Lecture 26: Sequence Alignment and Shortest Paths

COSC 311 *Algorithms*, Fall 2022

Overview

1. Sequence Alignment
2. Shortest Paths, Revisited

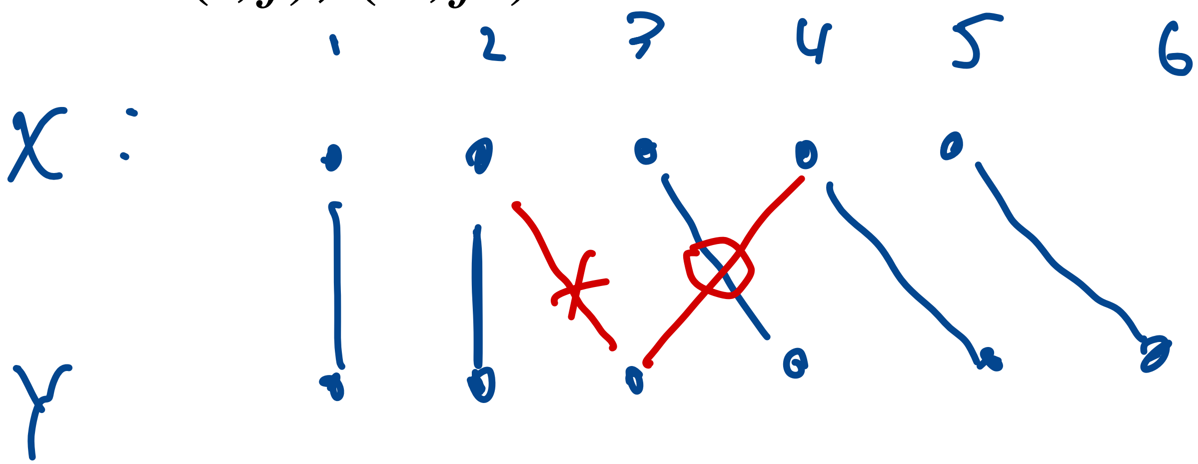
Matching Between Strings

Given strings X and Y form a *matching* between characters

- matching M is a set of pairs of matched indices

Rules for matching:

- each character is matched with at most one other character
  - some characters may be unmatched
- matched characters cannot "cross"
  - if $(i, j), (i', j')$ are matched with $i < i'$, then $j < j'$



