CS205 C/C++ Program Design Project2

> Name:章志轩
>
> SID:12010526

> Time:2023/03/20

Abstract: Java是由C++编写的高级编程语言，有着许多优秀的特性。本次项目旨在通过计算向量点积的性能来对比Java和C++两种编程语言的特点，产生更深刻的理解。

I. 项目介绍

在这个项目中，我们的目标是实现和比较两个计算两个向量的点积的程序。一个程序将用C或C++实现，另一个程序将用Java实现。向量的数据类型可以是int、float、double或其他类型。

II. 环境配置

Win10系统，AMD 锐龙 5 4600U处理器；Java： JDK 17.0.1; Cpp：c++11

III. 代码设计

在本次项目中我着眼于整型数和浮点数的运算。同时由于目的在于对比两种语言的特点，为减小硬件性能影像，我只会使用单线程。但在其它方面我则会尽可能提高运算性能。

a) Java

在大部分操作系统中基础运算大多依靠32位或64位寄存器进行。在Java中short是16位整型数，它在运算中会被转为32位的int类型整型数。故short只有在存储方面有优势，而在运算上不如int(short < int)。同时受硬件影响我无法肯定int和long(64位整型数)哪个运算效率更高，这取决于操作系统和编译器，同样现象存在于float(单精度浮点数)和double(双精度浮点数)。因此我计划对比int、long、float、double和混合不同参数类型向量的效率。为此我设计了`MyVector`类：

```
class MyVector {
    /* 由于不知道哪个更快，我决定都试一试(short除外，short不如int) */
//    private short[] shorts;
    private int[] ints;
    private long[] longs;
    private float[] floats;
    private double[] doubles;
    ...
    int size;
    int maxSize;
    /* int=3 < long=4 < float=5 < double=6 故通过类名可以区分type*/
    int elementType;
    String name;
    public MyVector(int size) {
        this.maxSize = size;
        ints = new int[maxSize];
        this.elementType = 3;
        this.size = 0;
```

```
        mapTable();
    }
}
```

类中设有四种类型的数组，是由于java类型固定不容易修改，而数组有着良好的性能。`size`和`maxSize`指向量当前大小和设置的最大长度(在构造器中传入)。`name`是向量的名字，方便我打印。`MyVector`的设计理念是根据添加的数据类型，动态的提升存储类型，以此在单独类型(如全为int)有着更高的效率，方便进行不同数据类型性能对比。`elementType`就是指向当前存储数据类型的标签，根据类型的长度可表示为：int(3), long(4), float(5), double(6)，也方便理解代码。当`MyVector`被实例化，初始的数据类型是int。

```
<T> void add(T e) {
        ...
        // 如果碰到short就转成int
        if(type.equals("short"))
            type = "int";
        ...
        // typeVal是输入元素e的类型
        if (elementType < typeVal) {
            // 实际上只有6种类型转换方式，只能从低到高如ints -> double,不存在double->float
            switch (elementType+typeVal) {
                case 7 -> {
                    longs = new long[maxSize];
                    longs = Arrays.stream(ints).mapToLong(i -> i).toArray();
                    // 错误的方法
                    // System.arraycopy(ints, 0, longs, 0, size);
                    ints = null; // 指向null，回收内存
                }
                // 9有两种：int(3)->double(6) 或 long(4)->float(5)
                case 8,9,10,11 -> 类似于7,实际实现略有区别
                default -> 打印错误
            }
            elementType = typeVal;
        }
        switch (elementType) {
            case 3 -> ints[size++] = Integer.parseInt(e.toString());
            case 4,5,6 -> 类似于3，实际实现略有区别
            default -> System.err.println("未知错误，出现在数组类型指针，超出[3,6]范围。");
        }
    }
```

