Portland State University

W'21 CS 584/684
Algorithm Design & Analysis

Fang Song

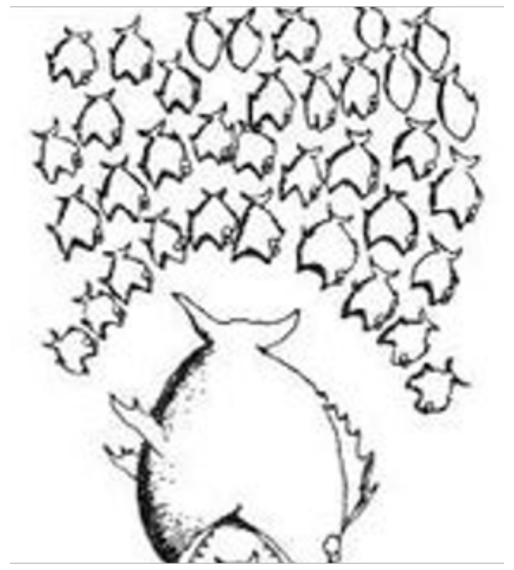
Lecture 8

- Longest common subsequence
- Bellman-Ford algorithm

Credit: based on slides by K. Wayne

Essence of dynamic programming

Top-down



Credit: Mary Wootters

- DP is about smart recursion (i.e. without repetition) by momoization.
- Usually easy to express by building up a table iteratively.

Bottom up

A recipe for DP

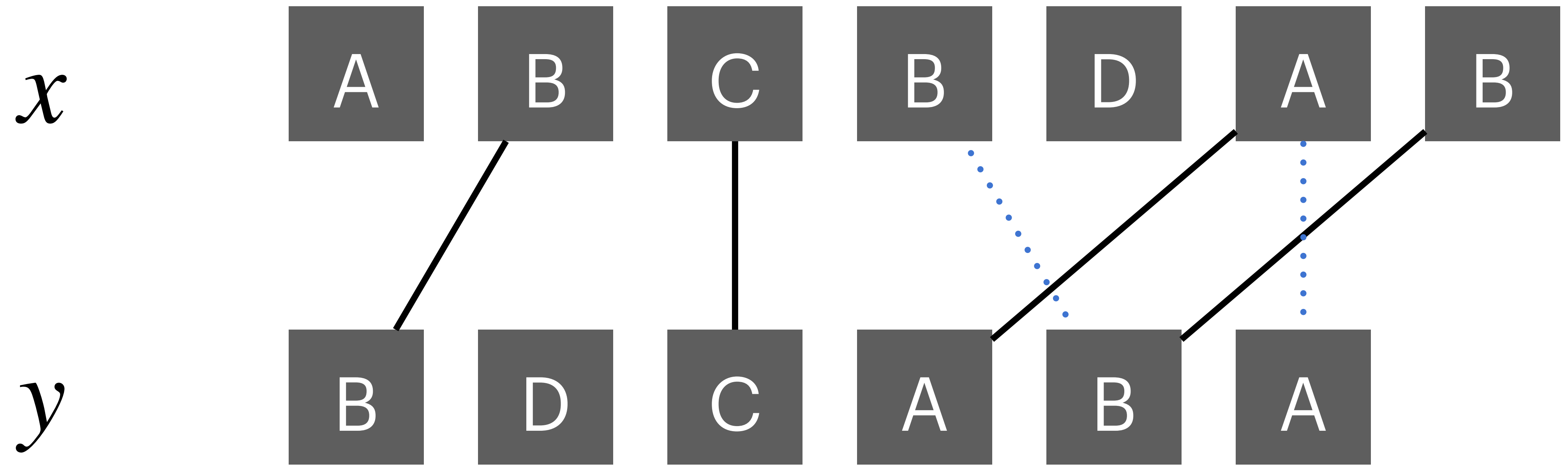
1. **Formulate the problem recursively (key step).**
   a. Specification. Describe what problems to solve (not how).
   b. Recursion. Give a recursive formula for the whole problem in terms of answers to smaller instances of the same problem.
   c. Step back and double check.
2. **Build solutions to your recurrence (kinda routine).**
   a. Identify subproblems.
   b. Choose a memoization data structure.
   c. Identify dependencies and find a good order (DAG in topological order).
   d. Write down your algorithm.
   e. Analyze time. Find possible improvement if possible.

We usually go with bottom-up approach in this class.

Longest common subsequence (LCS)

