Matrix Exponentiation - GeeksforGeeks

Source: https://www.geeksforgeeks.org/matrix-exponentiation/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Matrix Exponentiation Last Updated : 23 Jul, 2025 Matrix Exponentiation is a technique used to calculate a matrix raised to a power efficiently, that is in logN time. It is mostly used for solving problems related to linear recurrences. Idea behind Matrix Exponentiation: Similar to Binary Exponentiation which is used to calculate a number raised to a power, Matrix Exponentiation is used to calculate a matrix raised to a power efficiently. Let us understand Matrix Exponentiation with the help of an example: We can calculate matrix M^(N - 2) in logN time using Matrix Exponentiation . The idea is same as Binary Exponentiation: When we are calculating (M N), we can have 3 possible positive values of N: Case 1: If N = 0, whatever be the value of M, our result will be Identity Matrix I . Case 2: If N is an even number, then instead of calculating (M N), we can calculate ((M 2) N/2) and the result will be same. Case 3: If N is an odd number, then instead of calculating (M N), we can calculate (M * (M (N – 1)/2) 2). Use Cases of Matrix Exponentiation: Finding nth Fibonacci Number: The recurrence relation for Fibonacci Sequence is F(n) = F(n - 1) + F(n - 2) starting with F(0) = 0 and F(1) = 1. Below is the implementation of above idea: C++ // C++ Program to find the Nth fibonacci number using // Matrix Exponentiation #include <bits/stdc++.h> using namespace std ; int MOD = 1e9 + 7 ; // function to multiply two 2x2 Matrices void multiply (vector < vector < long long > >& A , vector < vector < long long > >& B) { // Matrix to store the result vector < vector < long long > > C (2 , vector < long long > (2)); // Matrix Multiply C [0][0] = (A [0][0] * B [0][0] + A [0][1] * B [1][0]) % MOD ; C [0][1] = (A [0][0] * B [0][1] + A [0][1] * B [1][1]) % MOD ; C [1][0] = (A [1][0] * B [0][0] + A [1][1] * B [1][0]) % MOD ; C [1][1] = (A [1][0] * B [0][1] + A [1][1] * B [1][1]) % MOD ; // Copy the result back to the first matrix A [0][0] = C [0][0]; A [0][1] = C [0][1]; A [1][0] = C [1][0]; A [1][1] = C [1][1]; } // Function to find (Matrix M ^ expo) vector < vector < long long > > power (vector < vector < long long > > M , int expo) { // Initialize result with identity matrix vector < vector < long long > > ans = { { 1 , 0 }, { 0 , 1 } }; // Fast Exponentiation while (expo) { if (expo & 1) multiply (ans , M); multiply (M , M); expo >>= 1 ; } return ans ; } // function to find the nth fibonacci number int nthFibonacci (int n) { // base case if (n == 0 || n == 1) return 1 ; vector < vector < long long > > M = { { 1 , 1 }, { 1 , 0 } }; // Matrix F = {{f(0), 0}, {f(1), 0}}, where f(0) and // f(1) are first two terms of fibonacci sequence vector < vector < long long > > F = { { 1 , 0 }, { 0 , 0 } }; // Multiply matrix M (n - 1) times vector < vector < long long > > res = power (M , n - 1); // Multiply Resultant with Matrix F multiply (res , F); return res [0][0] % MOD ; } int main () { // Sample Input int n = 3 ; // Print nth fibonacci number cout << nthFibonacci (n) << endl ; } Java // Java Program to find the Nth fibonacci number using // Matrix Exponentiation import java.util.