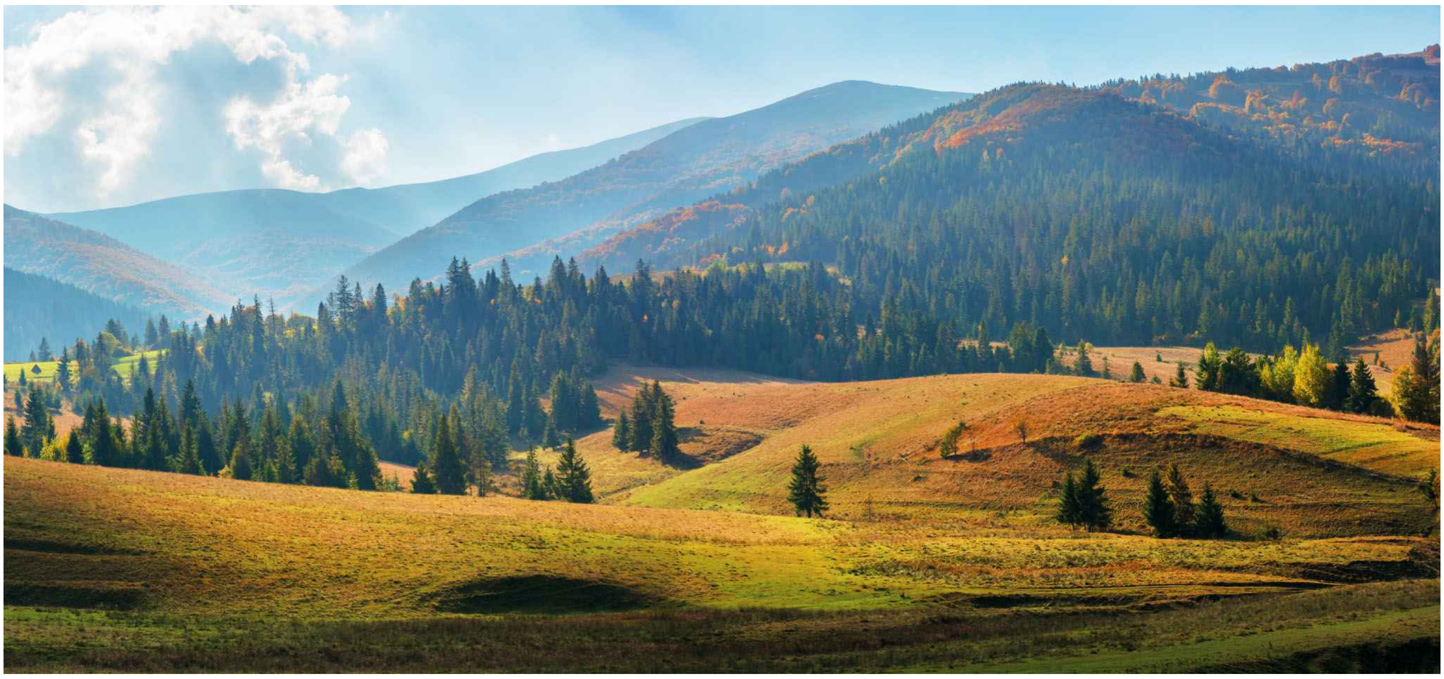


24-Recursion

Author: Vincent Lau

What is Recursion

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.

Base condition in recursion

- In the recursive program, the solution to the base case is provided and the solution to the bigger problem is expressed in terms of smaller problems.

How is a particular problem solved using recursion?

- The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

```
// Traditional way by for loop
int factorial(int n) {
  int i, fact = 1;
  // 1 * 2 * 3, where n = 3
  // 1 * 2 * 3 * 4 * 5, where n = 5
  for(i = 1; i <= n; i++){
      fact = fact * i;
  }
  System.out.println("Factorial of "+number+" is: "+fact);
}

// Recursive way
int factorial(int n) {
    if (n <= 1)  { // base case
        return 1;
    }
    return n * factorial(n - 1);
    // return 5 * (4 * (3 * ( 2 * 1)))
}
```

Why Stack Overflow error occurs in recursion?