Input: two sequences $x[1,\ldots,m]$ and $y[1,\ldots,n]$

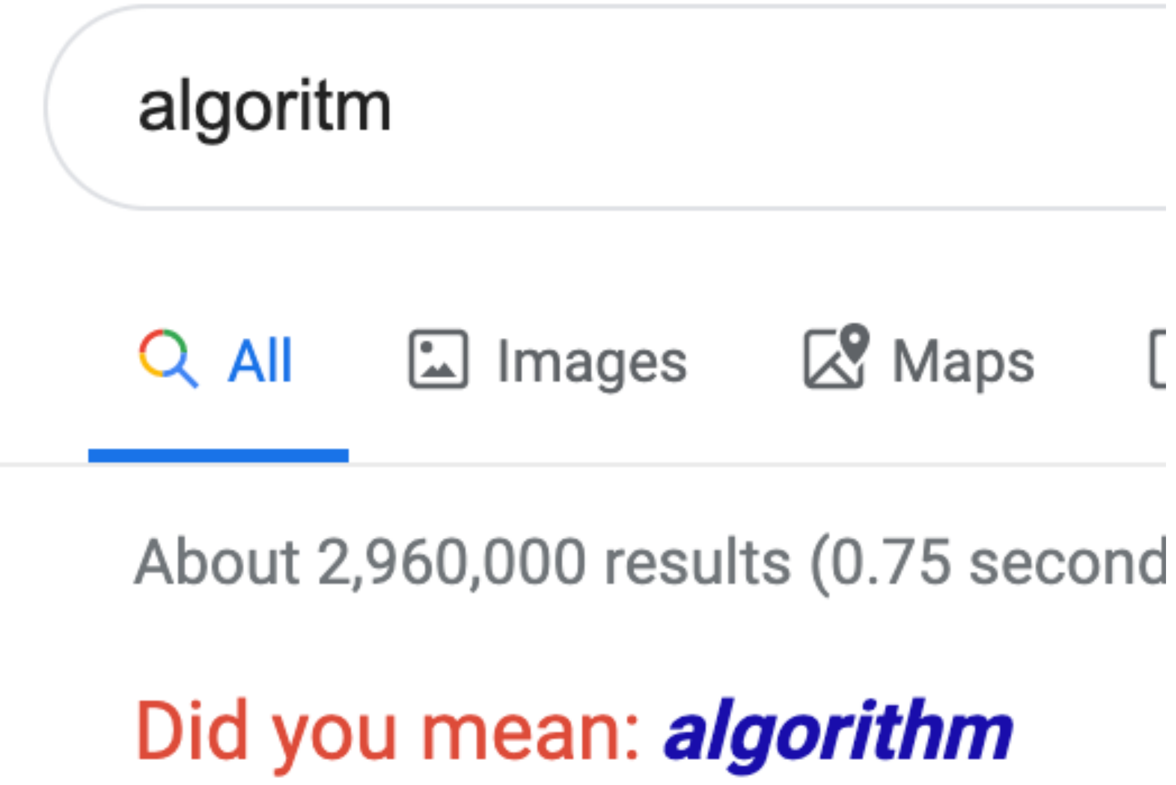
Output: A longest subsequence common to both.



- **Other names you may've heard of**
  - Sequence alignment
  - Edit distance: $n - length(LCS(x, y))$

Motivation

- **String matching** [Levenshtein 1965]
  - Auto corrector
  - Spell checker
  - Speech recognition
  - Machine translation



- **Computational biology** [Needleman-Wunsch, 1970's]
  - Simple measure of genome similarity

ACGTACGTACGTACGTACGTACGTATCGTACGT
AACGTACGTACGTACGTACGTACGTACGTACGT

ACGTACGTACGTACGTACGTACGTA T CGTACGT
AACGTACGTACGTACGTACGTACGTA CGTACGT

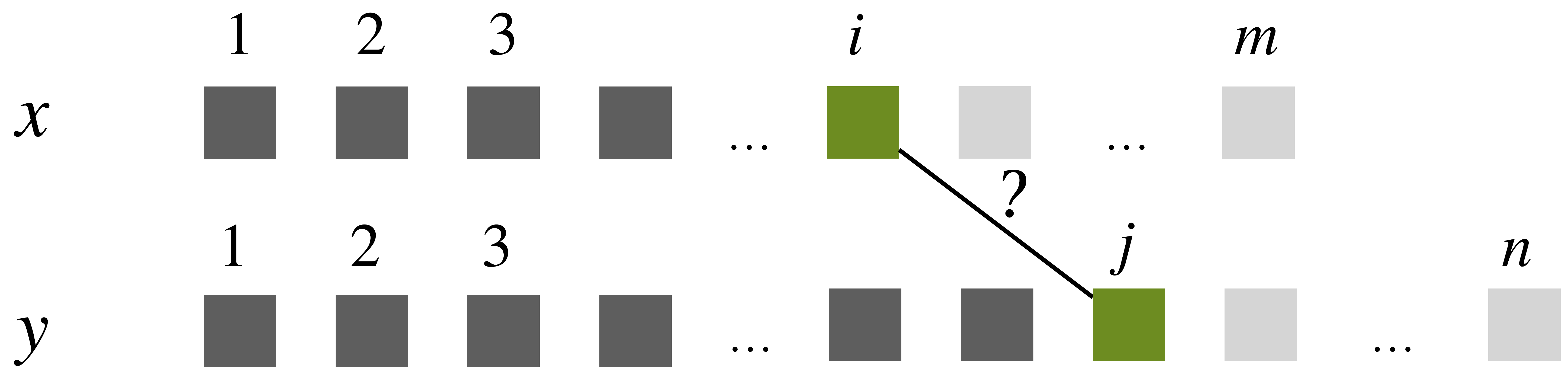
DP1: develop a recurrence

- **Simplification: look at the length of a longest-common subsequence**
  - Extend the algorithm to find the LCS itself

- **1.a Specification. What problems to solve?**
  - Definition. $c(i,j) := |LCS(x[1,\ldots,i], y[1,\ldots,j])|$. $|s|$: length of string s
  - Goal. Find $c(m,n)$.

- **1.b Recursion. Recurrence to solve an subproblems from smaller ones.**
  - Base. $c(i,j) = 0$, if $i = 0$ or $j = 0$.
  - How to compute $c(i,j)$ recursively?