$M: (1,1), (2,2), (3,4), (4,5), (5,6)$

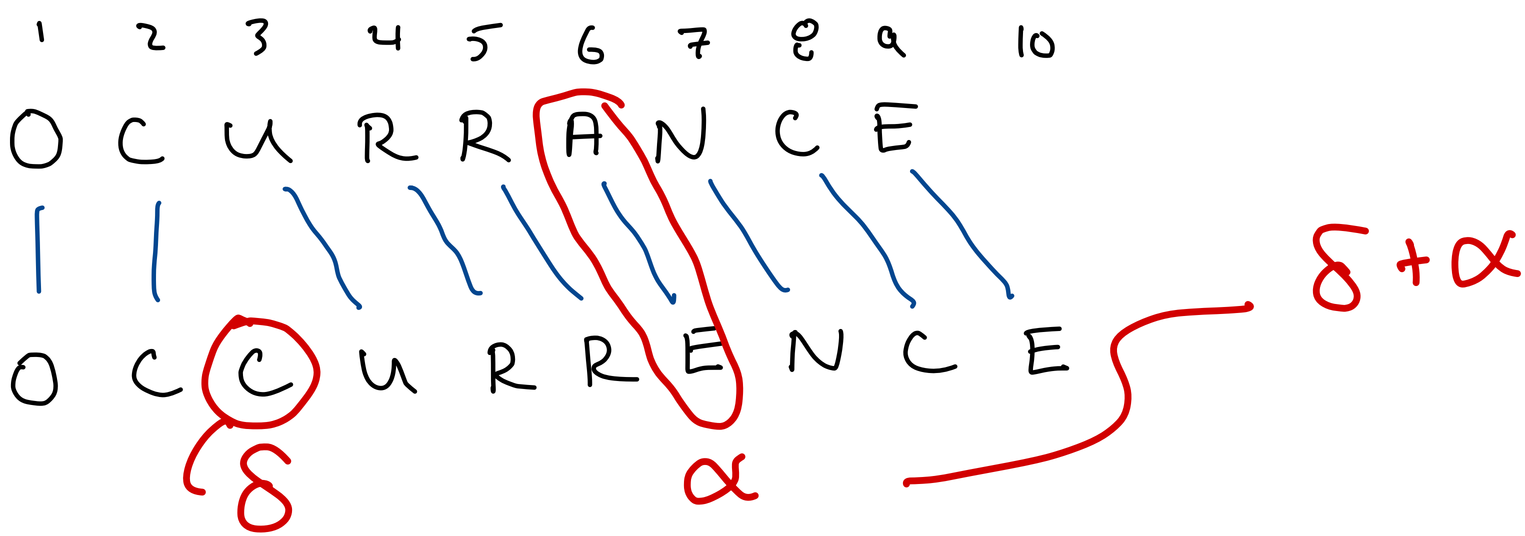
Sequence Alignment Problem

Input:

- Sequences X and Y of characters of length n and m, respectively
- Penalties δ, α for omission/mismatch

Output:

- A matching M between indices of X and Y
- M minimizes total penalty of matching



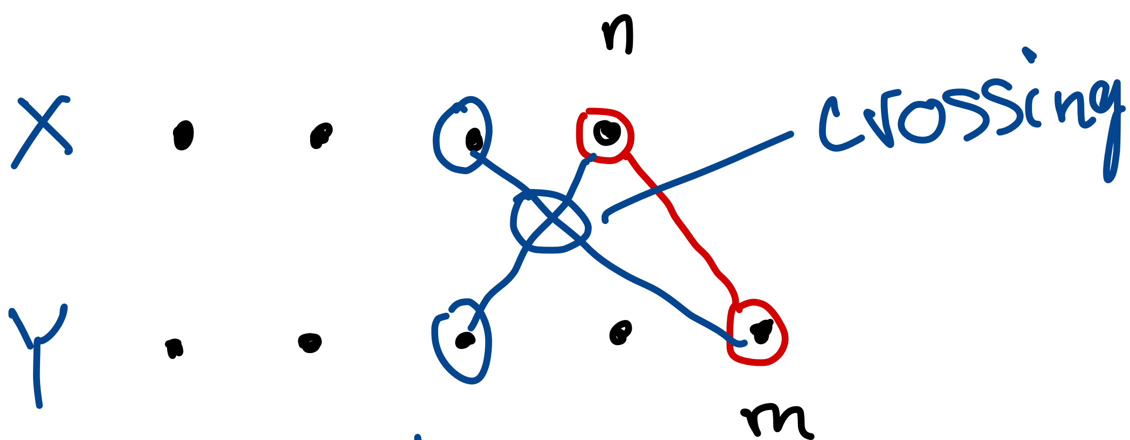
An Observation

Suppose

- X sequence of length n
- Y sequence of length m
- M a matching between $[1, n]$ and $[1, m]$

Claim. Then at least one of the following holds:

1. (n, m) is in M
2. n is unmatched in M
3. m is unmatched in M



Why?

Excluded possibility:
n, m matched, but not w/ each other
(If so, they would cross.)

