实验 2　算法设计策略

Pb15111604
金泽文

1. 实验要求

- ex1：实现求矩阵链乘问题的算法。对 n 的取值分别为：　5、10、20、30 ，随机生成 n+1 个整数值（ p_0、p_1、…、p_n ）代表矩阵的规模，其中第 i 个矩阵(1≤i ≤ n)的规模为 $p_{i-1} \times p_i$ ，用动态规划法求出矩阵链乘问题的最优乘法次序，统计算法运行所需时间 ，画出时间曲线。

- ex2：实现 FFT 算法，对 n 的取值分别为 4、16、32、60(注意当 n 取值不为 2 的整数幂时的处理方法)，随机生成 2n 个实数值（a_0、a_1、…、a_{n-1}）和（b_0、b_1、…、b_{n-1}）分别作为多项式 A(x)和 B(x)的系数向量，使用 FFT 计算多项式 A(x)与多项式 B(x)的乘积，统计算法运行所需时间 ，与普通乘法进行比较，画出时间曲线。

2. 实验环境

编译环境：

编程语言：Java SE8
机器内存：16G
时钟主频：2.3GHz

为了便于检查时可以在 Linux 等环境下编译运行，可以在源文件中将 CHECKMODE 设置为 true，以便找到正确路径。

3. 实验过程

1. 生成随机数
   a) Ex1：
      i. 生成 31 个(int)(Math.random()*32767 + 1)到 input.txt 中。
   b) Ex2：
      i. 生成 120 个(int)(Math.random()*32767 + 1)到 input.txt 中。（实验中为了观察 n 比较大时两个算法的时间，我在这里生成了 65536 个系数。）
2. Ex1 动态规划求最优矩阵相乘次序：
   i. 输入：
      用 Scanner 依次读入 int 数据，存入 p 数组。
   ii. 动态规划算法的实现：
      创建二维表 s，m：s[i][j]用来存储 Ai..Aj 相乘所应选择的 p；m[i][j]用来存储 Ai..Aj 的最小代价。剩下的与教材中的伪代码相似。唯一需要注意的是数组的下标。
   iii. 输出：
      输出的串通过 ii.中得到的 s 二维表递归得到。为了满足实验的要求，通过正则表达式的匹配与子串的替换，得到更加美观的表示。
   iv. 检验正确性：
      通过教材上的例子：

矩阵	A_1	A_2	A_3	A_4	A_5	A_6
规模	30×35	35×15	15×5	5×10	10×20	20×25

得到的输出：

```
number of tasks: 6
        result: ((A1,(A2,A3)),((A4,A5),A6))
number of tasks: 6
        time: 20888 nanoseconds.
```

教材上的结果：

$$((A_1(A_2A_3))((A_4A_5)A_6))$$

结果正确。

3. Ex2 多项式相乘的 FFT 实现:
   i. 输入：
      用 Scanner 依次读入 int 数据，前 n 个作为 a 数组系数，后面的 n 个作为 b 数组的系数。
      同时扩充 a,b，以满足 2 的幂，再乘以二。
   ii. FFT 乘法的实现：
      通过教材中的卷积定理（如下图），实现多项式的 FFT 乘法。
      通过 `iterativeFFT` 函数得到 `DFT`，通过 `plotMult` 点乘

若此。

定理 30.8(卷积定理) 对任意两个长度为 n 的向量 a 和 b，其中 n 是 2 的幂，

$$a \otimes b = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a) \cdot \mathrm{DFT}_{2n}(b))$$

其中向量 a 和 b 用 0 填充，使其长度达到 $2n$，并用"."表示 2 个 $2n$ 个元素组成向量的点乘。

再通过 revFFT 得到逆 DFT 的结果。

iii. 普通乘法的实现

iv. 输出

v. 检验正确性：

通过比对两种实现得到的系数数组，再通过网上的在线多项式相乘计算器比对，检验正确。

4. 制图

4. 实验关键代码截图（结合文字说明）

最后的目录结构：

```
Reaper@KZ:/mnt/d/USTC/algorithm/PB15111604-project2$ tree
.
├── ex1
│   ├── input
│   │   └── input.txt
│   ├── output
│   │   ├── result.txt
│   │   └── time.txt
│   └── src
│       ├── DynamicProgramming.java
│       └── GenerateIntegers.java
├── ex2
│   ├── input
│   │   └── input.txt
│   ├── output
│   │   ├── result.txt
│   │   └── time.txt
│   └── src
│       ├── GenerateDoubles.java
│       └── PolynomialMultiplication.java
└── PB15111604-金泽文-project2.doc
```

