

**实 验 报 告**

（ 2021/2022 学年 第 2 学期）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 课程名称 | 算法分析与设计 | | | | | |
| 实验名称 | 动态规划法 | | | | | |
| 实验时间 | 2022 | 年 | 4 | 月 | 18 | 日 |
| 指导单位 | 南京邮电大学计算机学院、软件学院 | | | | | |
| 指导教师 | 陈春玲 | | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 学生姓名 | 张宸冉 | 学 号 | B20060527 |
| 学院(系) | 计算机学院 | 专 业 | 计算机科学与技术 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | | 动态规划法 | | | | 指导教师 | | | 陈春玲 |
| 实验类型 | | 验证 | 实验学时 | | 2 | 实验时间 | | | 2022.4.18 |
| 1. **实验任务和目的**   1、任务：  一、用动态规划法和备忘录方法实现求两序列的最长公共子序列问题。要求掌握动态规划法思想在实际中的应用，分析最长公共子序列的问题特征，选择算法策略并设计具体算法，编程实现两输入序列的比较，并输出它们的最长公共子序列。  二、用动态规划法和备忘录方法求解矩阵相乘问题，求得最优的计算次序以使得矩阵连乘总的数乘次数最少，并输出加括号的最优乘法算式。  2、目的：  加深对动态规划法的算法原理及实现过程的理解，学习用动态规划法解决实际应用中的最长公共子序列问题和矩阵连乘问题，体会动态规划法和备忘录方法的异同。 | | | | | | | | | |
| 1. **实验环境**   硬件：AMD Ryzen 7 4800H with Radeon Graphics  软件：Windows操作系统，Visual C++ 6.0。 | | | | | | | | | |
| **三、核心算法设计及分析**  1、核心算法设计   1. 最长公共子序列   **最优子结构性质：**  （1）若 xm=yn，则 zk=xm=yn，且 Zk-1是 Xm-1和 Yn-1的最长公共子序列；  （2）若 xm≠yn且 zk≠xm，则 Z是 Xm-1和 Y的最长公共子序列；  （3）若 xm≠yn且 zk≠yn，则 Z是 X和 Yn-1的最长公共子序列。  假设目前的子序列是a0a1a2a3a4a5…an, 在序列之后增加一个元素xm使得这个序列成为X和Y的长度为k+1的公共子序列，这和a0a1a2a3a4a5…an是X和Y的LCS是一个矛盾。  **动态规划递推公式：**    **重叠子问题性质：**  从递推关系式可以看出，最长公共自序列在求解过程中会重复使用之前已经得到的结果  **算法细节：**  用二维数组c[i][j]记录LCS的长度，其中第0行和第0列是0  当两个序列xi=yj，c[i][j]=c[i-1][j-1]+1;  当xi≠yj,c[i][j]=max{c[i][j-1],c[i-1][j]}  同时用二维数组s[i][j]记录LCS构造的路径，  当xi=yj，c[i][j]=c[i-1][j-1]+1，s[i][j]=1  当c[i][j]=c[i-1][j],s[i][j]=2  当c[i][j]=c[i][j-1],s[i][j]=3    **思考题：**   1. 通过备忘录方法，结合分治法的思想，自顶向下递归调用，但是和传统分治法不同，这里通过备忘录保存了所有的计算结果，利用了重叠子问题性质。   具体就是在代码中增加了：  **if (c[i][j]!=0) return c[i][j];**  直接返回结果，不通过之后的递归调用过程。   1. 常规版本的s矩阵主要是记录LCS构造的相关信号，但这些信息其实可以通过对c矩阵进行访问得到，没有必要记录s矩阵，可以通过这个优化节省空间复杂度。   **if (i==0||j==0) return;**  **if (a[i]==b[j])**  **{**  **CLCS(i-1,j-1);**  **cout<<a[i];**  **}**  **else**  **if (c[i-1][j]>=c[i][j-1]) CLCS(i-1,j);**  **else CLCS(i,j-1);**   1. 如果没有构造最后最长公共子序列的需求，只需要得出最长公共子序列的长度，除了不需要记录s矩阵以外，通过观察c矩阵的构造过程可以发现，每一行的构造只和上一行相关，所以只需要一个两行的c矩阵就可以求出最长公共子序列的长度。 2. 常规的输出最长公共子序列的函数，对于c[i-1][j]>=c[i][j-1]的情况都是默认向上一行前进，这样就只会输出一个最长公共子序列，然而**c[i-1][j]>c[i][j-1]**的时候应该是向上回溯，**c[i-1][j]=c[i][j-1]**的时候应该是向上和向左两条回溯道路都存在，所以在这里自然想到了递归调用可以运用系统栈记录断点，而常规的输出LCS的函数直接运用了递归的方法，会产生冲突，所以这里使用了一个集合记录所有的最长公共子序列，所有的最长公共子序列在回溯过程中先记录的是从后往前的，然后在回溯结束之后再进行反向操作即可。 3. 