Week 11 – S02
Dynamic Programming contd.

50.004 Introduction to Algorithms

Dr. Subhajit Datta

ISTD, SUTD

The story so far …

- Dynamic programming (DP)
  - A general techniques for algorithm design
- "Programming" does not necessarily refer to writing computer programs
  - Although, coding does help understand the subtleties better ☺
- Algorithms using DP solve problems by combining solution to sub-problems
- We have seen DP applications in
  - Fibonacci series computation, finding the least cost path in a tree, optimal rod-cutting problem, optimal text justification problem

What is common to all these problems?

- Overlapping sub-problem space
  - If you solve one sub-problem once, you can reuse the solution later!

Think!

- Is a recursive formulation of a problem a necessary and a sufficient condition for an effective DP solution?
  - If yes, prove it
  - If **no**, give a counter-example

Fibonacci: Bottom-up (Iterative)

```
table = {}
def fiboBottomUp(n):
    for i in range(1,n+1):
        if i <= 2:
            fibo = 1
        else:
            fibo = table[i-1]+table[i-2]
        table[i] = fibo
    return table[n]
```

Think further!

- Does DP always involve memoization?
  - If yes, prove it
  - If **no**, give a counter-example

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

How can you guess the running time of a DP algorithm? Without looking at the algorithm ☺

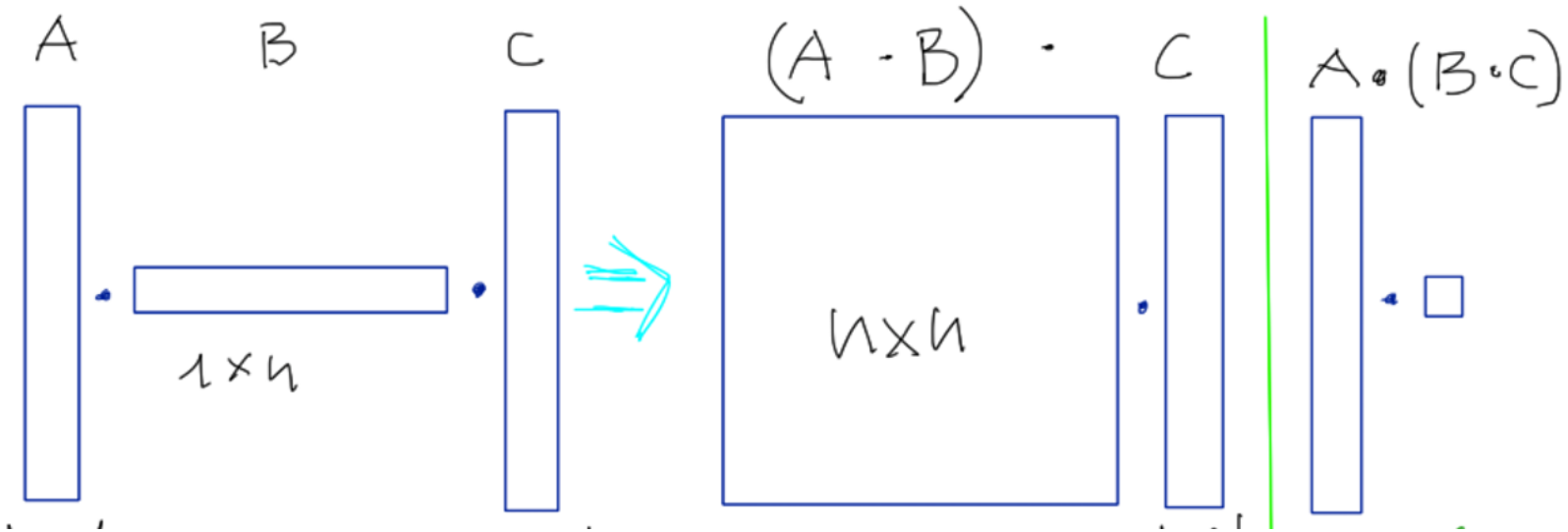
Informally, the running time of a dynamic-programming algorithm depends on the product of two factors: the number of subproblems overall and how many choices we look at for each subproblem. In rod cutting, we had $\Theta(n)$ subproblems overall, and at most n choices to examine for each, yielding an $O(n^2)$ running time.