Ex1 动态规划求最优矩阵相乘次序：

a) 代码框架：

大致框架如下。

```
13  public class DynamicProgramming {
14
15      public static final boolean CHECKMODE = false;
16
17      private static int[] p;
18      private static int[][] m;
19      private static int[][] s;
20      private static long start;
21      private static long end;
22      private static long endurance;
23      private static PrintWriter printResult;
24      private static PrintWriter printTime;
25      private static Scanner inputIntegers;
26
27      public static void main(String[] args)throws IOException{...}
59
60      // 输入
61      public static void input()throws IOException{...}
67
68      // 输出
69      public static void output(int n)throws IOException{...}
91
92      // 动态规划算法实现
93      public static void matrix(int[] p){...}
111
112     // 递归生成次序串
113 @   private static String printResultAug(int i, int j){...}
123 }
124
```

a) main：main 函数首先负责初始化：输入输出文件初始化，任务序列：5,10,20,30,的初始化。然后负责迭代处理：针对每一个长度，计时，运行 matrix 函数，结束计时，输出。

```
    public static void main(String[] args)throws IOException{
        // 初始化
        if(CHECKMODE){
            inputIntegers = new Scanner(Paths.get( first: "../input/input.txt"), charsetName: "utf-8");
            input();
            printResult = new PrintWriter( fileName: "../output/result.txt", csn: "utf-8");
            printTime = new PrintWriter( fileName: "../output/time.txt", csn: "utf-8");
        }
        else {
            inputIntegers = new Scanner(Paths.get( first: "D:/USTC/algorithm/PB15111604-project2/ex1/input/input.txt"),
 charsetName: "utf-8");
            input();
            printResult = new PrintWriter( fileName: "D:/USTC/algorithm/PB15111604-project2/ex1/output/result.txt",
 csn: "utf-8");
            printTime = new PrintWriter( fileName: "D:/USTC/algorithm/PB15111604-project2/ex1/output/time.txt",
 csn: "utf-8");
        }

        int[] tasks = {30, 20, 10, 5};
//      int[] tasks = {6};
```

```
//          int[] tasks = {6};
            // 迭代处理
            for(int task : tasks){
                int[] tmpChain = Arrays.copyOf(p, newLength: task+1);
//                int[] tmpChain = {30, 35, 15, 5, 10, 20, 25};
                // 开始计时
                start = System.nanoTime();
                // 运行
                matrix(tmpChain);
                // 结束计时
                end = System.nanoTime();
                endurance = end - start;
                // 输出
                output(task);
            }
        }
```

b) input：内容简短。

```
p = new int[31];
for(int i = 0; i < 31; i++){
    p[i] = inputIntegers.nextInt();
}
```

c) output：首先根据 printResultAug 函数得到得到形如

(((A1,A2),A3),A4),A5)

的串。再根据正则表达式的匹配与子串的替换，得到更美观的输出。

```
        // 得到形如(((A1,A2),A3),A4),A5)的串
        String result = printResultAug( i: 1, n);
        // 用正则表达式匹配替换，以使输出更加美观
        result = result.replaceAll( regex: "(\\d+)(A)", replacement: "$1,$2");
        result = result.replaceAll( regex: "(\\))(\\()", replacement: "$1,$2");
        result = result.replaceAll( regex: "(\\))(A)", replacement: "$1,$2");
        result = result.replaceAll( regex: "(A)(\\()", replacement: "$1,$2");
        result = result.replaceAll( regex: "(\\d+)(\\()", replacement: "$1,$2");
```

再依次输出到 output 文件中。

d) printResultAug：类似于教材中的 PRINT-OPTIMAL-PARENS 函数

```
112         // 递归生成次序串
113 @       private static String printResultAug(int i, int j){
114             if( i == j ){
115                 return "A"+i;
116             }
117             else{
118                 String l = printResultAug(i, s[i][j]);
119                 String r = printResultAug( i: s[i][j]+1, j);
120                 return "("+l+r+")";
121             }
122         }
123     }
```

e) matrix：类似于教材中的 MATRIX-CHAIN-ORDER 函数。创建二维表 s，m：s[i][j]用来存储 Ai..Aj 相乘所应选择的 p；m[i][j]用来存储 Ai..Aj 的最小代价。剩下的与教材中的伪代码相似。唯一需要注意的是数组的下标。

```
92      // 动态规划算法实现
93      public static void matrix(int[] p){
94          int n = p.length - 1;
95          m = new int[n+1][n+1];
96          s = new int[n][n+1];
97          for(int l = 2; l <= n; l++){
98              for(int i = 1; i <= n-l+1; i++){
99                  int j = i + l - 1;
100                 m[i][j] = 2_147_483_647; //max
101                 for(int k = i; k <= j-1; k++){
102                     int q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
103                     if (q < m[i][j]){
104                         m[i][j] = q;
105                         s[i][j] = k;
106                     }
107                 }
108             }
109         }
110     }
111
```

