算法课程大作业-传统项目版

> 姓名：赵治宇
>
> 学号：2301210565

股票交易策略分析

在股票市场中，制定有效的交易策略是投资者追求最大利润的关键。股票价格波动较大，投资者需要在不同时间点做出买入和卖出决策，以获得最大的收益。

问题定义

输入：

1. 一系列股票的历史价格数据，以时间序列的形式给出。
2. 算法需要考虑的参数，如手续费、最大交易次数等。

输出：

1. 买卖股票的时间点，即何时买入和何时卖出。
2. 最大利润，即通过这一系列交易获得的最大收益

约束条件：

1. 交易次数限制：可能存在最大交易次数的限制，即在整个时间范围内，最多可以进行多少次交易。
2. 手续费：每次买入或卖出都可能涉及手续费，需要考虑在内。
3. 冷冻期：每次交易之间会有冷冻期限制，规避频繁交易。

算法设计

贪心

贪心算法基于局部最优选择，即在每个阶段选择当前最优的解决方案，希望最终能够得到全局最优解。在股票交易中，具体的贪心策略为在每次价格上涨时买入，价格下跌时卖出。

优点：

简单易行，容易实现。算法复杂度较低，只需遍历一遍数据。空间复杂度低。

缺点：

不是最优解，稳定有收益（不亏损），但不一定是最大收益。当不限制交易次数时也可得到最大收益。

代码实现：

```
#include <iostream>
#include <vector>

using namespace std;
```

```cpp
struct StockTrade {
    int buyDay;
    int sellDay;
};

// 贪心算法实现
vector<StockTrade> greedyStockTrade(const vector<int>& prices) {
    vector<StockTrade> trades;

    int n = prices.size();
    if (n < 2) {  // 边界条件处理，避免非法输入
        cout << "非法交易数据" << endl;
        return trades;
    }

    for (int i = 0; i < n - 1; ++i) {
        if (prices[i + 1] > prices[i]) {
            StockTrade trade;
            trade.buyDay = i;
            while (i < n - 1 && prices[i + 1] > prices[i]) {
                ++i; // 找到价格峰值
            }
            trade.sellDay = i;
            trades.push_back(trade);
        }
    }

    return trades;
}

int main() {
    // 示例股票价格数据
    // 用户输入股票价格数据
    vector<int> prices;
    int price;
    cout << "输入股票价格序列（输入非字符值结束）：";
    while (cin >> price) {
        prices.push_back(price);
    }

    // 贪心算法
    vector<StockTrade> trades = greedyStockTrade(prices);

    // 输出交易结果
    if (trades.empty()) {
        cout << "非法交易" << endl;
    } else {
        cout << "交易策略" << endl;
        int totalProfit = 0;
```

```
        for (const auto& trade : trades) {
            int profit = prices[trade.sellDay] - prices[trade.buyDay];
            totalProfit += profit;
            cout << "购买日: " << trade.buyDay << ", 卖出日: " << trade.sellDay << ", 收
益: " << profit << endl;
        }
        cout << "总收益: " << totalProfit << endl;
    }

    return 0;
}
```

运行结果:

```
(base) zhaozhiyu@bogon algorithm_class % ./greedy
输入股票价格序列（输入非字符值结束）: 100 120 130 70 60 80 e
交易策略
购买日: 0, 卖出日: 2, 收益: 30
购买日: 4, 卖出日: 5, 收益: 20
总收益: 50
```

该测试用例说明贪心算法具有**正确性**，但是倘若增加交易次数限制呢?

```
// ...其他代码
// 最大交易次数限制
int maxTrades = 2;

vector<StockTrade> greedyStockTrade(const vector<int>& prices, int maxTrades) {
    vector<StockTrade> trades;

    int n = prices.size();
    if (n < 2) {  // 边界条件处理，避免非法输入
        cout << "非法交易数据" << endl;
        return trades;
    }

    int tradesCount = 0;
    for (int i = 0; i < n - 1 && tradesCount < maxTrades; ++i) {
        if (prices[i + 1] > prices[i]) {
            StockTrade trade;
            trade.buyDay = i;
            while (i < n - 1 && prices[i + 1] > prices[i]) {
                ++i; // 找到价格峰值
            }
            trade.sellDay = i;
            trades.push_back(trade);
            ++tradesCount;
        }
    }
```

```
    return trades;
}
```

