Dynamic Programming Overview And 1-Dimensional Example

When There's No Greedy Choice Property, We Have To Evaluate All Possible Subproblem Divisions And Pick The Optimal Division

Dynamic Programming

- Applied to optimization problems
- There should still be the optimal substructure property.
  - Just like in greedy algorithms
- Doesn't require the greedy choice property.
  - Have to evaluate all possible divisions into different sub-problems
  - This means dynamic programming can be also applied to the optimization problems we solved using greedy algorithms (but usually less efficient).
- "Programming" in that we complete a table of solutions of sub-problems, not meaning coding in a programming language.

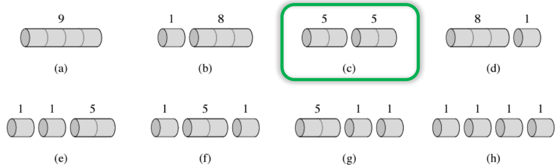
1-Dimensional Dynamic Programming Example

- Rod-cutting problem
  - Rod pieces of different lengths are sold at different prices (table of prices)
  - Given a rod of original length n (probably quite long), what would be the maximum revenue obtainable by cutting up the rod and selling the pieces?

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

Sample price table (CLRS Fig. 15.1)

Optimal: Max revenue 10, compared to 9 (a, b, d), 7 (e, f, g), 4 (h)



All possible cuts of a rod of length 4 (CLRS Fig. 15.2)

How To Solve Rod-Cutting Problem

- Brute-force?
  - 2^{n-1} ways to cut an n-inch rod w/ duplicates (e.g., 1/1/2, 1/2/1, 2/1/1)
  - Eliminating duplicates, it's still huge: $\cong e^{\pi\sqrt{\frac{2n}{3}}}/4n\sqrt{3}$ (CLRS pp.361 footnote)
- Optimal subproblem structure property
  - Given an n-inch rod, assume an optimal solution is to cut the rod into $i_1, i_2 \dots, i_k$ inches ($n = i_1 + i_2 + \cdots + i_k$ for some $1 \leq k \leq n$), revenue r_n.
  - Then, no matter what i_1's value is, the remaining cuts ($i_2, i_3, \dots, i_k$) must be an optimal solution of cutting an $(n - i_1)$-inch rod!
  - Sounds simple and obvious, but needs proof.
    - Proof by contradiction, using "cut-and-paste" technique.

Proving Optimal Subproblem Structure Property: Proof By Contradiction

- Hypothesis: Suppose $(i_2, i_3, \dots, i_k)$ is not an optimal solution of cutting an $(n - i_1)$-inch rod. Let's call its revenue r_{n-i_1} $(= p_{i_2} + p_{i_3} + \cdots + p_{i_k})$.
- Then there must be a different cut of an $(n - i_1)$-inch rod that gives a higher revenue than r_{n-i_1}.
  - Let's call such a cut $(i'_1, i'_2, \dots, i'_{k'})$ and its revenue r'_{n-i_1} (we know $r'_{n-i_1} > r_{n-i_1}$).

Cut-And-Paste Technique

- Then consider the cut $(i_1, i'_1, i'_2, \dots, i'_{k'})$ of an n-inch rod.
  - Its revenue is now:
    - $p_{i_1} + r'_{n-i_1}$
    - $> p_{i_1} + r_{n-i_1}$ (Use $r'_{n-i_1} > r_{n-i_1}$)
    - $= r_n$ (the revenue of the originally assumed optimal solution).
  - This is a contradiction!
    - There's a revenue bigger than the optimal (maximum) revenue.
  - This means the hypothesis must be wrong.
  - Therefore, $(i_2, i_3, \dots, i_k)$ must be an optimal solution of cutting an $(n - i_1)$-inch rod. Q.E.D.

Cut-And-Paste Explained

- We "cut" the solution to the sub-problem and assume it's not optimal.
- Then we "pasted" another solution that's optimal, and derive a contradiction, leading us to conclude that the "cut" solution must be optimal.
- A typical pattern of optimal subproblem structure property proof
- Will omit the cut-and-paste proofs on future problems.
  - But everyone is expected to correctly write this kind of proof for any optimization problem that can be solved using either dynamic programming or greedy method.

