GeeksMan
Recursion
Lesson 1

RECURSION

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

Moreover , Recursion is a method of solving a problem that involves breaking down a problem into smaller subproblems until you get to a small enough problem (Base condition) that it can be solved trivially . Recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

HOW DOES IT WORK

There are three steps needed to solve a recursive problem:

1.   **Base Case**: A recursive problem must have a terminating condition after which the function stops calling itself. If we fail to do so, we will get a segmentation fault telling us that there is no more memory available for us.

2.   **Recursive Call**: In this step we call the function again in the function itself but we pass a smaller subpart as the argument.

3.   **Calculation**: Generally we perform a necessary calculation in this step.

We can exchange the position of the 2nd and 3rd step according to the nature of our problem.

The Concept of recursion is actually based on the Principle of Mathematical Induction . When we use PMI to prove a theorem , we have to show that the base case (usually for x = 0 or x = 1) is true and , the induction hypothesis for case x=k is true must imply that case x = k+1 is also true.

We can now understand that the steps we followed in recursion are actually based on induction steps.

Now let's look at some very basic examples:

To calculate factorial of a number:

```
int fact(int n)
{
   if ((n==0)||(n==1))
   return 1;
   else
   return n*fact(n-1);

}
```

With each function call we are passing a smaller part of the number and when it reaches the base case i.e 1 then it starts returning value to the bigger function calls and hence we get the final answer.

This flowchart shows the working of this code.

(a) Sequence of recursive calls.

(b) Values returned from each recursive call.
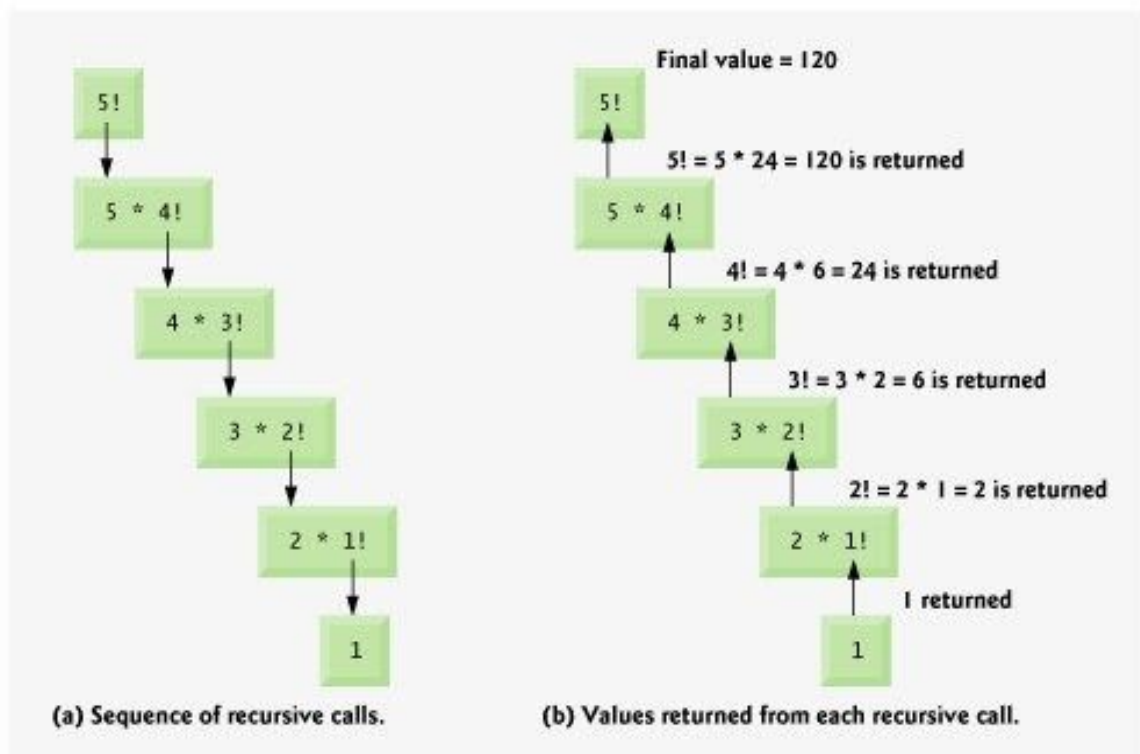
How recursion uses a stack

