Recursion:

The term "recursion" refers to a programming or problem-solving technique in which a function calls itself directly or indirectly in order to solve a larger problem by breaking it down into smaller, similar subproblems.

Recursion is a process in which a function solves a problem by calling itself with modified arguments, aiming to reach a base case (or termination condition) where the problem can be solved directly without further recursive calls. Each recursive call typically reduces the problem into a smaller or simpler instance of the same problem until the base case is reached.

The "base case" is a fundamental concept in recursive programming and problem-solving. It refers to the termination condition in a recursive algorithm or function, where the recursion stops, and the function returns a result without making further recursive calls.

```cpp
#include <iostream>
using namespace std;


int sum = 0;


int  fn(int n, int N) {
    if (n == N) {
        return n;
    }
    sum += fn(n + 1, N);

}


int main() {
    int N = 5;
    fn(0, N);
    cout << sum;  // Print the final sum after the recursion is
done.
}
```


return garbage
value

```
#include <iostream>
using namespace std;


int sum = 0;


int  printN(int n, int N) {
    if (n == N) {
        return n;
    }
   sum=n+printN(n+1,N);
   return 0;
}


int main() {
    int N = 5;
    printN(1, N);
    cout << sum;  // Print the final sum after the recursion is
done.
}
```

prints 5 boz not returning sum

```
#include <iostream>
using namespace std;


int i = -1;
int sum = 0;


int printN(int N[9]) {
    i++;
    if(i==9)
    {
        return 0;
    }
    sum=N[i]+printN(N);
    return sum;
}


int main() {
    int N[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    printN(N);


    cout << sum;
    return 0;
}
```

```
#include <iostream>
using namespace std;

int arraySum(int arr[], int size) {
    // Base case: If the array is empty (size is 0), return 0.
    if (size == 0) {
        return 0;
    }

    // Recursive case: Calculate the sum recursively.
    // The sum of the array is the sum of the first element and the sum
of the rest.
    return arr[0] + arraySum(arr + 1, size - 1);
}

int main() {
    int size;
    cout << "Enter the size of the array: ";
    cin >> size;

    int arr[size];

    cout << "Enter the elements of the array: ";
    for (int i = 0; i < size; i++) {
        cin >> arr[i];
    }

    int sum = arraySum(arr, size);

    cout << "Sum of the array elements: " << sum << endl;

    return 0;
}
```

```cpp
#include <iostream>
using namespace std;
 int x=0 ;

int printN(int N) {

    if (N>0) {
        x++;
        return printN(N-1)+x;
    }
    return 0;
}


int main() {
    int N = 5;
  cout<< printN(N)<<endl;
  cout<<printN(N)<<endl;
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

int printN(int N) {

   Static int x=0 ;

    if (N>0) {
        x++;
        return printN(N-1)+x;
    }
    return 0;
}


int main() {
    int N = 5;
  cout<< printN(N)<<endl;
  cout<<printN(N)<<endl;
    return 0;
```

```
#include<iostream>
using namespace std;
void fn(int n)
{
if(n==0)
{
    return;
}
cout<<n;
fn(n--);
}
int main()
{

int n =4;

fn(n);

}
```

```cpp
//0 1 2 3 5 8 13
#include<iostream>
using namespace std;
int fn(int n)
{
if(n==0)
{
    return 0;
}
return n%10+fn(n/10);
}

int main()
{

int n =1234;
cout<<fn(n);

}
```

Revers of a number

```
//0 1 2 3 5 8 13
#include<iostream>
using namespace std;
int sum=0;
void fn(int n)
{
if(n==0)
{
    return;
}
int rem=n%10;
sum=sum*10+rem;
fn(n/10);
}



int main()
{


int n =504;
fn(n);
cout<<sum;




}
```

Check given sum can be achieve from the sum of elements of the array

```cpp
#include<iostream>
using namespace std;



bool sum(int arr[], int size, int k, int current_sum, int index) {

    if (current_sum == k) {
        return true;
    }
    if (index == size || current_sum > k) {
        return false;
    }
    return sum(arr, size, k, current_sum, index + 1) ||
           sum(arr, size, k, current_sum + arr[index], index + 1);
}


int main() {
    int arr[] = {3, 7, 1, 2, 1};
    int k = 15;
    int size = 5;
    int current_sum = 0;
    int index = 0;

    if (sum(arr, size, k, current_sum, index)) {
        cout << "Yes, there exists a subset with the sum " << k << "." << endl;
    } else {
        cout << "No, there is no subset with the sum " << k << "." << endl;
    }


    return 0;
}
```

