CE100 Algorithms and Programming II

Week-5 (Dynamic Programming)

Spring Semester, 2021-2022

Download DOC, SLIDE, PPTX

<iframe width=700, height=500 frameBorder=0 src="../ce100-week-5-dp.md_slide.html"></iframe>

Quicksort Sort

Outline

- Convex Hull (Divide & Conquer)
- Dynamic Programming
  - Introduction
  - Divide-and-Conquer (DAC) vs Dynamic Programming (DP)

- Fibonacci Numbers
  - Recursive Solution
  - Bottom-Up Solution
- Optimization Problems
- Development of a DP Algorithms

- Matrix-Chain Multiplication
  - Matrix Multiplication and Row Columns Definitions
  - Cost of Multiplication Operations (pxqxr)
  - Counting the Number of Parenthesizations

- The Structure of Optimal Parenthesization
  - Characterize the structure of an optimal solution
  - A Recursive Solution
    - Direct Recursion Inefficiency.
  - Computing the optimal Cost of Matrix-Chain Multiplication
  - Bottom-up Computation

- Algorithm for Computing the Optimal Costs
  - MATRIX-CHAIN-ORDER
- Construction and Optimal Solution
  - MATRIX-CHAIN-MULTIPLY
- Summary

Dynamic Programming - Introduction

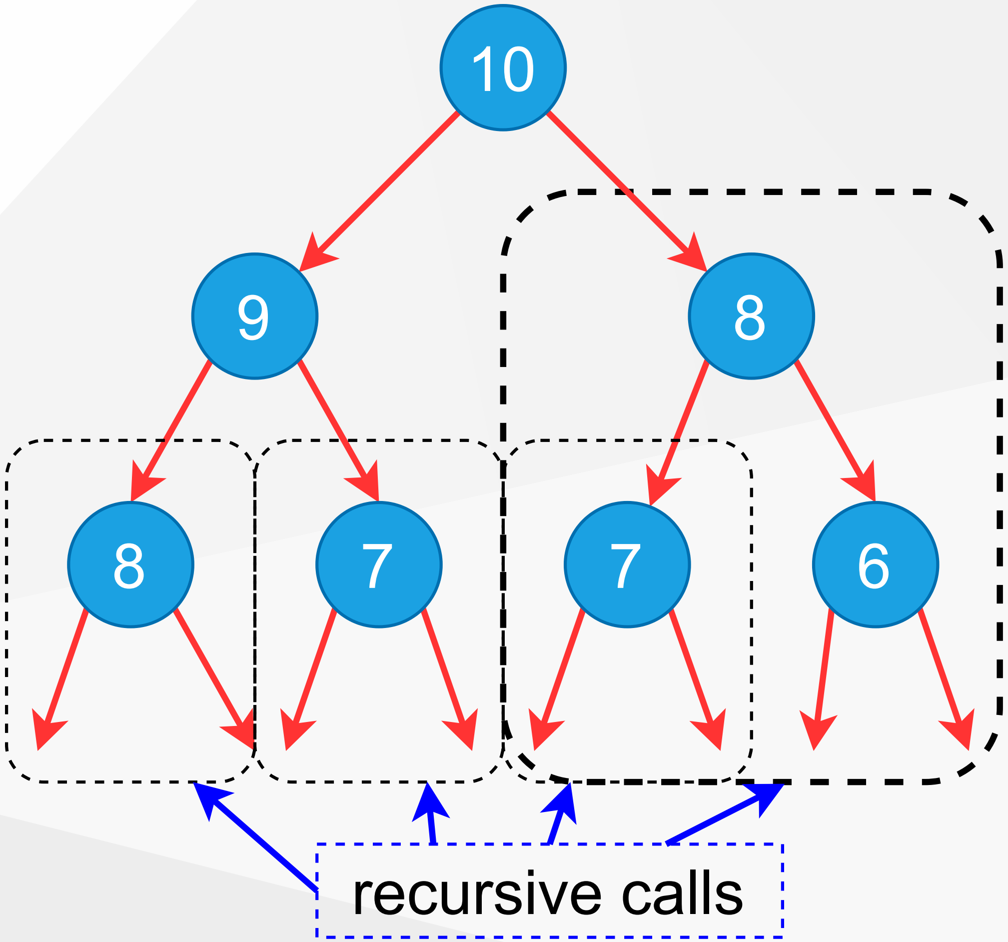
- An algorithm design paradigm like divide-and-conquer
- **Programming:** A tabular method (not writing computer code)
    - Older sense of planning or scheduling, typically by filling in a table
- **Divide-and-Conquer (DAC):** subproblems are independent
- **Dynamic Programming (DP):** subproblems are not independent
- Overlapping subproblems: subproblems share sub-subproblems
    - In solving problems with overlapping subproblems
        - A DAC algorithm **does redundant** work
            - Repeatedly solves common subproblems
        - A DP algorithm solves each problem just once
            - **Saves** its result **in a table**

Problem 1: Fibonacci Numbers
Recursive Solution

- **Reminder:**

$$F(0) = 0 \text{ and } F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

```
REC-FIBO(n)
  if n < 2
    return n
  else
    return REC-FIBO(n-1) + REC-FIBO(n-2)
```



- Overlapping subproblems in different recursive calls. Repeated work!

Problem 1: Fibonacci Numbers Recursive Solution

- **Recurrence:**
  - *exponential runtime*

$$T(n) = T(n-1) + T(n-2) + 1$$

- Recursive algorithm inefficient because it recomputes the same $F(i)$ repeatedly in different branches of the recursion tree.

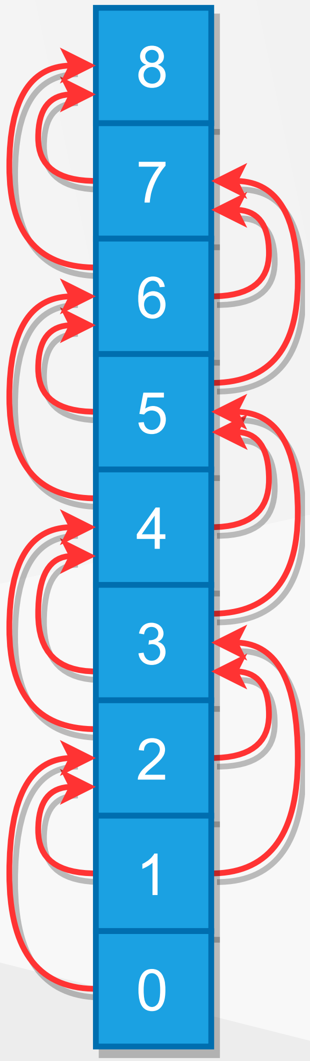
Problem 1: Fibonacci Numbers Bottom-up Computation

- Reminder:

$$F(0) = 0 \text{ and } F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

- Runtime $\Theta(n)$

```
ITER-FIBO(n)
  F[0] = 0
  F[1] = 1
  for i = 2 to n do
    F[i] = F[i-1] + F[i-2]
  return F[n]
```



Optimization Problems

- **DP** typically applied to optimization problems
- In an optimization problem
  - There are many possible solutions (feasible solutions)
  - Each solution has a value
  - Want to find an optimal solution to the problem
    - *A solution with the optimal value (min or max value)*
  - Wrong to say **the** optimal solution to the problem
    - *There may be several solutions with the same optimal value*

Development of a DP Algorithm

Step-1. Characterize the structure of an optimal solution

Step-2. Recursively define the value of an optimal solution

Step-3. Compute the value of an optimal solution in a bottom-up fashion

Step-4. Construct an optimal solution from the information computed in **Step 3**

Problem 2: Matric Chain Multiplication

- **Input:** a sequence (chain) $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices
- **Aim:** compute the product $A_1 \cdot A_2 \cdot \dots A_n$
- 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 a pair of parentheses.

$$\left(A_i(A_{i+1}A_{i+2}\dots A_j)\right)$$
$$\left((A_iA_{i+1}A_{i+2}\dots A_{j-1})A_j\right)$$
$$\left((A_iA_{i+1}A_{i+2}\dots A_k)(A_{k+1}A_{k+2}\dots A_j)\right) \text{ for } i \le k < j$$

