斐波那契数列比较递归与循环

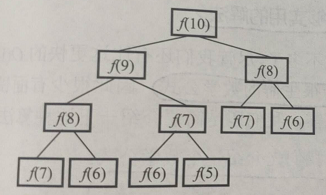
# 斐波那契数列Fibonacci

## 递归算法

\* 利用递归算法计算斐波那契序列

\* 递归算法存在重复计算：如**计算F(9)=F(8)+F(7)**; **F(8)=F(7)+F(6)**;

\* 计算F(9)和F(8) 内部都要计算F(7)，存在重复计算，利用树形图表示：



public int **Fibonacci**(int n){

if(n <= 0) return 0;

if(n == 1) return 1;

return **Fibonacci(n-1)+Fibonacci(n-2)**;

}

优点：代码简洁；

缺点：存在重复计算，效率较低。

## 循环算法（应用比较广泛）

\* 循环算法： 保存上次计算结果，不断更新前两个值，用于计算后面的值。

public int **Fibonacci**(int n){

if(n<=0) return 0;

if(n==1) return 1;

int first = 0,second = 1;

while(n-- >= 2){

**int third = first + second;**

first = second;

second = third;

}

return second;

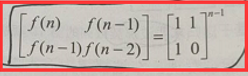
}

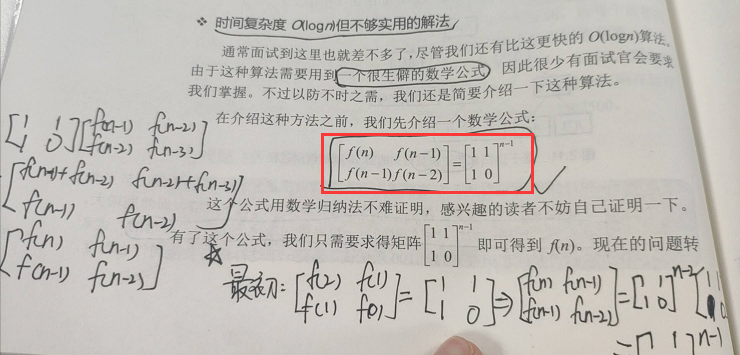
## 利用矩阵实现(应用比较少,编程比较难)

/\*\*

\* 算法3：利用矩阵实现

\*/





**f(n) f(n-1)**

**\***

**=**

f(n) f(n-1)

f(n-1) f(n-2)

1 1

1 0

f(n-1) f(n-2)

f(n-2) f(n-3)

**f(n-1) f(n-2)**

**当n=2时；**

1 1

1 0

f(2) f(1)

f(1) f(0)

**相等；只需要**

f(n) f(n-1)

f(n-1) f(n-2)

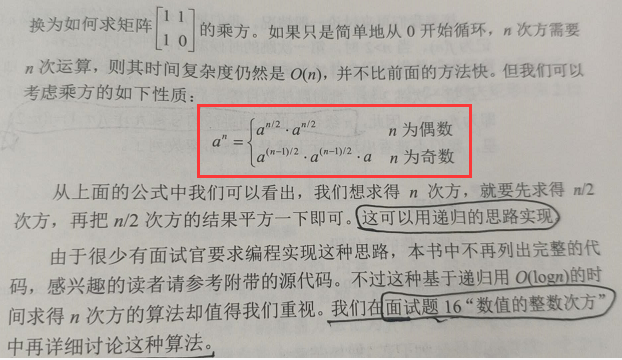
**n-1**

1 1

1 0

**=**

利用**幂的指数可加性**，如an=(an/2)2;最终的时间复杂度为O(log2n)



# 跳台阶问题

## 题目描述

一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法（先后次序不同算不同的结果）。

## 思路分析

\* 思路分析：跳n级台阶，分成两种情况，第一次跳1台阶后跳n-1台阶和第一次跳2台阶后跳n-2台阶，

\* **即正好满足斐波那契数列f(n)=f(n-1)+f(n-2)。**

## Java代码

/\*\*

\* 题目描述

\* 一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法（先后次序不同算不同的结果）。

\*/

/\*\*

\* 递归方法

\*/

public int JumpFloor2(int n) {

if(n<=0) return 0;

if(n==1||n==2) return n;

