第1.2题：一元稀疏多项式计算器

实验报告

题目：设计一个一元稀疏多项式简单计算器。

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%

1. **需求分析**
2. 按照格式输入多项式。
3. 多项式输出按照指数降序排列，输出形式为类数学表达式。
4. 多项式 a 和 b 相加，建立多项式 a+b。
5. **概要设计**
6. 多项式的存储使用双向链表类，其中，每个节点有两个成员变量：cterm、eterm，分别用来存储系数、指数。其中，cterm和eterm的格式应该根据使用场景设定，而根据提供的测试数据将本程序中的cterm设为double格式，eterm设为int格式。为防止老师之后验证程序发现指数的小数点后数字丢失，特此说明。
7. 本程序的特点是所输入的多项式为乱序、有可能指数相等，而输出需要按照指数递减，所以链表的**查重->将相同指数项合并**和**排序**就变得格外重要。
8. 算法部分，加法的实现比较直接：对两个**已经排序后的链表**进行顺序遍历，将指数相等的项相加，不等的项直接放入链表中。
9. 此外，其他的操作都是链表中比较常见的，构造析构拷贝构造赋值重载等等。
10. 求导和减法属于拓展操作，求导思路很简单：就像人类一样c\_out = c \* e, e\_out = e-1。减法则是基于加法，对减数求相反数即可。
11. **节点类 node**

对象：

node \*next; //链表中指向下一节点的指针。

node \*prev; //链表中指向上一节点的指针。

double cterm; //存储系数

int eterm; //存储指数

操作：

node(); //构造函数，对参数进行初始化。

1. **链表类 linknode**

对象：

node \*head, \*rear; //双向链表的头指针和尾指针，指向链表头尾

int length; //记录链表长

操作：

friend std::ostream & operator <<(std::ostream &os, linknode target);

//重载输出函数，输出多项式

linknode(double \*c, int \*e, int n);

//构造函数，根据得到的数组c和e初始化一个链表

~linknode();

//析构函数，返还链表开辟的空间

linknode(const linknode &a);

//拷贝构造函数，返回一个链表的拷贝，该拷贝占用空间为新开辟的空间

linknode operator+(linknode &another);

//加法，用以得到两个多项式加法运算结果

linknode & operator=(const linknode &target);

//赋值重载，和拷贝函数有点类似，代码都很相似，返回一个linknode的引用

void simplify();

//简化函数。对链表中指数相等的项进行合并

linknode sortme();

//排序函数，将链表中的所有节点按照指数降序排序

linknode derivate();

//求导函数，返回当前多项式的求导结果

linknode linknode::operator-(linknode &another);

//减法函数，返回两个多项式的减法

1. **详细设计**
2. **节点类 node**

*node.h*

class node

{

public:

node(); //构造函数，对参数进行初始化。

node \*next; //链表中指向下一节点的指针。

node \*prev; //链表中指向上一节点的指针。

double cterm; //存储系数

int eterm; //存储指数

};

*node.cpp*

//单纯的初始化

node::node()

{

cterm = 0;

eterm = 0;

}

1. **链表类 linknode**

*linknode.h*

class linknode

{

friend std::ostream & operator <<(std::ostream &os, linknode target);

//重载输出函数，输出多项式

public:

node \*head, \*rear; //双向链表的头指针和尾指针，指向链表头尾

int length; //记录链表长

linknode(double \*c, int \*e, int n);

//构造函数，根据得到的数组c和e初始化一个链表

~linknode();

//析构函数，返还链表开辟的空间

linknode(const linknode &a);

//拷贝构造函数，返回一个链表的拷贝，该拷贝占用空间为新开辟的空间

linknode operator+(linknode &another);

//加法，用以得到两个多项式加法运算结果

linknode & operator=(const linknode &target);

//赋值重载，和拷贝函数有点类似，代码都很相似，返回一个linknode的引用

void simplify();

