实验一

姓名：黄丹禹

学号：2012030

日期：2022/10/26

一、 实验平台

a) 基于 Visual Studio 2022，使用+语言编写

b) 调用函数库：

```
#include <iostream>
#include <vector>
#include <string>
```

二、 实验目的

多序列操作（对于无限长序列应完车即时允许情况下应完成即时操作，可采取每一个时刻两个序列依次输入的方式）

a) 满足加法操作

b) 满足乘法操作

c) 满足卷积操作

 i. 线性卷积

 ii. 圆周卷积（未实现）

d) 满足序列相似性比对操作

 i. 滑动窗的相似性比对

 ii. 归一化的相似性比对

e) 分析滑动窗序列的效率和优化

三、 实验核心公式及问题

a) 加法操作中，两序列相加前应该使两序列对齐，分别将两个序列的 begin 和 end 进行大小比较，然后相应的进行前后补零操作，补零操作结束后就才可以相加。

b) 注意分辨滑动窗与归一性相似性比对两种不同的方法的差别，注意在调用卷积函数用以方便相似性计算时分别是否要进行序列反转。

四、 实验设计

a） 加法操作

通过对运算符重载’+’完成加法操作，首先进行前后补零，后再按位进行加法：

```
//加法
void operator+(seq & b) {
    //补零
    if (this->begin < b.begin) {
        b = b.supplement(1, b.begin - this->begin);
    }
    else {
        supplement(1, this->begin - b.begin);
    }
    if (this->end < b.end) {
        supplement(0, b.end-this->end);
    }
    else {
        b = b.supplement(0, this->end - b.end);
    }
/* ... */
    //处理
    cout << "result:";
    for (int i = 0; i < a.size(); i++) {
        cout << a[i] + b.a[i] << ' ';
    }
    cout << endl;
}
```

测试数据：A={1,2,3} ，n=0:2 ；B={1,2,3} ，n=-2:0 ：

```
加法结果:result:1 2 4 2 3
```

无限长序列加法操作：

初始化 A={1,2,3} ，n=0:2　　B={1,2,3},n=-2:0 （下图中蓝色框为新输入值，绿色框为新结果值）

```
初始化结果序列:
result:1 2 4 2 3
1
5
result:1 2 4 2 3 6
5
8
result:1 2 4 2 3 6 13
```

b） 乘法操作

通过对运算符重载’-’完成加法操作，首先进行前后补零，后再按位进行乘法，原理和加法相似：

```
//乘法
void operator-(seq & b) {
    //补零
    if (this->begin < b.begin) {
        b = b.supplement(1, b.begin - this->begin);
    }
    else {
        supplement(1, this->begin - b.begin);
    }
    if (this->end < b.end) {
        supplement(0, b.end - this->end);
    }
    else {
        b = b.supplement(0, this->end - b.end);
    }
    //处理
    cout << "result:";
    for (int i = 0; i < a.size(); i++) {
        cout << a[i] * b.a[i] << ' ';
    }
    cout << endl;
}
```

测试数据：A={1,2,3} ，n=0:2 ；B={1,2,3} ，n=-2:0 ：

```
乘法结果:result:0 0 3 0 0
```

无限长序列乘法操作：

初始化 A={1,2,3} ，n=0:2　　B={1,2,3},n=-2:0

过程与加法类似：

```
初始化结果序列:
result:0 0 3 0 0
5
9
result:0 0 3 0 0 45
4
5
result:0 0 3 0 0 45 20
```

c)　线性卷积

思路为翻转 B 序列后滑动，并每次对位相乘后累加，每次滑动得到最后卷积结果序列中的其中一个值，通过循环完成每次相乘累加和滑动，得到 result 数组为卷积的结果。

```
//卷积
seq operator*(seq b) {
    vector<double> result;
    int temp;
    int t2 = 0;
    for (int i = 0; i < this->a.size() + b.a.size() - 1; i++) {
        for (int j = 0; j < this->a.size(); j++) {
            if (((i - j )> (b.a.size()-1)) || (i - j < 0)) {
                temp = 0;
            }
            else temp = b.a[i-j];
            t2=t2+ temp * a[j];
        //  result.push_back(temp * a[j]);
        }
        result.push_back(t2);
        t2 = 0;
    }
    for (int i = 0; i < result.size(); i++) {
        cout << result[i] << ' ';
    }
    cout << endl;
    seq t(0, result.size(), result);
    return t;
}
```

