1137. N-th Tribonacci Number

Easy Memoization Math Dynamic Programming Leetcode Link

Problem Description

The Tribonacci sequence is a variation of the Fibonacci sequence where each number is the sum of the three preceding ones. The sequence begins with `T_0 = 0`, `T_1 = 1`, and `T_2 = 1`. For all integers `n >= 0`, the `n+3`rd number of the sequence is calculated as `T_n+3 = T_n + T_n+1 + T_n+2`. The problem asks for the `T_n` value of the sequence given an integer `n`.

Intuition

The intuitive way to calculate the `n`th Tribonacci number would be to use recursion or iteration, starting from the base cases `T_0`, `T_1`, and `T_2`. However, this can be computationally expensive for larger `n` due to repeated calculations.

To optimize, one can use matrix exponentiation, realizing that each Tribonacci number can be obtained by multiplying a transformation matrix with the vector of the three previous Tribonacci numbers. The transformation matrix is:

```
[(1, 1, 0),
 (1, 0, 1),
 (1, 0, 0)]
```

By successively squaring this matrix and reducing the power, we apply an exponentiation by squaring technique which significantly reduces the number of multiplications needed. We only multiply the `res` vector by the transformation matrix when the current bit in the binary representation of `n` is set (using the bitwise AND operation `n & 1`). The operation `n >>= 1` is used to right shift the bits of `n` by one position to check the next bit in the next iteration. The intuition behind this approach comes from a realization that `n`th power of the transformation matrix can be broken down into powers of 2, which allows the sequence to be computed in a time complexity that is logarithmic to `n` instead of linear, making it much more efficient for large values of `n`.

Solution Approach

The solution is implemented using a class named `Solution` with a method `tribonacci` that takes an integer `n` as its parameter and returns the `n`th Tribonacci number.

To calculate the Tribonacci number efficiently, the solution applies matrix exponentiation. Here is a step-by-step explanation of the implementation:

1. **Base Cases**: For `n == 0`, return `0`. For `n < 3` (which covers `n == 1` or `n == 2`), return `1`.
2. **Transformation Matrix**: The `factor` variable is defined as a 3×3 transformation matrix using `numpy`:

   ```
   factor = np.mat([(1, 1, 0), (1, 0, 1), (1, 0, 0)], np.dtype("O"))
   ```

   Each multiplication of this matrix by a vector [T[n], T[n+1], T[n+2]] will result in [T[n+1], T[n+2], T[n+3]].
3. **Result Vector**: The `res` variable is defined as a 1×3 matrix to keep track of the current values of [T[n], T[n+1], T[n+2]]:

   ```
   res = np.mat([(1, 1, 0)], np.dtype("O"))
   ```

   Initially, it's set to represent [T[0], T[1], T[2]] (0, 1, 1) before the loop.
4. **Exponentiation by Squaring**: Before entering the loop, `n` is decremented by 3 as the initial `res` already accounts for the first three numbers. The loop then iterates, applying the following steps until `n` becomes 0:
   - **Conditional Multiplication**: If the least significant bit of `n` is 1 (`n & 1`), `res` is multiplied by the `factor` matrix.
   - **Matrix Squaring**: The `factor` matrix is squared which is equivalent to doubling the exponent.
   - **Right Shift**: The bits of `n` are shifted to the right (`n >>= 1`) to process the next bit in the next iteration of the loop.

   The loop effectively multiplies `res` by `factor` raised to the power of `n`, but it does so in logarithmic time relative to the value of `n`.
5. **Final Sum**: After the loop ends, the sum of the `res` array will contain the value of T[n], and this is returned as the final result:

   ```
   return res.sum()
   ```

In terms of data structures, the solution utilizes `numpy` matrices for efficient manipulation and multiplication of large arrays, which is pivotal in matrix exponentiation. The use of bit manipulation to check individual bits of the integer `n` is an efficient pattern often used in algorithms involving repetitive multiplication, as it greatly reduces the number of calculations required.

Example Walkthrough

Let's use `n = 4` as a small example to illustrate the solution approach.