* ; public class GFG { static final int MOD = 1000000007 ; // Function to multiply two 2x2 matrices public static void multiply (long [][] A , long [][] B) { // Matrix to store the result long [][] C = new long [2][2] ; // Matrix multiplication C [0][0] = (A [0][0] * B [0][0] + A [0][1] * B [1][0]) % MOD ; C [0][1] = (A [0][0] * B [0][1] + A [0][1] * B [1][1]) % MOD ; C [1][0] = (A [1][0] * B [0][0] + A [1][1] * B [1][0]) % MOD ; C [1][1] = (A [1][0] * B [0][1] + A [1][1] * B [1][1]) % MOD ; // Copy the result back to the first matrix for (int i = 0 ; i < 2 ; i ++) { for (int j = 0 ; j < 2 ; j ++) { A [i][j] = C [i][j] ; } } } // Function to find (Matrix M ^ expo) public static long [][] power (long [][] M , int expo) { // Initialize result with identity matrix long [][] ans = { { 1 , 0 }, { 0 , 1 } }; // Fast exponentiation while (expo > 0) { if ((expo & 1) != 0) { multiply (ans , M); } multiply (M , M); expo >>= 1 ; } return ans ; } // Function to find the nth Fibonacci number public static int nthFibonacci (int n) { // Base case if (n == 0 || n == 1) { return 1 ; } long [][] M = { { 1 , 1 }, { 1 , 0 } }; // F(0) = 1, F(1) = 1 long [][] F = { { 1 , 0 },

{ 0 , 0 } }; // Multiply matrix M (n - 1) times long [][] res = power (M , n - 1); // Multiply resultant with matrix F multiply (res , F); return (int)((res [0][0]) % MOD); } public static void main (String [] args) { // Sample input int n = 3 ; // Print nth Fibonacci number System . out . println (nthFibonacci (n)); } } Python # Python Program to find the Nth fibonacci number using # Matrix Exponentiation MOD = 10 ** 9 + 7 # function to multiply two 2x2 Matrices def multiply (A , B): # Matrix to store the result C = [[0 , 0], [0 , 0]] # Matrix Multiply C [0][0] = (A [0][0] * B [0][0] + A [0][1] * B [1][0]) % MOD C [0][1] = (A [0][0] * B [0][1] + A [0][1] * B [1][1]) % MOD C [1][0] = (A [1][0] * B [0][0] + A [1][1] * B [1][0]) % MOD C [1][1] = (A [1][0] * B [0][1] + A [1][1] * B [1][1]) % MOD # Copy the result back to the first matrix A [0][0] = C [0][0] A [0][1] = C [0][1] A [1][0] = C [1][0] A [1][1] = C [1][1] # Function to find (Matrix M ^ expo) def power (M , expo): # Initialize result with identity matrix ans = [[1 , 0], [0 , 1]] # Fast Exponentiation while expo : if expo & 1 : multiply (ans , M) multiply (M , M) expo >>= 1 return ans def nthFibonacci (n): # Base case if n == 0 or n == 1 : return 1 M = [[1 , 1], [1 , 0]] # F(0) = 0, F(1) = 1 F = [[1 , 0], [0 , 0]] # Multiply matrix M (n - 1) times res = power (M , n - 1) # Multiply Resultant with Matrix F multiply (res , F) return res [0][0] % MOD # Sample Input n = 3 # Print the nth fibonacci number print (nthFibonacci (n)) C# // C# Program to find the Nth fibonacci number using // Matrix Exponentiation using System ; using System.Collections.Generic ; public class GFG { static int MOD = 1000000007 ; // function to multiply two 2x2 Matrices public static void Multiply (long [][] A , long [][] B) { // Matrix to store the result long [][] C = new long [2][] { new long [2], new long [2] }; // Matrix Multiply C [0][0] = (A [0][0] * B [0][0] + A [0][1] * B [1][0]) % MOD ; C [0][1] = (A [0][0] * B [0][1] + A [0][1] * B [1][1]) % MOD ; C [1][0] = (A [1][0] * B [0][0] + A [1][1] * B [1][0]) % MOD ; C [1][1] = (A [1][0] * B [0][1] + A [1][1] * B [1][1]) % MOD ; // Copy the result back to the first matrix A [0][0] = C [0][0]; A [0][1] = C [0][1]; A [1][0] = C [1][0]; A [1][1] = C [1][1]; } // Function to find (Matrix M ^ expo) public static long [][] Power (long [][] M , int expo) { // Initialize result with identity matrix long [][] ans = new long [2][] { new long [] { 1 , 0 }, new long [] { 0 , 1 } }; // Fast Exponentiation while (expo > 0) { if ((expo & 1) > 0) Multiply (ans , M); Multiply (M , M); expo >>= 1 ; } return ans ; } // function to find the nth fibonacci number public static int NthFibonacci (int n) { // base case if (n == 0 || n == 1) return 1 ; long [][] M = new long [2][] { new long [] { 1 , 1 }, new long [] { 1 , 0 } }; // F(0) = 0, F(1) = 1 long [][] F = new long [2][] { new long [] { 1 , 0 }, new long [] { 0 , 0 } }; // Multiply matrix M (n - 1) times long [][] res = Power (M , n - 1); // Multiply Resultant with Matrix F Multiply (res , F); return (int)((res [0][0] % MOD)); } public