Lakehead
UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

Jan. 30, 2023 (Lecture 5)

Lakehead UNIVERSITY

Dynamic Programming

Introduction

- Dynamic programming is a way of improving on inefficient divide-and-conquer algorithms.
- By “inefficient”, we mean that the same recursive call is made over and over.
- If same subproblems is solved several times, we can use **table** to store result of a subproblem the first time it is computed and thus never have to recompute it again.
- Dynamic programming is applicable when the subproblems are dependent, that is, when subproblems share sub-sub-problems.
- “Programming” refers to a tabular method.

Example 1: Cut Rod

Rod Cutting Example

- **Rod cutting problem**

The rod cutting problem is the following. Given a rod of length n inches and a table of price p_i for i = 1, . . . , n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. The following is an example of price table.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	19	17	17	20	24	30

Rod Cutting Example

- **Rod cutting problem**

For n = 4, we may cut as: (1, 1, 1, 1),(1, 1, 2),(2, 2),(1, 3),(4),
the correspondent prices are: 4, 7, 10, 9, 9, respectively.

• So the optimal revenue is cutting the 4-inch rod into two 2-inch pieces.

Rod Cutting Example

By inspection, we can obtain the optimal decomposition as follows.

r_1 = 1 from solution 1 = 1 (no cuts)
r_2 = 5 from solution 2 = 2 (no cuts)
r_3 = 8 from solution 3 = 3 (no cuts)
r_4 = 10 from solution 4 = 2 + 2
r_5 = 13 from solution 5 = 2 + 3
r_6 = 17 from solution 6 = 6 (no cut)
r_7 = 18 from solution 7 = 1 + 6 or 7 = 2 + 2 + 3
r_8 = 22 from solution 8 = 2 + 6
r_9 = 25 from solution 9 = 3 + 6
r_{10} = 30 from solution 10 = 10 (no cuts)

Rod Cutting Example

In general, for a rod of length n, we can consider 2^{n-1} different cutting ways, since we have an independent option of cutting or not cutting at distance i inches from one end.

Suppose an optimal solution cuts the rod into k pieces with lengths $i_1, i_2, \ldots, i_k$ that

