Recursion 1

INTRODUCTION

Since computer programming is a fundamental application of mathematics, so first, let's try to understand the mathematical logic behind recursion.

In general, we all are aware of the concept of functions. Briefly, functions are the mathematical equations that produce an output on providing input. Different types of inputs can have a single value of output, but different types of outputs can't have a single input. **For example:** Suppose F(x) is a function defined by:

$$\mathbf{F(x) = x^2 + 4}$$

where F(x) is the output received for a particular input x. We can observe that if we put x = 2, we receive output as 8 and if we put x = -2, then also we receive output as 8. Representing this in the form of a computer program:

```
int F(int x) {
        return (x * x + 4);
}
```

Now, we can pass different values of x to this function and receive our output accordingly.

This proves **one-to-one mapping** of programming functions with mathematical functions.

As we vaguely recall, there was another concept called "**Principle of Mathematical Induction**" in mathematics, let's try to figure out if we have something in programming for this concept.

Before we dig deeper into it, let's revise. Principle of Mathematical Induction (PMI) is a technique for proving a statement, a formula, or a theorem that is asserted about every natural number. It has following three steps:

1. Step of trivial case: In this step, we will prove the desired statement for n = 1.

2. Step of assumption: In this step, we will assume that the desired statement is valid for n = k.
3. To prove step: From the results of assumption step, we will prove that n = k + 1 also holds for the desired equation.

For Example: Let's prove that:

S(n): 1 + 2 + 3 + ... + n = (n * (n + 1))/2 (the sum of first n natural numbers)

Proof:

Step 1: For n = 1, S(1) = 1 is true.

Step 2: Assume, the given statement is true for n = k, i.e.,

1 + 2 + 3 + ... + k = (k * (k + 1))/2

Step 3: Let's prove the statement for n = k + 1 using step 2.

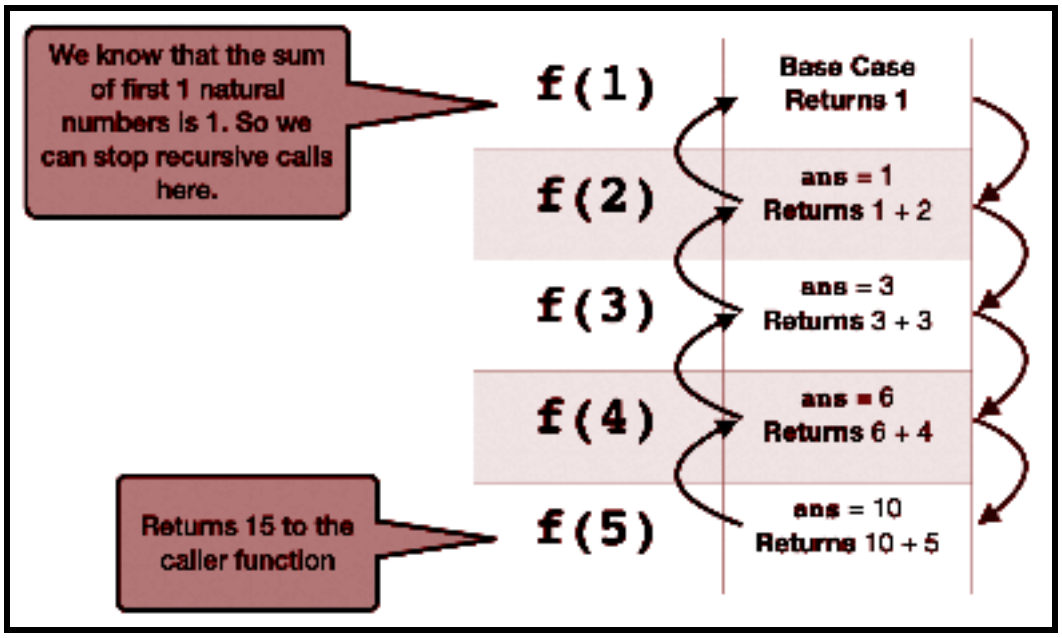
Adding (k+1) to both LHS and RHS in the result obtained on step 2: **1 + 2 + 3 + ... + (k+1) = (k*(k+1))/2 + (k+1)** *Now, taking (k+1) common from RHS side:* **1 + 2 + 3 + ... + (k+1) = (k+1)*((k + 2)/2)** *According the statement that we are trying to prove:* **1 + 2 + 3 + ... + (k+1) = ((k+1)*(k+2))/2** *Which is the same as we obtained above. Hence proved*

