Dynamic Programming

Matrix Chain Multiplication

Introduction

- Greedy problem solving technique, dynamic programming and divide and conquer are all similar in that they try to solve the problem by dividing it into smaller sub-problems and solving the sub-problems in order to get the solution of the original problem
- Dynamic programming and greedy techniques are used in solving **optimization problems**
- Unlike divide and conquer, the dynamic programming technique is useful when the subproblems overlap
- The idea in dynamic programming is to solve the overlapping subproblems only once and save the result which can be *looked up* in future rather than solving the same problem again
- When subproblems do not overlap, dynamic programming may not be a good approach

When to apply Dynamic Programming

- Optimization Problem – the problem has several solutions including potentially more than one optimal solution and the task is to find **an** optimal solution
- Optimal substructure – the solution to the optimization problem involves solving the generated subproblems optimally (and the subproblems can be optimized independently).
- Overlapping Subproblems – the space of the generated subproblems is **small** in the sense that many subproblems overlap
- It is an example of a technique where we can increase space complexity to reduce time complexity i.e. there exists **space-time trade-off**

Steps involved in Dynamic Programming

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution.
- Construct an optimal solution from computed information.

Matrix-Chain Multiplication

- We are given a sequence (chain) $< A_1, A_2, \ldots A_n >$ of n matrices to be multiplied, and we wish to compute the product
$$A_1 A_2 \ldots A_n$$
- The number of computations required for calculating the above product might change dramatically with the order of computation of products
- The computation problem in Matrix-Chain multiplication is to find an optimal order of multiplication for the n matrices so that the number of computations involved is minimum

Example

- Suppose, A1, A2 and A3 are three matrices having dimensions (10x100), (100x5) and (5x50) respectively
- ((A1,A2)A3) – (10x100x5) + (10x5x50) = 7500 scalar multiplications
- (A1(A2,A3)) – (10x100x50)+(100x5x50) = 50000+25000 = 75000 scalar multiplications
- The problem in matrix chain multiplication is not actually calculating the matrix product but finding an optimal sequence of matrices for calculating the product so that the number of scalar multiplications is minimum

Brute Force Approach

- This would involve enumerating all possible sequences of multiplication for n matrices and calculating the cost for each
- A minimum cost sequence will comprise the solution to the problem
- Multiplication of a sequence of n matrices can be obtained by multiplying two subsequences of size k and n-k each where we put a parenthesis between kth and (k+1)th matrices
- The two sequences can in turn be evaluated by again putting the parenthesis between any two matrices in the sequence
- Thus, we get a recurrence relation for solving the problem

$$P(n) = \begin{cases} 1, & if\ n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k), & if\ n \geq 2 \end{cases}$$

- Using an inductive proof assuming $P(n) \geq 2^n - 1 \Rightarrow \Omega(2^n)$
- We can use substitution method to show that $P(n) = \Omega(2^n)$
- Thus, the brute-force approach yields an algorithm that grows exponentially with the problem size n

Applying Dynamic Programming

- We note that once we have generated two parenthesized sequences of k and n-k lengths
- We must solve each of them optimally in order to get the optimal solution to the problem
- If it were not so then there would be some other sequence within them which could give even smaller number of multiplications and therefore, a solution better than what we have got which contradicts our assumption of finding the optimal sequence
- Therefore, we must solve both the sub problems optimally in order to get optimal solution to the original problem
- Further, both the subproblems are independent as the optimal sequence of the first k matrix products has nothing in common with optimal sequence of remaining n-k products

- We further note that for each possible value of the initial division performed at k, we get two subsequences, each of which has many subsequences that are common for different values of k
- Thus, while enumerating the possible sequences for multiplication recursively we are generating subproblems that overlap considerably
- Thus, the space of subproblems is not very large and we are actually generating same subproblems over and over again
- Therefore, the **optimal substructure and overlapping subproblems**, two important guiding properties for a dynamic programming based solution, both hold true for this problem
- Therefore, we can look for a dynamic programming based solution where we only solve a subproblem once and use its result every time the same subproblem is generated

