

Dynamic programming

Longest Common Subsequence

Longest Common Subsequence

- A **subsequence** of a sequence is the same sequence with 0 or more elements left out (deleted)
- **Substring** is different from subsequence, substring is consecutive string.
- $X = \{A\ G\ G\ G\ C\ T\}$
Subsequences of $X = A\ C\ ,\ G\ G\ G\ ,\ G\ C\ T\ ,\ G\ T\ ,\ \dots$
 $G\ T$ is subsequence of X but it is not substring of X

Longest Common Subsequence

- **Common Subsequence:** A common subsequence of 2 DNA sequences is a subsequence present in both sequences

X = A G C G T A G

Y = G T C A G A

Common subsequences of X and Y = GT, GTA, G A, A G, G C A,

- **Longest Common subsequence** is the longest sequence among common subsequences.

X = A **G** **C** **G** T **A** G

Y = **G** T **C** A **G** **A**

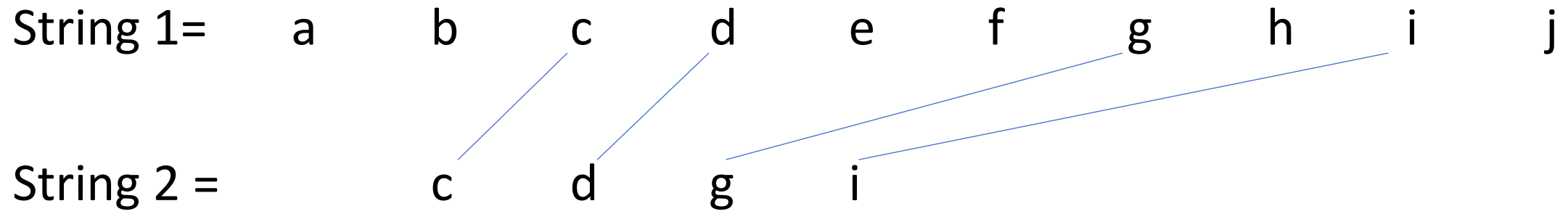
LCS = GCGA

LCS – example 1

String 1= a b c d e f g h i j

String 2 = c d g i

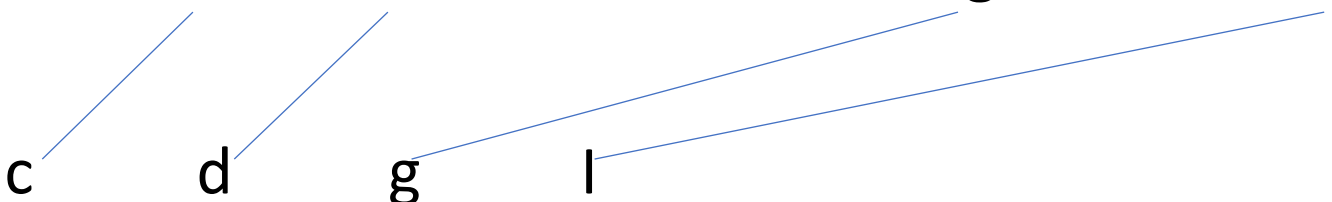
LCS



LCS

String 1= a b c d e f g h i j

String 2 = c d g l



LCS = cdgi

$|LCS| = 4$

We are not looking for exact match. String 2 is present in the string 1. The characters are not together but are in same order as they are in string 2.

