

Dynamic Programming

Outline

- Recap sequence alignment in quadratic time and space
- Hirschberg's algorithm: sequence alignment in linear space via divide-and-conquer
- Longest common subsequence

String Similarity

How similar are two strings?

- `ocurrence`
- `occurrence`

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| o | c | u | r | r | a | n | c | e | - |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| o | c | c | u | r | r | e | n | c | e |
|---|---|---|---|---|---|---|---|---|---|

6 mismatches, 1 gap

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| o | c | - | u | r | r | a | n | c | e |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| o | c | c | u | r | r | e | n | c | e |
|---|---|---|---|---|---|---|---|---|---|

1 mismatch, 1 gap

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| o | c | - | u | r | r | - | a | n | c | e |
|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| o | c | c | u | r | r | e | - | n | c | e |
|---|---|---|---|---|---|---|---|---|---|---|

0 mismatches, 3 gaps

Edit Distance

Applications.

- Basis for Unix diff.
- Speech recognition.
- Computational biology.

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ ; mismatch penalty α_{pq} .
- Cost = sum of gap and mismatch penalties.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| C | T | G | A | C | C | T | A | C | C | T |
|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| C | C | T | G | A | C | T | A | C | A | T |
|---|---|---|---|---|---|---|---|---|---|---|

$$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$$

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| - | C | T | G | A | C | C | T | A | C | C | T |
|---|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| C | C | T | G | A | C | - | T | A | C | A | T |
|---|---|---|---|---|---|---|---|---|---|---|---|

$$2\delta + \alpha_{CA}$$

Sequence Alignment

Goal: Given two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$ find alignment of minimum cost.

Def. An **alignment** M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings.

Def. The pair $x_i - y_j$ and $x_{i'} - y_{j'}$ **cross** if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i: x_i \text{ unmatched}} \delta + \sum_{j: y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Ex: CTACCG **vs.** TACATG.

Sol: $M = x_2 - y_1, x_3 - y_2, x_4 - y_3, x_5 - y_4, x_6 - y_6$.

| x_1 | x_2 | x_3 | x_4 | x_5 | | x_6 |
|-------|-------|-------|-------|-------|---|-------|
| C | T | A | C | C | - | G |

| | y_1 | y_2 | y_3 | y_4 | y_5 | y_6 |
|---|-------|-------|-------|-------|-------|-------|
| - | T | A | C | A | T | G |

Sequence Alignment: Problem Structure

Def. $OPT(i, j)$ = min cost of aligning strings $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$.

- Case 1: OPT matches x_i - y_j .
 - pay mismatch for x_i - y_j + min cost of aligning two strings $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_{j-1}$
- Case 2a: OPT leaves x_i unmatched.
 - pay gap for x_i and min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_j$
- Case 2b: OPT leaves y_j unmatched.
 - pay gap for y_j and min cost of aligning $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_{j-1}$

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

```
Sequence-Alignment( $m, n, x_1x_2\dots x_m, y_1y_2\dots y_n, \delta, \alpha$ ) {  
  for  $i = 0$  to  $m$   
     $M[i, 0] = i\delta$   
  for  $j = 0$  to  $n$   
     $M[0, j] = j\delta$   
  
  for  $i = 1$  to  $m$   
    for  $j = 1$  to  $n$   
       $M[i, j] = \min(\alpha[x_i, y_j] + M[i-1, j-1],$   
                     $\delta + M[i-1, j],$   
                     $\delta + M[i, j-1])$   
  
  return  $M[m, n]$   
}
```

Analysis. $\Theta(mn)$ time and space.

English words or sentences: $m, n \leq 10$.

Computational biology: $m = n = 100,000$. 10 billions ops OK, but 10GB array?

Outline

- Recap sequence alignment in quadratic time and space
- Hirschberg's algorithm: sequence alignment in linear space via divide-and-conquer
- Longest common subsequence

Sequence Alignment: Linear Space

Q. Can we avoid using quadratic **space**?

Easy. Optimal **value** in $O(m)$ space and $O(mn)$ time.

- Compute $\text{OPT}(i, \cdot)$ from $\text{OPT}(i-1, \cdot)$.
- No longer a simple way to recover alignment itself.

Theorem. [Hirschberg 1975] Optimal **alignment** in $O(m + n)$ space and $O(mn)$ time.

- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

Programming
Techniques

G. Manacher
Editor

A Linear Space
Algorithm for
Computing Maximal
Common Subsequences

D.S. Hirschberg
Princeton University

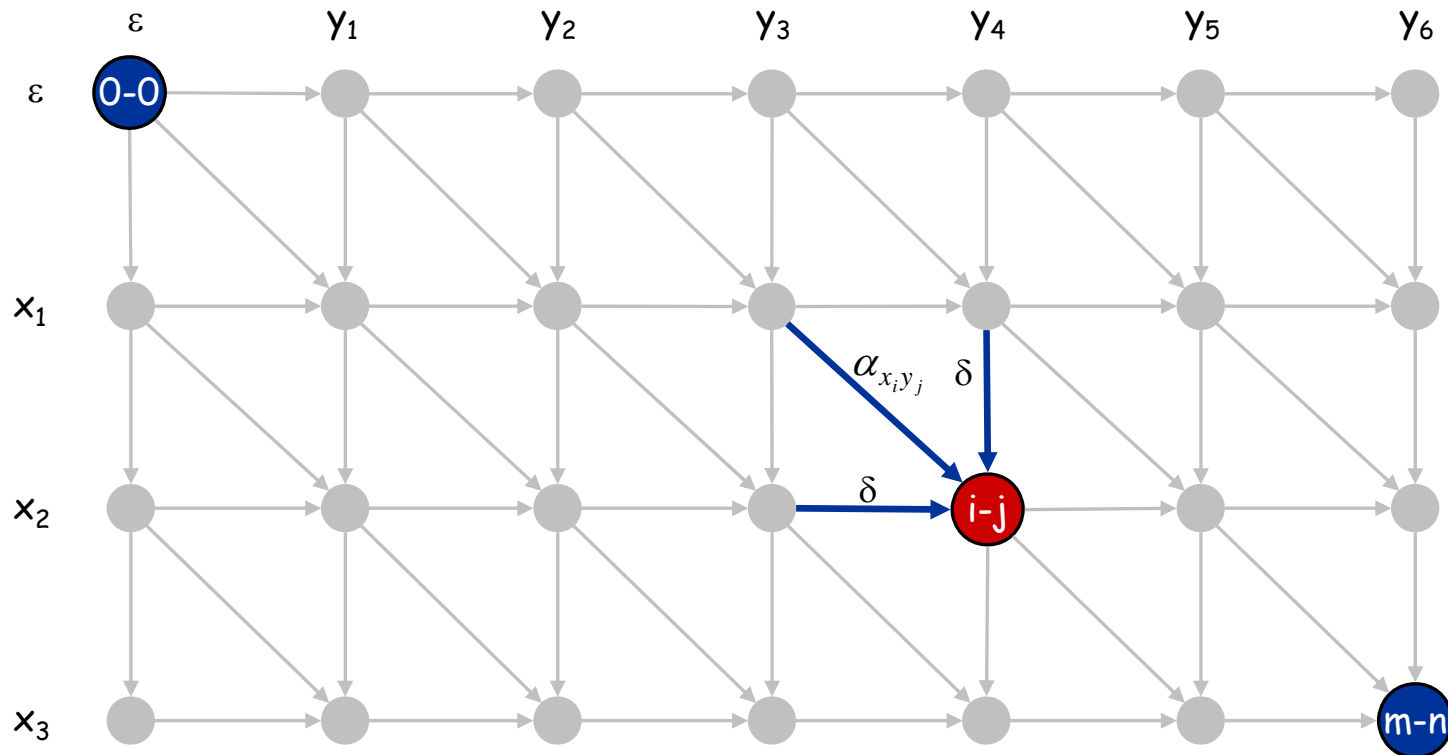
The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space. An algorithm is presented which will solve this problem in quadratic time and in linear space.

