Recursive solutions involve more execution overhead than their iterative counterparts, but their main advantage is that they simplify the code and make it more compact and elegant. Recursive algorithms are easier to understand because the code is shorter and clearer.

Recursion should be used when the underlying problem is recursive in nature or when the data structure on which we are operating is recursively defined like trees. Iteration should be used when the problem is not inherently recursive, or the stack space is limited.

For some problems which are very complex, iterative algorithms are harder to implement and it is easier to solve them recursively. In these cases, recursion offers a better way of writing our code which is both logical and easier to understand and maintain. So sometimes it may be worth sacrificing efficiency for code readability.

Recursion can be removed by maintaining our own stack or by using iterative version.

5.8 Tail recursion

Before studying about tail recursive functions, let us see what tail recursive calls are. A recursive call is tail recursive if it is the last statement to be executed inside the function.

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

void display2(int n)
{
    if(n==0)
        return;
    display2(n-1); /*Not a Tail recursive Call*/
    printf("%d ",n);
}/*End of display2()*/
```

In non void functions, if the recursive call appears in return statement and that call is not part of an expression then the call is tail recursive.

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

long fact(int n)
{
    if(n==0)
            return(1);
    return(n * fact(n-1)); /*Not a tail recursive call*/
}/*End of fact()*/
```

Here the call `fact(n-1)` appears in the return statement but it is not a tail recursive call because the call is part of an expression. Now let us see some functions which have more than one recursive calls.

```
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 ); /*Not tail recursive call*/
    printf("Move Disk %d from %c-->%c\n",ndisk,source,dest);
    tofh(ndisk-1,temp,source,dest ); /*Tail recursive call*/
}/*End of tofh()*/
```

Here the first recursive call is not a tail recursive call while the second one is a tail recursive call. In the next function we have two recursive calls and both are tail recursive.

```
int binary_search(int arr[],int item, int low, int up)
{
        int mid;
        if(up<low)
                return -1;
        mid = (low+up)/2;
        if(item>arr[mid])
                return binary_search(arr,item,mid+1,up);/*tail recursive call*/
        else if(item<arr[mid])
                return binary_search(arr,item,low,mid-1);/*tail recursive call*/
        else
                return mid;
}/*End of binary_search()*/
```

Here only one recursive call will be executed in each invocation of the function, and that recursive call will be the last one to be executed inside the function. So both the calls in this function are tail recursive.

The two recursive calls in `fibonacci()` function(P5.9) are not tail recursive because these calls are part of an expression, and after returning from the call, the return value has to be added to the return value of other recursive call.

A function is tail recursive if all the recursive calls in it are tail recursive. In the examples given earlier the tail recursive functions are - `display1()`, `GCD()`, `binary_search()`. The functions `display2()`, `tofh()`, `fact()` are not tail recursive functions.

Tail recursive functions can easily be written using loops, because as in a loop there is nothing to be done after an iteration of the loop finishes, in tail recursive functions there is nothing to be done after the current recursive call finishes execution. Some compilers automatically convert tail recursion to iteration for improving the performance.

In tail recursive functions, the last work that a function does is a recursive call, so no operation is left pending after the recursive call returns. In non void tail recursive functions(like GCD) the value returned by the last recursive call is the value of the function. Hence in tail recursive functions, there is nothing to be done in the unwinding phase.

Since there is nothing to be done in the unwinding phase, we can jump directly from the last recursive call to the place where recursive function was first called. So there is no need to store the return address of previous recursive calls and values of their local variables, parameters, return values etc. In other words there is no need of pushing new AR for all recursive calls.

Some modern compilers can detect tail recursion and perform tail recursion optimization. They do not push a new activation record when a recursive call occurs; rather they overwrite the previous activation record by current activation record, while retaining the original return address. So we have only one activation record in the stack at a time, and this is for the currently executing recursive call. This improves the performance by reducing the time and memory requirement. Now it doesn't matter how deep the recursion is, the space required will be always be constant.