**return JumpFloor(n-1)+JumpFloor(n-2);**

}

/\*\*

\* 非递归方法

\*

\*/

public int JumpFloor(int n){

if(n<=0) return 0;

if(n == 1) return 1;

int first = 0,second = 1;

while(n-- > 0){

int third = first + second;

first = second;

second = third;

}

return second;

}

# 变态跳台阶问题

## 题目描述

一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

## 规律总结

\* 台阶树：n

\* 若n=1，则返回f(1)=1；{[1]}

\* 若n=2，则返回f(2)=f(1)+1=2；{[1,f(1)],[2]}

\* 若n=3，则返回f(3)=f(2)+f(1)+1=4；{[1,f(2)],[2,f(1)],[3]}

\* 若n，则返回2^(N-1);{[1,f(n-1)],[2,f(n-2)],...,[n-1,f(1)],[n]}

\* 即**f(n)=f(n-1)+f(n-2)+...+f(1)+1**

**f(n-1)= f(n-2)+...+f(1)+1**

**则f(n)=f(n-1)+f(n-1)=2\*f(n-1)**

\* 由于f(1)=2^(1-1);f(2)=2^(2-1);.....;**数学归纳法**，f(n) = 2^(n-1)。

## 实现base的n次方

思路分析：求解2的10次方，只需要求解2的5次方，然后再平方即可；

求解2的5次方只需要求解2的2次方，再平方，再乘以2即可；

求解2的2次方，只需要求解2的1次方，再平方即可。

最终再逐层返回，最终返回2的10次方。

//利用**递归**实现base的n次方

public int **power**(int base,int n){

if(n == 0) return 1;

if(n == 1) return base;

int result = power(base,n>>1);//核心

result \*= result;//关键

**if((n&1)==1) result \*= base;**

return result;

}

## Java代码

### 利用公式：f(n) = 2(n-1)

//方法1:直接利用公式

//利用数学归纳方法证明

public int **JumpFloorII**(int n) {

return **(int)Math.pow(2,n-1)**;

}

### 利用2倍关系

/\*\*

\* 递归方法：就是2倍的关系

\*/

public int JumpFloorIIBasedOnFibonacci2(int n) {

if(n <= 0) return 0;

if(n == 1) return 1;

**return 2\*JumpFloorIIBasedOnFibonacci(n-1);**

}

/\*\*

\* 非递归方法：就是2倍的关系

\*/

public int JumpFloorIIBasedOnFibonacci3(int n) {

if(n <= 0) return 0;

if(n == 1) return 1;

int result = 1;

// while(--n>0) result \*= 2;

// while(--n>0) result <<= 1;//利用左移替代乘2

**//先计算2^n，再乘以result**

**result \*= power(2,n-1);**

return result;

}

### 类比于Fibonacci序列：效率低下，重复计算严重

public int JumpFloorIIBasedOnFibonacci(int n) {

if(n <= 0) return 0;

if(n == 1) return 1;

if(n == 2) return 2;

int result = 0;

**while(--n>=0) result += JumpFloorIIBasedOnFibonacci(n);**

return result + 1;

}

# Leetcode\_70\_ClimbingStairs\_Easy

## 题目介绍

\* 难度：Easy

\* https://leetcode.com/problems/climbing-stairs/description/

\* 题目介绍：

\* You are climbing a stair case. It takes n steps to reach to the top.

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

\* Note: Given n will be a positive integer.

\* Example 1: Input: 2 Output: 2

\* Explanation: There are two ways to climb to the top.

