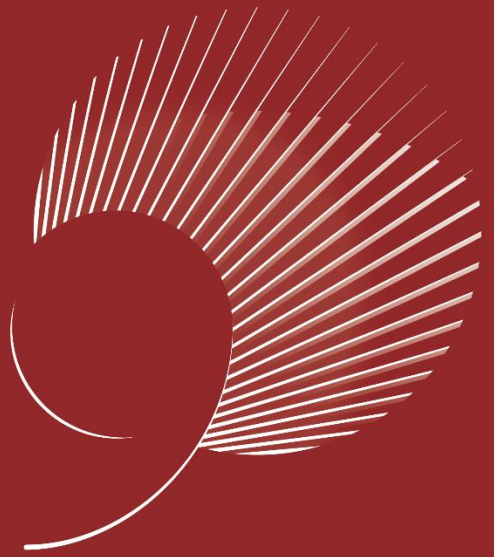
Chapter 15: Dynamic Programming (part I)

Chi-Yeh Chen

陳奇業

成功大學資訊工程學系



藏行顯光
成就共好

Achieve Securely
Prosper Mutually

國立成功大學 九十週年
90th Anniversary of NCKU

Dynamic Programming

- Not a specific algorithm, but a technique (like divide-and-conquer).
- “Programming” in this context refers to a tabular method, not to writing computer code.
- Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
- Dynamic programming applies when the subproblems overlap-that is, when subproblems share subsubproblems.

- Used for optimization problems:
  - Find a solution with the optimal value.
  - Minimization or maximization. (We'll see both.)
- We call such a solution an optimal solution to the problem, as opposed to the optimal solution.

Four-step method

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution in a bottom-up fashion.
- Construct an optimal solution from computed information.

Rod cutting

NCKU
National Cheng Kung University

藏行顯光
成就共好
Achieve Securely
Prosper Mutually
國立成功大學 九十週年
90th Anniversary of NCKU

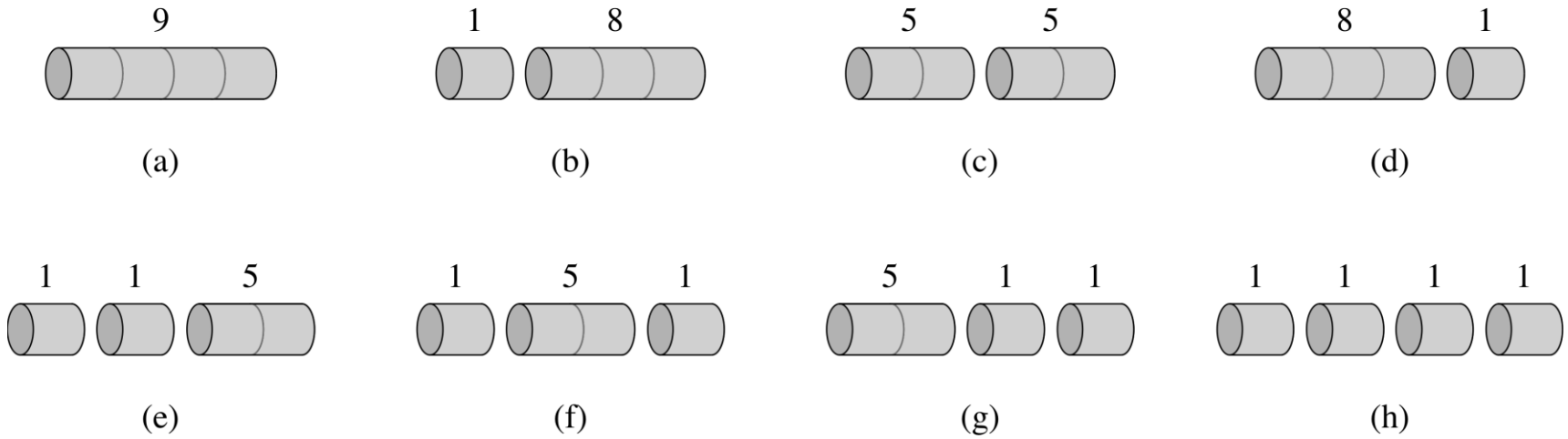
Rod cutting

- How to cut steel rods into pieces in order to maximize the revenue you can get?
- Each cut is free. Rod lengths are always an integral number of inches.
- **Input:** A length n and table of prices p_i, for $i = 1, 2, \ldots, n$.
- **Output:** the maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods .
- If p_n is large enough, an optimal solution might require no cuts, i.e., just leave the rod as n inches long.

- **Example**:[Using the first 8 values from the example in the book.]

Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

- Can cut up a rod in 2^{n-1} different ways, because can choose to cut or not cut after each of the first $n-1$ inches.
- Here are all 8 ways to cut a rod of length 4, with the costs from the example:



- The best way is to cut it into two 2-inch pieces, getting a revenue of $p_2 + p_2 = 5 + 5 = 10$.
- Let r_i be the maximum revenue for a rod of length i. Can express a solution as a sum of individual rod lengths.

- Can determine optimal revenues r_i for the example, by inspection:

i	r_i	optimal solution
1	1	1(no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	2 + 2
5	13	2 + 3
6	17	6 (no cuts)
7	18	1 + 6 or 2 + 2 + 3
8	22	2 + 6

Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

- Can determine optimal revenue r_n by taking the maximum of
  - p_n: the price we get by not making a cut,
  - $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod $n - 1$ inches,
  - $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n - 2$ inches, …
  - $r_{n-1} + r_1$.
- That is,

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

Optimal substructure

- **Optimal substructure**: To solve the original problem of size n, solve subproblems on smaller sizes. After making a cut, we have two subproblems. The optimal solution to the original problem incorporates optimal solutions to the subproblems. We may solve the subproblems independently.
- *Example*: For $n = 7$, one of the optimal solutions makes a cut at 3 inches, giving two subproblems, of lengths 3 and 4. We need to solve both of them optimally. The optimal solution for the problem of length 4, cutting into 2 pieces, each of length 2, is used in the optimal solution to the original problem with length 7.

A simpler way to decompose the problem

- **A simpler way to decompose the problem**: Every optimal solution has a leftmost cut. In other words, there's some cut that gives a first piece of length i cut off the left end, and a remaining piece of length $n - i$ on the right.
  - Need to divide only the remainder, not the first piece.
  - Leaves only one subproblem to solve, rather than two subproblems.

- Say that the solution with no cuts has first piece size $i = n$ with revenue p_n, and remainder size 0 with revenue $r_0 = 0$.
- Gives a simpler version of the equation for r_n:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

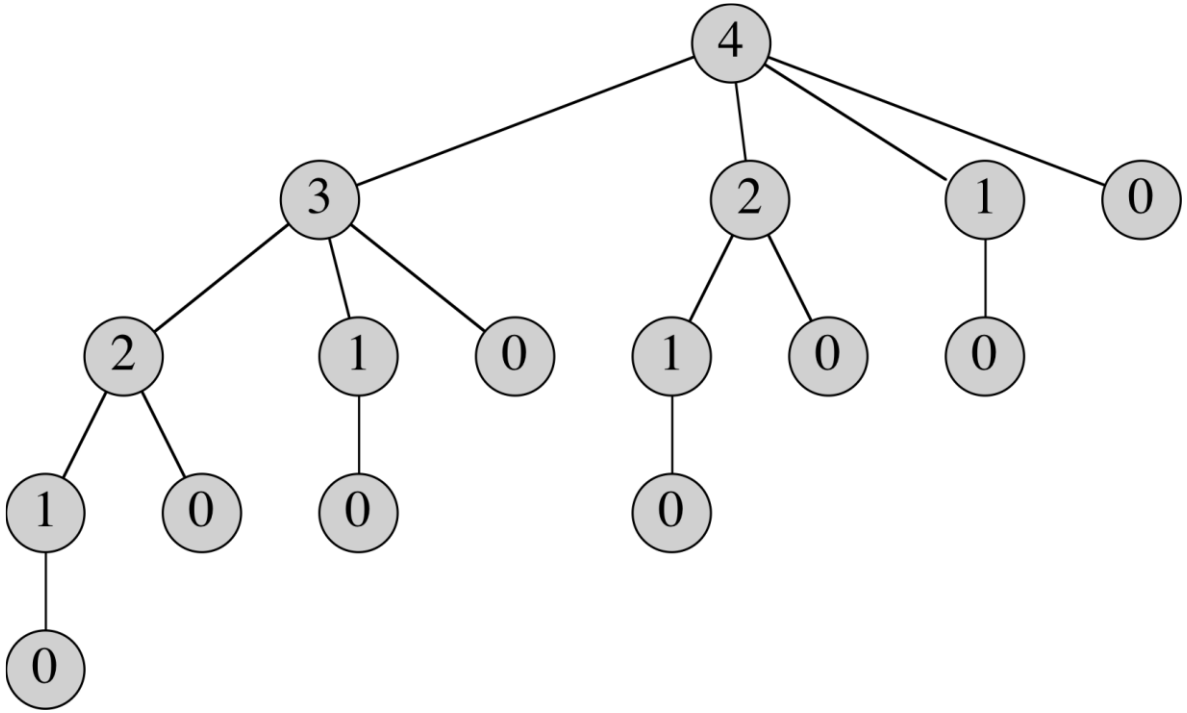
Recursive top-down

- **Recursive top-down**
  - Direct implementation of the simpler equation for r_n.
  - The call CUT-ROD(p, n) returns the optimal revenue r_n:

CUT-ROD(p, n)

```
1 if n == 0 then
2     return 0
3 q = −∞
4 for i = 1 to n do
5     q = max(q, p[i] + CUT-ROD(p, n − i))
6 return q
```

- This procedure works, but it is terribly *inefficient*. If you code it up and run it , it could take more than an hour for $n = 40$. Running time almost doubles each time n increases by 1.
- ***Why so inefficient?*:** CUT-ROD calls itself repeatedly, even on subproblems it has already solved. Here's a tree of recursive calls for $n = 4$. Inside each node is the value of n for the call represented by the node:



- Lots of repeated sub problems. Solve the subproblem for size 2 twice, for size 1 four times, and for size 0 eight times.