Matrix chain multiplication

Our next example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. We are given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product

$$A_1 A_2 \cdots A_n$$

What is parenthesization?



How many ways can we parenthesize the following matrices? 5 ways!

$$\langle A_1, A_2, A_3, A_4 \rangle$$

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

How many scalar multiplication are required?

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

*The number of **scalar multiplications**!*

We can multiply two matrices A and B only if they are ***compatible***: the number of columns of A must equal the number of rows of B. If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr.

Exercise

- Given a matrix chain

$$\langle A_1, A_2, A_3 \rangle \text{ of three matrices}$$

- Dimensions of the matrices are

$$10 \times 100,\ 100 \times 5,\ \text{and}\ 5 \times 50$$

- How many scalar multiplications do we need for the multiplications with each of the following parenthesizations?

$$((A_1 A_2) A_3)$$

$$(A_1 (A_2 A_3))$$

Exercise

- Given a matrix chain

$$\langle A_1, A_2, A_3 \rangle \text{ of three matrices}$$

- Dimensions of the matrices are

$$10 \times 100,\ 100 \times 5,\ \text{and}\ 5 \times 50$$

- How many scalar multiplications do we need for the multiplications with each of the following parenthesizations?

$$((A_1 A_2) A_3)$$

If we multiply according to the parenthesization $((A_1 A_2) A_3)$, we perform $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product $A_1 A_2$, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications.

Exercise

- Given a matrix chain

$$\langle A_1, A_2, A_3 \rangle \text{ of three matrices}$$

- Dimensions of the matrices are

$$10 \times 100,\ 100 \times 5,\ \text{and}\ 5 \times 50$$

- How many scalar multiplications do we need for the multiplications with each of the following parenthesizations?

$(A_1(A_2A_3))$

If instead we multiply according to the parenthesization $(A_1(A_2A_3))$, we perform $100 \cdot 5 \cdot 50 = 25{,}000$ scalar multiplications to compute the 100×50 matrix product A_2A_3, plus another $10 \cdot 100 \cdot 50 = 50{,}000$ scalar multiplications to multiply A_1 by this matrix, for a total of 75,000 scalar multiplications.

Exercise

- Given a matrix chain

$$\langle A_1, A_2, A_3 \rangle \text{ of three matrices}$$

- Dimensions of the matrices are

$$10 \times 100,\ 100 \times 5,\ \text{and}\ 5 \times 50$$

- How many scalar multiplications do we need for the multiplications with each of the following parenthesizations?

$$((A_1 A_2) A_3)$$

$$(A_1 (A_2 A_3))$$

7500 versus 75000; factor of 10 is a big deal!

This is why we need to worry about optimal parenthesization!

- Given a matrix chain

$$\langle A_1, A_2, A_3 \rangle \text{ of three matrices}$$

- Dimensions of the matrices are

$$10 \times 100,\ 100 \times 5,\ \text{and}\ 5 \times 50$$

- How many scalar multiplications do we need for the multiplications with each of the following parenthesizations?

$((A_1 A_2) A_3)$

$(A_1 (A_2 A_3))$

7500 versus 75000; factor of 10 is a big deal!

This is not about actually multiplying matrices ☺

We state the ***matrix-chain multiplication problem*** as follows: given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 7500 scalar multiplications instead of 75,000).

What is the time complexity of this?

Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. When $n = 1$, we have just one matrix and therefore only one way to fully parenthesize the matrix product. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and $(k+1)$st matrices for any $k = 1, 2, \ldots, n-1$. Thus, we obtain the recurrence

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

Let us assume that $P(k) \geq c2^k$ for all $k < n$. Let us prove that $P(n) \geq c2^n$

For $n > 1$,

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

$\geq \sum_{k=1}^{n-1} c2^k c2^{n-k} = \sum_{k=1}^{n-1} c^2 2^n = (n-1)c^2 2^n$

$\geq c2^n$, *provided that* $(n-1)c > 1$

Because $n>1$, there exists threshold, and $c >$ threshold make $(n-1)c > 1$.

$$\Omega(2^n)$$

Exponential

Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. When $n = 1$, we have just one matrix and therefore only one way to fully parenthesize the matrix product. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and $(k+1)$st matrices for any $k = 1, 2, \ldots, n-1$. Thus, we obtain the recurrence

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

Let us assume that P(k) ≥ c2^k for all k < n. Let us prove that P(n) ≥ c2^n

For n > 1,

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

≥ $\sum_{k=1}^{n-1} c2^k c2^{n-k} = \sum_{k=1}^{n-1} c^2 2^n = (n-1)c^2 2^n$

≥ c2^n , *provided that* $(n-1)c > 1$

Because n>1, there exists threshold, and c > threshold make $(n-1)c > 1$.

$$\Omega(2^n)$$

How can we improve by using DP?

Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. When $n = 1$, we have just one matrix and therefore only one way to fully parenthesize the matrix product. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and $(k+1)$st matrices for any $k = 1, 2, \ldots, n-1$. Thus, we obtain the recurrence

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

Let us assume that $P(k) \geq c2^k$ for all $k < n$. Let us prove that $P(n) \geq c2^n$

For $n > 1$,

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

$\geq \sum_{k=1}^{n-1} c2^k c2^{n-k} = \sum_{k=1}^{n-1} c^2 2^n = (n-1)c^2 2^n$

$\geq c2^n$, *provided that* $(n-1)c > 1$

Because $n>1$, there exists threshold, and $c >$ threshold make $(n-1)c > 1$.

$$\Omega(2^n)$$

What is the optimal substructure of this problem?

For convenience, let us adopt the notation $A_{i..j}$, where $i \le j$, for the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$. Observe that if the problem is nontrivial, i.e., $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \le k < j$. That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

The optimal substructure of this problem is as follows. Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, we split the product between A_k and A_{k+1}. Then the way we parenthesize the "prefix" subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$. Why? If there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then we could substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce another way to parenthesize $A_i A_{i+1} \cdots A_j$ whose cost was lower than the optimum: a contradiction. A similar observation holds for how we parenthesize the subchain $A_{k+1} A_{k+2} \cdots A_j$ in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} \cdots A_j$.

Developing a recursive solution

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$. Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the lowest-cost way to compute $A_{1..n}$ would thus be $m[1, n]$.

We can define $m[i, j]$ recursively as follows. If $i = j$, the problem is trivial; the chain consists of just one matrix $A_{i..i} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, $m[i, i] = 0$ for $i = 1, 2, \ldots, n$. To compute $m[i, j]$ when $i < j$, we take advantage of the structure of an optimal solution from step 1. Let us assume that to optimally parenthesize, we split the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, where $i \le k < j$. Then, $m[i, j]$ equals the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$, plus the cost of multiplying these two matrices together. Recalling that each matrix A_i is $p_{i-1} \times p_i$, we see that computing the matrix product $A_{i..k} A_{k+1..j}$ takes $p_{i-1} p_k p_j$ scalar multiplications. Thus, we obtain

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

Recursive definition of minimum cost

Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

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

The $m[i,j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i,j]$ to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization. That is, $s[i,j]$ equals a value k such that $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$.

What will happen of we build an algorithm based on this recurrence?

Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

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

The $m[i, j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i, j]$ to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Such a recursive algorithm will take exponential time

Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

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

The $m[i, j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i, j]$ to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Instead …

We shall implement the tabular, bottom-up method in the procedure MATRIX-CHAIN-ORDER, which appears below. This procedure assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$. Its input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $p.length = n + 1$. The procedure uses an auxiliary table $m[1..n, 1..n]$ for storing the $m[i, j]$ costs and another auxiliary table $s[1..n-1, 2..n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$. We shall use the table s to construct an optimal solution.