DP based solution

- We solve the problem step-by-step as given in slide 4
- In the first step we assume that the optimal division takes place at the kth matrix (characterizing the optimal solution)
- The two subsequences generated after this step must also give optimal solution for generating the optimal solution to original problem (recursively defining optimal solution)
- We will then compute the value of the optimal solution
- The fourth step might not always be required, if required some extra information may need to be stored for constructing the solution

- So, now we can characterize the optimal solution for a sequence from i to j as follows:

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

Here, m[i,j] denotes the total number of scalar multiplications required for a sequence of length j-i+1 and each matrix Ai has dimension $p_{i-1}p_i$

Above is a recursive equation and we need to solve the subproblems to get the optimal solution

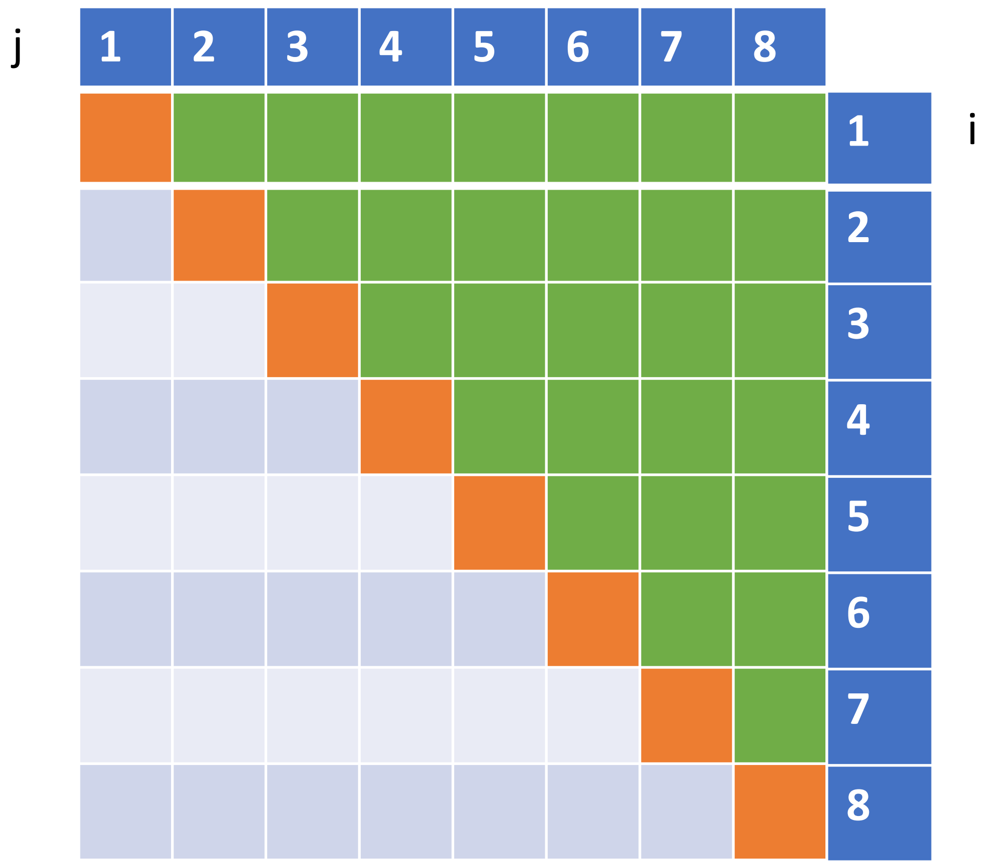
The above process is repeated for all possible values of k

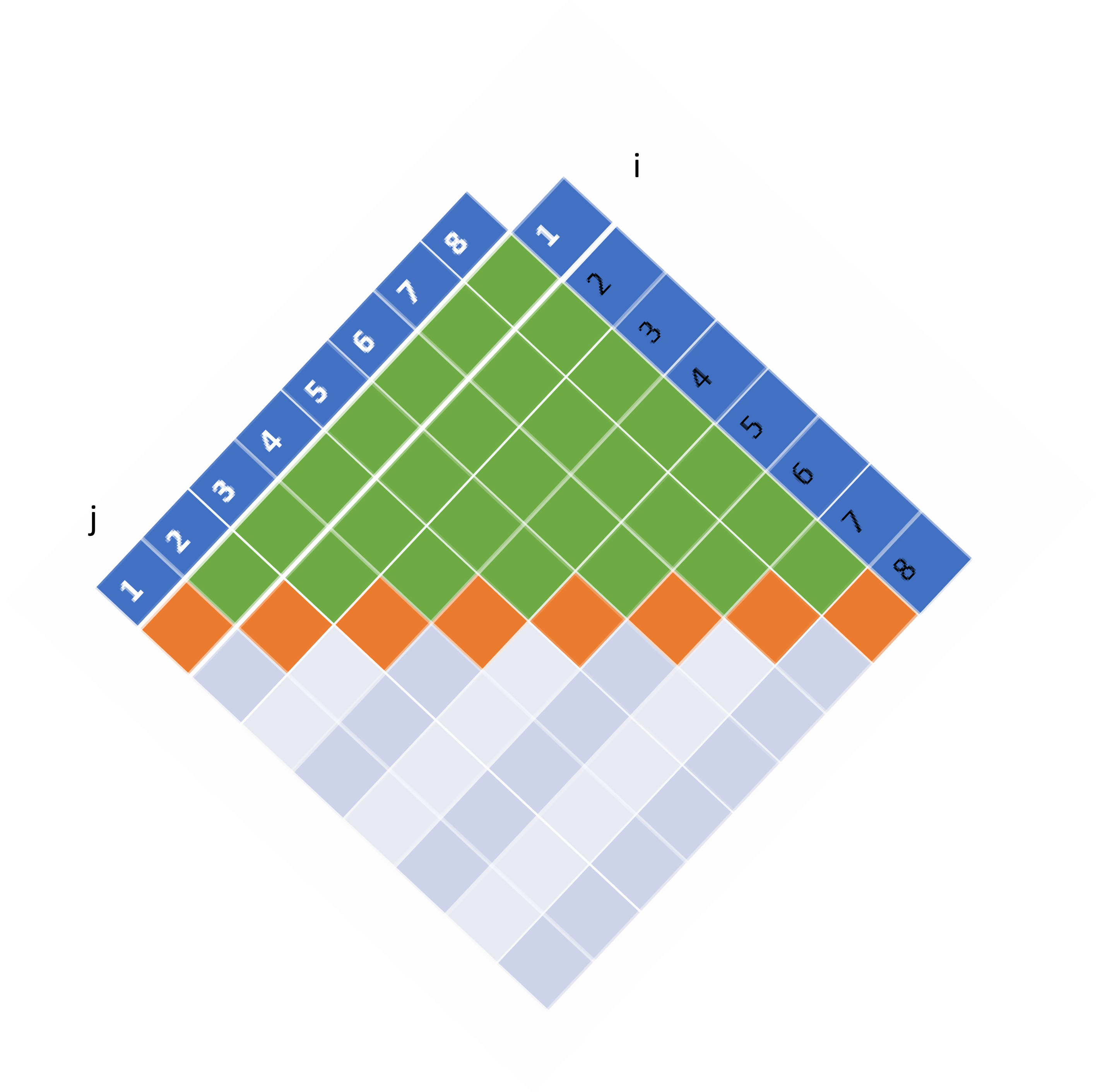
Thus the recursive equation becomes:

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

- The above equation will be solved in a bottom-up manner to get the optimal sequence
- Each matrix A_i has dimension $p_{i-1}p_i$
- The method takes as input a sequence $< p_o , p_1 , \ldots p_n >$ of length n+1 and uses the matrix m[i,j] to store the optimal cost for the sequence i to j
- It also uses an auxiliary matrix s[i,j] to store the index of the optimal split

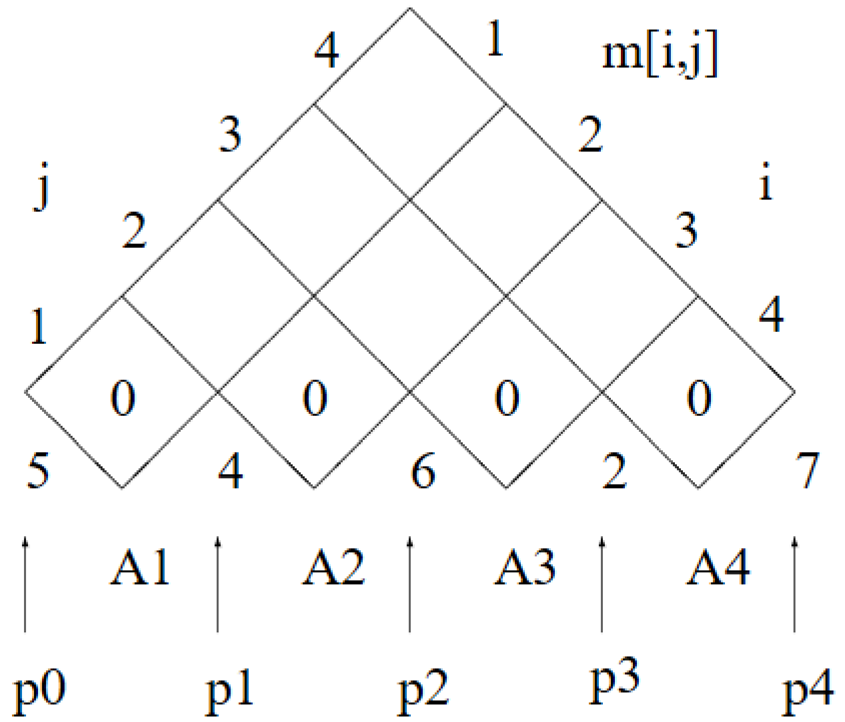
```
MATRIX-CHAIN-ORDER($p$)
$n = p.length - 1$
let $m[1 \mathinner{..} n, 1 \mathinner{..} n]$ and $s[1 \mathinner{..} n-1, 2 \mathinner{..} n]$ be new tables
for $i = 1$ to $n$
    $m[i,i] = 0$
for $l = 2$ to $n$                    // $l$ is the chain length
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $m[i,j] = \infty$
        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 $m$ and $s$
```


j
1
2
3
4
5
6
7
8
i
1
2
3
4
5
6
7
8


i
j
1
2
3
4
5
6
7
8
1
2
3
4
5
6
7
8

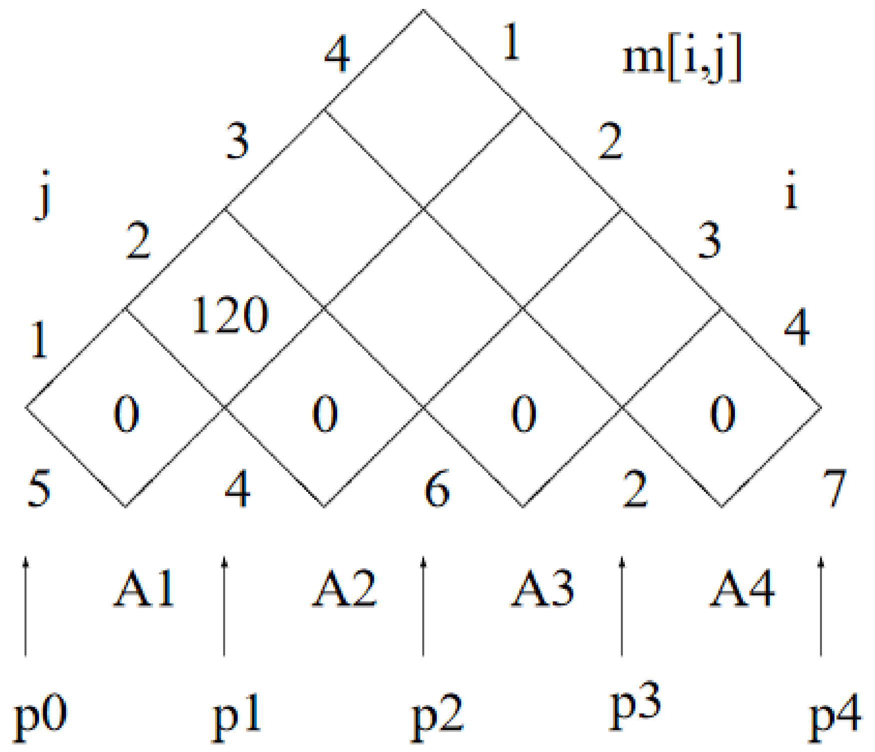
Example: Given a chain of four matrices A_1, A_2, A_3 and A_4, with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$. Find $m[1,4]$.

S0: Initialization



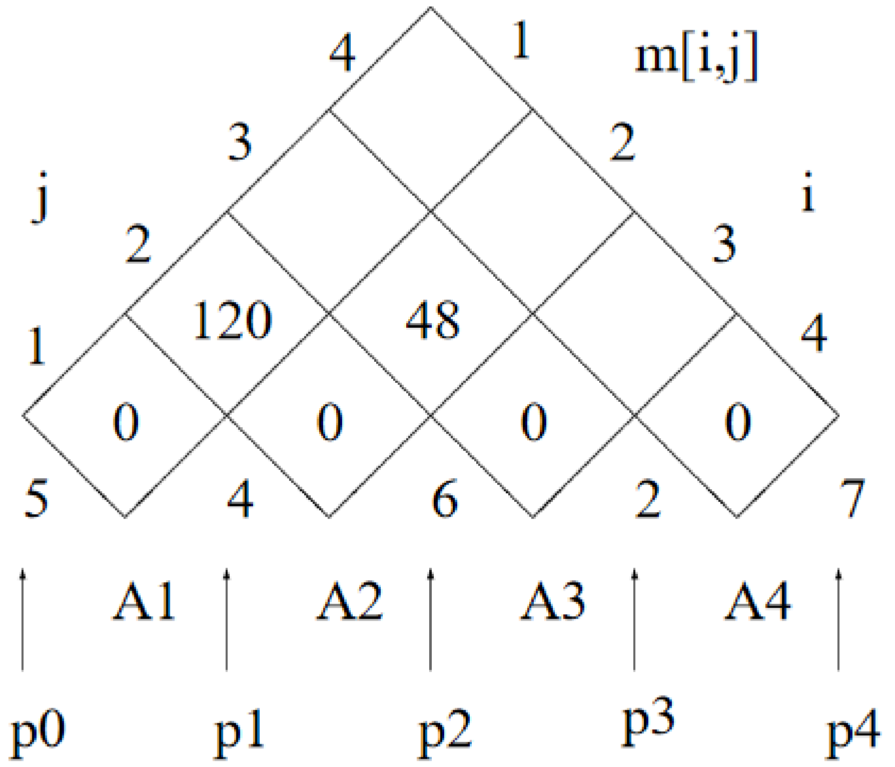
Stp 1: Computing $m[1,2]$ By definition

$$\begin{aligned} m[1,2] &= \min_{1\le k<2}(m[1,k]+m[k+1,2]+p_0p_kp_2) \\ &= m[1,1]+m[2,2]+p_0p_1p_2 = 120. \end{aligned}$$



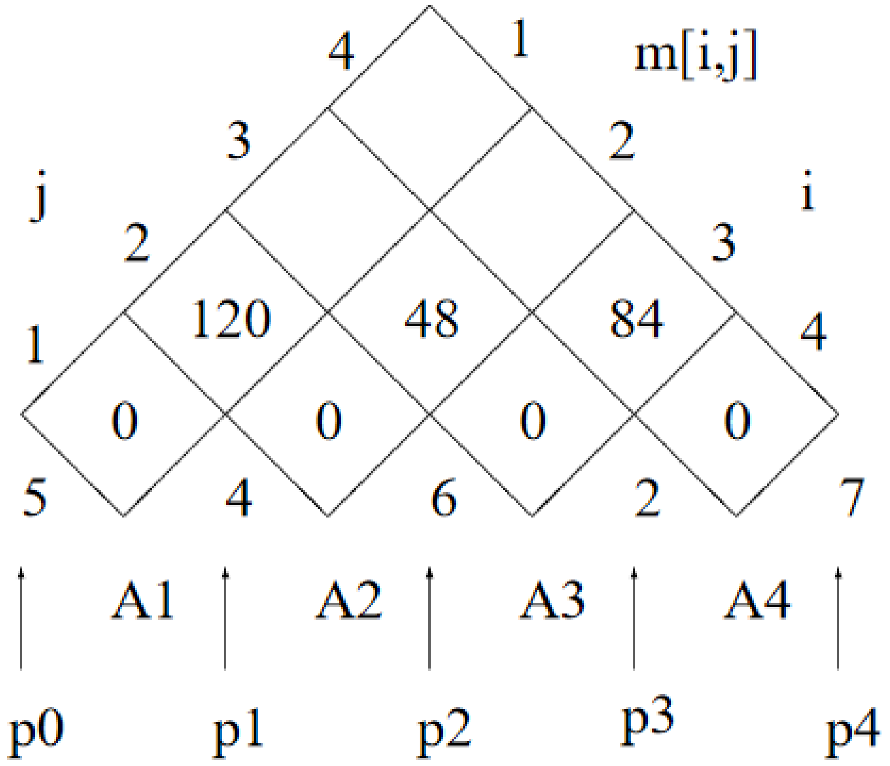
Stp 2: Computing $m[2,3]$ By definition

$$\begin{aligned} m[2,3] &= \min_{2 \le k < 3} (m[2,k] + m[k+1,3] + p_1 p_k p_3) \\ &= m[2,2] + m[3,3] + p_1 p_2 p_3 = 48. \end{aligned}$$



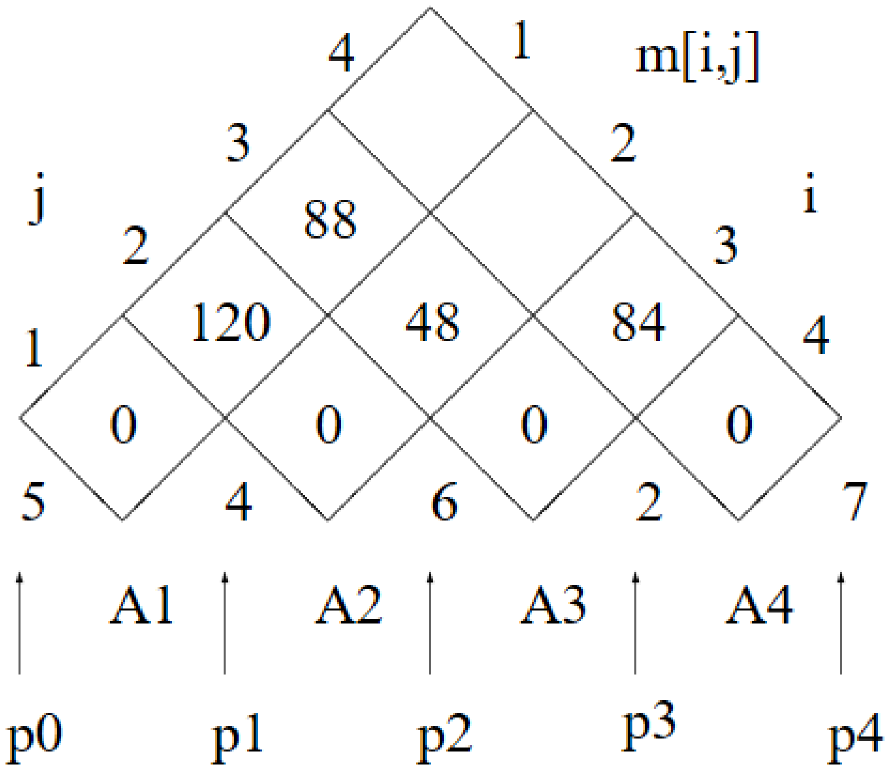
Stp3: Computing $m[3,4]$ By definition

$$\begin{aligned} m[3,4] &= \min_{3\le k<4}(m[3,k]+m[k+1,4]+p_2p_kp_4) \\ &= m[3,3]+m[4,4]+p_2p_3p_4 = 84. \end{aligned}$$



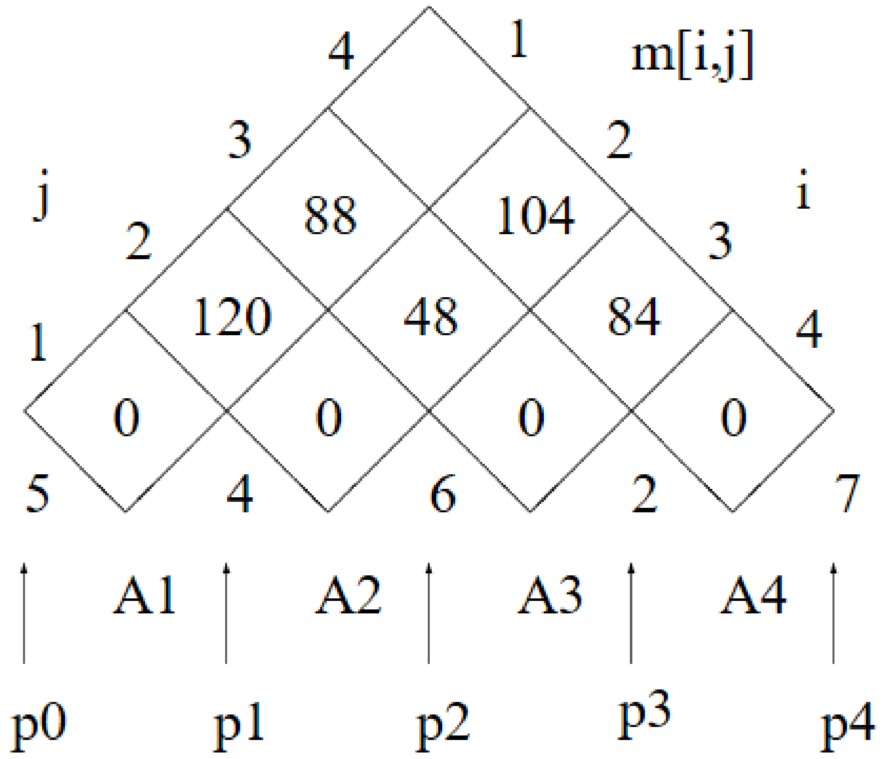
Stp4: Computing $m[1,3]$ By definition

$$
\begin{aligned}
m[1,3] &= \min_{1\le k<3}\left(m[1,k]+m[k+1,3]+p_0p_kp_3\right)\\
&= \min\left\{\begin{array}{l} m[1,1]+m[2,3]+p_0p_1p_3 \\ m[1,2]+m[3,3]+p_0p_2p_3 \end{array}\right\}\\
&= 88.
\end{aligned}
$$



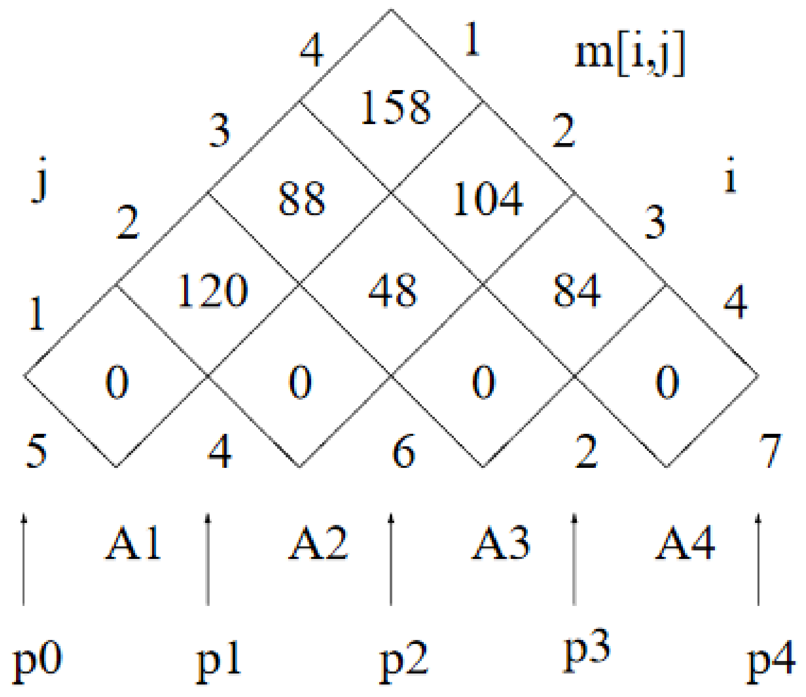
Stp5: Computing $m[2,4]$ By definition

$$\begin{aligned} m[2,4] &= \min_{2 \le k < 4} (m[2,k] + m[k+1,4] + p_1 p_k p_4) \\ &= \min \left\{ \begin{array}{l} m[2,2] + m[3,4] + p_1 p_2 p_4 \\ m[2,3] + m[4,4] + p_1 p_3 p_4 \end{array} \right\} \\ &= 104. \end{aligned}$$



St6: Computing $m[1,4]$ By definition

$$\begin{aligned} m[1,4] &= \min_{1\le k<4}(m[1,k]+m[k+1,4]+p_0p_kp_4) \\ &= \min\left\{\begin{array}{l} m[1,1]+m[2,4]+p_0p_1p_4 \\ m[1,2]+m[3,4]+p_0p_2p_4 \\ m[1,3]+m[4,4]+p_0p_3p_4 \end{array}\right\} \\ &= 158. \end{aligned}$$



Top-Down v/s Bottom Up Approach

- For solving such problems, instead of using a recursive algorithm directly, we apply a tabular bottom-up approach
- A recursive algorithm applies a top-down approach solving a sub-problem just when it is encountered, we do not want to do it here because sub problems overlap
- In the bottom-up approach as we shall see, we start with the smallest instance, solve it and save the result and then move further
- In this method, whenever a new problem is encountered all its sub-problems are already solved and therefore, we only have to look at the optimal solution of the subproblem already stored somewhere

- Since, the smallest instance is solved first, the bottom-up approach does not use a recursive algorithm for problem solving
- For solving such problems, top down approach may also be applied with a technique called **memoization**
- Memoization refers to memorization of the results of sub-problems generated while solving the problem
- The algorithm before making a recursive call checks if the result is already available or not
- If yes, then the saved result is used rather than solving the subproblem again
- Among the two approaches bottom-up algorithm is useful if all the subproblems are generated at least once, this is because it won't involve the overhead of making recursive calls
- Top-down approach on the other hand, is useful when not all the subproblems need be generated, it saves time by only solving the subproblems that are generated in the course of solving a particular problem instance

```
RECURSIVE-MATRIX-CHAIN$(p, i, j)$
1  if $i == j$
2      return 0
3  $m[i, j] = \infty$
4  for $k = i$ to $j - 1$
5      $q =$ RECURSIVE-MATRIX-CHAIN$(p, i, k)$
            $+$ RECURSIVE-MATRIX-CHAIN$(p, k + 1, j)$
            $+ p_{i-1} p_k p_j$
6      if $q < m[i, j]$
7          $m[i, j] = q$
8  return $m[i, j]$
```

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length - 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j - 1
6          q = LOOKUP-CHAIN(m, p, i, k)
                + LOOKUP-CHAIN(m, p, k + 1, j) + $p_{i-1} p_k p_j$
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

```
PRINT-OPTIMAL-PARENS($s, i, j$)
1  if $i == j$
2      print "$A$"$_i$
3  else print "("
4      PRINT-OPTIMAL-PARENS($s, i, s[i, j]$)
5      PRINT-OPTIMAL-PARENS($s, s[i, j] + 1, j$)
6      print ")"
```