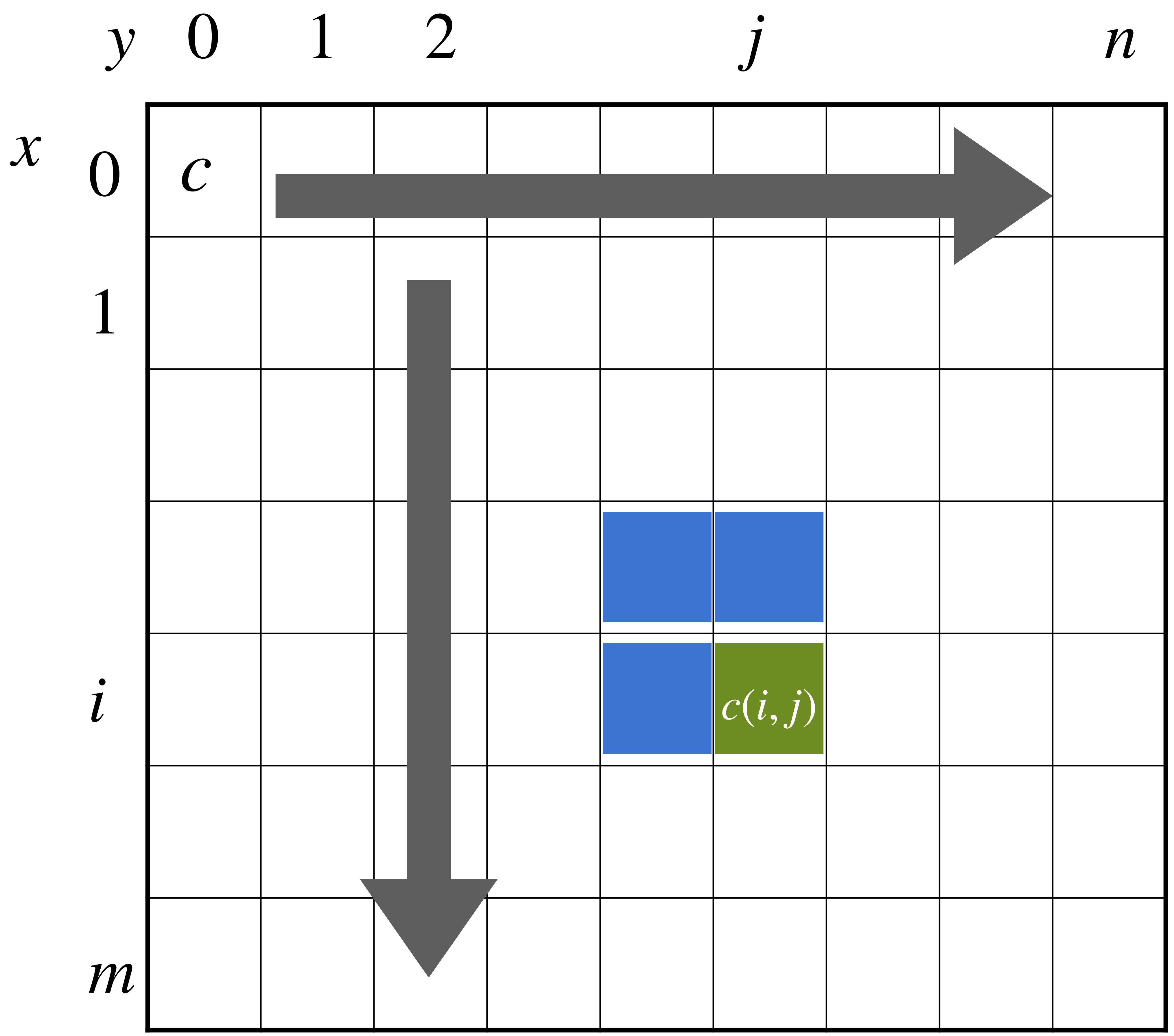
DP1: develop a recurrence, cont'd



- **Case 1.** $x[i] = y[j]$. $c(i,j) = c(i-1, j-1) + 1$
- **Case 2.** $x[i] \neq y[j]$. $c(i,j) = \max\{c[i-1, j], c[i, j-1]\}$