static void Main (string [] args) { // Sample Input int n = 3 ; // Print nth fibonacci number Console . WriteLine (NthFibonacci (n)); } } JavaScript const MOD = 1e9 + 7 ; // Function to multiply two 2x2 matrices function multiply (A , B) { // Matrix to store the result const C = [[0 , 0], [0 , 0]]; // Matrix Multiply C [0][0] = (A [0][0] * B [0][0] + A [0][1] * B [1][0]) % MOD ; C [0][1] = (A [0][0] * B [0][1] + A [0][1] * B [1][1]) % MOD ; C [1][0] = (A [1][0] * B [0][0] + A [1][1] * B [1][0]) % MOD ; C [1][1] = (A [1][0] * B [0][1] + A [1][1] * B [1][1]) % MOD ; // Copy the result back to the first matrix A [0][0] = C [0][0]; A [0][1] = C [0][1]; A [1][0] = C [1][0]; A [1][1] = C [1][1]; } // Function to find (Matrix M ^ expo) function power (M , expo) { // Initialize result with identity matrix const ans = [[1 , 0], [0 , 1]]; // Fast Exponentiation while (expo) { if (expo & 1) multiply (ans , M); multiply (M , M); expo >>= 1 ; } return ans ; } // Function to find the nth fibonacci number function nthFibonacci (n) { // Base case if (n === 0 || n === 1) return 1 ; const M = [[1 , 1], [1 , 0]]; // F(0) = 0, F(1) = 1 const F = [[1 , 0], [0 , 0]]; // Multiply matrix M (n - 1) times const res = power (M , n - 1); // Multiply Resultant with Matrix F multiply (res , F); return res [0][0] % MOD ; } // Sample Input const n = 3 ; // Print nth fibonacci number console . log (nthFibonacci (n)); Output 2 Time Complexity: O(logN), because fast exponentiation takes O(logN) time. Auxiliary Space: O(1) Finding nth Tribonacci Number: The recurrence relation for Tribonacci Sequence is T(n) = T(n - 1) + T(n - 2) + T(n - 3) starting with T(0) = 0, T(1) = 1 and T(2) = 1. Below is the implementation of above idea: C++ // C++ Program to find the nth tribonacci number #include <bits/stdc++.h> using namespace std ; // Function to multiply two 3x3 matrices void multiply (vector < vector < long long > >& A , vector < vector < long long > >& B) { // Matrix to store the result vector < vector < long long > > C (3 , vector < long long > (3)); for (int i = 0 ; i < 3 ; i ++) { for (int j = 0 ; j < 3 ; j ++) { for (int k = 0 ; k < 3 ; k ++) { C [i][j] = (C [i][j] + ((A [i][k]) * (B [k][j]))); } } } // Copy the result back to the first matrix for (int i = 0 ; i < 3 ; i ++) { for (int j = 0 ; j < 3 ; j ++) { A [i][j] = C [i][j]; } } } // Function to calculate (Matrix M) ^ expo vector < vector < long long > > power (vector < vector < long long > > M , int expo) { // Initialize result with identity matrix vector < vector < long long > > ans = { { 1 , 0 , 0 }, { 0 , 1 , 0 }, { 0 , 0 , 1 } }; // Fast Exponentiation while (expo) { if (expo & 1) multiply (ans , M); multiply (M , M); expo >>= 1 ; } return ans ; } // function to return the Nth tribonacci number long long tribonacci (int n) { // base

```
condition if ( n == 0 || n == 1 ) return n ; // Matrix M to generate the next tribonacci number vector <
vector < long long > > M = { { 1 , 1 , 1 }, { 1 , 0 , 0 }, { 0 , 1 , 0 } }; // Since first 3 number of tribonacci
series are: // trib(0) = 0 // trib(1) = 1 // trib(2) = 1 // F = {{trib(2), 0, 0}, {trib(1), 0, 0}, {trib(0), 0, // 0}} vector
< vector < long long > > F = { { 1 , 0 , 0 }, { 1 , 0 , 0 }, { 0 , 0 , 0 } }; vector < vector < long long > > res =
power ( M , n - 2 ); multiply ( res , F ); return res [ 0 ][ 0 ]; } int main () { // Sample Input int n = 4 ; //
Function call cout << tribonacci ( n ); return 0 ; } Java // Java program to find nth tribonacci number
import java.util.* ; public class Main { // Function to multiply two 3x3 matrices static void multiply ( long
[][] A , long [][] B ) { // Matrix to store the result long [][] C = new long [ 3 ][ 3 ] ; for ( int i = 0 ; i < 3 ; i ++ ) {
