重庆师范大学

实验报告

实验课程名称	算法设计与分析
实验序号	6
实验内容	矩阵连乘（动态规划）
班级	23 计科 6 班
姓名	周子依
学号	2023051603162

2024 年 4 月 19 日

实验目的与要求

学习目标：

1. 掌握动态规划的思想。
2. 理解矩阵连乘寻找最小乘法次数的思路。
3. 掌握如何将动态规划的思想用到矩阵连乘中。
4. 对以下的实际例子求解连乘的顺序使得乘法次数最少。

p0	p1	p2	p3	p4	p5	p6
30	35	15	5	10	20	25

实验内容

1、理解问题

假设有一个 p*q 规模的矩阵 A，一个 q*r 规模的矩阵 B，并且我们现在再加上一个规模为 r*s 的矩阵 C，那么这三个矩阵的乘积 ABC 有两种计算顺序：

- （AB）C
- A（BC）

$$mult[(AB)C] = pqr + prs$$

$$mult[A(BC)] = qrs + pqs$$

假设 p=5,q=4,r=6 并且 s=2，则：

$$mult[(AB)C] = 180,$$
$$mult[A(BC)] = 88.$$

由此可见，矩阵连乘的顺序不同，所需计算的乘法次数，会有很大不同！！！所以，我们需要寻找连乘顺序使得做乘法次数最少，以大大节省时间。

矩阵连乘可以被递归成如下形式：

$$A_1A_2A_3\cdots A_{s-1}A_s = A_1(A_2(A_3\cdots(A_{s-1}A_s)))$$

比如有四个矩阵，共有五种连乘顺序：

$$\begin{aligned} A_1A_2A_3A_4 &= (A_1A_2)(A_3A_4) \\ &= A_1(A_2(A_3A_4)) = A_1((A_2A_3)A_4) \\ &= ((A_1A_2)A_3)(A_4) = (A_1(A_2A_3))(A_4) \end{aligned}$$

2、使用动态规划算法解决矩阵连乘

动态规划一般适用于最优子结构问题，在矩阵连乘中找到一个乘积所需计算次数最小的计算顺序（打括号方式），就是我们的最优子结构。

无论我们以一个什么样的乘法顺序来计算，最后一步都是一样的，即把最后生成的两个中间矩阵 Ai…k 和 Ak+1…j 相乘，其中 k 可以是在合法范围内的任意一个值：

$$A_{i..j} = (A_i\cdots A_k)(A_{k+1}\cdots A_j) = A_{i..k}A_{k+1..j}$$

决定最优乘积计算顺序的问题就可以分解为下面两个子问题：

- 我们应该在乘积序列中的哪一个地方使用大括号把该序列分成两个子序列呢？（即 K 的值到底应该设为多少？可以暴力列举出所有合法的 K 值）
- 对于分割出来的两个子乘积序列 Ai…k 和 Ak+1…j，我们又该如何对他们再进行一步划分呢？

我们定义 m[i,j]的值为计算 Ai…j 所需要要的最少的计算次数，因此，这个最少计算次数的问题可以用下面的递归式来描述：

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i\le k<j}\left(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\right) & i < j \end{cases}$$

对于 Ai…j 的任何一个计算顺序，我们都可以把它拆分成下面两个计算步骤：

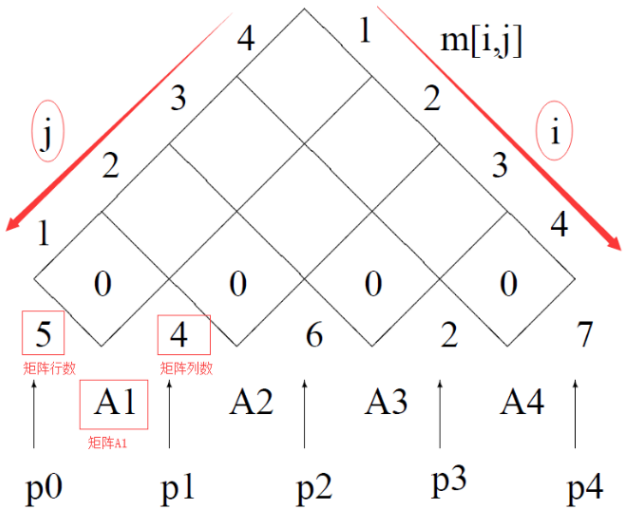
$$A_{i..j} = A_{i..k}A_{k+1..j}$$

则 Ai..j 的最少乘积次数的计算公式为:

