Dynamic Programming

交通大學 資訊工程系

Overview

- **Dynamic programming**
  - Not a specific algorithm, but a technique (like divide-and-conquer).
  - Solves problems by combining the solutions to subproblems.
  - "Programming" in this context refers to a ***tabular method***.
  - Applicable when the subproblems are not independent (subproblems share subsubproblems).
  - Solve every subsubproblems just once and then saves its answer in a table.
  - Used for optimization problems.

Overview

- Follow a sequence of four steps to develop a dynamic-programming algorithm:
  - 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.
- Note:
  - If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4.

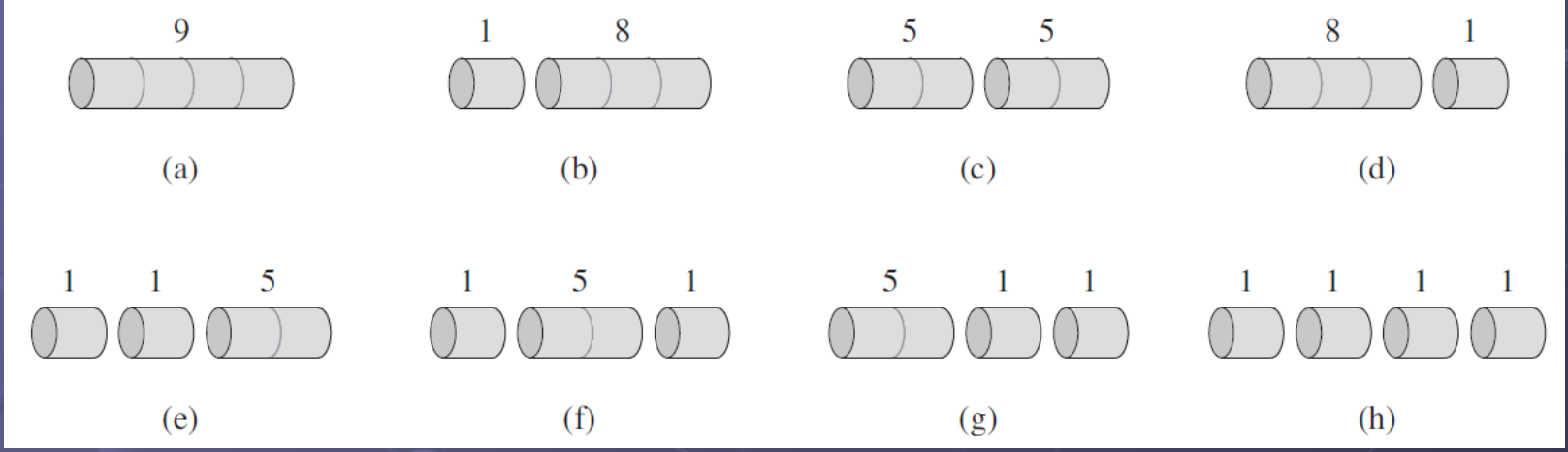
Rod Cutting[1]

- ***Rod cutting problem***
  - Given a rod of length n inches and a table of prices p_i for i= 1, 2, …, n.
  - Determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.
  - Rod lengths are always an integral number of inches.
- A sample price table

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

Rod Cutting[2]

☞ Example: The case when $n = 4$.



- Cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

Rod Cutting[3]

- There are 2^{n-1} different ways to cut a rod of length n.
  - There is an independent option of cutting, or not cutting, at distance i inches from the left end, for $i = 1, 2, \cdots, n-1$.
- An optimal solution cuts the rod into k pieces, for some $1 \leq k \leq n$, then
  - An optimal decomposition $n = i_1 + i_2 + \cdots + i_k$
  - Provides maximum corresponding revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$.

Rod Cutting[4]

- The optimal solution can be obtained by determining the *optimal revenue figure* r_i, for $i = 1, 2, \cdots, n$, with the corresponding optimal decompositions.

$$
\begin{aligned}
r_1 &= 1 \quad \text{from solution } 1 = 1 \quad \text{(no cuts)}, \\
r_2 &= 5 \quad \text{from solution } 2 = 2 \quad \text{(no cuts)}, \\
r_3 &= 8 \quad \text{from solution } 3 = 3 \quad \text{(no cuts)}, \\
r_4 &= 10 \quad \text{from solution } 4 = 2 + 2, \\
r_5 &= 13 \quad \text{from solution } 5 = 2 + 3, \\
r_6 &= 17 \quad \text{from solution } 6 = 6 \quad \text{(no cuts)}, \\
r_7 &= 18 \quad \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3, \\
r_8 &= 22 \quad \text{from solution } 8 = 2 + 6, \\
r_9 &= 25 \quad \text{from solution } 9 = 3 + 6, \\
r_{10} &= 30 \quad \text{from solution } 10 = 10 \quad \text{(no cuts)}.
\end{aligned}
$$

Rod Cutting[5]

- We can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

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

- ***Optimal substructure***
  - Optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

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

  - In this formulation, an optimal solution embodies the solution to only one related subproblem, the ***remainder***.