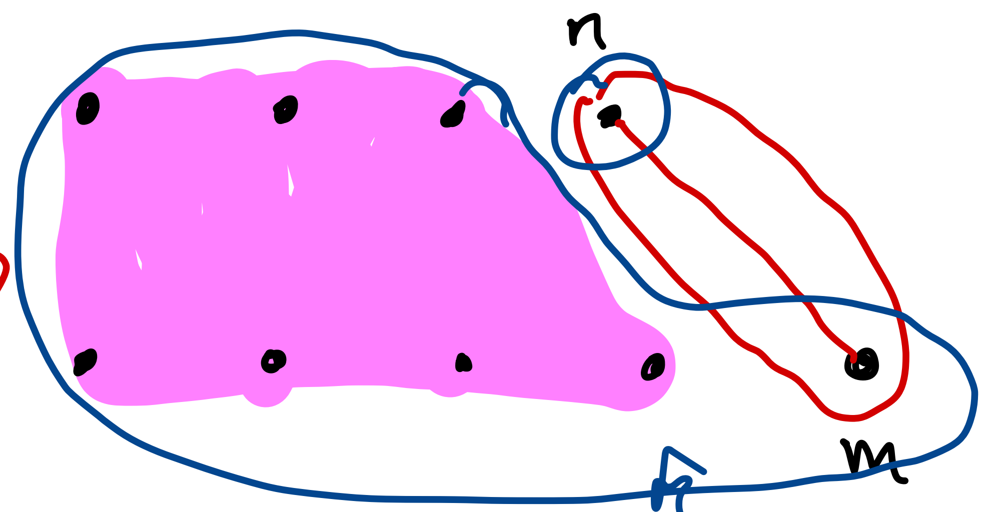
A Recursive Solution?

Idea. Use previous claim to give recursive characterization of optimal alignment.

How?

① n, m are matched

find opt sol'n here (recursive)



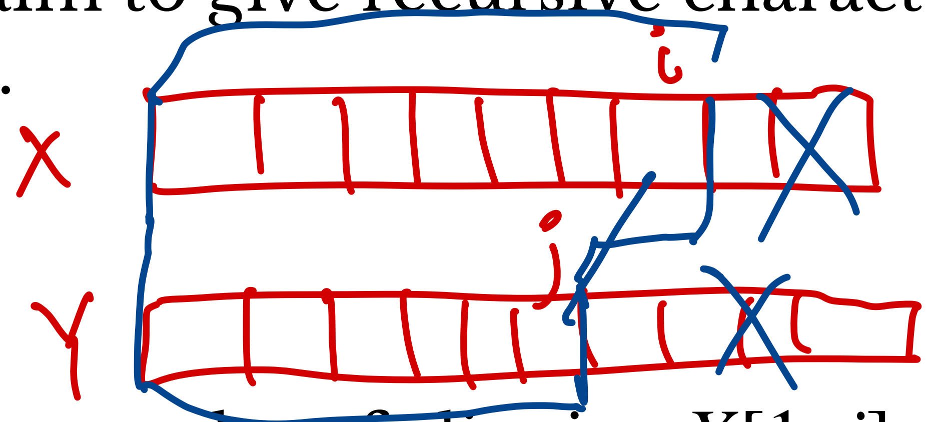
② n unmatched

remove n, find opt matching in remaining chars

③ m unmatched — same procedure

A Recursive Solution?

Idea. Use previous claim to give recursive characterization of optimal alignment.



How?

Define

- $\text{opt}(i, j)$ = minimum penalty of aligning $X[1..i]$ and $Y[1..j]$
- $M_{i,j}$ is minimum penalty matching between $X[1..i]$ and $Y[1..j]$
- by claim, there are three cases
  1. $(i, j) \in M_{i,j}$
  2. i unmatched in $M_{i,j}$
  3. j unmatched in $M_{i,j}$

Recursive Solution?

