

算法设计与应用 作业3

姓名：TRY

学号：

专业：计算机科学与技术

1. 【Leetcode 455】分发饼干

(1) 算法思路

- 这道题是比较简单的**贪心问题**。
- 可以用以下**两个思路**来设计算法：
 - 给一个孩子的饼干 应当尽量小并且又能满足该孩子，这样大饼干才能拿来给满足度比较大的孩子。
 - 满足度最小的孩子最容易得到满足，所以应先满足满足度最小的孩子。
- 因此，可以编写下面的算法，得到**最优的分配方法**：
 - 将胃口数 g 数组和饼干大小数组 s ，进行**从小到大的排序**。
 - 从小到大开始**遍历两个数组**：
 - 如果当前 $g[i] \leq s[j]$ ，则可以满足该孩子的胃口， $result++$ ， $i++$ ， $j++$ 。
 - 否则， $j++$ ，看下一个饼干的大小能否满足对应孩子的胃口。
 - 当 i 或 j 到达对应数组的结尾时，跳出循环，返回结果值 $result$ 。

(2) 复杂度分析

- **时间复杂度**： $O(m \log m)$ ，其中， $m = \max(n_1, n_2)$ ， n_1, n_2 分别为 g ， s 数组的大小
 - sort 算法的复杂度为 $O(n \log n)$
 - while 循环的复杂度是 $O(n_1 + n_2)$
 - 所以总的时间复杂度是 $O(m \log m)$ 。
- **空间复杂度**： $O(1)$ ，只开辟了常数变量的大小

(3) 代码

```
int findContentChildren(vector<int>& g, vector<int>& s) {
    sort(g.begin(), g.end());
    sort(s.begin(), s.end());
    int i = 0, j = 0;
    int result = 0;
    while (i != g.size() && j != s.size())
    {
        if (g[i] <= s[j])
        {
            result++;
        }
        i++;
        j++;
    }
}
```

```
        i++;
        j++;
    }
    else
        j++;
}
return result;
}
```

(4) 截图

执行结果：通过 [显示详情](#)

执行用时：**84 ms**，在所有 C++ 提交中击败了 **79.61%** 的用户

内存消耗：**16.9 MB**，在所有 C++ 提交中击败了 **6.98%** 的用户

2. 【Leetcode 984】不含AAA或BBB的字符串

(1) 算法思路

- 同样，这是一道简单的**贪心算法**
- 由样例可知，把握以下**两个思路**编写算法：
 - 在允许情况下，先写入**当前字母数量多**的字母 x 。（否则，会出现大量字母连在一起的情况，不符合题目要求）
 - 如果前2个字母已经是 x 的情况下，则写入另一个字母。
- 因此，可以编写下面思路的**贪心算法**，得到字符串：
 - 定义变量 `atemp`, `btemp`，分别记录当前位置前面连续的 `a`, `b` 字母的个数，初始化为 0。
 - 选择当前数量最多的字母 x ，并试图写入 x ：
 - 当 x 的 `xtemp` 值小于2时，说明可以写入字母 x ，则写入字母 x ，并通过 `x--` 来更新 x （当前字母的剩余量），`xtemp++`, `yemp=0`。
 - 否则，说明不可以写入字母 x ，则写入字母 y ，并更新：`y--`, `ytemp++`, `xtemp=0`。
 - 一直循环，直到字符串的长度等于原来的 `A+B`，退出循环。
- **注意：**由于题目中说明“对于输入的A,B，必定存在字符串s满足要求”，所以在 **x 不满足要求的时候，可直接写入 y** 。

(2) 复杂度分析

- **时间复杂度：** $O(A+B)$
 - 循环进行A+B次
- **空间复杂度：** $O(1)$ ，只开辟了常数空间

(3) 代码

- 处理一：清晰分类

```
string strWithout3a3b(int A, int B) {
    int i = 0;
    string s;
    int atemp = 0, btemp = 0;
    int temp = A + B;
    while (s.size() < temp)
    {
        if (A > B)
        {
            if (atemp < 2)
            {
                s.push_back('a');
                A--;
                atemp++;
                btemp = 0;
            }
            else
            {
                s.push_back('b');
                B--;
                atemp = 0;
                btemp++;
            }
        }
        else
        {
            if (btemp < 2)
            {
                s.push_back('b');
                B--;
                atemp = 0;
                btemp++;
            }
            else
            {
                s.push_back('a');
                A--;
                btemp = 0;
                atemp++;
            }
        }
    }
    return s;
}
```

