1 Fibonacci Numbers with Linear Algebra: Matrix Exponentiation

(dix) Fibonacci Number

Let's talk about Fibonacci numbers. Yes, those numbers that start with 0 and 1, where every number after that is the sum of the two before it: 0,1,1,2,3,5,8,.... You know the drill.

On LeetCode the constraint is $n \leq 30$ for Fibonacci Number. I don't like small constraint here, just too, you know, small. Let's try another approach but first pay some respect to the old school solution.

1.0.1 1. Normal approach

Every genius will know how to do this using Dynamic Programming. Easy peasy. Let's have a quick recap.

The DP approach avoids the repetitive calculations of recursion by storing previously calculated Fibo values. This allows us to build up the sequence in one single pass, with time complexity of $O(n)$ and a space complexity of $O(1)$ with an iterative solution.

Here's the Python code:

```python
def fibonacci_dp(n):
  if n == 0
    return 0
  elif n == 1:
    return 1
  a, b = 0, 1

  for _ in range(2, n + 1):
    a, b = b, a + b

  return b
```

And now comes the interesting part.

1.0.2 2. Matrix exponentiation approach

1.0.2.1 What the heck is matrix exponentiation?

According Wikipedia

> In mathematics, the matrix exponential is a matrix function on square matrices analogous to the ordinary exponential function. It is used to solve systems of linear differential equations. In the theory of Lie groups, the matrix exponential gives the exponential map between a matrix Lie algebra and the corresponding Lie group.

Wtf? Don't panic, It is basically multiplying a square matrix by itself repeatedly.

1.0.2.2 How is a matrix related to Fibonacci number?

The Fibonacci sequence F_n is defined by the recurrence relation:

$$F_n = F_{n-1} + Fn - 2$$

with initial conditions $F_0 = 0$ and $F_1 = 1$.

This relation can be represented in matrix form as:

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

Define the tranformation matrix T as:

$$T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Then we can rewrite the recurrence relation as:

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = T \cdot \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$$

1.0.2.2.1 Base case

We check the base case with $n = 1$:

$$\begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = T \cdot \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

This is correct since $F_2 = 1$ and $F_1 = 1$

1.0.2.2.2 Induction step

Assume that for some $k \geq 1$, we have:

$$\begin{bmatrix} F_{k+1} \\ F_k \end{bmatrix} = T^k \cdot \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

We need to show that implies:

$$\begin{bmatrix} F_{k+2} \\ F_{k+1} \end{bmatrix} = T^{k+1} \cdot \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

Using the recurrence relation:

$$\begin{bmatrix} F_{k+2} \\ F_{k+1} \end{bmatrix} = T \cdot \begin{bmatrix} F_{k+1} \\ F_k \end{bmatrix}$$

Substitute the inductive hypothesis:

$$= T \cdot \left(T^k \cdot \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} \right) = T^{k+1} \cdot \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

Thus, by induction, the matrix representation for all $n \geq 1$. $Q.E.D.$

Okay, just let's these numbers and matrices go away. Lameeeeee!!!!. Give me the code

1.0.2.2.3 Implementation

```python
import numpy as np

def matrix_power(mat, power):
  result = np.identity(len(mat), dtype=object) #use dtype object to handle overflow
number
  base = mat.copy()
  while power > 0:
    if power % 2 == 1:
      result = np.dot(result, base)
    base = np.dot(base, base)
    power //= 2

    return result

def fibonacci_matrix_exp(n):
  if n == 0:
    return 0
  if n == 1:
    return 1

  T = np.array([[1, 1], [1, 0]], dtype=object)
  T_n_minus_1 = matrix_power(T,  n - 1)
  F_vector = np.array([1, 0], dtype=object)

  F_n_vector = np.dot(T_n_minus_1, F_vector)

  return F_n_vector[0]
```

The time complexity is $O(logn)$ and space complexity is $O(1)$. Why $O(logn)$?

- If n is even, you calculate $T^{\frac{n}{2}}$ and then square it to get T^n
- If n is odd, you multiply the matrix by T^{n-1}, where n is even, and you can apply the same halving process.

The number of times you need to halve the n to reach 1 is $log_2 n$. In each step, you either square the result or multiply by the matrix T once more. Thus, the total number of steps if $O(logn)$.

1.0.2.3 Benchmark

Let's create a benchmark function to evaluate 2 approaches, especially with large numbers.

```
def benchmark_fibonacci():
    test_values = [10, 100, 1000, 50000, 100000]  # Different values of n
    results = []

    for n in test_values:
        # Time DP Approach
        start_time = time.time()
        dp_result = fibonacci_dp(n)
        dp_time = time.time() - start_time

        # Time Matrix Exponentiation Approach
        start_time = time.time()
        matrix_result = fibonacci_matrix_exp(n)
        matrix_time = time.time() - start_time

        # Ensure both methods give the same result for correctness
        assert dp_result == matrix_result, f"Mismatch at n={n}"

        results.append((n, dp_time, matrix_time))
```

```python
    # Display results
    print("|   n    | DP Time (s)      | Matrix Exp Time (s) |")
    print("|--------|------------------|----------------------|")
    for n, dp_time, matrix_time in results:
        print(f"| {n:<6} | {dp_time:<16.6f} | {matrix_time:<20.6f} |")
```

And this is the result

```
|   n    | DP Time (s)     | Matrix Exp Time (s) |
|--------|------------------|----------------------|
| 10     | 0.000000         | 0.000000             |
| 100    | 0.000000         | 0.000000             |
| 1000   | 0.000000         | 0.000000             |
| 50000  | 0.048000         | 0.005000             |
| 100000 | 0.126000         | 0.014000             |
```

You can see that the matrix approach is quite better with large input n.

Yeah that's it for today!. Hope through this random note you can find it interesting to apply math into some coding problems.

> Nature is written in mathematical language.
>
> – Galileo Galilei

See you space cowboy.