition is satisfied.

BRANCHING

1. if else statement

It is used to carry out one of the two possible actions depending on the outcome of a logical test. The else portion is optional.

The syntax is

If (expression) statement1 [if there is no else part]

Or

If (expression)

Statement 1

else

Statement 2

Here expression is a logical expression enclosed in parenthesis. If expression is true, statement 1 or statement 2 is a group of statements, they are written as a block using the braces { }

Example:

1. if(x<0) printf("\n x is negative");

2. if(x<0)

printf("\n x is negative");

else

printf("\n x is non negative");

3.if(x<0)

{

x=-x;

s=sqrt(x);

}

else

s=sqrt(x);

2. nested if statement

Within an if block or else block another if -- else statement can come. Such statements are called nested if statements.

The syntax is

If (e1)

s1

if (e2)

s2

else

s3

else

s4

3. Ladder if statement

Inorder to create a situation in which one of several courses of action is executed we use ladder – if statements.

The syntax is

If (e1) s1

else if (e2) s2

else if (e3) s3

....

else sn

Example: if(mark>=90) printf('\n excellent");

else if(mark>=80) printf("\n very good");

else if(mark>=70) printf("\n good");

else if(mark>=60) printf("\n average");

else

printf("\n to be improved");

SWITCH STATEMENT

It is used to execute a particular group of statements to be chosen from several available options. The selection is based on the current value of an expression with the switch statement.

The syntax is:

switch(expression)

{

```
case value1:
s1
break;
case value 2:
s2
break;
.......
........
default:
sn
}
```

All the option are embedded in the two braces { }.Within the block each group is written after the label case followed by the value of the expression and a colon. Each group ends with '***break***' statement. The last may be labelled ***'default'.*** This is to avoid error and to execute the group of statements in default if the value of the expression does not match value1, value2,........

LOOPING

1. The while statement

This is to carry out a set of statements to be executed repeatedly until some condition is satisfied.

The syntax is:

While (expression) statement

The statement is executed so long as the expression is true. Statement can be simple or compound.

Example 1:

```
#include<stdio.h>
while(n > 0)
{
printf("\n");
n = n - 1;
}
```

Example 2:

```
#include<stdio.h>
main()
{
int i=1;
while(x<=10)
{
printf("%d",i);
++i;
}
}
```

2. *do while statement*

This is also to carry out a set of statements to be executed repeatedly so long as a condition is true.

The syntax is: **do statement while(expression)**

Example:

```
#include<stdio.h>
main()
{
int i=1;
do
{
printf("%d",i);
++i;
}while(i<=10);
}
```

THE DIFFERENCE BETWEEN while loop AND do – while loop

1) In the while loop the condition is tested in the beginning whereas in the other case it is done at the end.

2) In while loop the statements in the loop are executed only if the condition is true. whereas in do – while loop even if the condition is not true the statements are executed atleast once.

3. for loop

It is the most commonly used looping statement in C.

The general form is **For(expression1;expression2;expression3)statement**

Here expression1 is to initialize some parameter that controls the looping action. expression2 is a condition and it must be true to carry out the action. expression3 is a unary expression or an assignment expression.

Example:

```
#include<stdio.h>
main()
{
int i;
for(i=1;i<=10;++i)
printf("%d",i);
}
```

Here the program prints *i* starting from 1 to 10.First *i* is assigned the value 1 and then it checks whether *i*<=10 If so i is printed and then *i* is increased by one. It continues until *i*<=10. An example for finding the average of 10 numbers.

```
#include<stdio.h>
main()
{
int i;
float x,avg=0;
for(i=1;i<=10;++i)
{
scanf("%f",&x);
avg += x;
}
avg /= 10;
printf("\n average -%f",avg);
}
```

Note [illegible]

Example [illegible]

for(j=1;j<=10;++j);

The break statement

The break statement is used to terminate the loop or to exit from a switch. It is used in for, while, do-while and switch statement.

The syntax is ***break;***

Example 1: A program to read the sum of positive numbers only

```
#include<stdio.h>
main()
{
int x, sum=0;
int n=1;
while(n<=10)
{
scanf("%d",&x);
if(x<0) break;
sum+=x;
}
printf("%d",sum);
}
```

Example 2 :A program for printing prime numbers between 1 and 100:

```
#include <stdio.h>
#include <math.h>
main()
{
int i, j;
printf("%d\n", 2);
for(i = 3; i <= 100; i = i + 1)
{
for(j = 2; j < i; j = j + 1)
{
if(i % j == 0)
break;
if(j > sqrt(i))
{
printf("%d\n", i);
break;
}
}
}
return 0;
}
```

Here while loop breaks if the ınput for x is –ve.

The continue statement

It is used to bypass the remainder of the current pass through a loop. The loop does not terminate when continue statement is encountered, but statements after continue are skipped and proceeds to the next pass through the loop.

In the above example of summing up the non negative numbers when a negative value is input, it breaks and the execution of the loop ends. In case if we want to sum 10 nonnegative numbers, we can use *continue* instead of *break*

Example :

```
#include<stdio.h>
main()
{
int x, sum=0, n=0;
while(n<10)
{
scanf("%d",x);
if(x<0) continue;
sum+=x;
++n;
}
printf("%d",sum);
}
```

GO TO statement

It is used to alter the normal sequence of program execution by transferring control to some other part of the program .

The syntax is **goto** *label* ;

Example :

```
#include<stdio.h>
main( )
{
int n=1,x,sum=0;
while(n<=10)
{
scanf("%d" ,&x);
if(x<0)goto error;
sum+=x;
++n;
}
error:
printf("\n the number is non negative");
}
```

Review Questions

1. Compare while loop and for loop with example.
2. What is looping in C? What are the advantages of looping?
3. What is the role played by the break statement within the switch statement. Explain with example.
4. What is use of continue in C.
5. Differentiate between if-else-if and switch statement.
6. Explain for loop?
7. Explain nested for loop with an example
8. Explain nested if – else with example.
9. Explain switch statement with its syntax and example.
10. Explain syntax and use of Do__While statement
11. How does Switch statement differ from Nested if?
12. What are the different decision control structure available in C. Explain with examples.
13. What are the various loop constructs available in C. Distinguish between while and do- while loops.
14. What is Nested if else explain with an example?
15. What is use of if statement?
16. What the term 'Nesting' refers to? Explain with the help of an example.
17. Why do we avoid the use of goto statements in programs?
18. write a syntax of while loop?
19. Write disadvantages of goto statement.

CHAPTER 6 FUNCTIONS

Functions are programs .There are two types of functions- library functions and programmer written functions. We are familiarised with library functions and how they are accessed in a C program.

The advantages of function programs are many

1) A large program can be broken into a number of smaller modules.

2) If a set of instruction is frequently used in program and written as function program, it can be used in any program as library function.

Defining a function

Generally a function is an independent program that carries out some specific well defined task. It is written after or before the main function. A function has two components-definition of the function and body of the function.

Generally it looks like

```
datatype function name(list of arguments with type)
{
statements
return;
}
```

If the function does not return any value to the calling point (where the function is accessed).

The syntax looks like

```
function name(list of arguments with type)
{
statements
return;
}
```

If a value is returned to the calling point, usually the return statement looks like return(value).In that case data type of the function is executed. Note that if a function returns no value the keyword **void** can be used before the function name

Example:

(1)

```
writecaption(char x[] );
{
printf("%s",x);
return;
}
```

(2)

```
int maximum(int x, int y)
{
int z ;
z=(x>=y)? x : y ;
return(z);
}
```

(3)

```
maximum( int x,int y)
{
int z;
z=(x>=y) ? x : y ;
printf("\n maximum =%d",z);
return ;
}
```

Note: In example (1) and (2) the function does not return anythir g.

Advantages of functions

1. It appeared in the main program several times, such that by making it a function, it can be written just once, and the several places where it used to appear can be replaced with calls to the new function.

2. The main program was getting too big, so it could be made (presumably) smaller and more manageable by lopping part of it off and making it a function.

3. It does just one well-defined task, and does it well.

4. Its interface to the rest of the program is clean and narrow

5. Compilation of the program can be made easier.

Accessing a function

A function is accessed in the program (known as calling program)by specifying its name with optional list of arguments enclosed in parenthesis. If arguments are not required then only with empty parenthesis.

The arguments should be of the same data type defined in the function definition.

Example:

1) int a,b,y;

y=maximum(a,b);

2) char name[50] ;

writecaption(name);

3) arrange();

If a function is to be accessed in the main program it is to be defined and written before the main function after the preprocessor statements.

Example:

```
#include<stdio.h>
int maximum (int x,int y)
{
int z ;
z=(x>=y) ? x : y ;
return (z);
}
main( )
{
int a,b,c;
scanf("%d%d",&a,&b);
c=maximum(a,b);
printf("\n maximum number=%d",c);
}
```

FUNCTION PROTOTYPE

It is a common practice that all the function programs are written after the main( ) function. When they are accessed in the main program, an error of prototype function is shown by the compiler. It means the computer has no reference about the programmer defined functions, as they are accessed before the definition .To overcome this, i.e to make the compiler aware that the declarations of the function referred at the calling point follow, a declaration is done in the beginning of the program immediately after the preprocessor statements. Such a declaration of function is called prototype declaration and the corresponding functions are called function prototypes.

Example 1:

```
#include<stdio.h>
int maximum(int x,int y);
main( )
{
int a,b,c;
scanf("%d%d",&a,&b);
c=maximum(a,b);
printf("\n maximum number is : %d",c);
}
int maximum(int x, int y)
{
int z;
z=(x>=y) ? x : y ;
return(z);
}
```

Example 2:

```
#include<stdio.h>
void int factorial(int m);
main( )
{
int n;
scanf("%d",&n);
factorial(n);
}
void int factorial(int m)
{
int i,p=1;
for(i=1;i<=m;++i)
p*=i;
printf("\n factorial of %d is %d ",m,p);
return( );
}
```

Note: In the prototype decleration of function, if it return no value, in the place of data-type we use void.

Eg: void maximum(int x,int y);

Passing arguments to a function

The values are passed to the function program through the arguments. When a value is passed to a function via an argument in the calling statement, the value is copied into the formal argument of the function (may have the same name of the actual argument of the calling function). This procedure of passing the value is called passing by value. Even if formal argument changes in the function program, the value of the actual argument does not change.

Example:

```
#include<stdio.h>
void square (int x);
main( )
{
int x;
scanf("%d",&x);
square(x):
}
void square(int x)
{
x*=x ;
printf("\n the square is %d",x);
return;
}
```

In this program the value of x in the program is unaltered.

RECURSION

It is the process of calling a function by itself, until some specified condition is satisfied. It is used for repetitive computation (like finding factorial of a number) in which each action is stated in term of previous result

Example:

```
#include<stdio.h>
long int factorial(int n);
main( )
{
int n;
long int m;
scanf("%d",&n);
m=factorial(n);
printf("\n factorial is : %d", m);
}
long int factorial(int n)
{
if (n<=1)
return(1);
else
return(n*factorial(n-1));
}
```

In the program when n is passed the function, it repeatedly executes calling the same function for n, n-1, n-2,..................1.

Review Questions

1. List out the advantages of function.
2. List out the rules used in return statement
3. What is mean by call by reference & call by value.
4. What is the function and list out advantages and disadvantages of functions
5. What is the purpose of return statement
6. What is the recursive function. List out their merits and demerits.
7. Explain function with argument and return type.
8. Explain recursion?
9. Explain the difference between calling function and called function?
10. Explain void function?
11. State three advantages of function?
12. What is call by value?
13. What is function ?how function is defined.
14. What is function?
15. What is recursion explain with suitable example.
16. What is the difference between call by value and call by reference
17. What is the purpose of the library function fflush()?
18. What is user defined functions and built-in functions. Enlist them.

CHAPTER 7 STORAGE CLASS

Earlier we mentioned that variables are characterized by their data type like integer, floating point type, character type etc. Another characteristic of variables or arrays is done by storage class. It refers to the permanence and scope of variables or arrays within a program.

There are 4 different storage class specifications in C

- automatic,
- external,
- static, and
- register.

They are identified by the key words auto, external, static, and register respectively.

AUTOMATIC VARIABLES

They are declared in a function. It is local and its scope is restricted to that function. They are called so because such variables are created inside a function and destroyed automatically when the function is exited. Any variable declared in a function is interpreted as an automatic variable unless specified otherwise. So the keyword auto is not required at the beginning of each declaration.

EXTERNAL VARIABLE (GLOBAL VARIABLE)

The variables which are alive and active throughout the entire program are called external variables. It is not centered to a single function alone, but its scope extends to any function having its reference. The value of a global variable can be accessed in any program which uses it. For moving values forth and back between the functions, the variables and arrays are declared globally i.e., before the main program. The keyword *external* is not necessary for such declaration, but they should be mentioned before the main program.

STATIC VARIABLES

It is, like automatic variable, local to functions is which it is defined. Unlike automatic variables static variable retains values throughout the life of the program, i.e. if a function is exited and then re-entered at a later time the static variables defined within the function will retain their former values. Thus this feature of static variables allows functions to retain information permanently throughout the execution of the program. Static variable is declared by using the

keyword static.

Example :

```
static float a ;
Static int x ;
```

Consider the function program:

```
# include<stdio.h>
long int Fibonacci (int count )
main()
```

```
{
int i, m=20;
for (i =1 ; i < m ; ++i)
printf( "%ld\t",fibonacci(i));
}
long int Fibonacci (int count )
{
static long int fl=1, f2=1 ;
long int f ;
f = (count < 3 ) ? 1 : fl + f2 ;
f2 = fl
fl= f ;
return (f ) ;}
```

In this program during the first entry to the function fl and f2 are assigned 1, later they are replaced by successive values of fl and f. as fl and f2 are declared static storage class. When the function is exited the latest values stored infl and f2 will be retained and used when the function is re-entered.

Review Questions

1. How are the data elements initialized in the case of static type variable
2. How can data be initialized in the automatic variable
3. What is mean by register variable and what the scope of it?
4. What is the automatic variable and what is the use of it.
5. What is the storage class used in recursive function
6. What is the use of external data type in C
7. Differentiate between local variable and global variable?
8. Explain Automatic storage class specifier
9. Explain Extern storage class
10. Explain Register storage class
11. Explain Static storage classs
12. Explain the various storage classes in C.
13. How static variable are define and initialized?
14. What are register variables? What are the advantage of using register variables?
15. What is mean by storage class of variable?

CHAPTER 8

ARRAYS

ARRAYS

An array is an identifier to store a set of data with common name. Note that a variable can store only a single data. Arrays may be one dimensional or multi dimensional.

Defining an array one dimensional arrays

Definition: Arrays are defined like the variables with an exception that each array name must be accompanied by the size (i.e. the max number of data it can store).For a one dimensional array the size is specified in a square bracket immediately after the name of the array.

The **syntax** is

data-type array name[size];

So far, we've been declaring simple variables: the declaration

int i;

declares a single variable, named i, of type int. It is also possible to declare an *array* of several elements. The declaration

int a[10];

declares an array, named a, consisting of ten elements, each of type int. Simply speaking, an array is a variable that can hold more than one value. You specify which of the several values you're referring to at any given time by using a numeric *subscript*. (Arrays in programming are similar to vectors or matrices in mathematics.) We can represent the array a above with a picture like this:

eg:

int x[100];

float mark[50];

char name[30];

Note: With the declaration int x[100],computer creates 100 memory cells with name x[0],x[1],x[2],.........,x[99].Here the same identifier x is used but various data are distinguished by the subscripts inside the square bracket.

Array Initialization

Although it is not possible to assign to all elements of an array at once using an assignment expression, it is possible to initialize some or all elements of an array when the array is defined. The syntax looks like this:

int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

The list of values, enclosed in braces {}, separated by commas, provides the initial values for successive elements of the array.

If there are fewer initializers than elements in the array, the remaining elements are automatically initialized to 0. For example,

```
int a[10] = {0, 1, 2, 3, 4, 5, 6};
```

would initialize a[7], a[8], and a[9] to 0. When an array definition includes an initializer, the array dimension may be omitted, and the compiler will infer the dimension from the number of initialisers.

For example,

```
int b[] = {10, 11, 12, 13, 14};
```

Example :

```
int x[ ] ={0,1,2,3,4,5}; or
int x[6]={0,1,2,3,4,5};
```

Even if the size is not mentioned (former case) the values 0,1,2,3,4 are stored in x[0],x[1],x[2],x[3],x[4],x[5]..If the statement is like

int x[3]={0,1,2,3,4,5}; then x[0],x[1],x[2] are assigned the values 0,1,2.

Note: If the statement is like

```
int x[6]={0,1,2};
```

then the values are stored like x[0]=0, x[1]=1, x[2]=2, x[3]=0, x[4]=0 and x[5]=0.

Processing one dimensional array

1) Reading arrays: For this normally we use for- loop.

If we want to read n values to an array name called 'mark' , the statements look like

```
int mark[200],i,n;
for(i=1;i<=n;++i)
scanf("%d",&x[i]);
```

Note: Here the size of array declared should be more than the number of values that are intended to store.

2) Storing array in another: To store an array to another array. Suppose a and b are two arrays and we want to store that values of array a to array b. The statements look like

```
float a[100],b[100];
int I;
for(i=1;i<=100;++i)
b[i]=a[i];
```

Problem: To find the average of a set of values.

```
#include<stdio.h>
main( )
{
int x,i;
float x[100],avg=0;
printf("\n the no: of values ");
scanf("%d",&n);
printf("\n Input the numbers");
for(i=1;i<=n;++i)
{
scanf("%f",&x[i]);
avg=avg+x[i];
}
avg=avg/n;
printf("\n Average=%f",avg);
}
```

PASSING ARRAYS TO FUNCTION

Remember to pass a value to a function we include the name of the variable as an argument of the function. Similarly an array can be passed to a function by including array name (without brackets) and size of the array as arguments. In the function defined the array name together with empty square brackets is an argument.

Example:

(calling function)-avg= average (n, x); where n is the size of the data stored in the array x[].

(function defined)- float average (int n, float x[]);

Now let us see to use a function program to calculate the average of a set of values.

```
#include<stdio.h>
float average(int n,float y[]);
main()
{
int n;
float x[100],avg;
printf("\n Input the no: of values");
scanf("%d",&n);
printf("\n Input the values");
for(i=1;i<=n;++i)
scanf("%f",&x[i]);
avg=average(n,x);
printf("\n The average is %f",avg);
}
float average(int n, float y[]);
{
float sum=0;
int i;
for(i=1;i<=n;++i)
sum=sum+y[i];
sum=sum/n;
return(sum);
}
```

Note:

1) In the function definition the array name together with square brackets is the argument. Similarly in the prototype declaration of this function too, the array name with square brackets is the argument

2) We know that changes happened in the variables and arrays that are in function will not be reflected in the main (calling) program even if the same names are usual. If we wish otherwise the arrays and variables should be declared globally. This is done by declaring them before the main program.

Example:

```
#include<stdio.h>
void arrange(int n,float x[]);
main();
{
..........
arrange(n,x);
.............
}
arrange(int n,float x[]);
{
.........
return;
}
```

Problem : Write a program to arrange a set of numbers in ascending order by using a function program with global declaration.

MULTI-DIMENSIONAL ARRAYS

Multi-dimensional arrays are defined in the same manner as one dimensional arrays except that a separate pair of square brackets is required to each subscript.

Example:

float matrix[20][20] (two dimensional)

Int x[10][10][5] (3-dimensional)

Initiating a two dimensional array we do as int x[3][4]={1,2,3,4,5,6,7,8,9,10,11,12} Or

int x[3][4]= {{1,2,3,4};

{5,6,7,8};

{89,10,11,12};}

NOTE: The size of the subscripts is not essential for initialization. For reading a two dimensional array we use two for-loop.

Example:

for(i=1;i<=2;++i)

for(j=1;j<=3;++j)

scanf("%f",&A[i][j]);

NOTE: If x[2][3] is a two dimensional array, the memory cells are identified with name x[0][0],x[0][1],x[0][2],x[1][0],x[1][1] and x[1][2].

ARRAYS AND STRINGS.

A string is represented as a one dimensional array of character type.

Example : char name[20];

Here name is an array that can store a string of size 20.

If we want to store many strings(like many names or places) two dimensional array is used. Suppose we want to store names of 25 persons, then declare name as charname[25][ ]. Note that the second square bracket is kept empty if the length of string is not specified.

If the declaration is char name[25][30], 25 names of maximum size 30 can be stored. The various names are identified by name[0], name[1], name[2],........, name[24].

These names are read by the command.

For(i=0; i<25,++i)

Scanf("%[^\n]", name(i));

PROBLEM: Write a program to store the names and places of students in your class.

