Matrix Chain Multiplication - GeeksforGeeks

Source: https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/

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 Chain Multiplication Last Updated : 23 Jul, 2025 Given the dimension of a sequence of matrices in an array arr[] , where the dimension of the i th matrix is (arr[i-1] * arr[i]) , the task is to find the most efficient way to multiply these matrices together such that the total number of element multiplications is minimum . When two matrices of size m*n and n*p when multiplied, they generate a matrix of size m*p and the number of multiplications performed is m*n*p . Examples: Input: arr[] = [2, 1, 3, 4] Output: 20 Explanation: There are 3 matrices of dimensions 2x1, 1x3, and 3x4, Let the input 3 matrices be M 1 , M 2 , and M 3 . There are two ways to multiply ((M 1 x M 2) x M 3) and (M 1 x (M 2 x M 3)), Please note that the result of M 1 x M 2 is a 2 x 3 matrix and result of (M 2 x M 3) is a 1 x 4 matrix. ((M 1 x M 2) x M 3) requires (2 x 1 x 3) + (2 x 3 x 4) = 30 (M 1 x (M 2 x M 3)) requires (1 x 3 x 4) + (2 x 1 x 4) = 20 The minimum of these two is 20. Input: arr[] = [1, 2, 3, 4, 3] Output: 30 Explanation: There are 4 matrices of dimensions 1×2, 2×3, 3×4, 4×3. Let the input 4 matrices be M 1 , M 2 , M 3 and M 4 . The minimum number of multiplications are obtained by ((M 1 M 2)M 3)M 4 . The minimum number is 1*2*3 + 1*3*4 + 1*4*3 = 30 Input: arr[] = [3, 4] Output: 0 Explanation: As there is only one matrix so, there is no cost of multiplication. Try it on GfG Practice Table of Content [Naive Approach] Using Recursion - O(2^n) and O(n) Space [Better Approach 1] Using Top-Down DP (Memoization) - O(n*n*n) and O(n*n) Space [Better Approach 2] Using Bottom-Up DP (Tabulation) - O(n*n*n) and O(n*n) Space [Naive Approach] Using Recursion - O(2^n) and O(n) Space We can solve the problem using recursion based on the following facts and observations: Now, for a given chain of n matrices , the first partition can be done in n-1 ways. For example, sequence of matrices M 1 , M 2 , M 3 and M 4 can be grouped as (M 1)(M 2 x M 3 x M 4), (M 1 x M 2) x (M 3 x M 4) or ((M 1 x M 2 x M 3) x M 4) in these 3 ways. For n matrices M 1 , M 2 , ... M n . we can put the first bracket n-1 ways (M 1) x (M 2 x M 3 x M 4 ..... M n-1 x M n) (M 1 x M 2) x (M 3 x M 4 ..... M n-1 x M n) (M 1 x M 2 x M 3) x (M 4 ..... M n-1 x M n) .... (M 1 x M 2 x M 3 x M 4 .....M n-1) x (M n) We put the first bracket at different n-1 places and then recursively call for the two parts. At the end, we return the minimum of all partitions. To write a recursive function we use a range of indexes [i, j]. And run a loop for k = i + 1 to j . For k = i + 1, we put the first bracket after the first matrix which has dimensions arr[i] x arr[i+1] and before the remaining matrices which have dimensions arr[i+1] x arr[i+2], arr[i+2] x arr[i+3], ... arr[j-1] x arr[j] For every k , we make two subproblems : (a) Chain from i to k (b) Chain from k to j Each of the subproblems can be further partitioned into smaller subproblems and we can find the total required multiplications by solving for each of the groups. The base case would be when we have only one matrix left which means when j is equal to i+1 . C++ // C++ code to implement the // matrix chain multiplication using recursion #include <bits/stdc++.h> using namespace std ; // Matrix Ai has dimension arr[i-1] x arr[i] int minMultRec (vector < int > & arr , int i , int j) { // If there is only one matrix if (i + 1 == j) return 0 ; int res = INT_MAX ; // Place the first bracket at different // positions or k and for every placed // first bracket, recursively compute // minimum cost for remaining brackets // (or subproblems) for (int k = i + 1 ; k < j ; k ++) { int curr = minMultRec (arr , i , k) + minMultRec (arr , k , j) + arr [i] * arr [k] * arr [j]; res = min (curr , res); } // Return minimum count return res ; } int matrixMultiplication (vector < int > & arr) { int n = arr . size (); return minMultRec (arr , 0 , n - 1); } int main () { vector < int > arr = { 2 , 1 , 3 , 4 }; cout << matrixMultiplication (arr); return 0 ; } Java // Java code to implement the // matrix chain multiplication using recursion import java.util.* ; class GfG { // Matrix Ai has dimension arr[i-1] x arr[i] static int

