5 Recursion

Recursion is a process in which a problem is defined in terms of itself. The problem is solved by repeatedly breaking it into smaller problems, which are similar in nature to the original problem. The smaller problems are solved and their solutions are applied to get the final solution of the original problem. To implement recursion technique in programming, a function should be capable of calling itself. A recursive function is a function that calls itself.

```
main()
{
        .....
        rec();
        .....
}/*End of main()*/
void rec()
{
        .....
        rec();          /*recursive call*/
        .....
}/*End of rec()*/
```

Here the function `rec()` is calling itself inside its own function body, so `rec()` is a recursive function. When `main()` calls `rec()`, the code of `rec()` will be executed and since there is a call to `rec()` inside `rec()`, again `rec()` will be executed. It seems that this process will go on infinitely but in practice, a terminating condition is written inside the recursive function which ends this recursion. This terminating condition is also known as exit condition or the base case. This is the case when function will stop calling itself and will finally start returning.

Recursion proceeds by repeatedly breaking a problem into smaller versions of the same problem, till finally we get the smallest version of the problem which is simple enough to solve. The smallest version of problem can be solved without recursion and this is actually the base case.

5.1 Writing a recursive function

Recursion is actually a way of thinking about problems, so before writing a recursive function for a problem we should be able to define the solution of the problem in terms of a similar type of a smaller problem. The two main steps in writing a recursive function are-

1. Identification of the base case and its solution, i.e. the case where solution can be achieved without recursion. There may be more than one base case.
2. Identification of the general case or the recursive case i.e. the case in which recursive call will be made.

Identifying the base case is very important because without it the function will keep on calling itself resulting in infinite recursion.

We must ensure that each recursive call takes us closer to the base case i.e. the size of problem should be diminished at each recursive call. The recursive calls should be made in such a way that finally we arrive at the

base case. If we don't do so, we will have infinite recursion. So merely defining a base case will not help avoid infinite recursion, we should implement the function such that the base case is finally reached.

5.2 Flow of control in Recursive functions

Before understanding the flow of control in recursive calls, first let us see how control is transferred in simple function calls.

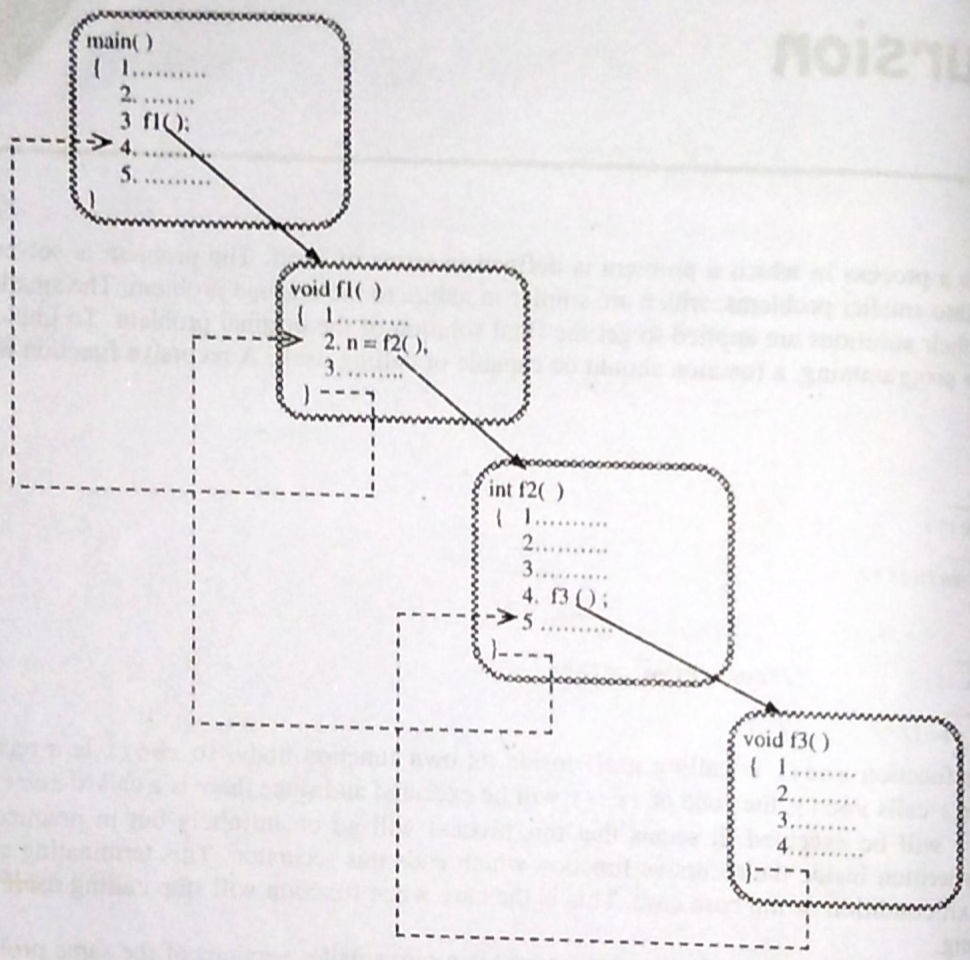


Figure 5.1 Flow of control in normal function calls

In the figure 5.1, the flow of control is-

* `main()` calls function `f1()` at line 3 so control is transferred to `f1()`.
* Inside function `f1()`, the function `f2()` is called at line 2 so control is transferred to function `f2()`.
* Inside function `f2()`, the function `f3()` is called at line 4 so control is transferred to function `f3()`.
* Inside function `f3()` no function is called so control returns back to function `f2()` at line 5.
* After finishing function `f2()`, control is returned to function `f1()` at line 2. Here control returned to line 2 because the work of assigning the return value of `f2()` to variable n still remains.
* Now after finishing the execution of function `f1()` the control is returned to `main()` at line 4.

When the execution of a function is finished we return to the caller(parent) function at the place where it had left it. Recursive calls are no different and they behave in a similar manner. In figure 5.2, the function that is being called each time is the same. We will call the different calls of `func()` as different instances of different invocations of `func()`.

* Initially `main()` calls `func()`, so in first instance of `func()` value of formal parameter n is 5. Line 1 of this function is executed, after this the terminating condition is checked and since it is false we don't return. Now line 4 is executed and then at line 5, `func()` is called with argument 3.

* Now control transfers to the second instance of `func()` and inside this instance, value of n is 3. Line 1 of this second instance of `func()` is executed, after this the terminating condition is checked and since it is false we don't return. Now line 4 is executed and then at line 5, `func()` is called with argument 1.

* Now control transfers to the third instance of `func()` and inside this instance, value of n is 1. Line 1 of this third instance of `func()` is executed, after this the terminating condition is checked and since it is true, we return.

* We return to second instance of `func()` at line 6. Note that now we are inside second instance of `func()` so value of n is 3. After executing lines 6 and 7 we return back to the line 6 of first instance.

* Now we are inside first instance of `func()` so value of n is 5. After executing lines 6 and 7 of first instance we return to `main()` at line 4.