Key Words and Phrases: subsequence, longest common subsequence, string correction, editing
CR Categories: 3.63, 3.73, 3.79, 4.22, 5.25

Sequence Alignment: Linear Space

Edit distance graph.

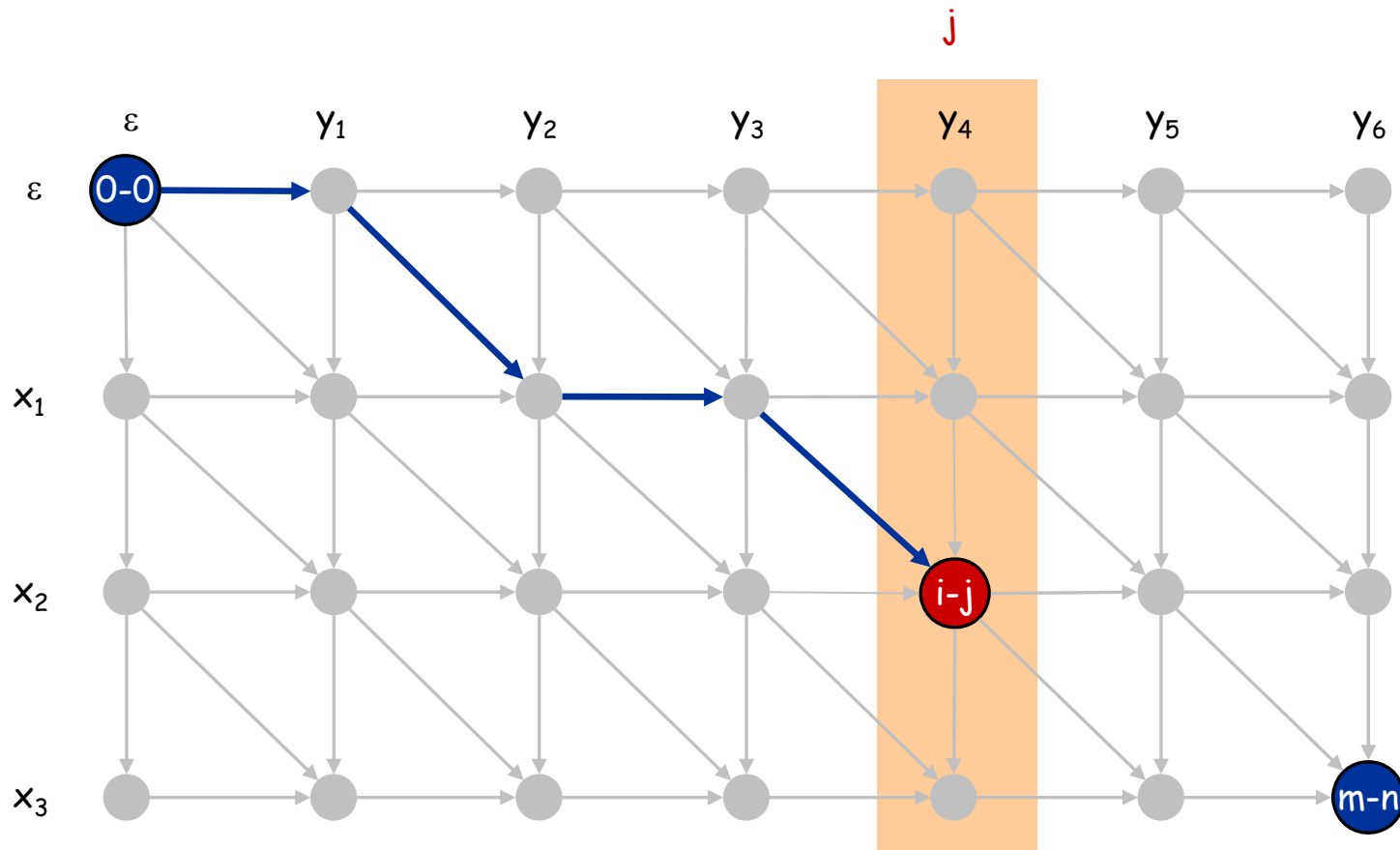
- Let $f(i, j)$ be shortest path from $(0,0)$ to (i, j) .
- Observation: $f(i, j) = \text{OPT}(i, j)$.



Sequence Alignment: Linear Space

Edit distance graph.