Optimal Subproblem Structure of Rod-Cutting

- Maximum revenue r_n of cutting an n-inch rod is the maximum of:
  - $p_1 + r_{n-1}$ (first cut is at 1-inch)
  - $p_2 + r_{n-2}$ (first cut is at 2-inch)
  - …
  - $p_{n-1} + r_1$ (first cut is at $(n-1)$-inch)
  - p_n (no cut, or first cut is at n-inch)
- In other words, the solution is equivalent to build a table of $r_1, r_2, \dots, r_{n-1}$ systematically ("programming") and apply the following equation:

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

Naïve/Inefficient Implementation

- Directly translate the recursive equation into recursive code:

```
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
```

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

- Very inefficient, duplicate computations: 2^n calls to CUT-ROD()!
  - Solve $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$ & $T(0) = 0$ ($T(n)$ is the number of calls to CUT-ROD()), and you get $T(n) = 2^n$.

Why So Inefficient: Call Graph

- Draw all CUT-ROD() calls in a tree and see how many duplicates.

```
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
```

- Very similar to recursive Fibonacci code.

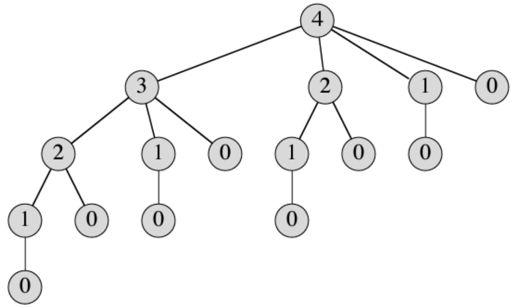
Memoization: Avoid Unnecessary Duplicate Computations By Remembering

- Set up a space (table) for remembering (memoizing) subproblem solutions
- Fill in a blank entry of the table as soon as the solution for that entry is found.
- Look it up first before making any further recursive calls on any other CUT-ROD() call.

```
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)
```

```
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
```

How Memoization Saves the Recursive Algorithm

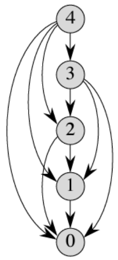


r[0]	r[1]	r[2]	r[3]	r[4]

Top-Down (Recursive) vs. Bottom-Up (Iterative)

- Recursive code is a direct translation of the optimal subproblem structure property equation that we must establish anyway.
- It's "top-down" in that we start from top (CUT(n)), getting down to the bottom (CUT(0)) and rolling up back to top.
- Care (memoization) must be taken to avoid unnecessary duplicate computation while rolling up back to top.
- Noticing that the actual solutions are filled in the order of r[0], r[1], r[2], r[3], r[4], and that to find r[j], we just need to know r[0], r[1], …, r[j-1],
  - Why not just immediately start from bottom (r[0]) and solving up to top?

Bottom-Up Iterative Algorithm and Insight



Subproblem graph (CLRS Fig. 15.4)

```
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]
```

Time/Space Complexities

- Time
  - BOTTOM-UP-CUT-ROD(): $1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2} = \Theta(n^2)$
  - MEMOIZED-CUT-ROD(): Same as above. $\Theta(n^2)$
- Space
  - BOTTOM-UP-CUT-ROD(): $\Theta(n)$ – The array r[0..n]
  - MEMOIZED-CUT-ROD(): $\Theta(n)$ – The array r[0..n] and the maximum recursion depth
    - Not worse than iterative code as before, because of the array needed.
- Still, iterative code performs better with smaller constant factors

Reconstructing a Solution: The Actual Cut

- Here, the "solution" means not the maximum revenue value, but the cut itself that gives us the maximum revenue (the actual optimal cut).
- Our code so far didn't consider the actual optimal cut.
- Not much more work, though:
  - Whenever we update the max-so-far (q), we also update the index number that gives that max-so-far value.
    - Meaning we need extra space to store those index numbers.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[1..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
```

Actual Reconstruction of the Cut

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
$s[i]$		1	2	3	2	2	6	1	2	3	10

```
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]
```

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

Solution To The Modified 10-Inch Rod Cutting

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

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[1..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
```

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	**21**	**22**

Matrix-Chain Multiplication Problem: A 2-Dimensional Dynamic Programming Example

When The Table We Need To Build Is 2-Dimensional

Matrix Multiplication

```
MATRIX-MULTIPLY(A, B)
if A.columns ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows × 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 · b_kj
    return C
```

- Given a $p \times q$ matrix A and a $q \times r$ matrix B,
  - Their product $C = AB$ is a $p \times r$ matrix.
  - And it takes pqr scalar multiplications.
- For example, consider a chain of three matrices $\langle X, Y, X \rangle$
  - Where X is 10×100, Y is 100×5, and Z is 5×50.
  - To compute XYZ, there are two ways: $(XY)Z$ and $X(YZ)$
    - $(XY)Z$: $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$ multiplications
    - $X(YZ)$: $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$ multiplications!!!
- It does matter how to decide the order of matrix multiplications!