When you call a function, the CPU's program counter must be saved so that when the function finishes , it knows where to continue from . Given that functions can nest inside functions , we need a structure that can save the program counter each time , and be recalled in the reverse order. That is a stack.
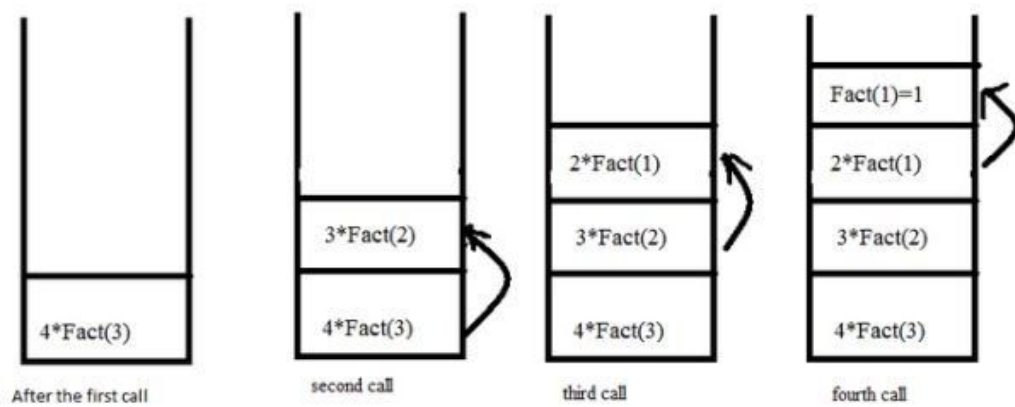
The stack is also used when local variables and parameters within a function are defined . If a function calls another function , the value of all those variables needs to be saved so that when the called function returns , the values are exactly as they were left .

A recursive function calls itself, so by definition it is a series of nested functions. It might not have any local variables, but it will always have a return address . So the stack facilitates this.
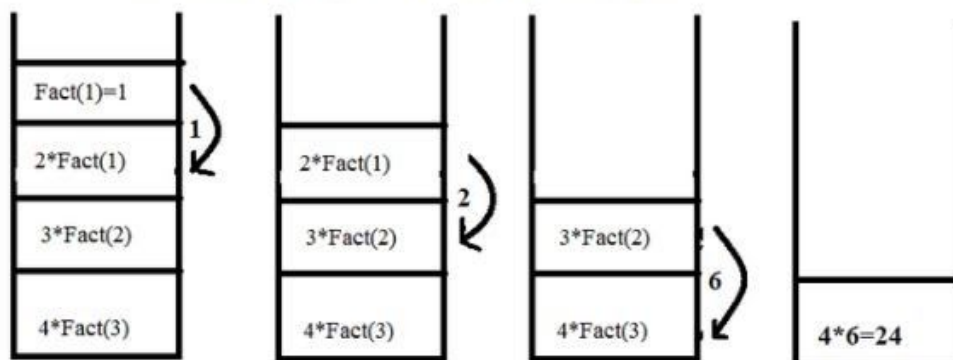
```
1.    int fact(int n)
2.    {
3.      if ((n==0)||(n==1))
4.      return 1;
5.      else
6.      return n*fact(n-1);
7.    }
```

Here, the only local stack storage needed is for the parameter <number> (plus the return address, which is dealt with automatically so we don't code that explicitly). On line 6, the function calls itself once. Each time it does that, it saves the local value of <number> on the stack (plus the current address). When the function returns, the stack unwinds back to where it was when the function was called, so <number> is back to the value it was at when the function was called , and execution continues from where it left off.

When function call happens previous variables gets stored in stack

			Fact(1)=1
		2*Fact(1)	2*Fact(1)
	3*Fact(2)	3*Fact(2)	3*Fact(2)
4*Fact(3)	4*Fact(3)	4*Fact(3)	4*Fact(3)
After the first call	second call	third call	fourth call

Returning values from base case to caller function

Fact(1)=1			
2*Fact(1)	2*Fact(1)		
3*Fact(2)	3*Fact(2)	3*Fact(2)	
4*Fact(3)	4*Fact(3)	4*Fact(3)	4*6=24

When any function is called from main() , the memory is allocated to it on the stack. A recursive function calls itself , the memory for a called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call . When the base case is reached , the function returns its value to the function by whom it is called and memory is deallocated and the process continues .

Types of recursion

1. **Linear Recursion**

A linear recursive function is a function that only makes a single call to itself each time the function runs (as opposed to one that would call itself multiple times during its execution) . The factorial function is a good example of linear recursion.

