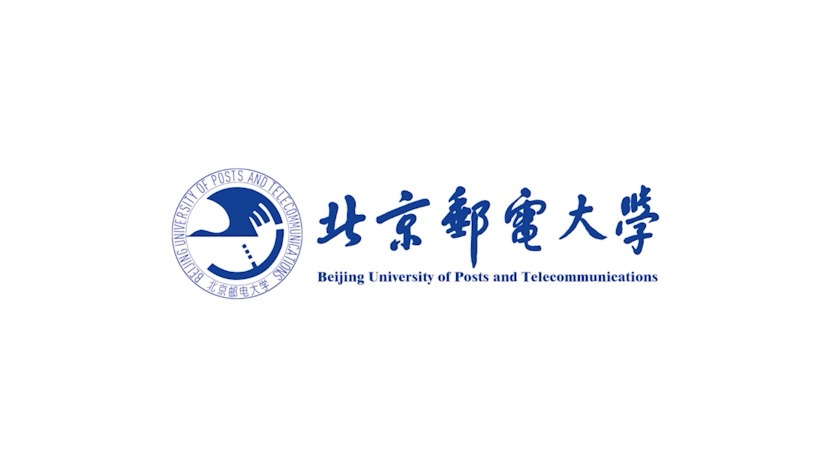
**算法设计与分析实验报告**



实验题目： 利用 FFT 算法改进大整数乘法的算法效率

姓名： 马天成

学号： 2020211376

日期： 2022-10-18

目录

[一、实验环境 4](#_Toc117018821)

[1.1 设备规格 4](#_Toc117018822)

[1.2 操作系统 4](#_Toc117018823)

[1.3 编程语言&编译器 4](#_Toc117018824)

[1.4 开发工具 5](#_Toc117018825)

[二、实验内容 5](#_Toc117018826)

[2.1 实验内容及要求 5](#_Toc117018827)

[2.2 基于传统分治的大整数乘法算法 6](#_Toc117018828)

[2.2.1 原理 6](#_Toc117018829)

[2.2.2 数据结构 6](#_Toc117018830)

[2.2.3 字符串计算 6](#_Toc117018831)

[2.2.3 测试 7](#_Toc117018832)

[2.3 FFT 7](#_Toc117018833)

[2.3.1 原理 7](#_Toc117018834)

[2.3.2 代码 8](#_Toc117018835)

[2.3.3 测试数据 10](#_Toc117018836)

[三、出现问题及解决 11](#_Toc117018837)

[四、总结 11](#_Toc117018838)

[4.1 时间复杂度空间复杂度 11](#_Toc117018839)

[4.2 实验理解 12](#_Toc117018840)

