Chapter 2

Recursion

Recursion is deceptively simple in statement but exceptionally complicated in implementation. Recursive procedures work fine in many problems. Many programmers prefer recursion through simpler alternatives are available. It is because recursion is elegant to use through it is costly in terms of time and space. But using it is one thing and getting involved with it is another.

In this unit we will look at "recursion" as a programmer who not only loves it but also wants to understand it! With a bit of involvement it is going to be an interesting reading for you.

2.1. Introduction to Recursion:

A function is recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself. For a computer language to be recursive, a function must be able to call itself.

For example, let us consider the function factr() shown below, which computers the factorial of an integer.

```
#include <stdio.h>
int factorial (int);
main()
{
        int num, fact;
        printf ("Enter a positive integer value: ");
        scanf ("%d", &num);
        fact = factorial (num);
        printf ("\n Factorial of %d =%5d\n", num, fact);
}

int factorial (int n)
{
        int result;
        if (n == 0)
                return (1);
        else
                result = n * factorial (n-1);

        return (result);
}
```

A non-recursive or iterative version for finding the factorial is as follows:

```
factorial (int n)
{
        int i, result = 1;
        if (n == 0)
```

```
        return (result);
    else
    {
        for (i=1; i<=n; i++)
            result = result * i;
    }
    return (result);
}
```

The operation of the non-recursive version is clear as it uses a loop starting at 1 and ending at the target value and progressively multiplies each number by the moving product.

When a function calls itself, new local variables and parameters are allocated storage on the stack and the function code is executed with these new variables from the start. A recursive call does not make a new copy of the function. Only the arguments and variables are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

When writing recursive functions, you must have a exit condition somewhere to force the function to return without the recursive call being executed. If you do not have an exit condition, the recursive function will recurse forever until you run out of stack space and indicate error about lack of memory, or stack overflow.

2.2. Differences between recursion and iteration:

- Both involve repetition.
- Both involve a termination test.
- Both can occur infinitely.

Iteration	Recursion
Iteration explicitly user a repetition structure.	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop continuation.	Recursion terminates when a base case is recognized.
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop so the extra memory assigned is omitted.	Recursion causes another copy of the function and hence a considerable memory space's occupied.
It reduces the processor's operating time.	It increases the processor's operating time.

2.3. Factorial of a given number:

The operation of recursive factorial function is as follows:
Start out with some natural number N (in our example, 5). The recursive definition is:

n = 0, 0 ! = 1	Base Case
n > 0, n ! = n * (n - 1) !	Recursive Case

Recursion Factorials:

5! =5 * 4! = 5 *___ = _____ factr(5) = 5 * factr(4) = __

4! = 4 *3! = 4 *___ = ___ factr(4) = 4 * factr(3) = __

3! = 3 * 2! = 3 * ___ = ___ factr(3) = 3 * factr(2) = __

2! = 2 * 1! = 2 * ___ = ___ factr(2) = 2 * factr(1) = __

1! = 1 * 0! = 1 * __ = __ factr(1) = 1 * factr(0) = __

0! = 1 factr(0) = __

5! = 5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 5*4*3*2*1*0! = 5*4*3*2*1*1 =120

We define 0! to equal 1, and we define factorial N (where N > 0), to be N * factorial (N-1). All recursive functions must have an exit condition, that is a state when it does not recurse upon itself. Our exit condition in this example is when N = 0.

Tracing of the flow of the factorial () function:

When the factorial function is first called with, say, N = 5, here is what happens:

FUNCTION:
Does N = 0? No
Function Return Value = 5 * factorial (4)

At this time, the function factorial is called again, with N = 4.

FUNCTION:
Does N = 0? No
Function Return Value = 4 * factorial (3)

At this time, the function factorial is called again, with N = 3.

FUNCTION:
Does N = 0? No
Function Return Value = 3 * factorial (2)

At this time, the function factorial is called again, with N = 2.

FUNCTION:
Does N = 0? No
Function Return Value = 2 * factorial (1)

At this time, the function factorial is called again, with N = 1.

FUNCTION:
Does N = 0? No
Function Return Value = 1 * factorial (0)

At this time, the function factorial is called again, with N = 0.

FUNCTION:
Does N = 0? Yes
Function Return Value = 1

Now, we have to trace our way back up! See, the factorial function was called six times. At any function level call, all function level calls above still exist! So, when we have N = 2, the function instances where N = 3, 4, and 5 are still waiting for their return values.