Matrix-Chain Multiplication Problem

- Given a chain $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices
  - Where for $i = 1,2,\dots,3$, matrix A_i has dimension $p_{i-1} \times p_i$
  - Find a way to compute the product $A_1 A_2 \cdots A_n$ (we call it fully parenthesizing) which requires the minimum number of scalar multiplications.
  - "Parenthesizing" because that determines how the matrix product is computed:
    - For $A_1 A_2 A_3$, we have $(A_1 A_2) A_3$ and $A_1 (A_2 A_3)$ and these are parenthesizations of the chain.
  - Brute-force search doesn't work, because # parenthesizations $P(n)$ is:
    - 1 if $n = 1$, $\sum_{k=1}^{n-1} P(k) P(n-k)$ if $n \geq 2$.
    - It's at least $\Omega(2^n)$ (Exercise 15.2-3 in CLRS)

Dynamic Programming To The Rescue

- Because there's the following optimal subproblem structure property:
- For matrix multiplication $A_i A_{i+1} \cdots A_j$ (we denote this as $A_{i..j}$) with $i \leq j$,
  - An optimal parenthesization of this chain must be split into 2 chains at a certain point between A_k and A_{k+1} ($i \leq k < j$).
  - And the two subchains corresponding to $A_{i..k}$ and $A_{k+1..j}$ must be optimal!
    - Why? Simple cut-and-paste technique application will do the proof.
    - We get these subproblems with arbitrary ranges.
      - So that's why we started with i and j, not with 1 and n.
  - The original problem is for $A_{1..n}$, finding optimal k, which results in two subproblems $A_{1..k}$ and $A_{k+1..n}$, finding optimal k' and k'', continuing.

Establishing Cost Function Recurrence

- Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute $A_{i..j}$.
- Observe that:
  - $m[i,j] = 0$ if $i = j$ (no matrix multiplication in range)
  - If $i < j$, $m[i,j]$ is the minimum of the following:
    - $m[i,i+1] + m[i+2,j] + p_{i-1} p_{i+1} p_j$ (Compute $A_{i..j}$ as $A_{i..i+1} A_{i+2..j}$)
    - $m[i,i+2] + m[i+3,j] + p_{i-1} p_{i+2} p_j$ (Compute $A_{i..j}$ as $A_{i..i+2} A_{i+3..j}$)
    - …
    - $m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$ (Compute $A_{i..j}$ as $A_{i..k} A_{k+1..j}$)
    - …
    - $m[i,j-1] + m[j,j] + p_{i-1} p_{j-1} p_j$ (Compute $A_{i..j}$ as $A_{i..j-1} A_{j..j}$)
    - $\min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \}$

Top-Down Recursive Implementation

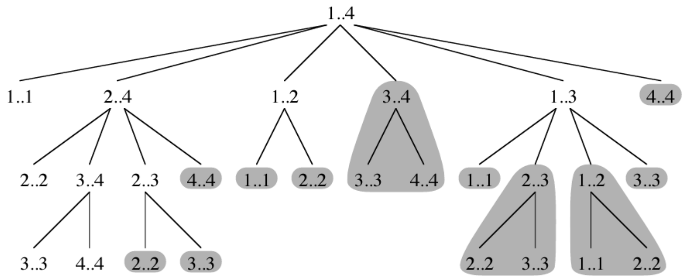
- Direct translation of the recurrence equations into code
- Overlapping subproblems
  - E.g., $A_{i..j}$ will show up in $A_{1..j}$ and $A_{i..n}$
  - In fact, there'll be many repeated duplicate calls with same i and j.
    - Can prove that it's $\Omega(2^n)$. See pp.386.
  - Thus, naïve top-down recursive code doesn't work.
  - Memoization w/ lookup helps

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
        + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
        + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

```
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] = ∞
return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    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]
```

