Fundamental Algorithm Techniques
EXO_3
Abusaid Aruzhan

Problem 1

“Fibonacci Super Fast!”

Purpose of this task is to calculate Fibonacci numbers very quickly.

So we all know that Fibonacci means? in simple words each number is the sum of the previous two:

Fn = Fn-1 + Fn-2

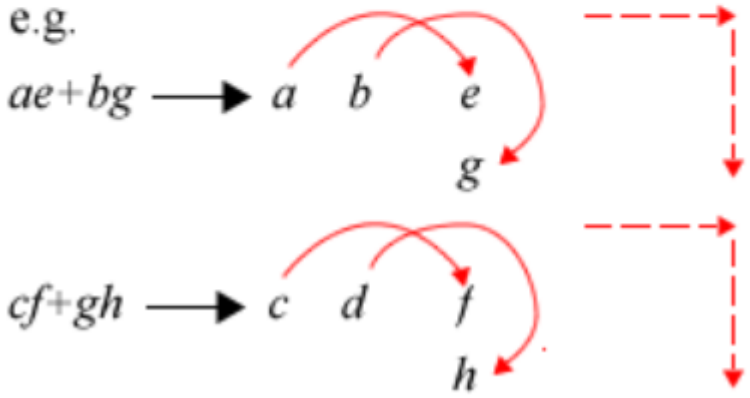
And supposedly, we were given a matrix formula to calculate fibonacci numbers faster:

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This means instead of doing many additions, you multiply a 2×2 matrix by itself many times. You can see the example for that below

1. $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e \\ g \end{pmatrix} = \begin{pmatrix} ae+bg \\ ce+dg \end{pmatrix}$
2. $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+gh \end{pmatrix}$

Notice the '7' pattern?



So the key idea is this:

Normal Fibonacci calculation takes a lot of time because it repeats many operations, thats why if we will use matrix multiplication with exponentiation by squaring (a trick to multiply fast), you can find only $\log_2(n)$ steps instead of n steps.

Simple explanation of "$\log_2(n)$":

Imagine you keep dividing n by 2 each time: 100 → 50 → 25 → 12 → 6 → 3 → 1
That's about $\log_2(100) \approx 7$ steps.

- Normal way: you add numbers again and again — O(n) steps. (1 million steps for $F_{1,000,000}$)
- Matrix way: you use fast exponentiation — multiply the matrix fewer times — only $\log_2(n)$ times. (only around 20 multiplications)

BUT what is this?

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \left(\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \right)^{n/2} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Step1:

What “complexity” means here:

We’re not being asked to calculate Fibonacci numbers.
We’re being asked: How many steps does it take to compute this matrix power?

Step2:

The “decomposition” (divide and conquer idea):

If you want to compute A to the power of n , you can do it like this:

If n is even:
  A^n = (A^(n/2)) × (A^(n/2))
If n is odd:
  A^n = A × (A^((n-1)/2)) × (A^((n-1)/2))

So each time, instead of multiplying the matrix *n times*, you cut the problem in half.

How it works??? I guess we have to look at this example

$$A^n = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}^n$$

Slow way: A × A × A × A × A × ... (it means that takes n multiplications)

Smart way: Use the rule: A^n = (A^(n/2)) × (A^(n/2))

That means we only need to find A^(n/2), then multiply it by itself.

So if n = 8:

- $A^8 = (A^4 \times A^4)$
- $A^4 = (A^2 \times A^2)$
- $A^2 = (A \times A)$

Conclusion, we only needed 3 multiplications, not 8, because each time we divide n by 2 until it becomes 1.

Step3:

The Master Theorem is a simple rule that tells us how many steps an algorithm like this takes when it keeps dividing the problem into smaller halves.

We describe the time needed as a recurrence:

$T(n) = T(n/2) + c$

where:

- T(n) = total time to compute A^n
- T(n/2) = time for the smaller problem (half of n)
- c = the small, constant work we do each step (like one matrix multiplication)

So the algorithm keeps solving smaller and smaller parts until n becomes 1.

If n = 8 → 8 → 4 → 2 → 1 → 3 divisions → $\log_2(8) = 3$
If n = 16 → 16 → 8 → 4 → 2 → 1 → 4 divisions → $\log_2(16) = 4$

Overall conclusion: We don't multiply the matrix n times.
We only multiply it $\log_2(n)$ times,
because each time we cut n in half (divide by 2).

That's why the total time complexity is $O(\log_2(n))$

there is ex of how it works:

```
#i try to make it fast using recursion and matrix multiplication
def multiply_matrix(a, b):
    #manually multiply 2x2 matrices with each other
    c = [[0, 0],
         [0, 0]]
    c[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0]
    c[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1]
    c[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0]
    c[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1]
    return c
```

below is the main recursive function.
It raises the matrix A to the power of n.
Instead of multiplying A by itself n times I divide the problem into halves:

- If n is even, I compute A^(n/2) once, then multiply it by itself.
- If n is odd, I do the same but multiply by one extra A at the end.

Each time the function divides n by 2, so the recursion depth is about $\log_2(n)$.
The variable counter just counts how many matrix multiplications happen.

```python
def power_matrix(A, n, counter=[0]):
    # recursive function to make matrix A to power n
    if n == 1:
        return A
    #if n even
    if n % 2 == 0:
        half = power_matrix(A, n // 2, counter)
        counter[0] += 1  #count multiplication
        return multiply_matrix(half, half)
    else:
        half = power_matrix(A, n // 2, counter)
        counter[0] += 2       #count mult
        return multiply_matrix(multiply_matrix(half, half), A)

#main part
A = [[1, 1],
     [1, 0]]

n = 16  #suppose n is 16
counter = [0]

result = power_matrix(A, n, counter)

print("A^", n, " = ", result)
print("Matrix multiplications:", counter[0])
```

```
A^ 16  =  [[1597, 987], [987, 610]]
Matrix multiplications: 4
```