One can think, why are we discussing these over here. To answer this question, we need to know that these three steps of PMI are related to the three steps of recursion, which are as follows:

1. Induction Step (IS) and Induction Hypothesis (IH): Here, the Induction Step is the main problem which we are trying to solve using recursion, whereas Induction Hypothesis is the sub-problem, using which we'll solve the induction step. Let's define Induction Step and Induction Hypothesis for our running example:
   Induction Step: Sum of first n natural numbers - F(n)
   Induction Hypothesis: This gives us the sum of first n-1 natural numbers - F(n-1)
2. Express F(n) in terms of F(n-1) and write code: F(n) = F(n-1) + n.

```
int f(int n) {
    int ans = f(n-1);    // Induction Hypothesis step
    return ans + n;    // Solving problem from result in previous step
}
```

3. The code is still not complete. The missing part is the base case. Now we will dry run to find the case where the recursion needs to stop.



4. After the dry run, we can conclude that for n equals 1 answer is 1, which we already know. so we'll use this as a base case; hence the final code becomes:

```
int f(int n) {
   if(n == 1) {                    // Base case
      return 1;
   }
   int ans = f(n-1);
   return ans + n;
}
```

This is the main idea to solve recursive problems. To summarize, we will always focus on finding the solution to our starting problem and tell the function to compute rest for us using the particular hypothesis. This idea will be studied in detail in further sections with more examples.

Now, we'll learn more about recursion by solving problems which contain smaller subproblems of the same kind. Recursion in computer science is a method where the solution to the question depends on solutions to smaller instances of the same problem. By the exact nature, it means that the approach that we use to solve the original problem can be used to solve smaller problems as well. So, in other words, in recursion, a function calls itself to solve smaller problems. Recursion is a popular approach for solving problems because recursive solutions are generally easier to think than their iterative counterparts, and the code is also shorter and easier to understand.

Working of recursion

We can define the steps of the recursive approach by summarizing the above three steps:

- **Base case:** A recursive function must have a terminating condition at which the process will stop calling itself. Such a case is known as the base case. In the absence of a base case, it will keep calling itself and get stuck in an infinite loop. Soon, the recursion depth* will be exceeded and it will throw an error.
- **Recursive call:** The recursive function will recursively invoke itself on the smaller version of the problem. We need to be careful while writing this step as it is crucial to correctly figure out what your smaller problem is. On the solution of smaller problem, the original problem's solution depends.
- **Small calculation:** Generally, we perform a calculation step in each recursive call. We can achieve this calculation step before or after the recursive call depending upon the nature of the problem.

Note: Recursion uses an in-built stack which stores recursive calls. Hence, the number of recursive calls must be as small as possible to avoid memory-overflow.

Now, let us see how to solve a few common problems using Recursion.

Example 1: Let's look at the model for a better explanation of the concept...

Problem statement: We want to find out the factorial of a natural number.

