CSE3318: Dynamic Programming
Matrix Chain Multiplication

by

Dr. Bhanu Jain

Unauthorized copying, distribution, or reproduction of this material is **strictly prohibited**.

All slides are based on: ***Introduction to Algorithms***, by Thomas H. Cormen, Charles E. Leiserson, Ronald E. Rivest, Clifford Stein,3rd edition (CLRS)

© Bhanu Jain 2025

Dynamic Programming

- **Divide and Conquer method:**
  - solve problems by combining the solutions to subproblems
  - partition the problem into disjoint subproblems
  - solve the subproblems recursively, and then combine their solutions to solve
  - does more work than necessary, repeatedly solving the common subproblems
- **Dynamic programming:**
  - applies when the subproblems overlap —that is, when subproblems share subproblems.
  - solves each subproblem just once and then saves its answer in a table
  - avoiding the work of recomputing the answer every time it solves each subproblem
  - applied to optimization problems
  - such subproblems can have many possible solutions
  - such solution has a value, find a solution with the optimal (minimum or maximum) value
  - call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution

© Bhanu Jain 2025

Dynamic Programming

- **Dynamic-programming algorithm**, follows a sequence of **four steps**:
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution, typically in a bottom-up fashion.
  4. Construct an optimal solution from computed information.

  - **Steps 1–3** form the basis of a dynamic-programming solution to a problem
  - If we need only the value of an optimal solution, and not the solution itself, omit step 4
  - When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution

© Bhanu Jain 2025

Dynamic Programming

- Like divide-and-conquer, solve problem by combining the solutions to sub-problems.
- Differences between divide-and-conquer and DP:
  - Independent sub-problems, solve sub-problems independently and recursively, (so same sub(sub)problems solved repeatedly)
  - DP is applicable when the sub-problems are not independent, i.e. when sub-problems share sub-sub-problems. It solves every sub-sub-problem just once and save the results in a table to avoid duplicated computation.

© Bhanu Jain 2025

Elements of Dynamic Programming Algorithms

- **Sub-structure:** decompose problem into smaller sub-problems. Express the solution of the original problem in terms of solutions for smaller problems.
- **Table-structure:** Store the answers to the sub-problem in a table, because sub-problem solutions may be used many times.
- **Bottom-up computation:** combine solutions on smaller sub-problems to solve larger sub-problems, and eventually arrive at a solution to the complete problem.

© Bhanu Jain 2025

Applicability to Optimization Problems

- **Optimal sub-structure (principle of optimality):** for the global problem to be solved optimally, each sub-problem should be solved optimally. This is often violated due to sub-problem overlaps. Often by being “less optimal” on one problem, we may make a big savings on another sub-problem.

- **Small number of sub-problems:** Many NP-hard problems can be formulated as DP problems, but these formulations are not efficient, because the number of sub-problems is exponentially large.

  Ideally, the number of sub-problems should be at most a polynomial number.

© Bhanu Jain 2025

Matrix-Chain Multiplication

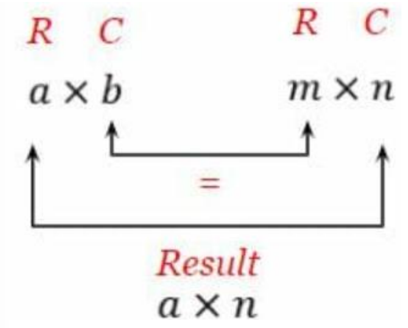
- An example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication
- Problem: Given a sequence (chain) $< A_1, A_2, \ldots\ldots, A_n >$ compute $A_1 * A_2 * \ldots * A_n$
- Matrix multiplication is associative, and so all parenthesizations yield the same product.
- A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.
- The chain of matrices $< A_1, A_2, A_3, A_4 >$ can be parenthesized in 5 distinct ways.
- Each of these 5 choices have a different cost for evaluating the product.

$$
\begin{aligned}
&(A_1(A_2(A_3A_4))) \ , \\
&(A_1((A_2A_3)A_4)) \ , \\
&((A_1A_2)(A_3A_4)) \ , \\
&((A_1(A_2A_3))A_4) \ , \\
&(((A_1A_2)A_3)A_4) \ .
\end{aligned}
$$

