ICPC Assiut Community

Newcomers Training

Recursion

ICPC Assiut
community

Recursion

- What is the mean of **Recursive** operation ?
  - It's mean that we decompose a problem to a set of sub-problems.
  - If the sup-problem is the same type as the problem but not the same constrain, this operation called **Recursion**.

- Example: Recall the factorial :
  - factorial(6) = **1 * 2 * 3 * 4 * 5** * 6
  - factorial(5) = **1 * 2 * 3 * 4** * 5
  - factorial(4) = **1 * 2 * 3** * 4
  - factorial(3) = **1 * 2** * 3
  - factorial(2) = **1** * 2
  - factorial(1) = 1

    - what is the relation between factorial(6) and factorial(5) ?

Factorial - Normal function code

```cpp
#include<iostream>
using namespace std;
int factorial(int n){

    int ans = 1;

    for(int i = 2; i <= n; i++){
        ans *= i;
    }
    return ans;
}
int main(){
    cout << factorial( 3) << endl; // 1 * 2 * 3 = 6
    cout << factorial( 4) << endl; // 1 * 2 * 3 * 4 = 24
                                   // factorial(3) * 4 = 24

    cout << factorial( 5) << endl; // 1 * 2 * 3 * 4 * 5 = 120
                                   // factorial(4) * 5 = 120

    cout << factorial( 6) << endl; // 1 * 2 * 3 * 4 * 5 * 6 = 720
                                   // factorial(5) * 6 = 720
    return 0;
}
```

Recursion

- **More Explain :**
  - We want to know the value of **factorial(6)** , *that's our problem !*
  - if we know the value of **factorial(5)** , would it help us ? **YES**
    how ?
    factorial(5) * 6 = factorial(6)
  - same logic for **factorial(5) = factorial(4) * 5** >> **factorial(n) = factorial(n-1) * n**
- Note : we decompose our problem to a sub-problems, **but when we stop doing that ?**
  - There is a case where no more sub-problems. we call it **base case**
    - Ex : **factorial(1) = 1** >> No need to decompose it

Factorial - Simple Recursion code

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

int factorial_1(){
    return 1; // base case. No sub problems
}
int factorial_2(){
    return factorial_1() * 2;
}
int factorial_3(){
    return factorial_2() * 3;
}
int factorial_4(){
    return factorial_3() * 4;
}
int factorial_5(){
    return factorial_4() * 5;
}

int main(){
    cout << factorial_5() << endl;
    return 0;
}
```

Factorial - Recursion code

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

int factorial(int n){
    if(n == 1)
        return 1; // base case. No sub problems

    return factorial(n - 1) * n;
}
int main(){
    cout << factorial(5) << endl;
    return 0;
}
```

- **Tracing the code:**
- We call factorial(5)
  - if 5 == 1 ? **False**
  - call **factorial(4) and multiply the result with 5**
- We call factorial(4)
  - if 4 == 1 ? **False**
  - call **factorial(3) and multiply the result with 4**
- We call factorial(3)
  - if 3 == 1 ? **False**
  - call **factorial(2) and multiply the result with 3**
- We call factorial(2)
  - if 2 == 1 ? **False**
  - call **factorial(1) and multiply the result with 2**
- We call factorial(1)
  - if 1 == 1 ? **True (base case)**
  - **Return 1**

Recursion - More Examples

- Given an integer number ***N*** and print the following triangle shape with ***N*** Levels :

  N = 5

```
* * * * *
* * * *
* * *
* *
*
```

Recursion - Print Triangle code

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

void print_triangle(int level){
    if(level == 0)
        return; // base case. No sub problems
    for(int i = 1; i <= level; i++){
        cout << "* ";
    }
    cout << endl;
    print_triangle(level - 1);
}
int main(){
    print_triangle(5);
    return 0;
}
```

- **Tracing the code:**
  - We call print_triangle(5) :
    - if 5 == 0 ? False
    - Print 5 stars and go to the next line
    - call print_triangle(4)
  - We call print_triangle(4) :
    - if 4 == 0 ? False
    - Print 4 stars and go to to next line
    - call print_triangle(3)
  - We call print_triangle(3) :
    - if 3 == 0 ? False
    - Print 3 stars and go to the next line
    - call print_triangle(2)
  - We call print_triangle(2) :
    - if 2 == 0 ? False
    - Print 2 stars and go to to next line
    - call print_triangle(1)
  - We call print_triangle(1) :
    - if 1 == 0 ? False
    - Print 1 star and go to to next line
    - call print_triangle(0)
  - We call print_triangle(0) :
    - if 0 == 0 ? True
    - Return

Recursion - More Examples

- Try to print this triangle using Recursion function:

N = 5