Since tail recursion can be efficiently implemented by compliers, we should try to make our recursive functions tail recursive whenever possible.

A recursive function can be written as a tail recursive function using an auxiliary parameter. The result is accumulated in this parameter and this is done in such a way that there is no pending operation left after the recursive call. For example we can rewrite the factorial function that we have written earlier as a tail recursive function.

```
long TRfact(int n, int result)
{
        if(n==0)
                return result;
        return TRfact(n-1, n*result);
}/*End of TRfact()*/
```

This function should be called as `TRfact(n,1)`. We can make a helper function to initialize the value of result to 1. The use of this helper function hides the auxiliary parameter.

```
TailRecursiveFact(int n)
{
        return TRfact(n,1);
}/*End of TailRecursiveFact()*/
```

Functions which are not tail recursive are called augmentive recursive functions and these types of functions have to finish the pending work after the recursive call finishes.

5.9 Indirect and direct Recursion

If a function `f1()` calls `f2()` and the function `f2()` in turn calls `f1()`, then this is indirect recursion, because the function `f1()` is calling itself indirectly.

```
f1()
{
      ........
      f2();
      ........
}
f2()
{
      ........
      f1();
      ........
}
```

The chain of functions in indirect recursion may involve any number of functions, for example `f1()` calls `f2()`, `f2()` calls `f3()`, `f3()` calls `f4()`, `f4()` calls `f1()`. If a function calls itself directly i.e. `f1()` is called inside its own function body, then that recursion is direct recursion. All the examples that we have seen in this chapter use direct recursion. Indirect recursion is complex and is rarely used.

Exercise

Find the output of programs from 1 to 16.

1.
```
main()
{
      printf("%d  %d\n",func1(3,8),func2(3,8));
}
func1(int a,int b)
{
      if(a>b)
            return 0;
      return b + func1(a,b-1);
}
func2(int a,int b)
{
      if(a>b)
            return 0;
      return a + func2(a+1,b);
}
```

2.
```
main()
{
      printf("%d \n",func(3,8));
}
int func(int a,int b)
{
```

```
    if(a>b)
        return 1000;
    return a + func(a+1,b);
}
```

3.
```
main()
{
    printf("%d\n",func(6));
    printf("%d\n",func1(6));
}
int func(int a)
{
    if(a==10)
        return a;
    return a + func(a+1);
}
int func1(int a)
{
    if(a==0)
        return a;
    return a + func1(a+1);
}
```

4.
```
main()
{
    printf("%d\n",func(4,8));
    printf("%d\n",func(3,8));
}
int func(int a,int b)
{
    if(a==b)
        return a;
    return a + b + func(a+1,b-1);
}
```

5.
```
main()
{
    func1(10,18);
    printf("\n");
    func2(10,18);
}
void func1(int a,int b)
{
    if(a>b)
        return;
    printf("%d  ",b);
    func1(a,b-1);
}
void func2(int a,int b)
{
    if(a>b)
        return;
    func2(a,b-1);
    printf("%d  ",b);
}
```

6.
```
main()
{
    func1(10,18);
    printf("\n");
    func2(10,18);
}
```

```
void func1(int a,int b)
{
      if(a>b)
            return;
      printf("%d  ",a);
      func1(a+1,b);
}
void func2(int a,int b)
{
      if(a>b)
            return;
      func2(a+1,b);
      printf("%d  ",a);
}
```

7.
```
main()
{
      printf("%d\n",func(3,8));
      printf("%d\n",func(3,0));
      printf("%d\n",func(0,3));
}
int func(int a,int b)
{
      if(b==0)
            return 0;
      if(b==1)
            return a;
      return a + func(a,b-1);
}
```

8.
```
main()
{
      printf("%d\n",count(17243));
}
int count(int n)
{
      if(n==0)
            return 0;
      else
            return 1 + count(n/10);
}
```

9.
```
main()
{
      printf("%d\n",func(14837));
}
int func(int n)
{
      return (n)? n%10 + func(n/10) : 0;
}
```