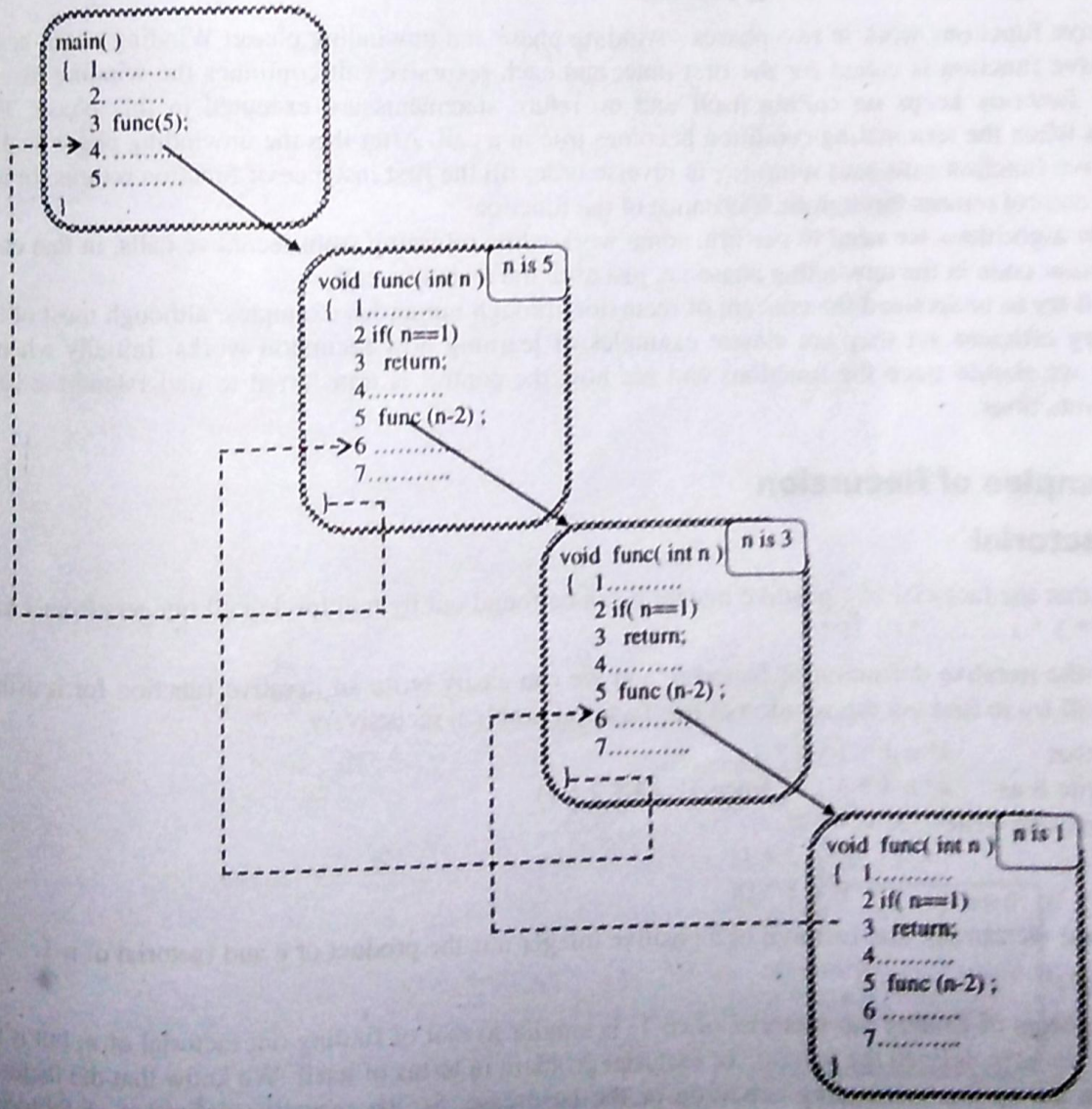


Figure 5.2 Flow of control in recursive function calls

So we can see that the recursive functions are called in a manner similar to that of regular functions, but here the same function is called each time. When execution of an instance of recursive function is finished, we return to the previous instance where we had left it.

We know each function has some local variables that exist only inside that function. The local variables of the called function are active while the local variables of the caller function are kept on hold or suspended. The same case occurs in recursion but here the caller function and called function are copies of the same function. When a function is called recursively, then for each instance a new set of formal parameters and local variables(except static) is created. Their names are same as declared in function but their memory locations are different and they contain different values. These values are remembered by the compiler till the end of function call, so that these values are available while returning. In the example of figure 5.2, we saw that there were three instances of `func()`, but each instance had its own copy of formal parameter n.

The number of times that a function calls itself is known as the recursive depth of that function. In the example of figure 5.2, the depth of recursion is 3.

5.3 Winding and unwinding phase

All recursive functions work in two phases - winding phase and unwinding phase. Winding phase begins when the recursive function is called for the first time, and each recursive call continues the winding phase. In this phase the function keeps on calling itself and no return statements are executed in this phase. This phase terminates when the terminating condition becomes true in a call. After this the unwinding phase begins and all the recursive function calls start returning in reverse order till the first instance of function returns. In unwinding phase the control returns through each instance of the function.

In some algorithms we need to perform some work while returning from recursive calls, in that case we put that particular code in the unwinding phase i.e. just after the recursive call.

We will try to understand the concept of recursion through numerous examples, although most of them may not be very efficient yet they are classic examples of learning how recursion works. Initially when learning recursion, we should trace the functions and see how the control is transferred to understand the behavior of recursive functions.

5.4 Examples of Recursion

5.4.1 Factorial

We know that the factorial of a positive integer n can be found out by multiplying all integers from 1 to n.
n! = 1 * 2 * 3 *.....* (n-1) * n

This is the iterative definition of factorial, and we can easily write an iterative function for it using a loop. Now we will try to find out the solution of this factorial problem recursively.

We know that - 4! = 4 * 3 * 2 * 1
We can write it as- 4! = 4 * 3! (since 3! = 3 * 2 * 1)
Similarly we can write- 3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!

So in general we can say that factorial of a positive integer n is the product of n and factorial of n-1.

n! = n * (n-1)!

Now problem of finding out factorial of (n-1) is similar to that of finding out factorial of n, but it is smaller in size. So we have defined the solution of factorial problem in terms of itself. We know that the factorial of 0 is 1. This can act as the terminating condition or the base case. So the recursive definition of factorial can be written as-

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & n > 0 \end{cases}$$

Now we will write a program, which finds out the factorial using a recursive function.

```
/*P5.1 Program to find the factorial of a number by recursive method*/
#include<stdio.h>
long int fact(int n);
main()
{
        int num;

        printf("Enter a number : ");
        scanf("%d", &num);
        if(num<0)
                printf("No factorial for negative number\n");
        else
                printf("Factorial of %d is %ld\n", num, fact(num));
}
long int fact(int n)
{
        if(n == 0)
                return(1);
        return(n * fact(n-1));
}/*End of fact()*/
```

