Matrix Multiplication - GeeksforGeeks

Source: https://www.geeksforgeeks.org/strassens-matrix-multiplication/

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 Multiplication Last Updated : 29 Aug, 2025 Given two matrices mat1[][] of size n × m and mat2[][] of size m × q , find their matrix product, where the resulting matrix has dimensions n × q Examples: Input: mat1[][] = [[1, 2, 3], [4, 5, 6]], mat2[][] = [[7, 8], [9, 10], [11, 12]] Output: [[58, 64], [139, 154]] Explanation: Matrix mat1[ ][ ] (2×3) is multiplied with Matrix mat2[ ][ ] (3×2) by computing the dot product of a's rows with b's columns. This results in a new matrix of size 2×2 containing the summed products for each position. Input: mat1[][] = [[17, 4], [17, 16]], mat2[][] = [9, 2], [7, 1]] Output: [[181, 38], [265, 50]] Explanation: Matrix mat1[ ][ ] (2×2) is multiplied with Matrix mat2[ ][ ] (2×2) by computing the dot product of a's rows with b's columns. This results in a new matrix of size 2×2 containing the summed products for each position. Try it on GfG Practice Table of Content [Approach 1] - Using Nested Loops - O(n * m * q) Time and O(n * q) Space [Approach 2] - Using Divide and Conquer - O(n * m * q) Time and O(n * q) Space [Approach 3] - Using Strassen's Method [Approach 1] - Using Nested Loops - O(n * m * q) Time and O(n * q) Space The main idea is to initializes the result matrix with zeros and fills each cell by computing the dot product of the corresponding row from a and column from . This is done using three nested loops: the outer two iterate over the result matrix cells, and the innermost loop performs the multiplication and addition. C++ #include <iostream> #include <vector> using namespace std ; vector < vector < int >> multiply (vector < vector < int >> & mat1 , vector < vector < int >> & mat2) { int n = mat1 . size (), m = mat1 [0]. size (), q = mat2 [0]. size (); // Initialize the result matrix with // dimensions n×q, filled with 0s vector < vector < int >> res (n , vector < int > (q , 0)); // Loop through each row of mat1 for (int i = 0 ; i < n ; i ++) { // Loop through each column of mat2 for (int j = 0 ; j < q ; j ++) { // Compute the dot product of // row mat1[i] and column mat2[][j] for (int k = 0 ; k < m ; k ++) { res [i][j] += mat1 [i][k] * mat2 [k][j]; } } } return res ; } int main () { vector < vector < int >> mat1 = { { 1 , 2 , 3 }, { 4 , 5 , 6 } }; vector < vector < int >> mat2 = { { 7 , 8 }, { 9 , 10 }, { 11 , 12 } }; vector < vector < int >> res = multiply (mat1 , mat2); // Print the resulting matrix for (int i = 0 ; i < res . size (); i ++) { for (int j = 0 ; j < res [i]. size (); j ++) { cout << res [i][j] << " " ; } cout << endl ; } return 0 ; } Java import java.util.ArrayList ; import java.util.Collections ; class GfG { static ArrayList < ArrayList < Integer >> multiply (int [][] mat1 , int [][] mat2) { int n = mat1 . length ; int m = mat1 [0] . length ; int q = mat2 [0] . length ; // Initialize the result matrix with // dimensions n×q, filled with 0s ArrayList < ArrayList < Integer >> res = new ArrayList <> (); for (int i = 0 ; i < n ; i ++) { ArrayList < Integer > row = new ArrayList <> (Collections . nCopies (q , 0)); res . add (row); } // Loop through each row of mat1 for (int i = 0 ; i < n ; i ++) { // Loop through each column of mat2 for (int j = 0 ; j < q ; j ++) { // Compute the dot product of // row mat1[i] and column mat2[][j] for (int k = 0 ; k < m ; k ++) { int val = res . get (i). get (j) + mat1 [i][k] * mat2 [k][j] ; res . get (i). set (j , val); } } } return res ; } public static void main (String [] args) { int [][] mat1 = { { 1 , 2 , 3 }, { 4 , 5 , 6 } }; int [][] mat2 = { { 7 , 8 }, { 9 , 10 }, { 11 , 12 } }; ArrayList < ArrayList < Integer >> res = multiply (mat1 , mat2); // Print the resulting matrix for (ArrayList < Integer > row : res) { for (int val : row) { System . out . print (val + " "); } System . out . println (); } } } Python def multiply (mat1 , mat2): n = len (mat1) m = len (mat1 [0]) q = len (mat2 [0]) # Initialize the result matrix with # dimensions n×q, filled with 0s res = [[0 for _ in range (q)] for _ in range (n)] # Loop through each row of mat1 for i in range (n): # Loop through each column of mat2 for j in range (q): # Compute the dot product of # row mat1[i] and column mat2[][j] for k in range (m): res [i][j] += mat1 [i][k] * mat2 [k][j] return res if __name__ == "__main__" : mat1 = [[1 , 2 , 3],[4 , 5 , 6]] mat2 = [[7 , 8],[9 , 10],[11 , 12]] res = multiply (mat1 , mat2) for row in res : for val in row : print (val , end = " ")

