# CS205 C/C++ Programming-Lab Assignment1 Report

**Name:** 邱逸伦 (Qiu Yilun)
**SID:** 12013006

## Part 1 - 函数传递和返回的参数问题

*Problem Description:*

在矩阵乘法的函数fast_matrix_exp中，我使用了mat_fac, mat_temp和mat_an作为求矩阵快速幂的中间矩阵，其中mat_an被初始化为行数为mat_a.m_row且列数为mat_a.m_col的单位矩阵，mat_temp被初始化为行数为mat_a.m_row且列数为mat_a.m_col的元素全为0的矩阵，mat_fac矩阵由mat_a使用copy_matrix函数复制而得。

根据快速幂算法的伪代码，矩阵快速幂仅仅只需将answer的值设为单位矩阵mat_an，将power_factor的值设为mat_fac。即可以得到矩阵快速幂的伪代码如下（其中矩阵乘法由@表示）。

```
Function matrix_quick_power(mat_a, n) -> Matrix:
    mat_an <- identity matrix;
    mat_fac <- mat_a;
    WHILE n is not 0:
        IF n is odd:
            mat_an <- ansmat_anwer @ mat_fac;
        mat_fac <- mat_fac @ mat_fac;
        n = floor(n/2);
    return mat_an;
```

根据矩阵乘法matrix_multiplication函数中传入的参数定义，传入的参数为两个进行矩阵乘法运算的矩阵和一个用于接受结果的矩阵（且传入的用于矩阵乘法运算的矩阵不能和接受结果的矩阵相同）。故而在代码中我使用mat_temp作为中间矩阵来存储中间运算的结果，在得到结果后通过copy_matrix函数拷贝mat_temp矩阵得到的输出。代码片段如下所示。

```
while (exp != 0) {
    if ((exp & 1) == 1) {
        matrix_multiplication(mat_an, mat_fac, mat_temp);
        mat_an = copy_matrix(mat_temp);
    }
    matrix_multiplication(mat_fac, mat_fac, mat_temp);
    mat_fac = copy_matrix(mat_temp);
    exp = floor(exp / 2);
}
```

经如此步骤后，得到存储最终结果的矩阵是mat_an，在最后我仍尝试使用copy_matrix函数拷贝得到mat_an中存储的数值如下代码所示。

```
mat_res = copy_matrix(mat_an);
```

但此时得到的最终结果始终是WA。

*Problem Analysis:*

为测试函数的运行结果，我定义了一个宏用来获取矩阵的元素的值，具体定义如下所示。

```
#define PRINT_MATRIX(matrix) \
for (int i = 0;i < (matrix.m_row); i++) {for (int j = 0;j < (matrix.m_col);j++) \
 {printf("%d ", get_by_index(matrix, i, j));}printf("\n");}
```

在测试函数运行时，我在fast_matrix_exp函数对mat_an和mat_res进行PRINT_MATRIX操作，在fast_matrix_exp函数中打印的结果都是正常得到的结果，而在主函数测试打印最终得到的mat_res函数却不正确。

为解决此问题，我通过debug发现问题的根源如下：

要解决此问题，应当从copy_matrix函数入手。copy_matrix函数的实现如下代码所示。

```
matrix copy_matrix(matrix mat) {
    matrix _mat = mat;
    _mat.m_data = malloc(_mat.m_data_size);
    memcpy(_mat.m_data, mat.m_data, _mat.m_data_size);
    return _mat;
}
```

> mat_res①：fast_martix_exp函数中的mat_res
> mat_res②：main函数（测试使用的函数）中的mat_res

在copy_matrix函数中，定义了一个新的矩阵_mat，这个_mat.m_data参数使用malloc方法动态申请了一段和原矩阵相同大小的内存，并使用memcpy方法拷贝了原矩阵这段内存的信息。在使用copy_matrix函数拷贝得到mat_an中存储的数值时，mat_an存储的数值是被拷贝到fast_matrix_exp函数的mat_res①中，此时的mat_res①因为作为copy_matrix函数的返回值，其m_data指向的内存空间地址已经在copy_matrix中重新申请而发生变化，故而此时修改的是mat_res①的m_data参数的内存空间，而mat_res②主函数中的m_data指向的内存空间并没有发生相应的改变，故而输出是错误的。

