

中国科学技术大学计算机学院

## 算法基础实验报告

### 实验二

### 动态规划与 FFT

学    号： PB18000203

姓    名： 汪洪韬

专    业： 计算机科学与技术

指导老师： 顾乃杰

中国科学技术大学计算机学院

2020 年 11 月 25 日

## 一、 实验内容

1. 求矩阵链乘的最佳方案；
2. 实现 FFT 算法。

## 二、 实验设备 and 环境

1. 实验设备：PC 机；
2. 实验环境：Visual Studio 2019

## 三、 实验方法和步骤

### 1. 矩阵链乘

源代码：

求矩阵链乘最佳方案子函数：

```
void matrix(int *p, long long *best) {
    int i, j, k, ll;
    int n = size;
    long long q;
    for(i = 1; i ≤ n; i++)
        m[i][i] = 0; // m数组初始化
    for(ll = 2; ll ≤ n; ll++){
        for(i = 1; i ≤ n - ll + 1; i++) { // 将第ll层遍历求值
            j = i + ll - 1;
            m[i][j] = MAX;
            for(k = i; k ≤ j - 1; k++) { // 将当前可能的组合遍历, 找出使得m[i, j]值最小的组合
                q = m[i][k] + m[k + 1][j] + (long long)p[i - 1] * p[k] * p[j];
                if(q < m[i][j])
                {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }
    *best = m[1][size]; // 将最佳结果赋值到best中
}
```

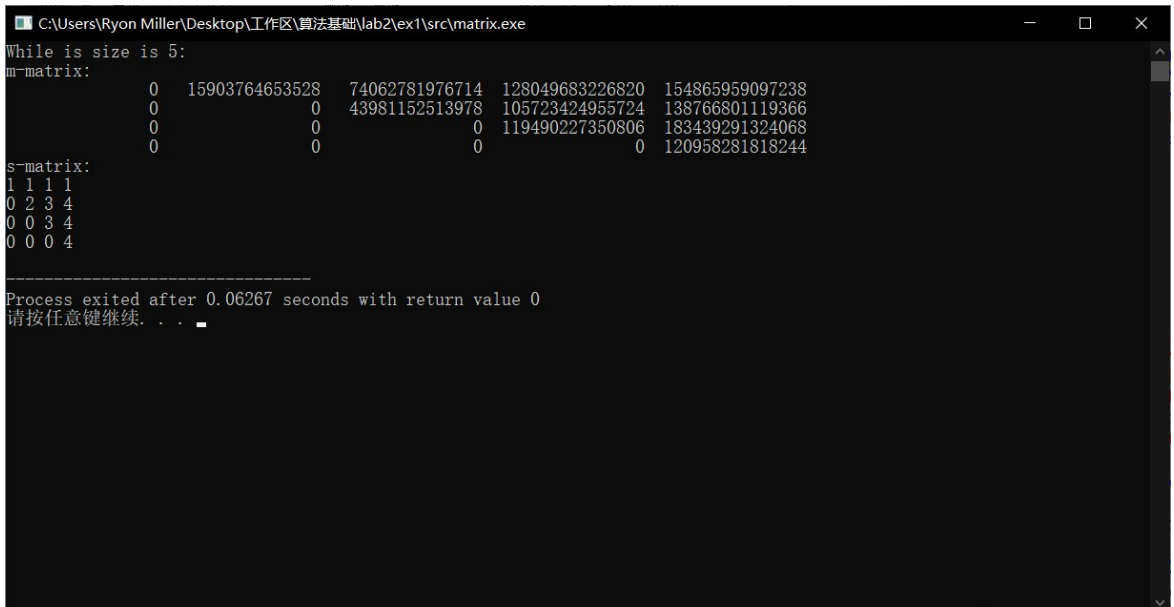
最佳配对输出函数：

```
void print_optimal_parens(FILE* fp, int i, int j) { // 配对输出函数
    if (i == j)
        fprintf(fp, "%d", i);
    else {
        fprintf(fp, "(");
        print_optimal_parens(fp, i, s[i][j]);
        print_optimal_parens(fp, s[i][j] + 1, j);
        fprintf(fp, ")");
    }
}
```