Rod Cutting[6]

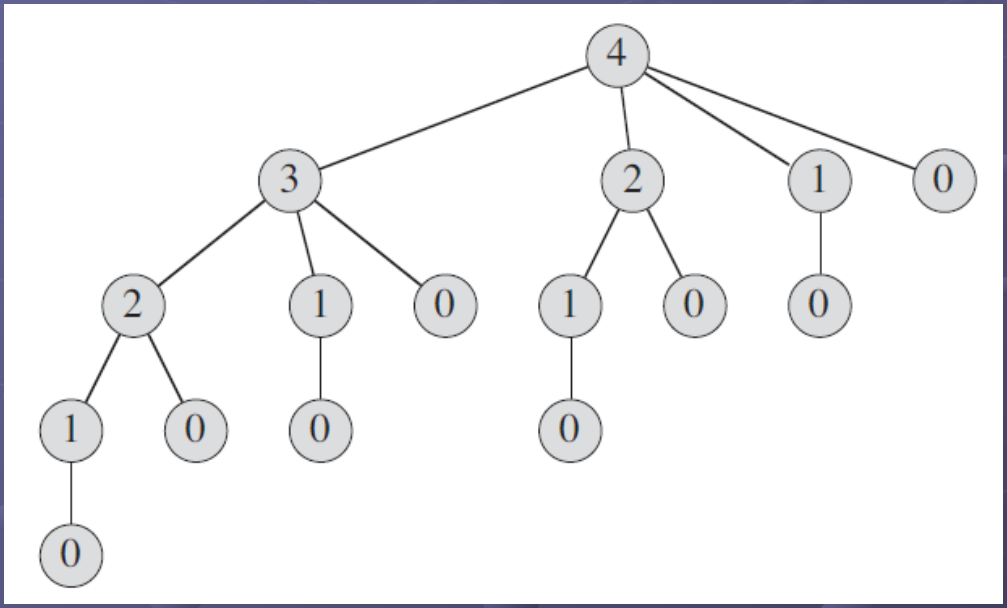
☞ Recursive top-down implementation:

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

- The inefficiency of CUT-ROD
  - It calls itself recursively over and over again.
  - It solves the same subproblems repeatedly.

Rod Cutting[7]

- The recursion tree of CUT-ROD, for $n = 4$.



- Each node label gives the size of the corresponding subproblem.
- A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n.

Rod Cutting[8]

- The running time of CUT-ROD

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

☞ Dynamic programming approach

- Use additional memory to save computation time, ***time-memory trade-off***.
- When does a dynamic programming approach run in polynomial time:

Rod Cutting[9]

- The number of distinct subproblems involved is polynomial, and
- Each subproblem can be solved in polynomial time.

- Two equivalent ways to implement a dynamic-programming approach:
  - *Top-down with memoization*
  - *Bottom-up method*

☞ **Top-down with memoization**

- Write the procedure recursively in a natural manner, but modified to save the result of each subproblem.

Rod Cutting[10]

```
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
```

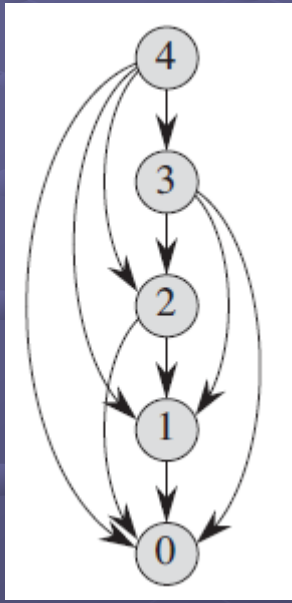
Rod Cutting[11]

- **Bottom-up method**
  - Solving any particular subproblem depends only on solving "smaller" subproblems.
  - We have already solved all of the smaller subproblems its solution depends upon, and saved their solutions.

```
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]
```

Rod Cutting[12]

- **Subproblem graph**
  - **Describe the set of subproblems involved and how subproblems depend on one another.**



Rod Cutting[13]

- The time to compute the solution to a subproblem is proportional to the degree of the corresponding vertex in the subproblem graph.
- The number of subproblems is equal to the number of vertices in the subproblem graph.
- The running time of dynamic programming is linear in the number of vertices and edges.

☞ Reconstructing a solution

- Extend to record not only the *optimal value* computed for each subproblem
- But also a *choice* that led to the optimal value.

Rod Cutting[14]

```
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
```

Rod Cutting[15]

```
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]
```

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

Matrix-Chain Multiplication[1]

- **Matrix-chain multiplication**
  - Given a sequence $\langle A_1, A_2, \cdots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product $A_1 A_2 \cdots A_n$.
- ***Fully parenthesized***
  - A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products.

$$
\begin{aligned}
&(A_1(A_2(A_3A_4))) ,\\
&(A_1((A_2A_3)A_4)) ,\\
&((A_1A_2)(A_3A_4)) ,\\
&((A_1(A_2A_3))A_4) ,\\
&(((A_1A_2)A_3)A_4) .
\end{aligned}
$$

Matrix-Chain Multiplication[2]

