Algorithms
Chapter 15
Dynamic Programming

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **Rod cutting**
- Matrix-chain multiplication
- Elements of dynamic programming
- Longest common subsequence
- Optimal binary search trees

Dynamic Programming$_{1/2}$

- Not a specific algorithm, but a technique, like divide-and-conquer.
- Dynamic programming is applicable when the subproblems are not independent.
- A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table.
- "Programming" in this context refers to a tabular method, not to writing computer code.
- Used for **optimization problems**:
  - Find **a** solution with **the** optimal value.
  - Minimization or maximization.

Dynamic Programming$_{2/2}$

- **Four-step method**

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 in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Rod cutting$_{1/2}$

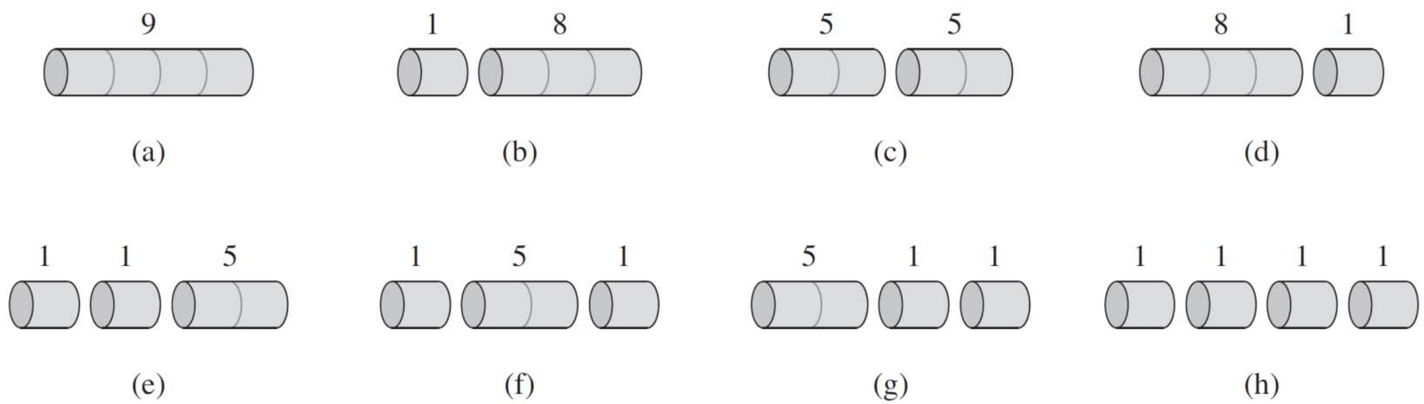
- How to cut steel rods into pieces in order to maximize the revenue you can get ?
  - Each cut is free.
  - Rod lengths are always an integral number of inches.
- **The rod-cutting problem problem**
  - **Input:** A length n and table of prices p_i, for i = 1, 2,…, n.
  - **Output:** The maximum revenue obtainable for rods whose lengths sum to n.
- If p_n is large enough, an optimal solution might require no cuts.
- We can cut up a rod of length n in 2^{n-1} different ways.
  - can choose to cut or not cut after each of the first $n - 1$ inches.

Rod cutting$_{2/2}$

- Consider the case when n = 4.

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

  - Here are all 8 ways to cut a rod of length 4.
  - The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.



Structure of an optimal solution

- Let r_i be the maximum revenue for a rod of length i.
- **Step 1:** Characterize the structure of an optimal solution.
  - Suppose a cut is made at distance j inches in an optimal solution of size n.
  - The optimal revenue $r_n = r_j + r_{n-j}$.
  - An optimal solution to a problem contains within it an optimal solution to subproblems.
  - This is **optimal substructure**.

Recursive solution

- **Step 2:** Recursively define the value of an optimal solution.
- Can determine optimal revenue r_n by taking the maximum of
  - p_n: the price we get by not making a cut,
  - $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of n − 1 inches,
  - $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inch and a rod of n − 2 inches,…
  - $r_{n-1} + r_1$.
- More generally, $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$.

