5 FUNCTIONS IN C

OBJECTIVES

In this chapter we shall introduce user-defined functions and learn the art of modular programming using functions. We also introduce the concept of recursion. We learn to do statistical computations for a frequency table and generating a calendar.

CONCEPT OF A FUNCTION

A function encapsulates some computation to enable the modular development. The function can be used repeatedly. We have seen the Gcd program. We write the same code as the 'Gcd function'.We compute the LCM of a and b. LCM = (a*b) / GCD.

```
int gcd(int a,int b)  /* gcd function definition */
{int r;          /* r is  local variable*/

 while ( a!= 0) /* a & b are parameters */
   {
     r = b % a;
     b = a;/* in main original value of a & b retained*/
     a = r;
    }

     return (abs(b));
 }

main()      /* By Anil Pedgaonkar        */
{
  int x,y,a,b,g;
  clrscr();    /* gcd function call a = 4 b = 6 */
  printf("gcd of 6 & 4 = %d", gcd(6,4));
  printf("\n give x & y");
  scanf("%d%d",&x,&y);
  printf("\ngcd of x = %d & y = %d is %d",x, y,gcd(x,y));
  printf("\n give a & b");
  scanf("%d%d",&a,&b);

  g = gcd(a,b);/*values of a & b not altered in main*/
  printf("\n gcd of a =  %d & b =  %d is %d",a,b,g);
  printf("\n lcm = %d",abs(a*b/g));
}
```

Output:
User responses are underlined

```
gcd of 6 & 4 = 2
 give x & y 3 5
gcd of x = 3 & y = 5 is 1
 give a & b 36 10
 gcd of a =  36 & b =  10 is 2
 lcm = 180
```

Characteristics of a function:

- The program *execution always begins with* the function *main*. The main can call other functions repeatedly. The gcd function is called thrice.
- Each function definition has the following structure.
- return-type Function-name(list of arguments separated by comma)
  {
  Body
  }
- The Body of the function is a compound statement and usually includes the return statement**. No function definition can be nested inside another function.** We observe that the main and the gcd function form separate blocks. One function can call other functions irrespective of the physical order in which they are defined. However the definition of the called function must appear ahead of the calling function else we need to have the declaration statement for the called function in the body of the calling function.
- At the function call, control is transferred to the called function. When the return statement is encountered in the called function, the control is transferred back to the calling function. The called function may return a single value to the calling function. A function may not return a value to the caller. The calling function may ignore the returned value. The gcd function returns a value.
- The arguments are called as parameters. These are dummy symbols. The same names may not be used in the call or the declaration, but the data types must be the same. The call to the gcd function is made with the arguments 6,4 and x, y also.
- **The call is by value**. The called function can not directly alter a variable in the calling function even though the variable is passed as a parameter in the call. The call gcd (a, b) is made. The variables a and b are 36 & 10 in main but their values are unaltered in main even though they are changed in the gcd function as seen from the output. The same is the story with the call gcd (x, y).

GCD OF A SET OF NUMBERS

Example: Find the GCD of the following set of numbers:

```
{ 80, 60, 500, 80, 305, 110, 45, 300 }
```

We use the function *gcd()*. The given set of numbers is stored in an array. We use register storage type for the variable *r* defined in the function *gcd()*. This storage type is used for efficiency. The variables are stored in the registers. Registers are memory locations in the CPU. Only *int* and *char* types can be assigned the storage class *register*. We cannot take the address of register variables. So the address operator & cannot be applied to register storage class. For this reason, the variables, a and b in the function main() cannot be assigned register class.

```
int gcd(a,b) /* A PROGRAM TO FIND GCD OF A SET OF NUMBERS */

{   register int r;

     while ( a != 0)
          r = b%a, b = a, a = r;

     return (abs(b));
}

main()
{
     int num[] = {80,60,500,80,305,110,45,300};
     int g = 0, n = size, k;

     clrscr();

     for ( k =  0 ; k < n ; ++k)
          g = gcd(g, num[k]);

     printf("gcd =  %d", g);
     getch();
}
```