Binary search

```
#include<iostream>
using namespace std;
int binary(int arr[],int target,int start,int end)
{
int m=(start+end)/2;
if(start>end)
{
    return -1;
}


if(arr[m]==target)
{
    return m;
}
if(target>arr[m])
{


return binary(arr,target,m+1,end);



}
if(target<arr[m])
{
   return  binary(arr,target,start,m-1);
}



}
int main()
{
int arr[]={1,2,5,6,12,22,34,55};
int target=3;
int start=0;
int end=sizeof(arr)/sizeof(arr[0]);
cout<<binary(arr,target,start,end-1);
}
```

Print all possible strings of length k that can be formed from a set of n characters

```
Input:
set[] = {'a', 'b'}, k = 3

Output:
aaa
aab
aba
abb
baa
bab
bba
bbb
```

Solution

```
#include<iostream>
using namespace std;
void findcom(char com[],int n,int k,string final)
{
if(k==0)
{
cout<<final<<endl;
return;
}
for(int i=0;i<n;i++)
{
string newfinal=final+com[i];


findcom(com,n,k-1,newfinal);
}
}
int main()
{
char com[]={'a','b'};
int k=3;
findcom(com,2,k,"");


}
```

There is N number of people at a party. Find the total number of handshakes such that a person can handshake only once.

Remove Character From an String

input:string s="baccadh";
output:bccdh

```
#include<iostream>
#include<string>
using namespace std;
string removech(string s,string e)
{
if(s.empty())
{
    return e;
}
if(s.at(0)=='a')
{
return removech(s.substr(1),e);
}
else
e=e+s.at(0);
return removech(s.substr(1),e);


}
int main()
{
string s="baccadh";
cout<<removech(s,"");
}
```

Subset of A String

```
string s="abc";
output:
abc
ab
ac
a
bc
b
c
```

```
#include<iostream>
using namespace std;
bool sub(string s,string news)
{
if(s.empty())
{
cout<< news<<endl;
return true ;
}
char ch=s.at(0);
sub(s.substr(1),ch+news);
sub(s.substr(1),news);


}


int main()
{


string s="abc";
sub(s,"");


}
```

```
#include<iostream>
using namespace std;
void  sub(string s,string news,int index)
{
if(index==s.length())
{
cout<< news<<endl;
return  ;
}
 sub(s,news+s[index],index+1);

 sub(s,news,index+1);

}


int main()
{
int index=0;
string s="abc";
sub(s,"",index);


}
```

Subset of A an integer array

```
int s []= {1,2,3};
output =
123
12
13
1
23
2
3
```

```
#include <iostream>
using namespace std;


void printSubsets(int  s[], int size,int t[], int index) {
    if (index == size) {
        for(int i=0;i<size;i++)
        {       if(t[i])
                 cout<<t[i];
        }
        cout<<endl;
        return;
    }
    t[index]=s[index];
    printSubsets(s, size,t,index + 1);

     t[index]=false;

    printSubsets(s,size, t, index + 1);

}


int main() {
    int s []= {1,2,3};
    int t[3]={0};
    printSubsets(s,3,t, 0);


    return 0;
}
```

Types of Recurssion

Direct Recursion

1. Tail Recursion
2. Head Recursion
3. Tree Recursion
4. Nested Recursion

InDirect Recursion

Tail Recursion

Tail Recursion: If a recursive function calling itself and that recursive call is the last statement in the function then it's known as **Tail Recursion**. After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling and it does nothing at returning time.

```
void fun(int n)
{
        if (n > 0) {
                cout << n << " ";

                // Last statement in the function
                fun(n - 1);
        }
}
int main()
{
        int x = 3;
        fun(x);
        return 0;
}
```

Head Recursion

Head Recursion: If a recursive function calling 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.

```
// Recursive function
void fun(int n)
{
        if (n > 0) {

                // First statement in the function
                fun(n - 1);

                cout << " "<< n;
        }
}

// Driver code
int main()
{
        int x = 3;
        fun(x);
        return 0;
}
```

Tree Recursion

Tree Recursion: To understand **Tree Recursion** let's first understand **Linear Recursion**. If a recursive function calling itself for one time then it's known as **Linear Recursion**. Otherwise if a recursive function calling itself for more than one time then it's known as **Tree Recursion**.

