0.4, a and b only

Yin.Zheng

January 30, 2022

1 Code examples

```python
def fib_matrix(n):
    if n == 1:
        return 1
    if n == 2:
        return 1
    else:
        F1 = [0, 1, 1, 1]
        return naive_matrix_power(F1, n)[0]


def naive_matrix_power(F1, n):

    B = [1, 1]
    for _ in range(n-1):
        B = matrix_multiply(F1, B)
        global number_of_matrix_muliply
        number_of_matrix_muliply += 1
        # print(B)
    return B


def matrix_multiply(F1, B):
    a, b, c, d = F1
    x, y = B
    global add_time
    global multi_time
    add_time += 2
    multi_time += 4
    return (a*x+b*y, c*x+d*y)


number_of_matrix_muliply = 0
add_time = 0
multi_time = 0
n = 8  # input your nth fibonacci number here
print("the", n, "number of fibonacci number is", fib_matrix(n))
print("the number of matrix muliply is", number_of_matrix_muliply)
print("the number of addition is performed is", add_time)
print("the number of multiplication is performed is", multi_time)
```

Listing 1: This is O(n) version of my fib matrix

```python

F1 = [0, 1, 1, 1]


def matrix_fib(n):
    if n == 1:
        return 1
    if n == 2:
        return 1
    else:
        result = matrix_power(F1, n)
        return result[1]


def matrix_power(F1, n):
    if n == 0:
        return [1, 0, 0, 1]
    elif n == 1:
        return F1
    else:
        B = F1
        i = 2
        while i <= n:
            # repeated square B until n = 2^q > m
            B = matrix_multiply_f2(B, B)
            global global_N
            global_N += 1
            i = i*2
        # add on the remainder
        R = matrix_power(F1, n-i//2)
        return matrix_multiply_f2(B, R)


def matrix_multiply_f1(F1, B):
    a, b, c, d = F1
    x, y = B
    return (a*x+b*y, c*x+d*y)


def matrix_multiply_f2(A, B):  # this function returns  matrix A*B
    a, b, c, d = A
    x, y, z, w = B
    global add_time
    global multi_time
    add_time += 4
    multi_time += 8
    return (
        a*x + b*z,
        a*y + b*w,
        c*x + d*z,
        c*y + d*w,
    )


# 1,1,2,3,5,8,13,21,34,55
global_N = 0
add_time = 0
```

```
multi_time = 0
n = 8
print("the ", n, "th fib number is ", matrix_fib(n))
print("The is O(logn)", global_N)
print("the number of addition is performed is", add_time)
print("the number of multiplication is performed is", multi_time)
```

Listing 2: This is O(log(n)) version of my fib with matrix

(a) Q1:Show that two 2 * 2 matrices can multiplied using 4 additions and 8 multiplications. But how many matrix multiplications does it take to compute Xn?

Answer: In my first version of matrix fib it takes $N-1$ steps to compute X^n, Because every time n increases by 1, it is multiplied once more by [0,1,1,1] so the time complexity is O(n)

```
def naive_matrix_power(F1, n):

    B = [1, 1]
    for _ in range(n-1):
        B = matrix_multiply(F1, B)
        global number_of_matrix_muliply
        number_of_matrix_muliply += 1
        # print(B)
    return B


def matrix_multiply(F1, B):
    a, b, c, d = F1
    x, y = B
    global add_time
    global multi_time
    add_time += 2
    multi_time += 4
    return (a*x+b*y, c*x+d*y)


number_of_matrix_muliply = 0
add_time = 0
multi_time = 0
n = 8  # input your nth fibonacci number here
print("the", n, "number of fibonacci number is", fib_mat
print("the number of matrix muliply is", number_of_matri
print("the number of addition is performed is", add_time
print("the number of multiplication is performed is", mu
```

```
Print Output:

the 8 number of fibonacci number is 21
the number of matrix muliply is 7
the number of addition is performed is 14
the number of multiplication is performed is 28
```

Figure 1: Screen shot for Q1

(b) Q2:Show that O(logn) matrix multiplications suffice for computing Xn. (Hint:Think about computing X8.)