- *Exponential growth*: Let $T(n)$ equal the number of calls to CUT-ROD with second parameter equal to n. Then

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n \geq 1 \end{cases}$$

- Summation counts calls where second parameter is $j = n - i$.
- Solution to recurrence if $T(n) = 2^n$.

We can verify that $T(n) = 2^n$ is a solution to the given recurrence by the substitution method. We note that for $n = 0$, the formula is true since $2^0 = 1$. For $n > 0$, substituting into the recurrence and using the formula for summing a geometric series yields

$$T(n) = 1 + \sum_{j=0}^{n-1} 2^j = 1 + (2^n - 1) = 2^n$$

Dynamic-programming solution

- Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
- Save the solution to a subproblem in a table and refer back to the table whenever we revisit the subproblem.
- “Store, don’t recompute” => time-memory trade-off.
- Can turn an exponential-time solution into a polynomial-time solution.
- Two basic approaches: top-down with memoization, and bottom-up.

Top-down with memoization

- Solve recursively, but store each result in a table.
- To find the solution to a sub problem, first look in the table. If the answer is there, use it. Otherwise, compute the solution to the sub problem and then store the solution in the table for future use.

- **Memorizing** is remembering what we have computed previously.
- Memorized version of the recursive solution, storing the solution to the subproblem of length i in array entry $r[i]$:

```
MEMOIZED-CUT-ROD(p, n)
1 let r[0...n] be a new array
2 for i = 0 to n do
3       r[i] = -∞
4 return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

MEMOIZED-CUT-ROD-AUX(p, n, r)

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

Bottom-up

- Sort the subproblems by size and solve the smaller ones first. That way, when solving a subproblem, have already solved the smaller subproblems we need.

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 do
4       q = -∞
5       for i = 1 to j do
6           q = max(q, p[i] + r[j - i])
7       r[j] = q
8 return r[n]
```

Running time

- Both the top-down and bottom-up versions run in $\Theta(n^2)$ time.
  - Bottom-up: Doubly nested loops. Number of iterations of inner for loop forms an arithmetic series.
  - Top-down: MEMOIZED-CUT-ROD solves each subproblem just once, and it solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size n, the **for** loop iterates n times => over all recursive calls, total number of iterations forms an arithmetic series. [*Actually using aggregate analysis, which Chapter* 17 *covers*.]

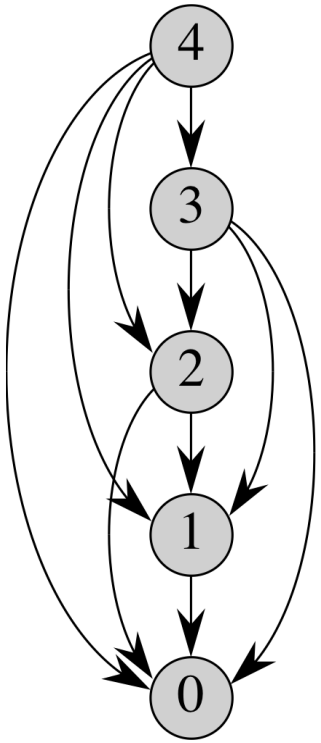
Subproblem graphs

- How to understand the subproblems involved and how they depend on each other.

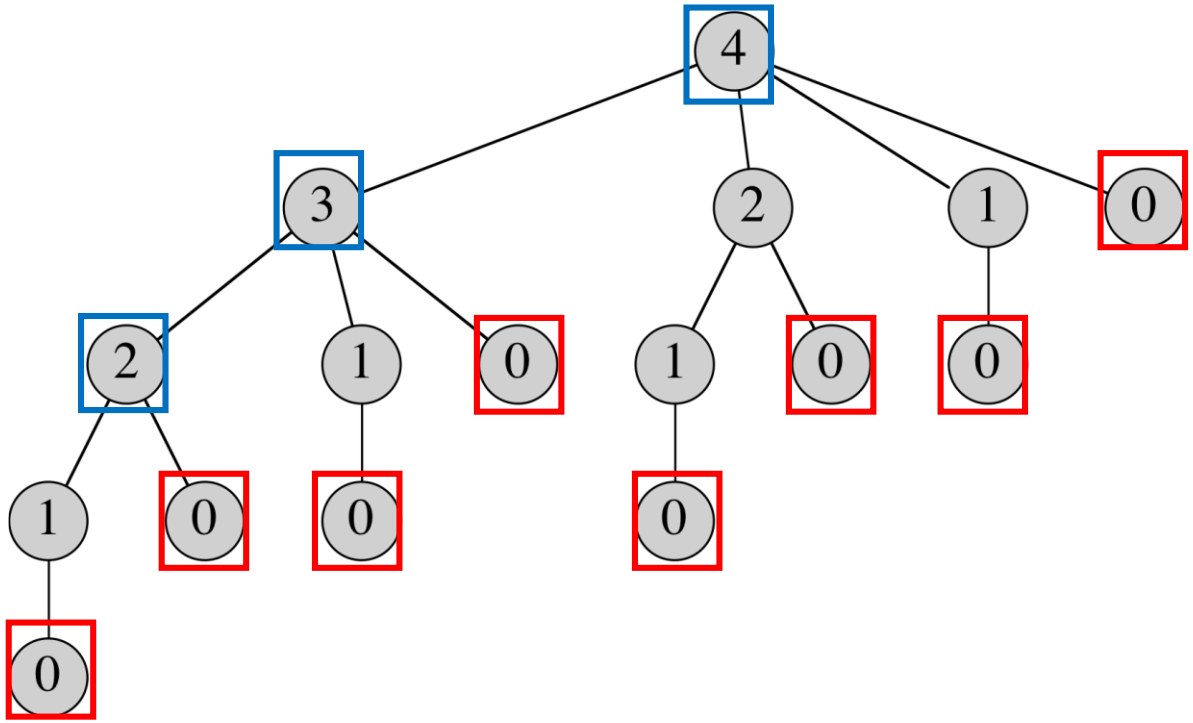
  Directed graph:

  - One vertex for each distinct subproblem.
  - Has a direct edge (x, y) if computing an optimal solution to subproblem x *directly* requires knowing an optimal solution to subproblem y.

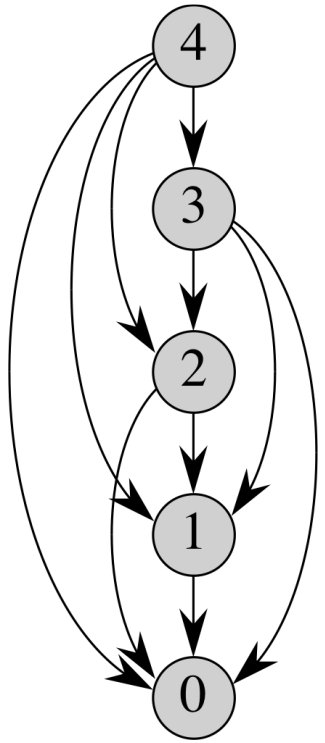
- **Example**: For rod-cutting problem with $n = 4$:



- Can think of the subproblem graph as a collapsed version of the tree of recursive calls, where all nodes for the same subproblem are collapsed into a single vertex, and all edges go from parent to child.
- Subproblem graph can help determine running time. Because we solve each subproblem just once, running time is sum of times needed to solve each subproblem.



Recursion tree



Subproblem graph

- Time to compute solution to a subproblem is typically linear in the out-degree(number of outgoing edges) of its vertex.
- Number of subproblems equals number of vertices.

- When these conditions hold, running time is linear in number of vertices and edges.

Reconstructing a solution

- So far, have focused on computing the value of an optimal solution, rather than the *choices* that produced an optimal solution.
- Extend the bottom-up approach to record not just optimal values, but optimal choices. Save the optimal choices in a separate table. Then use a separate procedure to print the optimal choices.