Output:

```
gcd =  5
```

STORAGE CLASSES IN C

Each identifier (variable) has a storage class, as well as type associated with it. The type tells about the nature of the value of the data item stored whereas the storage class specifies the initial value, the life, the storage location and the scope of the variable (the functions which can access the variable.) There are 4 storage classes. They are automatic, register, extern and

static. We show the difference between automatic (local) and extern (global) variables.

```
            /*EXPLAIN CALL BY VALUE*/

int a = 2,b= 3;             /*GLOBAL variables*/

int swap(int x,int y)
{int t= 0;     /*local auto variable in function swap */

 t = x; x= y; y = t;  /*This t different from t in main*/

 printf("\nx & y IN SWAP %d %d\n",x,y);
 t = a; a= b; b = t;  /*x & y Exchanged In      swap*/
 return (t);
}   /* function swap over. Returned value ignored in main */

main()
{int x = 9, y = 1, t = 5;  /* local variables in main*/
clrscr();

printf("STARTING FROM  MAIN x = %d y = %d\n",x,y);
printf("t IN MAIN = %d, GLOBAL a = %d b =%d",t,a,b);

swap(x,y);          /*call by value t in main unchanged */

printf("MAIN x = %d y = %d",x,y); /*x,y unchanged in main */

printf("\nt = %d a = %d b = %d",t,a,b); /*a & b global
variables  hence exchanged*/
}
```

Output:

```
STARTING FROM  MAIN x = 9 y = 1
t IN MAIN = 5, GLOBAL a = 2 b =3
x & y IN SWAP 1 9
MAIN x = 9 y = 1
t = 5 a = 3 b = 2
```

Analysis: The swap function is called from main. It exchanges the values of its parameters x & y, passed by main. They are exchanged in swap but when we return to main the original x & y retain their values. In the call x & y values are copied.

The variables a & b are defined outside any function. They are global and can be accessed by

any function like swap that is defined after their declaration. They are changed in swap and the change is reflected in a & b outside swap in function main as well .
There are two variables t. The t in main is distinct from t in swap. Both are local in the respective functions. The swap can not access the "t in main" unless it is passed as parameter. The local variables are private. They are hidden from other functions and when they are passed in the call, their values are copied onto the parameters.

Difference in Call by Value and Call by Reference :- We present another program**.** A function swap is used. It is defined ahead of main . Hence function declaration is unnecessary**.** We note that the elements of the array x [ ] declared in main () are passed by value as x[0] and x [1] to swap Their values are not exchanged
In this program another function "xchg" takes an int array as parameter. Since it returns nothing the keyword void is used in its definition.
The definition of xchg follows that of the main and hence it is **must** to use the declaration of the function xchg in main. It is declared as void xchg (int d [ ]); We could have written it as void xchg (int [ ]). .
But the array b declared in main () is passed by reference in the call as xchg (b); since the name of the array is the address of the array. So it's elements b [0] and b[1] are exchanged.

```
void  swap(int x,int y)
{int t= 0;     /*local auto variable in function swap */
t = x; x = y; y = t;
return;
}

 main ()
{int x[] = {9,1}; static int b[] = {4,7};  /*LOCAL IN MAIN*/
 void xchg(int d[]);    /*xchg function declaration*/
 clrscr();
 xchg(b);  /*function call x by value, b by reference*/
 swap(x[0], x[1]); /* x[0] and x[1] remain same in main */
 printf("x[0] = %d x[1] = %d\n", x[0],x[1);
 printf("b[0]= %d b[1]= %d", b[0],b[1]);
} /* b[0] & b[1] Exchanged in main  */

void xchg(int d[]) /* Function definition takes int array*/
{int temp;
 temp = x; x = y; y = temp;
 temp= d[0]; d[0] = d[1]; d[1]= temp;
 return;
} /*xchg def over it retuns nothing, observe key word void*/
```

```
Output:
x[0] = 9 X[1] = 1
b[0]= 7 b[1]= 4
```

Static and global variables:

- A global (*extern)* variable is defined outside the body of any function and it can be accessed and altered by any function whose definition follows the definition of the extern variable.
- If the function definition appears ahead the definition of the extern variable then the function body must have an explicit declaration for the global variable.
- Suppose the global variable is defined as char a; its declaration will appear like char *extern char a;*
- On the other hand the automatic (local) variables are the variables which are declared within a function. The keyword auto in their declaration is optional and usually omitted.
- The local (automatic) variables are reinitialized each time the function call is made
- The function parameters are treated as local to the body of the called function.
- The static variables are also local to the body of the function .
- **The static variables retain their values across the function calls.**