A simpler way to decompose the problem

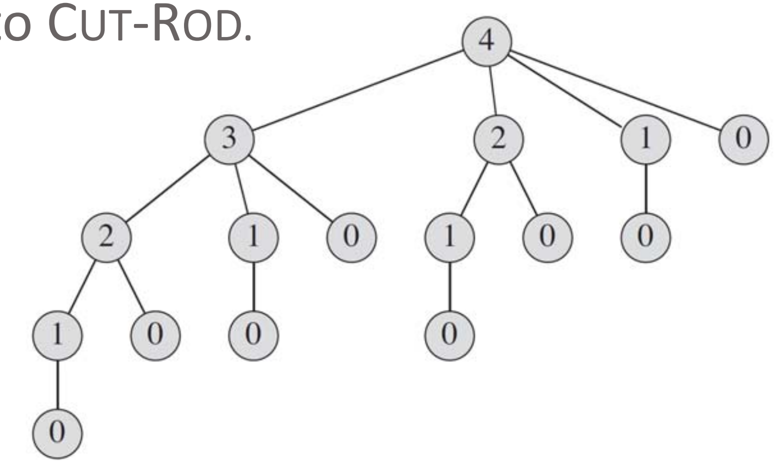
- Every optimal solution has a leftmost cut.
  - A first piece of length *i* cut off the left-hand end, and a remaining piece of length $n - i$ on the right.
  - Need to divide only the remainder, not the first piece.
  - Leaves only one subproblem to solve, rather than two subproblems.
  - $r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$.

```
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
```

Recursive top-down implementation

Running time of CUT-ROD(p, n)

- For n = 40, the program could take more than an hour.
- Each time you increase n by 1, the program's running time would approximately double.
- Why is CUT-ROD so inefficient?
  - It solves the same subproblems repeatedly.
- Running time:
  - $T(n)$: total number of calls made to CUT-ROD.
  - $T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n > 1. \end{cases}$
  - $T(n) = 2^n$. (exercise 15.1-1)



Dynamic programming

- Using dynamic programming for optimal rod cutting
  - Instead of solving the same subproblems repeatedly, arrange to solve each sub-problem just once.
  - Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.
  - "Store, don't recompute" ➔ time-memory trade-off.
  - Can turn an exponential-time solution into a polynomial-time solution.
- Two basic approaches: **top-down with memoization**, and **bottom-up method**.

Top-down with memoization

- Save the result of each subproblem in an array or hash table.
- The procedure first checks whether it has previously solved this subproblem.
  - Yes : return the saved value.
  - No : compute the value in the usual manner.
- **Memoizing** is remembering what we have computed previously.
  - Storing the solution of length *i* in array entry *r*[*i*].

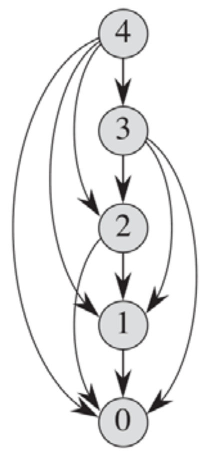
```
MEMOIZED-CUT-ROD-AUX(p,n, r)
    if r[n] ≥ 0
        return r[n]
    if n == 0
        q = 0
    else q = −∞
        for i = 1 to n
            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p,n − i, r))
    r[n] = q
    return  q
```

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

Bottom-up method

- **Step 3:** Compute the value of an optimal solution in a bottom-up fashion.
- The procedure solves subproblems of sizes j = 0, 1,…, n, in that order.
- When solving a subproblem, have already solved the smaller subproblems we need.

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



The subproblem graph.