`add`函数使用了泛类，允许多类型输入降低了代码复杂度。根据输入的数据类型选择是否提升存储数据类型。在检查输入类型后判断是否需要提升存储数据类型，将数组元素复制到新数组中。最后完成元素添加。由于类型只能往高的提升，旧的数组会直接指向`null`，方便被内存回收。

```
double dotProduct(MyVector v) {
        ...
        double result = 0;
        // 映射双方数据类型到 [6,21]中，<<2 等价于 ×4
        int xType = (this.elementType<<2)-v.elementType;
```

```
        switch (xType) {
            case 6 -> {
                for (int i = 0;i < size; i++)
                    result += this.ints[i] * v.doubles[i];
            }
            case 7~21 -> 类似于6
            default -> System.err.println("未知错误，出现在点乘时数组类型指针，超出[3,6]范围。");
        }
        return result;
    }
```

`dotProduct`函数实现了两个向量的点乘。根据前面的思路，我们要对两个实例向量的两个数组进行点乘，但每个向量都有4种情况(ints,longs,floats,doubles)，这就导致了有4×4=16种相乘方式。为简化判断方式我设计了`xType`变量，它由前一个向量类型的4倍减后一个向量类型，这样就可以将所有相乘情况映射到6~21中，再用switch进行跳转即可。值得注意的是该函数返回类型是double，这是因为高维向量点乘容易超出范围，这里其实是被迫使用double作为返回值。

值得注意的是该类并不是十分完善，它没有实现`delete`方法和`set`方法，只有`add`和`get`方法可以使用。输入的参数也只有short,int,longs,float,double有效，输出结果为double，存在范围限制-4.9e-324~1.8e+308。为避免数据超出范围(本次项目重点是测试效率)，我给各类型都限制了范围：

```
maxInteger = Short.MAX_VALUE + 1; minInteger = Short.MIN_VALUE;
maxLong = Integer.MAX_VALUE + 1L; minLong = Integer.MIN_VALUE;
maxFloat = 1.8e19f;     minFloat = -1.8e19f;
maxDouble = 1.3e+150;   minDouble = -1.3e+150;
```

b) C++

代码设计思路近似于java，但在向量元素上我使用了union联合体`Data`，它里面保存四种类型的数据，方便我选择性的插入值。

```
union Data
{
    // short shortData;
    int intData;
    long long longData;
    float floatData;
    double doubleData;
};
```

向向量随机添加值。用`Data`数组`vector`保存数据，`vType`是byte数组记录对应位置是什么类型的数据，`size`指向量长度，`limit`指限定的数据类型(0~5->short/int/long/float/double,其它数字代表不限定数据类型)。同时为保证结果不超范围，c++同样限定了数据范围，与java相同。

```
void generatingRandomNumber(Data * vector, byte * vType, int size, int limit) {
    Data *p = vector;
    byte *pType = vType;
    int type, symbol;
    int shortRange = SHRT_MAX - SHRT_MIN + 1;
```

```
    uniform_real_distribution<float> f(-1.8e19f, 1.8e19f);
    uniform_real_distribution<double> d(-1.3e+150, 1.3e+150);
    default_random_engine e(time(NULL));
    for (size_t i = 0; i < size; i++)
    {
        /* [0,5)-> short\int\long\float\double */
        type = rand() % 5;
        symbol = (rand() % 2) ? 1 : -1;
        switch (type)
        {
        case 0:
        case 1:
            p->intData = SHRT_MIN + rand() % shortRange;
            *pType = (byte) 0x01;
            break;
        case 2:
            p->longData = symbol * rand();
            *pType = (byte) 0x02;
            break;
        case 3:
            p->floatData = f(e);
            *pType = (byte) 0x03;
            break;
        case 4:
            p->doubleData = d(e);
            *pType = (byte) 0x04;
            break;
        default:
            i--;
            break;
        }
        p++;
        pType++;
    }
}
```