Review Questions

1. Distinguish between character array and string array
2. Explain applications of array
3. Summarise the purpose of string.h function.
4. What is an array indexing explain with an example
5. What is character array how it differs from other data types.
6. What is an array and how array variable differs from ordinary variable.
7. Can an array be used as an argument to a function? If yes. explain with examples.
8. Define array and how two – dimensional array is initialize?
9. Explain one dimensional array with an example
10. explain- a) strlen() b) strcat()
11. Explain any 4 string functions with suitable example?
12. Explain any two string functions?
13. Explain applications of array
14. Explain the concepts of multidimensional arrays in 'C' Language.
15. Explain the functions of the following : (a)Strcpy () (b)Strlen () (c)Strcat ()
16. Explain Two dimensional array with an example
17. For what purpose '\0' is used in string operations explain with suitable example.
18. In what ways does an array differ from an ordinary variable? What advantage is there in defining an array size in terms of symbolic constant rather than a fixed integer constant?
19. What are the rules to declare one dimensional array?
20. What are the rules used to declare a multi dimensional array
21. What are the rules used to declare a one dimensional array
22. What are the rules used to declare a two dimensional array
23. What do you understand by multi dimensional arrays.
24. What do you understand by strings? How do you declare a string?
25. What is an array? Explain the features of an array and their uses.
26. What is null string ? What is it's length?

CHAPTER 9 STRUCTURES AND UNIONS

We know an array is used to store a collection of data of the same type. But if we want to deal with a collection of data of various type such as integer, string, float etc we use structures in C language. It is a method of packing data of different types. It is a convenient tool for handling logically related data items of bio-data people comprising of name, place, date etc. , salary details of staff comprising of name, pay da, hra etc.

Defining a structure

In general it is defined with the syntax name **struct** as follows

```
Struct structure_name
{
Data type variable1;
Data type variable2;
...
}
```

For example

```
Struct account
{
Int accountno
Char name[50];
Float balance;
}customer[20]
```

Note :

Here accountno, name and balance are called members of the structure

```
Struct  date
{
Int month;
Int day;
Int year;
}dateofbirth;
```

In these examples customer is a structure array of type account and date of birth is a structural type of date. Within a structure members can be structures. In the following example of biodata structure date which is a structure is a member.

For example

```
struct date
{
```

```
Int day;
Int month;
Int year;
}
Struct biodata
{
Name char[30];
Int age ;
Date birthdate;
}staff[30];
```

Here staff is an array of structure of type biodata

Note: we can declare other variables also of biodata type structure as follows.

Struct biodata customer [20]; , Struct biodata student; etc

Processing a structure

The members of a structure are themselves not variable. They should be linked to the structure variable to make them meaningful members. The linking is done by period .)

If staff[] is structure array then the details of first staff say staff[1] is got by staff[1].name, staff[1].age, staff[1].birthdate.day, staff[1].birthdate.month, staff[1].birthdate.year

we can assign name, age and birthdate of staff[1] by

```
Staff[1].name="Jayachandran"
staff[1].age=26
staff[1].birthdate.day=11
staff[1].birthdate.month=6
staff[1].birthdate.year=1980
```

If 'employee' is a structure variable of type biodata as mentioned above then the details of 'employee' is got by declaring 'employee as biodata type by the statement biodata employee; The details of employee are got by employee.name, employee.age, employee.birthdate.year etc.

Note:

Structure initialisation

Like any other variable or array a structure variable can also be initalised. by using

syntax static

```
Struct record
{
Char name[30];
Int age;
Int weight;
}
```

Static struct record student1={"rajan", 18, 62}

Here student1 is of record structure and the name, age and weight are initialised as "rajan", 18 and 62 respectively.

1. Write a c program to read biodata of students showing name, place, pin, phone and grade

```
#include<stdio.h>
main()
{
Struct biodata
{
Char name[30];
Char Place[40]
Int pin;
Long Int phone;
Char grade;
};
Struct biodata student[50];
Int n;
Prinf("\n no of students");
Scanf("%d",n);
For(i=1;i<=n;++i)
{
Scanf("%s",student[i].name);
Scanf("%s",student[i].place);
Scanf("%d",student[i].pin);
Scanf("%ld",student[i].phone);
Scanf("%c",student[i].grade);
}
}
```

USER DEFIINED DATA TYPE

This is to define new data type equivalent to existing data types. Once defined a user-defined data type then new variables can be declared in terms of this new data type. For defining new data type we use the syntax typedef as follows

Typedef type new-type

Here type refers to existing data type

For example

Example 1:

```
Typedef int integer;
```

Now integer is a new type and using this type variable, array etc can be defined as

```
Integer x;
Integer mark[100];
```

Example 2:

```
Typedef struct
{
Int accno;
Char name[30];
Float balance;
}record;
```

Now record is structure type using this type declare customer, staff as record type

```
Record customer;
Record staff[100];
```

Passing structures to functions

Mainly there are two methods by which structures can be transferred to and from a function.

1 . Transfer structure members individually

2 . Passing structures as pointers (ie by reference)

Example 1

```
#include<stdio.h>
Typedef struct
{
Int accno;
Char name[30];
Float balance;
}record;
main()
{
.....
Record customer;
.....
Customer.balance=adjust(customer.name,customer.accno,balance)
.....
}
Float adjust(char name[], int accnumber, float bal)
{
Float x;
.....
X=
.....
Return(x);
}
```

Example 2

```
#include<stdio.h>
Typedef struct
{
Int accno;
Char name[30];
Float balance;
}record;
main()
{
Record customer;
Void adjust(record *cust)
......
Adjust(&customer);
Printf("\n %s\t%f",coustomer.name,customer.balance)
}
Void adjust(record *cust)
{
Float x;
....
Cust->balance=...
....
Return;
}
```

In the first example structure members are passed individually where as in the second case customer is passed entirely as a pointer named cust. The values of structure members are accessed by using -> symbol like cust->name, cust->balance etc.

UNIONS

Union is a concept similar to a structure with the major difference in terms of storage. In the case of structures each member has its own storage location, but a union may contain many members of different types but can handle only one at a time. Union is also defined as a structure is done but using the syntax union.

```
Union var
{
Int m;
Char c;
Float a;
}
Union var x;
```

Now x is a union containing three members m,c,a. But only one value can be stored either in x.m, x.c or x.a

BITWISE OPEATIONS

Bitwise operators operate on individual bits of integer (int and long) values. This is useful for writing low-level hardware or OS code where the ordinary abstractions of numbers, characters, pointers, and so on, are insufficient. Bit manipulation code tends to be less "portable". Code is "portable" if without any programmer intervention it compiles and runs correctly on different types of computers. The bitwise operations are commonly used with unsigned types. These operators perform bit wise logical operations on values. Both operands must be of the same type and width: the resultant value will also be this type and width.

Biwise operators are:

~	Bitwise Negation one's complement (unary)
&	Bitwise And
|	Bitwise Or
^	Bitwise Exclusive Or
>>	Right Shift by right hand side (RHS) (divide by power of 2)
<<	Left Shift by RHS (multiply by power of 2)

Definition of bitwise logical operators:

Bit a	Bit b	a & b	a | b	a ^ b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Example:

Bitwise Negation

```
 a=00000011
~a =11111100
```

Bitwise And

```
a   = 00000011
b   = 00010110
a&b=00000010
```

Bitwise Or

```
a  = 00000011
b  = 00010110
a|b=00010111
```

Bitwise Exclusive Or

```
a =  00000011
b =  00010110
a^b=00010101
```

Right Shift

```
a = 0000 0011 = 3
(a<<=1) = 00000110 = 6
(a<<=2) = 00011000 = 24
(a<<=3) = 11000000 = 192
```

Left Shift

```
a=11000000    =192
(a>>=1) = 01100000 = 96
(a>>=2) = 00011000 = 24
(a>>=3) = 0000 0011 = 3
```

Review Questions

1. Explain the salient features of typedef ?
2. How structure different from array
3. What is the difference between structure declaration and structure initialization
4. What is the structure and what are the uses of it.
5. Differentiate structure and array
6. Distinguish structure data type with other data type variables.,
7. Explain array of structure with example
8. Explain Nested structure with example.
9. Explain pointer to structure.
10. Explain the use of Typedef
11. What are the similarities and difference between structure and union.
12. What is mean by member or field of structure
13. What is structure?explain with suitable example
14. What is the advantage of UNION in C?
15. What is union? Explain with example.

CHAPTER 10 ALGORITHM

An algorithm, named for the ninth century Persian Mathematician al-Khowarizmi, is simply a set of rules used to perform some calculations, either by hand or more usually on a machine. In this subject we will refer algorithm as a method that can be used by the computer for solution of a problem. For instance performing the addition, multiplication, division or subtraction is an algorithm. Use of algorithm for some computations is a common practice. Even ancient Greek has used an algorithm which is popularly known as Euclid's algorithm for calculating the greatest common divisor of two numbers.

Basically algorithm is a finites set of instructions that can be used to perform certain task. In this chapter we will learn some basic concepts of algorithm. And for understanding how to write an algorithm we will discuss some examples.

ALGORITHM

An *Algorithm* is a set of rules for carrying out calculation either by hand or on a machine. An *Algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

$$\begin{tikzcd}
 & \text{Problem} \arrow[d] & \\
 & \text{Algorithm} \arrow[d] & \\
\text{Input} \arrow[r] & \text{Processor} \arrow[r] & \text{Output}
\end{tikzcd}$$

Every algorithm must satisfy the following criteria

Input	: Zero or more quantities are externally supplied.
Output	: Atleast one quantity is produced.
Definiteness	: Each instruction is clear and unambiguous.
Finiteness	: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminate after a finite number of steps.
Effectiveness	: Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil & paper.

CHARACTERISTICS OF ALGORITHM

- It is not depended on programming language, machine.
- Algorithm design is all about the mathematical theory behind the design of good programs.

ISSUES OR STUDY OF ALGORITHM:

- How to device or design an algorithm à creating and algorithm.
- How to express an algorithm à definiteness.
- How to analysis an algorithm à time and space complexity.
- How to validate an algorithm à fitness.
- Testing the algorithm à checking for error.

Algorithm Analysis Framework

Algorithm analysis framework involves finding out the time taken by a program and the memory space it requires to execute. It also determines how the input size of a program influences the running time of the program.

COMPLEXITY OF ALGORITHM

We can determine the efficiency of an algorithm by calculating its performance. The two factors that help us to determine the efficiency of an algorithm are:

- Amount of time required by an algorithm to execute
- Amount of space required by an algorithm to execute

These are generally known as time complexity and space complexity of an algorithm.

Time complexity

1. Time complexity of an algorithm is the amount of time required for it to execute.

2. The time taken by an algorithm is given as the sum of the compile time and the execution time. The compile time does not depend on the instance characteristics as a program once complied can be run many times without recompiling. So only the run time of the program matters while calculating time complexity and it is denoted by tp (instance characteristics).

3. It is difficult to calculate the time complexity in terms of physically clocked time.

For example in multi-user operating system, it depends on various factors such as:

- System load
- Number of programs running on the system
- Instruction set used
- Speed of the hardware

4. The time complexity of an algorithm is given in terms of frequency counts. Frequency count is the count that indicates the number of times the statement is executed.

5. The Time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion.

The time T(P) taken by a program P is the sum of the compute time and run time. The compile time does not depend on the instant characteristics. The run time is denoted by Tp

$\boxed{Tp(n)=Ca\ Add(n) + Cs\ Sub(n) + Cm\ Mul(n) + \ldots\ldots}$ where Ca, Cm ..... denotes the time needed for an addition, subtraction... and so on.

SPACE COMPLEXITY

Space complexity of an algorithm is the amount of storage required for it to execute.

The space required by an algorithm to execute is given as the sum of the following components:

1) A fixed part that is independent of the characteristics of inputs and outputs. This part includes instruction space (for example space for the code), space for simple variables and fixed size component variables and space for constants.

2) A variable part that consists of the space needed by the component variables whose size is dependent on a particular problem instance being solved and the space needed by the referenced variable.

Therefore to calculate the space complexity of an algorithm we have to consider two factors:

- Constant characteristic
- Instance characteristic

The following equation depicts the space requirement S(p) of an algorithm.

Where $\boxed{S(p) = C + Sp}$

- C is the constant part and indicates the space required for inputs and outputs which includes instructions, variables and identifiers.
- Sp defines the space required for an instance characteristic. This is a variable part whose space requirement depends on a particular problem.

Measuring input size

The time required to execute an algorithm depends on the input size of the algorithm. If the input size is longer, then the time taken to execute it is more. Therefore we can calculate the efficiency of an algorithm as a function to which the input size is passed as a parameter. Sometimes to implement an algorithm we need prior information of the input size.

Example when performing multiplication of two matrices we should know the order of these matrices, only then we can enter the elements of the matrices.

Measuring running time

The time measured for analyzing an algorithm is generally called as running time. For measuring the running time of an algorithm we consider the following:

1. First recognise the basic operation (the operation contributing the most to total run time) of an algorithm.

2. Identifying the basic operation of an algorithm is not that difficult. It is generally the most time consuming operation in the algorithm. Normally such operations are located in the inner loop of an algorithm.

For example, the basic operation of sorting algorithms is to compare the elements and place them in appropriate position.

The table gives an example of the concept of basic operation.

Basic Operations for Input Size

Problem statement	Input size	Basic operation
Computing GCD (greatest common divisor) of two numbers.	Two numbers	Division
Searching a key element from the list of n elements.	List of n elements.	Comparison of key elementwith every element of the list.
Performing matrix multiplication.	The two matrices with order n x n	Multiplication of the elements in the matrices.

Then calculate the total time taken by the basic operation using the following formula:

T(n) = Cop C(n)

Where T(n) is the running time of basic operation

Cop is the time taken by the basic operation to execute

C(n) is the number of time the operation needs to be executed Using this formula we can obtain the approximate computing time.

Best case, worst case and average case analysis

Best Case: If an algorithm takes the least amount of time to execute a specific set of input, then it is called best case time complexity.

Worst Case: If an algorithm takes maximum amount of time to execute a specific set of input, then it is called the worst case time complexity.

Average Case: If the time complexity of an algorithm for certain sets of inputs is on an average same then such a time complexity is called average case time complexity. The average case time complexity is not the just the average of best case and worst case time complexities.

Let us now consider the algorithm for sequential search and find its best, worst and average case time complexities.

Algorithm for sequential search

```
Algorithm Seq_search(H [0...n-1], key)
{
//Problem description: This algorithm searches the key elements for an
//array H [0...n-1] sequentially.
//Input: An array H [0...n-1] and search the key element
//Output: Returns the index of H where the key element is present
for p = 0 to n -1
{
 if (H [p] = key)
 {
 return p;
 }
}
}
```

Algorithm tracing for sequential search algorithm

1. //Let us consider n=4, H[ ] = {10, 14, 18, 20},
2. key = 14 for p = 0 to 4 -1 do // this loop iterates from p = 0 to 4-1
3. if {H [0] = key} then// the loop continues to iterate as H[0] is not the search element
4. return 1// finally the array retutns1 which is the position of the key element.

Best case time complexity

The above searching algorithm searches the element key from the list of n elements of the array H [0...n-1]. If the element key is present at the first location of the list (H [0...n-1]) then the time taken to execute the algorithm is the least .The time complexity depends on the number of times the basic operation is executed. Thus we get the best case time complexity when the number of basic operations is minimum. If the element to be searched is found at the first position, then the basic operation is only one and the best case complexity is achieved. The following equation denotes the best case time complexity as:

Cbest = 1

Worst case time complexity

In the above searching algorithm if the search element key is present at the nth position (the last position) of the list then the basic operations and time required to execute the algorithm is more and thus it gives the worst case time complexity. The following equation gives the worst case time complexity as:

Cworst = n

Average case time complexity

The Average complexity gives information about an algorithm on specific input. For instance in the sequential search algorithm:

Let the probability of getting successful search be P.

The total number of elements in the list is n.

If we find the first search element at the ith position then the probability of occurrence of the first search element is P/n for every ith element.

(1 - P) is the probability of getting unsuccessful search.

Therefore the average case time complexity Cavg(n) is given as:

Cavg(n) = Probability of successful search + Probability of unsuccessful search

$$Cavg(n) = \frac{P(1+n)}{2} + n(1 - P)$$

The above equation gives the general formula for computing average case time complexity.

If the search is complete and the search element is not found, then P = 0 which means that there is no successful search in such circumstances.

$Cavg(n) = 0(1 + n)/2 + n(1 + 0)$

$Cavg(n) = n$

From the above equation we can see that the average case time complexity Cavg(n) will be equal to n.

Thus calculating average case time complexity depends on the probability of getting a successful search and the number of operations required to perform the search. Therefore calculating average case time complexity is difficult compared to calculating worst case and best case complexities.

Methodologies for Analyzing Algorithms

In the previous section you have studied the analysis framework, time and space complexities, and tradeoff in algorithms. We also discussed measuring input size, running time of an algorithm, and best, worst and average case analysis. In this section we will study the methodologies used for analyzing the algorithms.

There are different methodologies for analyzing the algorithms. Following are some of the methodologies used for analyzing the algorithm:

- Pseudocode
- Random access machine model
- Counting primitive operations
- Analyzing recursive algorithms
- Testing and measuring over a range of instances

PSEUDOCODE

It is a compact and informal high level explanation of computer algorithms that uses the structural principles of a programming language. It is meant for human reading rather than machine reading.

It excludes some details from the algorithms which are not needed for human understanding such as variable declaration, system specific code and subroutines. The objective of using pseudocode is to make the programming language code easier for the human to understand.

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces {and}.
3. An identifier begins with a letter. The data types of variables are not explicitly declared.
4. Assignment of values to variables is done using the assignment statement.

<Variable>= <expression>;

5. There are two Boolean values TRUE and FALSE.

à Logical Operators AND, OR, NOT

àRelational Operators <, <=,>,>=, =, !=

6. The following looping statements are employed.

For, while and repeat-until

While Loop:

```
          While < condition >
          {
                    <statement-1>;
                              .
                              .
                              .
                    <statement-n>;
          }
```

For Loop:

```
          For variable= value-1 to value-2
{
          <statement-1>;
                    .
                    .
                    .
<statement-n>;
}
```

repeat-until:

```
            repeat
{
                  <statement-1>;
                        .
                        .
                        .
                  <statement-n>;
}
            until<condition>
```

7. A conditional statement has the following forms.

à If <condition> { <statement>;}

à If <condition> { <statement-1> ;}

Else {<statement-1>;}

Switch statement:

```
Switch<condition>
{
      Case <value-1>: <statement-1>;
                        .
                        .
                        .
      Case <value -n> : <statement-n>;
      Default : <statement-n+1>;
}
```

8. Input and output are done using the instructions read & write.

9. There is only one type of procedure:

Algorithm, the heading takes the form,

Algorithm Name (Parameter lists)

à As an example, the following algorithm fields & returns the maximum of 'n' given numbers:

```
1.  algorithm Max(A,n)
2.  // A is an array of size n
3.  {
```

```
4.    Result := A[1];
5.    for I:= 2 to n do
6.      if A[I] > Result then
7.          Result :=A[I];
8.     return Result;
9.    }
```

In this algorithm (named Max), A & n are procedure parameters. Result & I are Local variables.

Pseudocode for selection sort

Algorithm Selection sort (B)

```
{
for p = 1 to n-1
{
min x = B[p]
for j =m + 1 to n
{
If B[q] < min x
{
min q = q
min x = B[q]
B[min q] = B [p]
B[p]= min x
}
}
}
```

Counting primitive operations

While analyzing the algorithm we have to count the number of primitive operations that the algorithm executes. Examples of some of the primitive operations are as follows:

- Calling a method
- Returning a value from a function
- Performing some arithmetic operations like addition, subtraction etc.
- Assigning a value to a variable

- Comparing two variables
- Reference to the pointer
- Indexing an array

Counting primitive operations describes how to count the maximum number of primitive operations an algorithm executes.

Let us now consider the arrayMax algorithm which helps in finding the maximum number of primitive operations.

Algorithm factorial(x)

```
//Input x
//F is the factorial of the given number x Fact(x)
{
If x = 1
Return 1
Else
F= x*fact(x-1)// recursive function which stops when value of x =1
}
```

Let us now trace the factorial algorithm

Algorithm tracing for factorial algorithm

```
Let x = 5
Fact(5)
{
If x = 1// x is not equal to 1 so the else part gets executed
Return 1
Else
F= 5*fact(5-1)// recursive function which stops when value of x =1
// finally the algorithm returns 120 as the factorial of the number 5
}
```

Analysis of factorial algorithm

For analyzing this algorithm and computing the primitive operation we have to consider some operations that are repeated as they are enclosed in the body of the loop.

First getting the value for x corresponds to one primitive operation.

This step executes only once at the beginning of the algorithm and contributes one unit to the total count.

In the function Fact(x), there are two primitive operations.

- To check if the value of x is 1
- To multiply x and Fact(x-1)

The function recurs these two operations x number of times. Therefore it contributes a 2x value to the total count.

Hence the total number of primitive operations T(n) that the factorial algorithm executes is given in the following equation.

T(n) = 1+2x

Asymptotic Notations and Basic Efficiency Classes

To choose the best algorithm we need to check the efficiency of each algorithm. Asymptotic notations describe different rate-of-growth relations between the defining function and the defined set of functions. The order of growth is not restricted by the asymptotic notations, and can also be expressed by basic efficiency classes having certain characteristics.

Let us now discuss asymptotic notations of algorithms

ASYMPTOTIC NOTATIONS

Asymptotic notation within the limit deals with the behavior of a function, i.e. for sufficiently large values of its parameter. While analyzing the run time of an algorithm, it is simpler for us to get an approximate formula for the run- time.

The main characteristic of this approach is that we can neglect constant factors and give importance to the terms that are present in the expression (for T(n)) dominating the functions behavior whenever n becomes large. This allows dividing of un-time functions into broad efficiency classes.

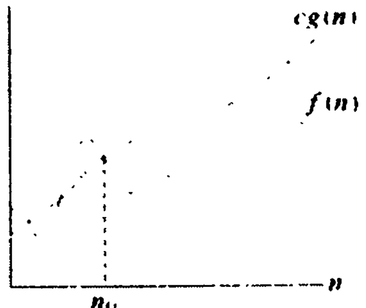
To give time complexity as "fastest possible", "slowest possible" or "average time", asymptotic notations are used in algorithms. Various notations such as &! (omega), T (theta), O (big o) are known as asymptotic notations.

Big Oh notation (O)

"O is the representation for big oh notation. It is the method of denoting the upper bound of the running time of an algorithm. Big Oh notation helps in calculating the longest amount of time taken for the completion of algorithm.

O(g)={f/f is a non negative function ,there exists constants c2 & n0 such that f(x) d" :2.g(n), ne"n0}

The graph of C h(n) and T(n) can be seen in the figure,As n becomes larger, the running time increases considerably. For example, consider T(n)=$13n^3$+$42n^2$+2nlogn+4n. Here as the value on n increases n^3 is much larger than n^2, nlogn and n. Hence it dominates the function T(n) and we can consider the running time to grow by the order of n3. Therefore it can be written as T(n)=O(n^3). The value of n for T(n) and C h(n) will not be less than n0.Therefore values less than n0 are considered as not relevant.



f(n) = O(g(n))

Big Oh Notation T(n) • O(h(n))

Example:

1. f(n)= $10n^3+5n^2+17$

    $10n^3$d" f(x)

    $10n^3$d" f(x)

    C1=10

    C1.n^3d" f(x) for all n e" 1 =n0

    F belongs to the class n^3

2. f(n)= $2n^3+3n+79$

    $2n^3$d" f(x)

    $2n^3$d" f(x)

    C1=2 & c2=84

    C1.n^3d" f(x) for all n e" 1 =n0

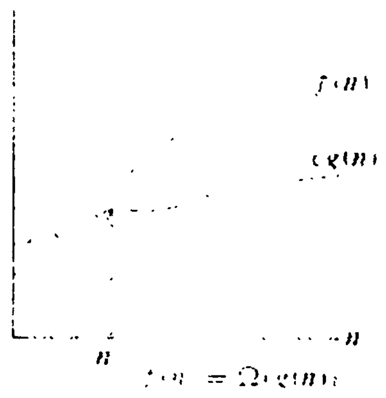
    F belongs to the class n^3

Omega notation (Ω)

"Ω" is the representation for omega notation. Omega notation represents the lower bound of the running time of an algorithm. This notation denotes the shortest amount of time that an algorithm takes.

Ω(g)={f / f is a non negative function ,there exists constants c1 & n0 such that c1.g(n) d" f(x), ne"n0}

The gı.ph of C h(n) and T(n) can be seen in the figure.



Example:

1. f(n)= $10n^3+5n^2+17$

   f(x) d" (10+5+7) n^3

   f(x) d" (32) n^3

   c2=32

   f(x) d" c2.n^3 for all n e" 1 =n0

   F belongs to the class n^3

2. f(n)= $2n^3+3n+79$

   f(x) d" (2+3+79) n^3

   f(x) d" (84) n^3

   c2=84

   f(x) d" c2.n^3 for all n e" 1 =n0

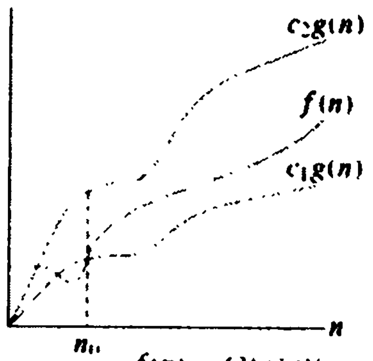
   F belongs to the class n^3

Theta notation (r·)

The depiction for theta notation is "T. This notation depicts the running time between the upper bound and lower bound.

O(g)={f / f is a non negative function ,there exists constants c1,c2 & n0 such that c1.g(n) d" f(x) d" c2.g(n), ne"n0}

The graph of C1 h(n), C2 h(n) and T(n) can be seen in the figure.



$f(n) = \Theta(g(n))$

Example:

1. f(n)= $10n^3+5n^2+17$

   $10n^3$d" f(x) d" (10+5+7) n^3

   $10n^3$d" f(x) d" (32) n^3

   C1=10 & c2=32

   C1.n^3d" f(x) d" c2.n^3 for all n e" 1 =n0

   F belongs to the class n^3

2. f(n)= $2n^3$+3n+79

   $2n^3$d” f(x) d” (2+3+79) n^3

   $2n^3$d” f(x) d” (84) n^3

   C1=2 & c2=84

   C1.n^3d” f(x) d” c2.n^3 for all n e” 1 =n0

   F belongs to the class n^3

3. a(n) = “$a_i n^i$,i=0,1,...,k a_k>o

   a belongs to the class n^k

4. x(n) =5nlogn + n

   x belongs to the class nlogn

5. x(n) = 2 + 1/n

   x belongs to the class 1

Practical Complexities

We have previously analyzed that we can obtain a different order of growth by means of constant multiple (C in C*f(n)). We used different types of notations for these orders of growth. But we do not restrict the classification of order of growth to &!, T and O.

There are various efficiency classes and each class possesses certain characteristic which is shown in table

Growth Order	Name of the Efficiency Class	Explanation	Example
1	Constant	Specifies that algorithm?s running time is not changed with increase in size of the input	Scanning the elements of array
log n	Logarithmic	For each iteration of algorithm a constant factor shortens the problem’s size	Performing the operations of binary search
n	Linear	Algorithms that examine the list of size n.	Performing the operations of sequential search
n logn	n-log-n	Divide and conquer algorithms.	Using merge sort or quick sort elements are sorted
n2	Quadratic	Algorithms with two embedded loops	Scanning the elements of matrix
n3	Cubic	Algorithms with three embedded loops.	Executing matrix Multiplication
2n	Exponential	Algorithms that generate all the subsets which are present in n – element sets	Generating all the subsets of n different elements
n!	Factorial	Algorithms that generate all the permutations of an n-element set	All the permutationsare generated

To get a feel for how the various functions grow with n, you are advised to study the following table and the figure very closely.

Logn	N	Nlogn	N2	N3	2n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	250	4,096	35,536
5	32	160	1,024	32,768	4,294,967,296

It is evident from the table and figure that the function 2^n grows rapidly with n In fact, if an algorithm needs 2^n steps for execution, then when n=40 , the number of steps needed is approximately 1.1. * 10^{12}. On a computer performing one billion steps per second, this would require about 18.3 minutes. If N=50, the same algorithm would run for about 13 days on this computer.when n =60 , abour 310.56 years are required to execute the algorithm and when n =100 about 4* 10^{13} years are needed. So, we may concluded that the utility of algorithms with exponential complexity is limited to small n(typically nd"40).

Algorithms that have a complexity that is a polynomial of high degree are also of limited utility.For example, if an algorithm needs n^{10} steps, then using out 1-billion-step-per-second computer, we need 10 seconds when n=10,3171 years when n=100 and 3.17 * 10^{13} years when n=1000. If the algorithm's complexity had been n^3 steps instead, then we would need one second when n=1000, 110.67 minutes when n=10,000 and 11.57 days when n=100,000.

The following table gives the time needed by a one-billion-steps-per-second computer to execute an algorithm of complexity f(n) instructions. You should note that currently only the fastest computers can execute about 2 billion instructions per second. From a practical standpoint, it is evident that for reasonably large(n>100), only algorithms of small complexity(sucn as n, nlogn,n^2 and n^3) are feasible.

Time for f(n) instructions on a 109 instr/sec computer						
N	F(n)=n	F(n)=nlogn	F(n)=n2	F(n)=n3	F(n)=n10	F(n)=2n
10	.01 μs	.03 μs	.1 μs	1 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	2.84 hr	1 ms
30	.03 μs	.15 μs	.9 μs	27 μs	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	121.36 d	18.3 m
50	.05 μs	.28 μs	2.5 μs	125 μs	3.1 yr	13 d
100	.1 μs	.66 μs	10 μs	1 ms	3171 yr	4.1013 yr
1000	1 μs	9.96 μs	1 ms	1 s	3.17 *1013 yr	31*10283 yr

Example

Let us analyze some recursive algorithm mathematically.

1. Factorial of a given number

The factorial of some number can be obtained by performing repeated multiplication.

In n=5

1) N!=5!
2) 4!*5
3) 3!*4*5
4) 2!*3*4*5
5) 1!*2*3*4*5
6) 0!*1*2*3*4*5
7) 1*1*2*3*4*5 [0!=1]

The above mentioned steps can be written in pseudocode form as

Algorithm Factorial(n)

```
{
if(n=0)
{
return 1;
}
else
{
return(Factorial((n-1)*n));
}
```

Program - fact.c

```
#include<stdio.hh>
#include<conio.h>
void main()
{
        int n,fact;
        int rec(int);
        clrscr();
        printf("Enter the number:->");
        scanf("%d",&n);
        fact=rec(n);
printf("Factorial Result are: %d", fact);
        getch();
}
```

```
rec(int x)
{
        int f;
        if(x==1)
                return(x);
        else
        {
                f=x*rec(x-1);
                return(f);
        }
}
```

MATHEMATICAL ANALYSIS

1) The factorial algorithm works for input size n.
2) The basic operation in computing factorial is multiplication.
3) The recursive function call can be formulated as

$F(n) = F(n-1) * n$ where $n>0$

Then the basic operation multiplication is given as $M(n)$. And $M(n)$ is multiplicat ion count to compute factorial (n).

$M(n)=M(n-1) + 1$

4) In step 3 the recurrence relation is obtained.

$M(n)=M(n-1) + 1$

Now we will solve recurrence using

a. Forward substitution

$M(1) = M(0)+1 \quad =1$

$M(2) = M(1)+1=1+1 \quad =2$

$M(3) = M(2)+1=2+1 = 3$

b. Backward Substitution

$M(n) \quad =M(n-1) + 1$

$\quad =[M(n-2)+1]+1=M(n-2)+2$

$\quad =[M(n-3)+1]+1=M(n-3)+3$

From the substitution methods we can establish a general formula as:

$M(n) =M(n-i) +i$

Now let us prove correctness of this formula usin ɜ mathematical induction as follows.

Prove $M(n) =n$ by using mathematical induction

Basis: Let n=0 the

$M(n) =0$

i.e. $M(0) =0=n$

Iduction : If we assume M(n-1) =n-1 then

M(n) =M(n-1)+1

=n-1+1

=n

i.e M(n)=n

Thus the time complexity of factorial is r•(n).

2. TOWERS OF HANOI

The problem "Towers of Hanoi" is a classic example of recursive function. The problem can be understood by following discussion. The initial setup is

There are 3 pegs named as A, B and C. The five disks of different diameters are placed on peg A. The arrangement of the disks is such that every smaller disk is placed on the larger disk.

The problem of "Tower Hanoi" states that move the five disks from peg A to peg C using peg B as an auxiliary.

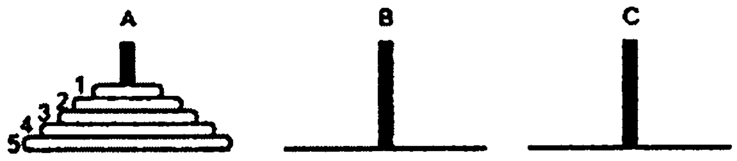
The conditions are

i) Only the top disk on any peg may be moved to any other peg.

ii) A larger disk should never rest on the smaller one.

The above problem is the classic example of recursion. The solution to this problem is very simple.

First of all let us number out the disks for our comfort.



The solution can be stated as

1. Move top n-1 disks from A to B using C as auxiliary.
2. Move the remaining disk from A to C.
3. Move the n-1 disks from B to C using A as auxiliary.

We can convert it to

Move disk 1 from A to B.

Move disk 2 from A to C.

Move disk 1 from B to C.

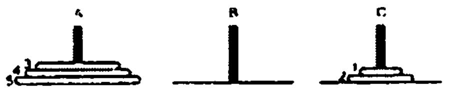
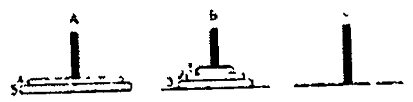
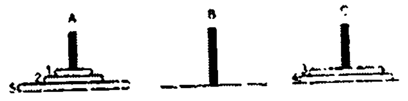


Fig. 1.12

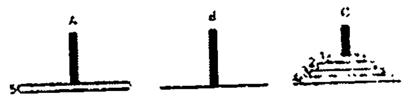
move disk 3 from A to B.
move disk 1 from C to A
move disk 2 from C to B.
move disk 1 from A to B.



move disk 4 from A to C.
move disk 1 from B to C
move disk 2 from B to A
move disk 1 from C to A
move disk 3 from B to C



move disk 1 from A to B
move disk 2 from A to C
move disk 1 from B to C.



Thus actually we have moved n-1 disks from peg A to C.

Algorithm TOH (n, A, C, B)

```
{
if (n=1)
{
write ("The peg moved from A to C");
return;
}
else
{
//move top n-1 disks from A to B using C
TOH (n-1,A,B,C);
//move remaining disk from B to C using A
TOH (n-1,B,C,A);
}
}
```

Mathematical analysis:

1. The input size is n. i.e. total number of disks.
2. The basic operation in the problem is moving disks from one peg to another. when n>1, then to move these disks from peg A to peg C using peg B, we first move recursively n-1 disks from peg A to peg B using auxiliary peg C .Then we move the largest disk directly from peg A to peg C and finally n-1 disks from peg B to peg C (using peg A auxiliary peg).

If n=1 then we simply move the disk from peg A to peg C.

3. The moves of disks are denoted by M(n). M(n) depends on number if disks n. The recurrence can then set up as.

$M(1)=1$ since Only I move is needed TOH(1,*,*,*)

If $n>1$ then we need two recursive calls plus one move. Hence

$M(n)=M(n-1)+1+M(n-1)$

$M(n)=2M(n-1)+1$

4. Solving the recurrence $M(n)=2M(n-1)+1$ using two substitution methods.

- Forward substitution

For $n>1$

$M(2) =2M(1)+1$
$=2+1$
$=3$

$M(3) =2M(2)+1$
$=2(3)+1$
$=7$

$M(4) =2M(3)+1$
$=2(7)+1$
$=15$

- Backward Substitution

$M(n)=2M(n-1)+1$
$=2[2M(n-2)+1]+1$
$=4M(n-2)+3$
$=4[2M(n-3)+1]+3$
$=8M(n-3)+7$

Above computation suggest us to compute next computation as

$M(n)=2^4M(n-4)+2^3+2^2+2+1$

From this we can establish a general formula as

$M(n)=2^iM(n-i)+2^{i-1}+2^{i-2}+\ldots..+2+1$

This can also be written as

$M(n)=2^iM(n-i)+2^i-1$

Thus for obtaining $M(n)$ we substitute n by n-1 in the equation

Let us mathematical induction to establish correctness of equation.

- **Basis:**

As in equation (2) we can obtain $M(n)$ by substituting $n=n-1$, assume initially $n=1$ then $n-i=1$ i.e. $i=n-1$

i.e. $M(n) = 2^iM(n-i)+2^i-1$ with $i=n-1$ can become

$=2^{n-1}M(n-n+1)+2^{n-1}-1$

$=2^{n-1}M(1)+2^{n-1}-1$

$=2^{n-1}+2^{n-1}-1$

$=2^{n}-1$

$M(n)=2^{n}-1$ Now if n=1 then

$M(n)=2-1=1$ is proved

- **Induction:**

From the equation

$M(n)=2M(n-1)+1$

But in basis of induction we have computed $M(n-1)=2^{n-1}-1$ then substitute this value in equation(3) and we will get

$M(n)=2(2^{n-1}-1)+1$

$= 2^{n}-2+1$

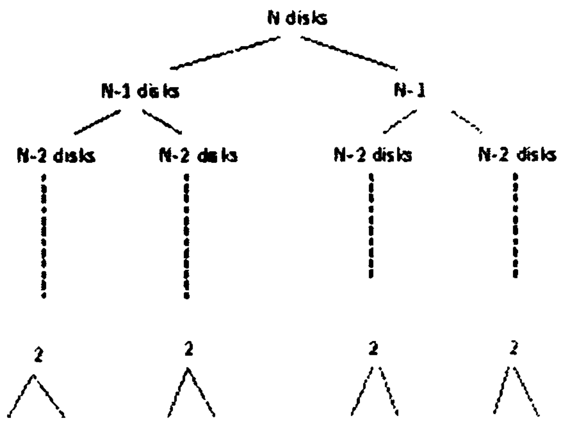
$M(n)=2^{n}-1$ is proved for n=n-1

"we get recurrence as

$M(n)=2^{n}-1$

From this we can conclude that Tower of Hanoi has a time complexity as $r\cdot(2^{n}-1)=r\cdot(2^{n})$

The tree can be drawn for showing the recursive calls made in the problem of Tower of Hanoi.



Summary

- Algorithm is nothing but a collection of unambiguous instructions occurring in some specific manner.
- The algorithm should possess different properties such as non ambiguous finite range of input, multiplicity, speed, finiteness.
- The algorithm should be written systematically. Certain notations are used while writing the algorithm.
- The greatest common divisor can be calculated using three methods: Euclid's algorithm, consecutive integer checking method and by finding repetitive factor.

- While developing an algorithm first the problem needs to be understood solve during exact problem solving or approximate problem solving algorithm. which data structure is to be used, and which algorithmic strategy will be most suitable to solve the problem efficiently.
- Then the algorithm can be designed by using natural language or pseudocode or a flow chart can be designed algorithm is verified with the help of some input set. Further the algorithm can be analyzed using time complexity and space complexity. The implementation of such algorithm can then be done with some suitable programming language.
- The computation problems can be classified as – Sorting, Searching, numerical problems, geometric problems, combinatorial problems, graph problems. String processing problems.
- The analysis framework is based on certain factors such as-

  The asymptotic notation is a shorthand way to represent the time complexity. Various notations such as &!, r• and O used are called asymptotic notions.
- Basic efficiency classes are – constant, logarithmic, linear, nlogn, quadratic, cubic, exponential and factorial.

Review Questions

1. *What is an Algorithm?*
2. *Define Pseudo code.*
3. *Define Flowchart.*
4. *Explain Algorithm's Correctness*
5. *What is Efficiency of algorithm?*
6. *What is generality of an algorithm?*
7. *What is algorithm's Optimality?*
8. *What do you mean by 3Worst case-Efficiency" of an algorithm?*
9. *What do you mean by 3Best case-Efficiency" of an algorithm?*
10. *Define the3 Average-case efficiency" of an algorithm?*
11. *How to measure the algorithm's efficiency?*
12. *What is called the basic operation of an algorithm?*
13. *Define Big oh notation*
14. *Define &! notation*
15. *Define È ▯ notation*
16. *What is the use of Asymptotic Notations?*
17. What is space complexity?
18. What is time complexity?

19. *Describe the steps in analyzing & coding an algorithm.*

20. *Discuss the fundamentals of analysis framework .*

21. *Explain the various asymptotic notations used in algorithm design.*

22. *Explain the general framework for analyzing the efficiency of algorithm.*

23. *Explain the basic efficiency classes.*

24. *Describe briefly the notions of complexity of an algorithm.*

25. *What is Pseudo-code? Explain with an example.*

26. Write Master's method for solving recurrence relations.

27. Discuss the basic steps in the complete development of an algorithm.

28. Write about randomized algorithm.

29. What is an algorithm? What are the criteria it should satisfy?

30. Develop the algorithm for towers of Hanoi without Recursive function.

31. Develop an algorithm for matrix addition and how much space and time is required?

32. Develop an algorithm for sum of the n integers and how much space and time is required?

33. Explain the asymptotic notations used in algorithm analysis.

Problems

1) What is big "oh" notation? Show that if $f(n) = a^m n^m + \ldots.. + a^1 n + a^0$ then $f(n) = O(n^m)$.

2) Show that $f(n) = 6{*}2^n + n^2 + 4n + 88 = O(2^n)$.

3) Solve the recurrence relation with intial condition as $T(1)=1$.Also find big Oh notation.

$T(n)=2T(n/2)+n$

4) Solve the recurrence realtion

$T(n) = kT(n/k)+n^2$ when $T(1)=1$, and k is any cosntant

5) Find the complexity of the following recurrence relation.

$T(n) = 9T(n/3)+n$

CHAPTER 11 SORTING ALGORITHMS

In computer science and mathematics, a **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output.

The general sorting problem is simple enough to describe: Given an Initially unordered array of N records, with one field distinguished as the Key, rearrange the records so they are sorted into increasing (or Decreasing) order, according to each record's key.

Example:

Sorting an array refers to putting data stored in array in a specified order:
ascending (lowest to highest) or descending order (highest to lowest).

Consider the following array elements
float weights [5] = {4.3, 2.0, 8.2, 23.5, 1.7};

```
Array elements sorted in ascending order are:   1.7  2.0  4.3  8.2  23.5
Array elements sorted in descending order are: 23.5  8.2  4.3  2.0   1.7
```

Uses:

Sorting algorithms are used in all kinds of applications and are necessary,for instance, if we plan to use efficient searching algorithms like Binary

Search or Interpolation Search, since these require their data to be sorted

Various Sorting Techniques: There are different techniques for putting array data in a given order.

1. Bubble sort
2. Bucket sort
3. Comparison sort
4. Heapsort
5. Insertion sort
6. Merge sort
7. Quick sort
8. Radix sort
9. Selection sort
10. Shell sort

BUBBLE SORT

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 means the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top (i.e. the beginning) of the list via the swaps. Because it only uses comparisons to operate on elements, it is a comparison sort. This is the easiest comparison sort to implement.

Bubble sort is a straightforward and simplistic method of sorting data that is used in computer science education. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. Although simple, this algorithm is highly inefficient and is rarely used except in education. A slightly better variant, cocktail sort, works by inverting the ordering criteria and the pass direction on alternating passes.

A C-language program bubble sort.

```
#include <stdio.h>
void main()
{
    int x[10];
    int i,j;
    printf("\nEnter 10 elements\n");

    /* accepting 10 elements from the user for sorting */
    for(i = 0; i<10 ; i++)
        scanf("%d",&x[i]);
    /* End of input */
    /* bubble sort */
    for(i=0;i<10;i++)
    {
        for(j=0;j<10-i;j++)
        {
            /*Condition to handle i=0 & j = 9. j+1 tries to access x[10] which is not there in zero based
array*/
            if(j+1 == 10)
              continue;
            if(x[j]>x[j+1])
            {
                temp = x[j];
                x[j]=x[j+1];
                x[j+1] = temp;
            }
        }
    }
    /* printing sorted array */
    for(i=0; i<10;i++)
        printf("\n %d",x[i]);
    /*end of program */
}
```

The bubble sort works by iterating down an array to be sorted from the first element to the last, comparing each pair of elements and switching their positions if necessary. This process is repeated as many times as necessary, until the array is sorted. Since the worst case scenario is that the array is in reverse order, and that the first element in sorted array is the last element in the starting array, the most exchanges that will be necessary is equal to the length of the array.

Here is a simple example:

Given an array 23154 a bubble sort would lead to the following sequence of partially sorted arrays: 21354, 21345, 12345. First the 1 and 3 would be compared and switched, then the 4 and 5. On the next pass, the 1 and 2 would switch, and the array would be in order.

We compute that the order of the outer loop (for(int x = 0; ..)) is O(n); then, we compute that the order of the inner loop is roughly O(n). Note that even though its efficiency varies based on the value of x, the average efficiency is n/2, and we ignore the constant, so it's O(n). After multiplying together the order of the outer and the inner loop, we have O(n^2).

SELECTION SORT

Selection sort is a simple sorting algorithm that improves on the performance of bubble sort. It works by first finding the smallest element using a linear scan and swapping it into the first position in the list, then finding the second smallest element by scanning the remaining elements, and so on. Selection sort is unique compared to almost any other algorithm in that its running time is not affected by the prior ordering of the list, it performs the same number of operations because of its simple structure.

Selection sort also requires only *n* swaps, and hence just È(*n*) memory writes, which is optimal for any sorting algorithm. Thus it can be very attractive if writes are the most expensive operation, but otherwise selection sort will usually be outperformed by insertion sort or the more complicated algorithms.

```
#include <stdlib.h>
#include <stdio.h>
#define uint32 unsigned int
typedef int (*CMPFUN)(int, int);
void ArraySort(int This[], CMPFUN fun_ptr, uint32 the_len)
```

```
{
 /* selection sort */

 uint32 indx;
 uint32 indx2;
 uint32 large_pos;
 int temp;
 int large;

 if (the_len <= 1)
  return;

 for (indx = the_len - 1; indx > 0; --indx)
 {
  /* find the largest number, then put it at the end of the array */
   large = This[0];
   large_pos = 0;

  for (indx2 = 1; indx2 <= indx; ++indx2)
   {
    temp = This[indx2];
    if ((*fun_ptr)(temp ,large) > 0)
    {
      large = temp;
      large_pos = indx2;
    }
   }
  This[large_pos] = This[indx];
  This[indx] = large;
 }
}

#define ARRAY_SIZE 14
int my_array[ARRAY_SIZE];
```

```
void fill_array()
{
 int indx;

 for (indx=0; indx < ARRAY_SIZE; ++indx)
 {
  my_array[indx] = rand();
 }
 /* my_array[ARRAY_SIZE - 1] = ARR[illegible]
}

int cmpfun(int a, int b)
{
  if (a > b)
   return 1;
  else if (a < b)
   return -1;
  else
   return 0;
}

int main()
{
 int indx;
 int indx2;

 for (indx2 = 0; indx2 < 80000; ++indx2)
 {
   fill_array();
  ArraySort(my_array, cmpfun, ARRAY_SIZE);
  for (indx=1; indx < ARRAY_SIZE; ++indx)
  {
   if (my_array[indx - 1] > my_array[indx])
    {
      printf("bad sort\n");
      return(1);
     }
    }
  }

  return(0);
}
```

INSERTION SORT

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly-sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place. Shell sort (see below) is a variant of insertion sort that is more efficient for larger lists. This method is much more efficient than the bubble sort, though it has more constraints.

```
#include <stdlib.h>
#include <stdio.h>
#define uint32 unsigned int
typedef int (*CMPFUN)(int, int);
void ArraySort(int This[], CMPFUN fun_ptr, uint32 the_len)
{
 /* insertion sort */
 uint32 indx;
 int cur_val;
 int prev_val;
 if (the_len <= 1)
  return;
 prev_val = This[0];
 for (indx = 1; indx < the_len; ++indx)
 {
  cur_val = This[indx];
  if ((*fun_ptr)(prev_val, cur_val) > 0)
  {
   /* out of order: array[indx-1] > array[indx] */
   uint32 indx2;
   This[indx] = prev_val; /* move up the larger item first */

   /* find the insertion point for the smaller item */
   for (indx2 = indx - 1; indx2 > 0;)
   {
    int temp_val = This[indx2 - 1];
    if ((*fun_ptr)(temp_val, cur_val) > 0)
    {
     This[indx2—] = temp_val;
     /* still out of order, move up 1 slot to make room */
    }
    else
     break;
   }
   This[indx2] = cur_val; /* insert the smaller item right here */
  }
  else
  {
   /* in order, advance to next element */
   prev_val = cur_val;
  }
 }
}
#define ARRAY_SIZE 14
int my_array[ARRAY_SIZE];
uint32 fill_array()
{
 int indx;
 uint32 checksum = 0;
 for (indx=0; indx < ARRAY_SIZE; ++indx)
 {
   checksum += my_array[indx] = rand();
  }
 return checksum;
}
int cmpfun(int a, int b)
{
 if (a > b)
  return 1;
 else if (a < b)
  return -1;
 else
  return 0;
}

int main()
{
 int indx;
 int indx2;
 uint32 checksum1;
 uint32 checksum2;
 for (indx2 = 0; indx2 < 80000; ++indx2)
 {
  checksum1 = fill_array();
  ArraySort(my_array, cmpfun, ARRAY_SIZE);
  for (indx=1; indx < ARRAY_SIZE; ++indx)
  {
   if (my_array[indx - 1] > my_array[indx])
   {
    printf("bad sort\n");
    return(1);
   }
  }
  checksum2 = 0;
 for (indx=0; indx < ARRAY_SIZE; ++indx)
```

```
  {
   checksum2 += my_array[indx];
  }
 if (checksum1 != checksum2)
  {
   printf("bad checksum %d %d\n", checksum1, checksum2);
  }
 }
 return(0);
}
```

MERGE SORT

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (*i.e.* 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists.

```
#include <stdlib.h>
#include <stdio.h>
#define uint32 unsigned int
 typedef int (*CMPFUN)(int, int);
#define INSERTION_SORT_BOUND 8 /* boundary point to use insertion sort */
 void ArraySort(int This[], CMPFUN fun_ptr, uint32 the_len)
{
  uint32 span;
  uint32 lb;
  uint32 ub;
  uint32 indx;
  uint32 indx2;
  if (the_len <= 1)
    return;
  span = INSERTION_SORT_BOUND;
  /* insertion sort the first pass */
  {
    int prev_val;
    int cur_val;
```

```
   int temp_val;
   for (lb = 0; lb < the_len; lb += span)
   {
    if ((ub = lb + span) > the_len) ub = the_len;
    prev_val = This[lb];
    for (indx = lb + 1; indx < ub; ++indx)
    {
     cur_val = This[indx];
     if ((*fun_ptr)(prev_val, cur_val) > 0)
     {
      /* out of order: array[indx-1] > array[indx] */
      This[indx] = prev_val; /* move up the larger item first */
      /* find the insertion point for the smaller item */
      for (indx2 = indx - 1; indx2 > lb;)
      {
       temp_val = This[indx2 - 1];
       if ((*fun_ptr)(temp_val, cur_val) > 0)
       {
        This[indx2—] = temp_val;
        /* still out of order, move up 1 slot to make room */
       }
       else
        break;
      }
      This[indx2] = cur_val; /* insert the smaller item right here */
     }
     else
     {
     /* in order, advance to next element */
      prev_val = cur_val;
     }
    }
   }
  }
```

```
/* second pass merge sort */
{
 uint32 median;
 int* aux;

 aux = (int*) malloc(sizeof(int) * the_len / 2);

 while (span < the_len)
  {
   /* median is the start of second file */
   for (median = span; median < the_len;)
    {
     indx2 = median - 1;
     if ((*fun_ptr)(This[indx2], This[median]) > 0)
      {
       /* the two files are not yet sorted */
       if ((ub = median + span) > the_len)
        {
         ub = the_len;
        }
       /* skip over the already sorted largest elements */
       while ((*fun_ptr)(This[—ub], This[indx2]) >= 0)
        {
        }
       /* copy second file into buffer */
       for (indx = 0; indx2 < ub; ++indx)
        {
         *(aux + indx) = This[++indx2];
        }
       —indx;
       indx2 = median - 1;
       lb = median - span;
       /* merge two files into one */
        for (;;)
{
if ((*fun_ptr)(*(aux + indx), This[indx2]) >= 0)
 {
  This[ub—] = *(aux + indx);
  if (indx > 0) —indx;
   else
   {
    /* second file exhausted */
     for (;;)
     {
      This[ub--] = This[indx2];
      if (indx2 > lb) —indx2;
      else goto mydone; /* done */
     }
    }
  }
  else
  {
   This[ub—] = This[indx2];
   if (indx2 > lb) —indx2;
    else
    {
     /* first file exhausted */
      for (;;)
      {
       This[ub—] = *(aux + indx);
       if (indx > 0) —indx;
       else goto mydone; /* done */
      }
     }
   }
 }
}
mydone:
      median += span + span;
     }
    span += span;
   }

  free(aux);
 }
}
#define ARRAY_SIZE 250000
int my_array[ARRAY_SIZE];
uint32 fill_array()
{
 int indx;
 uint32 sum = 0;

 for (indx=0; indx < ARRAY_SIZE; ++indx)
 {
  sum += my_array[indx] = rand();
 }
 return sum;
}

int cmpfun(int a, int b)
{
 if (a > b)
  return 1;
 else if (a < b)
  return -1;
 else
  return 0;
}
int main()
{
 int indx;
```

```
uint32 checksum, checksum2;
checksum = fill_array();
 ArraySort(my_array, cmpfun, ARRAY_SIZE);
checksum2 = my_array[0];
 for (indx=1; indx < ARRAY_SIZE; ++indx)
 {
  checksum2 += my_array[indx];
  if (my_array[indx - 1] > my_array[indx])
   {
    printf("bad sort\n");
    return(1);
   }
 }

 if (checksum != checksum2)
 {
  printf("bad checksum %d %d\n", checksum, checksum2);
  return(1);
 }
 return(0);
}
```

Comparison of 4 sorting techniques

Name	Best	Average	Worst	Memory	Stable	Method
Bubble sort	O(n)	—	O(n2)	O(1)	Yes	Exchanging
Selection sort	O(n2)	O(n2)	O(n2)	O(1)	No	Selection
Insertion sort	O(n)	O(n + d)	O(n2)	O(1)	Yes	Insertion
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	Merging

Review Questions

1. Explain the procedure of Bubble Sort
2. Explain the procedure of Insertion Sort
3. Describe Merge Sort with Example
4. Describe DFS Method with Example
5. Describe the Dijikstra's Algorithm with example
6. Draw (a) direct Graph (b) Connected Graph
7. Write an Algorithm for Selection sort
8. Compare all 4 sorting algorithm
9. List out the famous Sorting Algorithms
10. What do you mean by "Sorting" problem?

CHAPTER 12 SEARCHING ALGORITHMS

Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion. Thus the efficient storage of data to facilitate fast searching is an important issue.

SEARCHING ALGORITHM

Searching algorithms are closely related to the concept of dictionaries. Dictionaries are data structures that support search, insert, and delete operations. The Searching Algorithm includes:

(i) Sequential Search

(ii) Binary Search

Algorithm for sequential search

Algorithm Seq_search(H [0...n-1], key)

```
{
//Problem description: This algorithm searches the key elements for an
//array H [0...n-1] sequentially.
//Input: An array H [0...n-1] and search the key element
//Output: Returns the index of H where the key element is present
for p = 0 to n -1
{
 if (H [p] = key)
{
 return p;
}
}
}
```

Algorithm tracing for sequential search algorithm

```
1.    //Let us consider n=4, H[ ] = {10, 14, 18, 20},
2.    key = 14 for p = 0 to 4 -1 do // this loop iterates from p = 0 to 4-1
3.    if {H [0] = key} then// the loop continues to iterate as H[0] is not the search element
4.    return 1// finally the array retutns1 which is the position of the key element.
```

BINARY SEARCH

Binary search is an efficient searching method. While searching the elements using this method the most essential thing is that the elements in the array should be sorted one.

Binary search algorithm is a technique for finding a particular value in a ~orted 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 largest value(key), 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 approach iteratively, it narrows the search by a factor of two each time, and finds the target value.

Algorithm Bin search(a,n,x)

```
// Given an array a[1:n] of elements in non-decreasing
//order, n>=0,determine whether 'x' is present and
// if so, return 'j' such that x=a[j]; else return 0.
{
low:=1; high:=n;
while (low<=high) do
{
    mid:=[(low+high)/2];
    if (x<a[mid]) then high;
    else if(x>a[mid]) then
low=mid+1;
  else return mid;
 }
 return 0;
}
```

Example

Let us select the 14 entries.

-15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

à Place them in a[1:14], and simulate the steps Binsearch goes through as it searches for different values of 'x'.

à Only the variables, low, high & mid need to be traced as we simulate the algorithm.

à We try the following values for x: 151, -14 and 9.

for 2 successful searches & 1 unsuccessful search.

Table: Shows the traces of Bin search on these 3 steps.

X=151	low	high	mid
	1	14	7
	8	14	11
	12	14	13
	14	14	14

Found

x=-14

	low	high	mid
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	**Not found**

x=9

	low	high	mid
	1	14	7
	1	6	3
	4	6	5

Found

Analysis

The time complexity for Binary search problem may be written as:

$T(n) = K \qquad n=1$

$\quad =T(n/2) +K, \qquad n>1$

$T(n) = T(n/2) +K,$

$\quad = T(n/2^2) +2K,$

$\quad = T(n/2^3) +3K,$

....

$=T(1) + rk$, where $2^r =n$, r is a positive integer.

$=(r+1)K$

$=(1+\log_2 n)k= O(\log n)$

Time Complexity O(logn)

Time Complexity of Binary Search is

Best case	Average case	Worst case
$\Theta(1)$	Θ (log2n)	Θ (log2n)

Pros and Cons

Pros

Binary search is an optimal searching algorithm using which we can search the desired element very efficiently.

Cons

This algorithm requires the list to be sorted. Then only this method is applicable.

Applications of Binary Search

1. The binary search is an efficient searching method and is used to search desired record from database.
2. For solving non linear equations with one unknown this method is used.

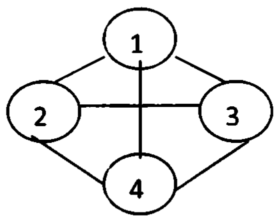
GRAPH:

Graph is a nonlinear data structure which is used in many applications. Graph is basically a collection of nodes or vertices that are connected by links called edges. There is a large variety of problems that can be solved using graph.

In this chapter we will first discuss how to represent a graph and what the algorithms used for traversal. We will understand the concept of connected components. Finally we will discuss the minimum spanning tree algorithm.

A graphs g consists of a set V of vertices (nodes) and a set E of edges (arcs). We write G=(V,E). V is a finite and non-empty set of vertices. E is a set of pair of vertices; these pairs are called as edges . Therefore, V(G).read as V of G, is a set of vertices and E(G),read as E of G is a set of edges. An edge e=(v, w) is a pair of vertices v and w, and to be incident with v and w.

A graph can be pictorially represented as follows,



We have numbered the graph as 1,2,3,4.

Therefore, V(G)=(1,2,3,4) and E(G) = {(1,2),(1,3),(1,4),(2,3),(2,4)}.

BASIC TERMINOLGIES OF GRAPH:

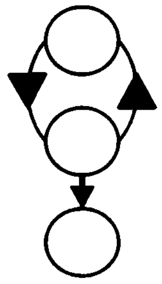
(a) ***UNDIRECTED GRAPH:***

An undirected graph is that in which, the pair of vertices representing the edges is unordered.

(b) ***DIRECTED GRAPH:***

An directed graph is that in which, each edge is an ordered pair of vertices, (i.e.) each edge is represented by a directed pair. It is also referred to as digraph.

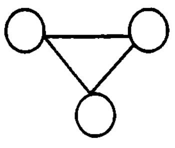
DIRECTED GRAPH



(c)*COMPLETE GRAPH:*

An n vertex undirected graph with exactly n(n-1)/2 edges is said to be complete graph. The graph G is said to be complete graph .

(d)*CONNECTED GRAPH*



An unconnected graph is said to be connected if for every pair of

distinct vertices V_i and V_i in V(G) there is a graph from for V_i to V_j in G.

TECHNIQUES FOR GRAPHS:

- The fundamental problem concerning graphs is the reach-ability problem.
- simplest form requires us to determine whether there exist a path in the given graph, G +(V,E) such that this path starts at vertex 'v' and ends at vertex 'u'.
- A more general form is to determine for a given starting vertex 'v' to a vertex 'u' such that there is a path from if it u.
- This problem can be solved by starting at vertex 'v' and systematically searching the graph 'G' for vertex that can be reached from 'v'.
- We describe 2 search methods for this.

i. Breadth first Search and Traversal.

ii. Depth first Search and Traversal.

BREADTH FIRST SEARCH AND TRAVERSAL:

Breadth first search:

In Breadth first search we start at vertex v and mark it as having been reached. The vertex v at this time is said to be unexplored. A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from v are visited next. There are new unexplored vertices. Vertex v has now been explored. The newly visited vertices have not been explored and are put onto the end of the list of unexplored vertices. The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left. The list of unexplored vertices acts as a queue and can be represented using any of the standard queue representations.

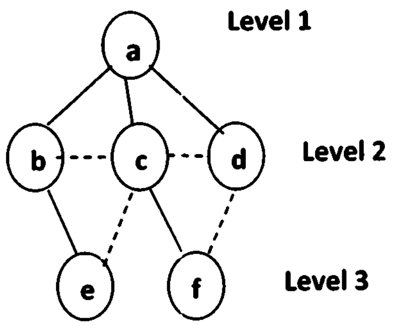
- In Breadth First Search we start at a vertex 'v' and mark it as having been reached (visited).
- The vertex 'v' is at this time said to be unexplored.
- A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjust from it.
- All unvisited vertices adjust from 'v' are visited next. These are new unexplored vertices.
- Vertex 'v' has now been explored. The newly visit vertices have not been explored and are put on the end of a list of unexplored vertices.
- The first vertex on this list in the next to be explored. Exploration continues until no unexplored vertex is left.
- The list of unexplored vertices operates as a queue and can be represented using any of the start queue representation.

Tree edge

In a graph G containing an edge (u, v), if a new unvisited vertex v is reached from the current vertex then edge (u, v) is called tree edge. The edges between these vertices are tree edges.

Cross edge

If an edge leading to its previously visited vertex other than its immediate predecessor is encountered then that edge is called cross edge. The cross edges are shown by dotted line and the tree edges are shown by solid edges.



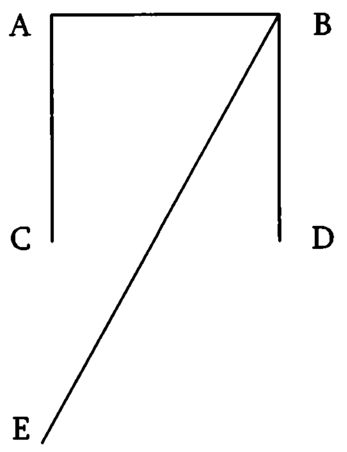
Algrothim : Breadth First Traversal

```
Algorithm BFT(G,n)
  {
   for i= 1 to n do
      visited[i] =0;
   for i =1 to n do
    if (visited[i]=0)then BFS(i)
  }
```

here the time and space required by BFT on an n-vertex e-edge graph one O(n+e) and O(n) resp if adjacency list is used. if adjancey matrix is used then the bounds are $O(n^2)$ and O(n) resp

Example

Graph is created with 5 vertices.



The Adjacency Matrix of the Graph is

```
01100
10011
10000
01000
01000
```

BREADTH FIRST TRAVERSAL

A==>> B==>> C==>> D==>> E

DEPTH FIRST SEARCH

A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At this time the exploration of the new vertex u begins. When this new vertex has been explored, the exploration of u continues. The search terminates when all reached vertices have been fully explored. This search process is best-described recursively.

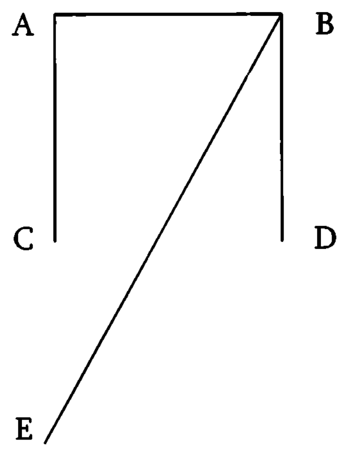
Algorithm

Algorithm DFS(v)

```
{
visited[v]=1
for each vertex w adjacent from v do
{
If (visited[w]=0)then
DFS(w);
}
}
```

Example

Graph is created with 5 vertices.



The Adjacency Matrix of the Graph is

```
0 1 1 0 0
0 0 0 1 1
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

DEPTH FIRST TRAVERSAL

A==>> B==>> D==>> E==>> C==>>0

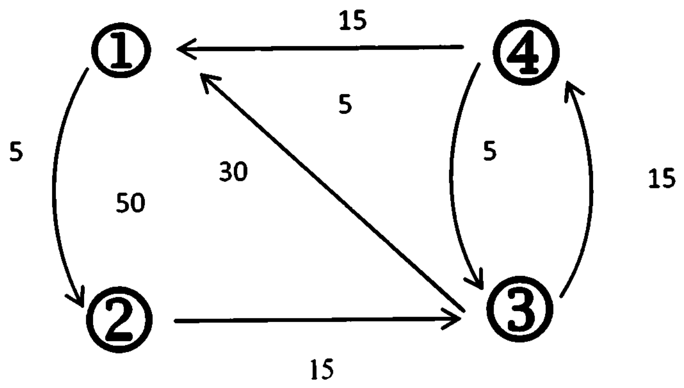
ALL PAIR SHORTEST PATH

- ❖ The all-pair shortest path algorithm is proposed by **R.Floyd.** Hence this algorithm is sometimes called as Floyd's algorithm
- ❖ Let G=<N,E> be a Weighted graph 'N' is a set of nodes and 'E' is the set of edges.
- ❖ Each edge has an associated non-negative length.
- ❖ We want to calculate the length of the shortest path between each pair of nodes.
- ❖ Suppose the nodes of G are numbered from 1 to n, so N={1,2,...N},and suppose G matrix W gives the length of each edge, with W(i,j)=0 for i=1,2...n, W(i,j)>=for all i & j, and W(i,j)=infinity, if the edge (i,j) does not exist.
- ❖ The principle of optimality applies: if k is the node on the shortest path from i to j then the part of the path from i to k and the part from k to j must also be optimal, that is shorter.
- ❖ First, create a cost adjacency matrix for the given graph.
- ❖ Copy the above matrix-to-matrix D, which will give the direct distance between nodes.
- ❖ We have to perform N iteration after iteration k. the matrix D will give you the distance between nodes with only (1,2...,k)as intermediate nodes.
- ❖ At the iteration k, we have to check for each pair of nodes (i,j) whether or not there exists a path from i to j passing through node k.

Algorithm

```
Algorithm Allpair(L,W)
{
for i=1 to n do
{
for j=1 to n do
{
D[I,j]=w[I,j]
}
}
fork=1 to n do
{
for i=1 to n do
{
for j=1 to n do
{
D [ i , j ] = min (D[ i, j ], D[ i, k ] + D[ k, j ] )
}
        }
}
```

Example



COST ADJACENCY MATRIX:

❖ At 0th iteration it nil give you the direct distances between any 2 nodes

D0= 0 5 μ μ

50 0 15 5

30 μ 0 15

15 μ 5 0

❖ At 1st iteration we have to check the each pair(i,j) whether there is a path through node 1.if so we have to check whether it is minimum than the previous value and if it is so then the distance through 1 is the value of d1(i,j).at the same time we have to solve the intermediate node in the matrix position p(i,j).

0 5 μ μ

50 0 15 5 p[3,2]= 1

D1= 30 **35** 0 15 p[4,2]= 1

15 **20** 5 0

likewise we have to find the value for N iteration (ie) for N nodes.

0 5 **20** **10** P[1,3] = 2

D2= 50 0 15 5 P[1,4] = 2

30 35 0 15

15 20 5 0

0 5 20 10

D3= **45** 0 15 5 P[2,1]=3

30 35 0 15

15 20 5 0

0 5 **15** 10

20 0 **10** 5 P[1,3]=4

D4= 30 35 0 15 P[2,3]=4

15 20 5 0

❖ D4 will give the shortest distance between any pair of nodes.

❖ If you want the exact path then we have to refer the matrix p. The matrix will be,

0 0 4 2

3 0 4 0 0 direct path

P= 0 1 0 0

0 1 0 0

❖ Since ,p[1,3]=4,the shortest path from 1 to3 passes through 4.

❖ Looking now at p[1,4]&p[4,3] we discover that between 1 & 4, we have to go to node 2 but that from 4 to 3 we proceed directly.

❖ Finally we see the trips from 1 to 2, & from 2 to 4, are also direct.

❖ The shortest path from 1 to 3 is 1,2,4,3.

Analysis of All pair shortest path

This algorithm takes a time of q (n^3)

SINGLE SOURCE SHORTEST PATH

If we want to travel from Mumbai to Bangalore then one giving the road map of India, we can find out the routes between these two cities. If we want to find the shortest path between these two cities then we find out all possible routes along with their distances of Mumbai to Bangalore. And we will select the route with minimum distance. But if we choose a distance from Mumbai to Jaipur is far away from Banglore then of course it will be poor choice. Because Jaipur is far away from Bangalore. These types of problems in computer science are solved by graph Theory. We can define shortest path. Weight from a to b by:

ä(a, b) = min{path(a, b)}	where path exist between a and b
= á	otherwise

DIJKSTRA'S ALGORITHM

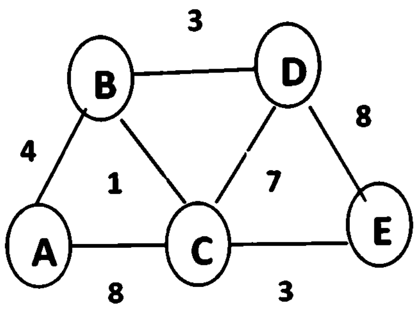
Dijkstra's algorithm is a popular algorithm for finding shortest path. This algorithm is called single source shortest path algorithm. In this algorithm, for a given vertex called source the shortest path to all other vertices is obtained. In this algorithm the main focus is not to find only one single path but to find the shortest paths form any vertex to all other remaining vertices. This algorithm applicable to graphs with non-negative weights only.

Dijkstra's algorithm find shortest paths to graph's vertices in order of that distance from given source. In this process of finding shortest path, first it finds the shortest path from the source to a vertex nearest and so on.

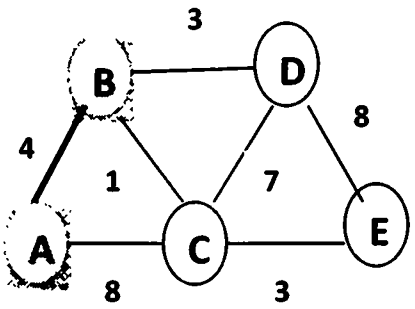
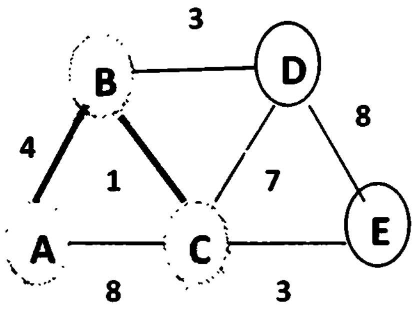
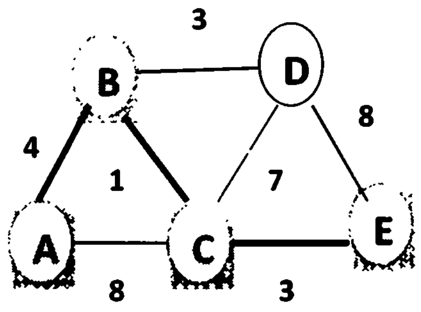
Algorithm Dijikstra (cost, source, dist)

```
{
for i=1 to tot_nodes
{
dist[i]=cost[source, i];
s[i]=0;
path[i]=source;
}
s[source]=1;
for i=1 to tot_nodes
{
min_dist=á;
v1=-1;
for i=1 to tot_nodes
{
If(s[j]==0)then
{
If(dist[j]<min_dist)then
{
min_dist=dist;
v1=j;
}
}
}
s[v1]=1;
for v2=1 to tot_nodes-1
{
If(s[v2]==0)then
{
If(dist[v1]+cost[v1,v2]<dist[v2])then
{
dist[v2]=cost[v1,v2]+dist[v1]);
path[v2]=v1;
}
}
}
}
}
```

Example : Consider a weighted connected graph as given below



Now we will consider each vertex as a source and will find the shortest distance from this vertex to every other remaining vertex. Let us start with vertex A

Source vertex	Distance with other vertices	Path shown in graph
A	**A-B, path =4** A-c, path =8 A-D, path=a A-E, path=a	
B	**B-C, path =4+1** B-D, path = 4+3 B-E, path=a	
C	C-D, path =5+7=12 **C-E, path= 5+3=8**	
D	D-E , path =7+8=15	

But we have one shortest distance obtained from A to E and that is A-B-C- E with path length =4+1+3=8.Similarly other shortest paths can be obtained by choosing appropriate source and destination.

Analysis

The performance of Dijkstra's algorithm depends on the time complexity of the function min-cost . We know from the chapter on heap that it is O(log V). The body of the while loop is executed —V— times, and hence the total time for all calls to min-cost is O(V log V). The for loop is executed O(E) times altogether, since the sum of the lengths of all adjacency lists is —E—. Within the for loop, the test for membership in vertex list can be implemented in constant time by keeping a bit for each vertex that tells whether or not it is in vertex list, and updating the bit when the vertex is removed from vertex list. The assignment in line path[v2]=v1 involves an implicit DECREASE-KEY operation on the min-heap, which can be implemented in a binary min-heap in O(log V) time.

Thus, the total time for Dijkstra's algorithm is O(V log V + E log V) = O(E log V).

Review Questions

1. What do you mean by 3Searching" problem?
2. Explain the procedure of Sequential Search
3. What is time complexity for Dijikstra's algorithm
4. Explain the procedure of Binary Search
5. Justify your answer with example , Which search is best one either Binary Search or Sequential Search
6. List out the applications of Binary Search
7. Define Graph
8. Describe BFS Method with Example
9. Describe DFS Method with Example
10. Describe the Dijikstra's Algorithm with example
11. Draw (a) direct Graph (b) Connected Graph

Write an Algorithm for All pair shortest path.

CHAPTER 13 POINTERS

A pointer is a variable that represents the location or address of a variable or array element.

Uses

1. They are used to pass information back and forth between a function and calling point.

2. They provide a way to return multiple date items.

3. They provide alternate way to access individual array elements.

When we declare a variable say x, the computer reserves a memory cell with name x. the data stored in the variable is got through the name x. another way to access data is through the address or location of the variable. This address of x is determined by the expression &x, where & is a unary operator (called address operator). Assign this expression &x to another variable px(i.e. px=&x).this new variable px is called a pointer to x (since it points to the location of x. the data stored in x is accessed by the expression *px where * is a unary operator called the indirection operator.

Ex: if x=3 and px=&x then *px=3

Declaration and Initialisation

A pointer variable is to be declared initially. It is done by the syntax.

Data type *pointer variable

Int *p declares the variable p as pointer variable pointing to a an integer type data. It is made point to a variable q by p= &q. In p the address of the variable q is stored. The value stored in q is got by *p.

If y is a pointer variable to x which is of type int, we declare y as int *y ;

Ex: float a;

float *b;

b=&a;

Note : here in 'b' address of 'a' is stored and in '*b' the value of a is stored.

Passing pointers to a function

Pointers are also passed to function like variables and arrays are done. Pointers are normally used for passing arguments by reference (unlike in the case if variable and arrays, they are passed by values). When data are passed by values the alteration made to the data item with in the function are not carried over to the calling point; however when data are passed by reference the case is otherwise. Here the address of data item is passed and hence whatever changes occur to this within the function, it will be retained throughout the execution of the program. So generally pointers are used in the place of global variables.

Example:

```
#include<stdio.h>

void f(int*px, int *py

main()
{
int x = 1;
int y=2;
f(&X,&y);
printf("\n %d%d", x,y);
}
Void f(int *px, int *py);
*px=*px+1;
*py=*py+2;
return;
}
```

Note:

1. here the values of x and y are increased by 1 and 2 respectively.

2. arithmetic operations *, +, -, / etc can be applied to operator variable also.

Pointer and one dimensional arrays

An array name is really a pointer to the first element in the array i.e. if x is a one dimensional array, the name x is &x[0] and &x[i] are x + i for i= 1,2,....... So to read array of numbers we can also use the following statements

```
int x[100],n;
for (i=1 ; i<=n; ++i)
scanf ('%d", x + i ) (in the place of scanf ("%d", &x[i] ) )
```

Note : the values stored in the array are got by * (x + i) in the place x[i].

DYNAMIC MEMORY ALLOCATION

Usually when we use an array in c program, its dimension should be more than enough or may not be sufficient. To avoid this drawback we allocate the proper (sufficient) dimensions of an array during the run time of the program with the help of the library functions called memory management functions like 'malloc','calloc', 'realloc' etc. The process of allocating memory at run time is known as dynamic memory allocation.

Example :

To assign sufficient memory for x we use the following statement

x= (int *) malloc (n* sizeof (int)) , for in the place of initial declaration int x[n]

Similarly in the place of float y [100] we use y = (float *) malloc (m* sizeof (float));

Example to read n numbers and find their sum

```
main()
{
int *x, n, i, sum=0;
Printf("\n Enter number of numbers");
Scanf("%d", &n);
x=(int *)malloc(n * sizeof(int));
for(i=1;i<=n,++i)
{
scanf("%d", x+i):
sum += *(x+i);
}
Printf("\nThe sum is %d ", sum);
}
```

Passing function to other function

A pointer to a function can be passed to another pointer as an assignment. Here it allows one function to be transferred as if the function were a variable.

Pointers and Structure

Like we have array of integers, array of pointer etc, we can also have array of structure variables. And to make the use of array of structure variables efficient, we use **pointers of structure type**. We can also have pointer to a single structure variable, but it is mostly used with array of structure variables.

Example

```
#include <stdio.h>
#include <string.h>
struct student
{
    int id;
    char name[30];
    float percentage;
};
int main()
{
    int i;
   struct student record1 = {1, "Raju", 90.5};
   struct student *ptr;
   ptr = &record1;
      printf("Records of STUDENT1: \n");
      printf("  Id is: %d \n", ptr->id);
      printf("  Name is: %s \n", ptr->name);
     printf("  Percentage is: %f \n\n", ptr->percentage);
   return 0;
}
```

Review Questions

1. Discuss the various arithmetic operations performed on a pointer.
2. Distinguish bet malloc and calloc()
3. Explain array of pointers.
4. Explain dynamic memory allocation?
5. Explain how to access a value using pointer?give a suitable example.
6. Explain pointer to function in detail.
7. Explain pointer to structure in detail.
8. Explain what is pointer?expalin with suitable example
9. What do you understand by pointers? Give the syntax of declaration of a pointer.
10. What is a null pointer?
11. What is advantage of representing an array of string by an array of pointer to string.
12. What is the difference between the array of pointer and pointer to the arra
13. Write a short note on pointer to pointer.

CHAPTER 14 DATA FILES

Data Files are to store data on the memory device permanently and to access whenever is required.

There are two types of data files

1 Stream Oriented data files

2 System Oriented data files

Stream oriented data files are either text files or unformatted files. System oriented data files are more closely related to computer's operating system and more complicated to work with. In this session we go through stream oriented data files.

Opening and Closing data files

The first step is to create a buffer area where information is stored temporarily before passing to computer memory. It is done by writing

File *fp;

Here fp is the pointer variable to indicate the beginning of the buffer area and called stream pointer .

The next step is to open a data file specifying the type i.e. read only file, write only file, read / write file. This is done by using the library function fopen

The syntax is

fp = fopen(filename,filetype)

The filetype can be

1. 'r' (to open an existing file for reading only)
2. 'w' (to open a new file for writing only. If file with filename exists, it will be destroyed and a new file is created in its place)
3. 'a' (to open an existing file for appending. If the file name does not exist a new file with that file name will be created)
4. 'r+' (to open an existing file for both reading and writing)
5. 'w+' (to open a new file for reading and writing. If the file exists with that name, it will be destroyed and a new one will be created with that name)
6. 'a+' (to open an existing file for reading and writing. If the file does not exist a new file will be created).

For writing formatted data to a file we use the function fprintf.

The syntax is

fprintf(fp,"conversion string", value);

For example to write the name "rajan" to the file named 'st.dat'

File *fp;

```
fp=fopen("st.dat",'w');
fprintf(fp,"%[^\n]","rajan");
```

The last step is to close the file after the desired manipulation. This is done by the library function fclose.
The syntax is

fclose(fp);

Example

1. To create a file of biodata of students with name 'st.dat'.

```
#include<stdio.h>
#include<string.h>
Tpedef struct
{
Int day;
Int month;
Int year;
}date;
Typedef Struct
{
char name(30);
char place(30);
int age;
date birthdate;
}biodata;
main()
{
File *fp;
biodata student;
fp=fopen("st.dat",'w');
printf("Input data");
scanf("%[^\n]",student.name);
scanf("%[^\n]",student.place);
scanf("%d",&student.age);
scanf("%d",&student.birthdate.day);
scanf("%d",&student.birthdate.month):
scanf("%d",&student.birthdate.year);
fprintf(fp,"%s%s%d%d%d%d",student.name,student.place,student.age,student.birthdate.day,
student.birthdate.month, student.birthdate.year)
fclose(fp);
}
```

Example 2: To write a set of numbers to a file.

```
#include<stdio.h>
main()
{
file *fp;
Int n; float x
fp=fopen("num.dat",'w');
printf("Input the number of numbers");
scanf("%d",&n);
for(i=1;i<=n;++i)
{
scanf("%d",&x);
fprintf(fp,"%f\n",x);
}
fclose(fp);
}
```

Processing formatted data File

To read formatted data from a file we have to follow all the various steps that discussed above. The file should be opened with read mode. To open the existing file 'st.dat' write the following syntax :

```
file *fp;
fp=fopen("st.dat", 'r+');
```

For reading formatted data from a file we use the function fscanf

Example:

```
Tpedef struct
{
Int day;
Int month;
Int year;
}date;
Typedef Struct
{
```

```
char name(30);
char place(30);
int age;
date birthdate;
}biodata;
main()
{
File *fp;
biodata student;
fp=fopen("st.dat",'r+');
fscanf(fp,"%s",student.name);
printf("%s",student.name);
fclose(fp);
}
```

Processing Unformatted data files

For reading and writing unformatted data to files we use the library functions fread and fwrite in the place of fscanf and fprintf.

The syntax for writing data to file 'st.dat' with stream pointer fp is

```
Fwrite(&student, sizeof(record),1,fp);
```

Here student is the structure of type biodata

Example:

To write biodata to a file

```
Tpedef struct
{
Int day;
Int month;
Int year;
}date;
Typedef Struct
{
char name(30);
char place(30);
int age;
date birthdate;
}biodata;
```

```
Main()
{
File *fp;
fp=fopen("st.dat",'a+')
biodata student;
Printf("Input data");
Scanf("%[^\n]",student.name);
Scanf("%[^\n]",student.place);
Scanf("%d",&student.age);
Scanf("%d",&student.birthdate.day);
Scanf("%d",&student.birthdate.month):
Scanf("%d",&student.birthdate.year);
Fwrite(&student,sizeof(record),1,fp);
Fclose(fp);
}
```

Example 2: To read biodata from the file.

```
Tpedef struct
{
Int day;
Int month;
Int year;
}date;
Typedef Struct
{
char name(30);
char place(30);
int age;
date birthdate;
}biodata;
main()
{
File *fp;
fp=fopen("st.dat",'a+`)
biodata student;
fread(&student,sizeof(record),1,fp);
printf("%s\n",student.name);
printf("%s\n]",student.place);
printf("%d\n",&student.age);
printf("%d\n",&student.birthdate.day);
printf("%d\n",&student.birthdate.month):
printf("%d\n",&student.birthdate.year);
fclose(fp);
}
```

Review Questions

1. Explain the following i)rewind ii)feof
2. How does the fopen() works?Explain it with example.
3. What role does the fseek() plays and how many arguments does it Have?
4. Write a short note on file handling in C
5. Explain is EOF and BOF.
6. Explain the following types of file i)sequential,ii)Index sequential and iii)Direct file
7. List any three file mode in C?
8. What is EOF and what value does usually have?

1. *The Hello program*

```
/* hello.c — The most famous program of them all ..
 */
#include <stdio.h>
int main(void) {
 printf("Hello World!\n");
 // return 0;
}
```

2. *Computing powers of 2*

```
/* power2.c — Print out powers of 2: 1, 2, 4, 8, .. up to 2^N
 */
#include <stdio.h>
#define N 16
int main(void) {
 int n;        /* The current exponent */
 int val = 1;   /* The current power of 2 */
 printf("\t n \t  2^n\n");
 printf("\t================\n");
 for (n=0; n<=N; n++) {
  printf("\t%3d \t %6d\n", n, val);
   val = 2*val;
 }
 return 0;
}
```

3. *Adding two integers*

```
/* add2.c — Add two numbers and print them out together
          with their sum
          AUTHOR:
               DATE:
 */
#include <stdio.h>
int main(void) {
 int first, second;
 printf("Enter two integers > ");
 scanf("%d %d", &first, &second);
 printf("The two numbers are: %d  %d\n", first, second);
 printf("Their sum is %d\n", first+second);
}
```

4. ***Adding n integers***

```
/* addn.c — Read a positive number N. Then read N integers and
 *      print them out together with their sum.
 */
#include <stdio.h>
int main(void) {
 int n;     /* The number of numbers to be read */
 int sum;    /* The sum of numbers already read  */
 int current; /* The number just read            */
 int lcv;    /* Loop control variable, it counts the number
             of numbers already read */
 printf("Enter a positive number n > ");
 scanf("%d",&n); /* We should check that n is really positive*/
 sum = 0;
 for (lcv=0; lcv < n; lcv++) {
  printf("\nEnter an integer > ");
  scanf("%d",&current);
  /*  printf("\nThe number was %d\n", current); */
  sum = sum + current;
 }
 printf("The sum is %d\n", sum);
 return 0;
}
```

5. ***Adding a sequence of positive integers***

```
/* add.c — Read a sequence of positive integers and print them
 *      out together with their sum. Use a Sentinel value
 *      (say 0) to determine when the sequence has terminated.
 */
#include <stdio.h>
#define SENTINEL 0
int main(void) {
 int sum = 0; /* The sum of numbers already read */
 int current; /* The number just read */
 do {
   printf("\nEnter an integer > ");
   scanf("%d", &current);
    if (current > SENTINEL)
      sum = sum + current;
  } while (current > SENTINEL);
 printf("\nThe sum is %d\n", sum);
}
```

6. ***Computing the factorial of a number***

```
/* factorial.c — It computes repeatedly the factorial of an integer entered
 *     by the user. It terminates when the integer entered is not
 *      positive.
 */
#include <stdio.h>
int fact(int n);
int main(void) {
 int current;
 printf("Enter a positive integer [to terminate enter non-positive] > ");
 scanf("%d", &current);
 while (current > 0) {
   printf("The factorial of %d is %d\n", current, fact(current));
   printf("Enter a positive integer [to terminate enter non-positive] > ");
   scanf("%d", &current);
  }
}
/* n is a positive integer. The function returns its factorial */
int fact(int n) {
 int lcv;   /* loop control variable */
 int p;     /* set to the product of the first lcv positive integers */
 for(p=1, lcv=2; lcv <= n; p=p*lcv, lcv++);
 return p;
}
```

7. ***Determining if a number is a prime***

```
/* prime1.c It prompts the user to enter an integer N. It prints out
 * if it is a prime or not. If not, it prints out a factor of N.
 */
#include <stdio.h>
int main(void) {
 int n;
 int i;
 int flag;
 printf("Enter value of N > ");
 scanf("%d", &n);
 flag = 1;
 for (i=2; (i<(n/2)) && flag; ) { /* May be we do not need to test
                                  values of i greater than the square root of n? */
   if ((n % i) == 0) /* If true n is divisible by i */
     flag = 0;
   else
     i++;
 }
  if (flag)
  printf("%d is prime\n", n);
 else
  printf("%d has %d as a factor\n", n, i);
 return 0;
}
```

***8.	<u>What are in C the values of TRUE and FALSE?</u>***

```
/* true.c — What are in C the values of TRUE and FALSE?
 */
 #include <stdio.h>
 int main(void) {
   printf("The value of 1<2 is %d\n", (1<2));
   printf("The value of 2<1 is %d\n", (2<1));
 }
 /* The program prints out
 The value of 1<2 is 1
 The value of 2<1 is 0
*/
```

***9.	<u>Computing Fibonacci numbers</u>***

```
/* fibo.c — It prints out the first N Fibonacci
 *        numbers.
 */
#include <stdio.h>
int main(void) {
  int n;      /* The number of fibonacci numbers we will print */
  int i;      /* The index of fibonacci number to be printed next */
  int current; /* The value of the (i)th fibonacci number */
  int next;   /* The value of the (i+1)th fibonacci number */
  int twoaway; /* The value of the (i+2)th fibonacci number */
  printf("How many Fibonacci numbers do you want to compute? ");
  scanf("%d", &n);
  if (n<=0)
   printf("The number should be positive.\n");
  else {
   printf("\n\n\tI \t Fibonacci(I) \n\t=====================\n");
    next = current = 1;
    for (i=1; i<=n; i++) {
          printf("\t%d \t  %d\n", i, current);
          twoaway = current+next;
          current = next;
          next   = twoaway;
    }
  }
}
```

10. <u>*Simple example on scope rules*</u>

```
/*scope1.c — Simple example showing effects of the scope rules */
#include <stdio.h>
int a=0;   /* This is a global variable */
void foo(void);
int main(void) {
int a=2;  /* This is a variable local to main */
int b=3;  /* This is a variable local to main */

printf("1. main_b = %d\n", b);
printf("main_a = %d\n", a);
foo();
printf("2. main_b = %d\n", b);
}
void foo(void){
int b=4;  /* This is a variable local to foo */
printf("foo_a = %d\n", a);
printf("foo_b = %d\n", b);
}
```

11. ***Reading, writing, reversing an integer array***

```
/* array2.c — Read/writing/reversing integer arrays */
#include <stdio.h>
#define NMAX 10
void intSwap(int *x, int *y);
int getIntArray(int a[], int nmax, int sentinel);
void printIntArray(int a[], int n);
void reverseIntArray(int a[], int n);
int main(void) {
 int x[NMAX];
 int hmny;
 hmny = getIntArray(x, NMAX, 0);
 printf("The array was: \n");
 printIntArray(x,hmny);
 reverseIntArray(x,hmny);
 printf("after reverse it is:\n");
 printIntArray(x,hmny);
}
void intSwap(int *x, int *y)
   /* It swaps the content of x and y */
{
 int temp = *x;
 *x = *y;
 *y = temp;
}
void printIntArray(int a[], int n)
   /* n is the number of elements in the array a.
    * These values are printed out, five per line. */
{
 int i;

 for (i=0; i<n; ){
  printf("\t%d ", a[i++]);
  if (i%5==0)
   printf("\n");
 }
 printf("\n");
}
```

```
int getIntArray(int a[], int nmax, int sentinel)
   /* It reads up to nmax integers and stores then in a; sentinel
    * terminates input. */
{
 int n = 0;
 int temp;
 do {
  printf("Enter integer [%d to terminate] : ", sentinel);
  scanf("%d", &temp);
  if (temp==sentinel) break;
  if (n==nmax)
    printf("array is full\n");
   else
    a[n++] = temp;
 }while (1);
 return n;
}
void reverseIntArray(int a[], int n)
   /* It reverse the order of the first n elements of a */
{
 int i;
for(i=0;i<n/2;i++){
  intSwap(&a[i],&a[n-i-1]);
 }
}
```

12. ***<u>Numeric value of printable characters</u>***

```
/* codes.c — It prints out the numerical codes of the printable ascii
 *         characters
 */
#include <stdio.h>
int main(void){
 int c;
 printf("\tCharacter Code\n"
      "\t==============\n");
 for (c=32; c<127; c++)
  printf("\t %c    %4d\n", c, c);
}
```

13. Reading an array and doing linear searches on it

```
/* linear.c — Read an integer array and then do linear searches.
 */
#include <stdio.h>
#define NMAX 10
int getIntArray(int a[], int nmax, int sentinel);
void printIntArray(int a[], int n);
int linear(int a[], int n, int who);
int main(void) {
 int x[NMAX];
 int hmny;
 int who;
 int where;
 hmny = getIntArray(x, NMAX, 0);
 printf("The array was: \n");
 printIntArray(x,hmny);
 printf("Now we do linear searches on this data\n");
 do{
  printf("Enter integer to search for [0 to terminate] : ");
  scanf("%d", &who);
  if(who==0)break;
  where = linear(x,hmny,who);
  if (where<0){
   printf("Sorry, %d is not in the array\n",who);
  }else
   printf("%d is at position %d\n",who,where);
 }while(1);
}
void printIntArray(int a[], int n)
   /* n is the number of elements in the array a.
    * These values are printed out, five per line. */
{
 int i;

 for (i=0; i<n; ){
  printf("\t%d ", a[i++]);
  if (i%5==0)
```

```
    printf("\n");
  }
 printf("\n");
}
int getIntArray(int a[], int nmax, int sentinel)
   /* It reads up to nmax integers and stores then in a; sentinel
    * terminates input. */
{
 int n = 0;
 int temp;
 do {
  printf("Enter integer [%d to terminate] : ", sentinel);
  scanf("%d", &temp);
  if (temp==sentinel) break;
  if (n==nmax)
    printf("array is full\n");
   else
    a[n++] = temp;
 }while (1);
 return n;
}

int linear(int a[], int n, int who)
   /* Given the array a with n elements, searches for who.
    * It returns its position if found, otherwise it returns
    * -1.
    */
{
 int lcv;
 for (lcv=0;lcv<n;lcv++)
  if(who == a[lcv])return lcv;
 return (-1);
}
```

14. String Functions

```
/* string1.c — Simple string operations
            String literals.
                      printf, scanf, %s, %c
```

```
                    strlen
                    strcpy
                    strcmp
 */
#include <stdio.h>
#define MAXBUFF 128
int main(void) {
 char c[] = "012345";
 char line[MAXBUFF];
 int lcv;
 int cmp;

 printf("sizeof(c)= %d\n", sizeof(c));
 printf("sizeof(line)= %d\n", sizeof(line));
 for (lcv=0; lcv<=strlen(c); lcv++)
  printf("c[lcv]= %d = %c\n",c[lcv],c[lcv]);
 printf("Please enter a string : ");
 scanf("%s",line);
 printf("strlen(line) = %d\n", strlen(line));
 printf("line = [%s]\n",line);
 cmp = strcmp(c,line);
 if(cmp<0)
  printf("%s is less than %s\n", c, line);
 else if (c==0)
  printf("%s is equal to %s\n", c, line);
 else
  printf("%s is greater than %s\n", c, line);
 strcpy(line,c);  /*copy the string c into line */
 cmp = strcmp(c,line);
 if(cmp<0)
  printf("%s is less than %s\n", c, line);
 else if (cmp==0)
  printf("%s is equal to %s\n", c, line);
 else
  printf("%s is greater than %s\n", c, line);
}
```

15. *Using enumerations*

```
/* enum1.c — Starting to use enumerated types: Printing for each
 *       day of the week, today, yesterday, and tomorrow, both
 *        as a string and as a number.
 */
#include <stdio.h>
/* Introducing an enumerated data type */
enum days {monday,tuesday,wednesday,thursday,friday,saturday,sunday};
typedef enum days days; // This allows us to use "days" as an abbreviation
                // for "enum days"

/* Two useful functions */
days yesterday(days today){
 return (today+6)%7;
}
days tomorrow(days today){
 return (today+1)%7;
}
// A useful array: thedays is an array of constant (i.e you cannot
// modify them) pointers to constant (i.e. you cannot modify them) strings
const char * const thedays[] =
           {"monday", "tuesday", "wednesday", "thursday",
                  "friday", "saturday", "sunday"};
int main(void){
 days today;
 printf("today   \tyesterday \ttomorrow\n"
     "============================================\n");
 for (today=monday;today<=sunday;today++)
   printf("%s = %d \t %s = %d \t %s = %d\n",
          thedays[today], today,
          thedays[yesterday(today)], yesterday(today),
          thedays[tomorrow(today)], tomorrow(today));
}
```

16. *Reading an array and doing binary searches on it*

```
/* binary.c - Binary search using two methods. The first is more intuitive, but it is slower. */
#include <stdio.h>
#include <sys/time.h>
```

```
/* Given a sorted array with n elements, we search for who using binary search. We return a position where
found, or -1 if not there */
int binary1(int n, int a[n], int who) {
  int left = 0;
  int right = n-1;
  while (left <= right) {
        int mid = left + (right-left)/2;
        if (who < a[mid])
          right = mid - 1;
        else if (who > a[mid])
          left = mid + 1;
        else
          return mid;
  }
  return -1;
}
/* Given a sorted array with n elements, we search for who using binary search. We return a position where
found, or -1 if not there */
int binary2(int n, int a[n], int who)
{
  int p = n/2;
  while (n > 0) {
        n = n/2;
        if (who < a[p]) {
          p -= n;
        } else if (who > a[p]) {
          p += n;
        } else
          return p;
  }
  return -1;
}
/* Returns the difference in microseconds between before and after */
long timediff(struct timeval before, struct timeval after) {
  long sec = after.tv_sec - before.tv_sec;
  long microsec = after.tv_usec - before.tv_usec;
  return 1000000*sec + microsec;
}
```

```
int main() {
   int a[] = {1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33};
   int n = sizeof(a)/sizeof(int);
   int where;
   struct timeval before;
   struct timeval after;
   int k;
   int j;
   gettimeofday(&before, NULL);
   for (j = 0; j < 1000000; j++)
   for (k = 0; k < 2*n+1; k++) {
            where = binary1(n, a, k);
//          printf("who = %d, \tvalue = %d\n", k, where);
   }
   gettimeofday(&after, NULL);
   printf("before=[%ld,%ld], after=[%ld,%ld]\n", before.tv_sec, before.tv_usec,
                                after.tv_sec, after.tv_usec);
   printf("The difference is %ld\n", timediff(before, after));
   printf("————————————————————————————————\n");
   gettimeofday(&before, NULL);
   for (j = 0; j < 1000000; j++)
   for (k = 0; k < 2*n+1; k++) {
            where = binary2(n, a, k);
//          printf("who = %d, \tvalue = %d\n", k, where);
   }
   gettimeofday(&after, NULL);
   printf("before=[%ld,%ld], after=[%ld,%ld]\n", before.tv_sec, before.tv_usec,
                                after.tv_sec, after.tv_usec);
   printf("The difference is %ld\n", timediff(before, after));
   return 0;
}
```

17. ***Sorting an array of integers with Selection Sort***

```
/* selection.c — Read an integer array, print it, then sort it and
 * print it. Use the selection sort method.
 */
#include <stdio.h>
#define NMAX 10
```

```
/* It reads up to nmax integers and stores then in a; sentinel
 * terminates input. '/
int getIntArray(int nmax, int a[nmax], int sentinel);
/* n is the number of elements in the array a.
 * These values are printed out, five per line. */
void printIntArray(int n, int a[]);
/* It sorts in non-decreasing order the first N positions of A. It uses
 * the selection sort method.
 */
void selectionSort(int n, int a[]);
int main(void) {
 int x[NMAX];
 int hmny;
 hmny = getIntArray(NMAX, x, 0);
 if (hmny==0)
  printf("This is the empty array!\n");
 else{
  printf("The array was: \n");
  printIntArray(hmny, x);
  selectionSort(hmny, x);
  printf("The sorted array is: \n");
  printIntArray(hmny, x);
 }
 return 0;
}
/* n is the number of elements in the array a.
 * These values are printed out, five per line. */
void printIntArray(int n, int a[n])
{
 int i;
 for (i=0; i<n; ){
  printf("\t%d ", a[i++]);
  if (i%5==0)
   printf("\n");
 }
 printf("\n");
}
```

```
/* It reads up to nmax integers and stores then in a; sentinel
 * terminates input. */
int getIntArray(int nmax, int a[nmax], int sentinel)
{
 int n = 0;
 int temp;
 do {
  printf("Enter integer [%d to terminate] : ", sentinel);
  scanf("%d", &temp);
  if (temp==sentinel) break;
  if (n==nmax)
    printf("array is full\n");
   else
    a[n++] = temp;
 }while (1);
 return n;
}
/* It sorts in non-decreasing order the first N positions of A. It uses
 * the selection sort method.
 */
void selectionSort(int n, int a[])
{
 int lcv;
 int rh;     /*Elements in interval rh..n-1 are in their final position*/
 int where;  /*Position where we have current maximum*/
 int temp;   /*Used for swapping*/

 for(rh=n-1;rh>0;rh—){
  /*Find position of largest element in range 0..rh*/
  where = 0;
  for (lcv=1;lcv<=rh;lcv++)
   if (a[lcv]>a[where])
          where = lcv;
  temp = a[where];
  a[where] = a[rh];
  a[rh] = temp;
 }
}
```

18. Sorting an array of integers with Bubble Sort

```
/* bubble.c — Read an integer array, print it, then sort it and
 * print it. Use the bubble sort method.
 */
#include <stdio.h>
#define NMAX 10
int getIntArray(int a[], int nmax, int sentinel);
void printIntArray(int a[], int n);
void bubbleSort(int a[], int n);
int main(void) {
 int x[NMAX];
 int hmny;
 int who;
 int where;
 hmny = getIntArray(x, NMAX, 0);
 if (hmny==0)
  printf("This is the empty array!\n");
 else{
  printf("The array was: \n");
  printIntArray(x,hmny);
  bubbleSort(x,hmny);
  printf("The sorted array is: \n");
  printIntArray(x,hmny);
 }
}
void printIntArray(int a[], int n)
  /* n is the number of elements in the array a.
   * These values are printed out, five per line. */
{
 int i;

 for (i=0; i<n; ){
  printf("\t%d ", a[i++]);
  if (i%5==0)
   printf("\n");
 }
 printf("\n");
}
```

```
int getIntArray(int a[], int nmax, int sentinel)
   /* It reads up to nmax integers and stores then in a; sentinel
    * terminates input. */
{
 int n = 0;
 int temp;

 do {
  printf("Enter integer [%d to terminate] : ", sentinel);
  scanf("%d", &temp);
  if (temp==sentinel) break;
  if (n==nmax)
    printf("array is full\n");
   else
    a[n++] = temp;
 }while (1);
 return n;
}
void bubbleSort(int a[], int n)
/* It sorts in non-decreasing order the first N positions of A. It uses
 * the bubble sort method.
 */
{
  int lcv;
  int limit = n-1;
  int temp;
  int lastChange;
  while (limit) {
    lastChange = 0;
    for (lcv=0;lcv<limit;lcv++)
     /* Notice that the values in positions LIMIT+1 .. N are in
      * their final position, i.e. they are sorted right */
           if (a[lcv]>a[lcv+1]) {
             temp = a[lcv];
             a[lcv] = a[lcv+1];
             a[lcv+1] = temp;
             lastChange = lcv;
```

```
        }
    limit = lastChange;
  }
}
```

19. <u>MergeSort</u>

```
#include<stdio.h>
#define MAX 50
void mergeSort(int arr[],int low,int mid,int high);
void partition(int arr[],int low,int high);
int main(){
  int merge[MAX],i,n;
  printf("Enter the total number of elements: ");
  scanf("%d",&n);
  printf("Enter the elements which to be sort: ");
  for(i=0;i<n;i++){
     scanf("%d",&merge[i]);
  }
  partition(merge,0,n-1);
  printf("After merge sorting elements are: ");
  for(i=0;i<n;i++){
     printf("%d ",merge[i]);
  }
  return 0;
}
void partition(int arr[],int low,int high){
  int mid;
  if(low<high){
     mid=(low+high)/2;
     partition(arr,low,mid);
     partition(arr,mid+1,high);
     mergeSort(arr,low,mid,high);
  }
}
void mergeSort(int arr[],int low,int mid,int high){
  int i,m,k,l,temp[MAX];
  l=low;
  i=low;
  m=mid+1;
  while((l<=mid)&&(m<=high)){
      if(arr[l]<=arr[m]){
         temp[i]=arr[l];
         l++;
      }
      else{
         temp[i]=arr[m];
         m++;
      }
      i++;
  }
  if(l>mid){
     for(k=m;k<=high;k++){
        temp[i]=arr[k];
        i++;
     }
  }
  else{
     for(k=l;k<=mid;k++){
        temp[i]=arr[k];
        i++;
     }
  }

  for(k=low;k<=high;k++){
      arr[k]=temp[k];
  }
}
```

20. __Simple uses of structures__

```
// struct.c - Simple uses of structures
#include <stdio.h>
struct student {
  int  mid;
  int final;
  int hmws;
};
```

```
void main(void){
  struct student sam = {85, 90, 88};
  struct student tom = {93, 88, 91};
  struct student *he = &tom; // We can access a structure and its fields through a pointer
  // We cannot read or write directly a studentas we can integers, and reals, etc.
  // Also, we cannot compare directly two students (i.e. we cannot say (sam == tom)].
  // But we can assign a student to another student. For example:
  tom = sam;
  // Then we can output the value of tom by printing its various fields:
  printf("tom = {%d, %d, %d}\n", tom.mid, tom.final, tom.hmws);
     // Notice the dot notation for accessing the fields, of tom.
     // The output will be tom = {85, 90, 88}
  printf("*he = {%d, %d, %d}\n", he->mid, he->final, he->hmws);
     // Notice the arrow notation for accessing the fields from he.
     // The output will be *he = {85, 90, 88}
  // We would get the same output if we access the fields in a different way:
  printf("*he = {%d, %d, %d}\n", (*he).mid, (*he).final, (*he).hmws);
}
```

21. *Reading and writing to a file an array of structures*

```
/* studentarray.c - Reads a file containing a sequence of records
 *            representing students, places them into an
 *            array, then writes that array out to a new files.
 *            The names of the files are passed in as command
 *             line parameters.
 */
#include <stdio.h>
#define SIZE 10
#define NAMESIZE 25
typedef struct {
 char name[NAMESIZE];
 int  midterm;
 int final;
 int homeworks;
} student;
void writeStudentArray(char filename[], student a[], int n)
  /* n is the number of elements in the array a.
```

```
    * filename is the name of the file where we will
     * write.
     */
{
 FILE *fd;  /* File descriptor used for filename */
 int i;
 if(n<=0)
  return;
 if((fd=fopen(filename,"w"))==NULL){
  perror("fopen");
   exit(1);
  }
  for (i=0;i<n;i++){
   fprintf(fd,"%s %d %d %d\n",
               a->name, a->midterm, a->final, a->homeworks);
    a++;
  }
  fclose(fd);
 }
 int readStudentArray(char filename[], student a[], int nmax)
    /* It reads up to nmax student records
     * from file filename and stores them in a.
     * It returns the number of records actually read.
      */
 {
  FILE *fd;  /* File descriptor used for filename */
  int i=0;
  if((fd=fopen(filename,"r"))==NULL){
   perror("fopen");
    exit(1);
   }
  while(fscanf(fd,"%s %d %d %d",
                 a->name, &a->midterm, &a->final, &a->homeworks)!=EOF){
   if(++i==nmax)break;  /* We have filled up the table */
    a++;
   }
   fclose(fd);
 return i;
}
int main(int argc, char *argv[]){
 int n;
 student table[SIZE];
 if(argc!=3){
  printf("Usage: %s infile outfile\n", argv[0]);
  exit(0);
 }
 n = readStudentArray(argv[1],table,SIZE);
 writeStudentArray(argv[2],table,n);
}
```

22. *Create a binary file from a text file and then read from it*

```
/* makebinfile.c - Reads a file containing a sequence of text records
 *            and writes it out to a new binary files.
 *            The names of the files are passed in as command
 *            line parameters.
 */
#include <stdio.h>
#define SIZE 10
#define NAMESIZE 25
typedef struct {
  char name[NAMESIZE];
  int  midterm;
  int final;
  int homeworks;
} student;
int writeastudent(FILE *fdout, student * who){
  /* Write to an open binary file fdout the content of who.
   * Return the number of bytes that were written out.
   */
  char * p;      /* Cursor in outputting a byte at a time */
  char * limit = ((char *)who)+sizeof(student); /*Address just past who */
  for (p=(char *)who;p<limit;p++){
   fputc(*p, fdout);
  }
  return (limit - (char *)who);
 }
 int main (int argc, char *argv[]){
  int n = 0;       /* Number of records read */
  int m;           /* Number of bytes in a record */
  student who;     /* Buffer for a record */
  FILE *fdin;  /* File descriptor for input file */
  FILE *fdout; /* File descriptor for output file */
  if(argc!=3){
   printf("Usage: %s infile outfile\n", argv[0]);
   exit(0);
  }

 if((fdin=fopen(argv[1],"r"))==NULL){
  perror("fopen");
   exit(1);
 }
 if((fdout=fopen(argv[2],"w"))==NULL){
  perror("fopen");
  exit(1);
 }
 while(fscanf(fdin,"%s %d %d %d",
          who.name, &who.midterm, &who.final, &who.homeworks)!=EOF){
  m = writeastudent(fdout, &who);
  printf("m=%d\n", m);
  n++;
 }
 printf("n=%d\n", n);
 fclose(fdin);
 fclose(fdout);
}
```

EXERCISE

1. Write a C program to find the maximum of three numbers using conditional operators
2. Write a C Program to sort an array in ascending order
3. Write a C Program to sort an array in descending order
4. Write a C Program to find sum of digits in a given number

5. Write a C Program to print square of all numbers 1 to 20 and print sum squares
6. Write a C Program to check if given number is present in an array or not
7. Write a C Program to find the position of given number in array
8. Write a C Program to print transpose of matrix
9. Write a C Program to print equivalent binary number of given decimal number
10. Write a C Program to print equivalent octal number of given decimal number
11. Write a C Program to print equivalent hex number of given decimal number
12. Write a C Program to draw a circle with radius 10
13. Write a C Program to calculate factorial of a given number using recursion
14. Write a C Program to draw star symbol at the center of the screen
15. Write a C Program to copy contents of text.dat file to txt2.data file
16. Write a C Program to draw following object
17. Write a C Program to print all numbers between 1 to n divisible by 7
18. Write a C Program to find sum of 1 + 2 + 3 + ... + n
19. Write a C Program to find sum of 2 + 4 + 6 + ... + n
20. Write a C Program to find sum of 7 + 14 + 21 + ... + n
21. Write a C Program to find sum of 1/1 + 1/2 + 1/3 + ... + 1/n
22. Write a C Program to print 15 terms of 1 . 2 , 4, 7, 11, 16, ...
23. Write a C Program to print even and odd number from an array
24. Write a C Program to read character from keyboard and display message whether
25. character is alphabet , digit or special symbol
26. Write a C Program to read a string and count number of vovels in it
27. Write a program for the addition of a 2*2 matrix?
28. Write a program to multiply two 3*3 matrix.
29. Write a program to pick up the largest number from any 5*5 matrix.
30. Write a program to find out the transpose of a matrix.
31. What is difference between Structure and Unions?
32. What are macros? what are its advantages and disadvantages?
33. What is a NULL Macro? What is the difference between a NULL Pointer and a NULL Macro?
34. What is the difference between Extern and Global variable?
35. What are the differences between structures and arrays?
36. What is a structure? Explain the advantages of using a structure.

Model Question Paper

Subject Name : Programming in C and Algorithms

Subject code : HDAH

Section A (2 x 12 Marks = 24 Marks)

Answer Any 2 Questions in about 500 Words

1. What are the different control statement available in C. Explain with examples.

2. Explain the procedure of Bubble Sort & Selection Sort.

3. Explain with examples on various file handling methods in C

Section B (2 x 7 Marks = 14 Marks)

Answer Any 2 of the following Questions in about 300 Words

4. What are identifier and keywords? Explain it with suitable example.

5. Explain the various storage classes in C.

6. *Explain the various asymptotic notations used in algorithm design.*

Section C (5 x 4 Marks = 20 Marks)

Answer Any 2 of the following Questions in about 200 Words

7(a) What are Operators? Mention their types in C.

7(b) Explain - a) printf() b) scanf()

7(c) What is recursion explain with suitable example.

7(d) Write a program to adding of n integers

7(e) What is structure?explain with suitable example

7(f) Explain the procedure of Binary Search

7(g) Explain dynamic memory allocation?

Section C (6 x 2 Marks = 12 Marks)

Answer All Questions in about 100 Words

8(a) Write a C program to Generate the Fibanocci Series

8(b) Explain with example ++i and i++.

8(c) What is use of continue in C.

8(d) State two advantages of function?

8(e) What is union? Explain with example.

8(f) *Define the3 Average-case efficiency" of an algorithm?*

Answers

Section A (2 x 12 Marks = 24 Marks)

1. ***What are the different control statement available in C. Explain with examples.***

Control statements are of two types – branching and looping.

BRANCHING

LOOPING

BRANCHING

1. if else statement

It is used to carry out one of the two possible actions depending on the outcome of a logical test. The else portion is optional.

The syntax is

If (expression) statement1 [if there is no else part]

Or

if (expression)

Statement 1

else

Statement 2

Here expression is a logical expression enclosed in parenthesis. If expression is true, statement 1 or statement 2 is a group of statements, they are written as a block using the braces { }

Example:

```
1. if(x<0) printf("\n x is negative");

2. if(x<0)
printf("\n x is negative");
else
printf("\n x is non negative");

3.if(x<0)
{
x=-x;
s=sqrt(x);
}
else
s=sqrt(x);
```

2. nested if statement

Within an if block or else block another if – else statement can come. Such statements are called nested if statements.

The syntax is

If (e1)

s1

if (e2)

s2

else

s3

else

s4

3. Ladder if statement

Inorder to create a situation in which one of several courses of action is executed we use ladder – if statements.

The syntax is

If (e1) s1

else if (e2) s2

else if (e3) s3

....

else sn

Example: if(mark>=90) printf("\n excellent");

else if(mark>=80) printf("\n very good");

else if(mark>=70) printf("\n good");

else if(mark>=60) printf("\n average");

else

printf("\n to be improved");

SWITCH STATEMENT

It is used to execute a particular group of statements to be chosen from several available options. The selection is based on the current value of an expression with the switch statement.

The syntax is:

```
switch(expression)
{
case value1:
s1
break;
case value 2:
s2
break;
.......
........
default:
sn
}
```

All the option are embedded in the two braces { }.Within the block each group is written after the label case followed by the value of the expression and a colon. Each group ends with '***break***' statement. The last may be labelled ***'default'***. This is to avoid error and to execute the group of statements in default if the value of the expression does not match value1, value2,........

LOOPING

1. The while statement

This is to carry out a set of statements to be executed repeatedly until some condition is satisfied.

The syntax is:

While (expression) statement

The statement is executed so long as the expression is true. Statement can be simple or compound.

Example

```
#include<stdio.h>
main()
{
int i=1;
while(x<=10)
{
```

```
printf("%d",i);
++i;
}
}
```

2. *do while statement*

This is also to carry out a set of statements to be executed repeatedly so long as a condition is true.

The syntax is:

do statement while(expression)

Example:

```
#include<stdio.h>
main()
{
int i=1;
do
{
printf("%d",i);
++i;
}while(i<=10);
}
```

THE DIFFERENCE BETWEEN while loop AND do – while loop

1) In the while loop the condition is tested in the beginning whereas in the other case it is done at the end.

2) In while loop the statements in the loop are executed only if the condition is true. whereas in do – while loop even if the condition is not true the statements are executed atleast once.

3. for loop

It is the most commonly used looping statement in C. The general form is

For(expression1;expression2;expression3)statement

Here expression1 is to initialize some parameter that controls the loopingaction.expression2 is a condition and it must be true to carry out theaction.expression3 is a unary expression or an assignment expression.

Example:

```
#include<stdio.h>
main()
```

```
{
int i;
for(i=1;i<=10.++i)
printf("%d",i);
}
```

Here the program prints *i* starting from 1 to 10.First *i* is assigned the value 1 and than it checks whether *i*<=10 If so i is printed and then *i* is increased by one. It continues until *i*<=10. An example for finding the average of 10 numbers;

```
#include<stdio.h>
main()
{
int i;
float x,avg=0;
for(i=1;i<=10;++i)
{
scanf("%f",&x);
avg += x;
}
avg /= 10;
printf("\n average=%f",avg);
}
```

Note: Within a loop another for loop can come

Example :

```
for(i=1;i<=10;++i)
for(j=1;j<=10;++j);
```

The break statement

The break statement is used to terminate4 loop or to exit from a switch. It is used in for, while, do-while and switch statement.

The syntax is ***break;***

Example A program to read the sum of positive numbers only

```
#include<stdio.h>
```

10

```
main()
{
int x, sum=0;
int n=1;
while(n<=10)
{
scanf("%d",&x);
if(x<0) break;
sum+=x;
}
printf("%d",sum);
}
```

2. ***Explain the procedure of Bubble Sort & Selection Sort.***

BUBBLE SORT

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 means the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top (i.e. the beginning) of the list via the swaps. Because it only uses comparisons to operate on elements, it is a comparison sort. This is the easiest comparison sort to implement.

Bubble sort is a straightforward and simplistic method of sorting data that is used in computer science education. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. Although simple, this algorithm is highly inefficient and is rarely used except in education. A slightly better variant, cocktail sort, works by inverting the ordering criteria and the pass direction on alternating passes.

A C-language program bubble sort.

```
#include <stdio.h>
void main()
{
    int x[10];
    int i,j;
   printf("\nEnter 10 elements\n");
/* accepting 10 elements from the user for sorting */
    for(i = 0; i<10 ; i++)
        scanf("%d",&x[i]);
    /* End of input */
    /* bubble sort */
    for(i=0;i<10;i++)
    {
        for(j=0;j<10-i;j++)
        {
/*Condition to handle i=0 & j = 9. j+1 tries to access x[10] which is not there in zero based array*/
            if(j+1 == 10)
              continue;
            if(x[j]>x[j+1])
            {
                temp = x[j];
                x[j]=x[j+1];
                x[j+1] = temp;
            }
        }
    }
    /* printing sorted array */
    for(i=0; i<10;i++)
        printf("\n %d",x[i]);
    /*end of program */
}
```

The bubble sort works by iterating down an array to be sorted from the first element to the last, comparing each pair of elements and switching their positions if necessary. This process is repeated as many times as necessary, until the array is sorted. Since the worst case scenario is that the array is in reverse order, and that the first element in sorted array is the last element in the starting array, the most exchanges that will be necessary is equal to the length of the array.

Here is a simple example:

Given an array 23154 a bubble sort would lead to the following sequence of partially sorted arrays: 21354, 21345, 12345. First the 1 and 3 would be compared and switched, then the 4 and 5. On the next pass, the 1 and 2 would switch, and the array would be in order.

We compute that the order of the outer loop (for(int x = 0; ..)) is O(n); then, we compute that the order of the inner loop is roughly O(n). Note that even though its efficiency varies based on the value of x, the average efficiency is n/2, and we ignore the constant, so it's O(n). After multiplying together the order of the outer and the inner loop, we have O(n^2).

SELECTION SORT

Selection sort is a simple sorting algorithm that improves on the performance of bubble sort. It works by first finding the smallest element using a linear scan and swapping it into the first position in the list, then finding the second smallest element by scanning the remaining elements, and so on. Selection sort is unique compared to almost any other algorithm in that its running time is not affected by the prior ordering of the list, it performs the same number of operations because of its simple structure. Selection sort also requires only *n* swaps, and hence just È(*n*) memory writes, which is optimal for any sorting algorithm. Thus it can be very attractive if writes are the most expensive operation, but otherwise selection sort will usually be outperformed by insertion sort or the more complicated algorithms.

```
#include <stdlib.h>

#include <stdio.h>

 #define uint32 unsigned int

typedef int (*CMPFUN)(int, int);

 void ArraySort(int This[], CMPFUN fun_ptr, uint32 the_len)

{

  /* selection sort */

  uint32 indx;

  uint32 indx2;

  uint32 large_pos;

  int temp;

  int large;

  if (the_len <= 1)

    return;

  for (indx = the_len - 1; indx > 0; —indx)

  {

   /* find the largest number, then put it at the end of the array */
```

```
    large = This[0];
    large_pos = 0;
   for (indx2 = 1; indx2 <= indx; ++indx2)
   {
     temp = This[indx2];
     if ((*fun_ptr)(temp ,large) > 0)
     {
       large = temp;
       large_pos = indx2;
     }
   }
   This[large_pos] = This[indx];
   This[indx] = large;
  }
}
#define ARRAY_SIZE 14
int my_array[ARRAY_SIZE];
void fill_array()
{
 int indx;
 for (indx=0; indx < ARRAY_SIZE; ++indx)
 {
  my_array[indx] = rand();
 }
 /* my_array[ARRAY_SIZE - 1] = ARRAY_SIZE / 3; */
}
int cmpfun(int a, int b)
{
 if (a > b)
   return 1;
  else if (a < b)
   return -1;
  else
   return 0;
}
int main()
{
 int indx;
 int indx2;
 for (indx2 = 0; indx2 < 80000; ++indx2)
 {
   fill_array();
  ArraySort(my_array, cmpfun, ARRAY_SIZE);
  for (indx=1; indx < ARRAY_SIZE; ++indx)
  {
   if (my_array[indx - 1] > my_array[indx])
    {
     printf("bad sort\n");
     return(1);
    }
  }
 }
 return(0);
}
```

3. Explain with examples on various file handling methods in C

Data Files are to store data on the memory device permanently and to access whenever is required.

There are two types of data files

1 Stream Oriented data files

2 System Oriented data files

Stream oriented data files are either text files or unformatted files. System oriented data files are more closely related to computer's operating system and more complicated to work with. In this session we go through stream oriented data files.

Opening and Closing data files

The first step is to create a buffer area where information is stored temporarily before passing to computer memory. It is done by writing

File *fp;

Here fp is the pointer variable to indicate the beginning of the buffer area and called stream pointer .

The next step is to open a data file specifying the type i.e. read only file, write only file, read /write file. This is done by using the library function fopen

The syntax is

fp=fopen(filename,filetype)

the filetype can be

1 'r' (to open an existing file for reading only)

2 'w' (to open a new file for writing only. If file with filename exists, it will be destroyed and a new file is created in its place)

3 'a' (to open an existing file for appending. If the file name does not exist a new file with that file name will be created)

4 'r+' (to open an existing file for both reading and writing)

5 'w+' (to open a new file for reading and writing. If the file exists with that name, it will be destroyed and a new one will be created with that name)

6 'a+' (to open an existing file for reading and writing. If the file does not exist a new file will be created).

For writing formatted data to a file we use the function fprintf.

The syntax is

Fprintf(fp,"conversion string", value);

For example to write the name "rajan" to the file named 'st.dat'

File *fp;

Fp=fopen("st.dat",'w');

Fprintf(fp,"%[^\n]","rajan");

The last step is to close the file after the desired manipulation. This is done by the library function fclose.

The syntax is

fclose(fp);

Example

1. To create a file of biodata of students with name 'st.dat'.

#include<stdio.h>

#include<string.h>

```
Tpedef struct
{
Int day;
Int month;
Int year;
}date;
Typedef Struct
{
char name(30);
char place(30);
int age;
date birthdate;
}biodata;
main()
{
File *fp;
biodata student;
fp=fopen("st.dat",'w');
Printf("Input data");
Scanf("%[^\n]",student.name);
Scanf("%[^\n]",student.place);
Scanf("%d",&student.age);
Scanf("%d",&student.birthdate.day);
Scanf("%d",&student.birthdate.month):
Scanf("%d",&student.birthdate.year);
Fprintf(fp,"%s%s%d%d%d%d",student.name,student.place,student.
age,student.birthdate.day, student.birthdate.month,
student.birthdate.year)
Fclose(fp);
}
```

Section B (2 x 7 Marks = 14 Marks)

4. ***What are identifier and keywords? Explain it with suitable example.***

IDENTIFIER AND KEYWORDS

(i) Identifier are names given to various program elements like variables, arrays and functions. The name should begin with a letter and other characters can be letters and digits and also can contain underscore character (_)Example: area, average, x12 , name_of_place etc.........

(ii) Keywords are reserved words in C language. They have predicted meanings and are used for the intended purpose. Standard keywords are auto,break, case, char, const, continue, default, do, double, else enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while. (Note that these words should not be used as identities.)

5. ***Explain the various storage classes in C.***

There are 4 different storage class specifications in C

- automatic,
- external,
- static, and
- register.

They are identified by the key words auto, external, static, and register respectively.

AUTOMATIC VARIABLES

They are declared in a function. It is local and its scope is restricted to that function. They are called so because such variables are created inside a function and destroyed automatically when the function is exited. Any variable declared in a function is interpreted as an automatic variable unless specified otherwise. So the keyword auto is not required at the beginning of each declaration.

EXTERNAL VARIABLE (GLOBAL VARIABLE)

The variables which are alive and active throughout the entire program are called external variables. It is not centered to a single function alone, but its scope extends to any function having its reference. The value of a global variable can be accessed in any program which uses it. For moving values forth and back between the functions, the variables and arrays are declared globally i.e., before the main program. The keyword *external* is not necessary for such declaration, but they should be mentioned before the main program.

STATIC VARIABLES

It is, like automatic variable, local to functions is which it is defined. Unlike automatic variables static variable retains values throughout the life of the program, i.e. if a function is exited and then re-entered at a later time the static variables defined within the function will retain their former values. Thus this feature of static variables allows functions to retain information permanently throughout the execution of the program. Static variable is declared by using the

keyword static.

Example :

```
 static float a ;
Static int x ;
```

Consider the function program:

```
# include<stdio.h>
long int Fibonacci (int count )
main()
{
int i, m=20;
for (i =1 ; i < m ; ++i)
printf( "%ld\t",fibonacci(i));
}
long int Fibonacci (int count )
{
static long int f1=1, f2=1 ;
long int f ;
f = (count < 3 ) ? 1 : f1 + f2 ;
f2 = f1
f1= f ;
return (f ) ;
}
```

6. *Explain the various asymptotic notations used in algorithm design.*

Asymptotic Notations and Basic Efficiency Classes

To choose the best algorithm we need to check the efficiency of each algorithm. Asymptotic notations describe different rate-of-growth relations between the defining function and the defined set of functions. The order of growth is not restricted by the asymptotic notations, and can also be expressed by basic efficiency classes having certain characteristics.

Let us now discuss asymptotic notations of algorithms

ASYMPTOTIC NOTATIONS

Asymptotic notation within the limit deals with the behavior of a function, i.e. for sufficiently large values of its parameter. While analyzing the run time of an algorithm, it is simpler for us to get an approximate formula for the run- time.

The main characteristic of this approach is that we can neglect constant factors and give importance to the terms that are present in the expression (for T(n)) dominating the functions behavior whenever n becomes large. This allows dividing of un-time functions into broad efficiency classes.

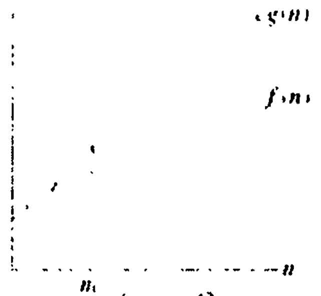
To give time complexity as "fastest possible", "slowest possible" or "average time`, asymptotic notations are used in algorithms. Various notations such as &! (omega), T (theta), O (big o) are known as asymptotic notations.

Big Oh notation (O)

"O is the representation for big oh notation. It is the method of denoting the upper bound of the running time of an algorithm. Big Oh notation helps in calculating the longest amount of time taken for the completion of algorithm.

O(g)={f / f is a non negative function ,there exists constants c2 & n0 such that f(x) d" c2.g(n), ne"n0}

The graph of C h(n) and T(n) can be seen in the figure,As n becomes larger, the running time increases considerably. For example, consider $T(n)=13n^3+42n^2+2n\log n+4n$. Here as the value on n increases n^3 is much larger than n^2, nlogn and n. Hence it dominates the function T(n) and we can consider the running time to grow by the order of n3. Therefore it can be written as $T(n)=O(n^3)$. The value of n for T(n) and C h(n) will not be less than n0.Therefore values less than n0 are considered as not relevant.



f(n) = O(g(n))

Big Oh Notation T(n) • O(h(n))

Example:

1. f(n)= $10n^3+5n^2+17$

$10n^3$d" f(x)

$10n^3$d" f(x)

C1=10

C1.n^3d" f(x) for all n e" 1 =n0

F belongs to the class n^3

2. f(n)= $2n^3+3n+79$

$2n^3$d" f(x)

$2n^3$d" f(x)

C1=2 & c2=84

C1.n^3d" f(x) for all n e" 1 =n0

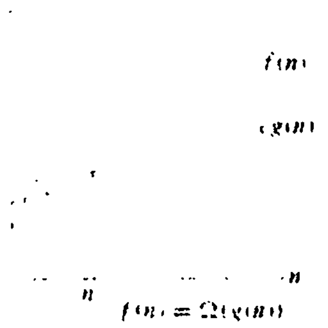
F belongs to the class n^3

Omega notation (&!)

"&! is the representation for omega notation. Omega notation represents the lower bound of the running time of an algorithm. This notation denotes the shortest amount of time that an algorithm takes.

&!(g)={f / f is a non negative function ,there exists constants c1 & n0 such that c1.g(n) d" f(x), ne"n0}

The graph of C h(n) and T(n) can be seen in the figure.



Example:

1. f(n)= $10n^3+5n^2+17$

    f(x) d" (10+5+7) n^3

    f(x) d" (32) n^3

    c2=32

    f(x) d" c2.n^3 for all n e" 1 =n0

    F belongs to the class n^3

2. f(n)= $2n^3+3n+79$

    f(x) d" (2+3+79) n^3

    f(x) d" (84) n^3

    c2=84

    f(x) d" c2.n^3 for all n e" 1 =n0

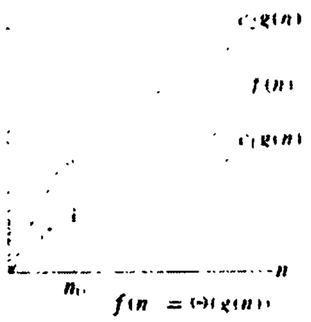
    F belongs to the class n^3

Theta notation (r•)

The depiction for theta notation is "T. This notation depicts the running time between the upper bound and lower bound.

O(g)={f / f is a non negative function ,there exists constants c1,c2 & n0 such that c1.g(n) d" f(x) d" c2.g(n), ne"n0}

The graph of C1 h(n), C2 h(n) and T(n) can be seen in the figure.



Example:

1. f(n)= $10n^3+5n^2+17$

   $10n^3$d” f(x) d” (10+5+7) n^3

   $10n^3$d” f(x) d” (32) n^3

   C1=10 & c2=32

   C1.n^3d” f(x) d” c2.n^3 for all n e” 1 =n0

   F belongs to the class n^3

2. f(n)= $2n^3+3n+79$

   $2n^3$d” f(x) d” (2+3+79) n^3

   $2n^3$d” f(x) d” (84) n^3

   C1=2 & c2=84

   C1.n^3d” f(x) d” c2.n^3 for all n e” 1 =n0

   F belongs to the class n^3

3. a(n) = “$a_i n^i$,i=0,1,...,k a_k>o

   a belongs to the class n^k

4. x(n) =5nlogn + n

   x belongs to the class nlogn

5. x(n) = 2 + 1/n

   x belongs to the class 1

Section C (5 x 4 Marks = 20 Marks)

7(a) What are Operators? Mention their types in C.

Integer division :

Division of an integer quantity by another is referred to integer division. This operation results in truncation. i.e. When applied to integers, the division operator /discards any remainder, so 1 / 2 is 0 and 7 / 4 is 1. But when either operand is a floating-point quantity (type float or double), the division operator yields a floating-point result, with a potentially nonzero fractional part. So 1 / 2.0 is 0.5,and 7.0 / 4.0 is 1.75.

Example : int a, b, c;

a=5;

b=2;

c=a/b;

Here the value of c will be 2

Actual value will be resulted only if a or b or a and b are declared floating type. The value of an arithmetic expression can be converted to different data type by the statement (data type) expression.

Example : int a, b;

float c;a=5;b=2;

c=(float) a/b

Here c=2.5

UNARY OPERATORS

A operator acts up on a single operand to produce a new value is called a unary operator.

(1) the **decrement and increment** operators - ++ and — are unary operators. They increase and decrease the value by 1. if x=3 ++x produces 4 and –x produces 2.

Note : in the place of ++x , x++ can be used, but there is a slight variation. In both case x is incremented by 1, but in the latter case x is considered before increment.

(2) sizeof is another unary operator

int x, y;

y=sizeof(x);

The value of y is 2 . the *sizeof* an integer type data is 2 that of float is 4, that of double is 8, that of char is 1.

RELATIONAL AND LOGICAL OPERATORS

< (less than),

<= (less than or equal to),

> (greater than),

>= (greater than or equal to),

= = (equal to) and

!= (not equal to) are relational operators.

LOGICAL OERATORS

The relational operators work with arbitrary numbers and generate true/false values. You can also combine true/false values by using the *Boolean operators*, which take true/false values as operands and compute new true/false values.

The three Boolean operators are:

- && and
- || or
- ! not (takes one operand; "*unary*")

The && ("and") operator takes two true/false values and produces a true (1) result if both operands are true (that is, if the lefthand side is true **and** the right-hand side is true).

The || ("or")operator takes two true/false values and produces a true (1) result if either operand is true. The ! ("not") operator takes a single true/false value and negates it, turning false to true and true to false (0 to 1 and nonzero to 0).

&& (and) and || (or) are logical operators which are used to connect logical expressions.

Where as ! (not) is unary operator, acts on a single logical expression.

ASSIGNMENT OPERATORS

These operators are used for assigning a value of expression to another identifier.

=, + =, - = , * =, /= and %= are assignment operators.

a = b+c results in storing the value of b+c in 'a'.

a += 5 results in increasing the value of a by 5

a /= 3 results in storing the value a/3 in a and it is equivalent a=a/3

7(b) Explain - a) printf() b) scanf()

scanf (control string, list of arguments);

The control string consists of group of characters, each group beginning % sign and a conversion character indicating the data type of the data item. The conversion characters are c,d,e,f,o,s,u,x indicating the type resp. char decimal integer, floating point value in exponent form, floating point value with decimal point, octal integer, string, unsigned integer, hexadecimal integer. ie, "%s", "%d" etc are such group of characters.

An example of reading a data:

#include<stdio.h>

```
main( )
{
char name[30], line;
int x;
float y;
.........
..........
scanf("%s%d%f", name, &x, &y);
scanf("%c",line);
}
```

NOTE:

1) In the list of arguments, every argument is followed by & (ampersand symbol) except string variable.

2) s-type conversion applied to a string is terminated by a blank space character. So string having blank space like "Govt. Victoria College" cannot be read in this manner. For reading such a string constant we use the conversion string as "%[^\n]" in place of "%s".

Example:

```
char place[80];
................
scanf("%[^\n]", place);
.................
```

with these statements a line of text (until carriage return) can be input the variable 'place'.

printf function

This is the most commonly used function for outputting a data of any type.

The syntax is

printf(control string, list of arguments)

Here also control string consists of group of characters, each group having % symbol and conversion characters like c, d, o, f, x etc.

Example:

```
#include<stdio.h>
' main()
```

```
{
int x;
scanf("%d",&x);
x*=x;
printf("The square of the number is %d",x);
}
```

Note that in this list of arguments the variable names are without &symbol unlike in The case of scanf( ) function. In the conversion string one can include the message to be displayed. In the above example "The square of the number is" is displayed and is followed by the value of x. For writing a line of text (which include blank spaces)the conversion string is "%s" unlike in scanf function. (There it is "[^\n]").

More about printf statement

There are quite a number of format specifiers for printf. Here are the basic ones :

%d	print an int argument in decimal
%ld	print a long int argument in decimal
%c	print a character
%s	print a string
%f	print a float or double argument
%e	same as %f, but use exponential notation
%g	use %e or %f, whichever is better
%o	print an int argument in octal (base 8)
%x	print an int argument in hexadecimal (base 16)
%%	print a single %

7(c) What is recursion explain with suitable example.

It is the process of calling a function by itself, until some specified condition is satisfied. It is used for repetitive computation (like finding factorial of a number) in which each action is stated in term of previous result

Example:

```
#include<stdio.h>
long int factorial(int n);
main( )
{
int n;
long int m;
scanf("%d",&n);
m=factorial(n);
printf("\n factorial is : %d", m);
}
long int factorial(int n)
{
if (n<=1)
return(1);
else
return(n*factorial(n-1));
}
```

In the program when n is passed the function, it repeatedly executes calling the same function for n, n-1, n-2,...................1.

7(d) Write a program to adding of n integers

```
#include <stdio.h>


int main(void) {
 int n;     /* The number of numbers to be read */
 int sum;    /* The sum of numbers already read  */
 int current; /* The number just read           */
 int lcv;    /* Loop control variable, it counts the number
               of numbers already read */
 printf("Enter a positive number n > ");
 scanf("%d",&n); /* We should check that n is really positive*/
 sum = 0;
 for (lcv=0; lcv < n; lcv++) {
  printf("\nEnter an integer > ");
  scanf("%d",&current);
  /*  printf("\nThe number was %d\n", current); */
  sum = sum + current;
 }
 printf("The sum is %d\n", sum);
 return 0;
}
```

7(e) What is structure?explain with suitable example

We know an array is used to store a collection of data of the same type. But if we want to deal with a collection of data of various type such as integer, string, float etc we use structures in C language. It is a method of packing data of different types. It is a convenient tool for handling logically related data items of bio-data people comprising of name, place, date etc., salary details of staff comprising of name, pay da, hra etc.

Defining a structure.

In general it is defined with the syntax name **struct** as follows

```
Struct structure_name

{

Data type variable1;

Data type variable2;

...

}
```

For example

```
1 Struct account

{

Int accountno

Char name[50];

Float balance;

}customer[20]
```

7(f) Explain the procedure of Binary Search

BINARY SEARCH

Binary search is an efficient searching method. While searching the elements using this method the most essential thing is that the elements in the array should be sorted one.

Binary search algorithm 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 largest value(key), 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 approach iteratively, it narrows the search by a factor of two each time, and finds the target value.

Algorithm Bin search(a,n,x)

```
// Given an array a[1:n] of elements in non-decreasing

//order, n>=0,determine whether 'x' is present and
```

```
// if so, return 'j' such that x=a[j]; else return 0.
{
low:=1; high:=n;
while (low<=high) do
{
      mid:=[(low+high)/2];
      if (x<a[mid]) then high;
      else if(x>a[mid]) then
low=mid+1;
    else return mid;
  }
  return 0;
}
```

7(g) Explain dynamic memory allocation?

Usually when we use an array in c program, its dimension should be more than enough or may not be sufficient. To avoid this drawback we allocate the proper (sufficient) dimensions of an array during the run time of the program with the help of the library functions called memory management functions like 'malloc','calloc', 'realloc' etc. The process of allocating memory at run time is known as dynamic memory allocation.

Example:

To assign sufficient memory for x we use the following statement

x= (int *) malloc (n* sizeof (int)) , for in the place of initial declaration int x[n]

Similarly in the place of float y [100] we use y = (float *) malloc (m* sizeof

(float));

Section D (6 x 2 Marks = 12 Marks)

8(a) Write a C program to Generate the Fibanocci Series

```
#include <stdio.h>
int main(void) {
  int n;     /* The number of fibonacci numbers we will print */
  int i;     /* The index of fibonacci number to be printed next */
  int current; /* The value of the (i)th fibonacci number */
  int next;   /* The value of the (i+1)th fibonacci number */
  int twoaway; /* The value of tl.e (i+2)th fibonacci number */
  printf("How many Fibonacci numbers do you want to compute? ");
  scanf("%d", &n);
  if (n<=0)
   printf("The number should be positive.\n");
  else {
  printf("\n\n\tI \t Fibonacci(I) \n\t=====================\n");
   next = current = 1;
   for (i=1; i<=n; i++) {
     printf("\t%d \t  %d\n", i, current);
     twoaway = current+next;
     current = next;
     next   = twoaway;
   }
  }
}
```

8(b) Explain with example ++i and i+ CONSTANTS

There are 4 basic types of constants. They are integer constants, floating-point constants, character constants and string constants.

(a) **integer constants**: It is an integer valued numbers, written in three different number system, decimal (base 10) , octal(base8), and hexadecimal(base 16).

A decimal integer constant consists of 0,1,.....,9..

Example : 75 6,0,32, etc.....

5,784, 39,98, 2-5, 09 etc are **not** integer constants.

An octal integer constant consists of digits 0,1,...,7. with 1st digit 0 to indicate that it is an octal integer.

Example : 0, 01, 0756, 032, etc.....

32, 083, 07.6 etc..... are **not** valid octal integers.

A hexadecimal integer constant consists of 0,1, ...,9,A, B, C, D, E, F. It begins with 0x.

Example: 0x7AA2, 0xAB, etc......

0x8.3, 0AF2, 0xG etc are **not** valid hexadecimal constants.

Usually negative integer constant begin with (-) sign. An unsigned integer constant is identified by appending U to the end of the constant like 673U, 098U, 0xACLFUetc. Note that 1234560789LU is an unsigned integer constant.

(b) floating point constants : It is a decimal number (ie: base 10) with a decimal point or an exponent or both. Ex; 32.65, 0.654, 0.2E-3, 2.65E10 etc. These numbers have greater range than integer constants.

(c) character constants : It is a single character enclosed in single quotes like 'a'. '3', '?', 'A' etc. each character has an ASCII to identify. For example 'A' has the ASCII code 65, '3' has the code 51 and so on.

(d) escape sequences: An escape sequence is used to express non printing character like a new line, tab etc. it begin with the backslash (\ ) followed by letter like a, n, b, t, v, r, etc. the commonly used escape sequence are

\a : for alert	\n : new line	\0 : null
\b : backspace	\f : form feed	\? : question mark
\f : horizontal tab	\r : carriage return	\' : single quote
\v : vertical tab	\" : quotation mark	

(e) string constants : it consists of any number of consecutive characters enclosed in double quotes .Ex : " C program" , "mathematics" etc......

VARIABLES

A variable is an identifier that is used to represent some specified type of information. Only a single data can be stored in a variable. The data stored in the variable is accessed by its name. before using a variable in a program, the data type it has to store is to be declared.

Example : int a, b, c,

a=3; b=4;

c=a+b;

8(c) What is use of continue in C.

It is used to bypass the remainder of the current pass through a loop. The loop does not terminate when continue statement is encountered, but statements after continue are skipped and proceeds to the next pass through the loop.

In the above example of summing up the non negative numbers when a negative value is input, it breaks and the execution of the loop ends. In case if we want to sum 10 nonnegative numbers, we can use *continue* instead of *break*

Example :
```
#include<stdio.h>
main()
{
int x, sum=0, n=0;
while(n<10)
{
scanf("%d",x);
if(x<0) continue;
sum+=x;
++n;
}
printf("%d",sum);
}
```

8(d) State two advantages of function?

Advantages of functions

1. It appeared in the main program several times, such that by making it a function, it can be written just once, and the several places where it used to appear can be replaced with calls to the new function.

2. The main program was getting too big, so it could be made (presumably) smaller and more manageable by lopping part of it off and making it a function.

3. It does just one well-defined task, and does it well.

4. Its interface to the rest of the program is clean and narrow

5. Compilation of the program can be made easier.

8(e) What is union? Explain with example.

Union is a concept similar to a structure with the major difference in terms of storage. In the case of structures each member has its own storage location, but a union may contain many members of different types but can handle only one at a time. Union is also defined as a structure is done but using the syntax union.

```
Union var
{
Int m;
Char c;
Float a;
}
```

8(f) Define the3 Average-case efficiency" of an algorithm?

The Average complexity gives information about an algorithm on specific input. For instance in the sequential search algorithm:

Let the probabiliy of getting successful search be P.

The total number of elements in the list is n.

If we find the first search element at the ith position then the probability of occurrence of the first search element is P/n for every ith element.

(1 - P) is the probability of getting unsuccessful search.

Therefore the average case time complexity Cavg(n) is given as:

Cavg(n) = Probability of successful search + Probability of unsuccessful search

$$\text{Cavg}(n) = \frac{P(1+n)}{2} + n(1 - P)$$