VARIABLE TYPE	AUTOMATIC	REGISTER	STATIC	EXTERN
Location	Memory	CPU registers	Memory	Memory
Initial value	Garbage	Garbage	Zero.	Zero.
Scope	Local to the block (function)	Local to the block (function)	Local to the block (function)	Global (accessible to all the functions)
Life	Created anew on each entry to the block and cease to exist when the block is exited	Created anew on each entry to the block and cease to exist when the block is exited	The value persists after the block is exited and again the block is reentered	The life of the program
Definition and declaration	auto char c; OR char c;	Register int k;	static float sum;	float sum; at the start of the program and ahead of any function. If the definition is following the function then extern declaration is required in the function as extern float sum;

We note that the storage class register can be used only for int and char variables of auto class. We can not take the address of the register variables as they are stored in the CPU registers and not in RAM. The compiler may ignore our register declaration and store them in RAM

We present a program to illustrate the concept of static variables.

```
/*EXPLAIN LOCAL(AUTO), STATIC and GLOBAL VARIABLES*/

int t;                /*global variables default value 0*/
int fun()     /* The function def returns int takes
nothing*/
{
     return (-1);
}

void dummy()
{int z = 0;  /* auto variable local in dummy */
static int x =0; /*local to dummy & static default value 0*/
printf("z = %d x = %d,t = %d", ++z, ++x,++t);
return;
}             /* definition of dummy over*/


main()
{ int z = 3, x = 3;          /*local in main*/
  clrscr();         /*x & z of main hidden from dummy */

  printf("dummy call 1\n");
  dummy();

  t = fun(); ; z+= 2,x+=2 ;

  printf("\n dummy call 2\n");
  dummy();
}
```

Output:

```
dummy call 1
z = 1 x = 1,t = 1
dummy call 2
z = 1 x = 2,t = 0
```

Analysis of the program:

We have two functions, *fun* and *dummy*.

The *fun* does not take any argument but returns an int.

Here, t is a global (extern) variable, x is a static variable in dummy, and z is a local (auto) variable in dummy.

There are another variables, x and z defined as local variables in main.

Both main and dummy have no access to the x, and z variables of the other function unless they are passed as arguments in the function call, and can not alter the values of the x and y of the other function in their body.

The global variable t can be accessed and altered by any function. The static z and the extern t are initialized to 0 by default.

At the second call of dummy the z in dummy is again 1 as z is reinitialized to 0 when the body of dummy is reentered.

But the static variable x retains its value 1 after the first call and thus x is 2 in the call.

The extern t is changed in the main via the function call fun ( ); and thus we have the value of t as 0.

We note that it is possible that a local and the global variable have same name.

In such case within the body of the function the local variable will be considered.

One must avoid global and static variables as far as possible since they violate the principles of information hiding.

Function Declarations

We have not used function declarations as we have placed function definitions ahead of the main. If the function definition follows the definition of the calling function then the calling function *must have a function declaration for the called function in it's body. Even otherwise, it is a good practice to always include declarations of all the functions in main. We have not done only to reduce the length of the program text, looking at the psychology of the student.*

Examples of declarations:

- Declare a function accepting one char one int and returning a char.
  char f (char a, int b); or char f(char, int);
- Declare a function accepting one array of int, one float and returning nothing.
  void f (int a[], float x); or void (int[], float);
- Declare a function returning an int and accepting nothing.
  int f( );
- Declare a function accepting nothing and returning nothing.
  void f( );
  The keyword void stands for a non-existent type.
- Declare a function accepting an array of chars and returning a char.
  char f(char x[]); or char f(char []);
- Declare a function accepting nothing and returning a float
  float f( );

- We note that a function can not return an array as function returns only single value.

THE SORTING PROGRAM USING FUNCTIONS

The functions encourage modular development and their use is highly recommended. We reorganize our sorting program using three functions, input sort and print to take care of three different jobs. The input function asks the user to enter the exact size of array, stores it in the variable n and accepts input for the elements. The array a and n are passed to sort and then to the print function.

```
#define size 10000
int input(int a[]) /* function to input numbers in an array
*/
{    int n, i;
     clrscr();
     printf("\n how many numbers to sort?");
     scanf("%d", &n);

     for ( i  = 0; i < n; ++i)
     {
          printf("give a[%d]",i);
          scanf("%d", &a[i]);
     }
 return(n);
}

void print(int a[], int n)
 /* function  to print the array */
{    int i;
     clrscr();

     for ( i = 0  ; i < n;  ++i)
     {
          if (i%10 == 0)/* 10 numbers are printed per line
*/
               printf("\n");

          printf(" %2d", a[i]); /* 10 numbers per line */
     }
 return;
}
```

```
void sort(int a[], int n)  /* function to sort */
{    int i, j , t;

     for( i= 0; i < n-1 ; i++)

          {for(j =0; j < n; j++)

               { if ( a[i] > a[j])

                     { t =a[i]; a[i] = a[j]; a[j] =t;
                     }
                }
           }

return;
}

main()
{
     int a[size], n;
     n = input(a);
     sort(a, n);
     print(a, n);
     getch();
}
```