点乘与java思路相同，由于数据类型有4种(short按int存)，故有4*4=16种情况，用`switch`处理。

```
double dotProduct(Data * vector1, byte * v1Type, Data * vector2, byte *v2Type, int
size) {
    double result = 0;
    int xType;
    for (size_t i = 0; i < size; i++) {
        xType = (((int) v1Type[i])<<2) - (int) v2Type[i];
        switch (xType)
        {
        case 0:
            result += vector1[i].intData * vector2[i].doubleData;
            break;
        case 1~15：与0类似
        }
```

```
    }
    return result;
}
```

IV. 性能分析

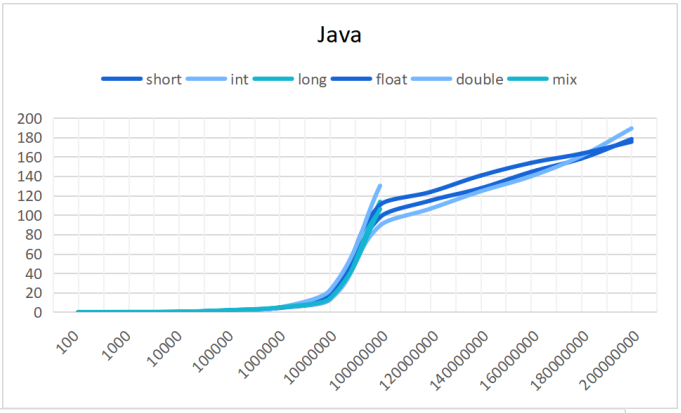
两份代码思路相同(实现逻辑相近)，环境相同，语言不同，适合进行性能对比。需要注意的是项目注重的是点乘的效率，也就是说运算的效率，但我将生成随机向量的时间一同记录了下来当作扩展探究内容。需要注意的是我期望的实验结果是不同数据类型有不同的效率，为此设计混合数据类型mix将有助于提高点乘效率。(mix指向量每一个值的数据类型都是short/int/long/float/double之间随机一种)。

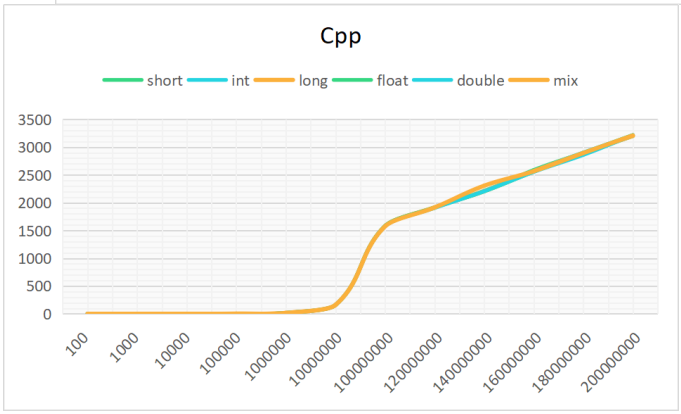
备注：java使用的System.nanoTime()计时，Cpp使用std::chrono::high_resolution_clock::now()计时。java可以精确到纳秒，chrono可以精确到微秒，但有限定范围且无法保留小数。因此在Cpp实验中当时间>1000ms，则自动换用ms计时，否则使用us计时。当然为方便对比在表格中单位统一为ms。

