Project2 动态规划实验报告

徐煜森 PB16110173

一． 实验要求

实验 1：实现矩阵链乘问题的求解算法。对 n 的取值分别为：5、10、20、30，随机生成 n+1 个整数值（p_0、p_1、…、p_n）代表矩阵的规模，其中第 i 个矩阵($1 \le i \le n$)的规模为 $p_{i-1} \times p_i$，用动态规划法求出矩阵链乘问题的最优乘法次序，统计算法运行所需时间，画出时间曲线，进行性能分析。

实验 2：实现最长公共子序列问题的求解算法。序列 X 的长为 m，序列 Y 的长为 n，序列 X 和 Y 的元素从 26 个大写字母中随机生成，m 和 n 的取值：

第 1 组：(15，10)，(15，20)，(15，30)，(15，40)，(15，50)，(15，60)

第 2 组：(15，25)，(30，25)，(45，25)，(60，25)，(75，25)，(90，25)

给出算法运行所需的时间，画出时间曲线，进行性能分析。

二． 实验环境

1. Windows10 64 位 x86，机器内存 8G，时钟主频 2.59GHz
2. 软件环境：Visual Studio 2017

三． 实验过程

0. Makefile

Makefile

```
g++ -std=c++11 ex1_matmul.cpp -o ex1_matmul

```

Makefile ex1\... Makefile ex2\...

```
g++ -std=c++11 ex2_LCS.cpp -o ex2_LCS

```

为方便助教在 Linux 环境下进行测试，为两个实验写了两个 Makefile。不过我实验是在 Windows 10 下使用 visual studio 2017 做的。

1. Ex1

1.1 生成数据

generate_ex1.cpp

```cpp
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <time.h>
using namespace std;

int main()
{
    ofstream fileout;
    fileout.open("../input/input.txt");

    srand(unsigned(time(0)));                    //初始化随机种子
    for (int i = 0; i < 31; i++) {
        fileout << rand() % 30 + 1 << endl;
    }
    return 0;
}
```

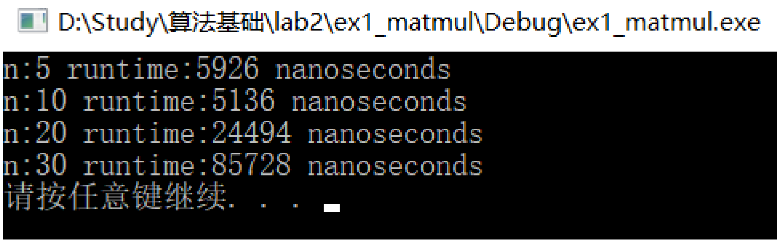
使用 rand()%30 + 1 生成[1, 30]区间的数据，每行一个数据，共 31 行。

1.2 矩阵链乘的动态规划算法实现与课堂上介绍的一致，将在代码截图与解析中说明。

1.3 实验大体测试时间的框架与 project 1 相同，更改了输入输出部分和算法。

1.4 实验结果截图

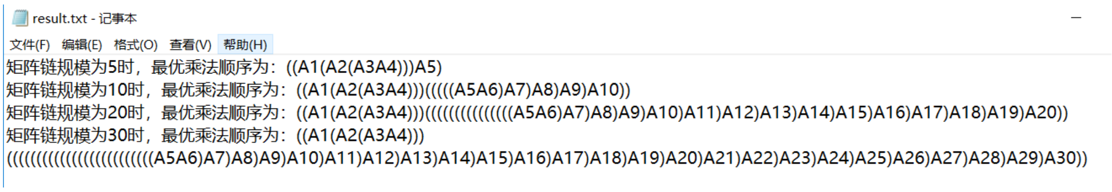
屏幕输出的运行时间（运行时间也按要求输出至文件）：



D:\Study\算法基础\lab2\ex1_matmul\Debug\ex1_matmul.exe

```
n:5 runtime:5926 nanoseconds
n:10 runtime:5136 nanoseconds
n:20 runtime:24494 nanoseconds
n:30 runtime:85728 nanoseconds
请按任意键继续. . .
```

