CS205 C/C++ Program Design-Assignment1

Name: 陈逸飞 Chen Yifei

Student ID: 12010502

Develop environment: Apple clang version 12.0.5 (clang-1205.0.22.11)

Part 1 - Analysis

This assignment required us to write 6 functions in **assign.c**. Each of functions solve certain problems.

> **quick_power**: Solve x^n using Divide and Conquer methods.
>
> **matrix_addition**: Add two matrices.
>
> **matrix_multiplication**: Multiplie two matrices.
>
> **naive_matrix_exponentiation**: Using regular loops to calculate matrix exponentiation A^n with complex degree $O(n^3)$.
>
> **fast_matrix_exponentiation**: Using methods in quick_power to calculate matrix exponentiation A^n with complex degree $O(logn)$.
>
> **fast_cal_fib**: Using matrix exponentiation to calculate Fibonacci Sequence.

Since we are provided with functions to do matrix operations in assign1_mat.c, so I don't have to create the fundemental elements and functions such as constructing the matrixs with arrays. Therefore, in this assignment, our main concern is to design algorithms which solve the problems in the most efficiency way. According to the testing cases, I think the orientation of this project is mainly to deal with large matrices and high power calculations, since the data type of the input parameter of functions are mainly integer. In my personal view, the main problems of the assignment are power calculation and how to apply quick-power to matrix calculaions, within proper memory management.

1.1Methods

Provided functions:

> 1.**Structure struct matrix**: Structure for matrix.
>
> 2.**Function create_matrix_all_zero**: Create a matrix filled by zeros.
>
> 3.**Function delete_matrix**: Delete the data segment of a matrix. You MUST call this function whenver a matrix is no longer needed.

> 4.**Function copy_matrix**: Copy a matrix. Do NOT use = to copy a matrix.
>
> 5.**Funciton set_by_index**: Set an entry of matrix to a specified value.
>
> 6.**Function get_by_index**: Get the value in an specified entry of matrix.

With the help of these functions, i can apply the matrix knowledge directly.

1.1.1 quick_power(20pts)

The tradtional power use regular loop, so its complex degree is $O(n)$, but as input index grow bigger, the algorithms will fail. This is because as index increasing, the result of power shows an explosive growth, which loops have to be execute many times. However, the quick_power use the method of divding, splitting the n times multiplication operations into $n/2$ times multiplication.

> if n is odd，x^n = x^(n/2) * x^(n/2)
>
> if n is even, x^n = x^(n/2) * x^(n/2) *x

An example: 2^(10) = 2^(5) * 2^(5) = $(2*2)^5$= 32 * 32 = 1024

When we calculate 2^(10), we need to circulate 10 times, but as we split it into 2^(5) * 2^(5) , we reduced 5 times circulation. Since the index can not be floating-point type, we can divide $4^5$to $4^4 * 4^1$. And then we can continue divide 4^4 into $16^2 * 16^2$. We can continue to use circulation.

1.1.2 matrix_addition(20pts)

> Matrix is useful in linear algebra and computer science.

According to basic linear algebra knowledge, we know matrix addition is to add two matrix elements together, whose size are identical. By using two circulations, we can go though over all the elements of two matrices and add them together to form a new matrix.

For example:

$$result = \begin{bmatrix} x_1 + y_1 & x_2 + y_2 & x_3 + y_3 \\ x_4 + y_4 & x_5 + y_5 & x_6 + y_6 \\ x_7 + y_7 & x_8 + y_8 & x_9 + y_9 \end{bmatrix}$$

1.1.3 matrix_multiplication(30pts)

Matrix multiplication is a little harder than matrix addition. As we know, when we do matrix multiplication, we multiple each row of left matrix with each column of right matrix. Therefore, after go though over all the elements, I apply an other circulation, to realize the step of rows multiple columns.

1.1.4 naive_matrix_exponentiation(15pts)

Matrix exponentiation is to do premultiplication to the matrix n times.Since only square matrix can do exponentiation again and again, we can use function **matrix_multiplication** for n times, which complex degree is $O(n * size^3)$.

1.1.5 fast_matrix_exponentiation(15pts)

This function apply the method in task1(**fast_power**) , splitting n times to odd cases and even cases.

1.1.6 fast_cal_fib(20pts)

This is one application of fast_matrix_exponentiation. The traditional method to calculate the Fibonacci Sequence is using a loop and calculating the recursive derivation in $O(n)$. Here we have recursive formula:

> $fn = fn-1 + fn-2$, else
>
> $f0 = 0, f1 = 1$

However, we can use **fast_matrix_exponentiation** to calculate required elements in Fibonacci Sequence.

We first construct matrix:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^m where, m = n - 1$$

Then do premultiplication to the first two elements of Fibonacci sequence:

$$\begin{bmatrix} f_1 \\ f_0 \end{bmatrix}$$

The result will be

$$\begin{bmatrix} f_n \\ f_m \end{bmatrix}$$

The nth element of Fibonacci Sequence is f_n.

1.2 Error Handling

> Please make sure your code will not crash if the input is incorrect. Your code should give some notice when the input is incorrect. All the required error handling will be noticed in the assignment document.