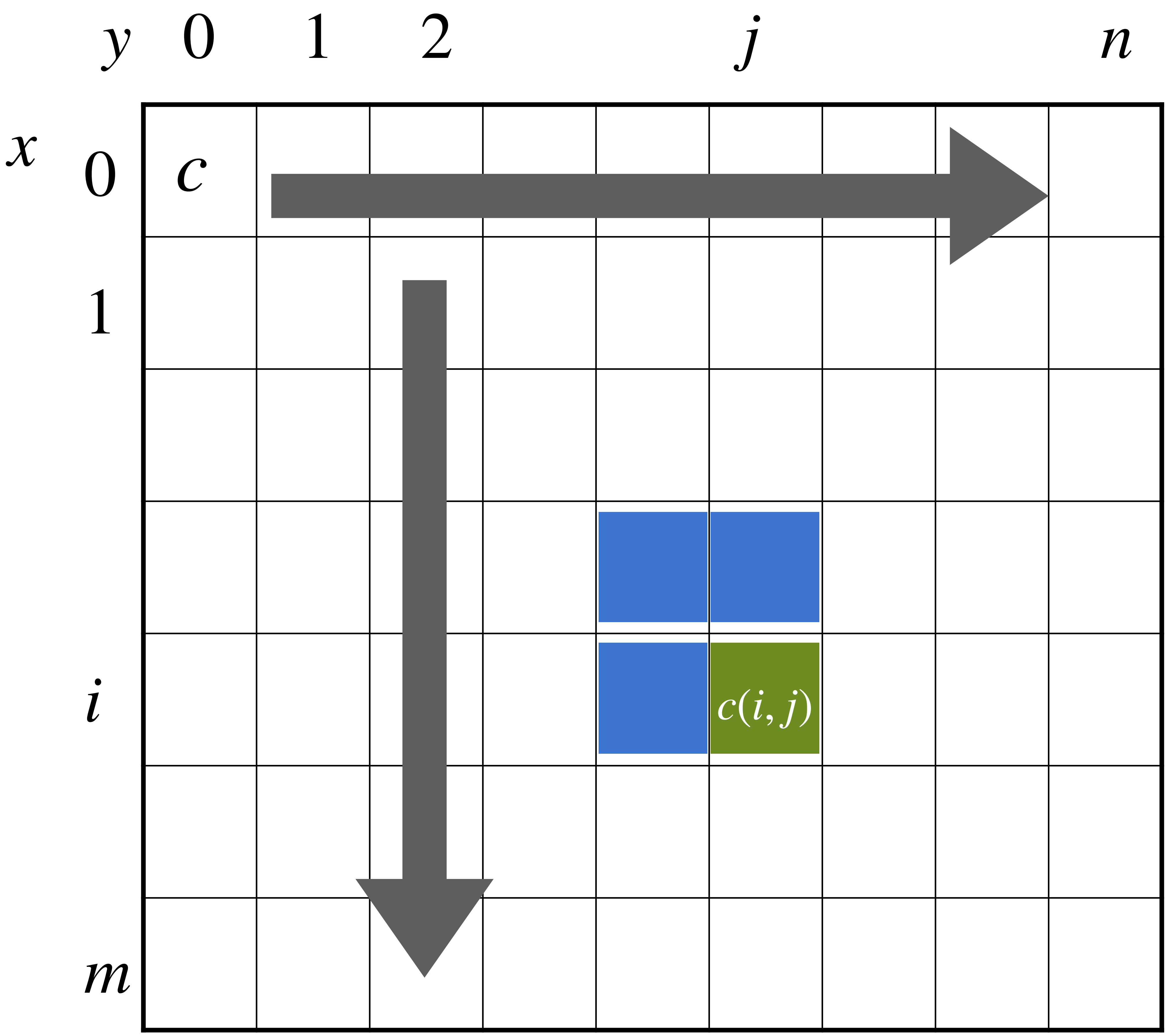
$$c(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ c(i-1, j-1) + 1, & \text{if } x[i] = y[j] \\ \max\{c[i-1, j], c[i, j-1]\}, & \text{if } x[i] \neq y[j] \end{cases}$$

DP2: build up solutions



- **Subproblems.** $O(mn)$
- **Memoization data structure**
  - 2-D array $c[0,\ldots,m,0,\ldots,n]$
- **Dependencies**
  - Each $c(i,j)$ depends on its 3 neighbors: $c(i-1,j-1)$, $c(i,j-1)$, $c(i-1,j)$.
- **Evaluation order**
  - Left-to-right, row by row

DP2: build up solutions, cont'd




```
LCSLen(x[1,…,m], y[1,…,n]):
// c[i, j] store subproblem values
1.  For j = 0,…,n  c[0,j] ← 0
2.  For i = 1,…,m // row by row
3.      c[i,0] ← 0
4.      For j = 1,…,n // left to right
            If x[i] = y[j]
                c(i, j) = c(i − 1,j − 1) + 1
            If x[i] = y[j]
                c(i, j) = max{c(i, j − 1), c(i − 1,j)}
5.  Return c[m, n]
```

- **Running time:** $O(mn)$.

Example

x = BDCABA

y = ABCBDAB

y		A	B	C	B	D	A	B
x	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								
A								
B								
A								4

DP3: constructing an optimal solution

x = BDCABA

y = ABCBDAB

- **Reconstruct LCS by tracing backwards**
  - $LCS(x, y) =$ BCBA
  - Multiple solutions possible.

- **Space:** $O(mn)$
  - Can you do it in $\min\{m, n\}$?
    [Hint: divide-&-conquer]

	y	A	B	C	B	D	A	B
x	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

Improved algorithms

- [MasekPatersen 1980] $O(n^2/\log n)$
- How about $O(n^{1.9999})$?

