CMPS 6610 – Fall 2018

Dynamic Programming

Carola Wenk

Slides by Carola Wenk, based on slides by Charles Leiserson

Dynamic programming

- Algorithm design technique
- A technique for solving problems that have
  1. an optimal substructure property (recursion)
  2. overlapping subproblems
- **Idea:** Do not repeatedly solve the same subproblems, but solve them only once and store the solutions in a **dynamic programming table**

Example: Fibonacci numbers

- $F(0)=0$; $F(1)=1$; $F(n)=F(n-1)+F(n-2)$ for $n \geq 2$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Dynamic-programming hallmark #1

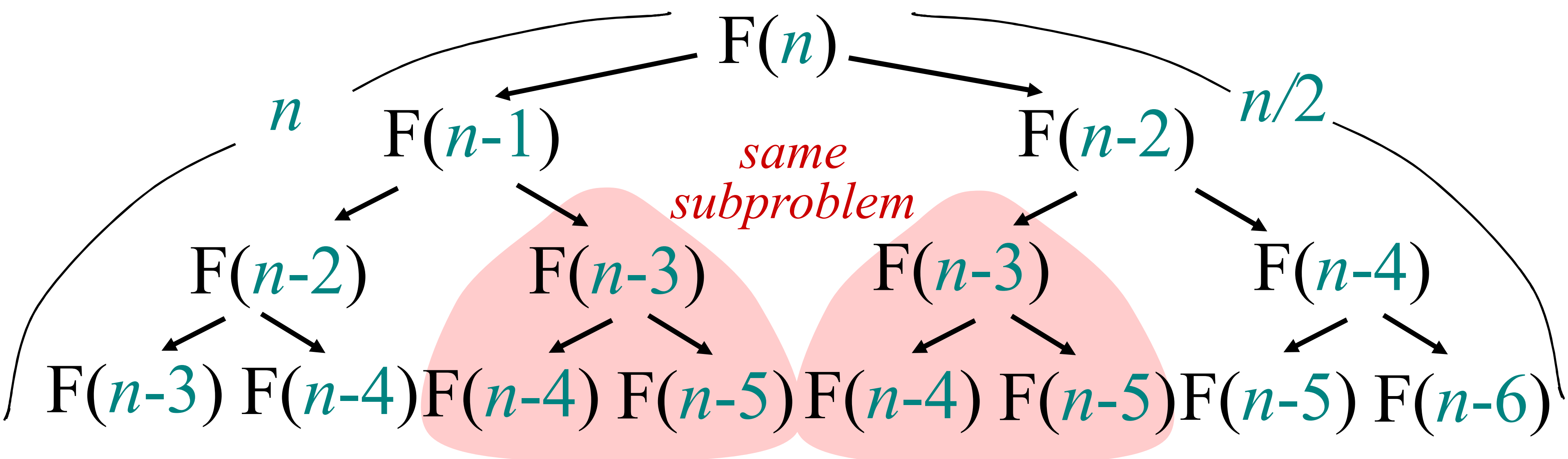
Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

→ ***Recursion***

Example: Fibonacci numbers

- F(0)=0; F(1)=1; F(n)=F(n-1)+F(n-2) for $n \geq 2$
- Implement this recursion directly:



- Runtime is exponential: $2^{n/2} \leq T(n) \leq 2^n$
- But we are repeatedly solving the same subproblems

Dynamic-programming hallmark #2

Overlapping subproblems

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

The number of distinct Fibonacci subproblems is only n.

Dynamic-programming

There are two variants of dynamic programming:

1. Bottom-up dynamic programming (often referred to as “dynamic programming”)

2. Memoization

Bottom-up dynamic-programming algorithm

- Store 1D DP-table and fill bottom-up:

F:											
	0	1	1	2	3	5	8				

fibBottomUpDP(n)

  F[0] $\leftarrow$ 0

  F[1] $\leftarrow$ 1

  for ($i \leftarrow 2$, $i \leq n$, i++)

    F[i] $\leftarrow$ F[i-1]+F[i-2]

  return F[n]

