Dynamic programming

Chapter 15 from textbook

Algorithmic Paradigms

- Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
  - **Example:** Merge sort, Binary search
- Greedy. Build up a solution incrementally, myopically optimizing some local criterion.
  - **Example:** Shortest path using Dijkstra’s algorithm, MST using Prim’s/Kruskal’s algorithm
- Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
  - **Example:** Rod cutting, Matrix multiplication, Largest common subsequence, Edit distance, All pair shortest path

Dynamic Programming History

- Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

- Etymology.
  - Dynamic programming = planning over time.
  - Secretary of Defense was hostile to mathematical research.
  - Bellman sought an impressive name to avoid confrontation.
    - "something not even a Congressman could object to"

Reference: Bellman, R. E. Eye of the Hurricane, An Autobiography.

Dynamic programming

- In Divide-and-conquer approach
  - We partition the problem into “independent” subproblems
  - Solve the subproblems recursively
  - Then combine the solutions to solve the original problem
- Dynamic programming is applicable when
  - We have “overlapping” subproblems
  - Naïve recursive method would solve these overlapping subproblems many many times resulting in exponential (in problem size) running time
  - Dynamic programming avoids this by solving subproblems in a particular order from smaller to larger subproblems such that each subproblem is solved exactly once and saves this in a table for later reuse

Introduction

- Dynamic Programming(DP) applies to optimization problems
  - Such problems can have many possible solutions
  - Each solution has a value and we wish to find a solution having optimal (minimum or maximum) value
  - We call such a solution as an optimal solution as oposed to the optimal solution, since there may be sevaral solutions that achieve the optimal value.
- When we develop dynamic programming algorithm we follow a sequence of four steps
  - Characterize the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Compute the value of an optimal solution in a bottom-up fashion
  - Construct an optimal solution from computed information

Dynamic Programming Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ….
- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Ford-Marshall algorithm for all pair shortest path shortest path
  - Cocke-Kasami-Younger for parsing context free grammars.

Problem #1: Fibonacci numbers

- Let's consider calculating the Fibonacci numbers:
  - 0,1,1,2,3,5,8,13,21,44,65,…
  - Any number is sum of the previous two numbers
- A naïve recursive algorithm will look something like this:

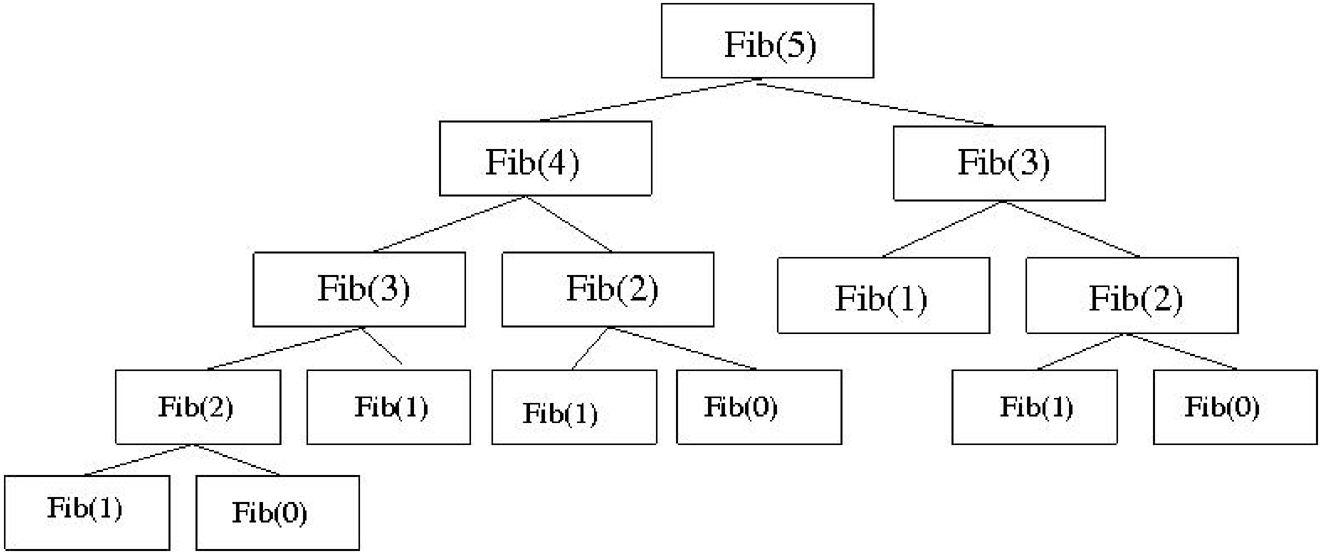
Recursive Algorithm:

```
Fib(n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
   Return Fib(n-1)+Fib(n-2)
}
```

- What is the problem here?

Problem #1: Fibonacci numbers

- Same subproblems are solved repeatedly many times resulting in exponential running time
- T(n)=?



Problem #1: Fibonacci numbers

- Better solution is obtained by
  - Observing optimal substructure $F(n)=F(n-1)+F(n-2)$
  - And solving subproblems in particular order smaller subproblems first and reusing these results
- To compute $F(n)$, maintain an array $M[0,1,\ldots,n]$
  - Set $M[0]=0$, $M[1]=1$
  - For $i=2$ to n

    $M[n]=M[n-1]+M[n-2]$

- Note that this is not an optimization problem but illustrates the main idea

Problem #2: Rod cutting

- Suppose a company buys long rods and sells by cutting them into small pieces for a profit as per the following 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

- **Question:** We are given a rod of length n, and want to maximize revenue, by cutting up the rod into pieces and selling each of the pieces.