矩阵连乘   **最优子结构性质：**  用A[0:n-1]表示A0A1…An-1的矩阵连乘，若在Ak处断开，即  A[0:n-1]=(A[0:k]@A[k+1:n-1])，若A[0:n-1]的计算量= A[0:k]的计算量加上A[k+1:n-1]的计算量，再加上这两个矩阵相乘的计算量，那么这个次序（断开的位置）必然是最优的  **递推关系式：**  m[i][j]表示A[i:j]的最小计算量，其中pi\*pk+1\*pn+1为两个矩阵相乘的计算量。  m[i][j]=0，当i=j  m[i][j]=min{m[i][k]+m[k+1][j]+pi\*pk+1\*pn+1}，当i≠j  用s[i][j]记录A[i:j]最优计算次序断开的位置k  **重叠子问题性质：**  在计算上三角矩阵的时候，每一步计算m[i][j]的时候都要求，m[i][k],m[k+1][j]（其中）这两个参数都是计算好的，然后才能进行比较，得出最小值。  **算法思路：**  计算m[i][j]之前，所有的m[i][j]到m[i][j-1]和m[i+1][j]到m[j-1][j]都是已经计算完成的，所以按照m矩阵从主对角线开始（已经初始化计算完毕），逐级向矩阵的右上角计算，每次扩充步长，对从i到j的所有可能的断点进行计算，比较出最小的计算量，最后，矩阵的m[0][n-1]就是最终的解。  在计算m矩阵的同时，每次还要向s矩阵中，写入从i到j的矩阵连乘的断点k，方便回溯。  回溯时，如果是单个矩阵直接输出，如果当前序号i小于s矩阵中的数字，用（）将分割元素内的矩阵连乘包裹并且递归调用。大于同理。    **思考题：**  矩阵连乘的备忘录实现  使用了递归的方法，自顶向下计算矩阵连乘最优计算量的方案，其中，使用了备忘录的方案，保存了已经计算过的结果，不需要对已经计算出结果的重新进行递归。  if (m[i][j] > 0) { //如果问题已经得到解决 直接返回结果  return m[i][j];  }  2、核心算法分析  1）LCS时间复杂度分析  int LCSLength()函数的时间复杂度是O（m\*n），mn是两个序列的长度  void CLCS()函数的时间复杂度是O（m+n）  空间复杂度分析，常规情况下是O（m\*n）  2）矩阵连乘的时间复杂度分析  由于函数有3个for循环，时间复杂度是,  空间复杂度是O() | | | | | | | | | |
| **四、实验及结果分析**  1、实验  1）最长公共子序列  **测试用例：**  char a[8] = { '0','a','b','c','b','d','a','b' };  char b[7] = { '0','b','d','c','a','b','a' };  **实验结果：**  常规结果：  最长公共子序列的长度是：4  Bcba  不使用s数组的结果：  最长公共子序列的长度是：4  bcba  如果没有输出子序列需求的结果：  最长公共子序列的长度是：4  使用备忘录的结果：  最长公共子序列的长度是：4  bcba  输出所有子序列：  最长公共子序列的长度是：4  bcab  bcba  bdab  2）矩阵连乘  1.不用备忘录方法求矩阵连乘的数乘最少次数  最少数乘次数k=15125  ((A0 (A1 A2 ))((A3 A4 )A5 ))  m=  0 1 2 3 4 5  0 0 15750 7875 9375 11875 15125  1 0 2625 4375 7125 10500  2 0 750 2500 5375  3 0 1000 3500  4 0 5000  5 0  s=  0 1 2 3 4 5  0 0 0 0 2 2 2  1 1 1 2 2 2  2 2 2 2 2  3 3 3 4  4 4 4  5 5  2.用备忘录方法求矩阵连乘的数乘最少次数  最少数乘次数k=15125  ((A0 (A1 A2 ))((A3 A4 )A5 ))  m=  0 1 2 3 4 5  0 0 15750 7875 9375 11875 15125  1 0 2625 4375 7125 10500  2 0 750 2500 5375  3 0 1000 3500  4 0 5000  5 0  s=  0 1 2 3 4 5  0 0 0 0 2 2 2  1 1 1 2 2 2  2 2 2 2 2  3 3 3 4  4 4 4  5 5  2、实验结果分析  1）对程序进行单步调试可以看出，程序构造c矩阵的过程符合预期结果，通过s矩阵进行递归回溯也和预期效果一样。  输出所有的公共子序列，通过单步调试，可以看出操作系统通过系统栈记录了当序列中两个元素不相等，但是在c矩阵中上和左两个元素相等的情况下，需要进行两个方向的调用，先完成一个，然后回到递归调用点，沿着另外一个方向继续。最终输出的3个结果和手工计算的结果一致。  2）程序输出结果符合最优结果时应该的组合，单步调试符合算法思路 | | | | | | | | | |
| **五、实验小结（包括问题和解决办法、心得体会、建议和意见等）**  动态规划法，要求最优化问题具有最优子结构性质和重叠子问题性质，他不需要像贪心法一样不仅需要最优子结构性质，还需要贪心准则。同时他也不会和分治法一样重复计算很多已经计算过的结果，动态规划会利用重叠子问题性质，使用记录好的计算结果。  动态规划法是在最优化问题中相对合理的算法。 | | | | | | | | | | |
| **六、指导教师评语** | | | | | | | | | | |
| 成 绩 |  | | 批阅人 |  | | | 日 期 |  | | |