According to assignment document, we should consider all possibility of input error.

1.2.1 Exceptions input

The integer we input is in range $x \geq 0 \cap n \geq 0$, which we do not need to take actions. However, when we deal with the matrix, we always need to check size and making exclusion of special circumstances.

1.2.1.1 Size checking

The matrix multiplication require that the rows and columns of input matrix are equal. In the same time, index of exponentiation calculation should also be especially considered, because when n=0, result is a unit matrix, and if n=0, result will be just be the input matrix. These preprocessing steps contribute to save times and preventing program failure.

1.2.2 Data field

The data that needs to be processed is very large, which means data needs to apply modulo arithmetic. When calculating the quotient value, the quotient value is rounded to negative infinity; Keep the quotient as small as possible.

1.3 Memory management

In last five functions, we need to create new matrix to help calculation. After **mat_res** get the answer, I use **delete_mat** to delete unused matrix to free the memory.

Part 2 - Code(Fundamental)

```
#include <stdio.h>
#include <math.h>
//
int copy_value(matrix mat_origin, matrix mat_copy);


//Part 1 Task1 20pts
//Power of x^n
int quick_power(int x, int n)
{
    unsigned long long answer = 1;
    long long x1 = x % MODULO;
    long long n1 = n;

    for(;n1 != 0; n1 /= 2)
    {
        if(n1 % 2 == 1)
        {
            answer = (answer * x1) % MODULO;
        }
        x1 = ((x1 % MODULO) * (x1 % MODULO)) % MODULO;
    }
    return answer;
}

//Task2-1 20pts
int matrix_addition(matrix mat_a, matrix mat_b, matrix mat_res)
{
    matrix mat1 = copy_matrix(mat_a);
    matrix mat2 = copy_matrix(mat_b);

    //check size
     if (mat2.m_row != mat1.m_row || mat2.m_col != mat1.m_col ||
        mat2.m_row != mat_res.m_row || mat2.m_col != mat_res.m_col)
```

```
    {
        return 1;
    }
    //Addition
    for(int i=0; i< mat_res.m_row; ++i)
    {
        for(int j=0; j < mat_res.m_col; ++j)
        {
            long val = (((long long) get_by_index(mat1, i, j)) + ((long long)
get_by_index(mat2, i, j))) % MODULO;
            set_by_index(mat_res, i, j, val);
        }
    }
    delete_matrix(mat1);
    delete_matrix(mat2);
    return 0;
}

//Tas2-2
int matrix_multiplication(matrix mat_a, matrix mat_b, matrix mat_res)
{
    long long temp = 0;
    matrix mat1 = copy_matrix(mat_a);
    matrix mat2 = copy_matrix(mat_b);
    //check matRes
    //check size
    if (mat1.m_col != mat2.m_row)
    {
        return 1;
    }
    if ( mat1.m_row != mat_res.m_row || mat2.m_col != mat_res.m_col)
    {
        return 1;
    }

    //Mul
    for(int i=0; i < mat1.m_row; i++)
    {
        for(int j=0; j < mat2.m_col; j++)
        {
            set_by_index(mat_res, i, j, 0);
            for(int k=0; k < mat1.m_col; k++)
            {
                temp = (temp + ((long long)get_by_index(mat1,i,k) * (long
long)get_by_index(mat2,k,j)) % MODULO) % MODULO;
            }
            set_by_index(mat_res,i,j,(int)temp);
            temp = 0;
        }
```

```
    }
    delete_matrix(mat1);
    delete_matrix(mat2);
    return 0;
}

//Task3
int naive_matrix_exp(matrix mat_a, int exp, matrix mat_res)
{
    int i,j,times,val,q=0;
    long long tmp = 0;
    long long temp = 0;
    matrix matIn = copy_matrix(mat_a);    //Inputs matrix Unchanged
    matrix mat_temp = copy_matrix(mat_a); //temporary matrix, after multiplied
with matIn

    //check size
    if(matIn.m_row != mat_res.m_row || matIn.m_col != mat_res.m_col)
    {
        return 1;
    }
    if(exp<0)
    {
        return 1;
    }
     //check exp
    if (exp < 0){ return 1; }
    //Check if exp = 0
    if(exp == 0)
    {
        for(i=0; i<=matIn.m_row; ++i)
        {
            for(j=0; j<=matIn.m_col; ++j)
            {
                if(i==j)
                {
                    set_by_index(mat_res,i,j,1);
                }else{
                    set_by_index(mat_res,i,j,0);
                }
            }
        }
        return 0;
    }

    //exp = 1
    if(exp==1)
    {
        copy_value(matIn,mat_res);
```

```
        return 0;
    }
    //exp > 0
    for(times=1; times<exp; times++)
    {
        //Matrix mul