- All parenthesizations yield the same product; matrix product is associative

Matrix-chain Multiplication: An Example Parenthesization

- **Input:** $\langle A_1, A_2, A_3, A_4 \rangle$ (5 distinct ways of full parenthesization)

$$\left(A_1\left(A_2(A_3A_4)\right)\right)$$

$$\left(A_1\left((A_2A_3)A_4\right)\right)$$

$$\left((A_1A_2)(A_3A_4)\right)$$

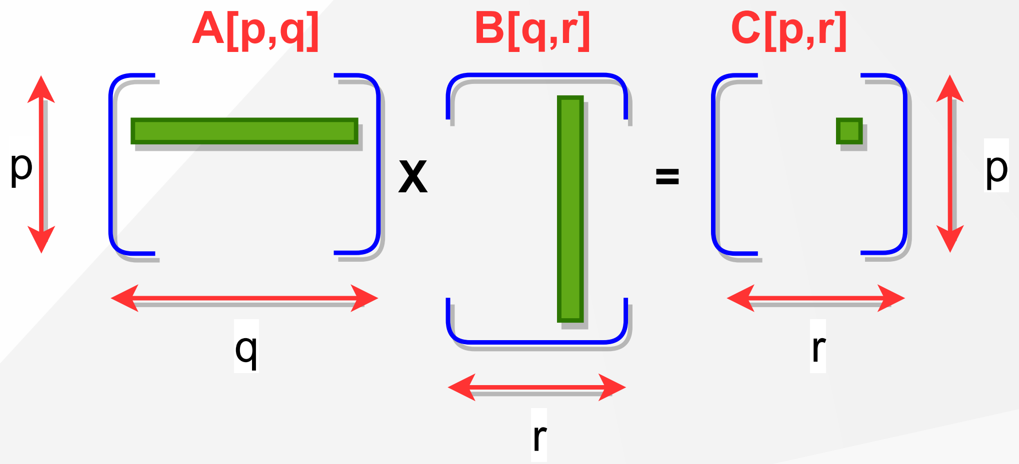
$$\left(\left(A_1(A_2A_3)A_4\right)\right)$$

$$\left(\left((A_1A_2)A_3\right)A_4\right)$$

- The way we parenthesize a chain of matrices can have a dramatic effect on the cost of computing the product

Matrix-chain Multiplication: Reminder

```
MATRIX-MULTIPLY(A, B)
  if cols[A]!=rows[B] then
    error("incompatible dimensions")
  for i=1 to rows[A] do
    for j=1 to cols[B] do
      C[i,j]=0
      for k=1 to cols[A] do
        C[i,j]=C[i,j]+A[i,k]·B[k,j]
  return C
```



rows(A)= p
cols(A) = q

rows(B)= q
cols(B) = r

rows(C)= p
cols(C) = r

Note : matrix[row,column]

A: p x q
B: q x r

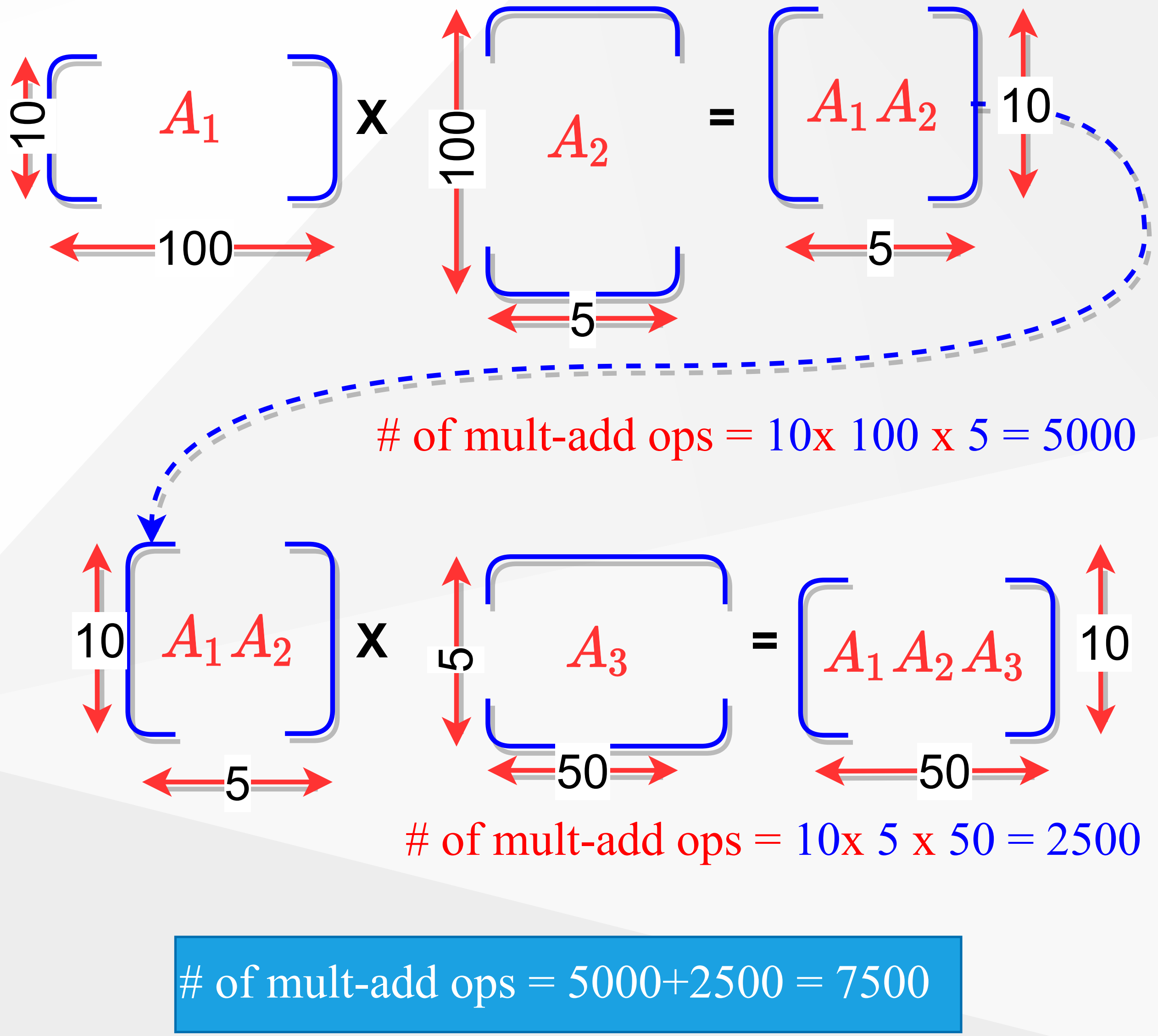
C:p x r

# of mult-add ops = rows[A]x cols[B] x cols[A]

# of mult-add ops = p x q x r

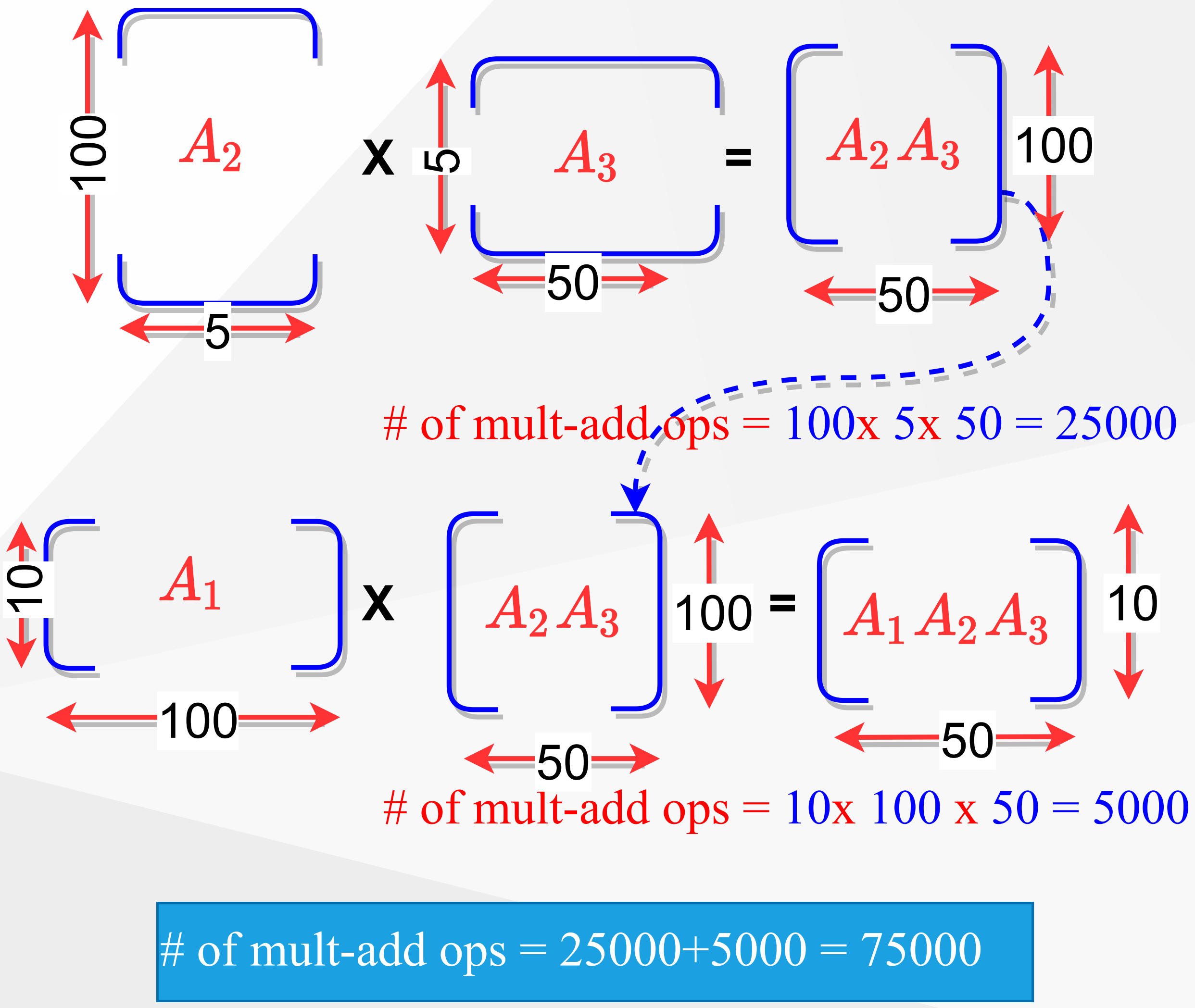
Matrix Chain Multiplication: Example

- $A1 : 10\text{x}100$, $A2 : 100\text{x}5$, $A3 : 5\text{x}50$
  - Which paranthesization is better? $(A1A2)A3$ or $A1(A2A3)$?