10.
```
main()
{
      printf("%d\n",count(123212,2));
}
int count(long int n,int d)
{
      if(n==0)
            return 0;
      else if(n%10 == d)
            return 1 + count(n/10,d);
      else
            return count(n/10,d);
```

```
    }
```

11.
```
main()
{
    int arr[10]={1,2,3,4,8,10};
    printf("%d\n",func(arr,6));
}
int func(int arr[],int size)
{
    if(size==0)
        return 0;
    else if(arr[size-1]%2==0)
        return 1 + func(arr,size-1);
    else
        return func(arr, size-1);
}
```

12.
```
main()
{
    int arr[10]={1,2,3,4,8,10};
    printf("%d\n",func(arr,6));
}
int func(int arr[],int size)
{
    if(size==0)
        return 0;
    return arr[size-1] + func(arr,size-1);
}
```

13.
```
main()
{
    char str[100],a;
    printf("Enter a string :");
    gets(str);
    printf("Enter a character :");
    scanf("%c",&a);
    printf("%d\n",f(str,a));
}
int f(char *s,char a)
{
    if(*s=='\0')
        return 0;
    if(*s==a)
        return 1 + f(s+1,a);
    return f(s+1,a);
}
```

14.
```
main()
{
    func1(4);
    func2(4);
}
void func1(int n)
{
    int i;
    if(n==0)
        return;
    for(i=1; i<= n; i++)
        printf("*");
    printf("\n");
    func1(n - 1);
}
void func2(int n)
```

```
{
    int i;
    if(n==0)
        return;
    func2(n-1);
    for (i=1; i<= n; i++)
        printf("*");
    printf("\n");
}
```

15.
```
main()
{
    int arr[10]={2,3,1,4,6,34};
    printf("%d\n",func(arr,6));
}
int func(int arr[],int size)
{
    int m;
    if(size==1)
        return arr[0];
    m = func(arr, size-1);
    if(arr[size-1] < m )
        return arr[size-1];
    else
        return m;
}
```

16.
```
main()
{
    int arr[10]={3,4,2,11,8,10};
    printf("%d\n",func(arr,0,5) );
}
int func(int arr[],int low,int high)
{
    int mid, left, right;
    if(low==high)
        return arr[low];
    mid = (low+high)/2;
    left = func(arr,low,mid);
    right = func(arr,mid+1,high);
    if(left < right)
        return left;
    else
        return right;
}
```

17. Write a recursive function to input and add n numbers.

18. Write a recursive function to enter a line of text and display it in reverse order, without storing the text in a array.

19. Write a recursive function to count all the prime numbers between numbers a and b(both inclusive).

20. A positive proper divisor of n is a positive divisor of n, which is different from n. For example 1,3,5,9, are positive proper divisors 45, but 45 is not a proper divisor of 45.Write a recursive function that displays the proper divisors of a number and returns their sum.

21. A number is perfect if the sum of all its positive proper divisors is equal to the number, for example 28 i perfect number since 28 = 1 + 2 + 4 + 7 + 14. Write a recursive function that finds whether a number is perf or not.

22. Write a recursive function to find the sum of all even numbers in an array.

23. Write a recursive function that finds the sum of all elements of an array by repeatedly partitioning it two almost equal parts.

24. Write a function to reverse the elements of an array.
25. Write a recursive function to find whether the elements of an array are in strict ascending order or not.
26. Write a recursive function that displays a positive integer in words, for example if the integer is 2134 then it is displayed as - two one three four.
27. Write a recursive function that reverses an integer. For example if the input is 43287 then the function should return the integer 78234.
28. Write a recursive function to find remainder when a positive integer a is divided by positive integer b.
29. Write a recursive function to find quotient when a positive integer a is divided by positive integer b.
30. The computation of a^n can be made efficient if we apply the following procedure instead of multiplying times -