- Time $= \Theta(n)$, space $= \Theta(n)$

Memoization algorithm

Memoization: Use recursive algorithm. After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

fibMemoization(n)
    for all i**:** F[i] = null
    fibMemoizationRec(n,F)
    return F[n]

fibMemoizationRec(n,F)
    if (F[n]= null)
        if (n=0) F[n] $\leftarrow$ 0
        if (n=1) F[n] $\leftarrow$ 1
        F[n] $\leftarrow$ fibMemoizationRec(n-1,F)
                + fibMemoizationRec(n-2,F)
    return F[n]

- Time = $\Theta(n)$, space = $\Theta(n)$

Longest Common Subsequence

Example: ***Longest Common Subsequence (LCS)***

- Given two sequences $x[1\,.\,.\,m]$ and $y[1\,.\,.\,n]$, find a longest subsequence common to them both.

"a" *not* "the"

x: A B C B D A B

y: B D C A B A

BCBA = LCS(x, y)

functional notation, but not a function

Brute-force LCS algorithm

Check every subsequence of $x[1 \,.\,.\, m]$ to see if it is also a subsequence of $y[1 \,.\,.\, n]$.

Analysis

- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).
- Hence, the runtime would be exponential !

Towards a better algorithm

Two-Step Approach:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$.

Strategy: Consider ***prefixes*** of x and y.

- Define $c[i, j] = |\mathrm{LCS}(x[1 \,..\, i], y[1 \,..\, j])|$.
- Then, $c[m, n] = |\mathrm{LCS}(x, y)|$.

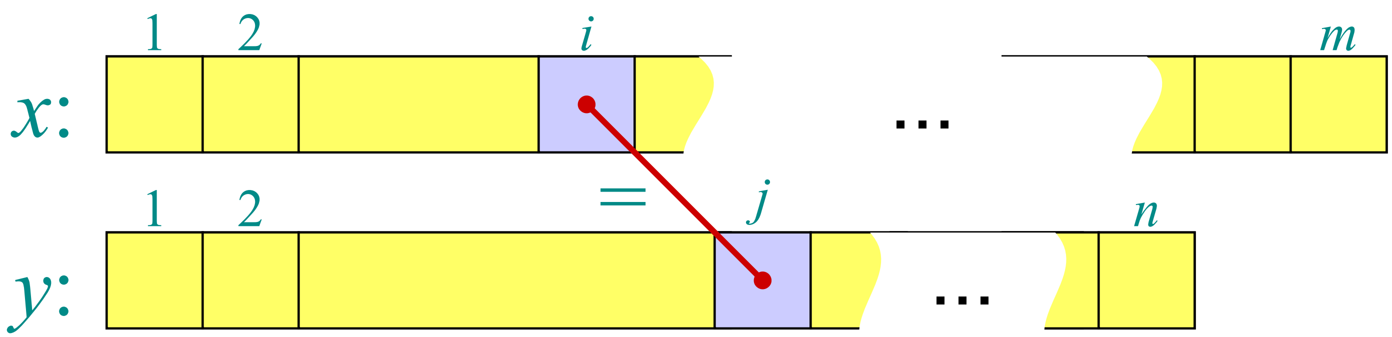
Recursive formulation

Theorem.

Longest common subsequence

$$c[i, j] = \begin{cases} 0 & \text{, if } i=0 \text{ or } j=0 \\ c[i-1, j-1] + 1 & \text{, if } i,j>0 \text{ and } x[i] = y[j] \\ \max\{c[i-1, j], c[i, j-1]\} & \text{, otherwise} \end{cases}$$

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



Let $z[1 \,.\,.\, k] = \text{LCS}(x[1 \,.\,.\, i], y[1 \,.\,.\, j])$, where $c[i, j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1 \,.\,.\, k-1]$ is CS of $x[1 \,.\,.\, i-1]$ and $y[1 \,.\,.\, j-1]$.

Proof (continued)

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

Suppose w is a longer CS of $x[1\,..\,i{-}1]$ and $y[1\,..\,j{-}1]$, that is, $|\,w\,| > k{-}1$. Then, ***cut and paste***: $w \,\|\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1\,..\,i]$ and $y[1\,..\,j]$ with $|\,w \,\|\, z[k]\,| > k$. Contradiction, proving the claim.

Thus, $c[i{-}1, j{-}1] = k{-}1$, which implies that $c[i, j] = c[i{-}1, j{-}1] + 1$.