计算结果输出：



result.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
矩阵链规模为5时，最优乘法顺序为：((A1(A2(A3A4)))A5)
矩阵链规模为10时，最优乘法顺序为：((A1(A2(A3A4)))(((((A5A6)A7)A8)A9)A10))
矩阵链规模为20时，最优乘法顺序为：((A1(A2(A3A4)))(((((((((((((((A5A6)A7)A8)A9)A10)A11)A12)A13)A14)A15)A16)A17)A18)A19)A20))
矩阵链规模为30时，最优乘法顺序为：((A1(A2(A3A4)))
(((((((((((((((((((((((((A5A6)A7)A8)A9)A10)A11)A12)A13)A14)A15)A16)A17)A18)A19)A20)A21)A22)A23)A24)A25)A26)A27)A28)A29)A30))
```

2. Ex2

2.1 生成数据

按要求生成 A、B 两组数据，每组六对，具体规模见实验要求。

generate_ex2.cpp

```cpp
int main()
{
    ofstream fileoutA, fileoutB;
    fileoutA.open("../input/inputA.txt");
    fileoutB.open("../input/inputB.txt");

    srand(unsigned(time(0)));                          //初始化随机种子
    // inputA
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 15; j++) {
            fileoutA << (char)('A' + rand() % 26);
        }
        fileoutA << endl;
        for (int j = 0; j < (i + 1) * 10; j++) {
            fileoutA << (char)('A' + rand() % 26);
        }
        fileoutA << endl;
    }
    // inputB
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < (i + 1) * 15; j++) {
            fileoutB << (char)('A' + rand() % 26);
        }
        fileoutB << endl;
        for (int j = 0; j < 25; j++) {
            fileoutB << (char)('A' + rand() % 26);
        }
        fileoutB << endl;
    }
    return 0;
}

```

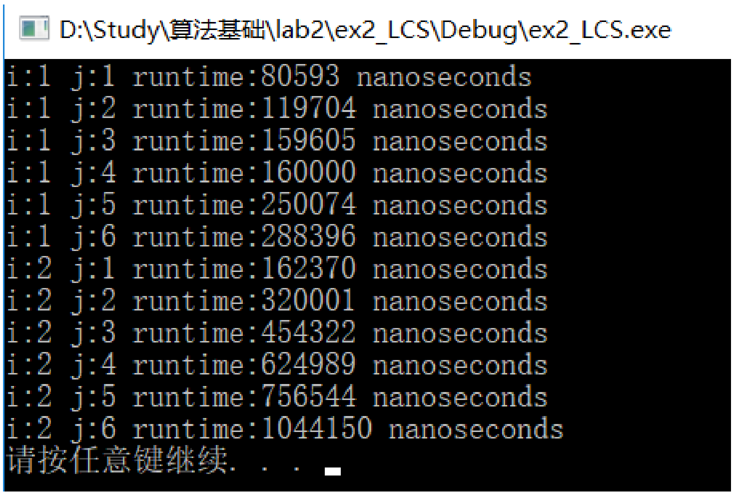
2.2 最长公共子序列的动态规划算法实现与课堂上介绍的一致，将在代码截图与解析中说明

2.3 实验框架与实验 1 相同，更改了输入输出和算法。

2.4 实验结果截图

屏幕输出的运行时间（运行时间也按要求输出至文件）：

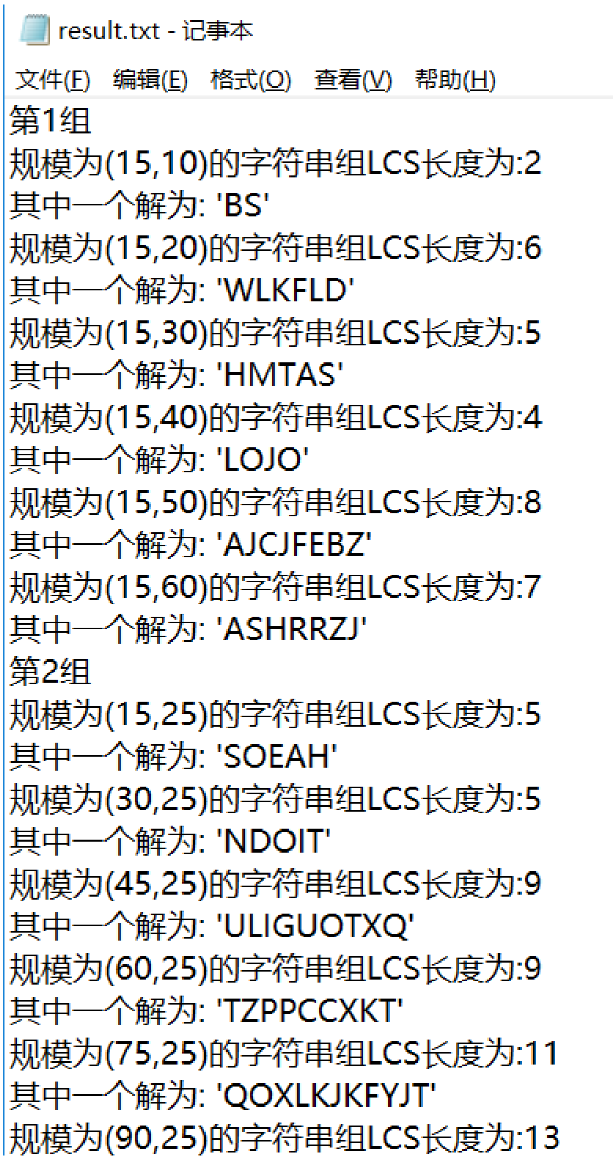
其中 i 表示组，j 表示对，如 i = 1，j = 1 表示第 1 组第一对数据。



D:\Study\算法基础\lab2\ex2_LCS\Debug\ex2_LCS.exe

