Recursion

Problem: Count your Position in Line

You are standing in a Cashier Line and you want to count at what number you are standing.

Problem: Count your Position in Line

How can you count at which number you are standing in the line?

Solution 1: Count your Position in Line

One way is to start the count from the start of the line and then count the number of people till your position.

Solution 1: Count your Position in Line

Now, let's generate the data for the problem.

```
struct Person
{
    Person * next;
};
```

```
Person* generateData()
{
    Person * p1 = new Person();
    Person * p2 = new Person();
    Person * p3 = new Person();
    Person * p4 = new Person();
    p4->next = p3;
    p3->next = p2;
    p2->next = p1;
    p1->next = NULL;
    return p4;
}
```

```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```

Solution 1: Count your Position in Line

Now, let's see the solution of the problem.

```
int positionInLine(Person *person)
{
    Person * temp = person;
    int count = 0;
    while(temp != NULL)
    {
        count++;
        temp = temp->next;
    }
    return count;
}
```

Solution 2: Count your Position in Line

Let's see another lazy solution in which you don't want to do all the work.

Solution 2: Count your Position in Line

You call the next person from you and ask him/her at which position he/she is standing.

Solution 2: Count your Position in Line

He/She doesn't know his/her position, so he/she calls the next person from him/her.

Solution 2: Count your Position in Line

He/She also doesn't know his/her position, so he/she calls the next person from him/her.

Solution 2: Count your Position in Line

The first person knows that he/she is the number 1 in the line so he/she says I'm number 1.

Solution 2: Count your Position in Line

Previous person from the start then adds 1 in the number and says i'm number 2.

Solution 2: Count your Position in Line

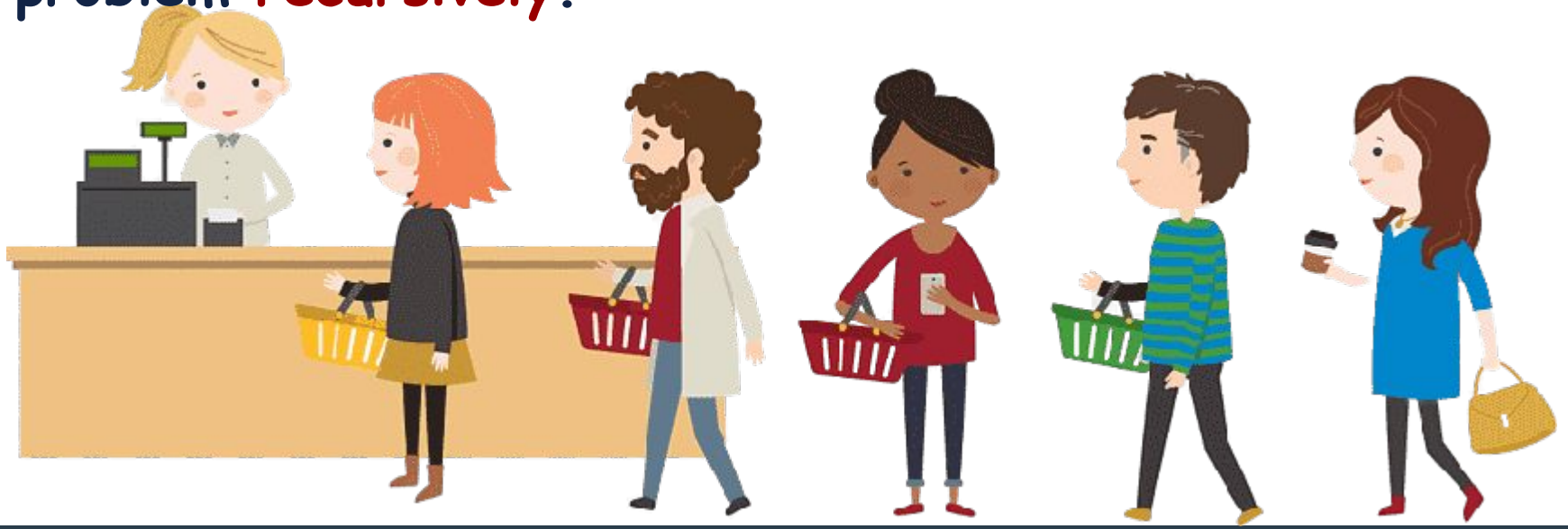
Previous person then adds 1 in the number and says i'm number 3.

Solution 2: Count your Position in Line

Now, you add 1 in the number and knows you are number 3.

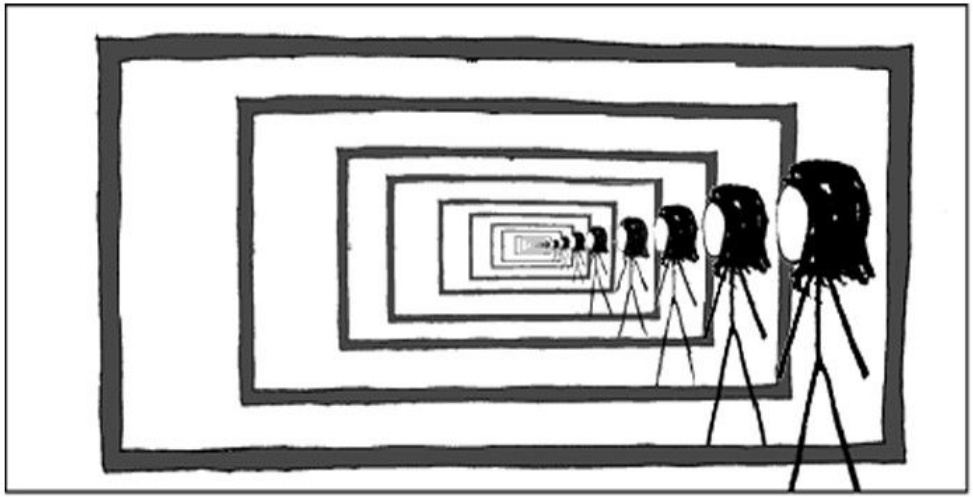
Count your Position in Line

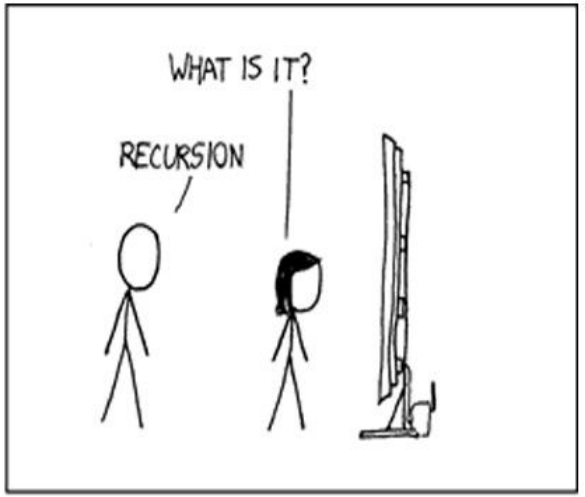
In the first Solution, you were solving the problem **iteratively**. In the second Solution, you are solving the problem **recursively**.



Recursion

Recursion is the technique of making a function call itself.





Recursion

Before moving to solve the problem recursively, let's see a simple program first.

What will be the Output?