- **Example:** we are given a 4 inches rod. What is the best solution to cut up?

Problem #2: Rod cutting

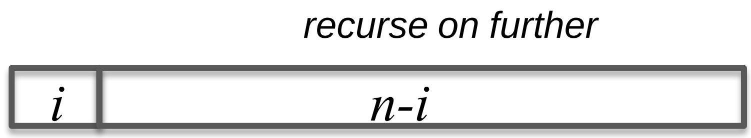
- **Example:** we are given a 4 inches rod. Best solution to cut up? We'll first list the solutions:

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

  - **1.)** Cut into 2 pieces of length 2: $p_2 + p_2 = 5 + 5 = 10$
  - **2.)** Cut into 4 pieces of length 1: $p_1 + p_1 + p_1 + p_1 = 1 + 1 + 1 + 1 = 4$
  - **3-4.)** Cut into 2 pieces of length 1 and 3 (3 and 1): $p_3 + p_1 = 8 + 1 = 9$
  - **5.)** Keep length 4: $p_4 = 9$
  - **6-8.)** Cut into 3 pieces, length 1, 1 and 2 (and all the different orders)
    $p_1 + p_1 + p_2 = 7$ $\quad p_1 + p_2 + p_1 = 7$ $\quad p_2 + p_1 + p_1 = 7$
  - **Total:** 8 cases for $n = 4$ ($= 2^{n-1}$). We can slightly reduce by always requiring cuts in non-decreasing order. But still a lot!

Problem #2: Rod cutting

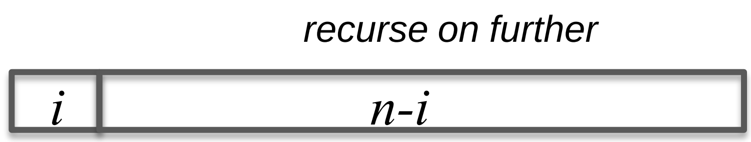
- **Note:** We've computed a brute force solution; all possibilities for this simple small example. But we want more optimal solution!
  - One possible solution is as follows:



  - What are we doing?
    - Cut rod into length i and n-i
    - Only remainder n-i can be further cut (recursed)
  - We need to define:
    - a.) **Maximum revenue** for rod of size n: $\boldsymbol{r_n}$ (that is the solution we want to find).
    - b.) **Revenue (price)** for single rod of length i: $\boldsymbol{p_i}$

Problem #2: Rod cutting

- **Note:** We've computed a brute force solution; all possibilities for this simple small example. But we want more optimal solution!
  - One possible solution is as follows:



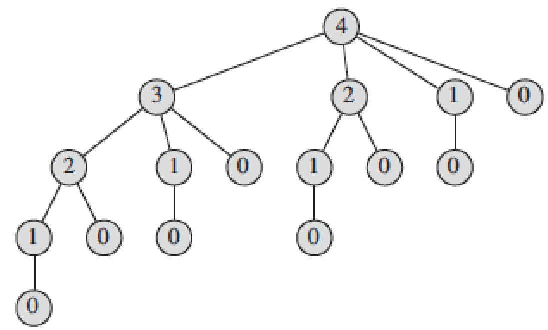
  - Revenue: $p_i + r_{n\text{-}i}$ can be seen by recursing on n-i
  - There many possible choices for i

$$r_n = \max \left\{ \begin{array}{c} p_1 + r_{n\text{-}1} \\ p_1 + r_{n\text{-}2} \\ \ldots \\ p_n + r_0 \end{array} \right\}$$

Problem #2: Rod cutting

- Consider naïve recursive (top-down approach)

```
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
```



  - Problem? Slow runtime (it's essential brute force)!
  - Why? Cut-rod calls itself repeatedly with the same parameter values (tree):
    - Node label: size of the subproblem called on
    - Can be seen by eye that many subproblems are called repeatedly (subproblem overlap)
    - Number of nodes exponential in n i.e., (2^n). therefore exponential number of calls.

Problem #2: Rod cutting

- We have seen, recursive solution is inefficient, since it repeatedly calculates a solution of the same subproblem (overlapping subproblem).
- Instead, solve each subproblem only once AND save its solution. Next time we encounter the subproblem look it up in a hash table or an array
  - This is called recursive **top-down solution with memoization**
- We will also talk about a second solution where we save the solution of subproblems of increasing size (i.e. in order) in an array.
  - Each time we will fall back on solutions that we obtained in previous steps and stored in an array (**bottom-up solution**).

Problem #2: Rod cutting

- Recursive top-down solution: **Cut-Rod with Memoization**
  - Step 1: Initialization

MEMOIZED-CUT-ROD(p, n)

let $r[0\,..\,n]$ be a new array
for $i = 0$ **to** n
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX(p, n, r)

Creates array for holding memoized results, and initialized to minus infinity. Then calls the main auxiliary function.

