第 1.1 题：长整形运算

实验报告

题目：编制一个实现任意长的整型进行加法运算的演示程序。
班级：F1702120 学号：517021910526 姓名：陈聪 作业贡献度：55%
班级：F1702127 学号：517021910737 姓名：叶嘉迅 作业贡献度：45%

一、 需求分析

1. 本演示程序中，输入数据限定为：按中国对于长整数的表示习惯，每四位一组，组间用逗号隔开。

2. 相加过程中不能破坏两个操作数链表；不能给长整数规定上限。

3. 程序允许的操作为：将两个按要求输入的数字进行相加（若需求为减法则第二个输入数字为负数），选作为乘除和次方。

4. 测试数据：

（1）0;0;应输出 0

（2）-2345,6789;-7654,3211; 应输出-1,0000,0000

（3）-9999,9999;1,0000,0000,0000; 应输出 9999,0000,0001

（4）1,0001,0001;-1,0001,0001; 应输出 0

（5）1,0001,0001;-1,0001,0000; 应输出 1

（6）-9999,9999,9999;-9999,9999,9999; 应输出-1,9999,9999,9998

二、 概要设计

本题最主要的特点是要求数字为无限长，可以较好实现这个特点的的方式是链表和字符串。本报告主要针对基于链表的方法。

具体实现思路为：链表表头存储数字的正负号，node 存储四位数字，加减则按照人类习惯从后向前遍历两链表，依次相加减。

实验要求实现加法，这其中比较复杂的部分在于加法的进位、减法的借位操作。在后续的报告中会着重介绍这部分的实现。

此外，当操作对象为(一个绝对值较小的正数 a;一个绝对值较大的负数 b;)时，人类一般常见的做法是将这种计算变成(-(-|b|+|a|))进行计算(比如 9989-10078，一般人习惯是计算-(10078-9989))，本报告也会着重介绍这一部分的实现。

1. 链表节点类 node

对象：

node *next; ：指向后一节点的指针

node *prev; ：指向前一节点的指针

int num; ：存储四位数字，

基本操作：

node(); ：创造一个节点

2. 链表类 longlongint

对象：

node *head,*rear; ：指向头节点和尾节点

int ranking; ：指定第一个/第二个数字，唯一作用是在 get()函数中将两个数字进行区分

int currentlength; ：存储当前链表长度

基本操作：

longlongint (int ranking); ：构造一个链表

~longlongint(); ：析构链表，返还所有开辟的空间

void get(std::string number); ：使用一个字符串对链表赋值，使链表中每个节点存储 4 位数字

longlongint(const longlongint &a); ：复制构造函数，开辟新的空间并将原链表中的所有元素进行复制

longlongint operator + (longlongint another); ：对加法进行重载，起封装作用，返回 longlongint 格式的加法结果

int operator <(longlongint number); ：对两个 longlongint 存储的数字进行绝对值比较，主要用于辅助处理前面所说的(一个绝对值较小的正数 a;一个绝对值较大的负数 b;)的情况

三、 详细设计

1. 节点类

1.1 node.h

```
class node {
public:
    node *next;    //指向前节点
    node *prev;    //指向后节点
    long num;      //存储数字
    node();        //构造新的节点
};
```

1.2 node.cpp

```
node::node()
{
    num = 0;       //初始化数字为 0
}
```

2. 链表类

2.1 longlongint.h

```
class longlongint
{
    friend std::ostream & operator<<(std::ostream & out, longlongint number);
    //重载输出函数，按人类习惯的模式输出链表中的数字
private:
    int currentlength;  //存储当前链表长度
    int ranking;    //指定第一个/第二个数字，唯一作用是在 get()函数中将两个数字进行区分

public:
    node *head,*rear;       //指向头节点和尾节点的指针

    longlongint (int ranking);//构造函数，初始化一个链表
    ~longlongint();         //析构函数，返还所有开辟的空间
    void get(std::string number);       //使用输入的字符串对链表进行赋值，得到一个每节点存储 4 个数字的链表
    longlongint(const longlongint &a); //拷贝构造函数，默认拷贝构造函数不适用于链表类，需要自定义新拷贝构造函数，
    longlongint operator + (longlongint another);     //重载加法，返回 longlongint 的加法结果
    int operator <(longlongint number);               //对两个 longlongint 存储的数字进行绝对值比较，主要用于辅助处理前面所说的(一个绝对值较小的正数 a;一个绝对值较大的负数 b;)的情况
};
```