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
1  let r[0...n] and s[0...n] be new arrays
2  r[0] = 0
3  for j = 1 to n do
4      q = -∞
5      for i = 1 to j do
6          if q < p[i] + r[j-i] then
7              q = p[i] + r[j-i]
8              s[j] = i
9      r[j] = q
10 return r and s
```

- Saves the first cut made in an optimal solution for a problem of size i in $s[i]$.
- To print out the cuts made in an optimal solution:

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0 do
3      print s[n]
4      n = n − s[n]
```

- **Examples**: For the example, EXTENDED-BOTTOM-UP-CUT-ROD returns

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$	0	1	2	3	2	2	6	1	2

- A call to PRINT-CUT-ROD-SOLUTION(p, 8) calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the above r and s tables. Then it prints 2, sets n to 6, prints 6, and finishes (because n becomes 0).

Modification of the rod-cutting problem

Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

MODIFIED-CUT-ROD(p, n)

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

Matrix-chain multiplication

NCKU
National Cheng Kung University

藏行顯光
成就共好
Achieve Securely
Prosper Mutually
國立成功大學 九十週年
90th Anniversary of NCKU

Matrix-chain multiplication

- A product of matrices is fully parenthesized if it is either a single matrix, or a product of two fully parenthesized matrix product, surrounded by parentheses.
- How to compute $A_1 A_2 \cdots A_n$ where A_i is a matrix for every i.
- Example: $A_1 A_2 A_3 A_4$
  $\big(A_1(A_2(A_3 A_4))\big), \big(A_1((A_2 A_3)A_4)\big), ((A_1 A_2)(A_3 A_4)), \big(((A_1 A_2)A_3)A_4\big)$

MARTIX-MULTIPLY(A, B)

```
1  if A.columns ≠ B.rows then
2      error "incompatible dimensions"
3  else
4      let C be a new A.rows × B.columns matrix
5      for i = 1 to A.rows do
6          for j = 1 to B.columns do
7              c_{i,j} = 0
8              for k = 1 to A.columns do
9                  c_{i,j} = c_{i,j} + a_{i,k} · b_{k,j}
10 return C
```

Complexity:

- Let A be a $p \times q$ matrix, and B be a $q \times r$ matrix. Then the complexity is $p \times q \times r$.
- Example: A_1 is a 10×100 matrix, A_2 is a 100×5 matrix, and A_3 is a 5×50 matrix. Then $\left((A_1 A_2)A_3\right)$ takes $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ time. However, $\left(A_1(A_2 A_3)\right)$ takes $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$ time.

The matrix-chain multiplication problem:

- 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.

Counting the number of parenthesizations:

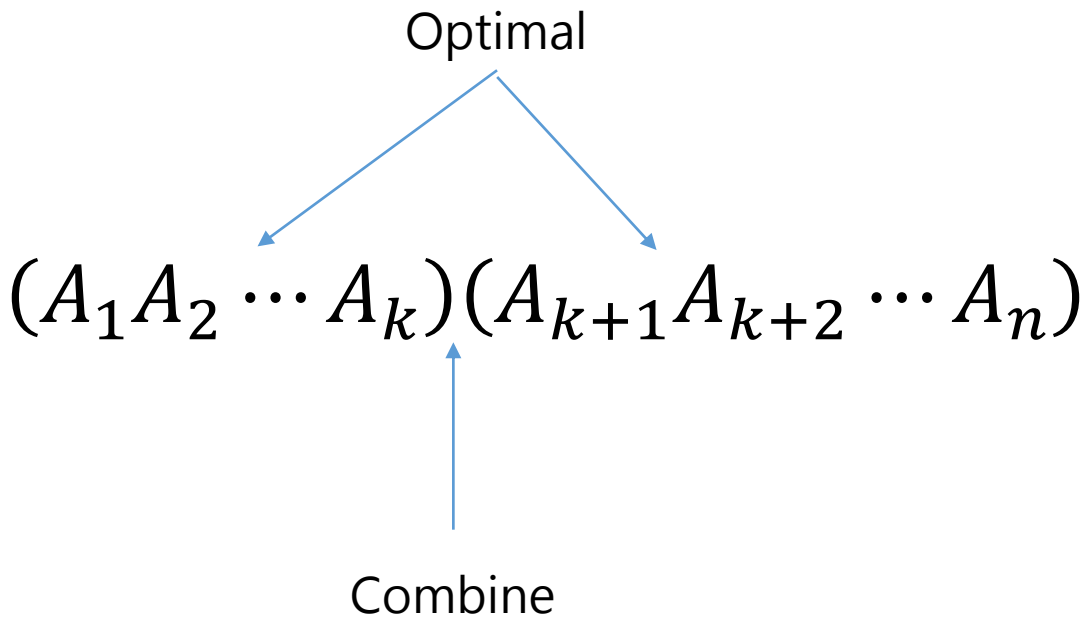
- Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$.
- When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and $(k+1)st$ matrices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

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

Counting the number of parenthesizations:

- $P(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$ [Catalan number]
- The number of solutions is thus exponential in n, and the brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain.

Step 1: The structure of an optimal parenthesization



Step 2: A recursive solution

- Define $m[i,j]$ = minimum number of scalar multiplications needed to compute the matrix $A_{i\cdots j} = A_i A_{i+1} \cdots A_j$.
- Goal: $m[1,n]$

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

Step 3: computing the optimal costs

- 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 achieve the optimal cost in computing $m[i, j]$. We shall use the table s to construct an optimal solution.

Step 3: computing the optimal costs

- In order to implement the bottom-up approach, we must determine which entries of the table we refer to when computing $m[i, j]$.
- 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.

- For $k = i, i + 1, \ldots, j - 1$, the matrix $A_{i..k}$ is a product of $k - i + 1 < j - i + 1$ matrices and the matrix $A_{k+1..j}$ is a product of $j - k < j - i + 1$ matrices.

