Computer Algorithms

Segment 3

Dynamic Programming

Why Dynamic Programming?

- Divide-and-Conquer: a top-down approach. Many smaller instances are computed more than once.
- Dynamic programming: a bottom-up approach. Solutions for smaller instances are stored in a table for later use.

- It sometimes happens that the natural way of dividing an instance suggested by the structure of the problem leads us to consider several overlapping subinstances.
- If we solve each of these independently, they will in turn create a large number of identical subinstances.

Why Dynamic Programming?....

- If we pay no attention to this duplication, it is likely that we will end up with an inefficient algorithm.
- If, on the other hand, we take advantage of the duplication and solve each subinstance only once, saving the solution for later use, then a more efficient algorithm will result.
- The underlying idea of dynamic programming is thus quite simple: avoid calculating the same thing twice, usually by keeping a table of known results, which we fill up as subinstances are solved.
- Dynamic programming is a bottom-up technique.

What is Dynamic Programming?

- ***Dynamic Programming*** is a general algorithm design technique.
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems.
- "Programming" here means "planning".

- Main idea:
    - solve several smaller (overlapping) subproblems.
    - record solutions in a table so that each subproblem is only solved once.
    - final state of the table will be (or contain) solution.

What is Dynamic Programming?...

- Dynamic programming solves *optimization problems* by combining solutions to subproblems
- "Programming" refers to a tabular method with a series of choices, not "coding"
- A set of choices must be made to arrive at an optimal solution
- As choices are made, subproblems of the same form arise frequently
- The key is to *store* the solutions of subproblems to be *reused* in the future

What is Dynamic Programming? ...

- Recall the divide-and-conquer approach
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine solutions of subproblems
- This contrasts with the dynamic programming approach
- Dynamic programming is applicable when *subproblems are not independent*
  - i.e., subproblems share subsubproblems
  - Solve every subsubproblem only once and store the answer for use when it reappears
- A divide-and-conquer approach will do more work than necessary

Elements of Dynamic Programming?

- Development of a dynamic programming solution to an optimization problem involves four steps

1. Characterize the structure of an optimal solution

   - Optimal substructures, where an optimal solution consists of sub-solutions that are optimal.

   - Overlapping sub-problems where the space of sub-problems is small in the sense that the algorithm solves the same sub-problems over and over rather than generating new sub-problems.

2. Recursively define the value of an optimal solution.

   - define the value of an optimal solution based on value of solutions to sub-problems.

3. Compute the value of an optimal solution in a bottom-up manner.

   - compute in a bottom-up fashion and save the values along the way

   - later steps use the save values of pervious steps

4. Construct an optimal solution from the computed optimal value

Matrix-chain Multiplication

- Suppose we have a sequence or chain A_1, A_2, …, A_n of n matrices to be multiplied
  - That is, we want to compute the product $A_1A_2 \ldots A_n$
- There are many possible ways (parenthesizations) to compute the product
- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:

1. $(A_1(A_2(A_3A_4)))$ 2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$ 4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication …

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices, matrix dimensions

Input: Matrices $A_{p \times q}$ and $B_{q \times r}$ (with dimensions $p \times q$ and $q \times r$)

Result: Matrix $C_{p \times r}$ resulting from the product $A \cdot B$

MATRIX-MULTIPLY($A_{p \times q}$, $B_{q \times r}$)

1. **for** $i \leftarrow 1$ **to** p
2.   **for** $j \leftarrow 1$ **to** r
3. $C[i, j] \leftarrow 0$
4.   **for** $k \leftarrow 1$ **to** q
5. $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6. **return** C

Scalar multiplication in line 5 dominates time to compute C
Number of scalar multiplications = pqr

Matrix-chain Multiplication …

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 =2{,}500$ scalar multiplications

    Total: 7,500
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 =50{,}000$ scalar multiplications

    Total: 75,000

Matrix-chain Multiplication …

- Matrix-chain multiplication problem
  - Given a chain A_1, A_2, …, A_n of n matrices, where for i=1, 2, …, n, matrix A_i has dimension $p_{i\text{-}1} \times p_i$
  - Parenthesize the product $A_1 A_2 \ldots A_n$ such that the total number of scalar multiplications is minimized