//简化函数。对链表中指数相等的项进行合并

linknode sortme();

//排序函数，将链表中的所有节点按照指数降序排序

linknode derivate();

//求导函数，返回当前多项式的求导结果

linknode linknode::operator-(linknode &another);

//减法函数，返回两个多项式的减法

};

*linknode.cpp*

std::ostream & operator <<(std::ostream &os, linknode target)

{

node \*temp = target.head->next;

int flag = 0;

//flag 用于记录是否有元素被输出，如果始终为0，则在最后输出一个0

for (int k = 0; k < target.length; k++)

{

if(temp->cterm>0 && temp != target.head->next){os<<"+";}

//输出正数时，cout并不会帮忙加上+号，所以自己加上

if(temp->cterm == 1 && temp->eterm != 1 && temp->eterm != 0)

// c = 1但e != 1时，不需要输出cterm的1

{

os<< "X^" << temp->eterm;

temp = temp->next;

flag++;

continue;

}

if(temp->cterm == -1 && temp->eterm != 1 && temp->eterm != 0)

//c = -1 但e != 1 时，不需要输出-1，只需要-号

{

os<< "-X^" << temp->eterm;

temp = temp->next;

flag++;

continue;

}

if(temp->cterm == 0 && k == target.length-1 && flag == 0)

//当所有元素都已经遍历，但还是没有元素被输出，输出一个0

{

os<<"0";

break;

}

if(temp->cterm == 0)

//当遇到c为0时，跳过不输出

{

temp = temp->next;

continue;

}

if(temp->cterm == 1 && temp->eterm == 1)

//当c和1都是1时，输出X

{

os<<"X";

temp = temp->next;

flag++;

continue;

}

if(temp->cterm == -1 && temp->eterm != 1 && temp->eterm != 0)

//当c和1都是1时，输出-X

{

os<< "-X^" << temp->eterm;

temp = temp->next;

flag++;

continue;

}

if(temp->eterm == 0)

//当e为0时，只需要输出c（为常数）

{

os<<temp->cterm;

temp = temp->next;

flag++;

continue;

}

if(temp->eterm ==1)

//当e为1时，（c!=1 因为前面已经判断过了）， 输出c和X

{

os<<temp->cterm<<"X";

temp = temp->next;

flag++;

continue;

}

//如果前面的所有特殊情况都不是，就很正常的输出cX^e

os<<temp->cterm<<"X^"<<temp->eterm;

flag++;

if(temp != target.rear)

{

temp = temp->next;

}

}

return os;

}

linknode::linknode(double \*c, int \*e, int n)

//构造函数，初始化变量和链表

{

length = n;

head = new node;

head->next = rear = new node;

rear->prev = head;

for (int k = 0; k < n; k++) //构造一个长度为n的链表

{

node \*temp = new node;

temp->cterm = c[k];

temp->eterm = e[k];

temp->next = rear;

temp->prev = rear->prev;

rear->prev->next = temp;

rear->prev = temp; } }

linknode::~linknode()

//析构函数，返还所有开辟的空间

{

node\* temp = head;

while(temp != rear)

//遍历返还空间

{

node \* temp1 = temp;

temp = temp->next;

delete temp1;

}

delete rear;

}

linknode::linknode(const linknode &a)

//拷贝构造函数，新建一个链表，将新建链表的节点值们赋值为现有链表的值

{

length = a.length;

head = new node;

head->next = rear = new node;

rear->prev = head;

node\* iter = a.head->next;

while (iter != a.rear)

//遍历赋值

{

node \*temp = new node;

temp->cterm = iter->cterm;

temp->eterm = iter->eterm;

temp->next = rear;

temp->prev = rear->prev;

rear->prev->next = temp;

rear->prev = temp;

iter = iter->next;