Running time

- Both the top-down and bottom-up versions run in $\Theta(n^2)$ time.
- **Bottom-up**
  - Doubly nested loops.
  - Number of iterations of inner for loop forms an arithmetic series.
- **Top-down**
  - MEMOIZED-CUT-ROD solves each subproblem just once.
  - It solves subproblems for sizes 0, 1,…,n.
  - To solve a subproblem of size n, the for loop iterates n times.
  - Total number of iterations also forms an arithmetic series.

Reconstructing a solution$_{1/2}$

- **Step 4:** Construct an optimal solution from computed information.
- Saves the first cut made in an optimal solution for a problem of size *i* in *s*[*i*].

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n]  and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
return r and s
```

Reconstructing a solution$_{2/2}$

- To print out the cuts made in an optimal solution.

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

- The call EXTENDED-BOTTOM-UP-CUT-ROD(*p*, *n*) return

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

- A call to PRINT-CUT-ROD-SOLUTION(*p*, 10) would print just 10.
- A call with *n* = 7 would print the cuts 1 and 6.

Outline

- Rod cutting
- **Matrix-chain multiplication**
- Elements of dynamic programming
- Longest common subsequence
- Optimal binary search trees

Matrix-chain multiplication

- When we multiply two matrices A and B, if A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix.
  - The number of scalar multiplications is ***pqr***.
- **Matrix-chain multiplication problem**
  - **Input:** A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices.
    (matrix A_i has dimension $p_{i-1} \times p_i$)
  - **Output:** A fully parenthesized product $A_1, A_2, \ldots, A_n$ that minimizes the number of scalar multiplications.
- For example: The dimensions of the matrices A_1, A_2, and A_3 are 10×100, 100×5, and 5×50, respectively.
  - $((A_1A_2)A_3) = 10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$.
  - $(A_1(A_2A_3)) = 100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$.

Counting the number of parenthesizations

- **Brute-force algorithm:**
  - Checking all possible parenthesizations
- Time: $\Omega(2^n)$. (Exercise 15.2-3)
  - Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$.