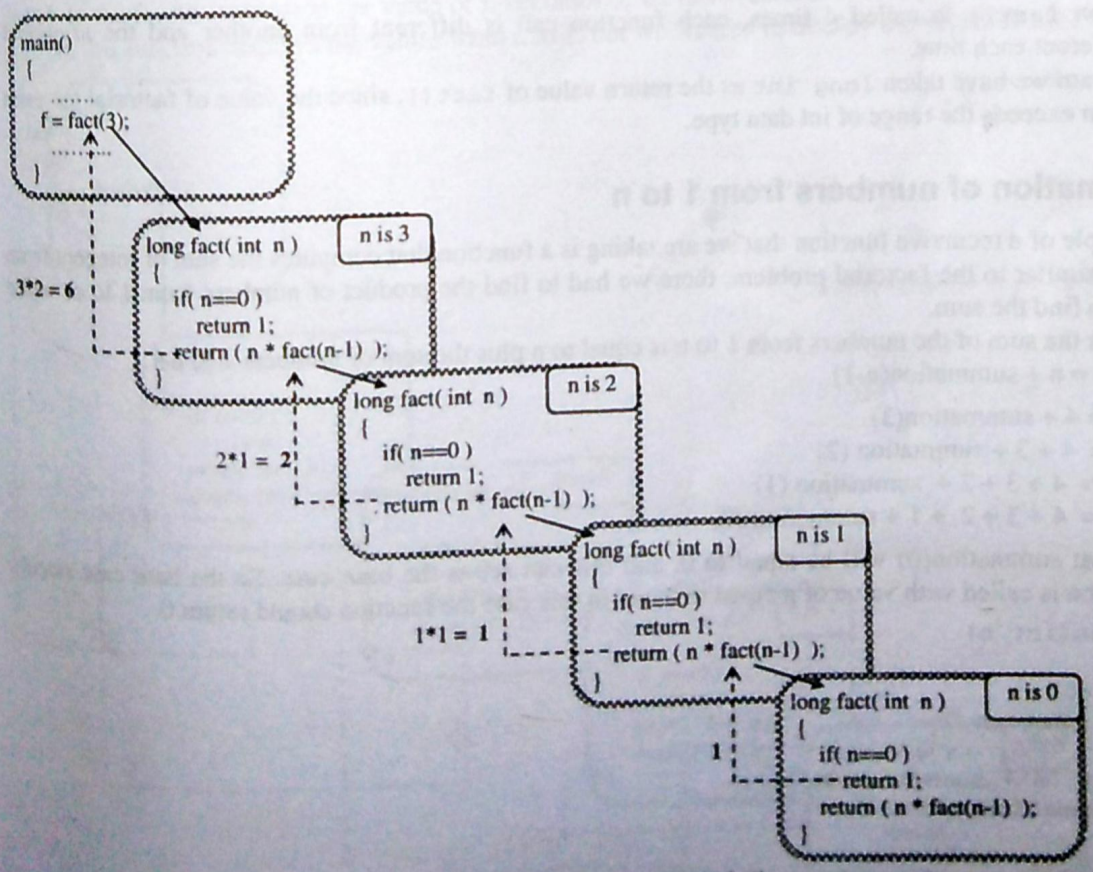


Figure 5.3 Finding Factorial of a number recursively

The function `fact()` returns 1 if the argument `n` is 0, otherwise it returns `n*fact(n-1)`. To return `n*fact(n-1)`, the value of `fact(n-1)` has to be calculated for which `fact()` has to be called again with an

argument of n-1. This process of calling `fact()` continues till it is called with an argument of 0. Suppose we want to find out the factorial of 3.

Initially `main()` calls `fact(3)`
Since 3>0, `fact(3)` calls `fact(2)`
Since 2>0, `fact(2)` calls `fact(1)`
Since 1>0, `fact(1)` calls `fact(0)`
} winding phase

First time when `fact()` is called, its argument is the actual argument given in the `main()` which is 3. So in the first invocation of `fact()` the value of n is 3. Inside this first invocation, there is a call to `fact()` with argument n-1, so now `fact()` is invoked for the second time and this time the argument is 2. Now the second invocation calls third invocation of `fact()` and this time argument is 1. The third invocation of `fact()` calls the fourth invocation with an argument of 0.

When `fact()` is called with n=0, the condition inside `if` statement becomes true, i.e. we have reached the base case, so now the recursion stops and the statement `return 1` is executed. The winding phase terminates here and the unwinding phase begins and control starts returning towards the original call.

Now every invocation of `fact()` will return a value to the previous invocation of `fact()`. These values are returned in the reverse order of function calls.

Value returned by `fact(0)` to `fact(1)` = 1
Value returned by `fact(1)` to `fact(2)` = 1 * `fact(0)` = 1*1 = 1
Value returned by `fact(2)` to `fact(3)` = 2 * `fact(1)` = 2*1 = 2
Value returned by `fact(3)` to `main()` = 3 * `fact(2)` = 3*2 = 6
} Unwinding Phase

The function `fact()` is called 4 times, each function call is different from another and the argument supplied is different each time.

In the program we have taken `long int` as the return value of `fact()`, since the value of factorial for even small value of n exceeds the range of int data type.

5.4.2 Summation of numbers from 1 to n

The next example of a recursive function that we are taking is a function that computes the sum of integers from 1 to n. This is similar to the factorial problem; there we had to find the product of numbers from 1 to n, while here we have to find the sum.

We can say that the sum of the numbers from 1 to n is equal to n plus the sum of numbers 1 to n-1.

summation(n) = n + summation(n-1)

summation(4) = 4 + summation(3)
= 4 + 3 + summation (2)
= 4 + 3 + 2 + summation (1)
= 4 + 3 + 2 + 1 + summation (0)

We know that summation(0) will be equal to 0, and this can act as the base case. So the base case occurs when the function is called with value of n equal to 0 and in this case the function should return 0.

```
int summation(int n)
{
      if(n==0)
            return 0;

      return (n + summation(n-1));
}/*End of summation()*/
```

5.4.3 Displaying numbers from 1 to n

In the previous two examples we had found out the product and sum of numbers from 1 to n, now we will write a recursive function to display these numbers. This function has to only display the numbers so it will not return any value and will be of type `void`.

We have written two functions `display1()` and `display2()`, these functions are traced in figures 5.4 and 5.5. Let us see which one gives us the desired output.

```
void display1(int n)
{
      if(n==0)
         return;
      printf("%d ",n);
      display1(n-1);
}/*End of display1()*/

void display2(int n)
{
      if(n==0)
         return;
      display2(n-1);
      printf("%d ",n);
}/*End of display2()*/
```

The function `display1()` is traced in figure 5.4. Initially the `printf()` inside the first invocation will be executed and in that invocation value of n is 3, so first 3 is displayed. After this the `printf()` inside second invocation is executed and 2 is displayed, and then `printf()` inside third invocation is executed and 1 is displayed. In the fourth invocation the value of n becomes 0, we have reached the base case and the recursion is stopped. So this function displays the values from n to 1, but we wanted to display the values from 1 to n.