```
*
* *
* * *
* * * *
* * * * *
```

Solution

Recursion - More Examples

- **Print (3N + 1)'s Sequence** :

- Start from number ***N***.
- **if this number is even**, the next number in the sequence is **(N / 2)**
- **if this number is odd**, the next number in the sequence is **(3 * N + 1)**
- **if this number is 1, the sequence end.**

- Ex : start from ***N*** = 5 : 5 16 8 4 2 1
- Ex : start from ***N*** = 9 : 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

- Write a **recursive function** that print this sequence.

Recursion - (3N + 1)'s Sequence code

```
#include<iostream>
using namespace std;
void print(int n){
    cout << n << " ";

    if(n == 1)
        return; // base case. No sub problems

    if(n % 2 == 0){
        print(n / 2);
    }
    else{
        print(3 * n + 1);
    }
}
int main(){
    print(5);
    return 0;
}
```

- **Tracing the code:**
  - We call print(5) :
    - **Print 5**
    - if 5 == 1 ? **False**
    - **5 is odd number >> call print(3 * 5 + 1)**
  - We call print_triangle(16) :
    - **Print 16**
    - if 16 == 1 ? **False**
    - **16 is even number >> call print(16 / 2)**
  - We call print_triangle(8) :
    - **Print 8**
    - if 8 == 1 ? **False**
    - **8 is even number >> call print(8 / 2)**
  - We call print_triangle(4) :