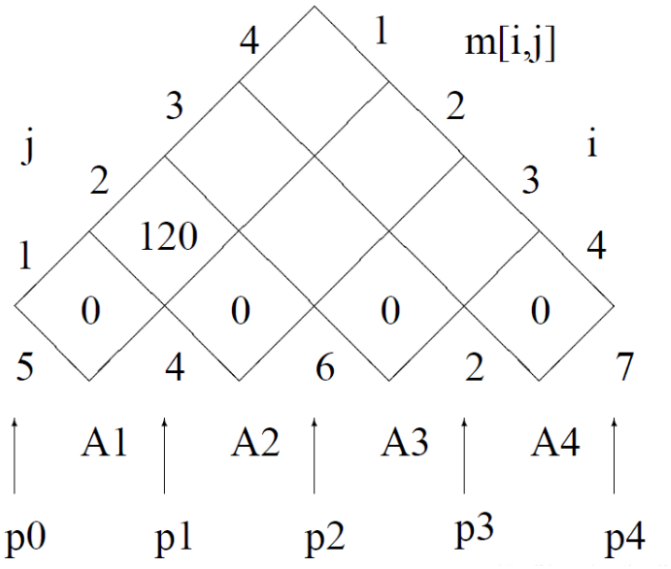
$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$$