Matrix-chain Multiplication …

1. The structure of an optimal solution

- Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i A_{i+1} \dots A_j$
- An optimal parenthesization of the product $A_1A_2\dots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \le k < n$
- First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$
- **Key observation**: parenthesizations of the subchains $A_1A_2\dots A_k$ and $A_{k+1}A_{k+2}\dots A_n$ must also be optimal if the parenthesization of the chain $A_1A_2\dots A_n$ is optimal (why?)
- That is, the optimal solution to the problem contains within it the optimal solution to subproblems

Matrix-chain Multiplication …

2. Recursive definition of the value of an optimal solution
   - Let $m[i, j]$ be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
   - Minimum cost to compute $A_{1..n}$ is $m[1, n]$
   - Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} for some integer k where $i \le k < j$

Matrix-chain Multiplication …

- $A_{i..j} = (A_i A_{i+1} \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_j) = A_{i..k} \cdot A_{k+1..j}$
- Cost of computing $A_{i..j}$ = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$
- Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ is $p_{i-1} p_k p_j$

- $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
  $for\ i \le k < j$
- $m[i, i] = 0$ for i=1,2,…,n
- But… optimal parenthesization occurs at one value of k among all possible $i \le k < j$
- Check all these and select the best one

Matrix-chain Multiplication …

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

- To keep track of how to construct an optimal solution, we use a table s
- $s[i, j]$ = value of k at which $A_i A_{i+1} \dots A_j$ is split for optimal parenthesization
- Algorithm: next slide
  - First computes costs for chains of length $l=1$
  - Then for chains of length $l=2,3, \dots$ and so on
  - Computes the optimal cost bottom-up

Matrix-chain Multiplication …

3. Computing the optimal costs

Input: Array $p[0 \ldots n]$ containing matrix dimensions and n

Result: Minimum-cost table m and split table s

```
MATRIX-CHAIN-ORDER(p[ ], n)
    for i ← 1 to n
        m[i, i] ← 0
    for l ← 2 to n
        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
```

Takes $O(n^3)$ time

Requires $O(n^2)$ space

Matrix-chain Multiplication …

4. Constructing an optimal solution

```
Print-Optimal-Parens(s, i, j)
    {
      if  i = j
          then print "A_i" :
      else
        { print "(";
          Print-Optimal-Parens(s, i, s[i, j]);
          Print-Optimal-Parens(s, s[i, j]+1, j);
          print ")" ;
        }
    }
```

Matrix-chain Multiplication Example

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Assign
$p_0 = 30$
$p_1 = 35$
$p_2 = 15$
$p_3 = 5$
$p_4 = 10$
$p_5 = 20$
$p_6 = 25$

m[i,i]
m[1,1]=0
m[2,2]=0
m[3,3]=0
m[4,4]=0
m[5,5]=0
m[6,6]=0

Matrix-chain Multiplication Example …

$m[1,2]=m[1,1] + m[2,2] + p_0p_1p_2 = 0+0+30.35.15=15750$

$m[2,3]=m[2,2] + m[3,3] + p_1p_2p_3 = 0+0+35.15.5=2625$

$m[3,4]=m[3,3] + m[4,4] + p_2p_3p_4 = 0+0+15.5.10=750$

$m[4,5]=m[4,4] + m[5,5] + p_3p_4p_5 = 0+0+5.10.20=1000$

$m[5,6]=m[5,5] + m[6,6] + p_4p_5p_6 = 0+0+10.20.25=5000$

m	i						
j		1	2	3	4	5	6
	6					5000	0
	5				1000	0	
	4			750	0		
	3		2625	0			
	2	15750	0				
	1	0					

s	i (value of k)					
j		1	2	3	4	5
	6					5
	5				4	
	4			3		
	3		2			
	2	1				

Matrix-chain Multiplication Example …

$$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + p_0p_1p_3 = 7875 \\ m[1,2] + m[3,3] + p_0p_2p_3 = 18000 \end{cases}$$

$$m[2,4] = \min \begin{cases} ? \end{cases} \qquad m[3,5] = \min \begin{cases} ? \end{cases} \qquad m[4,6] = \min \begin{cases} ? \end{cases}$$

m		i					
j		1	2	3	4	5	6
	6				3500	5000	0
	5			2300	1000	0	
	4		4375	750	0		
	3	7875	2625	0			
	2	15750	0				
	1	0					

s		i (value of k)				
j		1	2	3	4	5
	6				5	5
	5			3	4	
	4		3	3		
	3	1	2			
	2	1				

Matrix-chain Multiplication Example …

$$m[1,4]=\min\begin{cases} m[1,1]+m[2,4]+p_0p_1p_4 = ? \\ m[1,2]+m[3,4]+p_0p_2p_4 = ? \\ m[1,3]+m[4,4]+p_0p_3p_4 = 9375 \end{cases}$$

