

实验二

1. 实验要求

- 1) 实现矩阵链乘算法
- 2) 实现 FTT 算法
- 3) 记录算法运行时间
- 4) 画出算法在不同输入规模下的运行时间曲线图
- 5) 分析算法渐进性能

2. 实验环境

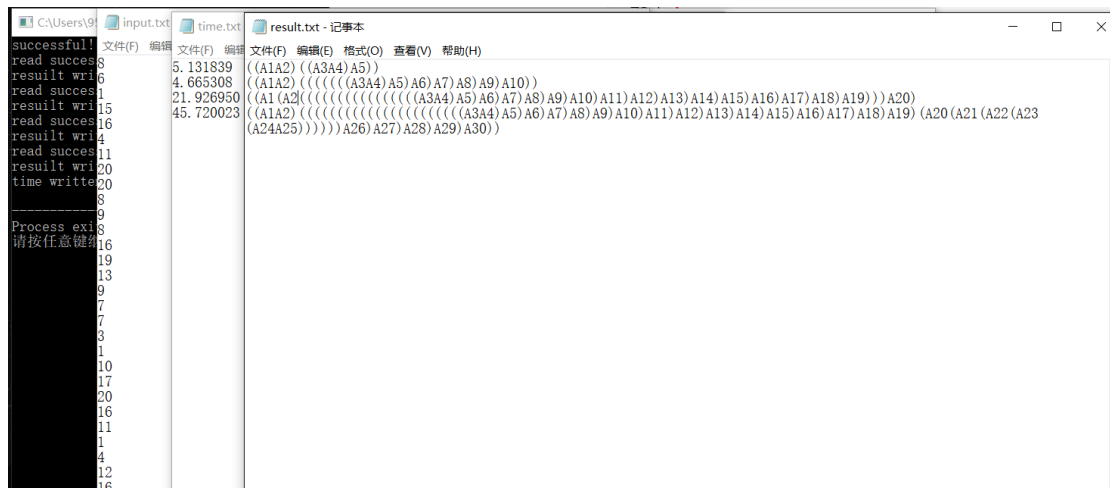
编译环境：Dev C++ 5.11

机器内存：16G

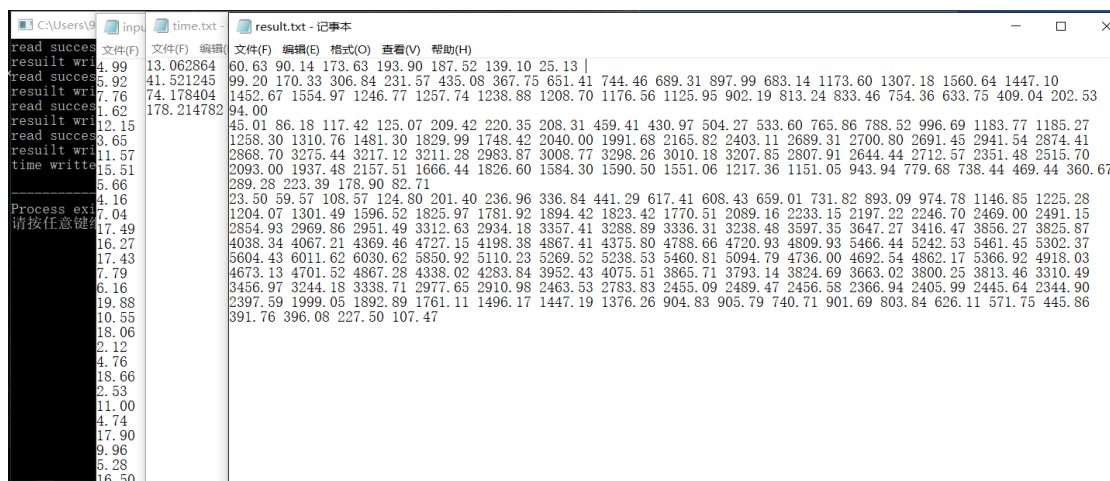
时钟主频：2.20GHz

3. 实验过程

矩阵链乘算法：



FFT 算法；



4. 实验关键代码截图（结合文字说明）

算法导论上的自底向上的矩阵链乘动态规划算法：

```
void MATRIX_CHAIN_ORDER(int p[],int n)
{
    int i,j,k,l,q;
    memset(s,0,sizeof(s));
    for(l = 2;l <= n;l++)
    {
        for(i = 1;i <= n - l + 1;i++)
        {
            j = i + l - 1;
            cost[i][j] = INF;
            for(k = i;k <= j - 1;k++)
            {
                q = cost[i][k] + cost[k + 1][j] + p[i - 1] * p[k] * p[j];
                if(q < cost[i][j])
                {
                    cost[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }
}
```

算法导论上的输出函数，此处用文件写入函数 fprintf 替代 printf。

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

Input.txt 中的随机数生成函数：

```

void Random_Num(void)
{
    int i,m = 40;
    int num[m];
    srand((unsigned)time(NULL)); // 以系统时间作为随机数种子
    for(i = 0;i < m;i++)
    {
        num[i] = 1 + rand()%20;
    }

    FILE *fp;
    if((fp = fopen("../input/input.txt","w"))== NULL)
    {
        printf("error");
    }
    else
    {
        for(i = 0;i < m;i++)
        {
            fprintf(fp,"%d\n",num[i]);
        }
        printf("successful!\n");
        fclose(fp);
    }
}

```

记录算法运行时间的工具:

```

#include "windows.h"
LARGE_INTEGER nFreq; // 时钟频率
LARGE_INTEGER t1; // 开始
LARGE_INTEGER t2; // 结束
