Topic 3:
Dynamic Programming

Dynamic Programming

- Dynamic program is not a specific algorithm, but a technique (like divide-and-conquer).
- Dynamic programming, like divide-and-conquer, solves problems by combining the solutions to subproblems.
- ***Programming*** means a "***tabular method***."
- Dynamic programming applies when the ***subproblems overlap***. That is when subproblems share subsubproblems.
  - A dynamic programming algorithm solves each subsubproblem just once and then saves its answer in a table.
    - The adopted tabular method avoids recomputing the answer every time it solves each subsubproblem.
  - In this kind of problems, a divide-and-conquer algorithm does more work than necessary.
    - Divide-and-conquer algorithms partition the problem into ***disjoint subproblems***, solve the problem recursively, and then combine their solutions to solve the original problem.

Dynamic Programming (Cont.)

- Dynamic programming is typically applied to solve ***optimization problems***.
- Four-step method to find an optimal solution (maximization or minimization) with dynamic programming:
  - Characterize the structure of an optimal solution.
  - Recursively define the value of an optimal solution.
  - Compute the value of an optimal solution, typically in a bottom-up fashion.
  - Construct an optimal solution from computed information.

Outline

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

Rod Cutting

Rod Cutting

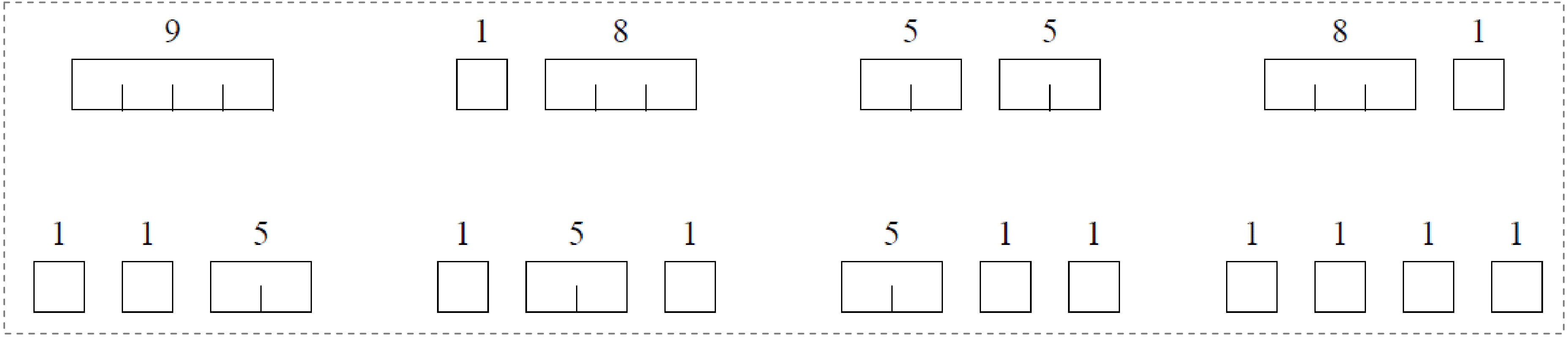
- How to cut steel rods into pieces in order to maximize the revenue you can get?
  - Each cut is free.
  - Rod lengths are always an integral number of inches.
- ***Input***:
  - A length n and table of prices p_i , for $i = 1, 2, \ldots, n$.
- ***Output***:
  - The maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.

An Example of Rod Cutting

- An *n-inch* rod of can be cut up in 2^{n-1} ways, because we can choose to cut or not cut after each of the first n-1 inches.

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

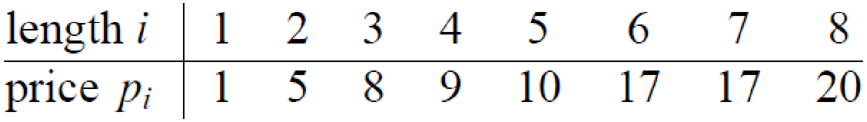
- ***Example***: A 4-inch rod



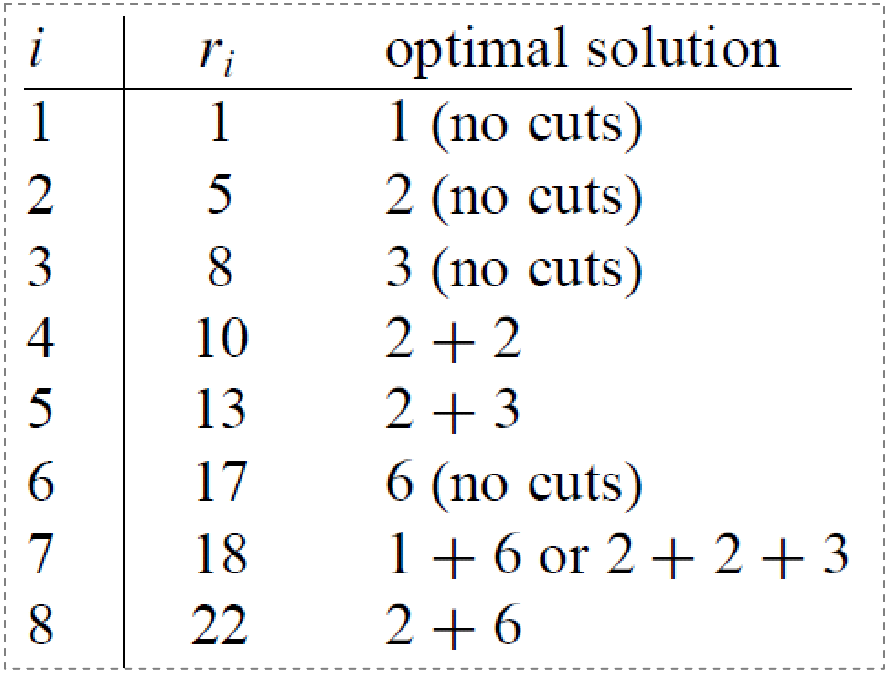
The best way is to cut it into two 2-inch pieces, getting a revenue of **$p_1 + p_2 = 5 + 5 = 10$**

An Example of Rod Cutting (Cont.)

- Let r_i be the maximal revenue for a rod of length i. The optimal revenues r_i for the example, by inspection:



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



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	$2 + 6$

- To determine the optimal revenue r_n by taking the maximum of
  - p_n: the price of no cut
  - $r_1 + r_{n-1}$: the maximum revenue from a rod of *1* inch and a rod of *n-1* inches,
  - $r_2 + r_{n-2}$: the maximum revenue from a rod of *2* inches and a rod of *n-2* inches, …
  - $r_{n-1} + r_1$: the maximum revenue from a rod of *n-1* inches and a rod of *1* inch.

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

Optimal Substructure

- 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 Decomposition

- 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 (***revenue p_i***), and a remaining piece of ***length n - i*** on the right (***revenue r_{n-i}***).
    - Need to divide only the remainder, not the first piece.
    - Leave only one subproblem to solve, rather than two subproblems.
    - The solution with no cuts has first piece size ***$i = n$*** with revenue ***p_n***, and remainder size ***0*** with revenue ***$r_0 = 0$***.
    - Give a simpler version of the equation for ***r_n***:

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

A Simpler Decomposition (Cont.)

- The call CUT-ROD(*p*, *n*) returns the optimal revenue r_n:
  - 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.

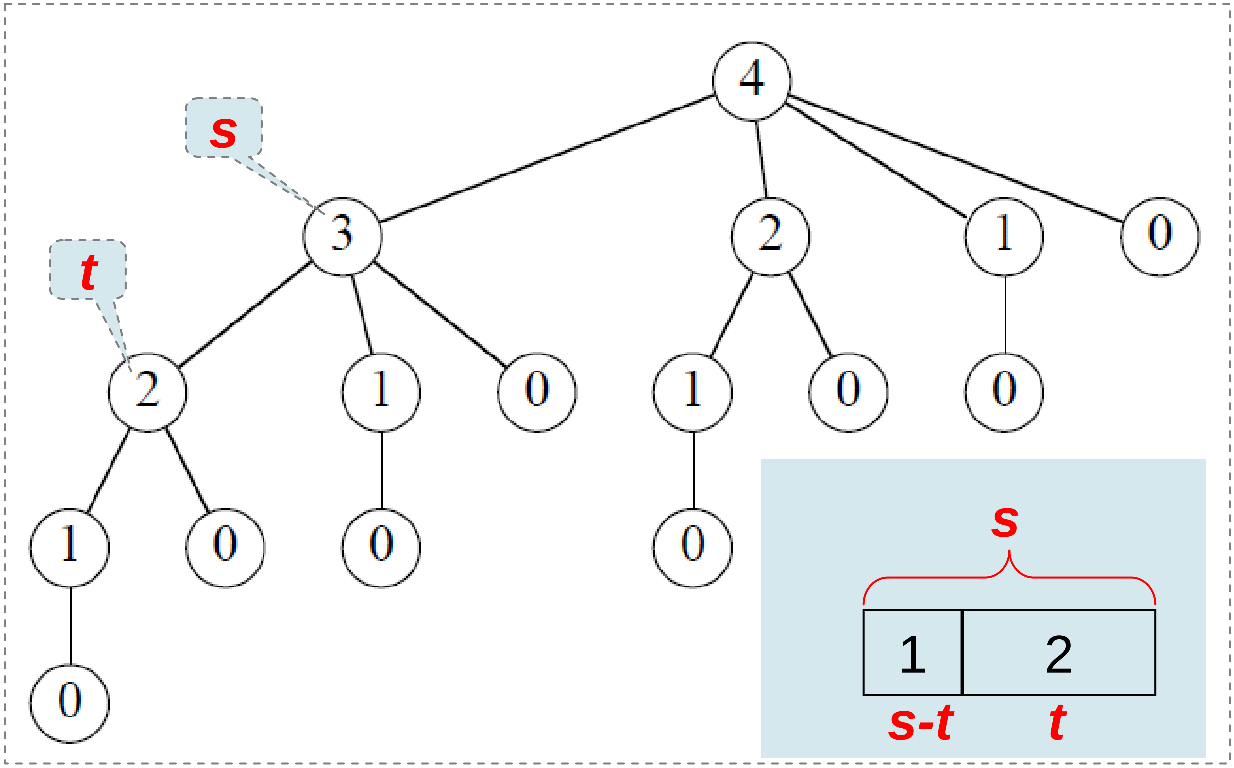
```
CUT-ROD($p, n$)
if $n == 0$
    return 0
$q = -\infty$
for $i = 1$ to $n$
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
return $q$
```

Adopt divide-and-conquer technique

A Simpler Decomposition (Cont.)