Matrix Chain Multiplication: Example

- $A1 : 10 \times 100$, $A2 : 100 \times 5$, $A3 : 5 \times 50$
  - Which paranthesization is better? $(A1A2)A3$ or $A1(A2A3)$?



Matrix Chain Multiplication: Example

- $A1 : 10 \times 100$, $A2 : 100 \times 5$, $A3 : 5 \times 50$
  - Which paranthesization is better? $(A1A2)A3$ or $A1(A2A3)$?

In summary:

- $(A1A2)A3$ = $\#$ of multiply-add ops: 7500
- $A1(A2A3)$ = $\#$ of multiple-add ops: 75000

First parenthesization yields **10x faster** computation

Matrix-chain Multiplication Problem

- **Input:** A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices,
  - where A_i is a $p_{i-1} \times p_i$ matrix
- **Objective:** Fully parenthesize the product
  - $A_1 \cdot A_2 \ldots A_n$
    - such that the number of **scalar mult-adds** is minimized.

Counting the Number of Parenthesizations

- **Brute force approach:** exhaustively check all parenthesizations
- $P(n)$: $\#$ of parenthesizations of a sequence of n matrices
- We can split sequence between k^{th} and $(k+1)^{st}$ matrices for any $k = 1, 2, \dots, n-1$, then parenthesize the two resulting sequences independently, i.e.,

$$(A_1 A_2 A_3 \dots A_k \overbrace{)(}^{break-point} A_{k+1} A_{k+2} \dots A_n)$$

- We obtain the recurrence

$$P(1) = 1 \text{ and } P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$$

Number of Parenthesizations:

- $P(1) = 1$ and $P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$
- The recurrence generates the sequence of **Catalan Numbers** Solution is $P(n) = C(n-1)$ where

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$$

- The number of solutions is **exponential** in n
- Therefore, brute force approach is a poor strategy

The Structure of Optimal Parenthesization

- **Notation:** $A_{i..j}$: The matrix that results from evaluation of the product: $A_i A_{i+1} A_{i+2} \ldots A_j$
- **Observation:** Consider the last multiplication operation in any parenthesization: $(A_1 A_2 \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_n)$
  - There is a k value $(1 \leq k < n)$ such that:
    - First, the product $A_1 \ldots k$ is computed
    - Then, the product $A_{k+1 \ldots n}$ is computed
    - Finally, the matrices $A_{1 \ldots k}$ and $A_{k+1 \ldots n}$ are multiplied

Step 1: Characterize the Structure of an Optimal Solution

- An optimal parenthesization of product $A_1 A_2 \ldots A_n$ will be:
  $(A_1 A_2 \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_n)$ for some k value
- The **cost of this optimal parenthesization** will be:
  $=$ Cost of computing $A_{1 \ldots k}$
  $+$ Cost of computing $A_{k+1 \ldots n}$
  $+$ Cost of multiplying $A_{1 \ldots k} \cdot A_{k+1 \ldots n}$

Step 1: Characterize the Structure of an Optimal Solution

- **Key observation:** Given optimal parenthesization
  - $(A_1 A_2 A_3 \dots A_k) \cdot (A_{k+1} A_{k+2} \dots A_n)$
- Parenthesization of the subchain $A_1 A_2 A_3 \dots A_k$
- Parenthesization of the subchain $A_{k+1} A_{k+2} \dots A_n$

should both be optimal

- Thus, optimal solution to an instance of the problem contains optimal solutions to subproblem instances
  - **i.e.**, optimal substructure within an optimal solution exists.

Step 2: A Recursive Solution

- **Step 2:** Define the value of an optimal solution recursively in terms of optimal solutions to the subproblems
- Assume we are trying to determine the min cost of computing $A_{i...j}$
- $m_{i,j}$: min $\#$ of scalar multiply-add opns needed to compute $A_{i...j}$
  - **Note:** *The optimal cost of the original problem:* $m_{1,n}$
- How to compute $m_{i,j}$ recursively?

Step 2: A Recursive Solution

- Base case: $m_{i,i} = 0$ (single matrix, no multiplication)
- Let the size of matrix A_i be $(p_{i-1} \times p_i)$
- Consider an optimal parenthesization of chain
  - $A_i \dots A_j : (A_i \dots A_k) \cdot (A_{k+1} \dots A_j)$
- The optimal cost: $m_{i,j} = m_{i,k} + m_{k+1,j} + p_{i-1} \times p_k \times p_j$
- **where:**
  - $m_{i,k}$: Optimal cost of computing $A_{i \dots k}$
  - $m_{k+1,j}$: Optimal cost of computing $A_{k+1 \dots j}$
  - $p_{i-1} \times p_k \times p_j$: Cost of multiplying $A_{i \dots k}$ and $A_{k+1 \dots j}$

Step 2: A Recursive Solution

- In an optimal parenthesization: k must be chosen to minimize m_{ij}
- The recursive formulation for m_{ij}:

$$m_{ij} = \begin{cases} 0 & if \;\; i = j \\ \underset{i \leq k < j}{MIN} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \} & if \;\; i < j \end{cases}$$

Step 2: A Recursive Solution

- The m_{ij} values give the **costs of optimal solutions** to subproblems
- In order to keep track of how to construct an optimal solution
  - Define s_{ij} to be the value of k which yields the optimal split of the subchain $A_{i \dots j}$
    - That is, $s_{ij} = k$ such that
      - $m_{ij} = m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j$ holds

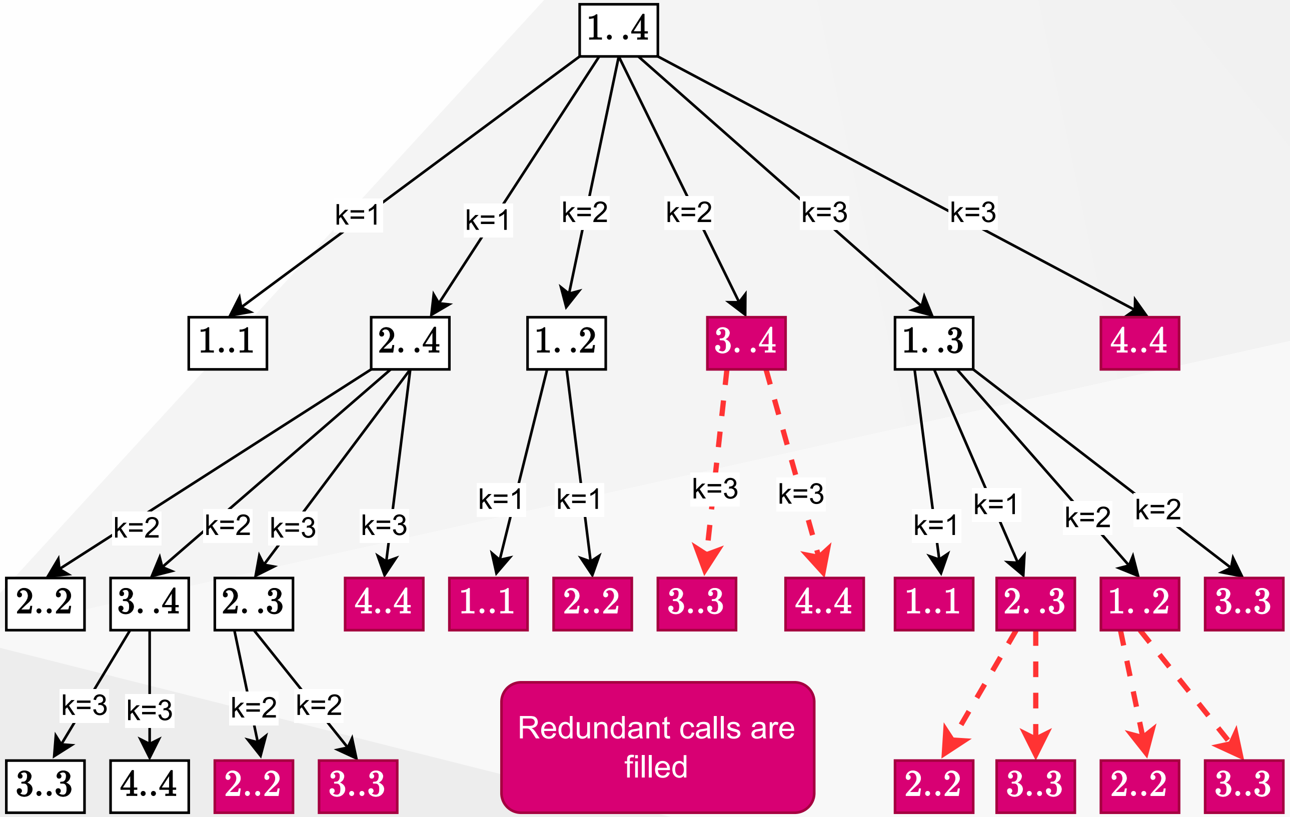
Direct Recursion: Inefficient!

