动态规划

最长斐波那契子序列的长度

中等/动态规划

学习目标

拉勾教育
—互联网人实战大学—

- 掌握动态规划的概念
- 掌握动态规划的基本解题步骤
- 掌握动态规划思想在实际问题的运用

题目描述

如果序列 X_1, X_2, ..., X_n 满足下列条件，就说它是斐波那契式 的：

- n >= 3
- 对于所有 i + 2 <= n，都有 X_i + X_{i+1} = X_{i+2}

给定一个严格递增的正整数数组形成序列，找到 A 中最长的斐波那契式的子序列的长度。如果一个都不存在，返回 0 。

注：子序列指从原序列 A 中删掉任意数量的元素（也可以不删），而不改变其余元素的顺序。

例如， [3, 5, 8] 是 [3, 4, 5, 6, 7, 8] 的一个子序列）

输入： [1,3,7,11,12,14,18]
输出： 3
解释：
最长的斐波那契式子序列有：
[1,11,12]，[3,11,14] 以及 [7,11,18] 。

一. Comprehend 理解题意

找到数组中的斐波那契子序列

1. 确定斐波那契子序列的两个起始元素(起始元素确定后，整个数列的值就确定了)
2. 根据两个起始元素依次向后寻找

- 在[1,3,7,11,12,14,18]中，斐波那契子序列可能的起始元素有[1,3],[1,7],...,[3,7],....，一共21种
- 如果起始元素是[1,3]，那么这个斐波那契序列一定是[1,3,4,7,11,18,29,...]
- 问题转化为判断[1,3,4,7,11,18,29,...] 有多少元素在数组中依次出现

二. Choose 数据结构及算法思维选择

基础解法：暴力搜索（暴力解法）

- 遍历所有起始元素组合，搜索可能的斐波那契子序列的长度
- 数据结构：数组
- 算法思维：暴力搜索

三. Code 基本解法及编码实现

基础解法： 暴力搜索

输入数组

子序列起始元素组合

每一个起始元素组合确定的序列

在原数组中查找序列元素

三. Code 基本解法及编码实现

基础解法： 暴力搜索算法流程

对于输入数组A，遍历每一个起始元素对(x,y)=(A[i],A[j]):

- 计算下一个预期值，y=A[i]+A[j]
- 查找y是否存在于数组A中
- 存在则更新（x,y）=(y,x+y)，否则退出查找，并记录当前序列长度

三. Code 基本解法及编码实现

基础解法： 暴力搜索边界和细节问题

边界问题

- 对于一个起始对，何时结束查找？

细节问题

- 查找一个元素是否存在数组中该如何高效实现？

三. Code 基本解法及编码实现

基础解法： 暴力搜索编码实现

```
class Solution {
    public int lenLongestFibSubseq(int[] A) {
        int N = A.length;
        Set<Integer> S = new HashSet();// 使用set来存储便于快速查找
        for (int x: A) S.add(x);
        int res = 0;
        for (int i = 0; i < N; ++i)
            for (int j = i+1; j < N; ++j) {
                int x = A[j]; // 假设从pair(A[i], A[j]) 开始
                int y = A[i] + A[j];// y 代表 A[i]->A[j] 的下一个斐波那契数列的元素
                int length = 2;
                while (S.contains(y)) { // 继续向后搜索以pair (A[i], A[j])开始的斐波那契数列
                    // x, y -> y, x+y
                    int tmp = y;
                    y += x;
                    x = tmp;
                    res = Math.max(res, ++length);
                }
            }

        return res >= 3 ? res : 0;
    }
}
```

执行结果： 通过 显示详情 >
执行用时： 87 ms ，在所有 Java 提交中击败了 86.42% 的用户
内存消耗： 38.4 MB ，在所有 Java 提交中击败了 95.03% 的用户

三. Code 基本解法及编码实现

基础解法： 暴力搜索复杂度分析

时间复杂度：O(N^2*log(M))

- N是输入数组的长度
- M是输入数组的最大值
- 一共有N(N-1)/2个起始对，O(N ^2)

空间复杂度：O(N)

