

Matrix chain multiplication is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, we want to find the most efficient way to multiply these matrices together. The problem is not actually to *perform* the multiplications, but merely to decide in what order to perform the multiplications.

We have many options because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A , B , C , and D , we would have:

$$(ABC)D = (AB)(CD) = A(BCD) = A(BC)D = ..$$

We could go through each possible parenthesization (brute force), but this would require time $O(2^n)$, which is very slow and impractical for large n .

One simple solution is called memoization: each time we compute the minimum cost needed to multiply out a specific subsequence, we save it. If we are ever asked to compute it again, we simply give the saved answer, and do not recompute it. Since there are about $n^2/2$ different subsequences, where n is the number of matrices, the space required to do this is reasonable. It can be shown that this simple trick brings the runtime down from $O(2^n)$ to $O(n^3)$, which is more than efficient enough for real applications. This is top-down dynamic programming.

Dynamic Programming: Another solution is to anticipate which costs we will need and pre-compute them. It works like this:

For each l from 2 to n , the number of matrices: Compute the minimum costs of each subsequence of length l , using the costs already computed.

```

/** dim is the sizes of the matrices
    Matrix Ai has the dimension dim[i-1] x dim[i].
    cost[i,j] is the best cost of multiplying matrices i through j
    s[i,j] saves the division point so you can actually report the
    matrix multiplication order
*/

void matrixChainOrder(int [] dim)
{
    int [][] cost = new int [SIZE+1][SIZE+1] ;
    int [][] s = new int [SIZE+1][SIZE+1];
    int matrixCt = dim.length - 1;
    for (int i = 1; i <= matrixCt; i++) // actually not needed in Java
        cost[i][i] = 0;

    // Fills in matrix by diagonals
    for (int l=2; l<= matrixCt; l++) { // l is chain length
        for (int i=1; i<= matrixCt -l+1; i++) {
            int j = i+l-1;
            cost[i][j] = Integer.MAX_VALUE; // computing the best cost so far
            // Try all possible division points i..k and k..j
            for (int k=i; k<=j-1; k++) {
                int thisCost = cost[i][k] + cost[k+1][j] + dim[i-1]*dim[k]*dim[j];
                if (thisCost < cost[i][j]) {
                    cost[i][j] = thisCost;
                    s[i][j] = k;
                }
            }
        }
    }
}

```

Cost of matrix Multiplies

	A	B	C	D	E	F
A	0	5000	2100	2250	2220	2820
B	0	0	2000	2300	2150	3130
C	0	0	0	3000	690	8330
D	0	0	0	0	90	330
E	0	0	0	0	0	1800
F	0	0	0	0	0	0

s: Save division points

	A	B	C	D	E	F
A	0	1	1	3	3	5
B	0	0	2	3	3	3
C	0	0	0	3	3	3
D	0	0	0	0	4	5
E	0	0	0	0	0	5
F	0	0	0	0	0	0

The $s[1][6]=5$ means divide ABCDEF after the fifth matrix so divide as (ABCDE)F

To find the next division point, see $s[1][5] = 3$ so ((ABC)(DE))F

To find the next division point, see $s[1][3] = 1$ so ((A(BC))(DE))F