    - **Print 4**
    - if 4 == 1 ? **False**
    - **4 is even number >> call print(4 / 2)**
  - We call print_triangle(2) :
    - **Print 2**
    - if 2 == 1 ? **False**
    - **2 is even number >> call print(2 / 2)**
  - We call print_triangle(1) :
    - **Print 1**
    - if 1 == 1 ? **True**
    - **Return**

Recursion - More Information

- **How memory is allocated to different function calls in recursion?**
  - 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 calling function and 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.

- **Why Stack Overflow error occurs in recursion?**
  - **If the base case is not reached or not defined**, then the stack overflow problem may arise.

Recursion - More Examples

- Ex : Fibonacci series

$$F(n) \begin{cases} 0 & \text{if } (n == 0) \\ 1 & \text{if } (n == 1) \\ F(n-1) + F(n-2) & \text{if } (n >= 2) \end{cases}$$

F = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …}

- Given ***N*** , print the ***F[N]*** value in the fibonacci series.

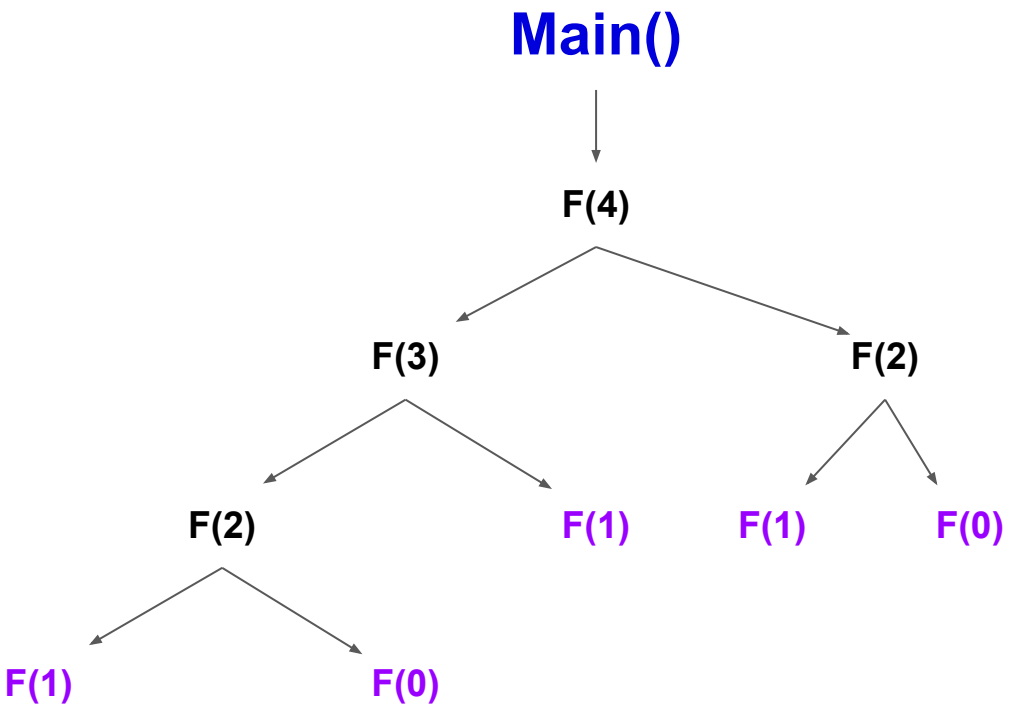
Recursion - Fibonacci

- **First**, What's our **base case** ?
  - if(n == 1) the answer is : 1
  - if(n == 0) the answer is : 0

- **Second**, if(n > 2) what's the functions we will call ?
  - **fib(n) = fib(n - 1) + fib(n - 2)**