- For n = 4:
  - Have lots of repeated subproblems.
  - Solve the subproblem for ***size 2 twice***, for ***size 1 four times***, and for ***size 0 eight times***.

An edge from ***a parent with label s*** to ***a child with label t*** corresponds to cutting off an initial piece of size ***s – t*** and leaving a remaining subproblem of size ***t***.



A Simpler Decomposition (Cont.)

- ***Exponential growth***

$$a + ar + ar^2 + ar^3 + \cdots + ar^n = \sum_{k=0}^{n} ar^k = a\frac{1 - r^{n+1}}{1 - r},$$

Geometric series

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

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

  - The initial *1* is for the call at the root.
  - T(*j*) counts the number of calls due to the call **CUT-ROD(*p*, *n-i*)**, where ***j* = *n* - *i***.

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

- Solution to recurrence is **T(*n*) = 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*.
  - Turn an exponential-time solution into a polynomial-time solution.
- Two basic approaches:
  - ***Top-down with memoization***, and
  - ***Bottom-up method***.

Top-Down with Memoization

- Solve recursively, but store each result in a table.
- To find the solution to a subproblem, first look in the table.
  - If the answer is there, use it.
  - Otherwise, compute the solution to the subproblem and then store the solution in the table for future use.
- ***Memoized*** version of the recursive solution, storing the solution to the subproblem of length ***i*** in array entry ***r[i]***
  $\Rightarrow$

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

The array to store the optimal of the solved subproblems.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Memoizing is remembering what we have computed previously.

Bottom-Up Method

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

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

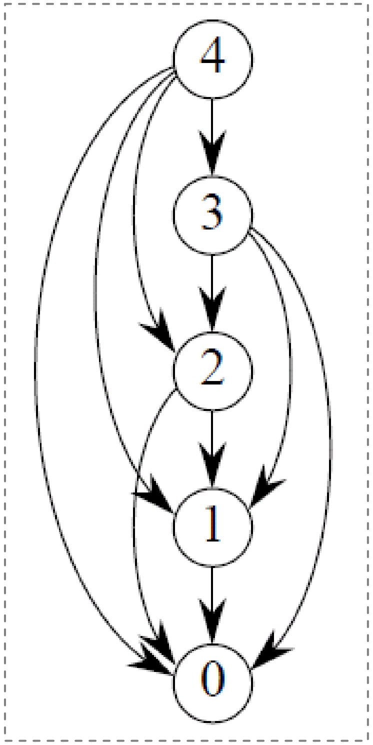
```
BOTTOM-UP-CUT-ROD($p, n$)
let $r[0..n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        $q = \max(q, p[i] + r[j - i])$
    $r[j] = q$
return $r[n]$
```

Running Time

- Both the *top-down* and *bottom-up* versions run in $O(n^2)$ 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 $n, n-1, \ldots, 0$.
    - To solve a subproblem of size $\boldsymbol{n}$, the **for** loop iterates $\boldsymbol{n}$ times. $\Rightarrow$ Over all recursive calls, total number of iterations forms an *arithmetic series*.

Subproblem Graphs

- Directed graph:
  - One ***vertex*** for each distinct subproblem.
  - A directed 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*:



- We 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.
- Because we solve each subproblem just once, the running time is sum of times needed to solve each subproblem.
  - 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*.

Thinking about a dynamic programming problem, we should understand how the set of subproblems involved and how subproblems depend on each other.

Reconstructing a Solution

- How to produce a choice that produces 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($p, n$)
let $r[0..n]$ and $s\underline{[0..n]}$ be new arrays
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        if $q < p[i] + r[j-i]$
            $q = p[i] + r[j-i]$
            $\underline{s[j]} = i$
    $r[j] = q$
return $r$ and $s$
```

```
PRINT-CUT-ROD-SOLUTION($p, n$)
$(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD($p, n$)
while $n > 0$
    print $s[n]$
    $n = n - s[n]$
```

Print out the cuts made in an optimal solution

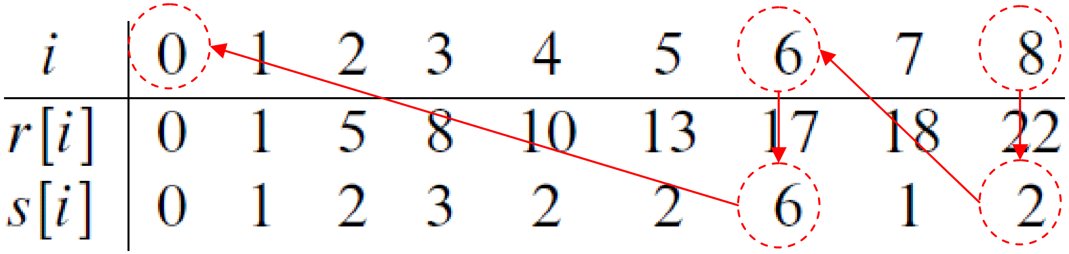
s[j] holds the optimal size ***i*** of the first piece to cut off when solving a subproblem of size ***j***.

Reconstructing a Solution (Cont.)

- ***Example***:

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

  - 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

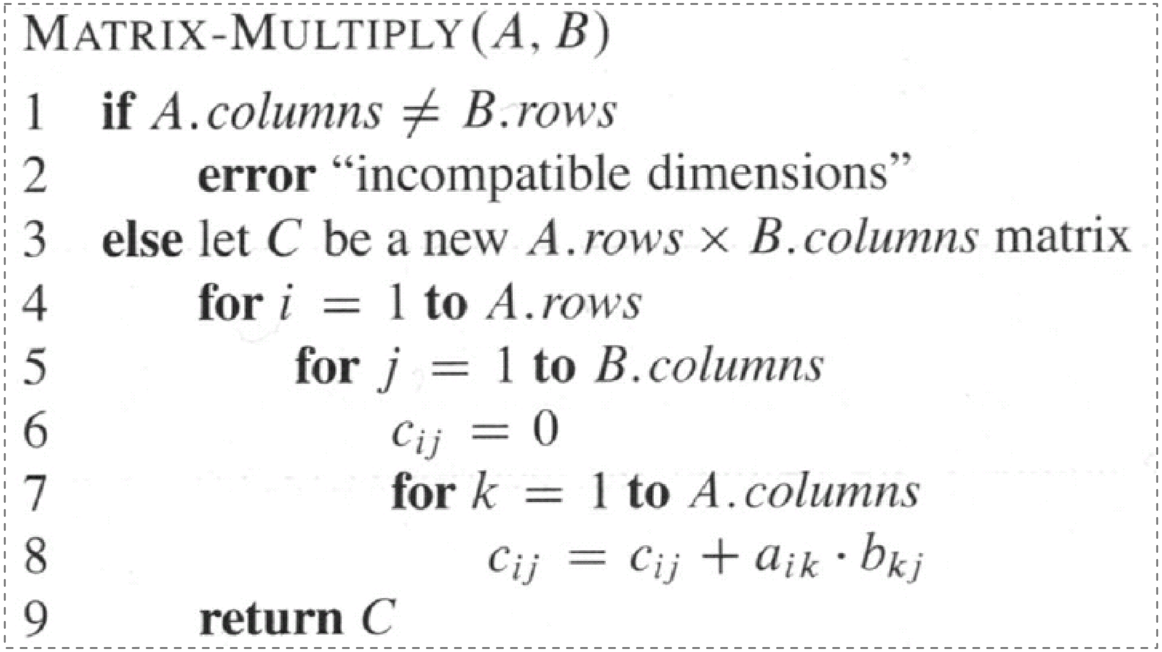
  - A call to PRINT-CUT-ROD-SOLUTION(p, ***8***) calls EXTENDED-BOTTOM-UPCUT-ROD to compute the above *r* and *s* tables.
  - Then it prints ***2***, sets n to 6, prints ***6***, and finishes (because n becomes 0).

Matrix-Chain Multiplication

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***.
- ***Example:***
  - ***Input:***
    - A chain of matrices is $<A_1, A_2, A_3, A_4>$
  - ***Output***:
    - Fully parenthesized matrices

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



