# 题目

斐波那契数，通常用 F(n) 表示，形成的序列称为斐波那契数列。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：

F(0) = 0,   F(1) = 1

F(N) = F(N - 1) + F(N - 2), 其中 N > 1.

给定 N，计算 F(N)。

**示例 1：**

输入：2

输出：1

解释：F(2) = F(1) + F(0) = 1 + 0 = 1.

**示例 2：**

输入：3

输出：2

解释：F(3) = F(2) + F(1) = 1 + 1 = 2.

**示例 3：**

输入：4

输出：3

解释：F(4) = F(3) + F(2) = 2 + 1 = 3.

**提示：**

0 ≤ N ≤ 30

类似题目：剑指offer 10-I

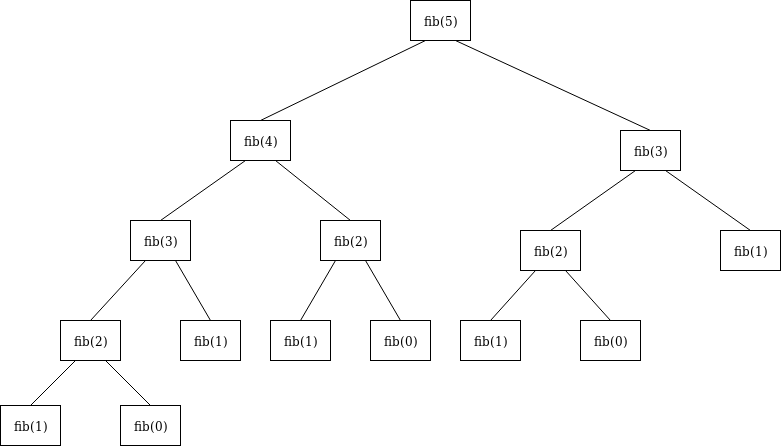
# 分析

## 方法一：递归

**思路：**

使用递归计算给定整数的斐波那契数。

注：该方法适用于N比较小的情况，如果N比较大会栈溢出，使用动态规划。



**代码：**

class Solution {

public:

int fib(int N) {

if (N <= 1) { // 递归的终止条件

return N;

}

return fib(N-1) + fib(N-2);

}

};

## 方法二：动态规划

**思路：**

斐波那契数列的定义是f(n + 1) = f(n) + f(n - 1)，生成第n项的做法有以下几种：

**递归法：**

原理：把f(n)问题的计算拆分成f(n−1)和f(n−2)两个子问题的计算，并递归，以f(0)和f(1)为终止条件。

缺点：大量重复的递归计算，例如f(n)和f(n−1)两者向下递归需要各自计算f(n - 2)的值。

**记忆化递归法：**

原理：在递归法的基础上，新建一个长度为n的数组，用于在递归时存储f(0)至f(n)的数字值，重复遇到某数字则直接从数组取用，避免了重复的递归计算。

缺点：记忆化存储需要使用O(N)的额外空间。

**动态规划：**

原理：以斐波那契数列性质f(n + 1) = f(n) + f(n - 1)为转移方程。

从计算效率、空间复杂度上看，动态规划是本题的最佳解法。

**动态规划解析：**

**状态定义：**设dp为一维数组，其中dp[i]的值代表斐波那契数列第i个数字。

**转移方程：**

dp[i+1]=dp[i]+dp[i-1]，即对应数列定义f(n+1) = f(n)+f(n-1)；

**初始状态：**

dp[0] = 0, dp[1] = 1，即初始化前两个数字；

**返回值：**

dp[n]，即斐波那契数列的第n个数字。

空间复杂度优化：

若新建长度为n的dp列表，则空间复杂度为O(N) 。

由于dp列表第i项只与第i−1和第i−2项有关，因此只需要初始化三个整形变量sum,a,b，利用辅助变量sum使a,b两数字交替前进即可（具体实现见代码）。

节省了dp列表空间，因此空间复杂度降至O(1) 。

**代码：**

class Solution {

public:

int fib(int n) {

if(0==n) return 0;

if(1==n) return 1;

vector<int> dp(n+1);

dp[0] = 0;

dp[1] = 1;

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

{

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

}

return dp[n];

}

};

**或：**

class Solution {

public:

int fib(int n) {

if(0==n) return 0;

vector<int> dp(n+1,0);

dp[0] = 0;

dp[1] = 1;

//该情况不能使用递归，栈溢出，使用动态规划

for(int i=2;i<dp.size();i++)

dp[i] = (dp[i-1] + dp[i-2]) % 1000000007;

return dp[n];

}

};