

CH-231-A

Algorithms and Data Structures

ADS

Lecture 31

Dr. Kinga Lipskoch

Spring 2020

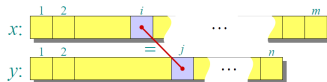
Strategy

- ▶ Look at length of longest-common subsequence.
- ▶ Let $|s|$ denote the length of a sequence s .
- ▶ To find $LCS(x, y)$, consider **prefixes** of x and y (i.e., we go from right to left)
- ▶ **Definition**: $c[i, j] = |LCS(x[1..i], y[1..j])|$.
In particular, $c[m, n] = |LCS(x, y)|$.
- ▶ **Theorem** (recursive formulation):

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

Proof (1)

Case $x[i] = y[j]$:



Let $z[1..k] = LCS(x[1..i], y[1..j])$ with $c[i, j] = k$.

Then, $z[k] = x[i] = y[j]$ (else z could be extended).

Thus, $z[1..k-1]$ is CS of $x[1..i-1]$ and $y[1..j-1]$.

Claim: $z[1..k-1] = LCS(x[1..i-1], y[1..j-1])$.

- ▶ Assume w is a longer CS of $x[1..i-1]$ and $y[1..j-1]$, i.e., $|w| > k-1$.
- ▶ Then the concatenation $w + z[k]$ is a CS of $x[1..i]$ and $y[1..j]$ with length $> k$.
- ▶ This contradicts $|LCS(x[1..i], y[1..j])| = k$.
- ▶ Hence, the assumption was wrong and the claim is proven.

Hence, $c[i-1, j-1] = k-1$, i.e., $c[i, j] = c[i-1, j-1] + 1$.

Proof (2)

Case $x[i] \neq y[j]$:

Then, $z[k] \neq x[i]$ or $z[k] \neq y[j]$.

▶ $z[k] \neq x[i]$:

Then, $z[1..k] = \text{LCS}(x[1..i-1], y[1..j])$.

Thus, $c[i-1, j] = k = c[i, j]$.

▶ $z[k] \neq y[j]$:

Then, $z[1..k] = \text{LCS}(x[1..i], y[1..j-1])$.

Thus, $c[i, j-1] = k = c[i, j]$.

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

Dynamic Programming Concept (1)

Step 1: Optimal substructure.

An optimal solution to a problem contains optimal solutions to subproblems.

Example:

If $z = LCS(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y .

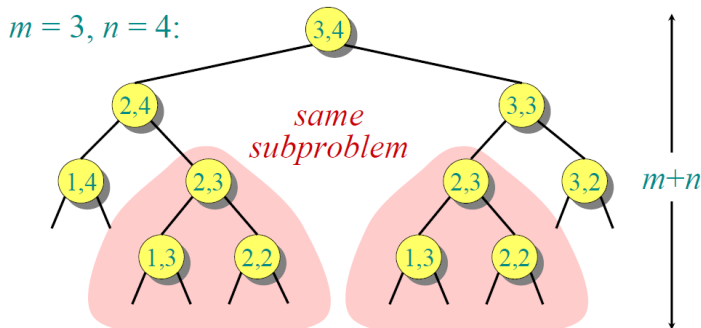
Recursive Algorithm

- ▶ Computation of the length of *LCS*:

```
1 LCSlength(x,y,i,j):  
2   if i == 0 or j == 0  
3     return 0  
4   else if x[i] == y[j]  
5     return LCSlength(x,y,i-1,j-1)+1  
6   else return max{LCSlength(x,y,i-1,j),  
7                   LCSlength(x,y,i,j-1)}
```

- ▶ Remark: if $x[i] \neq y[j]$, the algorithm evaluates two subproblems that are very similar.

Recursive Tree



Height = $m + n \Rightarrow$ work potentially exponential,
but we're solving subproblems already solved!

Dynamic Programming Concept (2)

Step 2: Overlapping subproblems.

A recursive solution contains a "small" number of distinct subproblems repeated many times.

Example:

The number of distinct *LCS* subproblems for two prefixes of lengths m and n is only $m \cdot n$.

Memoization Algorithm

Memoization:

- ▶ After computing a solution to a subproblem, store it in a table.
- ▶ Subsequent calls check the table to avoid repeating the same computation.

Recursive Algorithm with Memoization

Computation of the length of *LCS*:

```
1 LCSlength_(x,y,i,j):  
2   if c[i,j] == NIL  
3       then if i==0 or j==0  
4             c[i,j] = 0  
5   else if x[i] == y[j]  
6       c[i,j] = LCSlength_(x,y,i-1,j-1)+1  
7   else c[i,j] = max{LCSlength_(x,y,i-1,j),  
8                     LCSlength_(x,y,i,j-1)}  
9   return c[i,j]
```

Dynamic Programming (1)

Compute the table bottom-up:

	A	B	C	B	D	A	B
	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1
D	0	0	1	1	1	2	2
C	0	0	1	2	2	2	2
A	0	1	1	2	2	3	3
B	0	1	2	2	3	3	4
A	0	1	2	2	3	4	4

Dynamic Programming (2)

Compute the table bottom-up:

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

LCS-LENGTH(*X*, *Y*)

```

1  m = X.length
2  n = Y.length
3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
4  for i = 1 to m
5      c[i, 0] = 0
6  for j = 0 to n
7      c[0, j] = 0
8  for i = 1 to m
9      for j = 1 to n
10         if xi == yj
11             c[i, j] = c[i - 1, j - 1] + 1
12             b[i, j] = "↖"
13         elseif c[i - 1, j] ≥ c[i, j - 1]
14             c[i, j] = c[i - 1, j]
15             b[i, j] = "↑"
16         else c[i, j] = c[i, j - 1]
17             b[i, j] = "←"
18  return c and b
```

Complexity

- ▶ Time complexity: $T(m, n) = \Theta(m \cdot n)$
- ▶ Space complexity: $S(m, n) = \Theta(m \cdot n)$

Reconstructing LCS

- Trace backwards:

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

```

1  if  $i == 0$  or  $j == 0$ 
2      return
3  if  $b[i, j] == \nwarrow$ 
4      PRINT-LCS( $b, X, i - 1, j - 1$ )
5      print  $x_i$ 
6  elseif  $b[i, j] == \uparrow$ 
7      PRINT-LCS( $b, X, i - 1, j$ )
8  else PRINT-LCS( $b, X, i, j - 1$ )
    
```

- Time complexity: $O(m + n)$

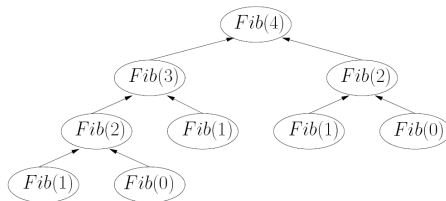
Fibonacci Numbers Revisited (1)

Recall:

- Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

- Recursion tree of brute-force implementation:



Fibonacci Numbers Revisited (2)

Dynamic programming solution:

- ▶ Avoid re-computations of same terms.
- ▶ Store results of subproblems in a table.
- ▶ Thus, $Fib(k)$ is computed exactly once for each k .
- ▶ This basically leads to the previously discussed bottom-up approach.
- ▶ Computation time is $T(n) = \Theta(n)$.