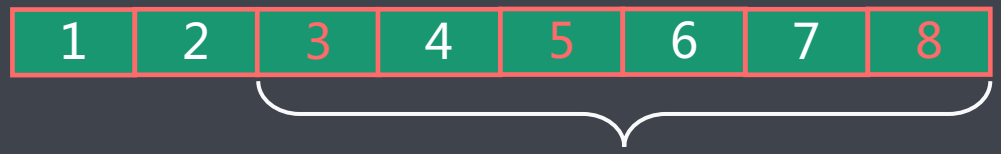
- 需要存储所有组合的元素需要O(N)

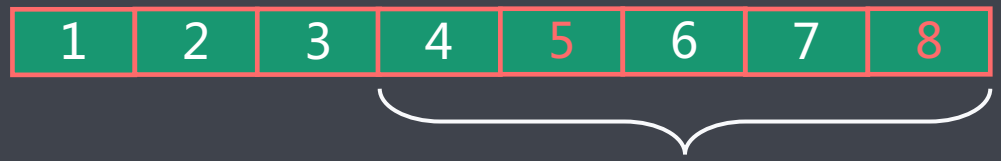
四. Consider 思考更优解

- 在思考更优解的时候需要考虑前面的解法中，何处是重复的可避免的？

输入数组

子序列起始元素组合





- 每一个起始对都要遍历后面的数组剩余的元素
- 能否有一种方法在遍历某一对数(3，5)的时候能否记录某些状态便于下一次遍历？

四. Consider 思考更优解

关键知识点： 动态规划

动态规划来源于**运筹学**，是求解**决策过程最优化**的一种数学方法。在20世纪50年代初美国数学家R. E. Bellman等提出的最优化原理，动态规划的**核心思想**是利用各阶段之间的关系，逐个求解，最终得到全局最优解。设计一个动态规划算法的关键点是确认**原问题与子问题、动态规划的状态、边界状态值和状态转移方程**等。

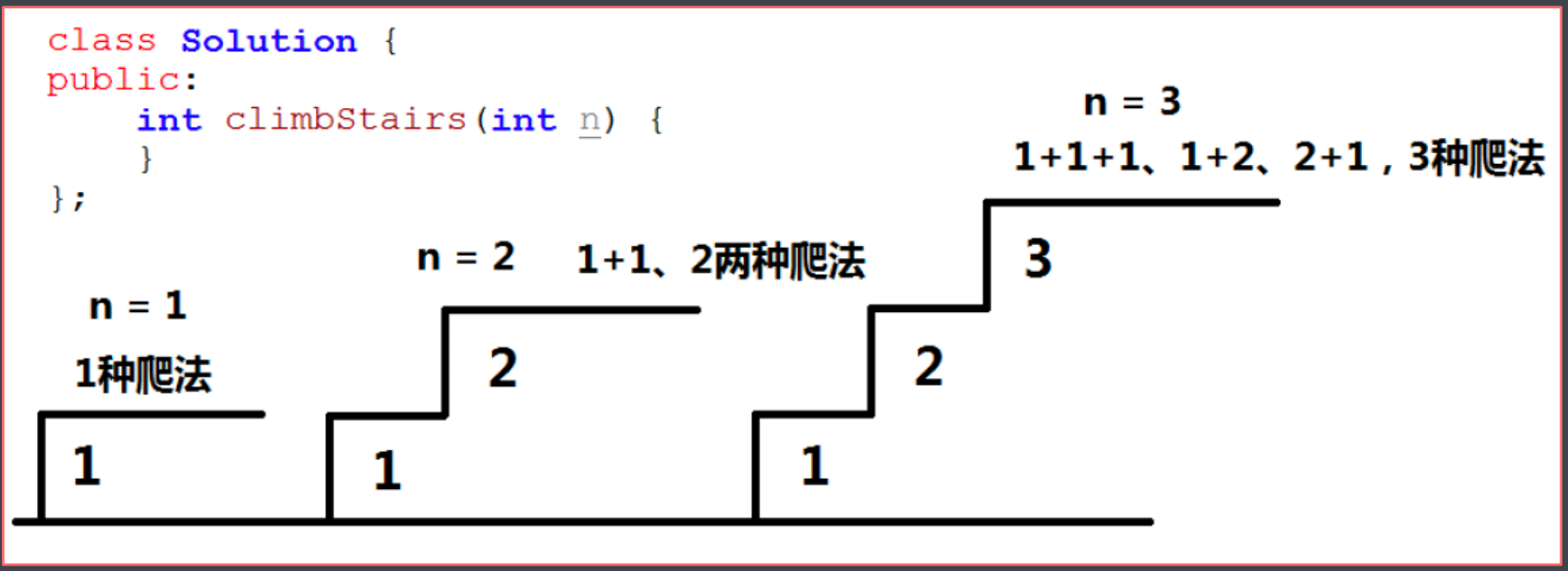
动态规划是面试**必考题爱考题**，特别是很多**大公司核心部门**的面试官（一般他们都喜欢感受巧妙解法解决复杂问题的快乐）。动态规划的题目一般是面试的最后一题，是否做出来是衡量这一轮面试**是否通过的重要指标**！