Question. What is a recurrence relation for opt(i, j)?

$$\text{opt}(i,j) = \text{Min}\begin{cases} \text{opt}(i-1, j-1) + \begin{cases}\alpha & \text{if } X[i] \neq Y[j] \\ 0 & \text{otherwise}\end{cases} \\ \text{opt}(i-1, j) + \delta \\ \text{opt}(i, j-1) + \delta \end{cases}$$

optimal value if (i, j) matched

opt value if i unmatched

opt value if j unmatched

penalty for not matching i

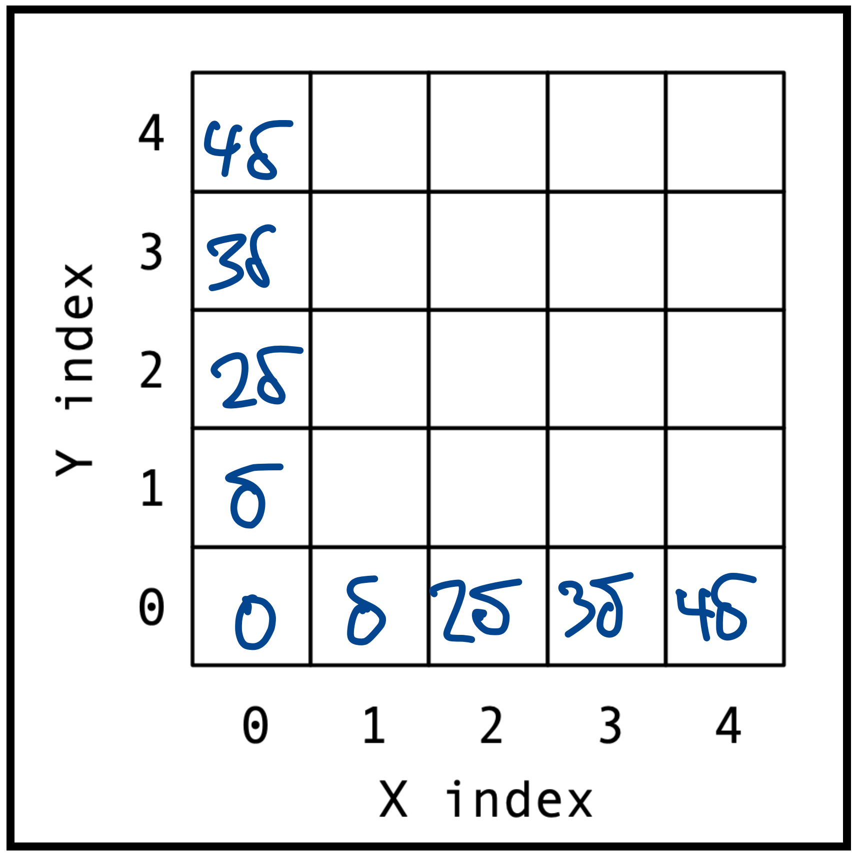
j

Iterative Solution

Construct a two dimensional array `p[0..n, 0..m]`

- `p[i, j]` should store opt(i, j)

Question 1. How to initialize `p`?



Iterative Solution

Construct a two dimensional array `p[0..n, 0..m]`

- `p[i, j]` should store opt(i, j)

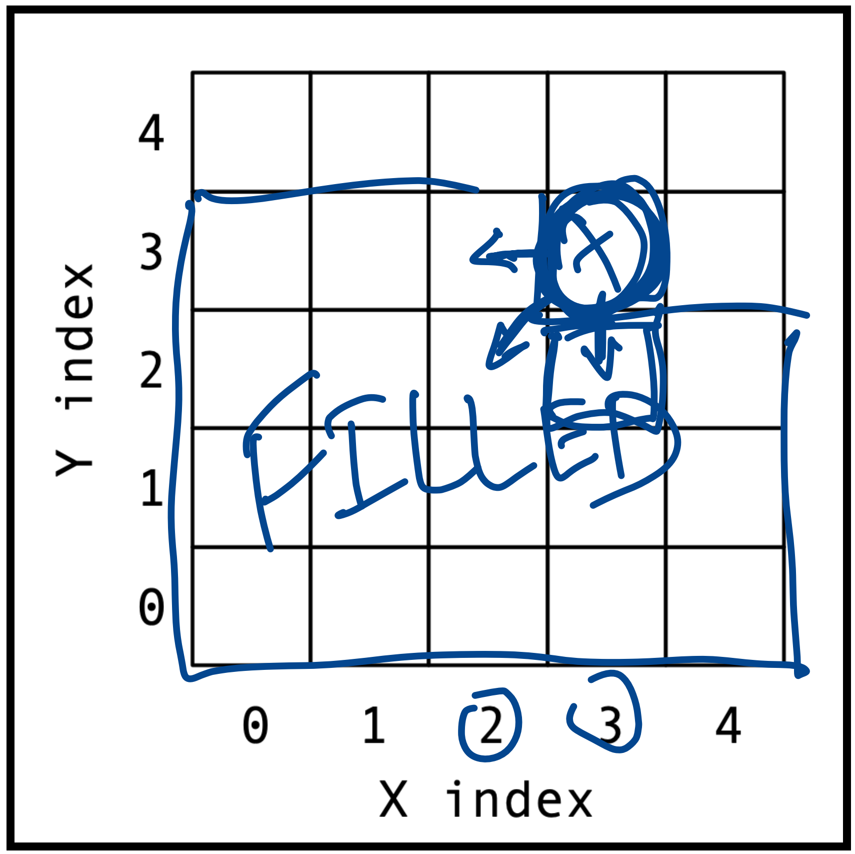
Question 1. How to initialize `p`?