main 函数主体:

```
for (ii = 0; ii < NUM; ii++) {
    fscanf(fp1, "%d", &size);
    for (j = 0; j ≤ size; j++)
        fscanf(fp1, "%d", &a[j]); // 按不同数据规模读入数据
    QueryPerformanceCounter(&time_start); // 开始计时
    matrix(a, &best); // 开始计算
    QueryPerformanceCounter(&time_over); // 结束计时
    if(ii == 0){ // 当n = 5时输出m和s数组
        printf("While is size is 5:\n");
        printf("m-matrix:\n");
        for (j = 1; j < l - 1; j++){
            for(i = 1; i < l; i++){
                printf("%16lld ", m[j][i]);
                printf("\n");
            }
        }
        printf("s-matrix:\n");
        for (j = 1; j < l - 1; j++){
            for(i = 2; i < l; i++){
                printf("%d ", s[j][i]);
                printf("\n");
            }
        }
    }
    run_time = 1000000 * (time_over.QuadPart - time_start.QuadPart) / dqFreq; // 计算所用时间
    fprintf(fp2, "%d matrixes cost %lfus\n", size, run_time);
    fprintf(fp3, "%d matrixes' best solution needs %lld multiplies\n", size, best);
    print_optimal_parens(fp3, 1, size);
    fprintf(fp3, "\n");
}
```

n=5 时的 m 与 s 数组:



```
C:\Users\Ryon Miller\Desktop\工作区\算法基础\lab2\ex1\src\matrix.exe
While is size is 5:
m-matrix:
      0  15903764653528  74062781976714  128049683226820  154865959097238
      0                0  43981152513978  105723424955724  138766801119366
      0                0                0  119490227350806  183439291324068
      0                0                0                0  120958281818244

s-matrix:
1 1 1 1
0 2 3 4
0 0 3 4
0 0 0 4

-----
Process exited after 0.06267 seconds with return value 0
请按任意键继续. . .
```

## 2. FFT

源代码:

求解 DFT(y)的子函数:

```
complex *FFT(int* a, int n) {
    int i;
    if (n == 1) { //当 n = 1时返回当前值
        complex *ca;
        ca = (complex*)malloc(sizeof(complex));
        (*ca).re = *a;
        (*ca).im = 0;
        return ca;
    }
    complex wn, w;
    wn.re = cos(2 * pi / n);
    wn.im = sin(2 * pi / n);
    w.re = 1;
    w.im = 0; //初始化定义
    int* a1, * a0;
    complex* y0, * y1, *y;
    a0 = (int*)malloc((n / 2) * sizeof(int));
    a1 = (int*)malloc((n / 2) * sizeof(int));
    for(i = 0; i < n / 2; i++){
        a0[i] = a[i * 2];
        a1[i] = a[i * 2 + 1];
    } //将系数数组进行分治
    y0 = (complex*)malloc((n / 2) * sizeof(complex));
    y1 = (complex*)malloc((n / 2) * sizeof(complex));
    y = (complex*)malloc(n * sizeof(complex));
    y0 = FFT(a0, n / 2);
    y1 = FFT(a1, n / 2); //分治计算
    for (i = 0; i < n / 2; i++) {
        y[i].re = y0[i].re + w.re * y1[i].re - w.im * y1[i].im;
        y[i].im = y0[i].im + w.re * y1[i].im + w.im * y1[i].re;
        y[i + n / 2].re = y0[i].re - w.re * y1[i].re + w.im * y1[i].im;
        y[i + n / 2].im = y0[i].im - w.re * y1[i].im - w.im * y1[i].re;
        w.re = w.re * wn.re - w.im * wn.im;
        w.im = w.re * wn.im + w.im * wn.re; //依据公式计算, 将分治计算的两部分进行合并
    }
    return y;
}
```