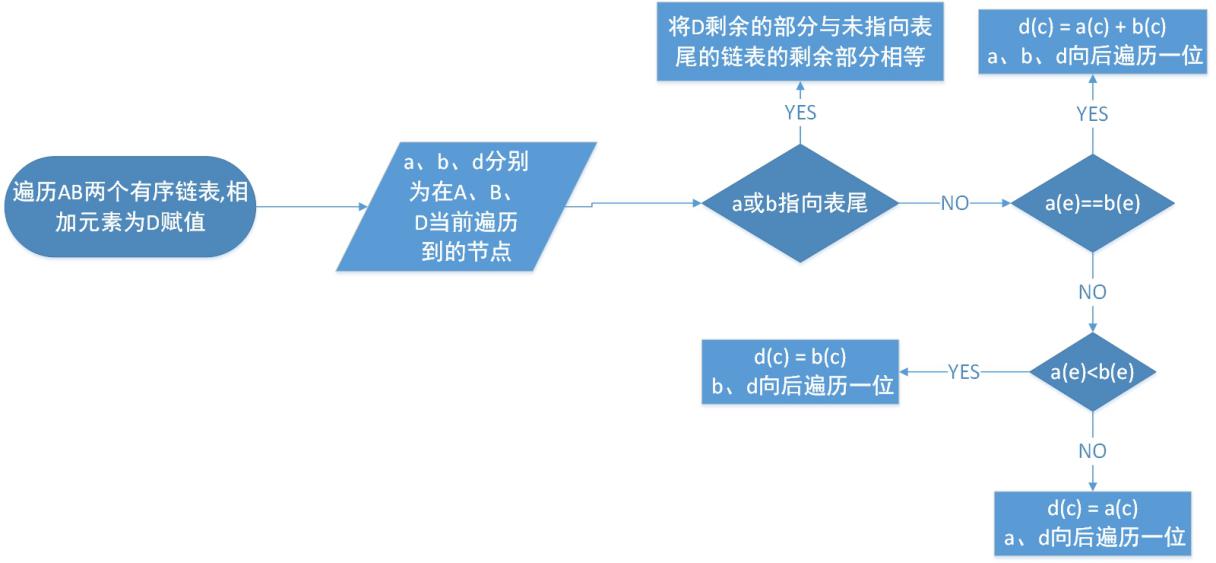
}

}

linknode linknode::operator+(linknode &another)

//整个题目的第一块算法部分，加法重载

//基本思路：



{

int k = length + another.length;

double \*c = new double[k]{0};

int \*e = new int [k]{0};

linknode result(c,e,k);

result.length = 0;

node\* iter\_this, \*iter\_another, \*iter\_target;

iter\_this = this->head->next;

iter\_another = another.head->next;

iter\_target = result.head->next;

for (int iter = 0; iter < k; iter++)

{ //当两个遍历链表的指针有一个指向了表尾时，把result剩余的节点赋值为另一个 //尚未被指向表尾的链表节点。

if(iter\_this == this->rear || iter\_another == another.rear)

{ //两种指向表尾的情况，分开讨论

//其实这里原则上来说可以用三目写，但可读性非常差，于是换用了if-else

if(iter\_this == this->rear)

{

while(iter\_another != another.rear)

{

iter\_target ->cterm = iter\_another->cterm;

iter\_target->eterm = iter\_another->eterm;

iter\_target = iter\_target->next;

iter\_another = iter\_another->next;

result.length++;

}

break;

}

else

{

while(iter\_this != this->rear)

{

iter\_target -> cterm = iter\_this->cterm;

iter\_target -> eterm = iter\_this->eterm;

iter\_target = iter\_target->next;

iter\_this = iter\_this->next;

result.length++;

}

break;

}

}

//第一个判断是否指向表尾的for结束

//接下来判断两个指针指向的指数是否相等

if(iter\_this->eterm != iter\_another->eterm)