So, the function call where N = 1 gets retraced first, once the final call returns 0. So, the function call where N = 1 returns 1*1, or 1. The next higher function call, where N = 2, returns 2 * 1 (1, because that's what the function call where N = 1 returned). You just keep working up the chain.

When N = 2, 2 * 1, or 2 was returned.
When N = 3, 3 * 2, or 6 was returned.
When N = 4, 4 * 6, or 24 was returned.
When N = 5, 5 * 24, or 120 was returned.

And since N = 5 was the first function call (hence the last one to be recalled), the value 120 is returned.

2.4. The Towers of Hanoi:

In the game of Towers of Hanoi, there are three towers labeled 1, 2, and 3. The game starts with n disks on tower A. For simplicity, let n is 3. The disks are numbered from 1 to 3, and without loss of generality we may assume that the diameter of each disk is the same as its number. That is, disk 1 has diameter 1 (in some unit of measure), disk 2 has diameter 2, and disk 3 has diameter 3. All three disks start on tower A in the order 1, 2, 3. The objective of the game is to move all the disks in tower 1 to entire tower 3 using tower 2. That is, at no time can a larger disk be placed on a smaller disk.

Figure 3.11.1, illustrates the initial setup of towers of Hanoi. The figure 3.11.2, illustrates the final setup of towers of Hanoi.

The rules to be followed in moving the disks from tower 1 tower 3 using tower 2 are as follows:

- Only one disk can be moved at a time.
- Only the top disc on any tower can be moved to any other tower.
- A larger disk cannot be placed on a smaller disk.

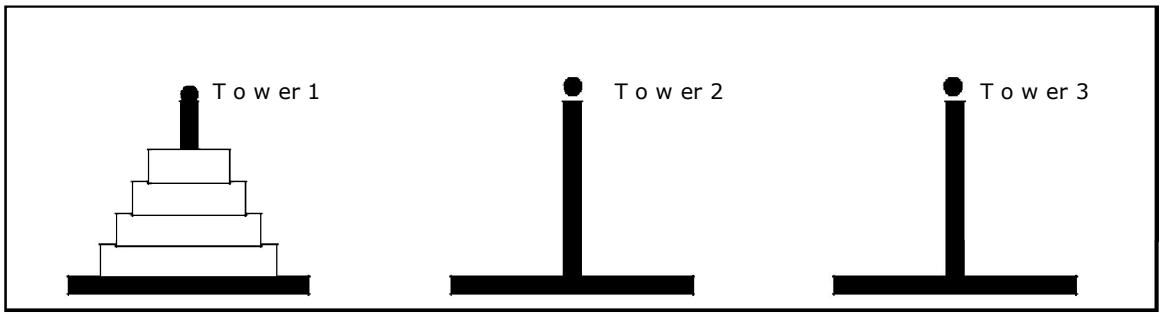


Fig. 3. 11. 1. Initial setup of Towers of Hanoi

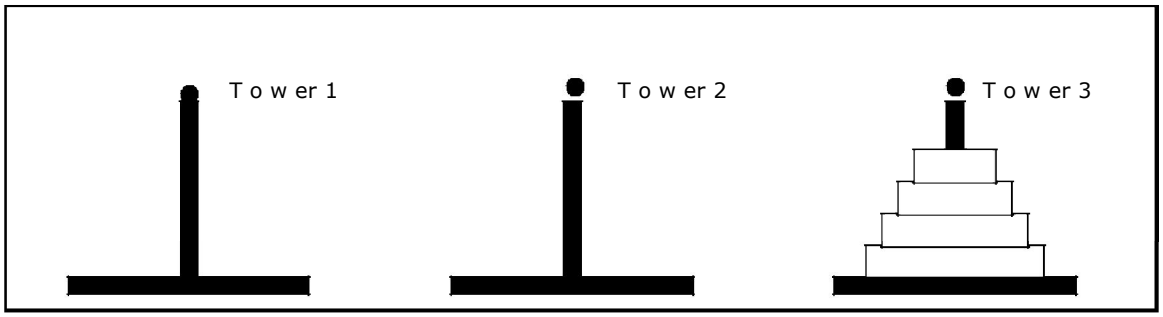


Fig 3. 11. 2. Final set up of Towers of Hanoi

The towers of Hanoi problem can be easily implemented using recursion. To move the largest disk to the bottom of tower 3, we move the remaining n – 1 disks to tower 2 and then move the largest disk to tower 3. Now we have the remaining n – 1 disks to be moved to tower 3. This can be achieved by using the remaining two towers. We can also use tower 3 to place any disk on it, since the disk placed on tower 3 is the largest disk and continue the same operation to place the entire disks in tower 3 in order.