```
int func3()
{
    int a = 1;
    return a;
}

int func2()
{
    int a = 1 + func3();
    return a;
}
```

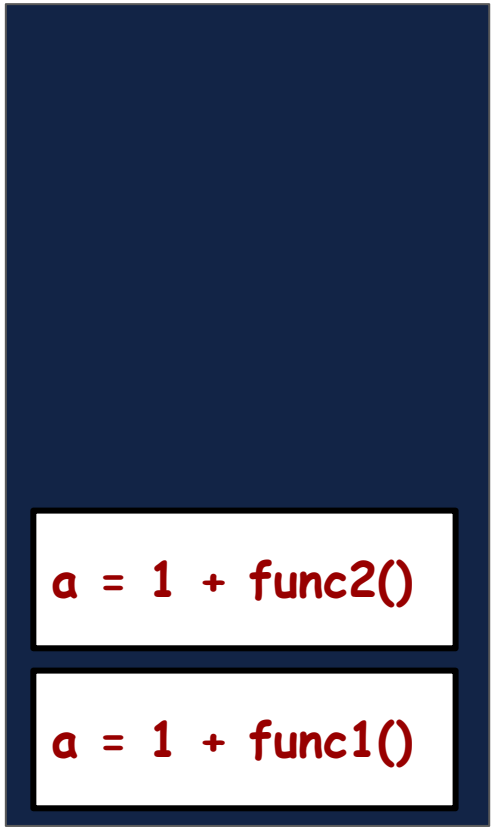
```
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```

Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```

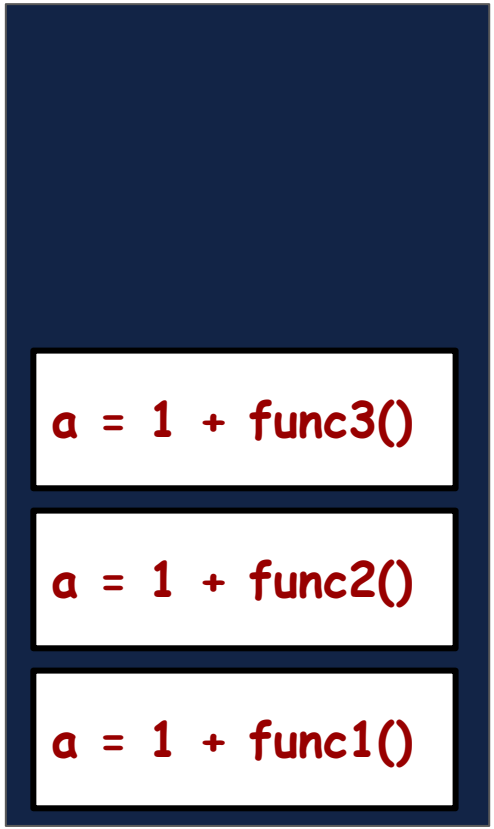
Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```



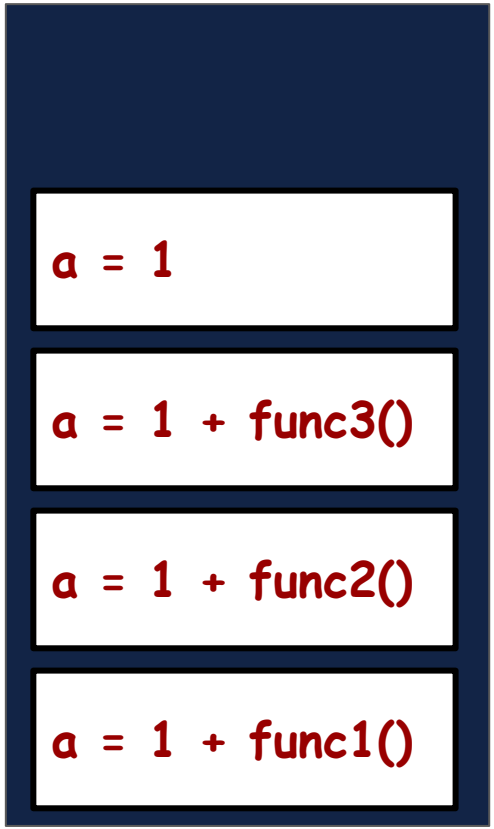
Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```



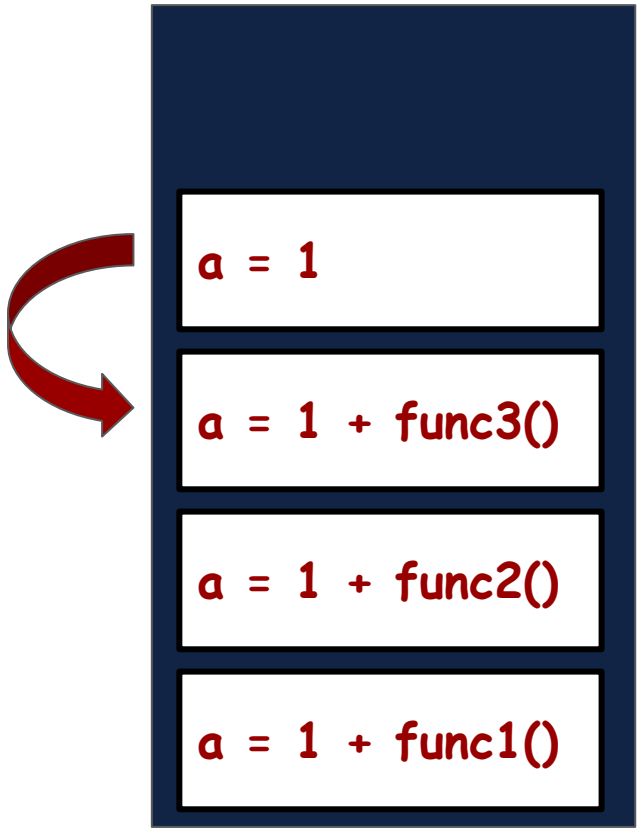
Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```



