# Project 2 report

## 实验内容与要求

### ex1

- 求解矩阵链乘最优方案，使得链乘过程中乘法次数最少
- 记录矩阵连乘最优方案，并记录运行时间，画曲线分析

### ex2

- 求最长公共子序列，打印最长子序列
- 记录运行时间，画出曲线进行分析

## 实验配置与环境

- 操作系统：Windows 11 专业版（version 21H2）
- 处理器：11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- 时钟主频：2.80 GHz
- 编译器：gcc version 8.1.0

## 实验方法与步骤

### ex1

1. **编写求解矩阵链乘最少次数方案的程序：**

```
long long m[50][50];
int solve[50][50];
long long matrix_chain_order(long long p[], int n){
    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] = INT64_MAX;
            for(int k = i; k <= j - 1; k++){
                long long q = m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j];
                if(q < m[i][j]){
                    m[i][j] = q;
                    solve[i][j] = k;
                }
            }
        }
    }
    return m[1][n];
}
```

由于题目中的乘法次数可能超过32位表示范围，所以这里采用long long 型变量进行运算。

代码的基本思路是基于矩阵链乘动态规划的递推式进行构建：

$$m[i][j] = \begin{cases} 0 & i = j \\ min_{i<=k<j}\{m[i][k] + m[k+1][j] + p_{i-1}p_jp_k\} & i < j \end{cases}$$

算法整体分为两步：

- 初始化边界条件，即将m中i=j的位置全部赋值为0
- 执行动态规划算法：首先求解长度为2的矩阵链乘问题，之后逐渐扩大规模，使用$m[i][j]$来保存第i个到第j个矩阵的最优链乘方案，使用递推式进行维护，并使用solve数组来保存最优方案的解。

最终返回$m[1][n]$即可获得最少乘法次数。

2. **编写构建最优解的程序：**

```
void print_optimal_parens(ofstream &fout, int i, int j){
    if(i == j) {
        cout << "A" << i;
        fout << "A" << i;
    }
    else{
        cout << "(";
        fout << "(";
        print_optimal_parens(fout, i, solve[i][j]);
        print_optimal_parens(fout, solve[i][j] + 1, j);
        cout << ")";
        fout << ")";
    }
}
```

这是借助solve数组进行递归恢复最优解方案的程序，利用了矩阵链乘全局最优解由局部最优解构成的特点，递归缩小问题规模，并输出最优括号化方案。

3. **使用测试用例测试并分析结果**

- 我首先对规模为5的测试用例进行手动计算，得出其最优解和最少链乘次数，并与算法运行结果进行比对，初步验证了其正确性。
- 我对单个规模运行100000次算法，使用clock()统计总时间恰好就是一次运行所需的纳秒数。
- 我对每个规模进行3次重复试验，算出平均时间，并用excel绘制图表。

## ex2

1. **编写求公共最长子序列的程序**

```
int b[50][50];
int c[50][50];
int LCS_length(char x[], int m, char y[], int n){
    for(int i = 1; i <= m; i++){
        c[i][0] = 0;
    }
    for(int j = 1; j < n; j++){
        c[0][j] = 0;
    }
    for(int i = 1; i <= m; i++){
        for(int j = 1; j <= n; j++){
            if(x[i] == y[j]){
                c[i][j] = c[i-1][j-1] + 1;
                b[i][j] = 0;
            }
            else if(c[i-1][j] > c[i][j-1]){
```

```
                c[i][j] = c[i-1][j];
                b[i][j] = 1;
            }
            else {
                c[i][j] = c[i][j-1];
                b[i][j] = -1;
            }
        }
    }
    return c[m][n];
}
```

这里我使用了b数组来保存最优序列的构建方法，使用$c[i][j]$来保存X的前i个字符和Y的前j个字符的最长公共子序列长度。

c数组的递推式如下：

$$c[i][j] = \begin{cases} 0 & i = 0或j = 0 \\ c[i-1][j-1]+1 & i,j>0且x[i]=y[j] \\ max\{c[i-1][j], c[i][j-1]\} & i,j>0且x[i]\neq y[j] \end{cases}$$