# 一、实验环境

## 1.1 设备规格

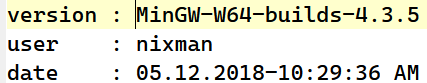


## 1.2 操作系统



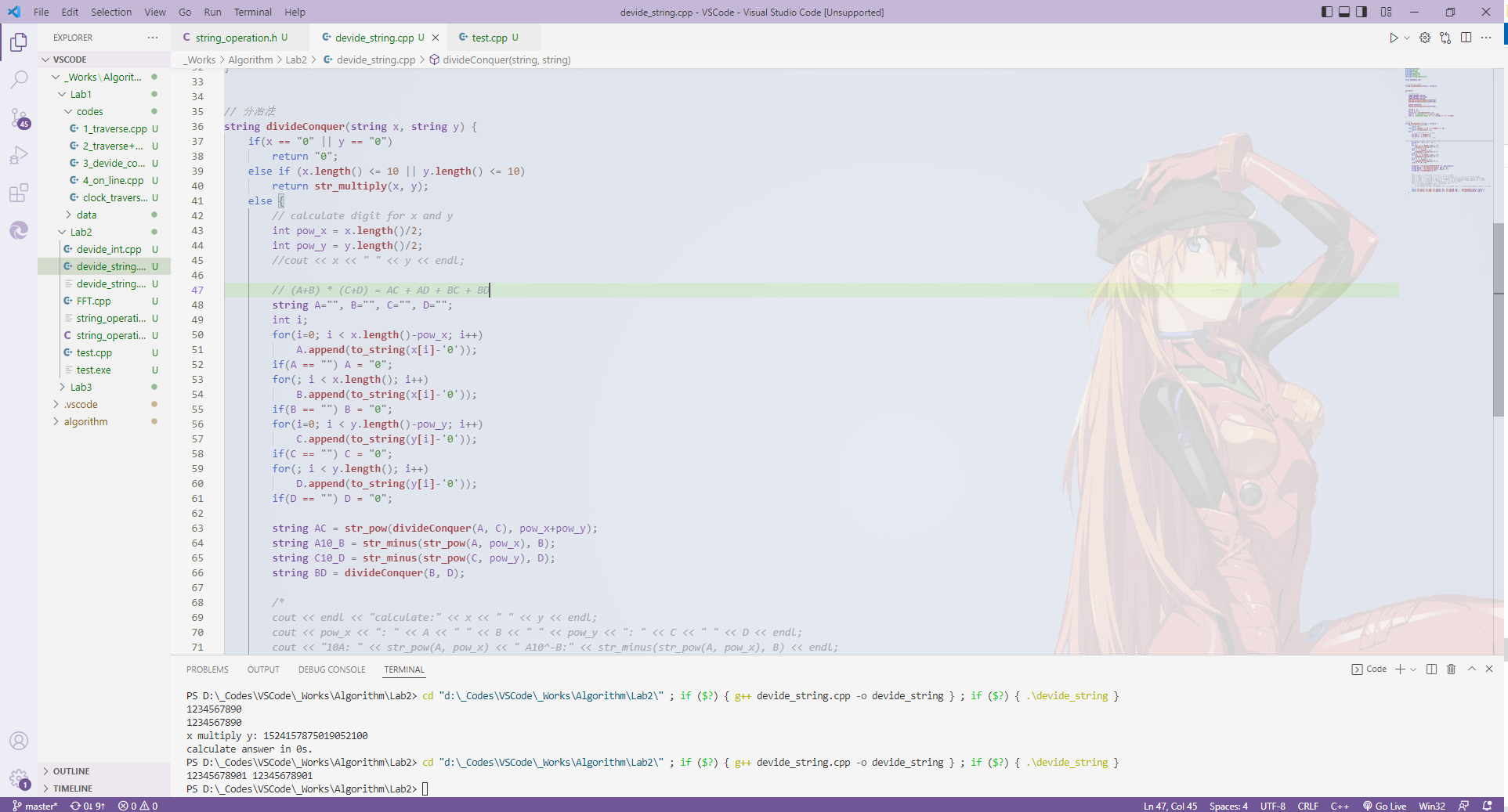
## 1.3 编程语言&编译器

C++11



## 1.4 开发工具

VSCode



# 二、实验内容

## 2.1 实验内容及要求

1. 算法的设计与实现

学习 FFT（快速傅里叶变换算法）的基本原理，基于快速傅里叶变换算法改进传统的大整数乘法分治算法，即将基于分治的大整数乘法的算法效率由 O(n 1.59)提高到 O(nlogn)。