{

//不相等则对两链表中较大者指针进行向后遍历，较小者指针不变

//这里用三目更合适一点，可读性高

int judge = iter\_this->eterm > iter\_another->eterm;

iter\_target->cterm = (judge == 1)? iter\_this->cterm : iter\_another->cterm;

iter\_target->eterm = (judge == 1)? iter\_this->eterm : iter\_another->eterm;

iter\_target = iter\_target->next;

iter\_this = (judge == 1) ? iter\_this->next : iter\_this;

iter\_another = (judge == 1) ? iter\_another : iter\_another->next;

result.length++;

continue;

}

else

//两个指针指向的e相等，则三个指针全遍历

{

iter\_target->cterm = iter\_this->cterm + iter\_another->cterm;

iter\_target->eterm = iter\_another->eterm;

iter++;

result.length++;

iter\_another = iter\_another->next;

iter\_this = iter\_this->next;

iter\_target = iter\_target->next;

continue;

}

}

while(iter\_target != result.rear)

//将初始化链表时，多建立的节点（因为最开始假设的表长是length + another.length） //删除

{

node \* temp = iter\_target;

iter\_target = iter\_target->next;

temp->prev->next = iter\_target;

iter\_target->prev = temp->prev;

delete temp;

}

return result;

}

linknode linknode::operator-(linknode &another)

//减法，在有了加法后，减法很好实现

//思路：将原链表的所有值取相反数建立新链表

{

double \* sizeofit1 = new double [another.length];//c

int \* sizeofit2 = new int [another.length];//e

linknode result(sizeofit1,sizeofit2,another.length);

node\* iter\_this= another.head->next;

node\* iter\_target = result.head->next;

while(iter\_this!=another.rear)

{

iter\_target->cterm = -1 \* iter\_this->cterm;

iter\_target->eterm = iter\_this->eterm;

iter\_target = iter\_target->next;

iter\_this = iter\_this->next;

}

return \*this + result;

}

linknode& linknode::operator=(const linknode &a)

//赋值函数重载，和拷贝构造函数的内容基本一样，注意返回的是引用

{

length = a.length;

head = new node;

head->next = rear = new node;

rear->prev = head;

node\* iter = a.head->next;

while (iter != a.rear)

{

node \*temp = new node;

temp->cterm = iter->cterm;

temp->eterm = iter->eterm;

temp->next = rear;

temp->prev = rear->prev;

rear->prev->next = temp;

rear->prev = temp;

iter = iter->next;

}

linknode result(a);

return result;

}

linknode linknode::sortme()

//第二部分算法，对链表进行排序，注意：这里的链表已经执行过simplify了，所以不会

//现具有相同e的节点



{

double \* sizeofit1 = new double [length];//c

int \* sizeofit2 = new int [length];//e

int temp = this->head->next->eterm;

linknode result(sizeofit1,sizeofit2,length);

node\* iter\_this= this->head->next;

node\* iter\_target = result.head->next;

iter\_target->cterm = iter\_this->cterm;

iter\_target->eterm = iter\_this->eterm;

//遍历一遍，得到最大值

for (int k = 0;k < length; k++)

{

if(iter\_this->eterm > temp)

{

temp = iter\_this->eterm;

iter\_target->cterm = iter\_this->cterm;

iter\_target->eterm = iter\_this->eterm;

}

iter\_this = iter\_this->next;

}

//length = 1的时候不需要再继续排序，直接返回

if(length == 1){return result;}

int e\_temp = temp;

double c\_temp = 0;

//每次k的循环取小于last\_biggest的最大值

for (int k =0; k < length -1; k++)

{

iter\_target = iter\_target->next;

iter\_this = this->head->next;

int last\_biggest = e\_temp;

int flag = 0;

for (int b = 0; b < length; b++)

{

//当遇到第一个小于e\_temp的值时，进行存储，flag = 1意味着此时已经找到了小于e\_temp的值，之后的任务是找比它大的，小于e\_temp的最大值

if(iter\_this->eterm < e\_temp && flag == 0)