- Recursive Matrix-Chain (**RMC**) Order

```
RMC(p,i,j)

  if (i == j) then
    return 0

  m[i, j] = INF

  for k=i to j-1 do

    q = RMC(p, i, k) + RMC(p, k+1, j) + p_{i-1} p_k p_j

    if q < m[i, j] then
      m[i, j] = q

  endfor

        return m[i, j]
```

Direct Recursion: Inefficient!

- Recursion tree for $RMC(p, 1, 4)$
- Nodes are labeled with i and j values



Computing the Optimal Cost (Matrix-Chain Multiplication)

An important observation:

- We have **relatively few subproblems**
  - one problem for each choice of i and j satisfying $1 \leq i \leq j \leq n$
  - total $n + (n-1) + \cdots + 2 + 1 = \frac{1}{2}n(n+1) = \Theta(n2)$ subproblems
- We can write a **recursive** algorithm based on recurrence.
- However, a recursive algorithm may encounter each subproblem many times in different branches of the recursion tree
- This property, **overlapping subproblems**, is the second important feature for applicability of **dynamic programming**

Computing the Optimal Cost (Matrix-Chain Multiplication)

- Compute the value of an optimal solution in a **bottom-up** fashion
  - matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \dots, n$
  - the input is a sequence $\langle p_0, p_1, \dots, p_n \rangle$ where $length[p] = n + 1$
- Procedure uses the following auxiliary tables:
  - $m[1 \dots n, 1 \dots n]$: for storing the $m[i, j]$ costs
  - $s[1 \dots n, 1 \dots n]$: records which index of k achieved the optimal cost in computing $m[i, j]$

Bottom-Up Computation

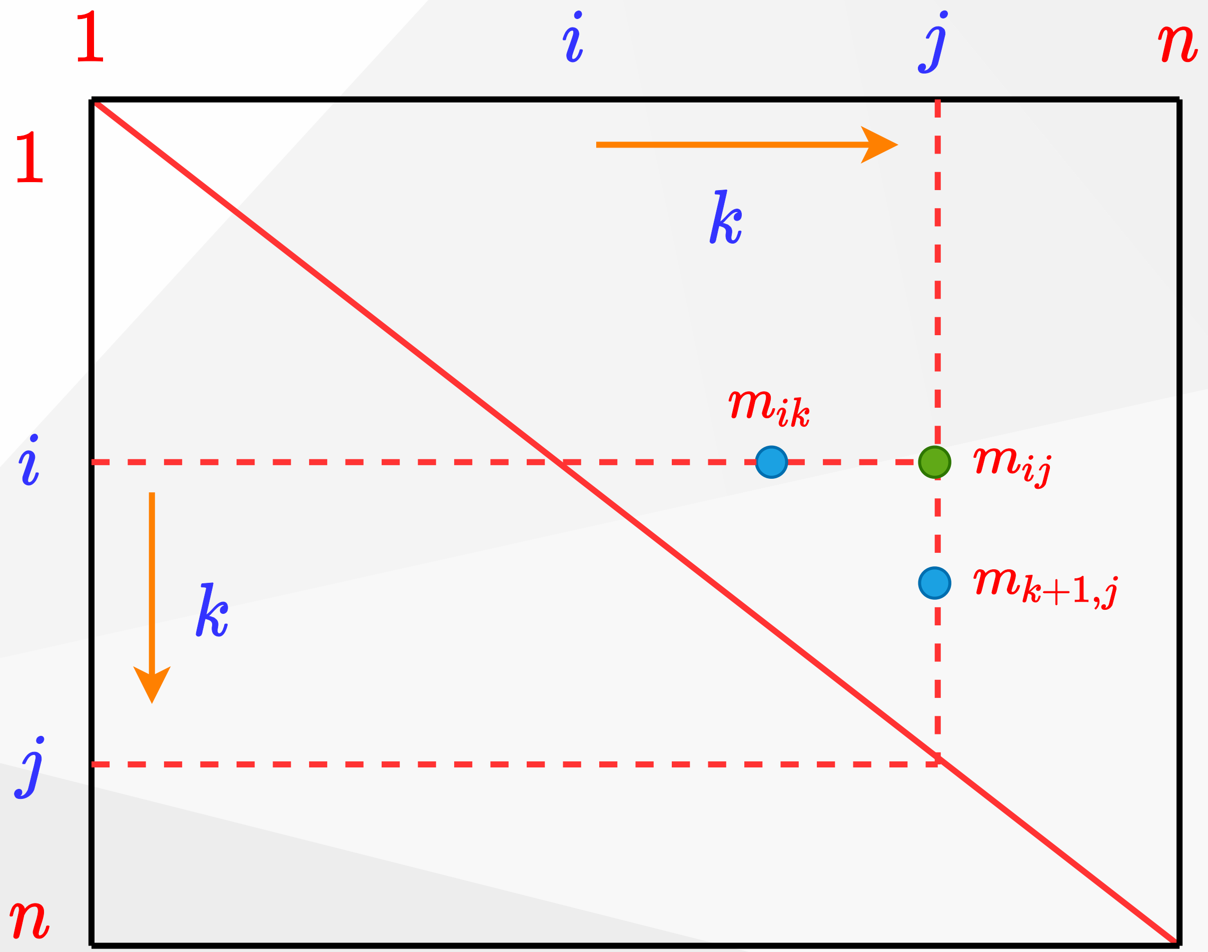
- How to choose the order in which we process m_{ij} values?
- Before computing m_{ij}, we have to make sure that the values for m_{ik} and $m_{k+1,j}$ have been computed for all k.

$$m_{ij} = \underset{i \leq k < j}{MIN} \{m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j\}$$

Bottom-Up Computation

$$m_{ij} = \underset{i \leq k < j}{MIN}\{m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j\}$$

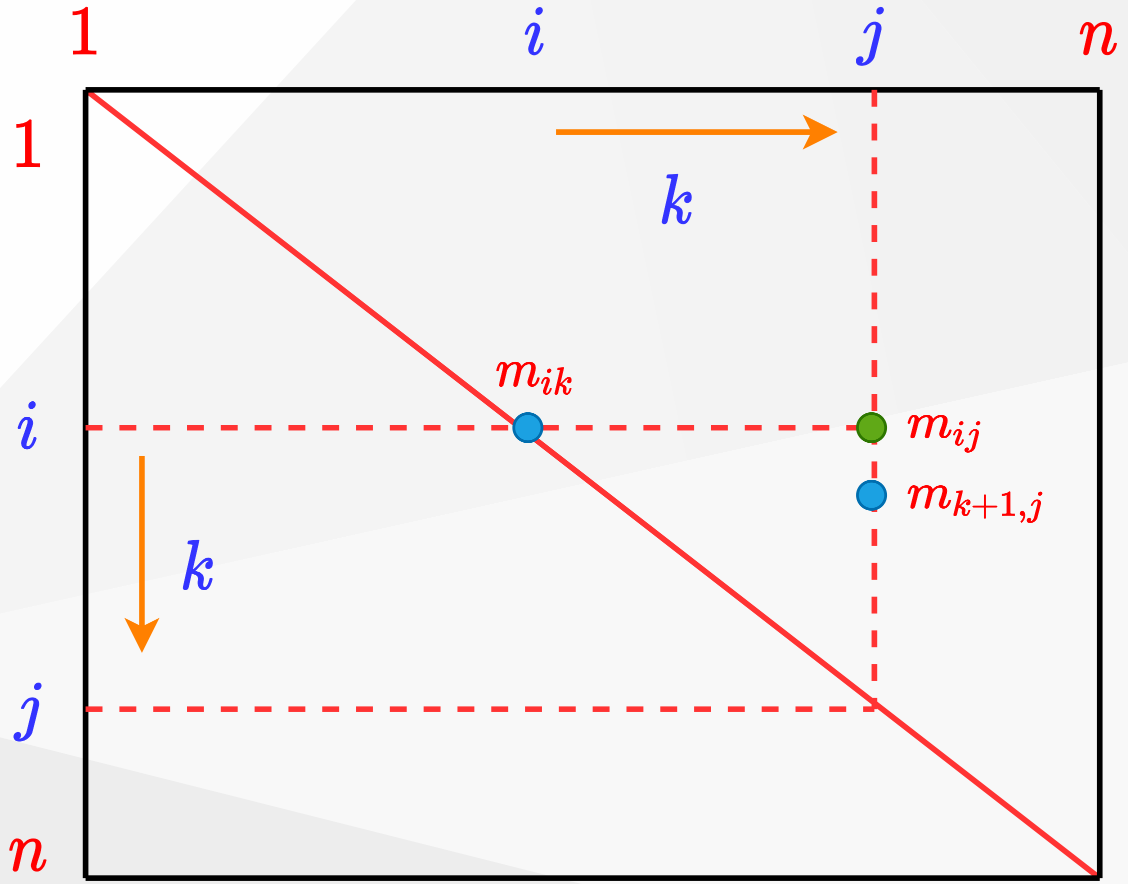
- m_{ij} must be processed after m_{ik} and $m_{j,k+1}$
- **Reminder:** m_{ij} computed only for $j > i$



