Project2 实验报告

PB15111672 林郅琦

实验要求

实验1

- 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（ p0、p1、...、pn ）代表矩阵的规模，其中第i 个矩阵(1≤i ≤ n)的规模为pi-1 × pi ，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。

实验2

- 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（a0、a1、...、an-1）和（b0、b1、...、bn-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。

实验环境

- 系统环境： MacOS High Sierra
- 编译环境：g++ 4.2.1; Apple LLVM version 9.0.0 (clang-900.0.38)
- 机器内存：8G
- CPU 主频：2.9 GHz Intel Core i5

实验过程

实验1

- 实验1要求解决矩阵链乘问题，使用的主要方法为动态规划，最终结果获得矩阵链乘的最优乘法次序。
- 动态规划核心等式： m[i][j] = min{ m[i][j], m[i][k] + m[k + 1][j] + data[i - 1] * data[k] * data[j] }
  其中 m[i][j] 维护矩阵乘法从第 i 个矩阵乘到第 j 个矩阵所可能的最小乘法次数。 i <= k <= j -1
- 当找到更小的乘法次数时，使用 s 二维数组记录分割点 k， s[i][j] = k
- 输出乘法次序：由于使用了 s[i][j] 记录第 i 个矩阵乘到第 j 个矩阵的分割点 k，采用递归方式用括号分割即可输出正确的乘法次序

实验2

- 多项式乘法可分为两种途径来计算：FFT 和 普通多项式乘法

基于FFT的多项式乘法

- 基于 FFT 的多项式乘法：
  - 通过 FFT 算法 计算出相乘两个多项式各自的 2n 个点值表示，其中点值采用复数单元 w_n 的形式进行计算，具体矩阵可参考算法导论课本。(O(nlog2n) 时间)
  - 通过两两值乘获得多项式积的结果。(O(n) 时间)
  - 通过逆向 FFT 将得到的乘积值转换为系数（ O(nlog2n) 时间)
- 核心 FFT 函数：
  - 采用递归形式调用，算法可参见算法导论，实质上使用了复数单元的性质使得原本需要 O(n^2) 的计算缩减到 O(nlog2n) 时间的计算
- 逆向 FFT 函数：
  - 将复数单元 wn 取倒数即可，其他与 FFT 完全相同
- 最后通过逆向 FFT 获得的系数值还要除以 n （这是因为范德蒙矩阵的逆矩阵的关系）
- [注]: FFT 要求多项式具备2的指数次项数目，如果实验中 n 不是2的指数次，则需要补0直到2的指数次，并且由于乘法需要输出2n个点值表示，还需要再将系数增长1倍（增长的部分补0）。

传统多项式乘法

- 传统多项式乘法思想主要来自于手工计算，让多项式 B(x) 的每一项系数乘以 A(x) 的每一项系数，进项加法计算。算法过程较为简单。

实验关键代码

实验1： 矩阵链乘算法

- 动态规划中核心代码如下：

```
// m[i][j] 维护第 i 个矩阵乘到第 j 个矩阵所需要的最小乘法次数，s 二维数组记录分割点
for(int l = 2; l <= n; ++l){
    for(int i = 1; i <= n - l + 1; ++i){
        int j = i + l - 1;
        m[i][j] = INT_MAX;
        for(int k = i; k <= j - 1; ++k){
            int q = q;
            if(q < m[i][j]){
                m[i][j] = q;
                s[i][j] = k;
            }
        }
    }
}
```

- 输出序列核心代码如下：

```
// s 为算法导论课本上用于记录矩阵乘分割点的二维数组，s[i][j] = k 表示先 i-k 矩阵
// 链乘后， k + 1 - j 矩阵链乘后，链乘结果在进行乘法计算
// 函数采用递归调用的方式进行输出，使用括号进行分割
void PrintParens(vector< vector<int> > s, int i, int j){
    if(i == j){
        cerr << "A" << i;
        result_file << "A" << i;
    }
    else{
        cerr << "(";
        result_file << "(";
        PrintParens(s, i, s[i][j]);
        PrintParens(s, s[i][j] + 1, j);
        cerr << ")";
        result_file << ")";
    }
}
```