以上课 PPT 中例子为测试样例： A={1,2,3} n=0:2 B={1,2,3} ， n=0:2

```
卷积结果:1 4 10 12 9
```

结果与手动计算的卷积结果一致。

测试数据：A={1,2,3} ， n=0:2 B={1,2,3} ， n= -2:0

```
卷积结果:0 0 1 4 10 12 9
```

无限长卷积操作：

确定卷积核为 B={1,2,3} ， n= 0:2 A 初始化为{1,2,3} ， n= 0:2

向其中输入后处理结果：

```
初始化结果序列:
3 8 14 8 3
5
3 8 14 23 13 5
8
3 8 14 23 37 21 8
9
3 8 14 23 37 48 26 9
```

d)　滑动窗的相似性比对

由上课讲授与课下查看教材得知，离散序列的相似性比对与卷积计算相似，即将 B 序列进行反转后与 A 序列卷积的结果：

研究式(2.67)，可以看到互相关的表达式与式(2.20a)的卷积表达式相似。若重写式(2.67)，这种相似性将会更加清楚：

$$r_{xy}[\ell] = \sum_{n=-\infty}^{\infty} x[n]y[-(\ell - n)] = x[\ell] \circledast y[-\ell] \qquad (2.70)$$

由上面可以推出，序列 $x[n]$ 与参考序列 $y[n]$ 的互相关就是序列 $x[n]$ 与 $y[-n]$（$y[n]$ 的时间反转形式的卷积和。同样，$x[n]$ 的自相关就是它自己与其时间反转形式的卷积和。

```
//滑动窗的相似性比对
void operator&(seq b) {
    seq temp = b.reverse();
    *this* temp;     //将this与反转后的b 两个序列卷积
}
```

测试数据：A={1,2,3} ，n=0:2　　B={3,2,1} ，n= -2:0

此处测试数据为卷积测试数据 2 的 A 与 B 的反转，所以此处测试的结果应该与卷积测试数据 2 结果相同，得到的结果符合预期：

```
相似性结果:0 0 1 4 10 12 9
```

不能操作无限长序列。

e)　归一化的相似性比对

先对序列 A，B 进行归一化，然后再翻转卷积，归一化公式如下并进行编程：

- 对一个向量 $A = \{A_i\}$，正规化

$$K = \sqrt{\sum A_i^2},\ A_i' = \frac{A_i}{K}$$

```cpp
//归一化的相似性比对
void operator^(seq b) {
    /* ... */
    double temp1 = 0;
    double temp2 = 0;
    for (int i = 0; i < a.size(); i++) {
        temp1 = temp1 + a[i] * a[i];
    }
    for (int i = 0; i < b.a.size(); i++) {
        temp2 = temp2 + b.a[i] * b.a[i];
    }
    double temp3 = sqrt(temp1);
    vector<double> tempA;
    vector<double> tempB;
    for (int i = 0; i < a.size(); i++) {
        tempA.push_back(a[i] / temp3);
    }
    temp3 = sqrt(temp2);
    for (int i = 0; i < b.a.size(); i++) {
        tempB.push_back(b.a[i] / temp3);
    }
    seq A2(this->begin, this->end, tempA);
    seq B2(b.begin, b.end, tempB);
    cout << "归一化原序列A:" << endl;
    A2.display();
    cout << "归一化原序列B:" << endl;
    B2.display();
    cout << "归一化后结果:";
    A2 & B2;
}
```

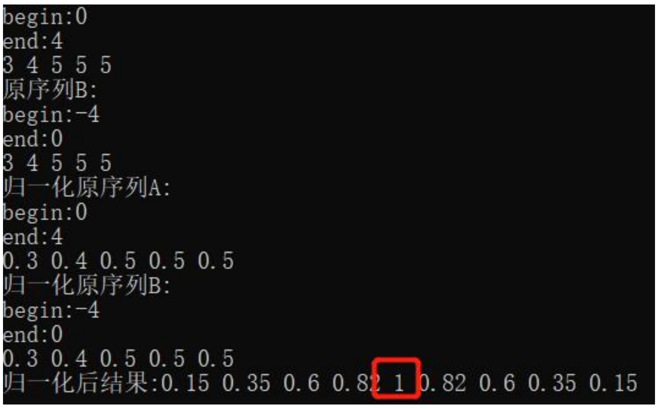
测试数据：A={3,4,5,5,5} ，n= 0:4　　　　B={3,4,5,5,5} ，n= 0:4