**附件：**

最长公共子序列：

#include<iostream>

#include<set>

using namespace std;

class LCS

{

public:

LCS(int nx, int ny, char\* x, char\* y);

~LCS();

int LCSLength();

int LCSLengthImprove();

int LCSLengthImproveWith2row();

int LCSLength\_Memo();

void CLCS();

void CLCSImprove();

void CLCSAllAnsImprove();

private:

int\*\* c; //用于动态规划记录路径长度的矩阵

int\*\* s; //用于记录路径方向的矩阵

int m; // 第一个子序列的长度

int n; //第二个子序列的长度

char\* a; //第一个子序列

char\* b; //第二个子序列

set<string> lcs; //存放所有子序列的输出

void CLCS(int i, int j);

void CLCSImprove(int i, int j);

void CLCSAllAnsImprove(int i ,int j ,string str);

int LCSLength\_Memo(int i,int j);

};

LCS::LCS(int nx, int ny, char\* x, char\* y) {

this->a = x;

this->b = y;

m = nx;

n = ny;

this->c = new int\* [m + 1];

this->s = new int\* [m + 1];

for (int i = 0; i <= m; i++) {

c[i] = new int[n + 1];

s[i] = new int[n + 1];

}

}

LCS::~LCS() {

for (int i = 0; i <= m; i++) {

delete c[i];

delete s[i];

}

delete c;

delete s;

}

int LCS::LCSLength() {

//初始化

for (int i = 0; i <= m; i++) {

c[i][0] = 0;

}

for (int i = 1; i <= n; i++) {

c[0][i] = 0;

}

for (int i = 1; i <= m; i++) {

for (int j = 1; j <= n; j++) {

if (a[i] == b[j]) { //对应元素相等 左上角元素+1

c[i][j] = c[i - 1][j - 1] + 1;

s[i][j] = 1;