2. Ex2 多项式相乘的 FFT 实现:

a) 代码框架：

代码架构如下：

GenerateDoubles.java | PolynomialMultiplication.java | input.txt

```
/**...*/

import ...

public class PolynomialMultiplication {

...

    public static void main(String[] args)throws IOException{...}

    // 初始化
    private static void init()throws IOException{...}

    // 输入
    private static void input()throws IOException{...}

    // 输出
    private static void output(int len_coef)throws IOException{...}

    // 为了保证2的幂,扩展为len
    // len -- 输出，为2的幂
    private static int fix(int n){...}

    // FFT相乘的框架
    // a,b -- 源数组
    // lenResult -- 最后输出所对应的数组长度
    //              ,为了避免扩充的影响而设置
    private static void fftMul(double[] a, double[] b, int lenResult)
{...}

    // 递归FFT
    // a -- 源数组
    // A -- 目的数组
    private static void iterativeFFT(double[] a, Complex[] A){...}

    // 针对FFT的逆序置换
    // a -- 源数组
    // A -- 目的数组
    // bits -- 位数
```

PolynomialMultiplication

PolynomialMultiplication.java

```
    private static void bitReverseCopy(double[]a, Complex[] A, int
bits){...}

    // 针对FFT的逆的逆序置换
    // a -- 源数组
    // A -- 目的数组
    // bits -- 位数
    private static void bitReverseCopy(Complex[]C, Complex[] c, int
bits){...}

    // 位的逆置换。(书上说的很"简单"的部分)
    // data -- 要置换的数
    // bits -- 数的位数
    private static int bitRev(int data, int bits){...}

    // 逐点相乘
    // C -- 结果存放
    // A,B -- 乘数数组
    private static void plotMult(Complex[] C, Complex[] A, Complex[] B)
{...}

    // FFT的逆
    // C -- 操作数
    // c -- 结果放在c
    private static void revFFT(Complex[] C){...}

    // 普通的多项式乘法
    // a,b -- 相乘数组
    // n -- 未扩充时a的系数个数
    private static void normalMul(double[] a, double[] b, int n){...}

    // 用于生成所要输出的系数字符串,
    // 形如(C0,C1,C2)
    private static String printResultAug(){...}
}

// 复数类
// 实现了简单的加减乘运算.
class Complex {...}
```

PolynomialMultiplication

b) 变量：

如下图所示，注释部分清晰易懂。

```
public class PolynomialMultiplication {

    public static final boolean CHECKMODE = false;
    // 如果要在自己的机器上跑，请置为true

    private static double[] originalData;   // 存放原始数据
    private static double[] a;              // 存放扩充的系数数组a
    private static double[] b;              // 存放扩充的系数数组a
    private static Complex[] c;             // 存放结果数组c
    private static Complex[] A;             // a的DFT
    private static Complex[] B;             // b的DFT
    private static Complex[] C;             // a,b点乘之后再逆DFT处理
    private static double[] Normal;         // 存放普通乘法的结果
    private static boolean ifFFT;           // 用于输出的标志
    private static long start;              // 计时开始
    private static long end;                // 计时结束
    private static long endurance;          // 时长
    private static PrintWriter printResult; // result.txt
    private static PrintWriter printTime;   // time.txt
    private static Scanner inputDoubles;    // input.txt
    public static final int MAX = 65536;    // 为了观察更大的n的效果
```

c) Main 函数：

先 init 初始化，再迭代处理：对于系数长度 n，依次计算普通乘法与 FFT 乘法，输出。

```
37      public static void main(String[] args)throws IOException{
38          // 初始化
39          init();
40
41  //          int[] coefficients = {MAX/2, MAX/4, MAX/16, MAX/32, 60,  32, 16, 4};      // 为了观察n比较大时，fft与普通乘法的效率。
42          int[] coefficients = {60,  32, 16, 4};
43
44          // 迭代处理
45          for(int len_coef : coefficients){
46              int len = 2*fix(len_coef);                                  // 为了保证2的幂，同时扩展为2n
47              a = Arrays.copyOfRange(originalData,  from: 0, len);
48              b = Arrays.copyOfRange(originalData, len_coef,  to: len_coef+len);
49
50              for(int i = len_coef; i < len; i++)
51                  a[i] = b[i] = 0;
52
53              start = System.nanoTime();                              // 开始计时
54              normalMul(a, b, len_coef);                                  // 运行
55              end = System.nanoTime();                                // 结束计时
56              endurance = end - start;
57              output(len_coef);                                           // 输出
58
59              start = System.nanoTime();                              // 开始计时
60              fftMul(a, b,  lenResult: len_coef * 2 - 1);                 // 运行
61              end = System.nanoTime();                                // 结束计时
62              endurance = end - start;
63              output(len_coef);                                           // 输出
64          }
65      }
66
```

d) 初始化：

根据是否检查模式来选择路径。初始化输入输出文件。

```
67      // 初始化
68      private static void init()throws IOException{
69          // 根据是否为检查模式，选择适合的路径
70          if(CHECKMODE){
71              // 检查模式，非IntelliJ Idea
72              inputDoubles = new Scanner(Paths.get( first: "../input/input.txt"),  charsetName: "utf-8");
73              input();
74              printResult = new PrintWriter( fileName: "../output/result.txt",  csn: "utf-8");
75              printTime = new PrintWriter( fileName: "../output/time.txt",  csn: "utf-8");
76          }
77
78          else{
79              // 非检查模式，IntelliJ Idea
80              inputDoubles = new Scanner(Paths.get( first: "D:/USTC/algorithm/PB15111604-project2/ex2/input/input.txt"),  charsetName: "utf-8");
81              input();
82              printResult = new PrintWriter( fileName: "D:/USTC/algorithm/PB15111604-project2/ex2/output/result.txt",  csn: "utf-8");
83              printTime = new PrintWriter( fileName: "D:/USTC/algorithm/PB15111604-project2/ex2/output/time.txt",  csn: "utf-8");
84          }
85      }
```