$$m[2,5]=\min\begin{cases} ? \end{cases} \qquad m[3,6]=\min\begin{cases} ? \end{cases}$$

m	i						
j		1	2	3	4	5	6
	6			5375	3500	5000	0
	5		7125	2300	1000	0	
	4	9375	4375	750	0		
	3	7875	2625	0			
	2	15750	0				
	1	0					

s	i (value of k)					
j		1	2	3	4	5
	6			3	5	5
	5		3	3	4	
	4	3	3	3		
	3	1	2			
	2	1				

Matrix-chain Multiplication Example …

$$m[1,5]=\min\begin{cases} m[1,1] + m[2,5] + p_0p_1p_5 = ? \\ m[1,2] + m[3,5] + p_0p_2p_5 = ? \\ m[1,3] + m[4,5] + p_0p_3p_5 = 11875 \\ m[1,4] + m[5,5] + p_0p_4p_5 = ? \end{cases}$$

$$m[2,6]=\min\begin{cases} ? \end{cases}$$

m	i						
j		1	2	3	4	5	6
	6		10500	5375	3500	5000	0
	5	11875	7125	2300	1000	0	
	4	9375	4375	750	0		
	3	7875	2625	0			
	2	15750	0				
	1	0					

s	i (value of k)					
j		1	2	3	4	5
	6		3	3	5	5
	5	3	3	3	4	
	4	3	3	3		
	3	1	2			
	2	1				

Matrix-chain Multiplication Example …

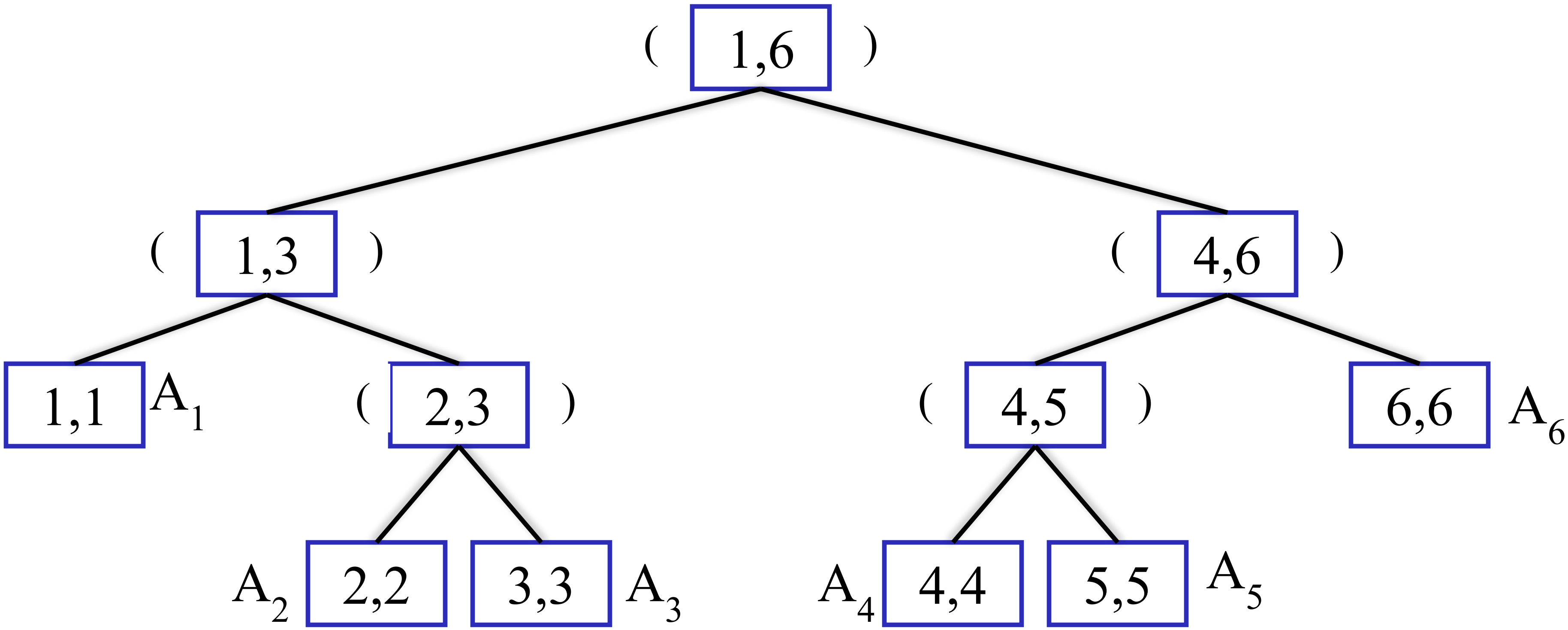
$$m[1,6] = \min \begin{cases} m[1,1] + m[2,6] + p_0p_1p_6 = ? \\ m[1,2] + m[3,6] + p_0p_2p_6 = ? \\ m[1,3] + m[4,6] + p_0p_3p_6 = 15125 \\ m[1,4] + m[5,6] + p_0p_4p_6 = ? \\ m[1,5] + m[6,6] + p_0p_5p_6 = ? \end{cases}$$