因此，我们在算法中，应现将i=0或j=0的c数组位置全部置0，之后对每一对i,j，都使用上述递推式进行递推，利用全局最优解是由局部最优解构成的特点，完成c中所有位置的计算，最后返回

$c[m][n]$即为最优解。

在这里，$b[i][j]=0$表示指向$b[i-1][j-1]$，$b[i][j]=1$表示指向$b[i-1][j]$，$b[i][j]=-1$表示指向$b[i][j-1]$，用以构建最优解。

2. **编写构建最优解的程序：**

```
void print_LCS(ofstream &fout, char x[], int i, int j){
    if(i == 0 || j == 0) return;
    if(b[i][j] == 0){
        print_LCS(fout, x, i-1, j-1);
        cout << x[i] << " ";
        fout << x[i] << " ";
    }
    else if(b[i][j] == 1) print_LCS(fout, x, i-1, j);
    else print_LCS(fout, x, i, j-1);
}
```

这个程序利用从m,n进行回溯递归，来正向打印出最优的子序列。

3. **使用测试用例测试并分析结果**
    - 我对单个规模运行100000次算法，使用clock()统计总时间恰好就是一次运行所需的纳秒数。
    - 我对每个规模进行3次重复试验，算出平均时间，并用excel绘制图表。
    - 对于10规模下的输入，我手工计算了最优子序列，初步验证了算法的正确性

# 实验结果与分析

## ex1

- 规模为5的输出结果：

```
matrix_linkmul.cpp U    result.txt U ×    LCS_length.cpp U    ASM ex7_9.s U
ex1 > output > result.txt
Scale = 5
The multiply time of scale 5 is:
        1               2               3               4               5
5       154865959097238 128049683226820 74062781976714  15903764653528  0
4       138766801119366 105723424955724 43981152513978  0
3       183439291324068 119490227350806 0
2       120958281818244 0
1       0

The solution of scale 5 is:
        1   2   3   4   5
5       1   1   1   1   0
4       4   3   2   0
3       4   3   0
2       4   0
1       0
```