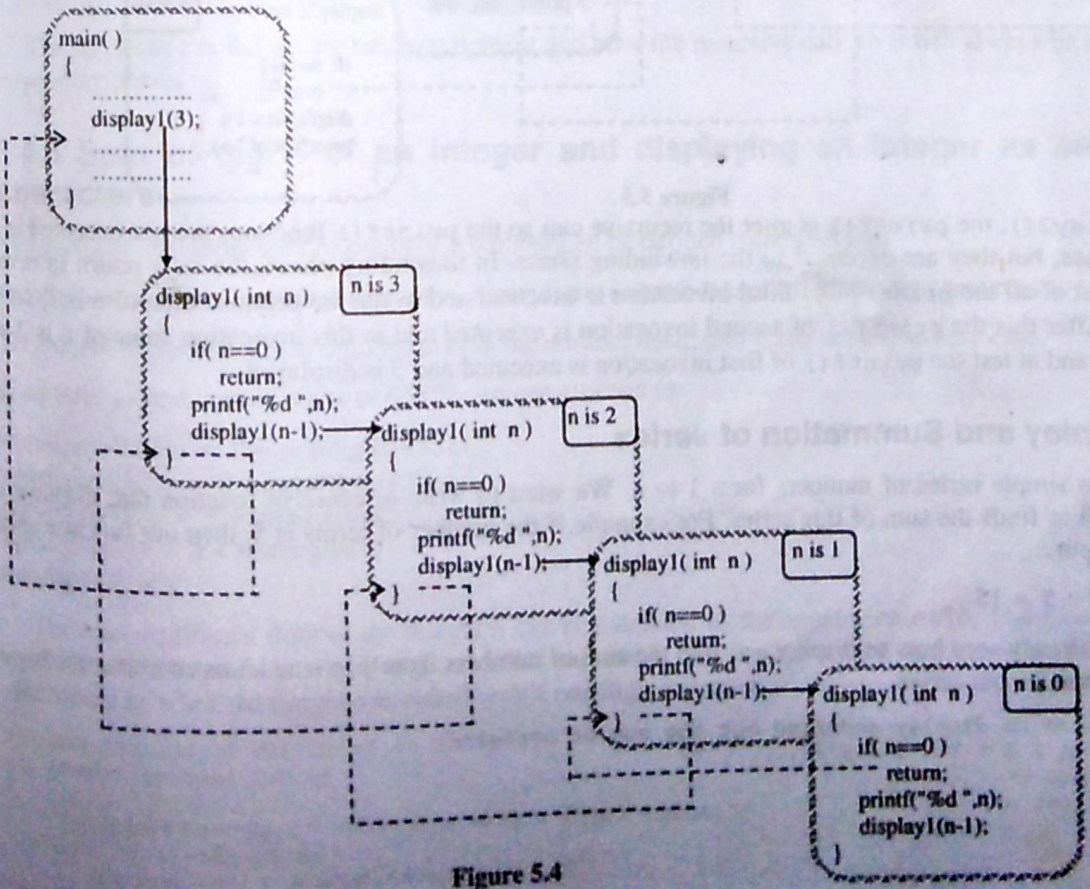


Figure 5.4

Now let us trace the function `display2()`.

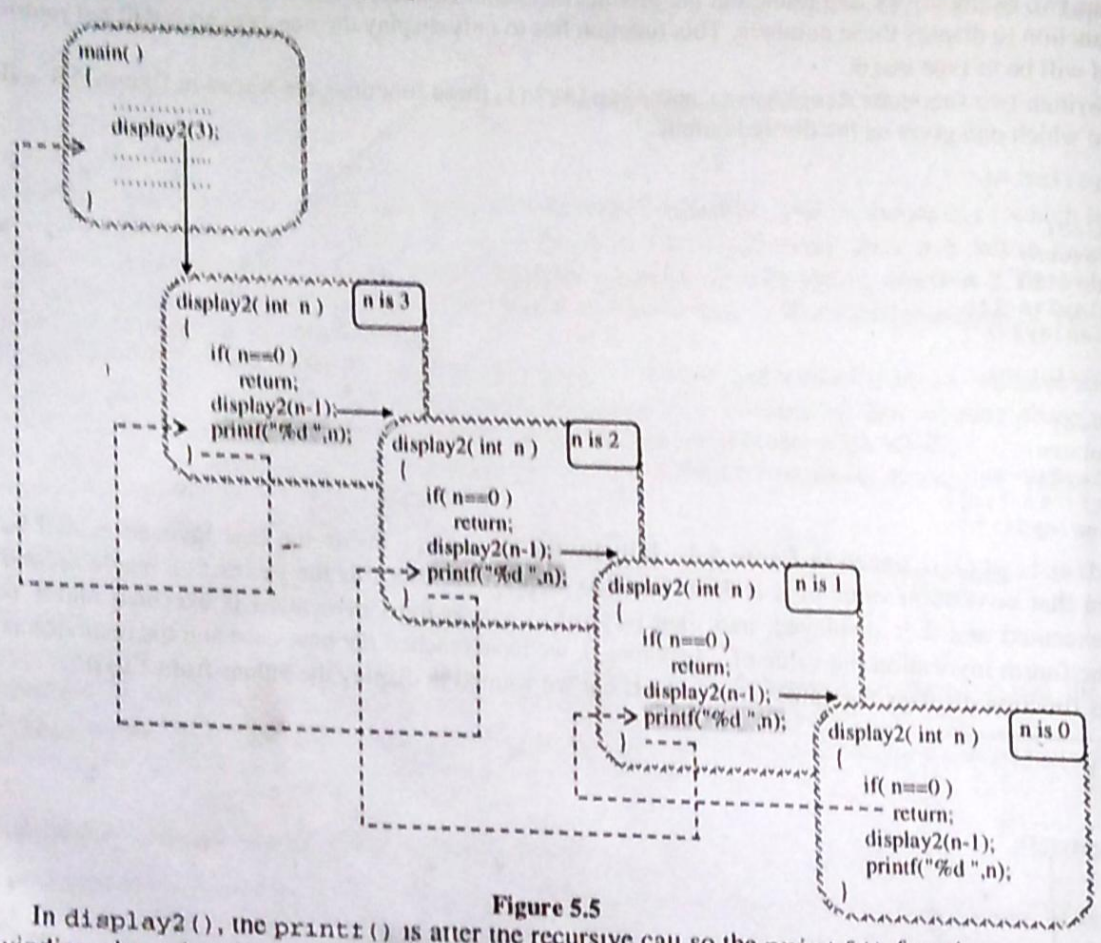


Figure 5.5

In `display2()`, the `printf()` is after the recursive call so the `printf()` functions are not executed in the winding phase, but they are executed in the unwinding phase. In unwinding phase, the calls return in reverse order, so first of all the `printf()` of third invocation is executed and in this invocation value of n is 1, so 1 is displayed. After this the `printf()` of second invocation is executed and in this invocation value of n is 2 so 2 is displayed and at last the `printf()` of first invocation is executed and 3 is displayed.

5.4.4 Display and Summation of series

Let us take a simple series of numbers form 1 to n. We want to write a recursive function that displays this series as well as finds the sum of this series. For example if the number of terms is 5, then our function should give this output.

1 + 2 + 3 + 4 + 5 = 15

We have already seen how to display and find the sum of numbers from 1 to n so let us combine that logic to write a function for our series.

```
/*P5.3 Program to display and find out the sum of series*/
/* Series : 1 + 2 + 3 + 4 + 5 +..... */
#include<stdio.h>
int rseries(int n);
main( )
{
        int n;
```



```
        printf("Enter number
        scanf("%d", &n);

        printf("\b\b = %d\n\n
}/*End of main()*/
int rseries(int n)
{
        int sum;
        if (n == 0)
                return 0;
        return n + rseries(n
        printf("%d + ",n);
}/*End of rseries()*/
```

Let us find out whether the f
the sum of the series but will n
control returns to the previous
(return n + rseries(n-
function. To make the function

```
int rseries(int n)
{
        int sum;
        if(n == 0)
                return 0;
        sum = (n + rserie
        printf("%d + ",n)
        return sum;
}/*End of rseries()*/
```

The `printf()` is before
unwinding phase.