- If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.
- If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

```
int fact(int n) {
    // wrong base case (it may cause stack overflow)
    if (n == 100) {
        return 1;
    }
    return n * fact(n - 1);
}
```

Direct & Indirect Recursion

- Most of the scenario can be solved by direct recursion. We have to understand the concept of both direct & indirect recursion.
- For this course, we just learn and focus on direct recursion.

```
// Direct recursion
```

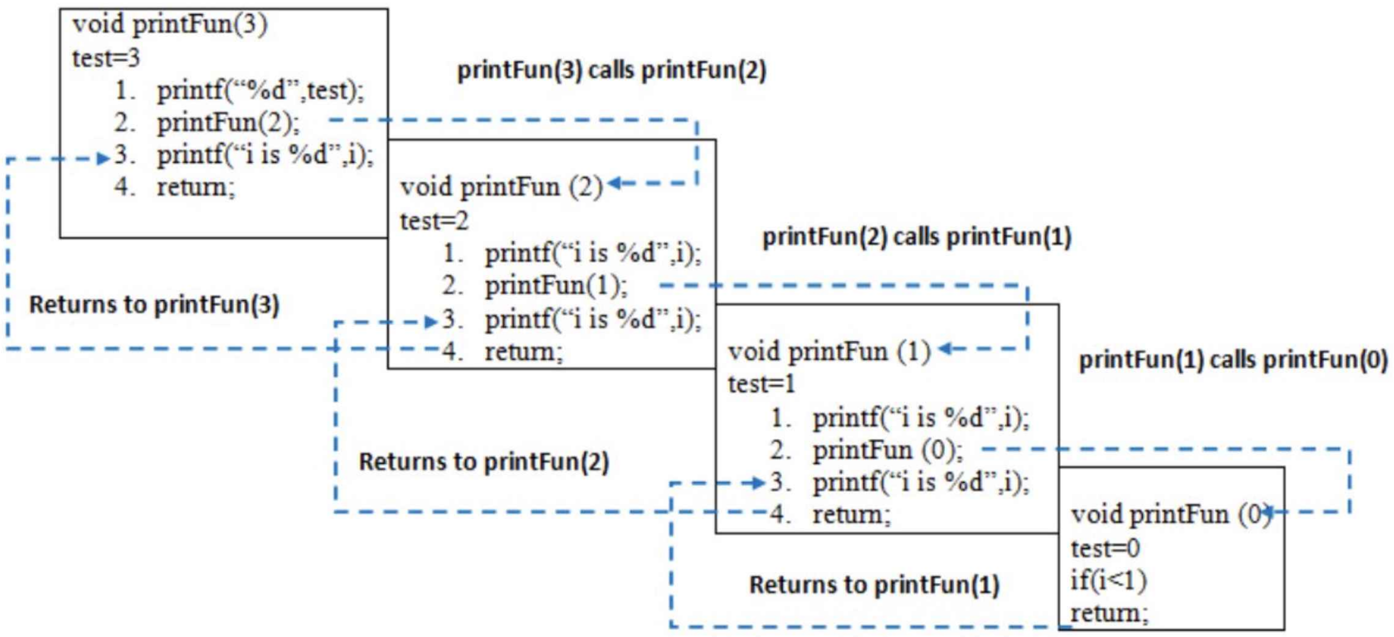
```
void directRecFun() {
    // Some code....
    directRecFun();
    // Some code...
}
// Indirect recursion
void indirectRecFun1() {
    // Some code...
    indirectRecFun2();
    // Some code...
}

void indirectRecFun2() {
    // Some code...
    indirectRecFun1();
    // Some code...
}
```

