CS205 C/C++ Program Design Assignment 1

Author: gdjs2, chris, oierVICTOR
Semester: 2022 Fall

Introduction

In Assignment 1, you are required to finish some simple functions related to **POWER(Exponentiation)** and **MATRIX** to help you master the basic C/C++ syntax. As result, you can find a definitely novel way to calculate Fibonacci sequence.

Tasks

Part 1. Quick Power (20pts)

Part 2. Matrix Addition and Matrix Multiplication (50pts)

Part 3. Naive Matrix Exponentiation (15pts)

Part 4. Fast Matrix Exponentiation (15pts)

Part 5. Fibonacci Sequence (20pts)

(You can find an online doc for your tasks via https://cs205-s22.github.io/assign1)

POWER

POWER is a critical concept in mathematics. As we know, we can use repeatition to calculate x^n in $\mathrm{O}(n)$, which is named by **TRADITIONAL POWER**. In this part, we will introduce another more efficient method, the **QUICK POWER**, to calculate power in $\mathrm{O}(\log n)$.

To help you get started, we will give you the pseudo code for **TRADITIONAL and QUICK POWER** separately.

Traditional Power

TRADITIONAL POWER calculating x^n:

```
Function traditional_power(x, n) -> Integer:
    answer <- 1;
    REPEAT for n times:
        answer = answer * x;
    RETURN answer;
```

Part One - Quick Power

The **QUICK POWER** is a typical application of *Divide and Conquer*, suppose we need to calculate x^n:

1. If n is even, we can recursively derive the result $x^n = x^{n/2} \times x^{n/2}$.
2. Otherwise, $x^n = x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} \times x$.
3. The boundary condition is $x^n = 1$, when $n == 0$.

QUICK POWER calculating x^n with recursion:

```
Function quick_power_recursion(x, n) -> Integer:
    IF n is 0:
        RETURN 1;
    partition_factor <- quick_power_recursion(x, floor(n/2));
    IF n is odd:
        RETURN partition_factor * partition_factor * x;
    ELSE:
        RETURN partition_factor * partition_factor;
```

Also, we can optimize the recursive version by non-recursive one:

```
Function quick_power_non_recursion(x, n) -> Integer:
    answer <- 1;
    power_factor <- x;
    WHILE n is not 0:
        IF n is odd:
            answer <- answer * power_factor;
        power_factor <- power_factor * power_factor;
        n = floor(n/2);
```

WHAT YOU SHOULD DO (20 pts in total):

1. Read the document **CAREFULLY**!
2. Implement the function `quick_power` (20 pts) in `assign1.c`.

ATTENTION:

1. **NO GRADES** will be given unless you implement the correct **QUICK POWER**!

2. **ALL TEST CASES** are valid, i.e., $x \geq 0 \cap n \geq 0$. You do not need to handle exceptions.
3. Because the result r may be too large. You **need** to calculate a, in which $a \equiv r \pmod{10^9 + 7}$. i.e., you need to modulo all results by $10^9 + 7$ when necessary. The constant MODULO has been defined as $10^9 + 7$ in `assign1_mat.h`. You can use the variable directly.
4. Please pay more attention to your code style. After all this is not ACM-ICPC contest. You will get deduction if your code style is terrible. You can read Google C++ Style Guide, NASA C Style Guide or some other guide for code style.

MATRIX

Matrix is useful in linear algebra and computer science. In this part, you are required to finish several functions corresponding to matrix, including addition and multiplication. We suppose that you all have basic background knowledge about matrix. If not, you can check the website or search the internet by yourself.

Matrix

We will not introduce the concept of matrix here, but the matrix structure we provide to you in `assign1_mat.h` and `assign1_mat.c`. You do not need to understand the functions' implementation and the definition of the structure while you need to know how to use the APIs provided to manipulate a matrix.

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.

You can assume the Matrix structure as a two-dimensional array, which has `col` and `row`. Like the index of array, both `col` and `row` start from `0`. So be careful when you index element by `set_by_index` and `get_by_index`.