- Standard matrix multiplication algorithm

```
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 time of matrix multiplication dominated by the number of scalar multiplications.

Matrix-Chain Multiplication[3]

- Different costs incurred by different parenthesizations.

  A_1 (10×100), A_2 (100×5), A_3 (5×50).
  $((A_1A_2)A_3)$: 5,000 + 2,500 = 7,500
  $(A_1(A_2A_3))$: 25,000 + 50,000 = 75,000

☞ The *matrix-chain multiplication problem*

- Given a chain $\langle A_1, A_2, \cdots, A_n \rangle$ of n matrices, where for $i = 1, 2, \cdots, n$, matrix A_i has dimension $p_{i-1} \times p_i$.
- Fully parenthesize the product $A_1A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.
- The goal is only to determine an order for multiplying matrices that has the lowest cost.

Matrix-Chain Multiplication[4]

- Counting the number of parenthesizations
  - Let $P(n)$ denote the number of alternative parenthesizations of a sequence of n matrices.
  - The split between two subproducts may occur between the kth and $(k+1)$st matrices, for $k = 1, 2, \ldots, n-1$. Thus we obtain the recurrence

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

  - The solution to the recurrence is $\Omega(2^n)$.

Matrix-Chain Multiplication[5]

- Applying dynamic programming
  - Characterize the structure of an optimal solution.
  - Recursively define the value of an optimal solution.
  - Compute the value of an optimal solution.
  - Construct an optimal solution from computed information.
- Step 1: The structure of an optimal parenthesization.
  - Adopt $A_{i..j}$, where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$.
  - The optimal substructure of this problem
    - Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, split the product between A_k and A_{k+1}.

Matrix-Chain Multiplication[6]

- Subchains $A_iA_{i+1}\ldots A_k$ and $A_{k+1}A_{k+2}\ldots A_j$ must be optimal parenthesization respectively.

☞ **Step 2: A recursive solution.**

- Subproblems: Determining the minimum cost of a parenthesization of $A_iA_{i+1}\ldots A_j$ for $1 \le i \le j \le n$.
- $m[i, j]$: The minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$. (The dimension of matrix A_i is $p_{i-1} \times p_i$.)

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

Matrix-Chain Multiplication[7]

- **Step 3: Computing the optimal costs**
  - **There are relative few distinct subproblems:**
    - **One subproblem for each choice of i and j satisfying $1 \le i \le j \le n$, or $\binom{n}{2} + n = \Theta(n^2)$ in all.**
  - **Implement the tabular, bottom-up method**
    - **Input $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $p.length = n + 1$.**
    - **Auxiliary table $m[1..n,1..n]$ for storing $m[i, j]$.**
    - **Auxiliary table $s[1..n,1..n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$.**

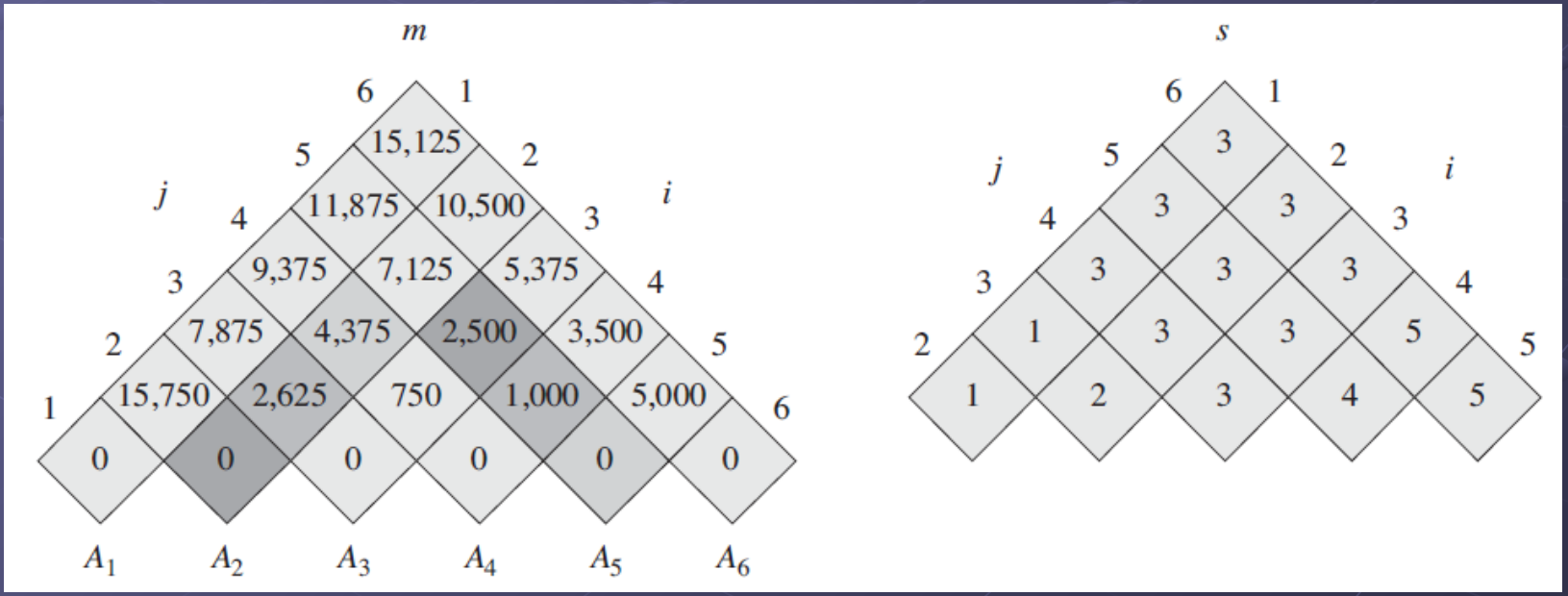
Matrix-Chain Multiplication[8]

```
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
```

Matrix-Chain Multiplication[9]

- Example: A chain of $n = 6$ matrices.

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-Chain Multiplication[10]

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

- Running time $O(n^3)$.
- Requires $\Theta(n^2)$ space to store the m and s tables.

☞ Step 4: Constructing an optimal solution.

- Each entry $s[i, j]$ records a value of k such that an optimal parenthesization of $A_iA_{i+1}\ldots A_j$ splits the product between A_k and A_{k+1}.

Matrix-Chain Multiplication[11]

```
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 ")"
```

- The result of the example with a chain of $n = 6$ matrices is $((A_1(A_2A_3))((A_4A_5)A_6))$.

Elements of Dynamic Programming[1]

- Two ingredients that an optimization problem must have in order for dynamic programming to be applicable:
  - *Optimal substructure*
  - *Overlapping subproblems*
- Optimal substructure
  - A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.
  - Ensure that the range of subproblems we consider includes those used in an optimal solution.

Elements of Dynamic Programming[2]

- Common pattern in discovering optimal substructure:
  - A solution to the problem consists of ***making a choice***. Making this choice leaves one or more subproblems to be solved.
  - Suppose for a given problem, you are given the choice that leads to an optimal solution.
  - Given this choice, determine which subproblems ensue and how to best characterize the resulting space of subproblems.
  - Use "***cut-and-paste***" technique to show that the solutions to the subproblems used within the optimal solution to the problem must themselves be optimal.

Elements of Dynamic Programming[3]

- "***Cut-and-paste***" technique
  - Suppose that one of the subproblem solutions is not optimal.
  - *Cut* it out.
  - *Paste* in an optimal solution.
  - Get a better solution to the original problem.
- How to characterize the space of subproblems:
  - Keep the space as simple as possible.
  - Then to expand it as necessary.

Elements of Dynamic Programming[4]

- 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.
- Rod-cutting problem
  - A rod of size n uses just one subproblem.
  - n choices for i in order to determine which one yields an optimal solution.

Elements of Dynamic Programming[5]

- **Matrix-chain multiplication**
  - **Two subproblems**
  - ***j* – *i* choices**
- **The running time of a dynamic-programming algorithm depends on the product of two factors:**
  - **The number of subproblems overall.**
  - **How many choices we look at for each subproblem.**
  - **In rod-cutting, $\Theta(n)$ subproblems and at most n choices for each.**
  - **In matrix-chain multiplication, $\Theta(n^2)$ subproblems and at most $n - 1$ choices in each.**

Elements of Dynamic Programming[6]

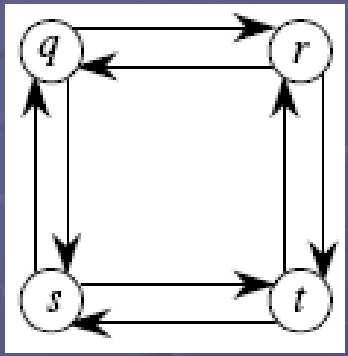
- Dynamic programming uses optimal substructure bottom up.
  - First find optimal solutions to subproblems.
  - Then choose which to use in optimal solution to the problem.
- The cost of the problem solution:
  - The subproblem costs.
  - A cost directly attributable to the choice itself.
    - Rod-cutting problem: p_i.
    - Matrix-chain multiplication: $p_{i-1}p_k p_j$.

Elements of Dynamic Programming[7]

- **Subtleties**
  - One should be careful not to assume that optimal substructure applies when it does not.
- **Example: *Unweighted shortest path***
  - Find path $u \rightarrow v$ with fewest edges. Must be *simple* (no *cycles*), since removing a cycle from a path gives a path with fewer edges.
  - Exhibits optimal substructure.
- **Example: *unweighted longest simple path***
  - Find *simple* path $u \rightarrow v$ with most edges. If didn't require simple, could repeatedly traverse a cycle to make an arbitrarily long path.

Elements of Dynamic Programming[8]

- Does not have optimal substructure.



The path $q \to r \to t$ is a longest simple path, but $q \to r$ is not a longest simple path from q to r, Nor is $r \to t$.

- An *NP*-complete problem.
- The subproblems in finding the longest simple path are not *independent*.
  - *Independent*: The solution to one subproblem does not affect the solution to another subproblem.

Elements of Dynamic Programming[9]

- Why are the subproblems *independent* for finding a shortest path?
  - The subproblems do not share resources.

☞ ***Overlapping subproblems***

- Two subproblems are overlapping if they are really the same subproblem that occurs as a subproblem of different problems.
- Dynamic programming algorithms solve each subproblem once and then store the solution in a table where it can be looked up when needed.

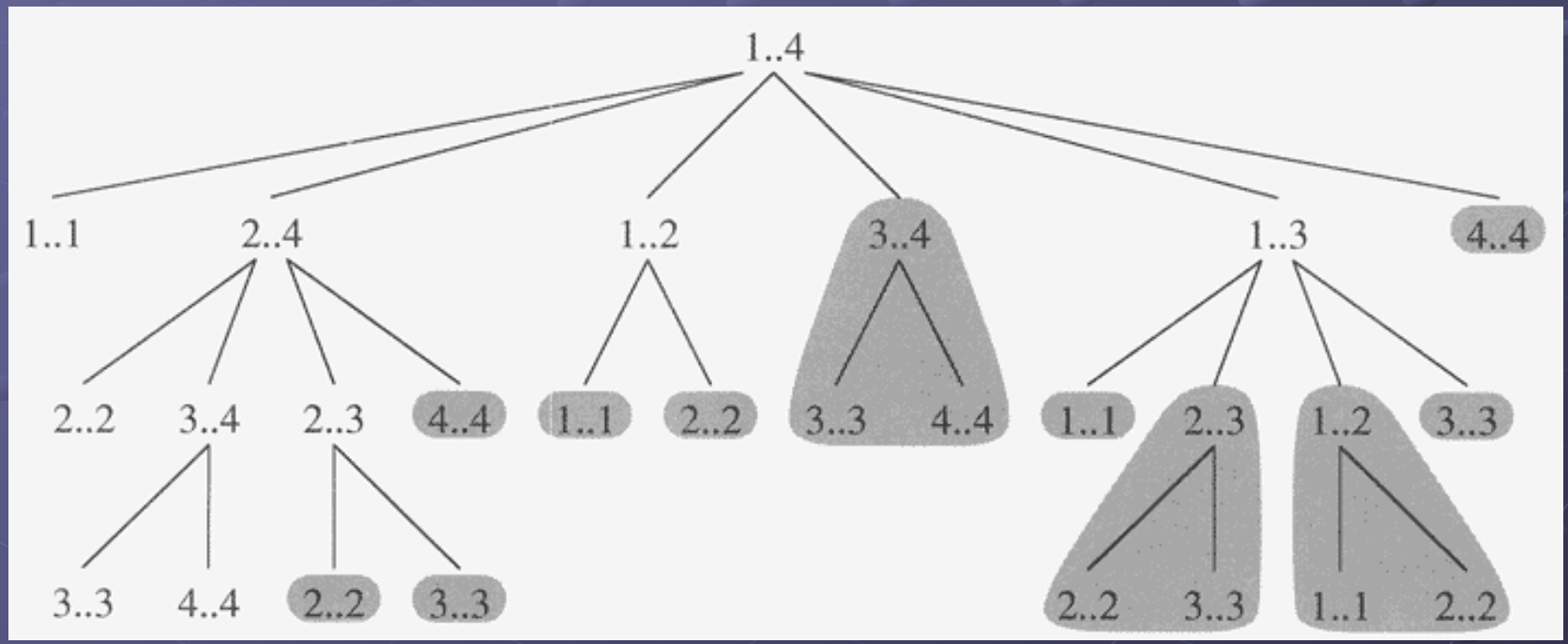
Elements of Dynamic Programming[10]

- **Recursive matrix-chain multiplication algorithm**

```
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]
```

Elements of Dynamic Programming[11]

- The corresponding recursion tree



- The computations of shaded subtrees are replaced by table lookup in MEMOIZED-MATRIX-CHAIN.

Elements of Dynamic Programming[12]

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

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n$$

$$T(1) \geq 1 = 2^0$$

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

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

Elements of Dynamic Programming[13]

- **Reconstruction of an optimal solution**
  - **We often store which choice we made in each subproblem in a table.**
  - **Then we can reconstruct the solution by table lookup.**
- ***Memoization***
  - **"Store, don't recompute."**
  - **Make a table indexed by subproblem.**
  - **When solving a subproblem:**
    - **Lookup in table.**
    - **If answer is there, use it.**
    - **Else, compute answer, then store it.**

Elements of Dynamic Programming[14]

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4       for j = i to n
5            m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)

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

Elements of Dynamic Programming[15]

- Running time: $O(n^3)$.
  - LOOKUP-CHAIN always returns the value of $m[i, j]$, but it computes it only if this is the first time been called.
- Without memoization, the natural recursive algorithm runs in exponential time.

☞ If all subproblems must be solved at least once

- Bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor.

☞ If some subproblems need not be solved at all

- The memoized solution has the advantage of solving only those subproblems that are definitely required.

Longest Common Subsequence[1]

- In biological applications
  - A strand of DNA consists of a string of molecules called *bases*:
    - Adenine (A)
    - Guanine (G)
    - Cytosine (C)
    - Thymine (T)
  - One goal of comparing two strands of DNA is to determine how "similar" the two strands are.
  - Two strands are similar if the number of changes needed to turn one into the other is small.

Longest Common Subsequence[2]

Subsequence

- Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Z = \langle z_1, z_2, \ldots, z_k \rangle$.
- Z is a *subsequence* of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$.
- For example: $Z = \langle B, C, D, B \rangle$ and $X = \langle A, B, C, B, D, A, B \rangle$ with index sequence $\langle 2, 3, 5, 7 \rangle$.

Common subsequence

- Given two sequences X and Y.
- Z is a subsequence of both X and Y.

Longest Common Subsequence[3]

- ***Longest-common-subsequence*** **(LCS) problem**
  - **Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$.**
  - **Find a maximum-length common subsequence of X and Y.**
- **Step 1: Characterizing a longest common subsequence.**
  - **Brute-force approach**
    - **Enumerate all subsequences of X.**
    - **There are 2^m subsequences.**
  - ***i*th prefix**
    - **Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, the *i*th prefix of X, for $i = 1, 2, \ldots, m$, is $X_i = \langle x_1, x_2, \ldots, x_i \rangle$.**

Longest Common Subsequence[4]

- *Theorem 15.1* (*Optimal substructure of an LCS*)
  - Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, ..., z_k \rangle$ 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 $z_k \neq x_m$ implies Z is an LCS of X_{m-1} and Y.
    3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies Z is an LCS of X and Y_{n-1}.
- Theorem 15.1 tells us
  - An LCS of two sequences contains within it an LCS of prefixes of the two sequences.

Longest Common Subsequence[5]

- **Step 2: A recursive solution.**
  - **Let $c[i, j]$ be the length of an LCS of the sequences X_i and Y_j.**

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

- **Step 3: Computing the length of an LCS.**
  - **Stores the $c[i, j]$ values in a table $c[0..m, 0..n]$ whose entries are computed in *row-major* order.**

Longest Common Subsequence[6]

- ***b*[*i*, *j*] points to the table entry corresponding to the optimal subproblem solution chosen when computing *c*[*i*, *j*].**

☞ **Procedure LCS-LENGTH.**

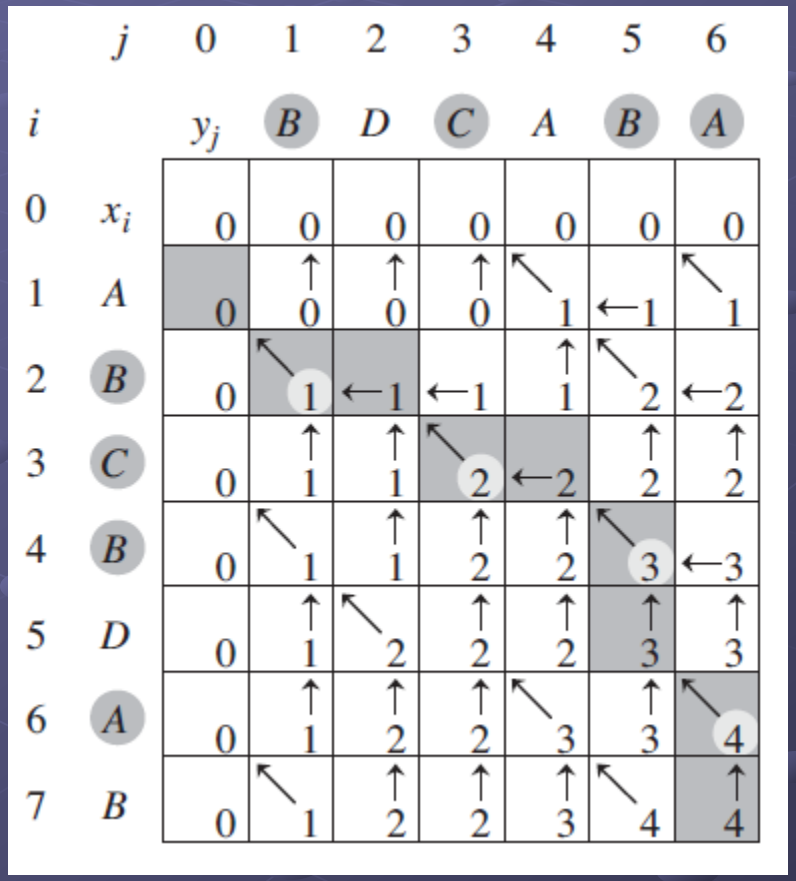
```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
```

Longest Common Subsequence[7]