其中，每个表格的最左端一列和最上方一行为序号，中间的三角矩阵为不同规模下乘法总次数和解的序号。

- 其余规模的最优方案与最少乘法次数

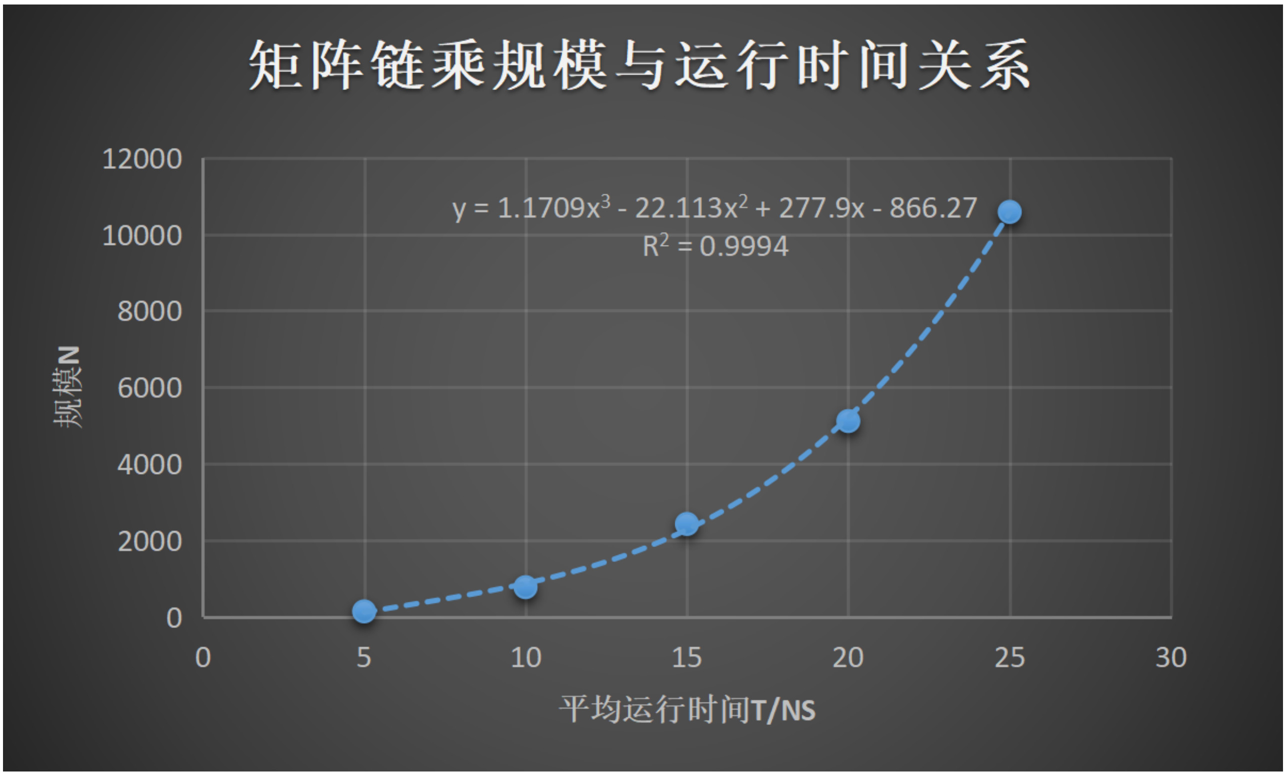
```
Scale = 10
Least multiply time:    42524697503391
Divide method:  ((A1A2)(((((((A3A4)A5)A6)A7)A8)A9)A10))

Scale = 15
Least multiply time:    5400945319618
Divide method:  ((((((((((((((A1A2)A3)A4)A5)A6)A7)A8)A9)A10)A11)A12)A13)A14)A15)

Scale = 20
Least multiply time:    319329979644400
Divide method:  ((A1(A2(A3(A4(A5(A6(A7(A8(A9(A10(A11(A12(A13(A14A15))))))))))))))((((A16A17)A18)A19)A20))

Scale = 25
Least multiply time:    574911761218280
Divide method:  ((A1(A2(A3(A4(A5(A6(A7(A8(A9(A10A11))))))))))(((((((((((((A12A13)A14)A15)A16)A17)A18)A19)A20)A21)A22)A23)A24)A25))
```

- 运行时间分析

三次运行的结果如下：

| 矩阵链乘规模与运行时间关系 | | | | |
|---|---|---|---|---|
| 规模n | 运行时间t1/ns | 运行时间t2/ns | 运行时间t3/ns | 平均运行时间t/ns |
| 5 | 151 | 141 | 132 | 141.3333333 |
| 10 | 755 | 782 | 785 | 774 |
| 15 | 2344 | 2384 | 2550 | 2426 |
| 20 | 5174 | 5042 | 5130 | 5115.333333 |
| 25 | 10162 | 10698 | 10881 | 10580.33333 |