Quadratic Barrier

[BackursIndyk'STOC2015] **Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false)**

[BEG'SODA2018] Approximating Edit Distance in Truly Subquadratic Time: Quantum and MapReduce*†

[CDGKS'FOCS2018] **Approximating Edit Distance Within Constant Factor in Truly Sub-Quadratic Time**

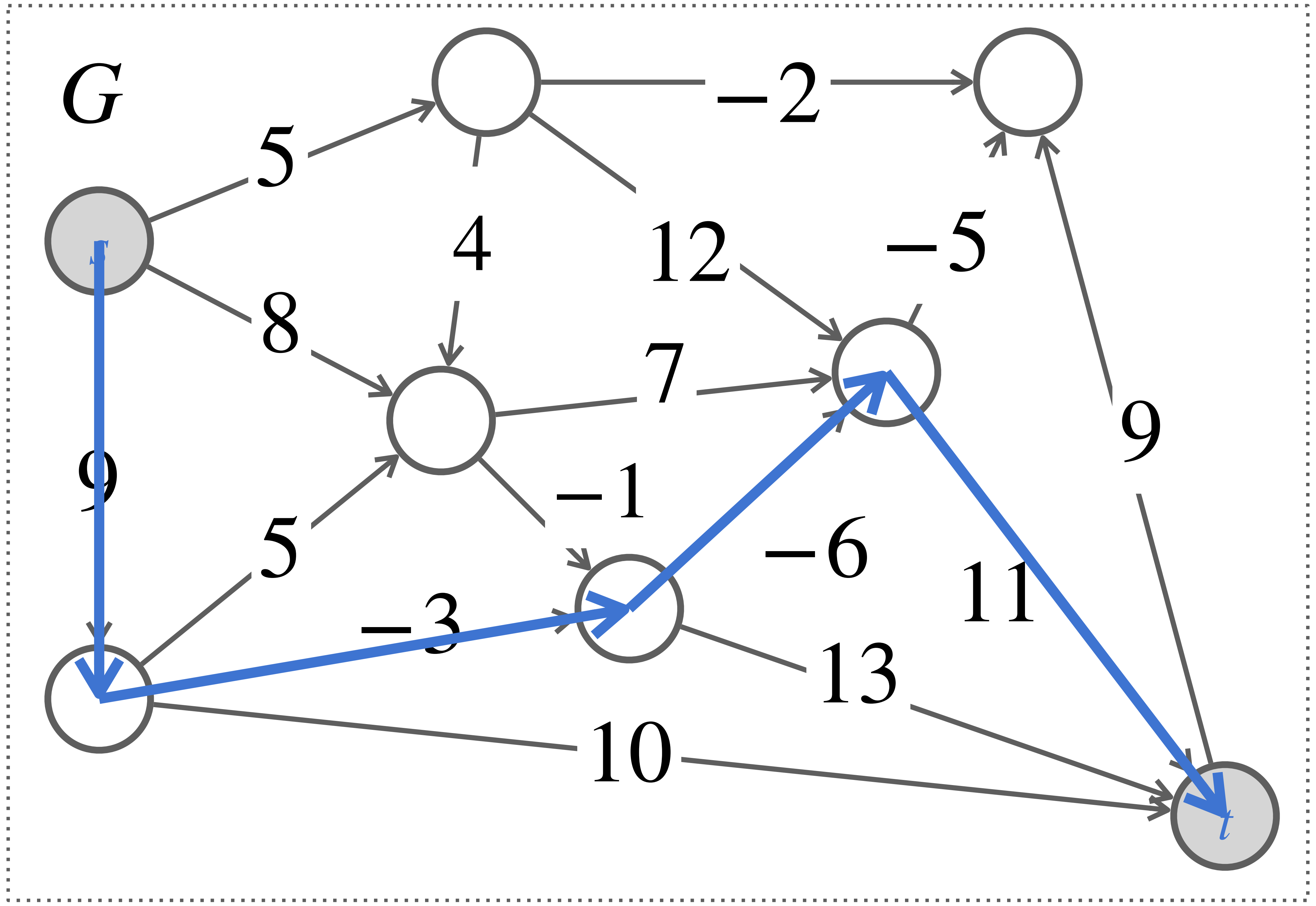
… Check STOC'20 for further improvements

Shortest path problem, revisited

Input: Graph G, nodes s and t .

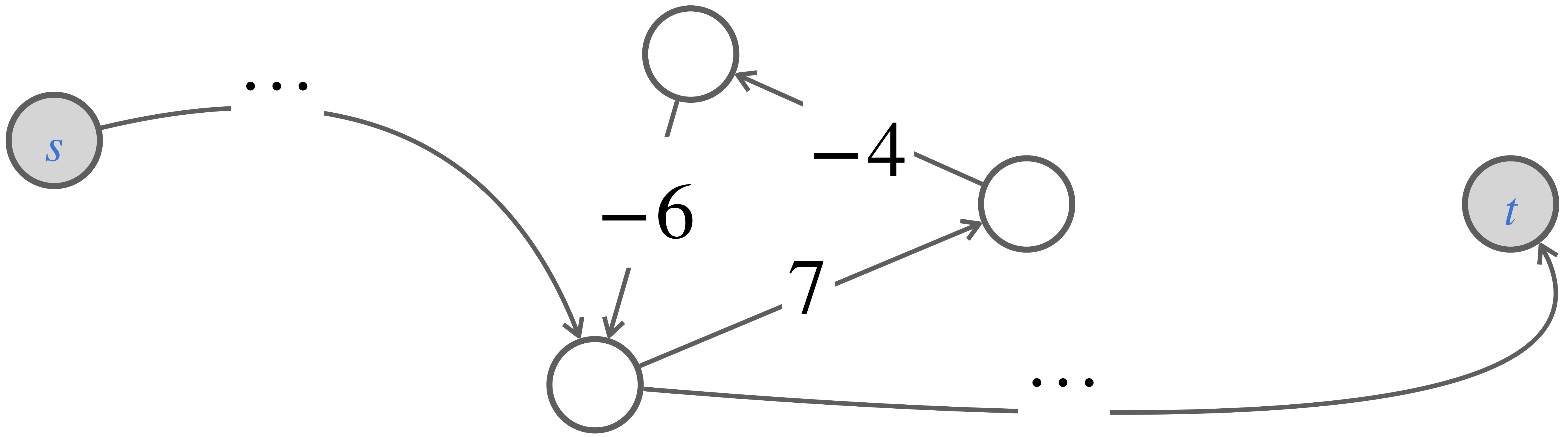
Output: $dist(s, t)$.

