

**HUBEIUNIVERSITY OF AUTOMOTIVE TECHNOLOGY**



**算法设计与分析**

**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验项目 | 实验三 | 实验类别 | | | 验证性 |
| 学生姓名 | 章崇文 | 学生学号 | | | 202202296 |
| 学生班级 | 计算机222 | | | | |
| 完成日期 | 2024-05-12 | | | | |
| 实验成绩 |  | | 评阅日期 |  | |
| 评阅教师 |  | | | | |

**实验三：动态规划法**

**【实验目的】**

应用动态规划算法思想求解矩阵连乘的顺序问题。

**【实验性质】**

验证性实验。

**【实验要求】**

应用动态规划算法的最优子结构性质和子问题重叠性质求解此问题。分析动态规划算法的基本思想，应用动态规划策略写出算法及相应的程序，求解此题。要读懂读透A[i,j]，A[1,n]=A[1,k] ×A[k+1,n]，m[i][j]，s[i][j]各式所表达的含义并正确加以应用。m[i][j]的递归定义：

0 （ i=j ）

m[i][j]=

min{m[i][k]＋ m[k+1][j]+ni-1nknj  （ i<j）

要求完成：⑴算法描述⑵写出程序代码⑶完成调试⑷进行过程与结果分析。

**【实验内容】**

对于下列所描述的问题，给出相应的算法描述，并完成程序实现与时间复杂度的分析。该问题描述为：一般地，考虑矩阵A1，A2，… ，An的连乘积，它们的维数分别为d0,d1,…,dn,即Ai的维数为di-1×di (1≤i≤n)。确定这n个矩阵的乘积结合次序，使所需的总乘法次数最少。对应于乘法次数最少的乘积结合次序为这n个矩阵的最优连乘积次序。按给定的一组测试数据对根据算法设计的程序进行调试：6个矩阵连乘积A=A1×A2×A3×A4×A5×A6，各矩阵的维数分别为：A1：10×20，A2：20×25，A3：25×15，A4：15×5，A5：5×10，A6：10×25。完成测试。

**【算法思想及处理过程】**

动态规划法是一种通过把原问题分解为相对简单的子问题，并存储子问题的解以避免重复计算，从而求得原问题解的方法。其基本思想主要包括以下几个步骤：

**1.定义状态：**明确问题的“状态”，即问题在不同阶段的不同表现形式。每个状态都应能通过某些决策或操作从其他状态达到。

**2.确定状态转移方程：**基于当前状态和可能的选择，定义如何从一个状态转移到另一个状态的规则或公式。这通常涉及到解决子问题并合并结果来得到更大问题的解。

**3.初始化边界条件：**确定最简单或最基本的状态值，通常是问题规模极小的情况。

**4.自底向上/自顶向下计算：**根据状态转移方程，以一种有序的方式（通常是从最小的问题规模开始逐步构建到原问题规模）计算所有需要的状态值，利用之前计算过的子问题的结果避免重复工作。

**应用到矩阵链乘法问题中的处理算法：**

定义状态

设状态 m[i][j] 表示计算矩阵 Ai...Aj（即从矩阵 Ai 到矩阵 Aj 的连续乘积）的最小乘法次数。

**状态转移方程**

对于计算 Ai...Aj 的最优解，考虑将乘积链在某个位置 k （i ≤ k < j）处分割为两部分：Ai...Ak 和 Ak+1...Aj。因此，最小乘法次数为：

*m*[*i*][*j*]=*i*≤*k*<*j*min​(*m*[*i*][*k*]+*m*[*k*+1][*j*]+*pi*−1​⋅*pk*​⋅*pj*​)

这里，p\_{i-1}, p\_k, 和 p\_j 分别是分割前后的矩阵维度，乘积表示额外的乘法次数。

初始化边界条件

边界条件为单个矩阵的乘法次数为 0，即当 i=j 时，m[i][i] = 0，因为单个矩阵不需要额外的乘法操作。

**计算顺序**

可以采用自底向上的方法，先解决小规模的子问题，然后利用这些子问题的解去解决更大规模的问题。具体而言，按子序列长度递增的顺序（长度为 2 开始直到整个序列），依次计算所有可能的 m[i][j] 值。

