<https://leetcode.com/problems/n-th-tribonacci-number/description/>

The Tribonacci sequence Tn is defined as follows:

T0 = 0, T1 = 1, T2 = 1, and Tn+3 = Tn + Tn+1 + Tn+2 for n >= 0.

Given n, return the value of Tn.

**Example 1:**

**Input:** n = 4

**Output:** 4

**Explanation:**T\_3 = 0 + 1 + 1 = 2T\_4 = 1 + 1 + 2 = 4

**Example 2:**

**Input:** n = 25

**Output:** 1389537

**Constraints:**

0 <= n <= 37

The answer is guaranteed to fit within a 32-bit integer, ie. answer <= 2^31 - 1

**Attempt 1: 2025-5-31**

**Solution 1: Native DFS (10 min, TLE 35/38)**

class Solution {

    public int tribonacci(int n) {

        if(n == 0) {

            return 0;

        }

        if(n == 1 || n == 2) {

            return 1;

        }

        return tribonacci(n - 1) + tribonacci(n - 2) + tribonacci(n - 3);

    }

}

Time Complexity: O(3^n) - Exponential due to repeated calculations

Space Complexity: O(n) - Recursion stack depth

**Solution 2: Memoization (10 min)**

class Solution {

    public int tribonacci(int n) {

        Integer[] memo = new Integer[n + 1];

        return helper(n, memo);

    }

    private int helper(int n, Integer[] memo) {

        if(n == 0) {

            return 0;

        }

        if(n == 1 || n == 2) {

            return 1;

        }

        if(memo[n] != null) {

            return memo[n];

        }

        return memo[n] = helper(n - 1, memo) + helper(n - 2, memo) + helper(n - 3, memo);

    }

}

Time Complexity: O(n) - Each subproblem solved once

Space Complexity: O(n) - For memoization array

**Solution 3: DP (10 min)**

class Solution {

    public int tribonacci(int n) {

        if(n == 0) {

            return 0;

        }

        if(n == 1 || n == 2) {

            return 1;

        }

        int[] dp = new int[n + 1];

        dp[0] = 0;

        dp[1] = 1;

        dp[2] = 1;

        for(int i = 3; i <= n; i++) {

            dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3];

        }

        return dp[n];

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 4: Space Optmized DP (10 min)**

class Solution {

    public int tribonacci(int n) {

        if(n == 0) {

            return 0;

        }

        if(n == 1 || n == 2) {

            return 1;

        }

        int[] dp = new int[n + 1];

        int a = 0;

        int b = 1;

        int c = 1;

        int d = 0;

        for(int i = 3; i <= n; i++) {

            d = a + b + c;

            a = b;

            b = c;

            c = d;

        }

        return d;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Solution 5: Math (60 min)**

class Solution {

    public int tribonacci(int n) {

        if(n == 0) {

            return 0;

        }

        if(n == 1 || n == 2) {

            return 1;

        }

        // Transformation matrix

        long[][] M = {{0, 1, 0},

                      {0, 0, 1},

                      {1, 1, 1}};

        // Initial state vector [T0, T1, T2]

        long[][] M0 = {{0},

                       {1},

                       {1}};

        // Compute M^n

        long[][] Mn = matrixPower(M, n);

        // Multiply M^n \* M0 to get final state

        // (order matters, must Mn \* M0, not M0 \* Mn)

        long[][] result = matrixMultiply(Mn, M0);

        return (int) result[0][0];

    }

    private long[][] matrixPower(long[][] matrix, int power) {

        // Identity matrix Property:

        // Multiplying any matrix by the identity matrix leaves it unchanged: A × I = A

        long[][] result = {{1, 0, 0},

                           {0, 1, 0},

                           {0, 0, 1}};

        long[][] tmp = matrix.clone();

        while(power > 0) {

            // If power is odd

            if(power % 2 == 1) {

                // Multiply by current temp, and in first time multiply

                // it will keep the tmp unchanged

                result = matrixMultiply(result, tmp);

            }

            // Square the temp matrix

            tmp = matrixMultiply(tmp, tmp);

            // Halve the power

            power /= 2;

        }

        return result;

    }

    private long[][] matrixMultiply(long[][] A, long[][] B) {

        int aRows = A.length;

        int aCols = A[0].length;

        int bCols = B[0].length;

        long[][] result = new long[aRows][bCols];

        for(int i = 0; i < aRows; i++) {

            for(int j = 0; j < bCols; j++) {

                for(int k = 0; k < aCols; k++) {

                    result[i][j] += A[i][k] \* B[k][j];

                }

            }

        }

        return result;

    }

}

Time Complexity: O(logn)

Space Complexity: O(n^2)

**Refer to Deepseek**

**Problem Description**

The Tribonacci sequence Tn is defined as follows:

T0 = 0, T1 = 1, T2 = 1

Tn+3 = Tn + Tn+1 + Tn+2 for n >= 0

Given n, return the value of Tn.