- Every edge has a length ℓ_e.
- Length of a path $\ell(P) = \sum_{e \in P} \ell_e$.
- Distance $dist(s, t) = \min_{P: u \rightsquigarrow v} \ell(P)$.



- **Special cases**
  - All edges have equal length: BFS $O(m + n)$.
  - DAG: DP in topological order $O(m + n)$.

A technical issue: negative length cycles



- **Observation**
  - If some $s \rightsquigarrow t$ path contains a negative length cycle, there does not exist a shortest $s \rightsquigarrow t$ path.
  - Otherwise there exists a simple (i.e., no repetition node) path $\leq n-1$ edges.

- **For simplicity, assuming G has no NegativeLengthCycle**
  - Can be detected with little overhead.

DP1: develop a recurrence

- **Simplification: look at the length of a shortest path.**

- **1.a Specification. What problems to solve?**
  - Definition. $OPT(i, v) :=$ length of shortest $v \rightsquigarrow t$ path P using $\leq i$ edges.
  - Goal. Find $OPT(n-1, s)$.

- **1.b Recursion. Recurrence to solve an subproblems from smaller ones.**
  - Base. $OPT(i, v) = 0$ or ∞ if $i = 0$.
  - How to compute $OPT(i, v)$ recursively?

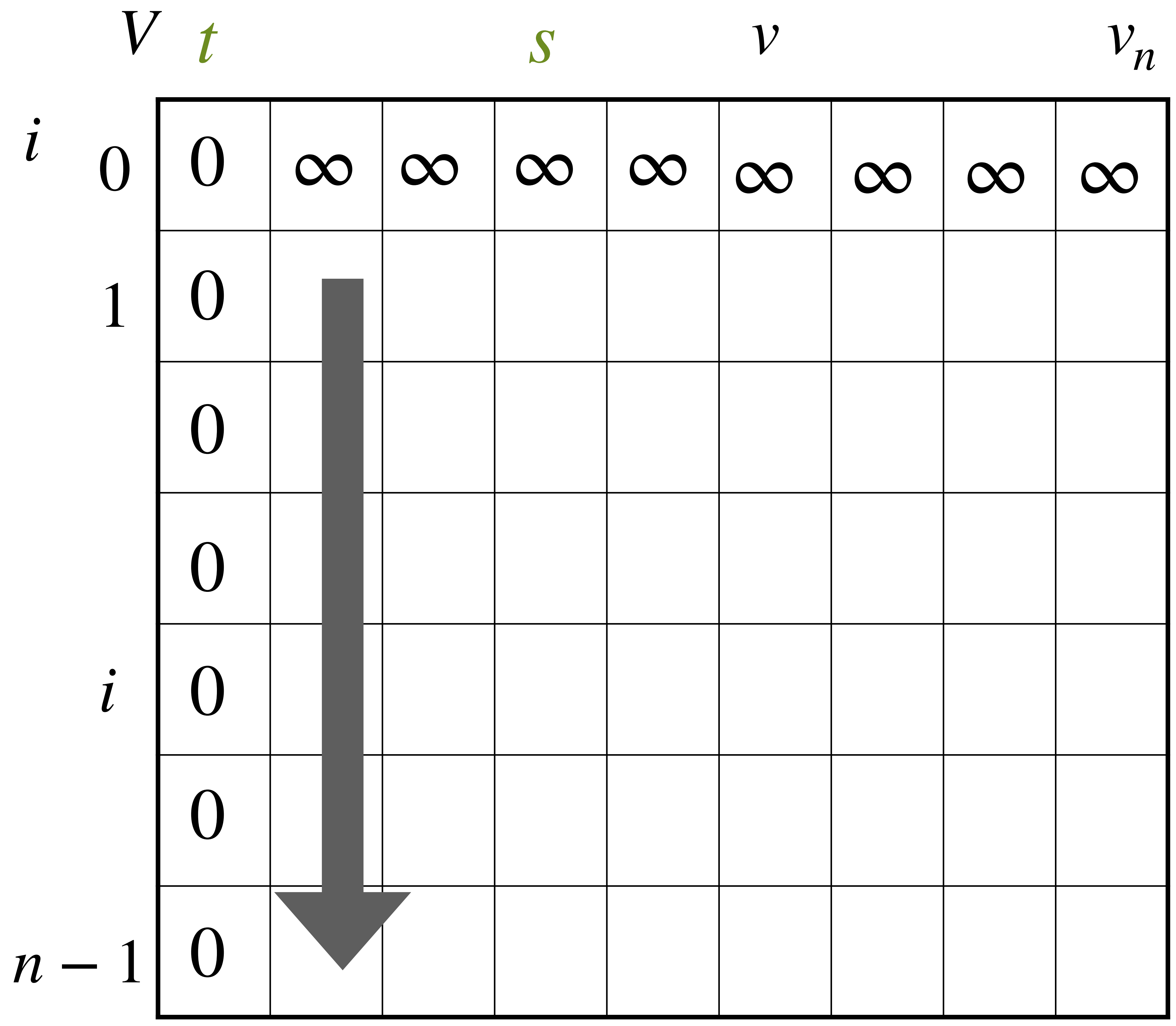
DP1: develop a recurrence, cont'd

$OPT(i, j) :=$ length of shortest $v \rightsquigarrow t$ path P using $\leq i$ edges.