$a^8 = (a^2)^4$	$a^{11} = a * (a^2)^5$	$a^{19} = a * (a^2)^9$	$a^{20} = (a^2)^{10}$
$a^4 = (a^2)^2$	$a^5 = a * (a^2)^2$	$a^9 = a * (a^2)^4$	$a^{10} = (a^2)^5$
$a^2 = (a^2)^1$	$a^2 = (a^2)^1$	$a^4 = (a^2)^2$	$a^5 = a * (a^2)^2$
$a^1 = a * (a^2)^0$	$a^1 = a * (a^2)^0$	$a^2 = (a^2)^1$	$a^2 = (a^2)^1$
		$a^1 = a * (a^2)^0$	$a^1 = a * (a^2)^0$

Write a recursive function to compute a^n using this procedure.
31. Write a recursive function to multiply two numbers by Russian peasant method. Russian peasant method multiplies any two positive numbers using multiplication by 2, division by 2 and addition. Here the first number is divided by 2(integer division), and the second is multiplied by 2 repeatedly until the first number reduces to 1. Suppose we have to multiply 19 by 25, we write the result of division and multiplication by 2, in the two columns like this-

19	25	Add
9	50	Add
4	100	
2	200	
1	400	Add
	475	

Now to get the product we'll add those values of the right hand column, for which the corresponding left column values are odd. So 25, 50, 400 will be added to get 475, which is the product of 19 and 25.
32. Write recursive functions to find values of $\lfloor \log_2 N \rfloor$ and $\lfloor \log_b N \rfloor$.
33. Write a recursive function to find the Binomial coefficient C(n,k) which is defined as-

$$C(n,0)=1$$
$$C(n,n)=1$$
$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

34. Write a recursive function to compute Ackermann's function A(m,n) which is defined as-

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0,\ n=0 \\ A(m-1,\ A(m,n-1)) & \text{otherwise} \end{cases}$$

35. Write a recursive function to count the number of vowels in a string.
36. Write a recursive function to replace each occurrence of a character by another character in a string.
37. Write a recursive function to reverse a string.
38. Write a recursive function to return the index of first occurrence of a character in a string.
39. Write a recursive function to return the index of last occurrence of a character in a string.
40. Write a recursive function to find whether a string is palindrome or not. A palindrome is a string that is read the same way forward and backward for example "radar", "hannah", "madam".

41. In the program of pre
lowercase differences are i
recognized as palindromes.
42. Write a function to con
43. Write a function to con
44. Write a function to pr
possible permutations are
45. Write a function to pri

1 | 1234
12 | 123
123 | 12
1234 | 1

46. A triangular numbe
numbers are 1, 3, 6, 10.

Write a recursive func
47. Write a recursive f
number of elements a
48. Write a recursive
49. Write a recursive
50. Write a recursive
51. Write a recursive

41. In the program of previous problem, make changes so that spaces, punctuations marks, uppercase and lowercase differences are ignored. The strings "A man, a plan, a canal – Panama!", "Live Evil" should be recognized as palindromes.

42. Write a function to convert a positive integer to string.

43. Write a function to convert a string of numbers to an integer.

44. Write a function to print all possible permutations of a string. For example if the string is "abc" then all possible permutations are abc, acb, bac, bca, cba, cab.

45. Write a function to print these pyramids of numbers.

	1 2 3 4	4 3 2 1
1	1 2 3	3 2 1
1 2	1 2	2 1
1 2 3	1	1
1 2 3 4		

46. A triangular number is the number of dots required to fill an equilateral triangle. The first 4 triangular numbers are 1, 3, 6, 10.

*	*	*	*
	* *	* *	* *
		* * *	* * *
			* * * *

Write a recursive function that returns n^{th} triangular number.

47. Write a recursive function to check if two linked lists are identical. Two lists are identical if they have same number of elements and the corresponding elements in both lists are same.

48. Write a recursive function to create a copy of a single linked list.

49. Write a recursive function to print alternate nodes of a single linked list.

50. Write a recursive function to delete a node from a single linked list.

51. Write a recursive function to insert a node in a sorted single linked list.