2.2 longlongint.cpp

```
//构造函数，初始化各参数与头尾节点
longlongint::longlongint(int n)
{
ranking = n;
head = new node;
head->next = rear = new node;
rear->prev = head;
currentlength = 0;
head->num = 1;
}

//析构函数，从尾节点向前遍历返还所有开辟的空间
longlongint::~longlongint()
{
    while(head != rear)
    {
        node *temp = head;
```

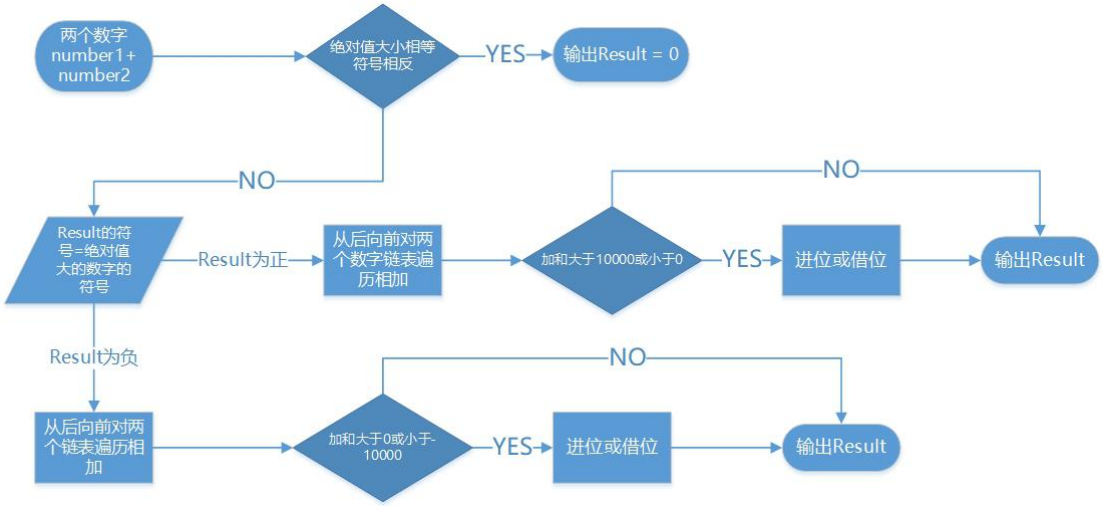
```
        head = head->next;
        delete temp;
    }

}



//拷贝构造函数，返回一个完全新开辟的、各节点值完全相等的新链表
longlongint::longlongint(const longlongint &target)
{
    node *temp = target.head;          //temp 用于遍历 target 中的各节点，并对其中值进行复制
    node *temp_rear = target.rear;     //temp_rear 用于在 temp 指向表尾时停止遍历

    head = new node;
    head->next = rear = new node;
    rear->prev = head;
    currentlength = 0;
    head->num = temp->num;             //新建链表并初始化 head、rear 节点

    temp = temp->next;
    node *add = this->head;            //add 用于在新建链表中接入新节点
    while(temp != temp_rear)
    {
        node *link = new node;
        link->num = temp->num;
        add->next = link;
        link->next = this->rear;
        this->rear->prev = link;
        link->prev = add;

        add = add->next;
        temp = temp->next;
        currentlength ++;              //将 target 链表中各元素复制到新建链表中
    }
}
```

```
//get 函数用于对 longlongint 对象进行赋值，得到一个各节点存储四个数字的链表
void longlongint::get(std::string number)     //参数为输入的字符串
{
    int change = int(number.find(';'));        //确定输入字符串中';'的位置

    int start = ranking ==1? 0:change + 1;
    int ending = ranking ==1? change - 1 : number.length()-2;
    //根据 ranking 的值对字符串进行不同的划分：
    //如果 ranking 为 1，则取;前的部分，如果 ranking 为 2，则取;后的部分
    if (number[start] == '-')
        {
            head->num = -1;
            start++;
        }                          //单独考虑负号，如果有，则输入头节点
    number = number.substr(start,ending - start + 1);      //根据 ranking 取相应子链

    node* temp = new node;
    rear->next = temp;
    temp->prev = rear;
    rear = rear->next;
    currentlength ++;                                  //先接入一个节点

    int flag = 0;
    int tens[] = {1,10,100,1000};                          、
    //flag 用于选取 tens 种所要乘的数，例如 5879   = 5*1000+8*100+7*10+9
    while(number.length() != 0)
    {
        int length = number.length() - 1;
        if(number[length] == ',')
    //输入中的,分隔了四个数字，依此进行遍历，遍历方法为：对每个位上的数字乘上它应
    //有的权值，并求和如 5879   = 5*1000+8*100+7*10+9
            {    number.erase(length);   //擦去,，并接入一个新的节点
                 flag = 0;
                 node* temp = new node;
                 temp->next = head->next;
                 head->next = temp;
                 temp->prev = head;
                 temp->next->prev = temp;
                 currentlength ++;
                 length --;
            };
        head->next->num += tens[flag] * (number[length]-48);
        number.erase(length);   //每加上一位数字，就擦去一位
        flag ++;}}
```

```
//重载输出函数，以人类的习惯输出链表中的数字
std::ostream & operator << (std::ostream & out , longlongint number)
{
    if(number.head->num == -1){out << '-';}//如果为负，输出一个-
    int tens[4] =   {1000,100,10,1};
    //以下循环防止了 将 1 输出为 0001 这种多余 0 出现的情况，即，将链表从前向后遍
    //历，跳过值为 0 的节点
    while(number.head->next->num == 0 && number.currentlength != 1)
    {
        node *temp1 = number.head->next;
        number.head->next = temp1->next;
        temp1->next->prev = number.head;
        delete temp1;
        number.currentlength--;
    }
    //对链表遍历，输出所有节点的数字的绝对值（绝对值是为了完善后面减法的实现）
    //此外，每 4 个数字遍历一次保证了 10010 不会输出为 110（防止直接输出 int 类型丢
失前面应有的 0）
    node *temp = number.head->next;
    while (temp != number.rear)
    {
        int the_number = abs(temp->num);
        if(temp != number.head->next)
        {
            for (int iter = 0; iter <= 3; iter ++)
            {
                out<<the_number / tens[iter];
                the_number = the_number - (the_number / tens[iter])*tens[iter];
            }
        }
        else
        {
            out<<abs(temp->num);
        }
        temp = temp->next;
        if(temp != number.rear)
        {
            out<<",";
        }
    }
}
```

```
//重载比较函数，只比较绝对值！
int longlongint::operator < (longlongint number)
{
    //更长的显然绝对值更大，这里本来打算用三目写，但可惜分大于等于小于三种情况
    if(this->currentlength < number.currentlength){return 1;}
    else{;}
    if(this->currentlength > number.currentlength){return 0;}
    else{;}
    //如果长度相等，从前往后遍历，如果一直相等，那就相等
    node* number1 = this->head->next;
    node* number2 = number.head->next;
    for (int i = 1; i <= currentlength; i++)
    {
        if (number1->num != number2->num)
        {
            return (number1->num < number2->num)? 1:0;
        }
        else
        {
            number1 = number1->next;
            number2 = number2->next;
        }
    }
    return 2;
}
```

//重载加法，没有写在 main 中主要起封装以减少 main 函数冗余度和提高代码复用率的作用
longlongint longlongint::operator + (longlongint another)
总计近 100 行，代码比较繁杂，逻辑比较难说，用流程图表示：



将循环压缩，只展示整体代码结构：

```
longlongint longlongint::operator + (longlongint another)
{
    longlongint target(1);                  //返回的对象
    node *rear_temp = target.rear;          //指向目标表尾的指针，用于添加节点
    node *number1 = this->rear->prev;       //number1 指向 this 的第一个含数字节点
    node *number2 = another.rear->prev;     //number2 指向 another 的第一个含数字节点
    int sign1 = this->head->num;            //number1 的符号
    int sign2 = another.head->num;          //number2 符号
    int carry = 0;                          //进位/借位用
    int judge = *this < another;            //判定 number1 和 number2 绝对值大小关系，2
                                            //为相等，1 为 this 小，0 为 another 小
    //绝对值相等，符号相反，返回 0
    if((judge == 2 && this->head->num * another.head->num == -1)||
       (judge ==2 && this->currentlength ==1 && this->head->next->num == 0))
    {
        node* temp = new node;
        temp->num = 0;
        target.currentlength = 1;
        rear_temp->prev = temp;
        target.head->next = temp;
        temp->next = rear_temp;
        temp->prev = target.head;
        rear_temp = rear_temp->prev;
        return target;
    }
```

```
    //结果的符号等于绝对值大者的符号
    target.head->num = judge==1? another.head->num:this->head->num;

    int range_max;
    int range_min;
    range_max = target.head->num ==1? 9999:0;
    range_min = target.head->num ==1? 0:-9999;
    int carry_ini = -target.head->num;

    //构造一个足够大的链表，以装下结果，构建的链表长为 max(length) + 1
    int length = this->currentlength > another.currentlength ? currentlength :
another.currentlength;
    for (int i = 0; i< length; i++)
    {
        node* temp = new node;
        rear_temp->prev = temp;
        target.head->next = temp;
        temp->next = rear_temp;
        temp->prev = target.head;
        rear_temp = rear_temp->prev;
    }
    rear_temp = target.rear->prev;

    //将 number1 的值从后向前赋值入 target 中，相当于 target (0)+ number1
    while (number1 != this->head)
    {
        rear_temp->num += sign1 * number1->num;
        rear_temp = rear_temp->prev;
        number1 = number1->prev;
    }
    rear_temp = target.rear->prev;

    //将 target + number1 再从后向前加上 number2，相当于 target + number1+number2
    while (number2 != another.head)
    {
        rear_temp->num += sign2 * number2->num + carry;
        carry = 0;
        //在 num 小于此时的最小值时，借位（carry = -1）
        if(rear_temp->num < range_min && rear_temp != target.head->next)
        {
            rear_temp->num += 10000;
            carry = -1;
        }
```

```
        //在 num 大于此时的最大值时，进位（carry = 1）
        if(rear_temp->num >range_max && rear_temp != target.head->next)
        {
            rear_temp->num -= 10000;
            carry = 1;
        }
        //如果此时循环到了结果的第一个节点，但此时该节点的绝对值大于等于 10000，此时已经没有再前面一位可以进了，于是新建一个节点并存入
        if(abs(rear_temp->num) >=10000 && rear_temp == target.head->next)
        {
            node *temp = new node;
            node *temp1 = target.head->next;
            temp->num = 1;
            temp1->num -= -carry_ini * 10000;
            target.head->next = temp;
            temp->prev = target.head;
            temp1->prev = temp;
            temp->next = temp1;
        }
        rear_temp = rear_temp->prev;
        number2 = number2->prev;
    }
    //第二个循环主要针对 number2，但如果 number2 长度小于 number1，那么有可能虽然遍历了 number2，但仍有位未进
    if (judge == 0)
    {
    rear_temp->num += carry;
    }
    return target;
}



//重载赋值运算，功能和拷贝构造基本一样，故代码实现和拷贝构造也很像，不再赘述
longlongint & operator = (const longlongint & target);
```

3. 主程序模块

主程序模块逻辑比较清楚，因为封装的比较好所以主要都是调用，调用了构造、get、重载的+、重载的输出。

```
int main()
{
	string number;
	cout<<"Please input two number following the basic rule:";
	cin>>number;
	//input the numbers to be added
	longlongint number1(1), number2(2);
	number1.get(number);
	number2.get(number);
	cout<<endl;
	cout<<"The first number is: "<<number1;
	cout<<endl;
	cout<<"The second number is: "<<number2;
	cout<<endl;
	longlongint target_plus = number1 + number2;
	cout<<"The sum is: "<<target_plus;
	cout<<endl;
	//longlongint target_mul = number1 * number2; //乘法被注释掉了
	//cout<<"The product is: "<<target_mul;		//想要看实现效果的话可以取消注释
	//cout<<endl;
	return 0;
	}
```

4. 选作分析

在这里我主要写出乘除的思路，其中乘法在代码中有实现。

法一：暴力

乘法：

```
Target:a*b
While (b!=0)
{ a = a + a;
b = b-1;
}
```

代码实现：（在 main 中被注释了，想要看效果的话可以取消注释）

```
longlongint longlongint::operator * (longlongint another)
{

    int judge = *this < another;
    longlongint target = judge==2 ? *this : another;        //取较大的作为循环 a+=a 的对象
    longlongint other = judge==2 ? another : *this;         //取较小的作为 b-=1 的对象
    int sign2 = other.head->num;
    longlongint number1(1),number2(2);
    std::string temp = sign2 == 1? "-1":"1";
    number1.get(temp);
    number2.get("0");
    longlongint adder = target;
    other = (other + number1);
    while(other<number2 != 2)
    {
        target = target + adder;
        other = (other + number1);}                         //a = a + a;b = b-1;
    return target;}
```

除法：(向绝对值大取整)

因为和乘法思路类似，而且时间复杂度依然很高所以没有实现。

```
Target:a/b
c = abs(a); d = abs(b)
While(c>0)
{c = c - d;
Target++;
}
Return sign1 * sign2 * Target
```

法 2：类比加法

按照人类习惯，乘法应该是对其中一个数字从后向前遍历，带权值地乘另一个数，然后将所有得到的数字相加。

但遇到的问题是，没有好的结构可以存储遍历相乘后的结果，因为这个结果往往比链表本身还要长，唯一想到的办法是用链表嵌套一个链表。如果这样做的话需要构造一个新的类，而且时间空间复杂度依然不低。

四、 调试分析

1. 设计早期很多运算符没有重载，产生了很多意想不到的 bug。比如赋值运算符，本来没有想到要重载，在写乘法的过程中发现有一个 bug 始终定位不到，花了很久才意识到是默认赋值运算不能达到我想要的效果。

2. 一些函数参数传递类型设计的不恰当。在所有程序完成后才开始修改传递参数，其中加法重载因为涉及改变原始值，必须要使用拷贝构造，所以没有传递引用。其他函数都使用了传递引用以提高运算效率，此外，对于不改变值的函数比如小于号重载，还需要加 const 以防止输入对象值被改变。

3. 链表调试过程中，单步调试相较于输出调试（cout 变量值）更有效。链表实现过程中，很容易出现内存泄漏，而对于内存泄漏，输出很难检查出问题究竟出在哪，并且编译器在内存泄漏时并不会指明在哪一行出现了内存泄漏。单步调试对于这种情况十分有效，当某步运行后，程序陷入死循环则很可能出现了内存泄漏。当然，两者结合也可以提高单步调试的效率。

4. 算法的复杂度分析。

1） 时间复杂度

使用了双向链表，设有头尾节点，每个节点存储一个 int 格式的数字，各种操作复杂度较合理。

friend std::ostream & operator<<(std::ostream & out,longlongint &number);重载输出函数的时间复杂度为 O(n)

longlongint (int ranking); 构造函数仅在头节点后加一个节点，时间复杂度为 O(1)

~longlongint();析构函数，遍历一遍链表并返还内存，时间复杂度为 O(n)

longlongint(const longlongint &a);拷贝构造函数，遍历一遍链表，时间复杂度为 O(n)

void get(std::string number);为链表赋值，遍历一遍输入的字符串，时间复杂度为 O(n)

int operator <(const longlongint &number);比较函数，最好的情况是 O(1)，最坏的情况需要遍历一遍链表，时间复杂度为 O(n)

longlongint & operator = (const longlongint & target);重载赋值函数，与拷贝构造函数类似，时间复杂度为 O(n)

longlongint operator + (longlongint another);首先加法重载调用了时间复杂度为 O(n)的比较函数，之后创建了一个与比当前最长链表长度大 1 的链表，时间复杂度为 O(n)，之后分别遍历了两个链表，时间复杂度为 O(n)。因此，加法的时间复杂度为 O(n)。

longlongint operator * (longlongint &another);乘法调用了 min(number1, number2)次加法和赋值函数。时间复杂度为 O(n^2)。

2） 空间复杂度

longlongint(const longlongint &a); 拷贝构造函数新开辟了一个新的空间用以存储复制后的结果，空间复杂度为 O(n)

void get(std::string number); get 函数为链表开辟空间以存储字符串中的数字，空间复杂度为 O(n)

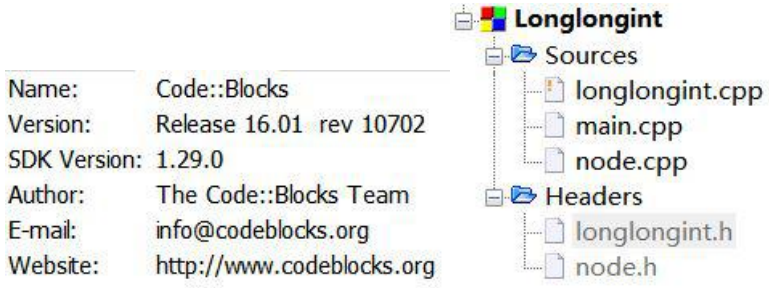
longlongint operator + (longlongint another); 加法重载中建立了新链表作为返回值，复杂度为 O(n)

longlongint & operator = (const longlongint & target);赋值函数构造一个新的链表作为返回值，空间复杂度为 O(n)

