实验 2 动态规划和FFT

- 实验设备和环境
- 实验 2.1 求矩阵链乘最优方案
  - 实验内容及要求
  - 实验过程（含具体方法和步骤）
  - 实验结果与分析
- 实验 2.2 FFT
  - 实验内容及要求
  - 实验过程（含具体方法和步骤）
  - 实验结果与分析
- 实验总结

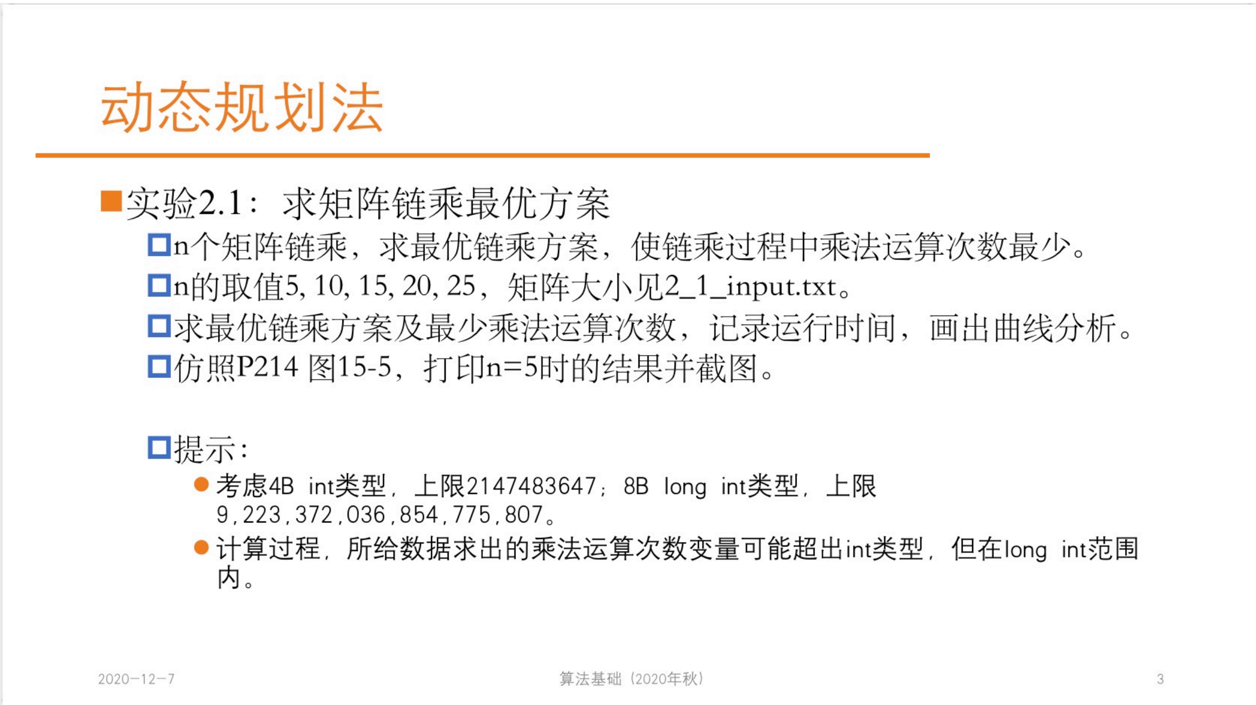
实验 2 动态规划和FFT

实验设备和环境

macOS Mojave 10.14.6 MacBook Pro (Retina, 13-inch, Early 2015) 处理器 2.9 GHz Intel Core i5

实验 2.1 求矩阵链乘最优方案

实验内容及要求



- ■实验2.1 矩阵链乘 输入输出
  - □ex1/input/2_1_input.txt（已给出）：
    - ●每个规模的数据占两行：
      - n
      - 矩阵大小向量$p = (p_0, p_1, \ldots, p_n)$，矩阵$A_i$大小为$p_{i-1} * p_i$
  - □ex1/output/
    - ●result.txt：每个规模的结果占两行
      - 最少乘法运算次数
      - 最优链乘方案（要求输出括号化方案，参考P215 print_opt_parens算法）
    - ●time.txt：每个规模的运行时间占一行
  - □同行数据间用空格隔开

