CSE331 Introduction to Algorithms
Lecture 12
Longest Common Subsequence

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

July 11, 2017

Introduction

- In this lecture, we study the *longest common subsequence (LCS)*.
- It gives another example of dynamic programming.

- **Reference**: Section 15.4 of the textbook (p. 390)
  Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.

Introduction

- Let $X = (A, B, C, B, D, A, B)$.
- We say that $Z = (B, D, A)$ is a *subsequence* of X.

Definition (subsequence)

A sequence $Z = (z_1, \ldots, z_k)$ is a subsequence of $X = (x_1, \ldots, x_m)$ if there is an increasing function φ such that $z_i = x_{\varphi_i}$ for all $i \in \{1, \ldots, k\}$.

- In the example above, $\varphi(1) = 2$, $\varphi(2) = 5$, $\varphi(3) = 6$,

$$z_1 = x_{\varphi(1)} = x_2 = B$$
$$z_2 = x_{\varphi(2)} = x_5 = D$$
$$z_3 = x_{\varphi(3)} = x_6 = A$$

- So the elements of the subsequence Z are taken from X, and appear in the same order.

Introduction

Definition (Common subsequence)

Given two sequences X and Y, we say that Z is a *common subsequence* of X and Y if Z is a subsequence of X and Y.

Example

$Z = (B, C, A)$ is a common subsequence of
$X = (A, B, C, B, D, A, B)$ and
$Y = (B, D, C, A, B, A)$

- In the example above, there is a *longer* common subsequence: (B, D, A, B).

Problem Statement

Problem (Longest common subsequence)

Given two sequences $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$, the longest common subsequence problem is to find a common subsequence Z of X and Y with maximum length. We say that Z is a longest common subsequence (LCS) of X and Y.

- Motivation: Measuring how similar two DNA strands are.
- Example: Two given strands
  $S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA$
  $S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA$
- Their LCS is
  $S_3 = GTCGTCGGAAGCCGGCCGAA$.
- The longer the strand S_3 is, the more similar S_1 and S_2 are.

Brute-Force Approach

Brute-Force Approach

For each subsequence of X, check whether it is a subsequence of Y.
Return the longest such subsequence of X and Y.

- What is the running time?
- There are 2^m subsequences of X, so the running time is $\Omega(2^m)$.
- This is exponential.
- It is too slow for DNA sequences, for instance.
- Therefore, we will try to use dynamic programming.

Structure of the Solution

Definition (prefix)

The ith *prefix* of a sequence $X = (x_1, \ldots, x_m)$, $m \geqslant i$ is the subsequence $X_i = (x_1, \ldots, x_i)$.

- In order to solve the problem by dynamic programming, we need a better understanding of the structure of the optimal solutions.

Theorem (Optimal substructure of an LCS)

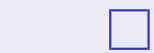
Let $Z_k = (z_1, \ldots, z_k)$ be an LCS of two sequences $X_m = (x_1, \ldots, x_m)$ and $Y_n = (y_1, \ldots, y_n)$.

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 is an LCS of X_m and Y_{n-1}, or an LCS of X_{m-1} and Y_n.*

Structure of the Solution

Proof.

1. (Case where $x_m = y_n$.) If $z_k \neq x_m$, then we can append x_m to Z_k and obtain a longer subsequence of X_m and Y_n. It contradicts the optimality of Z_k.
   Now suppose that $z_k = x_m$. The prefix Z_{k-1} is a subsequence of X_{m-1} and Y_{n-1}. If it were not an LCS of X_{m-1} and Y_{n-1}, then there would be a common subsequence Z' of X_{m-1} and Y_{n-1} with length more than $k-1$. After appending x_m to Z', we obtain a common subsequence of X_m and Y_n which is longer than Z_k, a contradiction.
2. As $x_m \neq y_n$, we must have $z_k \neq x_m$ or $z_k \neq y_n$. Without loss of generality, we assume that $z_k \neq x_m$. Therefore Z_k is a subsequence of X_{m-1}. We also know that Z_k is a subsequence of Y_n. Then Z_k must be an LCS of X_{m-1} and Y_n, as otherwise, there would be a longer subsequence of X_m and Y_n, contradicting the assumption that Z_k is an LCS of X_m and Y_n.



Structure of the Solution

- The theorem above is an *optimal substructure* property.
- We usually need this type of result in order to use dynamic programming.
- It gives a connection between the original problem and the subproblems whose solutions are recorded by the algorithm.

Computing the Length of an LCS

- Let $c(i, j)$ denote the length of an LCS of X_i and Y_j.
- The following recurrence relation follows from the theorem on Slide 8:

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

- This formula allows to compute the length of an LCS recursively. (See next slide.)

Computing the Length of an LCS: Naive Approach

Pseudocode

```
1: procedure NaiveLCSLength(X, Y, i, j)
2:     if i = 0 or j = 0 then
3:         return 0
4:     if X[i] = Y[j] then
5:         return 1+NaiveLCSLength(X, Y, i − 1, j − 1)
6:     return max(NaiveLCSLength(X, Y, i − 1, j),
7:                                NaiveLCSLength(X, Y, i, j − 1))
```

Computing the Length of an LCS

- The algorithm above runs in exponential time. (Why?)
- So we will use dynamic programming instead.
- More precisely, we will use the bottom-up method.

Computing the Length of an LCS

Pseudocode

```
1:  procedure LCSLength(X[1...m], Y[1...n])
2:      c[0...m, 0...n] ← new array
3:      for i = 0, m do
4:          c[i, 0] ← 0
5:      for j = 1, n do
6:          c[0, j] ← 0
7:      for i = 1, m do
8:          for j = 1, n do
9:              if X[i] = Y[j] then
10:                 c[i, j] ← c[i − 1, j − 1] + 1
11:             else
12:                 c[i, j] ← max(c[i − 1, j], c[i, j − 1])
13:     return c[m, n]
```

Computing the Length of an LCS

- Correctness of this algorithm follows from Equation (1), and the fact that at the time we compute $c[i, j]$, the values of the subproblems $c[i-1, j-1]$, $c[i-1, j]$ and $c[i, j-1]$ have already been computed.
- As usual with the bottom-up approach, we take advantage of an ordering of the solution. (Here we use the lexicographic order.)
- Analysis: This algorithm runs in $\Theta(mn)$ time due to the doubly-nested loops.
- This procedure only computes the length of an LCS. How do we recover an optimal subsequence $Z = (z_1, \ldots, z_k)$?

Computing an LCS

	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							

Table: $c[1 \ldots 7, 1 \ldots 6]$

- LCSLength computes the whole table $c[1 \ldots m, 1 \ldots n]$ of the LCSLengths of (X_i, Y_j) for all $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- LCSLength computes the whole table $c[1 \ldots m, 1 \ldots n]$ of the LCSLengths of (X_i, Y_j) for all $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$.
- We can use this information to construct an LCS *backward*.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- $x_7 = B$ and $y_6 = A$, so $x_7 \neq y_6$.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- $x_7 = B$ and $y_6 = A$, so $x_7 \neq y_6$.
- So an LCS of (X_7, Y_6) is an LCS of (X_6, Y_6) or (X_7, Y_5) by the Theorem on Slide 8.
- As $c[6, 6] = c[7, 5] = 4$, we can recurse on either side.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- We recurse on X_6, Y_6.
- $x_6 = y_6 = A$.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- We recurse on X_6, Y_6.
- $x_6 = y_6 = A$.
- So an LCS of (X_6, Y_6) is obtained by appending A to an LCS of (X_5, Y_5) by the Theorem on Slide 8.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- We recurse on X_5, Y_5.
- $x_5 = B$ and $y_5 = A$, so $x_5 \neq y_5$.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- We recurse on X_5, Y_5.
- $x_5 = B$ and $y_5 = A$, so $x_5 \neq y_5$.
- So an LCS of (X_5, Y_5) is an LCS of (X_4, Y_5) or (X_5, Y_4).
- As $c[5, 4] = 2$ and $c[4, 5] = 3$, it is an LCS of (X_4, Y_5)

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- We recurse on X_4, Y_5.
- $x_4 = B$ and $y_5 = B$, so $x_4 = y_5$.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- We recurse on X_3, Y_4.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- We recurse on X_3, Y_3.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- We recurse on X_2, Y_2.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- We recurse on X_2, Y_1.

Computing an LCS

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

Table: $c[1 \ldots 7, 1 \ldots 6]$

- We recurse on X_1, Y_0.
- End of recursion.
- BCBA is an LCS.
- Remark: It is not the only LCS. BDAB is another LCS.

Computing an LCS

- The procedure below prints an LCS in forward order, given the array $c[1 \ldots m, 1 \ldots n]$ from LCSLength.

Pseudocode

```
1:  procedure PrintLCS(c, X, Y, i, j)
2:      if i = 0 or j = 0 then
3:          return
4:      if X[i] = Y[j] then
5:          PrintLCS(c, X, Y, i − 1, j − 1)
6:          Print X[i]
7:      else if c[i − 1, j] = c[i, j] then
8:          PrintLCS(c, X, Y, i − 1, j)
9:      else
10:         PrintLCS(c, X, Y, i, j − 1)
```

Computing an LCS

- What is the running time of PrintLCS?
- It is $O(m+n)$, because at each recursive call, j or i is decremented, sot there are at most $m+n$ recursive calls.
- PrintLCS only prints one LCS. What would happen if we tried to print all the LCS?
- There are exponentially many LCS (why?), so it would take exponential time in the worst case.

Concluding Remarks

- Once again dynamic programming help us bring down the running time from exponential to polynomial.
- We could also easily reconstruct an optimal solution, given the optimal value (i.e. we could reconstruct an LCS after computing its length).
- In particular, computing the optimal length took quadratic time $\Theta(mn)$, but using the table computed by this procedure, we could print an LCS in linear time $O(m + n)$.
- We used the bottom-up method. It is also possible to use the top-down memoized approach, but the code is longer, and should be slower by a constant factor.