**Approaches**

**1. Recursion (Naive)**

**Time Complexity**: O(3^n) - Exponential due to repeated calculations

**Space Complexity**: O(n) - Recursion stack depth

**2. Dynamic Programming (Memoization)**

**Time Complexity**: O(n) - Each subproblem solved once

**Space Complexity**: O(n) - For memoization array

**3. Iterative DP (Optimal Space)**

**Time Complexity**: O(n)

**Space Complexity**: O(1) - Only stores last three values

**4. Matrix Exponentiation (Advanced)**

**Time Complexity**: O(log n)

**Space Complexity**: O(1)

**Solution Code**

**1. Recursive Solution (Naive)**

class Solution {

public int tribonacci(int n) {

if (n == 0) return 0;

if (n == 1 || n == 2) return 1;

return tribonacci(n-1) + tribonacci(n-2) + tribonacci(n-3);

}

}

**2. DP with Memoization**

class Solution {

public int tribonacci(int n) {

if (n == 0) return 0;

if (n == 1 || n == 2) return 1;

int[] memo = new int[n+1];

memo[0] = 0;

memo[1] = 1;

memo[2] = 1;

return helper(n, memo);

}

private int helper(int n, int[] memo) {

if (memo[n] != 0 || n == 0) return memo[n];

memo[n] = helper(n-1, memo) + helper(n-2, memo) + helper(n-3, memo);

return memo[n];

}

}

**3. Iterative DP (Optimal Space)**

class Solution {

public int tribonacci(int n) {

if (n == 0) return 0;

if (n == 1 || n == 2) return 1;

int a = 0, b = 1, c = 1;

for (int i = 3; i <= n; i++) {

int d = a + b + c;

a = b;

b = c;

c = d;

}

return c;

}

}

**4. Matrix Exponentiation (Logarithmic Time)**

class Solution {

public int tribonacci(int n) {

if (n == 0) return 0;

if (n == 1 || n == 2) return 1;

int[][] matrix = {{1, 1, 1}, {1, 0, 0}, {0, 1, 0}};

matrixPower(matrix, n - 2);

return matrix[0][0];

}

private void matrixPower(int[][] matrix, int power) {

int[][] result = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};

while (power > 0) {

if (power % 2 == 1) multiply(result, matrix);

multiply(matrix, matrix);

power /= 2;

}

System.arraycopy(result, 0, matrix, 0, result.length);

}

private void multiply(int[][] a, int[][] b) {

int[][] temp = new int[3][3];

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

temp[i][j] += a[i][k] \* b[k][j];

}

}

}

System.arraycopy(temp, 0, a, 0, temp.length);

}

}

**Key Takeaways**

**Recursive Solution** is simple but inefficient for larger n due to exponential time complexity.

**Memoization** improves performance by storing computed results.

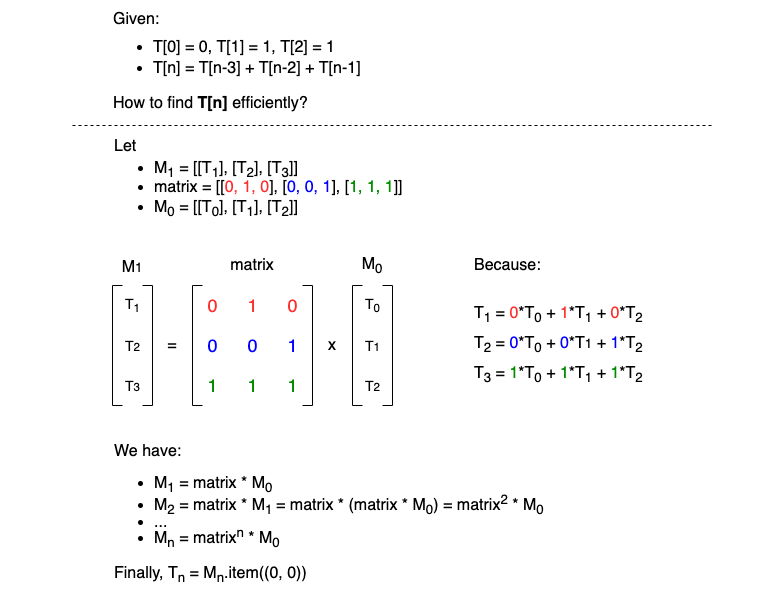
**Iterative DP** is the most space-efficient O(1) solution for this problem.

