CS205 C/C++ Programming - Project_2

Name: 刘乐奇（Liu Leqi）

SID：12011327

Part 1 - Analysis

Matries multiplication is a hard but significant in algebra. In this project, it refers to file I/O, data storage and calculation precision and time cost difference between `double` and `float`. I deside to use and one-dimentional dinamic array.

After all, if users want to calculate for more than once, the design of loop can make it.

Part 2 - Code

- head files

```
/*IO.hpp used to input and output the matrices*/

#pragma once

#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <iomanip>
using namespace std;

int getRow(const string filename);
int getCol(const string filename);
void inputdo(double *matrix, int row, int col, const string filename);
void outputdo(double *matrix,int row,int col,const string filename);
void inputfl(float *matrix, int row, int col, const string filename);
void outputfl(float *matrix,int row,int col,const string filename);
```

```
/*matrix.hpp used to create matrices and release, containing matrix
multiplication*/

#pragma once

class flMatrix {
public:
    int rowA, colA, rowB, colB;
    float *matrixA = new float[rowA * colA]();
    float *matrixB = new float[rowB * colB]();
    float *matrixRes = new float[rowA * colB]();

    flMatrix(int a, int b, int c, int d);

    void flMul(float *matrixA, int rowA, int colA, float *matrixB, int rowB, int
colB, float *mul);
```

```
    ~flMatrix();
};

class doMatrix {
public:
    int rowA, colA, rowB, colB;
    double *matrixA = new double[rowA * colA]();
    double *matrixB = new double[rowB * colB]();
    double *matrixRes = new double[rowA * colB]();

    doMatrix(int a, int b, int c, int d);

    void doMul(double *matrixA, int rowA, int colA, double *matrixB, int rowB,
int colB, double *mul);

    ~doMatrix();
};
```