5.4.5 Sum of digits
characters

The problem of finding su

sumdigits(n) = least signif

The sum of digits of a sin

If we have to find sum of

sumdigits(45329) = 9 +
sumdigits(4532) = 2 + s
sumdigits(453) = 3 + su
sumdigits(45) = 5 + su
sumdigits(4) = 4

The least significant
be made with the least
case would be when th

```
/*Finds the sum of
int sumdigits(long
{
        if(n/10 ==
                ret
        return n%10
}/*End of sumdigit
```

```
    printf("Enter number of terms : ");
    scanf("%d", &n);

    printf("\b\b = %d\n\n\n", rseries(n)); /* \b to erase last + sign*/
}/*End of main()*/
int rseries(int n)
{
    int sum;
    if (n == 0)
        return 0;
    return n + rseries(n-1);
    printf("%d + ",n);
}/*End of rseries()*/
```

Let us find out whether the function `rseries()` gives us the desired output or not. This function will return the sum of the series but will not display any term of the series. This is because in the unwinding phase when control returns to the previous invocation, the return statement `(return n + rseries(n-1))` is executed and so the function returns without executing the `printf()` function. To make the function work correctly we can write it like this -

```
int rseries(int n)
{
    int sum;
    if(n == 0)
        return 0;
    sum = (n + rseries(n-1));
    printf("%d + ",n);
    return sum;
}/*End of rseries()*/
```

The `printf()` is before the return statement and after the recursive call, so it will always be executed in the unwinding phase.

5.4.5 Sum of digits of an integer and displaying an integer as sequence of characters

The problem of finding sum of digits of a number can be defined recursively as-

sumdigits(n) = least significant digit of n + sumdigits (n with least significant digit removed)

The sum of digits of a single digit number is the number itself, and it can act as the base case.

If we have to find sum of digits or 45329, we can proceed as-

sumdigits(45329) = 9 + sumdigits(4532)
sumdigits(4532) = 2 + sumdigits(453)
sumdigits(453) = 3 + sumdigits(45)
sumdigits(45) = 5 + sumdigits(4)
sumdigits(4) = 4

The least significant digit of the integer n can be extracted by the expression `n%10`. The recursive call has to be made with the least significant digit removed and this is done by calling the function with `(n/10)`. The base case would be when the function is called with a one digit argument.

```
/*Finds the sum of digits of an integer*/
int sumdigits(long int n)
{
    if(n/10 == 0) /* if n is a single digit number*/
        return n;
    return n%10 + sumdigits(n/10);
}/*End of sumdigits()*/
```

```
/*Displays the digits of an integer*/
void display(long int n)
{
        if(n/10 == 0)
        {
                printf("%d",n);
                return;
        }
        display(n/10);
        printf("%d",n%10);
}/*End of display()*/
/*Displays the digits of an integer in reverse order*/
void Rdisplay(long int n)
{
        if(n/10==0)
        {
                printf("%d",n);
                return;
        }
        printf("%d",n%10);
        Rdisplay(n/10);
}/*End of Rdisplay()*/
```

5.4.6 Base conversion

Now we will write a recursive function which will convert a decimal number to binary, octal and hexadecimal base. To do this conversion we have to divide the decimal number repeatedly by the base till it is reduced to 0 and print the remainders in reverse order. If the base is hexadecimal then we have to print alphabets for remainder values greater than or equal to 10. We want to print the remainders in reverse order, so we can do this work in the unwinding phase.

```
/*P5.5 Program to convert a positive decimal number to Binary, Octal or Hexadecimal*/
#include<stdio.h>
void convert(int, int);
main()
{
        int num;
        printf("Enter a positive decimal number : ");
        scanf("%d",&num);
        convert(num, 2);        printf("\n");
        convert(num, 8);        printf("\n");
        convert(num, 16);       printf("\n");
}/*End of main()*/
void convert(int num,int base)
{
        int rem = num%base;
        if(num==0)
                return;
        convert(num/base,base);

        if(rem < 10)
                printf("%d",rem);
        else
                printf("%c",rem-10+'A');
}/*End of convert()*/
```

5.4.7 Exponentiation of a float by a positive integer

The iterative definition for finding a^n is

$a^n = a * a * a * \ldots\ldots\ldots$ n times

The recursive definition can be written as-

$$a^n = \begin{cases} 1 & n=0 \quad \text{(Base case)} \\ a * a^{n-1} & n>0 \quad \text{(Recursive case)} \end{cases}$$

```
/*P5.6 Program to raise a floating point number to a positive integer*/ #include<stdio.h>
float power(float a,int n);
main()
{
        float a,p;
        int n;
        printf("Enter a and n : ");
        scanf("%f%d",&a,&n);
        p = power(a,n);
        printf("%f raised to power %d is %f\n",a,n,p);
}/*End of main()*/
float power(float a,int n)
{
        if(n == 0)
                return(1);
        else
                return(a * power(a, n-1));
}/*End of power()*/
```

5.4.8 Prime Factorization

Prime factorization of a number means factoring a number into a product of prime numbers. For example prime factors of numbers 84 and 45 are -

84 = 2 * 2 * 3 * 7
45 = 3 * 3 * 5

To find prime factors of a number n, we check its divisibility by prime numbers 2,3,5,7,... till we get a divisor d. Now d becomes a prime factor and the problem reduces to finding prime factors of n/d. The base case occurs when problem reduces to finding prime factors of 1. Let us see how we can find prime factors of 84.

84 is divisible by 2, so 2 is a PF and now problem reduces to finding PFs of 84/2 = 42
42 is divisible by 2, so 2 is a PF and now problem reduces to finding PFs of 42/2=21
21 is not divisible by 2 so we check divisibility by next prime number which is 3,
21 is divisible by 3, so 3 is PF and now problem reduces to finding PFs of 21/3=7.
7 is not divisible by 3 so we check divisibility by next prime number which is 5,
7 is not divisible by 5 so we check divisibility by next prime number which is 7,
7 is divisible by 7, so 7 is a PF and now problem reduces to finding PFs of 7/7=1

The recursive function for printing the prime factors of a number are-

```
void PFactors(int num)
{
        int i = 2;
        if(num == 1)
                return;
        while(num%i != 0)
                i++;
        printf("%d ",i);
        PFactors(num/i);
}/*End of PFactors()*/
```

For simplicity we have checked the divisibility by all numbers starting from 2(prime and non prime), but will get only prime factors. For example 6 is non prime factor of 84 but it has already been removed as a factor of 2 and factor of 3.

5.4.9 Greatest Common Divisor

The greatest common divisor (or highest common factor) of two integers is the greatest integer that divides both of them without any remainder. It can be found out by using Euclid's remainder algorithm which states that-

$$GCD(a, b) = \begin{cases} a & b = 0 \quad \text{(Base case)} \\ GCD(b, a\%b) & \text{otherwise} \quad \text{(Recursive case)} \end{cases}$$

GCD(35, 21) (a = 35, b = 21)
= GCD(21, 14) (a = 21, b = 35 % 21 = 14)
= GCD(14, 7) (a = 14, b = 21 % 14 = 7)
= GCD(7, 0) (a = 7, b = 14 % 7 = 0)
= 7

GCD(12, 26) (a = 12, b = 26)
= GCD(26, 12) (a = 26, b = 12 % 26 = 12)
= GCD(12, 2) (a = 12, b = 26 % 12 = 2)
= GCD(2, 0) (a = 2, b = 12, % 2 = 0)
= 2

The recursive function can be written as-

```
int GCD(int a, int b)
{
        if(b==0)
           return a;
        return GCD(b, a%b);
}/*End of GCD()*/
```

Here nothing is to be done in the unwinding phase. The value returned by the last recursive call just becomes the return value of previous recursive calls, and finally it becomes the return value of the first recursive call.