**记录选择**

为了还原最优解的括号化方式，还需要记录每个状态下的最优分割位置 s[i][j]，即在计算 m[i][j] 时选择的 k 值。

**数据结构**

使用二维数组 m[N][N] 存储每个子问题的最小乘法次数。

使用二维数组 s[N][N] 记录每个子问题的最优分割点，便于回溯构建最优解。

**流程概述**

初始化 m[i][i]=0 对所有 i。

对于长度为 2 到 n 的所有子序列，按长度递增顺序： a. 计算所有可能的 m[i][j] 和对应的 s[i][j]。

使用 s[N][N] 回溯得到最优括号化的顺序。

**【程序代码】**

#include <iostream>

using namespace std;

const int N = 100;

int p[N] = {10, 20, 25, 15, 5, 10, 25}; // 数据的维数

int m[N][N]; // 最优解

int s[N][N]; // 断点位置

void MatrixChain(int n) {

int r, i, j, k;

for (i = 0; i <= n; i++) // 初始化对角线

m[i][i] = 0;

for (r = 2; r <= n; r++) { // r个矩阵连乘

for (i = 1; i <= n - r + 1; i++) {

j = i + r - 1;

m[i][j] = m[i][i] + m[i + 1][j] + p[i - 1] \* p[i] \* p[j];

s[i][j] = i;

for (k = i + 1; k < j; k++) { // 改换分隔位置，逐一测试

int t = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j];

if (t < m[i][j]) { // 如果变换后的位置更优，则替换原来的分隔方法

m[i][j] = t; // 记录最少计算次数

s[i][j] = k; // 记录断开位置

}

}

}

}

}

void Traceback(int i, int j) {

if (i == j) {

cout << "A" << i;

} else {

cout << "(";

Traceback(i, s[i][j]);

Traceback(s[i][j] + 1, j);

cout << ")";

}

}

int main() {

int n = 6; // 矩阵的个数

MatrixChain(n);

cout << "最小乘法次数为：" << m[1][n] << endl;

cout << "最优的括号化方式为：";

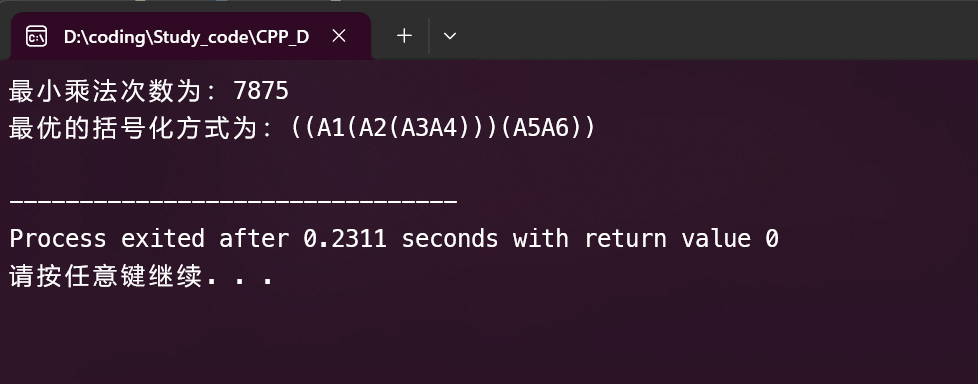
Traceback(1, n);

cout << endl;

return 0;

}

**【运行结果】**



**【算法分析】**

时间复杂度

主要的时间消耗在两个嵌套循环中，分别是：

外层循环 r 从 2 到 n，这个循环控制了我们考虑的矩阵链的长度。

内层循环有两个，第一个是 i 从 1 到 n - r + 1，第二个是 k 从 i + 1 到 j - 1。

第一层循环执行了 n - 1 次。

对于每一层 r，内层循环 i 执行了 n - r 次，而内层的 k 循环又执行了 j - i - 1 次。由于 j = i + r - 1，因此 k 的循环次数可简化为 r - 2。

因此，总的时间复杂度可以通过累加每层循环的迭代次数来估算，即：

*T*(*n*)=*r*=2∑*n*​(*n*−*r*)×(*r*−2)

简化这个表达式，我们得到一个关于 n 的多项式，最高次项是 n^3（因为展开后的主项是 n^3/6 - n^2/2 + n/3 中的 n^3/6），所以时间复杂度为 O(n^3)。

空间复杂度

二维数组 m[N][N] 和 s[N][N] 分别存储了不同长度链的最小乘法次数和断点位置，它们的大小都是 N x N，其中 N 是给定的最大矩阵数量（在这个例子中为 100）。因此，这部分的空间复杂度是 O(N^2)。

其他变量（如 r, i, j, k, t 等）占用的空间可以忽略不计，相对于 m 和 s 的空间来说是常数级别的。

综上所述，该算法的时间复杂度为 O(n^3)，空间复杂度为 O(N^2)，其中 n 是矩阵链中的矩阵个数，而 N 是预定义的最大矩阵数量。

【**实验总结**】