

**课程设计报告**

|  |  |  |
| --- | --- | --- |
| 课程设计题目 | ： | 动态规划算法的应用与实现 |
| 专 业 | ： | 计算机科学与技术 |
| 班 级 | ： | 2220704 |
| 学 号 | ： | 2022212080 |
| 学生姓名 | ： | 刘纪彤 |
| 指导教师 | ： | 刘自强 |
| 日 期 | ： | 2024 年 05 月 09 日 |

算法设计课程设计报告

动态规划算法的应用与实现

[动态规划算法的应用与实现 2](#_Toc166166874)

[1463.摘樱桃II 3](#_Toc166166875)

[问题描述 3](#_Toc166166876)

[解题思路 6](#_Toc166166877)

[代码实现 6](#_Toc166166878)

[2957.消除相邻近似相等字符 7](#_Toc166166879)

[问题描述 7](#_Toc166166880)

[解题思路 8](#_Toc166166881)

[代码实现 8](#_Toc166166882)

[代码优化 9](#_Toc166166883)

[总结 9](#_Toc166166884)

[数据结构与算法设计课程设计评分表 10](#_Toc166166885)

## [1463.摘樱桃II](https://leetcode.cn/problems/cherry-pickup-ii/)

### 问题描述

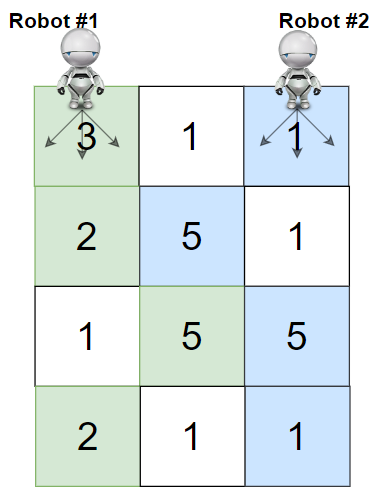
给你一个 rows x cols 的矩阵 grid 来表示一块樱桃地。 grid 中每个格子的数字表示你能获得的樱桃数目。

你有两个机器人帮你收集樱桃，机器人 1 从左上角格子 (0,0) 出发，机器人 2 从右上角格子 (0, cols-1) 出发。

请你按照如下规则，返回两个机器人能收集的最多樱桃数目：

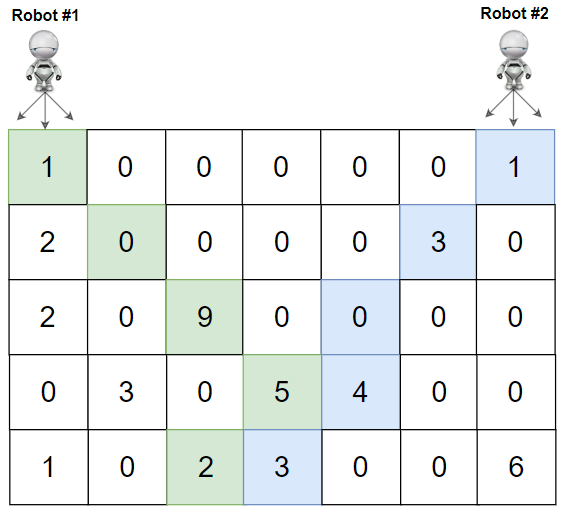
* 从格子 (i,j) 出发，机器人可以移动到格子 (i+1, j-1)，(i+1, j) 或者 (i+1, j+1) 。
* 当一个机器人经过某个格子时，它会把该格子内所有的樱桃都摘走，然后这个位置会变成空格子，即没有樱桃的格子。
* 当两个机器人同时到达同一个格子时，它们中只有一个可以摘到樱桃。
* 两个机器人在任意时刻都不能移动到 grid 外面。
* 两个机器人最后都要到达 grid 最底下一行。

**示例 1：**



输入：grid = [[3,1,1],[2,5,1],[1,5,5],[2,1,1]] 输出：24 解释：机器人 1 和机器人 2 的路径在上图中分别用绿色和蓝色表示。 机器人 1 摘的樱桃数目为 (3 + 2 + 5 + 2) = 12 。 机器人 2 摘的樱桃数目为 (1 + 5 + 5 + 1) = 12 。 樱桃总数为： 12 + 12 = 24 。

**示例 2：**



输入：grid = [[1,0,0,0,0,0,1],[2,0,0,0,0,3,0],[2,0,9,0,0,0,0],[0,3,0,5,4,0,0],[1,0,2,3,0,0,6]] 输出：28 解释：机器人 1 和机器人 2 的路径在上图中分别用绿色和蓝色表示。 机器人 1 摘的樱桃数目为 (1 + 9 + 5 + 2) = 17 。 机器人 2 摘的樱桃数目为 (1 + 3 + 4 + 3) = 11 。 樱桃总数为： 17 + 11 = 28 。

**示例 3：**

输入：grid = [[1,0,0,3],[0,0,0,3],[0,0,3,3],[9,0,3,3]] 输出：22

**示例 4：**

输入：grid = [[1,1],[1,1]] 输出：4

**提示：**

* rows == grid.length
* cols == grid[i].length
* 2 <= rows, cols <= 70
* 0 <= grid[i][j] <= 100

### 解题思路

先进行状态定义，定义 dfs(i,j,k)表示，A从(i,j)出发，B从(i,k)出发，到达最后一行所能够获得的最大樱桃数目。A和B均有三种情况，分别是向左、向右、向下，也就是有了9种情况，我们可以用一个三维数组 dfs(i,j,k)来存储

得到他的状态转移方程如下所示：

其中的 val = grid[i][j] + grid[i][k](j!=k)或 val=gird[i][j](j==k)，表示当前位置的所能采摘的樱桃数目。

做好出界判定，我们就对所有出界情况返回0，同时作为递归的终止条件。

为了降低时间复杂度，我们可以通过存储已经遍历过的数组来减少大量不需要的重复计算，达到降低时间复杂度的效果。

### 代码实现

class Solution {  
public:  
 int dfs(vector<vector<vector<int>>>& memo, vector<vector<int>>& grid, int i,  
 int j, int k) {  
 int m = grid.size(), n = grid[0].size();  
 // 越界处理  
 if (i == m || j < 0 || j >= n || k < 0 || k >= n)  
 return 0;  
 int& res = memo[i][j][k];  
 if (res != -1)  
 return res;  
 for (int j2 = j - 1; j2 <= j + 1; j2++) {  
 for (int k2 = k - 1; k2 <= k + 1; k2++) {  
 res = max(res, dfs(memo, grid, i + 1, j2, k2)); // 递归求解  
 }  
 }  
 res += grid[i][j];  
 if (k == j)  
 return res;  
 else {  
 res += grid[i][k];  
 return res;  
 }  
 }  
 int cherryPickup(vector<vector<int>>& grid) {  
  
 vector<vector<vector<int>>> memo(  
 grid.size(), vector<vector<int>>(  
 grid[0].size(),  
 vector<int>(grid[0].size(), -1))); //-1表示没有记忆  
 return dfs(memo, grid, 0, 0, grid[0].size() - 1);  
 }  
};

代码时间复杂度为，空间复杂度为。因为状态共种，每种状态只计算一次，所以时间复杂度为。存储需要存储种状态，所以空间复杂度为。

## [2957.消除相邻近似相等字符](https://leetcode.cn/problems/remove-adjacent-almost-equal-characters/)

### 问题描述

给你一个下标从 **0** 开始的字符串 word 。

一次操作中，你可以选择 word 中任意一个下标 i ，将 word[i] 修改成任意一个小写英文字母。

请你返回消除 word 中所有相邻 **近似相等** 字符的 **最少** 操作次数。

两个字符 a 和 b 如果满足 a == b 或者 a 和 b 在字母表中是相邻的，那么我们称它们是 **近似相等** 字符。

**示例 1：**

输入：word = "aaaaa" 输出：2 解释：我们将 word 变为 "acaca" ，该字符串没有相邻近似相等字符。 消除 word 中所有相邻近似相等字符最少需要 2 次操作。

**示例 2：**

输入：word = "abddez" 输出：2 解释：我们将 word 变为 "ybdoez" ，该字符串没有相邻近似相等字符。 消除 word 中所有相邻近似相等字符最少需要 2 次操作。

**示例 3：**

输入：word = "zyxyxyz" 输出：3 解释：我们将 word 变为 "zaxaxaz" ，该字符串没有相邻近似相等字符。 消除 word 中所有相邻近似相等字符最少需要 3 次操作

**提示：**

* 1 <= word.length <= 100
* word 只包含小写英文字母。

### 解题思路

先确定状态定义：创建初始数组，那么我们就设置一个dp数组用来存储次数，dp[i]就表示消除前i个字符中所有相邻的近似相等字符的最小操作次数。

确定状态转移方程：

当word[i]和word[i-1]近似相等之时，我们选择消除word[i-1]，保证了他与word[i-2]、word[i]互不相等，加上原有次数后dp[i] = dp[i-2] + 1，再进行继续搜索操作。直到整个字符串被遍历完毕。

对于初始状态的dp[0]，我们置零，对于dp[1]我们按照上述规则，如果0和1近似相等，我们就置dp[1] = 1，否则dp[1] = 0。

### 代码实现

class Solution {   
public:   
 bool isapprox(char a, char b) {   
 return (a - b) == 1 || a == b || (b - a) == 1;   
 }   
 int removeAlmostEqualCharacters(string word) {   
   
 int n = word.size();   
 if (n == 1)   
 return 0;   
 vector<int> dp(n);   
 dp[0] = 0;   
 if (isapprox(word[0], word[1]))   
 dp[1] = 1;   
 else   
 dp[1] = 0;   
 for (int i = 2; i < word.size(); i++) {   
 if (isapprox(word[i], word[i - 1]))   
 dp[i] = dp[i - 2] + 1;   
 else   
 dp[i] = dp[i - 1];   
 }   
 return dp[n - 1];   
 }   
};

### 代码优化

在这串代码中我们不难发现，其时间、空间复杂度均达到了O(n),再观察发现，我们并不需要存储所有的dp数组，我们只需要存储dp[i-2]和dp[i-1]即可，所以我们可以将空间复杂度降低到O(1)。

class Solution {  
public:  
 bool isapprox(char a, char b) {  
 return (a - b) == 1 || a == b || (b - a) == 1;  
 }  
 int removeAlmostEqualCharacters(string word) {  
  
 int n = word.size();  
 if (n == 1)  
 return 0;  
 vector<int> dp(2);  
 dp[0] = 0;  
 if (isapprox(word[0], word[1]))  
 dp[1] = 1;  
 else  
 dp[1] = 0;  
 for (int i = 2; i < word.size(); i++) {  
 int tmp = dp[1];  
 if (isapprox(word[i], word[i - 1])) {  
 dp[1] = dp[0] + 1;  
 dp[0] = tmp;  
 } else {  
 dp[0] = dp[1];  
 }  
 }  
 return dp[1];  
 }  
};

此时的空间复杂度就降低到了O(1)

## 总结

本次算法设计课程设计，我们通过两道题目的实现，对动态规划算法有了更深一步的理解，同时也对如何使用动态规划解决一些问题有了一定认识，动态规划的大致思路就是找到状态定义，找到状态转移方程，然后通过递归或者迭代的方式进行求解，同时我们也可以通过存储已经遍历过的数组来减少大量不需要的重复计算，达到降低时间复杂度的效果。参考示例一，我们可以看到，我们通过存储已经遍历过的数组，将时间复杂度从降低到了，这就是动态规划所带来的好处。

# 数据结构与算法设计课程设计评分表

班级：2220704 姓名： 刘纪彤 学号：2022212080

|  |  |  |  |
| --- | --- | --- | --- |
|  | 项目内容 | 满分 | 实 评 |
|
| 程  序  设  计 | 能体现创造性思维，或有独特见解 | 10 |  |
| 利用数据结构课程所学知识，设计并完成任务书所列功能的完整性 | 15 |  |
| 代码规范性 | 5 |  |
| 代码注视详细程度 | 10 |  |
| 实  验  报  告 | 内容完整性 | 8 |  |
| 测试用例 | 10 |  |
| 综合评价 | 14 |  |
| 报告排版质量 | 8 |  |
| 平时  成绩 | 平时表现（以考勤为主） | 20 |  |
|  | 总 分 | 100 |  |
| 指导教师评语：  指导教师签名：  年 月 日 | | | |