Other cases are similar. $\square$

Dynamic-programming hallmark #1

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

→ ***Recursion***

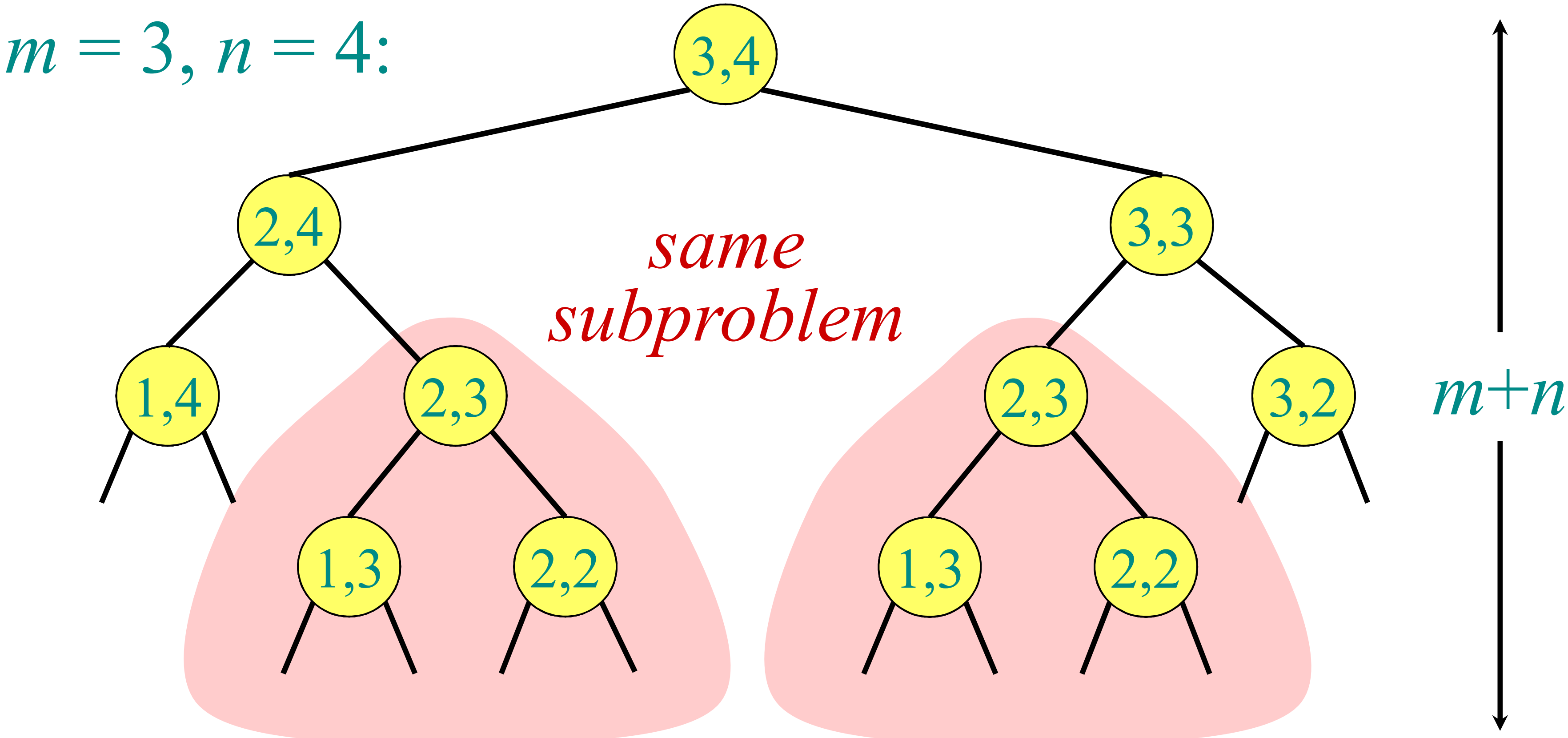
If $z = \mathrm{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

Recursive algorithm for LCS

LCS(x, y, i, j)
    if $(i{=}0 \text{ or } j{=}0)$
        $c[i, j] \leftarrow 0$
    else if $x[i] = y[\,j]$
        $c[i, j] \leftarrow$ LCS$(x, y, i{-}1, j{-}1) + 1$
    else $c[i, j] \leftarrow \max\{LCS(x, y, i{-}1, j)$,
                        LCS$(x, y, i, j{-}1)\}$
    return $c[i, j]$

Worst-case: $x[i] \neq y[\,j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion tree (worst case)



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

Dynamic-programming hallmark #2

Overlapping subproblems

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

The distinct LCS subproblems are all the pairs (i,j). The number of such pairs for two strings of lengths m and n is only mn.

Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS_mem(x, y, i, j)
    if $c[i, j] = \text{null}$
        if $(i=0 \text{ or } j=0)$
            $c[i, j] \leftarrow 0$
        else if $x[i] = y[j]$
            $c[i, j] \leftarrow$ LCS_mem$(x, y, i-1, j-1) + 1$
        else $c[i, j] \leftarrow \max\{$LCS_mem $(x, y, i-1, j)$,
                            LCS_mem $(x, y, i, j-1)\}$
    return $c[i, j]$

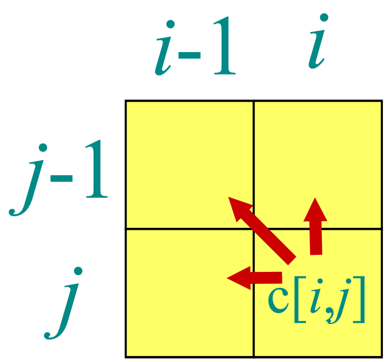
same as before

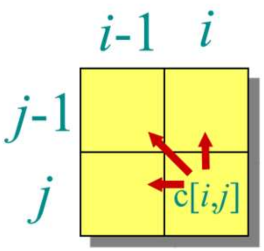
Space = time $= \Theta(mn)$; constant work per table entry.

Recursive formulation

$$c[i, j] = \begin{cases} 0 & \text{, if } i=0 \text{ or } j=0 \\ c[i-1, j-1] + 1 & \text{, if } i,j>0 \text{ and } x[i] = y[j] \\ \max\{c[i-1, j], c[i, j-1]\} & \text{, otherwise} \end{cases}$$

c:





Bottom-up dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

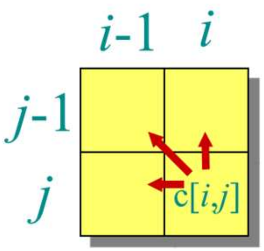
Space = $\Theta(mn)$.

$y\downarrow$	$x\rightarrow$	A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Bottom-up DP

```
LCS_bottomUp(x[1..m], y[1..n])
    for (i=0; i≤m; i++) c[i,0]=0;
    for (j=0; j≤n; j++) c[0,j]=0;
    for (j=1; j≤n; j++)
        for (i=1; i≤m; i++)
            if x[i] = y[ j] {
                c[i, j] ← c[i-1, j-1]+1
                arrow[i,j]="diagonal";
            } else { // compute max
                if (c[i-1, j]≥ c[i, j-1]){
                    c[i, j] ← c[i-1, j]
                    arrow[i,j]="left";
                } else{
                    c[i, j] ← c[i, j-1]
                    arrow[i,j]="up";
                }
            }
    return c and arrow
```

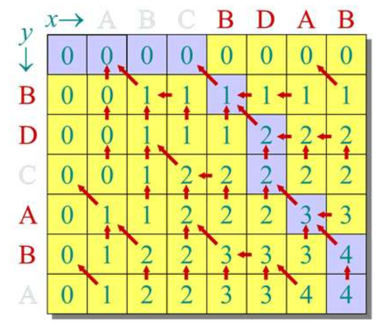
Space = time = $\Theta(mn)$; constant work per table entry.



Reconstruct LCS by backtracking

$y \downarrow$ \ $x \rightarrow$		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Reducing space



- We can compute the *length* of an LCS in $\Theta(mn)$ time using only $\Theta(\min\{m,n\})$ space by filling the table row-by-row and only keeping two rows (or column-by-column if columns are shorter). (Exercise: use only $\min\{m,n\}+\Theta(1)$ space.)
- However, without the whole DP table we cannot construct an LCS.
- Hirschberg's algorithm combines DP with divide-and-conquer to construct an LCS in $\Theta(mn)$ time using only $\Theta(\min\{m,n\})$ space

Two recursive formulas

Consider ***prefixes*** of x and y.