```
i:1  j:1  runtime:80593  nanoseconds
i:1  j:2  runtime:119704  nanoseconds
i:1  j:3  runtime:159605  nanoseconds
i:1  j:4  runtime:160000  nanoseconds
i:1  j:5  runtime:250074  nanoseconds
i:1  j:6  runtime:288396  nanoseconds
i:2  j:1  runtime:162370  nanoseconds
i:2  j:2  runtime:320001  nanoseconds
i:2  j:3  runtime:454322  nanoseconds
i:2  j:4  runtime:624989  nanoseconds
i:2  j:5  runtime:756544  nanoseconds
i:2  j:6  runtime:1044150  nanoseconds
请按任意键继续. . .
```

计算结果输出：



result.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

第1组
规模为(15,10)的字符串组LCS长度为:2
其中一个解为: 'BS'
规模为(15,20)的字符串组LCS长度为:6
其中一个解为: 'WLKFLD'
规模为(15,30)的字符串组LCS长度为:5
其中一个解为: 'HMTAS'
规模为(15,40)的字符串组LCS长度为:4
其中一个解为: 'LOJO'
规模为(15,50)的字符串组LCS长度为:8
其中一个解为: 'AJCJFEBZ'
规模为(15,60)的字符串组LCS长度为:7
其中一个解为: 'ASHRRZJ'
第2组
规模为(15,25)的字符串组LCS长度为:5
其中一个解为: 'SOEAH'
规模为(30,25)的字符串组LCS长度为:5
其中一个解为: 'NDOIT'
规模为(45,25)的字符串组LCS长度为:9
其中一个解为: 'ULIGUOTXQ'
规模为(60,25)的字符串组LCS长度为:9
其中一个解为: 'TZPPCCXKT'
规模为(75,25)的字符串组LCS长度为:11
其中一个解为: 'QOXLKJKFYJT'
规模为(90,25)的字符串组LCS长度为:13

四．　关键代码截图

1. Ex1

动态规划算法：