Question 2. How to fill out `p`?

$\downarrow$ = omit Y[j] + δ for omission

$\leftarrow$ = omit X[i] + δ for mission

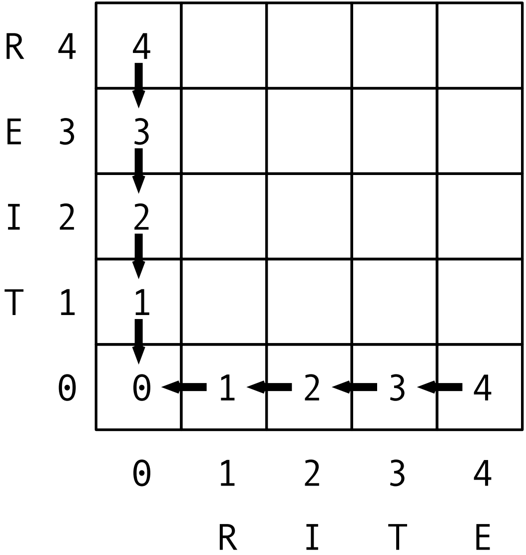
$\swarrow$ = match(i, j) + α if mismatch



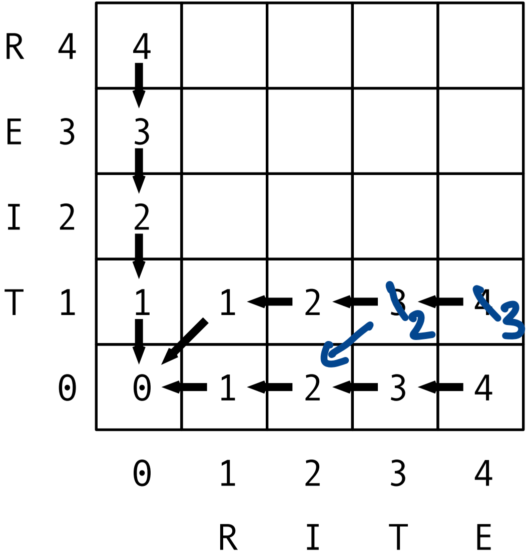
Example

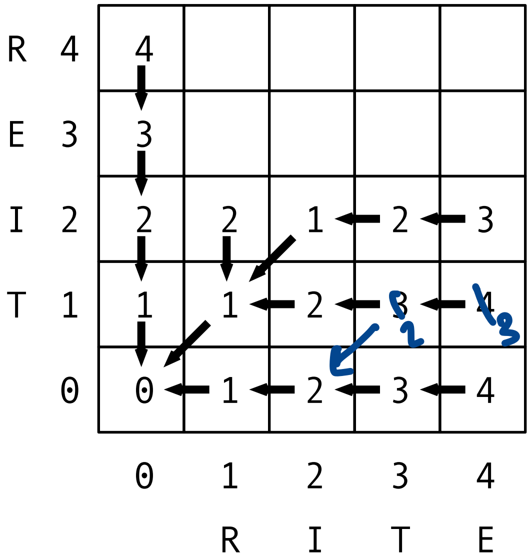
- $X = [R, I, T, E]$
- $Y = [T, I, E, R]$
- $\delta = \alpha = 1$