The algorithm, at last ☺

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 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 and s
```

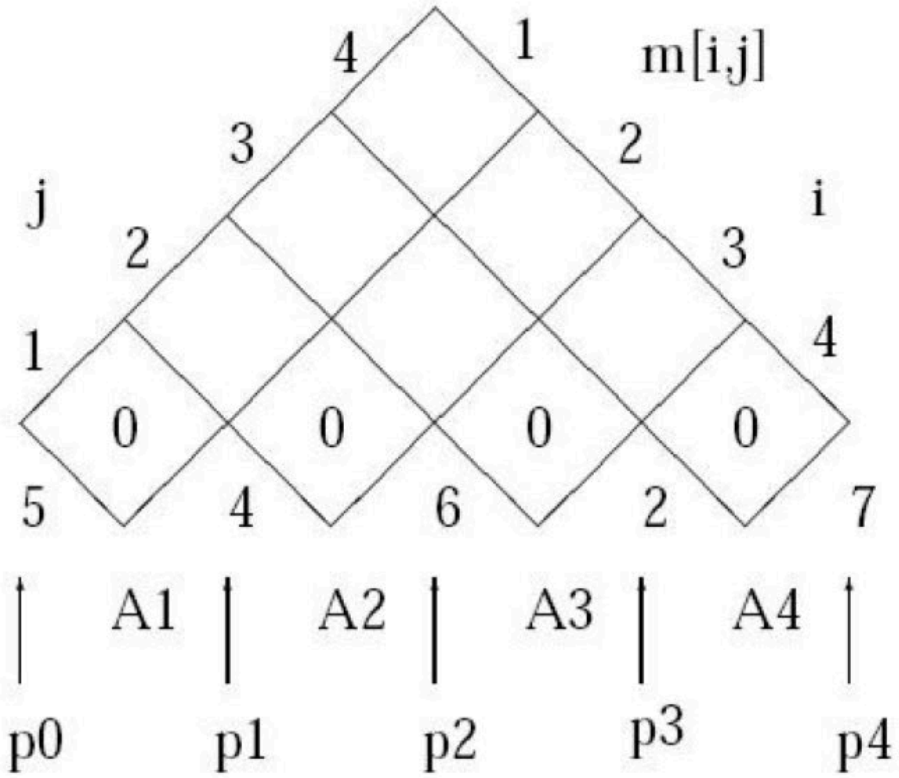
Time complexity? $O(n^3)$ of course!

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 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 and s
```

Example

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 $m[1, 4]$.