四. Consider 思考更优解

关键知识点： 动态规划 – 爬楼梯

在爬楼梯的时候，每次可以往上爬1阶台阶或2阶台阶，问n阶楼梯有多少种上楼的方式？

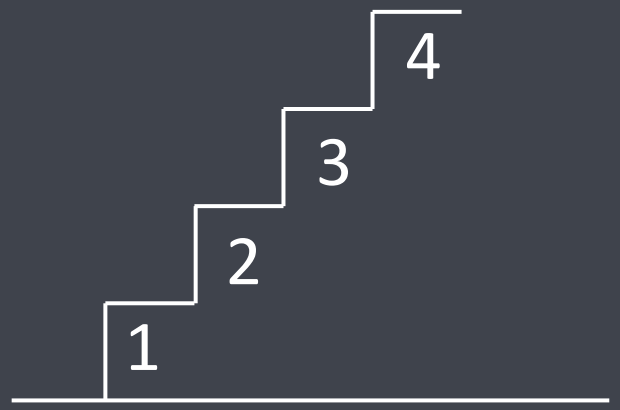
```
class Solution {
public:
    int climbStairs(int n) {
    }
};
```



四. Consider 思考更优解

拉勾教育
—互联网人实战大学—

关键知识点： 动态规划 – 爬楼梯



- 暴力解法中可以使用递归算法

```
class Solution {
    public int climbStairs(int n){
        if(n == 1 || n == 2){
            return n;
        }
        return climbStairs(n-1) + climbStairs(n-2);
    }

  }
```

执行结果： 超出时间限制 显示详情 >

最后执行的输入：

45

四. Consider 思考更优解

关键知识点： 动态规划 – 爬楼梯

分析

- 爬楼梯每次只能爬1阶或2阶
- 对于第i阶，它的上一步一定是到达i-1 或i-2阶
- 到达第i阶的爬法，只与到达i-1阶 i-2阶有关

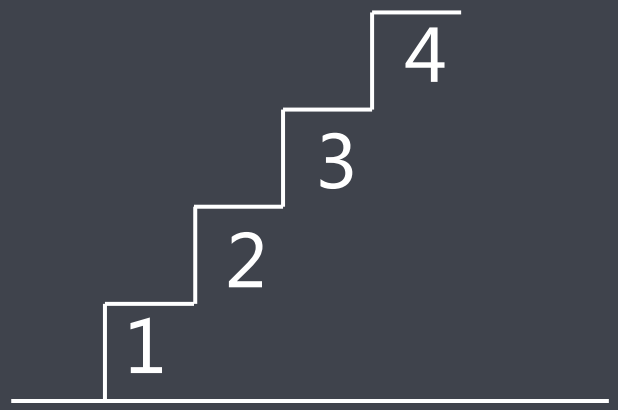
第i阶的爬法数量 = 第i-1阶爬法数量+第i-2阶爬法数量

四. Consider 思考更优解

关键知识点： 动态规划 – 爬楼梯

算法流程

1. 设置状态数组dp[n], dp[i] 代表到达第i阶的爬法数量，初始化为0
2. dp[0]=0, dp[1]=1, dp[2]=2
3. 利用i循环递推到第n阶 dp[i]=dp[i-1]+dp[i-2]



- dp[0]=0, dp[1]=1, dp[2]=2
- dp[3]=dp[2]+dp[1]=3
- dp[4]=dp[3]+dp[2]=5
- dp[5]=dp[4]+dp[3]=8
- ……

四. Consider 思考更优解

拉勾教育
—互联网人实战大学—

关键知识点： 动态规划 – 爬楼梯

```
class Solution {
    public int climbStairs(int n){
        // 注意构建n+3的数组来处理n=0,1,2的特殊情况
       int[] dp= new int[n+3];
       // 确定状态数组的前几个元素初值
       dp[1]=1;
       dp[2]=2;
       // 利用状态转移方程逐步求解最优解
       for(int i=3;i<=n;i++){
           dp[i]=dp[i-1]+dp[i-2];
       }
       return dp[n];
   }
}
```

执行结果： 通过 显示详情 >

执行用时： 0 ms , 在所有 Java 提交中击败了 100.00% 的用户

内存消耗： 34.9 MB , 在所有 Java 提交中击败了 97.80% 的用户

五. Code 最优解思路及编码实现

最优解： 动态规划

动态规划解题步骤

1. 想清楚原问题与子问题

- 爬楼梯中原问题是求n阶台阶走法的数量，子问题是求1，2，3,...,n-1阶走法

2. 设计状态（动态规划核心）

- 爬楼梯的状态可直接从目标值想到，第i个状态表示第i阶走法
- 本题第n个状态值即为目标值，但很多题目标值=f(第n个状态值)

3.设计状态转移方程

- 目标是设计方程从前面的状态值中直接求解第i个状态值
- 爬楼梯中为 dp[i]=dp[i-1]+dp[i-2](i>=3)

4.确定边界状态值

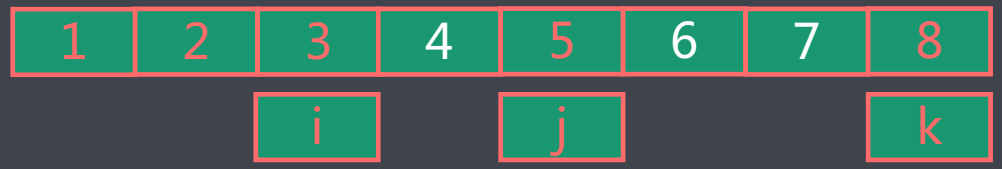
- 爬楼梯中为 dp[0]=0,dp[1]=1,dp[2]=2

五. Code 最优解思路及编码实现

最优解： 动态规划

思路：

- 将斐波那契子序列中的两个连续项 A[i], A[j] 视为单结点 (i, j)，整个子序列是这些连续结点之间的路径
- 例如，对于斐波那契式的子序列 (A[1] = 2, A[2] = 3, A[4] = 5, A[7] = 8, A[10] = 13)，结点之间的路径为 (1, 2) <-> (2, 4) <-> (4, 7) <-> (7, 10)
- 只有当 A[i] + A[j] == A[k] 时，两结点 (i, j) 和 (j, k) 才是连通的，我们需要这些信息才能知道这一连通



五. Code 最优解思路及编码实现

最优解： 动态规划

动态规划解题步骤

1. 想清楚原问题与子问题
   - 子问题为求解结点（i, j) 结尾的斐波那契子序列的长度
2. 设计状态
   - dp（i,j）结点（i, j）结尾的斐波那契子序列的长度
3. 设计状态转移方程
   - if（A[k]+A[i] == A[j]） dp（i, j）=dp（k,i）+1
   - 目标值 = max（ dp（i,j））
4. 确定边界状态值
   - dp（i,j）=2