2. **Tail recursion**

Tail recursion is a form of linear recursion . In tail recursion, the recursive call is the last thing the function does . All the operations are performed at the calling time , and the function will not be performing anything at returning time . As such , tail recursive functions can often be easily implemented in an iterative manner ; by taking out the recursive call and replacing it with a loop , the same effect can generally be achieved . In fact , a good compiler can recognize tail recursion and convert it to iteration in order to optimize the performance of the code.

```
1. #include <bits/stdc++.h>
2. #include <iostream>
3. using namespace std;
4.
5. int fact( int n, int a)
6. {
7.    if (n == 0)
8.        return a;
9.
10.    return fact(n-1, n*a);
11. }
12.
```

```
13.  int main()
14.  {
15.      cout << fact(5 , 1);
16.      return 0;
17.  }
```

```
1.int fact(int n )
2.{
3.    int s = 1;
4.    for(int i = 1;i<=n;i++)
5.    {
6.        s = s * i;
7.    }
8.    return s;
9.}
10.  int main()
11.  {
12.          cout << fact(5 );
13.      return 0;
14.  }
```

Space complexity while performing the question with a loop is O(1).

A good example of a tail recursive function is a function to compute the GCD , or Greatest Common Denominator, of two numbers:

3. Head recursion

If a recursive function calls itself and that recursive call is the first statement in the function then it's known as **Head Recursion.** There's no statement, no operation before the call. The function doesn't have to process or perform any operation at the time of calling and all operations are done at returning time.

```cpp
1. #include <bits/stdc++.h>
2. #include <iostream>
3. using namespace std;
4.
5. void fun(int n)
6. {
7.     if (n > 0)
8.     {
9.         fun(n - 1);
10.        cout << n;
11.    }
12. }
13.
14. int main()
15. {
16.    int x = 3;
17.    fun(x);
18.    return 0;
19. }
```

4. **Tree Recursion**

Function is calling itself more than one time.

Some recursive functions don't just have one call to themself, they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.

```cpp
1. #include <bits/stdc++.h>
2. #include <iostream>
3. using namespace std;
4.
5. void fun(int n)
6. {
7.     if (n > 0)
8.     {
9.         cout << n ;
10.
11.         // Calling once
12.         fun(n - 1);
13.
14.         // Calling twice
15.         fun(n - 1);
16.     }
17. }
18. int main()
19. {
20.    fun(3);
21.    return 0;
22. }
```

5. Nested Recursion

In nested recursion, one of the arguments to the recursive function is the recursive function itself! These functions tend to grow extremely fast. A good example is the classic mathematical function, "Ackerman's function. It grows very quickly (even for small values of x and y, Ackermann(x,y) is extremely large) and it cannot be computed with only definite iteration (a completely defined for() loop for example); it requires indefinite iteration (recursion, for example).

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

where m and n are non-negative integers

```
1.  int ackerman(int m, int n)
2.  {
3.     if (m == 0)
4.         return(n+1);
5.     else if (n == 0)
6.         return(ackerman(m-1,1));
7.     else return(ackerman(m-1,ackerman(m,n-1)));
8.  }
```

6.   **Indirect Recursion**

In this recursion, there may be more than one function and they are calling one another in a circular manner . *Indirect recursion* is when function 1 calls a different function , and this in turn calls the original calling function 1

```cpp
1. #include <iostream>
2. using namespace std;
3. void funB(int n);
4. void funA(int n)
5. {
6.     if (n > 0)
7.     {
8.         printf("%d ", n);
9.
10.         // Fun(A) is calling fun(B)
11.         funB(n - 1);
12.     }
13. }
14.  void funB(int n)
15.  {
16.    if (n > 1) {
17.        printf("%d ", n);
18.
19.        // Fun(B) is calling fun(A)
20.        funA(n / 2);
21.    }
22.  }
23.  int main()
24.  {
```

```
25.    funA(20);
26.    return 0;
27.  }
```

7. **Mutual recursion**:-

Two functions are called mutually recursive if the first function makes a recursive call to the second function and the second function, in turn, calls the first one.

A simple example of mutual recursion is a set of functions to determine whether an integer is even or odd.

```cpp
1.  #include <bits/stdc++.h>
2.  #include <iostream>
3.  using namespace std;
4.  bool odd( int  );
5.  bool even( int number )
6.  {
7.     if( number == 0 )
8.        return true;
9.     else
10.       return odd(abs(number)-1);
11. }
12.
13. bool odd( int number )
14. {
15.    if( number == 0 )
16.       return false;
17.    else
18.       return even(abs(number)-1);
19. }
20.
```

```cpp
21. int main()
22. {
23.   int a;
24.   cin >> a;
25.   if(even(a))
26.       cout << "even\n";
27.   else
28.       cout << "odd\n";
29.   return 0;
30. }
```

ADVANTAGES OF RECURSION OVER ITERATION

●   Recursive solutions often tend to be shorter and simpler than non-recursive ones.