运行结果：

可以看到当设置交易限制的时候，结果就不正确了，对于该测试用例，正确的最大收益应该为30，在第4日买入，第5日卖出。

动态规划

动态规划是一种通过拆分问题为子问题并保存其最优解的方法。在股票交易中，可以通过定义状态（持有或不持有股票）和状态转移方程来设计动态规划算法。动态规划通常能够找到全局最优解，但它可能需要更多的计算和空间复杂度。

优点：

可得到最优解。算法复杂度相对较高，但依旧是线性时间o(N)。

缺点：

需要额外的辅助空间。

算法设计：

对每一天维护两个状态：

1. **持有股票**：该状态下的最大利润可以是前一天就持有股票，或者今天买入股票（前一天没有股票）。
2. **不持有股票**：该状态下的最大利润可以是前一天就没有股票，或者今天卖出股票（前一天持有股票）。

`dp[i][0]`表示第i天不持有股票的最大利润，`dp[i][1]`表示第i天持有股票的最大利润.

具体定义：

```
// 初始条件：第0天不持有股票的最大利润为0，持有股票的最大利润为-prices[0]
dp[0][0] = 0;
dp[0][1] = -prices[0];

// 状态转移方程
dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
dp[i][1] = max(dp[i - 1][1], -prices[i]);
```

代码实现：

```
// ···代码
// 动态规划算法实现
```

```
vector<StockTrade> DPStockTrade(const vector<int>& prices) {
    vector<StockTrade> trades;

    int n = prices.size();
    if (n < 2) {  // 边界条件处理，避免非法输入
        cout << "非法交易数据" << endl;
        return trades;
    }

    // 创建动态规划数组，dp[i][0]表示第i天不持有股票的最大利润，dp[i][1]表示第i天持有股票的最大利润
    vector<vector<int>> dp(n, vector<int>(2, 0));

    // 初始条件：第0天不持有股票的最大利润为0，持有股票的最大利润为-prices[0]
    dp[0][0] = 0;
    dp[0][1] = -prices[0];

    for (int i = 1; i < n; ++i) {
        // 状态转移方程
        dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
        dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] - prices[i]);

        // 记录交易信息
        if (dp[i][0] > dp[i - 1][0] && dp[i][1] == dp[i - 1][1]) {
            StockTrade trade;
            trade.buyDay = i - 1;
            trade.sellDay = i;
            trades.push_back(trade);
        }
    }

    // 最终收益为在最后一天不持有股票的状态
    // return dp[n - 1][0];
    return trades;
}
```

运行结果：

```
(base) zhaozhiyu@bogon algorithm_class % ./dp
输入股票价格序列（输入非字符值结束）：100 120 110 130 80 90 110 75 e
总收益：70
```

该测试用例说明动态规划算法具有**正确性**。那么面对增加限制条件的情况呢？

限制交易次数

对于限制交易次数的情况，动态规划的状态可以扩展为三维数组，其中除了天数 `i` 外，还包括交易次数 `k`。我们可以使用 `dp[i][k][0]` 表示第 `i` 天结束时不持有股票的最大利润，`dp[i][k][1]` 表示第 `i` 天结束时持有股票的最大利润。修改后的状态转移方程如下：

```
// 状态转移方程
dp[i][k][0] = max(dp[i - 1][k][0], dp[i - 1][k][1] + prices[i]);
dp[i][k][1] = max(dp[i - 1][k][1], dp[i - 1][k - 1][0] - prices[i]);
```

入参增加最大交易数入参。函数签名更改如下：

```
vector<StockTrade> DPStockTrade(const vector<int>& prices, int maxTrades);
```

运行效果：

```
(base) zhaozhiyu@bogon algorithm_class % ./dp
输入股票价格序列（输入非字符值结束）：100 120 110 130 80 90 110 75 e
交易数限制为1的总收益：30
交易数限制为2的总收益：60
交易数限制为3的总收益：70
```

手续费

当考虑手续费时，我们需要在买入和卖出时扣除手续费。

更改函数签名：

```
vector<StockTrade> DPStockTrade(const vector<int>& prices, int fee)
```

更改初始条件：

```
// 初始条件：第0天不持有股票的最大利润为0，持有股票的最大利润为-prices[0]
dp[0][0] = 0;
dp[0][1] = -prices[0] - fee; // 考虑手续费
```

更改状态转移方程：

```
// 状态转移方程
dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] - prices[i] - fee); // 考虑手续费
```

运行效果：

```
(base) zhaozhiyu@bogon algorithm_class % ./dp
输入股票价格序列（输入非字符值结束）：100 120 110 130 80 90 110 75 e
总收益：20
```