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

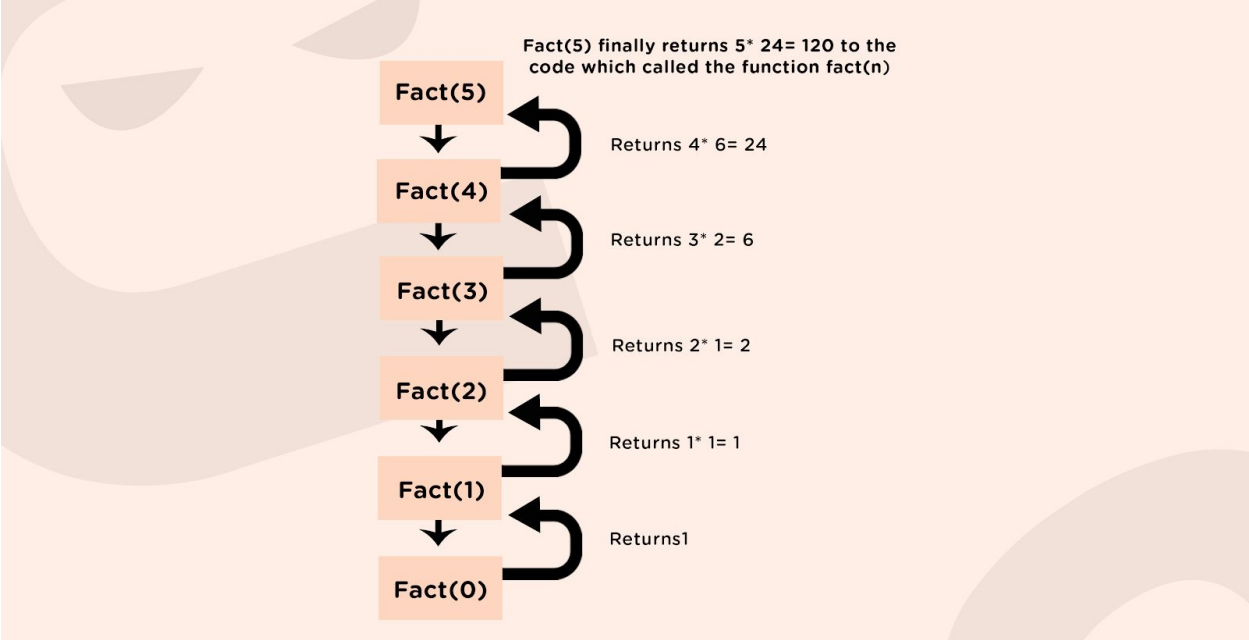
1. **Induction Step:** Calculating the factorial of a number n **- F(n)**

   Induction Hypothesis: We assume we have already obtained the factorial of n-1, through recursion **- F(n-1)**

2. Expressing F(n) in terms of F(n-1): **F(n)=n*F(n-1)**. Thus we get:

```
int f(int n) {
    int ans = f(n-1);                  //Assumption step
    return ans * n;                    //Solving problem from Assumption step
}
```

3. The code is still not complete. The missing part is the base case. Now we will dry run to find the case where the recursion needs to stop. Consider n = 5:



As we can see above, we already know the answer of n = 0, that is 1. So we will keep this as our base case. Hence, the code now becomes:

```
#include<iostream>
using namespace std;
int fact(int n) {
        if(n==0) {                     //Base Case
                return 1;
```

```
        }
        int ans = fact(n-1);            // Recursive call
        return n * ans;                 //Small calculation
}

int main() {
          int num;
          cin >> num;
          cout << fact(num);
          return 0;
}
```

Example 2: Let's look at another example of finding the Fibonacci numbers using recursion.

Function for Fibonacci series:

F(n) = F(n-1) + F(n-2),
with F(0) = 0 and F(1) = 1

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

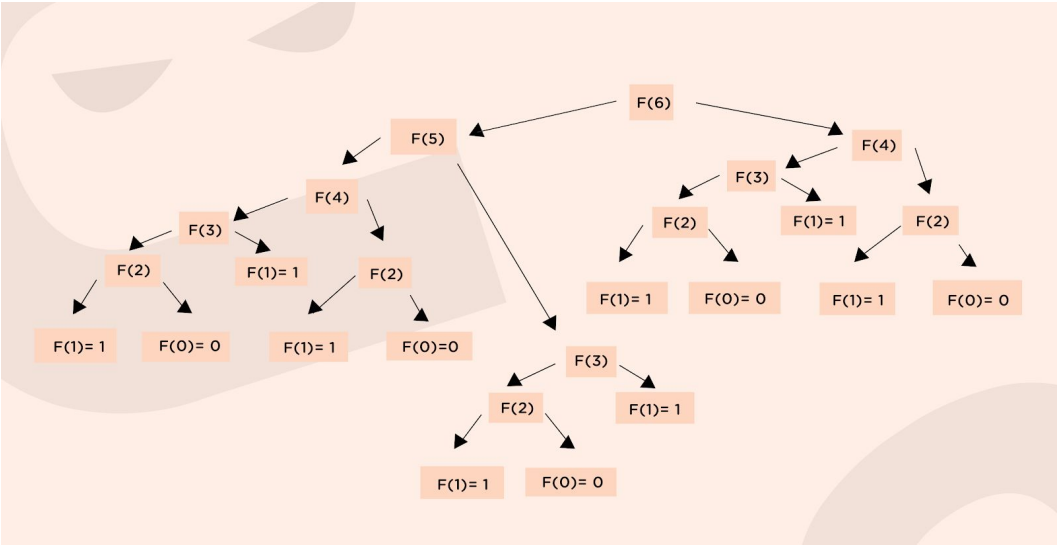
1. **Induction Step:** Calculating the nth Fibonacci number n.

