CSE221 Data Structures
Lecture 23
String Algorithms I

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

December 6, 2021

Introduction

- Assignment 4 due on Friday.
- I will not check memory leaks, so you don't need to implement a destructor.
- In Dijkstra's algorithm implementation, you may assume that $D[u]$ is always less than some large number such as 10^{10}.
- Final exam is on Wednesday 15 December, 20:00–22:00.
- This is a first lecture on algorithms for *strings*.
- We will also present another application of *dynamic programming*, which was already mentioned assignment 3.
- Reference for this lecture: Textbook Chapter 13.

Strings

Two strings

```
P = "CGTAAACTGCTTTAATCAAACGC"
S = "This is a string."
```

- A *string* is a sequence of characters taken from an alphabet Σ.

Examples

An English word is a string of characters in $\Sigma = \{a, b, c, \dots, z\}$.
For DNA sequences, the alphabet is $\Sigma = \{C, G, T, A\}$.

- We will assume that the alphabet is fixed, and thus $|\Sigma| = O(1)$.
- The *null string* is the string of length 0, which we may denote λ. (Often called *empty string*.)

Strings

	0	1	2	3	4	5	6	7	8	9	10	11
P	C	A	T	T	G	G	A	C	C	T	A	C

C	A	T	T	G	G	A	C	C	T	A	C

a substring of P

a prefix of P	C	A	T	T	G	G	A	C	C	T	A	C

C	A	T	T	G	G	A	C	C	T	A	C	a suffix of P

Strings

- We denote by $P = P[0]P[1] \dots P[m-1]$ a string of *length* m.
- We denote by $P[i \dots j]$ the *substring* $P[i]P[i+1] \dots P[j]$ whenever $0 \leqslant i \leqslant j < m$.
- $P[0 \dots i]$ is called a *prefix* of P and $P[j \dots m-1]$ is a *suffix*.
- When $i > j$, then $P[i \dots j]$ is the null string λ.
- A *proper substring* of P is a substring of P that is not equal to P. In other words, it means that $i > 0$ or $j < m - 1$.

STL Strings

Operation	Output
S.size()	16
S.at(5)	'f'
S[5]	'f'
S + "qrs"	"abcdefghijklmnopqrs"
S == "abcdefghijklmnop"	true
S.find("ghi")	6
S.substr(4, 6)	"efghij"
S.erase(4, 6)	"abcdklmnop"
S.insert(1, "xxx")	"axxxbcdklmnop"
S += "xy"	"axxxbcdklmnopxy"
S.append("z")	"axxxbcdklmnopxyz"

- Operations performed on the STL string "abcdefghijklmnop"

Introduction

- Let $X = CABCBDAB$.
- We say that $Z = BDA$ is a *subsequence* of Z.

Definition (subsequence)

A string $Z = z_0 z_1 \dots z_k$) is a subsequence of $X = x_0 x_1 \dots x_{m-1}$ if there is an increasing function φ such that $z_i = x_{\varphi_i}$ for all $i \in \{0, \dots, k\}$.

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

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

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

Introduction

Definition (Common subsequence)

Given two strings 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

$Z = BCA$ is a common subsequence of
$X = ABCBDAB$ and $Y = BDCABA$

- In the example above, there is a *longer* common subsequence: $BDAB$.

Problem Statement

Problem (Longest common subsequence)

Given two strings $X = x_0 \dots x_{m-1}$ and $Y = y_0 \dots y_{n-1}$, 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.
- So we will use a different approach: *dynamic programming*.

Structure of the Solution

- We denote by X_i and Y_j the prefixes of X and Y of lengths $i+1$ and $j+1$, respectively. So $X_i = x_0 x_1 \dots x_i$ and $Y_j = y_0 \dots y_j$.
- 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 $Z_k = z_0 \dots z_k$ be an LCS of two sequences $X_i = x_0, \dots, x_i$ and $Y_j = y_0, \dots, y_j$.

