CS205 C/ C++ Programming_Assignment1

Name: Lizinan

SID: 12011517

Part 1. Quick Power

- ### Analysis

  The algorithm is included in the document. Using Divide and conquer, the quick power algorithm has time complexity of O(logn).

- ### Code

```
//To increase code simplicity
int Mod(long long num) {
    return (int) (num % MODULO);
}
```

```
int quick_power(int x, int n) {
    long long ans = 1;
    long long power_factor = x % MODULO;
    while (n != 0) {
        if (n % 2 == 1) {
            ans = Mod(ans * power_factor);
        }
        power_factor = Mod(power_factor * power_factor);
        n = n / 2;
    }
    return (int) ans;
}
```

  To avoid the overflow during calculating, the type of ans and power_factor are set to long long.

- ### Result & Verification

```
//Test case 1
input: 959465599 478516499
output: 646341436
```

- ### Difficulties & Solutions

  None.

Part 2. Matrix Addition

- ### Analysis

  Check the size of input matrix and avoid the overflow.

- ### Code

```
int matrix_addition(matrix mat_a, matrix mat_b, matrix mat_res) {
    if (mat_res.m_row ≠ mat_a.m_row ||
        mat_res.m_col ≠ mat_a.m_col ||
        mat_res.m_row ≠ mat_b.m_row ||
        mat_res.m_col ≠ mat_b.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 val = Mod(Mod(get_by_index(mat_a, i, j)) + Mod(get_by_index(mat_b, i,
j)));
            set_by_index(mat_res, i, j, val);
        }
    }
    return 0;
}
```

- **Result & Verification**

```
//Test case 1
input:
8 6
35821 18763 20546 58794 52184 48332
19840 45964 85507 85237 26737 16378
17317 80350 5597 95712 64100 8403
73730 59033 52620 70634 89885 25324
35822 80579 75824 94951 83748 29821
37707 19568 48583 74604 94713 17119
39287 30904 63082 24793 16141 6170
41170 49809 2871 63119 45520 83323
8 6
87873 35601 42355 40492 6234 32239
82167 58407 12818 57990 53357 12917
4163 91063 32484 69097 82018 43548
86215 21305 74452 65648 46097 6944
88170 3619 56752 91040 83089 18623
90714 70961 54223 33069 27804 76808
81659 9970 35214 10828 84311 88570
23744 4825 95985 72579 73922 78002
output:
123694 54364 62901 99286 58418 80571
102007 104371 98325 143227 80094 29295
21480 171413 38081 164809 146118 51951
159945 80338 127072 136282 135982 32268
123992 84198 132576 185991 166837 48444
128421 90529 102806 107673 122517 93927
120946 40874 98296 35621 100452 94740
64914 54634 98856 135698 119442 161325
```

- **Difficulties & Solutions**

None

Part 3. Matrix Multiplication

- ## Analysis

  To improve the efficiency of this algorithm, I switch the order of matrix multiplication. The slow matrix multiplication is caused by **discontinuous memory access** in addition to the algorithm. When we use naive algorithm calculating matrix multiplication, each time we access object in mat_b will result in the interruption of memory access. By **switching the ordering of calculation** we can greatly increase the continuity of memory access. Using this method can **improve the performance of the function upto five-fold without increasing the amount of code**.

  To avoid overflow, type long long is used during the multiplication.

  In this function, I create a new matrix m to save the data and copy the data into mat_res at the end of function. In this way we can get rid of the initial value of may_res.

- ## Code

```
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 m = create_matrix_all_zero(mat_res.m_row, mat_res.m_col);
    for (int i = 0; i < mat_a.m_row; ++i) {
        for (int j = 0; j < mat_a.m_col; ++j) {
            int tmp = Mod(get_by_index(mat_a, i, j));//增加内存读取连续性
            for (int k = 0; k < mat_b.m_col; ++k) {
                long long tmp2 = (long long)tmp * (long long)Mod(get_by_index(mat_b, j,
k));
                set_by_index(m, i, k, Mod(get_by_index(m, i, k) + Mod(tmp2)));
            }
        }
    }
    memcpy(mat_res.m_data, m.m_data, mat_res.m_data_size);
    return 0;
}
```

- ## Result & Verification

```
//Test case
input:
1111111 1111111 1111111 1111111
1111111 1111111 1111111 1111111
1111111 1111111 1111111 1111111

9999 9999 9999 9999
9999 9999 9999 9999
9999 9999 9999 9999
9999 9999 9999 9999

output:
```

```
439995248 439995248 439995248 439995248
439995248 439995248 439995248 439995248
439995248 439995248 439995248 439995248
```

- ## Difficulties & Solutions

  At first I don't create a new matrix m to store the data, later when I write the fast matrix exponential function I notice that the initial value of mat_res could not be all zero, so I add matrix m. This change is convenient for me to reuse this function.

Part 4. Naive Matrix Exponentiation

- ## Analysis

  Using a loop and calculate the matrix exponentiation A_n in $O(n * size^3)$

- ## Code

```
int naive_matrix_exp(matrix mat_a, int exp, matrix mat_res) {
    if (mat_a.m_col ≠ mat_res.m_col ||
        mat_a.m_row ≠ mat_res.m_row ||
        mat_a.m_col ≠ mat_a.m_row ||
        exp < 0) {
        return 1;
    }
    matrix tmp = copy_matrix(mat_a);
    memcpy(mat_res.m_data, mat_a.m_data, mat_res.m_data_size);
    for (int i = 0; i < exp - 1; ++i) {
        matrix_multiplication(mat_a, tmp, mat_res);
        memcpy(tmp.m_data, mat_res.m_data, tmp.m_data_size);
    }
    return 0;
}
```

- ## Result & Verification

```
//Test case 1
input:
-1
99999 99999 99999 99999
99999 99999 99999 99999
99999 99999 99999 99999
99999 99999 99999 99999
output:
No change in mat_res
//Test case 2
input:
2
99999 99999 99999 99999
99999 99999 99999 99999
99999 99999 99999 99999
99999 99999 99999 99999
```

```
output:
999199731 999199731 999199731 999199731
999199731 999199731 999199731 999199731
999199731 999199731 999199731 999199731
999199731 999199731 999199731 999199731
```

- ## Difficulties & Solutions

  At first I forget that negative exp is not allowed.

Part 5. Fast Matrix Exponentiation

- ## Analysis

  It's the same as part1.

- ## Code

```
int fast_matrix_exp(matrix mat_a, long long exp, matrix mat_res) {
    if (mat_a.m_col ≠ mat_res.m_col ||
        mat_a.m_row ≠ mat_res.m_row ||
        mat_a.m_col ≠ mat_a.m_row ||
        exp < 0) {
        return 1;
    }
    matrix ans = create_matrix_all_zero(mat_res.m_row, mat_res.m_col);
    matrix tmp = create_matrix_all_zero(mat_res.m_row, mat_res.m_col);
    matrix power_factor = copy_matrix(mat_a);
    for (int i = 0; i < ans.m_row; ++i) {
        set_by_index(ans, i, i, 1);
    }
    while (exp ≠ 0) {
        if (exp % 2 == 1) {
            matrix_multiplication(ans, power_factor, ans);//It's good to ignore the
initial value of ans
        }
        matrix_multiplication(power_factor, power_factor, tmp);
        memcpy(power_factor.m_data, tmp.m_data, power_factor.m_data_size);
        exp = exp / 2;
    }
    memcpy(mat_res.m_data, ans.m_data, mat_res.m_data_size);
    return 0;
}
```

- ## Result & Verification

```
//Test case
input:
4 4 4386051
610446003 10819871 973870786 207662558
143512851 209222431 938140794 529468601
487530415 130922941 525057024 79299563
979399355 911257220 609210072 544595211
output:
260677687 893691021 328160037 906788090
484826694 681795293 811597451 173188768
116195179 550859957 183778738 110352072
353845041 998587269 341566667 733860558
```

- ## Difficulties & Solutions

  I set the matrix ans as a matrix all is 1 at beginning, but soon I realized it's wrong.

Part 6. Fibonacci Sequence

- ## Analysis

  We can prove by mathematical induction that the Fibonacci sequence satisfies: $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$

  So we can use fast_matrix_exp function we just finish to calculate F_n

- ## Code

```
int fast_cal_fib(long long n) {
    matrix mat = create_matrix_all_zero(2, 2);
    matrix mat_res = 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);
    fast_matrix_exp(mat, n, mat_res);
    return get_by_index(mat_res, 0, 1);
}
```

- ## Result & Verification

```
//Test case
input:
11111111111
output:
715537859//Moduled 1e9 + 7
```

- ## Difficulties & Solutions

  None