- source files

```
/*IO.cpp*/

#include "../inc/IO.hpp"

int getRow(const string filename)
{
    ifstream infile;
    infile.open(filename);
    if(!infile.is_open()){
        cout << "Could not open the file " << filename << endl;
        cout << "The program is terminated." << endl;
        exit(EXIT_FAILURE);
    }
    int rowcnt = 0;
    string c;
    getline(infile, c, '\n');
    while (infile.good())
    {
        rowcnt++;
        getline(infile, c, '\n');
    }

    cout << rowcnt << endl;

    if(infile.eof()) cout << "The end of file " << filename << " is reached" <<
endl;
    else if(infile.fail()) cout << "failure" << endl;

    infile.close();
    return rowcnt;
}

int getCol(const string filename)
{
    ifstream infile;
```

```
    infile.open(filename);
    if(!infile.is_open()){
        cout << "Could not open the file " << filename << endl;
        cout << "The program is terminated." << endl;
        exit(EXIT_FAILURE);
    }
    int colcnt=0;
    char c;
    c = infile.get();
    while(c != '\n' && c != '\r' && infile.good()){
        if(c == ' ') colcnt++;
        c = infile.get();
    }
    colcnt++;
    cout << colcnt << endl;

    if(infile.eof()) cout << "The end of file " << filename << " is reached" <<
endl;
    else if(infile.fail()) cout << "failure" << endl;

    infile.close();
    return colcnt;
}

void inputdo(double *matrix, int row, int col, const string filename)
{
    ifstream infile;
    infile.open(filename);
    while(infile.good()){
        for(int i=0;i<row;i++){
            for(int j=0;j<col;j++){
                infile >> *(matrix + j + i * row);
            }
        }
    }
    infile.close();
}

void outputdo(double *matrix,int row,int col,const string filename)
{
    ofstream outfile;
    outfile.open(filename);
    for(int i=0;i<row;i++)
    {
        for(int j=0;j<col;j++)
        {
            outfile << fixed << setprecision(6) << *(matrix + i*row + j) <<
'\t';
        }
        outfile<<endl;
    }
    outfile.close();
    cout << "The outcome is created!" << endl;
}

void inputfl(float *matrix, int row, int col, const string filename)
{
    ifstream infile;
```

```
    infile.open(filename);
    while(infile.good()){
        for(int i=0;i<row;i++){
            for(int j=0;j<col;j++){
                infile >> *(matrix + j + i * row);
            }
        }
    }
    infile.close();
}

void outputfl(float *matrix,int row,int col,const string filename)
{
    ofstream outfile;
    outfile.open(filename);
    for(int i=0;i<row;i++)
    {
        for(int j=0;j<col;j++)
        {
            outfile << fixed << setprecision(6) << *(matrix + i*row + j) <<
'\t';
        }
        outfile<<endl;
    }
    outfile.close();
    cout << "The outcome is created!" << endl;
}
```

```
/*matrix.cpp*/


#include "../inc/matrix.hpp"

flMatrix::flMatrix(int a, int b, int c, int d) : rowA(a), colA(b), rowB(c),
colB(d) {}

flMatrix::~flMatrix() {
    delete[] matrixA;
    delete[] matrixB;
    delete[] matrixRes;
}

void flMatrix::flMul(float *matrixA, int rowA, int colA, float *matrixB, int
rowB, int colB, float *mul) {
    for (int i = 0; i < rowA; i++) {
        for (int k = 0; k < colA; k++) {
            for (int j = 0; j < colB; j++) {
                *(mul + i * rowA + j) += *(matrixA + i * rowA + k) * (*(matrixB
+ k * colA + j));
            }
        }
    }
}
```

```
doMatrix::doMatrix(int a, int b, int c, int d) : rowA(a), colA(b), rowB(c),
colB(d) {}

doMatrix::~doMatrix() {
    delete[] matrixA;
    delete[] matrixB;
    delete[] matrixRes;
}

void doMatrix::doMul(double *matrixA, int rowA, int colA, double *matrixB, int
rowB, int colB, double *mul) {
    for (int i = 0; i < rowA; i++) {
        for (int k = 0; k < colA; k++) {
            for (int j = 0; j < colB; j++) {
                *(mul + i * rowA + j) += *(matrixA + i * rowA + k) * (*(matrixB
+ k * colA + j));
            }
        }
    }
}
```

```
/*main.cpp used to operate*/


#include "../inc/IO.hpp"
#include "../inc/matrix.hpp"
#include <ctime>

using namespace std;

clock_t startTime, endTime;

int main(int argc, char *argv[]) {
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);

    int rowA, colA, rowB, colB;
    string ma ="../proj2/matrices/in/";
    ma += argv[1];
    string mb = "../proj2/matrices/in/";
    mb += argv[2];
    rowA = getRow(ma);
    colA = getCol(ma);
    rowB = getRow(mb);
    colB = getCol(mb);

    if (colA != rowB) {
        cout << "The matrix dimension of A and B is unmatched" << endl;
        cout << "The program is terminated." << endl;
        exit(EXIT_FAILURE);
    }

    string out = "../proj2/matrices/out/";
    string outdo = out + "do-";
    outdo += argv[3];

    doMatrix doma(rowA, colA, rowB, colB);
```

```
    //input time used by double
    startTime = clock();
    inputdo(doma.matrixA, doma.rowA, doma.colA, ma);
    inputdo(doma.matrixB, doma.rowB, doma.colB, mb);
    endTime = clock();
    cout << "IN-double: " << (double)(endTime - startTime) / CLOCKS_PER_SEC <<
"s" << endl;

    //cal time used by double
    startTime = clock();
    doma.doMul(doma.matrixA, doma.rowA, doma.colA, doma.matrixB, doma.rowB,
doma.colB, doma.matrixRes);
    endTime = clock();
    cout << "CAL-double: " << (double)(endTime - startTime) / CLOCKS_PER_SEC <<
"s" << endl;

    //output time used by double
    startTime = clock();
    outputdo(doma.matrixRes, doma.rowA, doma.colB, outdo);
    endTime = clock();
    cout << "OUT-double: " << (double)(endTime - startTime) / CLOCKS_PER_SEC <<
"s" << endl;


    cout << " " << endl;


    string outfl = out + "fl-";
    outfl += argv[3];

    flMatrix fma(rowA, colA, rowB, colB);

    //input time used by float
    startTime = clock();
    inputfl(fma.matrixA, fma.rowA, fma.colA, ma);
    inputfl(fma.matrixB, fma.rowB, fma.colB, mb);
    endTime = clock();
    cout << "IN-float: " << (double)(endTime - startTime) / CLOCKS_PER_SEC <<
"s" << endl;

    //cal time used by float
    startTime = clock();
    fma.flMul(fma.matrixA, fma.rowA, fma.colA, fma.matrixB, fma.rowB, fma.colB,
fma.matrixRes);
    endTime = clock();
    cout << "CAL-float: " << (double)(endTime - startTime) / CLOCKS_PER_SEC <<
"s" << endl;

    //output time used by float
    startTime = clock();
    outputfl(fma.matrixRes, fma.rowA, fma.colB, outfl);
    endTime = clock();
    cout << "OUT-float: " << (double)(endTime - startTime) / CLOCKS_PER_SEC <<
"s" << endl;

}
```