1. **Initial Setup**: We are given the Tribonacci sequence with `T_0 = 0`, `T_1 = 1`, and `T_2 = 1`.
2. **Base Cases Handling**: For `n = 4`, it's not less than 3, so we don't return the base cases directly.
3. **Transformation Matrix**: We have a transformation matrix `factor`, which when multiplied by `[T_n, T_{n+1}, T_{n+2}]`, gives `[T_{n+1}, T_{n+2}, T_{n+3}]`.
4. **Result Vector Initialization**: We define the initial `res` vector as `[T_0, T_1, T_2]`, which is `[0, 1, 1]`.
5. **Exponentiation by Squaring Process**: Since `n = 4` and `res` covers the first three numbers we deduct 3 from `n`. Now `n` is `1`. The iteration would proceed as follows:
   - Start the loop with `n = 1`.
   - Since `n & 1` is `1` (`4 & 1` equals `1`), we multiply our `res` vector `[0, 1, 1]` by our `factor` matrix to get `[1, 1, 0]`.
   - We square our `factor` matrix, which doesn't affect the `res` vector for this small value of `n`.
   - Right-shift `n` by 1 (`n >>= 1`). Now `n` is `0`, which means our loop will terminate.
6. **Final Result**: The loop has ended, and `res` vector is `[1, 1, 0]`. The sum of `res` is `T_4 = 1 + 1 + 0 = 2`.

Therefore, the 4th Tribonacci number `T_4` is `2`.

Python Solution

```python
import numpy as np

class Solution:
    def tribonacci(self, n: int) -> int:
        # Base case: return 0 for the zero-th element in the sequence.
        if n == 0:
            return 0

        # Base cases: return 1 for the first or second element in the sequence.
        if n < 3:
            return 1

        # Define the transition matrix for the Tribonacci sequence.
        # This matrix represents the relationship between successive elements.
        transition_matrix = np.matrix([[1, 1, 0], [1, 0, 1], [1, 0, 0]], np.dtype("O"))

        # Initialize the result matrix as the identity matrix aligned with the sequence order.
        result_matrix = np.matrix([[1, 1, 0]], np.dtype("O"))
        n -= 3

        # Use the Fast Exponentiation algorithm to raise the matrix to the power of (n-3).
        while n:
            # When the current power is odd, multiply result_matrix with transition_matrix.
            if n & 1:
                result_matrix *= transition_matrix

            # Square the transition_matrix to get the next higher power of 2.
            transition_matrix *= transition_matrix

            # Shift right to divide n by 2, flooring it.
            n >>= 1

        # Return the sum of the elements in the resulting matrix as the answer.
        return result_matrix.sum()
```

Java Solution

```java
class Solution {
    // Calculates the n-th Tribonacci number
    public int tribonacci(int n) {
        // Base cases for n = 0, 1, 2
        if (n == 0) {
            return 0;
        }
        if (n < 3) {
            return 1;
        }
        // Transformation matrix for Tribonacci sequence
        int[][] transformationMatrix = {{1, 1, 0}, {1, 0, 1}, {1, 0, 0}};
        // Calculate the power of the matrix to (n-3), since we know the first three values
        int[][] resultingMatrix = matrixPower(transformationMatrix, n - 3);
        // Initialize answer
        int answer = 0;
        // Add the top row of the matrix to get the answer
        for (int element : resultingMatrix[0]) {
            answer += element;
        }

        return answer;
    }

    // Multiplies two matrices and returns the result
    private int[][] matrixMultiply(int[][] a, int[][] b) {
        int rows = a.length, cols = b[0].length;
        int[][] resultMatrix = new int[rows][cols];
        // Perform matrix multiplication
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                for (int k = 0; k < b.length; ++k) {
                    resultMatrix[i][j] += a[i][k] * b[k][j];
                }
            }
        }
        return resultMatrix;
    }

    // Calculates the exponentiation of a matrix 'a' raised to the power of 'n'
    private int[][] matrixPower(int[][] a, int n) {
        // Create an identity matrix for initial result
        int[][] result = {{1, 1, 0}};
        // Loop to do binary exponentiation
        while (n > 0) {
            // Multiply with 'a' when the least significant bit is 1
            if ((n & 1) == 1) {
                result = matrixMultiply(result, a);
            }
            // Square the 'a'
            a = matrixMultiply(a, a);
            // Right shift 'n' to process the next bit
            n >>= 1;
        }
        return result;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <numeric>

class Solution {
public:
    // Computes the n-th Tribonacci number using matrix exponentiation
    int tribonacci(int n) {
        // Base case when n is 0
        if (n == 0) {
            return 0;
        }
        // Base cases for n being 1 or 2
        if (n < 3) {
            return 1;
        }
        // Initial transformation matrix
        std::vector<std::vector<long long>> transformationMatrix = {
            {1, 1, 0},
            {1, 0, 1},
            {1, 0, 0}
        };
        // Raise the transformation matrix to the power (n - 3)
        std::vector<std::vector<long long>> resultMatrix = matrixPower(transformationMatrix, n - 3);
        // The sum of the first row of the result matrix gives us the nth Tribonacci number
        return std::accumulate(resultMatrix[0].begin(), resultMatrix[0].end(), 0);
    }

private:
    using ll = long long; // Alias for long long type for easier reading

    // Multiplies two matrices and returns the result
    std::vector<std::vector<ll>> matrixMultiply(std::vector<std::vector<ll>>& a, std::vector<std::vector<ll>>& b) {
        int rows = a.size(), columns = b[0].size();
        std::vector<std::vector<ll>> product(rows, std::vector<ll>(columns));

        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < columns; ++j) {
                for (int k = 0; k < b.size(); ++k) {
                    product[i][j] += a[i][k] * b[k][j];
                }
            }
        }
        return product;
    }

    // Calculates the power of a matrix to the given exponent n
    std::vector<std::vector<ll>> matrixPower(std::vector<std::vector<ll>> a, int n) {
        // Starting with the identity matrix of size 3x3
        std::vector<std::vector<ll>> result = {
            {1, 0, 0},
            {0, 1, 0},
            {0, 0, 1}
        };

        while (n) {
            if (n & 1) { // If n is odd, multiply the result by the current matrix a
                result = matrixMultiply(result, a);
            }
            a = matrixMultiply(a, a); // Square the matrix a
            n >>= 1; // Divide n by 2
        }
        return result;
    }
};
```

