Recursion

YAO ZHAO

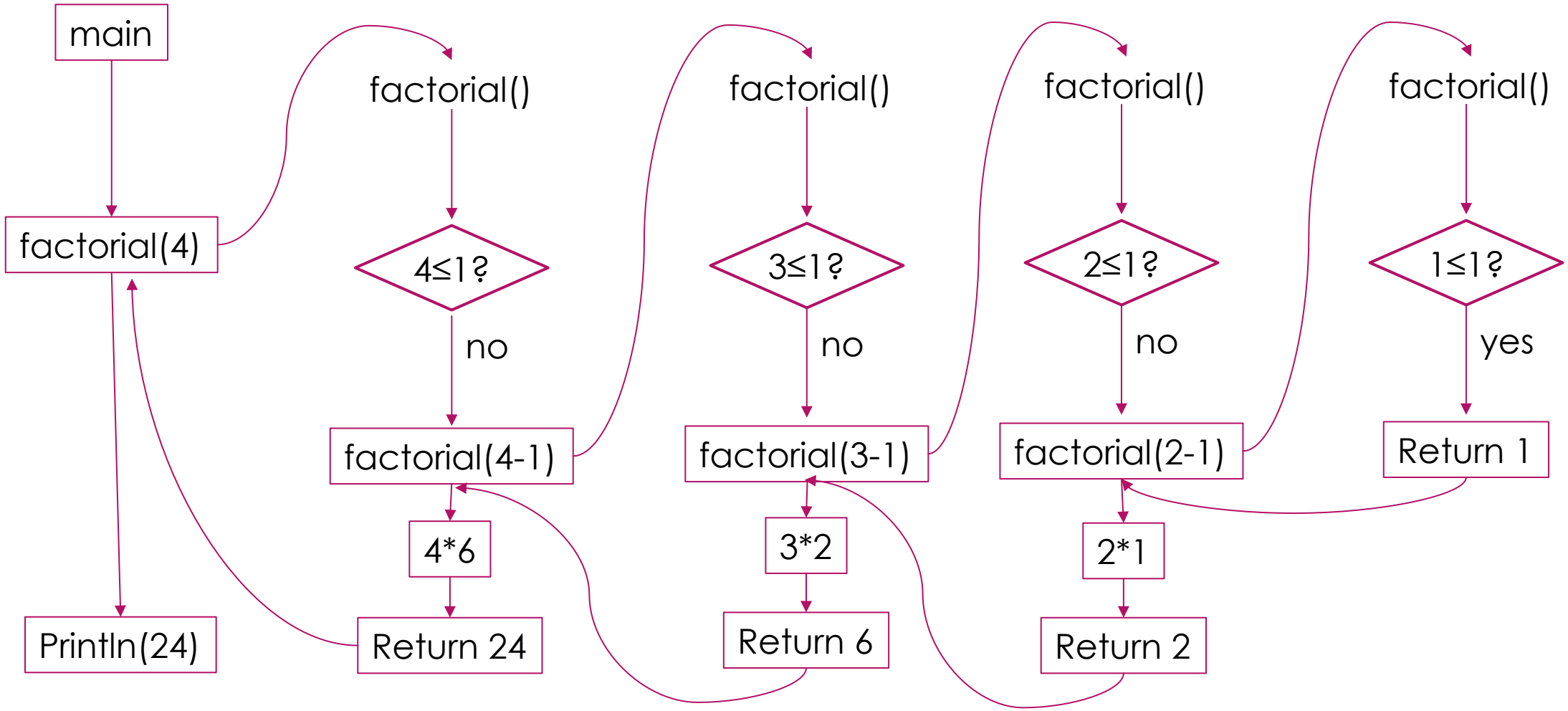
Factorial

The "Hello, World" for recursion is the *factorial* function, which is defined for positive integers *n* by the equation:
$n!=n\times(n-1)\times(n-2)\times\ldots\times2\times1$
The following relationship can be observed:
$n! = n \cdot (n-1)!$
We can implement the same problem by recursion.

```
public static long factorial(long n) {
    if (n <= 1) // test for base case
        return 1; // base cases: 0! = 1 and 1! = 1
    else // recursion step
        return n * factorial(n - 1);
}
```