5.4.10 Fibonacci Series

Fibonacci series is a sequence of numbers in which the first two numbers are 1, and after that each number is the sum of previous two numbers.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

The problem of finding the n^{th} Fibonacci number can be recursively defined as-

$$fib(n) = \begin{cases} 1 & n=0 \text{ or } n=1 \quad \text{(Base case)} \\ fib(n-1) + fib(n-2) & n>1 \quad \text{(Recursive case)} \end{cases}$$

```
/*P5.9 Program to generate fibonacci series*/
int fib(int n);
main()
{
```

```
    int nterms, i;
    printf("Enter number of terms : ");
    scanf("%d", &nterms);
    for(i=0; i<nterms; i++)
        printf("%d ",fib(i));
    printf("\n");
}/*End of main()*/
int fib(int n)
{
    if(n==0  || n==1)
        return(1);
    return(fib(n-1) + fib(n-2));
}/*End of fib()*/
```

Here the function `fib()` is called two times inside its own function body. The following figure shows the recursive calls of function `fib()` when it is called with argument 5.

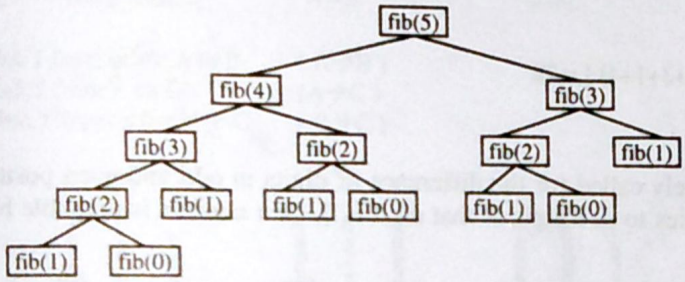


Figure 5.6

This implementation of Fibonacci is very inefficient because it performs same computations repeatedly, for example in the above example it computed the value of `fib(2)` 3 times.

5.4.11 Checking Divisibility by 9 and 11

We can easily check divisibility by any number using % operator but here we will develop functions for checking divisibility by 9 and 11 using the definitions that we have learnt in mathematics because these can help us improve our recursive thinking.
A number is divisible by 9 if and only if the sum of digits of the number is divisible by 9.

test(4968589) -> 4 + 9 + 6 + 9 + 8 + 5 + 8 + 9 = 58

test(58) -> 5 + 8 = 13

test (13) -> 1 + 3 = 4

test(4) -> not divisible by 9

test(1469358) -> 1 + 4 + 6 + 9 + 3 + 5 + 8 = 36

test(36) -> 3 + 6 = 9

test(9) -> divisible by 9

The function will be recursively called for the sum of digits of the number. The recursion will stop when the number reduces to one digit. If that digit is 9, number is divisible by 9. If that digit is less than 9 then number is not divisible by 9.

```
int divisibleBy9(long int n)
{
    int sumofDigits;
    if(n==9)
        return 1;
```

```
    if(n<9)
            return 0;
    sumofDigits = 0;
    while(n>0)
    {
            sumofDigits += n%10;
            n/=10;
    }
    divisibleBy9(sumofDigits);
}/*End of divisibleBy9()*/
```

This function returns 1 if number is divisible by 9 otherwise it returns 0.

Now let us make a function that checks divisibility by 11. A number is divisible by 11 if and only if the difference of the sums of digits at odd positions and even positions is either zero or divisible by 11.

test(91628153) -> [**28**(9+6+8+5) – **7**(1+2+1+3)] = 21
test(21) -> [2 – 1) = 1
test(1) -> Not divisible by 11

62938194 -> [**32**(6+9+8+9) – **10**(2+3+1+4)] = 22
test(22) -> [2 – 2] = 0
test(0) -> divisible by 11

The function will be recursively called for the difference of digits in odd and even positions. The recursion will stop when the number reduces to one digit. If that digit is 0, then number is divisible by 11 otherwise it is not divisible by 11.

```
int divisibleBy11(long int n)
{
        int s1=0,s2=0,diff;
        if(n==0)
                return 1;
        if(n<10)
                return 0;
        while(n>0)
        {
                s1 += n%10;
                n/=10;
                s2 += n%10;
                n/=10;
        }
        diff = s1>s2 ? (s1-s2) : (s2-s1);
        divisibleBy11(diff);
}/*End of divisibleBy11()*/
```

This function returns 1 if number is divisible by 11 otherwise it returns 0.

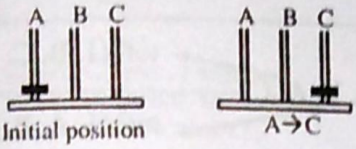
5.4.12 Tower of Hanoi

The problem of Tower of Hanoi is to move disks from one pillar to another using a temporary pillar. Suppose we have a source pillar A which has finite number of disks, and these disks are placed on it in a decreasing order i.e. largest disk is at the bottom and the smallest disk is at the top. Now we want to place all these disks on destination pillar C in the same order. We can use a temporary pillar B to place the disks temporarily whenever required. The conditions for this game are-

1. We can move only one disk from one pillar to another at a time.
2. Larger disk cannot be placed on smaller disk.

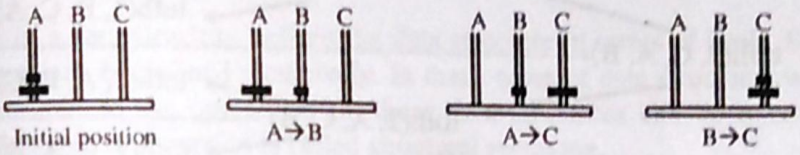
Suppose the number of disks on pillar A is n. First we will solve the problem for n=1, n=2, n=3 and then we will develop a general procedure for the solution. Here A is the source pillar, C is the destination pillar and B is the temporary pillar.

For n=1



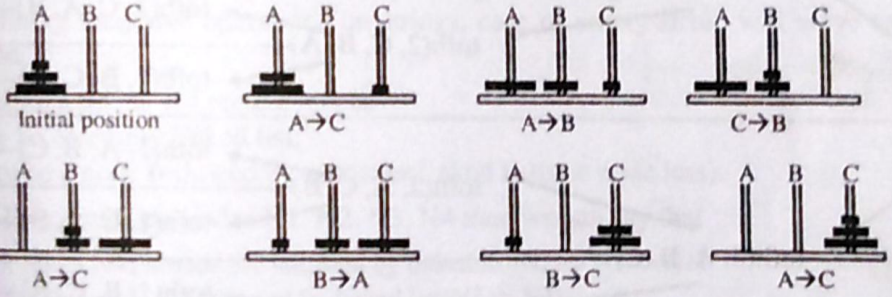
Move the disk from pillar A to pillar C. (A→C)

For n=2



(i) Move disk 1 from pillar A to B (A→B)
(ii) Move disk 2 from A to C (A→C)
(iii) Move disk 1 from pillar B to C (B→C)

For n=3



(i) Move disk 1 from pillar A to C (A→C)
(ii) Move disk 2 from pillar A to B (A→B)
(iii) Move disk 1 from pillar C to B (C→B)
(iv) Move disk 3 from pillar A to C (A→C)
(v) Move disk 1 from pillar B to A (B→A)
(vi) Move disk 2 from pillar B to C (B→C)
(vii) Move disk 1 from pillar A to C (A→C)

These were the solutions for n=1, n=2, n=3. From these solutions we can observe that first we move n-1 disks from source pillar (A) to temporary pillar (B) and then move the largest(n^{th}) disk to the destination pillar(C). So the general solution for n disks can be written as-

1. Move upper n-1 disks from A to B using C as the temporary pillar.
2. Move n^{th} disk from A to C.
3. Move n-1 disks from B to C using A as the temporary pillar.