实验过程（含具体方法和步骤）

全局变量有：输入规模 N、矩阵大小 array（设为 long int 就不需要考虑类型转换）、运算次数 m（设为 long int 因为 int 不够用）、最优方案记录 s。

```c
// global variables
long int array[26];
int N;
long int m[25][25];
int s[24][25];
```

参考课本，又考虑了 c 语言中以 0 为起始，实现了本算法中需要的 `MATRIX_CHAIN_ORDER()` 和 `PRINT_OPTIMAL_PARENS()`，如下：

MATRIX_CHAIN_ORDER() 值得一提的是，书中的无穷，可以用 0x7fffffffffffffff 表达。

```c
void MATRIX_CHAIN_ORDER(long int p[], int len){
    int n = len-1;
    int i, l, j, k;
    long int q;
    for(i = 0; i < n; i++)
        m[i][i] = 0;
    for(l = 1; l < n; l++)
        for(i = 0; i<n-l+1; i++){
            j = i+l;
            m[i][j] = 0x7fffffffffffffff; // max
            //printf("%ld\n", m[i][j]);
            for(k = i; k <= j-1; k++){
                q = p[i]*p[k+1]*p[j+1]+m[i][k]+m[k+1][j];
                if(q < m[i][j]){
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    return;
}
```

PRINT_OPTIMAL_PARENS() 将所有打印内容写进 time.txt

```c
void PRINT_OPTIMAL_PARENS(int i, int j, FILE *fp){
    if(i == j)
        fprintf(fp, "A%d", i+1);
    else{
        fprintf(fp, "(");
        PRINT_OPTIMAL_PARENS(i, s[i][j], fp);
        PRINT_OPTIMAL_PARENS(s[i][j]+1, j, fp);
        fprintf(fp, ")");
    }
}
```

核心计时写在了 MATRIX_CHAIN_ORDER() 前后：

```c
start_t = clock();
MATRIX_CHAIN_ORDER(array, N+1);
end_t = clock();
total_t = (double) (end_t - start_t) / CLOCKS_PER_SEC;
```

其他就是一些 I/O 上的问题了，见源代码。

实验结果与分析

- 打印 n=5 时的结果并截图

代码为：

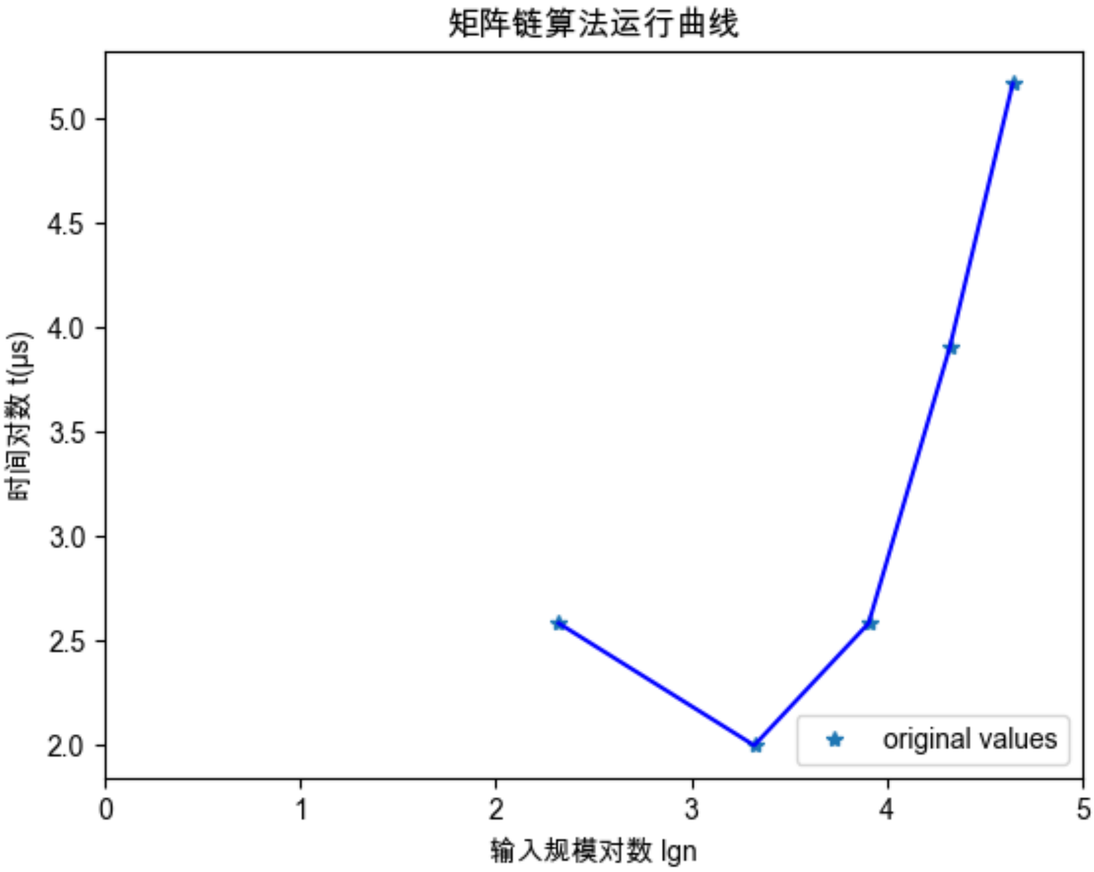
```c
if(N == 5){
    for(int i = 0; i < N; i++) {
        for (int j = i; j < N; j++)
            printf("%ld ", m[i][j]);
        printf("\n");
    }
}
```

