lab2 动态规划和FFT

实验内容及要求

- 动态规划法：
  - 求矩阵链乘最优方案，使得链乘过程中乘法运算次数最少
  - 求最少乘法运算次数，记录运行时间，画出曲线分析
  - 打印n=5时的结果并截图
- FFT
  - 求系数表示为$(a_0, a_1, \ldots, a_{n-1})$的多项式A(x)在不同规模下在$w_n^0, w_n^1, \ldots, w_n^{n-1}$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
  - Clion 2020.2.4
  - vscode
- 硬件
  - 处理器: 英特尔 Core i7-8750H @ 2.20GHz 六核
  - 速度 2.21 GHz (100 MHz x 22.0)
  - 处理器数量 核心数: 6 / 线程数: 12
  - 一级数据缓存 6 x 32 KB, 8-Way, 64 byte lines
  - 一级代码缓存 6 x 32 KB, 8-Way, 64 byte lines
  - 二级缓存 6 x 256 KB, 4-Way, 64 byte lines
  - 三级缓存 9 MB, 12-Way, 64 byte lines
  - 内存： 海力士 DDR4 2666MHz 8GB

实验方法和步骤

整体代码考虑

- 对于动态规划和FFT，都是分三步走
  - 处理文件读写
  - 处理计时函数
  - 处理算法具体的函数实现以及用到的数据结构
  - 两个实验都各自在一个main.cpp里实现了全部功能

代码设计思路

- 文件读写
  - 考虑使用fsteam头文件下的ofstream以及ifstream流来轻松实现文件读写。
  - 打印时利用`<iomanip>`来格式化输出
  - 需注意文件目录组织下，采用相对路径较为简单，个人因为使用Clion缘故，其运行时当前目录为cmake-build-debug,所以文件路径设置为

    `"..\\..\\input\\input.txt"`等

- 此外，Clion不支持中文路径，需要注意
- 在使用vscode时，则需换成绝对路径。

- 计时：采用网上推荐的格式，在经过试验后，发现微秒级的计时即可让结果显示较为方便。

  考虑采用<windows.h>头文件下一个us级的计时方式

```
//clock计时参数的声明
LARGE_INTEGER nFreq;
LARGE_INTEGER t1;
LARGE_INTEGER t2;
double dt;
```

```
QueryPerformanceFrequency(&nFreq);
QueryPerformanceCounter(&t1);
MATRIX_CHAIN_ORDER();//矩阵链乘计算
QueryPerformanceCounter(&t2);
dt =(t2.QuadPart-t1.QuadPart)/(double)nFreq.QuadPart;
time=dt*1000000;
```

  实验结果发现此时计时精度较高，每个数据规模下均有较为合理的测量结果。

- 动态规划矩阵链乘法计算函数
  - `void MATRIX_CHAIN_ORDER()`
    - 完成矩阵链乘法的自底向上表格法代码实现
    - 主要需要注意代码三重循环的逻辑：第一层是对矩阵链长度(从2到n)，第二层是该长度的各个矩阵链的最少乘法数的计算，第三层是对一个特定的矩阵链乘法，对其切割点作试探。
    - 此外对于`m[i][j]`的初始化，个人选择了-1，因为可能无法确定最大输入数据，所以用负数来消除可能的错误
    - 数据结构方面，考虑到表格法里数组空间确定，所以考虑使用三个全局数组或vector来避免传参的麻烦。

```
//存放矩阵链的维度
vector<long long> A;
//存放代价矩阵
long long m[30][30]={0};
//存放分割点矩阵
int s[30][30]={0};
```

      并且对5个量级数据共用同一个数组或vector

    - 代码如下

```
void MATRIX_CHAIN_ORDER(){
    int j=0;
    long long temp=0;
    long long q=0;
    int n=A.size()-1;
    for(int l=2;l<n+1;l++){//矩阵链长度从2到n
        for(int i=1;i<=n-l+1;i++){
            j=i+l-1;
            m[i][j]=-1;//取负来代替无穷
            for(int k=i;k<j;k++){
```

```
                    temp=A[i-1];
                    temp*=A[k];
                    temp*=A[j];
                    q=m[i][k]+m[k+1][j]+temp;
                    if(m[i][j]<0){
                        m[i][j]=q;
                        s[i][j]=k;
                    }
                    else if(q<m[i][j]){
                        m[i][j]=q;
                        s[i][j]=k;
                    }
                }
            }
        }
    }
```

- `void PRINT_OPTIMAL_PARENS()`
    - 完成递归的输出最优括号化方案
    - 此函数里同时在标准输出流和文件流(time.txt)中输出
- `int main()`
    - 采用一个for循环来对5个量级的数据读入，计时，运行矩阵链乘法算法，写入文件，再到清空全局变量，如此一来代码整体结构较为清晰。