```
MATRIX-MULTIPLY(A, B)
if $A.columns \neq B.rows$
    error "incompatible dimensions"
else let $C$ be a new $A.rows \times B.columns$ matrix
    for $i = 1$ to $A.rows$
        for $j = 1$ to $B.columns$
            $c_{ij} = 0$
            for $k = 1$ to $A.columns$
                $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
    return $C$
```

Matrix Multiplication

Matrix-Chain Multiplication (Cont.)

- If ***A*** is a $p \times q$ matrix and ***B*** is a $q \times r$ matrix, the result of multiplying ***A*** and ***B*** is a $p \times r$ matrix ***C***.
  - The time to computer *C* is dominated by ***the number of scalar multiplications***. That is $p \times q \times r$.
- ***Example:***
  - Given matrices $<A_1, A_2, A_3>$,
    - A_1 is a 10 × 100 matrix
    - A_2 is a 100 × 5 matrix
    - A_3 is a 5 × 50 matrix.
  - $A_1A_2 = 10 \cdot 100 \cdot 5 = 5{,}000$ multiplications to form a 10×5 matrix.
  - $A_2A_3 = 100 \cdot 5 \cdot 50 = 25{,}000$ multiplications to form a 100×50 matrix.
  - $((A_1, A_2), A_3)$
    $= 5{,}000 + 10 \cdot 5 \cdot 50 =$ *7,500* multiplications to form a 10×50 matrix.
  - $(A_1, (A_2, A_3))$
    $= 10 \cdot 100 \cdot 50 + 25{,}000 =$ *75,000* multiplications to form a 10 ×50 matrix.

Matrix-Chain Multiplication Problem

- ***Problem definition:***
  - Given a chain <A_1, A_2, …, A_n> of *n* matrices, where for *i = 1, 2, …, n*, matrix **A_i** has dimension ***$p_{i-1} \times p_i$***, fully parenthesize the product $A_1A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.
  - To represent the chain <A_1, A_2, …, A_n>, the input sequence $p = <\mathbf{p_0}, p_1, \ldots, p_n>$.
- Our goal is only to determine an order for multiplying matrices that has the lowest cost.

Counting the Number of Parenthesizations

- Exhaustively checking all possible parenthesizations does not yield an efficient algorithm.
- Let **P(n)** be the number of alternative parenthesizations of a sequence of *n* matrices.
  - When $n = 1$, 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.
    - The split between the two subproducts may occur between the *k*th and $(k+1)$st matrics for any $k = 1, 2, \ldots, n-1$.

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

→

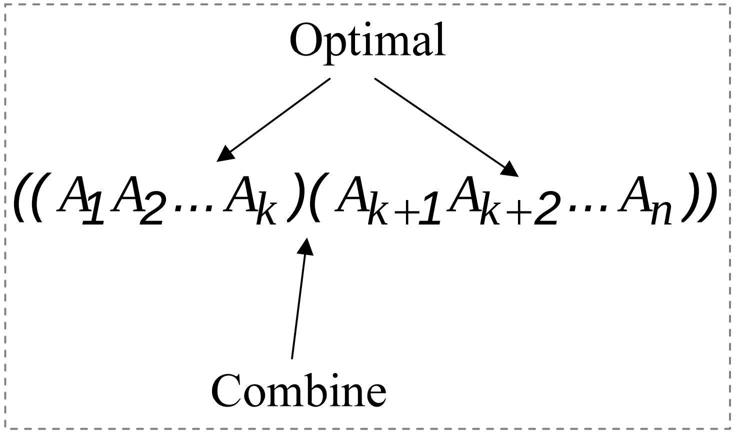
The number of solutions is **$\Omega(4^n/n^{3/2})$**

Applying Dynamic Programming

- Use dynamic-programming method to determining how to optimally parenthesize a matrix chain.
- The four-step sequence is
  - 1. Characterize the structure of an optimal solution.
  - 2. Recursively define the value of an optimal solution.
  - 3. Compute the value of an optimal solution.
  - 4. Construct an optimal solution from computed information.

Step 1.
The Structure of an Optimal Parenthesizatoin

- Let ***$A_{i..j}$*** denote the result of evaluating the product ***$A_iA_{i+1}..A_j$***, where $i \le j$.
- To parenthesize the product $A_iA_{i+1}..A_j$, the product between A_k and A_{k+1} for some integer k in the range $i \le k < j$ is split. That is
  - Comput $A_i A_{i+1}..A_k$ *and* $A_{k+1} A_{K+2}..A_j$ then
  - Multiply them together.



Optimal

$$((A_1A_2 \ldots A_k)(A_{k+1}A_{k+2} \ldots A_n))$$

Combine

Step 2
A Recursive Solution

- Let ***m[i, j]*** be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$.
  - If i = j, m[i..j] = 0 because no scalar multiplications.
  - If i < j, split $A_iA_{i+1}..A_j$ into $A_iA_{i+1}..A_k$ *and* $A_{k+1}A_{K+2}..A_j$ where $i \le k < j$.
    - There are $j - i$ possible values for k.
    - m[i..j] equals the minimum cost for computing the **$A_{i..k}$ and $A_{k+1..j}$**, plus the cost of **multiplying these two matrices together**.
- Since matrix A_i is $p_{i-1} \times p_i$, the product $A_{i..k}A_{k+1..j}$ takes $p_{i-1} \times p_k \times p_j$.
- The recursive definition for the minimum cost of parenthesizing the product $A_iA_{i+1}..A_j$ becomes:

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

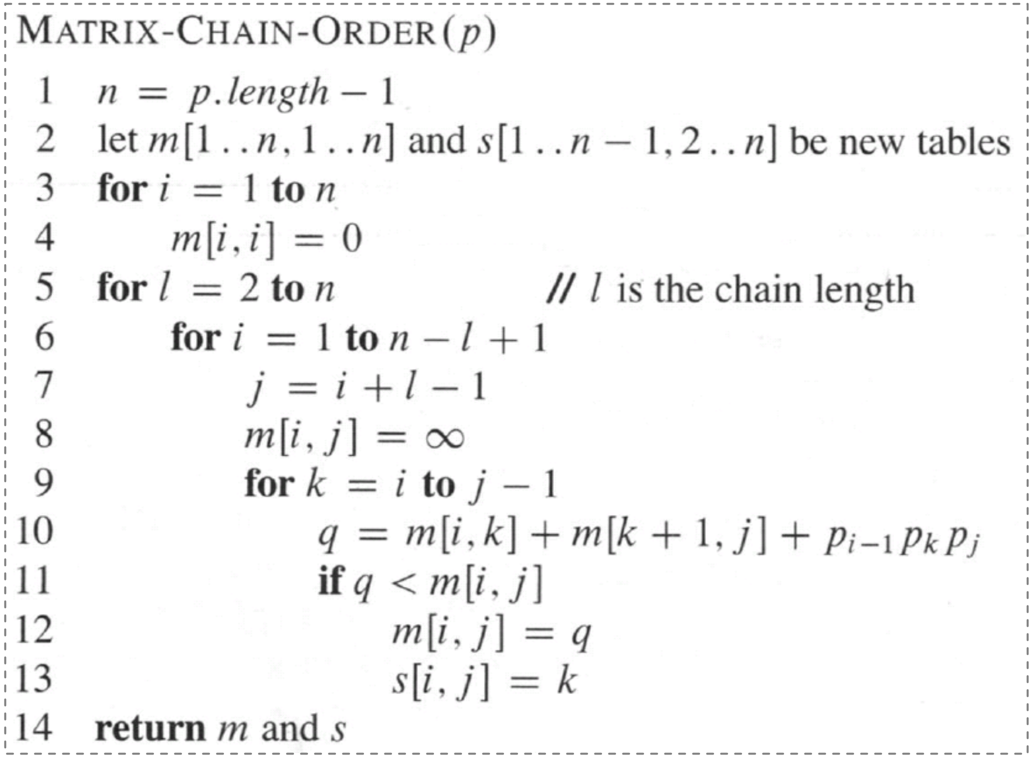
- The lowest cost way to compute $A_{1..n}$ is $m[1..n]$.

This recursive algorithm takes ***exponential time*** (similar to rod cutting) without adopting the ***tabular method***.

Step 3
Computing the Optimal Costs

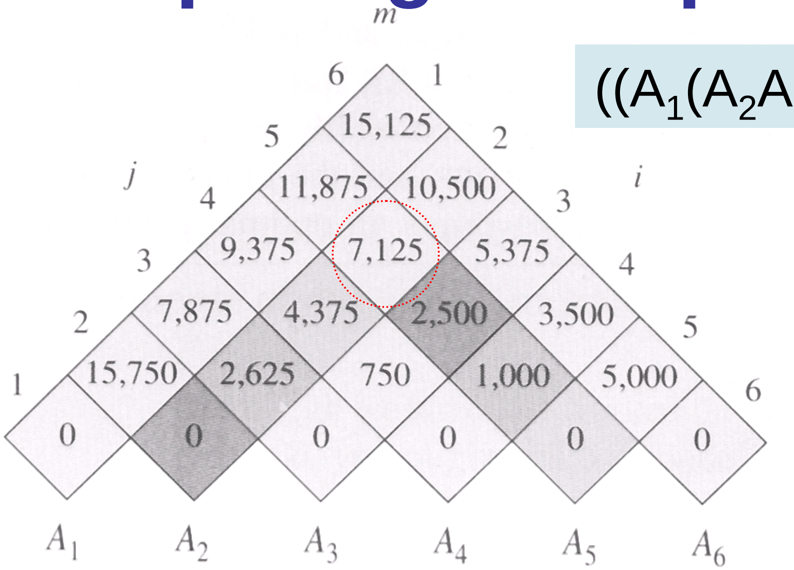
- The number of choices for i and j satisfying $1 \le i \le j \le n$ is $\mathbf{C_2^n + n} = n(n-1)/2 + n = n(n+1)/2 \mathbf{= \Theta(n^2)}$.
- A tabular, bottom-up approach:
  - Table m[1..*n*, 1..*n*] is to store the m[i, j] costs.
  - Table s[1..*n*-1, 2..*n*] records the *k* value achieving the optimal cost in computing m[i, j].

Running time $O(n^3)$.
Required space $\Theta(n^2)$.

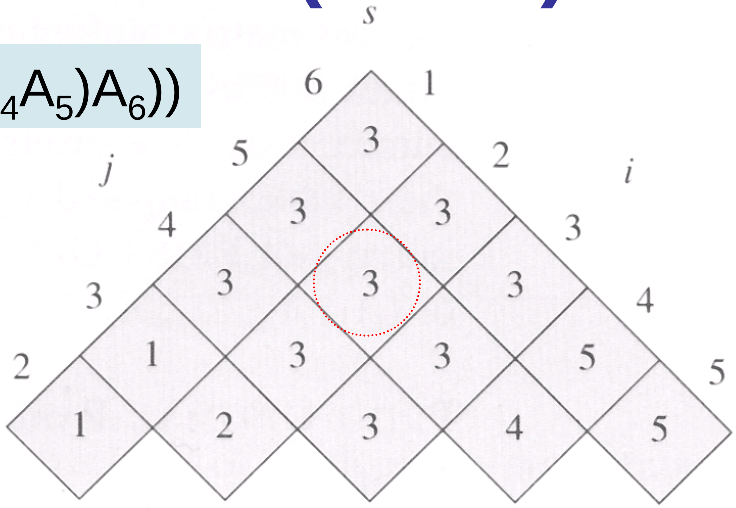