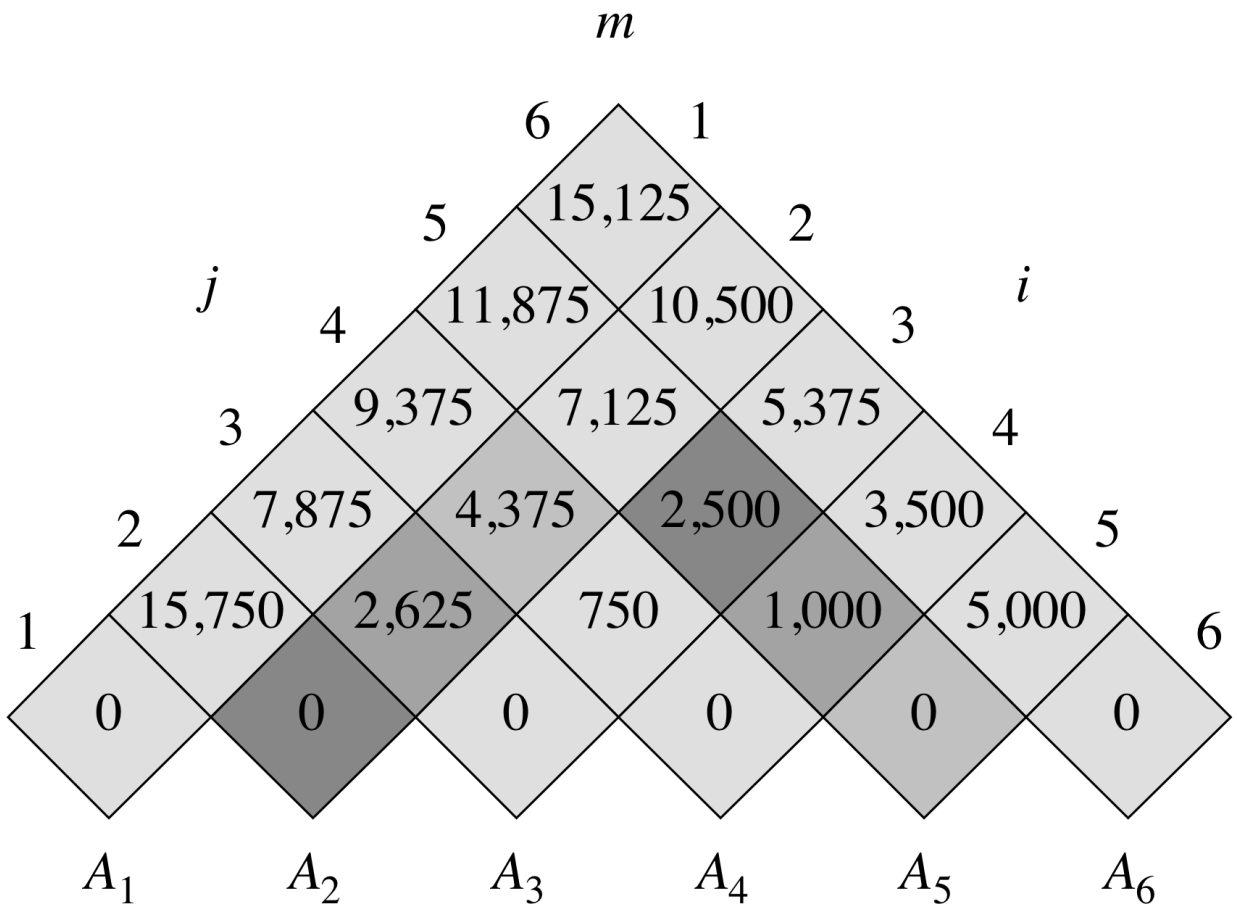
Step 3: computing the optimal costs

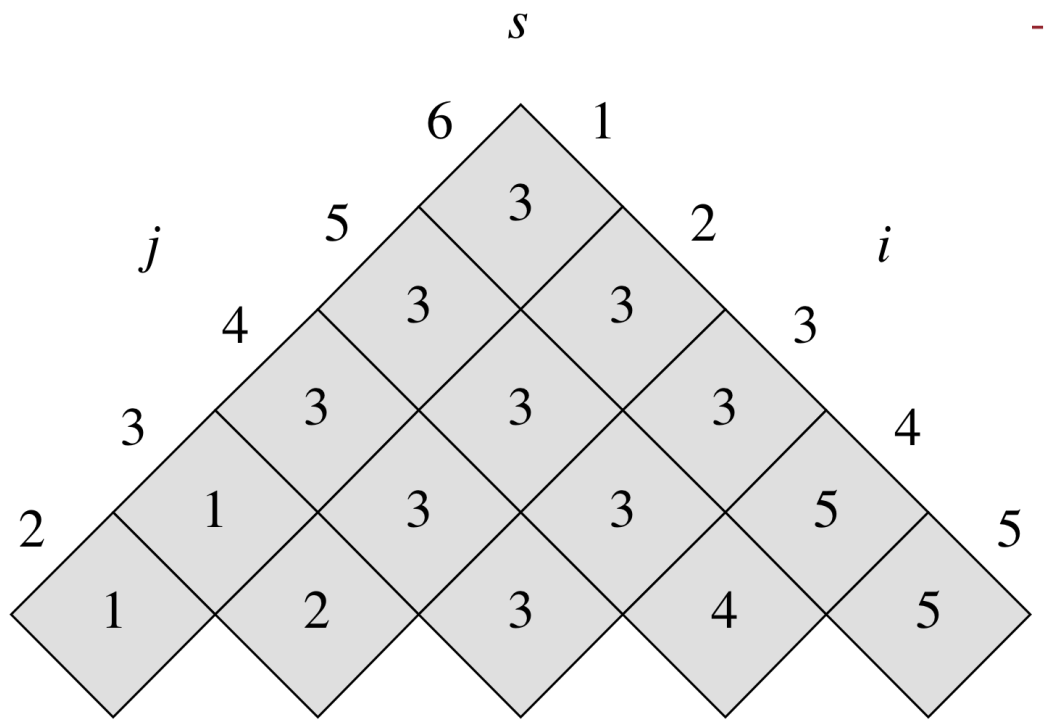
- Thus, the algorithm should fill in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length. For the subproblem of optimally parenthesizing the chain $A_i A_{i+1} \cdots A_j$, we consider the subproblem size to be the length $j - i + 1$ of the chain.