© Bhanu Jain 2025

Matrix Multiplication (A,B)

```
MATRIX-MULTIPLY(A, B)
if A.columns ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows × B.columns matrix
    for i = 1 to A.rows
        for j = 1 to B.columns
            c_ij = 0
            for k = 1 to A.columns
                c_ij = c_ij + a_ik · b_kj
    return C
```



$$\begin{bmatrix} 1 & 4 \\ -2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 3 & -4 \\ 6 & 8 & 0 \end{bmatrix}$$

$$(2 \text{ x } \underbrace{2) \cdot (2}_{\text{same}} \text{ x } 3)$$

$$\begin{bmatrix} 5 & -3 \\ 4 & 1 \\ 7 & -2 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 2 \end{bmatrix}$$

$$(3 \text{ x } \underbrace{2) \cdot (2}_{\text{same}} \text{ x } 1)$$

$$A = \begin{bmatrix} 1 & 3 \\ 1 & -2 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & -3 \\ 1 & 1 \end{bmatrix}$$

$$A \times B = \begin{bmatrix} 1 & 3 \\ 1 & -2 \end{bmatrix} \times \begin{bmatrix} 0 & -3 \\ 1 & 1 \end{bmatrix}$$

$$A \times B = \begin{bmatrix} (1)(0) + (3)(1) & (1)(-3) + (3)(1) \\ (1)(0) + (-2)(1) & (1)(-3) + (-2)(1) \end{bmatrix}$$

$$A \times B = \begin{bmatrix} 3 & 0 \\ -2 & -5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \text{ x } \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}$$

$$= \begin{bmatrix} 1\text{x}10 + 2\text{x}20 + 3\text{x}30 & 1\text{x}11 + 2\text{x}21 + 3\text{x}31 \\ 4\text{x}10 + 5\text{x}20 + 6\text{x}30 & 4\text{x}11 + 5\text{x}21 + 6\text{x}31 \end{bmatrix}$$

$$= \begin{bmatrix} 10+40+90 & 11+42+93 \\ 40+100+180 & 44+105+186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$

© Bhanu Jain 2025

Matrix Chain Products

- **Matrix Chain-Product:**
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is 3 × 100
  - C is 100 × 5
  - D is 5 × 5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

BC 3X5
(BC)D (3x5)* 5X5 => 3X5X5

© Bhanu Jain 2025

Enumeration Approach

- **Matrix Chain-Product Algorithm.:**
  - Try all possible ways to parenthesize $A=A_1*A_2*\ldots*A_n$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time:
  - The number of parenthesizations is equal to the number of binary trees with n nodes
  - This is **exponential**!

© Bhanu Jain 2025

Greedy Approach

- Idea #1: repeatedly select the product that uses the **fewest operations.**
- Counter-example:
  - A is 101 × 11
  - B is 11 × 9
  - C is 9 × 100
  - D is 100 × 99
  - Greedy idea #1 gives A*((B*C)*D)), which takes **109989**+**9900**+**108900**=228,789 ops

BC	11x9x100=>**9900**
(B*C)*D	11x100x99 =>**108900**
A*((B*C)*D))	101x11x99 =>**109989**

  - (A*B)*(C*D) takes **9999**+**89991**+**89100**=189090 ops

AxB	101x11x9 =>**9999**
C*D	9x100x99 =>**89100**
(A*B)*(C*D)	101x9x99 =>**89,991**

- The greedy approach is not giving us the optimal value.

