Recursion

Tags	
Created time	@November 14, 2022 10:58 PM
Created time 1	@November 14, 2022 10:58 PM
Last edited time	@November 19, 2022 2:07 AM
Status	In progress
URL	
URL 1	

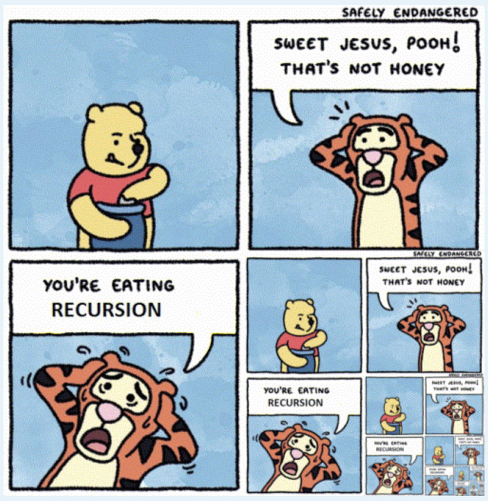
Definition - The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are **Towers of Hanoi (T.O.H.)**, **In-order/Pre-order/Post-order Tree Traversals**, **DFS of Graph**, etc. A recursive function solves a particular problem by calling a copy of itself and solving smaller sub-problems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.

src - full article (GfG).

Recursive can be easily implemented we just have to take of three objectives.

- Base Case (Solve for smallest sub-problem) [without this we will get a infinite recursion.]
- Induction Hypothesis (Assume for n-1)
- Induction Step (Solve for nth step)

💡 infinite recursion is a condition when we have not provided proper base case we will get a infinitive recursive calls like a never ending loop until we close it forcefully or the system memory is full.



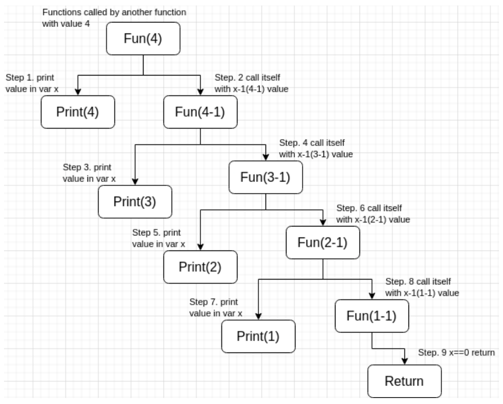
Example of Recursion

```
void Fun(int x){
  if(x < 1) return;//base-case
  print(x);
  Fun(x-1); //Recursive call
}
```

Tracing Recursion

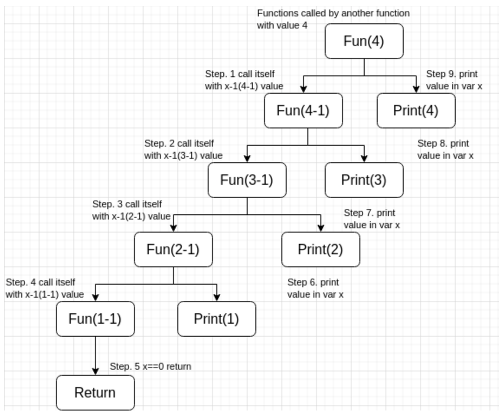
Recursive functions are traced in the form of a tree(graphical).

```
void Fun(int x){
  if(x < 1) return;//base-case
  print(x);
  Fun(x-1); //Recursive call
}
int main(){
  Fun(4);
}
```


Functions called by another function with value 4
Fun(4)
Step 1. print value in var x
Print(4)
Step. 2 call itself with x-1(4-1) value
Fun(4-1)
Step 3. print value in var x
Print(3)
Step. 4 call itself with x-1(3-1) value
Fun(3-1)
Step 5. print value in var x
Print(2)
Step. 6 call itself with x-1(2-1) value
Fun(2-1)
Step 7. print value in var x
Print(1)
Step. 8 call itself with x-1(1-1) value
Fun(1-1)
Step. 9 x==0 return
Return

```
console output: 4 3 2 1
```

```
void Fun(int x){
  if(x < 1) return;//base-case
  Fun(x-1); //Recursive call
  print(x);
}
int main(){
  Fun(4);
}
```