- Define $c[i, j] = |\,\mathrm{LCS}(x[1\,..\,i], y[1\,..\,j])\,|$.
- Then, $c[m, n] = |\,\mathrm{LCS}(x, y)\,|$.

$$c[i,j] = \begin{cases} 0 & \text{, if } i=0 \text{ or } j=0 \\ c[i-1, j-1] + 1 & \text{, if } i,j>0 \text{ and } x[i] = y[j] \\ \max\{c[i-1, j], c[i, j-1]\} & \text{, otherwise} \end{cases}$$

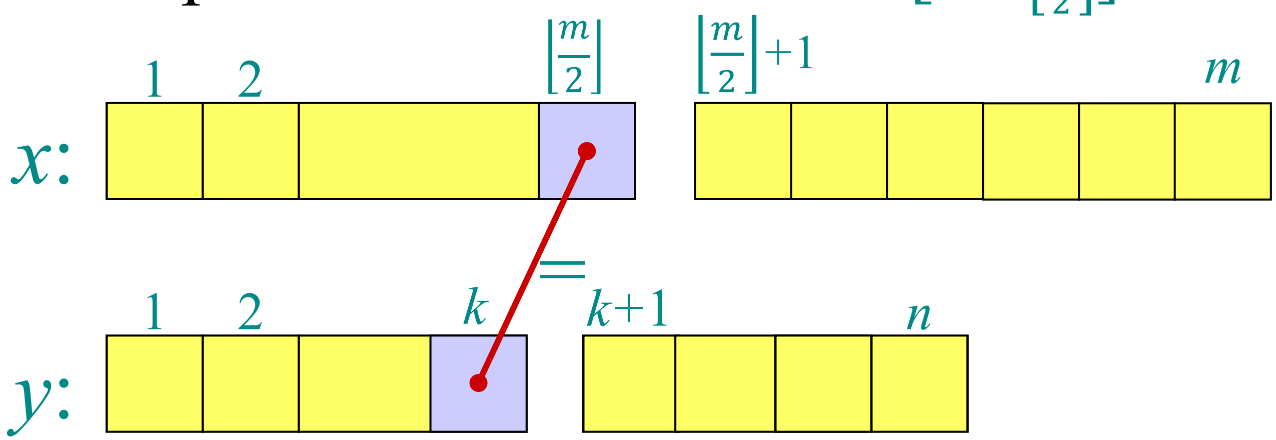
Equivalently, we can consider ***suffixes*** of x and y.

- Define $c'[i, j] = |\,\mathrm{LCS}(x[i\,..\,m], y[j\,..\,n])\,|$.
- Then, $c'[1,1] = |\,\mathrm{LCS}(x, y)\,|$.

$$c'[i,j] = \begin{cases} 0 & \text{, if } i=m+1 \text{ or } j=n+1 \\ c'[i+1, j+1] + 1 & \text{, if } i\le m,\, j \le n \text{ and } x[i] = y[j] \\ \max\{c'[i+1, j], c'[i, j+1]\} & \text{, otherwise} \end{cases}$$