Factorial: Trace this computation

```
public static void main(String[] args) {
    System.out.println(factorial(4));
}
```



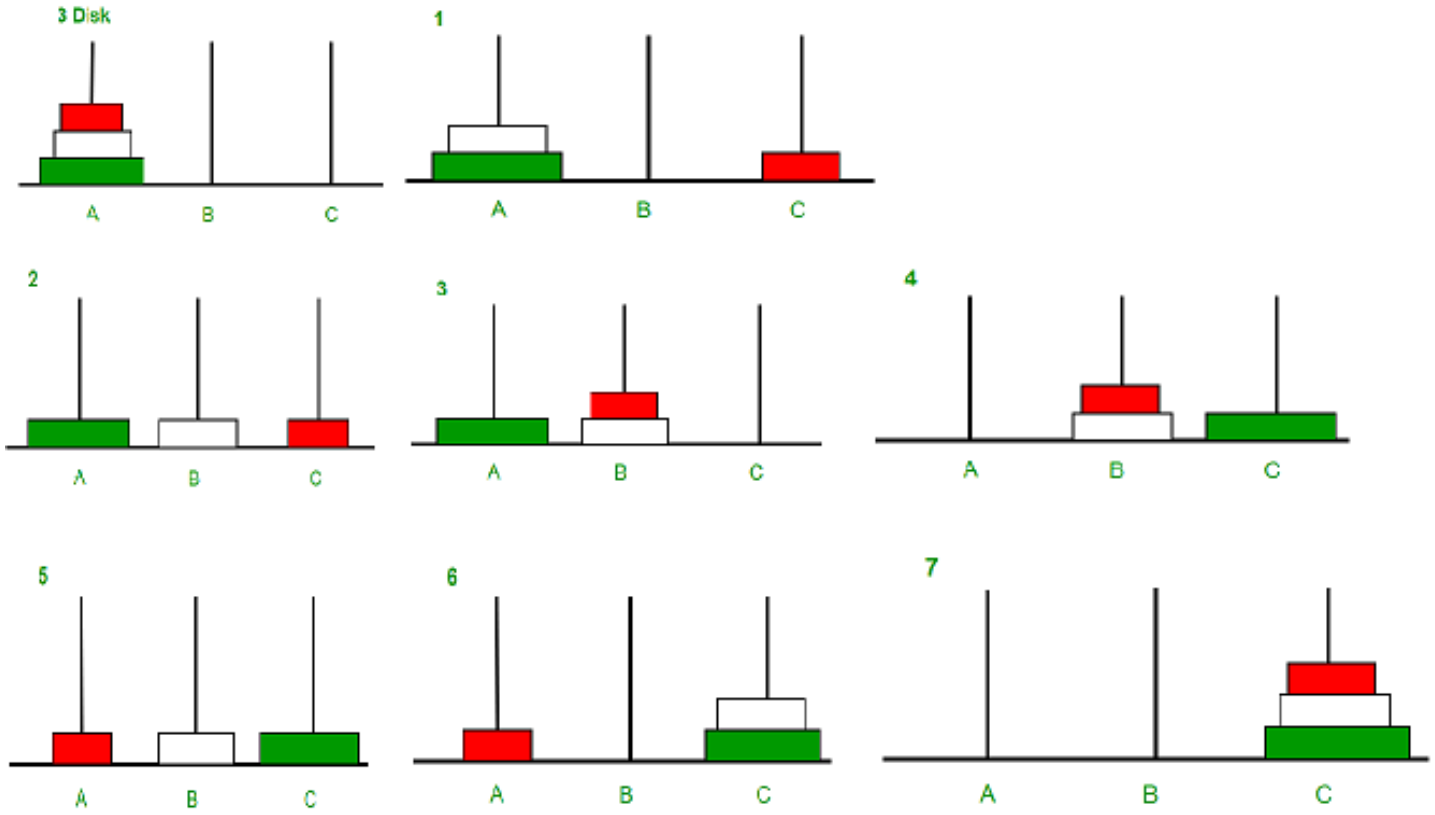
Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e., a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