实验1：不同数据类型对点乘运算效率的影响

语言：Java	单位：ms	精度：ns	测试三次	取平均			方差
size type	short	int	long	float	double	mix	VAR
100	0.005466667	0.006166667	0.010133333	0.012966667	0.008066667	0.006933333	7.92785E-06
1000	0.0246	0.020166667	0.021066667	0.0299	0.033166667	0.022866667	2.67455E-05
10000	0.169533333	0.164966667	0.303966667	0.2692	0.240233333	0.205566667	0.003095527
100000	1.7111	1.709266667	1.936166667	1.676966667	1.655233333	1.947566667	0.017616267
1000000	4.255933333	4.561466667	4.497533333	4.475466667	4.709066667	4.825466667	0.039199027
10000000	12.86866667	12.92036667	14.23886667	17.58093333	22.2619	13.9391	13.5099119
100000000	97.80793333	89.19783333	113.686	110.4235	130.1518	106.018433	198.0294817
120000000	114.8603	106.4981	OutOfMemory	123.7044333	OutOfMemory	OutOfMemory	74.03383167
140000000	127.5585333	124.4025333	OutOfMemory	140.5997	OutOfMemory	OutOfMemory	73.73009534
160000000	144.3609337	139.8275	OutOfMemory	153.7030667	OutOfMemory	OutOfMemory	50.05980335
180000000	158.1577333	160.7595667	OutOfMemory	163.2179	OutOfMemory	OutOfMemory	6.403037694
200000000	178.2202667	189.2266	OutOfMemory	175.4878667	OutOfMemory	OutOfMemory	52.89302947

语言：Cpp	单位：ms	精度：us	测试三次	取平均			方差
size type	short	int	long	float	double	mix	VAR
100	0.001	0.001	0.001	0.001	0.001	0.001	0
1000	0.015	0.015	0.015	0.015	0.015	0.015	0
10000	0.156333333	0.153	0.161666667	0.155	0.155	0.153	1.03111E-05
100000	1.558666667	1.811333333	1.552	1.555333333	1.567	1.646	0.010502863
1000000	15.55166667	16.06366667	15.976	16.04933333	15.885	15.54233333	0.057195822
10000000	157.9343333	158.0876667	159.421	157.9676667	158.091	158.1663333	0.320905556
100000000	1577.333333	1575.333333	1577	1580	1573	1572	8.785185185
120000000	1913.666667	1912.666667	1914.333333	1913	1903.666667	1911.666667	15.54444444
140000000	2204	2214	2206.333333	2202.666667	2204.333333	2304.333333	1619.174074
160000000	2587.333333	2569.666667	2566.333333	2555.666667	2553	2565	149.0555556
180000000	2896.666667	2901	2891.666667	2891.666667	2858.666667	2891.666667	228.6074074
200000000	3197	3208	3200.666667	3217	3212.333333	3209.333333	54.81851852