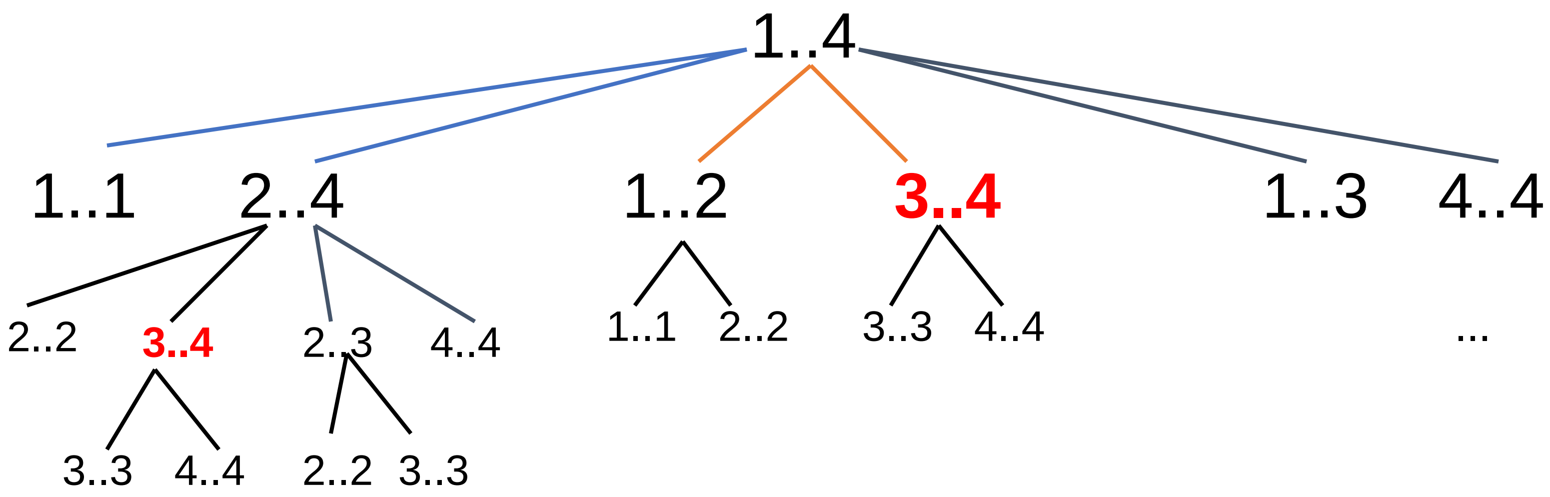
© Bhanu Jain 2025

Recursive Approach

- **Define subproblems:**
  - Find the best parenthesization of $A_i * A_{i+1} * \ldots * A_j$.
  - Let m[i, j] denote the number of operations done by this subproblem.
- **Subproblem optimality:** The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiplication is at index i: $(A_1 * \ldots * A_i) * (A_{i+1} * \ldots * A_n)$.
  - Then the optimal solution m[1, n] is the sum of two optimal subproblems, m[1, i] and m[i+1, n] plus the time for the last multiplication.

© Bhanu Jain 2025

Recursive Tree For The Computation of Recursive-Matrix-Chain (p,1,4)



- This divide-and-conquer recursive algorithm solves the overlapping problems *over and over.*
- **In contrast**, DP
  - solves the same (overlapping) subproblems only once (at the first time), then store the result in a table
  - when the same subproblem is encountered later, just look up the table to get the result.
- The divide-and-conquer is better for the problem which generates brand-new problems at each step of recursion.

© Bhanu Jain 2025

Subproblem Overlap

Algorithm ***RecursiveMatrixChain***(***S***, ***i***, ***j***):

Input: sequence ***S*** of ***n*** matrices to be multiplied

Output: number of operations in an optimal parenthesization of ***S***

if i=j

then return 0

for $k \leftarrow i$ **to** j **do**

$N_{i,j} \leftarrow \min\{N_{i,j},$ ***RecursiveMatrixChain***$(S, i, k)+$ ***RecursiveMatrixChain***$(S, k+1, j) + d_{i-1} d_k d_j\}$

return $N_{i,j}$

© Bhanu Jain 2025

Characterizing Equation

- The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiplication is at.
- Let us consider all possible places for that final multiplication:
  - m[i, j] = the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$ plus the cost of multiplying these two matrices together
  - Computing matrix product $A_{i..k}$ $A_{k+1..j}$ takes p_{i-1} p_k p_j scalar multiplications

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$$

  - The recursive definition for the minimum cost of parenthesizing the product A_i A_{i+1} …A_j is

$$m[i, j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j\ . \end{cases}$$

  - m[i, j] values give the costs of optimal solutions to subproblems, but do not provide all the information to construct an optimal solution
  - s[i, j] is a value of k at which to split the product A_i* A_2*…* A_j in an optimal paranthesization.
- Note that subproblems are not independent–the subproblems overlap.