打印结果如下：

```
0 15903764653528 74062781976714 128049683226820 154865959097238
0 43981152513978 105723424955724 138766801119366
0 119490227350806 183439291324068
0 120958281818244
0
```

- 比较实际复杂度和理论复杂度是否相同，给出分析。

实际拟合曲线如下：



横坐标为 [5, 10, 15, 20, 25] 的对数，时间为换算成微秒后取的对数。前两个点看起来误差有点大，不妨对后三个点做拟合，计算得到斜率为 3.492，与理论分析到的 O(n) 的复杂度接近。考虑到误差，可以认为实际复杂度和理论复杂度相同。

实验 2.2 FFT

实验内容及要求

FFT

■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2.2 FFT 输入输出

- □ex2/input/2_2_input.txt（已给出）：
  - 每个规模的数据占两行：
    - n
    - 多项式A的系数表示$(a_0, a_1, \ldots, a_{n-1})$
- □ex2/output/
  - result.txt：每个规模的结果占一行
    - DFT结果y的实部
  - time.txt：每个规模的运行时间占一行
- □同行数据间用空格隔开

实验过程（含具体方法和步骤）

核心是对 `RECURSIVE_FFT()` 的实现，要注意什么量必须为复数（即考虑类型转换）

```c
complex double *RECURSIVE_FFT(complex double a[], int n){
    if(n == 1)
        return a;
    complex double w_n = cos(2*PAI/n) + sin(2*PAI/n)*_Complex_I;
    complex double w = 1;
    complex double *a_0 = (complex double *)malloc(sizeof(complex double)*n/2);
    complex double *a_1 = (complex double *)malloc(sizeof(complex double)*n/2);
    for(int i = 0; i < n/2; i++){
        a_0[i] = a[2*i];
        a_1[i] = a[2*i+1];
    }
    complex double *y_0 = RECURSIVE_FFT(a_0, n/2);
    complex double *y_1 = RECURSIVE_FFT(a_1, n/2);
    complex double *y = (complex double *)malloc(sizeof(complex double)*n);
```

```
    for(int k = 0; k < n/2; k++){
        y[k] = y_0[k] + w*y_1[k];
        y[k+n/2] = y_0[k] - w*y_1[k];
        w *= w_n;
    }
    free(a_0);
    free(a_1);
    return y;
}
```

