Recursion-1

Introduction

The process in which a function calls itself is called **recursion** and the corresponding function is called a **recursive function**.

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

In general, we all are aware of the concept of functions. In a nutshell, functions are mathematical equations that produce an output on providing input. **For example:** Suppose the function **F(x)** is a function defined by:

$$F(x) = x^2 + 4$$

We can write the **Java Code** for this function as:

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

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

Before moving onto the recursion, let's try to understand another mathematical concept known as the **Principle of Mathematical Induction (PMI)**.

Principle of Mathematical Induction (PMI) is a technique for proving a statement, a formula, or a theorem that is asserted about a set of natural numbers. It has the following three steps:

1. **Step of the trivial case:** In this step, we will prove the desired statement for a base case like **n = 0** or **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 the assumption step, we will prove that, **n = k + 1** is also true for the desired equation whenever **n = k** is true.

For Example: Let's prove using the **Principle of Mathematical Induction** 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.

To Prove: `1 + 2 + 3 + ... + (k+1) = ((k+1)*(k+2))/2`

Proof:

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
```

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:

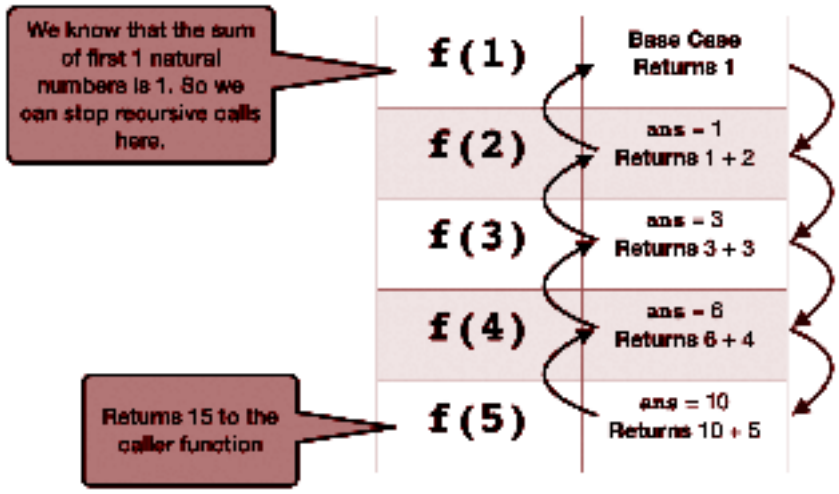
1. **Induction Step and Induction Hypothesis:** Here, the Induction Step is the main problem which we are trying to solve using recursion, whereas the Induction Hypothesis is the sub-problem, using which we'll solve the induction step. Let's define the Induction Step and Induction Hypothesis for our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

   Thus, we can write the Java code as:

```
public static 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, the answer is 1, which we already know. So we'll use this as our base case. Hence the final code becomes:

```
public static 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 the 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 invoke itself on a smaller version of the main problem. We need to be careful while writing this step as it is crucial to correctly figure out what your smaller problem is.
- **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. If the number of recursion calls exceeded the maximum permissible amount, the **recursion depth*** will be exceeded.
Now, let us see how to solve a few common problems using Recursion.

Problem Statement - Find Factorial of a Number

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.

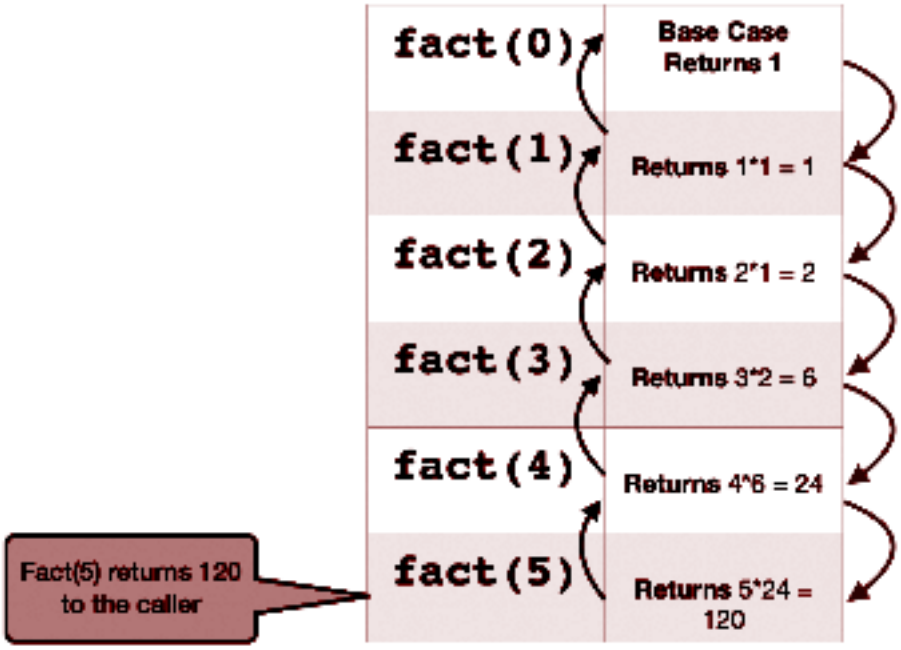
1. **Induction Step:** Calculating the factorial of a number n - **F(n)**

Induction Hypothesis: We have already obtained the factorial of n-1 - **F(n-1)**

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

```
public static int fact(int n){
     int ans = fact(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, which is 1. So we will keep this as our base case. Hence, the code now becomes:

```
public static int factorial(int n){
  if (n == 0) // base case
      return 1;
  else
      return n*factorial(n-1); // recursive case
}
```

Problem Statement - Fibonacci Number

Write a function *int fib(int n)* that returns nth fibonacci number. For example, if *n* = 0, then *fib(int n)* should return 0. If n = 1, then it should return 1. For n > 1, it should return F(n-1) + F(n-2), i.e., fibonacci of n-1 + fibonacci of n-2.

Function for Fibonacci series:

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

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

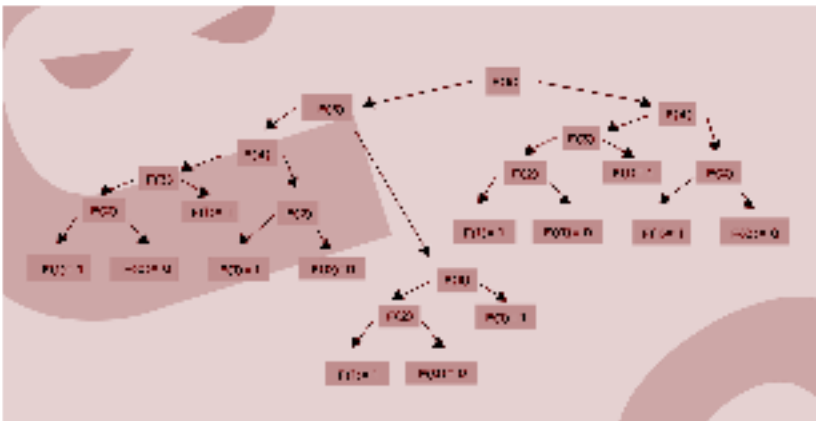
1. **Induction Step:** Calculating the n^{th} Fibonacci number n.

   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): $\mathbf{F_n = F_{n-1} + F_{n-2}}$.

```
public static 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:

```
public static int fib(int n){
  if (n <= 1)
      return n;
  else
      return (fib(n-1) + fib(n-2));
}
```

Recursion and array

Let us take an example to understand recursion on arrays.

Problem Statement - Check If Array Is Sorted.

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. **Induction hypothesis or Assumption step:** We assume that we have already obtained the answer to the array starting from 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 assumption 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.