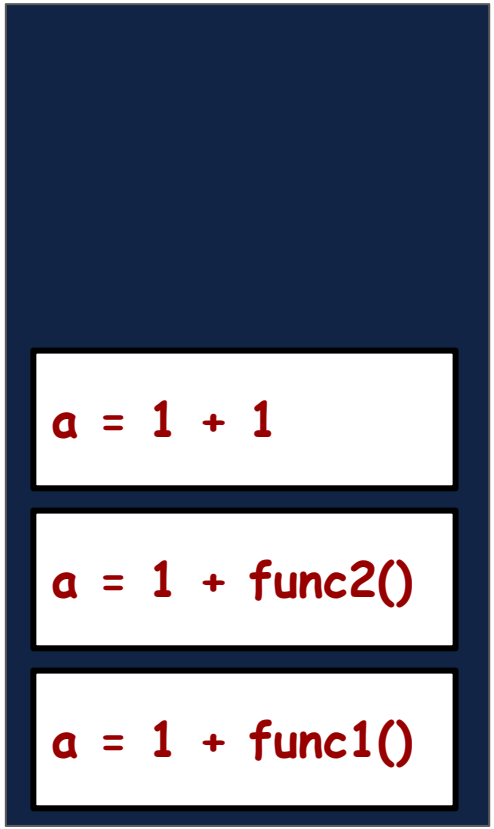
Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```



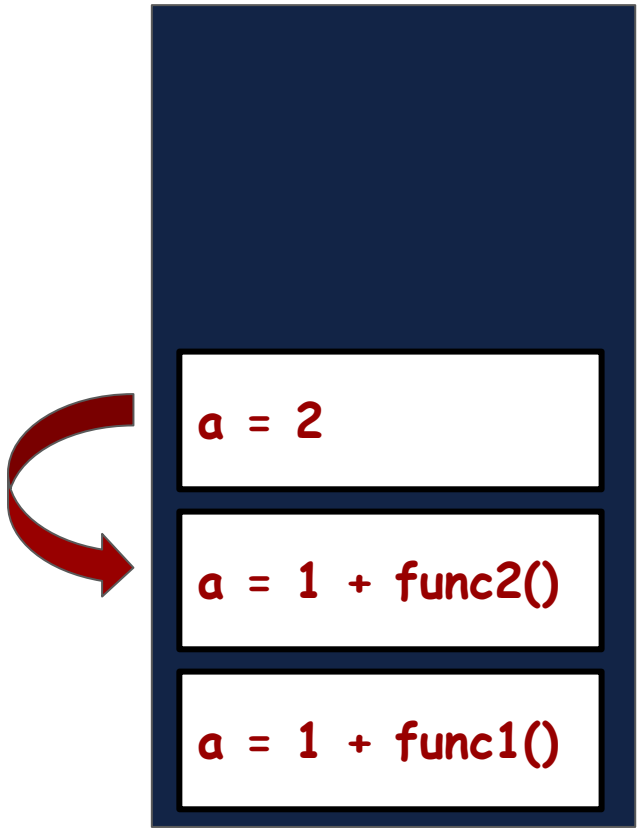
Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```



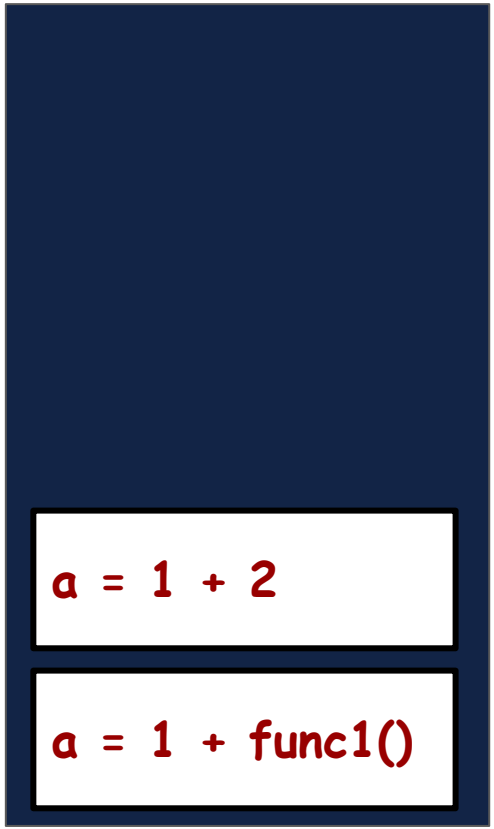
Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```



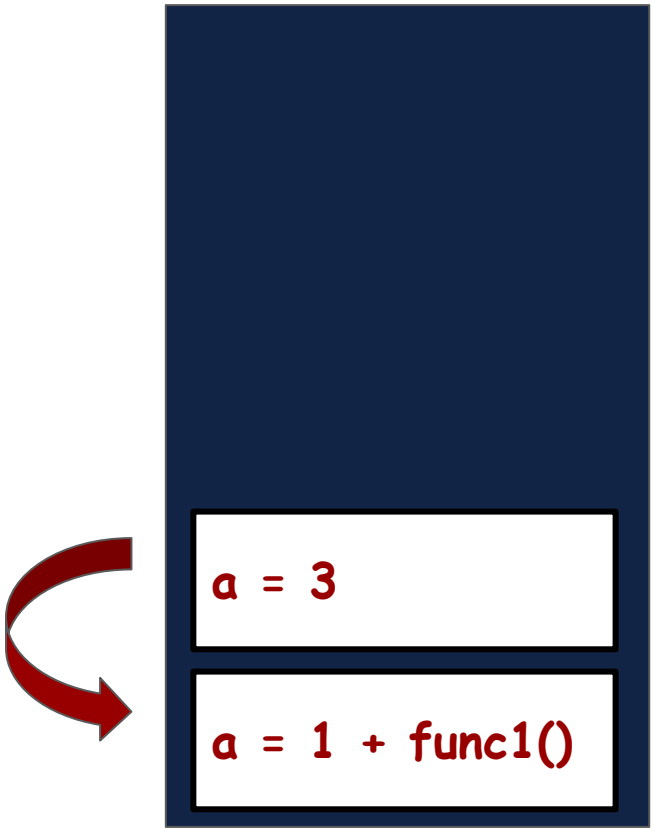
Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```



Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```



Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```

Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```

Lets see the Stack Calls

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```

I'm number 4

Repeating Statement

In this example, do you see some condition that keeps repeating in every function?