R	4	4				
E	3	3				
I	2	2				
T	1	1	1	2	2	3
	0	0	1	2	3	4
		0	1	2	3	4
			R	I	T	E

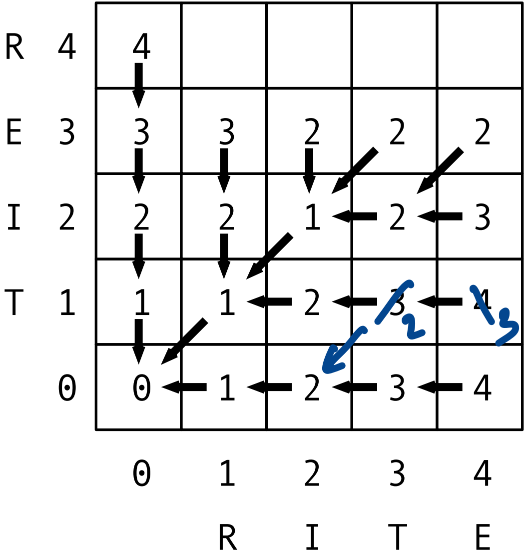


R 4 4
E 3 3
I 2 2
T 1 1
0 0 1 2 3 4
0 1 2 3 4
R I T E

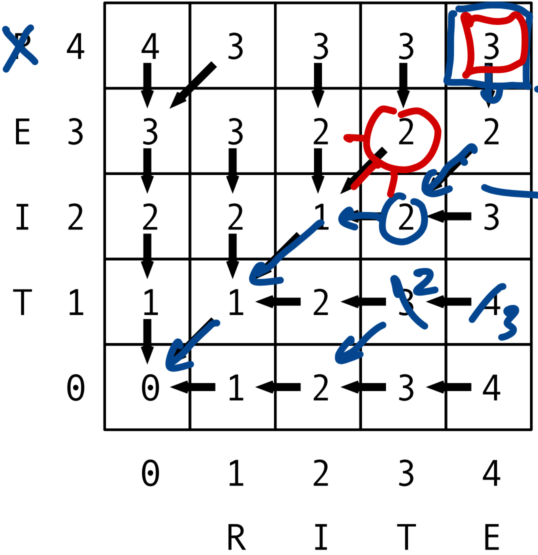

R 4 4
E 3 3
I 2 2
T 1 1 1 2 3 4
0 0 1 2 3 4
2
3
0 1 2 3 4
R I T E



R 4 4
E 3 3
I 2 2 2 1 2 3
T 1 1 1 2 3 4
0 0 1 2 3 4
0 1 2 3 4
R I T E
2
2



R 4 4
E 3 3 3 2 2 2
I 2 2 2 1 2 3
T 1 1 1 2 3 4
0 0 1 2 3 4
0 1 2 3 4
R I T E
2
3



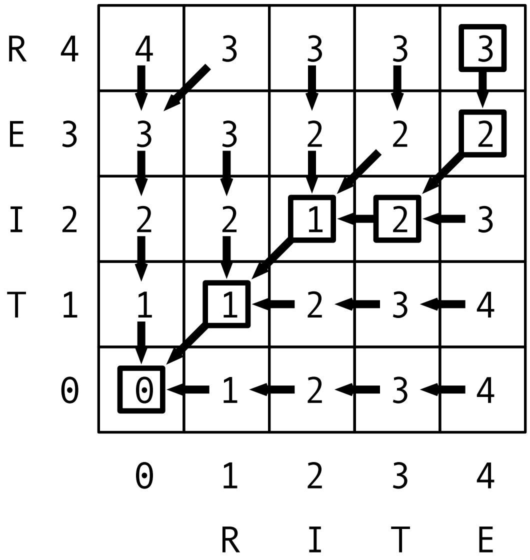
Omit R from Y

Match E, E

Omit T from X

Match I, I

Match R, T


R 4 4 3 3 3 3
E 3 3 3 2 2 2
I 2 2 2 1 2 3
T 1 1 1 2 3 4
0 0 1 2 3 4
0 1 2 3 4
R I T E

Algorithm Pseudocode

```
Alignment(X, Y, a, d):
  p <- 2d array of dimension (n+1) x (m+1)
  for i from 0 to n, p[i, 0] <- i * d
  for j from 0 to m, p[0, j] <- j * d
  for i from 1 to n
    for j from 1 to m
      unmatchX <- p[i-1, j] + d
      unmatchY <- p[i,j-1] + d
      match <- p[i-1,j-1]
      if X[i] != Y[j] then match <- match + a
      p[i, j] <- Min(unmatchX, unmatchY, match)
  return p[n, m]
```

n*m iter

Running time?