Tower of Hanoi：Sample n = 3

1.Disk 1 moved from A to C
2.Disk 2 moved from A to B
3.Disk 1 moved from C to B
4.Disk 3 moved from A to C
5.Disk 1 moved from B to A
6.Disk 2 moved from B to C
7.Disk 1 moved from A to C

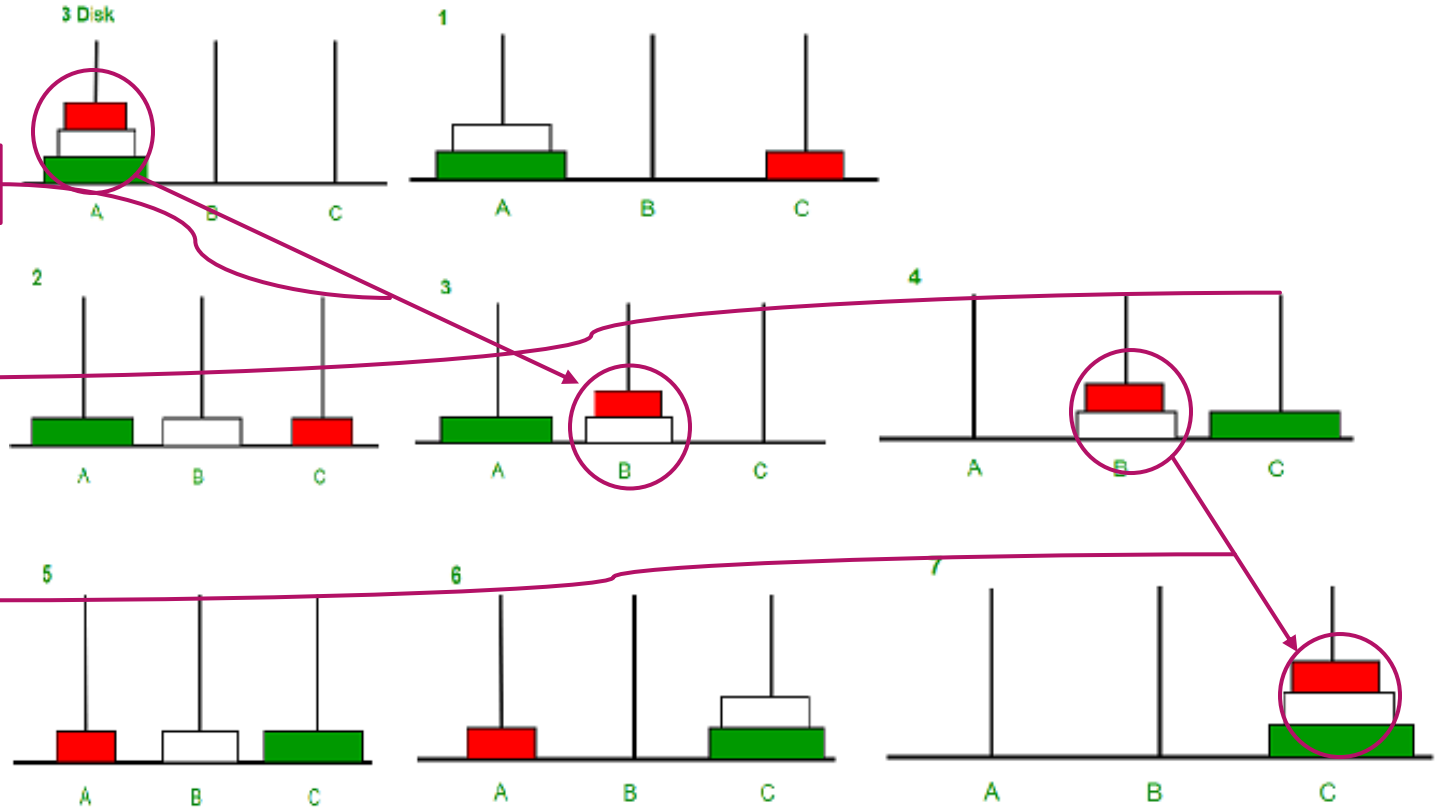


Tower of Hanoi：Sample n = 3

1~3. n-1 disks move from A to B

4. Disk n moved from A to C

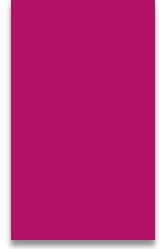
5~7. n-1 disks move from B to C



Tower of Hanoi：Code

```
static void towerOfHanoi(int n, char from_rod,
                         char to_rod, char aux_rod) {
    if (n == 1) {
        System.out.println("Move disk 1 from rod " + from_rod + " to rod " + to_rod);
        return;
    }
    towerOfHanoi( n: n - 1, from_rod, aux_rod, to_rod);
    System.out.println("Move disk " + n + " from rod " + from_rod + " to rod " + to_rod);
    towerOfHanoi( n: n - 1, aux_rod, to_rod, from_rod);
}
```

https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/
https://www.youtube.com/watch?v=YstLjLCGmgg



Practice: trace the computation for the tower of Hanoi like P.3

Euclid's algorithm

The greatest common divisor (gcd) of two positive integers is the largest integer that divides evenly into both of them. For example, the gcd(102, 68) = 34.
We can efficiently compute the gcd using the following property, which holds for positive integers p and q:
If p > q, the gcd of p and q is the same as the gcd of q and p % q.

```
public static int gcd(int p, int q) {
    if (q == 0) return p;
    else return gcd(q,  q: p % q);
}
```

Pitfalls of recursion

Code1:

```
public static double harmonic(int n) {
    return harmonic(n-1) + 1.0/n;
}
```

Pitfalls of recursion

Pitfall 1: *Missing base case*

The recursive function **harmonic** supposed to compute harmonic numbers, but is missing a base case.
If you call this function, it will repeatedly call itself and never return.