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```

Repeating Statement

In this example, do you see some condition that keeps repeating in every function?

1 + func();

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```

Stopping Condition

In this example, at what point we stopped calling the functions and started returning from the functions i.e., the function that is different from the rest?

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```

Stopping Condition

In this example, at what point we stopped calling the functions and started returning from the functions i.e., the function that is different from the rest?

Where we found a = 1

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```

2 Important Points

First point is that there is some similar statement that keeps repeating.

Second point is that there is a statement at which we stop calling the functions.

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```

2 Important Points

This is exactly what we do in recursive functions.
We keep on calling the same function itself and we define a terminating condition at which point we have to stop calling the function.

```
int func3()
{
    int a = 1;
    return a;
}
int func2()
{
    int a = 1 + func3();
    return a;
}
int func1()
{
    int a = 1 + func2();
    return a;
}

int main()
{
    int a = 1 + func1();
    cout << "I'm number " << a;
}
```

Recursive Solution

Now, let's see the recursive function of the starting problem.

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

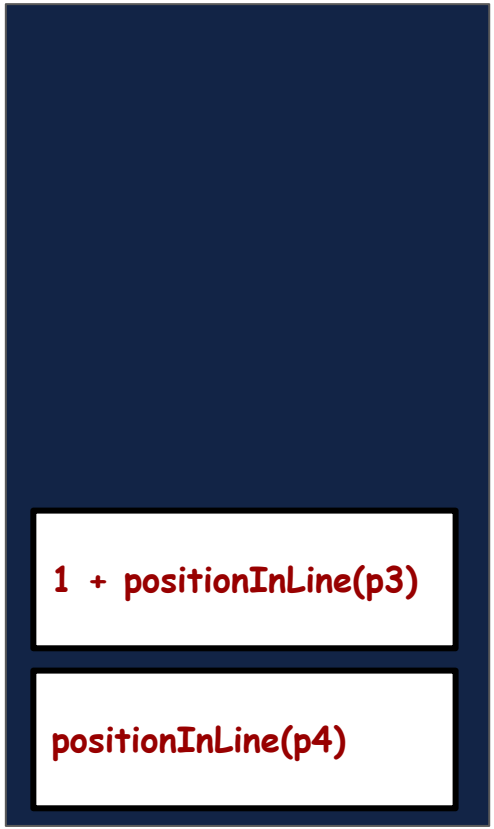
```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```

positionInLine(p4)

Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

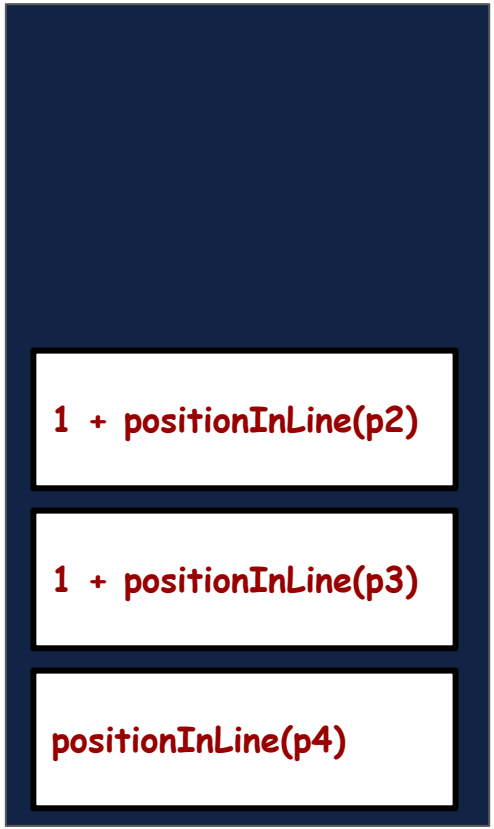
```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```



Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

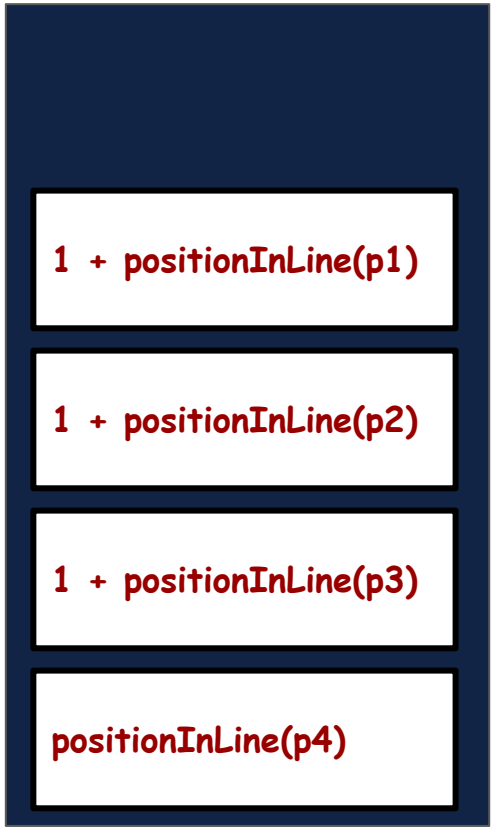
```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```



Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

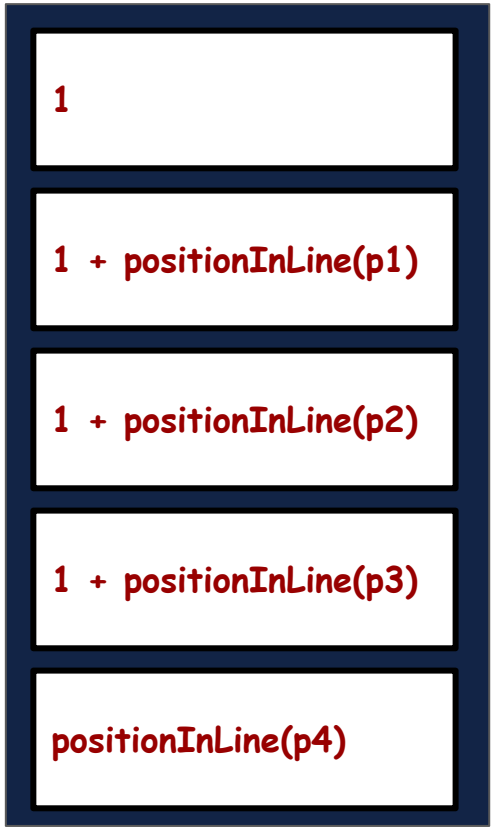
```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```



Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

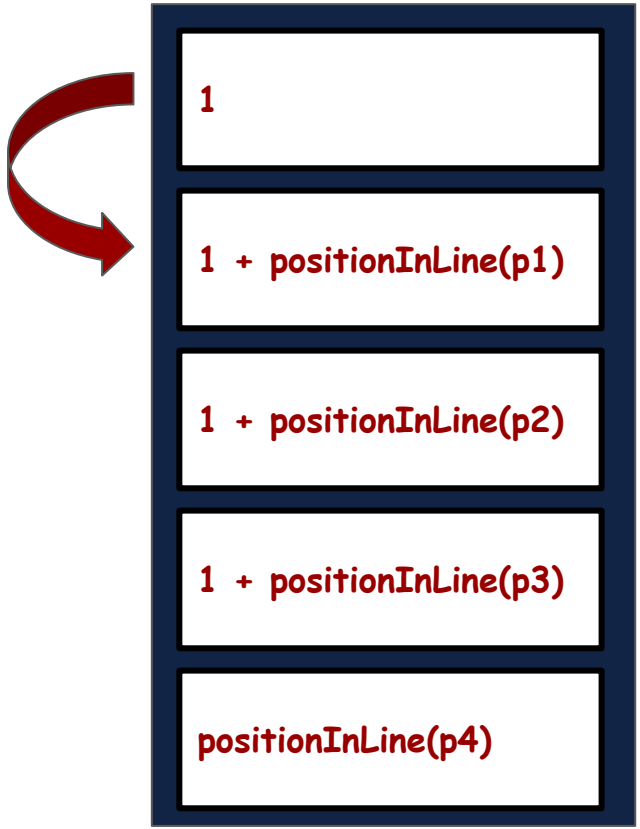
```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```



Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

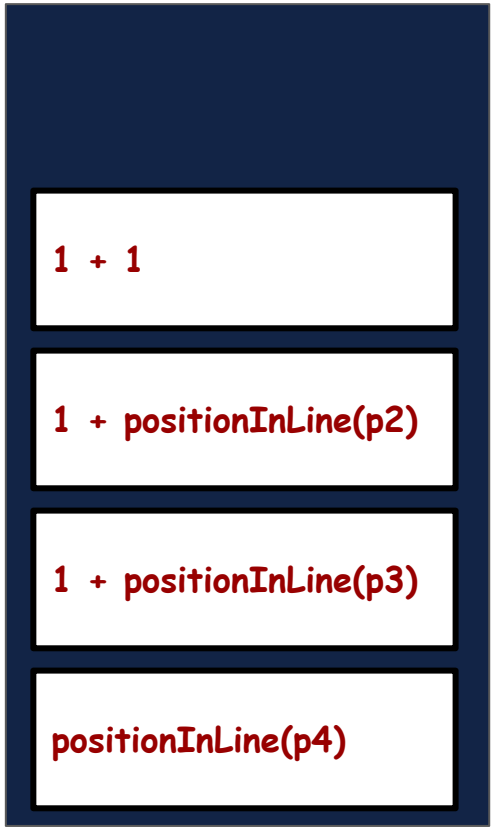
```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```



Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

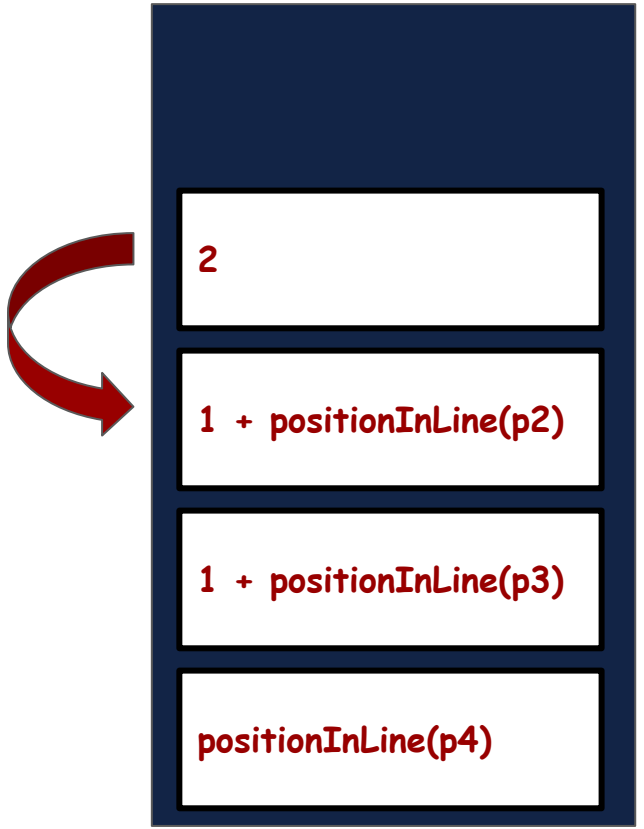
```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```



Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

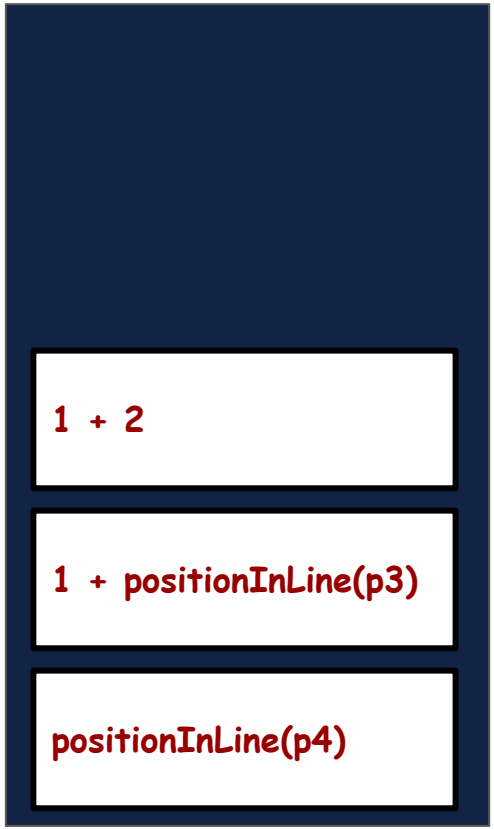
```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```



Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

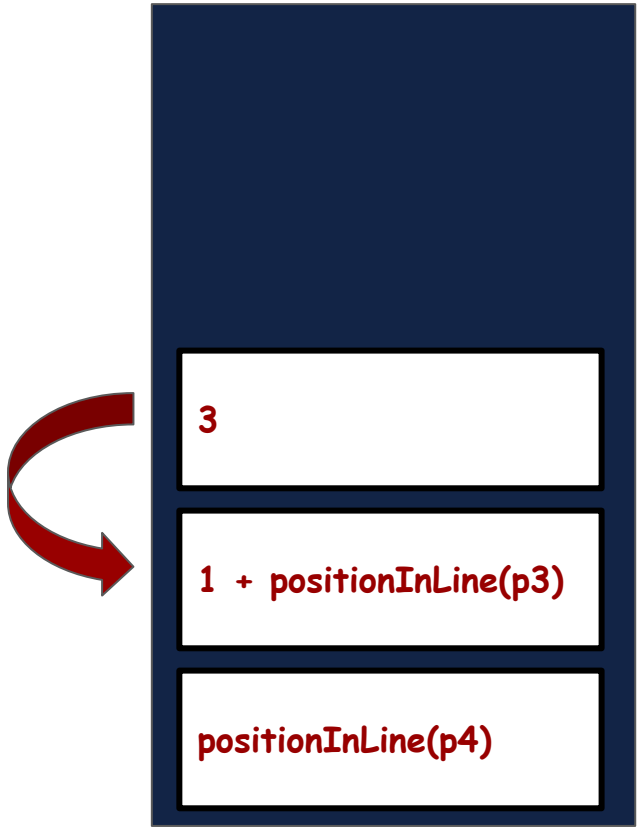
```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```



Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

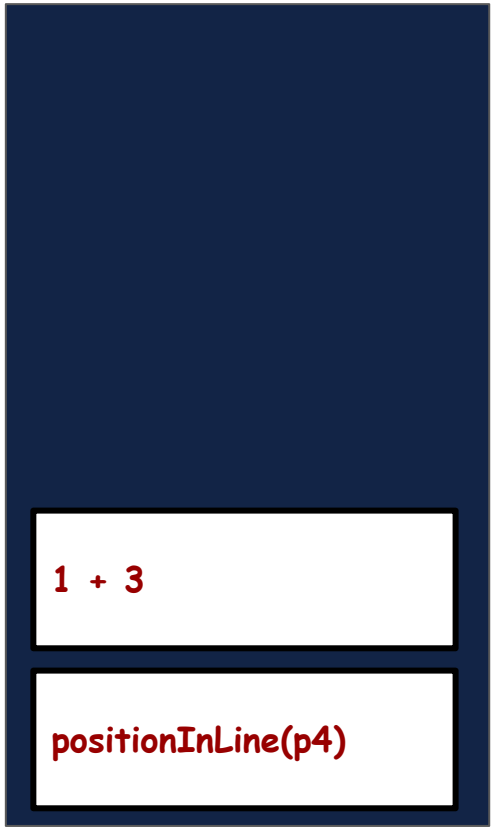
```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```



Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

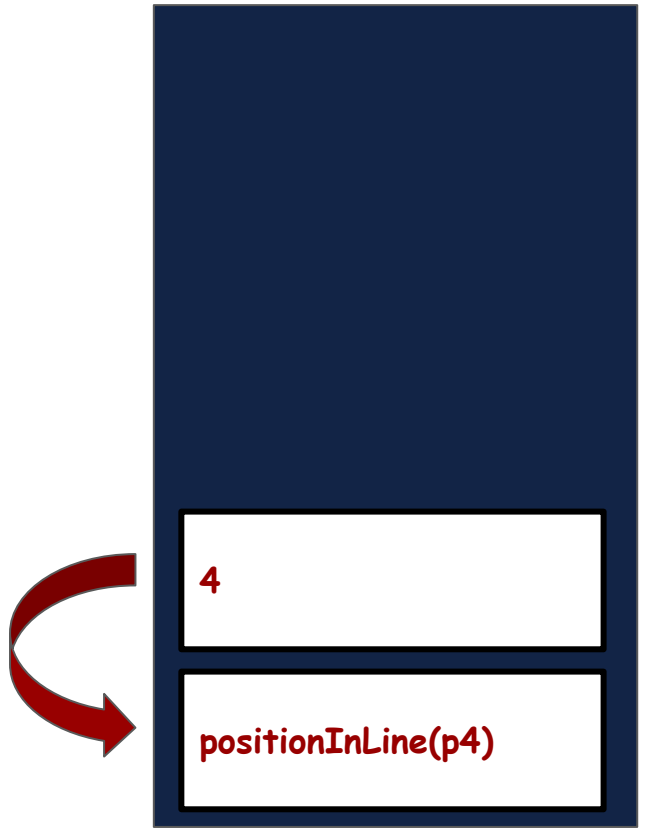
```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```



Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```



Recursive Function: Stack Calls

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```

I'm number 4

4

Recursive Function

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```

Terminating Condition

Recursive Function: Base Case

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```

Terminating Condition == Base Case

Recursive Function: Recursive Case

```
int positionInLine(Person *person)
{
    if(person->next == NULL)
    {
        return 1;
    }
    else
    {
        return 1 + positionInLine(person->next);
    }
}
```