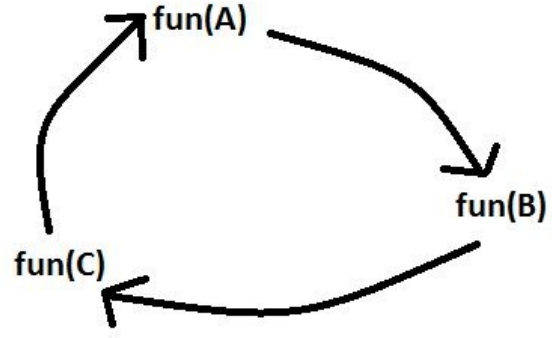
```
void fun(int n)
{
        if (n > 0)
        {
                cout << " " << n;

                // Calling once
                fun(n - 1);

                // Calling twice
                fun(n - 1);
        }
}
int main()
{
        fun(3);
        return 0;
}
```

Nested Recursion

```
#include<iostream>
using namespace std;
int subset(int n)
{
 if(n>100)
 {
    return n-10;
 }
 else
 subset(subset(n+11));
}
int main()
{
    cout<<subset(95);
}
```

Indirect Recursion

Indirect Recursion: In this recursion, there may be more than one functions and they are calling one another in a circular manner.




```
void funB(int n);

void funA(int n)
{
	if (n > 0) {
		cout <<" "<< n;

		// fun(A) is calling fun(B)
		funB(n - 1);
	}
}

void funB(int n)
{
	if (n > 1) {
		cout <<" "<< n;

		// fun(B) is calling fun(A)
		funA(n / 2);
	}
}

int main()
{
	funA(20);
	return 0;
}
```

```
fibonacci_tail_recursive(n, a=0, b=1)
  if (n == 0)
     return a
  else
     return fibonacci_tail_recursive(n - 1, b, a + b);
```

tail recursion

```
fibonacci_tree_recursive(n)
  if (n <= 0)
     return 0;
  elseif (n == 1)
     return 1;
  else
     return fibonacci_tree_recursive(n - 1) + fibonacci_tree_recursive(n - 2)
```

tree recursion

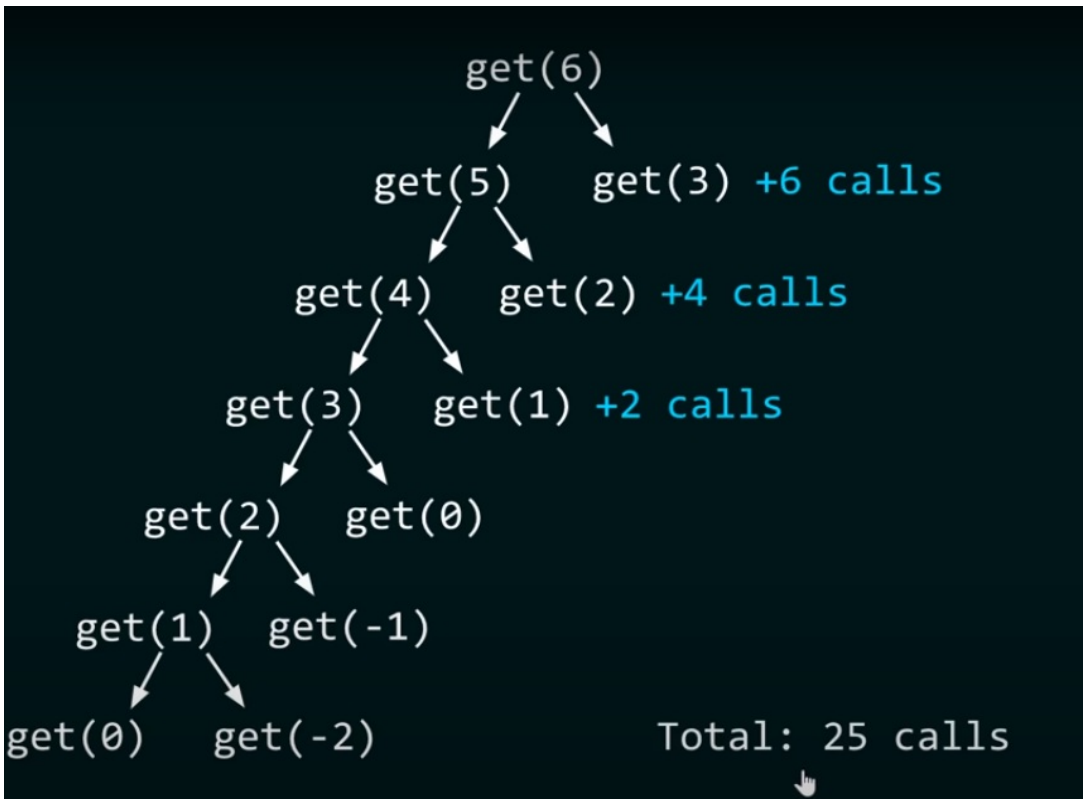
Consider the following recursive C function:

```
void get(int n) {
    if(n<1) return;
    get(n-1);
    get(n-3);
    printf("%d",n);
}
```

If get(6) function is being called in main() then how many times will the get() fun
be invoked before returning to the main()?

(A) 15 (B) 25
(C) 35 (D) 45

[GATE 2015 - 2 Mar



Determine, how many number of times the star will be printed on the screen:

```
void fun1(int n)
{
int i = 0;
if (n > 1)
    fun1(n-1);
for (i = 0; i < n; i++)
    printf(" * ");
}
```

a) n
b) n(n+1)/2
c) n*n
d) None of the above

For n = 2

Star will be printed 1+2 = 3 times

For n = 3

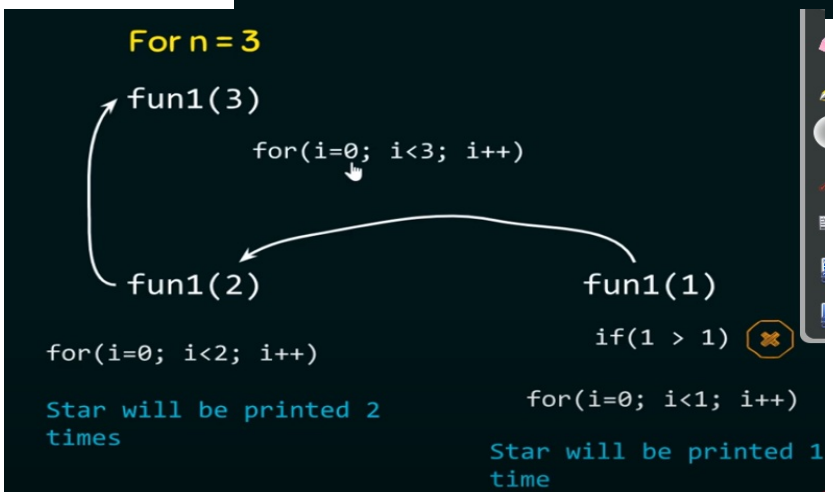
Star will be printed 1+2+3 = 6 times

For n = 4

Star will be printed 1+2+3+4 = 10 times

For n = k

Star will be printed $1 + 2 + 3 + 4 + \ldots + k = k(k+1)/2$



Consider the following C function:

```
int fun(int n)
{
  int x=1, k;
  if (n==1) return x;
  for (k=1; k<n; ++k)
     x = x + fun(k) * fun(n - k);
  return x;
}
```