- FFT函数实现
    - 考虑到递归调用的存在，所以全局变量共用不太现实，并且存在复数的计算，所以引入c++ 的vector和complex类
    - π通过宏定义设置，取`#define PI        3.1415926`
    - `vector<complex<double>> RECURSIVE_FFT(vector<double> &a)`
        - 完成主要的递归运算
        - 主要需要注意对于complex类实例的实部、虚部的初始化，然后因为complex类重载了*，+，-运算符，所以最后的for循环计算较为简洁。
        - 最后返回一个装有complex实例的vector作为计算结果y
        - 代码如下

```
vector<complex<double>> RECURSIVE_FFT(vector<double> &a){
    int n=a.size();
    //cout<<a[0]<<endl;
    vector<complex<double>> y = vector<complex<double>>(n);
    vector<double> a0,a1;

    complex<double> w_n {cos(2*PI/(double)n),sin(2*PI/(double)n)};
    complex<double> w {1.0,0.0};
    if(n==1){
        y[0].real(a[0]);
        y[0].imag(0.0);
        return y;
    }
    for(int i=0;i<n;i+=2){
        a0.push_back(a[i]);
        a1.push_back(a[i+1]);
    }
    vector<complex<double>> y0=RECURSIVE_FFT(a0);
```

```
    vector<complex<double>> y1=RECURSIVE_FFT(a1);

    for(int k=0;k<=(n/2-1);k++){
        y[k]=y0[k]+w*y1[k];
        y[k+n/2]=y0[k]-w*y1[k];
        w=w*w_n;
    }
    return y;
}
```

- `int main()`
  - 考虑反复用一个vector A来从文件读入每个量级下的参数a0,a1,...,an，并通过引用传递的方式传入RECURSIVE_FFT函数

实验结果与分析

动态规划

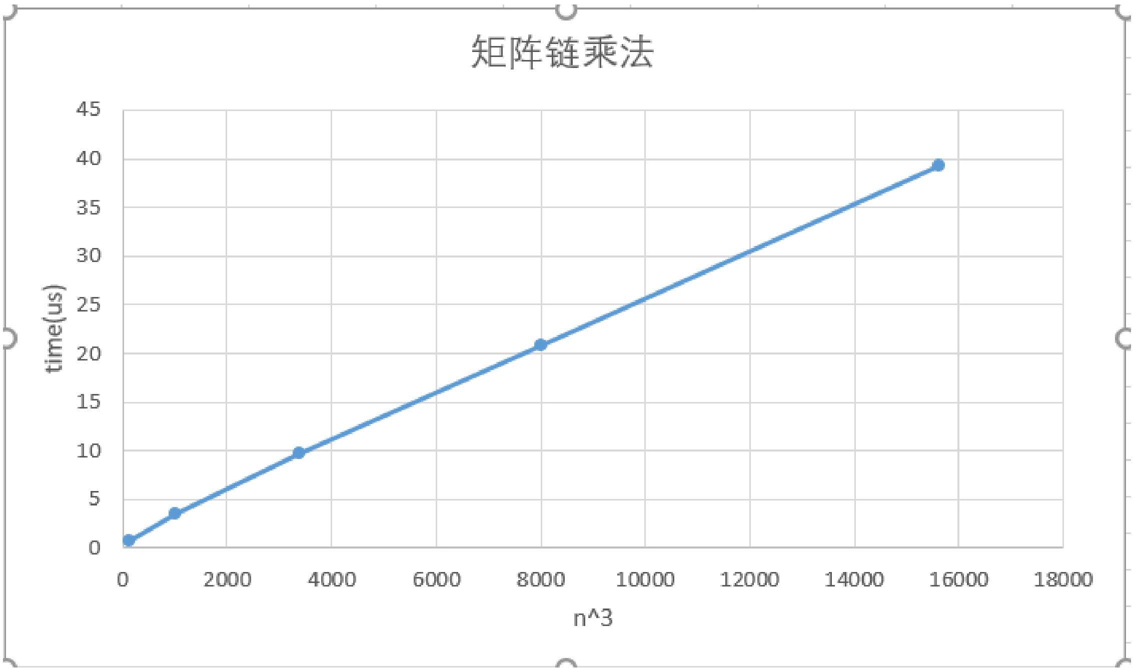
- n=5时的结果截图（clion下运行）

  包括m和s矩阵，以及最少乘法次数和最优括号化方案

```
m matrix
              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
s matrix
   1   1   1   1
   0   2   3   4
   0   0   3   4
   0   0   0   4
multi times  154865959097238
the best scheme:  (A1(((A2A3)A4)A5))
```