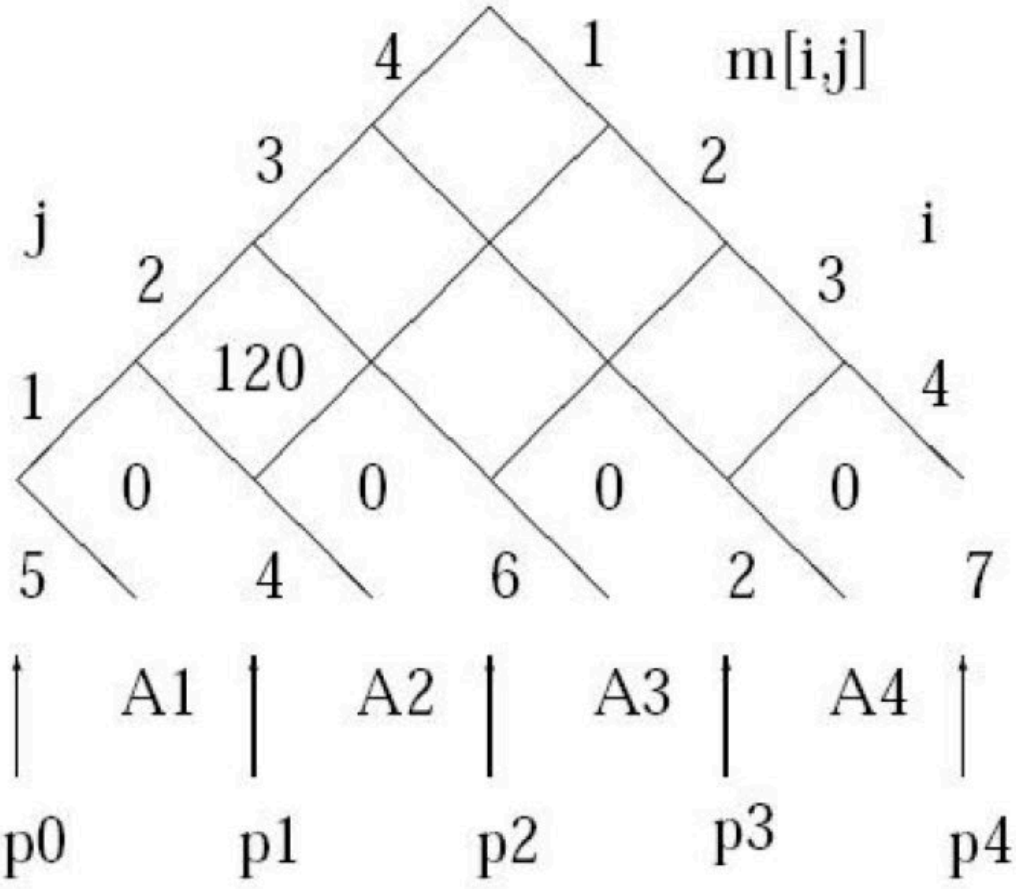
S0: Initialization



Step 1: Computing $m[1,2]$

By definition

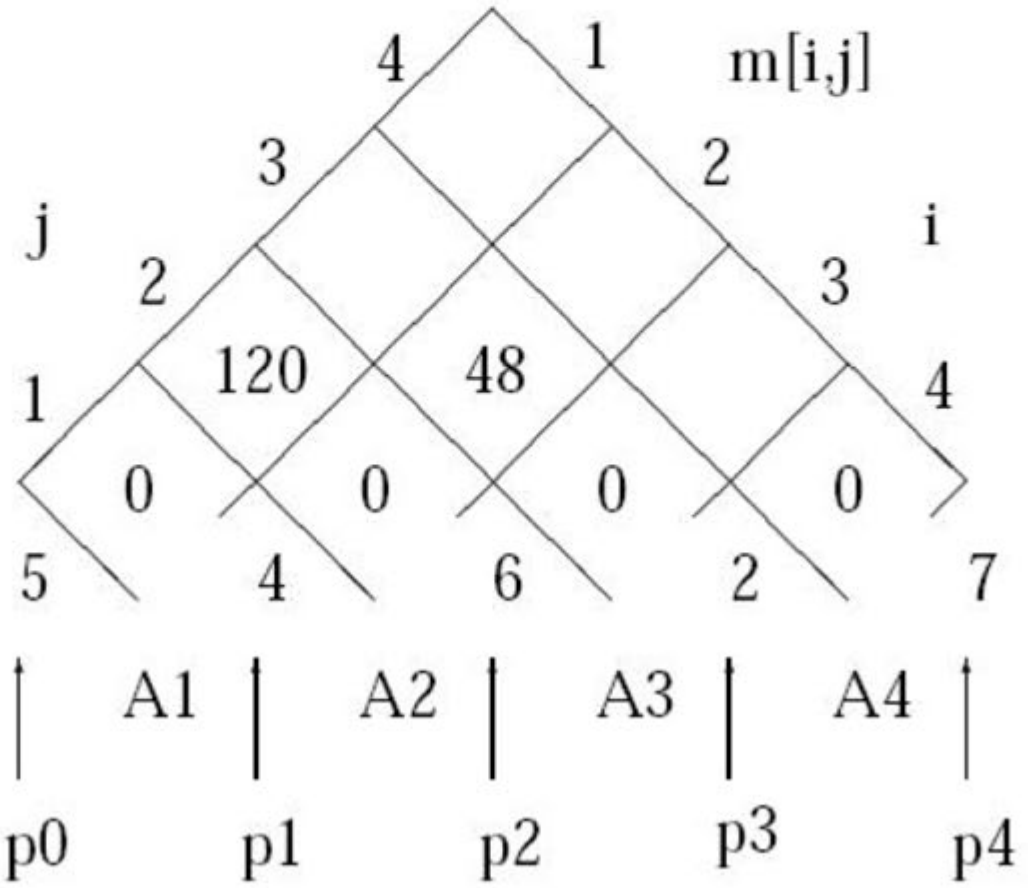
$$m[1,2] = \min_{1 \le k < 2} (m[1,k] + m[k+1,2] + p_0 p_k p_2)$$
$$= m[1,1] + m[2,2] + p_0 p_1 p_2 = 120.$$