Part 3 - Result & Verification

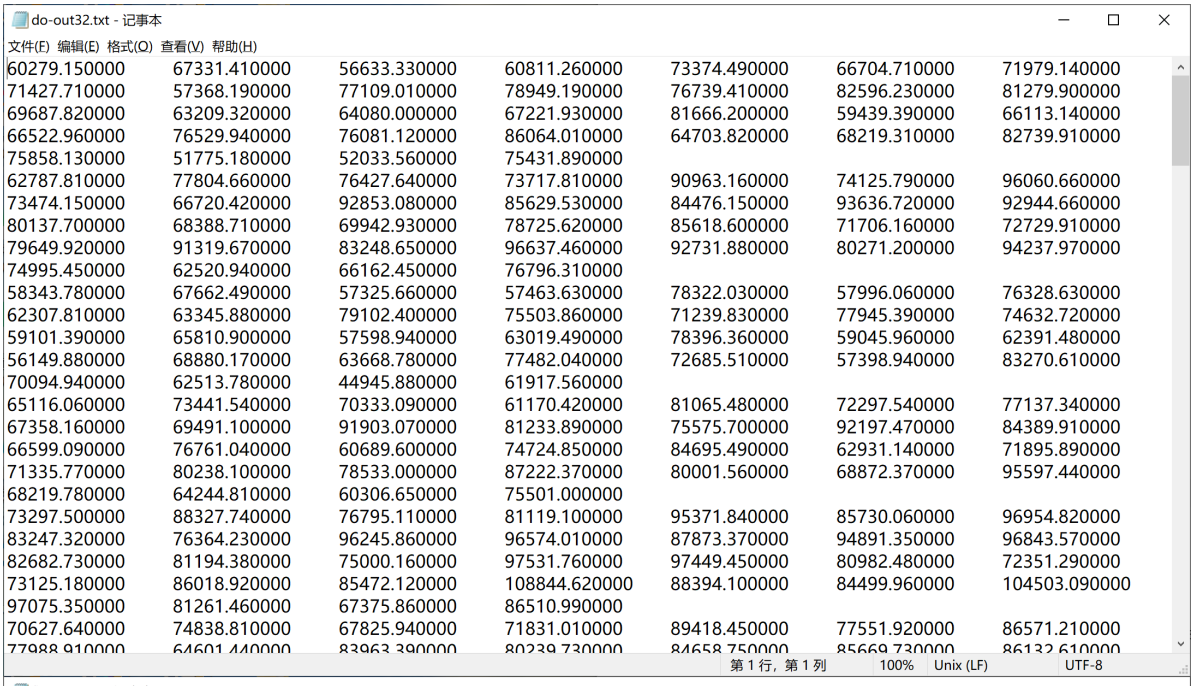
> In this part, I will present some instance of my program output.