Functions called by another function with value 4
Fun(4)
Step. 1 call itself with x-1(4-1) value
Step 9. print value in var x
Fun(4-1)
Print(4)
Step. 2 call itself with x-1(3-1) value
Step 8. print value in var x
Fun(3-1)
Print(3)
Step. 3 call itself with x-1(2-1) value
Step 7. print value in var x
Fun(2-1)
Print(2)
Step. 4 call itself with x-1(1-1) value
Step 6. print value in var x
Fun(1-1)
Print(1)
Step. 5 x==0 return
Return

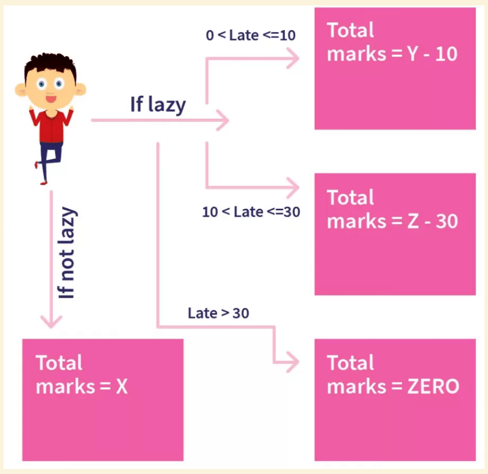
```
console output: 1 2 3 4
```

---

these diagram are used to predict the output of recursive function and also helps in designing new recursive algorithms as it gives a very visual and comprehensive visualisation of how recursion will behave with different output.

Like in these both cases with just swapping two we got the exact opposite output. Because they have different control flow.

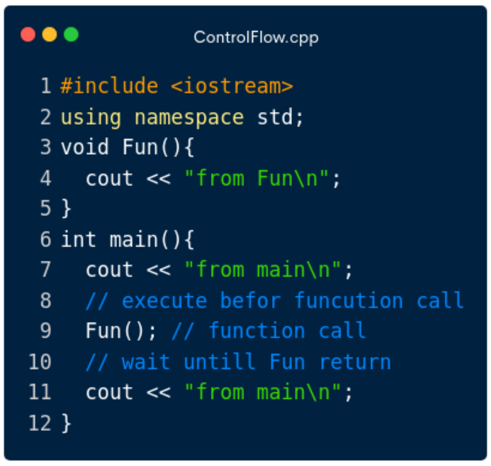
💡 control flow is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.



Control flow of recursive

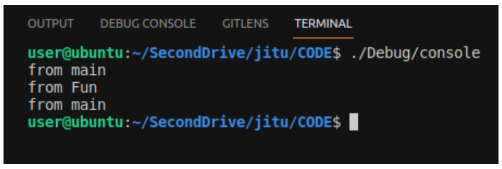
control flow of a recursive function is same as any other function its just important to study the flow of the functions to design any recursive function.

When we run a program in the computer, the computer sees the instructions(code). top to bottom line by line each statements instructions and function calls and inside each statement instruction it reads from left to right then if there are any variable or function calls it it keeps the function its in that time as hold and calls that function evaluates it first then return and then goes to next line. so any expression above function calls will execute before the function and expression after the function call will be on hold until the function return.and to keep track all of that there is a data structure called stack is used.


ControlFlow.cpp

```cpp
#include <iostream>
using namespace std;
void Fun(){
  cout << "from Fun\n";
}
int main(){
  cout << "from main\n";
  // execute befor funcution call
  Fun(); // function call
  // wait untill Fun return
  cout << "from main\n";
}
```

output -



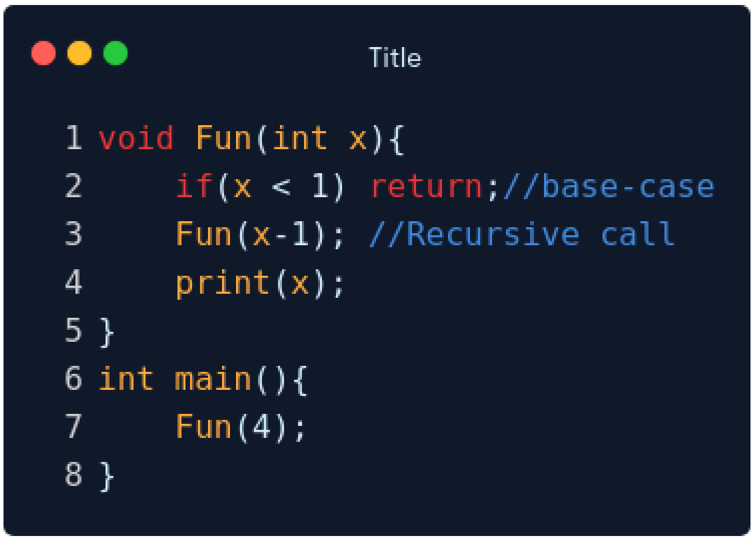
```
user@ubuntu:~/SecondDrive/jitu/CODE$ ./Debug/console
from main
from Fun
from main
user@ubuntu:~/SecondDrive/jitu/CODE$ 
```

> Stack - stack is a user defined linear data structure based on filo (first in last out) mechanism.