RECURSION

It is possible for a function to call itself. This is called recursion.

Sum of first n numbers using recursion

The sum can be defined as sum(n) = sum(n-1) + n , sum(1) = 1. We have the program.
Thhe function sum declared in main callss sum function again.

```
int sum(int n)   /*DEFINITION OF SUM FUNCTION */
{    int s;

     if ( n > 1)
          s = sum(n -1) + n;
     else
          s = 1;

     return(s);
}
```

The GCD program using recursion

We note that gcd(a,b) = gcd(b%a,a) , gcd(0,b) = b. We show the program and it's output.

```
int gcd(int a, int b) /*recursive program for gcd */
{
     if  ( a != 0)
     {
          printf("\n gcd of %d & %d", b%a,a);
          return gcd(b%a,a);
     }

      else
                return b;
      return;
}

main()
{
     int a,b;
     clrscr();
     printf("\n give 2 positive nos");
     scanf("%d%d", &a, &b);
     printf("\n Hence  gcd of %d & %d = %d", a,b,gcd(a,b));

}
```

Output:

```
give 2 positive nos
gcd of 10 & 36
gcd of 6 & 10
gcd of 4 & 6
gcd of 2 & 4
gcd of 0 & 2
Hence  gcd of 36 & 10 = 2
```

Computationally Inefficient use of Recursion

Though Recursion is a powerful tool the recursive program if poorly designed can be computationally inefficient involving many redundant and repetitive computations. Algorithms like sum of first n numbers, GCD or the product of first n numbers are inherently more suited to iteration using While or for Loop. We note that it is computationally inefficient to write

the Fibonacci number generation program recursively in the C language because the numbers prev and next are computed twice, though many books ask to write the program recursively
However there are some problems for which recursion is the natural way in which algorithm can be formulated. To program these algorithms non recursively necessarily involve building a stack which is a data structure discussed at a later stage in the book. There are languages like LISP, Prolog which use recursion rather than iteration as a fundamental programming construct. In older languages like Basic , Fortran recursion was not allowed but as we shall see it is a natural way and in some sense the analogue of Principal of Mathematical Induction which reader might have encountered in high School. Recurisve solutions are often elegant and shorter.

TOWERS OF HANOI PROGRAM

The Towers of Hanoi problem is very much suited for recursion.
We are given n disks numbered from 1 to N placed on tower A. The disks are arranged such that disk 1 is smaller in size than disk 2, disk 2 is smaller in size than disk 3 … and so on. They are placed on tower A in ascending order. We are given two towers B and C. The problem is to transfer the N disks from tower A to tower C using the spare tower B while preserving the order, subject to the following two conditions.

Condition One:- You should move only one disk at a time.
Condition Two :- You can not keep a bigger disk on a smaller disk.

The problem admits a simple recursive situation for n disks if we assume that N - 1 disks are already moved using the following three steps.

1. Move N - 1 disks from Tower A to Tower B using Tower C as spare.
2. Move disk N from Tower A To C
3. Move N – 1 disks from Tower B to Tower Using tower A Spare.

This is an example of the phenomenon of exponential computational complexity.
To move n disks we need to call the function to move N-1 disks twice. . So if O(n) is the number of moves then we have O(n 0 = 2O(n-1) + 1
For n =1 we require a single move. For n = 2 we require 3 moves and for n = 3 we require 7 moves as can be easily checked using the above algorithm. So for n disks the number is 2^n -1
We can use Mathematical induction to deduce this.
O(n) = 2O(n-1) + 1 = 2.(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n -1
It can be checked that number of moves for 10 disks are 1023 , and so on. It increases rapidly.
We write a recursive program to solve this puzzle. The reader should try to write a nonrecursive program and then will appreciate the power of recursion.

```
unsigned long count;

void hanoi(int n,   char s, char t,  char d)
{
   if ( n != 0)  /* Hanoi program by anil Pedgaonkar*/
    { hanoi(n-1, s, d, t);
          printf("\nMove disk %d from %c to %c number of
          moves  =     %lu" ,n, s, d, ++count);
      hanoi(n-1, t, s, d);
    }
}

main() /* Towers of Hanoi */
{   int n;   char s ='A', t = 'B', d ='C';
    clrscr();
    printf("\n give number of disks");
    scanf("%d", &n);

    hanoi(n,s,t,d);
    printf("\nHanoi tower by Anil Pedgaonkar”);
    printf(“  Number of moves = %lu",count);
}
```