}

else if (c[i - 1][j] >= c[i][j - 1]) { // 上面的元素比左边的元素大 就记为上边的元素

c[i][j] = c[i - 1][j];

s[i][j] = 2;

}

else {

c[i][j] = c[i][j - 1]; // 左边的元素比上边的大 就记为左边的元素

s[i][j] = 3;

}

}

}

return c[m][n]; // 右下角的元素就是最后的最优解值

}

//改进方法就是省略了S数组的空间

int LCS::LCSLengthImprove() {

//初始化

for (int i = 0; i <= m; i++) {

c[i][0] = 0;

}

for (int i = 1; i <= n; i++) {

c[0][i] = 0;

}

for (int i = 1; i <= m; i++) {

for (int j = 1; j <= n; j++) {

if (a[i] == b[j]) { //对应元素相等 左上角元素+1

c[i][j] = c[i - 1][j - 1] + 1;

}

else if (c[i - 1][j] >= c[i][j - 1]) { // 上面的元素比左边的元素大 就记为上边的元素

c[i][j] = c[i - 1][j];

}

else {

c[i][j] = c[i][j - 1]; // 左边的元素比上边的大 就记为左边的元素

}

}

}

return c[m][n]; // 右下角的元素就是最后的最优解值

}

//如果不需要输出子序列，就可以只用两行的矩阵记录结果

int LCS::LCSLengthImproveWith2row() {

//初始化

c[1][0] = 0; // 如果只是输出最后的最长公共子序列的长度，我们不需要记录每一个计算的结果 只需要两行的矩阵就可以计算最后的长度

for (int i = 1; i <= n; i++) {

c[0][i] = 0;

}

for (int i = 1; i <= m; i++) {

for (int j = 1; j <= n; j++) {

if (a[i] == b[j]) { //对应元素相等 左上角元素+1

c[1][j] = c[0][j - 1] + 1;

}

else if (c[0][j] >= c[1][j - 1]) { // 上面的元素比左边的元素大 就记为上边的元素

c[1][j] = c[0][j];

}

else {

c[1][j] = c[1][j - 1]; // 左边的元素比上边的大 就记为左边的元素

}

}

for (int j = 0; j <= n; j++) {

c[0][j] = c[1][j];

}

}

return c[1][n]; // 右下角的元素就是最后的最优解值

}

int LCS::LCSLength\_Memo() { // 备忘录递归算法 非递归接口

//初始化

for (int i = 0; i <= m; i++) {

for (int j = 0; j <= n; j++) {

c[i][j] = 0;

}

}

int ans=LCSLength\_Memo(m, n);

return ans;

}

int LCS::LCSLength\_Memo(int i, int j) {

if (i == 0 || j == 0) {

return 0;

}

if (c[i][j] != 0) {

return c[i][j]; //备忘录 如果有记录直接返回结果 不需要递归计算

}

else {

if (a[i] == b[j]) {

c[i][j] += LCSLength\_Memo(i - 1, j - 1) + 1;

s[i][j] = 1;

}

else {

if (LCSLength\_Memo(i - 1, j) >= LCSLength\_Memo(i, j - 1)) {

c[i][j] = LCSLength\_Memo(i - 1, j);

s[i][j] = 2;

}

else {

c[i][j] = LCSLength\_Memo(i, j - 1);

s[i][j] = 3;

}

}

}

return c[i][j];

}

void LCS::CLCS() {

CLCS(m, n);

}

//常规根据s数组回溯得到子序列

void LCS::CLCS(int i, int j) {

if (i == 0 || j == 0) {

return;

}

if (s[i][j] == 1) {

CLCS(i - 1, j - 1);

cout << a[i];

}

else if (s[i][j] == 2) {

CLCS(i - 1, j);

}

else {

CLCS(i, j - 1);

}

}

void LCS::CLCSImprove() {

CLCSImprove(m, n);

}

//通过c矩阵的信息省略s矩阵的空间

void LCS::CLCSImprove(int i, int j) {

if (i == 0 || j == 0) {

return;

}

if (a[i] == b[j]) {

CLCSImprove(i - 1, j - 1);

cout << a[i];