© Bhanu Jain 2025

Matrix-Chain-Order(p)

MATRIX-CHAIN-ORDER(p)

Input: Array ***p*** of size ***n+1*** representing the dimensions of n matrices

Output: Two tables: m[i,j] (min cost) and s[i,j] (split points)

```
n = length(p) - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i ← 1 to n
   m[i, i] ← 0
for l ←2 to n  // l is the chain length
   for i ← 1 to n - l + 1
      j ← i + l - 1
      m[i, j] ← ∞
      for k ← i to j - 1
         q ← m[i, k] + m[k+1, j] + p[i-1] * p[k] * p[j]
         if q < m[i, j]
            m[i, j] ← q
            s[i, j] ← k
return m, s
```

- This procedure assumes that matrix A_i has dimensions p_{i-1} p_i for i=1,2,…n
- Its input is a sequence p =<p_0, p_1,…,p_n>, where p.length= n+ 1
- The procedure uses an auxiliary table m[1..n, 1..n] for storing the m[i, j] costs
- The procedure uses an auxiliary table s[1..n-1, 2..n] that records which index of k achieved the optimal cost in computing m[i, j].
- We use the table s to construct an optimal solution.
- Recursive definition for the minimum cost of parenthesizing the product $A_iA_{i+1}\ldots A_j$ becomes

- Equation shows that the cost m[I,j] of computing a matrix-chain product of j – I +1 matrices depends only on the costs of computing matrix-chain products of fewer than j – I +1 matrices.
- The algorithm fills the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length

A = [2,3,2,4,5]

A1(2x3)   A2(3X2)   A3(2X4)   A4(4X5)

© Bhanu Jain 2025

Matrix-Chain-Order(p)

```
MATRIX-CHAIN-ORDER(p)
```

Input: Array ***p*** of size ***n+1*** representing the dimensions of n matrices

Output: Two tables: m[i,j] (min cost) and s[i,j] (split points)

```
1    n = length(p) - 1
2    let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
3    for i ← 1 to n
4        m[i, i] ← 0
5    for l ←2 to n  // l is the chain length
6        for i ← 1 to n - l + 1
7            j ← i + l - 1
8            m[i, j] ← ∞
9            for k ← i to j - 1
10               q ← m[i, k] + m[k+1, j] + p[i-1] * p[k] * p[j]
11               if q < m[i, j]
12                   m[i, j] ← q
13                   s[i, j] ← k
14   return m, s
```

- Line 3-4: First compute m[i,i]= 0 for i=1,2,… n (the minimum costs for chains of length 1)
- Line 5-13: Use recurrence to compute m[i, i+1] for i = 1, 2,… n – 1 (min cost for chains of length 2)
  - Then compute m[i, i+2] for i = 1, 2,… n – 2 (min cost for chains of length l= 3) an so on
- Lines 10–13: At each step, the m[i, j] cost computed depends only on table entries m[i, k] and m[k+1, j] already computed
- Since we have defined m[i, j] only for i ≤ j , only the portion of the table m strictly above the main diagonal is used.

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

A = [2,3,2,4,5]

A1(2x3)   A2(3X2)   A3(2X4)   A4(4X5)

© Bhanu Jain 2025

Matrix-Chain-Order(p)

MATRIX-CHAIN-ORDER(p)

Input: Array ***p*** of size ***n+1*** representing the dimensions of n matrices

Output: Two tables: m[i,j] (min cost) and s[i,j] (split points)

```
n = length(p) - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i ← 1 to n
   m[i, i] ← 0
for l ←2 to n  // l is the chain length
   for i ← 1 to n - l + 1
      j ← i + l - 1
      m[i, j] ← ∞
      for k ← i to j - 1
          q ← m[i, k] + m[k+1, j] + p[i-1] * p[k] * p[j]
          if q < m[i, j]
             m[i, j] ← q
             s[i, j] ← k
return m, s
```

- The nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm
- The loops are nested three deep, and each loop index (l, i, and k) takes on at most n – 1 values
- The algorithm requires $\Theta(n^2)$ space to store the m and s tables
- The algorithm is more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one

A = [2,3,2,4,5]

A1(2x3) A2(3X2) A3(2X4) A4(4X5)

© Bhanu Jain 2025

A = [2,3,2,4,5]

A1(2x3)   A2(3X2)   A3(2X4)   A4(4X5)

Matrix-Chain-Order(p)