main 函数主体:

```
complex* y;
int* a;
for (ii = 0; ii < NUM; ii++) {
    fscanf(fp1, "%d", &n);
    a = (int*)malloc(n * sizeof(int));
    y = (complex*)malloc(n * sizeof(complex));
    for (j = 0; j < n; j++)
        fscanf(fp1, "%d", &a[j]); //按不同数据规模读入数据
    QueryPerformanceCounter(&time_start); //开始计时
    y = FFT(a, n); //开始计算
    QueryPerformanceCounter(&time_over); //结束计时
    run_time = 1000000 * (time_over.QuadPart - time_start.QuadPart) / dqFreq; //计算所用时间
    if (ii == 0) //当数据规模为2^3时输出
        for (i = 0; i < n; i++)
            printf("A(W%d) = %lf + i * %lf\n", i, y[i].re, y[i].im);
    fprintf(fp2, "%d data size cost %lfus\n", n, run_time);
    for (i = 0; i < n; i++)
        fprintf(fp3, "%lf ", y[i].re); //将y的实部输出
    fprintf(fp3, "\n\n");
}
```

$n=2^3$  时的输出:

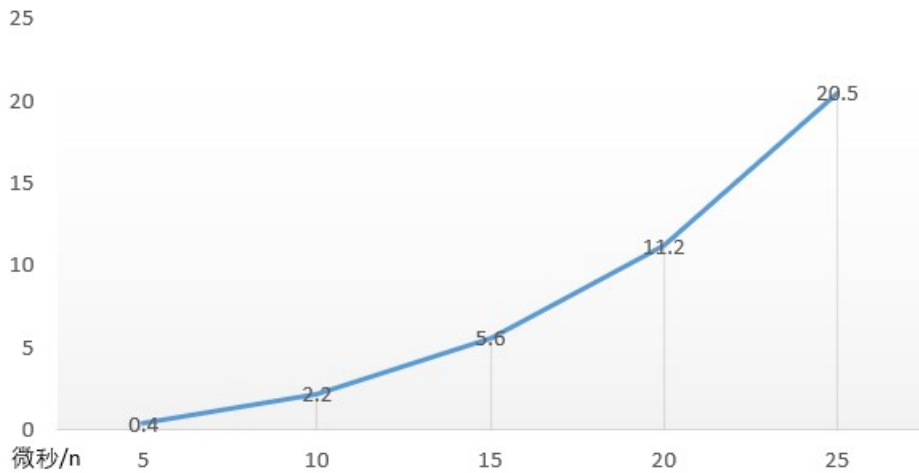
```
C:\Users\Ryon Miller\Desktop\工作区\算法基础\lab2\ex2\src\FFT.exe
A(W0) = -10.000000 + i * 0.000000
A(W1) = 8.000000 + i * -0.000001
A(W2) = 7.489593 + i * 7.770816
A(W3) = 8.000000 + i * 0.000000
A(W4) = -8.000000 + i * 0.000000
A(W5) = 8.000000 + i * 0.000000
A(W6) = 2.510407 + i * -7.770816
A(W7) = 8.000000 + i * 0.000000

-----
Process exited after 0.03367 seconds with return value 0
请按任意键继续. . .
```