Scalar Multiplication

To help you get started, we have finished the scalar multiplication in `assign1_mat.c`. It contains some usage of APIs to help you get started . We do following things in `scalar_multiplication` function:

1. Check whether the size of the result container variable `mat_res` matches the original matrix `mat_a`. If the size checking passes, we will continuing the calculation; or return 1 which represents the failing of size checking.
2. For each entry of the matrix, get the original value, do the multiplication and set to the entry of the result matrix.
3. We do the modulo opeartion (`%MODULO`) in the multiplication to prevent integer overflow. You need to follow this rule in the following parts needed to be implemented by yourself.

Part Two - Matrix Addition and Matrix Multiplication

You need to implement the matrix addition and multiplication.

WHAT YOU SHOULD DO (50 pts in total):

1. Read the document and code we provided to you CAREFULLY!
2. Implement the function `matrix_addition` (20 pts) and `matrix_multiplication` (30 pts) in `assign1.c`.

ATTENTION:

1. **NO GRADES** will be given unless you implement the correct **MARTIX ADDITION and MATRIX MULTIPLICATION**!
2. **Do** the size checking in the function and show the result of operation by return value. If the return value of your function is not 0, **DO NOT** modify the values in `mat_res`.
3. **DO** the modulo operation(`%MODULO`) during the multiplication and be attention to the overflow!
4. It is **GUARANTEED** that `mat_res` is different from `mat_a` or `mat_b` in both addition and multiplication when we test your functions. But **BE SURE** you will not call the these functions with the same `mat_res` and `mat_a` or same `mat_res` and `mat_b` either!

FAST MATRIX EXPONENTIATION

Till now, we have known quick power and matrix multiplication. Suppose we replace x in x^n by a matrix with size $size \times size$. We got the **EXPONENTIATION of MATRIX**. The **EXPONENTIATION of MATRIX** is very important in computing recursive derivation and spanning tree counting.

For example: the traditional method to calculate the Fibonacci Sequence is using a loop and calculating the recursive derivation in $\mathrm{O}(n)$. We have recursive formula: $f_n = f_{n-1} + f_{n-2}$, unless $f_0 = 0, f_1 = 1$.

We can transform the recursive formula into matrix multiplication in the following way:

1. Construct matrix $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
2. Suppose there is a vector $\begin{bmatrix} f_{n-1} \\ f_{n-2} \end{bmatrix}$
3. Do the multiplication between the first matrix and the second vector: $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} f_{n-1} \\ f_{n-2} \end{bmatrix} = \begin{bmatrix} f_{n-1} + f_{n-2} \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix}$. We successfully got f_n and the vector which can be used to calculate the next item f_{n+1}.
4. We can do the multiplication between the first matrix and the vector we got from step 3:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} f_n + f_{n-1} \\ f_n \end{bmatrix} = \begin{bmatrix} f_{n+1} \\ f_n \end{bmatrix}$$

5. If we put the initial value f_0 and f_1 into the vector and do the matrix multiplication:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} f_1 = 1 \\ f_0 = 0 \end{bmatrix} = \begin{bmatrix} f_2 = 1 \\ f_1 = 1 \end{bmatrix}$$

6. Because the matrix multiplication satisfies the law of association:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} f_2 = 1 \\ f_1 = 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} f_1 = 1 \\ f_0 = 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \times \begin{bmatrix} f_1 = 1 \\ f_0 = 0 \end{bmatrix} = \begin{bmatrix} f_3 = 2 \\ f_2 = 1 \end{bmatrix}$$

7. Finally, we got:

$$\begin{bmatrix} f_3 \\ f_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \times \begin{bmatrix} f_1 \\ f_0 \end{bmatrix}$$

8. It is easy to find:

$$\begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} f_1 \\ f_0 \end{bmatrix}$$

We will focus on how to calculate the exponentiation of a matrix like $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$ in the following.

Part Three - Naive Matrix Exponentiation

Just like traditional power calculation, we can also use a loop and calculate the matrix exponentiation A^n in $\mathrm{O}(n \times size^3)$, in which size is the one-dimensional size of matrix A. In this part, you are required to implement a naive matrix exponentiation by yourself.

WHAT YOU SHOULD DO (15 pts in total):

1. Implement the function `naive_matrix_exp` (15 pts) in `assign1.c`.

ATTENTION:

1. **NO GRADES** will be given unless you implement the correct **MATRIX EXPONENTIATION**!
2. **Do** the size checking in the function and show the result by return value. If the return value of your function is not 0, **DO NOT** modify the values in `mat_res`.
3. **DO** the modulo operation(`%MODULO`) during the calculation and be attention to the overflow!
4. **REUSE** the code you have implemented!
5. It is **GUARANTEED** that `mat_res` is different from `mat_a`. Be **CAREFUL** about the arguments you passed to the matrix multiplication!

Part Four - Fast Matrix Exponentiation

Comparing the **MATRIX EXPONENTIATION** to the **EXPONENTIATION of NUMBERS**, if we can calculate the power of numbers by **QUICK POWER**, how can we use the same way to optimize the **MATRIX EXPONENTIATION** and reduce the comlexity of the calculation to $\mathrm{O}(size^3 \times \log n)$?

WHAT YOU SHOULD DO (15 pts in total):

1. **REVIEW** the pseudo code of **QUICK POWER**.
2. Think about how to optimize the **MATRIX EXPONENTIATION**?
3. Implement the function `fast_matrix_exp` (15 pts) in `assignment1.c`.

ATTENTION:

1. **NO GRADES** will be given unless you implement the correct **FAST MATRIX EXPONENTIATION**!
2. **Do** the size checking in the function and show the result by return value. If the return value of your function is not 0, **DO NOT** modify the values in `mat_res`.
3. **DO** the modulo operation(`%MODULO`) during the calculation and be attention to the overflow!
4. **REUSE** the code you have implemented!
5. If you do not get full score in the **Naive Matrix Exponentiation** part but do in this part, you will eventually get full scores for both parts.
6. It is **GUARANTEED** that `mat_res` is different from `mat_a`. Be **CAREFUL** about the arguments you passed to the matrix multiplication!
7. Be **AWARE** of the type of parameter `exp`!

Part Five - Fibonacci Sequence

At the beginning of this section, we introduce the **MATRIX EXPONENTIATION** by Fibonacci Sequence. In this part, you are required to use the knowledge you obtain, to calculate the Fibonacci Sequence using **Fast Matrix Exponentiation**.

In our definition, $f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2}$, when $n \geq 2$.

WHAT YOU SHOULD DO (20 pts in total):

1. **REVIEW** the relationship between the **RECURSIVE DERIVATION** and **MATRIX EXPONENTIATION**.
2. Implement the function `fast_cal_fib` (20 pts) in `assign1.c`.

ATTENTION:

1. **NO GRADES** will be given unless you implement the function with correct time complexity!
2. **DO** the modulo operation(`%MODULO`) during the calculation and be attention to the overflow!
3. **REUSE** the code you have implemented!
4. It is **guaranteed** that the arguments are valid, i.e., $0 < n \leq 10^{18}$.

Tips

Warnings

- Make sure to **CREATE** the source file `assign1.c` for your assignment. This file is in which you should implement all the required functions.
- Make sure that you implement all the functions: `quick_power`, `matrix_addition`, `matrix_multiplication`, `naive_matrix_exp`, `fast_matrix_exp`, `fast_cal_fib`. Even if you do not know how to finish several of them or there are some bugs, please **IMPLEMENT** them in source file as well! Or you may get **ZERO** for the whole code part in assignment.

What to Submit

Submit two files to Blackboard.

- assign1.c
- your assignment report, it needs not to be long, but should be able to explain the difficulties you encountered in completing the assignment and how you solve them. If you think there are highlights in your code, please point them out in the report. Both Chinese and English are allowed.

How can you judge yourself's program?

Our online judge (http://120.25.240.87/) provides you several public test cases.

Your user name is your **student id**, the initial password is **123456**. (Please change your password ASAP)

If you have problems using the oj, you can contact the SA: He Zean (qq: 317576256, or find me in the group chat)

Public test cases and results are available at GitHub