MARTIX-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 do
4      m[i, i] = 0
5  for l = 2 to n do
6      ▶ l is the chain length
7      for i = 1 to n − l + 1 do
8          j = i + l − 1
9          m[i, j] = ∞
10         for k = i to j − 1 do
11             q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
12             if q < m[i, j] then
13                 m[i, j] = q
14                 s[i, j] = k
15 return m and s
```

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

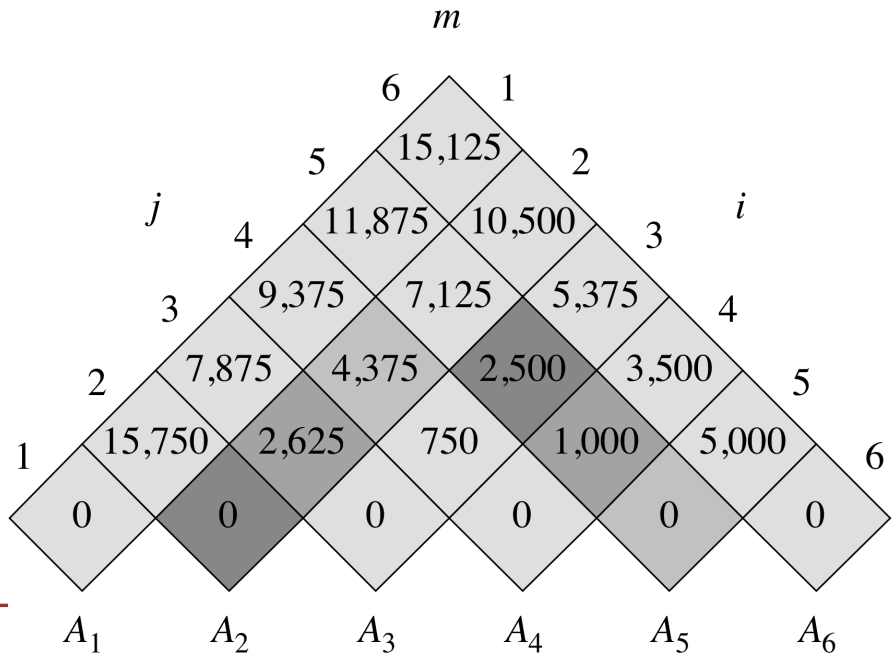


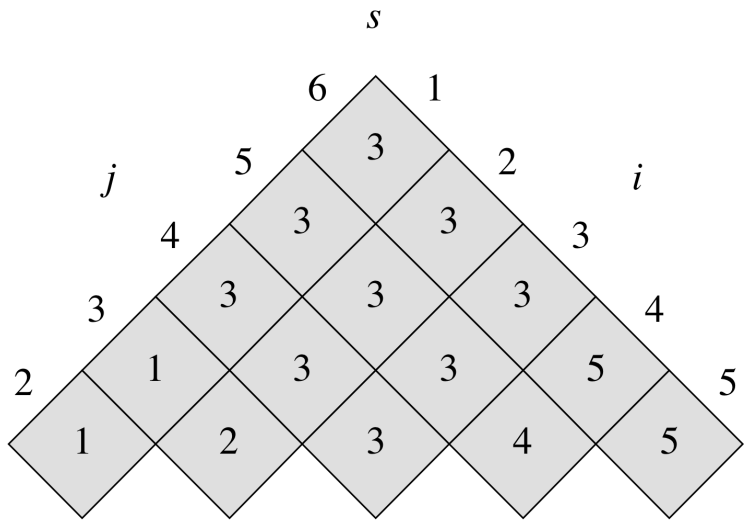


matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

- $m[2,5] =$

$$\min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$





- A simple inspection of 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 running time of this algorithm is in fact also $\Omega(n^3)$.
- The algorithm requires $\Theta(n^2)$ space to store the m and s tables.

- The total number of references for the entire table m is $\frac{n^3-n}{3}$
  each time the l-loop executes, the i-loop executes $n-l+1$ times. Each time the i-loop executes, the k-loop executes $j-i=l-1$ times, each time referencing m twice. Thus, the total number of times that an entry of m is referenced while computing other entries is $\sum_{l=2}^{n}(n-l+1)(l-1)2$ .Thus,

$$\sum_{l=2}^{n}(n-l+1)(l-1)2$$

$$=2\sum_{l=1}^{n-1}(n-l)l$$

$$=2\sum_{l=1}^{n-1}nl-2\sum_{l=1}^{n-1}l^2$$

$$=2\frac{n(n-1)n}{2}-2\frac{(n-1)n(2n-1)}{6}$$

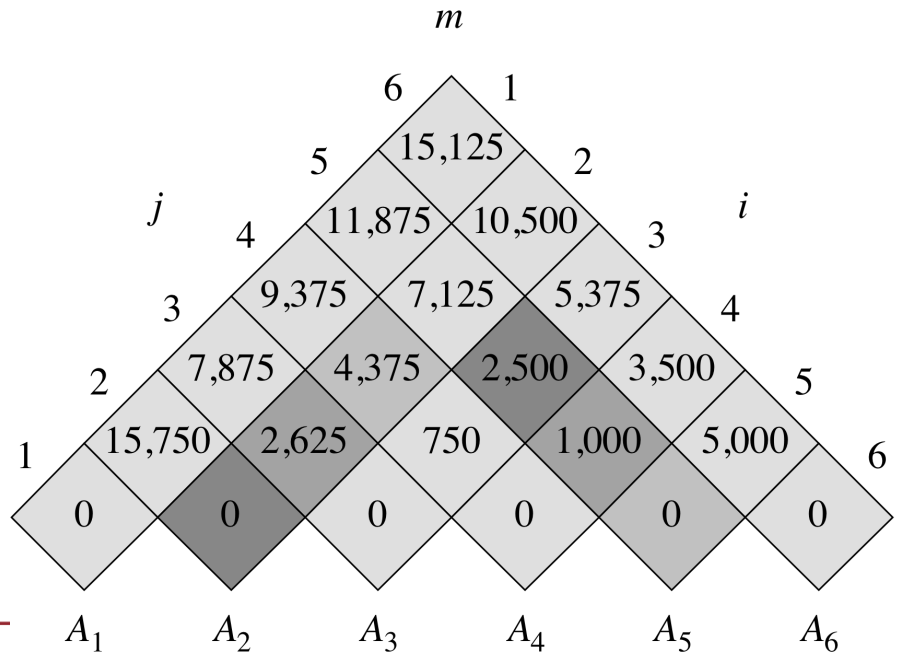
$$=n^3-n^2-\frac{2n^3-3n^2+n}{3}=\frac{n^3-n}{3}$$

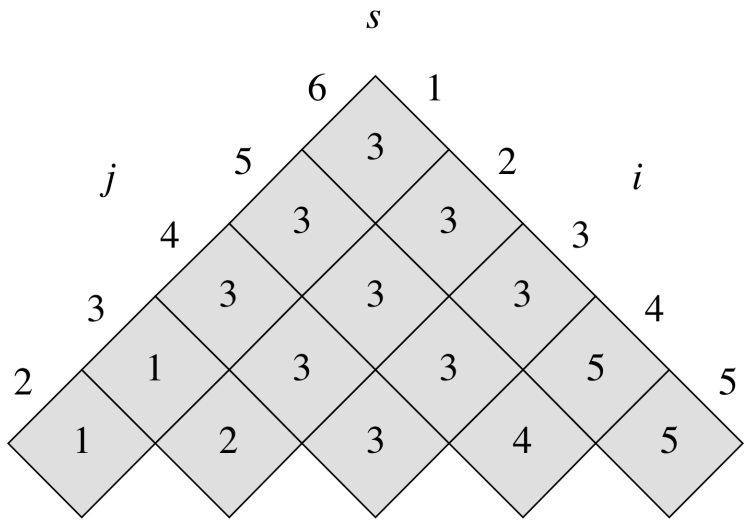
Step 4: Constructing an optimal solution

- The table $s[1..n-1, 2..n]$ gives us the information we need to do so.
- Each entry $s[i,\ j]$ records a value of k such that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}.