Recursion Tree With Or Without Memoization



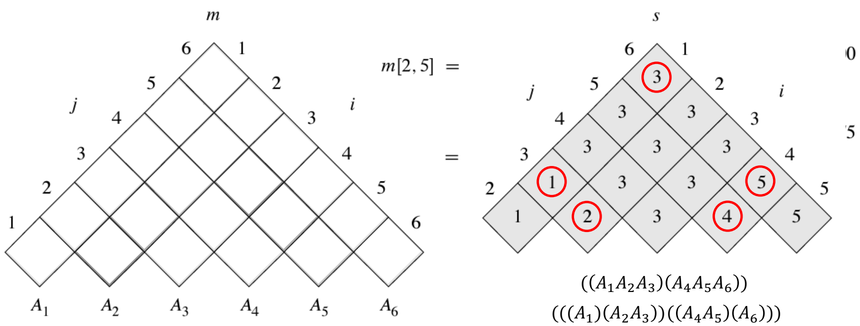
CLRS Figure 15.7

Bottom-Up Iterative Implementation

- Observe that to compute $m[i,j]$ for the chain $A_{i..j}$ where chain length $l = j - i + 1$,
  - We need to know $m[i', j']$ for all chains of lengths $(j' - i' + 1)$ up to $l - 1$.
  - This gives us the "bottom-up" approach:
    - Compute $m[i,j]$ for all chains of lengths 1. There are n such chains.
      - $m[1,1], m[2,2], \dots, m[n,n]$
    - Using the previous results, compute $m[i,j]$ for all chains of lengths 2. There are $n-1$ such chains.
      - $m[1,2], m[2,3], \dots, m[n-1,n]$
    - …
    - Using the previous results, compute $m[i,j]$ for all chains of length n
      - Of course there's only 1 such chain for $m[1,n]$.

Example (Figure 15.5, CLRS pp.376)

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



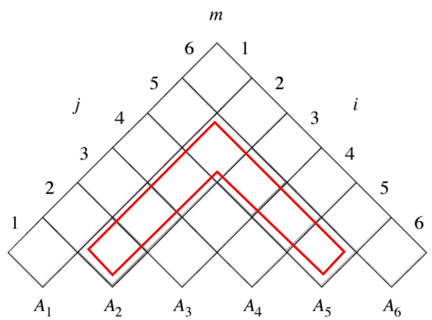
$((A_1 A_2 A_3)(A_4 A_5 A_6))$

$(((A_1)(A_2 A_3))((A_4 A_5)(A_6)))$

$(((A_1)((A_2)(A_3)))(((A_4)(A_5))(A_6)))$

Simplified (as in PRINT-OPTIMAL-PARENS()): $((A_1(A_2A_3))((A_4A_5)A_6))$

Actual Code



```
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
```

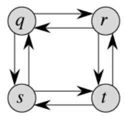
Time/Space Complexity Analysis

- MATRIX-CHAIN-ORDER() time:
  - Loops nested three deep, each with up to $n-1$ values → $O(n^3)$
  - In fact, it's also $\Omega(n^3)$, therefore $\Theta(n^3)$
- Space: m and s tables taking $\Theta(n^2)$
- Same time/space complexities for MEMOIZED-MATRIX-CHAIN()
  - Recursion call stack overhead for space complexity dominated by m and s tables.
  - Still, slower in constant factors, so use iterative MATRIX-CHAIN-ORDER() in production.

Insights Into Dynamic Programming

What Makes Dynamic Programming Possible

Optimal Substructure



- A hallmark property of an optimization problem that allows dynamic programming solution
- An optimal solution to the problem contains within it optimal solutions to sub-problems.
- Proof is possible using the "cut-and-paste" technique we've used so far.
- Not all optimization problems exhibit optimal substructure. See the graph on upper-right corner.
  - Shortest path: Yes. If $u \to \cdots \to w \to \cdots \to v$ is shortest from u to v, then $u \to \cdots \to w$ and $w \to \cdots \to v$ must be shortest for each of (u, w) and (w, v).
  - Longest simple path: No. $q \to r \to t$ is simply longest from q to t, but $q \to r$ is not for q to r and $r \to t$ is not for r to t.