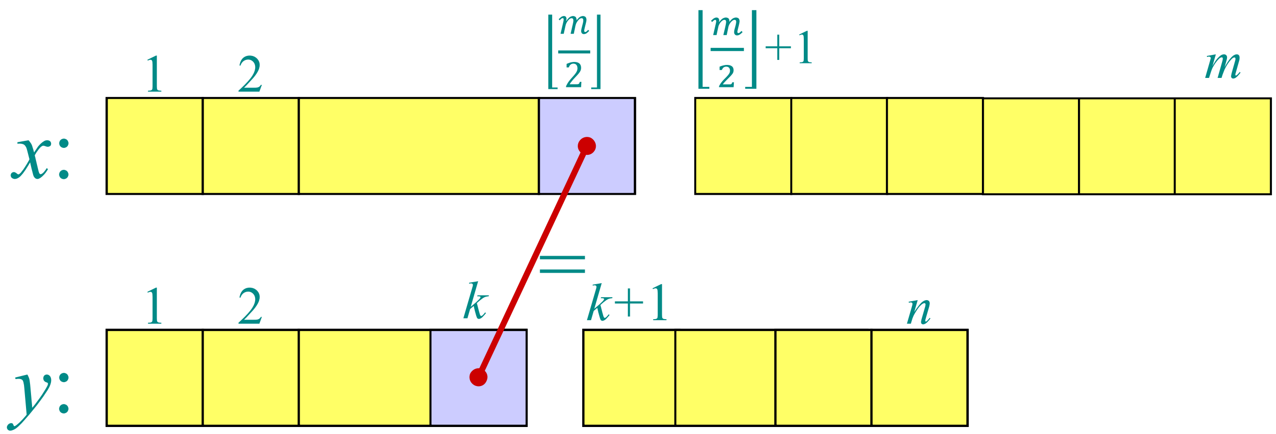
Hirschberg's D&C

- Without loss of generality assume $n \leq m$.
- **Idea:** Use divide-and-conquer on string x.
- Let z be an LCS for x and y, and consider the correspondence of matching characters between x and y described by z.
- Let $y[k]$ be the rightmost character in y that corresponds to a character in $x[1..\lfloor\frac{m}{2}\rfloor]$



Hirschberg's D&C

- Let $y[k]$ be the rightmost character in y that corresponds to a character in $x[1..\lfloor \frac{m}{2} \rfloor]$;
  or 0 if no such character exists.



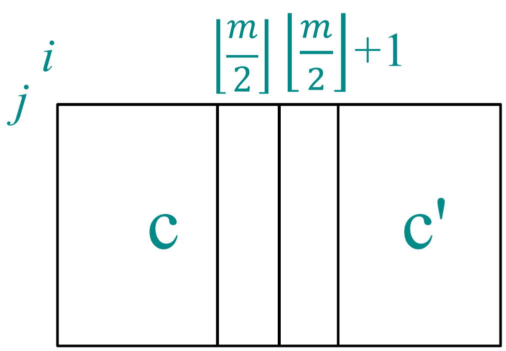
- Then:
  $$| \mathrm{LCS}(x, y) | = \max_{0 \le l \le n} \{ c[\lfloor \tfrac{m}{2} \rfloor, l] + c'[\lfloor \tfrac{m}{2} \rfloor + 1, l+1] \}$$

Algorithm

$$|\,\mathrm{LCS}(x, y)\,| = \max_{0 \le l \le n} \{ c[\lfloor \tfrac{m}{2} \rfloor, l] + c'[\lfloor \tfrac{m}{2} \rfloor + 1, l+1] \}$$

1. Find $k = \arg\max_{0 \le l \le n} \{ c[\lfloor \tfrac{m}{2} \rfloor, l] + c'[\lfloor \tfrac{m}{2} \rfloor + 1, l+1] \}$
2. Recursively compute $z_1 = \mathrm{LCS}(x[1..\lfloor \tfrac{m}{2} \rfloor], y[1..k])$ and $z_2 = \mathrm{LCS}(x[\lfloor \tfrac{m}{2} \rfloor + 1..m], y[k+1..n])$, and return the concatenation $z = z_1\, z_2$

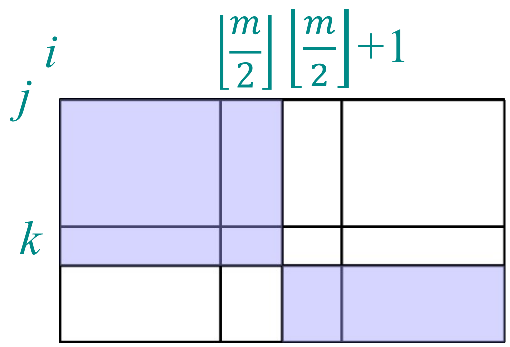
Step 1: Compute $c[\lfloor \tfrac{m}{2} \rfloor, l]$ and $c'[\lfloor \tfrac{m}{2} \rfloor + 1, l+1]$ for all $0 \le l \le n$.



This can be done in $O(mn)$ time and $O(n)$ space, using standard DP without storing the whole table.

Runtime analysis

In the root of the recursion tree the runtime is cmn.
The total work in each subsequent level is half:



Therefore the total runtime is at most:

$$cmn \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^{i} \in O(mn)$$

The total space needed is only the space of $O(n)$ used within each recursive call.