冷冻期

当考虑冷冻期时，我们需要在状态转移方程中考虑是否处于冷冻期。冷冻期的限制意味着如果在第 `i` 天卖出股票，那么在第 `i+1` 天就不能再买入股票。

更改状态转移方程：

```
// 计算处于冷冻期的前一天的状态
int prevDay = (i >= 2) ? dp[i - 2][0] : 0;

// 状态转移方程
dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
dp[i][1] = max(dp[i - 1][1], prevDay - prices[i]);
```

运行结果：

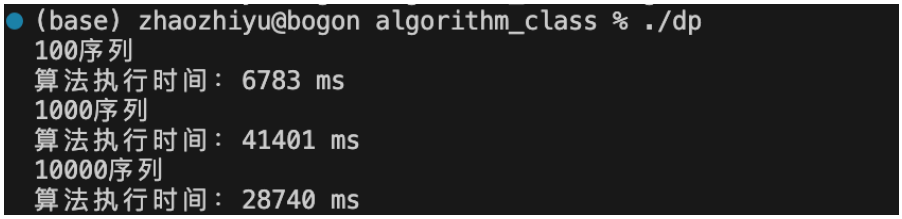
```
(base) zhaozhiyu@bogon algorithm_class % ./dp
输入股票价格序列（输入非字符值结束）：100 120 110 130 80 90 110 75 e
总收益：50
```

性能评估

应用`<chrono>`提供的计时器和`<random>`提供的随机数生成方法，生成股票序列，测试动态规划算法耗时：

```
// 测试算法性能
void testAlgorithmPerformance(const vector<int>& data) {
    auto start_time = chrono::high_resolution_clock::now();

    // 调用算法函数
    int result = DPStockTrade(data, 10);

    auto end_time = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end_time - start_time);

    // 输出执行时间
    cout << "算法执行时间：" << duration.count() << " ms" << endl;
}

// 生成随机股价数据集
vector<int> generateRandomStockPrices(int size, int minPrice, int maxPrice) {
    vector<int> prices;

    // 设置随机数生成器
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> priceDist(minPrice, maxPrice);

    // 生成随机股价数据集
    for (int i = 0; i < size; ++i) {
        prices.push_back(priceDist(gen));
    }

    return prices;
}
```

分别生成100、1000、10000长度序列，测试结果如下：



结果分析可知，由于限定交易次数，运行时间会出现数据量增大而减小的现象；此外分析可知该算法的时间复杂度为o(N)，只需遍历一遍输入序列，效率较高，可用于大量数据计算。

另外空间方面，需要额外开辟辅助存储空间，复杂度为o(N·K)（K为限制交易次数）。

性能优化

为优化性能，提高算法运行效率，可以考虑在工程实现层面加入缓存、并行计算等机制（考虑到动态规划迭代需要依赖前置结果，此处并行限于输入输出处理）等。此外还可以考虑使用机器学习等人工智能算法模型，但对应的开销也会增大。

挑战和解决方案

C++编译环境的搭建

问题描述：

由于课业要求使用C++解决问题，故需要配置本地开发环境，以实现本地开发运行实验程序。本人使用的代码编辑器为VSCode，编译CPP文件需要配置编译器信息。但由于电脑自身链接问题无法绑定编译器。

解决方案：

依旧VSCode编辑代码，但编译运行直接命令行操作。

股票预测场景与解决方案调研

问题描述：

由于网络上股票信息纷繁复杂，需要对股票序列进行建模。通过Google等搜索引擎协助，了解了股票预测的一般形式和应用场景。但依然有不懂、不明白的地方。

解决方案：

利用学院交叉学科优势，咨询金科方向同学相关问题，并基于自身技术背景为股票预测建模，验证方案的可行性。

总结与改进方向

项目本身是从算法层面，对股票序列单一建模的方式进行的分析比较和实验，但实际股票预测场景更为复杂多变，需要建立高效、可用的系统来解决实际的股票预测与策略生成。故改进方向为，可以建立股票交易系统，应用该项目研究的股票交易算法，实现真正的股票交易。

项目代码和评估结果演示视频已上传至：https://github.com/Breeze-P/Algorithm-Project

`LeetCode`主页：https://leetcode.cn/u/zhao-shao-feng-e/

最后，感谢这一学期张老师的教学指导，本人在算法层面获得了足够的训练、提升和进步，附图为本学期的`LeetCode`刷题情况：