```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, 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
```

Step 3
Computing the Optimal Costs (Cont.)



$((A_1(A_2A_3))((A_4A_5)A_6))$



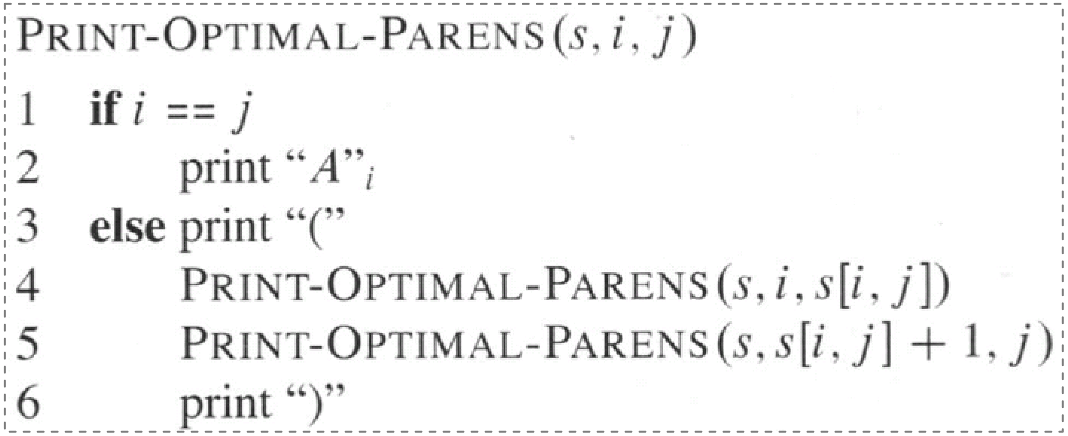
matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

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

$$= 7125 .$$

Step 4
Constructing an Optimal Solution

- Each entry *s[i, j]* records a value of k such that an optimal parenthesization of $A_i A_{i+1} .. A_j$, splits the product between A_k and A_{k+1}.
  - That is $A_{1..s[1, n]} A_{s[1,n]+1..n}$.
  - Find subproducts recursively:
    - $A_{1..s[1, n]}$ could be split at s[1, s[1, n]].
    - $A_{s[1, n]+1..n}$ could be split at s[s[1, n]+1, 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 ")"
```

Elements of Dynamic Programming

Elements of Dynamic Programming

- Two key elements that an optimization problem could be solved by dynamic programming:
  - *Optimal substructure*
    - An optimal solution to the problem contains ***within its optimal solution to subproblems***.
    - Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply.
    - We build an optimal solution to the problem from optimal solutions to subproblems.
  - *Overlapping subproblems*
    - When a recursive algorithm ***revisits the same problem repeatedly***, it has overlapping subproblems.
    - ***The total number of distinct subproblems is a polynominal 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 the recursion.

Common Pattens of Optimal Substructure

- A solution to the problem consists of ***making a choice***, Making this choice leaves ***one or more subproblems*** to be solved.
- For a given problem, you are given the choice that leads to an optimal solution.
- Given this choice, you determine which subproblems ensue (接著發生) and how to best characterize the ***resulting space of subproblems***.
- The solutions to the subproblems used within the optimal solution to the problem must themselves be optimal by using a “***cut-and-paste***” technique.
  - *Cutting out* the nonoptimal solution to each subproblem and *pasting in* the optimal one.

Key Points of Optimal Substructure

- Optimal substructure varies across problem domains in two ways:
  - ***How many subproblems*** are used in an optimal solution to the original problem.
  - ***How many choices*** we have in determining which subproblem(s) to use in an optimal solution.
- In rod cutting, ***$O(n)$ subproblems*** overall, and at most ***n choices*** to exam for each → ***$O(n^2)$*** running time.
  - With ***subproblem graph***, each ***vertex*** corresponds to a ***subproblem***, and the ***choices*** for a problem are the ***edges*** incident to that subproblem.
- In maxtirx-chain multiplication, ***$O(n^2)$ subproblems*** overall, and at most ***n choices*** to exam for each → ***$O(n^3)$*** running time.
  - With subproblem graph, there are ***$\Theta(n^2)$ vertices*** and each vertex would have degree at most ***n***.

Subtleties

- One should be careful not to assume that optimal substructure applies when it does not.
- Consider the following two problems in which we are given a directed graph $G = (V, E)$ and vertices u, $v \in V$.
  - ***Unweighted shortest path:***
    - Find a path from u to v consisting of the fewest edges. *Good for Dynamic programming*.
  - ***Unweighted longest simple path:***
    - Find a simple path from u to v consisting of the most edges. *Not good for Dynamic programming.*

Unweighted Shortest-Path Problem

- The unweighted shortest-path problem exhibits optimal substructure (***because subproblems do not share resources***).
  - Suppose that $u \neq v$. Any path p from u to v must contain an intermediate vertex w.
  - Decompose $u \overset{p}{\leadsto} v$ into subpaths $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$
  - Clearly, the number of edges in p equals the number of edges in p_1 **plus** that in p_2.
    - Proof: If p_1 or p_2 is not optimal and p'_1 or p'_2 is optimal, then $p'_1 + p_2 < p$ or $p_1 + p'_2 < p$ → ***Contradict that p is optimal.***

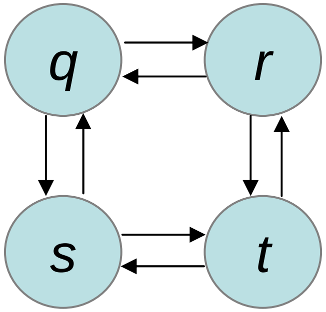
In matrix-chain multiplication, subchains are disjoint.
In rod-cutting, subproblems are disjoint.

$u \overset{p'_1}{\leadsto} w \overset{p_2}{\leadsto} v$

$u \overset{p_1}{\leadsto} w \overset{p'_2}{\leadsto} v$

Unweighted Longest-Path Problem

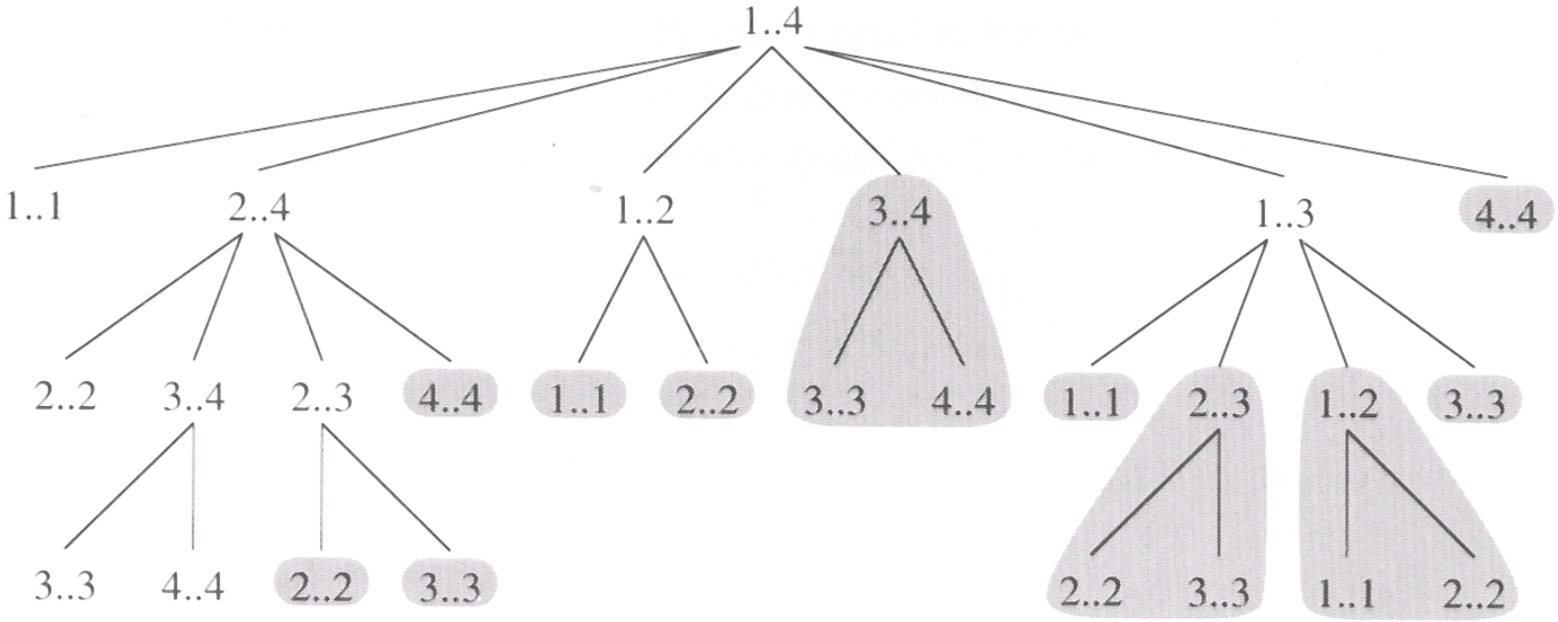
- Suppose that $u \neq v$. Any path p from u to v must contain an intermediate vertex w.
- Decompose $u \overset{p}{\rightsquigarrow} v$ into subpaths $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$
  - The p_1 might not be a longest path from u to w.
  - The p_2 might not be a longest path from w to v.



- ***Example***: Simple path means no cycle in the path.
  - One simple longest simple path from q to t is $q \rightarrow r \rightarrow t$.
  - ***Subproblems:***
    - $q \rightarrow r$ is not a simple longest path from from q to r. (Optimal: $q \rightarrow s \rightarrow t \rightarrow r$)
    - $r \rightarrow t$ is not a simple longest path from from r to t. (Optimal: $r \rightarrow q \rightarrow s \rightarrow t$)
    - Combine the above two suboptimals. The resulting path is not a simple path.
  - No optimal substructure exists because the subproblems in finding the longest simple path are ***not independent***.
    - One subproblem affects the solution to another subproblem.
    - E.g., $q \rightarrow s \rightarrow t \rightarrow r$ let the other not be able to select s and t. (due to "simple" path)

Overlapping Subproblems

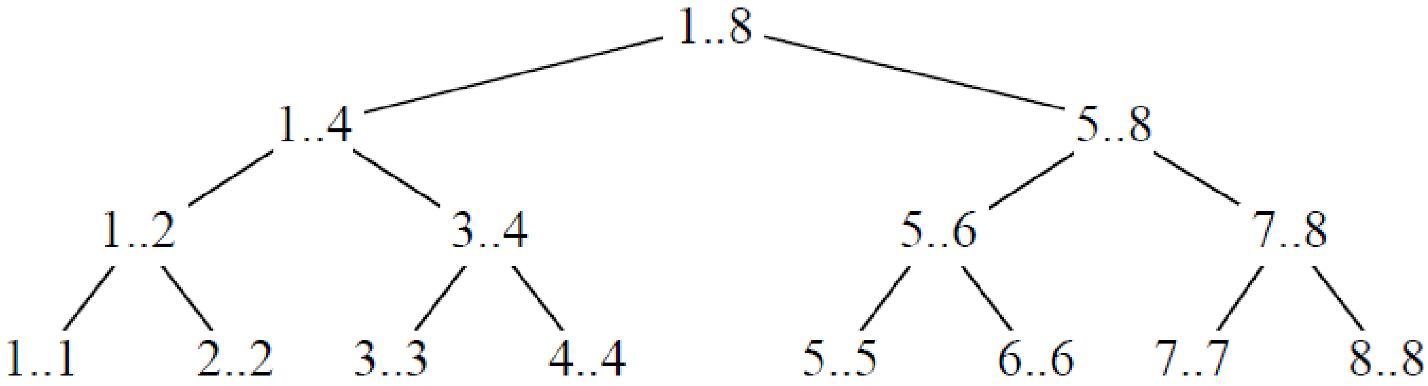
- An optimization problem for dynamic programming to apply must have “*small*” number of subproblems.
- Dynamic-programming algorithms typically solves each subproblem once and then stores the solution in a table for the future lookup.
  - For example, in matrix-chain multiplication, *m[3, 4]* is referenced four times: during the computations of *m[2, 4]*, *m[1, 4]*, *m[3, 5]*, and *m[3, 6]*.



Overlapping Subproblems (Cont.)

- Good divide-and-conquer algorithms usually generate a brand new problem at each stage of recursion.

Example: merge sort



Recursive Matrix Chain

- Let **T(n)** denote the time to compute an optimal parenthesization of a chain of *n* matrices.

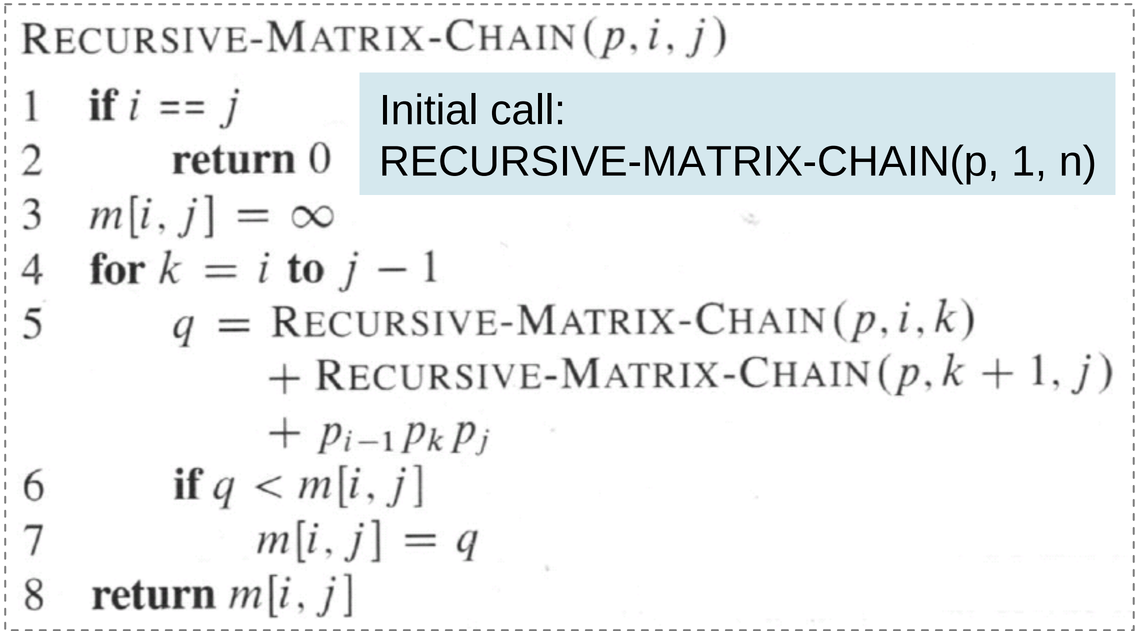
Lines 1, 2

Lines 6, 7 and multiplication

$$\begin{cases} T(1) \geq 1, \\ T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{for } n > 1 \end{cases} \Longrightarrow T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n$$

- Prove that T(n) ≥ $\Omega(2^n)$ using the substitution method:
  - Let $T(n) \geq 2^{n-1}$
    $T(1) \geq 1 = 2^0$ for $n \geq 1$

$$\begin{aligned} T(n) &\geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 \sum_{i=0}^{n-2} 2^{i} + n \\ &= 2(2^{n-1} - 1) + n \\ &= (2^n - 2) + n \geq 2^{n-1} \end{aligned}$$

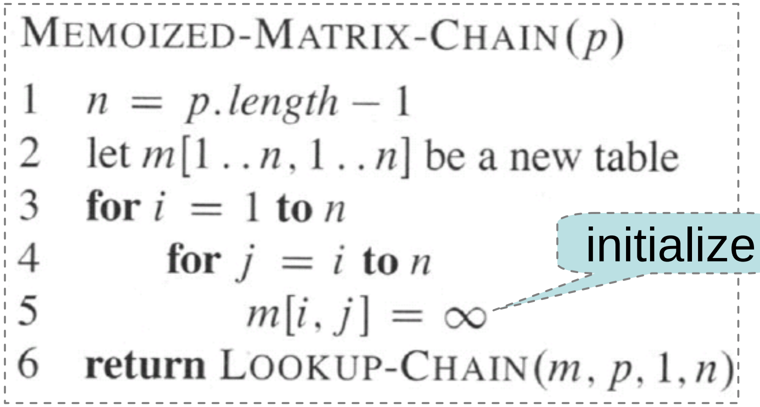


```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j - 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
             + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
             + $p_{i-1} p_k p_j$
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

Initial call:
RECURSIVE-MATRIX-CHAIN(p, 1, n)

Memoization

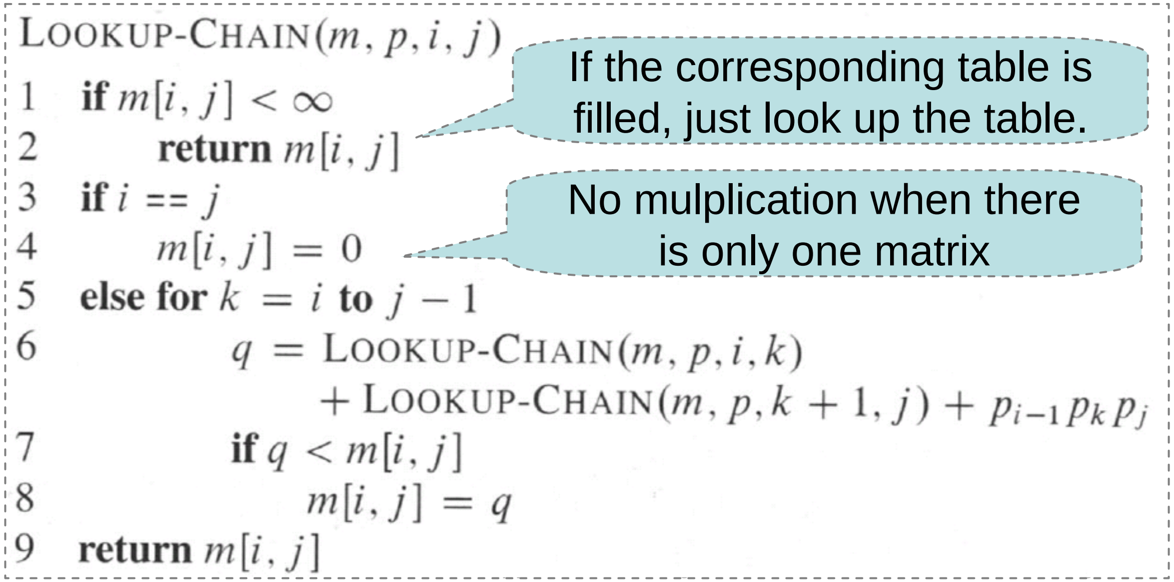
- In general, if all subproblems must be solved at least once,
    - A bottom-up DP algorithm usually outperforms the corresponding top-down memoized algorithm by ***a constant factor***.
    - The bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table.



```
MEMOIZED-MATRIX-CHAIN(p)
    $n = p.length - 1$
    let $m[1..n, 1..n]$ be a new table
    for $i = 1$ to $n$
        for $j = i$ to $n$
            $m[i, j] = \infty$
    return LOOKUP-CHAIN$(m, p, 1, n)$
```

initialize

- A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem.
- Time complexity: O(n^3)
    - **$\Theta(n^2)$** distinct subproblems.
    - Whenever a given call of LOOKUP-CHAIN makes recursive calls, it makes **O(n)** of them.



```
LOOKUP-CHAIN$(m, p, i, j)$
    if $m[i, j] < \infty$
        return $m[i, j]$
    if $i == j$
        $m[i, j] = 0$
    else for $k = i$ to $j - 1$
            $q = $ LOOKUP-CHAIN$(m, p, i, k)$
                $+$ LOOKUP-CHAIN$(m, p, k + 1, j) + p_{i-1} p_k p_j$
            if $q < m[i, j]$
                $m[i, j] = q$
    return $m[i, j]$
```

If the corresponding table is filled, just look up the table.

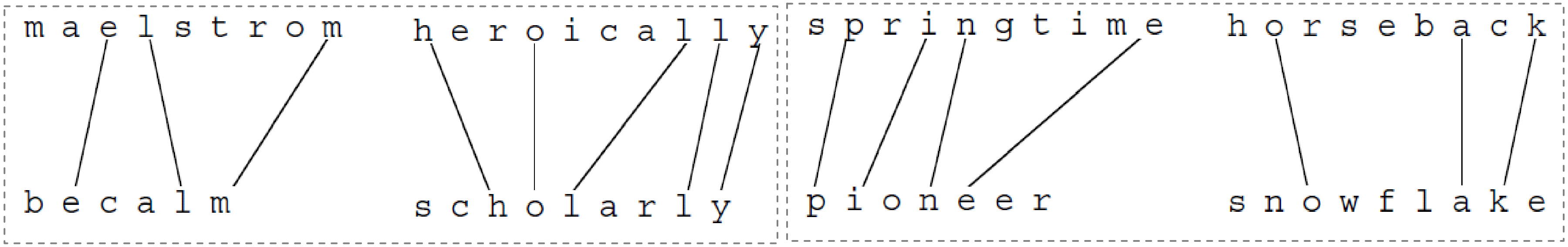
No mulplication when there is only one matrix

Longest Common Sequence

Longest Common Subsequence (LCS)

LCS is frequently adopted in DNA pattern matching.

- ***Input:***
  - Given 2 sequences, X = <x_1, …, x_m> and Y = <y_1, …, y_n>.
- ***Output:***
  - Find a subsequence common to both whose length is longest.
  - A subsequence doesn't have to be consecutive, but it has to be in order.



- Brute-force algorithm:
  - For every subsequence of X, check whether it's a subsequence of Y.
  - Time: **$Q(n2^m)$**.
    - **2^m** subsequences of X to check.
    - Each subsequence takes **O(n)** time to check: Scan **Y** for first letter, from there scan for second, and so on

Optimal Substructure

Notation:

$$X_i = \text{prefix } \langle x_1, \ldots, x_i \rangle$$

$$Y_i = \text{prefix } \langle y_1, \ldots, y_i \rangle$$

Theorem (Optimal substructure of an LCS)

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1}.

An LCS of two sequences contains as a prefix an LCS of prefixes of the sequences.

Optimal Substructure (Cont.)

1. First show that $z_k = x_m = y_n$. Suppose not. Then make a subsequence $Z' = \langle z_1, \ldots, z_k, x_m \rangle$. It's a common subsequence of X and Y and has length $k + 1 \Rightarrow Z'$ is a longer common subsequence than $Z \Rightarrow$ contradicts Z being an LCS.

   Now show Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}. Clearly, it's a common subsequence. Now suppose there exists a common subsequence W of X_{m-1} and Y_{n-1} that's longer than $Z_{k-1} \Rightarrow$ length of $W \geq k$. Make subsequence W' by appending x_m to W. W' is common subsequence of X and Y, has length $\geq k + 1$ $\Rightarrow$ contradicts Z being an LCS.
2. If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. Suppose there exists a subsequence W of X_{m-1} and Y with length $> k$. Then W is a common subsequence of X and $Y \Rightarrow$ contradicts Z being an LCS.
3. Symmetric to 2. ■ (theorem)

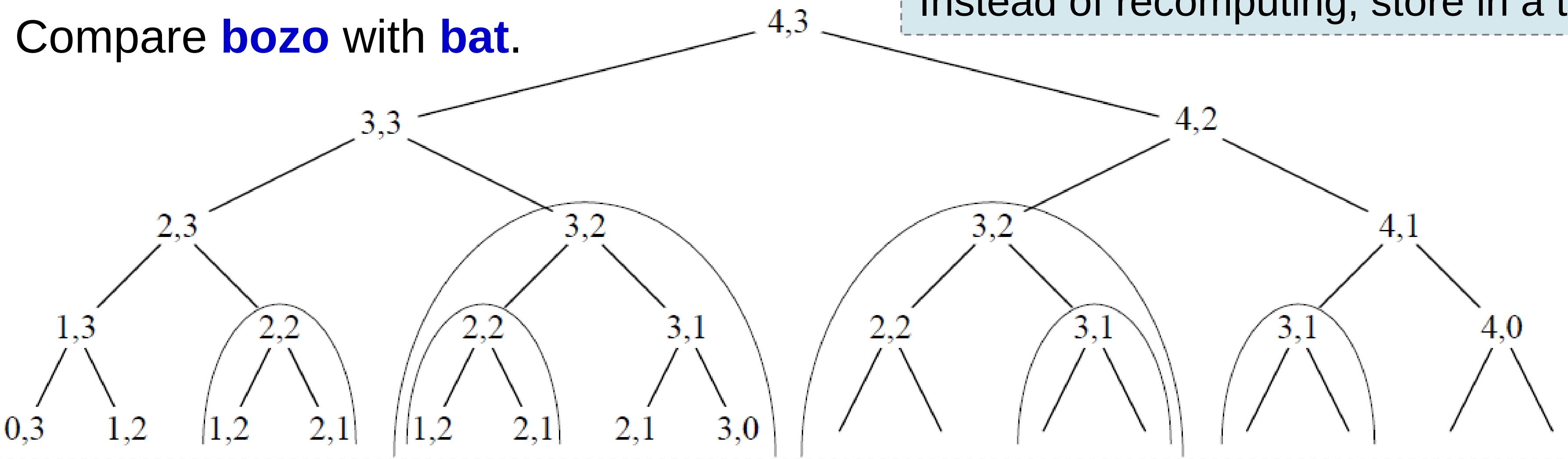
Recursive Formulation

Define $c[i,j]$ = length of LCS of X_i and Y_j. We want $c[m,n]$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i-1,j],c[i,j-1]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

Lots of repeated subproblems.
Instead of recomputing, store in a table.

Compare **bozo** with **bat**.



Compute Length of Optimal Solution