double dt[6]; // 时间差
QueryPerformanceFrequency(&nFreq); // 获取CPU频率
QueryPerformanceCounter(&t1); // 开始时间
INSERTION_SORT(str,m);
QueryPerformanceCounter(&t2); // 结束时间
dt[count] = (t2.QuadPart - t1.QuadPart) / (double)nFreq.QuadPart * 1000000;

```

为了支持虚数的计算，引入结构体 complex 表示虚数:

```

struct Complex
{
    double real; // 实部
    double imag; // 虚部
}; // complex

```

```
Complex A[MAX_SIZE], B[MAX_SIZE], C[MAX_SIZE], w[MAX_SIZE];
```

为表示方便，引入重载定义复数的四则运算:

//以下为重载定义复数的四则运算和虚部取反

Complex operator+(Complex a,Complex b)

```
{  
    Complex r;  
    r.real = a.real + b.real;  
    r.imag = a.imag + b.imag;  
    return r;  
} //plus
```

Complex operator-(Complex a,Complex b)

```
{  
    Complex r;  
    r.real = a.real - b.real;  
    r.imag = a.imag - b.imag;  
    return r;  
} //minus
```

Reverse 重新排列输入数组元素的下标:

void Reverse(int* id,int size,int m)

```
{  
    for(int i = 0;i < size;i++)  
    {  
        for(int j = 0;j < (m + 1) / 2;j++)  
        {  
            int v1 = (1 << (j) & i) << (m - 2 * j - 1);  
            int v2 = (1 << (m - j - 1) & i) >> (m - 2 * j - 1);  
            id[i] |= (v1 | v2);  
        }  
    }  
}; // 重新排列输入数组的元素下标
```

计算 w:

void Compute_W(Complex w[],int size)

```
{  
    for(int i = 0;i < size/2;i++)  
    {  
        w[i].real = cos(2 * PI * i/size);  
        w[i].imag = sin(2 * PI * i/size);  
        w[i + size/2].real = -w[i].real;  
        w[i + size/2].imag = -w[i].imag;  
    }  
}; // 预先计算FFT中需要的w值
```