        for(i=0; i < matIn.m_row; i++)
        {
            for(int j=0; j < matIn.m_col; j++)
            {
                set_by_index(mat_res, i, j, 0);
                for(int k=0; k < matIn.m_col; k++)
                {
                    tmp = (tmp + (((long long)get_by_index(matIn,i,k) %
MODULO) * ((long long)get_by_index(mat_temp,k,j)%MODULO)) % MODULO) % MODULO;
                }
                set_by_index(mat_res,i,j,(int)tmp);
                tmp = 0;
            }
        }
        mat_temp = copy_matrix(mat_res);
    }
    delete_matrix(matIn);
    delete_matrix(mat_temp);
    return 0;
}

//Task4
int fast_matrix_exp(matrix mat_a, long long exp, matrix mat_res)
{
    long long n = exp;
    matrix mat_In = copy_matrix(mat_a);
    matrix mat_Ide = create_matrix_all_zero(mat_a.m_row, mat_a.m_col);

    //单位矩阵
    for (int i = 0; i < mat_a.m_row; i++)
    {
        for (int j = 0; j < mat_a.m_col; j++)
        {
            if (i == j)
                set_by_index(mat_Ide, i, j, 1);
            else
                set_by_index(mat_Ide, i, j, 0);
        }
    }
    //check size
    if (mat_In.m_row != mat_res.m_row ||
        mat_In.m_col != mat_res.m_col)
```

```
        return 1;
    //check exp
    if (n < 0)
    {
        delete_matrix(mat_In);
        delete_matrix(mat_Ide);
        return 1;
    }
    else if (n == 0)
    {
        mat_res = copy_matrix(mat_Ide);
        delete_matrix(mat_Ide);
        delete_matrix(mat_In);
        return 0;
    }

    else if (n == 1)
    {
        mat_res = copy_matrix(mat_a);
        delete_matrix(mat_In);
        delete_matrix(mat_Ide);
        return 0;
    }
    else
    {
        matrix mat_Buffer = create_matrix_all_zero(mat_a.m_row, mat_a.m_col);
        matrix mat_Temp = create_matrix_all_zero(mat_a.m_row, mat_a.m_col);
        matrix mat_Out = create_matrix_all_zero(mat_a.m_row, mat_a.m_col);
        mat_Temp = copy_matrix(mat_a);

        mat_Out = copy_matrix(mat_Ide);
        delete_matrix(mat_Ide);

        while (n != 0)
        {
            //odd number
            if (n % 2 == 1)
            {
                matrix_multiplication(mat_Out, mat_Temp, mat_Buffer);

                copy_value(mat_Buffer,mat_Out);
            }
            //even number
            {
                matrix_multiplication(mat_Temp, mat_Temp, mat_Buffer);

                copy_value(mat_Buffer,mat_Temp);
            }
            n = n / 2;
```

```
        }
        //Copy value
        copy_value(mat_Out,mat_res);

        delete_matrix(mat_In);
        delete_matrix(mat_Buffer);
        delete_matrix(mat_Temp);
        delete_matrix(mat_Out);
        return 0;
    }
}

//Task5
int fast_cal_fib(long long n)
{
    long long answer;
    if(n == 0)
    {
        answer = 0;
        return answer;
    }

    if (n == 1 || n == 2)
    {
        answer = 1;
        return answer;
    }
    else
    {
    matrix mat_init = create_matrix_all_zero(2,2);
    matrix mat_mul_out = create_matrix_all_zero(2,2);
    matrix mat_temp = create_matrix_all_zero(2,1);
    matrix mat_Fib = create_matrix_all_zero(2,1);
    //设置初始矩阵
    // 1 1
    // 1 0
    set_by_index(mat_init,0,0,1);
    set_by_index(mat_init,0,1,1);
    set_by_index(mat_init,1,0,1);
    set_by_index(mat_init,1,1,0);
    // 1
    // 0
    set_by_index(mat_temp,0,0,1);
    set_by_index(mat_temp,1,0,0);
    //fast_matrix_exp
    fast_matrix_exp(mat_init,n-1,mat_mul_out);
    matrix_multiplication(mat_mul_out,mat_temp,mat_Fib);
    answer = (get_by_index(mat_Fib,0,0)) % MODULO;
```

```
    delete_matrix(mat_init);
    delete_matrix(mat_mul_out);
    delete_matrix(mat_temp);
    delete_matrix(mat_Fib);
    return answer;
    }
}

//For copy matrix value to another
int copy_value(matrix mat_origin, matrix mat_copy)
{
    for (int i = 0; i < mat_origin.m_row; i++)
    {
        for (int j = 0; j < mat_origin.m_col; j++)
        {
            long long temp = get_by_index(mat_origin, i, j);
            set_by_index(mat_copy, i, j, (int)temp);
        }
    }
    return 0;
}
```

When I use the function **copy_matrix**, it causes the copied matrix arguments to be overwritten and not get the correct values when accessed by external functions. So I added **copy_value** to **assign1.c** to copy only the values of the matrix.