Answer:In my second version of matrix Fib, it takes it takes $log(n)$ steps to compute X^n Because in this version matrix Multiplication grows exponentially For example when n=8, we are calculating the 8th fib number. This is what happens in this loop:

```
        while i <= n:
            # repeated square B until n = 2^q > m
            B = matrix_multiply_f2(B, B)
            global global_N
            global_N += 1
            i = i*2

```

Listing 3: loop code

- When i=2 B became B^2
- when i=4 B^2 became B^4
- when i=8 B^4 became B^8
- when i=16 jump out of the loop

As you can see the matrix multiply function execute 3 times, $3 = log(8)$,so $O(logn)$ matrix multiplications suffice for computing X^n

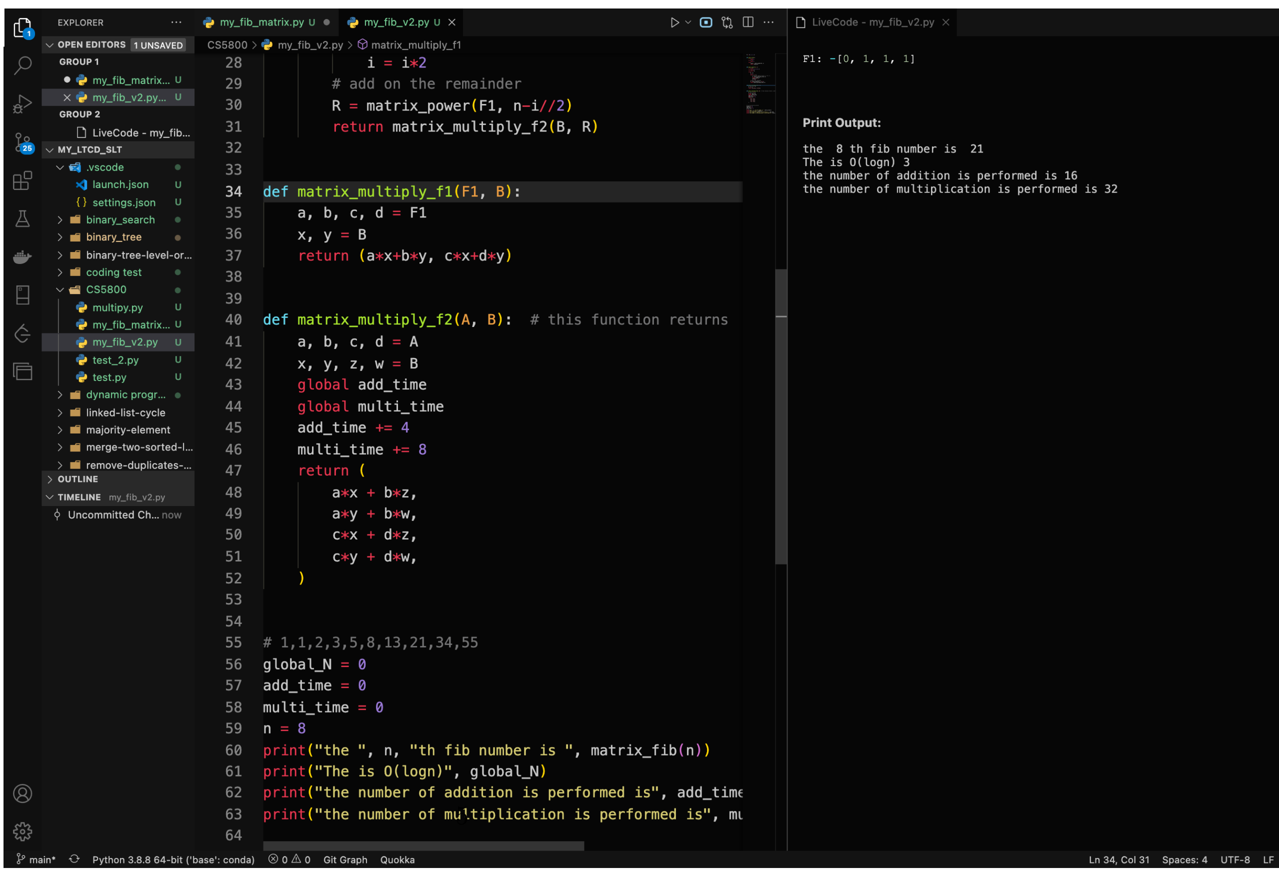

Figure 2: Screen shot for Q2