e) 输入：

```
82      // 输入
83      private static void input()throws IOException{
84          originalData = new double[MAX];
85          for(int i = 0; i < MAX; i++) {
86              originalData[i] = inputDoubles.nextDouble();
87          }
88      }
```

f) 输出：

通过 output 函数：

```
96      // 输出
97      private static void output(int len_coef)throws IOException{
98          int n = len_coef;
99          String result = printResultAug();
100         String typeMul = ifFFT ? "FFT   " : "Normal";
101
102         System.out.println("number of coefficients: " + n);
103         printResult.println("number of coefficients: " + n);
104         System.out.println("    "+ typeMul + "\tresult: " + result);
105         printResult.println("    "+ typeMul + "\tresult: " + result);
106         printResult.flush();
107         System.out.println("number of coefficients: " + n);
108         printTime.println("number of coefficients: " + n);
109         System.out.println("    "+ typeMul + "\ttime: " + endurance  + "\tnanoseconds.");
110         printTime.println("    "+ typeMul + "\ttime: " + endurance  + "\tnanoseconds.");
111         printTime.flush();
112     }
```

以及 printResultAug 函数输出。

这里需要说明的是为了方便比较实数，对 double 的小数部分进行了取舍，只保留 5 位小数。

```
274     // 用于生成所要输出的系数字符串，
275     // 形如(C0,C1,C2)
276     private static String printResultAug(){
277         String s = "(";
278         int i;
279
280         DecimalFormat df = new DecimalFormat( pattern: "################################.00000");
281         if(ifFFT){
282             df.setRoundingMode(RoundingMode.CEILING);
283             for(i = 0; i < c.length-1; i++){
284                 s += df.format(c[i].r) + ",";
285             }
286             s += df.format(c[i].r);
287         }
288         else{
289             for(i = 0; i < Normal.length-1; i++){
290                 s += df.format(Normal[i]) + ",";
291             }
292             s += df.format(Normal[i]);
293         }
294         s += ")";
295         return s;
296     }
297 }
```

g) FFT 乘法：

通过 fftMul 函数实现 FFT 乘法。

DFT 的过程由 iterativeFFT 函数实现

A=DFT（a）

B=DFT（b）

之后通过点乘运算存储到 C 中

C=A • B

再通过 revFFT 函数实现 DFT 的逆运算。

c=revDFT（C）。

其中，revDFT 的实现与 iterativeFFT 基本相似，只需要修改 sin 为-sin，在最后除以 n 即可。

代码截图如下：

```java
    // FFT相乘的框架
    // a,b -- 源数组
    // lenResult -- 最后输出所对应的数组长度
    //              ,为了避免扩充的影响而设置
    private static void fftMul(double[] a, double[] b, int lenResult){
        int n = a.length;
        A = new Complex[n];
        B = new Complex[n];
        for(int i = 0; i < n; i ++){
            A[i] = new Complex();
            B[i] = new Complex();
        }
        C = new Complex[n];
        iterativeFFT(a, A);
        iterativeFFT(b, B);
        plotMult(C, A, B);
        revFFT(C);
        c = Arrays.copyOfRange(c,  from: 0, lenResult);
        ifFFT = true;
    }
```

```java
    // 迭代FFT
    // a -- 源数组
    // A -- 目的数组
    private static void iterativeFFT(double[] a, Complex[] A){
        int n = a.length;
        int log = (int)Math.round(Math.log(n)/Math.log(2));

        bitReverseCopy(a, A, log);

        Complex wm = new Complex(), w = new Complex(), t = new Complex(), u = new Complex();
        for(int s = 2; s <= n; s <<= 1){
            int m = s;
            wm.r = Math.cos(2 * Math.PI / m);
            wm.i = Math.sin(2 * Math.PI / m);
            for(int k = 0; k < n; k += m){
                w.set(1, 0);
                for(int j = 0; j < m/2; j++){
                    t.set(w.mul(A[k+j+m/2]));
                    u.set(A[k+j]);
                    A[k+j] = u.add(t);
                    A[k+j+m/2] = u.minus(t);
                    w.set(w.mul(wm));
                }
            }
        }
    }
```

```java
    // FFT的逆
    // C -- 操作数
    // c -- 结果放在c
    private static void revFFT(Complex[] C){
        int n = C.length;
        c = new Complex[n];
        int log = (int)Math.round(Math.log(n)/Math.log(2));

        bitReverseCopy(C, c, log);

        Complex wm = new Complex(), w = new Complex(), t = new Complex(), u = new Complex();
        for(int s = 2; s <= n; s <<= 1){
            int m = s;
            wm.r = Math.cos(2 * Math.PI / m);
            wm.i = - Math.sin(2 * Math.PI / m);
            for(int k = 0; k < n; k += m){
                w.set(1, 0);
                for(int j = 0; j < m/2; j++){
                    t.set(w.mul(c[k+j+m/2]));
                    u.set(c[k+j]);
                    c[k+j] = u.add(t);
                    c[k+j+m/2] = u.minus(t);
                    w.set(w.mul(wm));
                }
            }
        }
        for(int i = 0; i < n; i++)
            c[i].r /= n;
    }
```

```java
    // 针对FFT的逆的逆序置换
    // a -- 源数组
    // A -- 目的数组
    // bits -- 位数
    private static void bitReverseCopy(Complex[]C, Complex[] c, int bits){
        int n = c.length;
        for(int i = 0; i < n; i++)
            c[bitRev(i, bits)] = C[i];
    }

    // 位的逆置换。(书上说的很“简单”的部分)
    // data -- 要置换的数
    // bits -- 数的位数
    private static int bitRev(int data, int bits){
        int low = 0;
        int high = bits-1;
        int bit_low;
        int bit_high;
        while(low < high){
            bit_low = data & (1<<low);
            bit_high = data & (1<<high);
            if(bit_low == 0){
                data &= ~(1 << high);                // set 0
            }
            else{
                data |= (1 << high);                 // set 1
            }
            if(bit_high == 0){
                data &= ~(1 << low);                // set 0
            }
            else{
                data |= (1 << low);                 // set 1
            }
            low ++;
            high --;
        }
        return data;
    }

    // 逐点相乘
    // C -- 结果存放
    // A,B -- 乘数数组
    private static void plotMult(Complex[] C, Complex[] A, Complex[] B){
        int n = A.length;
        for(int i = 0; i < n; i++){
            C[i] = new Complex(A[i].mul(B[i]));
        }
    }
```

