lab2动态规划和FFT

实验内容及要求

- ■实验2.1：求矩阵链乘最优方案
  - □n个矩阵链乘，求最优链乘方案，使链乘过程中乘法运算次数最少。
  - □n的取值5, 10, 15, 20, 25，矩阵大小见2_1_input.txt。
  - □求最优链乘方案及最少乘法运算次数，记录运行时间，画出曲线分析。
  - □仿照P214 图15-5，打印n=5时的结果并截图。

- ■实验2.2：FFT
  - □多项式$A(x)=\sum_{i=0}^{n-1} a_i x^i$，系数表示为$(a_0,a_1,\ldots,a_{n-1})$。
  - □n取$2^3,2^4,\ldots,2^8$，不同规模下的A见2_2_input.txt。
  - □用FFT求A在$\omega_n^0,\omega_n^1,\ldots,\omega_n^{n-1}$处的值。
  - □记录运行时间，画出曲线分析；打印$n=2^3$时的结果并截图。

实验设备和环境

- 编译运行环境
  - Windows10-mingw-w64
  - vscode
CPU	AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx
CPU核心数	4
逻辑处理器	8
主频	2100 Mhz
外频	100 Mhz
一级数据缓存	4 * 32KB
一级指令缓存	4 * 64KB
二级缓存	4 * 512KB
数据宽度	64 bit

实验方法和步骤

计时函数

- 上次实验采用了C++的计时函数，效果不是很理想，本次采用了`<windows.h>`中的`QueryPerformance`来计时，精度更高，避免了之前数据量较小时时间为0的情况
- 开始计时

```
LARGE_INTEGER t1, t2, tc;
QueryPerformanceFrequency(&tc);
QueryPerformanceCounter(&t1);
```

- 进行函数调用
- 结束计时并计算用时，以微秒为单位

```
QueryPerformanceCounter(&t2);
double time = (double)(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
```

文件读写

- 采用c++文件读写流进行文件的读写
- 因为两个输入文件格式基本相同，故两个部分文件读写采用的方式基本相同，这里以矩阵链乘为例
- 采用二维 vec 来存放读入的数据
- 采用基础路径+需要的更深层的路径来描述文件路径（使用相对路径在debug时会出错）
- 输出时采用`<iomanip>`对数据进行格式化（对齐、保留一定的位数）
- 读

```
string base_path = "D:\\study\\algorithm_lab\\lab2\\ex1\\";
infile.open(base_path + "input\\2_1_input.txt");
cout << "Reading from the file 2_1_input.txt" << endl;

for (int i = 0; i < 5; ++i){
    infile >> matrix_num;
    vector<int> temp;
    for (int j = 0; j <= matrix_num; ++j){
        infile >> in_temp;
        temp.push_back(in_temp);
    }
    data_vec.push_back(temp);
}
infile.close();
```

- 写入文件（只写入最终结果）并输出到屏幕（输出整个矩阵）

```
        for (int j = 1; j < data_vec[i].size(); ++j){
            for (int k = 1; k < data_vec[i].size(); ++k){
                cout << left << setw(20) << m[j][k] << " ";
            }
            cout << endl;
        }
        outfile_result << m[1][data_vec[i].size() - 1] << endl;
```

矩阵链乘

关键函数

```
void matrix_chain_order(vector<int> p){
    int n = p.size() - 1;
    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] = MAX;
            for (int k = i; k <= j - 1; ++k){
                long long q = m[i][k] + m[k + 1][j];
                long long temp = p[i - 1];
                temp *= p[k];
                temp *= p[j];
                //cout << temp << endl;//容器不能在一个式子中多次引用
                q += temp;
                if(q < m[i][j])
                {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }
}
```

- 命名和课本中伪代码保持一致
- 需要注意课本中正无穷对应这里 long long的上限，在声明部分将其定义为MAX
- 需要特别注意的是三重循环的初始值和终止条件
    - 第一重循环对应矩阵链长度
    - 第二重循环计算当前长度所需的最少乘法次数
    - 第三重循环对切割的位置进行试探，确定代价最小的方案
- m、s均为全局二维数组，每次调用结束后要进行清零，避免出现错误

输出函数

```
void print_optimal_parens(int s[][MATRIX_SIZE], int i, int j){
    if(i == j){
        cout << "A";
        outfile_result << "A";
        return;
    }
    else{
        cout << "(";
        outfile_result << "(";
        print_optimal_parens(s, i, s[i][j]);
        print_optimal_parens(s, s[i][j] + 1, j);
        cout << ")";
        outfile_result << ")";
        return;
    }
```

```
}
```