  - A fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts.
  - The split between the two subproducts may occur between the kth and $(k + 1)$st matrices.
  - Thus, we have $P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$

Step 1: The structure of an optimal solution

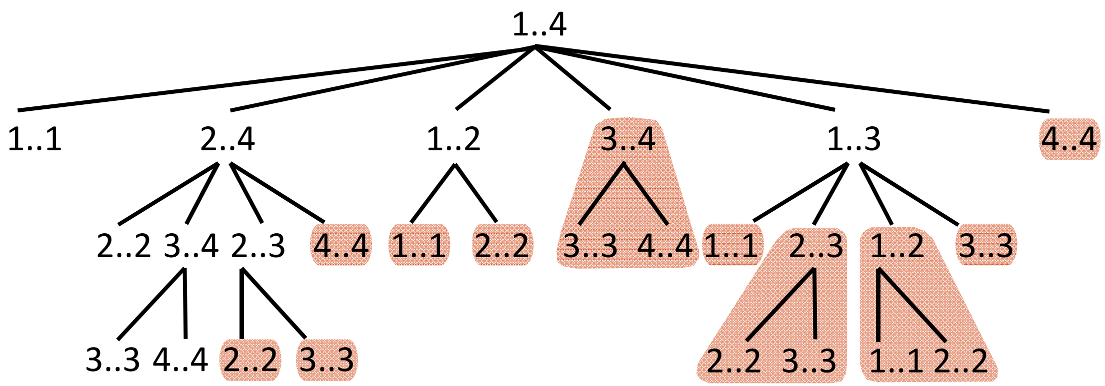
- An optimal solution to an instance contains optimal solutions to subproblem instances.
- For example:
  - If $((A_1A_2)A_3)(A_4(A_5A_6))$ is an optimal solution to $A_1, A_2, \ldots, A_6$.
  - Then, $((A_1A_2)A_3)$ is an optimal solution to A_1, A_2, A_3 and $(A_4(A_5A_6))$ is an optimal solution to A_4, A_5, A_6.

Step 2: A recursive solution

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

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

- The recursion tree for the computation of $m[1,4]$.



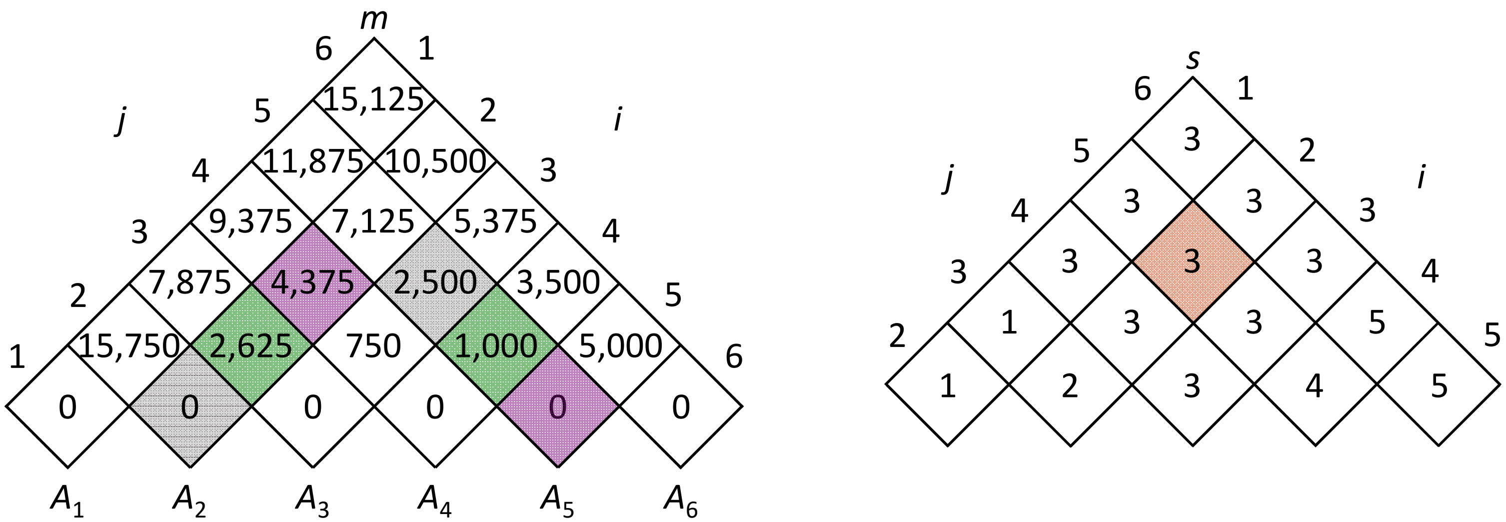
Step 3: Computing the optimal costs

- Based on the recursive formula, we could easily write an exponential-time recursive algorithm to compute the compute the minimum cost $m[1, n]$ for multiplying $A_1A_2 \ldots A_n$.

- There are only $\binom{n}{2} + n = \Theta(n^2)$ distinct subproblems, one problem for each choice of i and j satisfying $1 \leq i \leq j \leq n$.

- We can use dynamic programming to compute the solutions bottom up.

Dependencies between the subproblems



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

- $s[i, j]$: index k achieved the optimal cost in computing $m[i, j]$.

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

MATRIX-CHAIN-ORDER pseudocode

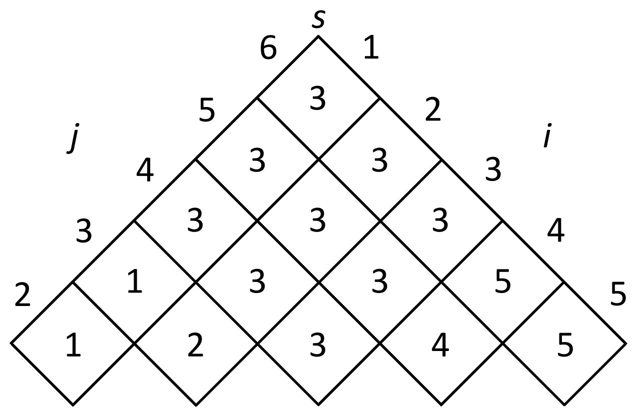
```
MATRIX-CHAIN-ORDER(p)
    n ← length[p] – 1
    for i ← 1 to n
        m[i, i] ← 0
    for ℓ ← 2 to n        /* ℓ is the chain length*/
        for i ← 1 to n – ℓ + 1
            j ← i + ℓ – 1
            m[i, j] ← ∞
            for k ← i to j – 1
                q ← m[i, k] + m[k + 1, j] + p_{i – 1} p_k p_j
                if q < m[i, j]
                    m[i, j] ← q
                    s[i, j] ← k
    return m and s
```

- The loops are nested three deep, and each loop index (ℓ, i, and k) takes on at most $n - 1$ values.
- Time:$O(n^3)$.

Step 4: Constructing an optimal solution

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