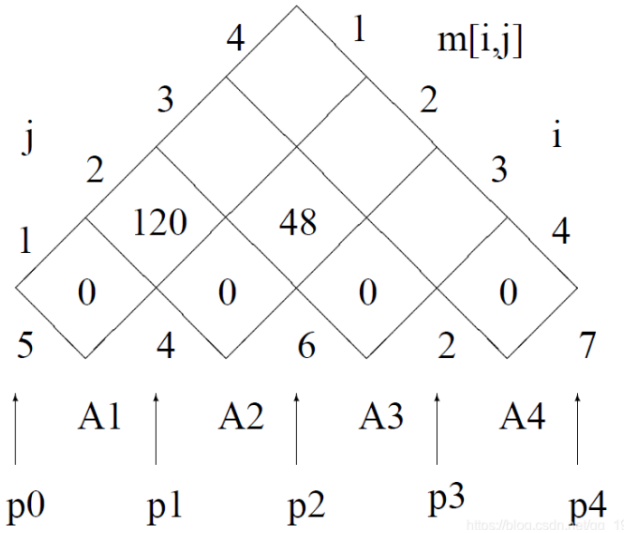
3、举例分析：

1) 使用动态规划算法解决，首先初始化数组 m[i,j]，为了便于较为直观的观察，数组 m[i,j]不按照正常“表格”的方式展示，而是按照下面的“画法”来展示：

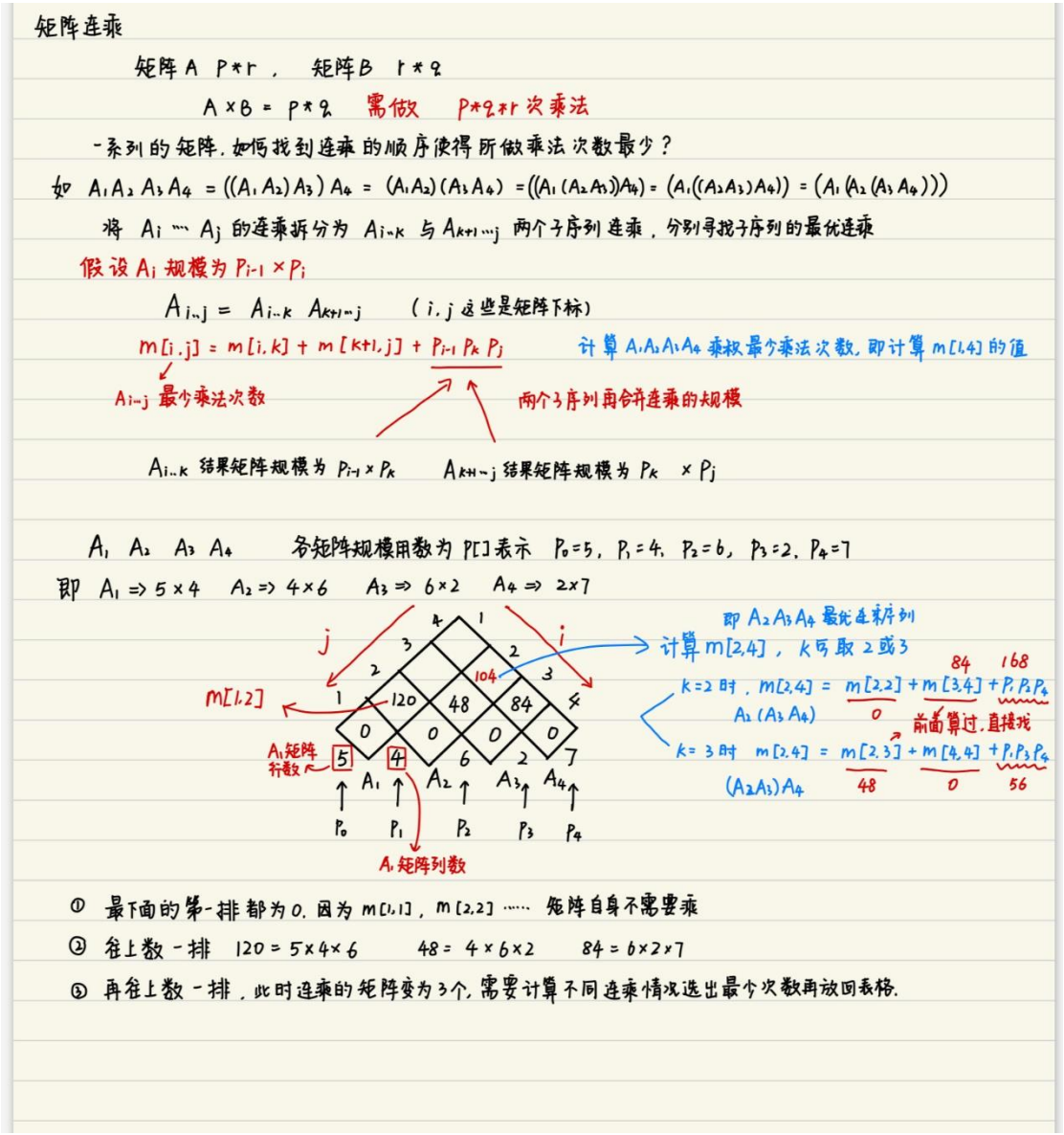


自下而上依次计算计算乘法次数


4
1
m[i,j]
3
2
j
i
2
3
120
1
4
0
0
0
0
5
4
6
2
7
A1
A2
A3
A4
p0
p1
p2
p3
p4


4
1
m[i,j]
3
2
j
i
2
3
120
48
1
4
0
0
0
0
5
4
6
2
7
A1
A2
A3
A4
p0
p1
p2
p3
p4

笔记整理如下：



① 最下面的第一排都为0，因为 m[1,1]，m[2,2] …… 矩阵自身不需要乘

② 往上数一排 120 = 5×4×6 48 = 4×6×2 84 = 6×2×7

③ 再往上数一排，此时连乘的矩阵变为3个，需要计算不同连乘情况选出最少次数再放回表格.

4、结合代码分析：

```
void MatrixChain(int *p, int n, int **m, int **s) {
    // 初始化对角线元素，单个矩阵相乘代价为0
    for (int i = 1; i <= n; i++)
        m[i][i] = 0;
    // r表示矩阵链的长度，从2开始（因为单个矩阵已初始化）
    for (int r = 2; r <= n; r++) {
        // i是矩阵链起始位置
        for (int i = 1; i <= n - r + 1; i++) {
            // j是矩阵链结束位置
            int j = i + r - 1;
            // 先假设一种划分方式（从i和i+1处划分），计算代价并记录分割点
            m[i][j] = m[i + 1][j] + p[i - 1] * p[i] * p[j];
            s[i][j] = i;
            // 尝试其他划分方式，k为分割点
            for (int k = i + 1; k < j; k++) {
                // 计算当前划分方式的代价
                int t = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                // 如果当前划分方式代价更小，则更新最优代价和分割点
                if (t < m[i][j]) {
                    m[i][j] = t;
                    s[i][j] = k;
                }
            }
        }
    }
```