\* 1. 1 step + 1 step

\* 2. 2 steps

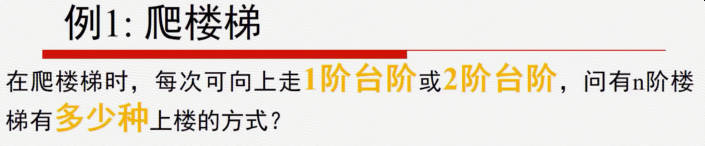
\* Example 2: Input: 3 Output: 3

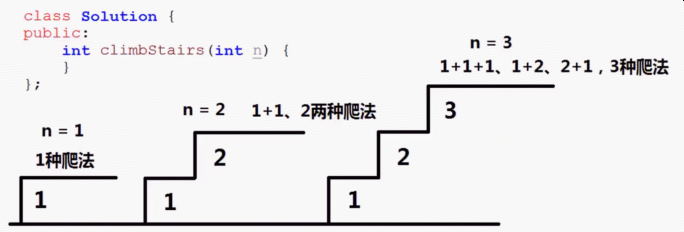
\* Explanation: There are three ways to climb to the top.

\* 1. 1 step + 1 step + 1 step

\* 2. 1 step + 2 steps

\* 3. 2 steps + 1 step





## 思路分析

\* 思路分析：动态规划思想。

\* f(n) = f(n-1) + f(n-2);

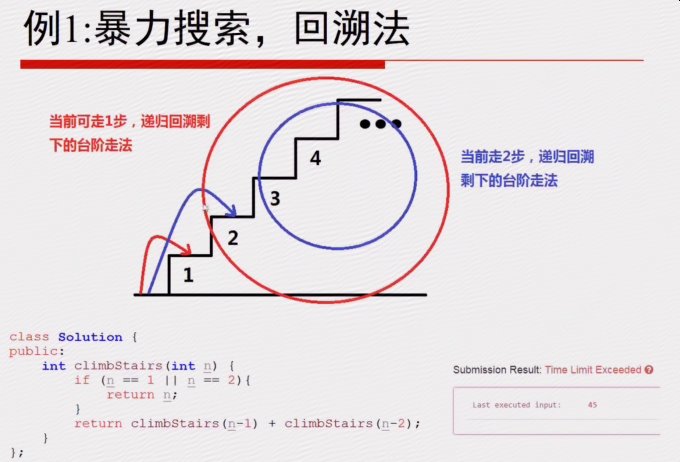
\* f(1) = 1;

\* f(2) = 2;

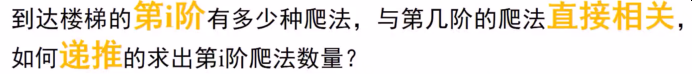
\* 凡是属于动态规划思路可以解决的问题，最好用dp表示出来，

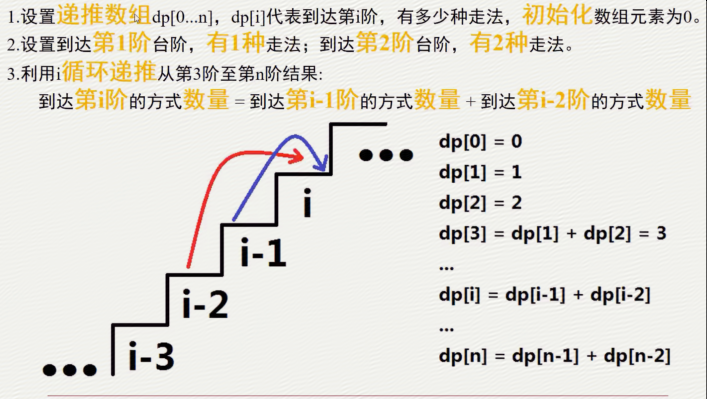
\* 面试官一看就知道采用的是DynamicProgramming思想。

### 思路：递归



### 思考





## Java代码

public int climbStairs(int n) {

if(n < 0) return 0;

if(n <= 2) return n;

int[] dp = new int[n+1];

for(int i = 0;i < 3;i++)//初始化,n必须大于2

dp[i] = i;