The program that uses recursion to produce a list of moves that shows how to accomplish the task of transferring the n disks from tower 1 to tower 3 is as follows:

```
#include <stdio.h>
#include <conio.h>

void towers_of_hanoi (int n, char *a, char *b, char *c);

int cnt=0;

int main (void)
{
        int n;
        printf("Enter number of discs: ");
        scanf("%d",&n);
        towers_of_hanoi (n, "Tower 1", "Tower 2", "Tower 3");
        getch();
}

void towers_of_hanoi (int n, char *a, char *b, char *c)
{
        if (n == 1)
        {
                ++cnt;
                printf ("\n%5d: Move disk 1 from %s to %s", cnt, a, c);
                return;
        }
        else
        {
                towers_of_hanoi (n-1, a, c, b);
                ++cnt;
                printf ("\n%5d: Move disk %d from %s to %s", cnt, n, a, c);
                towers_of_hanoi (n-1, b, a, c); return;

        }
}
```

Output of the program:

RUN 1:

Enter the number of discs: 3

1: Move disk 1 from tower 1 to tower 3.
2: Move disk 2 from tower 1 to tower 2.
3: Move disk 1 from tower 3 to tower 2.
4: Move disk 3 from tower 1 to tower 3.
5: Move disk 1 from tower 2 to tower 1.
6: Move disk 2 from tower 2 to tower 3.
7: Move disk 1 from tower 1 to tower 3.

RUN 2:

Enter the number of discs: 4

1: Move disk 1 from tower 1 to tower 2.
2: Move disk 2 from tower 1 to tower 3.
3: Move disk 1 from tower 2 to tower 3.
4: Move disk 3 from tower 1 to tower 2.
5: Move disk 1 from tower 3 to tower 1.
6: Move disk 2 from tower 3 to tower 2.
7: Move disk 1 from tower 1 to tower 2.
8: Move disk 4 from tower 1 to tower 3.
9: Move disk 1 from tower 2 to tower 3.
10: Move disk 2 from tower 2 to tower 1.
11: Move disk 1 from tower 3 to tower 1.
12: Move disk 3 from tower 2 to tower 3.
13: Move disk 1 from tower 1 to tower 2.
14: Move disk 2 from tower 1 to tower 3.
15: Move disk 1 from tower 2 to tower 3.

2.5. Fibonacci Sequence Problem:

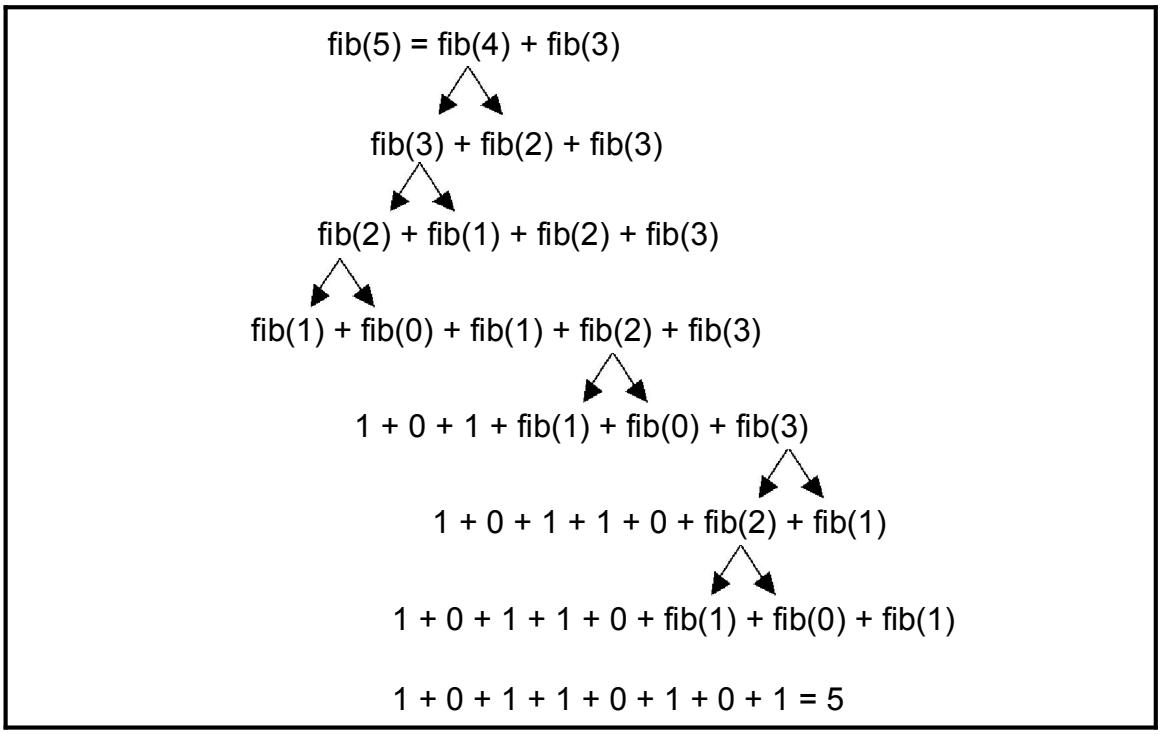
A Fibonacci sequence starts with the integers 0 and 1. Successive elements in this sequence are obtained by summing the preceding two elements in the sequence. For example, third number in the sequence is 0 + 1 = 1, fourth number is 1 + 1= 2, fifth number is 1 + 2 = 3 and so on. The sequence of Fibonacci integers is given below:

0 1 1 2 3 5 8 13 21 . . . . . .

A recursive definition for the Fibonacci sequence of integers may be defined as follows:

Fib (n) = n if n = 0 or n = 1
Fib (n) = fib (n-1) + fib (n-2) for n >=2

We will now use the definition to compute fib(5):



We see that fib(2) is computed 3 times, and fib(3), 2 times in the above calculations. We save the values of fib(2) or fib(3) and reuse them whenever needed.

A recursive function to compute the Fibonacci number in the n^{th} position is given below:

```
main()
{
        clrscr ();
        printf ("=nfib(5) is %d", fib(5));
}

fib(n)
int n;
{
        int x;
        if (n==0 | | n==1)
        return n;
        x=fib(n-1) + fib(n-2);
        return (x);
}
```

Output:

fib(5) is 5

2.6. Program using recursion to calculate the NCR of a given number:

```
#include<stdio.h>
float ncr (int n, int r);

void main()
{
        int n, r, result;
        printf("Enter the value of N and R :");
                scanf("%d %d", &n, &r);
        result = ncr(n, r);
        printf("The NCR value is %.3f", result);
}

float ncr (int n, int r)
{
        if(r == 0)
                return 1;
        else
                return(n * 1.0 / r * ncr (n-1, r-1));

}
```

Output:

```
Enter the value of N and R: 5    2
The NCR value is: 10.00
```

2.7. Program to calculate the least common multiple of a given number:

```
#include<stdio.h>

int alone(int a[], int n);
long int lcm(int a[], int n, int prime);

void main()
{
        int a[20], status, i, n, prime;
        printf ("Enter the limit: ");
                scanf("%d", &n);
        printf ("Enter the numbers : ");
        for (i = 0; i < n; i++)
                scanf("%d", &a[i]);
        printf ("The least common multiple is %ld", lcm(a, n, 2));
}

int alone (int a[], int n);
{
        int k;
        for (k = 0; k < n; k++)
                if (a[k] != 1)
                        return 0;
        return 1;
}
```

```
long int lcm (int a[], int n, int prime)
{
        int i, status;
        status = 0;
        if (allone(a, n))
                return 1;
        for (i = 0; i < n; i++)
                if ((a[i] % prime) == 0)
                {
                        status = 1;
                        a[i] = a[i] / prime;
                }
        if (status == 1)
                return (prime * lcm(a, n, prime));
        else
                return (lcm (a, n, prime = (prime == 2) ? prime+1 : prime+2));
}
```

Output:

```
Enter the limit: 6
Enter the numbers: 6 5 4 3 2 1
The least common multiple is 60
```