```
public static int isSorted(int[][] a, int size){
    if (a[0] > a[1]) // Small Calculation
        return false
    int isSmallerSorted = isSorted(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, the array is always sorted. Let's check the final code now:

```
public static int isSorted(int[][] a, int size){
    if (size == 0 or size == 1) // Base case
        return true;

    if (a[0] > a[1]) // Small calculation
        return false;

    int isSmallerSorted = isSorted(a+1, size-1); //Recursive call
    return isSmallerSorted;
}

// driver code
public static void main(String[] args) {
    int arr[] = {2, 3, 6, 10, 11};
    if(isSorted(arr, 5))
        System.out.println("Yes");
    else
        System.out.println("No");
}
```

Problem Statement - First Index of Number

Given an array of length **N** and an integer **x**, you need to find and return the first index of integer **x** present in the array. Return **-1** if it is not present in the array. The first index means that if **x** is present multiple times in the given array, you have to return the index at which **x** comes first in the array.

To get a better understanding of the problem statement, consider the given cases:

Case 1: `Array = {1,4,5,7,2}, Integer = 4`

Output: 1

Explanation: 4 is present at 1st position in the array.

Case 2: `Array = {1,3,5,7,2}, Integer = 4`

Output: -1

Explanation: 4 is not present in the array

Case 3: `Array = {1,3,4,4,4}, Integer = 4`

Output: 2

Explanation: 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**

Small calculation part:

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 startIndex which we will increment in each recursive call.

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

Recursive Call step:

- Since, in the running example, the startIndex element is not equal to 6, so we will have to make a recursive call for the remaining array: `[5,6,2,5]`, `x=6`. Though we will pass the same array but startIndex will be incremented.
- The recursive call will look like this:

```
f(arr, sI+1, x);
```

- In the recursive call, we are incrementing the startIndex pointer.
- We have to assume that the answer will come from the recursive call. The answer will come in the form of an integer.
- If the answer is -1, this denotes that the element is not present in the remaining array.
- If the answer is any other integer (other than -1), then this denotes that the element is present in the remaining array.

Base case step:

- 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:** Consider the array [5, 5, 6, 2, 5] and x = 10. On dry running, we can conclude that the base case will be the one when the startIndex exceeds size of the array.
- Therefore , then we will return -1. This is because if the base case is reached, then this means that the element is not present in the entire array.
- We can write the base case as:

```
if(sI == arr.length) // Base Case
    return -1;
```

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

Problem Statement - Last Index of Number

Given an array of length **N** and an integer **x**, you need to find and return the first index of integer **x** present in the array. Return **-1** if it is not present in the array. The last index means that if **x** is present multiple times in the given array, you have to return the index at which **x** comes last in the array.

Case 1: `Array = {1,4,5,7,2}, Integer = 4`

Output: 1 (**Explanation:** 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}, Integer = 4`

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

Case 3: `Array = {1,3,4,4,4}, Integer = 4`

Output: 4 (**Explanation**: 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[sI] == x)
    return sI;
```

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**. We will start with startIndex from the last index of the array.

The recursive call will look like this:

```
f(arr, sI-1, x);
```

- In the recursive call, we are decrementing the startIndex pointer..
- We have to assume that the answer will come for a recursive call. The answer will come in the form of an integer.

Base Case:

- 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 startIndex passes beyond left of index 0 as we started from the rightmost index.

- When startIndex becomes -1, then we will return -1.
- This is because if the startIndex reaches -1, then this means that we have traversed the entire array and we were not able to find the target element.

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 a new array (in increasing order) and return that array.

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

Case 3: `Array = {1,3,4,4,4}, Integer = 4`
Output: [2, 3, 4], 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 startIndex reaches the size of the array. In this case, we should simply return an empty array, i.e., an array of size 0, since there are no elements left.

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

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

Recursive Call

```
int[][] output = fun(arr, startIndex+1, size, x);
```

Small Calculation:

1. If the element at startIndex of array is equal to the x, then create a new array of size of output+1. Now copy paste all the elements of the output array to the new array starting from 1st index and at the 0th index add the element of startIndex in original array. Finally return this new output.
2. Else is the case when element at startIndex do not match with x. So simply return the output.

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.