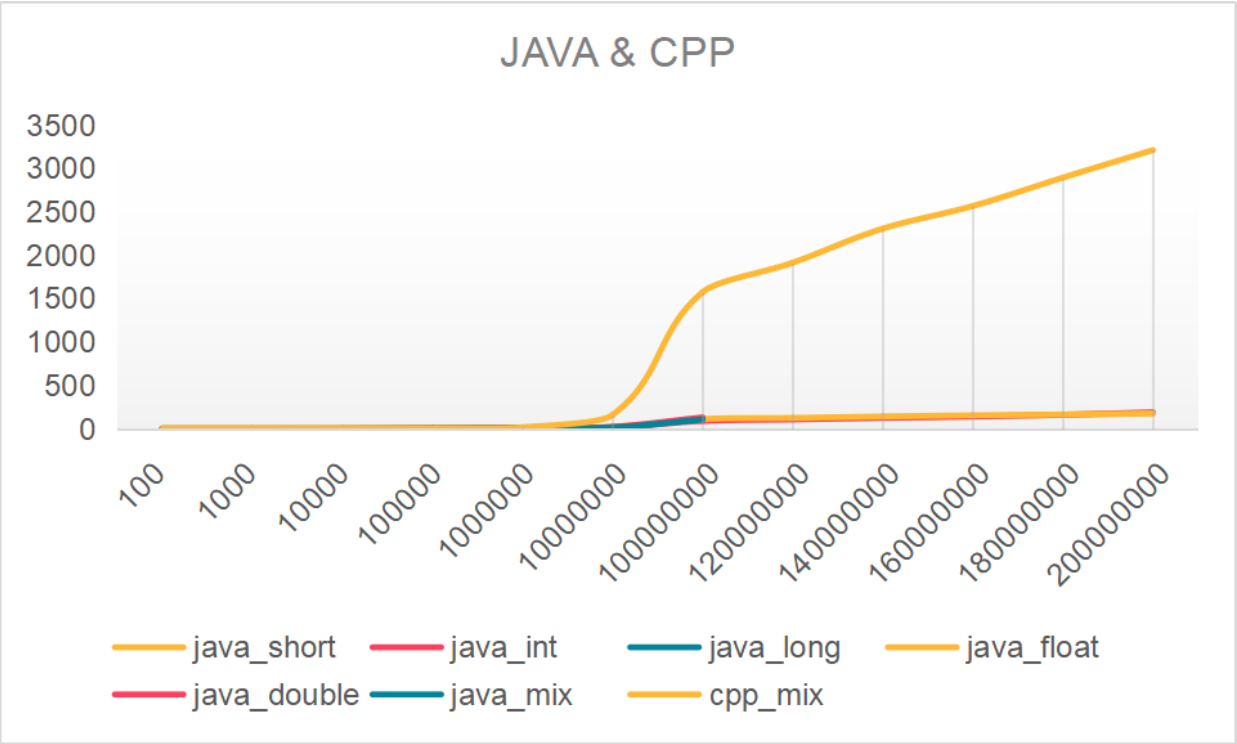


从方差可以看出对于java语言，不同数据类型在数据量小的时候运算开销基本相同，但随着数据量增大方差也增大。增大一方面是因为数据值变大，浮动自然变大，但通过曲线图可以看出各类型确实存在偏差，表现为double > float > long > mix > short ≈ int。其中我将short保存为int，所以两者相近很正常，long是64位比int32位慢也可以理解，但是据我所知在现在64位系统下，long并不一定比int慢，还需要看操作系统和编译器。至于double > float也是一样的道理，且现在有浮点数专门的ALU，所以浮点数的快慢也不能仅凭位数决定。但一般浮点数>整型数(浮点数储存形式更复杂，涉及到指数位的进退，位运算更复杂)。

而对于Cpp而言，结果却不大相同。各数据类型线条相对重合，方差随数据增大提升的也不多，再结合曲线图可以看出数据类型对Cpp程序运算效率影响不大。而且cpp的曲线表现出高度的线性，随向量长度增长正比例增长，而java的增长速率却不确定。原因是cpp编译后生成的可执行文件，执行效率更高，而java生成的是字节码为解释性语言，需要JVM对字节码进行额外的解释，还需要管理内存、满足安全机制(检查越界等)和完成一些高级特性。这些都需要编译器进行额外开销，导致时间的不确定性。

综上，可以看出在该实验环境下(见p1)数据类型对程序运算效率影响有限，在java可以提高一定的效率，但对比起增加的代码复杂度显得有些不值得，对于cpp程序，数据类型并不能带来运算效率上的变化。同时由于cpp程序的执行时间更好预测，在追求高性能时更为方便。