●   Some algorithms just lend themselves to recursion because of the way they are designed (Fibonacci sequences, traversing a tree like structure, etc.). Recursion makes the algorithm more succinct and easier to understand (therefore shareable and reusable).

●   Tower of Hanoi Problem can be solved more effectively using recursion.Recursion follows a divide and conquer technique to solve problems.

●   Also, some recursive algorithms use Lazy evaluation which makes them more efficient than their iterations . This means that they only do the expensive calculations at the time they are needed rather than each time the loop runs.

●   In functional programming language implementations, sometimes, iteration can be very expensive and recursion can be very cheap. In many, recursion is transformed into a simple jump, but changing the loop variable (which is mutable) sometimes requires some relatively heavy operations, especially on implementations which support multiple threads of execution.

DISADVANTAGES OF RECURSION

● A recursive program has greater space requirements than an iterative program as each function call will remain in the stack until the base case is reached.

● It also has greater time requirements if some recursive function repeats the computations for some parameters, the run time can be prohibitively long even for very simple cases.

Key differences between Iteration and Recursion

● A conditional statement decides the termination of recursion while a control variable's value decides the termination of the iteration statement (except in the case of a while loop).

● Infinite recursion can lead to system crash whereas infinite iteration consumes CPU cycles.

● Recursion repeatedly invokes the mechanism, and consequently the overhead, of method calls. This can be expensive in both processor time and memory space while iteration doesn't.

● Recursion makes code smaller while iteration makes it longer.

Finding the time complexity of recursive and iterative code

Iterative:-

In iterative code we can simply find the time complexity by calculating the no. of cycle executed in the loop.

Recursive:-

Time complexity in recursive code can be found by getting the value of nth recursive call in terms of n-1th call and finally getting to the base condition

Below example will illustrate these things easily - have a look

Lets first get to the iterative code of factorial of a number:-

Iterative code:-

```
1. int fact(int n)
2. {
3.     int i,res=0;
4.     for(i=1;i<=n;i++)
5.         {
6.             res=res*i;
7.         }
8. return   res;
9. }
```

Here loop is having n cycles and in each cycle one multiplication operation if performed.

Thus complexity =n*1=O(n)

Recursive code:-

```
1. int fact(int n)
2. {
3.    if (n==0)
4.      return 1;
5.    else
6.      return n*fact(n-1);
7. }
```

Here we are doing three constant operations :

1.checking the value of n in each call

2. Multiplication

3.subtraction

Thus ,

T(n) = T(n-1) +3   ....(a)   (first call)

Second call :

T(n-1) = T(n-2) +3 ......(b)

Now putting eq b in a

T(n)=(T(n-2) +3)+3

T(n)=T(n-2) +6   (second call)

For kth call we get

T(n)=T(n-k) + 3k

And we will reach base condition when at T(0)

For nth call

T(n)=T(n-n)+3n

T(n)=T(0)+3n

T(n)=1+3n

Hence time complexity = O(3*n) = O(n)

Print Pattern

Print a sequence of numbers starting with N, *without using a loop*, in which A[i+1] = A[i] - 5, if A[i]>0, else A[i+1]=A[i] + 5 repeat it until A[i]=N.

Input:

The first line contains an integer T, number of test cases. Then following T lines contains an integer N.

Output:

For each test case, print the pattern in a newline with space separated integers.

Example:

Input:

2

16

10

Output:

16 11 6 1 -4 1 6 11 16

10 5 0 5 10

Explanation:

We basically first reduce 5 one by one until we reach a negative or 0. After we reach 0 or negative, we one by one add 5 until we reach N.

Solution:

```cpp
1.  #include<iostream>
2.  #include<bits/stdc++.h>
3.  using namespace std;
4.  void prnt_pat_asc(int prev,int &n)
5.  {
6.        if((prev+5)==n)
7.   {
8.        cout<<(prev+5)<<" ";
9.        return;
10.      }
11.  cout<<(prev+5)<<" ";
12.      prnt_pat_asc(prev+5,n);
13.  }
14.
15.  void prnt_pat_des(int prev,int &n)
16.  {
17.            if(prev>0)
18.      {
19.          cout<<prev-5<<" ";
20.          prnt_pat_des(prev-5,n);
21.
22.      }
23.      else
24.          prnt_pat_asc(prev,n);
25.  }
```

```cpp
26.
27.   void solve()
28.   {
29.     int n;
30.     cin>>n;
31.     cout<<n<<" ";
32.     prnt_pat_des(n,n);
33.
34.
35.   }
36.   int main()
37.   {
38.      int t;
39.     cin>>t;
40.     while(t--)
41.     {
42.     solve();
43.         cout<<endl;
44.     }
45.   }
```