LCS – example 2

String 1= a b c d e f g h i j

String 2 = e c d g i

LCS

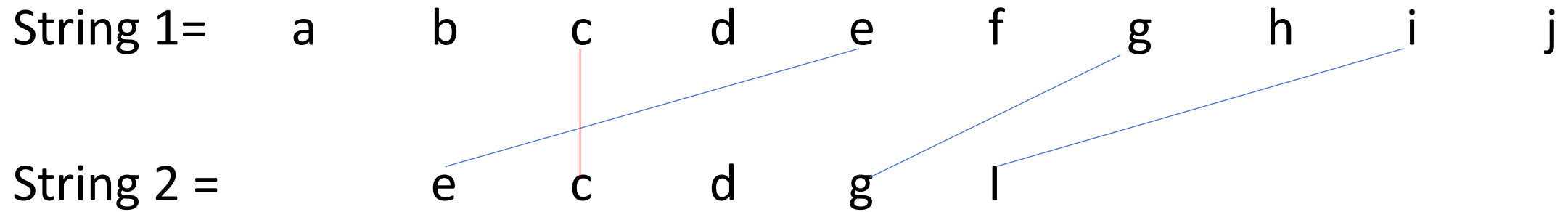
String 1= a b c d e f g h i j

String 2 = e c d g l

The diagram illustrates the Longest Common Subsequence (LCS) problem. It shows two strings: String 1 = "a b c d e f g h i j" and String 2 = "e c d g l". Blue lines connect the characters 'e', 'g', and 'i' in String 1 to 'e', 'g', and 'l' in String 2 respectively, illustrating a common subsequence [egi].

[egi] is a common subsequence. But its not the longest common subsequence.

LCS

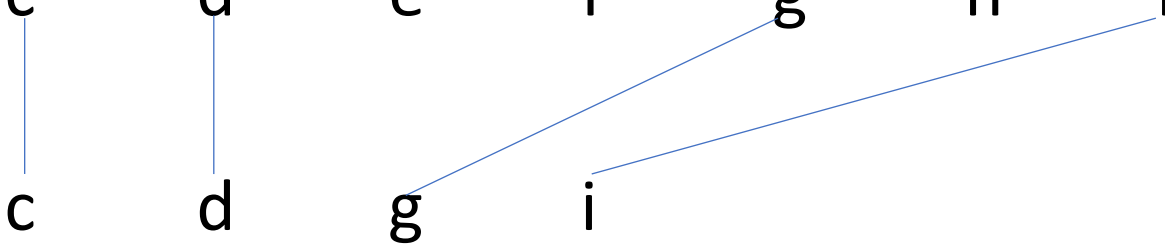


This intersection (red line) is not allowed. Characters should be in same order in string 1 as they are in string 2.

LCS

String 1= a b c d e f g h i j

String 2 = e c d g i



LCS = [cdgi]

LCS – example 3

String 1= a b d a c e

String 2 = b a b c e

LCS – example 3

String 1=

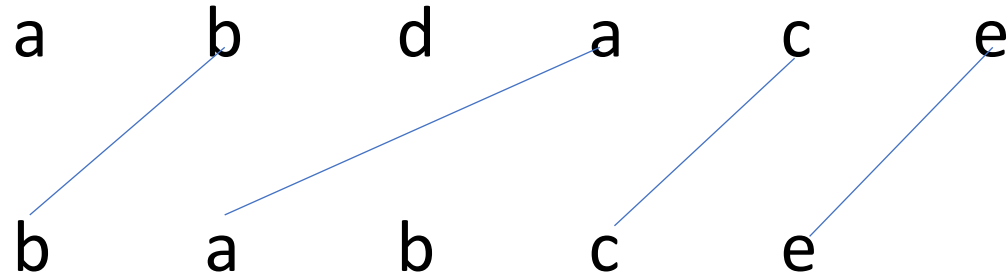
a b d a c e

String 2 =

b a b c e

LCS = [bace]

|LCS| = 4



LCS – example 3

String 1=

a b d a c e

String 2 =

b a b c e

LCS = [abce]

|LCS| = 4

LCS – Brute Force Algorithm

- Brute force algorithm would compute all subsequences of both sequences and find the common and print the longest.

OR

- Compute all subsequences of one sequence and check if it is also present in the other sequence. Print the longest common sequence.

LCS – Brute Force Algorithm

- How many subsequences are there in a sequence of n elements?
- Think about the definition of a subsequence
- A subsequence is same sequence with 0 or more elements left out.
- For each of the n elements, we have an option, delete it or keep it.
- 2 possibilities for each of the n elements so total subsequences =
- $2 * 2 * 2 \dots * 2 = 2^n$

LCS – Brute Force Algorithm

- if $|X| = m$, $|Y| = n$, then there are 2^m subsequences of x ; we must compare each with Y (n comparisons)

- So the running time of the brute-force algorithm is $O(n 2^m)$

The brute force algorithm will take exponential time since computing all subsequences of any one sequence will take exponential time.