$$n = i_1 + i_2 + \cdots + i_k$$

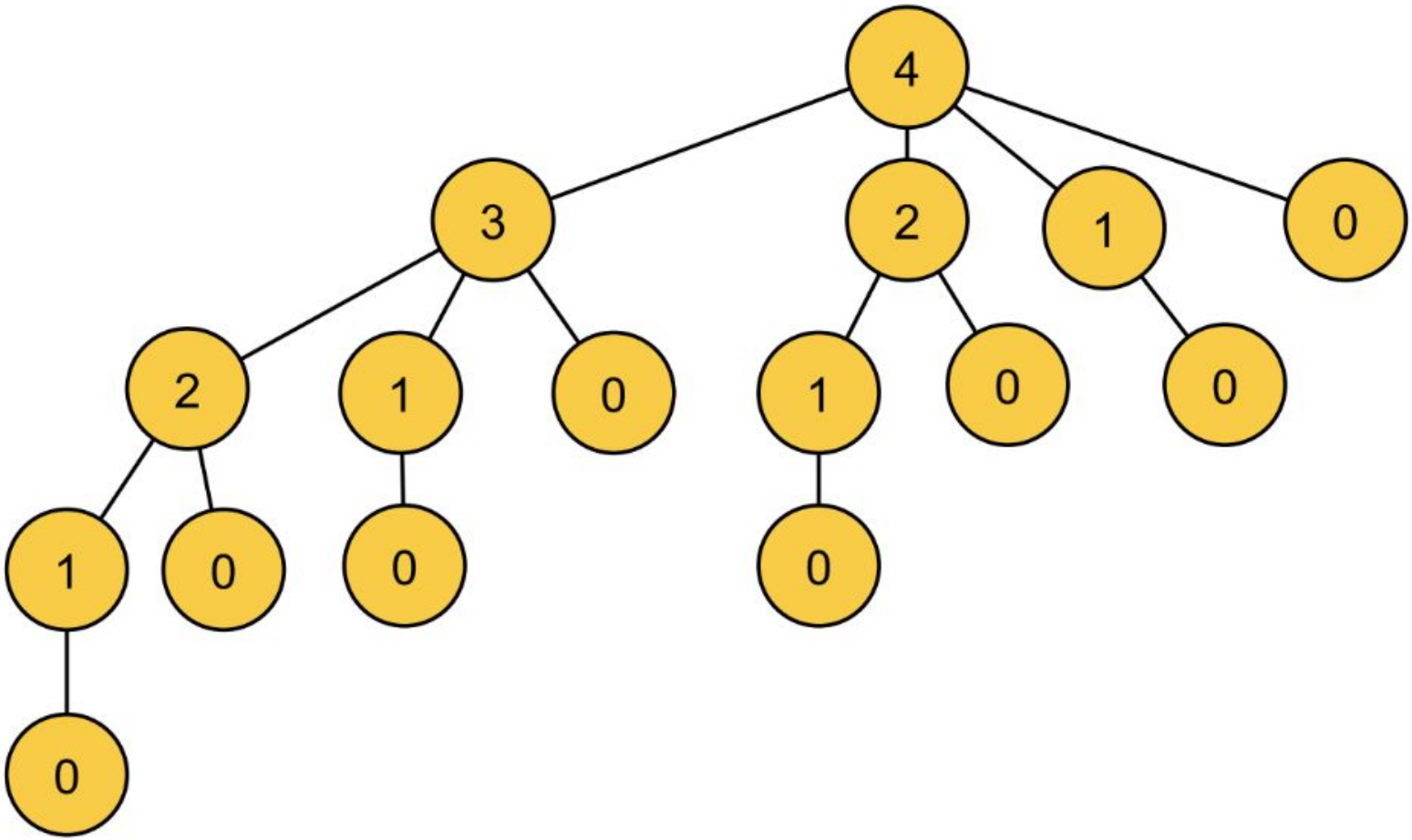
and the corresponding optimal revenue is

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}.$$

One approach:

```
procedure 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
```

It is inefficient and why?


4
3
2
1
0
2
1
0
1
0
0
1
0
0
0
0

From the recursion tree, we see that the same subproblem is computed again and again.

- In this example, Cut-Rod(p, 1) computed 4 times, Cut-Rod(p, 0) computed 8 times, etc.
- To improve the method, we will use the **dynamic-programming** method.
- The main idea of the dynamic-programming is to arrange for each subproblem to be solved only once.
  - Each time a subproblem is solved, the result will be stored for the next calling. So next time, when we need to solve this subproblem, we need just look it up.
  - Dynamic-programming uses additional memory to save the computation time.

Dynamic Programming Approaches

There are two approaches.

- The **first** approach is **top-down with memoization**. In this approach, the procedure runs recursively in a nature manner, but modified to save the result of each subproblem (in an array or hash table). The procedure now first checks to see if the subproblem has previous solved or not. If so, it just returns the saved result; if not, the procedure computes the result in the usual manner and returns and saves the result.

Top-Down Pseudocode - cut rod

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

```
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
```

Dynamic Programming Approaches

There are two approaches.

- The **second** approach is the **bottom-up method**. This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems. We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

Bottom-up Pseudocode - cut rod

```
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]
```

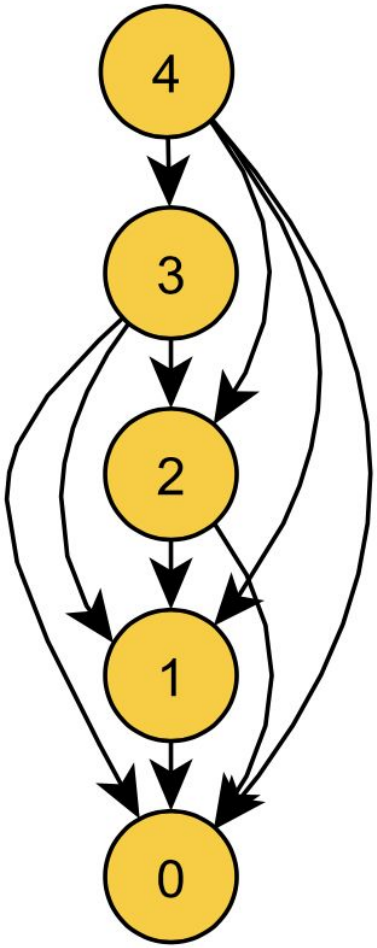
- When compute r[j], all the values of r[j − i] have been computed. Therefor the line 6 just use these values instead of using recursive callings (directly using for loop instead of recursive).

Running Time Analysis

- The running time of the Bottom-Up-Cut-Rod is $\Theta(n^2)$, because there is a double-nested for loop.
- The running time of the top-down approach is also $\Theta(n^2)$.
- Although the line 10 of Memoized-Cut-Rod-Aux uses recursive calling, each value of r[i] just computes once. Therefore the total number of iterations of its for loop forms an arithmetic series, which gives total of $\Theta(n^2)$ iterations.

Subproblem Graph

- When we think about a dynamic programming problem, it is important for us to understand the set of subproblems involved and how they depend on one another.



Subproblem Graph

- The subproblem graph is a digraph, in which each vertex represents a distinct subproblem, and each arc represents that an optimal solution of a subproblem needs the solution of the other subproblem.
- For the top-down approach, the graph shows the vertex 4 needs a solution of vertex 3, the vertex 3 needs a solution of 2, etc.
- The bottom-up approach first solves the vertex 1 from vertex 0, then solves vertex 2 from vertices 0 and 1, etc.

Subproblem Graph

- The size of the subproblem graph can help us determine the running time of the dynamic programming algorithm.
- Since each subproblem is solved only once, the running time is the sum of the times needed to solve each subproblem.
- Typically, the time to compute the solution to a subproblem is proportional to the degree (number of outgoing edges) of the corresponding vertex in the subproblem graph, and the number of subproblems is equal to the number of vertices in the subproblem graph.
- In this common case, the running time of dynamic programming is linear in the number of **vertices** and **edges**.

Extended Algorithm of Cut Rod

- The above dynamic programming solutions of the cut rod problem just give the value of the optimal revenue, but not the actual solutions (how to cut the rod).

```
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
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

Extended Algorithm of Cut Rod

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

EXTENDED-CUT-ROD-SOLUTION(p, 10) would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Dynamic Programming Hallmarks (Example 2: Fibonacci Numbers)

Elements of Dynamic Programming

There are two key ingredients that an optimization problem must have to apply dynamic programming: **optimal substructure** and **overlapping subproblems**.

➔ Optimal substructure means some algorithm that solves the optimal problem will depends on some optimal solutions of the subproblems. So we need not only to find some recursive method, but the recursive also on optimal problems.

Elements of Dynamic Programming

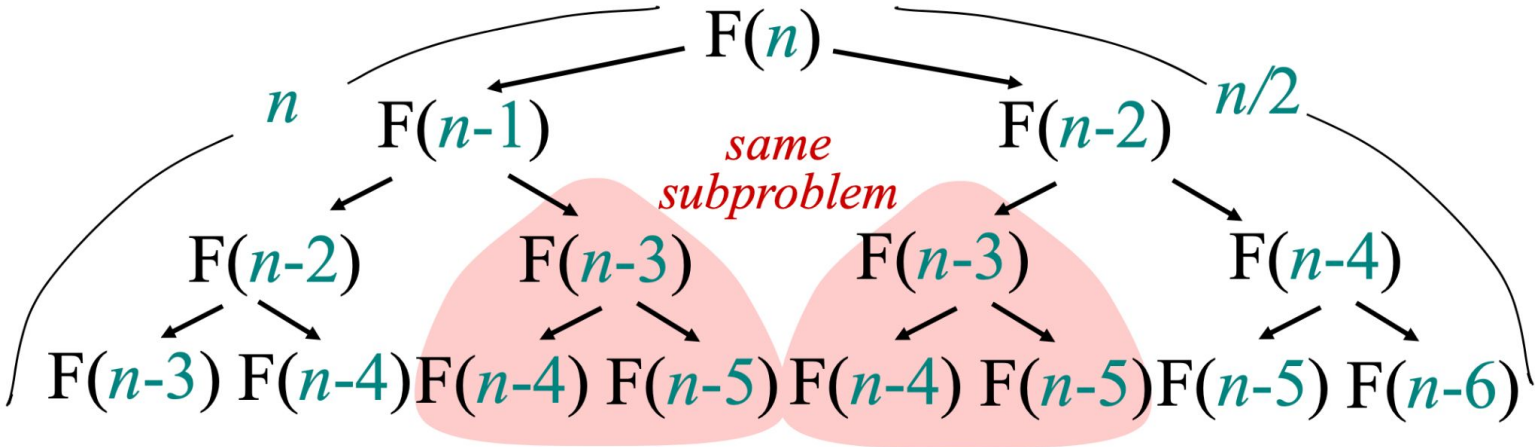
- The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be “small” in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems (**Overlapping subproblems**). Typically, the total number of distinct subproblems is a polynomial in the input size.

  In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of recursion.

Fibonacci Numbers

F(0)=0; F(1)=1; F(*n*)=F(*n*-1)+F(*n*-2) for *n*>1

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...



DP hallmark #1: Optimal substructure

Fibonacci Numbers

Implement this recursion directly:

- Runtime is exponential: $2^{n/2} \leq T(n) \leq 2^n$
- But we are repeatedly solving the same subproblems

DP hallmark #2: Overlapping subproblems

Fibonacci Numbers - bottom up

- The number of distinct Fibonacci subproblems is only n.
- Store 1D DP-table and fill bottom-up

F:	0	1	1	2	3	5	8				

```
fibBottomUpDP(n)
    F[0] ← 0
    F[1] ← 1
    for (i ← 2, i≤n, i++)
            F[i] ← F[i-1]+F[i-2]
    return F[n]
```

- Time $= \Theta(n)$, space $= \Theta(n)$

Fibonacci Numbers – Memorize

Memoization (top-down): Use recursive algorithm. After computing Subsequent calls check the table to avoid redoing work. a solution to a subproblem, store it in a table.

```
fibMemoization(n)
    for all i: F[i] = null
    fibMemoizationRec(n,F)
    return F[n]

fibMemoizationRec(n,F)
    if (F[n]= null)
            if (n=0) F[n] ← 0
            if (n=1) F[n] ← 1
            F[n] ← fibMemoizationRec(n-1,F)
                   + fibMemoizationRec(n-2,F)
    return F[n]
```

- Time = $\Theta(n)$, space = $\Theta(n)$

Example 3:

Matrix-Chain Multiplication Problem

Matrix-Chain Multiplication

- Suppose n matrices $A_1, A_2, \cdots, A_n$ are given, where the matrices are not necessary square. We need to compute the product $A_1A_2 \cdots A_n$.
- 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.
- For example, if the chain of matrices is $<A_1, A_2, A_3, A_4>$, then we can fully parenthesize the product $A_1A_2A_3A_4$ in five distinct ways:

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

  $((A_1(A_2A_3))A_4)$, $(((A_1A_2)A_3))A_4)$

Why Care about Parenthesization?

- Consider first the cost of multiplying two matrices.

```
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 main cost of the procedure is from line 4 to line 8, which is A.rows × B.columns × A.columns scalar multiplications.

Why Care about Parenthesization?

- When we compute the multiplication of more than two matrices, the order of the multiplication will affect the cost.
- Consider the multiplication $A_1A_2A_3$. Suppose the dimensions of A_1, A_2, A_3 are $10\times100, 100\times5, 5\times50$, respectively.
  - If we multiply according to $((A_1A_2)A_3)$, then we first perform $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute a 10×5 matrix. Then multiply the resulting matrix with A_3, which needs $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications. In this way, we need total 7500 multiplications.
  - But if we compute the multiply as $(A_1(A_2A_3))$, then a simple calculation shows that we need a total 75000 scalar multiplications.

Matrix-chain Multiplication Problem

Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for i = 1,2,...,n, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

This problem is to find out an optimal order of products for a matrix-chain multiplications, not actually multiplying matrices.

Number of Possible Parenthesizations

Denote the number of alternative parenthesizations of a sequence of n matrices by P(n). Then P(1) = 1. When n ≥ 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, . . . , n − 1. Thus we have

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

Number of Possible Parenthesizations

Using the substitution method, we can show that the solution to the recurrence in the previous slide is $\Omega(2^n)$. So an exhaustive search will be exponential in n.

We can see that in the recurrence, many value of P(k) is computed repeatedly. Therefore we can apply the dynamic programming.

Dynamic Programming

Step 1: The structure of an optimal parenthesization

- Let $A_{i..j}$, where $i \le j$, denote the matrix that results from evaluating the product $A_iA_{i+1} \cdot \cdot \cdot A_j$. When $i < j$, we need to split the product into two products and compute $A_{i..k}$ and $A_{k+1..j}$ for some $i \le k < j$, and then compute $A_{i..k}A_{k+1..j}$.
- The cost thus is the sum of the costs of compute $A_{i..k}$, $A_{k+1..j}$ and $A_{i..k}A_{k+1..j}$.

The optimal substructure of this problem is as follows.

Suppose that to optimally parenthesize $A_iA_{i+1} \cdot \cdot \cdot A_j$, we split the product between A_k and A_{k+1}. Then the subchain $A_iA_{i+1} \cdot \cdot \cdot A_k$ within this parenthesize must be optimal.

Otherwise, if we have a better parenthesize of $A_iA_{i+1} \cdot \cdot \cdot A_k$, then we can get a better parenthesize of $A_iA_{i+1} \cdot \cdot \cdot A_j$. By the similar argument, the parenthesize of $A_{k+1}A_{k+2} \cdot \cdot \cdot A_j$ is also optimal.

We can split the matrix-chain problem into two subproblems and find out the optimal solutions of these two subproblems.

Step 2: A recursive solution

➔ Let m[i, j] be the minimum number of scalar multiplications needed to compute $A_i A_{i+1} \cdots A_j$.

➔ We want to build m[i, j] recursively.

If we split $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, then the minimum costs will be $m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$

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

- The m[i, j] value gives the costs of optimal solution to problems, but they do not provide the construction of the optimal solution.
- We need to know the value of k which we used to split the product.
- We define s[i, j] to be a value of k at which we split the product $A_iA_{i+1} \cdot \cdot \cdot A_j$ in an optimal parenthesization.

Step 3: Computing the optimal costs

- From the recurrence, now the task is to find out the value m[1, n], which depends on m[i, j] for smaller chains with length j − i.
- So it is suitable to use the bottom-up method, i.e., start from computing m[i, j] for smaller l = j − i.
- Instead of use a recursive algorithm based on recurrence, we use a tabular, bottom-up method to compute the costs.

- We shall implement the tabular, bottom-up method in the procedure MATRIX-CHAIN-ORDER, which appears in the next slide.
- This procedure assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ 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 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].

```
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
```

- The main cost for the procedure is the three nested for loops, which yields a running time of $O(n^3)$.
- We can prove that the running time is also $\Omega(n^3)$.
- The space requirement is $\Theta(n^2)$.

Step 4: Constructing an optimal solution

➔ Now we are able to give the optimal solution (the optimal parenthesizing the chain), because we have determined the value k in s[i, j].

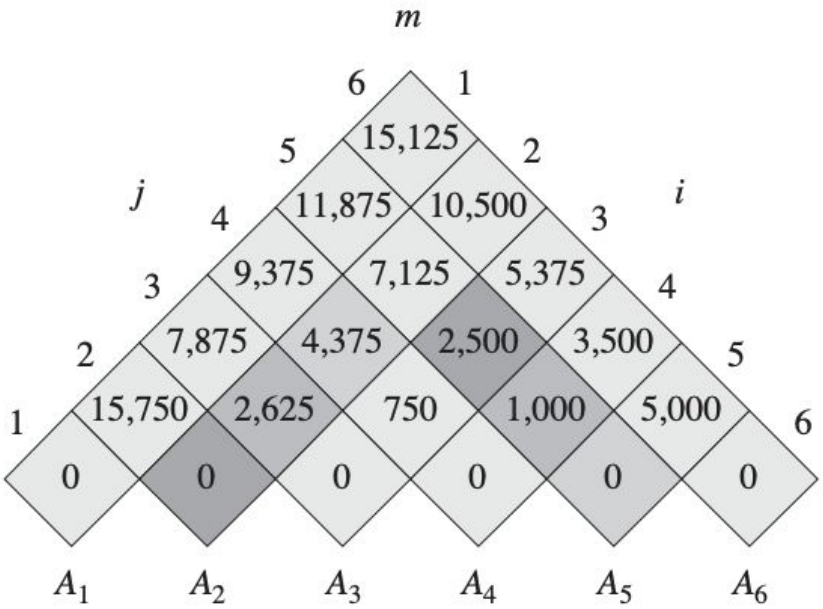
```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

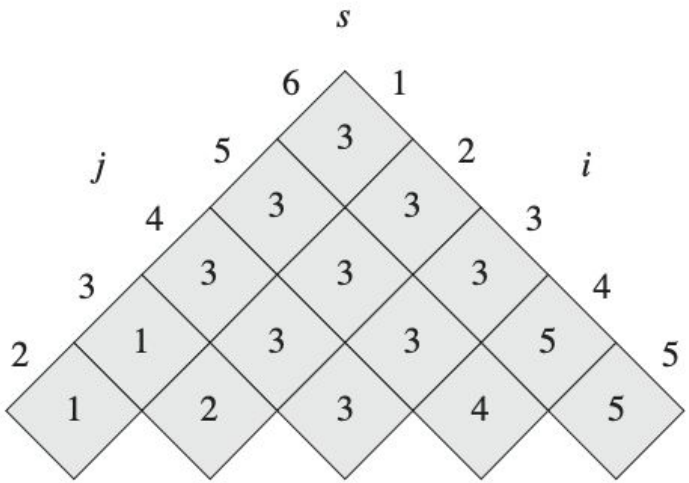
In the example, the call PRINT-OPTIMAL-PARENS(s, 1, 6) prints the parenthesization ((A1(A2A3))((A4A5)A6)).

An Example

Example: Suppose the following matrix-chain is given (p given)

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





An 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
	p0 p1	p2	p3	p4	p5	p6

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 &= 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 &= 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 &= 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11{,}375 \end{cases} \\
&= 7125\,.
\end{aligned}
$$

Call Matrix-Chain-Order(p) and Print-Optimal-Parens(s, 1, 6) prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

After Class and Next Class

- After class:

  read Part IV Chapter 15.1-15.3

- Next class – more examples of DP
  - Longest common subsequence
  - Optimal binary search trees