{

e\_temp = iter\_this->eterm;

c\_temp = iter\_this->cterm;

flag = 1;

}

//当满足小于e\_temp但大于当前存储的小于e\_temp的最大值时，赋值

if(iter\_this->eterm > e\_temp && iter\_this->eterm < last\_biggest && flag == 1)

{

e\_temp = iter\_this->eterm;

c\_temp = iter\_this->cterm;

}

iter\_this = iter\_this->next;

}

iter\_target->eterm = e\_temp;

iter\_target->cterm = c\_temp;

}

return result;

}

void linknode::simplify()

//简化函数，用于简化链表。思路很简单：循环遍历链表，将所有e相等的节点进行合并

{

if(length == 1){return;}

node\* temp1 = this->head->next;

node\* temp2 = temp1->next;

while(temp1 != this->rear)

{

temp2 = temp1->next;

while(temp2 != this->rear)

{

int flag = 0;

//当节点的e相等时，合并节点（c1 = c1 + c2 ），再删除c2节点

if(temp1->eterm == temp2->eterm)

{

temp1->cterm = temp1->cterm + temp2->cterm;

temp2->prev->next = temp2->next;

temp2->next->prev = temp2->prev;

length --;

node\* dele = temp2;

temp2 = temp2->next;

flag = 1;

//这里如果temp2不指向下一个的话，delete后temp2是野指针，无法实 现算法

delete dele;

}

temp2 = flag == 1? temp2 : temp2->next;

}

temp1 = temp1->next;

}

}

linknode linknode::derivate()

//求导函数，返回当前多项式的求导结果

//基本思路也很简单，就像正常的求导一样，对每一项c\_target = e \* c，e\_target = e-1即可

{

double \* sizeofit1 = new double [length];//c

int \* sizeofit2 = new int [length];//e

int temp = this->head->next->eterm;

linknode result(sizeofit1,sizeofit2,length);

node\* iter\_this= this->head->next;

node\* iter\_target = result.head->next;

while(iter\_this != this->rear)

{

iter\_target->cterm = iter\_this->cterm \* iter\_this->eterm;

iter\_target->eterm = iter\_this->eterm - 1; //c\_target = e \* c，e\_target = e-1

if(iter\_this != this->rear->prev)

{

iter\_this = iter\_this->next;

iter\_target = iter\_target->next;

continue;

}

break;

}

return result;

}

1. **调试分析**
2. 最开始的程序中没有考虑到c要取小数，所以c和e都是int。后来看测试数据才发现c可以取小数。不过还好这种格式取错了的情况，改几个变量的定义就可以，修改时间很快。
3. 这是三个作业中最后完成的一个，经历了三个程序之后，我最深的感触就是，链表的构造、析构、赋值重载、拷贝构造都是不能避免地需要自己写的。在这个程序中我曾经试图跳过以上说的那几个函数，最后都失败了。
4. const用的不熟练。比较成熟的代码风格应该是需要合理的使用const的，而我在写代码的过程中常常注意不到这个问题，以后需要注意这方面的问题。
5. 整体而言，整个程序和1.1有点类似，且编程难度低于1.1
6. 算法复杂度分析
7. **时间复杂度**

Node中：

node(); //构造函数，O(1)

Linknode中：

friend std::ostream & operator <<(std::ostream &os,const linknode &target);

//遍历一遍链表并输出，O(n)

linknode(double \*c, int \*e, int n);

//遍历两个数组对链表赋值，O(n)

~linknode();

//遍历链表并释放空间，O(n)

linknode(const linknode &a);

//遍历原链表，创建一个相等链表，O(n)

linknode operator+(linknode &another);

//算法图已放在第七页，相当于遍历一遍两个链表，O(n)

linknode & operator=(const linknode &target);

//和拷贝构造函数类似，O(n)

void simplify();

//简化函数，对每个链表节点都要进行后向遍历，O(n^2)

linknode sortme();