```
LCS-LENGTH($X, Y, m, n$)
let $b[1..m, 1..n]$ and $c[0..m, o..n]$ be new tables
for $i = 1$ to $m$
    $c[i, 0] = 0$
for $j = 0$ to $n$
    $c[0, j] = 0$
for $i = 1$ to $m$
    for $j = 1$ to $n$
        if $x_i == y_j$
            $c[i, j] = c[i-1, j-1] + 1$
            $b[i, j] =$ "$\nwarrow$"
        else if $c[i-1, j] \geq c[i, j-1]$
                $c[i, j] = c[i-1, j]$
                $b[i, j] =$ "$\uparrow$"
            else $c[i, j] = c[i, j-1]$
                $b[i, j] =$ "$\leftarrow$"
return $c$ and $b$
```

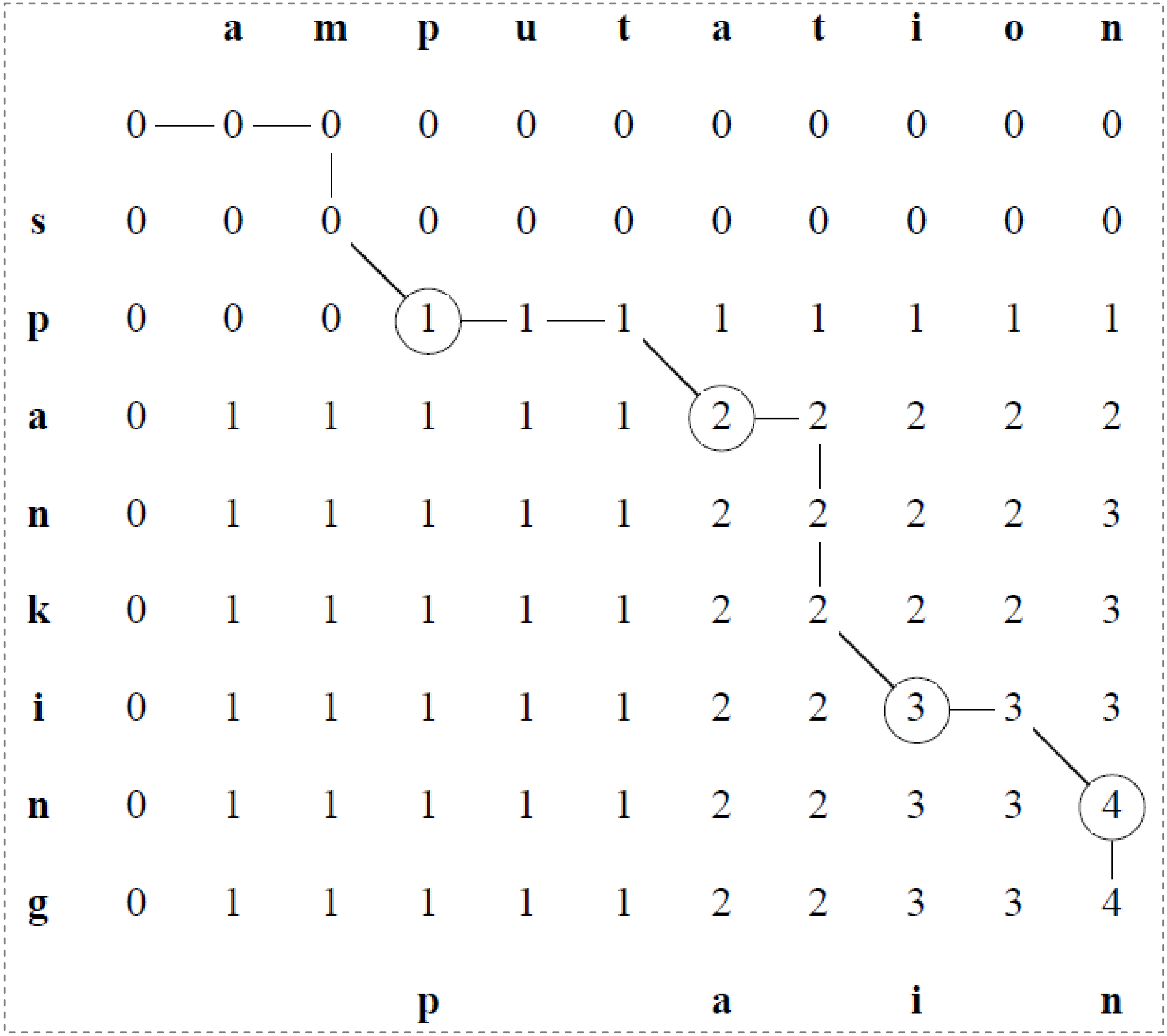
```
PRINT-LCS($b, X, i, j$)
if $i == 0$ or $j = 0$
    return
if $b[i, j] ==$ "$\nwarrow$"
    PRINT-LCS($b, X, i-1, j-1$)
    print $x_i$
elseif $b[i, j] ==$ "$\uparrow$"
    PRINT-LCS($b, X, i-1, j$)
else PRINT-LCS($b, X, i, j-1$)
```

- Initial call is PRINT-LCS(b, X, m, n).
- $b[i, j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j.
- When $b[i, j] = \nwarrow$, we have extended LCS by one character. So longest common subsequence = entries with $\nwarrow$ in them.



Demonstration

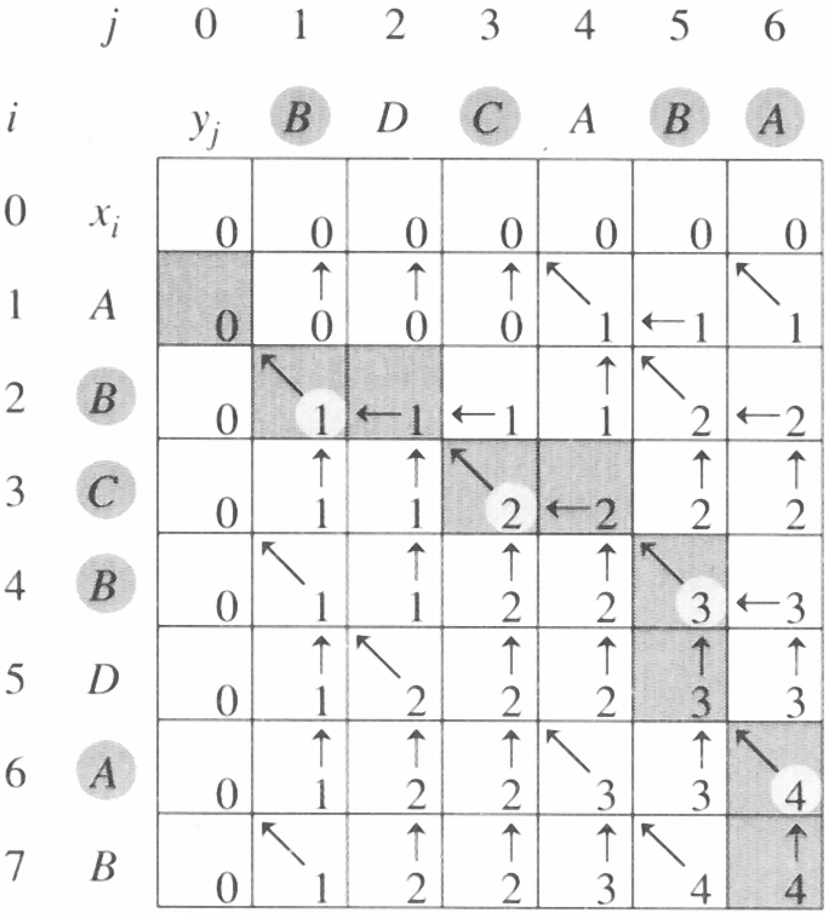
- ***spanking*** vs. ***amputation***
- Answer:
  - ***pain***
- Time:
  - **Θ(mn)**



		a	m	p	u	t	a	t	i	o	n
	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	1	1	1	1	1	1	1	1
a	0	1	1	1	1	1	2	2	2	2	2
n	0	1	1	1	1	1	2	2	2	2	3
k	0	1	1	1	1	1	2	2	2	2	3
i	0	1	1	1	1	1	2	2	3	3	3
n	0	1	1	1	1	1	2	2	3	3	4
g	0	1	1	1	1	1	2	2	3	3	4
				p			a		i		n

Demonstration (Cont.)

- ***ABCBDAB*** vs. ***BDCABA***
- Answer:
  - ***BCBA***
- Time:
  - **$\Theta(mn)$**



	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Optimal Binary Search Trees

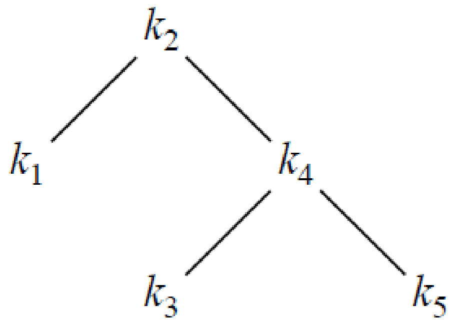
Optimal Binary Search Trees (BST)

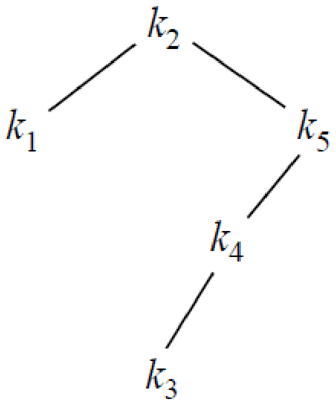
- Given sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys, sorted ($k_1 < k_2 < \ldots < k_n$).
- Want to build a binary search tree from the keys.
- For $\boldsymbol{k_i}$, have probability $\boldsymbol{p_i}$ that a search is for k_i .
- Want BST with minimum expected search cost.
- Actual cost = number of items examined.
  For key k_i , $cost = depth_T(k_i) + 1$,
  where $\mathrm{depth}_T(\mathrm{k}_i)$ = depth of k_i in BST T.

$$
\begin{aligned}
\mathrm{E}\,[\text{search cost in } T] \\
&= \sum_{i=1}^{n} (\mathrm{depth}_T(k_i) + 1) \cdot p_i \\
&= \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i \\
&= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i
\end{aligned}
$$

Example

i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3





i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	2	.1
4	1	.2
5	2	.6
		1.15

Therefore, E [search cost] = 2.15.

i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	3	.15
4	2	.4
5	1	.3
		1.10

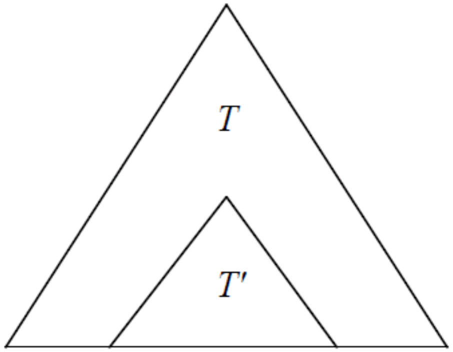
Therefore, E [search cost] = 2.10, which turns out to be optimal.

Observations

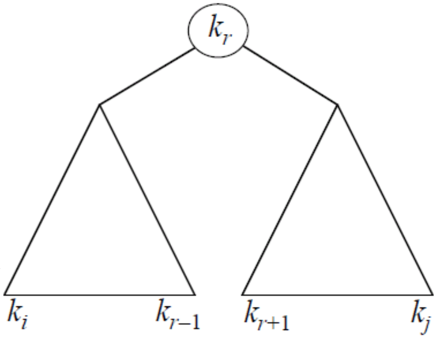
- Optimal BST might not have smallest height.
- Optimal BST might not have highest-probability key at root.
- Exhaustive checking:
  - Construct each n-node BST.
  - For each, put in keys.
  - Then compute expected search cost.
  - There are different $\Omega(4^n / n^{3/2})$ BSTs with *n* nodes.

Optimal Substructure

- Consider any subtree of a BST. It contains keys in a contiguous range $k_i, \dots, k_j$ for some $1 \le i \le j \le n$.
- If ***T*** is an optimal BST and ***T*** contains subtree ***T'*** with keys $k_i, \dots, k_j$, then ***T'*** must be an optimal BST for keys $k_i, \dots, k_j$.



- ***Proof***:
  - Use optimal substructure
    - Given keys $k_i, \dots, k_j$.
    - One of them, k_r , where $i \le r \le j$, must be the root.
    - Left subtree of k_r contains $k_i, \dots, k_{r-1}$.
    - Right subtree of k_r contains $k_{r+1}, \dots, k_j$.
  - If we examine all candidate roots $\boldsymbol{k_r}$, for $i \le r \le j$, and
  - we determine all optimal BSTs containing $k_i, \dots, k_{r-1}$ and containing $k_{r+1}, \dots, k_j$.
  - Then we're guaranteed to find an optimal BST for $k_i, \dots, k_j$.



Recursive Solution

Subproblem domain:

- Find optimal BST for $k_i, \ldots, k_j$, where $i \geq 1$, $j \leq n$, $j \geq i - 1$.
- When $j = i - 1$, the tree is empty.

Define $e[i, j]$ = expected search cost of optimal BST for $k_i, \ldots, k_j$.

If $j = i - 1$, then $e[i, j] = 0$.

If $j \geq i$,

- Select a root k_r, for some $i \leq r \leq j$.
- Make an optimal BST with $k_i, \ldots, k_{r-1}$ as the left subtree.
- Make an optimal BST with $k_{r+1}, \ldots, k_j$ as the right subtree.
- Note: when $r = i$, left subtree is $k_i, \ldots, k_{i-1}$; when $r = j$, right subtree is $k_{j+1}, \ldots, k_j$.

$e[j+1, j] = 0$

Recursive Solution (Cont.)

- When a subtree becomes a subtree of a node:
  - Depth of every node in subtree goes up by 1.
  - Expected search cost increases by

$$w(i,j) = \sum_{l=i}^{j} p_l$$

- If k_r is the root of an optimal BST for $k_i, \ldots, k_j$:

$$\begin{aligned} w(i,j) &= w(i,r-1) + p_r + w(r+1,j). \\ e[i,j] &= p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)). \\ &= e[i,r-1] + e[r+1,j] + w(i,j). \end{aligned}$$

- Try all candidates, and pick the best one:

$$e[i,j] = \begin{cases} 0 & \text{if } j = i-1, \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j. \end{cases}$$

Computing an Optimal Solution

As "usual," we'll store the values in a table:

$$e[\underbrace{1\,..\,n+1}_{\substack{\text{can store}\\ e[n+1,n]}},\ \underbrace{0\,..\,n}_{\substack{\text{can store}\\ e[1,0]}}]$$

- Will use only entries $e[i, j]$, where $j \geq i - 1$.
- Will also compute $root[i, j] =$ root of subtree with keys $k_i, \ldots, k_j$, for $1 \leq i \leq j \leq n$.
- One other table:

  Table $w[1\,..\,n+1, 0\,..\,n]$

  $w[i, i-1] = 0$ for $1 \leq i \leq n$

  $w[i, j] = w[i, j-1] + p_j$ for $1 \leq i \leq j \leq n$

Computing an Optimal Solution (Cont.)

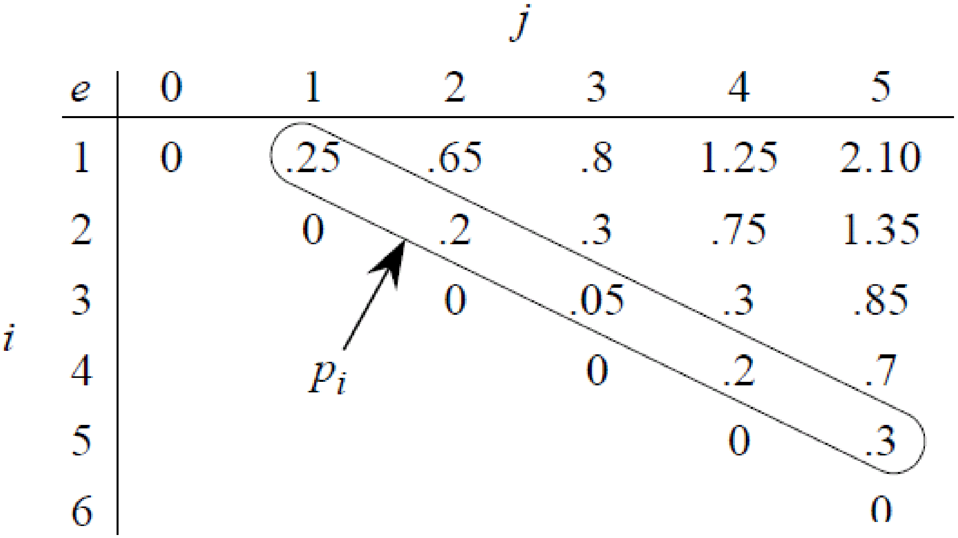
```
OPTIMAL-BST$(p, q, n)$
let $e[1..n+1, 0..n]$, $w[1..n+1, 0..n]$, and $root[1..n, 1..n]$ be new tables
for $i = 1$ to $n+1$
    $e[i, i-1] = 0$
    $w[i, i-1] = 0$
for $l = 1$ to $n$
    for $i = 1$ to $n-l+1$
        $j = i+l-1$
        $e[i, j] = \infty$
        $w[i, j] = w[i, j-1] + p_j$
        for $r = i$ to $j$
            $t = e[i, r-1] + e[r+1, j] + w[i, j]$
            if $t < e[i, j]$
                $e[i, j] = t$
                $root[i, j] = r$
return $e$ and $root$
```

When *l* = *1*, compute e[i, i] and w[i, i] for i=1…n. When *l* = *2*, compute e[i, i+1] and w[i, i+1] for i=1…n-1.

Try each candidate r

Time complexity: $\Theta(n^3)$

Computing an Optimal Solution (Cont.)



j

e	0	1	2	3	4	5
1	0	.25	.65	.8	1.25	2.10
2		0	.2	.3	.75	1.35
3			0	.05	.3	.85
4				0	.2	.7
5					0	.3
6						0

i

p_i

i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3

j

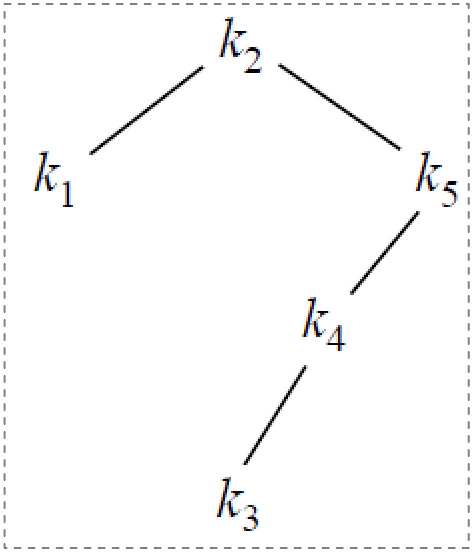
w	0	1	2	3	4	5
1	0	.25	.45	.5	.7	1.0
2		0	.2	.25	.45	.75
3			0	.05	.25	.55
4				0	.2	.5
5					0	.3
6						0

i

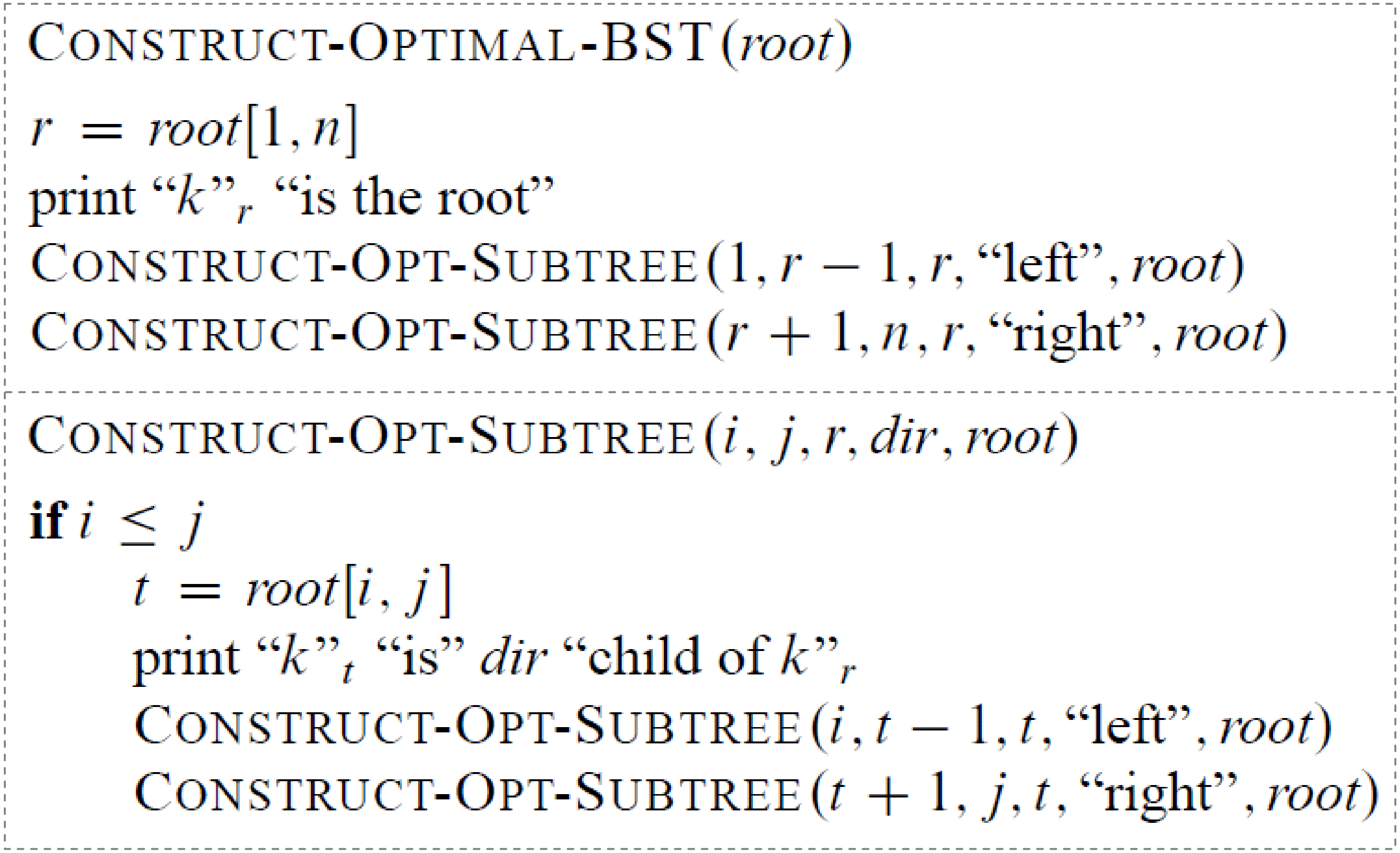
j

$root$	1	2	3	4	5
1	1	1	1	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

i



Construct an Optimal Solution



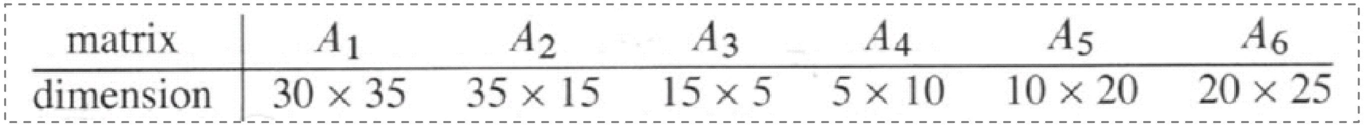
```
CONSTRUCT-OPTIMAL-BST(root)
$r = root[1, n]$
print "$k$"$_r$ "is the root"
CONSTRUCT-OPT-SUBTREE($1, r-1, r$, "left", $root$)
CONSTRUCT-OPT-SUBTREE($r+1, n, r$, "right", $root$)
```