Tail & Non-tail Recursion

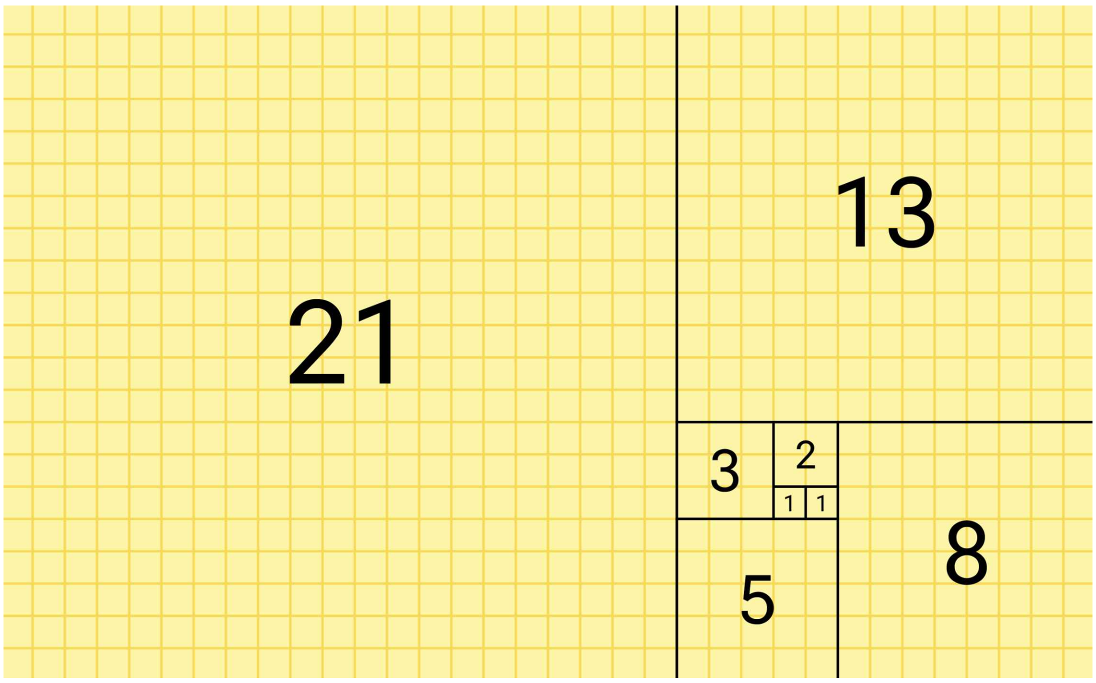
- A recursive function is tail recursive when recursive call is the last thing executed by the function, while non-tail recursive is ended with some other thing for the pattern.

```
// tail recursion
int factorial(int n) {
    if (n <= 1)  { // base case
        return 1;
    }
    return n * factorial(n - 1); // recursively call itself at tail
}

// non-tail recursion
class DemoFun {
  static void printFun(int test) {
    if (test < 1) {
      return;
    }
    System.out.printf("%d ", test);
    printFun(test - 1);
    System.out.printf("%d ", test); // there is still something to execute after recursive call
  }

  public static void main(String[] args) {
    printFun(3);
  }
} // print 3 2 1 1 2 3
```

- When **printFun(3)** is called from main(), memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints ‘3’. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statement 1 to 4 are pushed in the stack. Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun(0)** goes to if statement and it return to **printFun(1)**. Remaining statements of **printFun(1)** are executed and it returns to **printFun(2)** and so on. In the output, values from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in the diagram below.



Fibonacci (斐波那契数)



- In mathematics, the **Fibonacci numbers**, commonly denoted *F(n)*, form a sequence, the **Fibonacci sequence**, in which each number is the sum of the two preceding ones.
- The sequence commonly starts from 0 and 1, although some authors omit the initial terms and start the sequence from 1 and 1 or from 1 and 2. Starting from 0 and 1, the first few values in the sequence are:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 … …

By what we learned

- Complete it by FOR loop

```java
class DemoFibonacci {
     // position 1, 2, 3, 4, 5, 6, 7, 8, 9
    // fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...
    public static long fibonacci(long position) { // position >= 1
        if (position <= 2) {
            return position - 1;
        }
        // if position > 2
        long n1 = 0, n2 = 1, n3 = 1;
        for (int i = 2; i < position; i++) {
            n3 = n1 + n2;
            n1 = n2;
            n2 = n3;
        }
        return n3;
    }

    public static void main(String[] args) {
        System.out.println(DemoFibonacci.fibonacci(1)); // print 0
        System.out.println(DemoFibonacci.fibonacci(2)); // print 1
        System.out.println(DemoFibonacci.fibonacci(3)); // print 1
        System.out.println(DemoFibonacci.fibonacci(4)); // print 2
        System.out.println(DemoFibonacci.fibonacci(5)); // print 3
//          ...
    }
}
```

By recursion

- Write a fibonacci method to call itself recursively, which is to present the pattern of adding two preceding numbers.

```java
public class DemoFibonacci {
    // fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...
    public static long fibonacci(long number) {
        if (number <= 2) {
            return number - 1;
        }
        return fibonacci(number - 1) + fibonacci(number - 2);
        // Notes to calculate result:
        // 4 -> 3
            // f(3) -> 2
                // f(2) -> 2
                    // return 1;
                // f(1)
                    // return 1
            // f(2) -> 1
                // return 1
    }

    public static void main(String[] args) {
        System.out.println(DemoFibonacci.fibonacci(11)); // print 55
    }
}
```