h) 普通乘法：

```java
    // 普通的多项式乘法
    // a,b -- 相乘数组
    // n -- 未扩充时a的系数个数
    private static void normalMul(double[] a, double[] b, int n){
        int len = 2*n-1;
        Normal = new double[len];
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                Normal[i + j] += a[i] * b[j];
            }
        }
        ifFFT = false;
    }
```

i) Complex 类的实现：

实现了简单的复数加减乘运算。

```
298  // 复数类
299  // 实现了简单的加减乘运算.
300  class Complex {
301      public double r;            // 实部
302      public double i;            // 虚部
303      public Complex(double rr, double ii){
304          r = rr;
305          i = ii;
306      }
307      public Complex(){
308          r = 0;
309          i = 0;
310      }
311
312      public Complex(Complex c) {
313          r = c.r;
314          i = c.i;
315      }
316
317      public void set(double rr, double ii){
318          r = rr;
319          i = ii;
320      }
321
322      public void set(Complex c){
323          r = c.r;
324          i = c.i;
325      }
326
327      public Complex add(Complex c){
328          return new Complex( rr: r + c.r,  ii: i + c.i);
329      }
330
331      public Complex minus(Complex c){
332          return new Complex( rr: r - c.r,  ii: i - c.i);
333      }
334
335      public Complex mul(Complex c){
336          double tr = r*c.r - i*c.i;
337          double ti = i*c.r + r*c.i;
338          return new Complex(tr, ti);
339      }
340  }
```