Part 3-Optimum Programming

To test my optimum have worked, I set a clock to record time efficiency. (For different task, there will be some differences)

```
int main()
{
    clock_t start, finish;
    //clock_t为CPU时钟计时单元数
    start = clock();
    //clock()函数返回此时CPU时钟计时单元数
    printf("answer: %d\n",(call functions));
    finish = clock();
    //clock()函数返回此时CPU时钟计时单元数
    printf("time: %f\n",(double)(finish - start) / CLOCKS_PER_SEC);
    //finish与start的差值即为程序运行花费的CPU时钟单元数量，再除每秒CPU有多少个时钟单元，
即为程序耗时
    return 0;
}
```

In order to be able to test the processing time of large matrices (showing the significant advantage of the optimization approach in terms of time), I used **rand ()** to randomly generate the matrices of the specified dimensions to test the running efficiency and speed.

```
int set_random_matrix(matrix mat)
{
    for(int r=0;r<mat.m_row;++r)
    {
        for(int c=0;c<mat.m_col;++c)
        {
          set_by_index(mat,r,c,rand());
        }
    }
    return 0;
}
```

3.1 fast_power

In C language, **n % 2 == 1** can be replaced with faster bitwise operation, such as **n & 1**. If power is even, the last bit of its binary representation must be 0; If power is odd, the last bit of its binary representation must be 1. Add them to the binary of one, and you get the last digit of the power binary: 0 is even, 1 is odd.

Similarly, **n = n / 2** can be replaced with bitwise operation, where we simply move the binary representation of power one bit to the right to make it half.

```
int fast_Power(int x, int n)
{
    unsigned long long answer = 1;
    long long x1 = x % MODULO;
    long long n1 = n;
    while (n1 > 0) {
        if (n1 & 1) {//此处等价于if(n%2==1)
            answer = (answer * x1) % MODULO;
        }
        n1 >>= 1;//此处等价于n=n/2
        x1 = ((x1%MODULO) * (x1%MODULO)) % MODULO;
    }
    return answer;
}
```

In test case 8:

> Input: 959465599 478516499
>
> Output: 646341436

The origin fast_power:

```
ericchen@EricdeMacBook-Air code % cd "/Users/ericchen/Desktop/cppLab/Assignment1/code/" && gcc quickPower.c -o quickPower && "/Users/ericche
n/Desktop/cppLab/Assignment1/code/"quickPower
base: 959465599, power: 478516499
answer: 646341436
time: 0.000061
```

The fast_power using bitwise

```
ericchen@EricdeMacBook-Air code % cd "/Users/ericchen/Desktop/cppLab/Assignment1/code/" && gcc quickPower.c -o quickPower && "/Users/ericche
n/Desktop/cppLab/Assignment1/code/"quickPower
base: 959465599, power: 478516499
answer: 646341436
time: 0.000046
```

0.000046 is clearly faster than **0.000061**.

3.2 Optimization of matrix multiplication

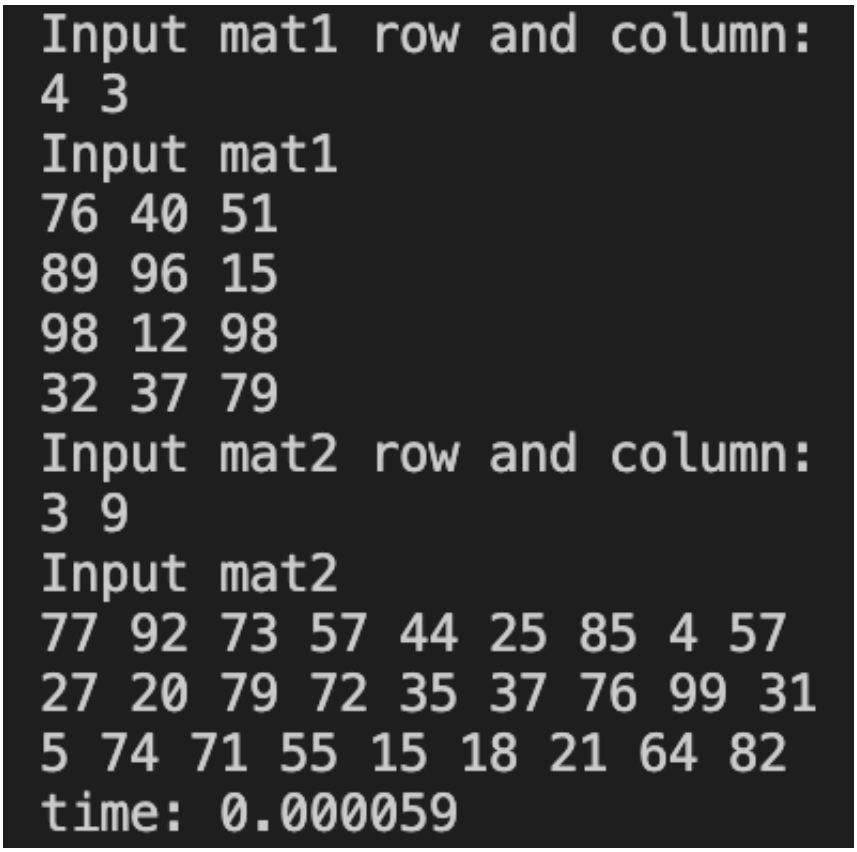
The original multiplication use three loops to calculate. It is very easy to read but it's also primitive.

get_by_index(mat2,k,j) Read data in memory, the process is discontinuous. At the bottom loop, **get_by_index(mat2,k,j)** jumps in memory as k increases by 1. This operation will result in a low cache hit ratio, and the looper will constantly move memory to the cache, causing a loss of efficiency.

```
    for(int i=0; i < mat1.m_row; i++)
    {
        for(int j=0; j < mat2.m_col; j++)
        {
            set_by_index(mat_res, i, j, 0);
            for(int k=0; k < mat1.m_col; k++)
            {
                temp = (temp + ((long long)get_by_index(mat1,i,k) * (long
long)get_by_index(mat2,k,j)) % MODULO) % MODULO;
            }
            set_by_index(mat_res,i,j,(int)temp);
            temp = 0;
        }
    }
```

So I found some ways to promote efficiency.