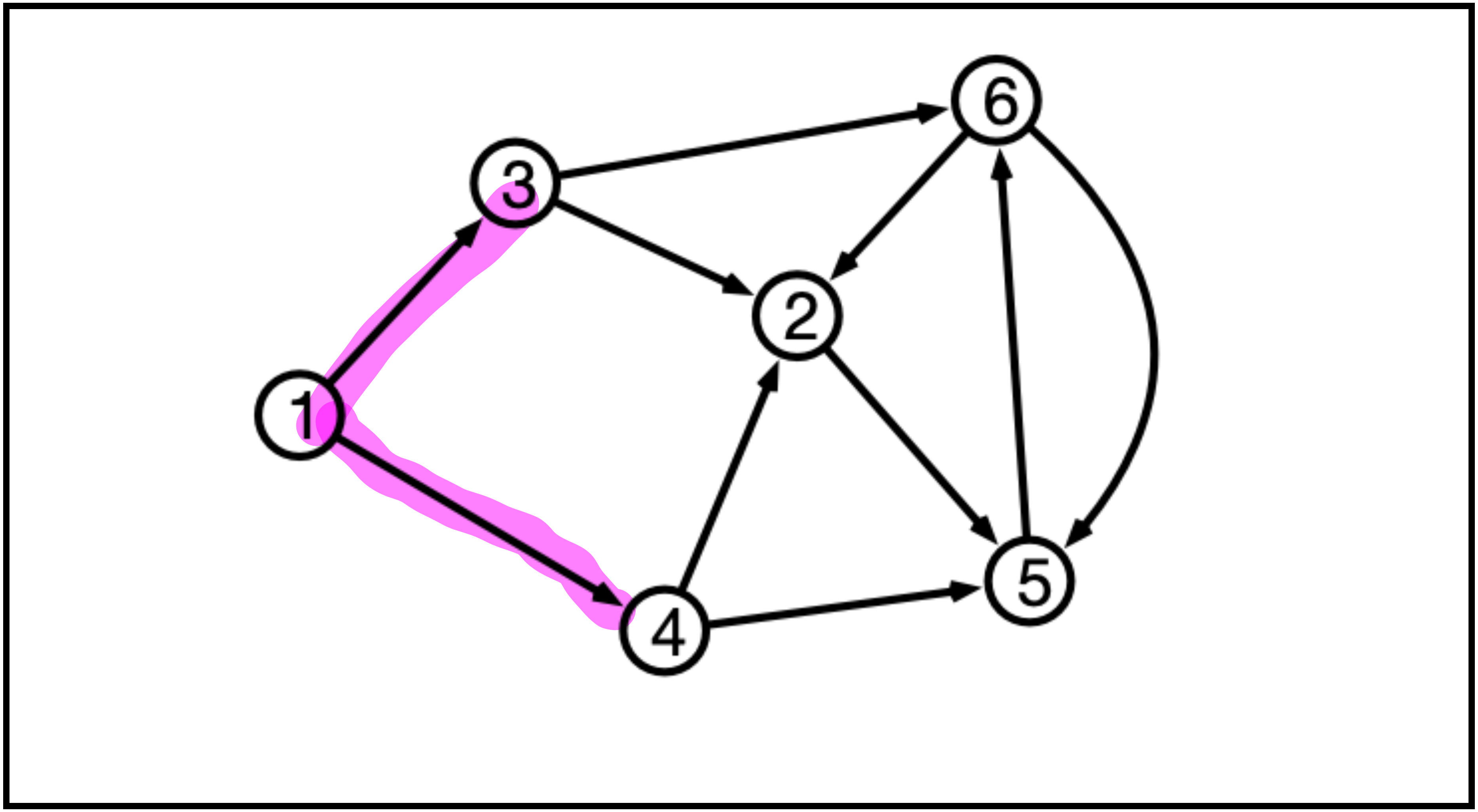
$O(n \cdot m)$

Conclusion

Optimal alignment between strings can be found in $O(nm)$ time where strings have lengths n and m, respectively.

Shortest Paths, Revisited

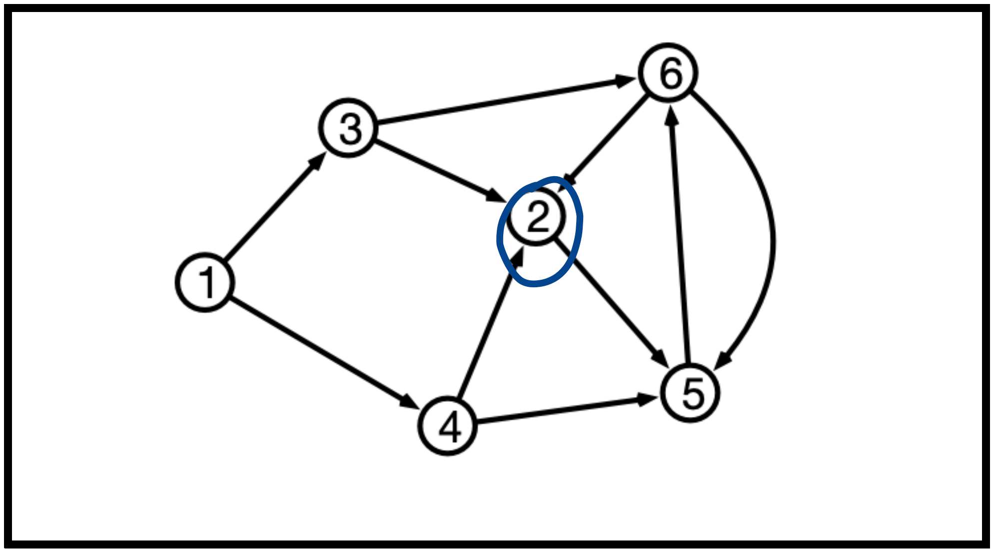
Directed Graphs and Paths



Representing Directed Graphs

Adjacency List

- *v*'s neighbors are *outgoing* neighbors



1: 3, 4
2: 5
3: 2, 6
4: 2, 5
5: 6
6: 2, 5

Previously

Single Source Shortest Paths (SSSP):

Input:

- (Directed) graph $G = (V, E)$, edge weights w
- Starting vertex u

Output:

- $d(v)$ = distance from u to v for every vertex v

Previous Algorithms

1. Breadth-first Search (BFS)
   - solves SSSP when all edge weights are 1
2. Dijkstra's Algorithm
   - solves SSSP when all edge weights are ≥ 0

Previous Algorithms

1. Breadth-first Search (BFS)
   - solves SSSP when all edge weights are 1
2. Dijkstra's Algorithm
   - solves SSSP when all edge weights are ≥ 0

Question. What if edge weights can be negative?