实验2： 多项式乘法

- FFT 函数核心代码：

```
typedef vector<complex<double> > CVector;
CVector RecursiveFft(const CVector a, bool forward = true){
    int n = a.size();
    if(n == 1){
        complex<double> res = a[0];
        return CVector(1, res);
    }
    int ndiv2 = n >> 1;
    complex<double> wn;
    if(forward)
        wn = polar(1.0, 2 * M_PI / n);
    else
        wn = polar(1.0, -2 * M_PI / n);
    complex<double> w = 1;
    CVector a0(ndiv2);
    CVector a1(ndiv2);
    for(int i = 0; i < ndiv2; ++i){
        a0[i] = a[i << 1];
        a1[i] = a[(i << 1) + 1];
    }
    CVector y0 = RecursiveFft(a0, forward);
    CVector y1 = RecursiveFft(a1, forward);
    CVector res(n);
    for(int k = 0; k < ndiv2; ++k){
        res[k] = y0[k] + w * y1[k];
        res[k + ndiv2] = y0[k] - w * y1[k];
        w = w * wn;
    }
    return res;
}
```

其中 `forward` 参数指示此次 FFT 采用 正向FFT 还是采用 逆向 FFT 计算，如果 `forward = true`，则采用正向 fft 计算。如果 `forward = false` 则采用逆向 fft 计算。

算法思路与算法导论书上的基本相同， 参数 `const CVector a` 在函数第一次调用是输入为系数（正向FFT）或者乘积值（逆向 FFT）。

- FFT 乘法核心流程

```
vector<double> MultiPoly(CVector a, CVector b, int poly){
    // 扩大系数向量(前面已经使系数向量扩大到2的指数次，a，b 分别为两个多项式系数向量)
    int len = a.size();
    CVector expansion(len, complex<double>(0, 0));
    a.insert(a.end(), expansion.begin(), expansion.end());
    b.insert(b.end(), expansion.begin(), expansion.end());
    // 对a，b 两个系数向量进行求 2n 个点值表示，FFT 算法
    CVector ya = RecursiveFft(a);
    CVector yb = RecursiveFft(b);
    CVector yc(len << 1);
    // 点值乘法
    for(int i = 0; i < (len << 1); ++i)
        yc[i] = ya[i] * yb[i];
    // 对结果进行逆向 FFT 获得系数向量
    CVector cc = RecursiveFft(yc, false);
    // 将取系数向量实部作为系数向量结果(实际上此时虚部均为0)
    int res_len = (poly << 1) - 1;
    vector<double> c(res_len);
    for(int i = 0; i < res_len; ++i){
        c[i] = cc[i].real() / (len << 1);
    }
    return c;
}
```