Factors Affecting Running Time Of Dynamic Programming Algorithms

- Two factors:
  - Number of sub-problems overall
  - How many choices we should look at for each subproblem
- E.g., Rod-cutting:
  - $\Theta(n)$ sub-problems overall, at most n choices to examine for each subproblem
  - Giving us an $O(n^2)$ running time.
- E.g., Matrix-chain multiplication:
  - $\Theta(n^2)$ sub-problems overall ($m[i,j]$ for $A_{i..j}$), at most $n-1$ choices to examine for each subproblem
  - Giving us an $O(n^3)$ running time.

Overlapping Sub-problems

- A subproblem appearing multiple times when solving other sub-problems
  - E.g., $A_{i..j}$ in matrix-chain problem could show up in $A_{1..j}$, $A_{i..n}$, $A_{i..j+1}$, … .
- Naïve recursive top-down code (that directly translates the cost (or benefit) function's recursive definition) would perform unnecessarily repeated duplicate computations. (Memoization helps.)
- Bottom-up iterative code solves smallest sub-problems first, then proceeds to solving next smallest sub-problems using the previous results, all the way up to the original problem.

Longest Common Subsequence Problem

Another 2-Dimensional Dynamic Programming Algorithm Example

Longest Common Subsequence (LCS)

- Given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$,
  - Another sequence $Z = \langle z_1, z_2, \dots, z_k \rangle$ is a subsequence of X if each element in Z corresponds to a distinct element in X, and their ordering is the same as in X.
  - E.g., Given $X = \langle A, B, C, B, D, A, B \rangle$, $Z = \langle B, C, D, B \rangle$ is a subsequence of X.
  - Note that <u>a subsequence is NOT a substring</u>!
    - No need for elements in Z to appear in X consecutively.
- Longest-common-subsequence problem
  - Given two sequences X and Y, find a maximum length common subsequence of them.
  - E.g., For $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$,
    - $\langle B, C, A \rangle$ is a common sequence, but not a longest.
    - $\langle B, C, B, A \rangle$ is a longest common subsequence (there's no common subsequence of length 5)
- Motivating example: Two DNA sequences and finding their similarity (CLRS)

Optimal Substructure Of LCS (Theorem 15.1)

- Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be two sequences, and let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y. Then the following three properties must hold:
  1) If $x_m = y_n$, then z_k must be equal to x_m and y_n, and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
     - S_i is a prefix subsequence $\langle s_1, s_2, \dots, s_i \rangle$ for a sequence $S = \langle s_1, s_2, \dots, s_l \rangle$ (with $i \le l$)
     - Obvious to see, prove by contradiction using cut-and-paste technique.
  2) If $x_m \ne y_n$ and $z_k \ne x_m$, then Z must be an LCS of X_{m-1} and Y.
     - Also obvious to see, also prove by contradiction using cut-and-paste.
  3) If $x_m \ne y_n$ and $z_k \ne y_n$, then Z must be an LCS of X and Y_{n-1}.
     - Symmetric to 2).

Recursive Definition Of Value Function To Maximize

- Let $c[i,j]$ be the length of an LCS of the prefix sequences X_i and Y_j.
- Base case definition: $c[i,j] = 0$ if $i = 0$ or $j = 0$.
  - One of the sequences is empty, so no common subsequence.
- And according to the theorem,
  - $c[i,j] = c[i-1,j-1] + 1$ if $i,j > 0$ and $x_i = y_j$.
  - $c[i,j] = \max(c[i,j-1], c[i-1,j])$ if $i,j > 0$ and $x_i \neq y_j$.
- Note some subproblems are ruled out based on conditions.
- Direct recursive top-down code possible, with exponential time.
  - Memoization would always help, but we prefer bottom-up/iterative code.

LCS Dynamic Programming Example (Manual)

- Manually computing the $c[i,j]$ table for $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$:
  - Fill in 0 for row 0 and column 0.
  - Filling in other blanks row-major, left-to-right.
    - Row 1, row 2, row 3, …
    - For each row, column 1, column 2, …
  - Trace back from $c[7,6]$ to recover an actual LCS from the table.
    - We can annotate each entry with arrows, just like in textbook, to help this process.