*Problem Handling：*

综上所述，想要拷贝mat_an矩阵的值至主函数的mat_res矩阵中，不能使用copy_matrix函数，而是应该循环遍历mat_an矩阵的每一个值单独去赋值给mat_res矩阵的相应位置。故而我实现了一个copy_function_matrix函数，函数具体实现如下所示。

```
matrix copy_function_matrix(matrix mat_ori, matrix mat_res){
    for (int i = 0; i < mat_ori.m_row; i++) {
        for (int j = 0; j < mat_ori.m_col; j++) {
            int val = get_by_index(mat_ori, i, j);
            set_by_index(mat_res, i, j, val);
        }
    }
    return mat_res;
}
```

最后在得到mat_an矩阵后，使用以下代码，即可得到正确的答案。

```
mat_res = copy_function_matrix(mat_an, mat_res);
```

对于naive_matrix_exp函数也是进行了同上的类似操作。

---

## Part 2 - 内存管理的分析

在实现矩阵的快速幂时，我初始化了mat_fac, mat_temp和mat_an等中间矩阵。在矩阵乘法运算过程中，我使用了多次copy_matrix函数去拷贝一个矩阵的值至另一矩阵，如以下代码：

```
mat_fac = copy_matrix(mat_temp);
```

该代码是位于计算矩阵快速幂的while循环中的，在此行代码执行前mat_fac已经存在并从堆申请了相应的内存空间，如果在循环内再次执行该段代码且不做任何其他处理，会再次申请一段未被使用的内存空间，显然我们不再需要之前的矩阵，而原来申请的那段内存空间则在内存中不再被使用且一直被原来的矩阵占用着。随着循环次数的增加和矩阵规模的增大，被占用的内存越来越大，若不及时释放，最终会导致程序崩溃。故而在每次执行copy_matrix函数前，我都调用了delete_matrix函数释放mat_fac矩阵申请的内存空间，在调用copy_matrix函数时允许它申请新的内存空间。通过debug操作我发现释放掉内存空间后，copy_matrix函数仍然申请的是之前释放的内存空间。同理对其他函数中的矩阵和中间矩阵最终也是调用delete_matrix函数释放了它们占用的内存空间。这样做轻量化了程序占用的内存，也有效避免了程序因为占用内存过大而发生错误。

在使用delete_matrix时，通过free函数释放了待删除矩阵的m_data指向的内存空间，但并没有将该指针置为空指针（null）。之所以这样是因为些待删除矩阵也都是定义在函数里，调用delete_matrix后不会再使用该删除的矩阵，无需考虑之后使用这些野指针会造成修改已经被释放的内存空间数据的问题。

## Part 3 - 使用中间矩阵的原因

*Problem Analysis:*

在fast_matrix_exp等函数进行矩阵运算中，我声明并初始化了mat_fac, mat_temp和mat_an等中间矩阵。虽然通过delete_matrix函数在运算完成后将这些中间矩阵占用的内存空间释放，但在运算时仍占用了部分内存，并且初始化也需要时间。

*Problem Explanation:*

在写代码之初，我尝试想直接用mat_res代替mat_an矩阵，但因为matrix_multiplication的第三个传入参数需要有一个矩阵来返回运算结果，故而mat_temp必须存在用于存储中间运算结果，而计算结果后要想更新mat_res有两种方法：

> 使用copy_matrix函数
> 使用copy_function_matirx函数

对于第一种方法，因为直接使用copy_matrix函数修改了mat_res.m_data的内存空间，即使在每次申请前我们可以通过delete_matrix函数释放原来申请的内存空间，去新申请空间，经过我的测试，大多数情况下新申请的内存空间就是我调用delete_matrix函数前mat_res.m_data指向的内存空间，但如果出现新申请的空间不是原来空间的情况下，在最终输出时主函数（测试函数）内的mat_res矩阵指向的内存空间仍无法进行相应的修改，输出结果会发生错误。

对于第二种方法，copy_function_matrix函数的时间复杂度为O($m\_row \times m\_col$)，在函数中多次调用会导致整体时间复杂度很高，故而不使用。

使用mat_fac, mat_temp和mat_an等中间矩阵时，初始化和copy_matrix、delete_matrix等操作时间复杂度都是O(1)，且对mat_res已申请的内存空间不会产生不良影响，故而我使用这种方式进行矩阵快速幂的计算。