```
int main()
{
    Person* you = generateData();
    cout << "I'm number " << positionInLine(you);
}
```

Recursive Condition

==

Recursive Case

Recursive Function

Recursive Function is just like you were doing some work.

Recursive Function

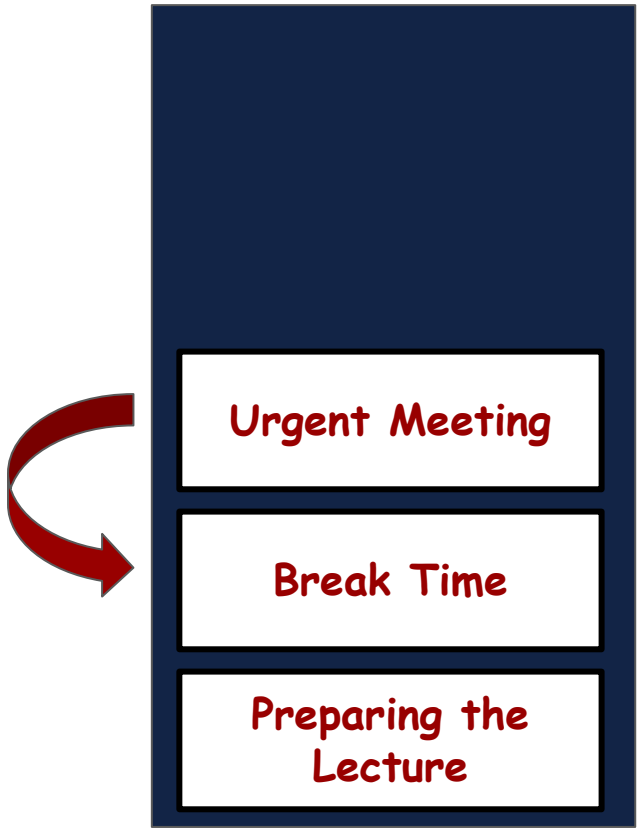
You got interrupted and went to have lunch.

Recursive Function

You got a Call from your Supervisor.

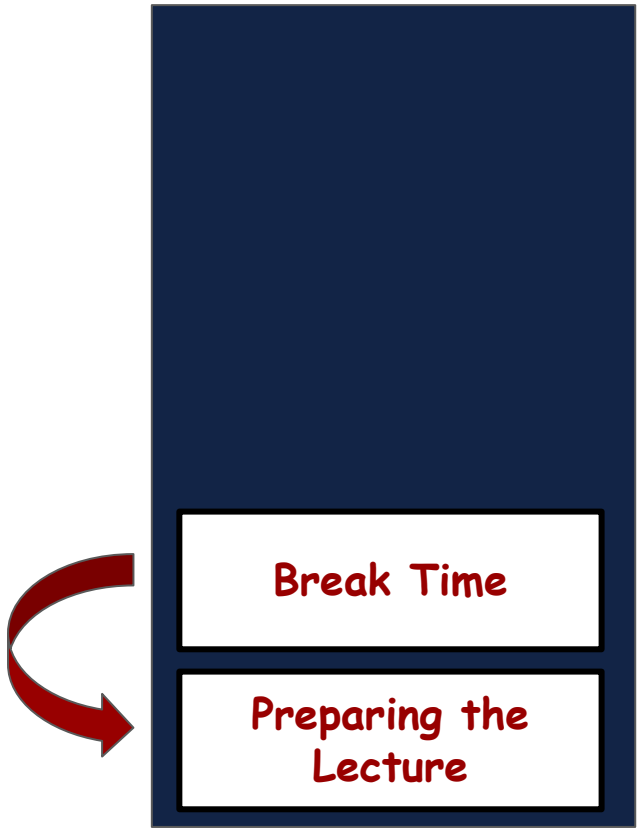
Recursive Function

Finished the meeting came back to finish the lunch.



Recursive Function

Finished the lunch and came back to prepare the lecture.

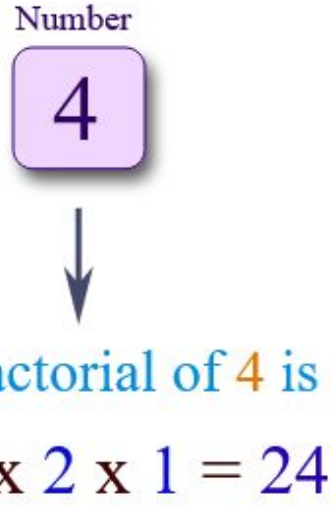


Recursive Function

Finished the lecture.

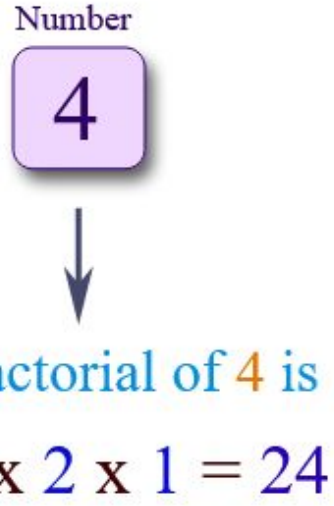
Working Example

Write a function to calculate the factorial of a number (a non-negative integer). The function accepts the number as the argument.



Working Example

Write a function to calculate the factorial of a number (a non-negative integer). The function accepts the number as the argument.



Factorial: Iterative Solution

```
int factIterative(int num)
{
     int fact = 1;
     for(int x = 2; x <= num; x++)
     {
          fact = fact * x;
     }
     return fact;
}
```

Factorial: Recursive Solution

```
int factRecursive(int num)
{
     if(num == 1)
     {
          return 1;
     }
     else
     {
          return num * factRecursive(num - 1);
     }
}
```

Recursion VS Iteration

```
int factRecursive(int num)
{
    if(num == 1)
    {
        return 1;
    }
    else
    {
        return num * factRecursive(num - 1);
    }
}
```

```
int factIterative(int num)
{
    int fact = 1;
    for(int x = 2; x <= num; x++)
    {
        fact = fact * x;
    }
    return fact;
}
```

Difficult to Debug	Easy to Debug

Recursion VS Iteration

```
int factRecursive(int num)
{
    if(num == 1)
    {
        return 1;
    }
    else
    {
        return num * factRecursive(num - 1);
    }
}
```

```
int factIterative(int num)
{
    int fact = 1;
    for(int x = 2; x <= num; x++)
    {
        fact = fact * x;
    }
    return fact;
}
```

Difficult to Debug	Easy to Debug
Needs extra memory for function calls	Doesn't need extra memory

Recursion VS Iteration

```
int factRecursive(int num)
{
    if(num == 1)
    {
        return 1;
    }
    else
    {
        return num * factRecursive(num - 1);
    }
}
```