2. 测试要求

设计测试数据集，编写测试程序，用于测试：

a) 正确性：所实现的三种算法的正确性；

b) 算法复杂性：针对两种算法，设计测试数据集，评价各个算法在算 法复杂性上的表现；（最差情况、平均情况）

3. 撰写评价报告

a) 结合实验结果，在理论上给予总结和评价两种算法在算法复杂性和效率上的表现。形成电子版实验报告。

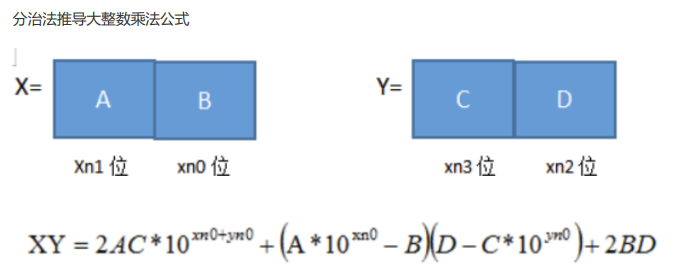
4. 作业清单

①程序（算法程序和测试程序）；

②测试数据集和测试程序；

## 2.2 基于传统分治的大整数乘法算法

### 2.2.1 原理



### 2.2.2 数据结构

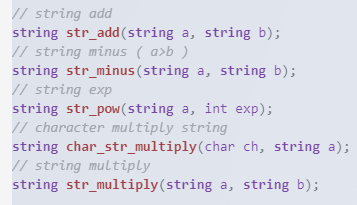
大整数一定是由字符串来存的。这样才能实现大整数乘法。

这里限制所有的输入都是正数。

这里就采用递归的手法了。非递归属是有点折磨人。虽然说写肯定能写，但是时间一定是来不及了。

### 2.2.3 字符串计算

这里提供了五个函数：



功能如注释所写。

**注意的地方：**

0. 所有地方注意0输入

1. 加法注意进位

2. 减法注意借位和去除前导0

3. 乘法注意pow

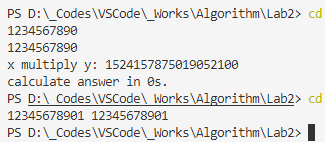
### 2.2.3 测试

* 最好最坏情况测试

假如我们采用这种方式

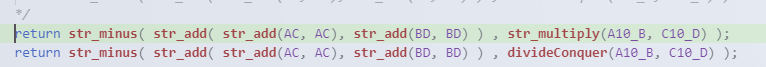


那么这两个乘法也会采用divide的方法。



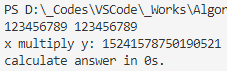
那么十位数就会爆掉。

但假如用这个就不会有太大的问题。因为他是指数级的减小规模。



* 平均情况

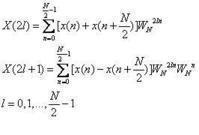
基本都可以在0秒内解决。（主要是过大的输入会直接爆栈）



## 2.3 FFT

### 2.3.1 原理

1. 设N点序列x(n), 将x(n)按奇偶分组：



2. 改写为：



3. 一个N点DFT分解为两个 N/2点的DFT：



4. 继续分解，[迭代](https://baike.baidu.com/item/%E8%BF%AD%E4%BB%A3?fromModule=lemma_inlink)下去，其运算量约为



**组合数基四FFT**

N≠2M时，可采取补零使其成为N=2，或者先分解为两个p、q的序列，其中p\*q=N，然后进行计算。

### 2.3.2 代码

#include<iostream>

#include<cstdio>

#include<cmath>

#include<time.h>

using namespace **std**;

const int MAXN = 1e7 + 10;

inline int **read**()

{

    char c = **getchar**(); int x = 0, f = 1;

    while (c < '0' || c > '9') {

        if (c == '-')f = -1;

        c = **getchar**(); }

    while (c >= '0' && c <= '9') {

        x = x \* 10 + c - '0';

        c = **getchar**();

    }

    return x \* f;

}

const double Pi = **acos**(-1.0);

struct **complex**

{

    double x, y;

**complex**(double xx = 0, double yy = 0) { x = xx, y = yy; }

} a[MAXN], b[MAXN];

**complex** **operator +** (**complex** a, **complex** b) {

    return **complex**(a.x + b.x, a.y + b.y);

}

**complex** **operator -** (**complex** a, **complex** b) {

    return **complex**(a.x - b.x, a.y - b.y);

}

**complex** **operator \*** (**complex** a, **complex** b) {

    return **complex**(a.x \* b.x - a.y \* b.y, a.x \* b.y + a.y \* b.x);

}

int N, M;

int l, r[MAXN];

int limit = 1;

void **fast\_fast\_tle**(**complex**\* A, int type)

{

*//Find out the sequence to iterate*

    for (int i = 0; i < limit; i++)

        if (i < r[i]) **swap**(A[i], A[r[i]]);

    for (int mid = 1; mid < limit; mid <<= 1) { *//Half the length of the interval to be combined*

**complex** **Wn**(**cos**(Pi / mid), type \* **sin**(Pi / mid)); *//Unit root*

        for (int R = mid << 1, j = 0; j < limit; j += R) { *//R is the length of the interval, and j is where it has been before*

**complex** **w**(1, 0); *//power*

            for (int k = 0; k < mid; k++, w **=** w **\*** Wn) { *//Enumerate the left half*

**complex** x = A[j + k], y = w **\*** A[j + mid + k];

                A[j + k] **=** x **+** y;

                A[j + mid + k] **=** x **-** y;

            }

        }

    }

}

int **main**() {

*// read data*

    cout **<<** "Please input two numbers started with bit\n";

    int N = **read**(), M = **read**();

    for (int i = 0; i <= N; i++) a[i].x = **read**();

    for (int i = 0; i <= M; i++) b[i].x = **read**();

*// start timer*

**clock\_t** start, end;

    start = **clock**();

    while (limit <= N + M) limit <<= 1, l++;

    for (int i = 0; i < limit; i++)

        r[i] = (r[i >> 1] >> 1) | ((i & 1) << (l - 1));