计算最下面一排
两个矩阵连乘结果

依次对上面排的矩阵（从三个矩阵开始）
计算最优连乘序列

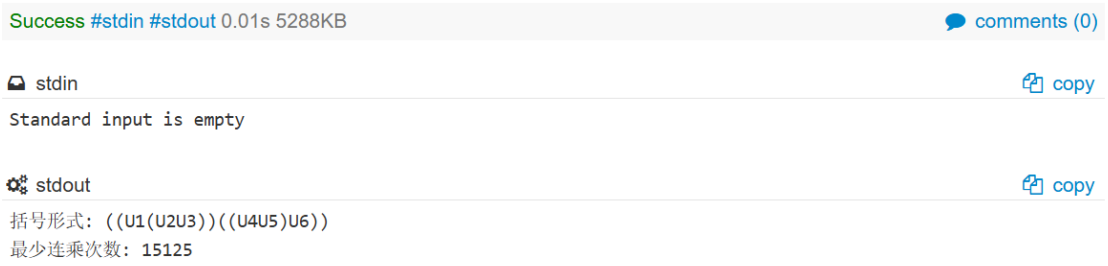
5、复杂度分析：

可以近似的认为矩阵连乘的复杂度就是矩阵连乘所需要的总次数，即O(p*q*r)。

代码附在最后

运行结果

为了便于直接观察，最终输出为括号形式。



总结与体会

1、 实验总结

本次实验围绕矩阵连乘问题展开，运用动态规划算法成功求解出了使得乘法次数最少的连乘顺序。通过实验，深入理解了动态规划思想，即把复杂问题分解为相互关联的子问题，通过求解子问题的最优解来得到原问题的最优解。在矩阵连乘场景中，明确了寻找最小乘法次数的核心在于合理划分连乘序列，确定分割点 k 。通过对不同规模矩阵连乘的分析与计算，熟练掌握了动态规划算法在该问题上的应用流程，包括状态定义（m[i, j]）表示计算（A_{i…j}）所需最少计算次数）、状态转移方程的推导与运用。

2、 踩坑与解决

在初始化数组(m[i, j])和实现状态转移计算时遇到了困难。初始化时，对边界条件的处理不够准确，导致后续计算出现偏差。比如在确定(m[i, i])（单个矩阵时乘法次数应为 0 ）的值时出现失误。通过反复查阅教材和参考资料，重新梳理了初始化逻辑，明确了每个边界值的意义，最终正确完成了初始化。在状态转移计算过程中，由于对嵌套循环和状态转移方程的理解不够透彻，计算顺序出现混乱。经过在纸上手动模拟计算过程，逐步理清了自下而上、从子问题到父问题的计算顺序，成功解决了该问题。

3、 学习收获

通过本次实验，在知识层面，不仅深入掌握了动态规划算法原理及其在矩阵连乘问题中的应用，还对矩阵乘法运算有了更深刻的理解，明白了不同连乘顺序对计算复杂度的巨大影响。在技能层面，提高了算法设计与实现能力，包

括代码编写、调试以及对复杂算法逻辑的梳理能力。同时，培养了问题分析和解决能力，面对算法实现过程中的问题，学会了从不同角度思考，通过查阅资料、手动模拟等方法寻找解决方案。在未来的学习和实践中，会更加注重对算法思想的理解与运用，不断提升自己解决实际问题的能力。