Original multiplication:



3.2.1 Memory access optimization

Just changing the order of the loops can dramatically improve performance.

```
    //Mul
    for(int i=0; i < mat1.m_row; i++)
    {
        for(int k=0; k < mat2.m_row; k++)
        {
            temp = (long long)get_by_index(mat1,i,k);
            for(int j=0; j < mat2.m_col; ++j)
            {
                long long t = get_by_index(mat_res,i,j) % MODULO;
                long long p = get_by_index(mat2,k,j) % MODULO;
                set_by_index(mat_res,i,j,t + temp * p);
            }
        }
    }
```

As we have mentioned, for a two-dimensional array, the elements that need to be accessed are mostly discontinuous in memory, which cost lot of time. Thus reducing reads (jumping to memory addresses) increases multiplication efficiency. Using a one-dimensional array can better reduce the number of jumping. 这一方法的关键在于提升了访存时的效率，而不是CPU/RAM计算的能力。

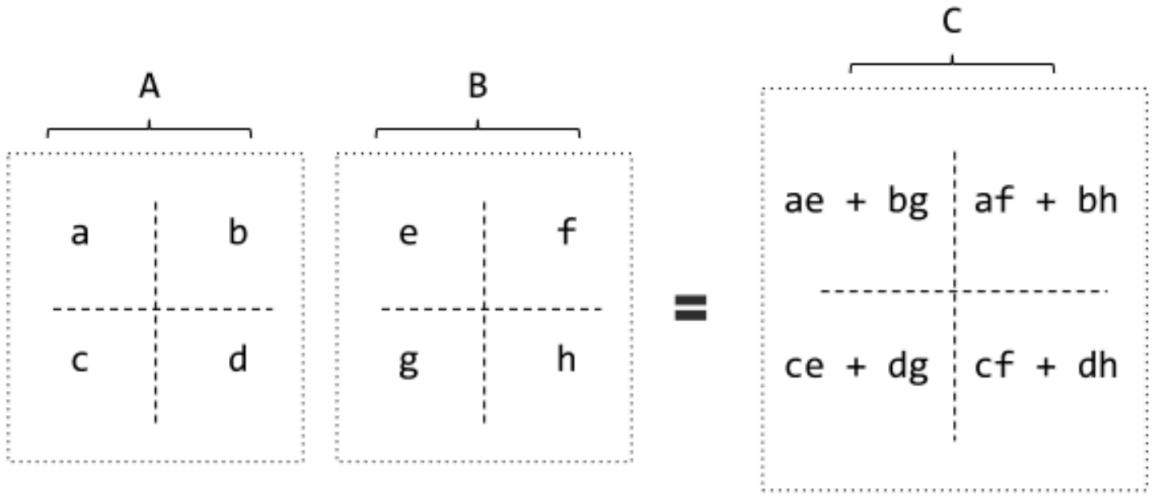
Same input：

```
Input mat1 row and column:
4 3
Input mat1
76 40 51
89 96 15
98 12 98
32 37 79
Input mat2 row and column:
3 9
Input mat2
77 92 73 57 44 25 85 4 57
27 20 79 72 35 37 76 99 31
5 74 71 55 15 18 21 64 82
time: 0.000047
```

3.2.2 Strassen Arithmetic

Generally speaking, when the amount of data is large, we tend to divide the large data into small data and process them separately. The complex degree of matrix multiplication is mostly about multiplication, and an extra addition or two doesn't add much to the complexity.

> Quote：https://www.jianshu.com/p/dc67e4a3c841
>
> 1969年，德国的一位数学家Strassen证明$O(N^3)$的解法并不是矩阵乘法的最优算法，他做了一系列工作使得最终的时间复杂度降低到了$O(n2.80)$。
>
> 首先将A，B，C矩阵分为相等大小的方块矩阵（限制条件就是相乘的两个矩阵必须是方阵，且幂大小为2）