for ( int j = 0 ; j < 3 ; j ++ ) { for ( int k = 0 ; k < 3 ; k ++ ) { C [ i ][ j ] += ( A [ i ][ k ] * B [ k ][ j ] ); } } } // Copy
the result back to the first matrix for ( int i = 0 ; i < 3 ; i ++ ) { for ( int j = 0 ; j < 3 ; j ++ ) { A [ i ][ j ] = C [ i ][
j ] ; } } } // Function to calculate (Matrix M) ^ expo static long [][] power ( long [][] M , int expo ) { //
Initialize result with identity matrix long [][] ans = { { 1 , 0 , 0 }, { 0 , 1 , 0 }, { 0 , 0 , 1 } }; // Fast
Exponentiation while ( expo > 0 ) { if (( expo & 1 ) == 1 ) multiply ( ans , M ); multiply ( M , M ); expo >>=
1 ; } return ans ; } // function to return the Nth tribonacci number static long tribonacci ( int n ) { // base
condition if ( n == 0 || n == 1 ) return n ; // Matrix M to generate the next tribonacci number long [][] M = {
{ 1 , 1 , 1 }, { 1 , 0 , 0 }, { 0 , 1 , 0 } }; // Since first 3 number of tribonacci series are: // trib(0) = 0 // trib(1)
= 1 // trib(2) = 1 // F = {{trib(2), 0, 0}, {trib(1), 0, 0}, {trib(0), // 0, 0}} long [][] F = { { 1 , 0 , 0 }, { 1 , 0 , 0 }, {
0 , 0 , 0 } }; long [][] res = power ( M , n - 2 ); multiply ( res , F ); return res [ 0 ][ 0 ] ; } public static void
main ( String [] args ) { // Sample Input int n = 4 ; // Function call System . out . println ( tribonacci ( n )); }
} Python # Python3 program to find nth tribonacci number # Function to multiply two 3x3 matrices def
multiply ( A , B ): # Matrix to store the result C = [[ 0 ] * 3 for _ in range ( 3 )] for i in range ( 3 ): for j in
range ( 3 ): for k in range ( 3 ): C [ i ][ j ] += A [ i ][ k ] * B [ k ][ j ] # Copy the result back to the first matrix
A for i in range ( 3 ): for j in range ( 3 ): A [ i ][ j ] = C [ i ][ j ] # Function to calculate (Matrix M) ^ expo def
power ( M , expo ): # Initialize result with identity matrix ans = [[ 1 , 0 , 0 ], [ 0 , 1 , 0 ], [ 0 , 0 , 1 ]] # Fast
Exponentiation while expo > 0 : if expo & 1 : multiply ( ans , M ) multiply ( M , M ) expo >>= 1 return ans
# Function to return the Nth tribonacci number def tribonacci ( n ): # Base condition if n == 0 or n == 1 :
return n # Matrix M to generate the next tribonacci number M = [[ 1 , 1 , 1 ], [ 1 , 0 , 0 ], [ 0 , 1 , 0 ]] #
Since first 3 numbers of tribonacci series are: # trib(0) = 0 # trib(1) = 1 # trib(2) = 1 # F = [[trib(2), 0, 0],
[trib(1), 0, 0], [trib(0), 0, 0]] F = [[ 1 , 0 , 0 ], [ 1 , 0 , 0 ], [ 0 , 0 , 0 ]] res = power ( M , n - 2 ) multiply ( res ,
F ) return res [ 0 ][ 0 ] # Sample Input n = 4 # Function call print ( tribonacci ( n )) C# // C# program to
find nth tribonacci number using System ; public class MainClass { // Function to multiply two 3x3
matrices static void Multiply ( long [][] A , long [][] B ) { // Matrix to store the result long [][] C = new long [
3 ][]; for ( int i = 0 ; i < 3 ; i ++ ) { C [ i ] = new long [ 3 ]; for ( int j = 0 ; j < 3 ; j ++ ) { for ( int k = 0 ; k < 3 ; k
++ ) { C [ i ][ j ] += A [ i ][ k ] * B [ k ][ j ]; } } } // Copy the result back to the first matrix A for ( int i = 0 ; i < 3
; i ++ ) { for ( int j = 0 ; j < 3 ; j ++ ) { A [ i ][ j ] = C [ i ][ j ]; } } } // Function to calculate (Matrix M) ^ expo
static long [][] Power ( long [][] M , int expo ) { // Initialize result with identity matrix long [][] ans = new
long [ 3 ][]; for ( int i = 0 ; i < 3 ; i ++ ) { ans [ i ] = new long [ 3 ]; ans [ i ][ i ] = 1 ; // Diagonal elements are
1 } // Fast Exponentiation while ( expo > 0 ) { if (( expo & 1 ) == 1 ) Multiply ( ans , M ); Multiply ( M , M );
expo >>= 1 ; } return ans ; } // Function to return the Nth tribonacci number static long Tribonacci ( int n )
{ // Base condition if ( n == 0 || n == 1 ) return n ; // Matrix M to generate the next tribonacci number long
