<https://leetcode.com/problems/fibonacci-number/description/>

The **Fibonacci numbers**, commonly denoted F(n) form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

F(0) = 0, F(1) = 1

F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate F(n).

**Example 1:**

**Input:** n = 2

**Output:** 1

**Explanation:** F(2) = F(1) + F(0) = 1 + 0 = 1.

**Example 2:**

**Input:** n = 3

**Output:** 2

**Explanation:** F(3) = F(2) + F(1) = 1 + 1 = 2.

**Example 3:**

**Input:** n = 4

**Output:** 3

**Explanation:** F(4) = F(3) + F(2) = 2 + 1 = 3.

**Constraints:**

0 <= n <= 30

**Attempt 1: 2025-05-27**

**Solution 1: Native DFS (10 min)**

class Solution {

    public int fib(int n) {

        if(n <= 1) {

            return n;

        }

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

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Solution 2: Memoization (10 min)**

class Solution {

    public int fib(int n) {

        Integer[] memo = new Integer[n + 1];

        return helper(n, memo);

    }

    private int helper(int n, Integer[] memo) {

        if(n <= 1) {

            return n;

        }

        if(memo[n] != null) {

            return memo[n];

        }

        return memo[n] = helper(n - 1, memo) + helper(n - 2, memo);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 3: DP (10 min)**

class Solution {

    public int fib(int n) {

        if(n <= 1) {

            return n;

        }

        int[] dp = new int[n + 1];

        dp[1] = 1;

        for(int i = 2; i <= n; i++) {

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

        }

        return dp[n];

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 4: Space Optimized DP (10 min)**

class Solution {

    public int fib(int n) {

        if(n <= 1) {

            return n;

        }

        int a = 0;

        int b = 1;

        int c = 0;

        for(int i = 2; i <= n; i++) {

            c = a + b;

            a = b;

            b = c;

        }

        return c;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to Deepseek**

**Problem Statement**

The Fibonacci numbers, commonly denoted F(n), form a sequence where each number is the sum of the two preceding ones, starting from 0 and 1. Given n, compute F(n).

**Approaches**

**1. Recursive (Naive) Approach**

Time Complexity: O(2^n) - Exponential due to repeated calculations

Space Complexity: O(n) - Recursion stack space

**2. Dynamic Programming (Memoization)**

Time Complexity: O(n) - Each subproblem solved once

Space Complexity: O(n) - For memoization array

**3. Iterative (Bottom-Up DP)**

Time Complexity: O(n)

Space Complexity: O(1) - Only stores last two values

**4. Matrix Exponentiation (Advanced)**

Time Complexity: O(log n)

Space Complexity: O(1)

**Solution Code**

**1. Recursive Solution**

class Solution {

public int fib(int n) {

if (n <= 1) return n;

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

}

}

**2. DP with Memoization**

class Solution {

public int fib(int n) {

if (n <= 1) return n;

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

memo[0] = 0;

memo[1] = 1;

return helper(n, memo);

}

private int helper(int n, int[] memo) {

if (memo[n] != 0 || n == 0) return memo[n];

memo[n] = helper(n - 1, memo) + helper(n - 2, memo);

return memo[n];

}

}

**3. Iterative DP (Optimal Space)**

class Solution {

public int fib(int n) {

if (n <= 1) return n;

int a = 0, b = 1;

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

int c = a + b;

a = b;

b = c;

}

return b;

}

}

**4. Matrix Exponentiation (Logarithmic Time)**

class Solution {

public int fib(int n) {

if (n <= 1) return n;

int[][] matrix = {{1, 1}, {1, 0}};

matrixPower(matrix, n - 1);

return matrix[0][0];

}

private void matrixPower(int[][] matrix, int power) {

if (power <= 1) return;

int[][] result = {{1, 0}, {0, 1}};

while (power > 0) {

if (power % 2 == 1) multiply(result, matrix);

multiply(matrix, matrix);

power /= 2;

}

System.arraycopy(result, 0, matrix, 0, result.length);

}

private void multiply(int[][] a, int[][] b) {

int x = a[0][0] \* b[0][0] + a[0][1] \* b[1][0];

int y = a[0][0] \* b[0][1] + a[0][1] \* b[1][1];

int z = a[1][0] \* b[0][0] + a[1][1] \* b[1][0];

int w = a[1][0] \* b[0][1] + a[1][1] \* b[1][1];

a[0][0] = x; a[0][1] = y;

a[1][0] = z; a[1][1] = w;

}

}

**Key Takeaways**

Recursive Solution is simple but inefficient for larger n due to exponential time complexity.