Bottom-Up Computation

$$m_{ij} = \underset{i \le k < j}{MIN}\{m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j\}$$

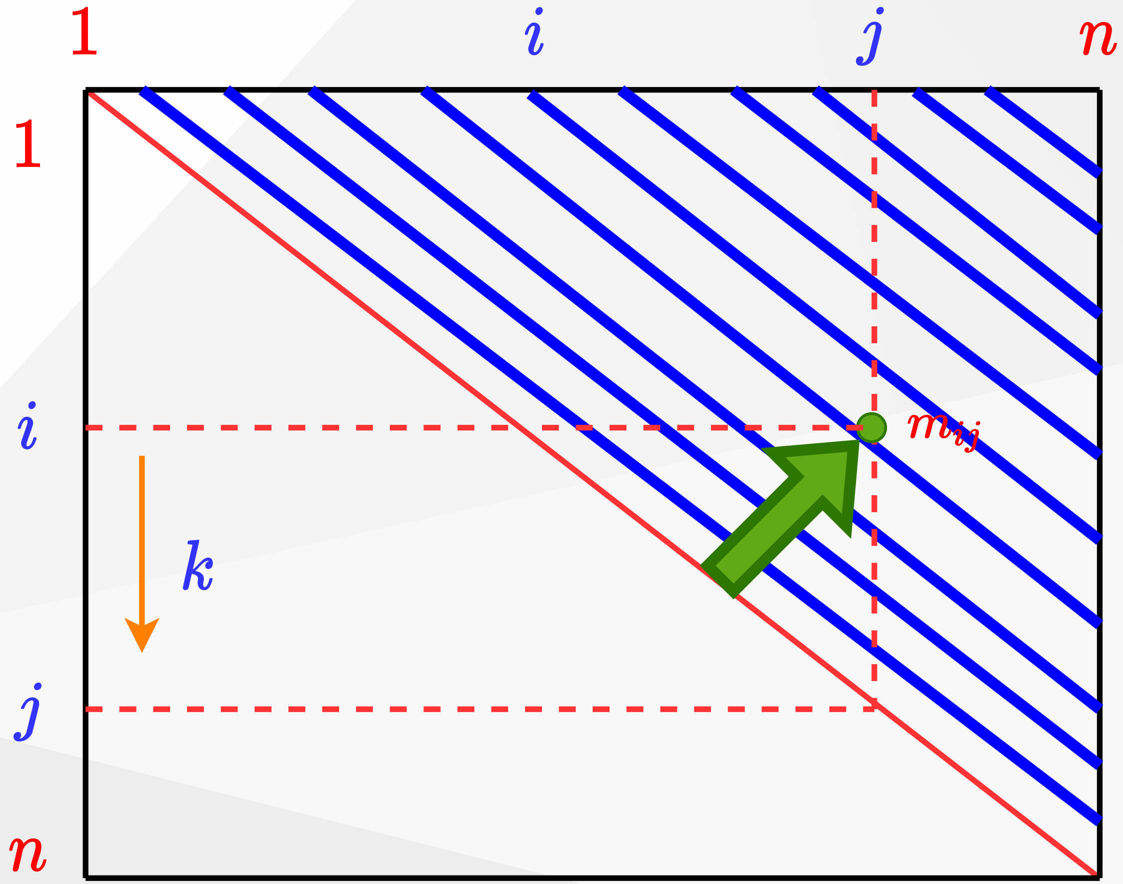
- m_{ij} must be processed after m_{ik} and $m_{j,k+1}$
- How to set up the iterations over i and j to compute m_{ij}?



Bottom-Up Computation

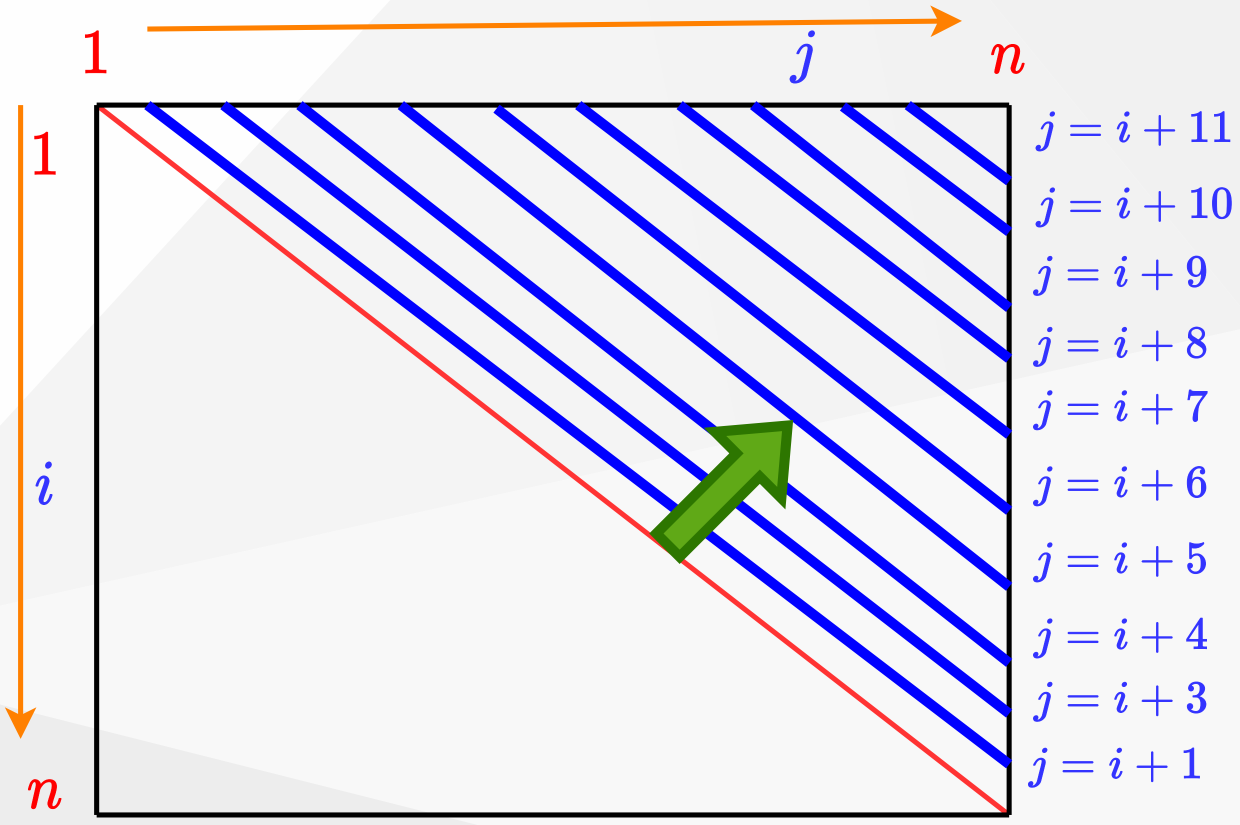
$$m_{ij} = \underset{i \leq k < j}{MIN}\{m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j\}$$

- If the entries m_{ij} are computed in the shown order, then m_{ik} and $m_{k+1,j}$ values are guaranteed to be computed before m_{ij}.