[][] M = new long [][] { new long [] { 1 , 1 , 1 }, new long [] { 1 , 0 , 0 }, new long [] { 0 , 1 , 0 } }; // Since
first 3 numbers of tribonacci series are: // trib(0) = 0 // trib(1) = 1 // trib(2) = 1 // F = [[trib(2), 0, 0], [trib(1),
0, 0], [trib(0), // 0, 0]] long [][] F = new long [][] { new long [] { 1 , 0 , 0 }, new long [] { 1 , 0 , 0 }, new long
[] { 0 , 0 , 0 } }; long [][] res = Power ( M , n - 2 ); Multiply ( res , F ); return res [ 0 ][ 0 ]; } public static void
Main ( string [] args ) { // Sample Input int n = 4 ; // Function call Console . WriteLine ( Tribonacci ( n )); }
} JavaScript // JavaScript Program to find nth tribonacci number // Function to multiply two 3x3 matrices
function multiply ( A , B ) { // Matrix to store the result let C = [ [ 0 , 0 , 0 ], [ 0 , 0 , 0 ], [ 0 , 0 , 0 ] ]; for ( let
i = 0 ; i < 3 ; i ++ ) { for ( let j = 0 ; j < 3 ; j ++ ) { for ( let k = 0 ; k < 3 ; k ++ ) { C [ i ][ j ] += A [ i ][ k ] * B [ k
][ j ]; } } } // Copy the result back to the first matrix A for ( let i = 0 ; i < 3 ; i ++ ) { for ( let j = 0 ; j < 3 ; j ++ )
{ A [ i ][ j ] = C [ i ][ j ]; } } } // Function to calculate (Matrix M) ^ expo function power ( M , expo ) { //
Initialize result with identity matrix let ans = [ [ 1 , 0 , 0 ], [ 0 , 1 , 0 ], [ 0 , 0 , 1 ] ]; // Fast Exponentiation
while ( expo > 0 ) { if ( expo & 1 ) multiply ( ans , M ); multiply ( M , M ); expo >>= 1 ; } return ans ; } //
Function to return the Nth tribonacci number function tribonacci ( n ) { // base condition if ( n === 0 || n
=== 1 ) return n ; // Matrix M to generate the next tribonacci number let M = [ [ 1 , 1 , 1 ], [ 1 , 0 , 0 ], [ 0 ,
1 , 0 ] ]; // Since first 3 numbers of tribonacci series are: // trib(0) = 0 // trib(1) = 1 // trib(2) = 1 // F =
[[trib(2), 0, 0], [trib(1), 0, 0], [trib(0), 0, 0]] let F = [ [ 1 , 0 , 0 ], [ 1 , 0 , 0 ], [ 0 , 0 , 0 ] ]; let res = power ( M
, n - 2 ); multiply ( res , F ); return res [ 0 ][ 0 ]; } // Main function function main () { // Sample Input let n =
```

4 ; // Function call console . log (tribonacci (n)); } // Call the main function main (); Output 4 Time Complexity: O(logN), because fast exponentiation takes O(logN) time. Auxiliary Space: O(1) Applications of Matrix Exponentiation: Matrix Exponentiation has a variety of applications. Some of them are: Any linear recurrence relation, such as the Fibonacci Sequence , Tribonacci Sequence or linear homogeneous recurrence relations with constant coefficients, can be solved using matrix exponentiation. The RSA encryption algorithm involves exponentiation of large numbers, which can be efficiently handled using matrix exponentiation techniques. Dynamic programming problems, especially those involving linear recurrence relations, can be optimized using matrix exponentiation to reduce time complexity. Matrix exponentiation is used in number theory problems involving modular arithmetic , such as finding large powers of numbers modulo some value efficiently. Advantages of Matrix Exponentiation: Advantages of using Matrix Exponentiation are: Matrix Exponentiation helps in finding Nth term of linear recurrence relations like Fibonacci or Tribonacci Series in log(N) time. This makes it much faster for large values of N. It requires O(1) space if we use the iterative approach as it requires constant amount of extra space . Large numbers can be handled without integer overflow using modulo operations. Disadvantages of Matrix Exponentiation: Disadvantages of using Matrix Exponentiation are: Matrix Exponentiation is more complex than other iterative or recursive methods. Hence, it is harder to debug . Initial conditions should be handled carefully to avoid incorrect results. Comment Article Tags: Article Tags: DSA Fibonacci series Modular Arithmetic matrix-exponentiation + 1 More