Memoization improves performance by storing computed results.

Iterative DP is the most space-efficient O(1) solution for this problem.

Matrix Exponentiation offers O(log n) time but is more complex to implement.

For most cases, the iterative DP solution (Solution 3) is recommended as it provides the best balance between simplicity and efficiency.

**Best Explain**

Leetcode

70 Climbing Stairs

> 类型：DP基础题

> Time Complexity O(N)

> Space Complexity O(1)

这道题作为DP的启蒙题(拥有非常明显的重叠子结构)，我在这详细的讲一讲LC大神们的答案是如何从一个毫无优化的做法，效率极低的递归解法，慢慢的进化到他们现在的答案，也给不会DP思路的同学补一补基础。

Top-Down

这道题自顶向下的思考：如果要爬到n台阶，有两种可能性:

在n-1的台阶处爬一层台阶

在n-2的台阶处爬两层台阶

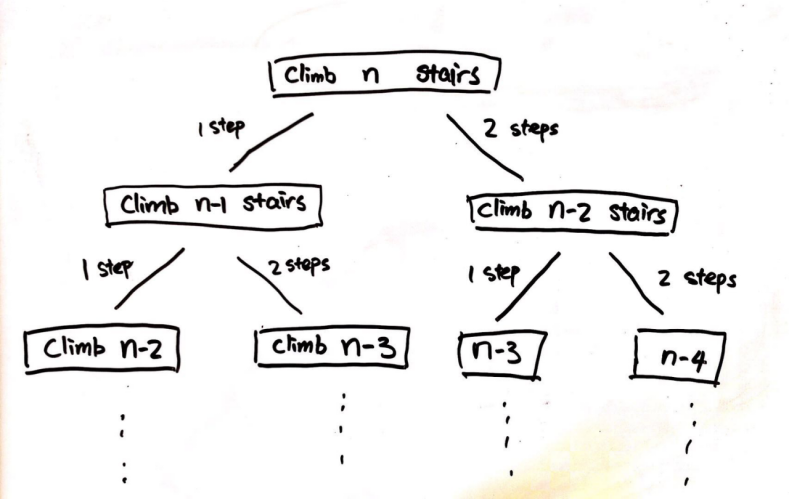
继续向下延伸思考，到达每一次层一共有几种方法这个问题就变成了2个子问题：

到达n-1层台阶有几种方法

到达n-2层台阶有几种方法

之后对返回子问题之和即可。

具体可以看下图。



根据以上的思路得到下面的代码

Solution 1: Recursion (TLE)

class Solution(object):

    def climbStairs(self, n):

        if n == 1: return 1

        if n == 2: return 2

        return self.climbStairs(n - 1) + self.climbStairs(n - 2)

TLE原因：

以上代码实现之所以会TLE，是因为递归的时候出现了很多次重复的运算。就如上图显示的爬n-2层的计算出现了2次，这种重复计算随着input的增大，会出现的越来越多，时间复杂度也会将以指数的级别上升。

优化思路：

这时候的思路为：如果能够将之前的计算好了的结果存起来，之后如果遇到重复计算直接调用结果，效率将会从之前的指数时间复杂度，变成O(N)的时间复杂度。

实现方法：

有了思路，实现方面则开辟一个长度为N的数组，将其中的值全部赋值成-1，具体为什么要用-1，是因为这一类问题一般都会要你求多少种可能，在现实生活中，基本不会要你去求负数种可能，所以这里-1可以成为一个很好的递归条件/出口。然后只要我们的数组任然满足arr[n] == -1，说明我们在n层的数没有被更新，换句话说就是我们还在向下递归或者等待返回值的过程中，所以我们继续递归直到n-1和n-2的值返回上来。这里注意我们递归的底层，也就是arr[0]和arr[1]，我们要对起始条件进行初始化：arr[0], arr[1] = 1, 2

Solution 2: Top-Down (using array)

class Solution(object):

    def climbStairs(self, n):

        if n == 1: return 1

        res = [-1 for i in range(n)]

        res[0], res[1] = 1, 2

        return self.dp(n-1, res)

    def dp(self, n, res):

        if res[n] == -1:

            res[n] = self.dp(n - 1, res) + self.dp(n - 2, res)

        return res[n]

这样是不是还是很抽象？我们可以举个例子，当n = 4的时候，我们在每一层返回之前打印一下当前的数组的值：

# print(n+1, res)

(2, [1, 2, -1, -1])

(1, [1, 2, -1, -1])

(3, [1, 2, 3, -1])

(2, [1, 2, 3, -1])

(4, [1, 2, 3, 5])

大家看到了，我们先从第4层开始递归，当递归到了1和2层的base case的时候，开始进行返回的计算，当到了第3层，将1和2层加起来，得到3，继续返回当到了第4层，将2和3层加起来，得到了5，这时候res[n] = 5，则满足出口条件，进行返回。

这就是Top-Down的思路，从大化小，思路就和DFS Tree中的分制一样。

Solution 2.5: Top-Down (using hash table)

这种的话就没有太多的篇幅，无脑塞哈希表就行。

class Solution(object):

    def \_\_init\_\_(self):

        self.memo = {0 : 0, 1 : 1, 2 : 2}

    def climbStairs(self, n):

        if n in self.memo:

            return self.memo[n]

        left = self.climbStairs(n - 1)

        self.memo[n - 1] = left

        right = self.climbStairs(n - 2)

        self.memo[n - 2] = right

        return left + right

or

class Solution(object):

    def \_\_init\_\_(self):

        self.memo = {0 : 0, 1 : 1, 2 : 2}

    def climbStairs(self, n):

        if n in self.memo:

            return self.memo[n]

        res = self.climbStairs(n - 1) + self.climbStairs(n - 2)

        self.memo[n] = res

        return res

Bottom-Up

Bottom-Up的思路则相反。我们不从n层向下考虑，而是解决我们每一层向上的子问题。在每一层，我们其实只需要知道在当前节点，我们的n-1和n-2的值是多少即可。

当我们有了第1层和第2层的base case，我们则可以直接从base case向上推得第3层，第4层以及第n层的答案，具体实现如下：

Solution 3: Bottom-Up (using array)

class Solution(object):

    def climbStairs(self, n):

        if n == 1: return 1

        res = [0 for i in range(n)]

        res[0], res[1] = 1, 2

        for i in range(2, n):

            res[i] = res[i-1] + res[i-2]

        return res[-1]

这种方法的使用的时候，我们发现其实在每一次更新当前的数的时候，我们最终返回的是最后一次更新的数，而我们的dependency是n-1 和n-2中的值，我们根本不需要一个数组去储存，直接用两个函数即可。所以底下为Bottom-Up的优化：

Solution 3: Bottom-Up (Constant Space)

class Solution(object):

    def climbStairs(self, n):

        if n == 1: return 1

        a, b = 1, 2

        for \_ in range(2, n):

            a, b = b, a + b

        return b

公瑾在刷题，加入我? : <https://github.com/yuzhoujr/leetcode/projects/1>

**Refer to**

[L70.Climbing Stairs](note://WEBce2fed61238e31617ae6d19bbb7f7a01)

[L746.Min Cost Climbing Stairs (Ref.L70)](note://WEB8aa33f758e5532cf447424bcc598b222)

[L842.Split Array into Fibonacci Sequence (Ref.L509)](note://WEBc3678ba168b711c5cb0a2b6c6755aef5)

[L873.Length of Longest Fibonacci Subsequence (Ref.L509)](note://WEB8078f50704f4e34bc32f4f9feccc01b9)