## Part 4 - 矩阵乘法的优化

*Problem Analysis:*

最开始我使用的矩阵乘法方式为最常规的以ijk顺序的三层循环嵌套的方式，具体的代码如下所示

```
int matrix_multiplication(matrix mat_a, matrix mat_b, matrix mat_res) {
    if (mat_a.m_col != mat_b.m_row || mat_a.m_row != mat_res.m_row || mat_b.m_col
!= mat_res.m_col) {
        return 1;
    }
    for (int i = 0; i < mat_a.m_row; i++) {
        for (int j = 0; j < mat_b.m_col; j++) {
            long long val = 0;
            for (int k = 0; k < mat_a.m_col; k++) {
                long long a_val = (long long) get_by_index(mat_a, i, k) % MODULO;
                long long b_val = (long long) get_by_index(mat_b, k, j) % MODULO;
                val = (val + (a_val * b_val) % MODULO) % MODULO;
            }
            set_by_index(mat_res, i, j, (int) val);
        }
    }
    return 0;
}
```

该方法我是通过先用参数i遍历矩阵mat_a的每一行，再用参数j遍历矩阵mat_b的每一列，循环最内层用参数k遍历矩阵mat_a的每一列（矩阵mat_b的每一行），在循环时我可以用get_by_index函数得到矩阵mat_a和矩阵mat_b对应位置的值并相乘，把矩阵乘法计算最终值赋值给矩阵mat_res。但此方法运行高维矩阵的效率较低，故而我尝试使用接下来的方法进行矩阵乘法的优化。

*Problem Explanation:*

此处对优化方法的解释主要参考了文章https://zhuanlan.zhihu.com/p/146250334。

以下代码为已经优化后的代码：

```
int matrix_multiplication(matrix mat_a, matrix mat_b, matrix mat_res) {
    if (mat_a.m_col != mat_b.m_row || mat_a.m_row != mat_res.m_row || mat_b.m_col
!= mat_res.m_col) {
        return 1;
    }
    matrix mat_temp = create_matrix_all_zero(mat_res.m_row, mat_res.m_col);
    for (int i = 0; i < mat_a.m_row; i++) {
        for (int k = 0; k < mat_a.m_col;k++) {
            long long a_val = get_by_index(mat_a, i, k);
            for (int j = 0; j < mat_b.m_col; j++) {
                long long b_val = get_by_index(mat_b, k, j);
                long long val = get_by_index(mat_temp, i, j);
                val = (val + (a_val * b_val) % MODULO) % MODULO;
                if (k == mat_a.m_col-1){
                    set_by_index(mat_res, i, j, val);
                }else{
                    set_by_index(mat_temp, i, j, val);
                }
            }
        }
    }
    delete_matrix(mat_temp);
    return 0;//Advanced Method
}
```

该方法的具体实现介绍如下：

首先，使用此方法我需要每次获取最终结果矩阵的元素进行运算，且最终结果矩阵的初始值一定为0，鉴于我们未知用于存储结果的矩阵mat_res的初始元素的值，我使用了一个中间矩阵mat_temp，mat_temp初始化为和mat_res的size相同以防止mat_res不为元素全为0的矩阵。在进行矩阵乘法运算时，我先使用参数i遍历矩阵mat_a的每一行,再使用参数k遍历矩阵mat_a的每一列，此时我定义了$a_val = mat_a[i][k]$，并使用参数j遍历矩阵mat_b的每一列，在循环最内部我定义了$b_val = mat_b[k][j]$，$val = mat\_temp[i][j]$，则$val$的值通过运算得到为$(val + (a_val * b_val)$。若k的值已经是循环能得到的最大值，该$val$将被直接赋值为$mat_res[i][j]$的最终值；若不是，该$val$值将被赋值为$mat_temp[i][j]$的值。当三层循环全部完成时，mat_res已经被赋值上计算得到的最终结果，此时我调用delete_matrix函数来释放mat_temp占用的内存空间。

在该assignment中，题目给定的matrix使用m_data用malloc方法申请了一段连续的内存空间用于存储矩阵的行和列的值，故而在此可以将矩阵视作使用一维的数组按行存储用于存储矩阵的值。若假设mat_a和mat_b的size相同为$n \times n$，该方法的理论时间复杂度和原来的方法同为O($n^3$)，在算法层面上并没有明显的优化，而是基于硬件方面进行优化。原来的方法访问矩阵中存储的每个元素的值是基于内存跳跃访问的，即内存访问不连续，这会导致cache（高速缓冲存储器）命中率不高。在文章https://zhuanlan.zhihu.com/p/146250334中定义了跳跃数的概念，用于衡量访问的不连续程度。此处跳跃数的单位为次，n次表示矩阵每次跳跃了n个单位元素去访问下一个我们需要访问到的元素。