> 然后通过定义7个变量：

> $P_1 = A(F - H)$
>
> $P_2 = (A + B)H$
>
> $P_3 = (C + D)E$
>
> $P_4 = D(G - E)$
>
> $P_5 = (A + D)(E + H)$
>
> $P_6 = (B - D)(G + H)$
>
> $P_7 = (A - C)(E + F)$
>
> 然后实现矩阵相乘

$$C = AB = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

Strassen algorithm is the matrix multiplication algorithm optimization of the two most commonly used algorithms.Therefore, it is necessary to continuously divide the matrix into blocks and recurse until the submatrix is small enough.

```
int strassen_launch(matrix mat_a, matrix mat_b, matrix mat_res)
{
    //方阵检查
    if(mat_a.m_row != mat_a.m_col || mat_b.m_row != mat_b.m_col)
    {
        printf("Must be square matrix!\n");
        return 1;
    }
    if (mat_a.m_row != mat_b.m_col)
    {
        printf("A and B must be same size!\n");
        return 1;
    }

    int n = mat_a.m_row;
     if(n & (n-1))   //check if n is 2^x
    {
        printf("Must be power of 2!\n");
        return 1;
    }
    //Normal
    strassen(mat_a,mat_b,mat_res);
    return 0;
}

int strassen(matrix mat_a, matrix mat_b, matrix mat_res)
{
    int n = mat_a.m_row;
    n /= 2;
```

```
matrix A11, A12, A21, A22, B11, B12, B21, B22;
//A11
matrix sub1 =create_matrix_all_zero(n,n);
sub_matrix(mat_a,sub1,0,0,n,n);
Copy_value(sub1,A11);
delete_matrix(sub1);
//A12
matrix sub2 =create_matrix_all_zero(n,n);
sub_matrix(mat_a,sub2,0,n,n,n);
Copy_value(sub2,A12);
delete_matrix(sub2);
//A21
matrix sub3 =create_matrix_all_zero(n,n);
sub_matrix(mat_a,sub3,n,0,n,n);
Copy_value(sub3,A21);
delete_matrix(sub3);
//A22
matrix sub4 =create_matrix_all_zero(n,n);
sub_matrix(mat_a,sub4,n,n,n,n);
Copy_value(sub4,A22);
delete_matrix(sub4);
//B11
matrix sub5 =create_matrix_all_zero(n,n);
sub_matrix(mat_b,sub5,0,0,n,n);
Copy_value(sub5,B11);
delete_matrix(sub5);
//B12
matrix sub6 =create_matrix_all_zero(n,n);
sub_matrix(mat_b,sub6,0,n,n,n);
Copy_value(sub6,B12);
delete_matrix(sub6);
//B21
matrix sub7 =create_matrix_all_zero(n,n);
sub_matrix(mat_b,sub7,n,0,n,n);
Copy_value(sub7,B21);
delete_matrix(sub7);
//B22
matrix sub8 =create_matrix_all_zero(n,n);
sub_matrix(mat_b,sub8,n,n,n,n);
Copy_value(sub8,B22);
delete_matrix(sub8);

matrix S1,S2,S3,S4,S5,S6,S7,S8,S9,S10;
matrix_subtraction(B12,B22,S1);
matrix_addition(A11,A12,S2);
matrix_addition(A21,A22,S3);
matrix_subtraction(B21,B11,S4);
matrix_addition(A11,A22,S5);
```

```
    matrix_addition(B11,B22,S6);
    matrix_subtraction(A12,A22,S7);
    matrix_addition(B21,B22,S8);
    matrix_subtraction(A11,A22,S9);
    matrix_addition(B11,B22,S10);
    //Strassen functions
    matrix F1,F2,F3,F4,F5,F6,F7;
    strassen(A11,S1,F1);
    strassen(S2,B22,F2);
    strassen(S3,B11,F3);
    strassen(A22,S4,F4);
    strassen(S5,S6,F5);
    strassen(S7,S8,F6);
    strassen(S9,S10,F7);

    matrix C11,C12,C21,C22;
    matrix_addition(F5,F4,C11);
    matrix_subtraction(C11,F2,C11);
    matrix_addition(C11,F6,C11);

    matrix_addition(F1,F2,C12);
    matrix_addition(F3,F4,C21);

    matrix_addition(F5,F1,C22);
    matrix_subtraction(C22,F3,C22);
    matrix_subtraction(C22,F7,C22);

    Sum(C11,C12,C21,C22,mat_res);
    delete_matrix(A11);
    delete_matrix(A12);
    delete_matrix(A21);
    delete_matrix(A22);
    delete_matrix(B11);
    delete_matrix(B12);
    delete_matrix(B21);
    delete_matrix(B22);
    delete_matrix(S1);
    delete_matrix(S2);
    delete_matrix(S3);
    delete_matrix(S4);
    delete_matrix(S5);
    delete_matrix(S6);
    delete_matrix(S7);
    delete_matrix(S8);
    delete_matrix(S9);
    delete_matrix(S10);
    delete_matrix(F1);
    delete_matrix(F2);
    delete_matrix(F3);
```

```
    delete_matrix(F4);
    delete_matrix(F5);
    delete_matrix(F6);
    delete_matrix(F7);
    return 0;
}

//求分块矩阵
matrix sub_matrix(matrix Mat, int row, int col, int row_sub, int col_sub)
{
    matrix sub = create_matrix_all_zero(row_sub,col_sub);
    long long temp = 0;
    for (int i = 0; i < row_sub; ++i)
    {
        for (int j = 0; j < col_sub; ++j)
        {
            temp = get_by_index(Mat,i+row,j+col);
            set_by_index(sub,i,j,temp);
        }
    }
    return sub;
}
//合并矩阵
matrix Sum(matrix C11, matrix C12, matrix C21, matrix C22)
{
    long long temp1, temp2, temp3, temp4;
    temp1=  temp2 =  temp3 =  temp4 = 0;
    matrix sum = create_matrix_all_zero(C11.m_row * 2, C11.m_col * 2);
    for (int i = 0; i < C11.m_row; ++i)
    {
        for (int j = 0; j < C11.m_col; ++j)
        {
            temp1 = get_by_index(C11,i,j);
            temp2 = get_by_index(C12,i,j);
            temp3 = get_by_index(C21,i,j);
            temp4 = get_by_index(C22,i,j);

            set_by_index(sum,i,j,temp1);
            set_by_index(sum,i,(j + C11.m_col),temp2);
            set_by_index(sum,(i + C11.m_row),j,temp3);
            set_by_index(sum,(i + C11.m_row),(j + C11.m_col),temp4);
        }
    }
    return sum;
}
```