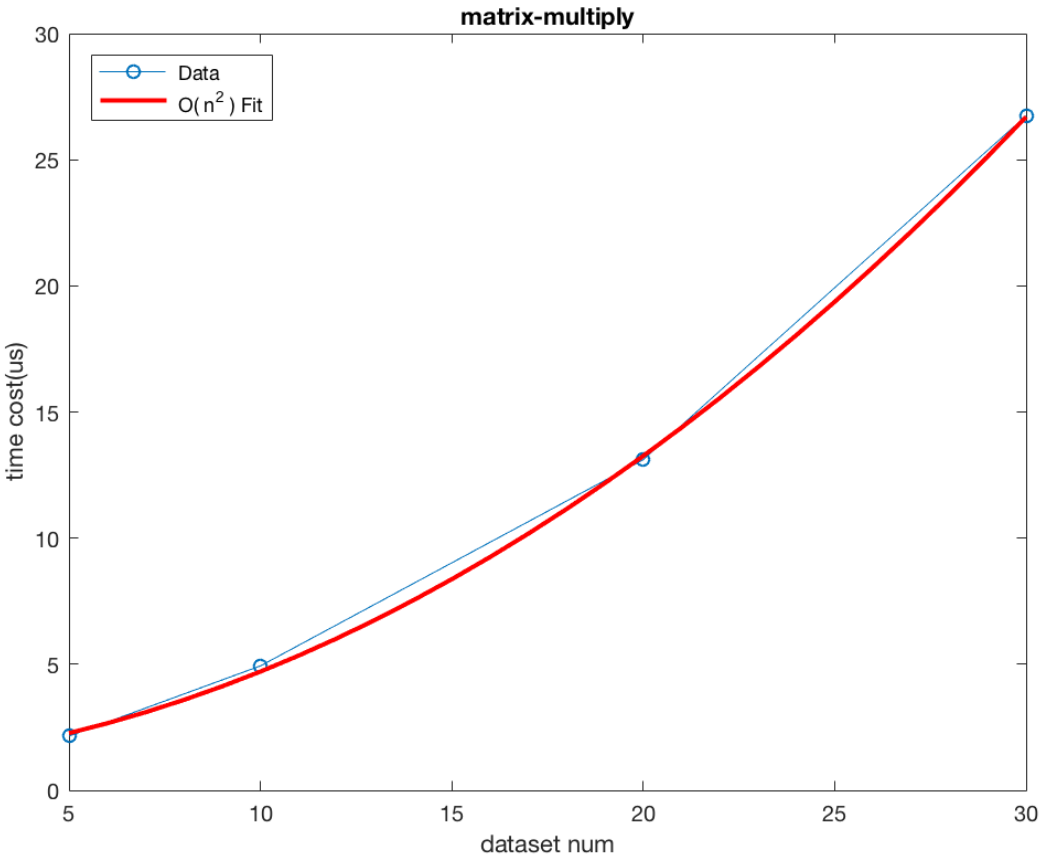
- 普通乘法核心流程

```
// 参数 a, b 表示两个多项式向
vector<double> PolyMulti(const vector<double>& a, const vector<double>& b){
    int result_len = a.size() + b.size() - 1;
    vector<double> res(result_len, 0);
    for(int i = 0; i < a.size(); ++i){
        for(int j = 0; j < b.size(); ++j){
            // x^(i) * x^(j) = x^(i + j)
            res[i + j] += a[i] * b[j];
        }
    }
    return res;
}
```

实验结果及分析

- 实验均能成功编译并且成功运行，运行结果核算后没有出现问题。
- 实验结果均存于输出 `output` 文件目录下。
- 通过运行时间进行拟合比对，使用 matlab 获得如下运行时间-规模曲线：