}

else if (c[i - 1][j] >= c[i][j - 1]) {

CLCSImprove(i - 1, j);

}

else {

CLCSImprove(i, j - 1);

}

}

//如果想要得出所有的子序列 需要对算法进行进一步的改进

void LCS::CLCSAllAnsImprove() {

string str; //记录每一个子序列

CLCSAllAnsImprove(m, n, str);

set<string>::iterator it = lcs.begin();

for (; it != lcs.end(); it++) //将集合中的元素输出

cout << \*it << endl;

}

void LCS::CLCSAllAnsImprove(int i, int j, string str) { //输出所有子序列 没有使用之前的完全递归 而是使用一个字符串保存

while (i>0&&j>0)

{

if (a[i] == b[j]) {

str.push\_back(a[i]); //两个字符串的字符相等 将该字符送入str保存

i--;

j--;

}

else

{

if (c[i - 1][j] > c[i][j - 1]) {

i--;

}

else if (c[i - 1][j] < c[i][j - 1]) {

j--;

}

else

{

CLCSAllAnsImprove(i - 1, j, str); //因为当两个字符不相等但是c矩阵的上下两个元素相等 会有两种可能的路径 所以很自然的想到了递归

CLCSAllAnsImprove(i, j - 1, str);

return;

}

}

}

reverse(str.begin(), str.end()); //相比于之前的版本，这里的字符串是从后往前的顺序，所以需要反向一下才是最终的结果

lcs.insert(str); //将结果存放到集合中保存

}

int main() {

char a[8] = { '0','a','b','c','b','d','a','b' };

char b[7] = { '0','b','d','c','a','b','a' };

LCS demo1(7, 6, a, b);

cout << "常规结果：" << endl;

cout << "最长公共子序列的长度是：" << demo1.LCSLength() << endl;

demo1.CLCS();

cout << endl<<endl;

LCS demo2(7, 6, a, b);

cout << "不使用s数组的结果：" << endl;

cout << "最长公共子序列的长度是：" << demo2.LCSLengthImprove() << endl;

demo2.CLCSImprove();

cout << endl<<endl;

LCS demo3(7, 6, a, b);

cout << "如果没有输出子序列需求的结果：" << endl;

cout << "最长公共子序列的长度是：" << demo3.LCSLengthImproveWith2row() << endl << endl;

LCS demo4(7, 6, a, b);

cout << "使用备忘录的结果：" << endl;

cout << "最长公共子序列的长度是：" << demo4.LCSLength\_Memo() << endl;

demo4.CLCS();

cout << endl << endl;

LCS demo5(7, 6, a, b);

cout << "输出所有子序列：" << endl;

cout << "最长公共子序列的长度是：" << demo5.LCSLengthImprove() << endl;

demo5.CLCSAllAnsImprove();

return 0;

}

矩阵连乘：

#include <iostream>

#include "math.h"

#include <iomanip>

using namespace std;

class MatrixChain

{

public:

MatrixChain(int mSize, int\* q); //创建二维数组m和s，一维数组p，并初始化

int MChain(); //一般动态规划法求最优解值

int LookupChain(); //备忘录方法求最优解值（程序7-4）

void Traceback(); //构造最优解的公有函数

void Output();

~MatrixChain();

private:

void Traceback(int i, int j); //构造最优解的私有递归函数

int LookupChain(int i, int j); //备忘录方法私有递归（程序7-4）

int\* p; //p用于存储矩阵的维度信息

int\*\* m;

int\*\* s;

int n;

};

MatrixChain::MatrixChain(int mSize, int\* q)

{

n = mSize;

m = new int\* [n];

s = new int\* [n];

p = new int[n + 1];

for (int i = 0; i < n; i++)

{

m[i] = new int[n];

m[i][i] = 0;

s[i] = new int[n];

s[i][i] = i;

p[i] = q[i];

}

p[n] = q[n];

}

MatrixChain::~MatrixChain() {

for (int i = 0; i <n; i++) {

delete m[i];

delete s[i];

}

delete m;

delete s;

delete p;