两个序列完全相同，相似度为 1：

```
原序列A:
begin:0
end:4
3 4 5 5 5
原序列B:
begin:0
end:4
3 4 5 5 5
归一化原序列A:
begin:0
end:4
0.3 0.4 0.5 0.5 0.5
归一化原序列B:
begin:0
end:4
0.3 0.4 0.5 0.5 0.5
归一化后结果:0.15 0.35 0.6 0.82 1 0.82 0.6 0.35 0.15
```

测试数据：A={3,4,5,5,5} ，n= 0:4　　　　　B={3,4,5,5,5} ，n= -4:0

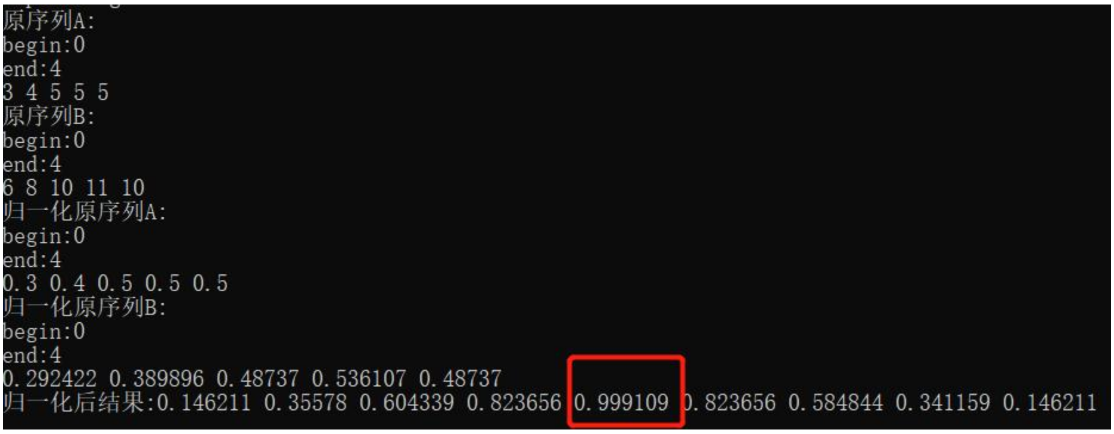
两个序列数据相同但是起始点不同，相似度也为 1：



```
begin:0
end:4
3 4 5 5 5
原序列B:
begin:-4
end:0
3 4 5 5 5
归一化原序列A:
begin:0
end:4
0.3 0.4 0.5 0.5 0.5
归一化原序列B:
begin:-4
end:0
0.3 0.4 0.5 0.5 0.5
归一化后结果:0.15 0.35 0.6 0.82 1 0.82 0.6 0.35 0.15
```

测试数据：A={3,4,5,5,5} ，n= 0:4　　　　　B={6,8,10,11,10} ，n= 0:4

B 序列由 A 变来，为 A 序列放大两倍后叠加{0,0,0,1,0}后的结果，相似度如图：



```
原序列A:
begin:0
end:4
3 4 5 5 5
原序列B:
begin:0
end:4
6 8 10 11 10
归一化原序列A:
begin:0
end:4
0.3 0.4 0.5 0.5 0.5
归一化原序列B:
begin:0
end:4
0.292422 0.389896 0.48737 0.536107 0.48737
归一化后结果:0.146211 0.35578 0.604339 0.823656 0.999109 0.823656 0.584844 0.341159 0.146211
```

不能操作无限长序列。

f) 分析滑动窗序列的效率和优化

使用滑动窗序列可以提高效率，不再需要遍历整个序列，只需要使用滑动窗中的数据即可，其他数据可以被丢弃，数据实现即来即走，可以减缓内存压力。

可以多次尝试得到滑动窗的最佳长度（或通过机器学习获取），过长会导致所需数据量过多计算复杂，长度过短会导致结果不准确，处理效果不佳。

可以精心设计滑动窗的内容，如图像处理中的卷积核，达到不同种类、不同程度的处理需求（如模糊处理的程度等）。