1. *If $x_i = y_j$, then $z_k = x_i = y_j$ and Z_{k-1} is an LCS of X_{i-1} and Y_{j-1}.*
2. *If $x_i \neq y_j$, then Z_k is an LCS of X_i and Y_{j-1}, or an LCS of X_{i-1} and Y_j.*

Structure of the Solution

Proof.

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

Recurrence Relation

- Let $L[i,j]$ denote the length of an LCS of X_i and Y_j.
- The following recurrence relation follows from the theorem on Slide 12:

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

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

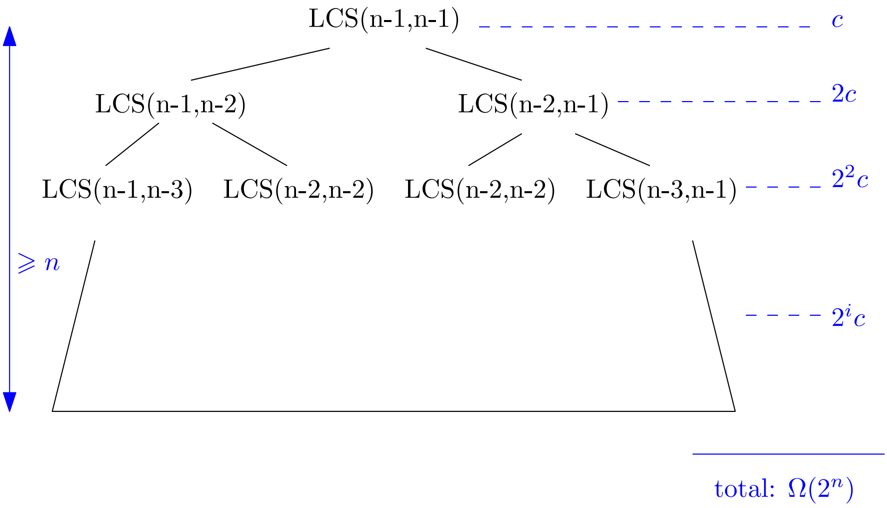
Computing the Length of an LCS

Naive Approach

```
procedure LCS(X, Y, i, j)
    if i = −1 or j = −1 then                ▷ empty substring
        return 0
    if x_i = y_j then
        return 1+LCS(X, Y, i − 1, j − 1)
    return max(LCS(X, Y, i − 1, j), LCS(X, Y, i, j − 1))
```

- The algorithm above runs in exponential time. (See next slide)
- So we will use a different approach, called *dynamic programming*.

Analysis by the Recursion Tree Method



Computing the Length of an LCS

Dynamic Programming Approach

procedure $\text{LCSLength}(X, Y)$
    $L[-1 \ldots m, -1 \ldots n] \leftarrow$ new array
    for $i \leftarrow -1, m-1$ **do**
        $L[i, -1] \leftarrow 0$
    for $j \leftarrow -1, n-1$ **do**
        $L[-1, j] \leftarrow 0$
    for $i \leftarrow 0, m-1$ **do**
        for $j \leftarrow 0, n-1$ **do**
            if $x_i = y_j$ **then**
                $L[i,j] \leftarrow L[i-1, j-1] + 1$
            else
                $L[i,j] \leftarrow \max(L[i-1,j], L[i,j-1])$
    return $L[m-1, n-1]$

Computing the Length of an LCS

- Correctness of this algorithm follows from Equation (1), and the fact that at the time we compute $L[i, j]$, the values of the subproblems $L[i-1, j-1]$, $L[i-1, j]$ and $L[i, j-1]$ have already been computed.
- Analysis: This algorithm runs in $\Theta(mn)$ time due to the doubly-nested loops.
- This is called dynamic programming because we record solutions of subproblems, to avoid recomputing them during the course of the algorithm. (See *CSE331: Introduction to Algorithms*.)
- This procedure only computes the length of an LCS. How do we recover an optimal subsequence $Z = z_1 \dots z_k$?