   Induction Hypothesis: We have already obtained the (n-1)th and (n-2)th Fibonacci numbers.

2. Expressing F(n)in terms of F(n-1) and F(n-2): $\mathbf{F_n = F_{n-1} + F_{n-2}}$.

```
int f(int n){
    int ans = f(n-1) + f(n-2);          //Assumption step
    return ans;                         //Solving problem from assumption step
}
```

3. Let's dry run the code for achieving the base case: (Consider n= 6)



From here we can see, that every recursive call either ends at 0 or 1 for which we already know the answer: F(0) = 0 and F(1) = 1; hence using this as our base case in the code below:

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

int fib(int n) {
        if (n == 0) {                   // Base Case for n = 0
                return 0;
        }

        if (n == 1) {                   // Base case for n  = 1
                return 1;
        }

        int smallOutput1 = fib(n - 1);          // Recursive call 1
        int smallOutput2 = fib(n - 2);          // Recursive call 2
        return smallOutput1 + smallOutput2;     // Small calculation
}

int main() {
        cout << fib(6) << endl;                 // Will give output as 8
}
```

Recursion and array

Let us take an example to understand recursion on arrays.

Problem statement: We have to tell whether the given array is sorted or not using recursion.

For example: If the array is {2, 4, 8, 9, 9, 15}, then the output should be **YES.**

If the array is {5, 8, 2, 9, 3}, then the output should be **NO.**

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

1. **Problem:** Given an array, find whether it is sorted.

   Assumption step: We assume that we have already obtained the answer to the array starting from the index 1. In other words, we assume that we know whether the array (starting from the first index) is sorted.

2. Solving the problem from the results of the "Assumption step": Before going to the assume step, we must check the relation between the first two elements. Find if the first two elements are sorted or not. If the elements are not in sorted order, then we can directly return false. If the first two elements are in sorted order, then we will check for the remaining array through recursion.

```
bool is_sorted(int a[], int size) {
        if (a[0] > a[1]) {                              // Small Calculation
                return false;
        }
        bool isSmallerSorted = is_sorted(a + 1, size - 1);      //Assumption step
        return isSmallerSorted;
}
```

3. We can see that in the case when there is only a single element left or no element left in our array, at that time, the array is always sorted. Let's check the final code now...

```
#include <iostream>
```

```
using namespace std;

bool is_sorted(int a[], int size) {
        if (size == 0 || size ==1) {                    // Base case
                return true;
        }

        if (a[0] > a[1]) {                              // Small calculation
                return false;
        }
        bool isSmallerSorted = is_sorted(a + 1, size - 1);      // Recursive call
        return isSmallerSorted;
}

int main() {
        int arr[ ] = {2, 3, 6, 10, 11};
        if(is_sorted(arr, 5)){
                cout << "Yes" << endl;
        } else {
                Cout << "No" << endl;
        }
}
```

First Index

Let us solve another problem named the **First Index of an array**. The question states that you are given an array of length N and an integer x, you need to find and return the first index (0-based indexing) of integer x present in the array. Return -1, if it is not present in the array. The first index denotes the index of the first occurrence of x in the input array.

Case 1: Array = {1, 4, 5, 7, 2}, target value = 4

Output: 1 (as 4 is present at 1st index in the array).

Case 2: Array = {1, 3, 5, 7, 2}, target value = 4

Output: -1 (as 4 is not present in the array).

Case 3: Array = {1, 3, 4, 4, 4}, target value = 4

Output: 2 (as 4 is present at 3 positions in the array; i.e., [2, 3, 4], but as the question says, we have to find out the first occurrence of the target value, so the answer should be 2).

Approach:

Now, to solve the question, we have to figure out the following three elements of the solution:

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let the array be: [5, 5, 6, 2, 5] and x = 6. Now, if we want to find 6 in the array, then first we have to check with the first index. This is the **small calculation part**.

Code:

```
if(arr[0] == x)
        return 0;
```

Since, in the running example, the 0^{th} index element is not equal to 6, so we will have to make a recursive call for the remaining array: [5, 6, 2, 5] and x = 6. This is the **recursive call step**.

The recursive call will look like this:

```
rec_call(arr+1, size-1, x)
```

In the recursive call, we are incrementing the pointer and decrementing the size of the array. We have to assume that the answer will come for the recursive call. The answer will come in the form of an integer. If the answer is -1, this denotes that element is not present in the remaining array. If the answer is any other integer (other than -1), then this denotes that element is present in the remaining array. So, if the element is present at the ith index in the remaining array, then it will be present at i+1 in the array. For instance, in the running example, 6 is present at index 1 in the remaining array and at index 2 in the array.

However, the base case is still left to be added. The base case for this question can be identified by dry running the case: when you are trying to find an element that is not present in the array. For example: [5, 5, 6, 2, 5] and x = 10. On dry running, we can conclude that the base case will be the one when the size of the array becomes zero. When the size

of the array becomes zero, then we will return -1. This is because if the size of the array becomes zero, then this means that we have traversed the entire array and we were not able to find the target element. Therefore, this is the **base case step**.

Code:

```
if( size == 0 )
        return -1;
```

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Last index

In this problem, we are given an array of length N and an integer x. You need to find and return the last index of integer x present in the array. Return -1 if it is not present in the array. The last index means: if x is present multiple times in the array, return the index at which x comes last in the array.

Case 1: Array = {1, 4, 5, 7, 2}, target value = 4
Output: 1 (as 4 is present at 1st position in the array, which is the last and the only place where 4 is present in the given array).

Case 2: Array = {1, 3, 5, 7, 2}, target value = 4
Output: -1 (as 4 is not present in the array).

Case 3: Array = {1, 3, 4, 4, 4}, target value = 4
Output: 4 (as 4 is present at 3 positions in the array; i.e., [2, 3, 4], but as the question says, we have to find out the last occurrence of the target value, so the answer should be 4).

Approach:
Now, to solve the question, we have to figure out the following three elements of the solution.

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let the array be: [5, 5, 6, 2, 5] and x = 6. Now, if we want to find 6 in the array, then first we have to check with the first index. This is the **small calculation part**.

Code:

```
if(arr[0] == x)
        return 0;
```

Since, in the running example, the 0^{th} index element is not equal to 6, so we will have to make a recursive call for the remaining array: [5, 6, 2, 5] and x = 6. This is the **recursive call step**.

The recursive call will look like this:

```
rec_call(arr+1, size-1, x)
```

In the recursive call, we are incrementing the pointer and decrementing the size of the array. We have to assume that the answer will come for a recursive call. The answer will come in the form of an integer. If the answer is -1, this denotes that element is not present in the remaining array; otherwise, we need to add 1 to our answer as for recursion, it might be the 0^{th} index, but for the previous recursive call, it was the first position. For instance, in the running example, 6 is present at index 1 in the remaining array and at index 2 in the array.

However, the base case is still left to be added. The base case for this question can be identified by dry running the case when you are trying to find an element that is not present in the array. For example: [5, 5, 6, 2, 5] and x = 10. On dry running, we can conclude that the base case will be the one when the size of the array becomes zero. When the size of the array becomes zero, then we will return -1. This is because if the size of the array becomes zero, then this means that we have traversed the entire array and we were not able to find the target element. Therefore, this is the **base case step**.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

All indices of a number

Here, given an array of length N and an integer x, you need to find all the indexes where x is present in the input array. Save all the indexes in an array (in increasing order) and return the size of the array.

Case 1: Array = {1, 4, 5, 7, 2}, target value = 4
Output: [1] and the size of the array will be 1 (as 4 is present at 1st position in the array, which is the only position where 4 is present in the given array).

Case 2: Array = {1, 3, 5, 7, 2}, target value = 4
Output: [] and the size of the array will be 0 (as 4 is not present in the array).

Case 3: Array = {1, 3, 4, 4, 4}, target value = 4
Output: [2, 3, 4] and the size of the array will be 3 (as 4 is present at three positions in the array; i.e., [2, 3, 4]).

Now, let's think about solving this problem...

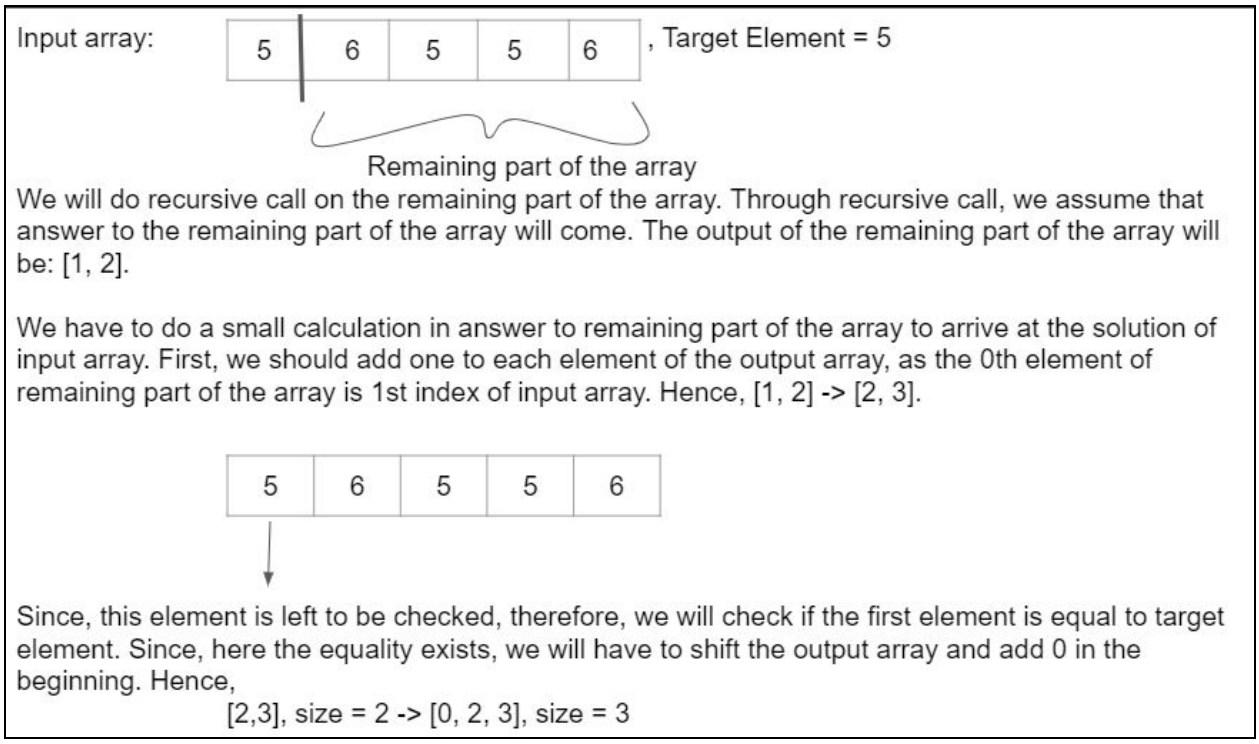
Approach:
Now, to solve the question, we have to figure out the following three elements of the solution:

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let us assume the given array is: [5, 6, 5, 5, 6] and the target element is 5, then the output array should be [0, 2, 3] and for the same array, let's suppose the target element is 6, then the output array should be [1, 4].

To solve this question, the base case should be the case when the size of the input array becomes zero. In this case, we should simply return 0, since there are no elements.

The next two components of the solution are Recursive call and Small calculation. Let us try to figure them out using following images:



So, following are the recursive call and small calculation components of the solution:

Recursive Call

Code:

```
int Size = fun(arr + 1, size - 1, x, output);
```

Small Calculation:

1. Update the elements of the output array by adding one to them.
2. If the equality exists, then shift the elements and add 0 at the first index of the output array. Moreover, increment the size, as we have added one element to the output array.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Using the same concept, other problems can be solved using recursion, just remember to apply PMI and three steps of recursion intelligently.