minMultRec (int arr [] , int i , int j) { // If there is only one matrix if (i + 1 == j) return 0 ; int res = Integer . MAX_VALUE ; // Place the first bracket at different // positions or k and for every placed // first bracket, recursively compute // minimum cost for remaining brackets // (or subproblems) for (int k = i + 1 ; k < j ; k ++) { int curr = minMultRec (arr , i , k) + minMultRec (arr , k , j) + arr [i] * arr [k] * arr [j] ; res = Math . min (curr , res); } // Return minimum count return res ; } static int matrixMultiplication (int arr []) { int n = arr . length ; return minMultRec (arr , 0 , n - 1); } public static void main (String [] args) { int arr [] = { 2 , 1 , 3 , 4 }; int res = matrixMultiplication (arr); System . out . println (res); } } Python # Python code to implement the # matrix chain multiplication using recursion import sys # Matrix Ai has dimension arr[i-1] x arr[i] def minMultRec (arr , i , j): # If there is only one matrix if i + 1 == j : return 0 res = sys . maxsize # Place the first bracket at different # positions or k and for every placed # first bracket, recursively compute # minimum cost for remaining brackets # (or subproblems) for k in range (i + 1 , j): curr = minMultRec (arr , i , k) \ + minMultRec (arr , k , j) \ + arr [i] * arr [k] * arr [j] res = min (curr , res) # Return minimum count return res def matrixMultiplication (arr): n = len (arr) return minMultRec (arr , 0 , n - 1) if __name__ == "__main__" : arr = [2 , 1 , 3 , 4] res = matrixMultiplication (arr) print (res) C# // C# code to implement the // matrix chain multiplication using recursion using System ; class GfG { // Matrix Ai has dimension arr[i-1] x arr[i] static int minMultRec (int [] arr , int i , int j) { // If there is only one matrix if (i + 1 == j) return 0 ; int res = int . MaxValue ; // Place the first bracket at different // positions or k and for every placed // first bracket, recursively compute // minimum cost for remaining brackets // (or subproblems) for (int k = i + 1 ; k < j ; k ++) { int curr = minMultRec (arr , i , k) + minMultRec (arr , k , j) + arr [i] * arr [k] * arr [j]; res = Math . Min (curr , res); } // Return minimum count return res ; } static int matrixMultiplication (int [] arr) { int n = arr . Length ; return minMultRec (arr , 0 , n - 1); } static void Main (string [] args) { int [] arr = { 2 , 1 , 3 , 4 }; int res = matrixMultiplication (arr); Console . WriteLine (res); } } JavaScript // JavaScript code to implement the // matrix chain multiplication using recursion function minMultRec (arr , i , j) { // If there is only one matrix if (i + 1 === j) return 0 ; let res = Number . MAX_SAFE_INTEGER ; // Place the first bracket at different // positions or k and for every placed // first bracket, recursively compute // minimum cost for remaining brackets // (or subproblems) for (let k = i + 1 ; k < j ; k ++) { let curr = minMultRec (arr , i , k) + minMultRec (arr , k , j) + arr [i] * arr [k] * arr [j]; res = Math . min (curr , res); } // Return minimum count return res ; } function matrixMultiplication (arr) { const n = arr . length ; return minMultRec (arr , 0 , n - 1); } let arr = [2 , 1 , 3 , 4]; let res = matrixMultiplication (arr); console . log (res); Output 20 [Better Approach 1] Using Top-Down DP (Memoization) - O(n*n*n) and O(n*n) Space Let's suppose we have four matrices (M 1 , M 2 , M 3 , M 4). Based on the recursive approach described above, we can construct a recursion tree . However, we can observe that some problems are computed multiple times. To avoid this redundant computation, we can implement memoization to store the results of previously computed inputs. Recursion Tree for Matrix Chain Multiplication If observed carefully you can find the following two properties: 1) Optimal Substructure: In the above case, we are breaking the bigger groups into smaller subgroups and solving them to finally find the minimum number of multiplications. Therefore, it can be said that the problem has optimal substructure property. 