# 数据结构与算法
# 实验报告

第一次



| | |
|---|---|
| 姓名 | 代珉玥 |
| 班级 | 软件 001 班 |
| 学号 | 2205223077 |
| 电话 | 18585038226 |
| Email | 2040257842@qq. com |
| 日期 | 2020-10-17 |

# 目录

实验 1..... 1
1. 题目..... 1
2. 数据结构设计..... 4
3. 算法设计..... 5
4. 主干代码说明..... 5
5. 运行结果展示..... 7
6. 总结和收获..... 7
7. 参考文献..... 7
实验 2..... 8
8. 题目..... 8
9. 数据结构设计..... 8
10. 算法设计..... 8
11. 主干代码说明..... 10
12. 运行结果展示..... 13
13. 总结和收获..... 14
14. 参考文献..... 14

# 实验 1

## 1.题目

在本次实验中，主要完成的任务是：依据指定的 List ADT 实现一个可以使用的 List 的数据结构（本次实验需要验证的输入数据都是 char 类型），并依据给定的输入数据文件进行验证。

指定的 List 的 ADT 描述如下：

```
public interface List<T> {
void insert(T newElement);
/*
* Precondition: List is not full and newElement is not full.
* PostCondition:
* Inserts newElement into a list after the cursor. If the list is empty, newElement is inserted as the first(and only)element in the list. In either case(empty or not empty),moves the cursor to newElement.
*/
void remove();
/*
* Precondition:
* List is not empty.
* PostCondition:
* Removes the element marked by the cursor from a list. If the resulting list is not empty, then moves the cursor to the element that followed the deleted element. If the deleted element was at the end of the list, then moves the cursor to the element at the beginning of the list.
*/
void replace(T newElement);
/*
* Precondition:
* List is not empty and newElement is not null.
* PostCondition:
* Replaces the element marked by the cursor with newElement. The cursor remains at newElement.
*/
void clear();
/*
* Precondition:
* None
* PostCondition:
```

```
* Removes all the elements in a list.
*/
boolean isEmpty();
/*
* Precondition:
* None
* PostCondition:
* Returns true if a list is empty. Otherwise, returns false.
*/
boolean isFull();
/*
* Precondition:
* None
* PostCondition:
* Returns true if a list is full. Otherwise, returns false.
*/
boolean gotoBeginning();
/*
* Precondition:
* None
* PostCondition:
* If a list is not empty, then moves the cursor to the beginning of the list and returns true.Otherwise, returns false.
*/
boolean gotoEnd();
/*
* Precondition:
* None
* PostCondition:
* If a list is not empty, then moves the cursor to the end of the list and returns true.Otherwise, returns false.
*/
boolean gotoNext();
/*
* Precondition:
* List is not empty.
* PostCondition:
* If the cursor is not at the end of a list, then moves the cursor to the next element in the list and return true. Otherwise, returns false.
*/
boolean gotoPrev();
/*
* Precondition:
* List is not empty.
* PostCondition:
```

```
* If the cursor is not at the beginning of a list, then moves the cursor to the preceding element in the list and returns true. Otherwise, returns false.
*/
T getCursor();
/*
* Precondition:
* List is not empty.
* PostCondition:
* Returns a copy of the element marked by the cursor.
*/
void showStructure();
/*
* Precondition:
* None
* PostCondition:
* Outputs the elements in a list and the value of cursor. If the list is empty, outputs "Empty list". Note that this operation is intended for testing/debugging purpose only.
*/
}
```

为了方便进行测试验证，对 List 的各种操作指定了相应的命令符号，具体的符号含义如下：

| | |
|---|---|
| + | insert |
| - | remove |
| = | replace |
| # | gotoBeginning |
| * | gotoEnd |
| > | gotoNext |
| < | gotoPrev |
| ~ | clear |

假如对一个空表做如下的操作列表，并且假设该线性表中插入的元素类型为 char 类型，那么下面表格中将示例对命令符号使用的运行结果：

| 命令行内容 | 执行结果（调用 List 的 showStructure 行为的输出内容） |
|---|---|
| +a +b +c +d | a b c d 3 |
| # > > | a b c d 2 |
| * < < | a b c d 1 |
| - | a c d 1 |
| +e +f +f | a c e f f d 4 |
| * - | a c e f f 0 |
| - | c e f f 0 |
| =g | g e f f 0 |
| ~ | Empty list -1 |

实验会给定一个数据文件，名为 testCase.txt，该文件中的每一行内容都类似于上表中的每

一行“命令行内容”列中所指示的内容。要求每执行一行，就调用 List 接口中的 showStructure 行为，用以验证该命令行的执行是否正确。我们会给定相应的执行结果列表（记录在 result.txt 文件中）。一旦发现某一行不正确，请进行修正，并在自己的实验报告中记录出错的原因。这种实验过程的记录是一个很重要的成绩考察方面，请大家重视。

【备注：对 testCase.txt 中命令行的命令的识别，可以借助于 JDK API 中的 StreamTokenizer 类辅助完成，该类的使用请自学。】

# 2. 数据结构设计

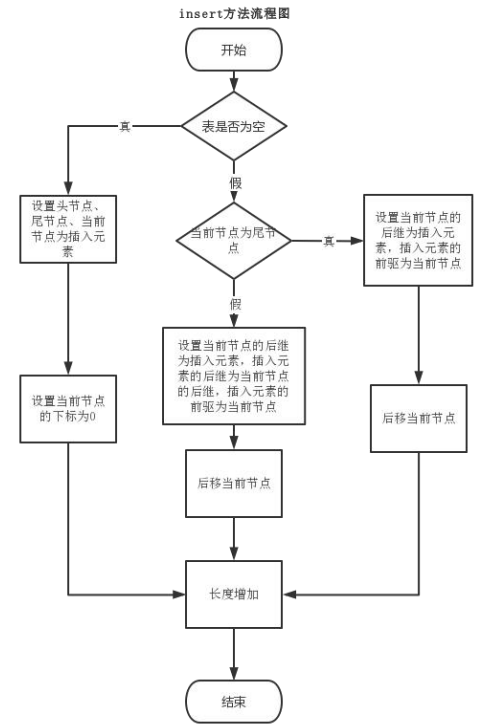
采用双向链表实现。
未采用哑节点，当前节点物理位置为其逻辑位置。

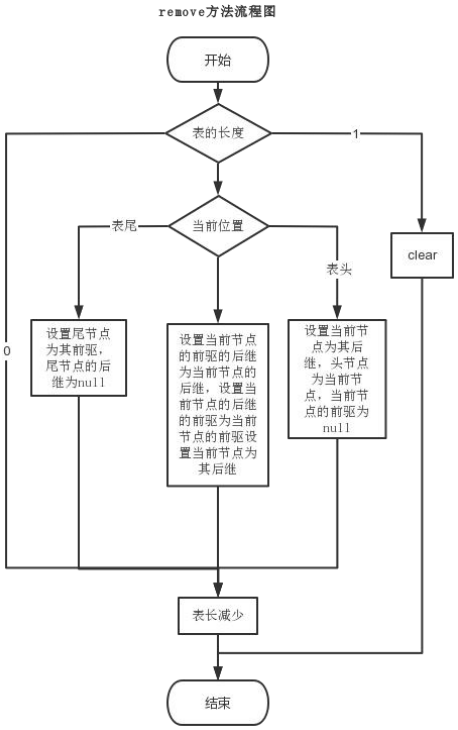
```
public class MyList implements List{
    private Node first;
    private Node end;
    private Node cursor;
    private int cursorValue;
    private int lenghth;
}
```

每个节点储存其前驱、后继和元素。

```
public class Node <T>{
    private   T element;          //节点储存的元素
    private   Node next;          //节点的后继
    private Node previous;        //节点的前驱
}
```

# 3. 算法设计




其他方法的实现省略。

# 4. 主干代码说明

```
@Override
public void insert(Object newElement) {
    if (lenghth == 0) {                                          //如果表空
        Node temp = new Node(newElement, null, null);            //设置一个临时节点储存
                                                                   该元素
        setCursor(temp);                                         //设置当前节点为该元素
        cursorValue = 0;                                         //设置当前节点下标为 0
        setFirst(temp);                                          //设置头节点为该元素
        setEnd(temp);                                            //设置尾节点为该元素
    } else if (cursor == end) {                                  //如果当前位置为表尾
        cursor.setNext(new Node(newElement, null, cursor)); //设置当前节点的后继为
                                                                   插入元素，插入元素的
                                                                   前驱为当前节点
        cursor = cursor.getNext();                               //后移当前节点
        cursorValue++;
        end = cursor;                                            //设置当前节点为尾节点
    } else {
        cursor.setNext(new Node(newElement, cursor.getNext(), cursor)); //设置当前
                                                                   节点的后继
```

```
                                                                为插入元素，
                                                                插入元素的
                                                                后继为当前
                                                                节点的后继，
                                                                插入元素的
                                                                前驱为当前
                                                                节点
            cursor = cursor.getNext();                          //后移当前节点
            cursorValue++;
            cursor.getNext().setPrevious(cursor);               //设置当前节点的后继的前
                                                                驱为当前节点
        }
        lenghth++;                                              //长度增加
    }

    @Override
    public void remove() {
        if (lenghth != 0) {             //表不为空
            if (lenghth == 1) {         //如果表中只有一个元素
                this.clear();           //置空表
            } else {                    //表中有多个元素
                if (cursor == end) {                //如果当前节点为尾节点
                    end = end.getPrevious();        //设置尾节点为其前驱
                    end.setNext(null);              //设置尾节点的后继为 null
                    this.gotoBeginning();           //设置当前节点为头节点
                } else if (cursor == first) {       //如果当前节点为头节点
                    cursor = cursor.getNext();      //设置当前节点为其后继
                    first = cursor;                 //设置头节点为当前节点
                    cursor.setPrevious(null);       //设置当前节点的前驱为 null
                } else {
                    cursor.getPrevious().setNext(cursor.getNext()); //设置当前节点的前
                                                                    驱的后继为当前节
                                                                    点的后继
                    cursor.getNext().setPrevious(cursor.getPrevious());//设置当前节点的
                                                                       后继的前驱为当
                                                                       前节点的前驱
                    cursor = cursor.getNext();      //设置当前节点为其后继
                }
                lenghth--;                          //表长减少
            }
        }
    }
```

# 5. 运行结果展示

测试结果截图如下：


结果为大小为 16.3 KB (16,709 字节)的 txt 文件，与 result.txt 文件比对发现测试结果正确（由于结尾空格的设置不同，字节数量不同，内容相同）。

# 6. 总结和收获

- 在测试时，发现 remove 方法编写出现问题。当只有一个元素时置空了表，设置表长为 0，但是最后仍然执行了 length--命令，导致之后调用其他方法时，（length==0）不成立，出现空指针异常。经过这次实验，今后对于变量的维护应该更注意，尤其在当前方法中调用其他方法，而其他方法修改了成员变量的时候，因为此时不易注意的变量已经修改。
- 在测试时，测试文档中出现了向空表中插入两个元素后删除两个元素后替换当前元素的值。而 replace 方法的前提是表不为空，编写时没有处理这个情况，所以测试时出现了空指针异常。在编写时还是要多处理几种情况。

# 7. 参考文献

无。

# 实验 2

## 8. 题目

任何一个编程语言，其定义的整数类型的范围都是有限的，比如 Java 语言，能够表示整数的基本类型中 long 型可以存储的最大整数是：9223372036854775807，存储的最小整数是：-9223372036854775808。但在很多应用场合下，整数值的大小超过了上面两个数所表示的范围，这个时候我们需要采用其他方式来完成这类型整数的算术运算了。请设计一个类 BigNumber，该类可以支持大整数的基本算术运算。具体要求如下：

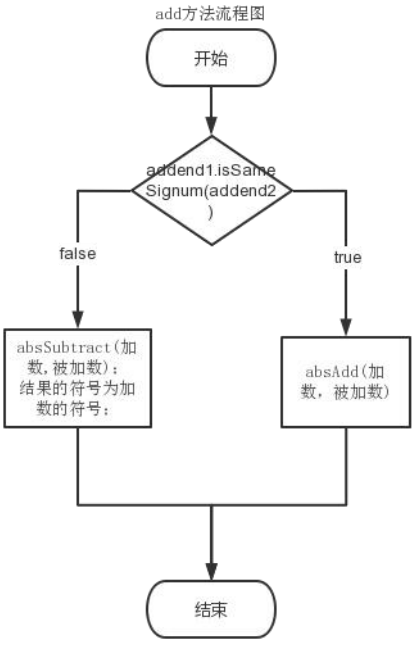
构建合理的数据结构存储这些需要运算的大整数；

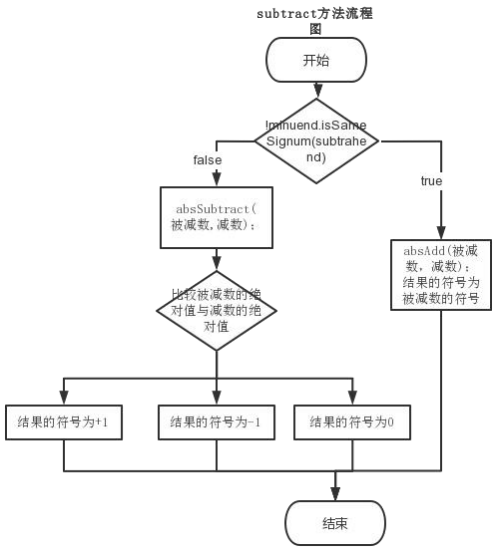
实现两个大整数之间的加法、减法、乘法和除法运算，其中除法运算只需要算出整数商即可。

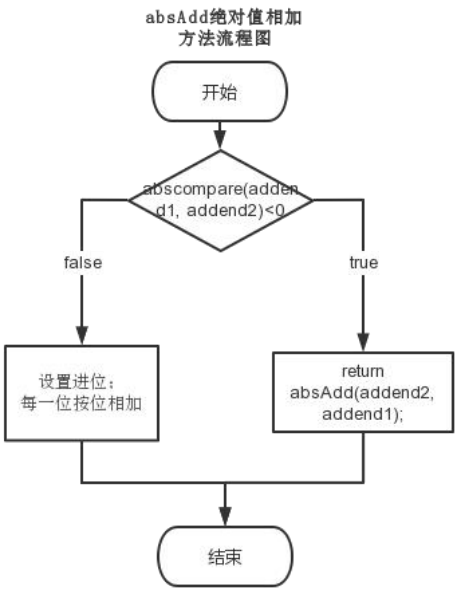
## 9. 数据结构设计

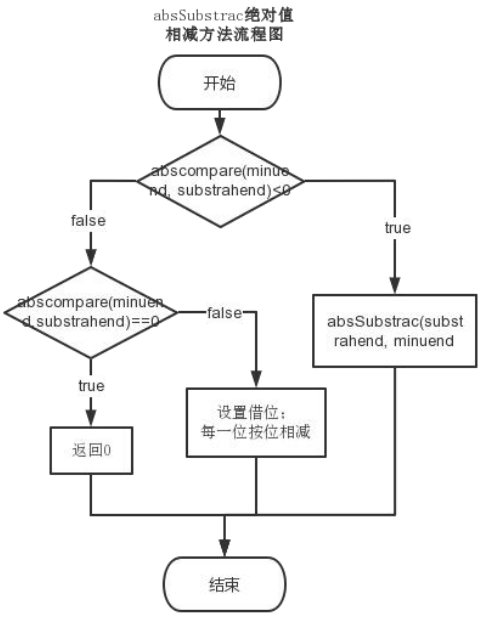
```
public class MyBigInteger {
    int signum;        //-1 表示负数，0 表示 0，1 表示正数
    int[] mag;         //分位储存该整数
    int length;        //该整数的位数
}
```

## 10. 算法设计






multiplication 乘法方法：

for (int i = 0; i < multiplier1.length; i++) {

for (int j = 0; j < multiplier2.length; j++) {

结果的第 i+j 位+=multiplier1 的 i 位*multiplier2 的第 j 位;

}

判断 result 的长度;

result+0;

判断 result 的 signum

结束;

division 除法方法：

if (除数为 0){

throw new ArithmeticException("除数不能为 0");

}

if (被除数为 0){

返回 0;

}

MyBigInteger temp = dividend;

temp.signum=1;

while(temp 的绝对值大于等于除数){

temp = temp 的绝对值减去除数的绝对值;

result = result 的绝对值加一;

}

result.signum=dividend.isSameSignum(divisor)?1:-1;

结束;

}

# 11. 主干代码说明

```
public static MyBigInteger add(MyBigInteger addend1, MyBigInteger addend2) {
    MyBigInteger result = null;
    if(addend1.isSameSignum(addend2)){
        result = MyBigInteger.absAdd(addend1,addend2);          //绝对值相加
        result.signum=addend1.signum;
    }else {
        result = MyBigInteger.absSubstrac(addend1,addend2);     //绝对值相减
        int compare = MyBigInteger.abscompare(addend1, addend2);
        if (compare > 0) {                                      //设定符号
            result.signum = addend1.signum;
        } else if(compare < 0){
            result.signum = addend2.signum;
        }else {
            result.signum=0;
        }
    }
    return result;
}

public static MyBigInteger subtract(MyBigInteger minuend, MyBigInteger subtrahend) {
    MyBigInteger result = null;
    if(!minuend.isSameSignum(subtrahend)){
        result = MyBigInteger.absAdd(minuend,subtrahend);           //绝对值相加
        result.signum=minuend.signum;
    }else{
        result = MyBigInteger.absSubstrac(minuend, subtrahend);     //绝对值相减
        int compare = MyBigInteger.abscompare(minuend, subtrahend);
        if (compare > 0) {                                          //设定符号
            result.signum = minuend.signum;
        } else if(compare < 0){
            result.signum = minuend.signum == 1 ? -1 : 1;
        }else {
            result.signum=0;
        }
    }
    return result;
}
```

```
public static MyBigInteger absAdd(MyBigInteger addend1, MyBigInteger addend2) {
    if (abscompare(addend1, addend2)<0) {
        return absAdd(addend2, addend1);
    } else {
        MyBigInteger result = new MyBigInteger();
        int carry = 0;                                          //设置进位;
        for (int i = 0; i < addend1.length; i++) {              //按位相加
            int temp = addend1.mag[i] + addend2.mag[i] + carry;
            carry = temp / 10;
            result.mag[i] = temp - carry * 10;
        }
        if (carry != 0) {                                       //设置长度（位数）
            result.mag[addend1.length] = carry;
            result.length = addend1.length + 1;
        } else {
            result.length = addend1.length;
        }
        result.signum = 1;
        return result;
    }
}

public static MyBigInteger absSubstrac(MyBigInteger minuend, MyBigInteger substrahend) {
    if (abscompare(minuend, substrahend)<0) {
        return absSubstrac(substrahend, minuend);
    } else if(abscompare(minuend,substrahend)==0){
        return new MyBigInteger();
    } else {
        MyBigInteger result = new MyBigInteger();
        boolean borrowing=false;                                //设置退位
        for (int i = 0; i < minuend.length; i++) {
            int temp = minuend.mag[i] - substrahend.mag[i]+(borrowing?-1:0);
            if (temp < 0) {
                temp += 10;
                borrowing=true;
            }else {
                borrowing=false;
            }
            result.mag[i] = temp;
        }
        for(result.length = minuend.length;result.length>0;result.length--){   //设置长度
            if (result.mag[result.length-1]!=0){
                break;
            }
        }
```

```
            result.signum = 1;
            return result;
        }
    }

    public static MyBigInteger multiplication(MyBigInteger multiplier1, MyBigInteger multiplier2)
    {
        if (multiplier1.signum==0||multiplier2.signum==0){
            return new MyBigInteger();
        }
        MyBigInteger result = new MyBigInteger();
        for (int i = 0; i < multiplier1.length; i++) {
            for (int j = 0; j < multiplier2.length; j++) {
                result.mag[i+j]+=multiplier1.mag[i]*multiplier2.mag[j];
            }
        }
        for(result.length = multiplier1.length+multiplier2.length;result.length>0;result.length--){
            if (result.mag[result.length-1]!=0){
                break;
            }
        }
        result = MyBigInteger.absAdd(result,new MyBigInteger("0")); //调整每一位为个位数
        result.signum=multiplier1.isSameSignum(multiplier2)?1:-1;   //设置长度
        return result;
    }

    public static MyBigInteger division(MyBigInteger dividend, MyBigInteger divisor) throws
    ArithmeticException{
        if (divisor.signum==0){
            throw new ArithmeticException("除数不能为 0");
        }
        if (dividend.signum==0){
            return new MyBigInteger();
        }
        MyBigInteger result = new MyBigInteger();
        MyBigInteger temp = dividend;
        temp.signum=1;
        while(MyBigInteger.abscompare(temp,divisor)>=0){
            temp = MyBigInteger.absSubstrac(temp,divisor);
            result = MyBigInteger.absAdd(result,new MyBigInteger("1"));
        }
        result.signum=dividend.isSameSignum(divisor)?1:-1;
        return result;
    }
```

# 12. 运行结果展示

测试代码：

```
public class Test {
    public static void main(String[] args) {
        MyBigInteger myBigInteger1 = new MyBigInteger( string: "999");
        MyBigInteger myBigInteger2 = new MyBigInteger( string: "-999");
        MyBigInteger myBigInteger3 = new MyBigInteger( string: "-123");
        MyBigInteger myBigInteger4 = new MyBigInteger( string: "0");
        MyBigInteger myBigInteger5 = new MyBigInteger( string: "35");
        MyBigInteger myBigInteger6 = new MyBigInteger( string: "7");
        //加法
        System.out.println(myBigInteger1+"+"+myBigInteger2+"="+MyBigInteger.add(myBigInteger1,myBigInteger2));
        System.out.println(myBigInteger5+"+"+myBigInteger3+"="+MyBigInteger.add(myBigInteger5,myBigInteger3));
        //减法
        System.out.println(myBigInteger1+"-"+myBigInteger2+"="+MyBigInteger.subtract(myBigInteger1,myBigInteger2));
        System.out.println(myBigInteger4+"-"+myBigInteger6+"="+MyBigInteger.subtract(myBigInteger4,myBigInteger6));
        //乘法
        System.out.println(myBigInteger1+"*"+myBigInteger4+"="+MyBigInteger.multiplication(myBigInteger1,myBigInteger4));
        System.out.println(myBigInteger5+"*"+myBigInteger6+"="+MyBigInteger.multiplication(myBigInteger5,myBigInteger6));
        //除法
        System.out.println(myBigInteger4+"/"+myBigInteger1+"="+MyBigInteger.multiplication(myBigInteger4,myBigInteger1));
        System.out.println(myBigInteger5+"/"+myBigInteger6+"="+MyBigInteger.multiplication(myBigInteger5,myBigInteger6));

        System.out.println("1/0="+MyBigInteger.division(new MyBigInteger( string: "1"),new MyBigInteger( string: "0")));
    }
}
```

运行结果：

```
D:\Javaaaa\ruanjian\bin\java.exe "-javaagent:D:\Javaaaa\ruanjian\idealIU1\IntelliJ
999+-999=0
35+-123=-88
999--999=1998
0-7=0
999*0=0
35*7=245
0/999=0
35/7=245
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : 除数不能为0
    at second.MyBigInteger.division(MyBigInteger.java:183)
    at second.Test.main(Test.java:26)
```

# 13. 总结和收获

- 刚开始想除法的时候不太有思路，感觉自己在做除法的时候大多在估计，不知道怎么在计算机当中完成估算。之后想到按照下述的方式：

  以 123456789/23 为例。

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 3 | | | | | | | |

  12<23,所以

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| | 2 | 3 | | | | | | |

  123/23=5；123%23=8；所以

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 8 | 4 | 5 | 6 | 7 | 8 | 9 |
| | | 2 | 3 | | | | | |

  这样就形成了递归。每一次处理的被除数和除数都相差 1 位或 0 位。

  但是最后没有这么写。因为要截取出一个数据的第 i 位到第 k 位（如第一步需要截取前两位），之前没考虑要写这样的构造方法，写除法的时候加减乘均已经完成，再改害怕前面的关联到的地方太多，需要修改的太多，所以最后还是按照一直减到被除数小于除数为止的方法来写了。

  乘法和除法是先写了代码再去写算法设计，加减是先算法设计再写的代码，之后还是先设计再写代码比较好，前者在写代码的过程中发现有问题会给修改带来麻烦。

- 在写减法的时候，刚开始的时候忘记设置位数了，出现了 999-999=000，增加了求位数这一步之后解决了。还是要记得对变量的维护。

# 14. 参考文献

无。