The base case would be when we have to move only one disk, in that case we can simply move the disk from source to destination pillar and return. The recursion tree for n=5 is given below.

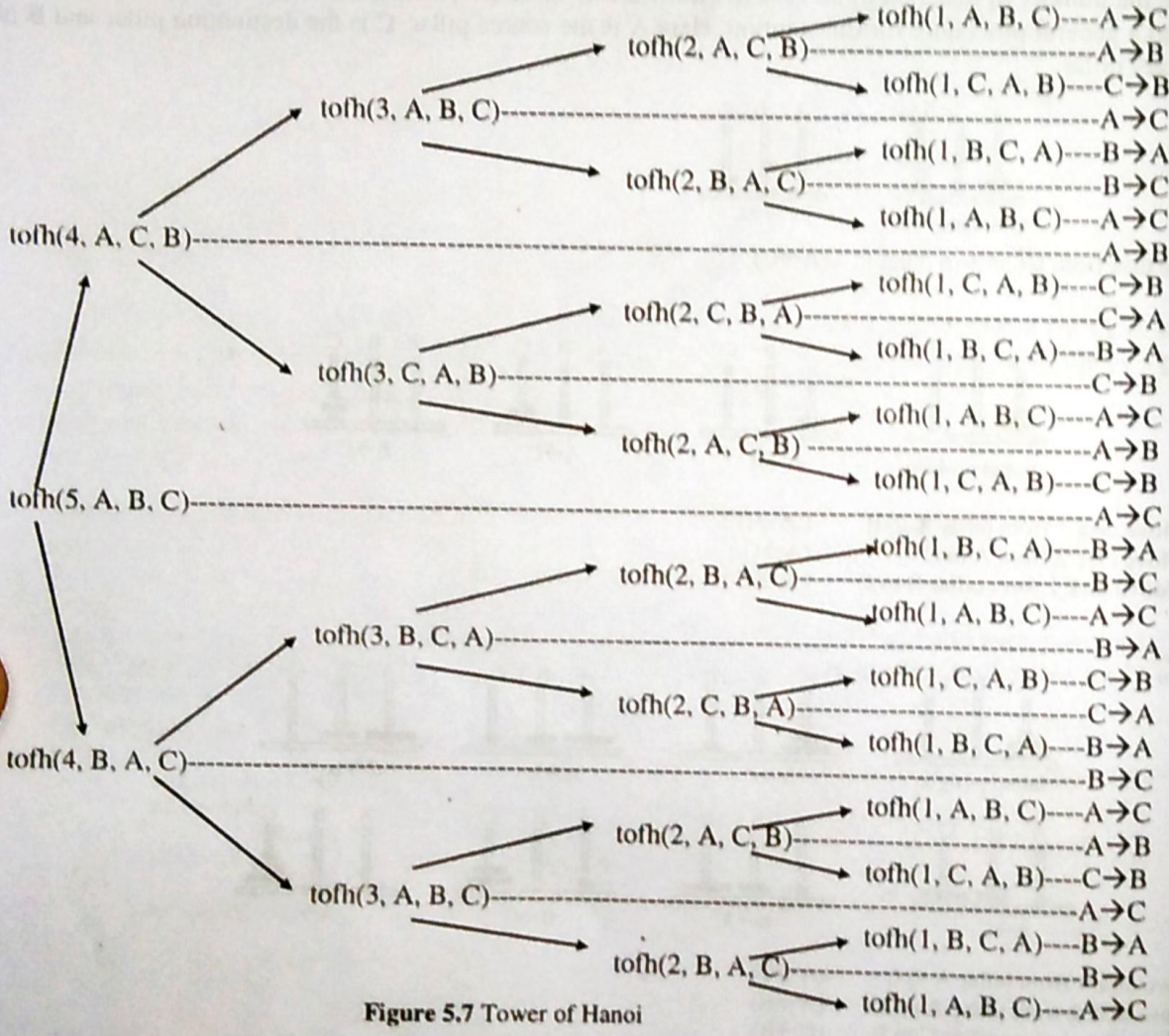


Figure 5.7 Tower of Hanoi

The recursive function for solving tower of Hanoi can be defined as-

$$\text{tofh}(n, \text{source}, \text{temp}, \text{dest}) = \begin{cases} \text{Move disk 1 from source to dest} & n=1 \\ \text{tofh}(n-1, \text{source}, \text{dest}, \text{temp}) \\ \text{Move } n^{th} \text{ disk from source to dest} & n>1 \\ \text{tofh}(n-1, \text{temp}, \text{source}, \text{dest}) \end{cases}$$

```
/*P5.11 Program to solve Tower of Hanoi problem using recursion*/
#include<stdio.h>
void tofh(int ndisk,char source,char temp,char dest);
main()
{
        char source='A',temp='B',dest='C';
        int ndisk;
        printf("Enter the number of disks : ");
        scanf("%d", &ndisk);
        printf("Sequence is :\n");
        tofh(ndisk,source,temp,dest);

}
void tofh(int ndisk,char source,char temp,char dest)
```

```
    if(ndisk==1)
    {
        printf("Move Disk %d from %c-->%c\n",ndisk,source,dest);
        return;
    }
    tofh(ndisk-1,source,dest,temp);
    printf("Move Disk %d from %c-->%c\n",ndisk,source,dest);
    tofh(ndisk-1,temp,source,dest);
}/*End of tofh()*/
```

5.5 Recursive Data structures

A recursive definition of a data structure defines the data structure in terms of itself. Some data structures like strings, linked lists, trees can be defined recursively. In these types of data structures we can take advantage of the recursive structure, and so the operations on these data structures can be naturally implemented using recursive functions. This type of recursion is called structural recursion.

A string can be defined recursively as-
1. A string may be an empty string.
2. A string may be a character followed by a smaller string(one character less).

The string "leaf" is character 'l' followed by string "eaf", similarly string "eaf" is character 'e' followed by string "af", string "af" is character 'a' followed by string "f", string "f" is character 'f' followed by empty string. So while defining recursive operations on strings, case of empty string will serve as the base case for terminating recursion.

Similarly we can define a linked list recursively as-
(i) A linked list may be an empty linked list.
(ii) A linked list may be a node followed by a smaller linked list(one node less).

If we have a linked list containing nodes N1, N2, N3, N4 then we can say that

linkedlist(N1-> N2-> N3-> N4) is node N1 followed by linked list(N2-> N3-> N4),
linkedlist(N2-> N3-> N4) is node N2 followed by linked list(N3-> N4),
linkedlist(N3-> N4) is node N3 followed by linked list(N4),
linkedlist(N4) is node N4 followed by an empty linked list.

While implementing operations on linked lists we can take the case of empty linked list as the base case for stopping recursion.

In this chapter we will study some recursive procedures to operate on strings and linked lists. In the next chapter we will study about another data structure called tree which can also be defined recursively. Some operations on trees are best defined recursively, so before going to that it is better if you understand recursion in linked lists.