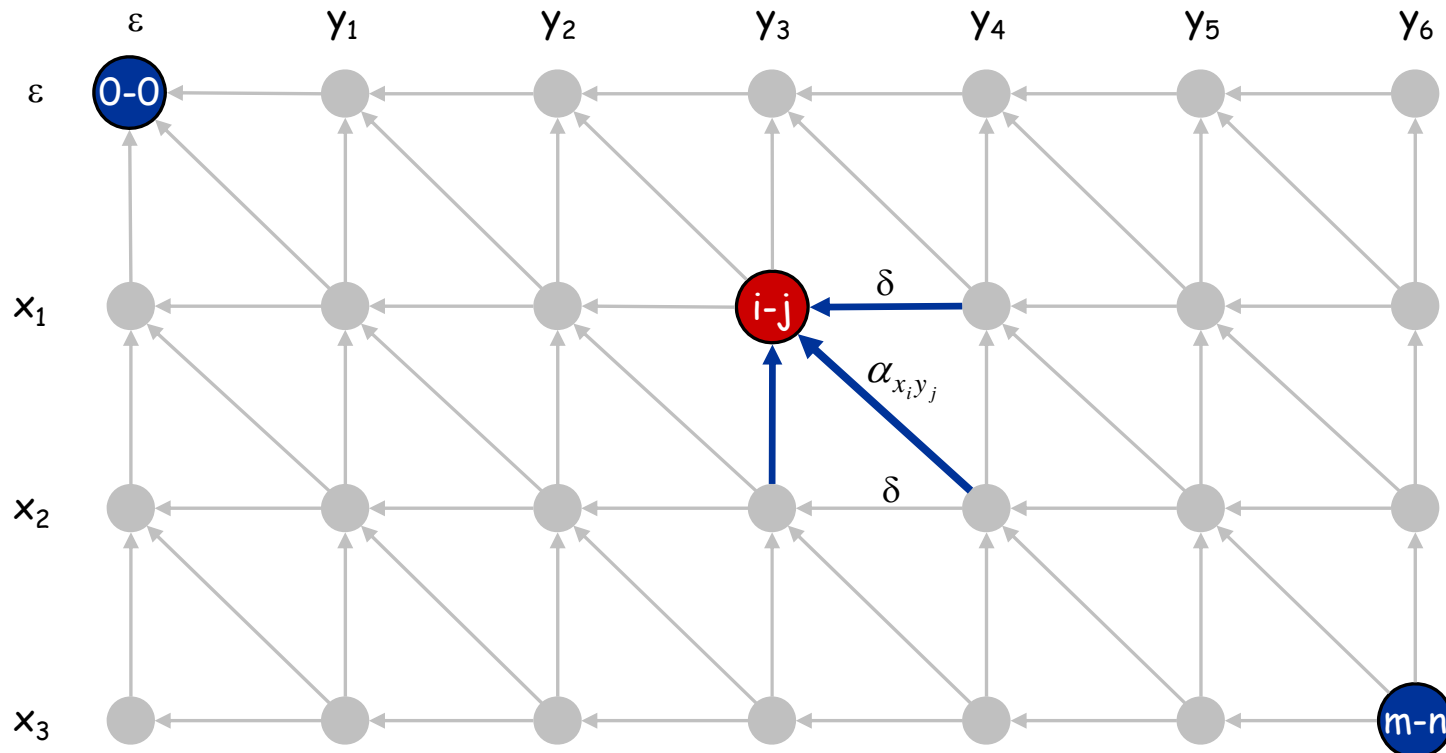
- Let $f(i, j)$ be shortest path from $(0,0)$ to (i, j) .
- Can compute $f(\cdot, j)$ for any j in $O(mn)$ time and $O(m)$ space.



Sequence Alignment: Linear Space

Edit distance graph.

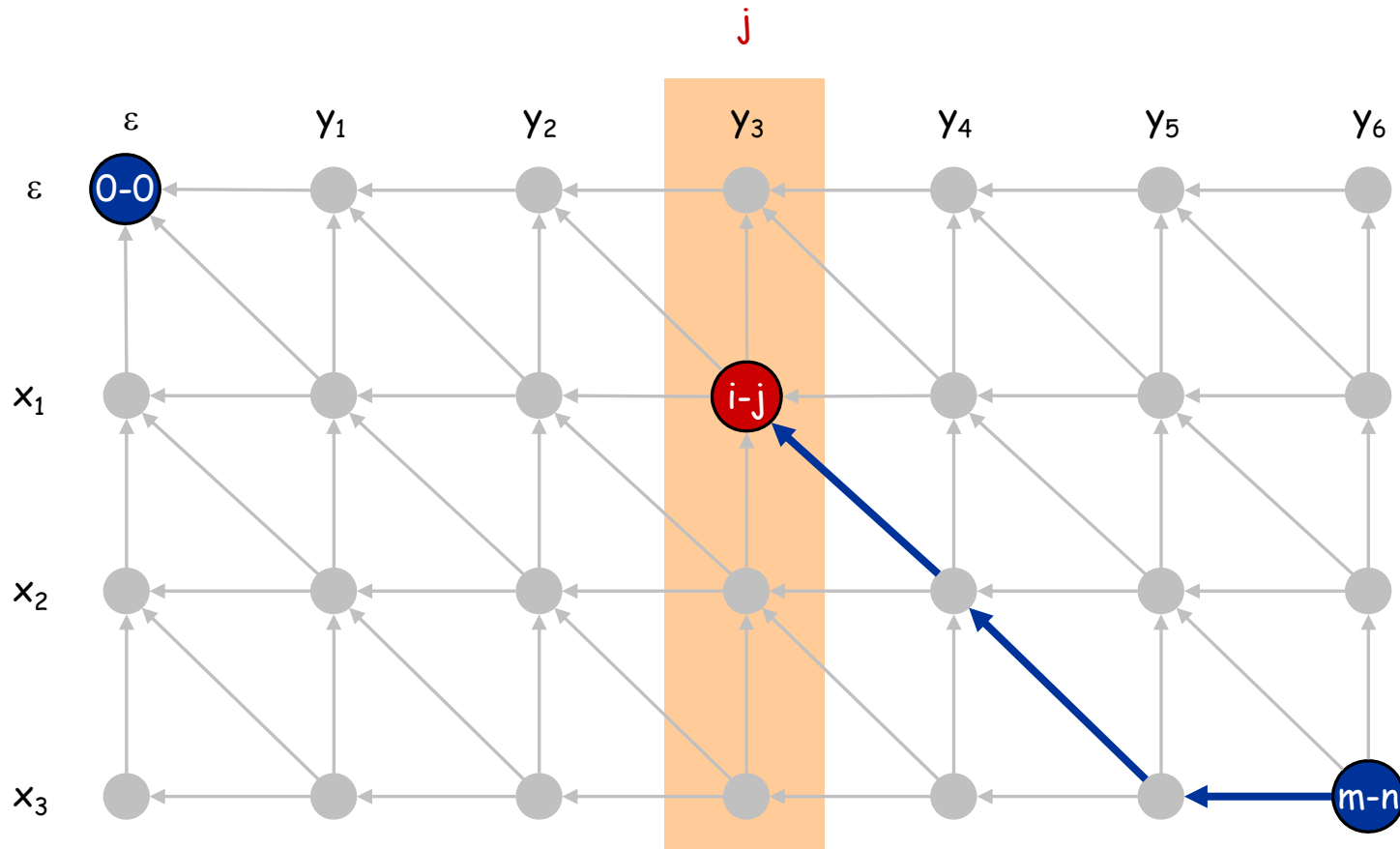
- Let $g(i, j)$ be shortest path from (i, j) to (m, n) .
- Can compute by reversing the edge orientations and inverting the roles of $(0, 0)$ and (m, n)