```
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 ")"
```



- The call PRINT-OPTIMAL-PARENS(s, 1, n) prints the parenthesization $((A_1(A_2A_3))\ ((A_4A_5)A_6))$.

Outline

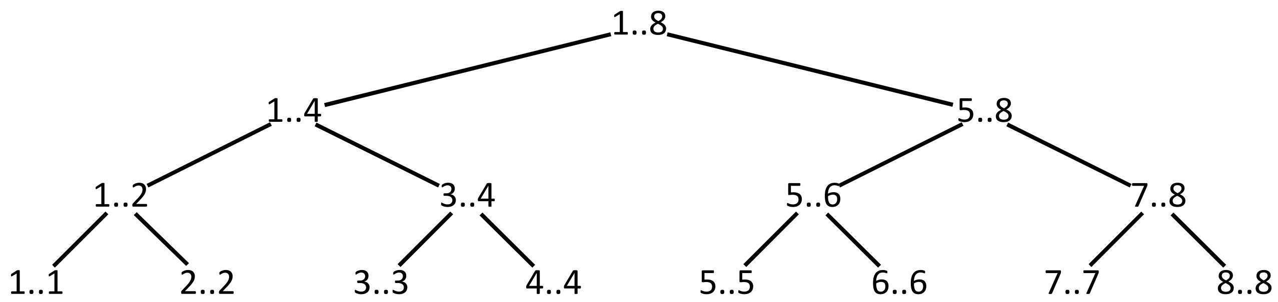
- Rod cutting
- Matrix-chain multiplication
- **Elements of dynamic programming**
- Longest common subsequence
- Optimal binary search trees

Elements of dynamic programming$_{1/2}$

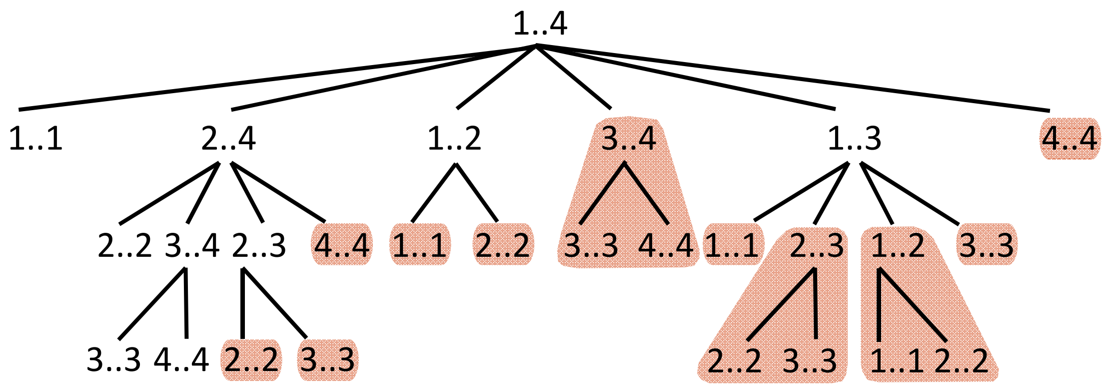
- **Optimal substructure**
  - An optimal solution to a problem contains an optimal solution to subproblems.
    - If $((A_1A_2)A_3)(A_4(A_5A_6))$ is an optimal solution to $A_1, A_2, \ldots, A_6$, then $((A_1A_2)A_3)$ is an optimal solution to A_1, A_2, A_3 and $(A_4(A_5A_6))$ is an optimal solution to A_4, A_5, A_6.
- **Overlapping subproblems**
  - A recursive algorithm revisits the same problem over and over again.
  - 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 the recursion.

Elements of dynamic programming$_{2/2}$

- Example: merge sort



- Example: matrix-chain



Outline

- Rod cutting
- Matrix-chain multiplication
- Elements of dynamic programming
- **Longest common subsequence**
- Optimal binary search trees

Longest-common-subsequence

- A **subsequence** is a sequence that can be derived from another sequence by deleting some elements.
- For example:
  - $\langle K, C, B, A\rangle$ is a subsequence of $\langle \boldsymbol{K}, G, \boldsymbol{C}, E, \boldsymbol{B}, B, \boldsymbol{A}\rangle$.
  - $\langle B, C, D, G\rangle$ is a subsequence of $\langle A, C, \boldsymbol{B}, E, G, \boldsymbol{C}, E, \boldsymbol{D}, B, \boldsymbol{G}\rangle$.
- **Longest-common-subsequence problem**
  - **Input:** 2 sequences, $X = \langle x_1, x_2,\ldots, x_m\rangle$ and $Y = \langle y_1, y_2,\ldots, y_n\rangle$.
  - **Output:** A maximum-length common subsequence of X and Y.
- For example: $X = \langle A, B, C, B, D, A, B\rangle$ and $Y = \langle B, D, C, A, B, A\rangle$.
  - $\langle B, C, A\rangle$is a common subsequence of both X and Y.
  - $\langle B, C, B, A\rangle$is an longest common subsequence (**LCS**) of X and Y.