**Matrix Exponentiation** offers O(log n) time but is more complex to implement.

For most cases, the **iterative DP solution** (Solution 3) is recommended as it provides the best balance between simplicity and efficiency.

**Explain more on Matrix solution ?**

Matrix exponentiation is an advanced technique to solve linear recurrence relations (like the Tribonacci sequence) in **O(log n) time**, significantly faster than the O(n) DP approach. Here's a detailed breakdown of how it works for the Tribonacci problem:

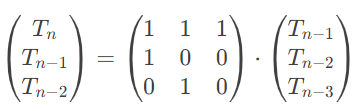


**Key Idea**

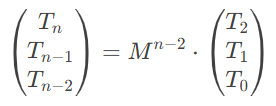
The Tribonacci sequence follows:

image.png

We can express this recurrence using **matrix multiplication**:



By repeatedly applying this transformation, we can derive:

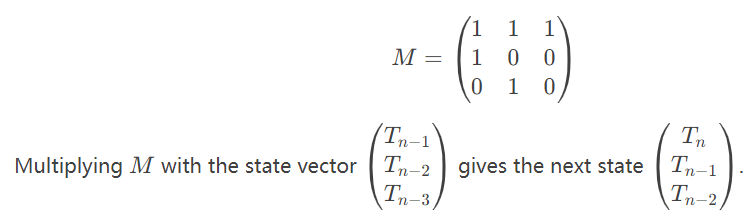


where *M* is the transformation matrix and *T*2​=1, *T*1​=1, *T*0​=0.

**Step-by-Step Explanation**

**1. Matrix Representation**

The transformation matrix *M* is:



**2. Matrix Exponentiation**

To compute *M^n* efficiently, we use **exponentiation by squaring**:

Compute *M*1,*M*2,*M*4,*M*8,… by repeatedly squaring.

Combine results based on the binary representation of *n*.

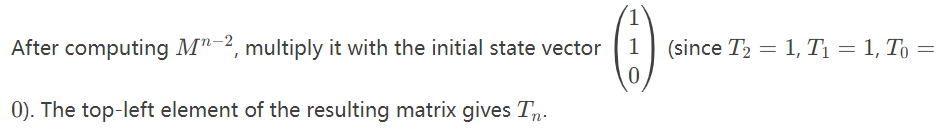
**Example**: For *n*=4: *M^*4=*M^*2⋅*M^*2

**3. Base Cases**

For *n*=0, return 0.

For *n*=1 or *n*=2, return 1.

**4. Result Extraction**



**Solution 1: NOT use the initial state Matrix, which is NOT intuitive**

class Solution {

public int tribonacci(int n) {

if (n == 0) return 0;

if (n == 1 || n == 2) return 1;

int[][] matrix = {{1, 1, 1}, {1, 0, 0}, {0, 1, 0}};

matrixPower(matrix, n - 2);

return matrix[0][0]; // T\_n is the top-left element

}

// Raises the matrix to the (power)-th exponent

private void matrixPower(int[][] matrix, int power) {

int[][] result = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}; // Identity matrix

while (power > 0) {

if (power % 2 == 1) {

multiply(result, matrix);

}

multiply(matrix, matrix);

power /= 2;

}

// Copy result back to the original matrix

System.arraycopy(result, 0, matrix, 0, result.length);

}

// Multiplies two 3x3 matrices

private void multiply(int[][] a, int[][] b) {

int[][] temp = new int[3][3];

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

temp[i][j] += a[i][k] \* b[k][j];

}

}

}

System.arraycopy(temp, 0, a, 0, temp.length);

}

}

**Solution 2: Use the initial state Matrix, which is intuitive**

class Solution {

public int tribonacci(int n) {

if (n == 0) return 0;

if (n == 1 || n == 2) return 1;

// Transformation matrix

long[][] M = {

{0, 1, 0},

{0, 0, 1},

{1, 1, 1}

};