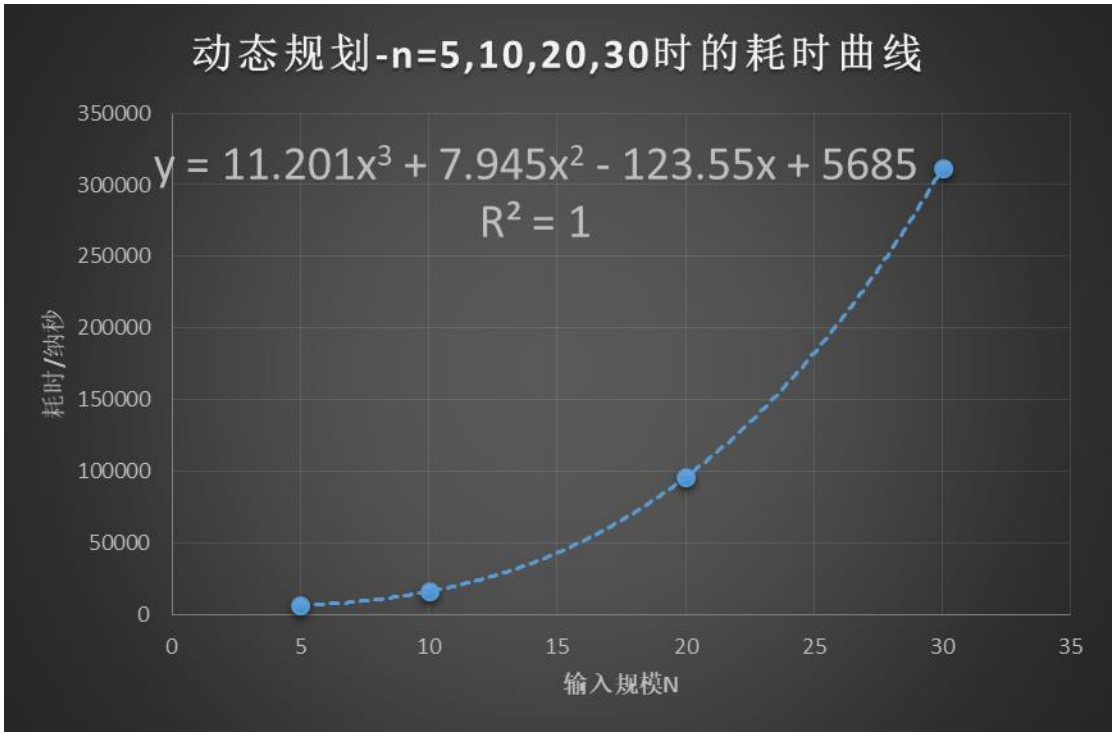
5. 实验结果、分析（结合相关数据图表分析）

a) Ex1 动态规划求最优矩阵相乘次序：

根据 time.txt 得到表格：

规模 n	5	10	20	30
耗时/ns	6666	16445	96000	311556

根据 Excel 得到如下拟合曲线：



从图中可以看出，虽然数据很少，但是拟合效果十分好，完全符合书中所说 $\Theta(n^3)$ 的时间复杂度。

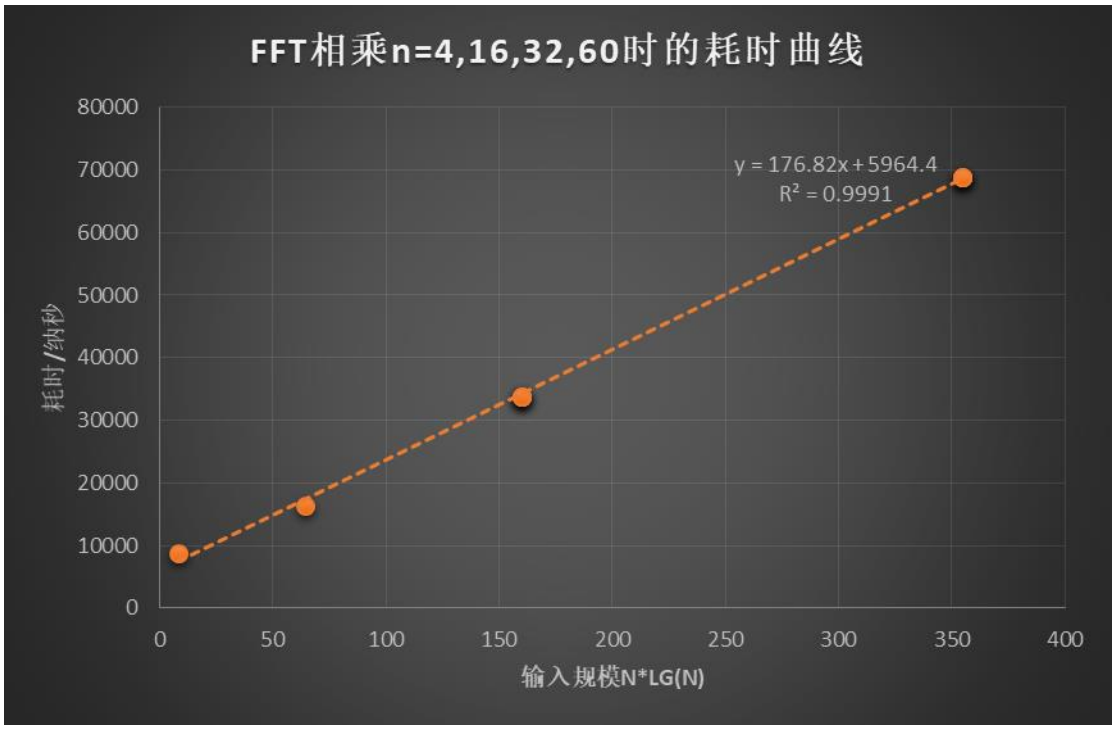
b) Ex2 多项式相乘的 FFT 实现：

根据 time.txt 得到以下几个结果：

这是 FFT 乘法在输入规模 n=4,16,32,60 时的表格：