Typescript Solution

```typescript
// Calculates the n-th Tribonacci number using matrix exponentiation.
function tribonacci(n: number): number {
    // Base cases for n = 0, 1, and 2.
    if (n === 0) {
        return 0;
    }
    if (n < 3) {
        return 1;
    }

    // Matrix representation of the Tribonacci relation.
    const tribonacciMatrix = [
        [1, 1, 0],
        [1, 0, 1],
        [1, 0, 0],
    ];

    // Raise the matrix to the (n-3)-th power and sum the top row for the result.
    return matrixPower(tribonacciMatrix, n - 3)[0].reduce((sum, element) => sum + element);
}

// Multiplies two matrices and returns the result.
function multiplyMatrices(matrix1: number[][], matrix2: number[][]): number[][] {
    const rows = matrix1.length;
    const cols = matrix2[0].length;
    // Create an empty matrix for the result.
    const result = Array.from({ length: rows }, () => Array.from({ length: cols }, () => 0));

    // Calculate the product of the two matrices.
    for (let i = 0; i < rows; ++i) {
        for (let j = 0; j < cols; ++j) {
            for (let k = 0; k < matrix2.length; ++k) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }

    return result;
}

// Raises a matrix to the power of n and returns the result.
function matrixPower(matrix: number[][], n: number): number[][] {
    // Initialize the result as an identity matrix in the shape of a 1 x 3 matrix.
    let result = [
        [1, 1, 0],
    ];

    while (n > 0) {
        // If the exponent is odd, multiply the result by the matrix.
        if (n & 1) {
            result = multiplyMatrices(result, matrix);
        }
        // Square the matrix.
        matrix = multiplyMatrices(matrix, matrix);
        // Halve the exponent by right-shifting it.
        n >>= 1;
    }

    return result;
}
```

Time and Space Complexity

The given Python code implements a fast matrix exponentiation approach to find the `n`-th number in the Tribonacci sequence. The Tribonacci sequence is a generalization of the Fibonacci sequence, where each number is the sum of the preceding three numbers in the sequence.

Time Complexity

The time complexity of this solution mainly depends on the matrix exponentiation part of the algorithm. The algorithm uses a loop that runs in O(log n) time, which accounts for the repeated squaring of the matrix `factor`. In each iteration of the loop, the matrix is either multiplied by itself (squaring) which takes O(1) time since the matrix size is fixed at 3×3, or the result matrix `res` is multiplied by `factor`, which also takes O(1) time for the same reason. Thus, the overall time complexity of the algorithm is `O(log n)`.

Space Complexity

The space complexity of the algorithm is determined by the size of the matrices used for the calculations. The matrices `factor` and `res` have a constant size of 3×3 and 1×3, respectively, so the space used does not depend on `n`. Consequently, the space complexity is `O(1)`, which means it uses constant space.