  - Step 2: The main auxiliary function, which goes through the lengths, computes answers to subproblems and memoizes if subproblem not yet encountered:

MEMOIZED-CUT-ROD-AUX(p, n, r)

if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ **to** n
        $q = \max(q, p[i] +$ MEMOIZED-CUT-ROD-AUX$(p, n - i, r))$
$r[n] = q$
return q

Problem #2: Rod cutting

- Running time of Recursive top-down solution with Memoization
  - Running time is $\Theta(n^2)$
    - Recursive call to a previously solved problem runs immediately.
    - Memoized-Cut-Rod solves each subproblem just once
    - It solves subproblem of size 0,1,2,…,n
    - To solve a subproblem of size n, line 6-7 iterates n times
    - Thus, total number of iterations of this for loop, over all recursive calls of Memoized-Cut-Rod forms an arithmetic series giving a total $\Theta(n^2)$ iterations

Problem #2: Rod cutting

- Bottom-up solution is even simpler
- Each time we reuse previously computed values stored in an array

```
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]
```

Compute maximum revenue if it hasn't already been computed.

- Running time is $\Theta(n^2)$

Problem #3: Matrix-Chain Multiplication

Problem: given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$, compute the product:

$$A_1 \cdot A_2 \cdots A_n$$

- Matrix compatibility:

$$C = A \cdot B \qquad\qquad C=A_1 \cdot A_2 \cdots A_i \cdot A_{i+1} \cdots A_n$$

$col_A = row_B$ $\qquad$ $col_i = row_{i+1}$

$row_C = row_A$ $\qquad$ $row_C = row_{A1}$

$col_C = col_B$ $\qquad$ $col_C = col_{An}$

MATRIX-MULTIPLY(A, B)

if columns[A] ≠ rows[B]

  then error "incompatible dimensions"

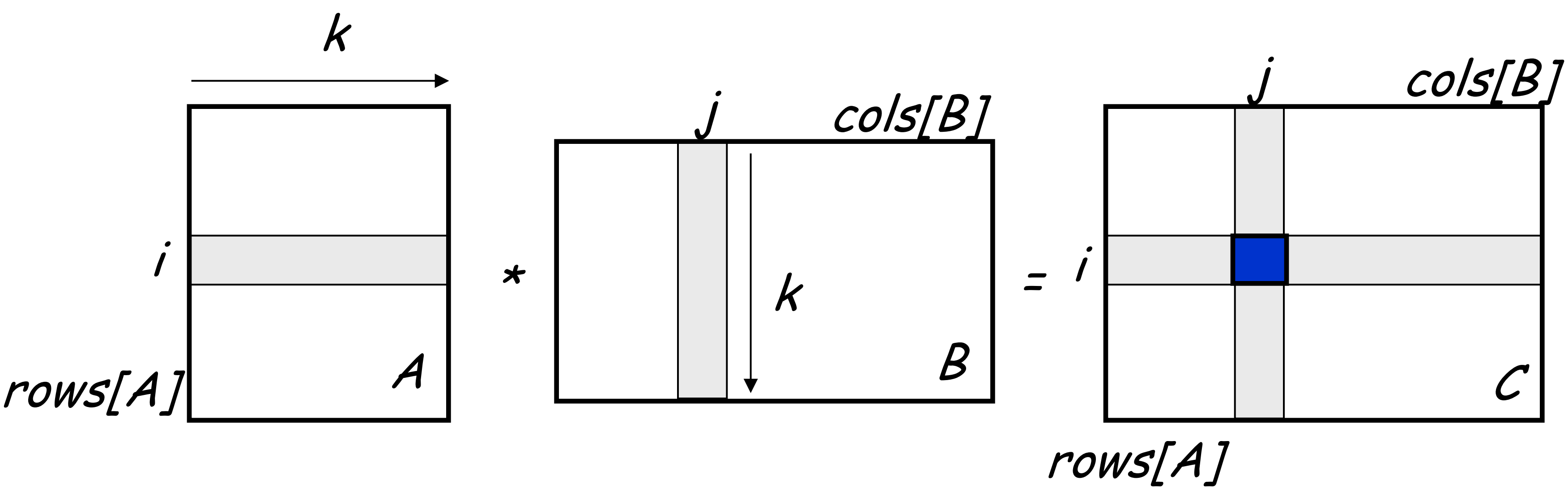
  else for i ← 1 to rows[A]

    do for j ← 1 to columns[B]

      do C[i, j] = 0

        for k ← 1 to columns[A]

          do C[i, j] ← C[i, j] + $\overbrace{A[i, k]\ B[k, j]}^{\text{rows[A] · cols[A] · cols[B] multiplications}}$



Matrix-Chain Multiplication

- In what order should we multiply the matrices?

  $$A_1 \cdot A_2 \cdots A_n$$

- Parenthesize the product to get the order in which matrices are multiplied

E.g.: $A_1 \cdot A_2 \cdot A_3 = ((A_1 \cdot A_2) \cdot A_3)$

$= (A_1 \cdot (A_2 \cdot A_3))$

- Which one of these orderings should we choose?
  - The order in which we multiply the matrices has a significant impact on the cost of evaluating the product

Example

$$A_1 \cdot A_2 \cdot A_3$$

- A_1: 10 x 100
- A_2: 100 x 5
- A_3: 5 x 50

1. $((A_1 \cdot A_2) \cdot A_3)$: $A_1 \cdot A_2 = 10 \times 100 \times 5 = 5{,}000$ (10 x 5)

   $((A_1 \cdot A_2) \cdot A_3) = 10 \times 5 \times 50 = 2{,}500$

   Total: 7,500 scalar multiplications

2. $(A_1 \cdot (A_2 \cdot A_3))$: $A_2 \cdot A_3 = 100 \times 5 \times 50 = 25{,}000$ (100 x 50)

   $(A_1 \cdot (A_2 \cdot A_3)) = 10 \times 100 \times 50 = 50{,}000$

   Total: 75,000 scalar multiplications

one order of magnitude difference!!