其他就是一些 I/O 上的问题了，见源代码。

实验结果与分析

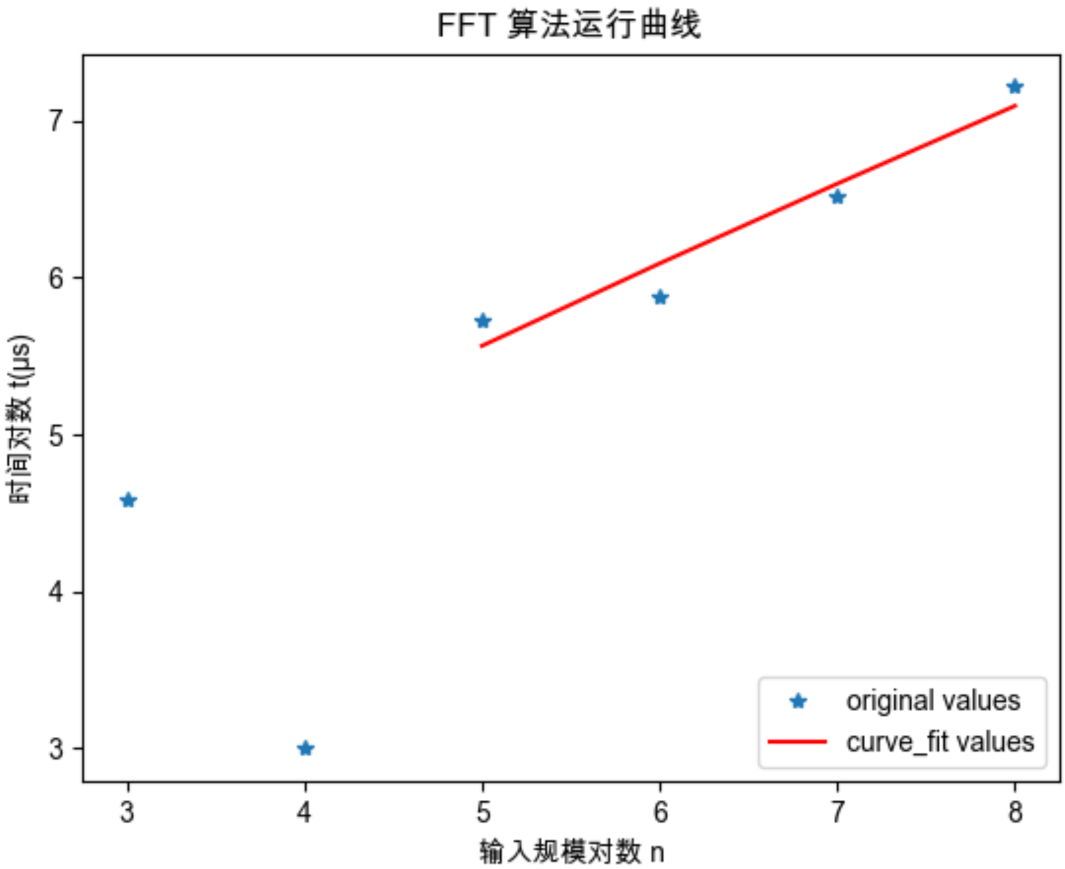
- 打印 n = 2^3 时的结果并截图

打印结果如下：

```
/Users/zengjing/CLionProjects/untitled27/cmake-build-debug/untitled27
n=8, time=0.000028s
-10.000000 15.778175 5.000000 0.221825 -8.000000 0.221825 5.000000 15.778175
n=16, time=0.000011s
n=32, time=0.000023s
n=64, time=0.000052s
n=128, time=0.000090s
n=256, time=0.000184s
```

- 比较实际复杂度和理论复杂度是否相同，给出分析。

实际运行曲线如下：



横坐标为 [8, 16, 32, 64, 128, 256] 的对数，时间为换算成微秒后取的对数。前两个点看起来误差有点大，不妨对后三个点做拟合。拟合思路（截取部分代码）如下：

```
def func(x, a, b):
    return a * (np.array(x) + np.log2(x)) + b
#非线性最小二乘法拟合
popt, pcov = curve_fit(func, x[2:], y[2:])
#获取popt里面是拟合系数
print(popt)
a = popt[0]
b = popt[1]
yvals = func(x[2:], a, b)

plot2 = plt.plot(x[2:], yvals, 'r', label='curve_fit values')
```

考虑到误差，可以认为实际复杂度和理论复杂度相同。

实验总结

动态规划法关于起始点为 0 的处理，让我思考了许久，理解更加深刻了。

FFT 对复数的处理也要考虑到课本没有提及的细节。

相比 lab1，这次实验的 I/O 处理更流畅了。