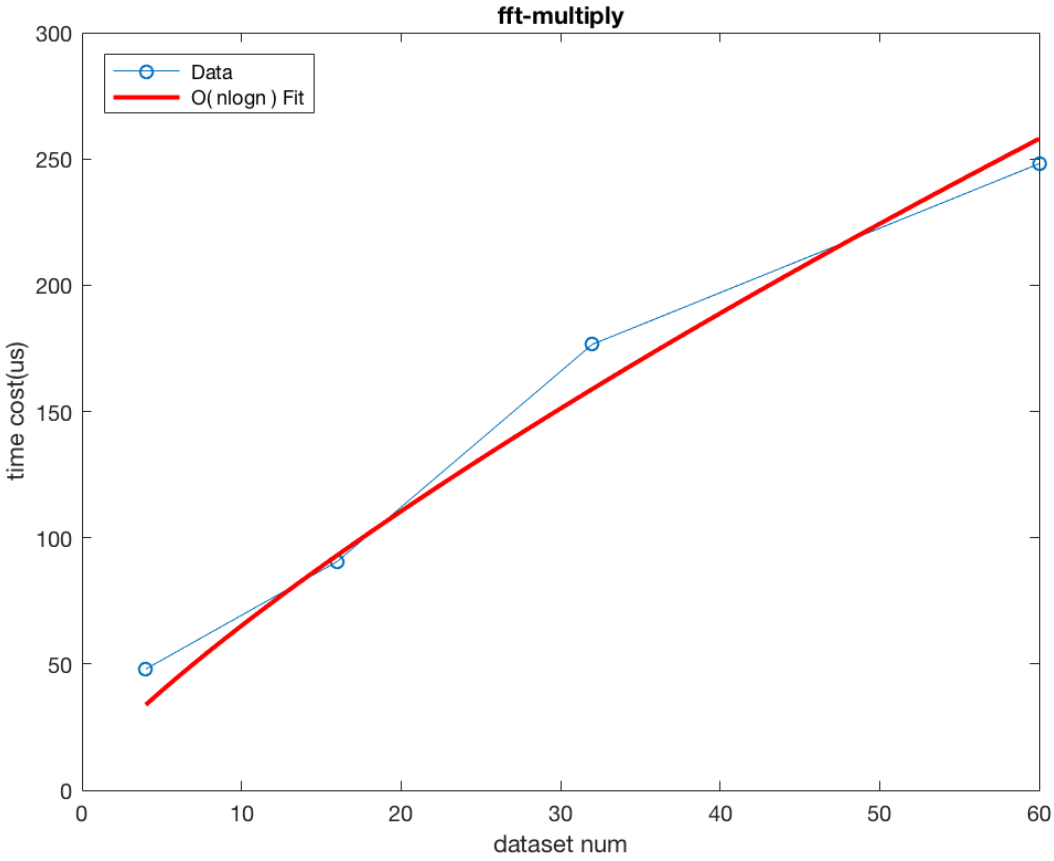
矩阵链乘



其中， 时间-数据量拟合结果为： `t = 0.0245 * x^2 + 0.1209 * x + 1.503`

从上面的实验结果可以看出，拟合结果较好，这是因为算法的优良时间局部性和空间局部性使得受到硬件干扰的情况较小。

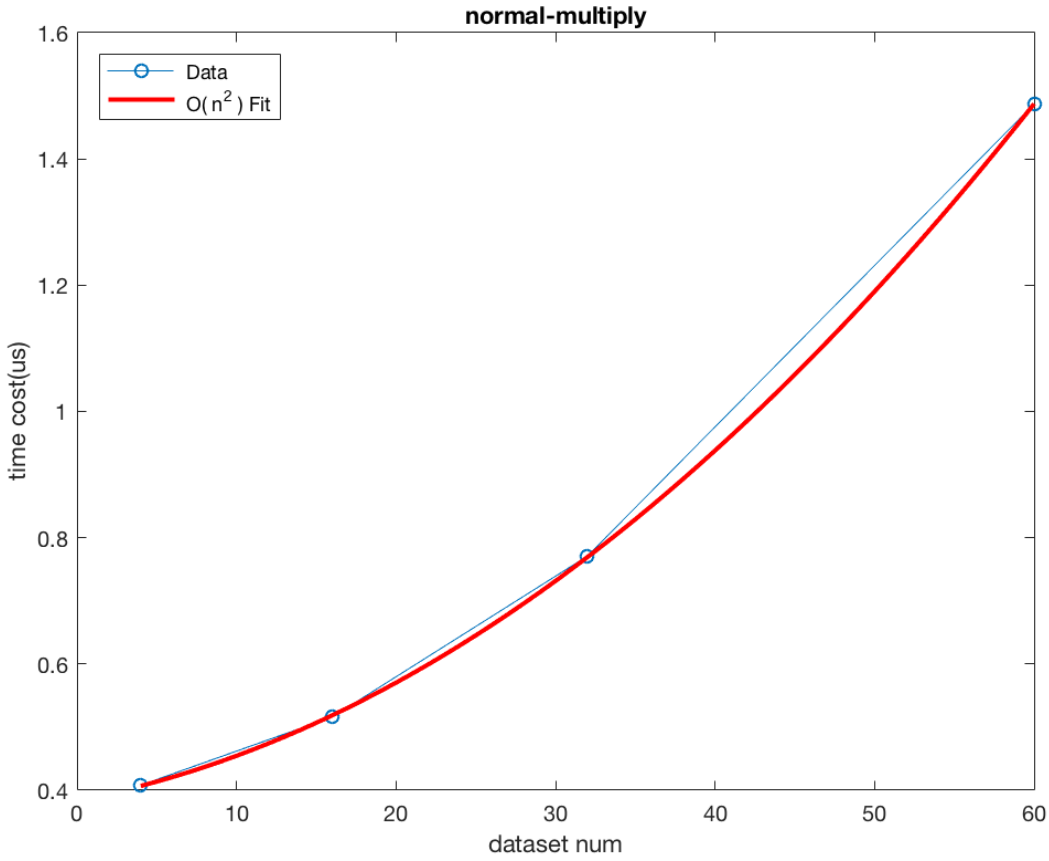
基于 FFT 的多项式乘法



其中，时间-数据量拟合结果为：`t = -0.6137nlog2(6.7344n) + 9.4872n + 7.5448`

从上面结果可以看出，仅仅这4个点不能很好的体现算法运行时间的 nlog(n) 的渐进复杂度（反而此时做线性拟合能获得更好的成果）。可能的原因为：首先是算法自身的问题，采用递归的 FFT 算法，虽然理论上的算法时间渐进复杂度为 O(nlogn)，然而真正在执行算法时由于计算机硬件的一些特性，如 cache 命中率（实际上递归 FFT 算法 cache 命中率不高）等因素对影响到算法执行，且随着数据规模的改变，cache 命中率也会出现波动，从而造成了并不能很好拟合的结果。

普通多项式乘法



其中，时间-数据量拟合结果为 `t =0.0002x^2 + 0.0048x + 0.3841`

从上面结果可以看出，此次实验数据较符合渐进时间复杂度为 O(n^2) 的结论，之所以能较为符合，原因是：首先整个算法的cache 命中率非常高，时间局部性和空间局部性非常好，因此对算法产生的干扰就会很小，算法的执行时间也逐渐趋于理想情况。