Sequence Alignment: Linear Space

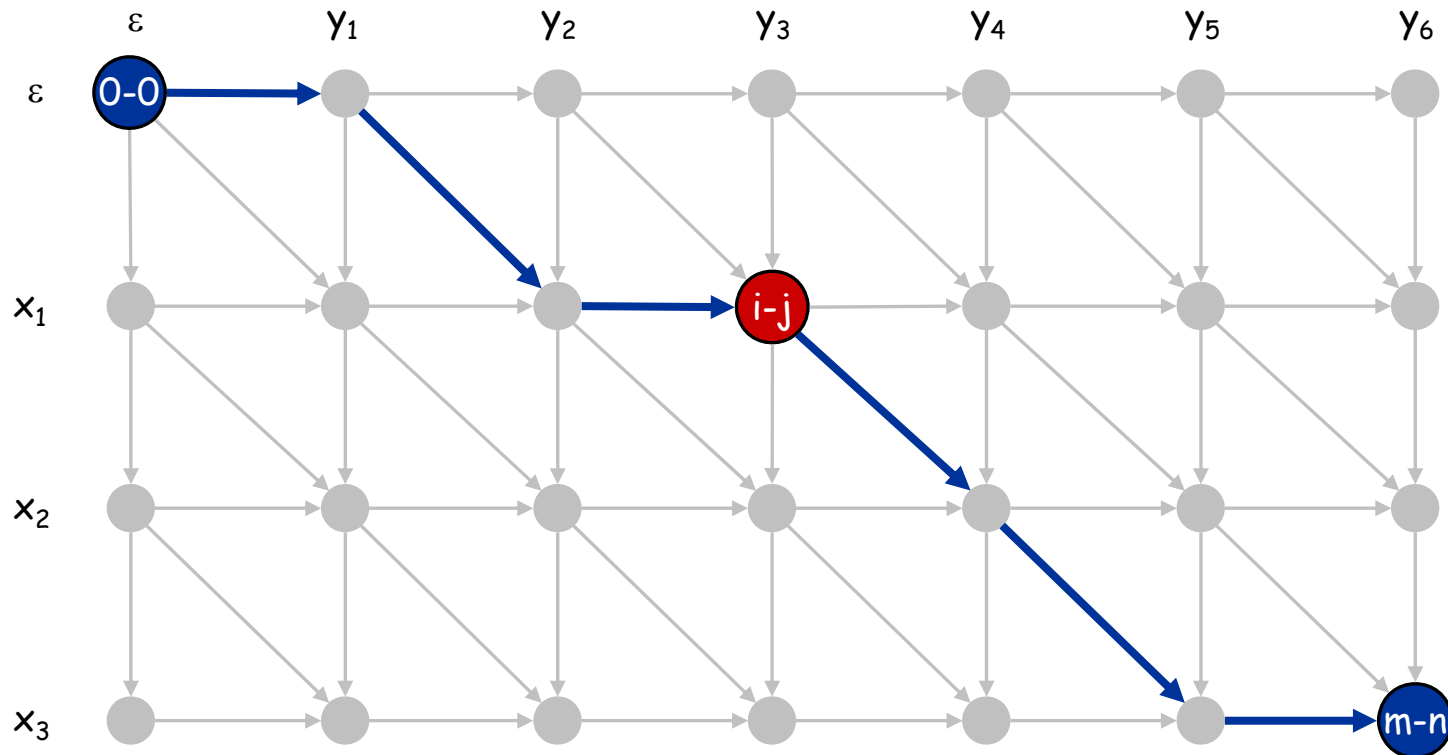
Edit distance graph.

- Let $g(i, j)$ be shortest path from (i, j) to (m, n) .
- Can compute $g(\cdot, j)$ for any j in $O(mn)$ time and $O(m)$ space.