```
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

- **Running time: O(*mn*).**

Longest Common Subsequence[8]



i		j	0	1	2	3	4	5	6
			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

Longest Common Subsequence[9]

- **Step 4: Constructing an LCS.**
  - **Call procedure PRINT-LCS(*b*, *X*, *X.length*, *Y.length*).**

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

  - **This procedure takes time *O*(*m* + *n*).**

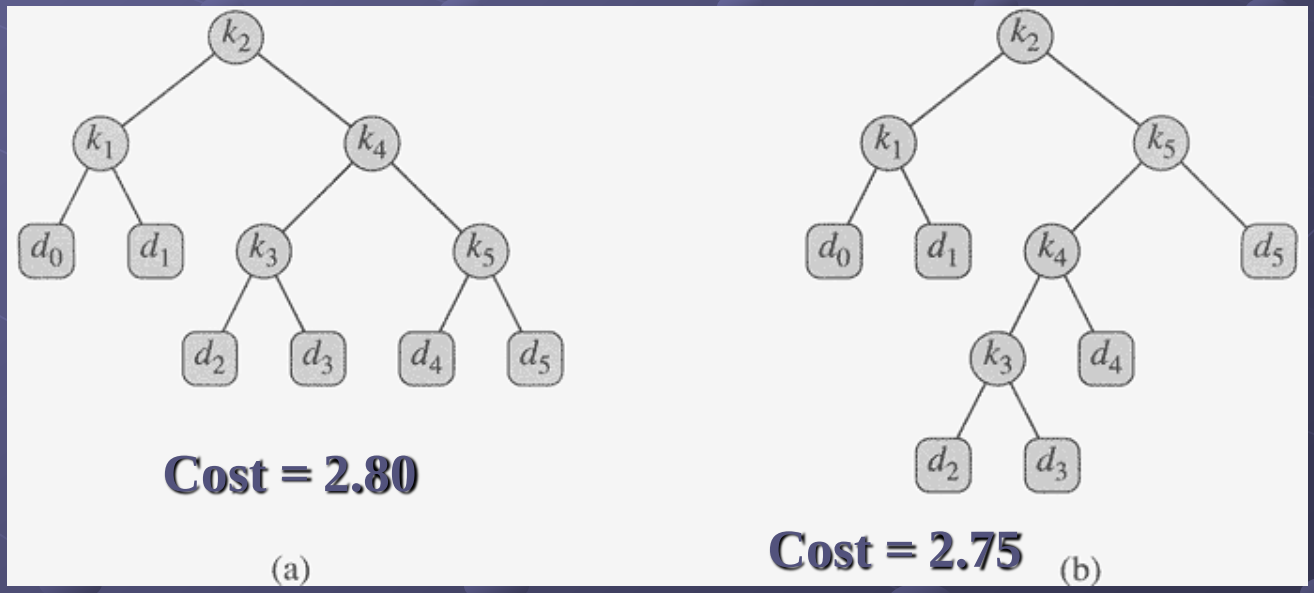
Optimal Binary Search Trees[1]

- **In language translation**
  - **We want words that occur frequently in the text to be placed nearer the root.**
- ***Optimal binary search tree***
  - **Given sequence $K = k_1, k_2, \ldots, k_n$ of n distinct keys, sorted ($k_1 < k_2 < \cdots < k_n$).**
  - **Want to build a binary search tree (BST) from the keys.**
  - **For k_i , have probability p_i that a search is for k_i .**
  - **For unsuccessful search, we have $n + 1$ "dummy keys" $d_0, d_1, d_2, \ldots, d_n$. For each d_i, we have probability q_i.**
  - **Want BST with minimum expected search cost.**

Optimal Binary Search Trees[2]

- **Example: Two binary search trees for a set of $n = 5$ keys with the following probabilities:**