- 处理二：合并分类

```
string strWithout3a3b(int A, int B) {
    string s;
    int atemp = 0, btemp = 0;
    int temp = A + B;
    while (s.size() < temp)
    {
```

```

    if (A > B && atemp < 2 || A <= B && btemp == 2)
    {
        s.push_back('a');
        A--;
        atemp++;
        btemp = 0;
    }
    else
    {
        s.push_back('b');
        B--;
        atemp = 0;
        btemp++;
    }
}
return s;
}

```

(4) 截图

执行结果：通过 [显示详情](#)

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

内存消耗：**6.3 MB**，在所有 C++ 提交中击败了 100.00% 的用户

3. 【Leetcode 120】三角形最小路径和

(1) 算法思路

- 这是一道中等难度的**动态规划问题**。（问题本身比较简单，但**难在优化**）
- 一开始看到题目，我的第一反应是“**回溯算法**”。但由于本题的**三角形并不是二叉树**，并且由于三角形的特殊性，有一定的**下标规律**可以完成求解，所以用**动态规划**即可完成求解。
- 问题的目标是**求三角形的最小路径和**，也就是要求出每一条从根节点到子节点的路径的距离之和，并比较取最小值。利用性质“**相邻的结点 在这里指的是 下标 与 上一层结点的下标 相同或者等于 上一层结点的下标 + 1 的两个结点**”即可知道：**到本单元的路径之和的最小值=上一行相邻节点的路径之和的最小值+本单元的元素值**。
 - 需要注意的是：在**每一行两边的数**在上一行只有1个相邻节点，所以需要特别处理。
- **【法一：空间O(n) + top-down】**
 - 题目中有要求，“**设计一个只使用 O(n) 的额外空间的算法**”。因此，可以定义数组 dp，大小：三角形的行数 n。在每一轮循环中定义数组 dp1，利用上一轮循环保存的结果 dp 得到本轮循环的结果，保存到 dp1 中，并在保存完成后，通过 dp=dp1 赋值，为下一轮循环做准备。直到进行到最后一行。
 - 这里**一定要设置两个数组 dp 和 dp1**，否则，会造成更新出错。
 - 上一行的1个的元素已经被更新，无法利用来进行比较。
 - 可以使用 stl 函数 min_element **求数组的最小值**。

- 【法二：空间 $O(1)$ + top-down】

- 其实我们可以直接在 `triangle` 数组原地实现更新。这样，就不需要开辟额外的空间，实现了内存的优化！！

- 【法三：空间 $O(1)$ + bottom-up】

- 由上面的性质可知，我们可以知道：可以实现top-down的算法，也可以实现bottom-up的算法。这样，就不需要进行额外的边界处理，可以统一用一句语句来实现计算。

(2) 复杂度分析

- 时间复杂度：

- 法一&法二&法三： $O(n^2)$ ， n 为三角形的行数。
 - 外层循环共 $n-1$ 次，内存循环遍历次数为对应的外层循环遍历的下标值。
 - 因此， $1+2+3+\dots+n = O(n^2)$ 。

- 空间复杂度：

- 法一： $O(n)$ ， n 为三角形的行数。
 - 每一轮循环开辟数组`dp1`，且每一轮循环会自己释放空间，下次再重新开。所以，复杂度为 $O(n)$ 。
- 法二&法三： $O(1)$ ，只开辟了常数变量的空间。

(3) 代码

- 法一：空间 $O(n)$ + top-down

```
int minimumTotal(vector<vector<int>>& triangle) {
    vector<double> dp(triangle.size(), 0);
    dp[0] = triangle[0][0]; // 第0行初始化
    for (int i = 1; i < triangle.size(); i++)
    {
        vector<double> dp1(triangle.size(), 0); // 本轮结果储存
        for (int j = 0; j <= i; j++)
        {
            if (j == 0) // 左边界
                dp1[0] = triangle[i][0] + dp[0];
            else if (j == i) // 右边界
                dp1[j] = triangle[i][j] + dp[j - 1];
            else
                dp1[j] = min(dp[j - 1], dp[j]) + triangle[i][j];
        }
        dp = dp1; // 重新赋值，为下一轮做准备
        // dp1.clear(); !!!不用自己删的！因为这是局部变量，这个大括号执行完之后会自己
delete掉
    }
    auto result = (int)*(min_element(dp.begin(), dp.end())); // 使用stl函数
    return result;
}
```

- 法二：空间O(1) + top-down

```
int minimumTotal(vector<vector<int>>& triangle) {
    for (int i = 1; i < triangle.size(); i++)
    {
        for (int j = 0; j <= i ; j++)
        {
            if (j == 0)//边界处理1
                triangle[i][j] += triangle[i - 1][0];
            else if (j == i)//边界处理2
                triangle[i][j] += triangle[i - 1][i - 1];
            else
                triangle[i][j] += min(triangle[i - 1][j - 1], triangle[i - 1][j]);
        }
    }
    auto result = (int)*(min_element(triangle[triangle.size()-1].begin(), triangle[triangle.size() - 1].end()));
    return result;
}
```

- 法三：空间O(1) + bottom-up

```
int minimumTotal(vector<vector<int>>& triangle) {
    for (int i = triangle.size()-2; i >=0; i--)
    {
        for (int j = 0; j <= i; j++)
            triangle[i][j] += min(triangle[i + 1][j], triangle[i + 1][j + 1]);
    }
    return triangle[0][0];
}
```

(4) 截图

- 法一的截图：

执行结果： 通过 [显示详情](#) >

执行用时： **8 ms** ，在所有 C++ 提交中击败了 **92.77%** 的用户

内存消耗： **8.9 MB** ，在所有 C++ 提交中击败了 **100.00%** 的用户

- 法二的截图：内存优化（原地做计算）

执行结果： 通过 [显示详情](#) >

执行用时： **8 ms** ，在所有 C++ 提交中击败了 **92.77%** 的用户

内存消耗： **8.3 MB** ，在所有 C++ 提交中击败了 **100.00%** 的用户

- 法三的截图：时间复杂度最优 (bottom-up)

执行结果： 通过 [显示详情](#)

执行用时： 4 ms，在所有 C++ 提交中击败了 99.74% 的用户

内存消耗： 8.3 MB，在所有 C++ 提交中击败了 100.00% 的用户

4. 【Leetcode 714】买卖股票的最佳时机含手续费

(1) 算法思路

- 法一：动态规划

- 这是一道**比较难**的动态规划。（难在想到要用DP而不是贪心算法）
- 首先，由于股票其实只需要知道前一天的最大收益，而不需要知道再前面的，所以，可以利用两个变量 `sell` 和 `hold` 来表示整个数组，作为DP的**状态记录变量**。
- **状态定义：**
 - **sell变量**：表示到当前天数，不持股（卖了股票）即持有现金的最大收益
 - **hold变量**：表示到当前天数，持有股票的最大收益。
 - 由于在**一天内，只能持有股票或卖出股票**，所以，可以用这两种状态来表示。
 - **注意**：不会在一天内又卖出有买入，因为这样相当于浪费了一次手续费，而没有任何收益，没有意义。
 - **约定**：在卖出的时候才扣除手续费。

- **状态转移方程：**

- **sell**：当天不持股，可以由昨天不持股 或者 昨天持股但转换而来。
 - 昨天不持股：说明今天什么都没做，只拿着现金。
 - 昨天持股：说明今天卖出了股票。并且需要扣除手续费。
 - 因此，可以得到以下公式：

```
sell = max(hold + prices[i] - fee, sell);
```

- **hold**：当天持股，可以由昨天不持股 和 昨天持股转换而来。
 - 昨天不持股：说明今天买入了股票。
 - 昨天持股：说明今天什么也没做，仍然持股。
 - 因此，可以得到以下公式：

```
hold = max(hold, sell - prices[i]);
```

- **注意**：这里不需要使用临时变量保存昨天的sell和hold，再更新，原因如下：

- 由于昨天的情况已定，所以**今天的最大利益只会有一种情况导致**：买入、卖出、保持昨天。
- 所以，**在一天中，hold和sell最多只会有一个值发生改变**。（买入hold变化，卖出sell变化，不买不卖hold和sell都不变）
- 也可以通过代入来解释。
- **初始化：**
 - 将 `sell` 初始化为 `0`，`hold` 初始化为 `-prices[0]`。
 - 买入时，是在当天就减去费用，所以hold初始化为负值。
- **结果：**
 - 最后一定要**卖完持有的股票**，所以最后返回遍历完成之后的 `sell`。