Test case #1:

```
lynchrocket@LAPTOP-FV00SMG9:/mnt/c/Users/Lynchrocket/Desktop/c、c++/Project/Project2/proj2$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt
2048
The end of file ../proj2/matrices/in/mat-A-2048.txt is reached
2048
2048
The end of file ../proj2/matrices/in/mat-B-2048.txt is reached
2048
IN-double: 2.63955s
CAL-double: 36.7403s
The outcome is created!
OUT-double: 2.37518s

IN-float: 1.55424s
CAL-float: 36.4182s
The outcome is created!
OUT-float: 2.33166s
lynchrocket@LAPTOP-FV00SMG9:/mnt/c/Users/Lynchrocket/Desktop/c、c++/Project/Project2/proj2$ ./matmul mat-A-256.txt mat-B-256.txt out256.
txtcl
256
The end of file ../proj2/matrices/in/mat-A-256.txt is reached
256
256
The end of file ../proj2/matrices/in/mat-B-256.txt is reached
256
IN-double: 0.043211s
CAL-double: 0.123119s
The outcome is created!
OUT-double: 0.116973s

IN-float: 0.091788s
CAL-float: 0.11523s
The outcome is created!
OUT-float: 0.067678s
lynchrocket@LAPTOP-FV00SMG9:/mnt/c/Users/Lynchrocket/Desktop/c、c++/Project/Project2/proj2$ ./matmul mat-A-32.txt mat-B-32.txt out32.txt
cl
32
The end of file ../proj2/matrices/in/mat-A-32.txt is reached
32
32
The end of file ../proj2/matrices/in/mat-B-32.txt is reached
32
IN-double: 0.00287s
CAL-double: 0.000265s
The outcome is created!
OUT-double: 0.005711s

IN-float: 0.002883s
CAL-float: 0.000235s
The outcome is created!
OUT-float: 0.004797s
```

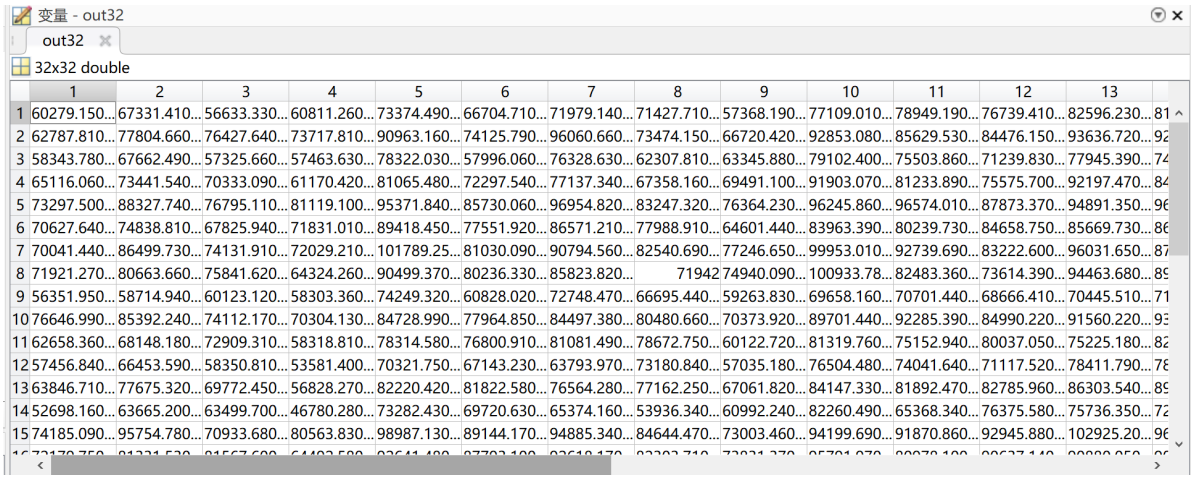
Notice that the above calculation all show that the speed of calcultion by float is faster than that by double.

I took `fl-out32.txt` and `do-out32.txt` for instance.



we can easily see from this test case that `float` type data can hold a more accurate outcome than `double` type.

To ensure the accuracy of the answer, I used Matlab to calculate the multiplication.