How stack is used in system memory

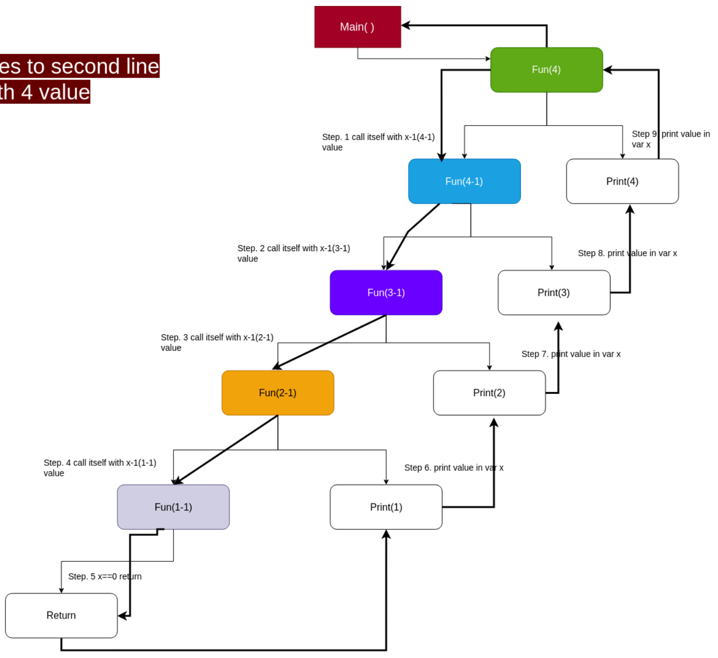
When we execute a program the a chunk of memory gets allocated for our program to load files keep variable etc. divided into parts (Code partition | Static and global variable partition | a part with memory for heap | and a chunk of free space(heap) for dynamic allocation.)it starts with the program with all the function local and global variable then starts loading function in stack starts with main and then if there is any function call inside the main function it creates a new instance in stack with space for new variable so every instance of a function has there own local variable with different addresses. Stack comes very handy when there are nested calls or recursive calls and call stack has multiple instance while returning it has the data every function as there local variable data stored so it doesn't get messed up with other same name variable. but this benefit comes with a price since we are creating new instance every function call we are using extra auxiliary space even if there is no variable in function and since we have a finite memory to run sometimes when the memory is fulled the program crashes and gives the “Stack-overflow” error. and one more down side(well its more like a “feature”)that since we are creating new variables for every function call sometimes we are gonna need some variables to be same so we have to create a function outside of the stack a variable for a function in which every instance of that function will share the same variable and if a function changes it the changes will be for all instance of that function that can be done by explicitly writing static in-front of the declaration and the variable will not be in stack so even if all the instance of the stack for popped and next time we all the function it will not get reset this variable is called static variable.