```
long long MatrixChainOrder(int p[], int length) {
    auto start = steady_clock::now();

    int n = length;
    for (int i = 1; i <= n; i++) {
        m[i][i] = 0;
    }
    for (int l = 2; l <= n; l++) {
        for (int i = 1; i <= n - l + 1; i++) {
            int j = i + l - 1;
            m[i][j] = 0xfffffff;
            for (int k = i; k <= j - 1; k++) {
                int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j]) {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }

    auto end = steady_clock::now();
    auto duration = duration_cast<nanoseconds>(end - start);
    return duration.count();
}
```

可以看到动态规划算法与课本上一致，其中值得注意的是：

a. 实验中 p，m，s 均为全局数组，在算法执行前需要进行必要的初始化。

b. 可以看出这里令 n = length，而课本上伪代码中令 n = length – 1。原因是我的实现中 length 为矩阵的个数，而课本上伪代码的 length 为数组的长度，即为矩阵个数 + 1。

结果输出：

输出的方法与课本一致，均为递归调用输出。

```cpp
void DoPrint(int i, int j) {
    if (i == j) {
        result_out << "A" << i;
    }
    else {
        result_out << "(";
        DoPrint(i, s[i][j]);
        DoPrint(s[i][j] + 1, j);
        result_out << ")";
    }
}

void PrintOptimalParens(int i, int j) {
    result_out << "矩阵链规模为" << int_to_string(j) << "时，最优乘法顺序为：";
    DoPrint(i, j);
    result_out << endl;
}
```

2. Ex2

动态规划算法：

```cpp
long long LCS_Length() {
    auto start = steady_clock::now();

    int m = X.length();
    int n = Y.length();
    for (int i = 1; i <= m; i++) {
        c[i][0] = 0;
    }
    for (int j = 0; j <= n; j++) {
        c[0][j] = 0;
    }
    // b[i][j] = 0 represent ↖ upleft
    // 1 represent ↑ up
    // 2 represent ← left
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i - 1] == Y[j - 1]) {
                c[i][j] = c[i - 1][j - 1] + 1;
                b[i][j] = 0;                    // upleft
            }
            else if (c[i - 1][j] >= c[i][j - 1]) {
                c[i][j] = c[i - 1][j];
                b[i][j] = 1;                    // up
            }
            else {
                c[i][j] = c[i][j - 1];
                b[i][j] = 2;                    // left
            }
        }
    }

    auto end = steady_clock::now();
    auto duration = duration_cast<nanoseconds>(end - start);
    return duration.count();
}
```

动态规划算法与课本上一致，其中使用 0,1,2 分别代表左上↖，上↑，左←。

结果输出：

```cpp
void DoPrint(int xlen, int ylen) {
    if (xlen == 0 || ylen == 0) {
        return;
    }
    if (b[xlen][ylen] == 0) {       // upleft
        DoPrint(xlen - 1, ylen - 1);
        result_out << X[xlen - 1];
    }
    else if (b[xlen][ylen] == 1) {  // up
        DoPrint(xlen - 1, ylen);
    }
    else {                          // left
        DoPrint(xlen, ylen - 1);
    }
}

void PrintLCS(int i, int j) {
    if (i == 1) {
        result_out << "规模为(15," << j * 10 << ")的字符串组LCS长度为:" << c[X.length()][Y.length()] << endl;