```
int factIterative(int num)
{
    int fact = 1;
    for(int x = 2; x <= num; x++)
    {
        fact = fact * x;
    }
    return fact;
}
```

Difficult to Debug	Easy to Debug
Needs extra memory for function calls	Doesn't need extra memory
Execution is Slow	Execution is Fast

Recursion VS Iteration

```
int factRecursive(int num)
{
    if(num == 1)
    {
        return 1;
    }
    else
    {
        return num * factRecursive(num - 1);
    }
}
```

```
int factIterative(int num)
{
    int fact = 1;
    for(int x = 2; x <= num; x++)
    {
        fact = fact * x;
    }
    return fact;
}
```

Difficult to Debug	Easy to Debug
Needs extra memory for function calls	Doesn't need extra memory
Execution is Slow	Execution is Fast
Code is comparatively Small	Code is comparatively Large

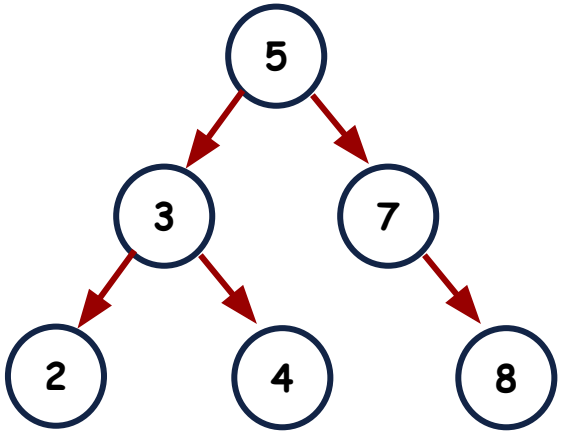
Why Bother?

Then, Why should we implement our solution with recursion when we can always solve the problems with iteration.

Recursion	Iteration
Difficult to Debug	Easy to Debug
Needs extra memory for function calls	Doesn't need extra memory
Execution is Slow	Execution is Fast
Code is comparatively Small	Code is comparatively Large
Difficult to think in terms of Recursion	Easier to think in terms of Iteration

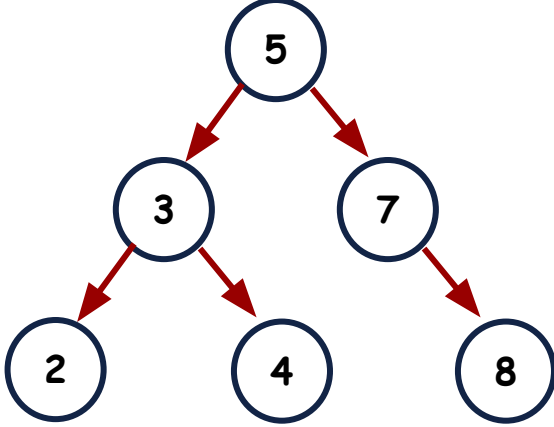
Recursion: better Solution for Trees

It's a lot easier to perform operations on trees using recursion.



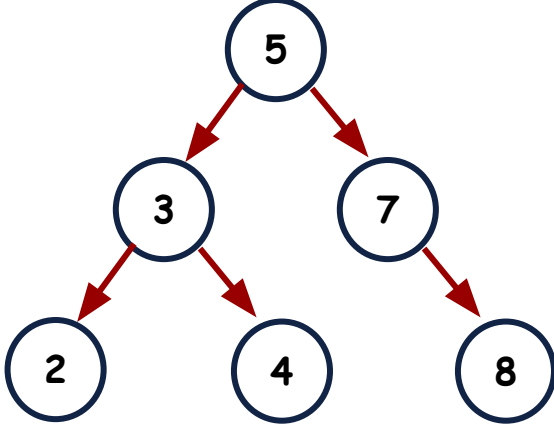
In-Order Traversal

```
void inOrderIterative()
{
    stack<TreeNode *> stack;
    TreeNode *curr = root;
    while (!stack.empty() || curr != NULL)
    {
        if (curr != NULL)
        {
            stack.push(curr);
            curr = curr->left;
        }
        else
        {
            curr = stack.top();
            stack.pop();
            cout << curr->val << " ";

            curr = curr->right;
        }
    }
}
```



In-Order Traversal

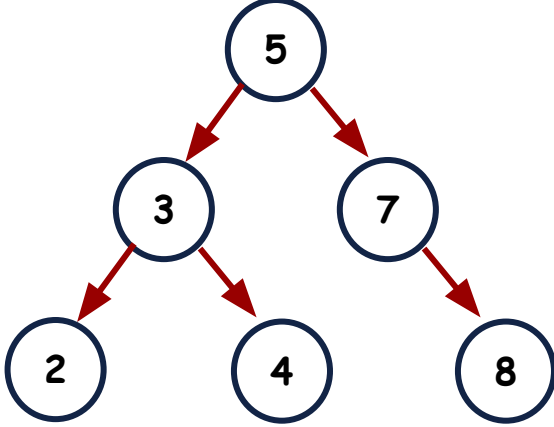
```
void inOrderIterative()
{
    stack<TreeNode *> stack;
    TreeNode *curr = root;
    while (!stack.empty() || curr != NULL)
    {
        if (curr != NULL)
        {
            stack.push(curr);
            curr = curr->left;
        }
        else
        {
            curr = stack.top();
            stack.pop();
            cout << curr->val << " ";

            curr = curr->right;
        }
    }
}
```




```
void inOrderRecursive(TreeNode *node)
    {
        if (node == NULL)
            return;

        inOrderRecursive(node->left);
        cout << node->val << " ";
        inOrderRecursive(node->right);
    }
```

Food For Thought

What will happen if you do not write base case in your recursive function?




```
void inOrderRecursive(TreeNode *node)
    {
        if (node == NULL)
            return;

        inOrderRecursive(node->left);
        cout << node->val << " ";
        inOrderRecursive(node->right);
    }
```

Learning Objective

Students should be able to write recursive **functions** and understand stack calls.

Self Assessment

What are the Types of Recursion?

Reading Activity:
https://www.javatpoint.com/types-of-recursion-in-c

Self Assessment

https://leetcode.com/problems/fibonacci-number/

https://leetcode.com/problems/search-in-a-binary-search-tree/description/

https://leetcode.com/problems/insert-into-a-binary-search-tree/description/

https://leetcode.com/problems/kth-smallest-element-in-a-bst/

https://leetcode.com/problems/delete-node-in-a-bst/