Step 2: Computing $m[2,3]$

By definition

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

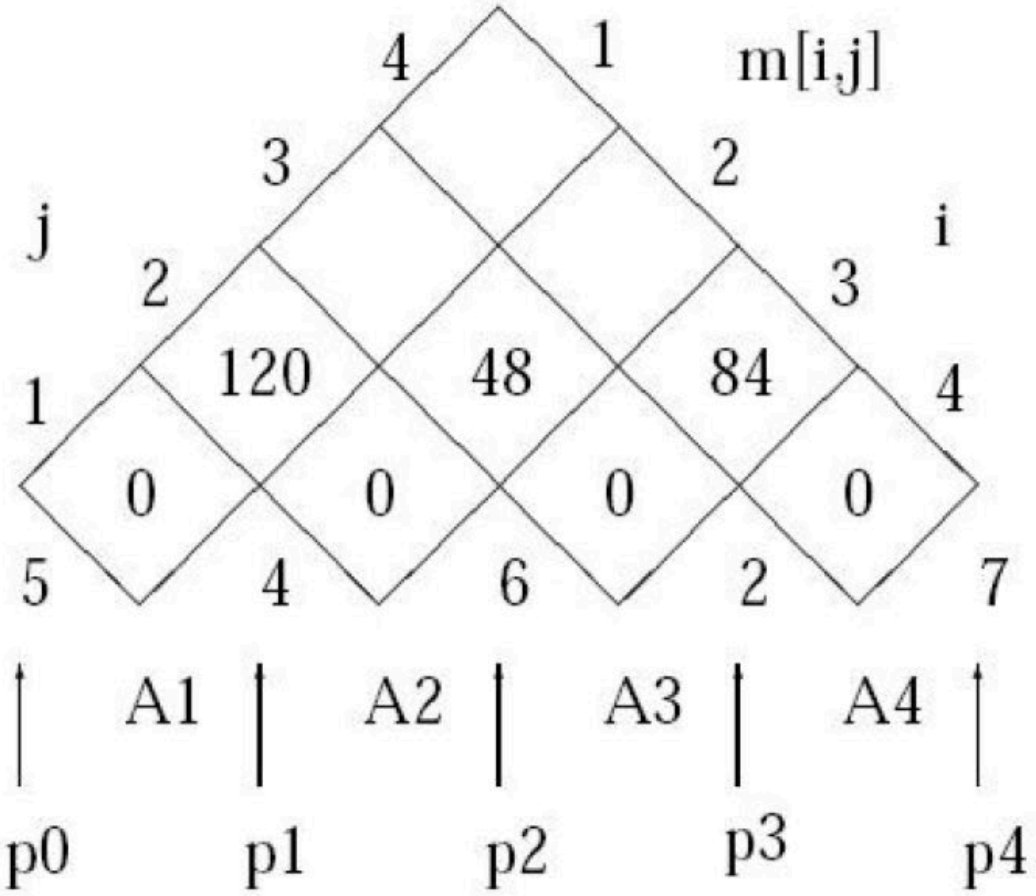


Step 3: Computing $m[3,4]$

By definition

$$m[3,4] = \min_{3 \le k < 4} (m[3,k] + m[k+1,4] + p_2 p_k p_4)$$

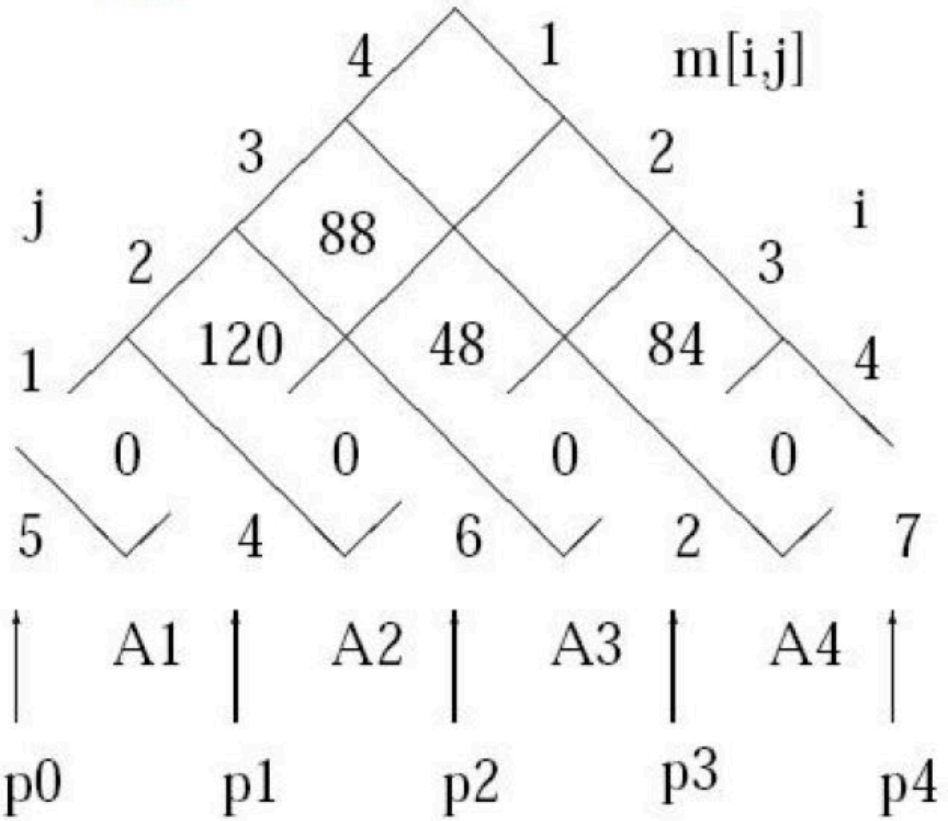
$$= m[3,3] + m[4,4] + p_2 p_3 p_4 = 84.$$