普通多项式乘法运行相同的数据规模会出现时间远远小于基于 FFT 的多项式乘法，这在数据规模量较少的情况下极其正常，当数据规模量大到2000以上是，基于 FFT 的多项式乘法才逐渐显现出优势。数据规模量小的时候 FFT 性能远低于普通乘法的原因为：尽管 FFT 渐进复杂度是 O(nlogn) 的，然而系数却非常大（这一点仅从两张拟合获得的曲线表达式即可看出），而系数非常大的原因是 cache 命中率的硬件原因和算法本身步骤的繁杂性所致，所以获得这样的结果非常正常。

实验心得

首先这两次的实验非常有意义，第一个动态规划实验能对动态规划算法有了很直观概念上的了解，从输出结果也能感受到动态规划算法的优越性。

其次是 FFT 多项式乘法：这个实验能让我们深刻了解 FFT 算法的巧妙性，同时理解即便理论上好的时间渐进复杂度也会因为算法本身和机器硬件而导致性能在规模较小时明显弱于传统算法。

实验源码编译与执行

- 实验1：矩阵链乘

  source 文件夹中：

  编译出可执行文件 `$make`

此时会产生两个可执行文件：data_gen 和 mulmatrix

data_gen 将随机生成数据放入 input 中； mulmatrix 将根据生成的 input 中的文件读取数据获取进行自动计算自动输出到文件夹。

使用 `$make clean` 将删除所有产生的可执行文件

- 实验2：多项式乘法

  source 文件夹中：

  编译出可执行文件：`$make`

  此时会产生三个可执行文件： data_gen; fft; norm

  data_gen 将随机生成数据放入 input 文件夹中；fft 将根据生成的 input 文件夹中的文件读取数据并执行基于 fft 的多项式乘法计算获得结果和时间输入进 output 文件夹中；norm 与 fft 功能相似，只是 norm 使用普通多项式乘法来得到最终结果。

  使用 `$make clean` 将删除所有产生的可执行文件

- 以上生成的所有可执行文件直接在终端执行即可，无需传入额外的参数。