五. Code 最优解思路及编码实现

最优解： 动态规划边界和细节问题

边界问题

- 输入数组长度小于3时如何处理？
- 对于一个起始对，何时结束查找？

细节问题

- 如何快速查找某个元素在数组中的位置？

五. Code 最优解思路及编码实现

最优解：动态规划编码实现

```
class Solution {
    public int lenLongestFibSubseq(int[] A) {
        int N = A.length;
        Map<Integer, Integer> index = new HashMap();// 建立值到下标的映射便于快速查找
        for (int i = 0; i < N; ++i)
            index.put(A[i], i);

        Map<Integer, Integer> longest = new HashMap();
        int ans = 0;

        for (int k = 0; k < N; ++k)
            for (int j = 0; j < k; ++j) {
                int i = index.getOrDefault(A[k] - A[j], -1);
                if (i >= 0 && i < j) {
                    // 把(i, j) 对应的位置编码成 (i * N + j)，这样就可以用一维数组来存储状态
                    int cand = longest.getOrDefault(i * N + j, 2) + 1; // 利用状态转移方程来求解
                    longest.put(j * N + k, cand);// 记录中间状态值，便于后续求解时调用
                    ans = Math.max(ans, cand);
                }
            }

        return ans >= 3 ? ans : 0;
    }
}
```

执行结果： 通过 显示详情 >
执行用时： 120 ms，在所有 Java 提交中击败了 15.03% 的用户
内存消耗： 38.1 MB，在所有 Java 提交中击败了 99.44% 的用户

五. Code 最优解思路及编码实现

最优解： 动态规划复杂度分析

时间复杂度：O(N^2)

- N是输入数组的长度
- 一共有N(N-1)/2个起始对，O(N ^2)

空间复杂度：O(Nlog(M))

- M是输入数组的最大值
- 可以证明子序列中的元素数量是有限的（复杂度 O(log(M/a))，其中 a 是子序列中最小的元素）

六. Change 变形延伸

题目变形

- （练习）最大子序和 (leetcode 53)

延伸扩展

- 动态规划是面试中的重点考察题目
- 设计状态和状态转移方程是动态规划解题的关键

本题来源

- Leetcode 873 https://leetcode-cn.com/problems/length-of-longest-fibonacci-subsequence/

总结

- 掌握动态规划的概念
- 掌握动态规划的解题步骤
- 掌握动态规划思想在实际问题的运用

课后练习

1. 打家劫舍(Leetcode 198 /简单)
2. 零钱兑换（ Leetcode 322 /中等 ）
3. 最长上升子序列（ Leetcode 300 /中等 ）
4. 做菜顺序（ Leetcode 1402 /困难 ）

课后练习

1. 打家劫舍(Leetcode 198 /简单)

   提示：你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。

```
输入: [1,2,3,1]
输出: 4
解释: 偷窃 1 号房屋 (金额 = 1) , 然后偷窃 3 号房屋 (金额 = 3)。
     偷窃到的最高金额 = 1 + 3 = 4 。
```

课后练习

2. 零钱兑换（Leetcode 322 /中等）

提示：给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。你可以认为每种硬币的数量是无限的。

```
输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1
```

课后练习

3. 最长上升子序列（Leetcode 300 /中等）

提示：给定一个无序的整数数组，找到其中最长上升子序列的长度。

```
输入: [10,9,2,5,3,7,101,18]
输出: 4
解释: 最长的上升子序列是 [2,3,7,101]，它的长度是 4。
```

课后练习

4. 做菜顺序（ Leetcode 1402 /困难 ）

提示：一个厨师收集了他 n 道菜的满意程度 satisfaction ，这个厨师做出每道菜的时间都是 1 单位时间。一道菜的 「喜爱时间」系数定义为烹饪这道菜以及之前每道菜所花费的时间乘以这道菜的满意程度，也就是 time[i]*satisfaction[i] 。请你返回做完所有菜 「喜爱时间」总和的最大值为多少。你可以按 任意 顺序安排做菜的顺序，你也可以选择放弃做某些菜来获得更大的总和。

输入： satisfaction = [-1,-8,0,5,-9]
输出： 14
解释： 去掉第二道和最后一道菜，最大的喜爱时间系数和为 (-1*1 + 0*2 + 5*3 = 14) 。每道菜都需要花费 1 单位时间完成。

拉勾教育

—互联网人实战大学—

下载「拉勾教育App」
获取更多内容