	1	2	3	4	5	6	7	8	9	10	11	12	13	
1	60279.150...	67331.410...	56633.330...	60811.260...	73374.490...	66704.710...	71979.140...	71427.710...	57368.190...	77109.010...	78949.190...	76739.410...	82596.230...	81
2	62787.810...	77804.660...	76427.640...	73717.810...	90963.160...	74125.790...	96060.660...	73474.150...	66720.420...	92853.080...	85629.530...	84476.150...	93636.720...	92
3	58343.780...	67662.490...	57325.660...	57463.630...	78322.030...	57996.060...	76328.630...	62307.810...	63345.880...	79102.400...	75503.860...	71239.830...	77945.390...	74
4	65116.060...	73441.540...	70333.090...	61170.420...	81065.480...	72297.540...	77137.340...	67358.160...	69491.100...	91903.070...	81233.890...	75575.700...	92197.470...	84
5	73297.500...	88327.740...	76795.110...	81119.100...	95371.840...	85730.060...	96954.820...	83247.320...	76364.230...	96245.860...	96574.010...	87873.370...	94891.350...	96
6	70627.640...	74838.810...	67825.940...	71831.010...	89418.450...	77551.920...	86571.210...	77988.910...	64601.440...	83963.390...	80239.730...	84658.750...	85669.730...	86
7	70041.440...	86499.730...	74131.910...	72029.210...	101789.25...	81030.090...	90794.560...	82540.690...	77246.650...	99953.010...	92739.690...	83222.600...	96031.650...	87
8	71921.270...	80663.660...	75841.620...	64324.260...	90499.370...	80236.330...	85823.820...	71942	74940.090...	100933.78...	82483.360...	73614.390...	94463.680...	89
9	56351.950...	58714.940...	60123.120...	58303.360...	74249.320...	60828.020...	72748.470...	66695.440...	59263.830...	69658.160...	70701.440...	68666.410...	70445.510...	71
10	76646.990...	85392.240...	74112.170...	70304.130...	84728.990...	77964.850...	84497.380...	80480.660...	70373.920...	89701.440...	92285.390...	84990.220...	91560.220...	93
11	62658.360...	68148.180...	72909.310...	58318.810...	78314.580...	76800.910...	81081.490...	78672.750...	60122.720...	81319.760...	75152.940...	80037.050...	75225.180...	82
12	57456.840...	66453.590...	58350.810...	53581.400...	70321.750...	67143.230...	63793.970...	73180.840...	57035.180...	76504.480...	74041.640...	71117.520...	78411.790...	78
13	63846.710...	77675.320...	69772.450...	56828.270...	82220.420...	81822.580...	76564.280...	77162.250...	67061.820...	84147.330...	81892.470...	82785.960...	86303.540...	89
14	52698.160...	63665.200...	63499.700...	46780.280...	73282.430...	69720.630...	65374.160...	53936.340...	60992.240...	82260.490...	65368.340...	76375.580...	75736.350...	72
15	74185.090...	95754.780...	70933.680...	80563.830...	98987.130...	89144.170...	94885.340...	84644.470...	73003.460...	94199.690...	91870.860...	92945.880...	102925.20...	96

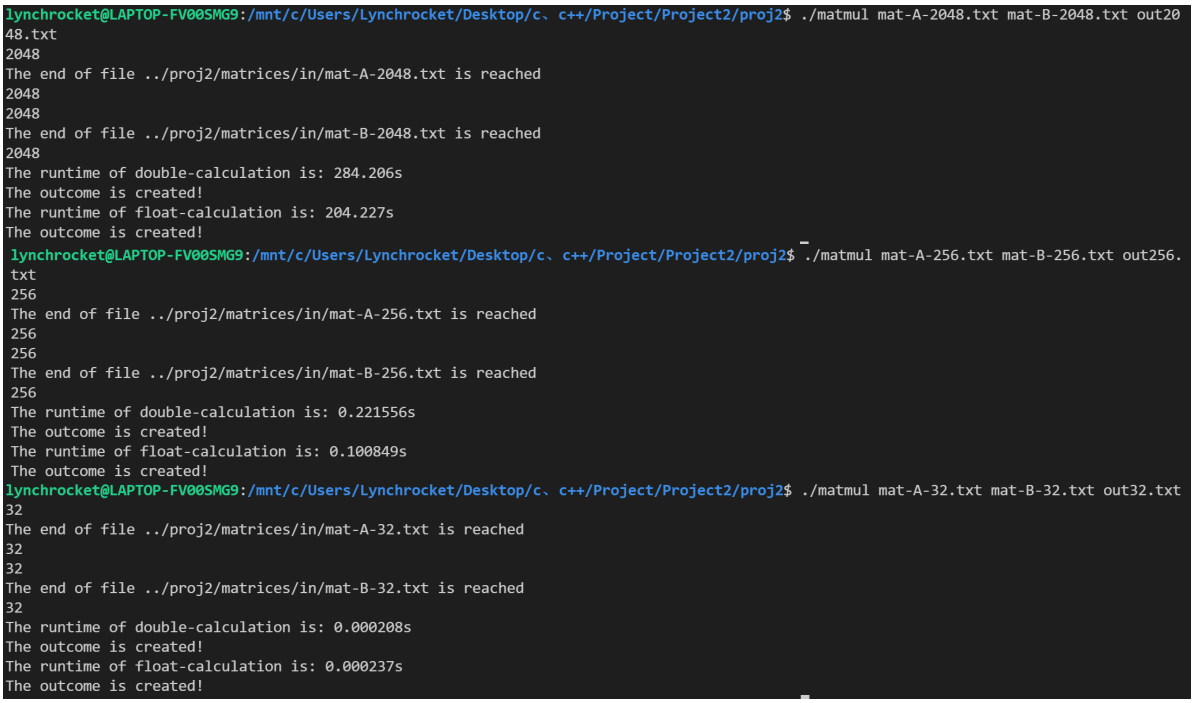
By the result, whether using `double` or `float`, the answer of the multiplication is reliable.

