算法课补分题作业

2154312 郑博远

习题1扩展：买卖股票的最佳时机

- 扩展一 设计一个算法来计算你所能获取的最大利润。你最多可以完成两笔交易。不能同时参与多笔交易（必须在再次购买前出售掉之前的股票）。
- 扩展二 设计一个算法来计算你所能获取的最大利润。你最多可以完成K笔交易。不能同时参与多笔交易（必须在再次购买前出售掉之前的股票）。

扩展一其实是关于扩展二当`K = 2`时的特解，因此只需要思考K笔交易时的情形，再代入K = 2的条件即可。对于扩展二，考虑如下数组以记录状态：

$$dp[i][k][j]$$

表示第i天时对于第k笔交易买入($j = 1$)或卖出($j = 0$)情况下的最大收益值。容易得到，对于第i天的第k笔交易，若是买入状态，则其最大收益值可能维持前一天的买入状态($dp[i-1][k][1]$)，也可能是在前一天卖出第$k-1$笔交易的情况下当天买入($dp[i-1][k-1][0] - prices[i-1]$)；同理，若是卖出状态，则其最大收益值可能维持前一天的卖出状态($dp[i-1][k][0]$)，也可能是在前一天买入第$k-1$笔交易的情况下当天卖出($dp[i-1][k][1] + prices[i-1]$)。综上所述有如下转移方程：

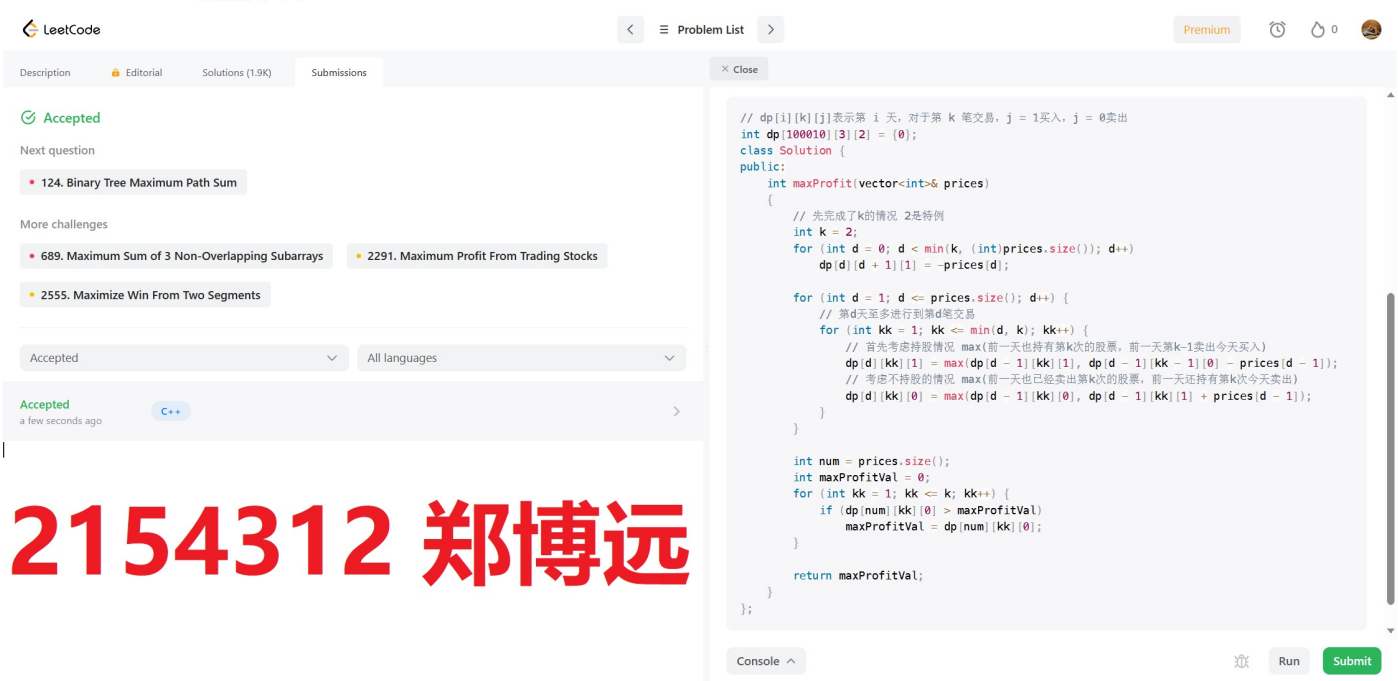
$$dp[i][k][j] = \begin{cases} \max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i-1]), & j = 1 \\ \max(dp[i-1][k][0], dp[i-1][k][1] + prices[i-1]), & j = 0 \end{cases}$$

代码如下：

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;