print () C# using System ; using System.Collections.Generic ; class GfG { static List < List < int >> multiply (int [,] mat1 , int [,] mat2) { // rows in mat1 int n = mat1 . GetLength (0); // cols in mat1 = rows in mat2 int m = mat1 . GetLength (1); // cols in mat2 int q = mat2 . GetLength (1); // Initialize the result matrix with dimensions n×q, filled with 0s List < List < int >> res = new List < List < int >> (); for (int i = 0 ; i < n ; i ++) { List < int > row = new List < int > (new int [q]); res . Add (row); } // Matrix multiplication for (int i = 0 ; i < n ; i ++) { for (int j = 0 ; j < q ; j ++) { for (int k = 0 ; k < m ; k ++) { res [i][j] += mat1 [i , k] * mat2 [k , j]; } } } return res ; } static void Main () { int [,] mat1 = {{ 1 , 2 , 3 },{ 4 , 5 , 6 }}; int [,] mat2 = {{ 7 , 8 },{ 9 , 10 },{ 11 , 12 }}; List < List < int >> res = multiply (mat1 , mat2); foreach (var row in res) { foreach (var val in row) { Console . Write (val + " "); } Console . WriteLine (); } } } JavaScript function multiply (mat1 , mat2) { let n = mat1 . length ; let m = mat1 [0]. length ; let q = mat2 [0]. length ; // Initialize the result matrix with // dimensions n×q, filled with 0s let res = Array . from ({ length : n }, () => Array (q). fill (0)); // Loop through each row of mat1 for (let i = 0 ; i < n ; i ++) { // Loop through each column of mat2 for (let j = 0 ; j < q ; j ++) { // Compute the dot product of // row mat1[i] and column mat2[][j] for (let k = 0 ; k < m ; k ++) { res [i][j] += mat1 [i][k] * mat2 [k][j]; } } } return res ; } // Driver code let mat1 = [[1 , 2 , 3],[4 , 5 , 6]]; let mat2 = [[7 , 8],[9 , 10],[11 , 12]]; let res = multiply (mat1 , mat2); for (let i = 0 ; i < res . length ; i ++) { console . log (res [i]. join (" ")); } Output 58 64 139 154 [Approach 2] - Using Divide and Conquer - O(n * m * q) Time and O(n * q) Space The main idea is to multiply two matrices by following the standard row-by-column multiplication method. For each element in the result matrix, it takes a row from the first matrix and a column from the second matrix, multiplies their corresponding elements, and adds them up to get a single value. This process is repeated for every position in the result matrix. C++ #include <iostream> #include <vector> using namespace std ; // Function to add two matrices of same dimensions r×c vector < vector < int >> add (vector < vector < int >> & mat1 , vector < vector < int >> & mat2) { int r = mat1 . size (); int c = mat1 [0]. size (); // Initialize result matrix with dimensions r×c vector < vector < int >> res (r , vector < int > (c , 0)); // Perform element-wise addition for (int i = 0 ; i < r ; ++ i) { for (int j = 0 ; j < c ; ++ j) { res [i][j] = mat1 [i][j] + mat2 [i][j]; } } return res ; } // Function to multiply mat1 (n×m) // with mat2 (m×q) vector < vector < int >> multiply (vector < vector < int >> & mat1 , vector < vector < int >> & mat2) { int n = mat1 . size (); int m = mat1 [0]. size (); int q = mat2 [0]. size (); // Initialize result matrix with dimensions n×q vector < vector < int >> res (n , vector < int > (q , 0)); // Matrix multiplication logic for (int i = 0 ; i < n ; ++ i) { for (int j = 0 ; j < q ; ++ j) { for (int k = 0 ; k < m ; ++ k) { res [i][j] += mat1 [i][k] * mat2 [k][j]; } } } return res ; } int main () { vector < vector < int >> mat1 = { { 1 , 2 , 3 }, { 4 , 5 , 6 } }; vector < vector < int >> mat2 = { { 7 , 8 }, { 9 , 10 }, { 11 , 12 } }; vector < vector < int >> res = multiply (mat1 , mat2); for (auto & row : res) { for (int val : row) { cout << val << " " ; } cout << endl ; } return 0 ; } Java import java.util.ArrayList ; import java.util.Collections ; class GfG { // Function to add two matrices of same dimensions r×c public static ArrayList < ArrayList < Integer >> add (int [][] mat1 , int [][] mat2) { int r = mat1 . length ; int c = mat1 [0] . length ; ArrayList < ArrayList < Integer >> res = new ArrayList <> (); for (int i = 0 ; i < r ; i ++) { ArrayList < Integer > row = new ArrayList <> (Collections . nCopies (c , 0)); res . add (row); } for (int i = 0 ; i < r ; i ++) { for (int j = 0 ; j < c ; j ++) { res . get (i). set (j , mat1 [i][j] + mat2 [i][j]); } } return res ; } // Function to multiply mat1 (n×m) with mat2 (m×q) public static ArrayList < ArrayList < Integer >> multiply (int [][] mat1 , int [][] mat2) { int n = mat1 . length ; int m = mat1 [0] . length ; int q = mat2 [0] . length ; ArrayList < ArrayList < Integer >> res = new ArrayList <> (); for (int i = 0 ; i < n ; i ++) { ArrayList < Integer > row = new ArrayList <> (Collections . nCopies (q , 0)); res . add (row); } for (int i = 0 ; i < n ; i ++) { for (int j = 0 ; j < q ; j ++) { for (int k = 0 ; k < m ; k ++) { int val = res . get (i). get (j) + mat1 [i][k] * mat2 [k][j] ; res . get (i). set (j , val); } } } return res ; } public static void main (String [] args) { int [][] mat1 = { { 1 , 2 , 3 }, { 4 , 5 , 6 } }; int [][] mat2 = { { 7 , 8 }, { 9 , 10 }, { 11 , 12 } }; ArrayList < ArrayList < Integer >> res = multiply (mat1 , mat2); for (ArrayList < Integer > row : res) { for (int val : row) { System . out . print (val + " "); } System . out . println (); } } } Python # Function to add two matrices of same dimensions r×c def add (mat1 , mat2): r = len (mat1) c = len (mat1 [0]) # Initialize result matrix with dimensions r×c res = [[0] * c for _ in range (r)] # Perform element-wise addition for i in range (r): for j in range (c): res [i][j] = mat1 [i][j] + mat2 [i][j] return res # Function to multiply mat1 (n×m) with mat2 (m×q) def multiply (mat1 , mat2): n = len (mat1) m = len (mat1 [0]) q = len (mat2 [0]) # Initialize result matrix with dimensions n×q res = [[0] * q for _ in range (n)] # Matrix multiplication logic for i in range (n): for j in range (q): for k in range (m): res [i][j] += mat1 [i][k] * mat2 [k][j] return res if __name__ == "__main__" : mat1 = [[1 , 2 , 3], [4 , 5 , 6]] mat2 = [[7 , 8], [9 , 10], [11 , 12]] res = multiply (mat1 , mat2) for row in res : print (" " . join (map (str , row))) C# using System ; using System.Collections.Generic ; class GfG { // Function to add two matrices and return result as List<List<int>> static List < List < int >> Add (int [,] mat1 , int [,] mat2) { int r = mat1 . GetLength (0); int c = mat1 . GetLength (1); List < List < int >> res = new List < List < int >> (); for (int i = 0 ; i < r ; i ++) { List < int > row = new List < int > (); for (int j = 0 ; j < c ; j ++) { row . Add (mat1 [i , j] + mat2 [i , j]); } res . Add (row); } return res ; } // Function to multiply mat1 (n×m) with mat2 (m×q) // and return result as List<List<int>> static List < List < int >> multiply (int [,] mat1 , int [,] mat2) { int n = mat1 . GetLength (0); int m = mat1 . GetLength (1); int q = mat2 . GetLength (1); List < List < int >> res = new List < List < int >> (); for (int i = 0 ; i < n ; i ++) { List < int > row = new List < int > (); for (int j = 0 ; j < q ; j ++) { int sum = 0 ; for (int k = 0 ; k < m ; k ++) { sum += mat1 [i , k] * mat2 [k , j]; } row . Add (sum); } res . Add (row); } return res ; } static void Main () { int [,] mat1 = { { 1 , 2 , 3 }, { 4 , 5 , 6 } }; int [,] mat2 = { { 7 , 8 }, { 9 , 10 }, { 11 , 12 } }; var res = multiply (mat1 , mat2); foreach (var row in res) { foreach (var val in row) { Console . Write (val + " "); } Console . WriteLine (); } } } JavaScript // Function to add two matrices of same dimensions r×c function add (mat1 , mat2) { let r = mat1 . length ; let c = mat1 [0]. length ; // Initialize result matrix with dimensions r×c let res = Array . from ({ length : r }, () => Array (c). fill (0)); // Perform element-wise addition for (let i = 0 ; i < r ; i ++) { for (let j = 0 ; j < c ; j ++) { res [i][j] = mat1 [i][j] + mat2 [i][j]; } } return res ; } // Function to multiply mat1 (n×m) // with mat2 (m×q) function multiply (mat1 , mat2) { let n = mat1 . length ; let m = mat1 [0]. length ; let q = mat2 [0]. length ; // Initialize result matrix with dimensions n×q let res = Array . from ({ length : n }, () => Array (q). fill (0)); // Matrix multiplication logic for (let i = 0 ; i < n ; i ++) { for (let j = 0 ; j < q ; j ++) { for (let k = 0 ; k < m ; k ++) { res [i][j] += mat1 [i][k] * mat2 [k][j]; } } } return res ; } // Driver Code let mat1 = [[1 , 2 , 3], [4 , 5 , 6]]; let mat2 = [[7 , 8], [9 , 10], [11 , 12]]; let res = multiply (mat1 , mat2); for (let row of res) { console . log (row . join (" ")); } Output 58 64 139 154 [Approach 3] - Using Strassen's Method Strassen’s algorithm originally applies to square matrices, but when adapted for multiplying