m	i						
j		1	2	3	4	5	6
	6	15125	10500	5375	3500	5000	0
	5	11875	7125	2500	1000	0	
	4	9375	4375	750	0		
	3	7875	2625	0			
	2	15750	0				
	1	0					

s	i (value of k)					
j		1	2	3	4	5
	6	3	3	3	5	5
	5	3	3	3	4	
	4	3	3	3		
	3	1	2			
	2	1				

Matrix-chain Multiplication Example …

- Constructing an optimal solution



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

Longest common subsequence (LCS)

The problem we shall consider is the longest-common-subsequence problem. A subsequence of a given sequence is just the given sequence with some elements (possibly none) left out. Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a **subsequence** of X if there exist 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 = <B, C, D, B> is a subsequence of X = <A, B, C, B, D, A, B> with corresponding index sequence <2, 3, 5, 7>

LCS…

Given two sequence X and Y, we say that a sequence Z is a **common subsequence** of X and Y if Z is a subsequence of both X and Y. For example, it X = <A, B, C, B, D, A, B> and Y = <B, D, C, A, B, A>, the sequence <B, C, A> is a common subsequence of both X and Y. The sequence <B, C, A> is not a longest common subsequence (LCS) of X and Y, however, since it has length 3 and the sequence <B, C, B, A>, which is also common to both X and Y, has length 4. The sequence <B, C, B, A> is an LCS of X and Y, as is the sequence <B, D, A, B>, since there is no common subsequence of length 5 or greater.

LCS…

In the longest-common-subsequence problem, we are given two sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum-length common subsequence of X and Y. Now we show that the LCS problem can be solved efficiently using dynamic programming.

Characterizing a LCS

A brute-force approach to solving the LCS problem is to enumerate all subsequence of X and check each subsequence to see if it is also a subsequence of Y, keeping track of the longest subsequence found. Each subsequence of Y corresponds to a subset of the indices {1, 2, …, m} of X. There are 2^m subsequences of X, so this approach requires exponential time, making it impractical for long sequences.

The LCS problem has an optimal-substructure property, however, the following theorem shows. As we shall see, the natural class of subproblems correspond to pairs of “prefixes” of the two input sequences. To be precise, given a sequence X = $<x_1, x_2, \ldots, x_m>$, we define the i^{th} **prefix** of X, for i = 0, 1, …, m, as $X_i = <x_1, x_2, \ldots x_i>$. For example, if X = <A, B, C, B, D, A, B>, then X_4 = <A, B, C, B> and X_0 is the empty sequence.

Characterizing a LCS…

Theorem 1 (Optimal substructure of an LCS)
Let X = <x_1, x_2, …,x_m>, and Y = <y_1, y_2, …, y_n> be sequences, and let Z = <z_1, z_2, …,z_k> 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 and 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 $z_k \neq y_m$ implies that Z is an LCS of X and Y_{n-1}.

Proof: (case 1: $x_m = y_n$)
Any sequence *Z'* that does not end in $x_m = y_n$ can be made longer by adding $x_m = y_n$ to the end. Therefore,
(1) longest common subsequence (LCS) *Z* must end in $x_m = y_n$.
(2) Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}, and
(3) there is no longer CS of X_{m-1} and Y_{n-1}, or *Z* would not be an LCS.

Characterizing a LCS…

Theorem 1 (Optimal substructure of an LCS)

Let X = <x_1, x_2, …,x_m>, and Y = <y_1, y_2, …, y_n> be sequences, and let Z = <z_1, z_2, …,z_k> 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 and 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 $z_k \neq y_m$ implies that Z is an LCS of X and Y_{n-1}.

Proof: (case 2: $x_m \neq y_n$, and $z_k \neq x_m$)

Since Z does not end in x_m,

(1) Z is a common subsequence of X_{m-1} and Y, and

(2) there is no longer CS of X_{m-1} and Y, or Z would not be an LCS.

Proof: (case 3: $x_m \neq y_n$, and $z_k \neq \mathrm{y}_m$)

Symmetric to (case 2)

A recursive solution to subproblems

The characterization of Theorem 1 shows that an LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal-substructure property. A recursive solution also has the overlapping-subproblems property, as we shall see in a moment.

Theorem 1 implies that there are either on or two subproblems to examine when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. If $x_m = y_n$ we must find and LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y. If $x_m \neq y_n$, then we must solve two subproblems: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}. Whichever of these two LCS's is longer is an LCS of X and Y.

A recursive solution to subproblems …

We can readily see the overlapping-subproblems property in the LCS problem. To find and LCS of X and Y, we may need to find the LCS's of X and Y_{n-1} and of X_{m-1} and Y. But each of these subproblems has the subsubproblem of finding the LCS of X_{m-1} and Y_{n-1}. Many other subproblems share subsubproblems.

A recursive solution to subproblems …

Like the matrix-chain multiplication problem, our recursive solution to the LCS problem involves establishing a recurrence of the cost of an optimal solution. Let us define c[i,j] to be the length of an LCS of the sequences X_i, and Y_j. If either i = 0 or j = 0, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula

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

Computing the length of an LCS

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

Procedure LCS-LENGTH takes two sequences $X = <x_1, x_2, \ldots, x_m>$ and $Y = <y_1, y_2, \ldots, y_n>$ as inputs. It stores the c[i,j] values in a table c[0..m,0..n] whose entries are computed in row-major order. (That is, the first row of c is filled in form left to right, then the second row, and so on.) It also maintains the table b[1.m,1..n] to simplify construction of an optimal solution. Intuitively, b[i,j] points to the table entry corresponding to the optimal subproblem solution chosen when computing b[i,j]. The procedure returns the b and c tables: c[m,n] contains the length of an LCS of X and Y.

Computing the length of an LCS ….

```
LCS-LENGTH (X, Y)
m ← length[X]
n ← length[Y]
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
```

$b[i, j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j.

$c[m,n]$ contains the length of an LCS of X and Y.

Computing the length of an LCS ….

	j	0	1	2	3	4	5	6
i		y_i	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

Figure 3.1 The c and b tables

Computing the length of an LCS ….

Figure 3.1 The c and b tables computed by LCS-LENGTH on the sequence X=<A, B, C, B, D, A, B> and Y=<B, D, C, A, B, A>. The square in row i and column j contains the value of c[i,j] and the appropriate arrow for the value of b[i,j]. The entry 4 in c[7,4]– the lower right-hand corner of the table—is the length of an LCS <B, C, B, A> of X and Y. For i,j > 0, entry c[i,j] depends only on whether $x_i = y_i$ and the values in entries c[i-1,j],c[i,j-1], and c[i-1,j-1], which are computed before c[i,j]. To reconstruct the elements of an LCS, follow the b[i,j] arrows from the lower right-hand corner; the path is shaded. Each "↖" on the path corresponds to an entry (highlighted) for which $x_i = y_i$ is a member of an LCS

Computing the length of an LCS ….

Figure 3.1 Shows the tables produced by LCS-LENGTH on the sequences X = <A, B, C, B, D, A, B> and Y= <B, D, C, A, B, A>. The running time of the procedure in O(mn), since each table entry O(1) time to compute.

Construction an LCS

The b table returned by LCS-LENGTH can be to quickly construct an LCS of $X = <x_1, x_2, \ldots, x_m>$ and $Y = <y_1, y_2, \ldots, y_n>$. We simply begin at b[m,n] and trace through the table following the arrows. Whenever we encounter a "↖" in entry b[i,j], it implies that $x_i = y_j$ is an element of the LCS. The elements of the LCS are encountered in reverse order by this method. The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial invocation is PRINT-LCS(b, X, length[x], lentgh[Y]).