FFT:

```

int* id = new int[size];
memset(id,0,sizeof(int)*size);
int m = log2((double) size);
Reverse(id,size,m);
Complex *resort = new Complex[size];
memset(resort,0,sizeof(Complex)*size);
int i,j,k,s;
for(i = 0;i < size;i++)
    resort[i] = in[id[i]];
for(i = 1;i <= m;i++){
    s = (int) pow((double)2,(double)i);
    for(j = 0;j < size/s;j++){
        for(k = j * s;k < j * s + s/2;k++){
            Complex k1 = resort[k] + w[size/s * (k - j * s)] * resort[k + s/2];
            resort[k + s/2] = resort[k] - w[size/s * (k - j * s)] * resort[k + s/2];
            resort[k] = k1;
        }
    }
}
for(i = 0;i < size;i++)
    in[i] = resort[i];
delete[] id;

```

IFFT:

```

int* id = new int[size];
memset(id, 0, sizeof(int)*size);
int m = log2((double) size);
Reverse(id,size,m);
Complex *resort = new Complex[size];
memset( resort, 0, sizeof(Complex)*size);
int i,j,k,s;
for(i = 0;i < size;i++)
    resort[i] = in[id[i]];
for(i = 1;i <= m;i++){
    s = (int) pow((double)2,(double)i);
    for(j = 0;j < size/s;j++){
        for(k = j*s;k < j * s + s/2;k++){
            Complex k1 = (resort[k] + (~w[size/s * (k - j * s)]) * resort[k + s/2]);
            resort[k + s/2] = (resort[k] - (~w[size/s * (k - j * s)]) * resort[k + s/2]);
            resort[k] = k1;
        }
    }
}
for(i = 0;i < size;i++)
    in[i] = resort[i]/size;
delete[] id;

```

处理 m 为非 2 的整数幂:

```

int m1 = pow(2,ceil(log2(2*m))); //处理非2的整数次幂

```

利用 FFT 进行多项式乘法计算以及记录运行时间:

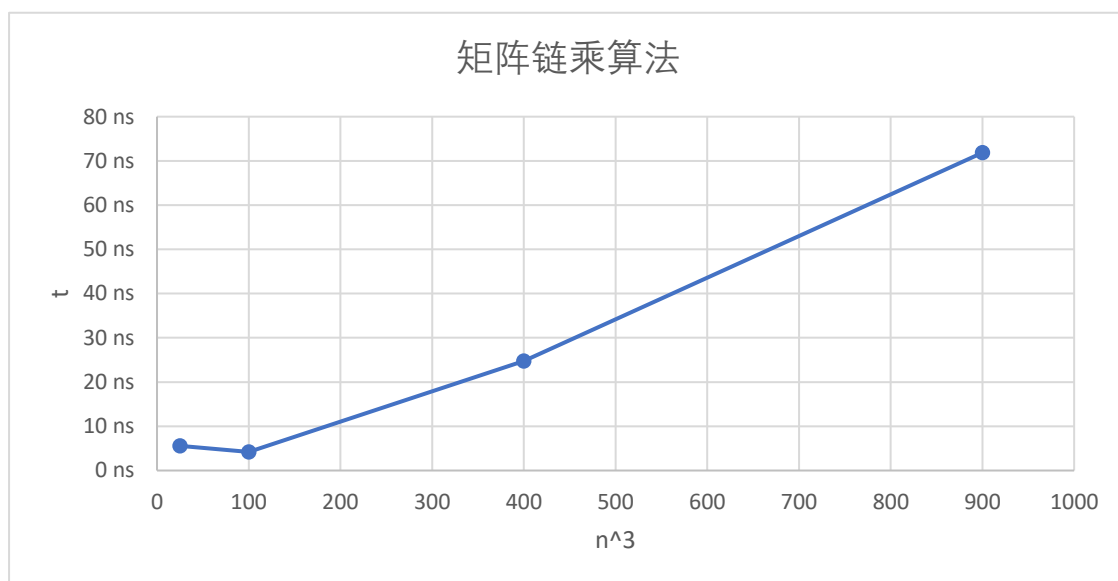
```

QueryPerformanceCounter(&t1);
Compute_W(w,m1);
FFT(A,m1);
FFT(B,m1);
for(i = 0;i < m1;i++)
    C[i] = A[i] * B[i];
IFFT(C,m1);
QueryPerformanceCounter(&t2);

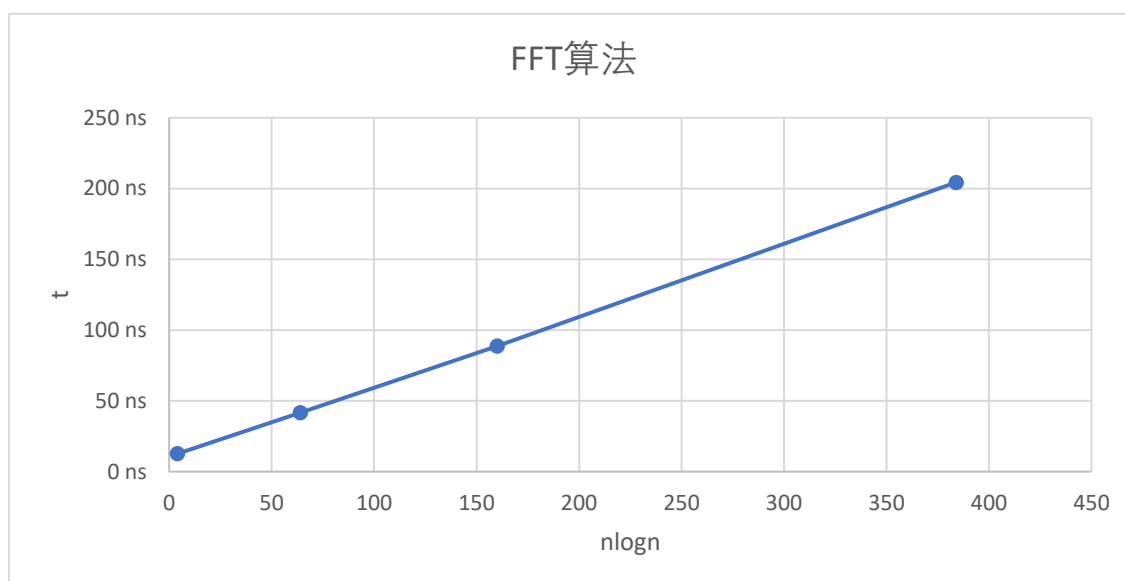
dt[count] = (t2.QuadPart - t1.QuadPart) / (double)nFreq.QuadPart * 1000000;

```

5. 实验结果、分析（结合相关数据图表分析）



图中已对 n 作了三次方处理，可以看出 t 与 n^3 基本呈线性关系，与算法导论中矩阵链乘算法 $\Omega(n^3)$ 的论断一致。 $n=100$ 时的反常点应该是与矩阵链乘算法需要额外 $\Theta(n^2)$ 内存空间来保存表 $cost$ 和 s 有关。



图中已对 n 作了 $n \log n$ 处理（其中 $n=60$ 时，取 $n=64$ ），可以看出 t 与 $n \log n$ 基本呈线性关系，与算法导论 $\Theta(n \log n)$ 的论断一致。

6. 实验心得

通过对矩阵链乘算法实现，我较清楚地理解了动态规划的基本原理，了解了动态规划的一般步骤，并对代码的优化有了一定的认识。通过对 FFT 算法的实现，我认识到了数学与计算机科学的紧密联系，并且复习了相关的数学知

识。通过对两个算法时间渐进性的分析进一步学习了怎样测量程序运行时间以及图表的制作。