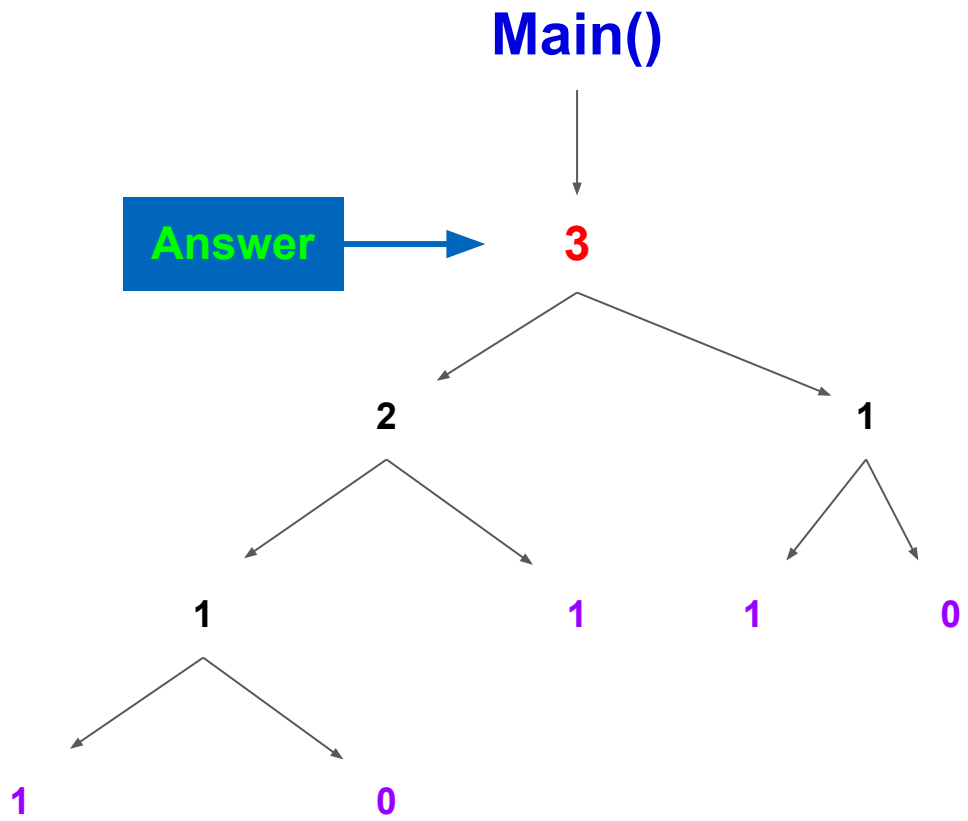
- **Let's draw a tree to trace it**

Recursion - Fibonacci




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

Recursion - Fibonacci




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

Recursion - Fibonacci

- How many calls does this function calls in every node ?
  - It calls **two functions** , that's mean :
    F[0] and F[1] no need to call, because they are Base case
    F[3] needs 4 calls
    F[4] needs 8 calls
    F[5] needs 16 calls

    - So, F[***n***] needs ***2^(n - 1) calls*** (***n*** is index), **We can ignore (n-1) to (n)**

      So the Complexity = O(2^n)

- **The Complexity depends on the number of nodes and the number of calls in each node**

Recursion

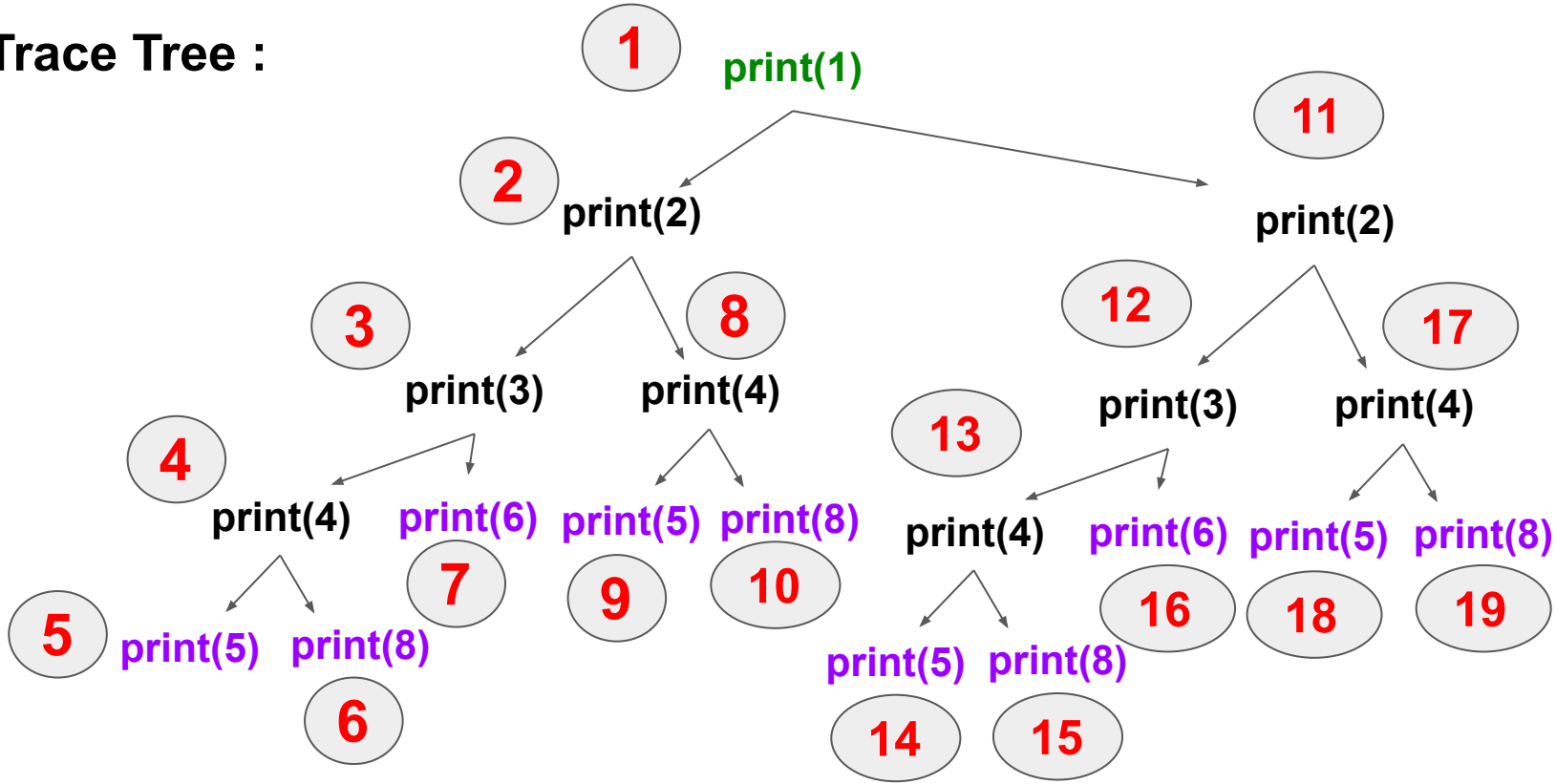
- What is the Output of this code ?
- Draw the trace tree for this code.

```
#include<iostream>
using namespace std;
void print(int x){
    if(x > 4){
        return;
    }
    cout << x << " ";
    print(x + 1);
    print(x * 2);
}
int main(){
    print(1);
    return 0;
}
```

Recursion

- `Output :1 2 3 4 4 2 3 4 4`

- **Trace Tree :**



For more information about **Recursion** visit this Link

Now it's time to practise and solve the problems of Recursion

Recursion Sheet

Good luck <3