5.5.1 Strings and recursion

We have seen the recursive definition of strings and the base case. Now we will write some recursive functions for operations on strings. The first one is to find the length of a string.
The length of empty string is 0 and this will be our terminating condition. In general case, the function is recursively called for a smaller string, which does not contain the first character of the string.

```
int length(char *str)
{
    if(*str == '\0')
        return 0;
    return (1 + length(str+1));
}/*End of length()*/
```

We can print a string by printing the first character of the string followed by printing of the smaller string, if the string is empty there is nothing to be printed so we will do nothing and return(base case).

```
void display(char *str)
{
        if(*str == '\0')
              return;
        putchar(*str);
        display(str+1);
}/*End of display()*/
```

By changing the order of printing statement and recursive call we can get the function for displaying string in reverse order.

```
void Rdisplay(char *str)
{
        if(*str == '\0')
              return;
        Rdisplay(str+1);
        putchar(*str);
}/*End of Rdisplay()*/
```

Note that for displaying string in standard order we have placed the printing statement before the recursive call while during the display of numbers from 1 to n (Section 5.4.3), we had placed the printing statement after the recursive call. We hope that you can understand the reason for this and if not then trace and find out.

5.5.2 Linked lists and recursion

The first recursive function for linked list that we will make is a function to find out the length of the linked list. Length of a linked list is 1 plus the length of smaller list(list without the first node) and length of empty list is zero(base case).

```
int length(struct node *ptr)
{
        if(ptr==NULL)
              return 0;
        return 1 + length(ptr->link);
}/*End of length()*/
```

Similarly we can make a function to find out the sum of the elements of linked list.

```
int sum(struct node *ptr)
{
        if(ptr==NULL)
              return 0;
        return ptr->info + sum(ptr->link);
}/*End of sum()*/
```

Next we will make a function for printing the elements of a linked list. We can print a list by printing the first element of list followed by printing of the smaller list, if the list is empty there is nothing to be printed so we will do nothing and return.

```
void display(struct node *ptr)
{
        if(ptr==NULL)
              return;
        printf("%d ",ptr->info);
        display(ptr->link);
}/*End of display()*/
```

We are just walking down the list till we reach NULL, and printing the info part in the winding phase. This function will be invoked as –

```
display(start);
```

Next we will make a function to print the list in reverse order, i.e. it will print all the elements starting fro the last element.

```
void Rdisplay(struct node *ptr)
{
      if(ptr==NULL)
            return;
      Rdisplay(ptr->link);
      printf("%d ",ptr->info);
}/*End of Rdisplay()*/
```

The next function searches for an element in the list and if it is present it returns 1, otherwise it returns 0.

```
int search(struct node *ptr, int item)
{
      if(ptr==NULL)
            return 0;
      if(ptr->info==item)
            return 1;
      return search(ptr->link,item);
}/*End of search()*/
```

The next function inserts a node at the end of the linked list.

```
struct node *insertLast(struct node *ptr,int item)
{
      struct node *temp;
      if(ptr==NULL)
      {
            temp = malloc(sizeof(struct node));
            temp->info = item;
            temp->link = NULL;
            return temp;
      }
      ptr->link = insertLast(ptr->link, item);
     return ptr;
}/*End of insertLast()*/
```

This function will be invoked as `start = insertLast(start);`

The next function deletes the last node from the linked list.

```
struct node *delLast(struct node *ptr)
{
      if(ptr->link==NULL)
      {
            free(ptr);
            return NULL;
      }
      ptr->link = delLast(ptr->link);
      return ptr;
}/*End of delLast()*/
```

This function will be invoked as `start = delLast(start);`

The next function reverses the linked list.

```
struct node *reverse(struct node *ptr)
{
      struct node *temp;

      if(ptr->link==NULL)
            return ptr;

      temp=reverse(ptr->link);
      ptr->link->link=ptr;
      ptr->link=NULL;

      return temp;
```

```
}/*End of reverse()*/
```

This function will be invoked as `start = reverse(start);`

5.6 Implementation of Recursion

We have seen that recursive calls execute just like normal function calls, so there is no special technique for implementing them. All function calls whether recursive or non recursive are implemented through run time stack and in the previous chapter we had seen how it is done. Here we will take the example of function `fact()` called by main with argument 3, and see the changes in the run time stack as the function `fact()` is evaluated.

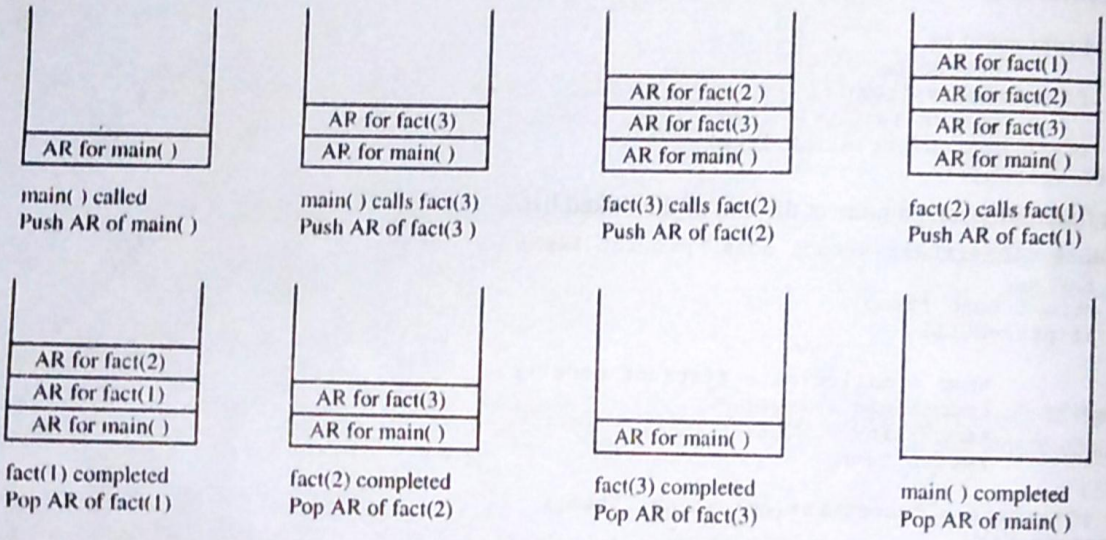


Figure 5.8

In the winding phase, the stack grows as new activation records are created and pushed for each invocation of the function. In the unwinding phase, the activation records are popped from the stack in LIFO manner till the original call returns.

5.7 Recursion vs. Iteration

All repetitive problems can be implemented either recursively or iteratively. First we will compare the workings of both approaches.

In loops, the same block of code is executed repeatedly, and repetition occurs when the block of code is finished or a continue statement is encountered. In recursion, the same block of code is repeatedly executed and repetition occurs when the function calls itself.

In loops, the variables inside the loop are modified using some update statement. In recursion, the new values are passed as parameters to the next recursive call.

For the loop to terminate, there is a terminating condition and the loop progresses in such a way that this condition is eventually hit. If this does not happen then we get an infinite loop. For the recursion to terminate, there is a terminating condition(base case) when function stops calling itself, and the recursion should proceed in such a way that we finally hit the base case. If this does not happen then we will have infinite recursion, and the function will keep on calling itself till the stack is exhausted and we get stack overflow error.

Recursion involves pushing and popping activation records of all the currently active recursive calls on the stack. So the recursive version of a problem is usually slower because of the time spent in pushing and popping these activation records. It is also expensive in terms of memory as it uses space in run time stack to store these activation records. If the recursion is too deep, then the stack may overflow and the program will crash. On the other hand the iterative versions do not have to pay for this function call overhead and so are faster and require less space.