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

LCS Dynamic Programming Code (Bottom-Up)

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..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] = "↖"
        elseif 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

Reconstructing Actual 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

Performance Analysis and Improvement

- Time: $\Theta(mn)$ – Obvious from the nested loop structure
  - This could be improved by a memoized recursive code for this problem:
    - Because some sub-problems may never need to be evaluated.
    - Best case of memoized recursive code would be $\Theta\left(\min(m, n)\right)$ if one is a suffix of the other.
    - Whereas, there's no such best case on iterative bottom-up implementation.
- PRINT-LCS() time: $O(m + n)$ – # shaded squares in the table
- Space: $\Theta(mn)$ – Obvious from the $c[i, j]$ table.
  - We don't really need the $b[i, j]$ table, because the arrow information can be derived from $c[i, j]$ table just fine.
    - Still this doesn't decrease the asymptotic space complexity.
  - This could be improved to $\Theta(n)$, however, if only for LCS length.
    - Because only 2 rows are needed at any time, not the entire table.
    - But this won't work for reconstructing an actual LCS.
      - Because reconstructing an actual LCS does require the entire table.

Optimal Binary Search Tree

Yet Another 2D Dynamic Programming Example, Very Similar To Matrix-Chain Multiplication, Thus Just Overview

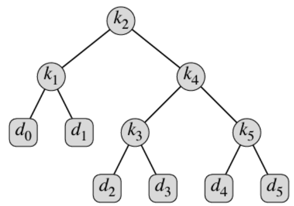
Optimal Binary Search Tree (BST)

- Non-equally likely lookup of values for a specific set of keys
  - Different probabilities for successful searches on existing keys
    - Keys $k_1, k_2, \ldots, k_n$ with probabilities $p_1, p_2, \ldots, p_n$
  - Different probabilities for unsuccessful searches on non-existing values
    - Dummy keys (for non-matches) $d_0, d_1, d_2, \ldots, d_n$ with probabilities $q_0, q_1, q_2, \ldots, q_n$
    - $d_0 < k_1 < d_1 < k_2 < \cdots < d_{n-1} < k_n < d_n$
- Exponentially many different BSTs for the k_i's and d_j's.
- What is the BST that gives the minimum expected search cost?

Search Cost Computation and Example

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

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



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

$$\begin{aligned} \text{E}\left[\text{search cost in } T\right] &= \sum_{i=1}^{n}(\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n}(\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i \end{aligned}$$

Optimal Substructure of Min. Cost BST

- Given keys $k_i, \dots, k_j$,
  - Say one of these keys k_r ($i \le r \le j$) is the root of an optimal BST containing $k_i, \dots, k_j$.
  - Then the left subtree of the root k_r must be an optimal BST for keys $k_i, \dots, k_{r-1}$ (together with dummy keys $d_{i-1}, \dots, d_{r-1}$)
  - And the right subtree of the root k_r must be an optimal BST for keys $k_{r+1}, \dots, k_j$ (together with dummy keys $d_r, \dots, d_j$)
  - Empty subtrees: Empty in the sense that there's no actual (matching) keys.
    - If $r = i$, then k_r's left subtree contains no actual keys, but only one dummy key d_{i-1}.
    - If $r = j$, then k_r's right subtree contains no actual keys, but only one dummy key d_j.

Recursive Definition of Cost Function

- Let $e[i, j]$ be the expected cost of searching an optimal BST containing keys $k_i, \dots, k_j$. Ultimately we want to compute $e[1, n]$.
- Empty subtree case is easy: $e[i, i-1] = q_{i-1}$ (prob. dummy key d_{i-1})
- Otherwise, we need to pick root k_r with two subtrees.
  - The expected cost of BST for $k_i, \dots, k_j$ will be sum of:
    - Expected cost of BST for $k_i, \dots, k_{r-1}$
    - Expected cost of BST for $k_{r+1}, \dots, k_j$
    - And sum of all probabilities for $d_{i-1}, k_i, d_i, \dots, k_j, d_j$ (call this sum $w(i, j)$)
      - Because the expected cost of the two subtrees are increased by each node's probability.
  - Thus $e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j)$ for an optimal root k_r.
  - To find k_r, we need to try all r for $i \le r \le j$ and pick r that gives the min cost:

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1\,, \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j\,. \end{cases} \quad (15.14)$$

Conclusion

- Very similar to matrix-chain multiplication, except:
  - Different indexing due to 3 parts (left subtree, root, right subtree) instead of 2 parts (left product, right product)
  - $w(i, j)$ computation.
- Other than these, absolutely same algorithm structure.
  - See OPTIMAL-BST() and Figure 15.10 in CLRS.
- Time/space complexities are the same as MATRIX-CHAIN-ORDER()
  - $\Theta(n^3)$ time, $\Theta(n^2)$ space