Step 4: Computing $m[1,3]$

By definition

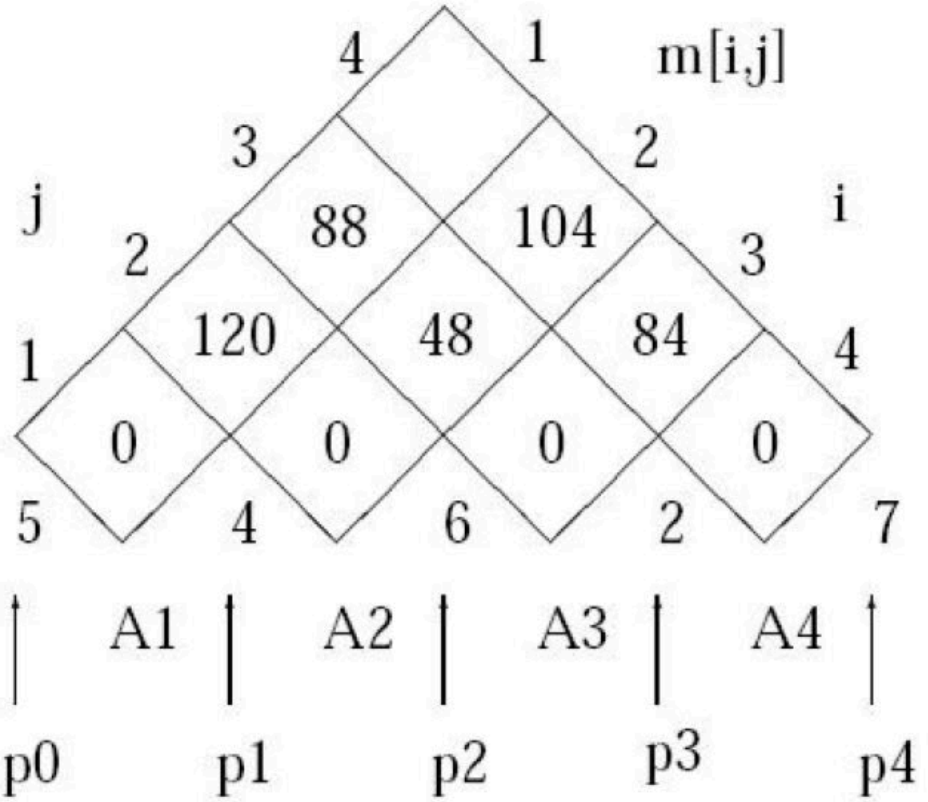
$$
\begin{aligned}
m[1,3] &= \min_{1 \le k < 3} (m[1,k] + m[k+1,3] + p_0 p_k p_3) \\
&= \min \left\{ \begin{array}{l} m[1,1] + m[2,3] + p_0 p_1 p_3 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 \end{array} \right\} \\
&= 88.
\end{aligned}
$$