Matrix-Chain Multiplication: Problem Statement

- Given a chain of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, where A_i has dimensions $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \cdot A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

$$A_1 \quad \cdot \quad A_2 \quad \cdots \quad A_i \quad \cdot \quad A_{i+1} \quad \cdots \quad A_n$$

$$p_0 \times p_1 \quad p_1 \times p_2 \qquad p_{i-1} \times p_i \quad p_i \times p_{i+1} \qquad p_{n-1} \times p_n$$

What is the number of possible parenthesizations?

- Exhaustively checking all possible parenthesizations is not efficient!
- It can be shown that the number of parenthesizations grows as $\Omega(4^n/n^{3/2})$

(see page 333 in your textbook)

1. The Structure of an Optimal Parenthesization

- Notation:

$$A_{i\ldots j} = A_i\, A_{i+1} \cdots A_j,\ i \leq j$$

- Suppose that an optimal parenthesization of $A_{i\ldots j}$ splits the product between A_k and A_{k+1}, where $i \leq k < j$

$$\begin{aligned} A_{i\ldots j} &= A_i\, A_{i+1} \cdots A_j \\ &= A_i\, A_{i+1} \cdots A_k\, A_{k+1} \cdots A_j \\ &= A_{i\ldots k}\, A_{k+1\ldots j} \end{aligned}$$

Optimal Substructure

$$A_{i\ldots j} = A_{i\ldots k}\, A_{k+1\ldots j}$$

- The parenthesization of the "prefix" $A_{i\ldots k}$ must be an optimal parentesization
- If there were a less costly way to parenthesize $A_{i\ldots k}$, we could substitute that one in the parenthesization of $A_{i\ldots j}$ and produce a parenthesization with a lower cost than the optimum $\Rightarrow$ contradiction!
- An optimal solution to an instance of the matrix-chain multiplication contains within it optimal solutions to subproblems

2. A Recursive Solution

- Subproblem: determine the minimum cost of parenthesizing $A_{i\ldots j} = A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$
- Let $m[i, j]$ = the minimum number of multiplications needed to compute $A_{i\ldots j}$
  - full problem ($A_{1..n}$): $m[1, n]$
  - $i = j$: $A_{i\ldots i} = A_i \Rightarrow m[i, i] = 0$ for $i=1, 2, \ldots, n$

2. A Recursive Solution

- Consider the subproblem of parenthesizing

$$A_{i\ldots j} = A_i\, A_{i+1} \cdots A_j \qquad \text{for } 1 \le i \le j \le n$$

$p_{i-1}p_kp_j$

$$= \underbrace{A_{i\ldots k}}_{m[i,\,k]}\; \underbrace{A_{k+1\ldots j}}_{m[k+1,j]} \qquad \text{for } i \le k < j$$

- Assume that the optimal parenthesization splits the product $A_i\, A_{i+1} \cdots A_j$ at k ($i \le k < j$)

$$m[i,j] = \underbrace{m[i,k]}_{} \quad + \quad \underbrace{m[k+1,j]}_{} \quad + \quad \underbrace{p_{i-1}p_kp_j}_{}$$

min # of multiplications to compute $A_{i\ldots k}$

min # of multiplications to compute $A_{k+1\ldots j}$

# of multiplications to compute $A_{i\ldots k}A_{k\ldots j}$

2. A Recursive Solution (cont.)

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$$

- We do not know the value of k
  - There are j – i possible values for k: k = i, i+1, …, j-1
- Minimizing the cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes:

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

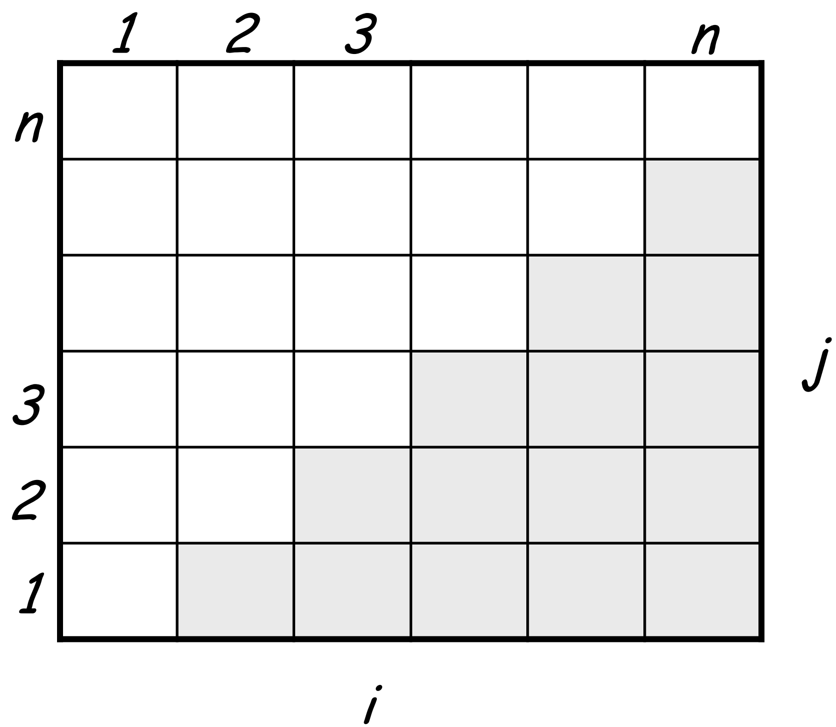
3. Computing the Optimal Costs

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

- Computing the optimal solution recursively takes exponential time!
- How many subproblems?