实验2：不同语言对点乘效率的影响



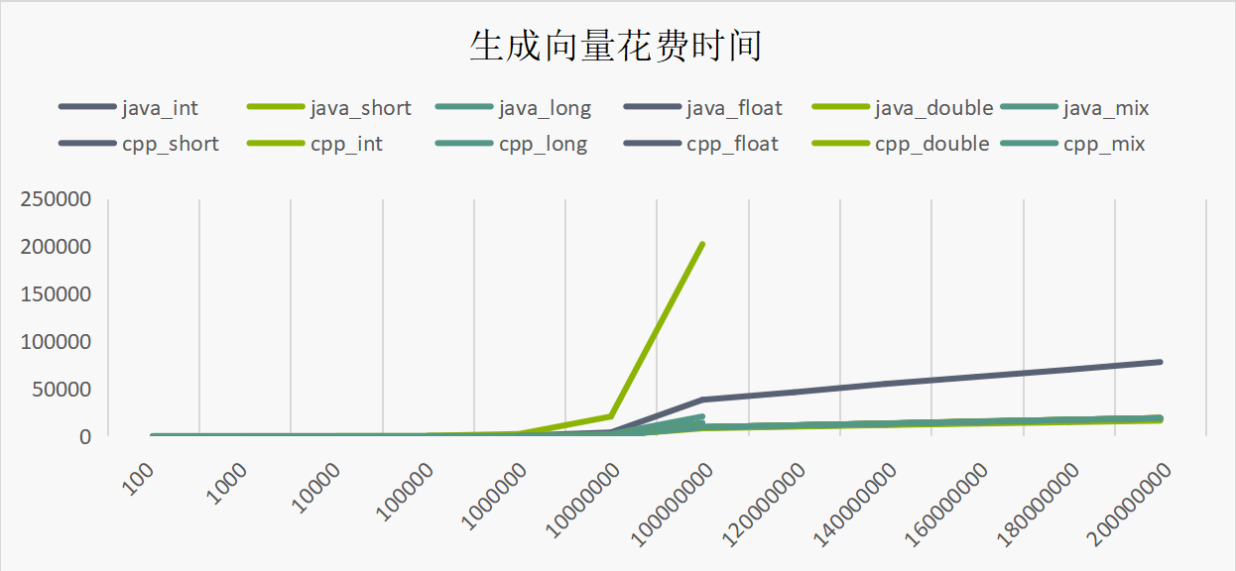
cpp选取最接近平均值的mix线与java的运算效率进行对比，可以很明显看出cpp的运行时间远大于java运行时间。就直观感受来说，这是不合理的因为cpp的运行时间一般都是远快于java程序的。但经过调研，我发现问题出在JIT(Just in Time)编译器上，JIT又称为即使热点引擎，它会在执行代码的时候让JVM记住被反复使用的机器码代码段，在后续调用中可以快速重用。也因此这可以看成是java编译器的动态优化特性，使得它的运行效果反超cpp。

(扩展)实验3：不同语言下生成随机向量

语言：Java	单位：s	精度：ns	测试三次	取平均			方差
size type	short	int	long	float	double	mix	VAR
100	0.0013081	0.000974867	0.006686633	0.004928967	0.012914133	0.0045228	1.90491E-05
1000	0.0041957	0.004124067	0.010327567	0.0213024	0.032608433	0.012059333	0.000122089
10000	0.012638033	0.0129298	0.024577267	0.0423494	0.122169533	0.0368438	0.001692915
100000	0.039511767	0.0388005	0.067190833	0.123572933	0.3946208	0.115822	0.018143383
1000000	0.125373033	0.133558967	0.2305928	0.500347967	2.322468567	0.426026533	0.716640798
10000000	0.899900633	0.946394967	1.528909233	3.9337354	20.66403023	2.266913167	59.8499872
100000000	8.263537133	8.905747567	14.14753347	38.2478529	201.8892648	20.71744903	5754.356848
120000000	10.097924	10.56740353	OutOfMemory	46.18420577	OutOfMemory	OutOfMemory	428.4994574
140000000	11.7605718	12.1201379	OutOfMemory	55.02761207	OutOfMemory	OutOfMemory	618.8695668
160000000	13.27228527	13.84070833	OutOfMemory	62.474451	OutOfMemory	OutOfMemory	797.7361906
180000000	14.74537017	15.82239517	OutOfMemory	69.9179201	OutOfMemory	OutOfMemory	995.2493445
200000000	16.32242237	17.41262147	OutOfMemory	77.94304197	OutOfMemory	OutOfMemory	1243.703517

语言：Cpp	单位：ms	精度：ns	测试三次	取平均		方差	
size type	short	int	long	float	double	VAR	
100	0.01	0.009	0.008666667	0.008	0.009	0.008666667	4.2963E-07
1000	0.105333333	0.088	0.094666667	0.086	0.087333333	0.087	5.62852E-05
10000	0.927	0.929333333	0.901333333	0.908333333	0.943666667	0.952666667	0.000389841
100000	9.82	9.418333333	9.514666667	9.317	9.444333333	9.334	0.033918907