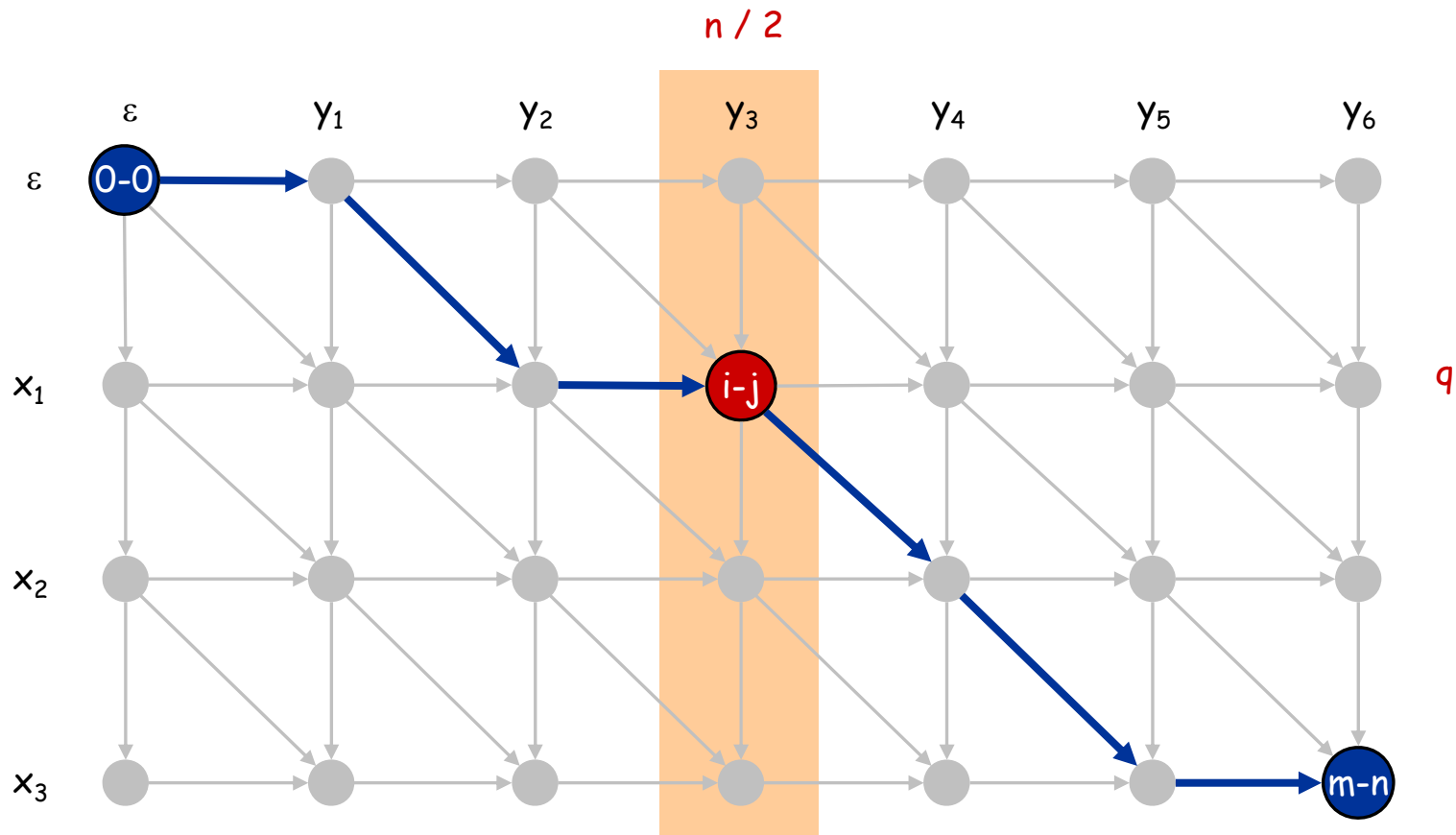
Sequence Alignment: Linear Space

Observation 1. The cost of the shortest path that uses (i, j) is $f(i, j) + g(i, j)$.



Sequence Alignment: Linear Space

Observation 2. let q be an index that minimizes $f(q, n/2) + g(q, n/2)$. Then, the shortest path from $(0, 0)$ to (m, n) uses $(q, n/2)$.

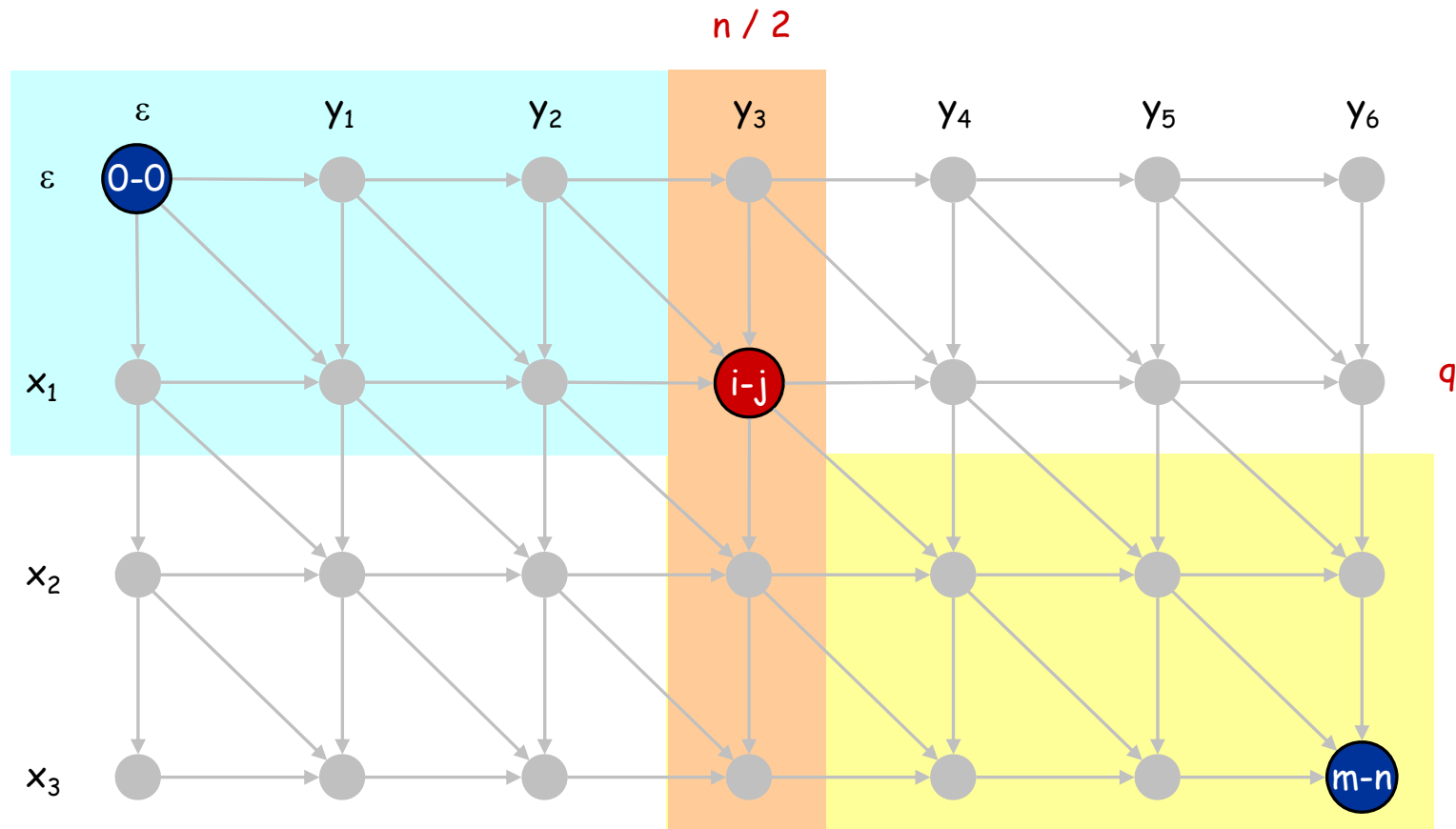


Sequence Alignment: Linear Space

Divide: find index q that minimizes $f(q, n/2) + g(q, n/2)$ using DP.

- Align x_q and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.



Divide-and-Conquer-Alignment Pseudocode

```
Divide-and-Conquer-Alignment( $X, Y$ )
  Let  $m$  be the number of symbols in  $X$ 
  Let  $n$  be the number of symbols in  $Y$ 
  If  $m \leq 2$  or  $n \leq 2$  then
    Compute optimal alignment using Alignment( $X, Y$ )
  Call Space-Efficient-Alignment( $X, Y[1:n/2]$ )
  Call Backward-Space-Efficient-Alignment( $X, Y[n/2 + 1:n]$ )
  Let  $q$  be the index minimizing  $f(q, n/2) + g(q, n/2)$ 
  Add  $(q, n/2)$  to global list  $P$ 
  Divide-and-Conquer-Alignment( $X[1:q], Y[1:n/2]$ )
  Divide-and-Conquer-Alignment( $X[q + 1:n], Y[n/2 + 1:n]$ )
  Return  $P$ 
```

*

- Maintain a global list P that stores nodes on the shortest corner-to-corner path.
- $|P| \leq m+n$ implies $O(m+n)$ space of the algorithm in total

Sequence Alignment: Running Time Analysis Warmup

Theorem. Let $T(m, n)$ = max running time of algorithm on strings of length at most m and n . $T(m, n) = O(mn \log n)$.

$$T(m, n) \leq 2T(m, n/2) + O(mn) \Rightarrow T(m, n) = O(mn \log n)$$

Remark. Analysis is not tight because two sub-problems are of size $(q, n/2)$ and $(m - q, n/2)$. In next slide, we save $\log n$ factor.