- 命名和课本中伪代码保持一致
- 采用递归输出，根据前面函数生成的备忘s进行输出到屏幕和文件

main函数

- 控制计时
- 控制文件读写
- 在恰当的时机对计算链乘次数的函数、输出函数进行调用
- 控制m、s的清零

FFT

关键函数

```
vector<complex<double>> recursive_fft(vector<complex<double>> a){
    int n = a.size();
    if(n == 1){
        return a;
    }
    complex<double> w_n = complex<double>(cos(2 * M_PI / n), sin(2 * M_PI / n));
    complex<double> w = complex<double>(1, 0);
    vector<complex<double>> a_0, a_1;
    for (int i = 0; i < n; i += 2){
        a_0.push_back(a[i]);
        a_1.push_back(a[i + 1]);
    }
    auto y_0 = recursive_fft(a_0);
    auto y_1 = recursive_fft(a_1);
    vector<complex<double>> y;
    for (int k = 0; k < n; ++k){
        y.push_back(complex<double>(0, 0));
    }
    for (int k = 0; k <= n / 2 - 1; ++k){
        y[k] = y_0[k] + w * y_1[k];
        y[k + n / 2] = y_0[k] - w * y_1[k];
        w *= w_n;
    }
    return y;
}
```

- 命名和课本中伪代码保持一致
- 采用c++的模板类 `complex<double>` 保存复数。template 中重载了复数的算符，计算较为方便
- 采用c++的容器 `vector` 保存向量，维度可以动态变化，较为方便
- 采用欧拉公式 $e^{\pm ix} = \cos x \pm i \sin x$ 表示w_n

main函数

- 控制计时
- 控制文件读写
- 在恰当的时机FFT函数进行调用

实验结果和分析

矩阵链乘

- n = 5时的结果截图

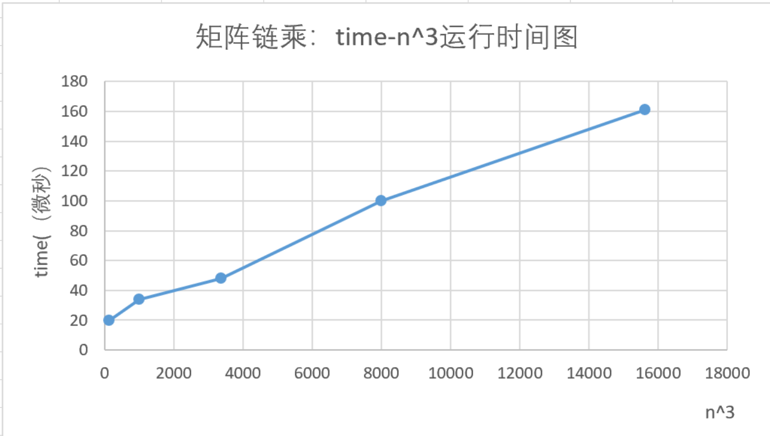
```
spend time = 1.97e-05
0               15903764653528          74062781976714          128049683226820         154865959097238
0               0                       43981152513978          105723424955724         138766801119366
0               0                       0                       119490227350806         183439291324068
0               0                       0                       0                       120958281818244
0               0                       0                       0                       0
```

- 最终结果以及最优括号化方案

```
154865959097238
(A(((AA)A)A))
```

- 运行时间分析（单位：微秒）

n	n^3	time(微秒)
5	125	20
10	1000	34
15	3375	48
20	8000	100
25	15625	161



由算法中的三重循环以及课后题15.2-5可知，该算法的时间复杂度为 Ω（n^3)

由上图可知，图像基本为一条直线，结果符合预期

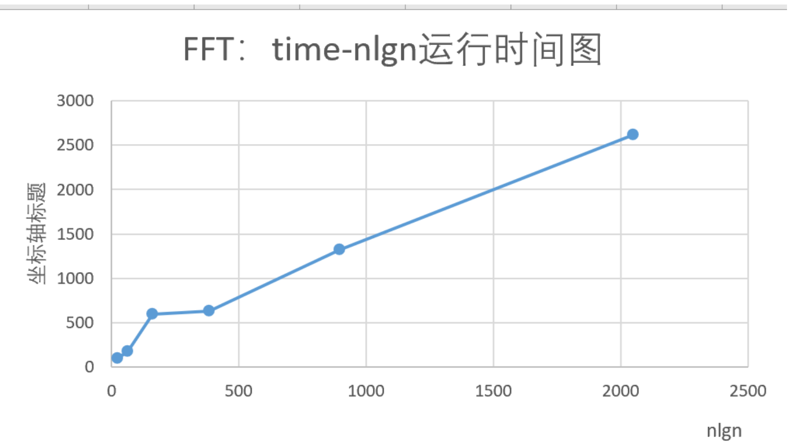
FFT

- n = 8时的结果截图（只打印了实部）

```
Reading from the file 2_2_input.txt
spend time = 0.000092
-10.000000 15.778175 5.000000  0.221825  -8.000000 0
.221825  5.000000  15.778175
```

- 时间复杂度分析

n	nlgn	time(微秒)
8	24	92
16	64	175
32	160	597
64	384	631
128	896	1322
256	2048	2616



该算法理论的时间复杂度为θ（nlgn）

由图像可知，图像基本为一条直线，结果符合预期

实验总结

- 经过本次实验，我对矩阵链乘算法、动态规划法的应用有了更加深入的理解
- 对于算法的空间和时间消耗也有了更加深刻的体会。特别是矩阵链乘中，通过记录s来实现后续最优括号化方案的输出
- 通过FFT实验，我对伪代码到实际代码中数据结构的使用有了更加深刻的体会，特别是模板类、容器的使用，可以使得代码大大简化