Only when the dimension of the square matrix is large enough, Strassen algorithm has its significant optimization effect. When I test 1024*1024 matrix, strassen algorithm take 7.5 seconds, faster than the origin methods (10.35s).

Part4-Obstacles and Solutions

4.1Modulo Arithmetic

Although i have used modulo in calculation before, I have never understood the definition and rules of modular operations in detail. This become a problem in this assignment. At first, I only do modulo arithmetic to the data itself, which cause Integer types cannot store the result (The program return 0 value). Then I check the definition of modulo arithmetic.

> Given a positive integer p, any integer n, there must be an equation:
> n = kp + r ； k and r are both integer.

> - **(a + b) % p = (a%p + b%p) %p**
> - **(a - b) % p = ((a%p - b%p) + p) %p**
> - **(a * b) % p = (a%p)*(b%p) %p**

This allowed me to learn the rules of data processing for large number operations.

4.2 Library function reference error

The giving files are regular **.hpp** and **.cpp**, but an error is reported at compile time:

> *Undefined* symbols for architecture x86_64

This is because when a template is instantiated during compilation, the compiler creates a new class with the given template parameter (templates cannot be compiled into code), only the result of the instantiation of the template. If Undefined symbols appears and does not belong to the structure of your own code, it is largely due to the library function reference error.

The solution is to use inline functions, putting the contents of**.cpp** directly into**.hpp**.

4.3 Segmentation fault

> zsh: segmentation fault

A segment error is when you access more memory than the program is allocated. Once a program has an out-of-bounds access, the CPU will generate the corresponding exception protection.

Since I 'm not familiar with the basic functions that construct matrix, I can only check it by set **printf** to ever step to check the value, Tracking possible locations of segment errors in the code. In the same time, the printf function can be included with conditional compilation directives **#ifdef DEBUG** and **#endif**, and the -**ddebug** parameter can be used to view debugging information during compilation.

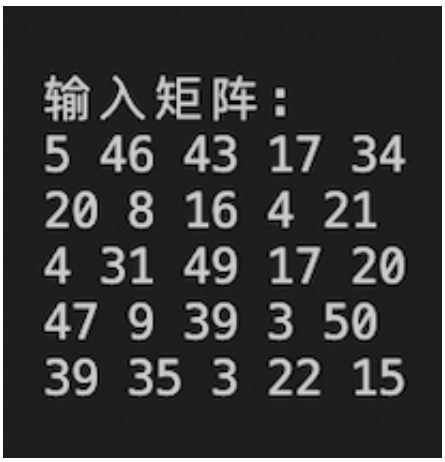
> 防止Segmentation fault的出现需要注意：
>
> - 定义了指针以后记得初始化，在使用的时候记得判断是否为NULL；
> - 在使用数组的时候是否被初始化，数组下标是否越界，数组元素是否存在等；