Pitfalls of recursion

```
public static double harmonic(int n) {
   if (n == 1) return 1.0;
   return harmonic(n) + 1.0/n;
}
```

Pitfalls of recursion

Pitfall 2: *No guarantee of convergence*

The recursive function **harmonic** goes into an infinite recursive loop for any value of its argument (except 1).
This is another common problem which is to include within a recursive function a recursive call to solve a subproblem that is not smaller than the original problem.

Pitfalls of recursion

```
public static double harmonic(int n) {
   if (n == 1) return 1.0;
   return harmonic(n-1) + 1.0/n;
}
```

Pitfalls of recursion

Pitfall 3:*Excessive memory requirements*

The recursive function **harmonic** correctly computes the nth harmonic number. However, calling it with a huge value of n (50000 for example)will lead to a **StackOverflowError**.
If a function calls itself recursively an excessive number of times before returning, the memory required by Java to keep track of the recursive calls may be prohibitive.

Pitfalls of recursion

```
public static long fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Pitfalls of recursion

Pitfall 4: *Excessive recomputation.*

This program is spectacularly inefficient. To see why it is futile to do so, consider what the function does to compute fibonacci(8) = 21. It first computes fibonacci(7) = 13 and fibonacci(6) = 8. To compute fibonacci(7), it recursively computes fibonacci(6) = 8 *again* and fibonacci(5) = 5. Things rapidly get worse. The number of times this program computes fibonacci(1) when computing fibonacci(n) is precisely F_n.

The temptation to write a simple recursive program to solve a problem must always be tempered by the understanding that a simple program might require exponential time (unnecessarily), due to excessive recomputation.