- Thus, we know that the final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]} A_{s[1,n]+1..n}$.
- We can determine the earlier matrix multiplications recursively, since $s[1, s[1, n]]$ determines the last matrix multiplication when computing $A_{1..s[1,n]}$ and $s[s[1, n] + 1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1..n}$.

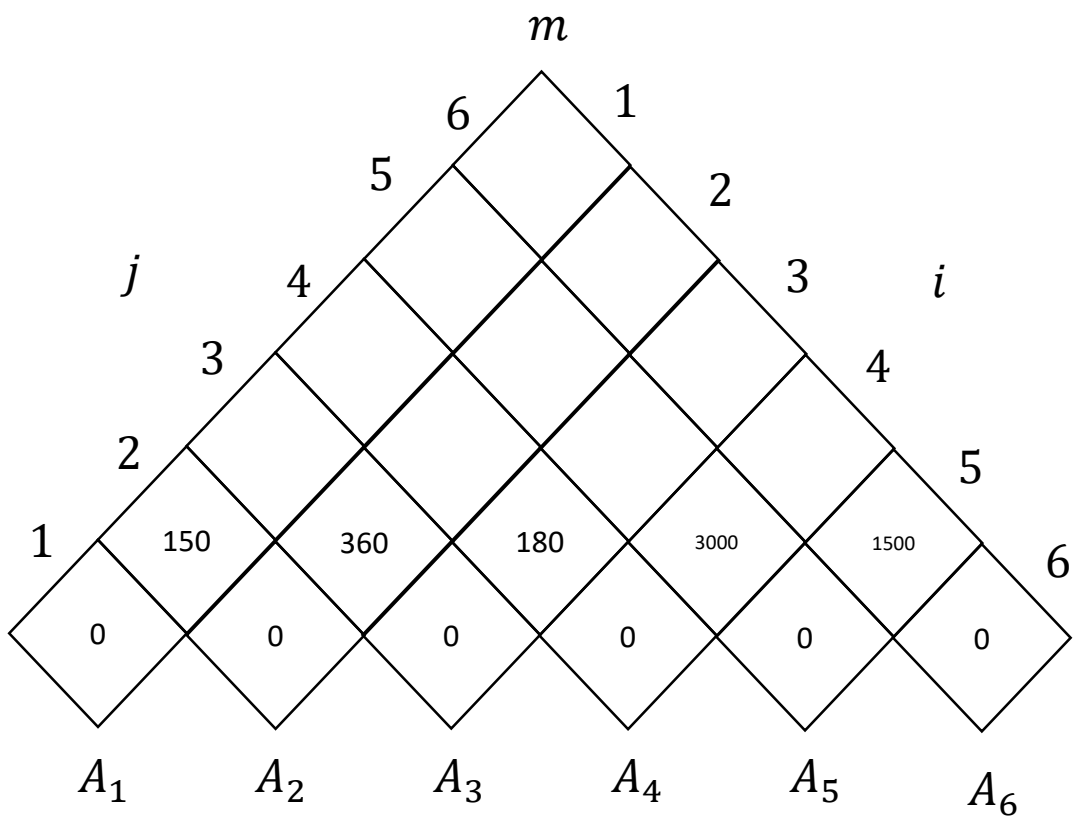
```
PRINT-OPTIMAL-PARENS(s, i, j)
1 if i == j then
2     print "A"_i
3 else
4     print "("
5     PRINT-OPTIMAL-PARENS(s, i, s[i, j])
6     PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
7     print ")"
```

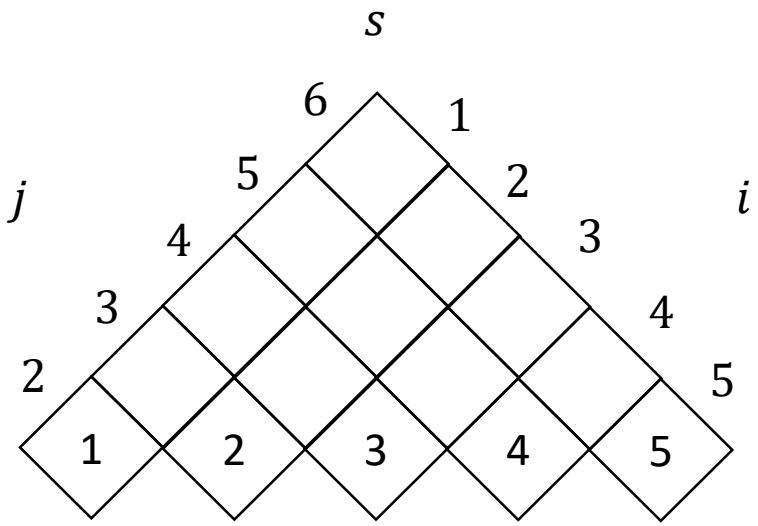
- In the example of Figure 15.5, the call PRINT-OPTIMAL-PARENS(s, 1, 6) prints the parenthesization $\big((A_1(A_2A_3))((A_4A_5)A_6)\big)$





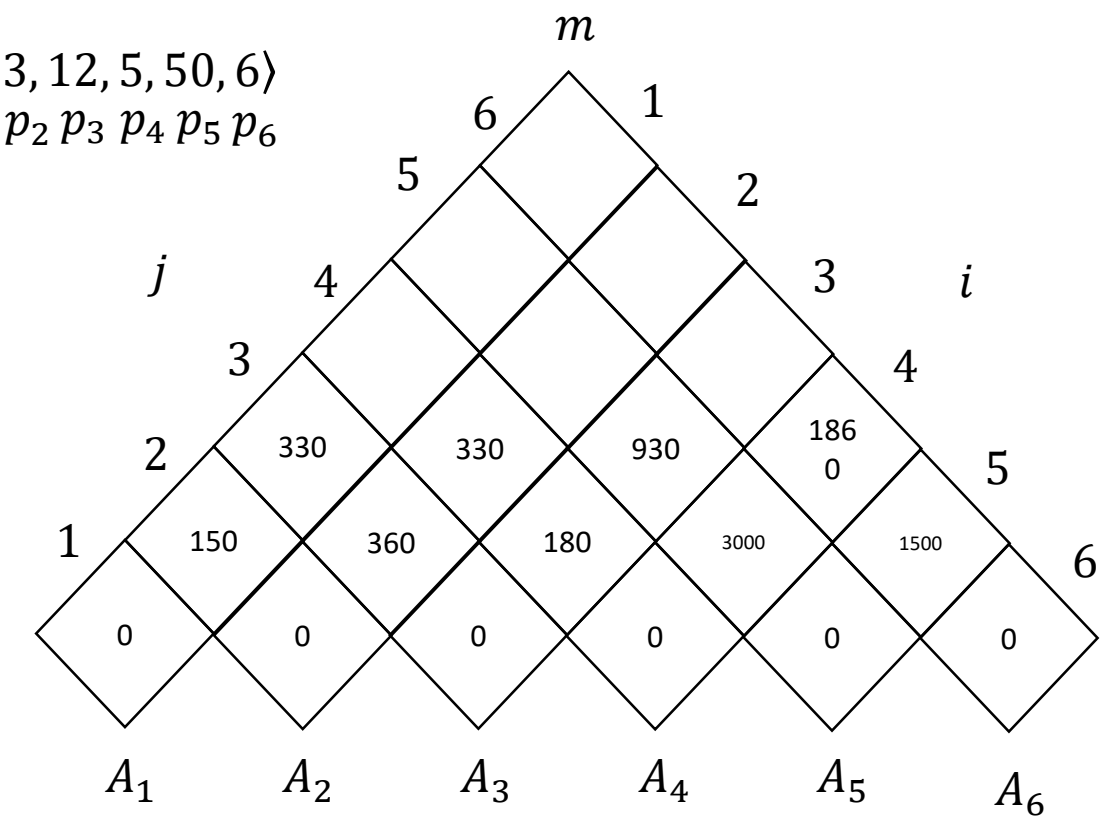
- Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

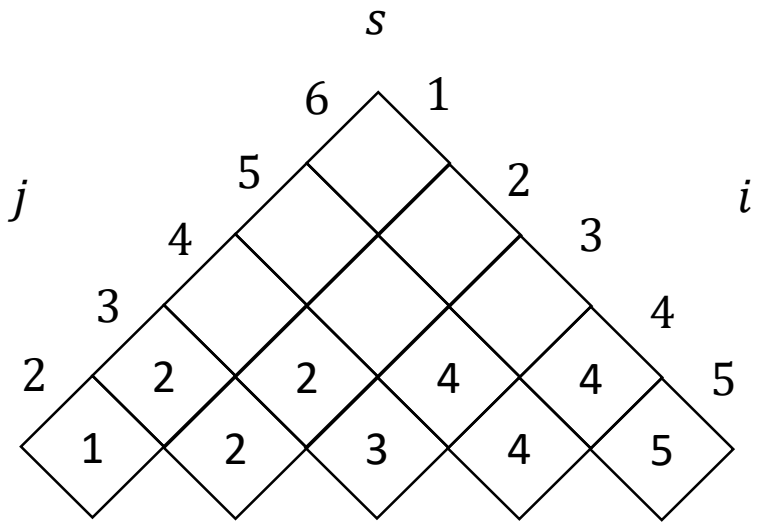




$\langle 5, 10, 3, 12, 5, 50, 6 \rangle$

$p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6$





$$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + p_0 p_1 p_3 = 0 + 360 + 5 \cdot 10 \cdot 12 = 960 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 = 150 + 0 + 5 \cdot 3 \cdot 12 = 330 \end{cases}$$

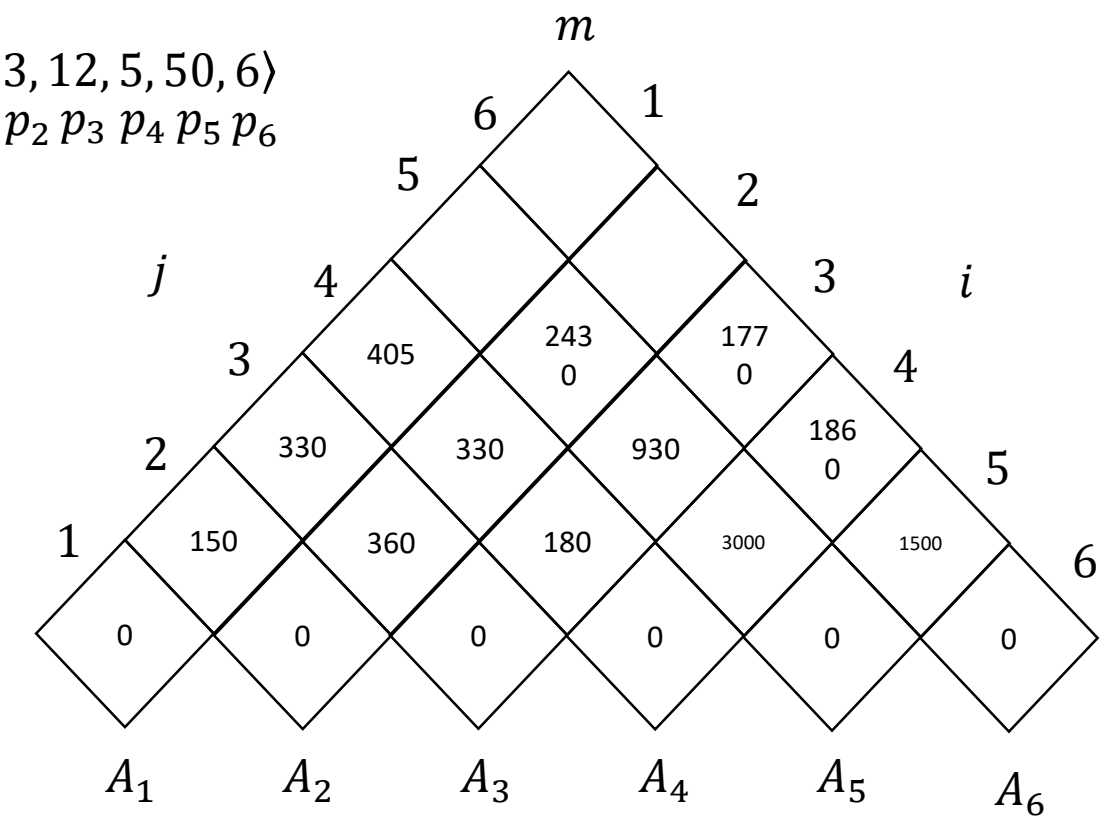
$$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + p_1 p_2 p_4 = 0 + 180 + 10 \cdot 3 \cdot 5 = 330 \\ m[2,3] + m[4,4] + p_1 p_3 p_4 = 150 + 0 + 10 \cdot 12 \cdot 5 = 750 \end{cases}$$

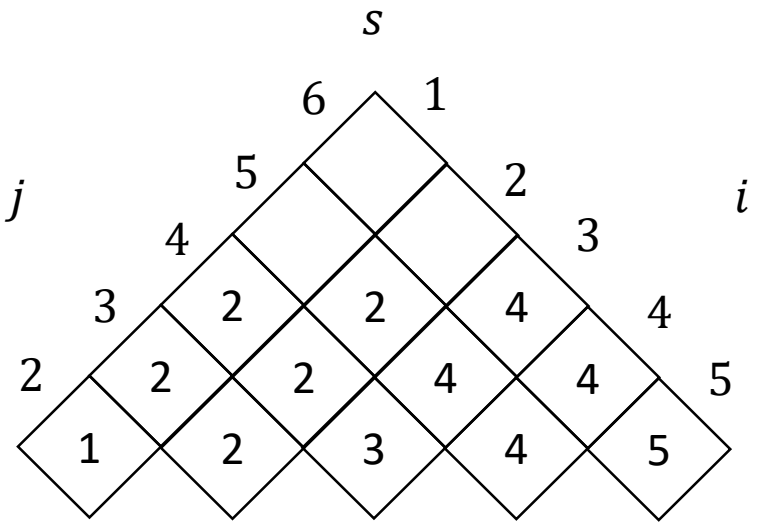
$$m[3,5] = \min \begin{cases} m[3,3] + m[4,5] + p_2 p_3 p_5 = 0 + 3000 + 3 \cdot 12 \cdot 50 = 4800 \\ m[3,4] + m[5,5] + p_2 p_4 p_5 = 180 + 0 + 3 \cdot 5 \cdot 50 = 930 \end{cases}$$