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



Optimal Binary Search Trees[3]

- **Actual cost = # of items examined. For key k_i, cost = $\text{depth}_T(k_i) + 1$, where $\text{depth}_T(k_i)$ = depth of k_i in BST T.**

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

$$E[\text{search cost in } T]$$

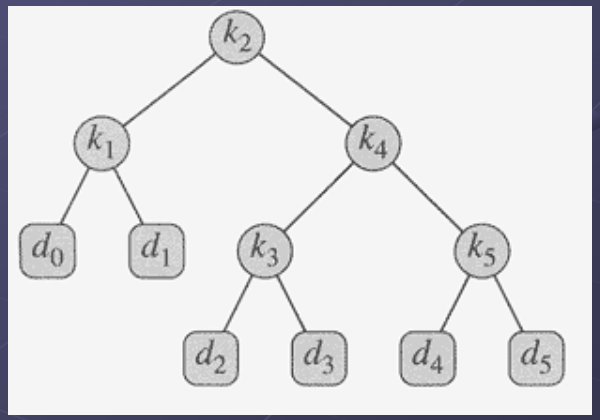
$$= \sum_{i=1}^{n} (depth_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n} (depth_T(d_i)+1)\cdot q_i$$

$$= 1 + \sum_{i=1}^{n} depth_T(k_i)\cdot p_i + \sum_{i=0}^{n} depth_T(d_i)\cdot q_i$$

Optimal Binary Search Trees[4]

☞ Expected search cost node by node of Fig. 15.9(a).

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80



Optimal Binary Search Trees[5]

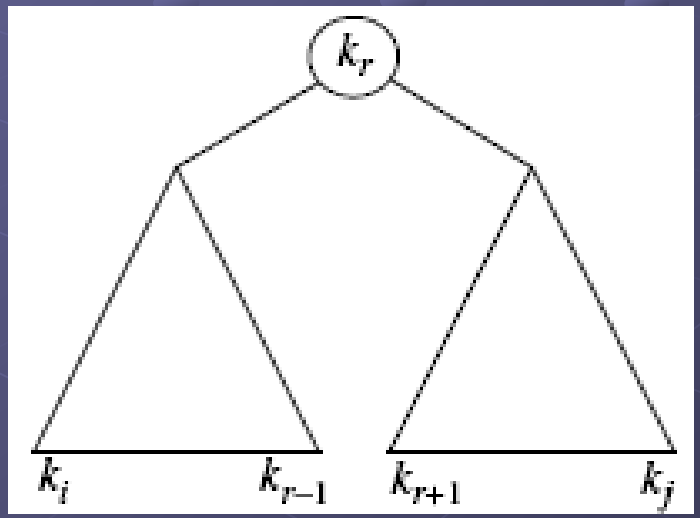
- Example of Fig. 15.9 shows
  - Optimal BST might not have smallest height.
  - Optimal BST might not have highest-probability key at root.
    - The lowest expected cost of any BST with k_5 at the root is 2.85.
- Build by exhaustive checking?
  - Construct each n-node BST.
  - For each, put in keys.
  - Then compute expected search cost.
  - But there are $\Omega(4^n/n^{3/2})$ different BSTs with n nodes.

Optimal Binary Search Trees[6]

- Step 1: The structure of an optimal BST.
  - Consider any subtree of a BST. It contains keys in a contiguous range $k_i, \ldots, k_j$ for some $1 \le i \le j \le n$.
  - If T is an optimal BST and T contains subtree T' with keys $k_i, \ldots, k_j$, then T' must be an optimal BST as well for keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$.
    - Proof: Cut-and-paste.
  - Use optimal substructure to construct an optimal solution to the problem from optimal solutions to subproblems:
    - Given keys $k_i, \ldots, k_j$ (the problem).

Optimal Binary Search Trees[7]

- One of them, k_r, where $i \le r \le j$, must be the root.
- Left subtree of k_r contains $k_i, \ldots, k_{r-1}$.
- Right subtree of k_r contains $k_{r+1}, \ldots, k_j$.



Optimal Binary Search Trees[8]

- If
  - we examine all candidate roots k_r, for $i \le r \le j$, and
  - we determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$,

  then we're guaranteed to find an optimal BST for $k_i, \ldots, k_j$.
- "Empty" subtrees
  - No actual keys but a single dummy key d_{i-1} or d_j.

Optimal Binary Search Trees[9]

- Step 2: A recursive solution.
  - Subproblem domain
    - Find optimal BST for k_i , …, 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, …, k_j.
    - If $j = i - 1$, then $e[i, j] = 0$.
  - When a subtree becomes a subtree of a node:
    - Depth of every node in subtree goes up by 1.

Optimal Binary Search Trees[10]

- **Expected search cost increases by the sum of all the probabilities in the subtree.**

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

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

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$$

But $w(i, j) = w(i, r-1) + p_r + w(r+1, j)$.