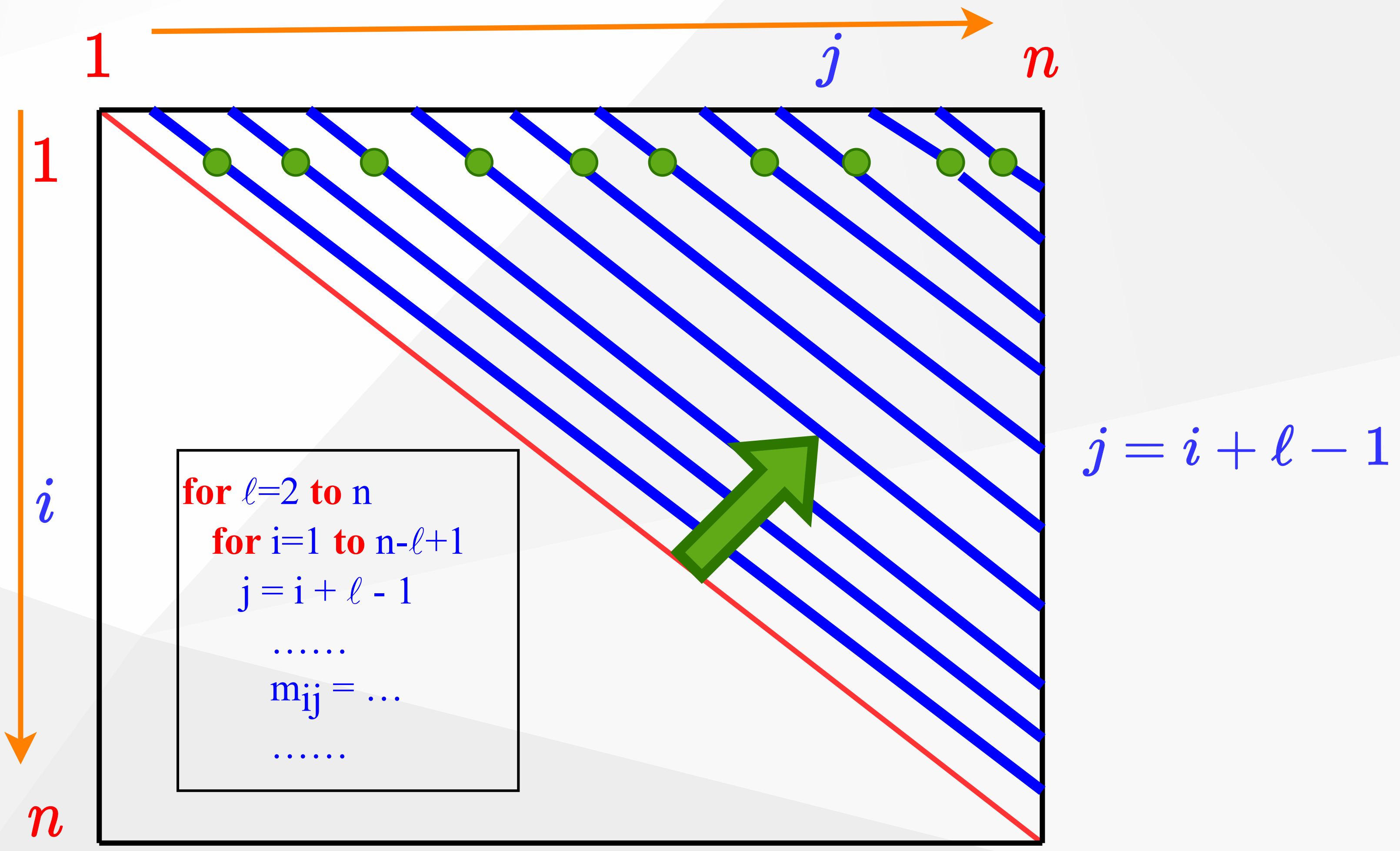
Bottom-Up Computation

$$m_{ij} = \underset{i \le k < j}{MIN} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \}$$



Bottom-Up Computation

$$m_{ij} = \underset{i \le k < j}{MIN}\{m_{ik} + m_{k+1,j} + p_{i-1}p_k p_j\}$$



Algorithm for Computing the Optimal Costs

- *Note*: l= ℓ and p_{i-1} p_k p_j $= p_{i-1}p_kp_j$

```
MATRIX-CHAIN-ORDER(p)
  n = length[p]-1
  for i=1 to n do
    m[i, i]=0
  endfor
  for l=2 to n do
    for i=1 to n n-l+1 do
      j=i+l-1
      m[i, j]=INF
      for k=i to j-1 do
        q=m[i,k]+m[k+1, j]+p_{i-1} p_k p_j
        if q < m[i,j] then
          m[i,j]=q
          s[i,j]=k
      endfor
    endfor
  endfor
  return m and s
```

Algorithm for Computing the Optimal Costs

- The algorithm first computes
  - $m[i, i] \leftarrow 0$ for $i = 1, 2, \ldots, n$ min costs for all chains of length 1
- **Then**, for $\ell = 2, 3, \ldots, n$ computes
  - $m[i, i + \ell - 1]$ for $i = 1, \ldots, n - \ell + 1$ min costs for all chains of length ℓ
- For each value of $\ell = 2, 3, \ldots, n$,
  - $m[i, i + \ell - 1]$ depends only on table entries $m[i, k] \& m[k + 1, i + \ell - 1]$ for $i \leq k < i + \ell - 1$, which are already computed

Algorithm for Computing the Optimal Costs

$$\begin{matrix} \text{compute } m[i,i+1] \\ \underbrace{\{m[1,2], m[2,3], \ldots, m[n-1,n]\}}_{(n-1)\text{ values}} \end{matrix} \left\{ \begin{array}{l} \ell = 2 \\ \text{for } i = 1 \text{ to } n-1 \text{ do} \\ \quad m[i,i+1] = \infty \\ \quad\quad \text{for } k = i \text{ to } i \text{ do} \\ \quad\quad\quad \vdots \end{array} \right.$$

$$\begin{matrix} \text{compute } m[i,i+2] \\ \underbrace{\{m[1,3], m[2,4], \ldots, m[n-2,n]\}}_{(n-2)\text{ values}} \end{matrix} \left\{ \begin{array}{l} \ell = 3 \\ \text{for } i = 1 \text{ to } n-2 \text{ do} \\ \quad m[i,i+2] = \infty \\ \quad\quad \text{for } k = i \text{ to } i+1 \text{ do} \\ \quad\quad\quad \vdots \end{array} \right.$$

$$\begin{matrix} \text{compute } m[i,i+3] \\ \underbrace{\{m[1,4], m[2,5], \ldots, m[n-3,n]\}}_{(n-3)\text{ values}} \end{matrix} \left\{ \begin{array}{l} \ell = 4 \\ \text{for } i = 1 \text{ to } n-3 \text{ do} \\ \quad m[i,i+3] = \infty \\ \quad\quad \text{for } k = i \text{ to } i+2 \text{ do} \\ \quad\quad\quad \vdots \end{array} \right.$$

Table access pattern in computing $m[i, j]$s for $\ell = j - i + 1$

for $k \leftarrow i$ to $j - 1$ do

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

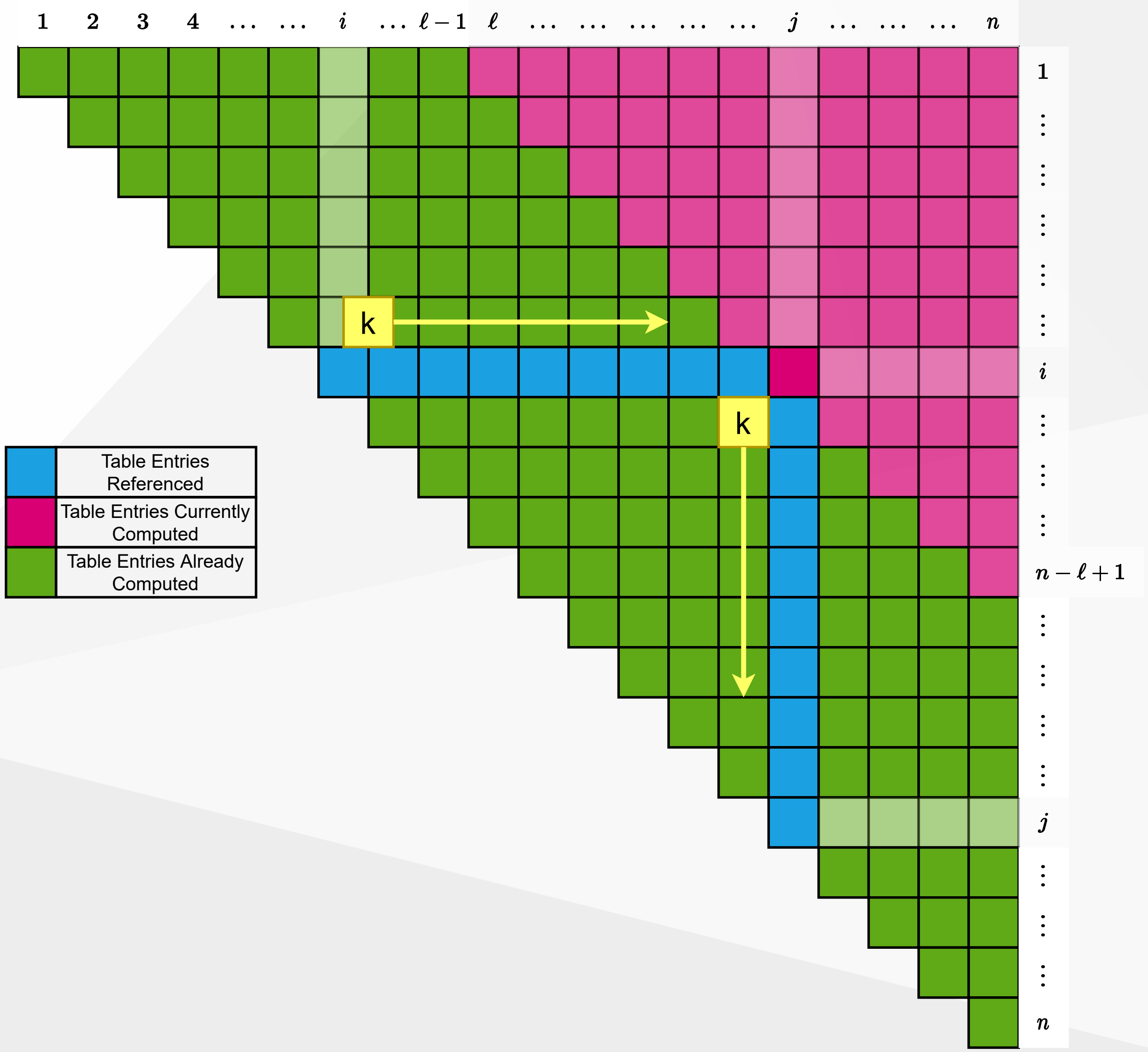


Table access pattern in computing $m[i,j]$s for $\ell = j - i + 1$

$$((A_i) \overbrace{\vdots}^{mult.} (A_{i+1}A_{i+2}\dots A_j))$$

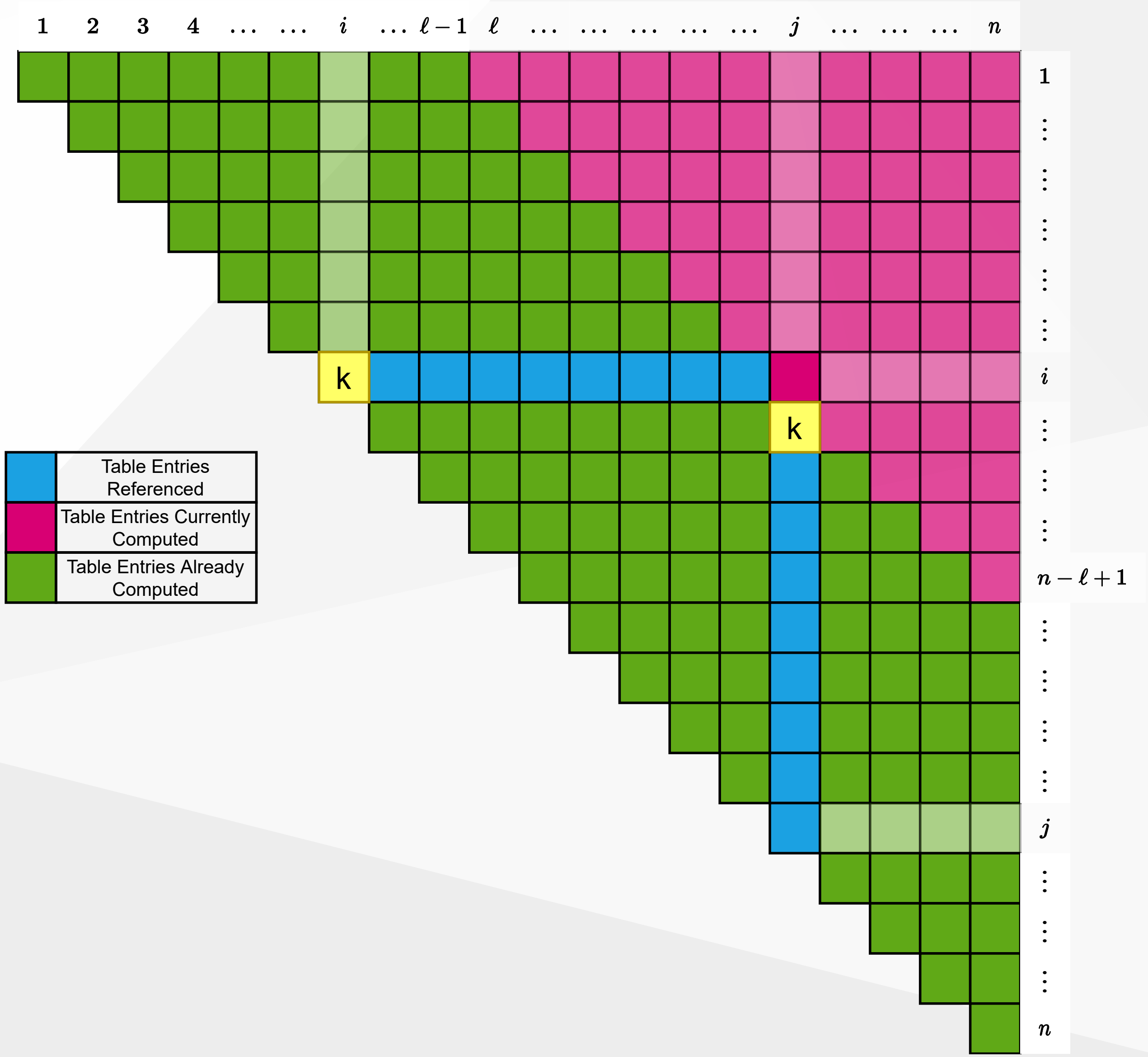


Table access pattern in computing $m[i, j]$s for $\ell = j - i + 1$

$$((A_i A_{i+1}) \overbrace{\vdots}^{mult.} (A_{i+2} \ldots A_j))$$

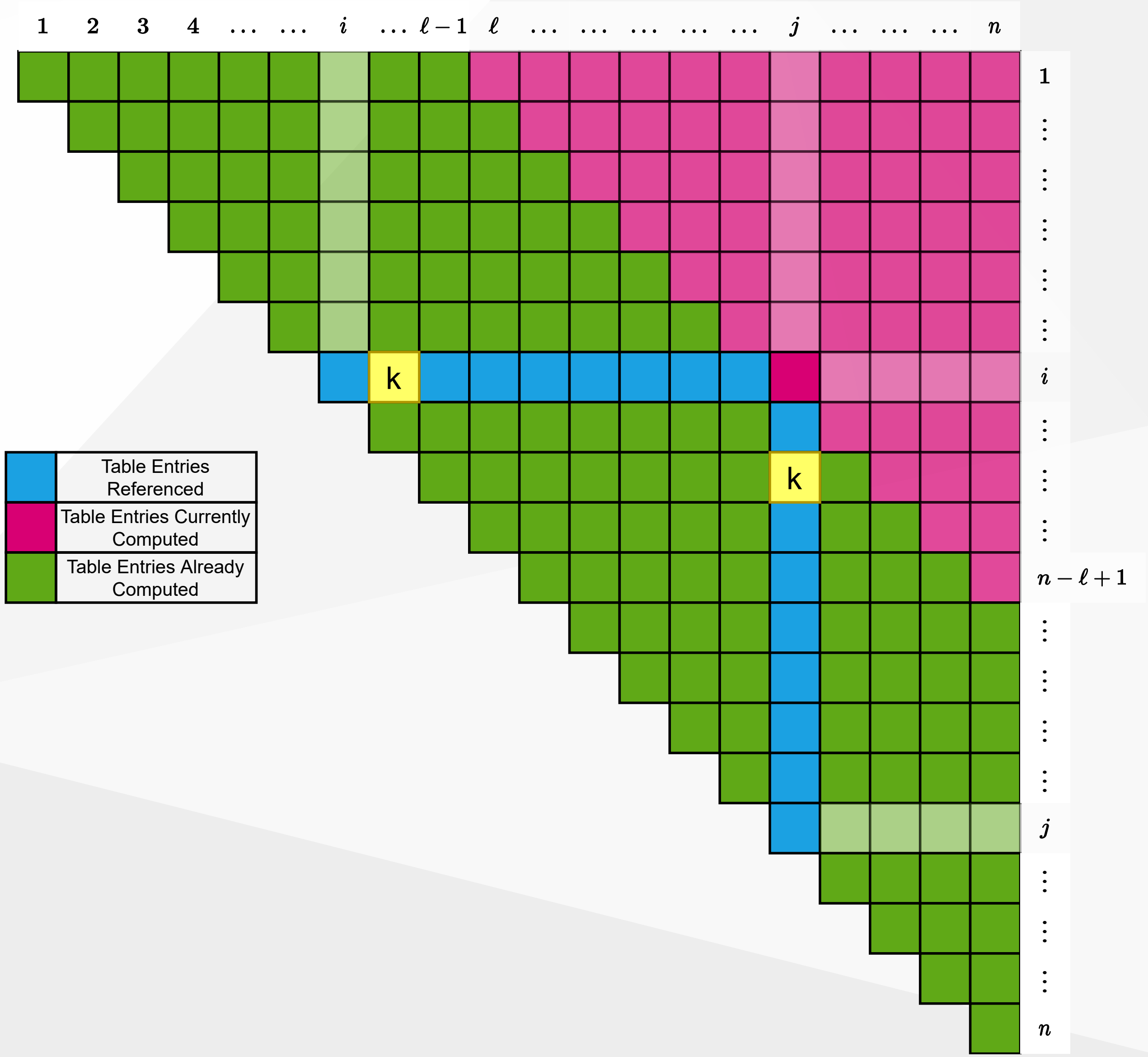


Table access pattern in computing $m[i, j]$s for $\ell = j - i + 1$

$$((A_i A_{i+1} A_{i+2}) \overbrace{\vdots}^{mult.} (A_{i+3} \ldots A_j))$$

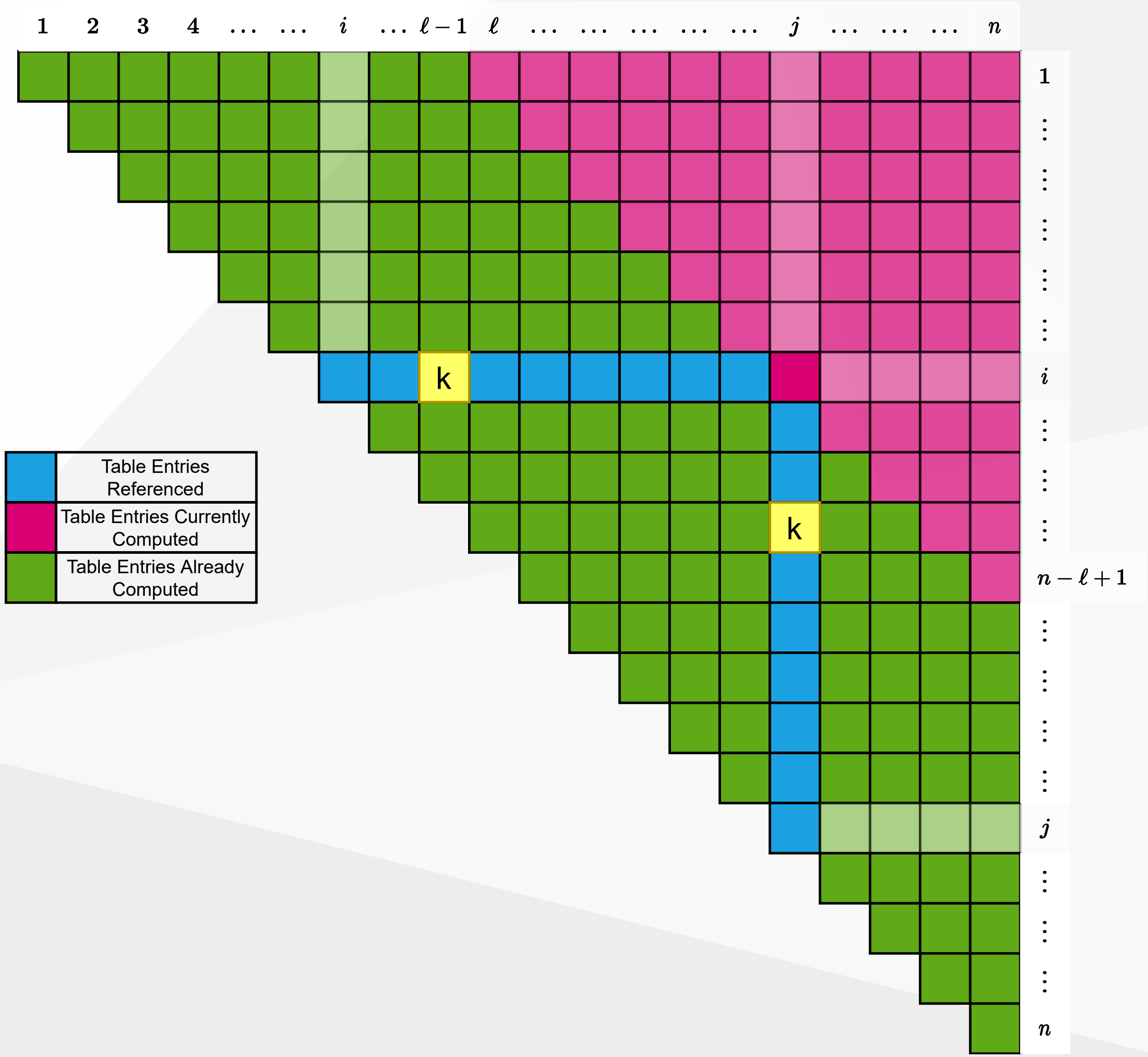
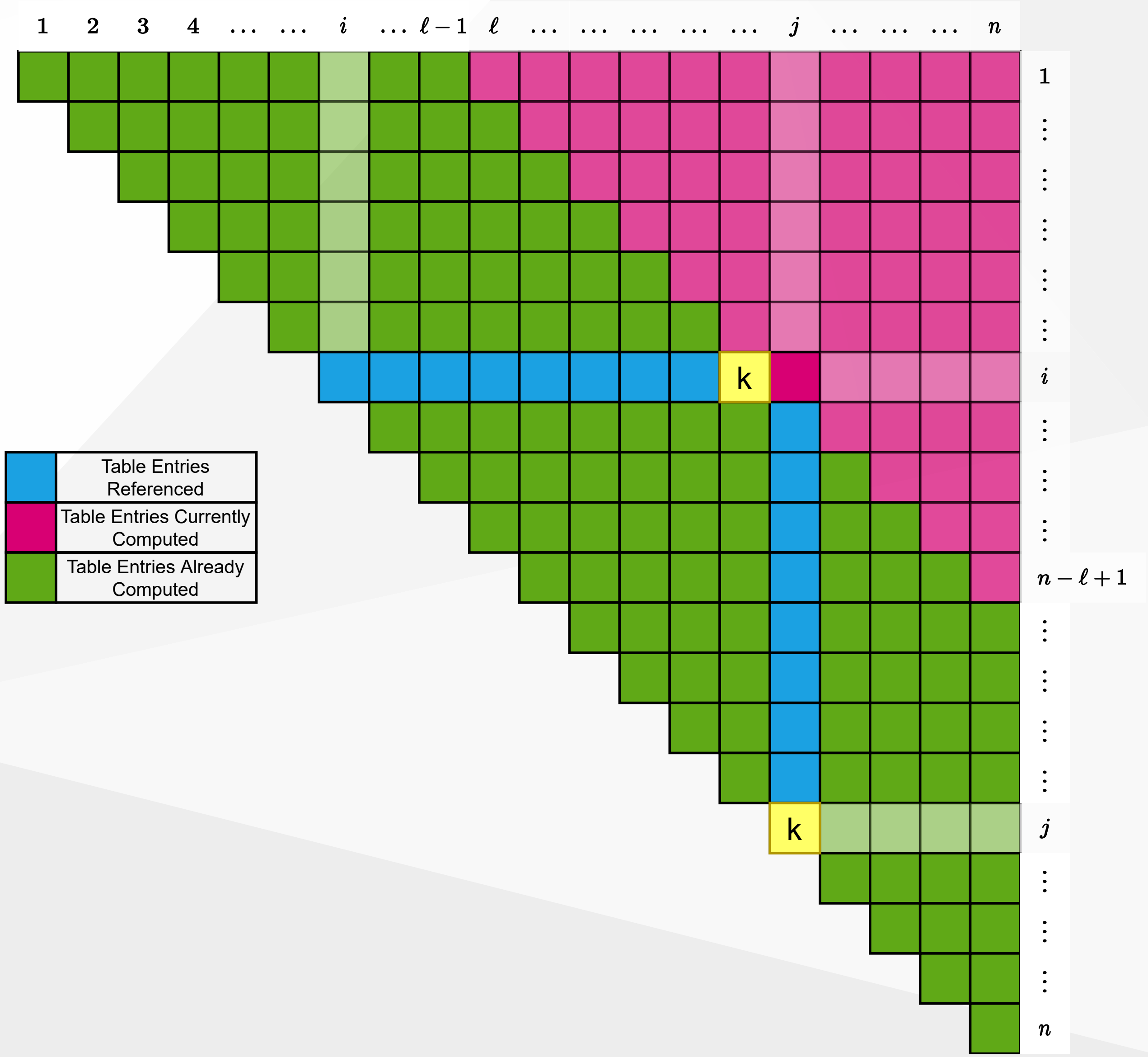


Table access pattern in computing $m[i, j]$s for $\ell = j - i + 1$

$$((A_i A_{i+1} \dots A_{j-1}) \overbrace{\vdots}^{mult.} (A_j))$$



References

$$-End - Of - Week - 5 - Course - Module-$$