Sequence Alignment: Running Time Analysis

Theorem. Let $T(m, n)$ = max running time of algorithm on strings of length m and n . $T(m, n) = O(mn)$.

Pf. (by induction on n)

- $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find index q .
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
- Choose constant c so that:

$$T(m, 2) \leq cm$$

$$T(2, n) \leq cn$$

$$T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$$

- Base cases: $m = 2$ or $n = 2$.
- Inductive hypothesis: $T(m, n) \leq 2cmn$.

$$\begin{aligned} T(m, n) &\leq T(q, n/2) + T(m - q, n/2) + cmn \\ &\leq 2cq n/2 + 2c(m - q)n/2 + cmn \\ &= cq n + cmn - cq n + cmn \\ &= 2cmn \end{aligned}$$

Outline

- Recap sequence alignment in quadratic time and space
- Hirschberg's algorithm: sequence alignment in linear space via divide-and-conquer
- Longest common subsequence

Outline

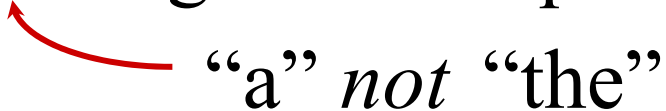
- Recap sequence alignment in quadratic time and space
- Hirschberg's algorithm: sequence alignment in linear space via divide-and-conquer
- Longest common subsequence

Longest Common Subsequence

E.g. Compare the similarity of two DNA strands.


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

Longest Common Subsequence

- Given two sequences $x[1 \dots m]$ and $y[1 \dots n]$, find a longest subsequence common to them both.
 “a” *not* “the”

Longest Common Subsequence

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

 “a” *not* “the”

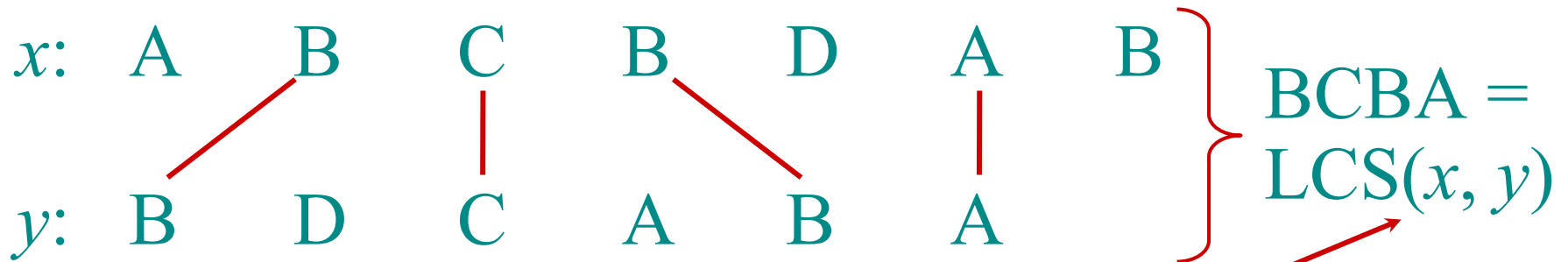
x : A B C B D A B

y : B D C A B A

Longest Common Subsequence

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

“a” *not* “the”



functional notation,
but not a function

Brute-force LCS algorithm

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

Brute-force LCS algorithm

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

Analysis

- Checking = $O(n)$ time per subsequence.
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time = $O(n2^m)$
= exponential time.

Towards a better algorithm

Simplification:

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

Towards a better algorithm

Simplification:

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 $|s|$.

Towards a better algorithm

Simplification:

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

Recursive formulation

Theorem.

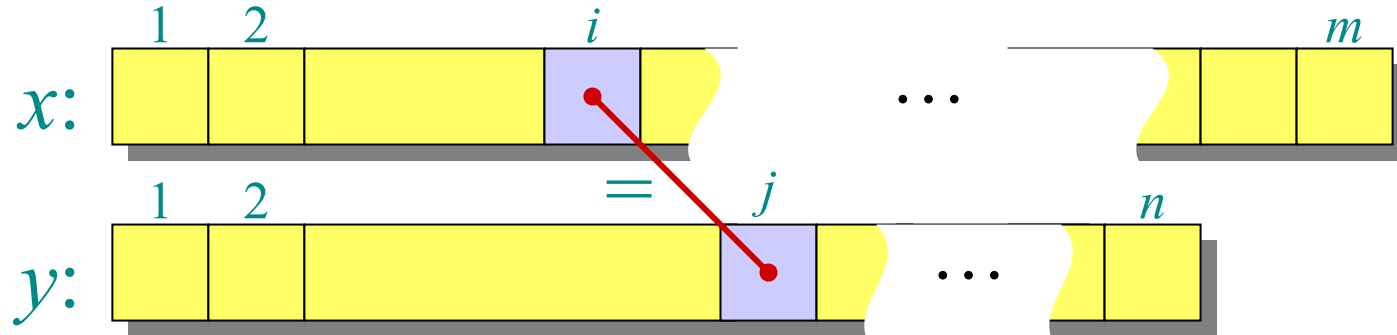
$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max \{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

Recursive formulation

Theorem.

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

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

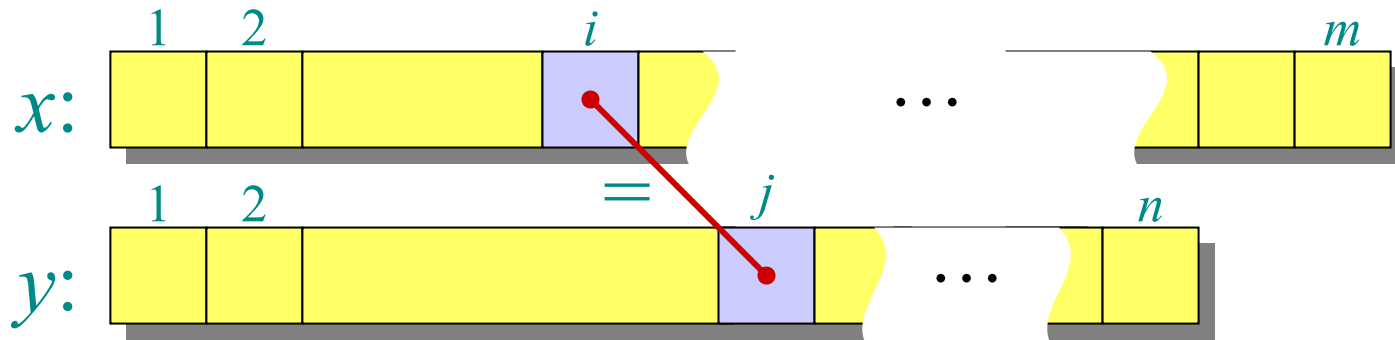


Recursive formulation

Theorem.