#define KMAX 100

class Solution {
public:
```

```cpp
    int maxProfit(int k, vector<int>& prices)
    {
        // dp[i][k][j]表示第 i 天，对于第 k 笔交易，j = 1买入，j = 0卖出
        int dp[1010][KMAX + 10][2] = {0};
        for (int d = 0; d < min(k, (int)prices.size()); d++)
            dp[d][d + 1][1] = -prices[d];

        for (int d = 1; d ≤ prices.size(); d++) {
            // 第d天至多进行到第d笔交易
            for (int kk = 1; kk ≤ min(d, k); kk++) {
                // 首先考虑持股情况 max(前一天也持有第k次的股票，前一天第k-1卖出今天买入)
                dp[d][kk][1] = max(dp[d - 1][kk][1], dp[d - 1][kk - 1][0] - prices[d - 1]);
                // 考虑不持股的情况 max(前一天也已经卖出第k次的股票，前一天还持有第k次今天卖出)
                dp[d][kk][0] = max(dp[d - 1][kk][0], dp[d - 1][kk][1] + prices[d - 1]);
            }
        }

        int num = prices.size();
        int maxProfitVal = 0;
        for (int kk = 1; kk ≤ k; kk++) {
            if (dp[num][kk][0] > maxProfitVal)
                maxProfitVal = dp[num][kk][0];
        }

        return maxProfitVal;
    }
};


int main()
{
    vector<int> nums = { 70, 49, 62, 89, 60 };
    Solution s;
    cout << s.maxProfit(3, nums);
    return 0;
}
```

以下是两题的Leetcode截图：



习题2扩展：最长回文子串

1. 给你一个字符串s，找出其中最长回文子序列，并返回该序列的长度。

   子序列定义为：不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。

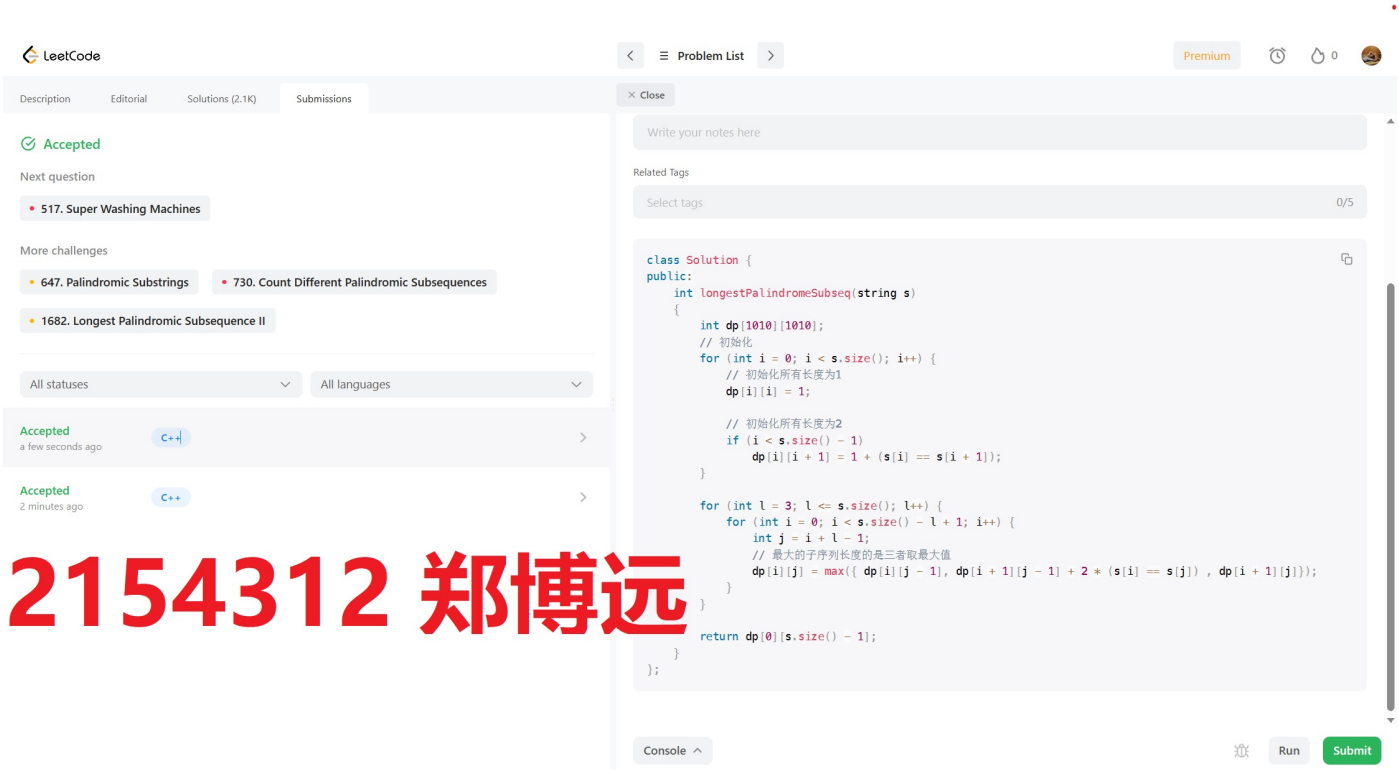
定义$dp[i][j]$表示从i到j的最长回文子串长度，有如下的状态转移方程：

$$dp[i][j] = \max(dp[i][j-1], dp[i+1][j-1] + 2 \cdot (s[i] == s[j]), dp[i+1][j])$$

因此可以写出代码如下：

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;

class Solution {
public:
    int longestPalindromeSubseq(string s)
    {
        int dp[1010][1010];
        // 初始化
        for (int i = 0; i < s.size(); i++) {
            // 初始化所有长度为1
            dp[i][i] = 1;

            // 初始化所有长度为2
            if (i < s.size() - 1)
                dp[i][i + 1] = 1 + (s[i] == s[i + 1]);
        }

        for (int l = 3; l ≤ s.size(); l++) {
            for (int i = 0; i < s.size() - l + 1; i++) {
                int j = i + l - 1;
                // 最大的子序列长度的是三者取最大值
                dp[i][j] = max({ dp[i][j - 1], dp[i + 1][j - 1] + 2 * (s[i] == s[j]) , dp[i + 1][j]});
            }
        }

        return dp[0][s.size() - 1];
    }
};

int main()
{
    string s = "bbbab";
    Solution sol;
    cout << sol.longestPalindromeSubseq(s);
    return 0;
}
```

以下是Leetcode截图：



2. 给定一个字符串s，你可以通过在字符串前面添加字符s'将其转换为回文串。找到并返回可以用这种方式转换的最短回文串。

若采用与课堂例题相似的方式求回文串长度则会TLE。因此采用Manacher算法求每个位置的最长子串。即记录下当前最靠右的回文子串始末位置，若遍历到的下标小于回文子串的末位置，则可以用对称的方式复制此前计算出的值。具体解释在代码注释中给出。

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;

class Solution {
public:
    string shortestPalindrome(string _s)
    {
        int n0 = _s.size();
        // 用'#'隔开每个字符用于之后的计算
        string s(2 * n0 + 1, '#');
        int n = n0 * 2 + 1;
        for (int i = 0; i < n0; i++)
            s[2 * i + 1] = _s[i];

        // Manacher 求每个位置的最长子串
        vector<int> d(n);
        for (int i = 0, l = 0, r = -1; i < n; i++) {
            // 如果超出了当前的最大r 就要用朴素算法求解
```

```
            // 否则则可以用对称的位置解决（超出部分继续用朴素算法）
            int k = (i > r) ? 1 : min(d[l + r - i], r - i + 1);
            // 朴素的逐步扩大求解
            while (0 ≤ i - k && i + k < n && s[i - k] == s[i + k]) {
                k++;
            }
            d[i] = k--;
            // 更新当前的l和r
            if (i + k > r) {
                l = i - k;
                r = i + k;
            }
        }

        int minAdd = n0;
        for (int i = 0; i < n - 1; i++) {
            // 回文串的起始位置必须是字符串头 否则无用
            if (d[i] == i + 1)
                // d[i]的长度是回文字符串的串长 + 1
                // 计算最小的串长
                minAdd = min(minAdd, n0 - d[i] + 1);
        }

        string ans;
        // 倒序添加字符串末尾
        ans.assign(_s.rbegin(), _s.rbegin() + minAdd);
        ans += _s;
        return ans;
    }
};

int main()
{
    string s = "abcd";
    Solution sol;
    cout << sol.shortestPalindrome(s);
    return 0;
}
```

以下是Leetcode截图：

3. 给你两个单词 word1和 word2，请返回将 word1 转换成 word2所使用的最少操作数(最小编辑距离)。

   操作类型：插入一个字符、删除一个字符、替换一个字符

由于插入与删除的等价，若欲使A、B二串相同这三种操作实际上可以被转换为：

1. 替换A中的一个字符
2. 删除A中的一个字符
3. 删除B中的一个字符（对应在A中插入一个字符）

定义$dp[i][j]$表示下标从$[0]$~$[i-1]$的字符串与下标从$[0]$~$[j-1]$的字符串的最短编辑距离。根据上述的三种转换，可以得到状态转移方程：

$$dp[i][j] = \begin{cases} dp[i-1][j-1], & i = j \\ \min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1, & i \neq j \end{cases}$$

因此写出代码如下：

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;

class Solution {
public:
```

```cpp
    int minDistance(string word1, string word2)
    {
        // dp[i][j]表示下标从[0] - [i - 1]的字符串与下标从[0] - [j - 1]的字符串的最短编辑距离
        vector<vector<int> > dp(word1.size() + 1, vector<int>(word2.size() + 1, 0));

        // 初始化
        for (int i = 0; i ≤ word1.size(); i++)
            dp[i][0] = i;
        for (int j = 1; j ≤ word2.size(); j++)
            dp[0][j] = j;

        for (int i = 1; i ≤ word1.size(); i++) {
            for (int j = 1; j ≤ word2.size(); j++) {
                // 不需要修改
                if (word1[i - 1] == word2[j - 1])
                    dp[i][j] = dp[i - 1][j - 1];
                else
                    // 替换 删除 插入
                    dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]}) + 1;
            }
        }

        return dp[word1.size()][word2.size()];
    }
};

int main()
{
    string w1 = "ros", w2 = "horse";
    Solution sol;
    cout << sol.minDistance(w1, w2);
    return 0;
}
```

以下是Leetcode截图：