Therefore, $e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j)$

Optimal Binary Search Trees[11]

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

- **Step 3: Computing the expected search cost of an optimal BST.**
  - **Store the values in a table:**

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

Optimal Binary Search Trees[12]

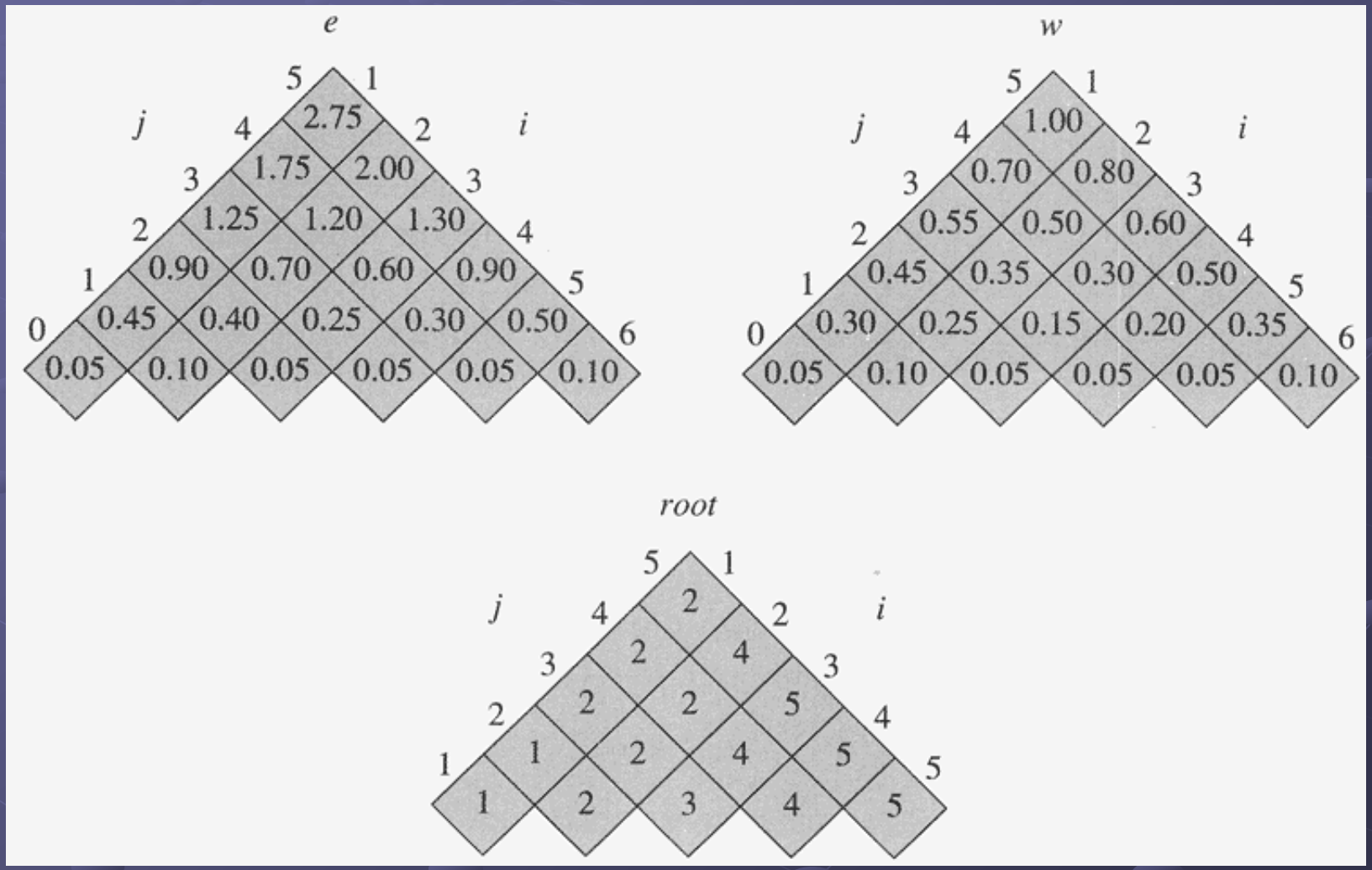
- Will use only entries $e[i, j]$, where $j \geq i - 1$.
- Will also compute

  $root[i, j]$ = root of subtree with keys $k_i, \ldots, k_j$, for $1 \leq i \leq j \leq n$.
- We need one other table $w(i, j)$
  - Table $w[1 \ldots n + 1, 0 \ldots n]$
  - $w[i, i - 1] = 0$, for $1 \leq i \leq n$.
  - $w[i, j] = w[i, j - 1] + p_j + q_j$, for $1 \leq i \leq j \leq n$.
  - Can compute all $\Theta(n^2)$ values in $O(1)$ time each.

Optimal Binary Search Trees[13]

```
OPTIMAL-BST(p, q, n)
 1  let e[1..n+1, 0..n], w[1..n+1, 0..n],
        and root[1..n, 1..n] be new tables
 2  for i = 1 to n + 1
 3      e[i, i − 1] = q_{i−1}
 4      w[i, i − 1] = q_{i−1}
 5  for l = 1 to n
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          e[i, j] = ∞
 9          w[i, j] = w[i, j − 1] + p_j + q_j
10          for r = i to j
11              t = e[i, r − 1] + e[r + 1, j] + w[i, j]
12              if t < e[i, j]
13                  e[i, j] = t
14                  root[i, j] = r
15  return e and root
```

Optimal Binary Search Trees[14]



Optimal Binary Search Trees[15]

- ***Running time:*** $O(n^3)$: for loops nested 3 deep, each loop index takes on $\leq n$ values. Can also show $\Omega(n^3)$. Therefore, $\Theta(n^3)$.

☞ **Step 4: 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)
```

Optimal Binary Search Trees[16]

```
CONSTRUCT-OPT-SUBTREE(i, j, r, dir, root)
if i ≤ j
   then 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)
```

Optimal Binary Search Trees[17]

k_2 is the root
k_1 is the left child of k_2
d_0 is the left child of k_1
d_1 is the right child of k_1
k_5 is the right child of k_2
k_4 is the left child of k_5
k_3 is the left child of k_4
d_2 is the left child of k_3
d_3 is the right child of k_3
d_4 is the right child of k_4
d_5 is the right child of k_5