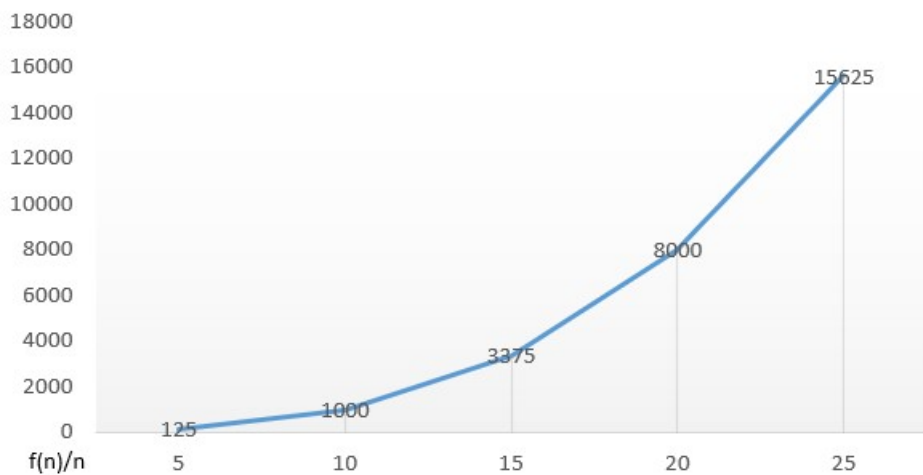
## 四、 实验结果与分析

### 1.矩阵链乘

矩阵链乘数据规模对运行时间的影响

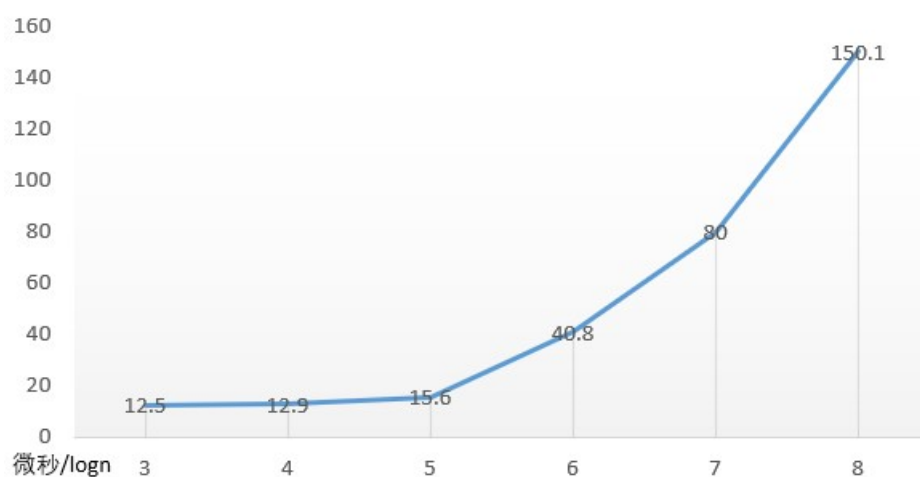


$f(x) = n^3$ 函数趋势

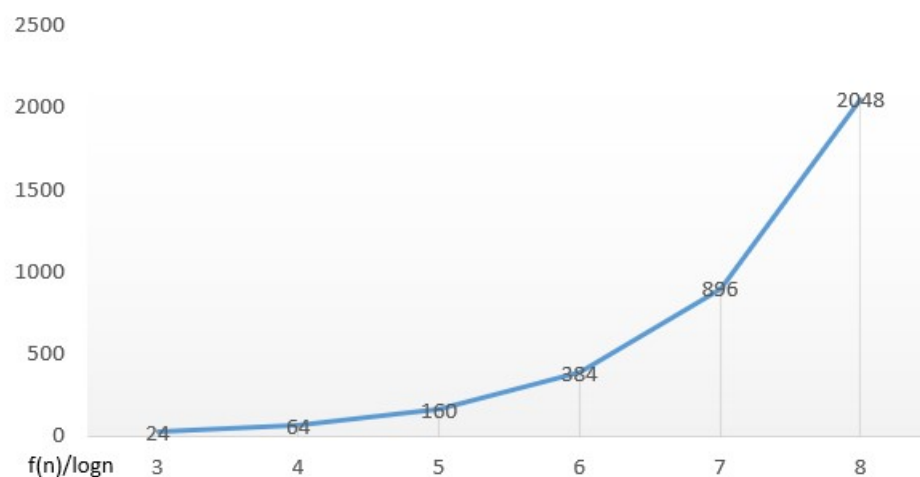


### 2.FFT

### FFT数据规模对运行时间的影响



### $f(x) = n \log n$ 函数趋势



由上图，矩阵链乘算法在不同输入规模下的时间曲线较好的符合  $O(n^3)$ ；FFT 算法在不同输入规模下的时间曲线较好的符合  $O(n \log n)$ 。