    }
    else if (i == 2) {
        result_out << "规模为(" << j * 15 << ",25)的字符串组LCS长度为:" << c[X.length()][Y.length()] << endl;
    }
    result_out << "其中一个解为: '";
    DoPrint(X.length(), Y.length());
    result_out << "'" << endl;
}
```

按照要求格式进行输出，输出 LCS 的方法与课本一致为递归输出。

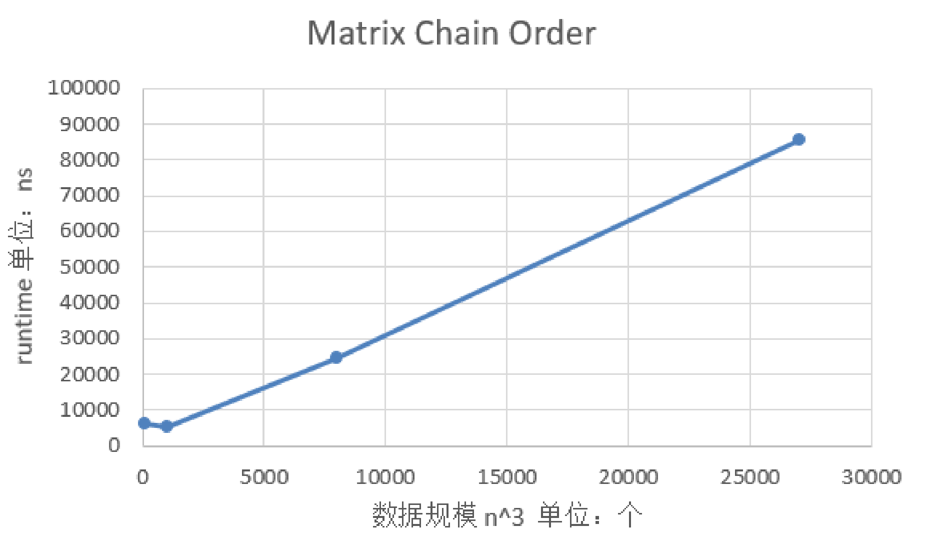
五． 实验结果及分析

1. Ex1

实验数据如下：

Matrix Chain Order		
n	n^3	runtime (ns)
5	125	5926
10	1000	5136
20	8000	24494
30	27000	85728

画出折线图如下：



分析：

a. 发现运行时间与数据规模 n^3 基本成线性关系，符合期望 O(n^3)。

b. 发现在 n=5 时运行时间反而比 n=10 时长，分析原因可能是 cache 存在的缘故。在 n=5 首次运算时各个数组被调入 cache，n=10 时发现大部分需要用到的数据都可以在 cache 中找到，减少了从内存调入 cache 的时间。

2. Ex2

实验数据如下：

第一组：

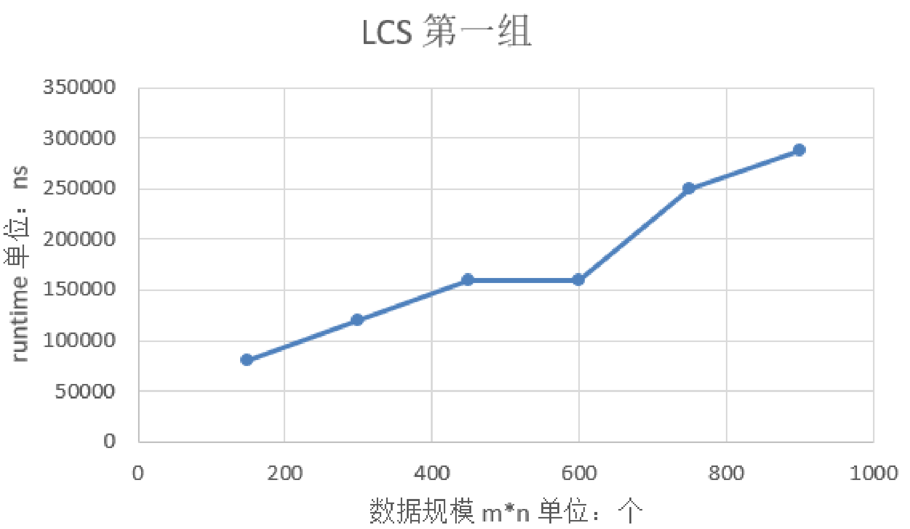
LCS	第一组		
m	n	m*n	runtime（ns）