Step 5: Computing $m[2,4]$

By definition

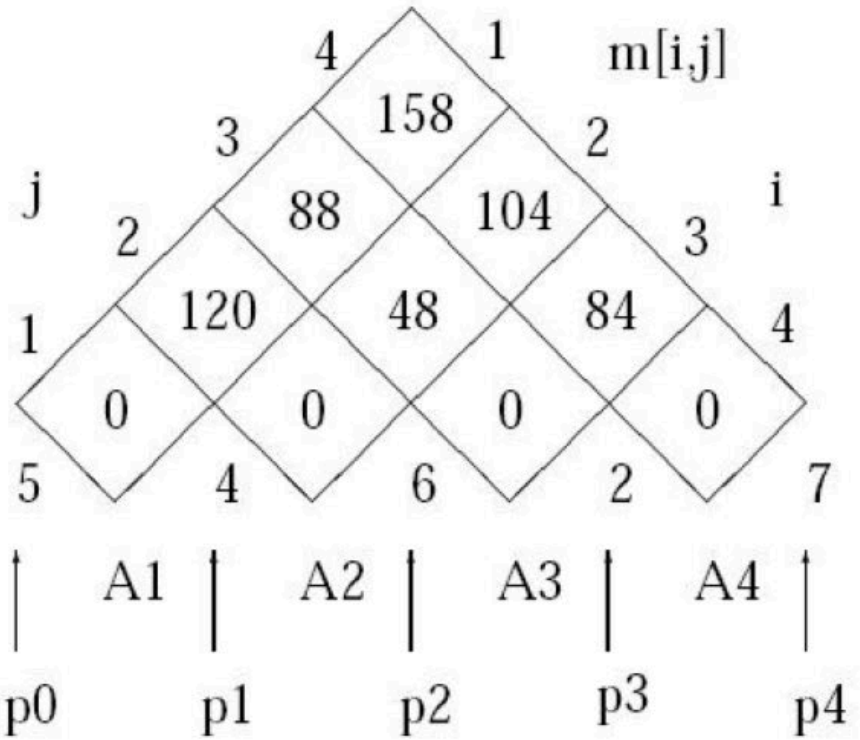
$$
\begin{aligned}
m[2,4] &= \min_{2 \le k < 4} (m[2,k] + m[k+1,4] + p_1 p_k p_4) \\
&= \min \left\{ \begin{array}{l} m[2,2] + m[3,4] + p_1 p_2 p_4 \\ m[2,3] + m[4,4] + p_1 p_3 p_4 \end{array} \right\} \\
&= 104.
\end{aligned}
$$



Step 6: Computing $m[1,4]$

By definition

$$
\begin{aligned}
m[1,4] &= \min_{1 \le k < 4} (m[1,k] + m[k+1,4] + p_0 p_k p_4) \\
&= \min \left\{ \begin{array}{l} m[1,1] + m[2,4] + p_0 p_1 p_4 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 \end{array} \right\} \\
&= 158.
\end{aligned}
$$



Another example

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25