- **Case 1.** P **uses at most** $i - 1$ **edges.** $OPT(i, v) = OPT(i-1, v)$
- **Case 2.** P **uses exactly** i **edges.**
  - If (v, w) is the first edge, then OPT uses (v, w) and then select best $w \rightsquigarrow t$ path using $\leq i - 1$ edges.
  - $OPT(i, v) = \min_{v \to w \in E} \{OPT(i-1, w) + \ell_{v \to w}\}$

$$OPT(i, v) = \begin{cases} 0, & \text{if } v = t \\ \infty, & \text{if } i = 0 \\ \min\{OPT(i-1, v), \min_{v \to w \in E}\{OPT(i-1, w) + \ell_{v \to w}\}\}, & \text{otherwise} \end{cases}$$

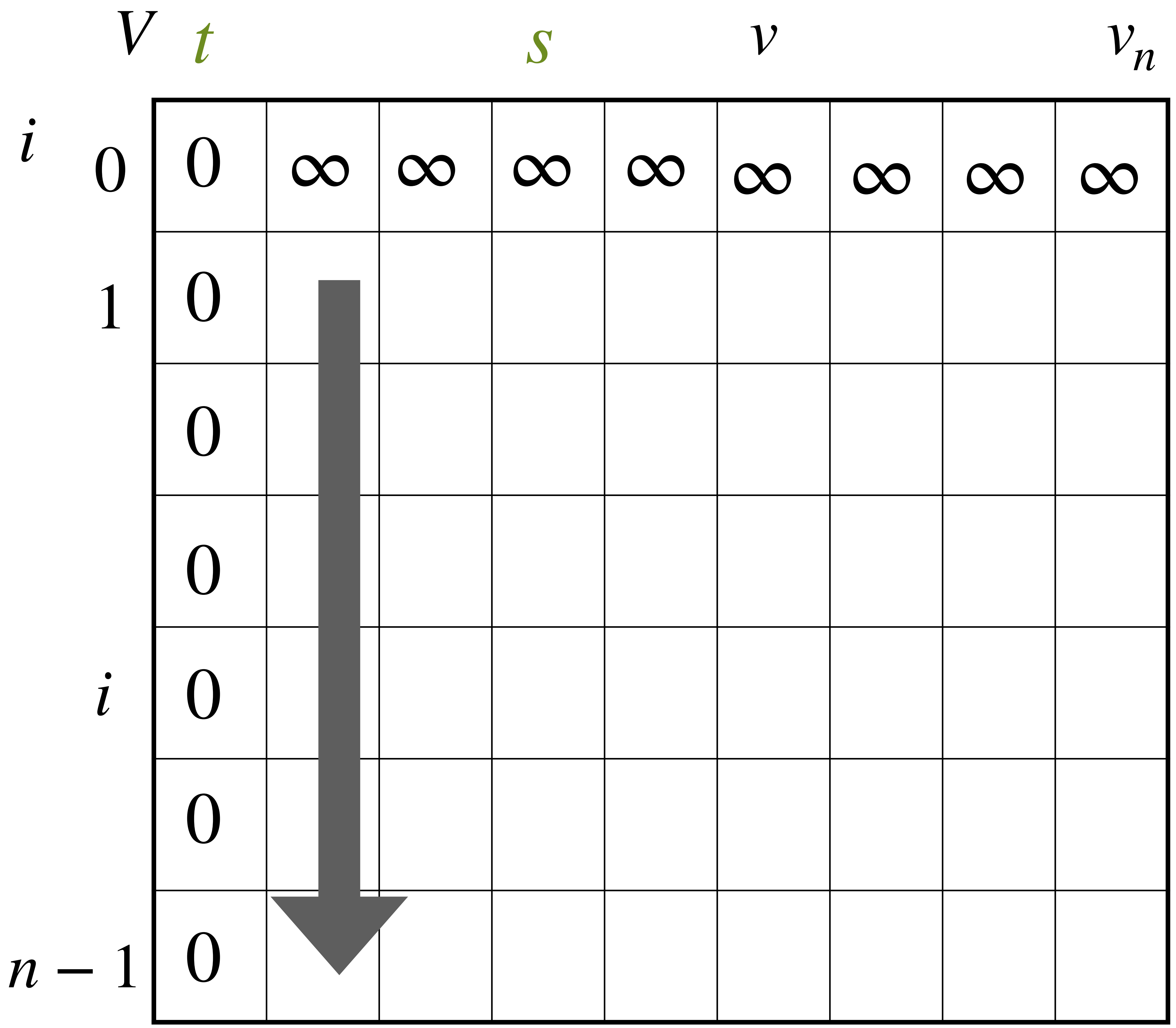
DP2: build up solutions



- **Subproblems.** $O(n^2)$
- **Memoization data structure**
  - 2-D array $M[0, \ldots, n-1, v_1, \ldots, v_n]$.
- **Dependencies**
  - Each $OPT(i, v)$ depends on subproblems in the row above.
- **Evaluation order**
  - Row by row, arbitrary within a row.

$$OPT(i, v) = \begin{cases} 0, & \text{if } v = t \\ \infty, & \text{if } i = 0 \\ \min\{OPT(i-1, v), \min_{v \to w \in E}\{OPT(i-1, w) + \ell_{v \to w}\}\}, & \text{otherwise} \end{cases}$$

DP2: build up solutions, cont'd



SPLen(G, s, t):

// $M[i, v]$ store subproblem values

// $M[0,t] = 0$, $M[0,v] = \infty$ otherwise.