15	10	150	80593
15	20	300	119704
15	30	450	159605
15	40	600	160000
15	50	750	250074
15	60	900	288396

第二组：

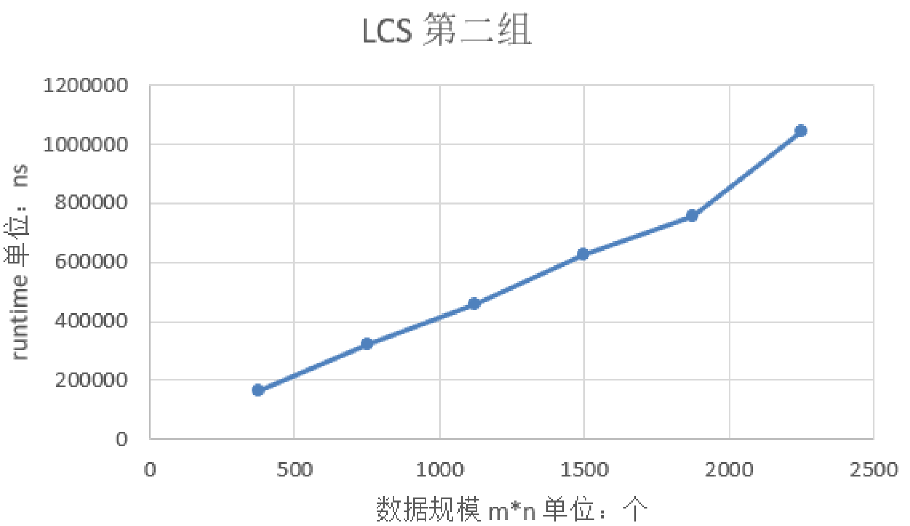
LCS	第二组		
m	n	m*n	runtime（ns）
15	25	375	162370
30	25	750	320001
45	25	1125	454322
60	25	1500	624989
75	25	1875	756544
90	25	2250	1044150

画出折线图：

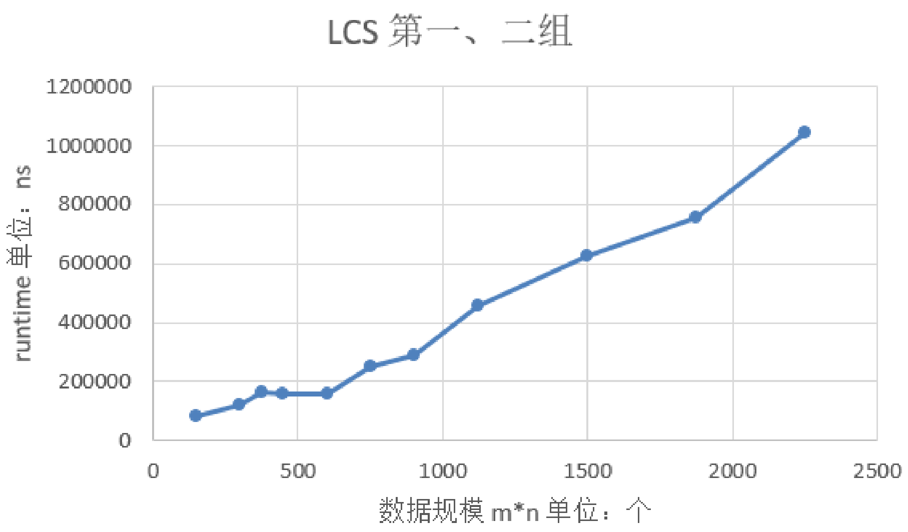
第一组：



第二组：



将两组数据画在同一张图上：



分析：

a. 综合看来，运行时间与数据规模 m*n 基本成线性关系，符合期望 O（m*n）。

b. 从第一组和第二组图中可以看出，除第一组图中有波动外，运行时间几乎与 m*n 成完美的线性关系。而将第一组和第二组的数据画在一张图上时，可以看出在 m*n 较小时，运行时间有明显波动。分析原因可能是在 m*n 较小时，对数组的初始化消耗时间不可忽略，导致运行时间的常数因子有所变化。

c. 当 m*n 较大时，运行时间基本稳定与 m*n 成线性关系。

六． 实验总结

本次实验中有些重要的小问题，就如在 Ex1 的实验过程中提到的 n=length 与课本不一致的问题。看似是个小疏忽，实际上反映出自己对算法的掌握仍有不足，对算法没有理解到位，值得反思。