Construction an LCS…..

```
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)
```

- Initial call is PRINT-LCS (b, X,m, n).
- When $b[i, j\,] =$, we have extended LCS by one character. So LCS = entries with in them.
- Time: $O(m+n)$

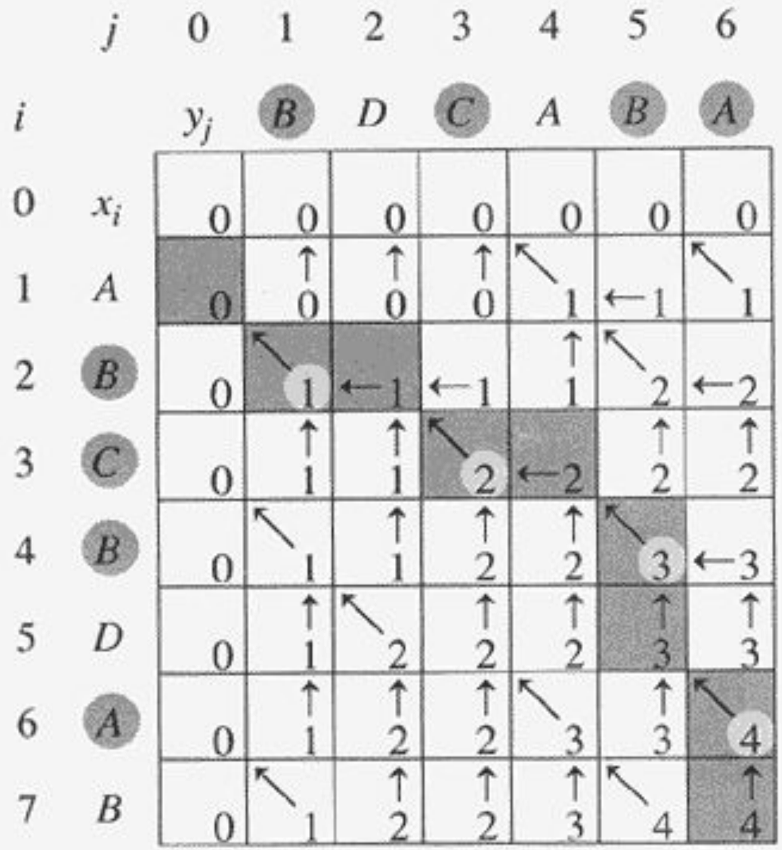


Figure 15.6 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the path is shaded. Each "$\nwarrow$" on the path corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS Example (0)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y	**B**	**D**	**C**	**A**	**B**
0	X	j					
1	i **A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB;   m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,4]

LCS Example (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y	**B**	**D**	**C**	**A**	**B**
0	X	j **0**	**0**	**0**	**0**	**0**	**0**
1	i **A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m   c[i,0] = 0
for j = 1 to n   c[0,j] = 0

LCS Example (2)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y	B	D	C	A	B
0	X	0	0	0	0	0	0
1	A	0	0				
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

*

LCS Example (3)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y	**B**	**D**	**C**	**A**	**B**
0	X		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**		
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (4)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y	**B**	**D**	**C**	**A**	**B**
0	X	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
  c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (5)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y	**B**	**D**	**C**	**A**	**B**
0	X		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (6)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y	**B**	**D**	**C**	**A**	**B**
0	X	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**				
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (7)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y	**B**	**D**	**C**	**A**	**B**
0	X	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

*

LCS Example (8)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y	**B**	**D**	**C**	**A**	**B**
0	X		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**					
4	**B**		**0**					

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (10)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y	**B**	**D**	**C**	**A**	**B**
0	X	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	1	1			
4	**B**	**0**					

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (11)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y	B	D	C	A	B
0	X		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2		
4	B		0					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (12)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y	**B**	**D**	**C**	**A**	**B**
0	X	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**					

if ($X_i == Y_j$)
   c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (13)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y	**B**	**D**	**C**	**A**	**B**
0	X		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**				

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (14)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y	B	D	C	A	B
0	X		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (15)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Y	**B**	**D**	**C**	**A**	**B**	
0	X	**0**	**0**	**0**	**0**	**0**	**0**	
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**	
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**	
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**	
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**	

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

*

Finding LCS

i	j	0	1	2	3	4	5
		Y	**B**	**D**	**C**	**A**	**B**
0	X	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

Finding LCS (2)

i	j	0	1	2	3	4	5
		Y	**B**	**D**	**C**	**A**	**B**
0	X	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

LCS (reversed order): **B C B**

LCS (straight order): **B C B**

(this string turned out to be a palindrome)