绘制图表并拟合曲线可得：




拟合的可决系数与1很接近，证明用n$^3$曲线拟合度较好，这也说明了算法真实的时间复杂度确实和理论的复杂度较为相同，为$\theta(n^3)$

## ex2

- 运行输出结果

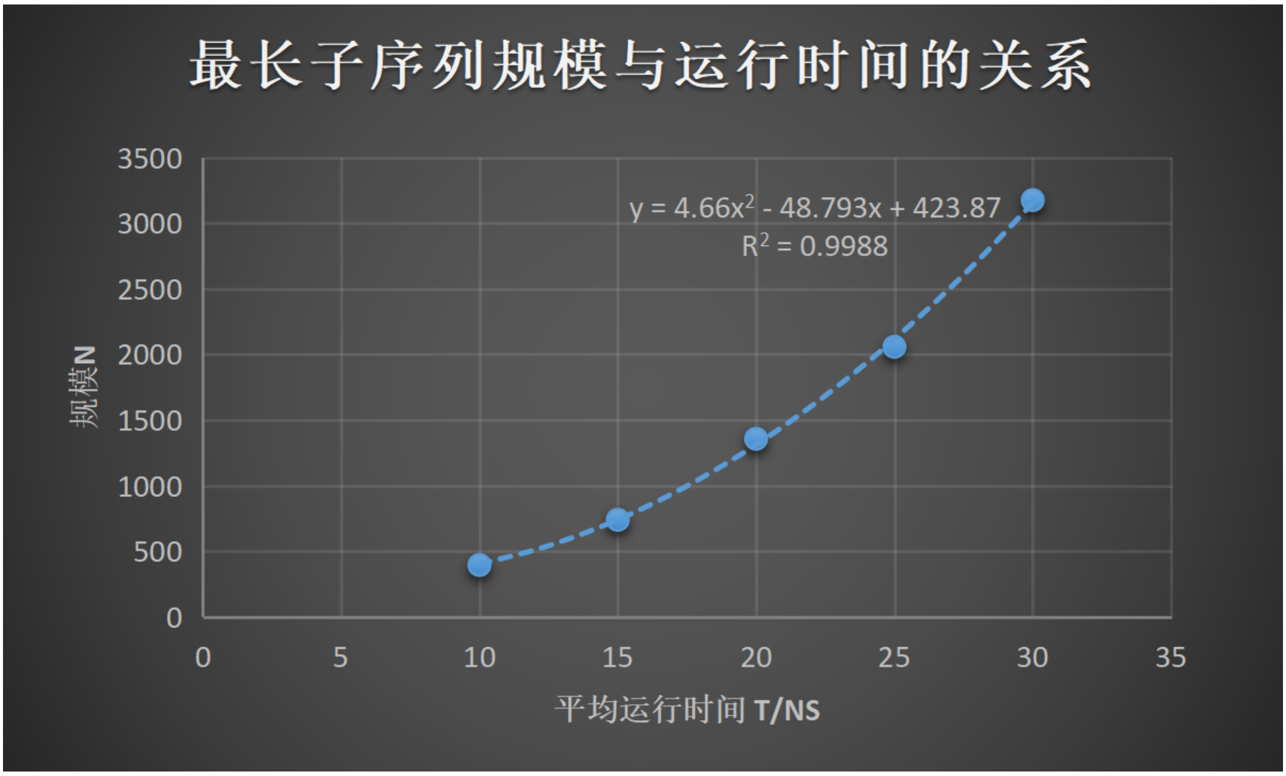
```
Scale = 10
LCS length: 5
LCS is: D A B A B

Scale = 15
LCS length: 8
LCS is: A C C B D C C C

Scale = 20
LCS length: 12
LCS is: C B A C D A D C B B D A

Scale = 25
LCS length: 14
LCS is: C D D B B B C C D D B A D D

Scale = 30
LCS length: 16
LCS is: A D D B B C D B B C D D D C B D
```

- 运行时间分析

  三次测得的运行时间如下：

| 最长子序列规模与运行时间的关系 | | | | |
|---|---|---|---|---|
| 规模n | 运行时间t1/ns | 运行时间t2/ns | 运行时间t3/ns | 平均运行时间t/ns |
| 10 | 396 | 373 | 416 | 395 |
| 15 | 722 | 744 | 752 | 739.3333333 |
| 20 | 1364 | 1334 | 1373 | 1357 |
| 25 | 2079 | 2072 | 2022 | 2057.666667 |
| 30 | 3030 | 3127 | 3371 | 3176 |

  绘制图表并进行拟合：




可以看到，实验结果与理论时间$\theta(mn)$（这里m=n）拟合程度较好，基本符合理论的时间复杂度

# 实验总结

通过本次实验，我有了如下收获：

- 进一步加深了对动态规划的理解
- 验证了这两个动态规划问题的真实时间复杂度
- 加强了自己的数据分析能力和数据处理能力