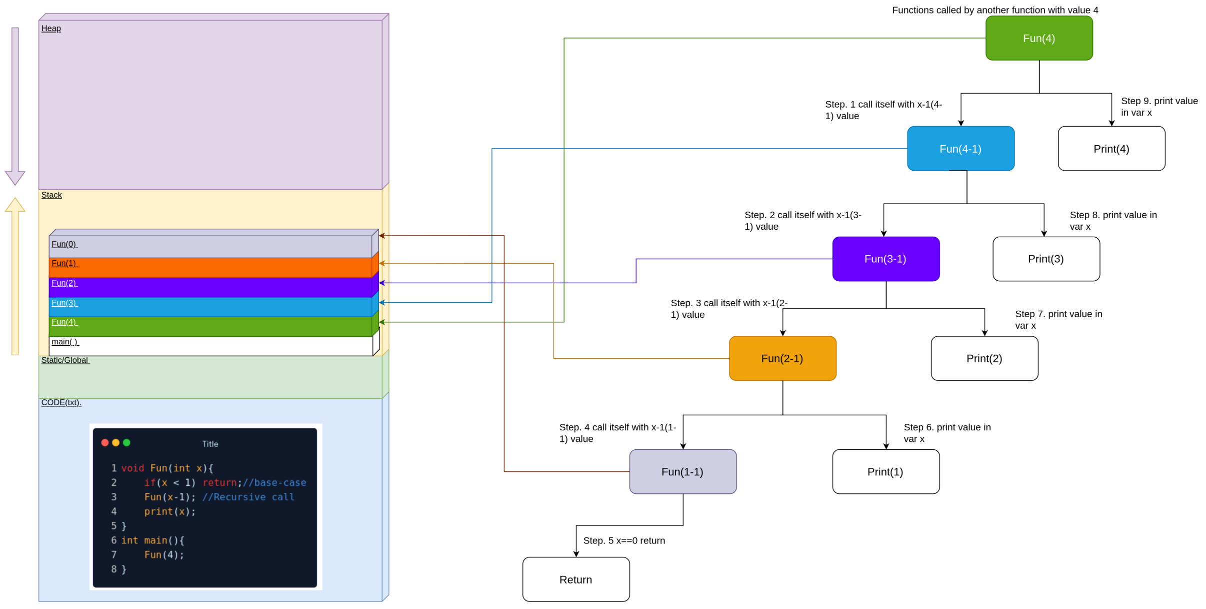
Dry run and visualisation of control flow and how a recursive function uses stack memory with example code.



```
void Fun(int x){
    if(x < 1) return;//base-case
    Fun(x-1); //Recursive call
    print(x);
}
int main(){
    Fun(4);
}
```

Line no 6. Main() -- > program starts with main goes to second line
Line no 7. Main() --> this line will call Fun(int x) with 4 value
Line no 1. Fun(4) --> In function
Line no 2. Fun(4) --> (x < 1) [false]
Line no 3. Fun(4) -- > Fun(x-1) with value 4-1 = 3
Line no 1. Fun(3) --> In function
Line no 2. Fun(3) --> (x < 1) [false]
Line no 3. Fun(3) -- > Fun(x-1) with value 3-1 = 2
Line no 1. Fun(2) --> In function
Line no 2. Fun(2) --> (x < 1) [false]
Line no 3. Fun(2) -- > Fun(x-1) with value 2-1 = 1
Line no 1. Fun(1) --> In function
Line no 2. Fun(1) --> (x < 1) [false]
Line no 3. Fun(1) -- > Fun(x-1) with value 1-1 = 0
Line no 1. Fun(0) --> In function
Line no 2. Fun(0) --> (x < 1) [True] return
Line no 4. Fun(1) --> Print(1) pop the stack
Line no 4. Fun(2) --> Print(2) pop the stack
Line no 4. Fun(3) --> Print(3) pop the stack
Line no 4. Fun(4) --> Print(4) pop the stack
Line no 8. Main() --> return 0;


Main( )
Fun(4)
Step. 1 call itself with x-1(4-1) value
Step 9. print value in var x
Fun(4-1)
Print(4)
Step. 2 call itself with x-1(3-1) value
Step 8. print value in var x
Fun(3-1)
Print(3)
Step. 3 call itself with x-1(2-1) value
Step 7. print value in var x
Fun(2-1)
Print(2)
Step. 4 call itself with x-1(1-1) value
Step 6. print value in var x
Fun(1-1)
Print(1)
Step. 5 x==0 return
Return


Heap
Stack
Fun(0)
Fun(1)
Fun(2)
Fun(3)
Fun(4)
main( )
Static/Global
CODE(txt)
Title
Functions called by another function with value 4
Fun(4)
Step. 1 call itself with x-1(4-1) value
Step 9. print value in var x
Fun(4-1)
Print(4)
Step. 2 call itself with x-1(3-1) value
Step 8. print value in var x
Fun(3-1)
Print(3)
Step. 3 call itself with x-1(2-1) value
Step 7. print value in var x
Fun(2-1)
Print(2)
Step. 4 call itself with x-1(1-1) value
Step 6. print value in var x
Fun(1-1)
Print(1)
Step. 5 x==0 return
Return

```
1 void Fun(int x){
2     if(x < 1) return;//base-case
3     Fun(x-1); //Recursive call
4     print(x);
5 }
6 int main(){
7     Fun(4);
8 }
```

Time And Space Complexity

In this illustration we can see when we call the function with ‘4’ input it created 4+1 instance in stack memory so we can say that for N input it will create N+1 function call thus N+1 instance in the stack memory from that we can get the space complexity from as the degree for this is n the space complexity will be O(n). and inside each call there are fixed(constant) amount of operation going on to each function’s time complexity is O(1) and since we are calling the function n time the total complexity of the program will be N * O(1) ⇒ O(n).