2.8. Program to calculate the greatest common divisor:

```
#include<stdio.h>

int check_limit (int a[], int n, int prime);
int check_all (int a[], int n, int prime);
long int gcd (int a[], int n, int prime);

void main()
{
        int a[20], stat, i, n, prime;
        printf ("Enter the limit: ");
                scanf ("%d", &n);
        printf ("Enter the numbers: ");
        for (i = 0; i < n; i ++)
                scanf ("%d", &a[i]);
        printf ("The greatest common divisor is %ld", gcd (a, n, 2));
}

int check_limit (int a[], int n, int prime)
{
        int i;
        for (i = 0; i < n; i++)
                 if (prime > a[i])
                        return 1;
        return 0;
}
```

```
int check_all (int a[], int n, int prime)
{
        int i;
        for (i = 0; i < n; i++)
                if ((a[i] % prime) != 0)
                        return 0;
        for (i = 0; i < n; i++)
                a[i] = a[i] / prime;
        return 1;
}

long int gcd (int a[], int n, int prime)
{
        int i;
        if (check_limit(a, n, prime))
                return 1;
        if (check_all (a, n, prime))
                return (prime * gcd (a, n, prime));
        else
           return (gcd (a, n, prime = (prime == 2) ? prime+1 : prime+2));
}
```

Output:

```
Enter the limit: 5
Enter the numbers: 99    55    22    77    121
The greatest common divisor is 11
```

Exercises

1. What is the importance of the stopping case in recursive functions?

2. Write a function with one positive integer parameter called n. The function will write 2^n-1 integers (where ^ is the exponentiation operation). Here are the patterns of output for various values of n:

   n=1: Output is: 1
   n=2: Output is: 1 2 1
   n=3: Output is: 1 2 1 3 1 2 1
   n=4: Output is: 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

   And so on. Note that the output for n always consists of the output for n-1, followed by n itself, followed by a second copy of the output for n-1.

3. Write a recursive function for the mathematical function:
   f(n) = 1 if n = 1
   f(n) = 2 * f(n-1) if n >= 2

4. Which method is preferable in general?
   a) Recursive method
   b) Non-recursive method

5. Write a function using Recursion to print numbers from *n* to 0.

6. Write a function using Recursion to enter and display a string in reverse and state whether the string contains any spaces. Don't use arrays/strings.

7. Write a function using Recursion to check if a number *n* is prime. (You have to check whether *n* is divisible by any number below *n*)

8. Write a function using Recursion to enter characters one by one until a space is encountered. The function should return the depth at which the space was encountered.

Multiple Choice Questions

1. In a single function declaration, what is the maximum number of statements that may be recursive calls? [ ]
   A. 1 B. 2
   C. n (where n is the argument) D. There is no fixed maximum

2. What is the maximum depth of recursive calls a function may make? [ ]
   A. 1 B. 2
   C. n (where n is the argument) D. There is no fixed maximum

3. Consider the following function: [ ]

```
void super_write_vertical (int number)
{
      if (number < 0)
      {
          printf(" - ");
          super_write_vertical(abs(number));
      }
      else if (number < 10)
          printf("%d\n", number);
      else
      {
           super_write_vertical(number/10);
           printf("%d\n", number % 10);
      }
  }
```

   What values of number are directly handled by the stopping case?
   A. number < 0 B. number < 10
   C. number >= 0 && number < 10 D. number > 10

4. Consider the following function: [ ]

```
void super_write_vertical(int number)
{
      if (number < 0)
      {
           printf(" - ");
           super_write_vertical (abs(number));
      }
      else if (number < 10)
           printf("%d\n", number);
      else
      {
           super_write_vertical(number/10);
           printf("%d\n", number % 10);
      }
  }
```

   Which call will result in the most recursive calls?
   A. super_write_vertical(-1023) B. super_write_vertical(0)
   C. super_write_vertical(100) D. super_write_vertical(1023)

5. Consider this function declaration: [ ]

```
void quiz (int i)
{
   if (i > 1)
   {
        quiz(i / 2);
        quiz(i / 2);
   }
         printf(" * ");
}
```

How many asterisks are printed by the function call quiz(5)?
A. 3 B. 4
C. 7 D. 8

6. In a real computer, what will happen if you make a recursive call without making the problem smaller? [ ]
A. The operating system detects the infinite recursion because of the "repeated state"
B. The program keeps running until you press Ctrl-C
C. The results are non-deterministic
D. The run-time stack overflows, halting the program

7. When the compiler compiles your program, how is a recursive call treated differently than a non-recursive function call? [ ]
A. Parameters are all treated as reference arguments
B. Parameters are all treated as value arguments C. There is no duplication of local variables D. None of the above

8. When a function call is executed, which information is not saved in the activation record? [ ]
A. Current depth of recursion.
B. Formal parameters.
C. Location where the function should return when done.
D. Local variables

9. What technique is often used to prove the correctness of a recursive function? [ ]
A. Communitivity. B. Diagonalization.
C. Mathematical induction. D. Matrix Multiplication.