  $\Rightarrow \Theta(n^2)$

  - Parenthesize $A_{i\ldots j}$ for $1 \le i \le j \le n$
  - One problem for each choice of i and j



3. Computing the Optimal Costs (cont.)

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

- How do we fill in the tables m[1..n, 1..n]?
  - Determine which entries of the table are used in computing $m[i, j]$

$$A_{i\ldots j} = A_{i\ldots k} A_{k+1\ldots j}$$

  - Subproblems' size is one less than the original size
  - **<u>Idea:</u>** fill in m such that it corresponds to solving problems of increasing length

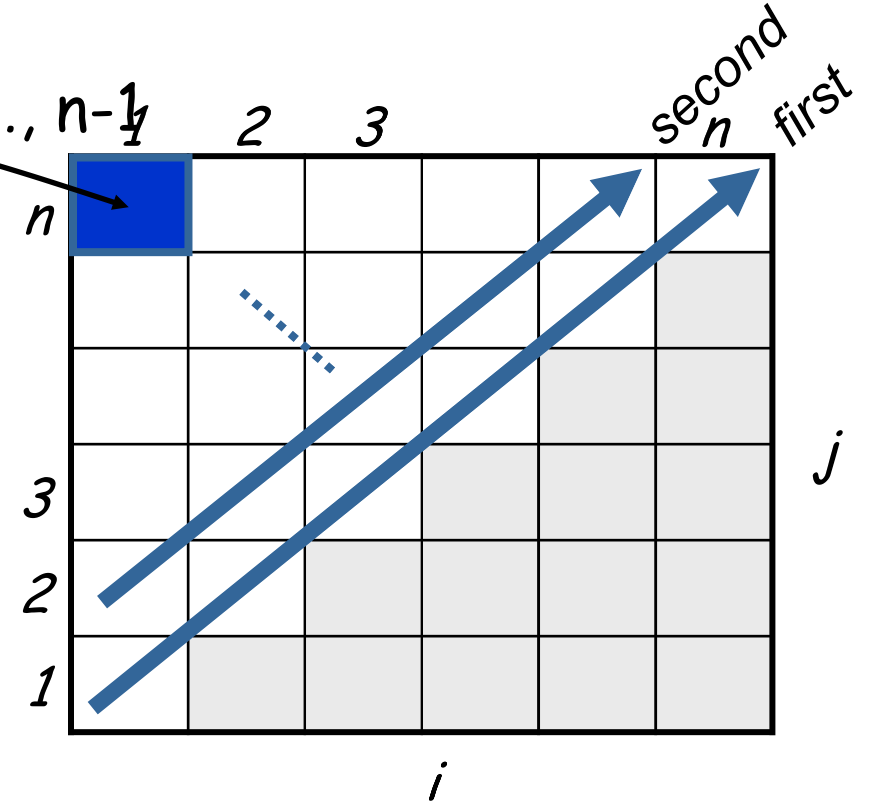
3. Computing the Optimal Costs (cont.)

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

- Length = 1: $i = j$, $i = 1, 2, \ldots, n$
- Length = 2: $j = i + 1$, $i = 1, 2, \ldots, n-1$

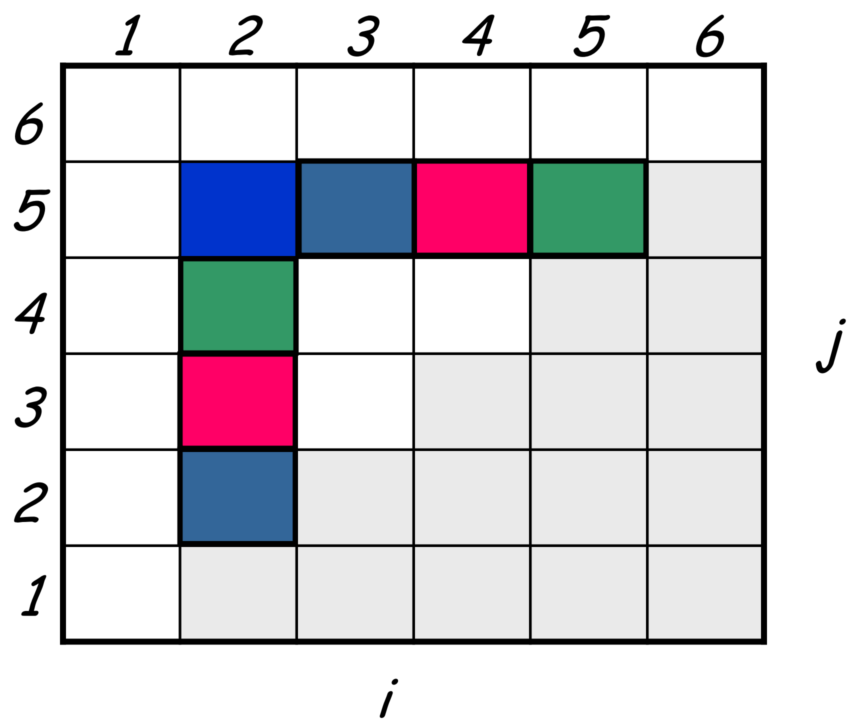
m[1, n] gives the optimal solution to the problem

Compute rows from bottom to top and from left to right



Example: $\min \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\}$

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1p_2p_5 & k = 2 \\ m[2, 3] + m[4, 5] + p_1p_3p_5 & k = 3 \\ m[2, 4] + m[5, 5] + p_1p_4p_5 & k = 4 \end{cases}$$



- *Values $m[i, j]$ depend only on values that have been previously computed*

Example $\min\{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\}$

	1	2	3
3	2 7500	2 25000	0
2	1 5000	0	
1	0		

Compute $A_1 \cdot A_2 \cdot A_3$

- A_1: 10 x 100 (p_0 x p_1)
- A_2: 100 x 5 (p_1 x p_2)
- A_3: 5 x 50 (p_2 x p_3)

$m[i, i] = 0$ for $i = 1, 2, 3$

$$m[1, 2] = m[1, 1] + m[2, 2] + p_0p_1p_2 \qquad (A_1A_2)$$

$$= 0 + 0 + 10 * 100 * 5 = 5{,}000$$

$$m[2, 3] = m[2, 2] + m[3, 3] + p_1p_2p_3 \qquad (A_2A_3)$$

$$= 0 + 0 + 100 * 5 * 50 = 25{,}000$$

$$m[1, 3] = \min \begin{cases} m[1, 1] + m[2, 3] + p_0p_1p_3 = 75{,}000 & (A_1(A_2A_3)) \\ m[1, 2] + m[3, 3] + p_0p_2p_3 = 7{,}500 & ((A_1A_2)A_3) \end{cases}$$

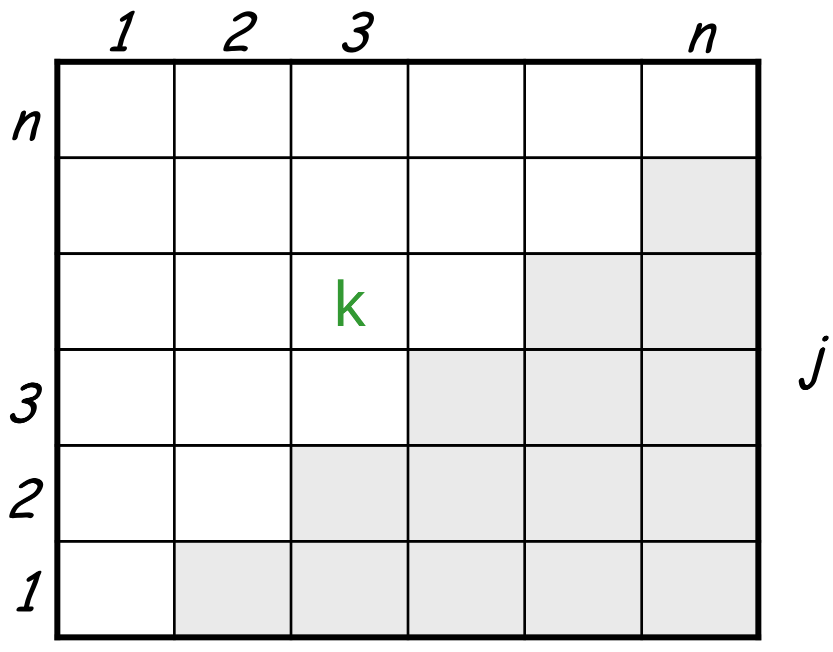
Matrix-Chain-Order(p)

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

$O(N^3)$

4. Construct the Optimal Solution

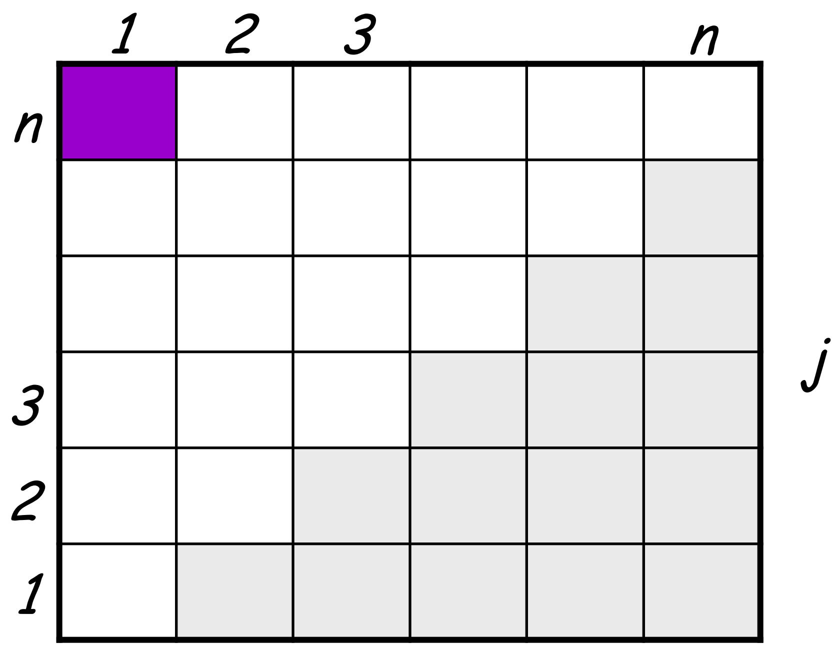
- In a similar matrix s we keep the optimal values of k
- $s[i, j]$ = a value of k such that an optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1}



4. Construct the Optimal Solution

- $s[1, n]$ is associated with the entire product $A_{1..n}$
  - The final matrix multiplication will be split at k = $s[1, n]$

    $$A_{1..n} = A_{1..s[1,n]} \cdot A_{s[1,n]+1..n}$$
  - For each subproduct recursively find the corresponding value of k that results in an optimal parenthesization



4. Construct the Optimal Solution

- $s[i, j]$ = 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}

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

j

i

- $s[1, 6] = 3 \Rightarrow A_{1..6} = A_{1..3}\ A_{4..6}$
- $s[1, 3] = 1 \Rightarrow A_{1..3} = A_{1..1}\ A_{2..3}$
- $s[4, 6] = 5 \Rightarrow A_{4..6} = A_{4..5}\ A_{6..6}$

4. Construct the Optimal Solution (cont.)

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

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

j

i

Example: $A_1 \cdot \cdot \cdot A_6$ $((A_1(A_2A_3))((A_4A_5)A_6))$

PRINT-OPT-PARENS(s, i, j) **s[1..6, 1..6]**

if *i = j*

  then *print "A"$_i$*

  else *print "("*

    PRINT-OPT-PARENS(s, i, s[i, j])

    PRINT-OPT-PARENS(s, s[i, j] + 1, j)

    print ")"

j \ i	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

P-O-P(s, 1, 6) s[1, 6] = 3

i = 1, j = 6 "(" P-O-P (s, 1, 3) s[1, 3] = 1

i = 1, j = 3 "(" P-O-P(s, 1, 1) ⇒ "A_1"

P-O-P(s, 2, 3) s[2, 3] = 2

i = 2, j = 3 "(" P-O-P (s, 2, 2) ⇒ "A_2"

P-O-P (s, 3, 3) ⇒ "A_3"

")"

")" …

Memoization

- Top-down approach with the efficiency of typical dynamic programming approach
- Maintaining an entry in a table for the solution to each subproblem
  - **memoize** the inefficient recursive algorithm
- When a subproblem is first encountered its solution is computed and stored in that table
- Subsequent "calls" to the subproblem simply look up that value

Memoized Matrix-Chain

Alg.: MEMOIZED-MATRIX-CHAIN(p)

1. n ← length[p] - 1
2. **for** i ← 1 **to** n
3. **do for** j ← i **to** n
4. **do** m[i, j] ← ∞
5. **return** LOOKUP-CHAIN(p, 1, n)

Initialize the m table with large values that indicate whether the values of m[i, j] have been computed

Top-down approach

Memoized Matrix-Chain

Alg.: LOOKUP-CHAIN(p, i, j)

Running time is $O(n^3)$

if $m[i, j] < \infty$
    then return $m[i, j]$
if $i = j$
  then $m[i, j] \leftarrow 0$
  else for $k \leftarrow i$ **to** $j - 1$
      do $q \leftarrow$ LOOKUP-CHAIN(p, i, k) +
          LOOKUP-CHAIN(p, k+1, j) + $p_{i-1}p_kp_j$
        if $q < m[i, j]$
          then $m[i, j] \leftarrow q$
return $m[i, j]$

Longest Common Subsequence

- Given two sequences

  $$X = \langle x_1, x_2, \ldots, x_m \rangle$$

  $$Y = \langle y_1, y_2, \ldots, y_n \rangle$$

  find a maximum length common subsequence (LCS) of X and Y

- *E.g.:*

  $$X = \langle A, B, C, B, D, A, B \rangle$$

- Subsequences of X:
  - A subset of elements in the sequence taken in order

    $\langle A, B, D \rangle$, $\langle B, C, D, B \rangle$, etc.

Example

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

- $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are longest common subsequences of X and Y (length = 4)
- $\langle B, C, A \rangle$, however is not a LCS of X and Y

Brute-Force Solution

- For every subsequence of X, check whether it's a subsequence of Y
- There are 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
- Running time: $\Theta(n2^m)$

Making the choice

$X = \langle A, B, D, E \rangle$

$Y = \langle Z, B, E \rangle$

- Choice: include one element into the common sequence (E) and solve the resulting subproblem

$X = \langle A, B, D, G \rangle$

$Y = \langle Z, B, D \rangle$

- Choice: exclude an element from a string and solve the resulting subproblem

Notations

- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ we define the i-th prefix of X, for $i = 0, 1, 2, \ldots, m$

  $$X_i = \langle x_1, x_2, \ldots, x_i \rangle$$

- `c[i, j]` = the length of a LCS of the sequences $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$

A Recursive Solution

Case 1: $x_i = y_j$

e.g.: $X_i = \langle A, B, D, E \rangle$

$Y_j = \langle Z, B, E \rangle$

$$c[i, j] = c[i - 1, j - 1] + 1$$

- Append $x_i = y_j$ to the LCS of X_{i-1} and Y_{j-1}
- Must find a LCS of X_{i-1} and Y_{j-1} $\Rightarrow$ optimal solution to a problem includes optimal solutions to subproblems

A Recursive Solution

Case 2: $x_i \neq y_j$

e.g.: $X_i = \langle A, B, D, G \rangle$

$Y_j = \langle Z, B, D \rangle$

$$c[i, j] = \max \{ c[i-1, j], c[i, j-1] \}$$

- Must solve two problems
  - find a LCS of X_{i-1} and Y_j: $X_{i-1} = \langle A, B, D \rangle$ and $Y_j = \langle Z, B, D \rangle$
  - find a LCS of X_i and Y_{j-1}: $X_i = \langle A, B, D, G \rangle$ and $Y_j = \langle Z, B \rangle$