Types of recursion

1. tail recursion
2. head recursion
3. Indirect recursion
4. nested recursion
5. tree recursion

Tail Recursion

when a recursive function has the recursive call at the very end of the body its called a tail recursion.

example.

```
void Func(int x){
  if(!x) return; //base case
  cout << x << endl;
  Func(x-1);
}
```

Head Recursion

when a recursive function has the recursive call in first line(ofc after base-case other wise it will become a infinite recursive function) its called a head recursion.

```
void Func(int x){
  if(!x) return; //base case
  Func(x-1);
  cout << x << endl;}
```

💡 Since recursive functions use linear space our goal is to always find if the same functionality can be archived iterative function by using loops and most of time it is possible with head and tail recursive functions.

ex tail function to iterative

```
void Func(int x){
  if(!x) return; //base case
  cout << x << endl;
  Func(x-1);
}
```

```
void Func(int x){
  while(x > 0)
    cout << x-- << endl;
}
```

ex. head function to iterative

```
void Func(int x){
  if(!x) return; //base case
  Func(x-1);
  cout << x << endl;}
```

```
void Func(int x){
  int i = 0;
  while(i < x)
    cout << i++ << endl;
}
```

Indirect Recursion

its happen when two or more function call each other. despite not looking like a regular recursive function this create a recursive order flow in a circular fashion and because they are not calling them self directly thus the name “indirect” recursive.

example.

```
void Fun1(int x){
  if(x < 1) return;
  cout << x << endl;
  Fun2(x-1);
}
void Fun2(int x){
  if(x < 1) return;
  cout << x << endl;
  Fun1(x/2);
}
```

Nested recursion

When a function(non-void) has a recursive call in returning statement its called Nested recursion.

example.

```
int Factorial(int x)
{
  if(x==2) return x;
  return x * Factorial(x-1);
}
```

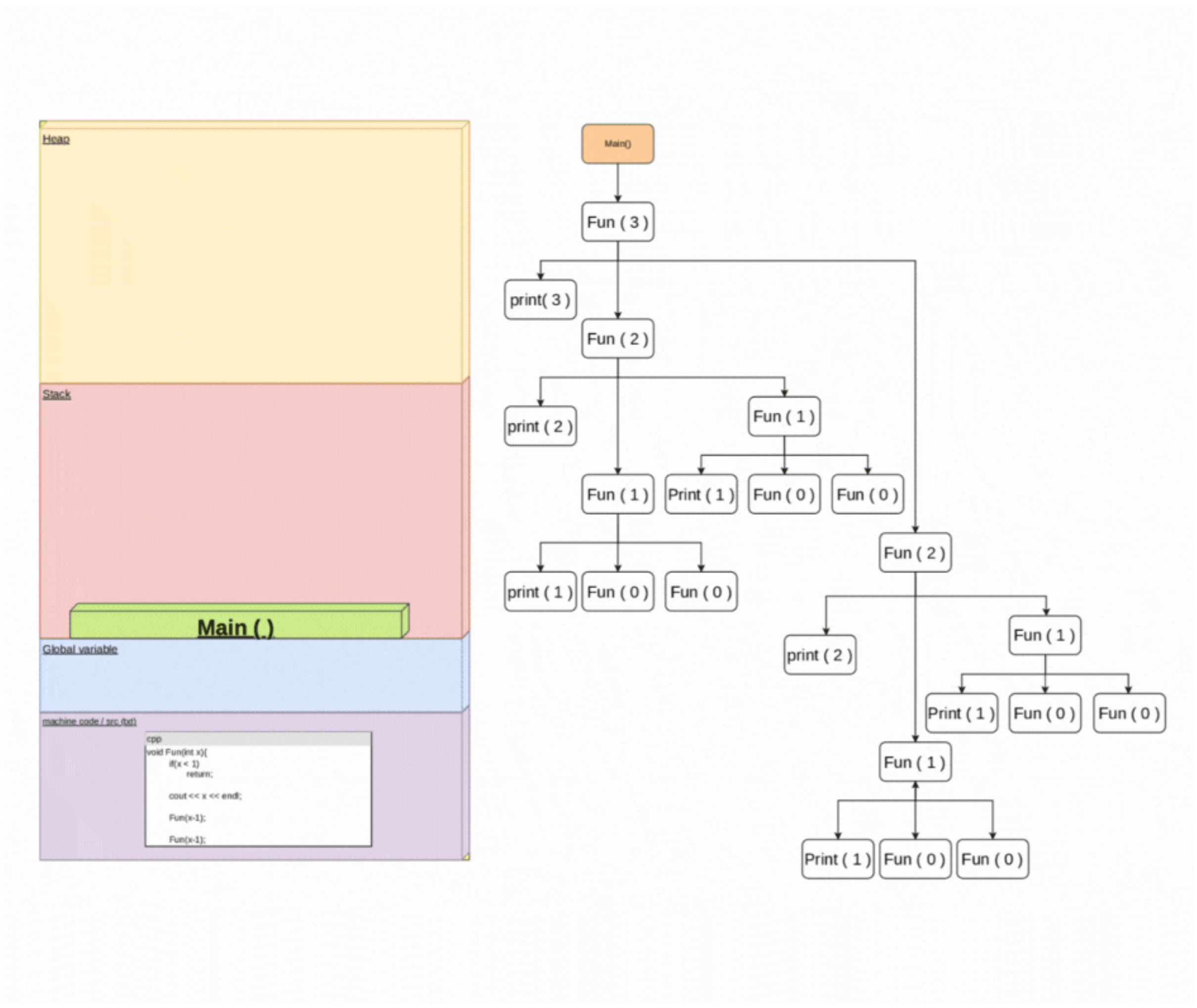
Tree recursion

When a function calls has more than one recursive call its called tree recursion.

example.

```
void Fun(int x){
  if(x < 1) return;
  cout << x << endl;
  Fun(x-1);
  Fun(x-1);
}
```

because it is calling itself more than one time this type of function can me very slow like exponentially or some times factorial time complexity though space complexity is not that much affected by this because that is more dependent on the base-case and input.



```cpp
void Fun(int x){
	if(x < 1)
		return;

	cout << x << endl;

	Fun(x-1);

	Fun(x-1);
```

Practice -

▼ print string n times

Description - (self explanatory)

```
void printNT(string str,int x,int i = 0)
{
  if(i > x) return;
  print(str);
  printNT(str,x,i+1);
}
```

▼ Print N to 1

Description -(self explanatory)

```
void printNum(int N)
{
  if(N<1) return;
  printNum(N-1);
  print(N);
}
```

▼ Print 1 to N

Description -(self explanatory)

```
void printNum(int x)
{
  if(x<1) return;
  printNum(x-1);
  print(x);
}
```

▼ Summation the first N numbers

description - return sum from 1 to N

```
Example -
input - 5
output - 15
-----------
explanation - 1 + 2 + 3 + 4 + 5 = 15
```

```
//code -
int sum(int x)
{
  if(x == 1) return 1;
  return x + sum(x-1);
}
```

▼ Factorial of a Number

Description - return product of number from 1 to N

```
Example -
Input - 5
Output - 120
---------------
explanation - 1 x 2 x 3 x 4 x 5 = 120
```

```
//code
int Fact(int x)
{
  if(x == 2) return 2;
  return x * Fact(x-1);
}
```

▼ Reverse an Array

Description - SE

```
void reverse(int arr[],int l,int f = 0)
{
  if(f>=l) return;
  swap(arr[f],arr[l]);
  reverse(arr,l-1,f+1);
}
```

▼ Palindrome Check

Description - check if given array is palindrome or not

> palindrome - a string on reversal reads the same

```
example.
input - 1 2 3 2 1
output - true
-----------------------
input - 1 2 3 4 2
output - false
```

```
bool checkPalindrome(int arr[],int l,int f = 0)
{
  if(f>=l) return true;
  if(arr[f] != arr[l]) return false;
  return checkPalindrome(arr,l-1,f+1);
}
```

▼ Fibonacci number

Description - return nth Fibonacci number.

```
int fib(int x)
{
  if(x < 2) return x;
  return fib(x-1) + fib(x-2);
}
```