$$m[4,6] = \min \begin{cases} m[4,4] + m[5,6] + p_3 p_4 p_6 = 0 + 1500 + 12 \cdot 5 \cdot 6 = 1860 \\ m[4,5] + m[6,6] + p_3 p_5 p_6 = 3000 + 0 + 12 \cdot 50 \cdot 6 = 6600 \end{cases}$$

$\langle 5, 10, 3, 12, 5, 50, 6 \rangle$

$p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6$





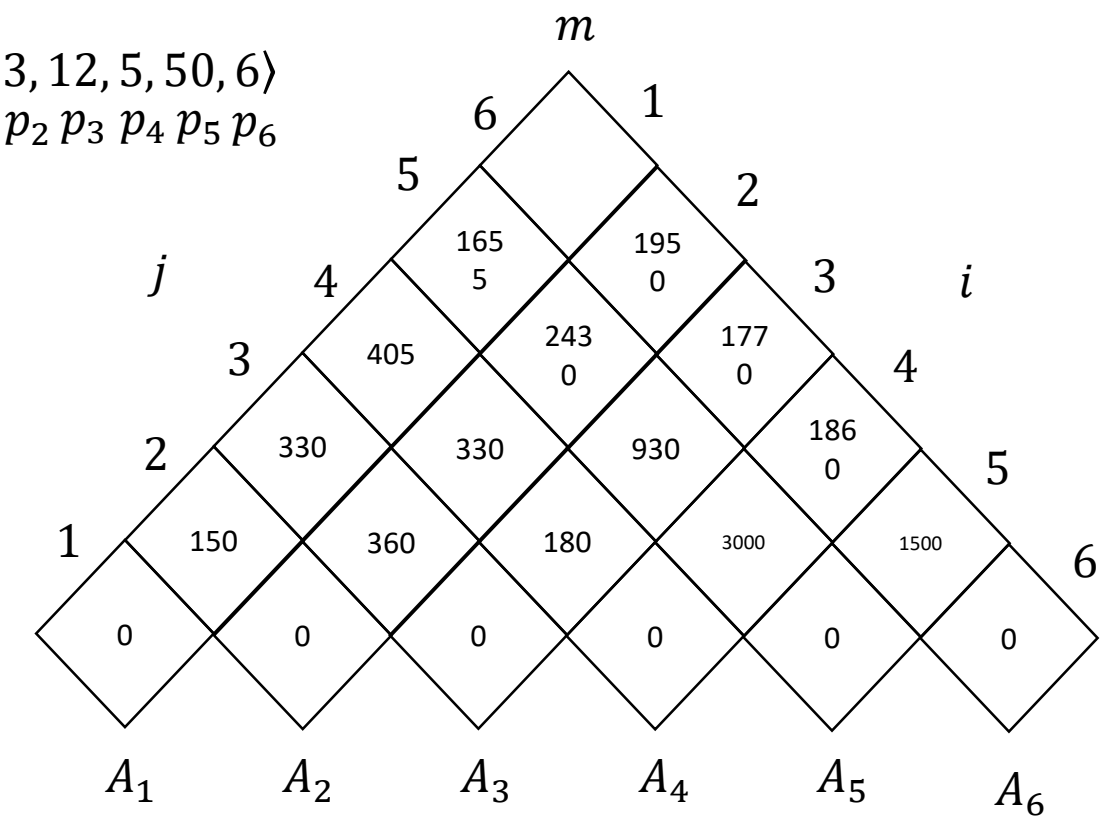
$$m[3,6] = \min \begin{cases} m[3,3] + m[4,6] + p_2 p_3 p_6 = 0 + 1860 + 3 \cdot 12 \cdot 6 = 2076 \\ m[3,4] + m[5,6] + p_2 p_4 p_6 = 180 + 1500 + 3 \cdot 5 \cdot 6 = 1770 \\ m[3,5] + m[6,6] + p_2 p_5 p_6 = 930 + 0 + 3 \cdot 50 \cdot 6 = 1830 \end{cases}$$

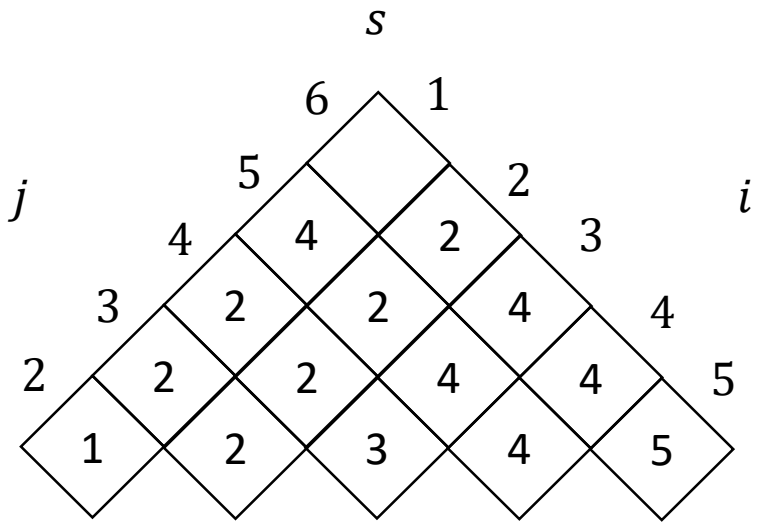
$$m[1,4] = \min \begin{cases} m[1,1] + m[2,4] + p_0 p_1 p_4 = 0 + 330 + 5 \cdot 10 \cdot 5 = 580 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 = 150 + 180 + 5 \cdot 3 \cdot 5 = 405 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 = 330 + 0 + 5 \cdot 12 \cdot 5 = 630 \end{cases}$$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 930 + 10 \cdot 3 \cdot 50 = 2430 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 360 + 3000 + 10 \cdot 12 \cdot 50 = 9360 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 330 + 0 + 10 \cdot 5 \cdot 50 = 2830 \end{cases}$$

$\langle 5, 10, 3, 12, 5, 50, 6 \rangle$

$p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6$



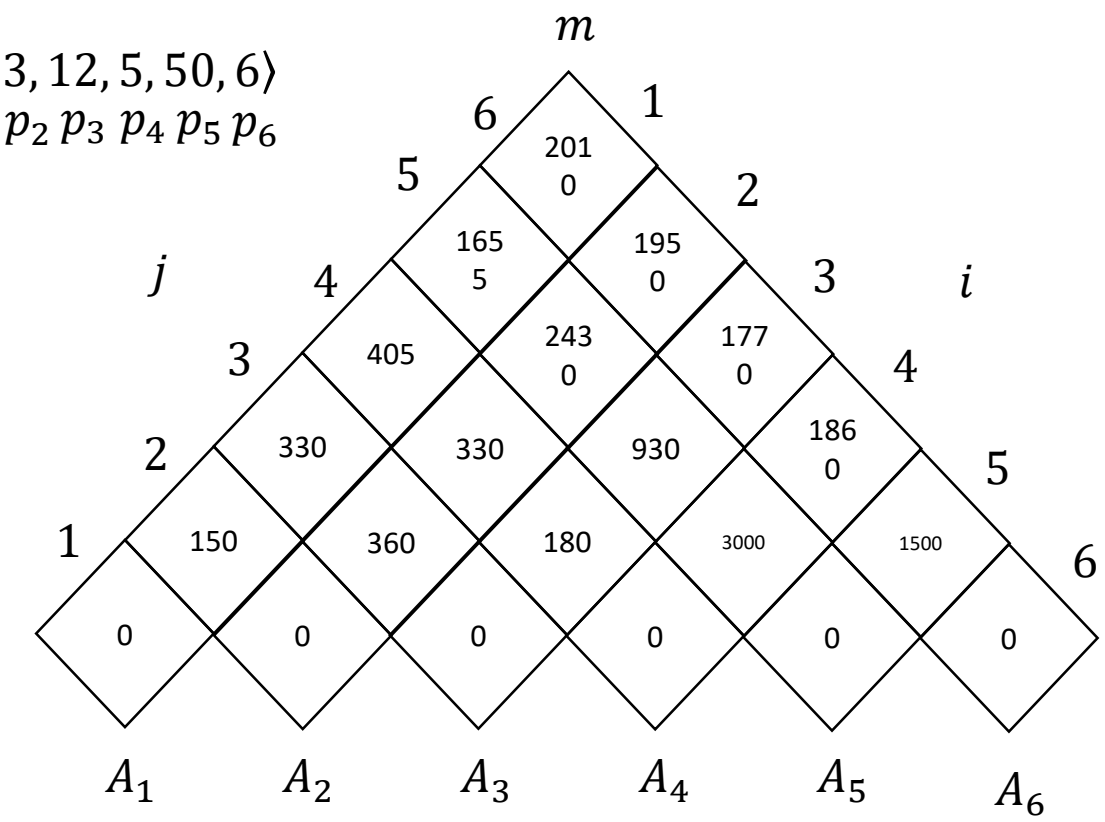


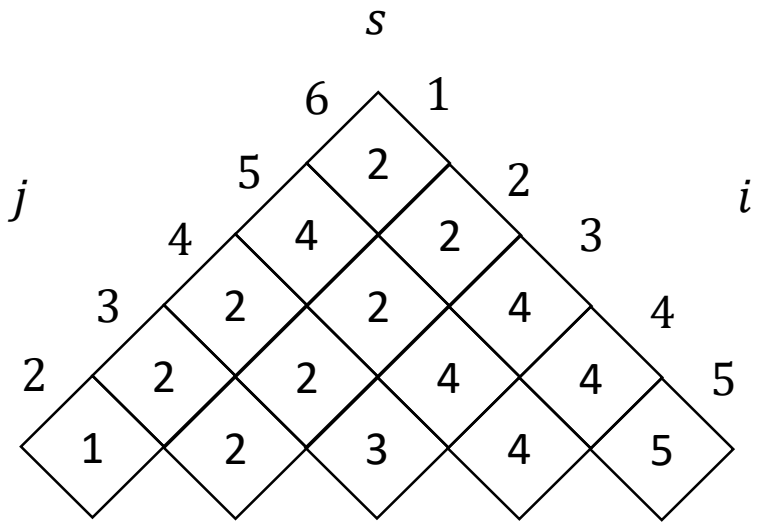
$$m[1,5] = \min \begin{cases} m[1,1] + m[2,5] + p_0 p_1 p_5 = 0 + 2430 + 5 \cdot 10 \cdot 50 = 4930 \\ m[1,2] + m[3,5] + p_0 p_2 p_5 = 150 + 930 + 5 \cdot 3 \cdot 50 = 1830 \\ m[1,3] + m[4,5] + p_0 p_3 p_5 = 330 + 3000 + 5 \cdot 12 \cdot 50 = 6330 \\ m[1,4] + m[5,5] + p_0 p_4 p_5 = 405 + 0 + 5 \cdot 5 \cdot 50 = 1655 \end{cases}$$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,6] + p_1 p_2 p_6 = 0 + 1770 + 10 \cdot 3 \cdot 6 = 1950 \\ m[2,3] + m[4,6] + p_1 p_3 p_6 = 360 + 1860 + 10 \cdot 12 \cdot 6 = 2940 \\ m[2,4] + m[5,6] + p_1 p_4 p_6 = 330 + 1500 + 10 \cdot 5 \cdot 6 = 2130 \\ m[2,5] + m[6,6] + p_1 p_5 p_6 = 2430 + 0 + 10 \cdot 50 \cdot 6 = 5430 \end{cases}$$