Output:

```
give number of disks
move disk 1 from A to C number of moves = 1
move disk 2 from A to B number of moves = 2
move disk 1 from C to B number of moves = 3
move disk 3 from A to C number of moves = 4
move disk 1 from B to A number of moves = 5
move disk 2 from B to C number of moves = 6
move disk 1 from A to C number of moves = 7
Hanoi tower by Anil Pedgaonkar Number of moves = 7
```

THE DAY ON A GIVEN DATE PROGRAM

We present the program. The days are numbered from 0 to 6 with Sunday as 0. We need to compute the remainder modulo 7 and consider leap years. The number of leap years up to and including 2000 are 484 and can be easily computed using the formula in the program. We use the symbolic constant START to store the day on Jan 1,2000 and the function week to print the day on corresponding to the number. We first present the output for three dates.

```
give date  as dd/mm/yyyy 01/01/2010
Day programmed by Anil Pedgaonkar   Friday
give date  as dd/mm/yyyy 17/12/1994
Day programmed by Anil Pedgaonkar   Saturday
give date  as dd/mm/yyyy 26/01/2011
Day programmed by Anil Pedgaonkar   Wedensday
```

```
#define start 6   /* DAY ON 1 JAN 2000 */
#define leapcount   484    /* Number of leap years elapsed till year 2000 */
int week( unsigned long  num)
{          if  ( num < 0)          num   +=   7;
           printf("\nDay programmed by Anil Pedgaonkar   ");
           switch( num)
           {       case 0 : printf("Sunday");          break;
                   case 1 : printf("Monday");          break;
                   case 2 : printf("Tuesday");         break;
                   case 3 :  printf("Wedensday");     break;
                   case 4 : printf("Thursday");        break;
                   case 5 : printf("Friday");          break;
                   case 6 : printf("Saturday");         break;
           }
   return (num);
}

main()                    /* Day on a Date By Anil Pedgaonkar */
{  int  leap, d, m,   leapyears,    i,   num =start,  years;     unsigned long y;
   static int month[] = {0,31, 28, 31,30,31,30,31,31,30,31,30,31};     clrscr();
   printf("\n give date  as dd/mm/yyyy ");     scanf("%d/%d/%lu", &d,  &m,  &y);
   printf("%02d/%02d/%4lu",  d,  m,  y);

      leap  =  ( (y%4 ==  0  &&   y %100 !=  0 )  ||  y%400  ==  0); /*Test whether leap
year*/
      years  =   y >= 2000 ?   (y  - 2000) % 7 :  -  ( (2000 -  y) %  7  );
      num   +=   years;

       leapyears   =   ( y/4  -   y / 100  +  y / 400  -   leapcount);
       leapyears  =   leapyears  >  0 ?    leapyears  % 7  :   -  ( abs (  leapyears ) % 7);
       if ( y  ==  2000)      --num;     /* Dont add a day for 2000 unless Feb is over */
       num    +=    leapyears;

       for ( i  =  1;   i < m;    ++i)        num  +=   month[i];
       if(  leap  &&      y   < 2000  )    --num;   /* Going in the Past */
       if (leap   &&     m > 2 )            ++num;                    /* Add a day after Feb  */

       num    +=     (d  -   1 ) ;
       num     =     num  >=  0?      num % 7 :  -   ( abs  (num)  %  7)  ;
       num    =   week  (  num  );          /* Finds the day */
 }
```

QUESTIONS

Q I. Write function declarations (function prototypes) for the following functions:

(1) A function which takes one int, one array of chars and returns nothing.
(2) A function taking no arguments but returning a float.
(3) A function accepting an int and returning a character.
(4) A function accepting no argument and returning nothing.
(5) A function taking no arguments and returning nothing.
(6) A function taking a double array and returning a long integer.

Q. II State whether the following are true or false by giving reasons:

(1) A function must return a value.
(2) A function can return an array.
(3) The value returned by a function must be used.
(4) The value of a static variable can not be changed outside the function in which it is defined.
(5) One can use the same name for a global and automatic variable.
(6) A register variable is always stored in a register.
(7) A function can call itself recursively even if it does not return a value.
(8) A float variable can not have the storage class register.
(9) A static variable, which is not initialized, has garbage as its initial value.
(10) #define can be used to define a formula or one line function.
(11) The local variable can not be altered by the function if the function has an automatic variable declaration with the same name.
(12) Even if an automatic variable defined in the calling function is not passed to the called function one can use the called function can alter its value in the calling function by reassigning the value returned by the calling function.
(13) An array value can be altered in the called function ,even though the array is automatic when the array name is passed as the argument.

1.

Q. III Answer the following questions in brief:

(1) Which are the storage classes in C? Illustrate.
(2) Explain the phenomenon of persistence of values of static variables across the function cals by an example
(3) Rewrite the prime number program using the register storage clas for num, factor ,and prime.
(4) What is a recursive function? Analyze the reasons as to why the fibonacci number generation program should not be written recursively.

Q. IV Find the output of the following programs after correcting errors, if any:

(1)

```
int sum;
main()
{
        int k = 1;
        ++k;
        sum += k;
        --sum;
        printf("%d",sum);
}
```

(2)

```
char c;
void reverse (char k)
{
k = k+ 'z' – 'a';
}
main()
{
        char k = 'a';printf("%c",reverse(c));
        k = c;
        reverse(k);
        c = k;
        printf("%c",c);
}
```

(3)

```
float sum = -1;
int sum(int k)
{       int sum = 0;
        sum += k;
        return;
}

main()
{
        int k = sum;
        k = sum(k);
        printf("%d",sum(k);
}
```

(4)

```
void addone()
{
        static int a; int b = 0;
        ++a; ++b;
        printf("%d %d ",a,b);
        return;
}

main()
{
        adone();
        return;
}
```

(5)

```
int k = 0;
int add()
{
        int a;
        a += k;
        ++k;
        return(a);
}
```

(6)

```
char swap(char a, char b)
{
        char temp;
        temp = a,
        a = b,
        b= temp;
        return(temp);
}
```

```
main()
{
        int a;
        a = add();
        printf("%d",a);
        a = add();
        printf("%d",a);
}
```

(7)

```
char a = 'a', b = 'b';
main()
{
        char swap(char a, char b);
        swap(a,b);
        printf("%c %c",a,b);
}

char swap(char a, char b)
{
        char temp;
        temp = a; a = b; b= temp;
        return(temp);
}
```

(8)

```
int sum;
main()
{
        int sum(int);
        printf("%d",sum(sum);)
}

int sum (int k)
{
        k = (k = 3)? k : sum(++k);
         return(k);
}
```

(9)

```
void sum()
static int k;
{
        if k = 3
                printf("%d",k);
        else
                sum(k++);}
main()
{
        sum();
}
```

(10)

```
#include <stdio.h>
char word = "mat"
char swap(char a,char b)
{
        char t = a; a= b; b = t; return(a);
}

void xchg(char x[])
{
        char  a = x[0], b = x[1], t = a; a= b; b = t;
        return;
}

main()
{
```

```
main()
{
        char a = 'a',b = 'b';
        a = swap(a,b);
        printf("%c %c",a b);
}
```

```
        char name = "am"
        swap (name[0], name[1]);   puts(name);
        xchg (name); puts(name);
        swap(word[0],word[1]);
        puts(word);
}
```

PRACTICAL EXERCISES

1. Write functions to input an array of floating point numbers, to print the array, and to copy the array to another array.
2. Write a function to compute mean deviation, from m where m can be mean mode or median for a set of observations entered by user along with their frequencies.
3. Write a function to compute mean deviation, from m where m can be mean mode or median for a frequency table.
4. Write a recursive function to reverse a string.
5. Write a recursive function to print the future values and present values for a given amount for n years at the rate of interest r%.
6. Write a recursive program to sort an array.
7. Write a function to find the correlation coefficient.
8. Write a the tower of hanoi programs by describing the disks and towers graphically.

∴ ∴ ∴ ∴ ∴