下面将对original method的跳跃数和advanced method的跳跃数进行分析（假定矩阵$mat_a$和矩阵$mat_b$的size同为$n \times n$）：

对于original method使用ijk顺序实现矩阵乘法，在此我依次计算了$mat_res$中的每个元素时需要将$mat_a$对应的行的元素与$mat_b$对应的列的元素依次相乘相加。根据上方的分析，矩阵$matrix$是以一维数组的方式按行存储的。在$mat_a$相应行中不断向右移动时，内存访问是连续的，按二维展开的形式要访问下一行的元素时直接由相应行的末尾连续访问到下一个元素即可。但$mat_b$相应列移动时，按二维展开形式是需要从前一行跳跃到下一行，在此我们使用连续一维内存存储时则需要跳跃n个元素模拟访问到下一行的元素，故而跳跃数为n次。计算$mat_res$的所有$n^2$个元素的跳转数为$n^3$次。

对于advanced method使用ikj顺序实现矩阵乘法，此方法对$mat_a$，$mat_b$和$mat_temp$的访问都是连续的。当计算$mat_temp$中任一行i的元素时先访问$mat_a$中相应行i的每一个元素$a_val = mat_a[i][k]$，k在$mat_a$每一行移动时都是对内存进行连续访问的，且外层i的值发生变化即遍历的$mat_a$的行发生变化时，我也可以连续地访问下一行的第一个元素（矩阵一维连续存储方式决定）；而对于每一个变化的k值我需要得到$mat_b$相应k行的每个值$b_val = mat_b[k][j]$，j在$mat_b$每一行移动时都是对内存连续进行访问的，且外层k的值发生变化时，我也可以连续地访问下一行的第一个元素（矩阵一维连续存储方式决定）；在i，j值确定后我可以得到$mat_temp$矩阵在行为i，列为j位置的值，并通过运算$val = (val + (a_val * b_val)$，得到其更新值并赋值给$mat_temp[i][j]$，此时因为在循环最里层以参数j在$mat_temp$的每一行移动时对内存的访问都是连续的，当k发生变化时，i在最外层尚未发生改变，此时对

$mat_temp$的求值需从$j=1$（下标始于1）重新开始，故而此处内存访问跳跃了n个元素，对于n个k值，跳跃数为$n^2$次；而最外层i发生改变为i+1时，内存是由访问$mat_temp[i][n]$的值到访问$mat_temp[i+1][1]$的值，这两值在内存中是连续存储的（矩阵一维连续存储方式决定）。故而计算$mat_temp$的所有元素的跳跃数为$n^2$。

由上方两种不同方法的跳跃数的对比显然发现advanced method的跳跃数显著小于original method，对于advanced method提升cache的命中率能有效地在硬件层面提高程序的性能，此处不列举ijk其他排列顺序计算矩阵乘法的跳跃数，通过比较可以得到以advanced method使用的ikj顺序进行矩阵乘法的计算所需跳跃数最小。

下方程序运行结果为两种矩阵乘法的方法运行速度的比较：

```
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ gcc assign1_test.c
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ ./a.out
Time cost of original method = 0.000045
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ gcc assign1_test.c
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ ./a.out
Time cost of advanced method = 0.000057
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$
```

Figure.1 size = 10 * 10

```
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ gcc assign1_test.c
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ ./a.out
Time cost of original method = 0.048698
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ gcc assign1_test.c
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ ./a.out
Time cost of advanced method = 0.027211
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$
```

Figure.2 size = 100 * 100

```
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ gcc assign1_test.c
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ ./a.out
Time cost of original method = 62.568338
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ gcc assign1_test.c
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$ ./a.out
Time cost of advanced method = 34.188671
olin@LAPTOP-K531Q585:~/Code/2022-Spring-C-and-C++/Assignment1$
```

Figure.3 size = 1000 * 1000

*Note: 第一个输出为使用original method得到的程序运行时间，第二个输出为使用advanced method得到的程序运行时间。*

由以上三张运行结果图可以发现当矩阵的size较小时，两种矩阵乘法的方法的程序运行时间差别不大；但当矩阵的size较大时，advanced method的程序运行时间显著低于original method，对于矩阵size = 1000 * 1000优化后的程序运行时间约为原来程序运行时间的54.64%。故而我选用advanced method作为计算矩阵乘法的最终方法。

用于测试程序运行时间的代码如下所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "assign1.c"
#include "assign1_mat.c"

int main(){
    int row = 10000;
    int col = 10000;
    matrix m = create_matrix_all_zero(row, col);
    matrix m2 = create_matrix_all_zero(row, col);
    matrix m_res = create_matrix_all_zero(row, col);
    int MAX = 2147483647;
    srand((unsigned int)time(0));
    for (int i = 0; i < row; i++){
        for (int j = 0; j < col; j++){
            int val = rand();
            val = val % (MAX - 1);
            set_by_index(m, i, j, val);
        }
    }
    for (int i = 0; i < row; i++){
        for (int j = 0; j < col; j++){
            int val = rand();
            val = val % (MAX - 1);
            set_by_index(m2, i, j, val);
        }
    }
    clock_t start, finish;
    double time;
    start = clock();
    matrix_multiplication(m, m2, m_res);
    finish = clock();
    time = (double) (finish - start) / CLOCKS_PER_SEC;
    printf("Time cost of original method = %lf\n", time);
    printf("Time cost of advanced method = %lf\n", time);
    //使用时以注释形式选用不同的method
}
```

# Part 5 - 最终代码

```
#include <stdlib.h>
#include <math.h>

#include "assign1.h"
#include "assign1_mat.h"

matrix set_identity_matrix(matrix mat);


matrix copy_function_matrix(matrix mat_ori, matrix mat_res);


matrix set_identity_matrix(matrix mat) {
    for (int i = 0; i < mat.m_row; i++) {
        for (int j = 0; j < mat.m_col; j++) {
            if (i == j) set_by_index(mat, i, j, 1);
            else set_by_index(mat, i, j, 0);
        }
    }
    return mat;
}


matrix copy_function_matrix(matrix mat_ori, matrix mat_res) {
    for (int i = 0; i < mat_ori.m_row; i++) {
        for (int j = 0; j < mat_ori.m_col; j++) {
            int val = get_by_index(mat_ori, i, j);
            set_by_index(mat_res, i, j, val);
        }
    }
    return mat_res;
}


int quick_power(int x, int n) {
    if (n == 0) {
        return 1;
    }
    long long partition_factor;
    partition_factor = quick_power(x, floor(n / 2)) % MODULO;
    if ((n & 1) == 1) {
        return ((partition_factor * partition_factor) % MODULO) * (x % MODULO) %
MODULO;
    } else {
        return (partition_factor * partition_factor) % MODULO;
    }
```

```
}


int matrix_addition(matrix mat_a, matrix mat_b, matrix mat_res) {
    if (mat_a.m_row != mat_b.m_row || mat_a.m_row != mat_res.m_row ||
        mat_a.m_col != mat_b.m_col || mat_a.m_col != mat_res.m_col) {
        return 1;
    }
    for (int i = 0; i < mat_res.m_row; i++) {
        for (int j = 0; j < mat_res.m_col; j++) {
            int a_val = (long long) get_by_index(mat_a, i, j) % MODULO;
            int b_val = (long long) get_by_index(mat_b, i, j) % MODULO;
            int val = (a_val + b_val) % MODULO;
            set_by_index(mat_res, i, j, val);
        }
    }
    return 0;
}


// int matrix_multiplication(matrix mat_a, matrix mat_b, matrix mat_res)
{//original method
//     if (mat_a.m_col != mat_b.m_row || mat_a.m_row != mat_res.m_row ||
mat_b.m_col != mat_res.m_col) {
//         return 1;
//     }
//     for (int i = 0; i < mat_a.m_row; i++) {
//         for (int j = 0; j < mat_b.m_col; j++) {
//             long long val = 0;
//             for (int k = 0; k < mat_a.m_col; k++) {
//                 long long a_val = (long long) get_by_index(mat_a, i, k) %
MODULO;
//                 long long b_val = (long long) get_by_index(mat_b, k, j) %
MODULO;
//                 val = (val + (a_val * b_val) % MODULO) % MODULO;
//             }
//             set_by_index(mat_res, i, j, (int) val);
//         }
//     }
//     return 0;
// }


int matrix_multiplication(matrix mat_a, matrix mat_b, matrix mat_res) {//advanced
method
    if (mat_a.m_col != mat_b.m_row || mat_a.m_row != mat_res.m_row || mat_b.m_col
!= mat_res.m_col) {
        return 1;
    }
    matrix mat_temp = create_matrix_all_zero(mat_res.m_row, mat_res.m_col);
    for (int i = 0; i < mat_a.m_row; i++) {
        for (int k = 0; k < mat_a.m_col; k++) {
            long long a_val = (long long) get_by_index(mat_a, i, k);
            for (int j = 0; j < mat_b.m_col; j++) {
```

```
                long long b_val = (long long) get_by_index(mat_b, k, j);
                long long val = (long long) get_by_index(mat_temp, i, j);
                val = (val + (a_val * b_val) % MODULO) % MODULO;
                if (k == mat_a.m_col - 1) {
                    set_by_index(mat_res, i, j, (int) val);
                } else {
                    set_by_index(mat_temp, i, j, (int) val);
                }
            }
        }
    }
    delete_matrix(mat_temp);
    return 0;
}


int naive_matrix_exp(matrix mat_a, int exp, matrix mat_res) {
    if (mat_a.m_col != mat_a.m_row || mat_a.m_col != mat_res.m_col || mat_a.m_row
!= mat_res.m_row) {
        return 1;
    }
    if (exp == 1) {
        mat_res = copy_function_matrix(mat_a, mat_res);
        return 0;
    } else if (exp == 0) {
        mat_res = set_identity_matrix(mat_res);
        return 0;
    }
    matrix mat_b = copy_matrix(mat_a);
    for (int i = 0; i < exp - 1; i++) {
        matrix mat_temp = create_matrix_all_zero(mat_a.m_row, mat_a.m_col);
        matrix_multiplication(mat_b, mat_a, mat_temp);
        delete_matrix(mat_b);
        mat_b = copy_matrix(mat_temp);
        delete_matrix(mat_temp);
    }
    mat_res = copy_function_matrix(mat_b, mat_res);
    delete_matrix(mat_b);
    return 0;
}


int fast_matrix_exp(matrix mat_a, long long exp, matrix mat_res) {
    if (mat_a.m_col != mat_a.m_row || mat_a.m_col != mat_res.m_col || mat_a.m_row
!= mat_res.m_row) {
        return 1;
    }
    if (exp == 1) {
        mat_res = copy_function_matrix(mat_a, mat_res);
        return 0;
    } else if (exp == 0) {
        mat_res = set_identity_matrix(mat_res);
        return 0;
    }
```

```
    /**
     * mat_fac denotes the factor matrix.
     * mat_temp is a transient matrix.
     * mat_an is the final matrix of result.
     */
    matrix mat_fac = copy_matrix(mat_a);
    matrix mat_temp = create_matrix_all_zero(mat_a.m_row, mat_a.m_col);
    matrix mat_an = create_matrix_all_zero(mat_a.m_row, mat_a.m_col);
    mat_an = set_identity_matrix(mat_an);
    while (exp != 0) {
        if ((exp & 1) == 1) {
            matrix_multiplication(mat_an, mat_fac, mat_temp);
            delete_matrix(mat_an);
            mat_an = copy_matrix(mat_temp);
        }
        matrix_multiplication(mat_fac, mat_fac, mat_temp);
        delete_matrix(mat_fac);
        mat_fac = copy_matrix(mat_temp);
        exp = floor(exp / 2);
    }
    delete_matrix(mat_fac);
    delete_matrix(mat_temp);
    mat_res = copy_function_matrix(mat_an, mat_res);
    delete_matrix(mat_an);
    return 0;
}


int fast_cal_fib(long long n) {
    if (n == 0) {
        return 0;
    } else if (n == 1 || n == 2) {
        return 1;
    }
    matrix mat = create_matrix_all_zero(2, 2);
    set_by_index(mat, 0, 0, 1);
    set_by_index(mat, 0, 1, 1);
    set_by_index(mat, 1, 0, 1);
    matrix mat_pow = create_matrix_all_zero(2, 2);
    fast_matrix_exp(mat, n - 1, mat_pow);
    matrix mat_ori = create_matrix_all_zero(2, 1);
    set_by_index(mat_ori, 0, 0, 1);
    matrix mat_res = create_matrix_all_zero(2, 1);
    matrix_multiplication(mat_pow, mat_ori, mat_res);
    delete_matrix(mat);
    delete_matrix(mat_pow);
    delete_matrix(mat_ori);
    int val = get_by_index(mat_res, 0, 0);
    delete_matrix(mat_res);
    return val;
}
```