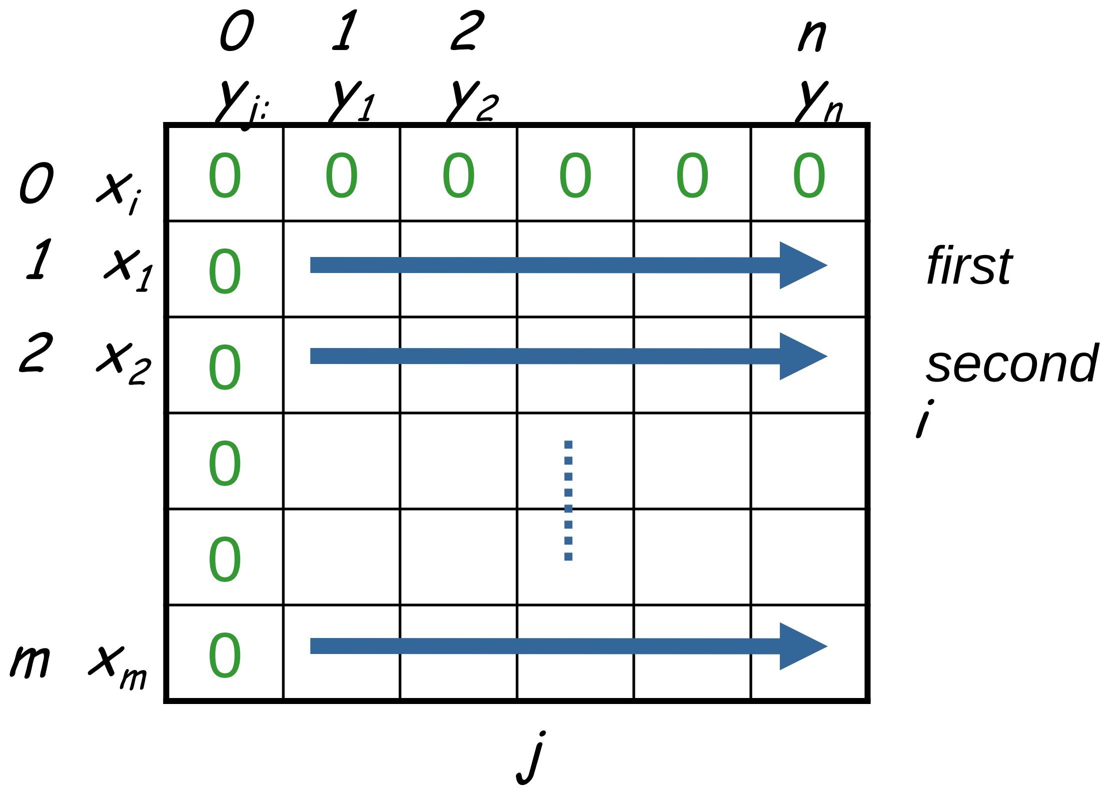
- Optimal solution to a problem includes optimal solutions to subproblems

Overlapping Subproblems

- To find a LCS of X and Y
  - We may need to find the LCS between X and Y_{n-1} and that of X_{m-1} and Y
  - Both the above subproblems has the subproblem of finding the LCS of X_{m-1} and Y_{n-1}
- Subproblems share subsubproblems

3. Computing the Length of the LCS

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



Additional Information

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

b & c:

		0	1	2	3		n	
		y_j:	A	C	D		F	
0	x_i	0	0	0	0	0	0	
1	A	0						
2	B	0			c[i-1,j]			
3	C	0		↖ c[i,j-1]	↑			*i*
		0						
m	D	0						

j

A matrix b[i, j]:

- *For a subproblem [i, j] it tells us what choice was made to obtain the optimal value*
- *If $x_i = y_j$*

  b[i, j] = "↖"
- *Else, if c[i - 1, j] ≥ c[i, j-1]*

  b[i, j] = " ↑ "

  else

  b[i, j] = " ← "

LCS-LENGTH(X, Y, m, n)

```
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i - 1, j - 1] + 1
                 b[i, j ] ← "↖"
            else if c[i - 1, j] ≥ c[i, j - 1]
                then 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
```

The length of the LCS if one of the sequences is empty is zero

Case 1: $x_i = y_j$

Case 2: $x_i \neq y_j$

Running time: $\Theta(mn)$

Example

X = ⟨A, B, C, B, D, A⟩

Y = ⟨B, D, C, A, B, A⟩

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

If $x_i = y_j$

$b[i, j] = $ "↖"

Else if

$c[i - 1, j] \geq c[i, j-1]$

$b[i, j] = $ " ↑ "

else

$b[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

4. Constructing a LCS

- Start at b[m, n] and follow the arrows
- When we encounter a "↖" in b[i, j] ⇒ $x_i = y_j$ is an element of the LCS

		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

PRINT-LCS(b, X, i, j)

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

Running time: $\Theta(m + n)$

Initial call: PRINT-LCS(b, X, length[X], length[Y])

Improving the Code

- What can we say about how each entry $c[i, j]$ is computed?
  - It depends only on $c[i - 1, j - 1]$, $c[i - 1, j]$, and $c[i, j - 1]$
  - Eliminate table b and compute in $O(1)$ which of the three values was used to compute $c[i, j]$
  - We save $\Theta(mn)$ space from table b
  - However, we do not asymptotically decrease the auxiliary space requirements: still need table c

Improving the Code

- If we only need the length of the LCS
  - LCS-LENGTH works only on two rows of c at a time
    - The row being computed and the previous row
  - We can reduce the asymptotic space requirements by storing only these two rows