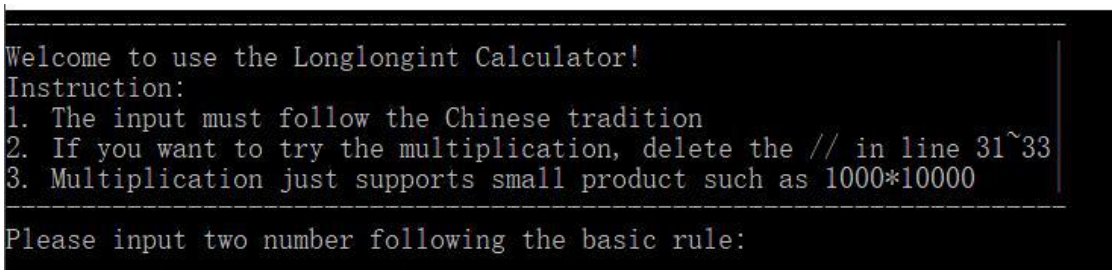
longlongint operator * (longlongint &another); 乘法调用了 min(number1, number2)次加法和赋值函数，空间复杂度为 O(n^2)

五、 用户手册

1. 本程序使用的 Code::Blocks 16.01 IDE，程序以项目（project）方式组织，如图 1 所示：



2. 依次点击菜单“Build”-> “Build and run”，显示文本方式的用户界面，如图 2 所示：



3. 按照中国表述习惯输入相应数字后，如果输入的内容存在非法字符，则返回 invalid input；
4. 输出结果

六、 执行结果

1. 输入为 0;0;

Please input two number following the basic rule:
0;0;

The first number is: 0
The second number is: 0
The sum is: 0

2. 输入为-2345,6789;-7654,3211;

Please input two number following the basic rule:
-2345,6789;-7654,3211;

The first number is: -2345,6789
The second number is: -7654,3211
The sum is: -1,0000,0000

3. 输入为-9999,9999;1,0000,0000,0000;

Please input two number following the basic rule:
-9999,9999;1,0000,0000,0000;

The first number is: -9999,9999
The second number is: 1,0000,0000,0000

The sum is: 9999,0000,0001
4. 输入为 1,0001,0001;-1,0001,0001;
Please input two number following the basic rule:
1,0001,0001;-1,0001,0001;

The first number is: 1,0001,0001
The second number is: -1,0001,0001
The sum is: 0
5. 输入为 1,0001,0001;-1,0001,0000;
Please input two number following the basic rule:
1,0001,0001;-1,0001,0000;

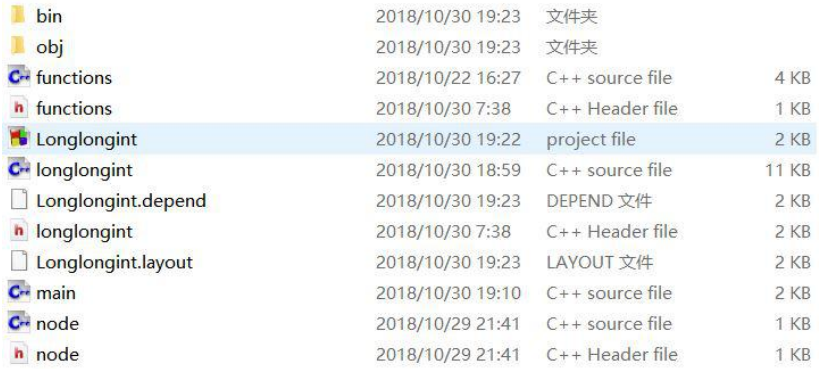
The first number is: 1,0001,0001
The second number is: -1,0001,0000
The sum is: 1
6. 输入为-9999,9999,9999;-9999,9999,9999;
Please input two number following the basic rule:
-9999,9999,9999;-9999,9999,9999;

The first number is: -9999,9999,9999
The second number is: -9999,9999,9999
The sum is: -1,9999,9999,9998
7. 输入为 1,0000,9999,9999;-1,0001,0001;
Please input two number following the basic rule:
1,0000,9999,9999;-1,0001,0001;

The first number is: 1,0000,9999,9999
The second number is: -1,0001,0001
The sum is: 9999,9998,9998

七、 附录

文件夹下文件清单：



其中，functions 中有一些函数的“初稿”和“废稿”，其中 obey 函数是很 naive 的非法输

入识别，这个题目想要把所有的非法输入全部识别(比如 400,500,4000;-1,00000,8000;)实在太困难了，我想到的方法程序都很冗长而且复杂度比较高，所以只实现了一个检验有没有非法字符的函数。这个函数的时间复杂度为 O(n)。