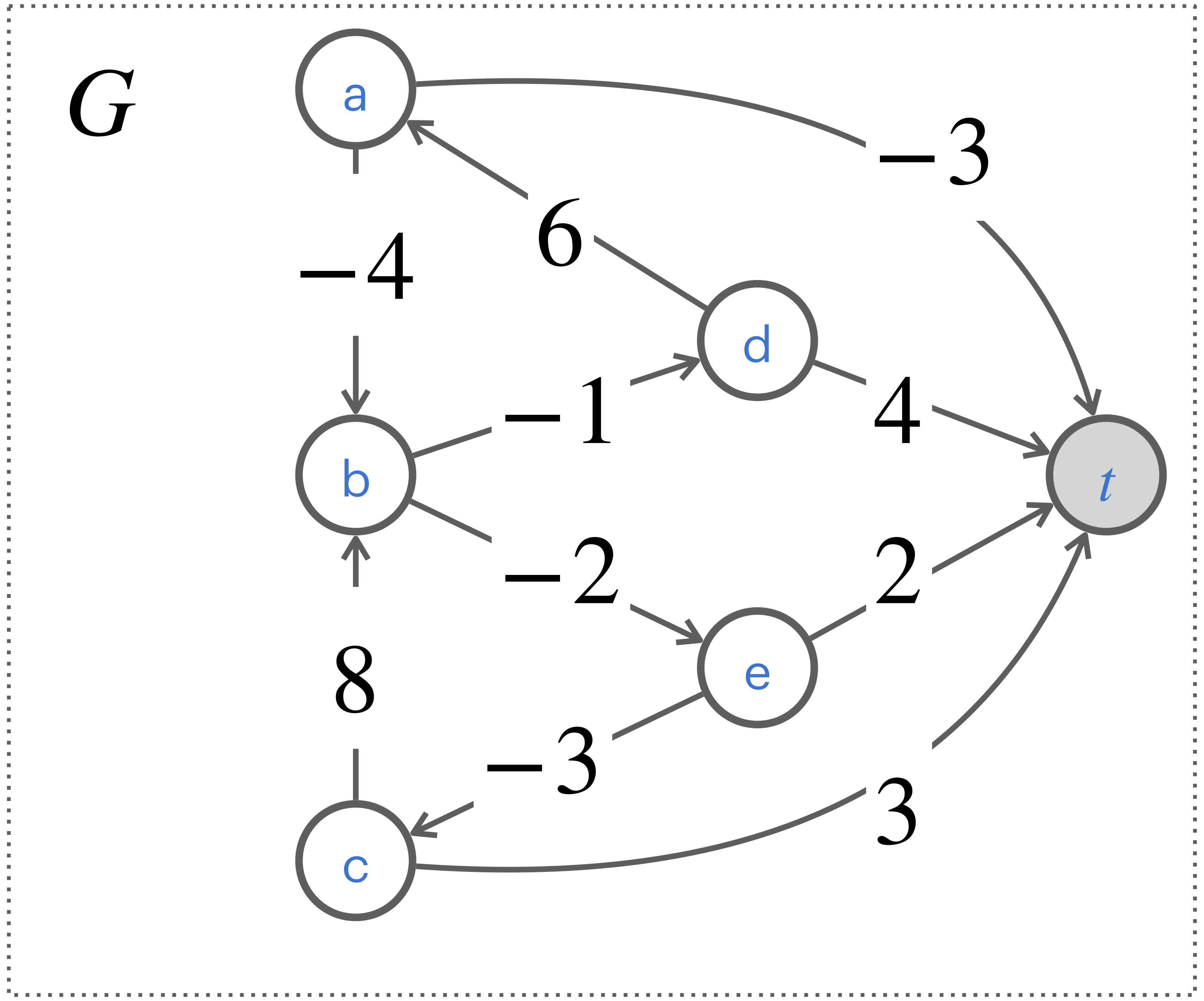
1. For $i = 1,\ldots,n-1$ // row by row
2. For $v \in V$ // arbitrary order

    $M[i, v] \leftarrow M[i-1,v]$ // case 1

    For edge $v \to w \in E$ // case 2

    $M(i, j) \leftarrow \min\{M[i, v], M[i-1,w] + \ell_{vw}\}$
3. Return $M[n-1,s]$

Example



V / i	t	A	B	C	D	E
0	0	∞	∞	∞	∞	∞
1	0					
2	0					
3	0					
4	0					
5	0					

For $v \in V$ // arbitrary order
$M[i, v] \leftarrow M[i-1, v]$ // case 1
For edge $v \rightarrow w \in E$ // case 2
$M(i, j) \leftarrow \min\{M[i, v], M[i-1, w] + \ell_{vw}\}$

Scratch