an n*m matrix with an m*q matrix, the matrices are padded to form square matrices of size s*s where s = next power of two ≥ max(n, m, q). The algorithm then performs 7 recursive multiplications on these square blocks. Note: Strassen's Method is often not preferred in practice due to its high constant factors, making the naive method faster for typical applications. It performs poorly with sparse matrices, where specialized algorithms work better. The recursive submatrix divisions require extra memory. Additionally, due to limited precision in floating-point arithmetic, Strassen’s algorithm tends to accumulate more errors than the naive approach. C++ #include <iostream> #include <vector> #include <cmath> using namespace std ; // Return the next power of two greater than or equal to n int nextPowerOfTwo (int n) { return pow (2 , ceil (log2 (n))); } // Resize a matrix to newR x newC and // fill extra space with zeros vector < vector < int >> resizeMatrix (vector < vector < int >> & mat , int newR , int newC) { vector < vector < int >> resized (newR , vector < int > (newC , 0)); for (int i = 0 ; i < mat . size (); ++ i) for (int j = 0 ; j < mat [0]. size (); ++ j) resized [i][j] = mat [i][j]; return resized ; } // Perform matrix addition or subtraction // of size×size matrices // sign = 1 for addition, -1 for subtraction vector < vector < int >> add (vector < vector < int >> a , vector < vector < int >> b , int size , int sign = 1) { vector < vector < int >> res (size , vector < int > (size)); for (int i = 0 ; i < size ; i ++) for (int j = 0 ; j < size ; j ++) res [i][j] = a [i][j] + sign * b [i][j]; return res ; } // Recursive implementation of Strassen's // matrix multiplication // Assumes both matrices are size×size // and size is a power of 2 vector < vector < int >> strassen (vector < vector < int >> mat1 , vector < vector < int >> mat2) { int n = mat1 . size (); // Base case: 1×1 matrix multiplication vector < vector < int >> res (n , vector < int > (n , 0)); if (n == 1) { res [0][0] = mat1 [0][0] * mat2 [0][0]; return res ; } // Split matrices into 4 submatrices int newSize = n / 2 ; vector < vector < int >> a11 (newSize , vector < int > (newSize)); vector < vector < int >> a12 (newSize , vector < int > (newSize)); vector < vector < int >> a21 (newSize , vector < int > (newSize)); vector < vector < int >> a22 (newSize , vector < int > (newSize)); vector < vector < int >> b11 (newSize , vector < int > (newSize)); vector < vector < int >> b12 (newSize , vector < int > (newSize)); vector < vector < int >> b21 (newSize , vector < int > (newSize)); vector < vector < int >> b22 (newSize , vector < int > (newSize)); // Fill the submatrices for (int i = 0 ; i < newSize ; i ++) for (int j = 0 ; j < newSize ; j ++) { a11 [i][j] = mat1 [i][j]; a12 [i][j] = mat1 [i][j + newSize]; a21 [i][j] = mat1 [i + newSize][j]; a22 [i][j] = mat1 [i + newSize][j + newSize]; b11 [i][j] = mat2 [i][j]; b12 [i][j] = mat2 [i][j + newSize]; b21 [i][j] = mat2 [i + newSize][j]; b22 [i][j] = mat2 [i + newSize][j + newSize]; } // Compute the 7 products using Strassen’s formulas auto m1 = strassen (add (a11 , a22 , newSize), add (b11 , b22 , newSize)); auto m2 = strassen (add (a21 , a22 , newSize), b11); auto m3 = strassen (a11 , add (b12 , b22 , newSize , -1)); auto m4 = strassen (a22 , add (b21 , b11 , newSize , -1)); auto m5 = strassen (add (a11 , a12 , newSize), b22); auto m6 = strassen (add (a21 , a11 , newSize , -1), add (b11 , b12 , newSize)); auto m7 = strassen (add (a12 , a22 , newSize , -1), add (b21 , b22 , newSize)); // Calculate result quadrants from m1...m7 auto c11 = add (add (m1 , m4 , newSize), add (m7 , m5 , newSize , -1), newSize); auto c12 = add (m3 , m5 , newSize); auto c21 = add (m2 , m4 , newSize); auto c22 = add (add (m1 , m3 , newSize), add (m6 , m2 , newSize , -1), newSize); // Combine result quadrants into final matrix for (int i = 0 ; i < newSize ; i ++) for (int j = 0 ; j < newSize ; j ++) { res [i][j] = c11 [i][j]; res [i][j + newSize] = c12 [i][j]; res [i + newSize][j] = c21 [i][j]; res [i + newSize][j + newSize] = c22 [i][j]; } return res ; } // Multiply mat1 (n×m) and mat2 (m×q) // using Strassen’s method vector < vector < int >> multiply (vector < vector < int >> & mat1 , vector < vector < int >> & mat2) { // Compute size of the smallest power of 2 ≥ max(n, m, q) int n = mat1 . size (), m = mat1 [0]. size (), q = mat2 [0]. size () ; int size = nextPowerOfTwo (max (n , max (m , q))); // Pad both matrices to size×size with zeros vector < vector < int >> aPad = resizeMatrix (mat1 , size , size); vector < vector < int >> bPad = resizeMatrix (mat2 , size , size); // Perform Strassen multiplication on padded matrices vector < vector < int >> cPad = strassen (aPad , bPad); // Extract the valid n×q result from the padded result vector < vector < int >> C (n , vector < int > (q)); for (int i = 0 ; i < n ; i ++) for (int j = 0 ; j < q ; j ++) C [i][j] = cPad [i][j]; return C ; } int main () { vector < vector < int >> mat1 = {{ 1 , 2 , 3 }, { 4 , 5 , 6 }}; vector < vector < int >> mat2 = {{ 7 , 8 }, { 9 , 10 }, { 11 , 12 }}; vector < vector < int >> res = multiply (mat1 , mat2); for (auto & row : res) { for (int val : row) { cout << val << " " ; } cout << endl ; } return 0 ; } Java import java.util.Scanner ; import java.lang.Math ; import java.util.ArrayList ; public class GfG { // Return the next power of two greater than or equal to n static int nextPowerOfTwo (int n) { return (int) Math . pow (2 , Math . ceil (Math . log (n) / Math . log (2))); } // Resize a matrix to newR x newC and // fill extra space with zeros static int [][] resizeMatrix (int [][] mat , int newR , int newC) { int [][] resized = new int [newR][newC] ; for (int i = 0 ; i < mat . length ; ++ i) for (int j = 0 ; j < mat [0] . length ; ++ j) resized [i][j] = mat [i][j] ; return resized ; } // Perform matrix addition or subtraction // of size×size matrices // sign = 1 for addition, -1 for subtraction static int [][] add (int [][] a , int [][] b , int size , int sign) { int [][] res = new int [size][size] ; for (int i = 0 ; i < size ; i ++) for (int j = 0 ; j < size ; j ++) res [i][j] = a [i][j] + sign * b [i][j] ; return res ; } static int [][] add (int [][] a , int [][] b , int size) { return add (a , b , size , 1); } // Recursive implementation of Strassen's // matrix multiplication // Assumes both matrices are size×size // and size is a power of 2 static int [][] strassen (int [][] mat1 , int [][] mat2) { int n = mat1 . length ; // Base case: 1×1 matrix multiplication int [][] res = new int [n][n] ; if (n == 1) { res [0][0] = mat1 [0][0] * mat2 [0][0] ; return res ; } // Split matrices into 4 submatrices int newSize = n / 2 ; int [][] a11 = new int [newSize][newSize] ; int [][] a12 = new int [newSize][newSize] ; int [][] a21 = new int [newSize][newSize] ; int [][] a22 = new int [newSize][newSize] ; int [][] b11 = new int [newSize][newSize] ; int [][] b12 = new int [newSize][newSize] ; int [][] b21 = new int [newSize][newSize] ; int [][] b22 = new int [newSize][newSize] ; // Fill the submatrices for (int i = 0 ; i < newSize ; i ++) for (int j = 0 ; j < newSize ; j ++) { a11 [i][j] = mat1 [i][j] ; a12 [i][j] = mat1 [i][j + newSize] ; a21 [i][j] = mat1 [i + newSize][j] ; a22 [i][j] = mat1 [i + newSize][j + newSize] ; b11 [i][j] = mat2 [i][j] ; b12 [i][j] = mat2 [i][j + newSize] ; b21 [i][j] = mat2 [i + newSize][j] ; b22 [i][j] = mat2 [i + newSize][j + newSize] ; } // Compute the 7 products using Strassen’s formulas int [][] m1 = strassen (add (a11 , a22 , newSize), add (b11 , b22 , newSize)); int [][] m2 = strassen (add (a21 , a22 , newSize), b11); int [][] m3 = strassen (a11 , add (b12 , b22 , newSize , - 1)); int [][] m4 = strassen (a22 , add (b21 , b11 , newSize , - 1)); int [][] m5 = strassen (add (a11 , a12 , newSize), b22); int [][] m6 = strassen (add (a21 , a11 , newSize , - 1), add (b11 , b12 , newSize)); int [][] m7 = strassen (add (a12 , a22 , newSize , - 1), add (b21 , b22 , newSize)); // Calculate result quadrants from m1...m7 int [][] c11 = add (add (m1 , m4 , newSize), add (m7 , m5 , newSize , - 1), newSize); int [][] c12 = add (m3 , m5 , newSize); int [][] c21 = add (m2 , m4 , newSize); int [][] c22 = add (add (m1 , m3 , newSize), add (m6 , m2 , newSize , - 1), newSize); // Combine result quadrants into final matrix for (int i = 0 ; i < newSize ; i ++) for (int j = 0 ; j < newSize ; j ++) { res [i][j] = c11 [i][j] ; res [i][j + newSize] = c12 [i][j] ; res [i + newSize][j] = c21 [i][j] ; res [i + newSize][j + newSize] = c22 [i][j] ; } return res ; } // Multiply mat1 (n×m) and mat2 (m×q) // using Strassen’s method static ArrayList < ArrayList < Integer >> multiply (int [][] mat1 , int [][] mat2) { int n = mat1 . length , m = mat1 [0] . length , q = mat2 [0] . length ; int size = nextPowerOfTwo (Math . max (n , Math . max (m , q))); // Pad both matrices to size×size with zeros int [][] aPad = resizeMatrix (mat1 , size , size); int [][] bPad = resizeMatrix (mat2 , size , size); // Perform Strassen multiplication on padded matrices int [][] cPad = strassen (aPad , bPad); // Extract the valid n×q result from the padded result ArrayList < ArrayList < Integer >> C = new ArrayList <> (); for (int i = 0 ; i < n ; i ++) { ArrayList < Integer > row = new ArrayList <> (); for (int j = 0 ; j < q ; j ++) { row . add (cPad [i][j]); } C . add (row); } return C ; } public static void main (String [] args) { int [][] mat1 = {{ 1 , 2 , 3 }, { 4 , 5 , 6 }}; int [][] mat2 = {{ 7 , 8 }, { 9 , 10 }, { 11 , 12 }}; ArrayList < ArrayList < Integer >> res = multiply (mat1 , mat2); for (ArrayList < Integer > row :

res) { for (int val : row) { System . out . print (val + " "); } System . out . println (); } } } Python import math # Return the next power of two greater than or equal to n def nextPowerOfTwo (n): return int (math . pow (2 , math . ceil (math . log2 (n)))) # Resize a matrix to newR x newC and # fill extra space with zeros def resizeMatrix (mat , newR , newC): resized = [[0 for _ in range (newC)] for _ in range (newR)] for i in range (len (mat)): for j in range (len (mat [0])): resized [i][j] = mat [i][j] return resized # Perform matrix addition or subtraction # of size×size matrices # sign = 1 for addition, -1 for subtraction def add (a , b , size , sign = 1): res = [[0 for _ in range (size)] for _ in range (size)] for i in range (size): for j in range (size): res [i][j] = a [i][j] + sign * b [i][j] return res # Recursive implementation of Strassen's # matrix multiplication # Assumes both matrices are size×size # and size is a power of 2 def strassen (mat1 , mat2): n = len (mat1) # Base case: 1×1 matrix multiplication res = [[0 for _ in range (n)] for _ in range (n)] if n == 1 : res [0][0] = mat1 [0][0] * mat2 [0][0] return res # Split matrices into 4 submatrices newSize = n // 2 a11 = [[0 for _ in range (newSize)] for _ in range (newSize)] a12 = [[0 for _ in range (newSize)] for _ in range (newSize)] a21 = [[0 for _ in range (newSize)] for _ in range (newSize)] a22 = [[0 for _ in range (newSize)] for _ in range (newSize)] b11 = [[0 for _ in range (newSize)] for _ in range (newSize)] b12 = [[0 for _ in range (newSize)] for _ in range (newSize)] b21 = [[0 for _ in range (newSize)] for _ in range (newSize)] b22 = [[0 for _ in range (newSize)] for _ in range (newSize)] # Fill the submatrices for i in range (newSize): for j in range (newSize): a11 [i][j] = mat1 [i][j] a12 [i][j] = mat1 [i][j + newSize] a21 [i][j] = mat1 [i + newSize][j] a22 [i][j] = mat1 [i + newSize][j + newSize] b11 [i][j] = mat2 [i][j] b12 [i][j] = mat2 [i][j + newSize] b21 [i][j] = mat2 [i + newSize][j] b22 [i][j] = mat2 [i + newSize][j + newSize] # Compute the 7 products using Strassen’s formulas m1 = strassen (add (a11 , a22 , newSize), add (b11 , b22 , newSize)) m2 = strassen (add (a21 , a22 , newSize), b11) m3 = strassen (a11 , add (b12 , b22 , newSize , - 1)) m4 = strassen (a22 , add (b21 , b11 , newSize , - 1)) m5 = strassen (add (a11 , a12 , newSize), b22) m6 = strassen (add (a21 , a11 , newSize , - 1), add (b11 , b12 , newSize)) m7 = strassen (add (a12 , a22 , newSize , - 1), add (b21 , b22 , newSize)) # Calculate result quadrants from m1...m7 c11 = add (add (m1 , m4 , newSize), add (m7 , m5 , newSize , - 1), newSize) c12 = add (m3 , m5 , newSize) c21 = add (m2 , m4 , newSize) c22 = add (add (m1 , m3 , newSize), add (m6 , m2 , newSize , - 1), newSize) # Combine result quadrants into final matrix for i in range (newSize): for j in range (newSize): res [i][j] = c11 [i][j] res [i][j + newSize] = c12 [i][j] res [i + newSize][j] = c21 [i][j] res [i + newSize][j + newSize] = c22 [i][j] return res # Multiply mat1 (n×m) and mat2 (m×q) # using Strassen’s method def multiply (mat1 , mat2): # Compute size of the smallest power of 2 ≥ max(n, m, q) n = len (mat1) m = len (mat1 [0]) q = len (mat2 [0]) size = nextPowerOfTwo (max (n , max (m , q))) # Pad both matrices to size×size with zeros aPad = resizeMatrix (mat1 , size , size) bPad = resizeMatrix (mat2 , size , size) # Perform Strassen multiplication on padded matrices cPad = strassen (aPad , bPad) # Extract the valid n×q result from the padded result C = [[0 for _ in range (q)] for _ in range (n)] for i in range (n): for j in range (q): C [i][j] = cPad [i][j] return C if __name__ == "__main__" : mat1 = [[1 , 2 , 3], [4 , 5 , 6]] mat2 = [[7 , 8], [9 , 10], [11 , 12]] res = multiply (mat1 , mat2) for row in res : for val in row : print (val , end = " ") print () C# using System ; using System.Collections.Generic ; class GfG { // Return the next power of 2 ≥ n static int nextPowerOfTwo (int n) { int power = 1 ; while (power < n) power *= 2 ; return power ; } // Resize a 2D array to newR x newC, padding with 0 static List < List < int >> resizeMatrix (int [,] mat , int newR , int newC) { List < List < int >> resized = new List < List < int >> (); for (int i = 0 ; i < newR ; i ++) { List < int > row = new List < int > (); for (int j = 0 ; j < newC ; j ++) { if (i < mat . GetLength (0) && j < mat . GetLength (1)) row . Add (mat [i , j]); else row . Add (0); } resized . Add (row); } return resized ; } // Add or subtract two matrices static List < List < int >> add (List < List < int >> a , List < List < int >> b , int size , int sign) { var res = new List < List < int >> (); for (int i = 0 ; i < size ; i ++) { var row = new List < int > (); for (int j = 0 ; j < size ; j ++) row . Add (a [i][j] + sign * b [i][j]); res . Add (row); } return res ; } // Strassen’s matrix multiplication (recursive) static List < List < int >> strassen (List < List < int >> mat1 , List < List < int >> mat2) { int n = mat1 . Count ; if (n == 1) { return new List < List < int >> { new List < int > { mat1 [0][0] * mat2 [0][0] } }; } int newSize = n / 2 ; List < List < int >> a11 = new List < List < int >> (); List < List < int >> a12 = new List < List < int >> (); List < List < int >> a21 = new List < List < int >> (); List < List < int >> a22 = new List < List < int >> (); List < List < int >> b11 = new List < List < int >> (); List < List < int >> b12 = new List < List < int >> (); List < List < int >> b21 = new List < List < int >> (); List < List < int >> b22 = new List < List < int >> (); for (int i = 0 ; i < newSize ; i ++) { a11 . Add (new List < int > ()); a12 . Add (new List < int > ()); a21 . Add (new List < int > ()); a22 . Add (new List < int > ()); b11 . Add (new List < int > ()); b12 . Add (new List < int > ()); b21 . Add (new List < int > ()); b22 . Add (new List < int > ()); for (int j = 0 ; j <

newSize ; j ++) { a11 [i]. Add (mat1 [i][j]); a12 [i]. Add (mat1 [i][j + newSize]); a21 [i]. Add (mat1 [i + newSize][j]); a22 [i]. Add (mat1 [i + newSize][j + newSize]); b11 [i]. Add (mat2 [i][j]); b12 [i]. Add (mat2 [i][j + newSize]); b21 [i]. Add (mat2 [i + newSize][j]); b22 [i]. Add (mat2 [i + newSize][j + newSize]); } } var m1 = strassen (add (a11 , a22 , newSize , 1), add (b11 , b22 , newSize , 1)); var m2 = strassen (add (a21 , a22 , newSize , 1), b11); var m3 = strassen (a11 , add (b12 , b22 , newSize , - 1)); var m4 = strassen (a22 , add (b21 , b11 , newSize , - 1)); var m5 = strassen (add (a11 , a12 , newSize , 1), b22); var m6 = strassen (add (a21 , a11 , newSize , - 1), add (b11 , b12 , newSize , 1)); var m7 = strassen (add (a12 , a22 , newSize , - 1), add (b21 , b22 , newSize , 1)); var c11 = add (add (m1 , m4 , newSize , 1), add (m7 , m5 , newSize , - 1), newSize , 1); var c12 = add (m3 , m5 , newSize , 1); var c21 = add (m2 , m4 , newSize , 1); var c22 = add (add (m1 , m3 , newSize , 1), add (m6 , m2 , newSize , - 1), newSize , 1); var res = new List < List < int >> (); for (int i = 0 ; i < n ; i ++) res . Add (new List < int > (new int [n])); for (int i = 0 ; i < newSize ; i ++) { for (int j = 0 ; j < newSize ; j ++) { res [i][j] = c11 [i][j]; res [i][j + newSize] = c12 [i][j]; res [i + newSize][j] = c21 [i][j]; res [i + newSize][j + newSize] = c22 [i][j]; } } return res ; } // Multiply two matrices using Strassen's algorithm static List < List < int >> multiply (int [,] mat1 , int [,] mat2) { int n = mat1 . GetLength (0); int m = mat1 . GetLength (1); int q = mat2 . GetLength (1); int size = nextPowerOfTwo (Math . Max (Math . Max (n , m), q)); var aPad = resizeMatrix (mat1 , size , size); var bPad = resizeMatrix (mat2 , size , size); var cPad = strassen (aPad , bPad); var result = new List < List < int >> (); for (int i = 0 ; i < n ; i ++) { var row = new List < int > (); for (int j = 0 ; j < q ; j ++) row . Add (cPad [i][j]); result . Add (row); } return result ; } static void Main () { int [,] mat1 = { { 1 , 2 , 3 }, { 4 , 5 , 6 } }; int [,] mat2 = { { 7 , 8 }, { 9 , 10 }, { 11 , 12 } }; var result = multiply (mat1 , mat2); foreach (var row in result) { foreach (var val in row) Console . Write (val + " "); Console . WriteLine (); } } } JavaScript // Return the next power of two greater than or equal to n function nextPowerOfTwo (n) { return Math . pow (2 , Math . ceil (Math . log2 (n))); } // Resize a matrix to newR x newC and // fill extra space with zeros function resizeMatrix (mat1 , newR , newC) { let resized = Array . from ({ length : newR }, (_ , i) => Array . from ({ length : newC }, (_ , j) => i < mat1 . length && j < mat1 [0]. length ? mat1 [i][j] : 0)); return resized ; } // Perform matrix addition or subtraction // of size×size matrices // sign = 1 for addition, -1 for subtraction function add (mat1 , mat2 , size , sign = 1) { let res = Array . from ({ length : size }, () => Array (size). fill (0)); for (let i = 0 ; i < size ; i ++) for (let j = 0 ; j < size ; j ++) res [i][j] = mat1 [i][j] + sign * mat2 [i][j]; return res ; } // Recursive implementation of Strassen's // matrix multiplication // Assumes both matrices are size×size // and size is a power of 2 function strassen (mat1 , mat2) { const n = mat1 . length ; const res = Array . from ({ length : n }, () => Array (n). fill (0)); // Base case: 1×1 matrix multiplication if (n === 1) { res [0][0] = mat1 [0][0] * mat2 [0][0]; return res ; } // Split matrices into 4 submatrices const newSize = n / 2 ; const a11 = [], a12 = [], a21 = [], a22 = []; const b11 = [], b12 = [], b21 = [], b22 = []; // Fill the submatrices for (let i = 0 ; i < newSize ; i ++) { a11 . push (mat1 [i]. slice (0 , newSize)); a12 . push (mat1 [i]. slice (newSize)); a21 . push (mat1 [i + newSize]. slice (0 , newSize)); a22 . push (mat1 [i + newSize]. slice (newSize)); b11 . push (mat2 [i]. slice (0 , newSize)); b12 . push (mat2 [i]. slice (newSize)); b21 . push (mat2 [i + newSize]. slice (0 , newSize)); b22 . push (mat2 [i + newSize]. slice (newSize)); } // Compute the 7 products using Strassen's formulas const m1 = strassen (add (a11 , a22 , newSize), add (b11 , b22 , newSize)); const m2 = strassen (add (a21 , a22 , newSize), b11); const m3 = strassen (a11 , add (b12 , b22 , newSize , - 1)); const m4 = strassen (a22 , add (b21 , b11 , newSize , - 1)); const m5 = strassen (add (a11 , a12 , newSize), b22); const m6 = strassen (add (a21 , a11 , newSize , - 1), add (b11 , b12 , newSize)); const m7 = strassen (add (a12 , a22 , newSize , - 1), add (b21 , b22 , newSize)); // Calculate result quadrants from m1...m7 const c11 = add (add (m1 , m4 , newSize), add (m7 , m5 , newSize , - 1), newSize); const c12 = add (m3 , m5 , newSize); const c21 = add (m2 , m4 , newSize); const c22 = add (add (m1 , m3 , newSize), add (m6 , m2 , newSize , - 1), newSize); // Combine result quadrants into final matrix for (let i = 0 ; i < newSize ; i ++) for (let j = 0 ; j < newSize ; j ++) { res [i][j] = c11 [i][j]; res [i][j + newSize] = c12 [i][j]; res [i + newSize][j] = c21 [i][j]; res [i + newSize][j + newSize] = c22 [i][j]; } return res ; } // Multiply mat1 (n×m) and mat2 (m×q) // using Strassen's method function multiply (mat1 , mat2) { const n = mat1 . length , m = mat1 [0]. length , q = mat2 [0]. length ; const size = nextPowerOfTwo (Math . max (n , m , q)); // Pad both matrices to size×size with zeros const aPad = resizeMatrix (mat1 , size , size); const bPad = resizeMatrix (mat2 , size , size); // Perform Strassen multiplication on padded matrices const cPad = strassen (aPad , bPad); // Extract the valid n×q result from the padded result const matrixC = []; for (let i = 0 ; i < n ; i ++) { matrixC . push ([]); for (let j = 0 ; j < q ; j ++) matrixC [i][j] = cPad [i][j]; } return matrixC ; } // Driver Code const mat1 = [[1 , 2 , 3], [4 , 5 , 6]]; const mat2 = [[7 , 8], [9 , 10], [

11 , 12]]; const res = multiply (mat1 , mat2); res . forEach (row => console . log (row . join (" "))); Output 58 64 139 154 Time Complexity: $O(s^3)$ to O(s^log■7) ≈ O(s^2.807), where s is the next power of two greater than max(n, m, q), due to only 7 recursive multiplications. Auxiliary Space: $O(s^2)$ because of the padding and the creation of intermediate submatrices during recursion. Comment Article Tags: Article Tags: Divide and Conquer Matrix DSA