规模 n	4	16	32	60
耗时/ns	8888	16444	33777	68889
nlg(n)	8	64	160	354.4134

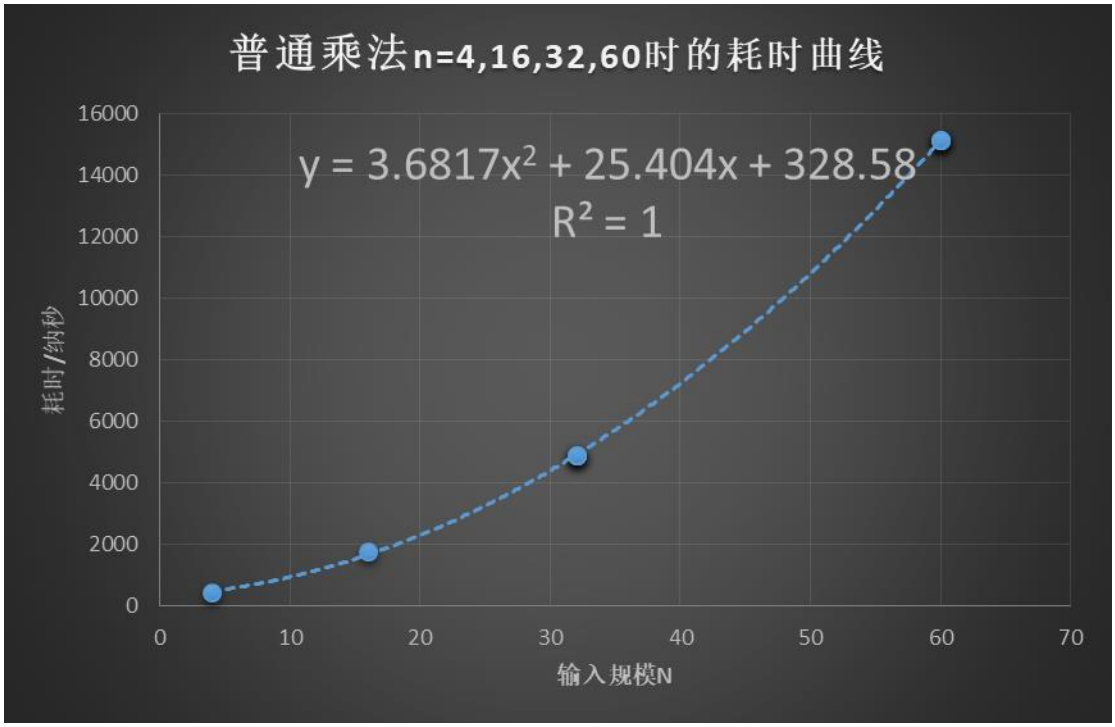
得到 FFT 在输入规模 n=4,16,32,60 时的耗时曲线如下图所示，注意横坐标为 n*lg(n)，可以看出 nlgn 与耗时 t 趋于线性关系，与书中所说的 $\Theta(n\lg n)$ 渐进时间复杂度一致。



这是普通乘法在输入规模 n=4,16,32,60 时的表格：

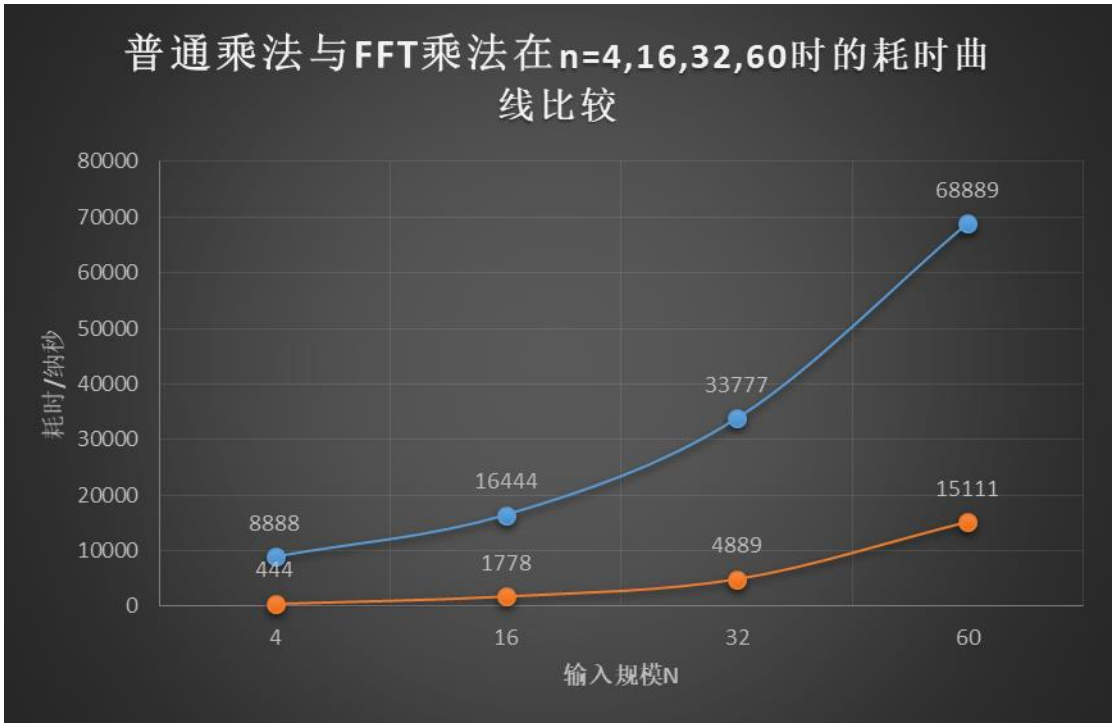
规模 n	4	16	32	60
耗时/ns	444	1778	4889	15111
nlg(n)	8	64	160	354.4134