// Initial state vector [T0, T1, T2]

long[][] M0 = {

{0}, // T0

{1}, // T1

{1} // T2

};

// Compute M^n

long[][] Mn = matrixPower(M, n);

// Multiply M^n \* M0 to get final state

long[][] result = matrixMultiply(Mn, M0);

return (int) result[0][0];

}

private long[][] matrixPower(long[][] matrix, int power) {

long[][] result = {

{1, 0, 0},

{0, 1, 0},

{0, 0, 1}

}; // Identity matrix

long[][] temp = matrix.clone();

while (power > 0) {

if (power % 2 == 1) {

result = matrixMultiply(result, temp);

}

temp = matrixMultiply(temp, temp);

power /= 2;

}

return result;

}

private long[][] matrixMultiply(long[][] A, long[][] B) {

int aRows = A.length;

int aCols = A[0].length;

int bCols = B[0].length;

long[][] res = new long[aRows][bCols];

for (int i = 0; i < aRows; i++) {

for (int j = 0; j < bCols; j++) {

for (int k = 0; k < aCols; k++) {

res[i][j] += A[i][k] \* B[k][j];

}

}

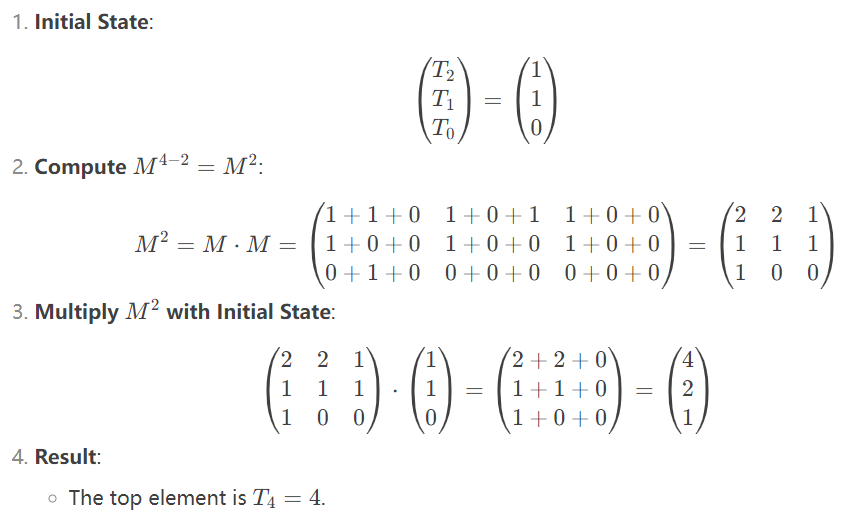
}

return res;

}

}

**Worked Example (n = 4)**



**Complexity Analysis**

|  |  |  |
| --- | --- | --- |
| Metric | Complexity | Explanation |
| **Time** | O(log n) | Due to exponentiation by squaring. |
| **Space** | O(1) | Fixed-size matrices (3x3) are used. |

**Why Use Matrix Exponentiation?**

**Efficiency**: For very large *n* (e.g., *n*=10^9), the O(log n) time is crucial.

**Generalization**: This method works for any linear recurrence (e.g., Fibonacci, Tetranacci).

**Key Takeaways**

The transformation matrix *M* encodes the recurrence relation.

Exponentiation by squaring reduces time from O(n) to O(log n).

The top-left element of *M^n*−2 gives *Tn*​.

This approach is optimal for large *n* but more complex to implement than DP for small inputs.

**Matrix multiply rules**

**a 3 \* 3 matrix multiply a 3 \* 1 matrix will result into what kind of matrix ?**

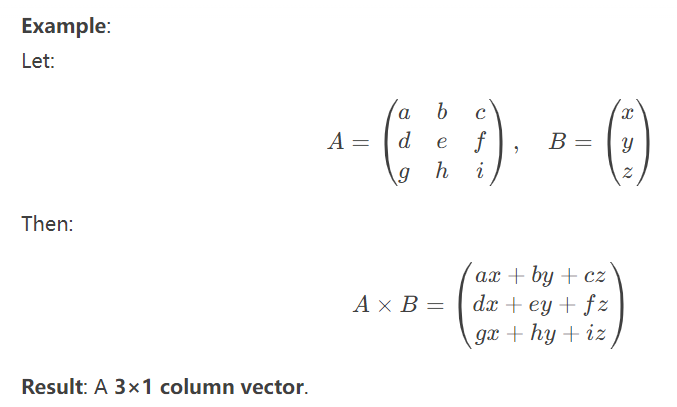
**a 1 \* 3 matrix multiply a 3 \* 3 matrix will result into what kind of matrix ?**

**1. 3×3 Matrix × 3×1 Matrix**

When you multiply a **3×3 matrix** by a **3×1 matrix**, the result is a **3×1 matrix** (a column vector).