语言：Cpp	单位：ms	精度：ns	测试三次	取平均		方差	
1000000	93.36333333	92.30866667	92.859	92.58566667	92.4	93.20066667	0.185513496
10000000	948.223	922.6253333	918.2303333	920.4533333	921.8466667	918.404	132.978038
100000000	9520	9454	9471	9463.666667	9511.333333	9464	771.7777778
120000000	11391.66667	11441	11354.33333	11465.33333	11377	11418.66667	1719.555556
140000000	13415	13235	13327.33333	13213	13300.66667	13246.66667	5592.685185
160000000	15461.66667	15609.33333	15276	15274	15308.66667	15238.66667	20860.10741
180000000	17243.33333	17198.66667	17131.66667	17197.33333	17225.66667	17245.66667	1800.32963
200000000	19100.33333	19112.66667	18940	19101	19099.66667	18999	5153.407407



通过图可以看到，在java，不同数据类型生成的效率有着天差地别，浮点数的生成要远慢于整型数，而高位数据也慢于低位数据。这是因为Java中的随机数生成器使用的是伪随机数算法，它依赖于一个称为"种子"的值。种子是一个整数，用于生成整数随机数，但是它不能用于生成浮点数随机数。因此，生成浮点数随机数的过程要比生成整数随机数稍微复杂一些，因此效率会慢一些。而C++中的随机数生成器使用的是一种称为伪随机数生成器（PRNG）的技术，它使用一个称为种子的初始值来生成一系列的数字。这些数字都是伪随机的，因为它们是由一个固定的算法生成的，所以它们的效率是相同的，无论是浮点数还是整型数。

V. 总结&收获的经验

通过上述三个实验，我发现了java在不同数据类型的运算与生成中有着不同的效率，运算上越复杂的数据类型越是需要花费更多的时间，其与硬件以及java的特性(安全机制、高级特性等)有关；而生成随机数则是与java随机数生成原理有关，且时间增幅十分明显。cpp则表现出更为稳定的特性，不同数据类型表现出相近的效率，且为高度线性。然而我还发现java在进行点乘运算时速度要远快于cpp，这是由JIT提供的代码优化；但在生成数据方面java又远慢于cpp，这是由于双方本身执行效率以及随机数生成方法决定。

为此，我明白了java程序不一定会慢于cpp程序，丰富的特性基于java多方面的优势；但cpp程序在设计上有着更大的自由，有着更可预测的时间以及在执行上大多情况下更优秀的效率，有着更多的可能性。没有哪个语言更优秀的说法。在未来学习与工作中，我们需要根据自己的需求、偏好决定自己想要的语言。

在完成课程项目中，我也遇见许多困难，它们同样给予了我启发。在设计代码时，我希望程序能够自己选择合适的数据类型保存。因为我知道"="右边是先根据自己的数据类型运算再赋值给左边的变量(测试混合数据类型向量的速度时有用)。为此我想到了Vector类，然后我失败了。首先Vector类并不比ArrayList之类的方便，性能也差很多，而且不指定类型在相加时还是需要自己判断然后强转。总之在性能和书写难度方面都很糟糕，最后还是回归数组(快且方便)。这也让我明白，代码不应该过度追求完备性与复杂性，有时候简单而重复的数据结构反而显得更优雅、更效率、更好理解。同时在思考数据类型对效率的影响时，我查阅了很多资料，这让我对基本数据类型有着更深的理解，明白不是数据越小就越快，数据越大就越全面。结合实验结果，我发现最终数据类型的选择还是需要根据实际情况来，在内存与效率间寻找平衡点。

VI. 源码

https://github.com/15775011722/CS205-Sustech-Cpp/tree/main/project/project2