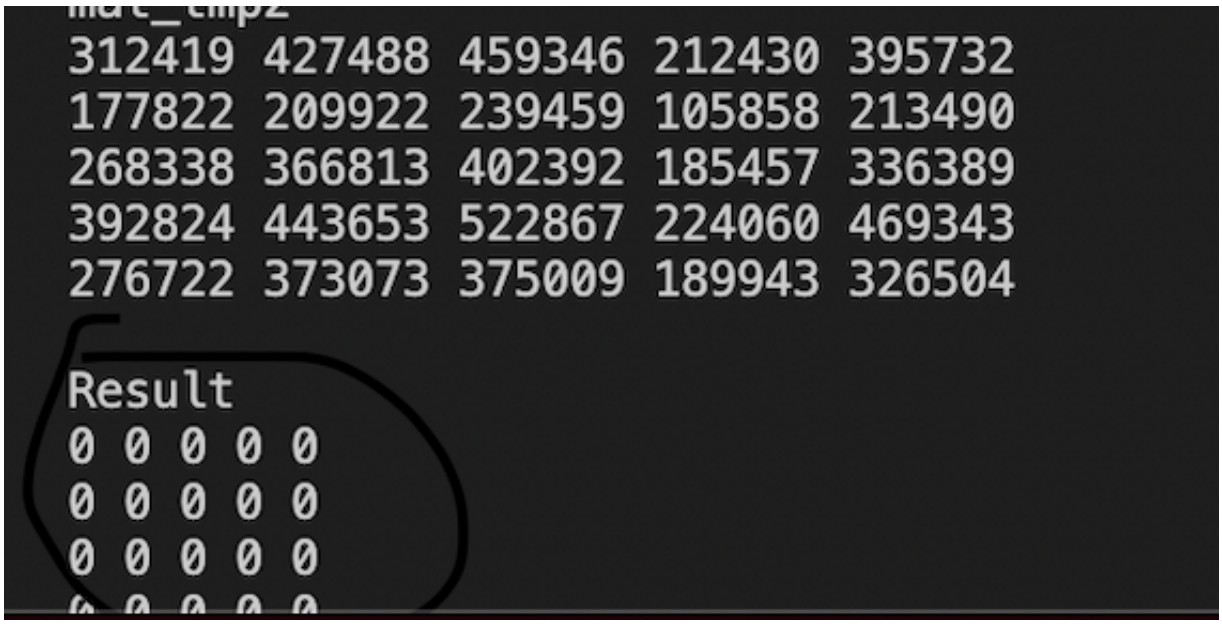
> - 在变量处理的时候变量的格式控制是否合理等；
>
>   Quote：https://blog.csdn.net/u010150046/article/details/77775114?spm=1001.2101.3001.6650.1&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1.pc_relevant_default&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1.pc_relevant_default&utm_relevant_index=2

I solve it by checking every loops and matrix i have created. And finally i found it is due to memory leak(delete unused matrix).

4.4 Matrix Address

The problem was found when I test my program locally. The output result matrix is always zero matrix. I put **printf** in the end of the functions, which prove my calculation is correct. Obviously, the only problem is when the function return mat_res value.





Originly, in the front of the function, in order to assure the mat_res is zero matrix, I set **mat_res** to zero matrix by using **create_matrix_all_zero**.

```
mat_res = creat_matrix_all_zero(mat_a.row, mat_a.col);
```

The function didn't make elements in matrix **mat_res** to be all zero, but instead, it create a new matrix(new struct address where puts all the value), so the calculation result will not be returned to the real parameter **mat_res**.

```
matrix create_matrix_all_zero(int row, int col) {
    matrix mat;
    mat.m_col = col;
    mat.m_row = row;
    mat.m_data_size = row * col * sizeof(int);
    mat.m_data = malloc(mat.m_data_size);
    memset(mat.m_data, 0, mat.m_data_size);
    return mat;
}
```

All the value of matrix elements is stored in pointer **m_data**.

```
typedef struct matrix {
    int m_col;  ///< number of columns in the matrix, aka. width
    int m_row;  ///< number of rows in the matrix, aka. height
    size_t m_data_size;  ///< number of *bytes* in the data zone, aka. m_col *
m_row * sizeof(int)
    void *m_data;
} matrix;
```

Although I didn't construct the matrix(struct) by myself and the lecture haven't mentioned knowledge about pointer and parameter, I manage to understand the concept of address.

Part 5-Conclusion

1.对于Strassen的方法的学习做一个小总结，

通过递归分治、Strassen算法将矩阵乘法原本的时间复杂度O(n^3)降到了O(n^(lg7)) = O(n ^ 2.81)，看起来确实是一个不错的优化。但在矩阵大小为2048 x 2048时，n ^ 3 = 8.59e9，n ^ 2.81 = 2.02e9，相差不超过一个数量级。但多次递归的开销抵消掉了分治带来的加速，导致运行速度变慢。至于如何减少递归次数、优化这个方法的结构还有待思考。矩阵乘法一般意义上还是选择的是朴素的方法，只有当矩阵变稠密，而且矩阵的阶数很大时，才会考虑使用Strassen算法。

采用Strassen算法作递归运算，需要创建大量的动态二维数组，且这个主程序的编写是基于给定的矩阵操作函数的，需要频繁的调用外部函数和递归运算，其中分配堆内存空间将占用大量计算时间，从而掩盖了Strassen算法的优势。对Strassen算法的改进，设定一个界限，将strassen优势最明显的范围突出出来，并对底层函数进行优化，使得调用所需的操作尽可能的少。由于项目作业的时间条件限制，这个工作没有彻底完成。需要更多的时间和数据进行测试。

2.从矩阵操作中的收获

矩阵计算作为众多领域的底层计算类别占据非常重要的地位，比如在机器人控制中，轨迹预测和规划都依靠着矩阵的计算。其计算的速度和质量，更是成为各个领域关键的发展因素。这一点也让我非常感兴趣。

本次作业最大的收获之一就是让我初步了解什么是内存泄漏以及学会了一些如何避免内存泄露的方法。小至占用内存，大至导致程序崩溃都是它会导致的问题，

对于本次作业的形式，虽然给定的底层函数操作函数让我们这些初学者能够非常轻松的上手完成任务，但是在深入了解考虑到优化时，发现只有了解底层函数的操作逻辑才能更好让程序执行起来更有效率、更加优雅（比如函数可以设置为void型函数）。这激发了我继续学习C/C++的兴趣。