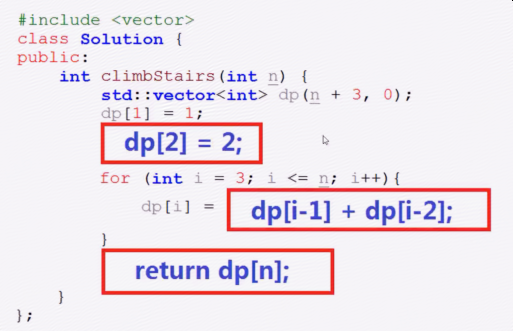
for(int i = 3;i <= n;i++){

dp[i] = dp[i-1] + dp[i-2];

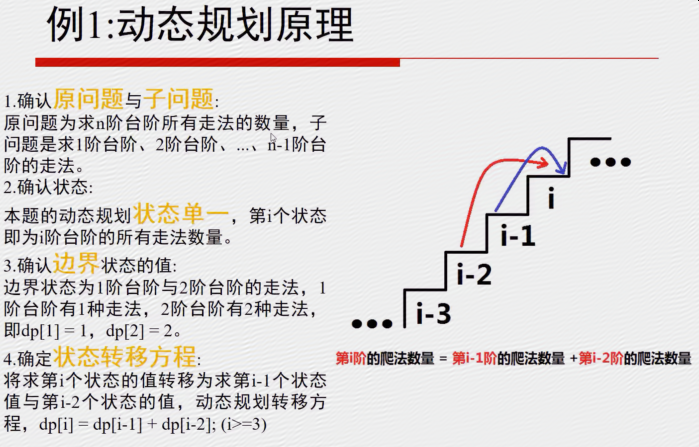
}

return dp[n];

}



## 动态规划原理



# Leetcode\_746\_MinCostClimbingStairs\_Easy

## 题目介绍

\* 难度：Easy

\* 类型分类：动态规划DynamicProgramming

\* <https://leetcode.com/problems/min-cost-climbing-stairs/description/>

\*

\* On a staircase, the i-th step has some non-negative cost cost[i] assigned (0 indexed).

\* Once you pay the cost, you can either climb one or two steps.

\* You need to find minimum cost to reach the top of the floor,

\* and you can either start from the step with index 0, or the step with index 1.

\*

\* Example 1:

\* Input: cost = [10, 15, 20] Output: 15

\* Explanation: Cheapest is start on cost[1], pay that cost and go to the top.

\* Example 2:

\* Input: cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1] Output: 6

\* Explanation: Cheapest is start on cost[0], and only step on 1s, skipping cost[3].

\* Note:

\* cost will have a length in the range [2, 1000].

\* Every cost[i] will be an integer in the range [0, 999].

## 思路分析

\* 思路分析：动态规划思路。

\* 题意理解：cost长度为len，则共len台阶，top相当于是len+1台阶。

\* 要想爬上top，需要计算之前的花费总和。

\* 编号从0开始，则top对应下标为len。

\* 爬上第n台阶的总花费，相当于爬上第n-1台阶的花费再加上n-1台阶本身的花费或者

\* 爬上第n-2台阶的花费再加上n-2台阶本身的花费；

\* 由于要获取最小的花费，取两种情况的最小值即可。

\* 那么爬上top的总花费就是爬上len-1台阶的花费再加上len-1台阶本身的花费与

\* 爬上len-2台阶的花费再加上len-2台阶本身的花费的较小者。

\* 定义：cost[n]表示第n台阶本身的花费；dp[n+1]表示爬上n+1花费的最小代价。

\* 动态规划状态转换为：

\* dp[n+1] = min{dp[n]+cost[n],dp[n-1]+cost[n-1]};n>=1

\* dp[0+1] = cost[0];

\* 为了节省内存空间，从低到高不断更新cost，即cost[n-1]=dp[n]；

\* 因此最后需要dp[n]，返回cost[n-1]和cost[n-2]的最小值即可。

## Java代码

public int **minCostClimbingStairs**(int[] cost) {

if(cost == null||cost.length == 2) return 0;

if(cost.length == 1) return cost[0];//边界情况

int len = cost.length;

for(int i = 2;i < len;i++)

**cost[i] += (cost[i-1]<cost[i-2]?cost[i-1]:cost[i-2]);**//或者使用Math.min()方法

return cost[len-1]<cost[len-2]?cost[len-1]:cost[len-2];

}