完整代码（cpp）：

```
#include <iostream>
#include <sstream>
#include <memory>
using namespace std;

// 计算矩阵链乘法最优代价和记录分割点
// p 是一个数组，存储矩阵链中每个矩阵的行数和列数（第一个矩阵的行数、后续矩阵的列数）
// n 是矩阵链中矩阵的数量
// m 是一个二维数组，用于存储矩阵链乘法的最优代价
// s 是一个二维数组，用于记录在最优代价下矩阵链的分割点
void MatrixChain(int *p, int n, int **m, int **s) {
    // 初始化对角线元素，单个矩阵相乘代价为 0
    for (int i = 1; i <= n; i++) {
        m[i][i] = 0;
    }
    // r 表示矩阵链的长度，从 2 开始（因为单个矩阵已初始化）
    for (int r = 2; r <= n; r++) {
        // i 是矩阵链起始位置
        for (int i = 1; i <= n - r + 1; i++) {
            // j 是矩阵链结束位置
            int j = i + r - 1;
```

```
22              // 先假设一种划分方式（从 i 和 i+1 处划分），计算代价并记录分割点
23              m[i][j] = m[i + 1][j] + p[i - 1] * p[i] * p[j];
24              s[i][j] = i;
25              // 尝试其他划分方式，k 为分割点
26              for (int k = i + 1; k < j; k++) {
27                  // 计算当前划分方式的代价
28                  int t = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
29                  // 如果当前划分方式代价更小，则更新最优代价和分割点
30                  if (t < m[i][j]) {
31                      m[i][j] = t;
32                      s[i][j] = k;
33                  }
34              }
35          }
36      }
37  }
38
39  // 追溯矩阵链乘法的加括号顺序并生成括号形式字符串
40  // i 是矩阵链起始位置
41  // j 是矩阵链结束位置
42  // s 是记录分割点的二维数组
43  string Traceback(int i, int j, int **s) {
44      // 如果起始和结束位置相同，即单个矩阵，直接返回矩阵名称
45      if (i == j) {
46          ostringstream oss;
47          oss << "U" << i;
```

```
48          return oss.str();
49      }
50      // 递归获取左子链的括号形式字符串
51      string left = Traceback(i, s[i][j], s);
52      // 递归获取右子链的括号形式字符串
53      string right = Traceback(s[i][j] + 1, j, s);
54      ostringstream oss;
55      // 将左右子链的括号形式字符串组合起来，形成当前矩阵链的括号形式字符串
56      oss << "(" << left << right << ")";
57      return oss.str();
58  }
59
60  // 输出正确格式的乘法顺序
61  // s 是记录分割点的二维数组
62  // i 是矩阵链起始位置
63  // j 是矩阵链结束位置
64  void PrintOptimalParens(int **s, int i, int j) {
65      // 如果起始和结束位置相同，即单个矩阵，直接输出矩阵名称
66      if (i == j) {
67          cout << "U" << i;
68          return;
69      }
70      // 输出左括号，表示开始计算当前矩阵链
71      cout << "(";
72      // 递归输出左子链的乘法顺序
73      PrintOptimalParens(s, i, s[i][j]);
74      // 递归输出右子链的乘法顺序
75      PrintOptimalParens(s, s[i][j] + 1, j);
```

```
76      // 输出右括号，表示当前矩阵链计算结束
77      cout << ")";
78  }
79
80  int main() {
81      // 定义数组 p，存储矩阵链中每个矩阵的行数和列数
82      int p[] = {30, 35, 15, 5, 10, 20, 25};
83      // 矩阵链中矩阵的数量
84      int n = sizeof(p) / sizeof(p[0]) - 1;
85
86      // 使用智能指针管理动态分配的二维数组 m，用于存储矩阵链乘法的最优代价
87      auto m = make_unique<int*[]>(n + 1);
88      // 使用智能指针管理动态分配的二维数组 s，用于记录在最优代价下矩阵链的分割点
89      auto s = make_unique<int*[]>(n + 1);
90
91      // 为二维数组 m 的每一行分配内存
92      for (int i = 0; i <= n; i++) {
93          m[i] = new int[n + 1];
94      }
95      // 为二维数组 s 的每一行分配内存
96      for (int i = 0; i <= n; i++) {
97          s[i] = new int[n + 1];
98      }
99
100     // 调用 MatrixChain 函数，计算矩阵链乘法的最优代价和分割点
101     MatrixChain(p, n, m.get(), s.get());
102
```

```
    // 调用 Traceback 函数，获取括号形式结果并输出
    string parenForm = Traceback(1, n, s.get());
    cout << "括号形式: " << parenForm << endl;

    // 输出最少连乘次数
    cout << "最少连乘次数: " << m[1][n] << endl;

    // 释放二维数组 m 动态分配的内存
    for (int i = 0; i <= n; i++) {
        delete[] m[i];
    }
    // 释放二维数组 s 动态分配的内存
    for (int i = 0; i <= n; i++) {
        delete[] s[i];
    }

    return 0;
}
```