- 运行时间分析，时间单位为微妙(us)

n^3	time(us)
125	0.7
1000	3.5
3375	9.7
8000	20.9
15625	39.3



- 由算法本身的实现里的三重循环可知，算法的理论时间复杂度为$O(n^3)$,所以如上作出了$time - n^3$的图像，发现为线性函数，所以实际复杂度与理论复杂度一致，从而验证了算法里的三重循环的分析确实正确。

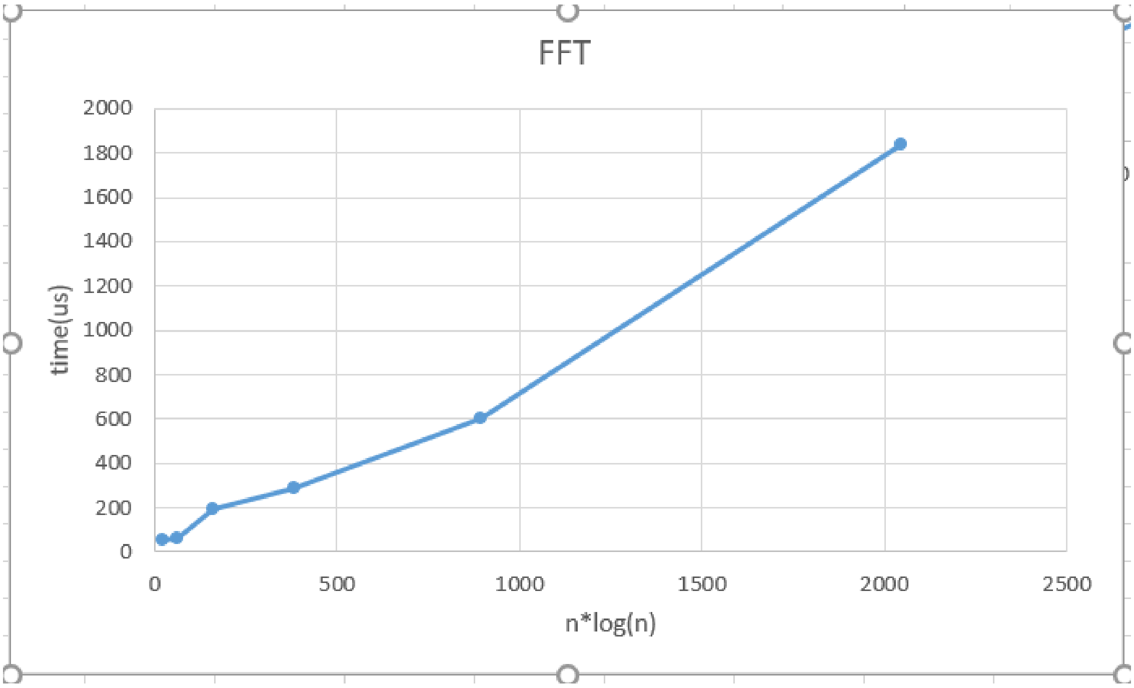
FFT

- 实验结果是通过vscode运行得到的，因为在clion下运行不知道什么原因n=8时的时间老是更大。但在vscode下运行则没有这种错误。
- 结果截图（n=2^3时）

```
8
A(w0):  real: -10  imag: 0
A(w1):  real: 15.7782  imag: -18.7782
A(w2):  real: 5  imag: 17
A(w3):  real: 0.221825  imag: 3.22183
A(w4):  real: -8  imag: 0
A(w5):  real: 0.221825  imag: -3.22183
A(w6):  real: 5  imag: -17
A(w7):  real: 15.7782  imag: 18.7782
16
```

- 运行时间分析，时间单位为微妙(us)

数据规模n	n*log(n)	time(us)
8	24	58.3
16	64	64.6
32	160	194.8
64	384	290.4
128	896	600.9
256	2048	1834.9



- 算法的理论复杂度为$\Theta(nlgn)$，这是通过主方法求出来的
- 而由实际图像time--n*log(n)也可发现，整个折线成线性关系，所以实际复杂度与理论复杂度一致。

实验总结

- 通过本次实验，我对课上所学的动态规划相关算法有了更深刻的认识，除了亲手实现算法的具体代码以外，也对动态规划的递归实现以及迭代实现(自底向上表格法)都有了更清晰的逻辑理解。
- 而对于FFT，其中对于复数的计算，自己通过对complex类和vector类的使用，极大简化了工作，简洁而高效的达到了算法实现的目的。

参考文档