Step 1: Characterizing an LCS

- **Brute-force algorithm:**
  - For every subsequence of X, check whether it is a subsequence of Y.
- Time: $\Theta(n2^m)$.
  - 2^m subsequences of X to check.
  - Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, from there scan for second, and so on.
- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, we define the ith **prefix** of X, as $X = \langle x_1, x_2, \ldots, x_i \rangle$.
- For example:
  - $X = \langle A, B, C, B, D, A, B \rangle$.
  - $X_4 = \langle A, B, C, B \rangle$ and X_0 is the empty sequence.

Optimal substructure of an LCS

- **Theorem 15.1**

  Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ 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 $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
  3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

- For example:
  - $X = \langle A, B, C, B, D, A, B \rangle$, $Y = \langle B, D, C, A, B \rangle$ and $Z = \langle B, C, A, B \rangle$ is an LCS of X and Y. If $x_7 = y_5$, then $z_4 = x_7 = y_5$ and $Z_3 = \langle B, C, A \rangle$ is an LCS of X_6 and Y_4.
  - $X = \langle A, B, C, B, D, A, D \rangle$, $Y = \langle B, D, C, B, A \rangle$ and $Z = \langle B, C, A \rangle$ is an LCS of X and Y. If $x_7 \neq y_5$, then $z_3 \neq x_7$ implies that $Z_3 = \langle B, C, A \rangle$ is an LCS of X_6 and Y_5.
  - $X = \langle A, B, C, B, D, A, A \rangle$, $Y = \langle B, D, C, A, B \rangle$ and $Z = \langle B, C, A \rangle$ is an LCS of X and Y. If $x_7 \neq y_5$, then $z_3 \neq y_5$ implies that $Z_3 = \langle B, C, A \rangle$ is an LCS of X_7 and Y_4.

Step 2: A recursive solution

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

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

Step 3: Computing the length of an LCS

- Based on the recursive formula, we could easily write an exponential-time recursive algorithm to compute the length of an LCS of two sequences.
- There are only $\Theta(mn)$ distinct subproblems.
- We can use dynamic programming to compute the solutions bottom up.

LCS-LENGTH pseudocode