• 法二：贪心算法

- 其实我的第一反应是贪心算法，但debug了一个多小时还是没有写出来，所以分享某位大佬的[题解](#)。过程非常的巧妙，是利用了波峰波谷来进行计算。
- 有手续费，意味着把峰值降低。
- 且这个方法最妙的是使用了 `in_hand=prices[i]-fee` 作为今天的价格。（如果想通这点，就可以利用贪心完成了）
- 以下，不再过多赘述。（但是，实在是超级妙！忍不住分享！比DP难想多了）

(2) 复杂度分析

- 时间复杂度： $O(n)$ ， n 为数组的长度
- 空间复杂度： $O(1)$ ，常数空间。

(3) 代码

- 动态规划算法：

```
int maxProfit(vector<int>& prices, int fee) {
    if (prices.size() == 1)
        return 0;
    int sell = 0;
    int hold = -prices[0]; // 买的时候没有算手续费，卖才算
    for (int i = 1; i < prices.size(); i++)
    {
        sell = max(hold + prices[i] - fee, sell);
        hold = max(hold, sell - prices[i]);
    }
    return sell;
}
```

(4) 截图

执行结果: 通过 [显示详情](#) >


执行用时: **280 ms** , 在所有 C++ 提交中击败了 **34.16%** 的用户

内存消耗: **51.7 MB** , 在所有 C++ 提交中击败了 **16.67%** 的用户

5. 【Leetcode 91】解码游戏

注: 由于本题我已经在三个月前完成并写过[题解](#), 所以, 以下部分内容来自我的题解。

 **C++ 双100% 一看就懂的动态规划 解码方法**

sleeping monster 发布于 2020-04-10  1.4k C++ 动态规划

(1) 算法思路

- 这是一道非常典型的**动态规划问题**。由于需要使用到 $v[i-2]$ 和 $v[i-1]$, 所以开辟数组 v 。(其实, 令两个变量也可以)
- 状态定义:**
 - 先初始化一个 v 数组, $v[i]$ 记录对应到 $s[i]$ 位置的解码方法的总数。
- 状态转移方程:**
 - 当 $s[i-1]$ 不是 "1" 或 "2" 的时候, 说明 $s[i]$ 不可以和 $s[i-1]$ 联合解码
 - 当 $s[i]$ 是 "0" 的时候, $s[i]$ 无法解码, 返回 0
 - 当 $s[i]$ 不是 "0" 的时候, $s[i]$ 位置独立解码, 此时 $v[i]=v[i-1]$
 - 当 $s[i-1]$ 是 "1" 或 "2" 的时候, 说明 $s[i]$ 可以和 $s[i-1]$ 联合解码
 - 如果 $s[i-1]s[i]$ 组成的数字小于等于 26, 则可以联合解码, $v[i]=v[i-1]+v[i-2]$
 - 否则, 还是不可以联合解码, $s[i]$ 独立解码, $v[i]=v[i-1]$
- 初始化:**
 - $v[0]=1$.
 - $v[1]$ 根据是否可以联合解码来进行初始化。
- 结果:**
 - 如果中间有不满足解码规则, 直接返回 0
 - 最后返回 $v[n-1]$.

(2) 复杂度分析

- 时间复杂度:** $O(n)$, n 为数组的大小
- 空间复杂度:** $O(n)$, n 为数组的大小。(开辟了数组 v)

(3) 代码

```

int numDecodings(string s) {
    int n = s.size();
    if (s[0] == '0')
        return 0;
    if (n == 1)
        return 1;
    vector<int> v(n, 0);
    v[0] = 1;
    if (s.substr(0, 2) <= "26")
        v[1] = s[1] == '0' ? 1 : 2;
    else
        v[1] = s[1] == '0' ? 0 : 1;
    for (int i = 2; i < n; i++)
    {
        if (s[i - 1] != '1' && s[i - 1] != '2') // s[i] 不可能与 s[i-1] 一起解码
        {
            if (s[i] == '0') // s[i] 位置的 0 无法解码，返回 0
                return 0;
            else // s[i] 位置独立解码
                v[i] = v[i - 1];
        }
        else // s[i] 可能与 s[i-1] 联合解码
        {
            if (s[i] == '0')
                v[i] = v[i - 2];
            else
                v[i] = (s.substr(i - 1, 2) <= "26") ? (v[i - 1] + v[i - 2]) :
v[i - 1];
        }
    }
    return v[n - 1];
}

```

(4) 截图

执行结果：通过 [显示详情 >](#)

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

内存消耗：**6.5 MB**，在所有 C++ 提交中击败了 100.00% 的用户