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

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



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

Proof (continued)

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

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

Proof (continued)

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

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

Dynamic-programming hallmark #1

Optimal substructure

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

Dynamic-programming hallmark #1

Optimal substructure

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

If $z = \text{LCS}(x, y)$, 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  $x[i] = y[j]$   
    then  $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$   
    else  $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j),$   
                                    $\text{LCS}(x, y, i, j-1) \}$ 
```

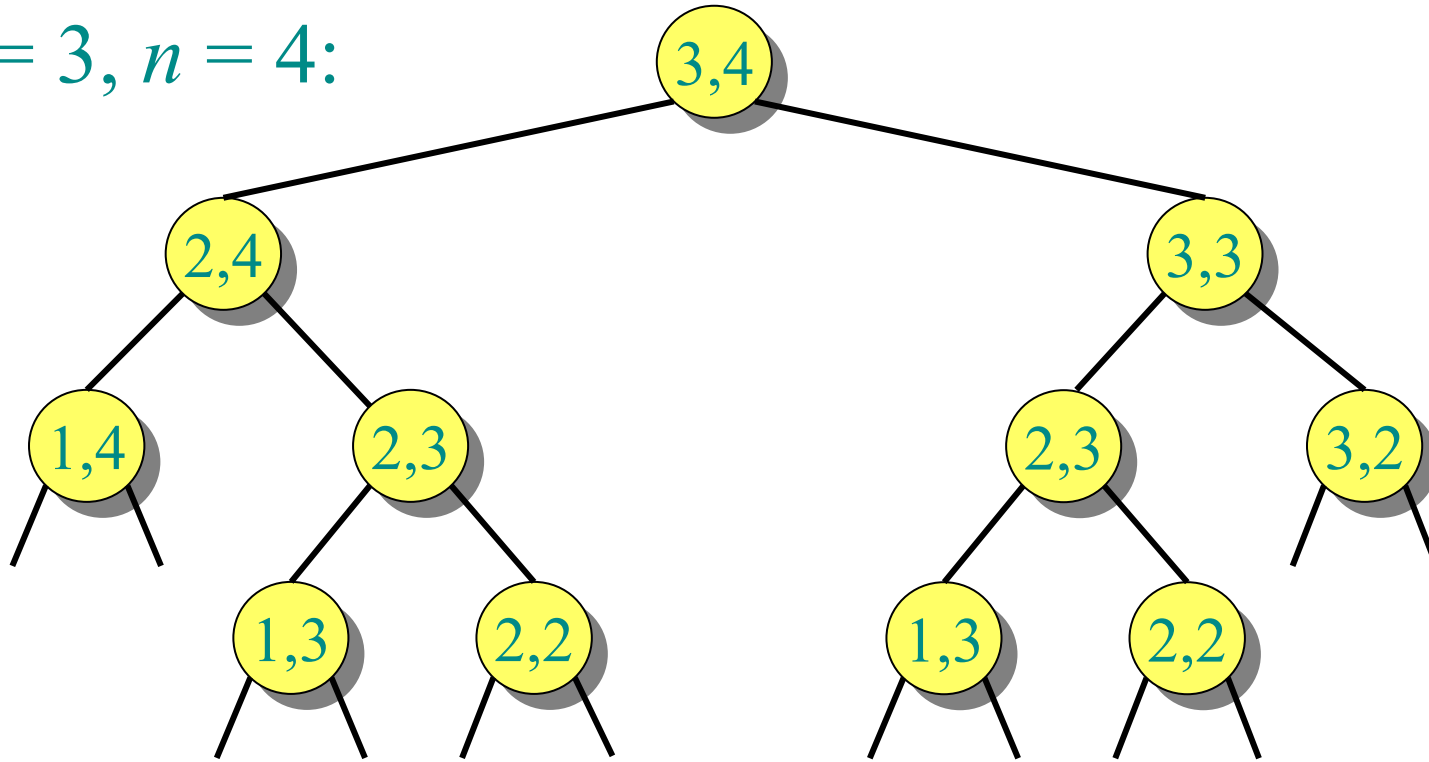
Recursive algorithm for LCS

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

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

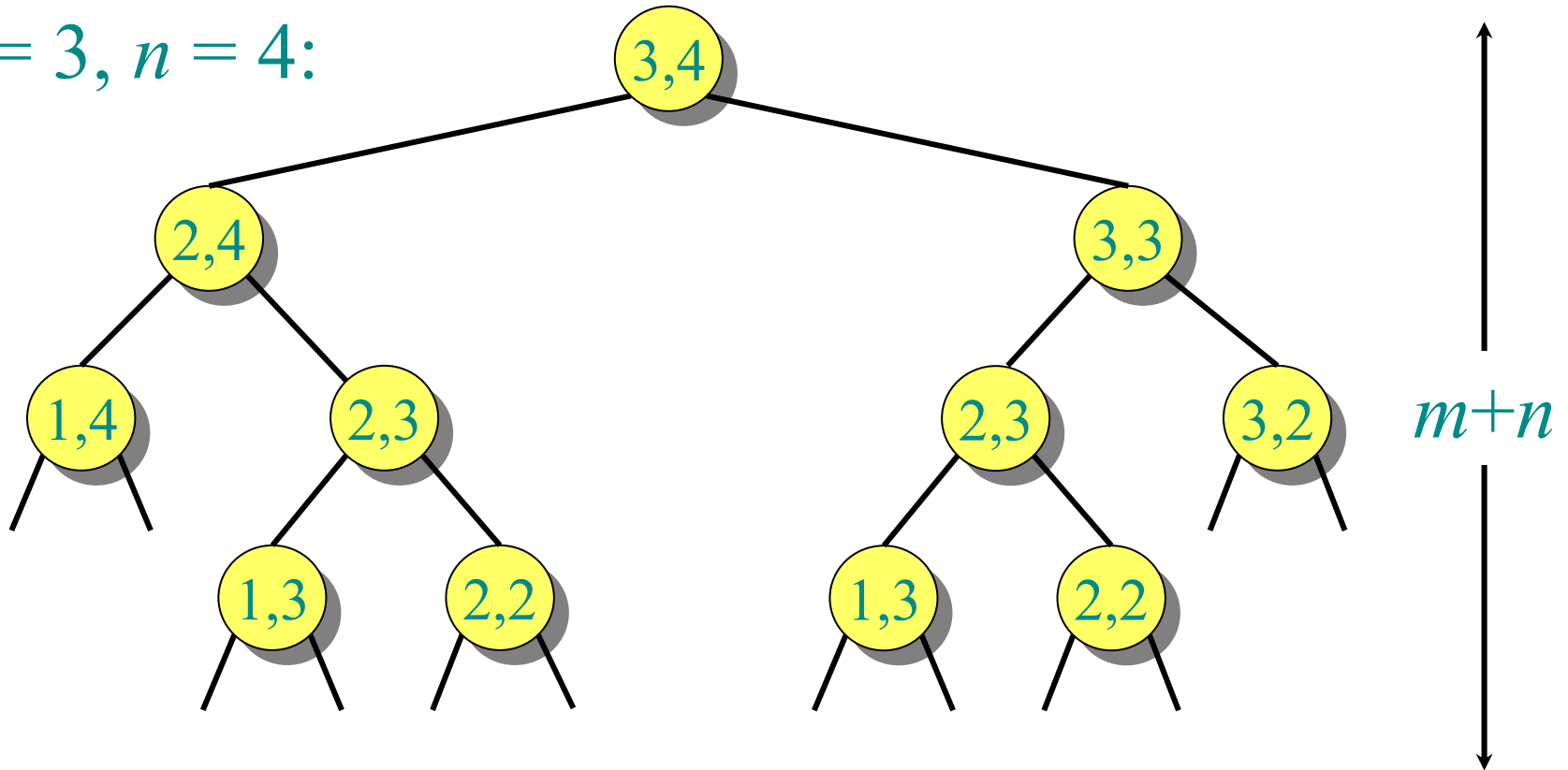
Recursion tree

$m = 3, n = 4$:



Recursion tree

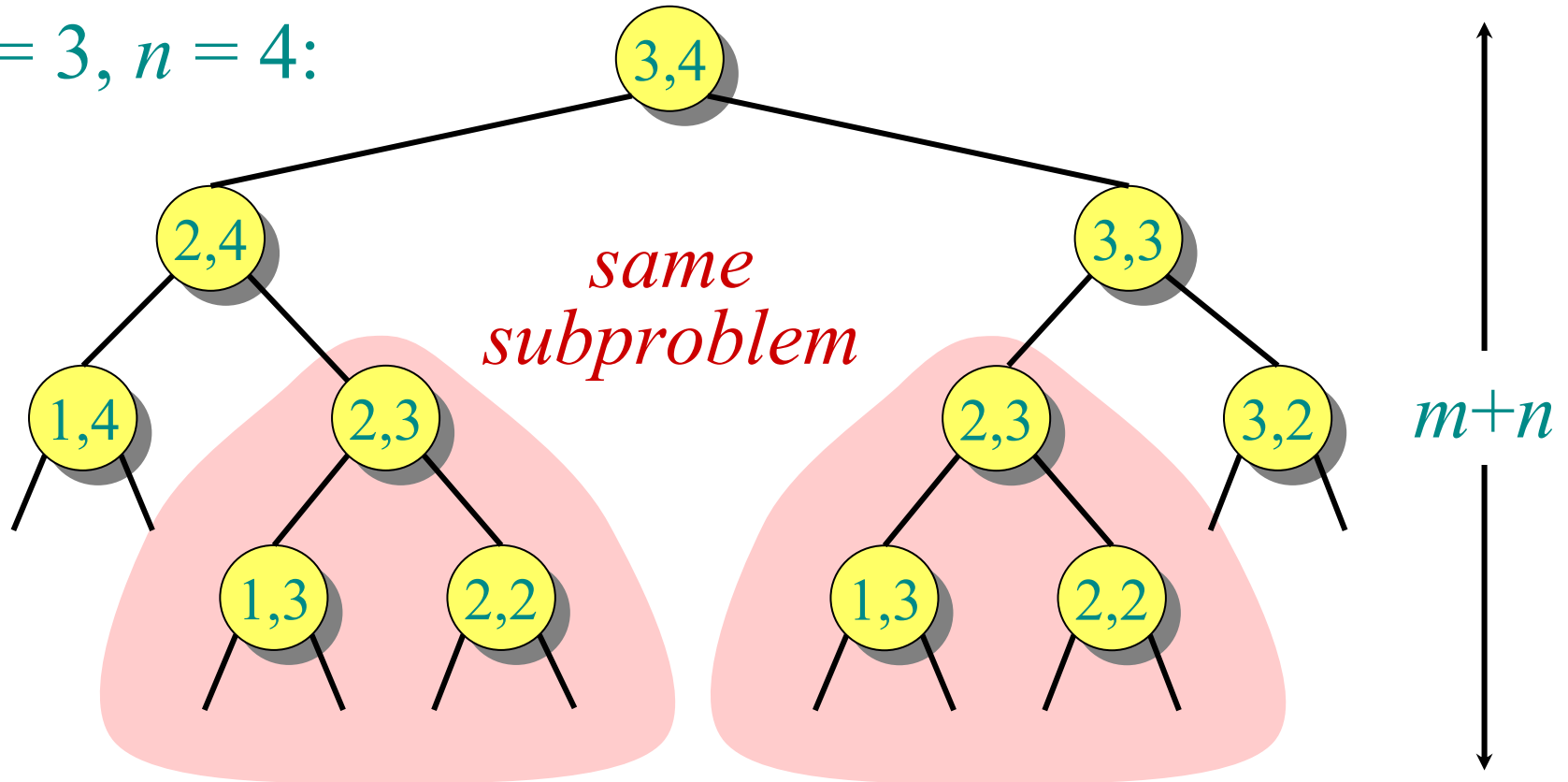
$m = 3, n = 4$:



Height = $m + n \Rightarrow$ work potentially exponential.

Recursion tree

$m = 3, n = 4$:



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.*

Dynamic-programming hallmark #2

Overlapping subproblems

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

The number of distinct LCS subproblems 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.

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

same as before

Memoization algorithm

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

$\text{LCS}(x, y, i, j)$

if $c[i, j] = \text{NIL}$

then if $x[i] = y[j]$

then $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$

else $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \}$

*same
as
before*

Time = $\Theta(mn)$ = constant work per table entry.

Space = $\Theta(mn)$.

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

| | | 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 |

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

| | A | B | C | B | D | A | B |
|---|---|---|---|---|---|---|---|
| B | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 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 algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

Space = $\Theta(mn)$.

Exercise:

$O(\min\{m, n\})$.

| | A | B | C | B | D | A | B |
|---|---|---|---|---|---|---|---|
| B | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| D | 0 | 0 | 1 | 1 | 2 | 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 |