}

int MatrixChain::MChain(){

for (int i = 0; i < n; i++) { //初始化矩阵对角线

m[i][i] = 0;

}

for (int r = 2; r <= n; r++) { // r控制不同级别对角线之间的步长

for (int i = 0; i < n - r + 1; i++) { // i控制计算起始的行数

int j = i + r - 1; // 根据偏移量计算出列数

m[i][j] = m[i + 1][j] + p[i] \* p[i + 1] \* p[j + 1]; //省略了m[i][i]=0项

s[i][j] = i;

for (int k = i + 1; k < j; k++) //遍历所有分割元素

{

int t = m[i][k] + m[k + 1][j] + p[i] \* p[k + 1] \* p[j + 1]; //计算当前计算数量

if (t < m[i][j])

{

m[i][j] = t; //更新当前的m矩阵和s矩阵

s[i][j] = k;

}

}

}

}

return m[0][n-1];

}

void MatrixChain::Traceback(int i, int j){

if (i == j) { //如果是单个矩阵 直接输出

cout << 'A' << i << " ";

return;

}

//i小于分割元素，用（）将分割元素内的矩阵连乘包裹并且递归调用

if (i < s[i][j]) {

cout << '(';

}

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

if (i < s[i][j]) {

cout << ')';

}

//i大于分割元素，用（）将分割元素内的矩阵连乘包裹并且递归调用

if (s[i][j] + 1 < j) {

cout << '(';

}

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

if (s[i][j] + 1 < j) {

cout << ')';

}

}

void MatrixChain::Traceback(){

cout << '(';

Traceback(0, n - 1);

cout << ')' << endl;

}

void MatrixChain::Output(){

//输出算法中的m矩阵和 s矩阵

int i, j;

cout << " m=" << endl;

cout << " ";

for (j = 0; j < n; j++)

if (j < 2) cout << setw(4) << j;

else cout << setw(6) << j;

cout << endl;

for (i = 0; i < n; i++)

{

cout << " " << i << " ";

for (j = 0; j < n; j++)

{

if (i < j) cout << setw(6) << m[i][j]; //setw(6), 指定输出域宽为6

else if (i == j) cout << setw(2) << m[i][j];

else cout << setw(6) << " ";

}

cout << endl;

}

cout << " s=" << endl;

cout << " ";

for (j = 0; j < n; j++) cout << j << " ";

cout << endl;

for (i = 0; i < n; i++)

{

cout << " " << i << " ";

for (j = 0; j < n; j++)

{

if (i <= j) cout << s[i][j] << " ";

else cout << " ";

}

cout << endl;

}

}

int MatrixChain::LookupChain(int i, int j){

if (m[i][j] > 0) { //如果问题已经得到解决 直接返回结果

return m[i][j];

}

if (i == j) {

return 0; //对角线上值为0

}

m[i][j] = LookupChain(i + 1, j) + p[i] \* p[i + 1] \* p[j + 1];

s[i][j] = i;

for (int k = i + 1; k < j; k++){

int v = LookupChain(i, k) + LookupChain(k + 1, j) + p[i] \* p[k + 1] \* p[j + 1];

if (v < m[i][j]){

m[i][j] = v;

s[i][j] = k;

}

}

return m[i][j];

}

int MatrixChain::LookupChain(){

return LookupChain(0, n - 1);

}

void main()

{

int nn = 6; //矩阵的个数

int k;

int pp[7] = { 30,35,15,5,10,20,25 }; //6个可以连乘的矩阵的7个维度信息

MatrixChain mm(nn, pp);

cout << endl << "1.不用备忘录方法求矩阵连乘的数乘最少次数" << endl;

k = mm.MChain();

cout << " 最少数乘次数k=" << k << endl;

mm.Traceback();

cout << endl;

mm.Output();

//下面是备忘录方法

cout << endl << "2.用备忘录方法求矩阵连乘的数乘最少次数" << endl;

MatrixChain a(nn, pp);

k = a.LookupChain(); cout << " 最少数乘次数k=" << k << endl;

a.Traceback();

cout << endl;

a.Output();

}