Part 4 - Difficulties & Solutions

- Originally, my algorithm to calculate the matrix multiplication is very crude.

```
/*ijk*/
for (int i = 0; i < rowA; i++) {
    for (int j = 0; j < colB; j++) {
        for (int k = 0; k < colA; k++) {
            *(mul + i * rowA + j) += *(matrixA + i * rowA + k) * (*(matrixB + k
* colA + j));
        }
    }
}
```

I had no sooner found that it was too slow (cases shown in below).



So I optimized it.

```
/*ikj*/
for (int i = 0; i < rowA; i++) {
    for (int k = 0; k < colA; k++) {
        for (int j = 0; j < colB; j++) {
            *(mul + i * rowA + j) += *(matrixA + i * rowA + k) * (*(matrixB + k
* colA + j));
        }
    }
}
```

Although it was still running slow, it was much more rapid, expecially for a large amount of data.

```
lynchrocket@LAPTOP-FV00SMG9:/mnt/c/Users/Lynchrocket/Desktop/c、c++/Project/Project2/proj2$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt
2048
The end of file ../proj2/matrices/in/mat-A-2048.txt is reached
2048
2048
The end of file ../proj2/matrices/in/mat-B-2048.txt is reached
2048
IN-double: 2.63955s
CAL-double: 36.7403s
The outcome is created!
OUT-double: 2.37518s

IN-float: 1.55424s
CAL-float: 36.4182s
The outcome is created!
OUT-float: 2.33166s
lynchrocket@LAPTOP-FV00SMG9:/mnt/c/Users/Lynchrocket/Desktop/c、c++/Project/Project2/proj2$ ./matmul mat-A-256.txt mat-B-256.txt out256.
txtcl
256
The end of file ../proj2/matrices/in/mat-A-256.txt is reached
256
256
The end of file ../proj2/matrices/in/mat-B-256.txt is reached
256
IN-double: 0.043211s
CAL-double: 0.123119s
The outcome is created!
OUT-double: 0.116973s

IN-float: 0.091788s
CAL-float: 0.11523s
The outcome is created!
OUT-float: 0.067678s
lynchrocket@LAPTOP-FV00SMG9:/mnt/c/Users/Lynchrocket/Desktop/c、c++/Project/Project2/proj2$ ./matmul mat-A-32.txt mat-B-32.txt out32.txt
cl
32
The end of file ../proj2/matrices/in/mat-A-32.txt is reached
32
32
The end of file ../proj2/matrices/in/mat-B-32.txt is reached
32
IN-double: 0.00287s
CAL-double: 0.000265s
The outcome is created!
OUT-double: 0.005711s

IN-float: 0.002883s
CAL-float: 0.000235s
The outcome is created!
OUT-float: 0.004797s
```

To find out the reason, I had searched for it on the Internet. And then I found the website.

(https://zhuanlan.zhihu.com/p/146250334)

对于矩阵乘法而言，一维数组显然比二维数组好得多（下文的分析也能看出这一点）。但是如果用矩阵类实现其他功能的话，可能其他功能用二维数组更方便一些。所以下面对于这两种实现方式都加以分析。

造成矩阵乘法慢的原因，除了算法上的 $O(n^3)$ 以外，还有**内存访问不连续**。这会导致cache命中率不高。所以为了加速，就要尽可能使内存访问连续，即不要跳来跳去。我们定义一个概念：**跳跃数**，来衡量访问的不连续程度。