得到普通乘法在输入规模 n=4,16,32,60 时的耗时曲线如下图所示。拟合效果十分良好，符合书上所说的' $\Theta(n^2)$ 的渐进时间复杂度。



根据下表，比较输入规模 n 较小时的 FFT 乘法与普通乘法性能，得图。

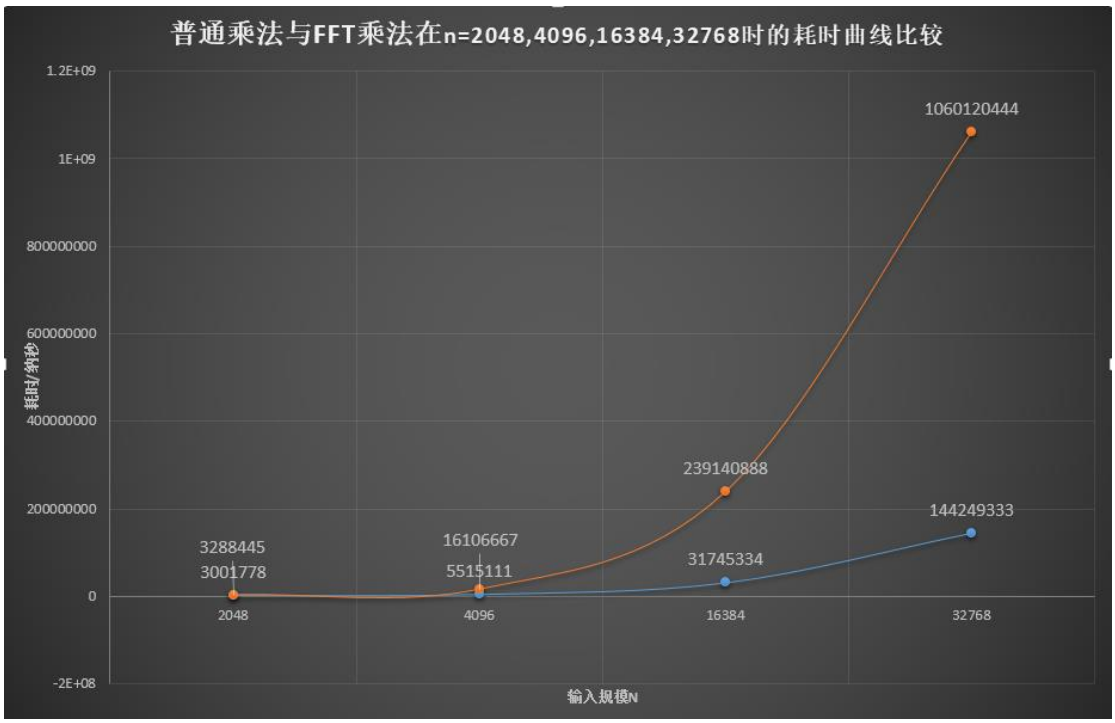
规模 n	4	16	32	60
fft 乘法耗时/ns	8888	16444	33777	68889
普通乘法耗时/ns	444	1778	4889	15111



由图可见，n 较小时，普通乘法比 FFT 乘法性能要好很多，FFT 乘法的额外常熟开销很大。

所以考虑 n 较大时的性能比较，根据下表得到 n 在输入规模比较大时，二者的性能比较图。

规模 n	2048	4096	16384	32768
fft 乘法耗时/ns	3001778	5515111	31745334	1.44E+08
普通乘法耗时/ns	3288445	16106667	2.39E+08	1.06E+09



由图可知，在输入规模为 2048 时，FFT 的性能与普通乘法性能接近，性能好一点点，在 2048 之后，由于二者渐进复杂度本身的显著差异，之后 FFT 与普通乘法的性能差距越拉越大。

6. 实验心得

a) 首先，通过本次实验，加深了对动态规划和 FFT 的理解，这是最重要的。尤其是通过实现各个算法，发现了很多以前没有考虑到的细节。这是最大的收获。

b) 其次，通过本次实验，发现了 FFT 的重要性与顾老师的良苦用心。一开始遇到 FFT，我的内心是抗拒的，因为 FFT 与信号处理密切相关，而信号处理是我不感兴趣的领域。但是本次实验 FFT 通过它的优秀的性能，实实在在、真真切切地让我感受到了它的魅力，以及学习它的必要性。

c) 再其次，通过本次实验，我对 java 有了进一步的理解，得到了很多细节性的认识。

d) 最后，和上一次一样就，感谢可爱哒助教读到这里（虽然没有上一次的 30 多页那么长），感谢的同时心疼一下。。。

e) 算法很美很重要，要加深理解与思考！

f) 祝好！