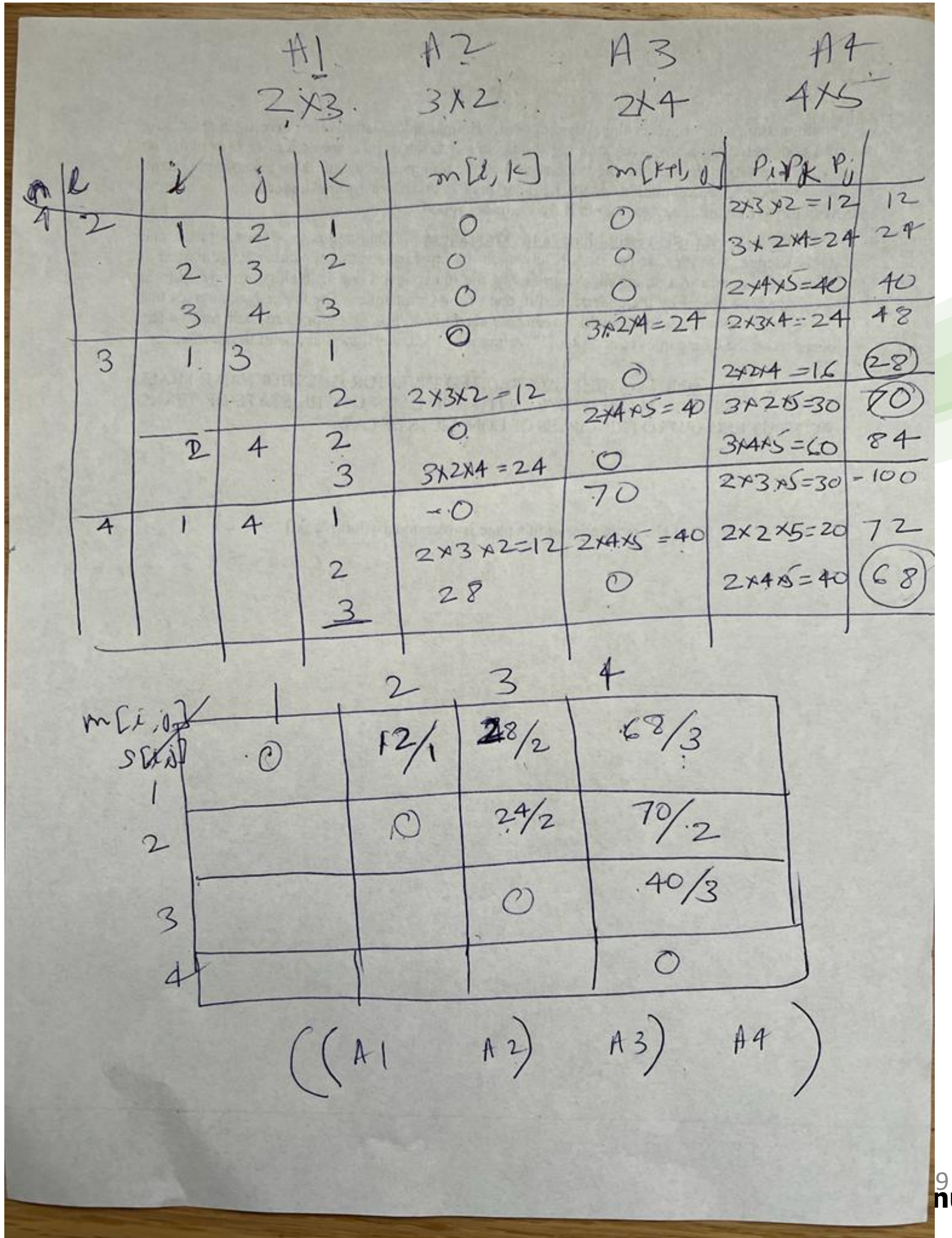
MATRIX-CHAIN-ORDER(p)

Input: Array ***p*** of size ***n+1*** representing the dimensions of n matrices

Output: Two tables: m[i,j] (min cost) and s[i,j] (split points)

```
n = length(p) - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i ← 1 to n
    m[i, i] ← 0
for l ←2 to n  // l is the chain length
    for i ← 1 to n - l + 1
        j ← i + l - 1
        m[i, j] ← ∞
        for k ← i to j - 1
            q ← m[i, k] + m[k+1, j] + p[i-1] * p[k] * p[j]
            if q < m[i, j]
                m[i, j] ← q
                s[i, j] ← k
 return m, s
```

m= [[0, 12, 28, 68],
[, 0, 24, 70],
[0, 40],
[0]]



The END!

© Bhanu Jain 2025