*// start fft*

**fast\_fast\_tle**(a, 1);

**fast\_fast\_tle**(b, 1);

    for (int i = 0; i <= limit; i++)

        a[i] **=** a[i] **\*** b[i];

**fast\_fast\_tle**(a, -1);

**string** temp;

    int\* ntemp = new int[N + M + 2];

    for (int i = 0;i < N + M + 2;i++)

        ntemp[i] = 0;

    int t = 0;

    for (int i = N + M; i >=0 ; i--) {

        t= (int)(a[i].x / limit + 0.5);

        ntemp[i] = (ntemp[i+1]+t) / 10;

        ntemp[i+1] = (ntemp[i+1] + t) % 10;

    }

    bool flag = true;

    for (int i = 0;i < N + M + 2;i++) {

        if (flag)

            ntemp[i] != 0 ? flag = false : flag = true;

        if(!flag)

            temp **+=** ntemp[i]+'0';

    }

*// output time and answer*

    cout **<<** temp**<<endl**;

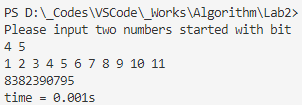
    end = **clock**();

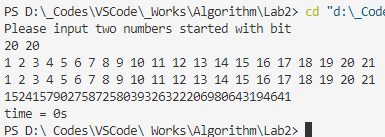
    cout **<<** "time = " **<<** double(end - start) / **CLOCKS\_PER\_SEC** **<<** "s" **<<** **endl**;

    return 0;

}

### 2.3.3 测试数据





只能说FFT算法没有爆栈速度也快，非常美丽。

# 三、出现问题及解决

**1. 编写字符串功能**

在编写字符串功能的时候，至少花费了6h。主要是经过各种各样的测试才能保证自己的算法是天衣无缝的。其中主要是0的处理以及倒转进位借位要处理，所以才会显得比较复杂。

**2. 经典分治的实现**

在实现的时候，我发现无论是边界的处理还是分治的位数处理都需要小心。比如2位数出现的0，或者因为边界值过小导致的问题规模太大而导致时间复杂度巨高。

**3. FFT的处理**

其中主要是罩着老师给的算法嗯看，看不懂也得写。所以还是非常折磨人的。

主要就是写转换的时候需要很多心思，但有了老师给的算法，那就比较简单了。

# 四、总结

## 4.1 时间复杂度空间复杂度

* 经典的分治算法

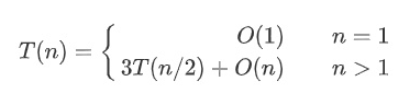
第一种写法：



这个显然复杂度很高，我一开始甚至认为是指数型的。

理想情况下：x、y位数相同

XY=AC\*10n+((A-B)(D-C)+AC+BD)\*10n/2+BD



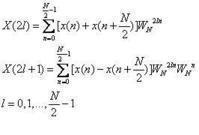
看起来似乎更复杂，其实只需要做三次n/2为整数的乘法。T(n)=O（n^log3）

非理想，则x，y位数不同。

空间复杂度也很高，每一个栈都需要与输入规模成正比的线性空间。

* FFT

1. 设N点序列x(n), 将x(n)按奇偶分组：



2. 改写为：



3. 一个N点DFT分解为两个 N/2点的DFT：



4. 继续分解，[迭代](https://baike.baidu.com/item/%E8%BF%AD%E4%BB%A3?fromModule=lemma_inlink)下去，其运算量约为



## 4.2 实验理解

首先是经典的分治算法。其实我一开始写了个int或者long型的，发现根本就是。。。天方夜谭。这能叫大整数？所以只好乖乖改成字符串。

在这之后，我又进行了字符串计算的编写。这个是比较耗时间的，因为我们这个东西确实是很多bug需要处理。从0的处理到位数的处理到分治的处理，都需要话一定时间来进行debug。

接下来是FFT。FFT老师上课也描述过一遍了，印象很深，但压根没怎么听懂哈哈哈。因为确实接受起来比较难。

我们需要明白：FFT算法实质上就是DFT算法的改良版，而DFT算法则是傅里叶变换的离散版。按傅里叶变换→DFT→FFT的思路推导，即可理解FFT。

在经历过这个算法的熏陶过后。我充分了解了算法的美丽。他居然能把我们认为的很普通的高复杂度问题简化成如此简单的一个问题。简直是太美妙了。这个分治的思想和点线巧妙的转换思路，我们也要灵活运用在以后的方方面面。