```
fibonacci(8)
   fibonacci(7)
      fibonacci(6)
         fibonacci(5)
            fibonacci(4)
               fibonacci(3)
                  fibonacci(2)
                     fibonacci(1)
                        return 1
                     fibonacci(0)
                        return 0
                     return 1
                  fibonacci(1)
                     return 1
                  return 2
               fibonacci(2)
                  fibonacci(1)
                     return 1
                  fibonacci(0)
                     return 0
                  return 1
               return 3
            fibonacci(3)
               fibonacci(2)
                  fibonacci(1)
                     return 1
                  fibonacci(0)
                     return 0
                  return 1
               fibonacci(1)
                  return 1
               return 2
            return 5
         fibonacci(4)
            fibonacci(3)
               fibonacci(2)
                .
                .
                .
```

Memoization(记忆化)

In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. Memoization has also been used in other contexts (and for purposes other than speed gains), such as in simple mutually recursive descent parsing. Although related to caching, memoization refers to a specific case of this optimization, distinguishing it from forms of caching such as buffering or page replacement. In the context of some logic programming languages, memoization is also known as tabling. (wikipedia)

How to use memoization to avoid ***Excessive recomputation?***

Problem: Climbing Stairs

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example 1:
Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps

Example 2:
Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

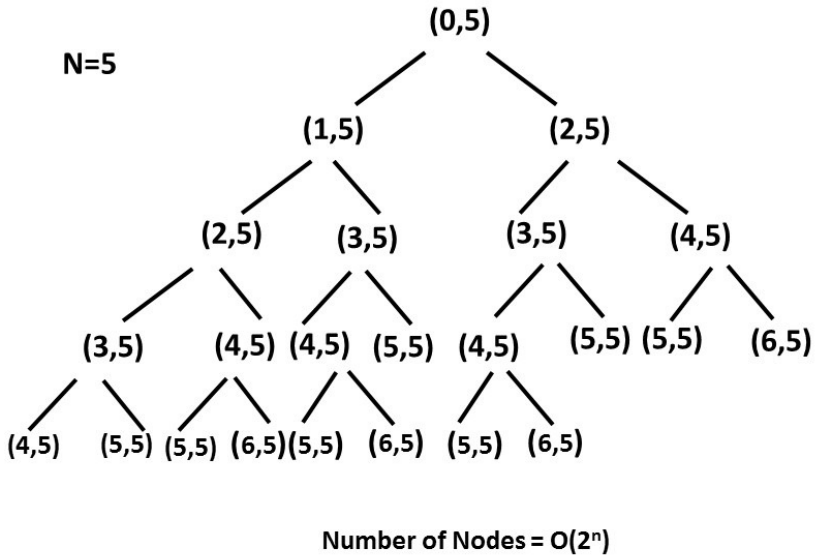
Source：LeetCode

Link：https://leetcode-cn.com/problems/climbing-stairs

Simple Recursion

climbStairs(*i*,*n*)=*climbStairs*(*i*+1,*n*)+*climbStairs*(*i*+2,*n*)

```
public class ClimbStairs {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        System.out.println(climbStairs(0,n));
    }
    public static int climbStairs(int i, int n) {
        if (i > n) {
            return 0;
        }
        if (i == n) {
            return 1;
        }
        return climbStairs(i + 1, n) + climbStairs(i + 2, n);
    }
}
```



Memoization Recursion

```
public class ClimbStairs {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        int memo[] = new int[n + 1];
        System.out.println(climbStairs(0, n, memo));
    }
    public static int climbStairs(int i, int n, int memo[]) {
        if (i > n) {
            return 0;
        }
        if (i == n) {
            return 1;
        }
        if (memo[i] > 0) {
            return memo[i];
        }
        memo[i] = climbStairs(i + 1, n, memo) + climbStairs(i + 2, n, memo);
        return memo[i];
    }
}
```

The result of each step is stored in the **memo** array, and each time the function is called again, we return the result directly from the **memo** array.

With the help of the **memo** array, we get a repaired recursion tree whose size is reduced to n.