The return value of fun(5) is

a) 0
b) 26
c) 51
d) 71

[GATE 2015 (Set 2)]

n = 4

1+1+1+1 = 4

1+1+2 = 4

1+2+1 = 4

2+1+1 = 4

2+2 = 4

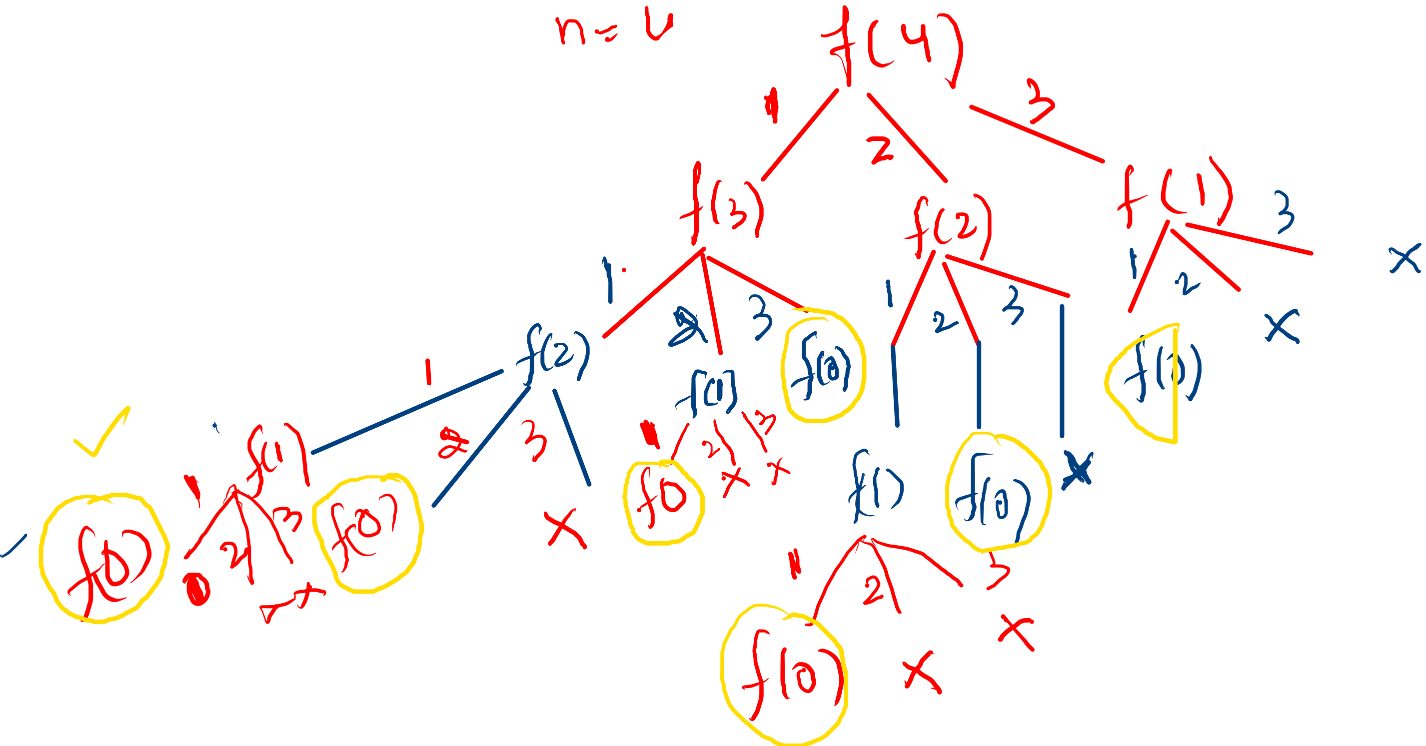
3+1 = 4

1+3 = 4

Stair case Problem

There are **n** stairs, a person standing at the bottom wants to climb stairs to reach the **nth** stair. The person can climb either **1** stair or **2** or 3 stairs stairs at a time, the task is to count the number of ways that a person can reach at the top.

```
#include<iostream>
using namespace std;
int step(int n)
{
    if(n==0)
    {
        return 1;
    }
    else if(n<0)
    {
        return 0;
    }
    return step(n-1)+step(n-2)+step(n-3);
}
int main()
{
   int n=4;
  cout<< step(n);
}
```


n = 4
f(4)
f(3)
f(2)
f(1)
f(0)

1
2
3
4
f(4)
f(3)
f(2)
f(1)
f(0)