对于最普通的实现方式（顺序： ijk ），它是依次计算 C 中的每个元素。当计算 C 中任一个元素时，需要将 A 对应的行与 B 对应的列依次相乘相加。之前已假设过，矩阵是按行存储的，所以在 A 相应行中不断向右移动时，内存访问是连续的。但 B 相应列不断向下移动时，内存访问是不连续的。计算完 C 的一个元素时， B 相应列中已经间断地访问了 n 次，而 A 只间断 1 次（这一次就是算完后跳转回本行的开头），故总共是 $n+1$ 次。这样计算完 C 中所有 n^2 个元素，跳转了 n^3+n^2 次。但刚才没有计数 C 的跳转次数，加上以后是 n^3+n^2+n 。（注意，在计算完 C 中每行的最后一个元素时， A 是从相应行末尾转到下一行开头。如果使用一维数组实现的话，这是连续地访问，要减掉这 n 次。同时， C 没有跳转次数了，还要减掉 n 次。因此对于一维数组，跳转数是 n^3+n^2-n 次）

而如果以顺序 ikj 实现，它将 C 中元素一行一行计算。当计算 C 中任一行的第一个元素时，先访问 A 中相应行的第一列元素，和 B 中第一列的第一行元素，然后 B 不断往右挪（不间断），算完后跳转到下一行（如果二维数组则间断一次，一维数组不间断），此时 A 往右挪一个元素（不间断）。依次这样挪动，这样算完 C 的这一行元素后，恰好按顺序将 B 遍历一遍，间断了 n 次（一维数组是 1 次），且恰好从左往右遍历了 A 的相应行，间断了 1 次（一维数组没有间断），加起来是 $n+1$ 次（一维数组是 1 次）。故算完 C 的所有 n 行后，跳转了 n^2+n 次（一维数组是 n 次）。刚才没有算 C 的跳转，算上后跳跃数是 $2n^2+n$ 次（一维数组是 n^2 次）。

Obviously, apart from `ijk` and `ikj`, similarly we have `kij`, `jik`, `kji`, `jki` as an alternative solution.

下面是各个循环顺序的跳跃数列表（下面是我写文章时现计算的，可能会因为粗心犯错）

- 顺序 ikj —— $2n^2+n$ （二维数组）—— n^2 （一维数组）
- 顺序 kij —— $3n^2$ （二维数组）—— $2n^2$ （一维数组）
- 顺序 jik —— n^3+2n^2 （二维数组）—— n^3+n^2+n （一维数组）
- 顺序 ijk —— n^3+n^2+n （二维数组）—— n^3+n^2-n （一维数组）
- 顺序 kji —— $2n^3+n$ （二维数组）—— $2n^3$ （一维数组）
- 顺序 jki —— $2n^3+n^2$ （二维数组）—— $2n^3+n^2$ （一维数组）

因此从速度来说：

$$ikj > kij > jik \vee ijk > kji > jki$$

实测速度：（ 1000 阶， gcc ， $-O3$ 优化）

- ikj : $847ms$
- kij : $1028ms$
- jik : $2733ms$
- ijk : $4552ms$
- kji : $17269ms$
- jki : $17197ms$

发现基本符合（ kji 与 jki 略微违反，其实它们很接近，回过头看理论分析也表明很接近）

- Eventually, I found that `cin` and `cout` are inefficient because of buffer in the below website.

(https://stackoverflow.com/questions/1042110/using-scanf-in-c-programs-is-faster-than-using-cin)

By using `ios::sync_with_stdio(0)`, we can eliminate the buffer of `iostream`.
And by using `cin.tie(0)` and `cout.tie(0)`, we can unbind `cin` and `cout`, reduce IO burden, and further speed up thre execution efficiency.

```
ios::sync_with_stdio(0),cin.tie(0),cout.tie(0);
```

Part 5 - Some Better Choices

- Algorithm

None of `ijk`-series algorithm is a good option for matrix multiplication (all with O(n^3) time complexity except for a constant coefficient).

However, there is a famous algrithm for matrix multiplication --- *Strassen algorithm*, which can attain a time complexity O($n^{2.807}$). It would not be faster at a small scale data.

(http://www.longluo.me/blog/2019/06/21/Strassens-Matrix-Multiplication-Algorithm/)

- Hardware
  GEMM, `<thread>` library, simd, etc.

  I found that with different computer, the time cost for calculation is different. It may be due to some optimization in CPU.