```
LCS-LENGTH(X, Y)
    m ← length[X]; n ← length[Y]
    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] ← "↖"
            else if c[i - 1, j] ≥ c[i, j - 1]
                c[i, j] ← c[i – 1, j]
                b[i, j] ← "↑"
            else  c[i, j] ← c[i, j – 1]
                b[i, j] ← "←"
    return c and b
```

	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

- Time: $O(mn)$.

Step 4: Constructing an LCS

- Whenever we encounter a "↖" in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the LCS.

```
PRINT-LCS(b, X, i, j)
    if i = 0 or j = 0
        return
    if b[i, j] = "↖"
        PRINT-LCS(b, X, i – 1, j – 1)
        print x_i
    elseif b[i, j] = "↑"
        PRINT-LCS(b, X, i – 1, j)
    else PRINT-LCS(b, X, i, j – 1)
```

	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

- This procedure prints "*BCBA*".

Outline

- Rod cutting
- Matrix-chain multiplication
- Elements of dynamic programming
- Longest common subsequence
- **Optimal binary search trees**

Optimal binary search trees

- **Input:** A sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order.
  A sequence $D = \langle d_0, d_1, \ldots, d_n \rangle$ of $n + 1$ dummy keys.
  - $k_1 < k_2 < \cdots < k_n$.
  - d_0 = all values < k_1. d_n = all values > k_n.
  - d_i = all values between k_i and k_{i+1}.
  - For each key k_i, a probability p_i that a search is for k_i.
  - For each key d_i, a probability q_i that a search is for d_i.

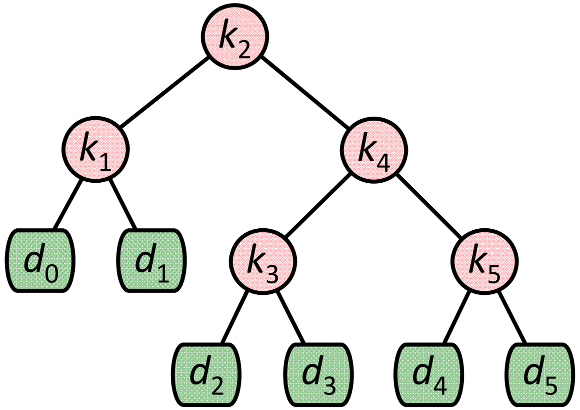
- **Output:** A BST with minimum expected search cost.
  - $\text{E[search cost in } T] = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=1}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$