```
CONSTRUCT-OPT-SUBTREE($i, j, r, dir, root$)
if $i \le j$
    $t = root[i, j]$
    print "$k$"$_t$ "is" $dir$ "child of $k$"$_r$
    CONSTRUCT-OPT-SUBTREE($i, t-1, t$, "left", $root$)
    CONSTRUCT-OPT-SUBTREE($t+1, j, t$, "right", $root$)
```

Project 4

- Use C language to implement the rod-cutting problem with dynamic programming.
  - The input file should be retrieved through ***argv[1]*** of main() function.
  - Use *fscanf()* to get integers from the input file.
    - The first integer indicates the length of the rod to cut.
    - The first integer also indicates the number of input integers in this file. The *i*-th input integer indicates the revenue of the rod of length *i*.
    - E.g., “***4 1 5 8 9***” means there is a 4-inch rod. 1, 5, 8, and 9 are the revenue of the rod of 1, 2, 3, and 4 inches, respectively.
  - Find and output cuts and the maximal revenue.
    - .E.g., ***2, 2: 10***
- Deadline: 24:00, 2010.10.18
  - Email the .c or .cpp program to me: johnsonchang@ntut.edu.tw
  - Email title: Algo_P4_學號_姓名

Project 5



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

- Use C language to implement the Matrix-chain multiplication problem with dynamic programming.
  - The input file should be retrieved through ***argv[1]*** of main() function.
  - Use *fscanf()* to get integers from the input file.
    - The first integer indicates the number of matrices.
    - E.g., “***6 30 35 15 5 10 20 25***” means there are 6 matrices (A1 to A6) and p_0 to p_6 are 30, 35, 15, 5, 10, 20, 25, respectively.
  - Find and output the minimal number of multiplications and the parenthesization of the matrices.
    - .E.g., ***15125, ((A1(A2A3))((A4A5)A6))***
- Deadline: 24:00, 2010.10.25
  - Email the .c or .cpp program to me: johnsonchang@ntut.edu.tw
  - Email title: Algo_P5_學號_姓名