2) Overlapping Subproblems: We can see in the recursion tree that the same subproblems are called again and again and this problem has the Overlapping Subproblems property. So Matrix Chain Multiplication problem has both properties of a dynamic programming problem. So recomputations of same subproblems can be avoided by constructing a temporary array memo[][] in a bottom up manner. Follow the below steps to solve the problem: Build a matrix memo[][] of size n*n for memoization purposes. Use the same recursive call as done in the above approach: When we find a range (i, j) for which the value is already calculated, return the minimum value for that range (i.e., memo[i][j]). Otherwise, perform the recursive calls as mentioned earlier. The value stored at memo[0][n-1] is the required answer. C++ // C++ code to implement the // matrix chain multiplication using memoization #include <bits/stdc++.h> using namespace std ; int minMultRec (vector < int > & arr , int i , int j , vector < vector < int >> & memo) { // If there is only one matrix if (i + 1 == j) return 0 ; // Check if the result is already // computed if (memo [i][j] != -1) return memo [i][j]; int res = INT_MAX ; // Place the first bracket at different positions or k and // for every placed first bracket, recursively compute // minimum cost for remaining brackets (or subproblems) for (int k = i + 1 ; k < j ; k ++) { int curr = minMultRec (arr , i , k , memo) + minMultRec (arr , k , j , memo) + arr [i] * arr [k] * arr [j]; res = min (curr , res); } // Store the result in memo table memo [i][j] = res ; return res ; } int matrixMultiplication (vector < int > & arr) { int n = arr . size (); vector < vector < int >> memo (n , vector < int > (n , -1)); return minMultRec (arr , 0 , n - 1 , memo); } int main () { vector < int > arr = { 2 , 1 , 3 , 4 }; int res = matrixMultiplication (arr); cout << res << endl ; return 0 ; } Java // Java code to implement the // matrix chain multiplication using memoization import java.util.Arrays ; class

GfG { // Function to compute minimum multiplication // recursively static int minMultRec (int [] arr , int i , int j , int [][] memo) { // If there is only one matrix if (i + 1 == j) return 0 ; // Check if the result is already computed if (memo [i][j] != - 1) return memo [i][j] ; int res = Integer . MAX_VALUE ; // Place the first bracket at different positions or // k and for every placed first bracket, recursively // compute minimum cost for remaining brackets (or // subproblems) for (int k = i + 1 ; k < j ; k ++) { int curr = minMultRec (arr , i , k , memo) + minMultRec (arr , k , j , memo) + arr [i] * arr [k] * arr [j] ; res = Math . min (curr , res); } // Store the result in memo table memo [i][j] = res ; return res ; } static int matrixMultiplication (int [] arr) { int n = arr . length ; int [][] memo = new int [n][n] ; for (int [] row : memo) Arrays . fill (row , - 1); return minMultRec (arr , 0 , n - 1 , memo); } public static void main (String [] args) { int [] arr = { 2 , 1 , 3 , 4 }; System . out . println (matrixMultiplication (arr)); } } Python # Python code to implement the # matrix chain multiplication using memoization import sys # Function to compute minimum multiplication # recursively def minMultRec (arr , i , j , memo): # If there is only one matrix if i + 1 == j : return 0 # Check if the result is already computed if memo [i][j] != - 1 : return memo [i][j] res = sys . maxsize # Place the first bracket at different positions or k # and for every placed first bracket, recursively compute # minimum cost for remaining brackets (or subproblems) for k in range (i + 1 , j): curr = (minMultRec (arr , i , k , memo) + minMultRec (arr , k , j , memo) + arr [i] * arr [k] * arr [j]) res = min (res , curr) # Store the result in memo table memo [i][j] = res return res def matrixMultiplication (arr): n = len (arr) memo = [[- 1 for _ in range (n)] for _ in range (n)] return minMultRec (arr , 0 , n - 1 , memo) if __name__ == "__main__" : arr = [2 , 1 , 3 , 4] res = matrixMultiplication (arr) print (res) C# // C# code to implement the // matrix chain multiplication using memoization using System ; class GfG { // Function to compute minimum multiplication // recursively static int MinMultRec (int [] arr , int i , int j , int [,] memo) { // If there is only one matrix if (i + 1 == j) return 0 ; // Check if the result is already computed if (memo [i , j] != - 1) return memo [i , j]; int res = int . MaxValue ; // Place the first bracket at different positions or // k and for every placed first bracket, recursively // compute minimum cost for remaining brackets (or // subproblems) for (int k = i + 1 ; k < j ; k ++) { int curr = MinMultRec (arr , i , k , memo) + MinMultRec (arr , k , j , memo) + arr [i] * arr [k] * arr [j]; res = Math . Min (curr , res); } // Store the result in memo table memo [i , j] = res ; return res ; } static int matrixMultiplication (int [] arr) { int n = arr . Length ; int [,] memo = new int [n , n]; for (int i = 0 ; i < n ; i ++) for (int j = 0 ; j < n ; j ++) memo [i , j] = - 1 ; return MinMultRec (arr , 0 , n - 1 , memo); } static void Main () { int [] arr = { 2 , 1 , 3 , 4 }; int res = matrixMultiplication (arr); Console . WriteLine (res); } } JavaScript // JavaScript code to implement the // matrix chain multiplication using memoization function minMultRec (arr , i , j , memo) { // If there is only one matrix if (i + 1 === j) return 0 ; // Check if the result is already computed if (memo [i][j] !== - 1) return memo [i][j]; let res = Number . MAX_SAFE_INTEGER ; // Place the first bracket at different positions or k // and for every placed first bracket, recursively // compute minimum cost for remaining brackets (or // subproblems) for (let k = i + 1 ; k < j ; k ++) { let curr = minMultRec (arr , i , k , memo) + minMultRec (arr , k , j , memo) + arr [i] * arr [k] * arr [j]; res = Math . min (res , curr); } // Store the result in memo table memo [i][j] = res ; return res ; } function matrixMultiplication (arr) { let n = arr . length ; let memo = Array . from ({ length : n }, () => Array (n). fill (- 1)); return minMultRec (arr , 0 , n - 1 , memo); } let arr = [2 , 1 , 3 , 4]; let res = matrixMultiplication (arr); console . log (res); Output 20 [Better Approach 2] Using Bottom-Up DP (Tabulation) - O(n*n*n) and O(n*n) Space In iterative approach, we initially need to find the number of multiplications required to multiply two adjacent matrices. We can use these values to find the minimum multiplication required for matrices in a range of length 3 and further use those values for ranges with higher length. The iterative implementation is going to be tricky here we initially know diagonal values (which are 0), our result is going to be at the top right corner (or dp[0][n-1]) and we never access lower diagonal values. So we cannot fill the matrix with a normal traversal, we rather need to fill in diagonal manner. We fill the matrix using a variable len that stores differences between row and column indexes. We keep increasing len until it becomes n-1 (for the top right element) C++ // C++ code to implement the // matrix chain multiplication using tabulation #include <bits/stdc++.h> using namespace std ; int matrixMultiplication (vector < int > & arr) { int n = arr . size (); // Create a 2D DP array to store the minimum // multiplication costs vector < vector < int >> dp (n , vector < int > (n , 0)); // Fill the DP array. // Here, len is the chain length for (int len = 2 ; len < n ; len ++) { for (int i = 0 ; i < n - len ; i ++) { int j = i + len ; dp [i][j] = INT_MAX ; for (int k = i + 1 ; k < j ; k ++) { int cost = dp [i][k] + dp [k][j] + arr [i] * arr [k] * arr [j]; dp [i][j] = min (dp [i][j], cost); } } } // The minimum cost is stored in dp[0][n-1] return dp [0][n - 1]; } int main () { vector < int > arr = { 2 , 1 , 3 , 4 }; cout << matrixMultiplication (arr); return 0 ; } Java // Java code to implement the // matrix chain multiplication using tabulation class GfG { // Function to calculate min multiplication cost static int matrixMultiplication

(int [] arr) { int n = arr . length ; // Create a 2D DP array to store minimum // multiplication costs int [][] dp = new int [n][n] ; // Fill the DP array // len is the chain length for (int len = 2 ; len < n ; len ++) { for (int i = 0 ; i < n - len ; i ++) { int j = i + len ; dp [i][j] = Integer . MAX_VALUE ; for (int k = i + 1 ; k < j ; k ++) { int cost = dp [i][k] + dp [k][j] + arr [i] * arr [k] * arr [j] ; if (cost < dp [i][j]) { dp [i][j] = cost ; } } } } // Minimum cost is in dp[0][n-1] return dp [0][n - 1] ; } public static void main (String [] args) { int [] arr = { 2 , 1 , 3 , 4 }; System . out . println (matrixMultiplication (arr)); } } Python # Python code to implement the # matrix chain multiplication using tabulation def matrixMultiplication (arr): n = len (arr) # Create a 2D DP array to store min # multiplication costs dp = [[0] * n for _ in range (n)] # Fill the DP array # length is the chain length for length in range (2 , n): for i in range (n - length): j = i + length dp [i][j] = float ('inf') for k in range (i + 1 , j): cost = dp [i][k] + dp [k][j] + arr [i] * arr [k] * arr [j] dp [i][j] = min (dp [i][j], cost) # Minimum cost is in dp[0][n-1] return dp [0][n - 1] arr = [2 , 1 , 3 , 4] print (matrixMultiplication (arr)) C# // C# code to implement the // matrix chain multiplication using tabulation using System ; class GfG { // Function to calculate min multiplication cost static int matrixMultiplication (int [] arr) { int n = arr . Length ; // Create a 2D DP array to store minimum // multiplication costs int [,] dp = new int [n , n]; // Fill the DP array // len is the chain length for (int len = 2 ; len < n ; len ++) { for (int i = 0 ; i < n - len ; i ++) { int j = i + len ; dp [i , j] = int . MaxValue ; for (int k = i + 1 ; k < j ; k ++) { int cost = dp [i , k] + dp [k , j] + arr [i] * arr [k] * arr [j]; if (cost < dp [i , j]) { dp [i , j] = cost ; } } } } // Minimum cost is in dp[0][n-1] return dp [0 , n - 1]; } static void Main () { int [] arr = { 2 , 1 , 3 , 4 }; Console . WriteLine (matrixMultiplication (arr)); } } JavaScript // JavaScript code to implement the // matrix chain multiplication using tabulation function matrixMultiplication (arr) { const n = arr . length ; // Create a 2D DP array to store min // multiplication costs const dp = Array . from ({ length : n }, () => Array (n). fill (0)); // Fill the DP array // len is the chain length for (let len = 2 ; len < n ; len ++) { for (let i = 0 ; i < n - len ; i ++) { let j = i + len ; dp [i][j] = Infinity ; for (let k = i + 1 ; k < j ; k ++) { let cost = dp [i][k] + dp [k][j] + arr [i] * arr [k] * arr [j]; dp [i][j] = Math . min (dp [i][j], cost); } } } // Minimum cost is in dp[0][n-1] return dp [0][n - 1]; } const arr = [2 , 1 , 3 , 4]; console . log (matrixMultiplication (arr)); Output 20 Related articles: Matrix Chain Multiplication (O(n*n) Solution) Printing brackets in Matrix Chain Multiplication Problem Applications: Minimum and Maximum values of an expression with * and + Comment Article Tags: Article Tags: Dynamic Programming Matrix DSA Microsoft Amazon matrix-chain-multiplication + 2 More