$$= \sum_{i=1}^{n} p_i + \sum_{i=1}^{n} q_i + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} \text{depth}_T(d_i) \cdot q_i$$

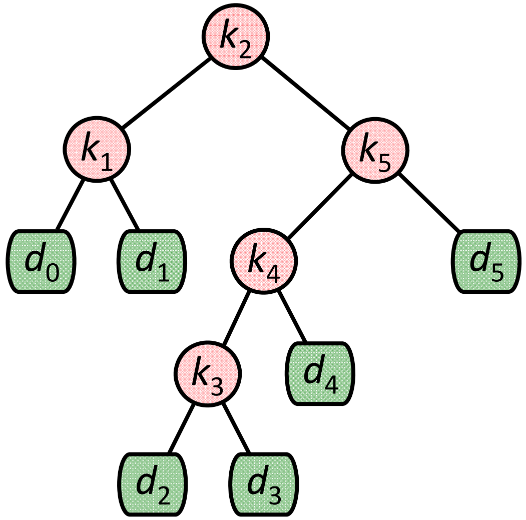
$$\boxed{\sum_{i=1}^{n} p_i + \sum_{i=1}^{n} q_i = 1}$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} \text{depth}_T(d_i) \cdot q_i$$

An example



Expected search cost 2.80.



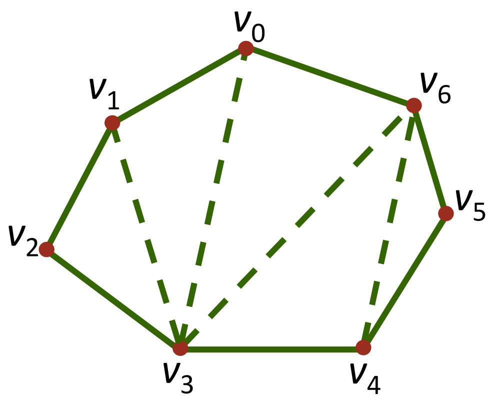
Expected search cost 2.75.
This tree is optimal.

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

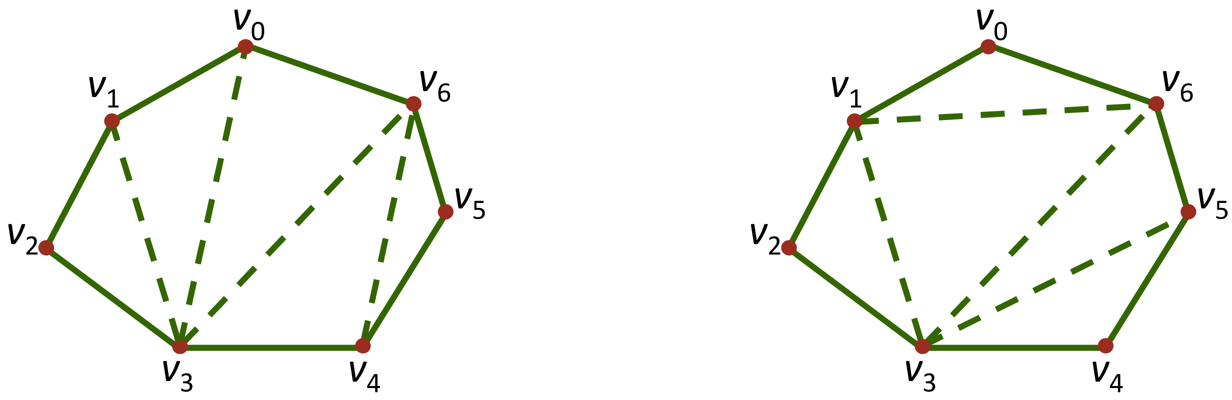
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Optimal polygon triangulation$_{1/2}$

- If $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$ is a **convex polygon**, it has *n* **sides**, $\overline{v_0 v_1}$, $\overline{v_1 v_2}$, …, $\overline{v_{n-1} v_0}$.
- Given two nonadjacent vertices v_i and v_j, the segment $v_i v_j$ is a **chord** of the polygon.
- A **triangulation** of a polygon is a set *T* of chords of the polygon that divide the polygon into disjoint triangles.



Optimal polygon triangulation$_{2/2}$



Two ways of triangulating a convex polygon.

- **Optimal polygon triangulation problem**
  - **Input:** A convex polygon $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$.
    - A weighting function w defined on triangles formed by sides and chords of P.
  - **Output:** A triangulation that minimizes the sum of the weights of the triangles in the triangulation.

0-1 knapsack problem-- using DP

- **Input:** A set $A = \{a_1, a_2, \ldots, a_n\}$ of n items and a knapsack of capacity C.
  - Each item a_i is worth v_i dollars and weighs w_i pounds.
- **Output:** A subset of items whose total size is bounded by C and whose profit is maximized.
  - **Each item must either be taken or left behind.**
- For example: