MATRIXCHAIN-ORDER

- The running time of MATRIXCHAIN-ORDER is $\Omega(n^3)$.
- The algorithm requires $\Theta(n^2)$ space to store the m and s tables.

➡ MATRIX-CHAINORDER is much more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one.

Step 4: Constructing an Optimal Solution (1/3)

- The table $s[1..n-1, 2..n]$gives us the information we need to show how to multiply the matrices.
  - Each entry $s[i, j]$ records a value of k such that an optimal parenthesization of $A_i A_{i+1} \ldots A_j$ splits the product between A_k and A_{k+1}.
  - The final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]} A_{s[1,n]+1..n}$.
  - $s[1, s[1, n]]$ 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 $A_{s[1,n]+1..n}$.

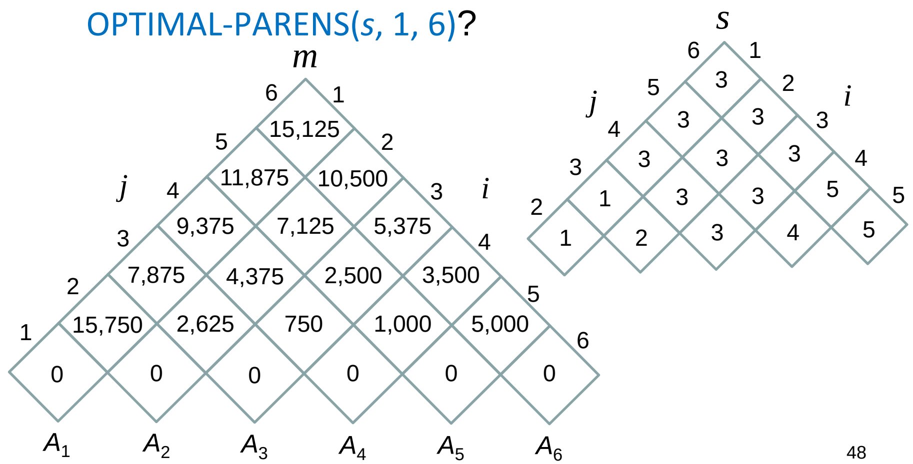
Step 4: Constructing an Optimal Solution (2/3)

- The initial call PRINT-OPTIMAL-PARENS(s, 1, n) prints an optimal parenthesization of $<A_1, A_2, \ldots, A_n>$.

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


Step 4: Constructing an Optimal Solution (3/3)

- What would be printed when we call PRINT-OPTIMAL-PARENS(s, 1, 6)?



Outine

- Rod Cutting
- Matrix-Chain Multiplication
- ***Elements of Dynamic Programming***
- Longest Common Subsequence
- Optimal Binary Search Trees

Elements of Dynamic Programming

- When should we look for a dynamic-programming solution to a problem?
- ➡ Two key ingredients that an optimization problem must have in order for dynamic programming to apply: optimal substructure and overlapping subproblems.

Optimal Substructure (1/4)

- The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution.
- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.
- ➡ Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply.
  - It also might mean that a greedy strategy applies, however.

Optimal Substructure (2/4)

- Optimal substructure in both of the problems we have examined so far:
  - The optimal way of cutting up a rod of length n involves optimally cutting up the two pieces resulting from the first cut.
  - An optimal parenthesization of $A_i A_{i+1} \ldots A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problems of parenthesizing $A_i A_{i+1} \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_j$.

Optimal Substructure (3/4)

- Optimal substructure varies across problem domains in two ways:

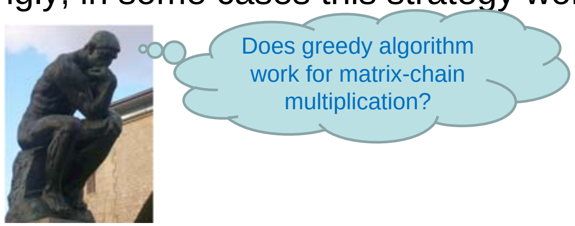
1. how many subproblems an optimal solution to the original problem uses, and
2. how many choices we have in determining which subproblem(s) to use in an optimal solution.

Optimal Substructure (4/4)

- In the rod-cutting problem, an optimal solution for cutting up a rod of size n uses just one subproblem (of size $n - i$), but we must consider n choices for i in order to determine which one yields an optimal solution.
- Matrix-chain multiplication for the subchain $A_i A_{i+1} \ldots A_j$ serves as an example with two subproblems and $j - i$ choices.

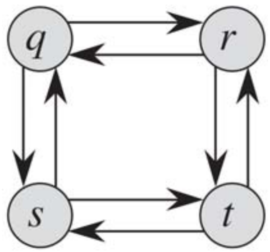
Dynamic Programming vs. Greedy

- Instead of first finding optimal solutions to subproblems and then making an informed choice, greedy algorithms first make a "greedy" choice—the choice that looks best at the time—and then solve a resulting subproblem, without bothering to solve all possible related smaller subproblems.
- Surprisingly, in some cases this strategy works!



Subtleties (1/3)

- You should be careful not to assume that optimal substructure applies when it does not.
- Unweighted longest simple path: Find a simple path from u to v consisting of the most edges.



Consider the path $q \to r \to t$, which is a longest simple path from q to t.
Is $q \to r$ a longest simple path from q to r?
Is $r \to t$ a longest simple path from r to t?

Subtleties (2/3)

- Why is the substructure of a longest simple path so different from that of a shortest path?

☞ Although a solution to a problem for both longest and shortest paths uses two subproblems, the subproblems in finding the longest simple path are not independent, whereas for shortest paths they are.
  - For the vertices used in the first subproblem can no longer be used in the second problem, since the combination of the two solutions would yield a path that is not simple.

Subtleties (3/3)

- Why, then, are the subproblems independent for finding a shortest path?
  - We claim that if a vertex w is on a shortest path p from u to v, then we can splice together any shortest path $u \overset{p_1}{\leadsto} w$ and any shortest path $w \overset{p_2}{\leadsto} v$ to produce a shortest path from u to v.
  - We are assured that, other than w, no vertex can appear in both paths p_1 and p_2. Why? (Hint: by contradiction)

Overlapping Subproblems (1/2)

- 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.

➡ When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has overlapping subproblems.

Overlapping Subproblems (2/2)

- In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion.
- To illustrate the overlapping-subproblems property in greater detail, let us reexamine the matrix-chain multiplication problem.