Computing an LCS

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

Table: $L[-1 \ldots 6, -1 \ldots 5]$

- LCSLength computes the whole table $L[-1 \ldots m-1, -1 \ldots m-1]$ of the LCS lengths of (X_i, Y_j) for all $0 \leqslant i \leqslant m-1$ and $0 \leqslant j \leqslant n-1$.

Computing an LCS

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

Table: $L[-1 \ldots 6, -1 \ldots 5]$

- LCSLength computes the whole table $L[-1 \ldots m-1, -1 \ldots m-1]$ of the LCSLengths of (X_i, Y_j) for all $0 \leqslant i \leqslant m-1$ and $0 \leqslant j \leqslant n-1$.
- We can use this information to construct an LCS *backward*.

Computing an LCS

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

Table: $L[-1\dots 6, -1\dots 5]$

- $x_6 = B$ and $y_5 = A$, so $x_6 \neq y_5$.

Computing an LCS

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

Table: $L[-1\dots6, -1\dots5]$

- $x_6 = B$ and $y_5 = A$, so $x_6 \neq y_5$.
- So an LCS of (X_6, Y_5) is an LCS of (X_5, Y_5) or (X_6, Y_4) by the Theorem on Slide 12.
- As $L[5,5] = L[6,4] = 4$, we can recurse on either side.

Computing an LCS

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

Table: $L[-1\dots 6, -1\dots 5]$

- We recurse on X_5, Y_5.
- $x_5 = y_5 = A$.

Computing an LCS

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

Table: $L[-1\ldots6, -1\ldots5]$

- We recurse on X_5, Y_5.
- $x_5 = y_5 = A$.
- So an LCS of (X_5, Y_5) is obtained by appending A to an LCS of (X_4, Y_4) by the Theorem on Slide 12.

Computing an LCS

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

Table: $L[-1\dots 6, -1\dots 5]$

- We recurse on X_4, Y_4.
- $x_4 = B$ and $y_4 = A$, so $x_4 \neq y_4$.

Computing an LCS

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

Table: $L[-1\dots 6, -1\dots 5]$

- We recurse on X_4, Y_4.
- $x_4 = D$ and $y_4 = B$, so $x_4 \neq y_4$.
- So an LCS of (X_4, Y_4) is an LCS of (X_3, Y_4) or (X_4, Y_3).
- As $L[4,3] = 2$ and $L[3,4] = 3$, it is an LCS of (X_3, Y_4)

Computing an LCS

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

Table: $L[-1\ldots 6, -1\ldots 5]$

- We recurse on X_3, Y_4.
- $x_3 = B$ and $y_4 = B$, so $x_3 = y_4$.

Computing an LCS

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

Table: $L[-1\ldots 6, -1\ldots 5]$

- We recurse on X_2, Y_3.

Computing an LCS

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

Table: $L[-1\ldots 6, -1\ldots 5]$

- We recurse on X_2, Y_2.

Computing an LCS

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

Table: $L[-1 \dots 6, -1 \dots 5]$

- We recurse on X_1, Y_1.

Computing an LCS

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

Table: $L[-1\ldots 6, -1\ldots 5]$

- We recurse on X_1, Y_0.

Computing an LCS

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

Table: $L[-1 \ldots 6, -1 \ldots 5]$

- We recurse on X_0, Y_{-1}.
- 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 $L[-1 \dots m-1, -1 \dots n-1]$ from LCSLENGTH.

Pseudocode

```
procedure PRINTLCS(L, X, Y, i, j)
    if i = −1 or j = −1 then
        return
    if x_i = y_j then
        PRINTLCS(L, X, Y, i − 1, j − 1)
        Print x_i
    else if L[i − 1, j] = L[i, j] then
        PRINTLCS(L, X, Y, i − 1, j)
    else
        PRINTLCS(L, 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?
- In the worst case there are exponentially many LCS, so it would take exponential time.

Concluding Remarks

- Dynamic programming helped us bring down the running time from exponential to polynomial.
- We were able to 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 $O(m+n)$.