This is because matrix multiplication follows the rule:

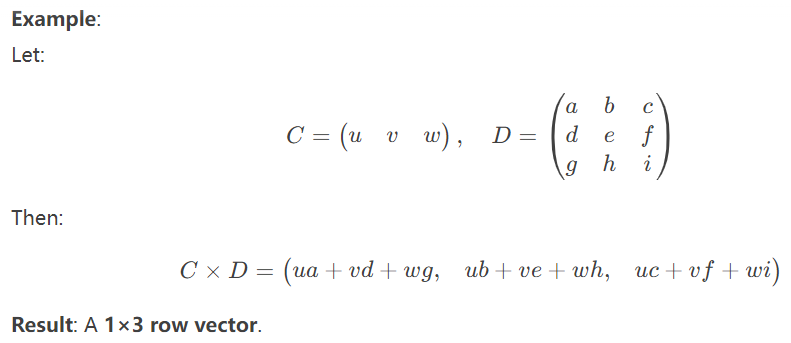
**(m×n) × (n×p) → (m×p)**



**2. 1×3 Matrix × 3×3 Matrix**

When you multiply a **1×3 matrix** by a **3×3 matrix**, the result is a **1×3 matrix** (a row vector).

Again, following **(m×n) × (n×p) → (m×p)**:



**Key Takeaways**

|  |  |
| --- | --- |
| Multiplication | Resulting Matrix |
| **3×3 × 3×1** (e.g., *M*×state vector) | **3×1** (column vector) |
| **1×3 × 3×3** (e.g., extracting a single value) | **1×3** (row vector) |

**Why This Matters in the Tribonacci Problem**

The **3×3 transformation matrix** *M* is multiplied by a **3×1 state vector** to compute the next state.

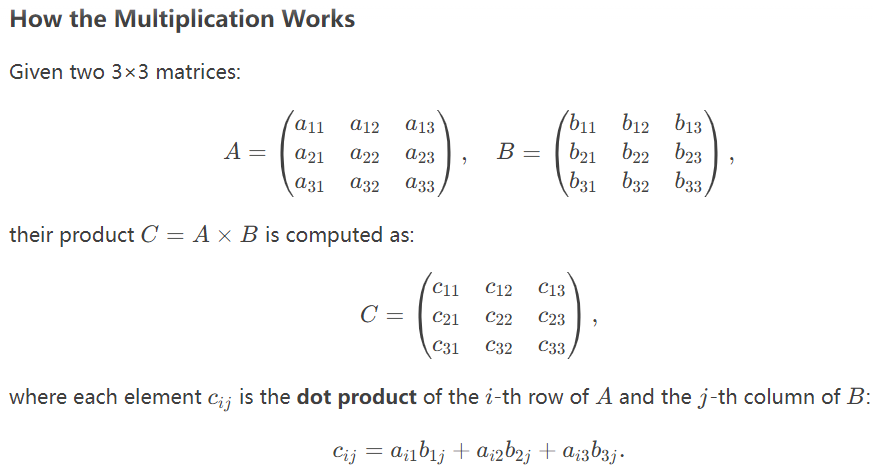
The **1×3 × 3×3** case is less common here, but it could be used to extract a specific value after exponentiation.

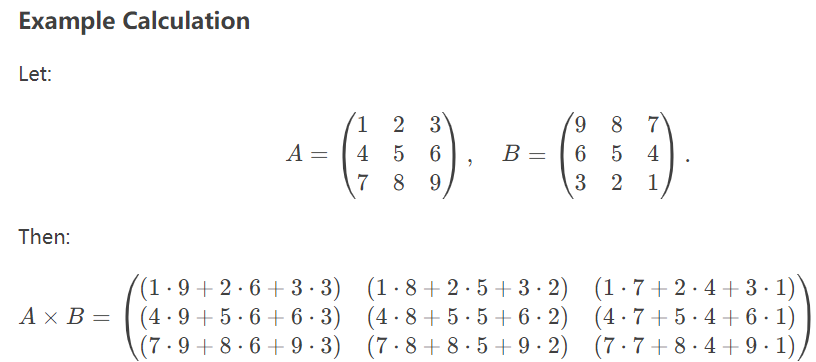
This structure ensures efficient computation of linear recurrences like the Tribonacci sequence.