Optimal Substructure in LCS

- LCS problem has *optimal substructure*: optimal solutions of sub-problems are parts of the final solution.
 - Sub-problems: “find LCS of pairs of *prefixes* of X and Y ”
 - If $X = \langle x_1, \dots, x_m \rangle$ and if $Y = \langle y_1, \dots, y_n \rangle$ are sequences, let $Z = \langle z_1, \dots, z_k \rangle$ be some LCS of x and y .
1. If $x_m = y_n$ then $z_k = x_m$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
 2. If $x_m \neq y_n$ and $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y
 3. If $x_m \neq y_n$ and $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}

Optimal Substructure in LCS

1. If $x_m = y_n$ then $z_k = x_m$

$X = \langle x_1, x_2, \dots, x_{m-2}, x_{m-1}, x_m \rangle$

$Y = \langle y_1, y_2, \dots, y_{n-2}, y_{n-1}, y_n \rangle$

$X = G C G T A G$

$Y = G T T C A G A G$

$Z = G C G A G$

Optimal Substructure in LCS

1. If $x_m = y_n$ then $z_k = x_m$

$X = \langle x_1, x_2, \dots, x_{m-2}, x_{m-1}, x_m \rangle$

$Y = \langle y_1, y_2, \dots, y_{n-2}, y_{n-1}, y_n \rangle$

Proof by Contradiction:

If $z_k \neq x_m$ then we could add $x_m = y_n$ to Z to get an LCS of length $k + 1$.
By contradiction it must be that $z_k = x_m = y_n$.

Optimal Substructure in LCS

1. If $x_m = y_n$ then $z_k = x_m$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}

$X = \langle x_1, x_2, \dots, x_{m-2}, x_{m-1}, x_m \rangle$

$Y = \langle y_1, y_2, \dots, y_{n-2}, y_{n-1}, y_n \rangle$

Proof by Contradiction:

If $z_k \neq x_m$ then we could add $x_m = y_n$ to Z to get an LCS of length $k + 1$.

By contradiction it must be that $z_k = x_m = y_n$.

$|Z_{k-1}| = k - 1$ and it is an LCS of X_{m-1} and Y_{n-1} .

It is an LCS, if not then suppose W is LCS of X_{m-1} and Y_{n-1} with $|W| > k - 1$ and so by appending $x_m = y_n$ to W we get a LCS of X and Y of length greater than k , a contradiction.

Optimal Substructure in LCS

2. If $x_m \neq y_n$ and $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y

$X = \langle x_1, x_2, \dots, x_{m-2}, x_{m-1}, x_m \rangle$

$Y = \langle y_1, y_2, \dots, y_{n-2}, y_{n-1}, y_n \rangle$

Proof:

If $z_k \neq x_m$ then Z is a LCS of X_{m-1} and Y .

If Z is not LCS then suppose W is LCS with of X_{m-1} and Y and $|W| > k$, then W would also be LCS of X and Y , a contradiction.

Optimal Substructure in LCS

3. If $x_m \neq y_n$ and $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}

$X = \langle x_1, x_2, \dots, x_{m-2}, x_{m-1}, x_m \rangle$

$Y = \langle y_1, y_2, \dots, y_{n-2}, y_{n-1}, y_n \rangle$

Proof:

Same as proof of 2

LCS – DP Recursive Formula

- Define X_i , Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i & Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

LCS – DP Recursive Formula

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 & Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j :
 $c[0, j] = c[i, 0] = 0$

LCS – DP Recursive Formula

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- When we calculate $c[i, j]$, we consider two cases:
- **First case:** $x[i] = y[j]$: one more symbol in strings X and Y matches, so the length of LCS X_i & Y_j equals to the length of LCS of smaller strings X_{i-1} & Y_{j-1} , plus 1

LCS – DP Recursive Formula

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- **Second case:** $x[i] \neq y[j]$
- As symbols don't match, our solution is not improved, and the length of $\text{LCS}(X_i, Y_j)$ is the same as before (i.e. maximum of $\text{LCS}(X_i, Y_{j-1})$ and $\text{LCS}(X_{i-1}, Y_j)$)