//排序函数，思路很像选择排序，O(n^2)

linknode linknode::derivate()

//求导函数，对链表进行遍历，O(n)

linknode linknode::operator-(linknode &another)

//减法函数，对其中一个链表遍历取相反数，并调用加法，O(n)

1. 空间复杂度

在linknode类中，以下四个函数空间复杂度都为O(n),其他函数全为O(1)

linknode(double \*c, int \*e, int n);

linknode(const linknode &a);

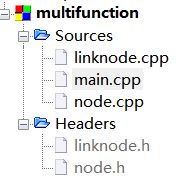
linknode operator+(linknode &another);

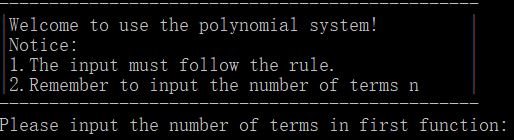
linknode & operator=(const linknode &target);

linknode sortme();

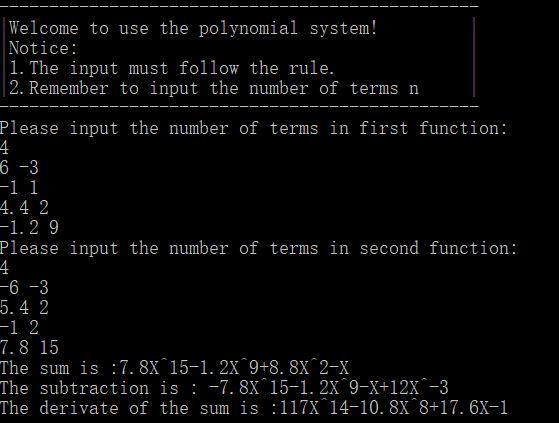
linknode linknode::derivate();

linknode linknode::operator-(linknode &another)

1. **用户手册**
2. 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：
3. 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：

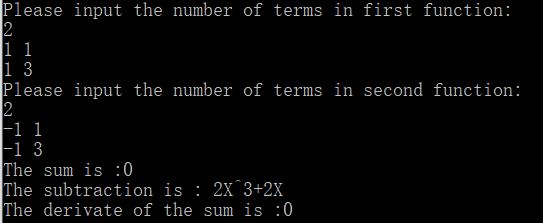
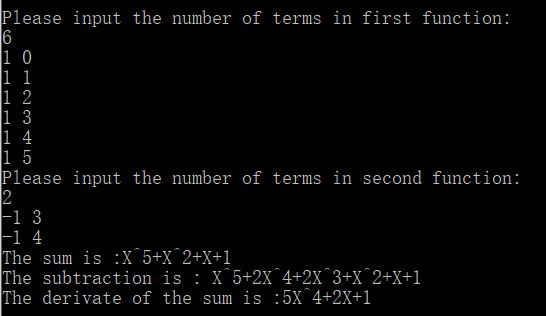
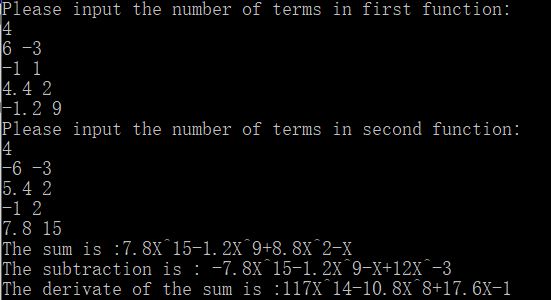
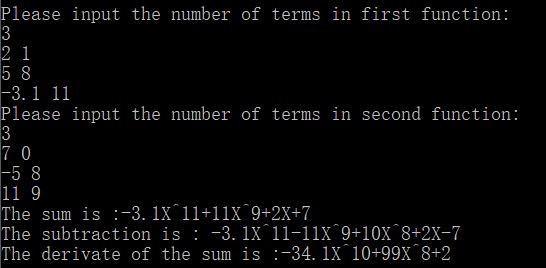