**How about a 3 \* 3 matrix multiply a 3 \* 3 matrix will result into what kind of matrix ?**

When you multiply a **3×3 matrix** by another **3×3 matrix**, the result is always another **3×3 matrix**. This follows the general rule of matrix multiplication where:(m×n) matrix × (n×p) matrix → (m×p) matrix

Since both matrices are 3×3 (i.e., *n*=3), the result is also 3×3.





**Key Properties**

**Non-Commutative**: A×B≠B×A(**order matters**!).

**Associative**: (*A*×*B*)×*C*=*A*×(*B*×*C*).

**Identity Matrix**: **Multiplying by the identity matrix *I* leaves the matrix unchanged**:*A*×*I*=*I*×*A*=*A*.

**Why This Matters in the Tribonacci Problem**

In the **matrix exponentiation solution for Tribonacci numbers**, we repeatedly multiply the transformation matrix *M* by itself to compute *M^n*. Since *M* is 3×3, all intermediate results remain 3×3 matrices, enabling efficient computation in *O*(log*n*) time.

**Summary**

|  |  |
| --- | --- |
| Multiplication | Result |
| **3×3 × 3×3** | **3×3 matrix** |

This operation is foundational for advanced algorithms like **matrix exponentiation** used in dynamic programming and linear recurrence problems.

**In both solutions, i want to how how below statement works ? and why result initialized as identity matrix ?**

if (power % 2 == 1) {

result = matrixMultiply(result, temp);

}

Great question! This is part of the **matrix exponentiation by squaring** algorithm, which efficiently computes matrix^power in O(log n) time. Let me break it down:

**Why Initialize result as Identity Matrix?**

long[][] result = {

{1, 0, 0},

{0, 1, 0},

{0, 0, 1}

}; // Identity matrix

**Identity Matrix Property:**

**Multiplying any matrix by the identity matrix leaves it unchanged: A × I = A**

**This is the matrix equivalent of 1 in regular multiplication (e.g., 5 × 1 = 5)**

**Purpose in Exponentiation**:

We start with result = I to:

Accumulate the final result gradually

Handle the case where power = 0 (though our code doesn't encounter this for Tribonacci)

Serve as a neutral starting point for multiplication

**How the Exponentiation Works**

while (power > 0) {

if (power % 2 == 1) { // If power is odd

result = matrixMultiply(result, temp); // Multiply by current temp

}

temp = matrixMultiply(temp, temp); // Square the temp matrix

power /= 2; // Halve the power

}

**Step-by-Step Execution (Example: power = 5)**

|  |  |  |  |
| --- | --- | --- | --- |
| power | action | result | temp (matrix) |
| 5 | power is odd → multiply | I × M = M | M |
|  | square temp | M | M² |
| 2 | power is even → no multiply | M | M² → M⁴ (after squaring) |
| 1 | power is odd → multiply | M × M⁴ = M⁵ | M⁴ → M⁸ (wouldn't be used) |

**Odd Power Detection** (power % 2 == 1):

When the power is odd, we "peel off" one multiplication and accumulate it into result

This ensures we don't lose the "leftover" matrix multiplication

**Squaring** (temp = matrixMultiply(temp, temp)):

Doubles the exponent of temp each time (M → M² → M⁴ → M⁸...)

This gives us exponential growth in the exponent

**Halving** (power /= 2):

Equivalent to right-shifting bits (power >>= 1)

Reduces the problem size logarithmically

**Why This is Efficient**

**Normal Approach**: M⁵ = M × M × M × M × M → 4 multiplications

**Exponentiation by Squaring**:

M⁵ = M × (M²)² → 3 multiplications

For large n (e.g., n=100), reduces from 99 to ~7 multiplications!

**Key Insight**

The identity matrix initialization and odd-power handling work together to:

Preserve accumulated results

Handle both even and odd powers correctly

Maintain the invariant: final\_result = initial\_matrix^power

This is exactly how we efficiently compute large matrix powers for the Tribonacci sequence!

**Refer to**

[L70.Climbing Stairs](note://WEBce2fed61238e31617ae6d19bbb7f7a01)

[L509.Fibonacci Number (Ref.L70,L746,L842,L873)](note://WEB5e91be389fb43d760647ae414e6240e8)