LCS – DP Algorithm

LCS-Length(X, Y)

1. $m = \text{length}(X)$ // get the # of symbols in X
2. $n = \text{length}(Y)$ // get the # of symbols in Y
3. for $i = 1$ to m $c[i,0] = 0$ // special case: Y_0
4. for $j = 1$ to n $c[0,j] = 0$ // special case: X_0
5. for $i = 1$ to m // for all X_i
6. for $j = 1$ to n // for all Y_j
7. if ($X_i == Y_j$)
8. $c[i,j] = c[i-1,j-1] + 1$
9. else $c[i,j] = \max(c[i-1,j], c[i,j-1])$
10. return c

LCS using DP

- $O(m*n)$.

LCS using DP

A = ¹ ²
 b d

B = ¹ ² ³ ⁴
 a b c d

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

	String B j		a	b	c	d
String A i		0	1	2	3	4
	0	0	0	0	0	0
b	1	0				
d	2	0				

LCS using DP

A = **1** **2**
 b d

B = **1** **2** **3** **4**
 a b c d

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

	String B		a	b	c	d
String A	Indices i, j	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0			
d	2	0				

LCS using DP

A = **1** **2**
 b d

B = **1** **2** **3** **4**
 a b c d

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

	String B		a	b	c	d
String A	indices	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1		
d	2	0				

LCS using DP

A = **1** **2**
 b d

B = **1** **2** **3** **4**
 a b c d

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

	String B		a	b	c	d
String A	indices	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1	1	
d	2	0				

LCS using DP

A = **1** **2**
 b d

B = **1** **2** **3** **4**
 a b c d

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

	String B		a	b	c	d
String A	indices	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1	1	1
d	2	0				

LCS using DP

A = **1** **2**
 b d

B = **1** **2** **3** **4**
 a b c d

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

	String B		a	b	c	d
String A	indices	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1	1	1
d	2	0	0			

LCS using DP

A = **1** **2**
 b d

B = **1** **2** **3** **4**
 a b c d

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

	String B		a	b	c	d
String A	indices	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1	1	1
d	2	0	0	1		

LCS using DP

A = **1** **2**
 b d

B = **1** **2** **3** **4**
 a b c d

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

	String B		a	b	c	d
String A	indices	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1	1	1
d	2	0	0	1	1	

LCS using DP

A = **1** **2**
 b d

B = **1** **2** **3** **4**
 a b c d

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

	String B		a	b	c	d
String A	indices	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1	1	1
d	2	0	0	1	1	2

LCS using DP - Backtracking

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

See where a particular entry is coming from?

Either from the previous diagonal or previous row or column

Whenever an entry is filled from previous diagonal, that character is part of LCS

	String B		a	b	c	d
String A	indices	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1	1	1
d	2	0	0	1	1	2

d

LCS using DP - Backtracking

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

See where a particular entry is coming from?

Either from the previous diagonal or previous row or column

Whenever an entry is filled from previous diagonal, that character is part of LCS

	String B		a	b	c	d
String A	indices	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1	1	1
d	2	0	0	1	1	2

d

LCS using DP - Backtracking

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

See where a particular entry is coming from?

Either from the previous diagonal or previous row or column

Whenever an entry is filled from previous diagonal, that character is part of LCS

	String B		a	b	c	d
String A	indices	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1	1	1
d	2	0	0	1	1	2

b

d

LCS using DP - Backtracking

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

See where a particular entry is coming from?

Either from the previous diagonal or previous row or column

Whenever an entry is filled from previous diagonal, that character is part of LCS

	String B		a	b	c	d
String A	indices	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1	1	1
d	2	0	0	1	1	2

b

d

LCS = [bd]
|LCS| = 2

Complexity = $O(m*n)$

Exercise – Optimal Solution

Modify the algorithm to get the optimal solution

Exercise - dry run the DP algorithm

		T	U	E	S	D	A	Y
		0	0	0	0	0	0	0
S	0							
A	0							
T	0							
U	0							
R	0							
D	0							
A	0							
Y	0							

Slide Credits

- COMP 3711H Design and Analysis of Algorithms
Fall 2014