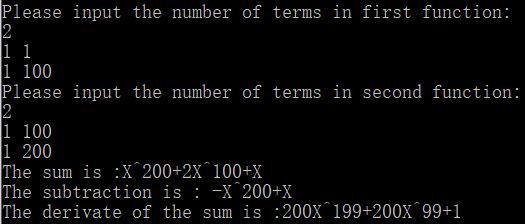
1. 接下来按照要求输入想要的多项式即可得到结果：例如：

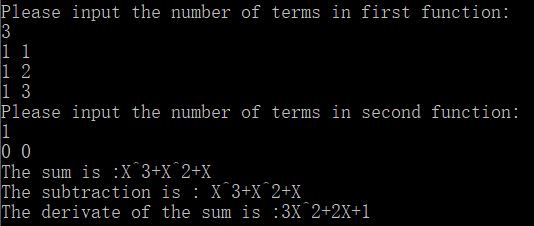


1. **测试结果**

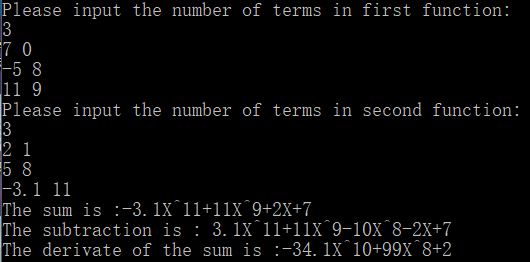
1.正序测试测试数据多项式

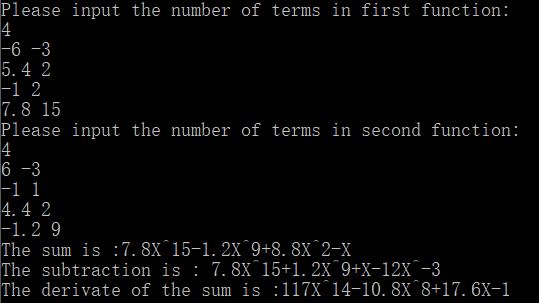


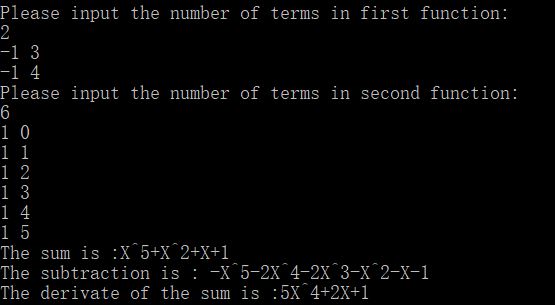


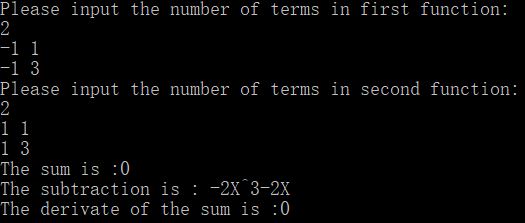


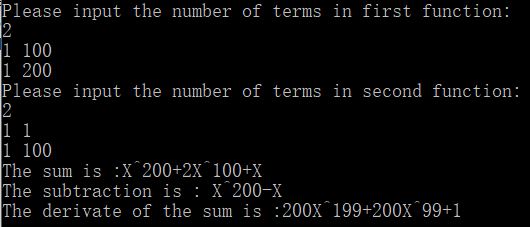
2. 逆序测试测试数据

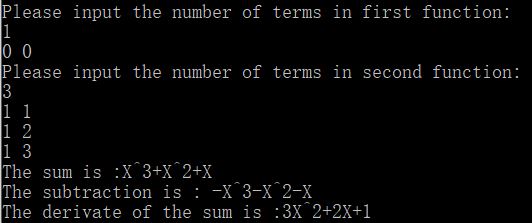












1. **附录**

目录：