$\langle 5, 10, 3, 12, 5, 50, 6 \rangle$

$p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6$

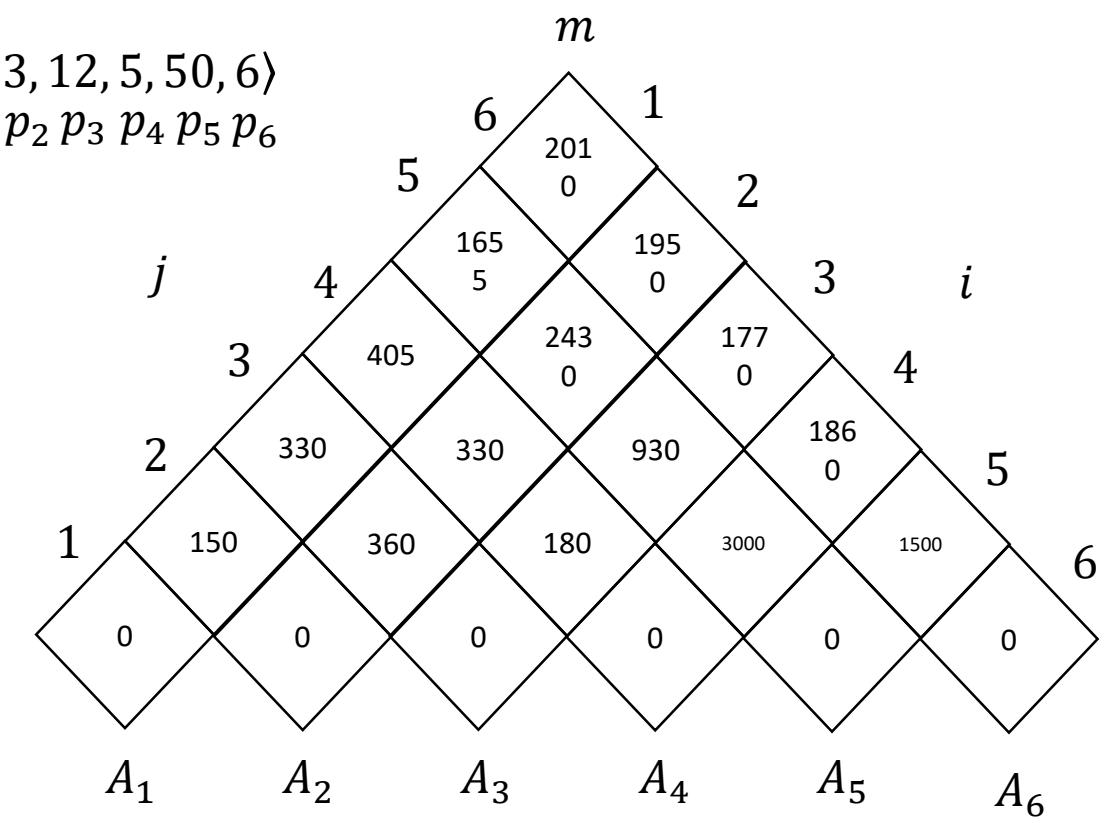


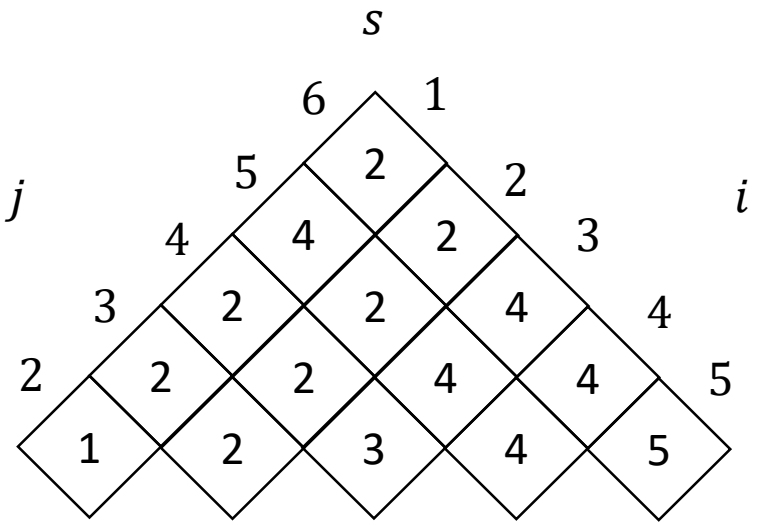


$$m[1,6] = \min \begin{cases} m[1,1] + m[2,6] + p_0 p_1 p_6 = 0 + 1950 + 5 \cdot 10 \cdot 6 = 2250 \\ m[1,2] + m[3,6] + p_0 p_2 p_6 = 150 + 1770 + 5 \cdot 3 \cdot 6 = 2010 \\ m[1,3] + m[4,6] + p_0 p_3 p_6 = 330 + 1860 + 5 \cdot 12 \cdot 6 = 2550 \\ m[1,4] + m[5,6] + p_0 p_4 p_6 = 405 + 1600 + 5 \cdot 5 \cdot 6 = 2185 \\ m[1,5] + m[6,6] + p_0 p_5 p_6 = 1655 + 0 + 5 \cdot 50 \cdot 6 = 3155 \end{cases}$$

$\langle 5, 10, 3, 12, 5, 50, 6 \rangle$

$p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6$





$$(A_1A_2A_3A_4A_5A_6)$$

$$\Rightarrow \big((A_1A_2)(A_3A_4A_5A_6)\big)$$

$$\Rightarrow \Big((A_1A_2)\big((A_3A_4)(A_5A_6)\big)\Big)$$

$\langle 5, 10, 3, 12, 5, 50, 6 \rangle$
$p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6$

$$(A_1A_2A_3A_4A_5A_6)$$
$$\Rightarrow \big((A_1A_2)(A_3A_4A_5A_6)\big)$$
$$\Rightarrow \Big((A_1A_2)\big((A_3A_4)(A_5A_6)\big)\Big)$$

$(A_1A_2) \Rightarrow 5 \cdot 10 \cdot 3 = 150$

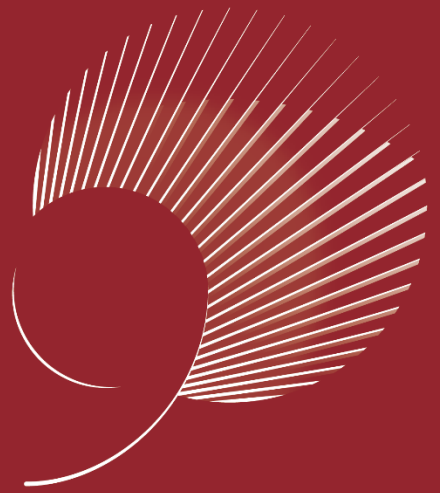
$(A_3A_4) \Rightarrow 3 \cdot 12 \cdot 5 = 180$

$(A_5A_6) \Rightarrow 5 \cdot 50 \cdot 6 = 1500$

$(A_3A_4)(A_5A_6) \Rightarrow p_2p_4p_6 = 3 \cdot 5 \cdot 6 = 90$

$(A_1A_2)\big((A_3A_4)(A_5A_6)\big) \Rightarrow p_0p_2p_6 = 5 \cdot 3 \cdot 6 = 90$

$150 + 180 + 1500 + 90 + 90 = 2010$



藏行顯光
成就共好

Achieve Securely
Prosper Mutually

國立成功大學 九十週年
90th Anniversary of NCKU

國立成功大學
National Cheng Kung University
1931