CSE 421 Winter 2025
Lecture 14: DP3

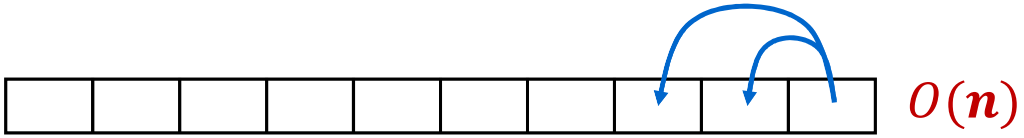
Nathan Brunelle

http://www.cs.uw.edu/421

Dynamic Programming Patterns

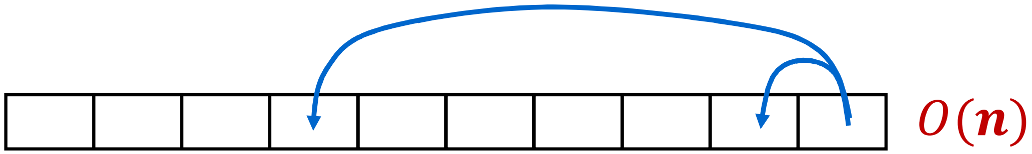
Fibonacci pattern:

- 1-D, $O(\mathbf{1})$ immediately prior
- $O(\mathbf{1})$ space



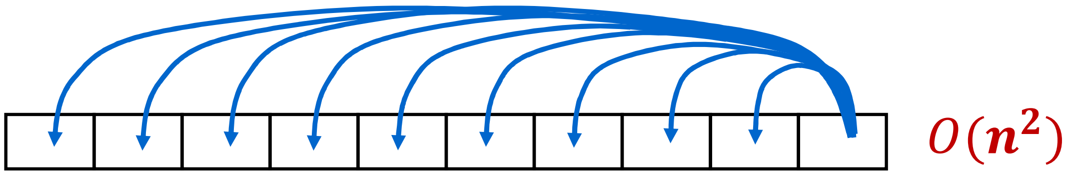
Weighted interval scheduling pattern:

- 1-D, $O(\mathbf{1})$ arbitrary prior
- $O(\boldsymbol{n})$ space



Longest increasing subsequence pattern:

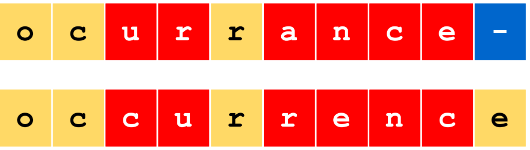
- 1-D, all $\boldsymbol{n}-\mathbf{1}$ prior
- $O(\boldsymbol{n})$ space



String Similarity

How similar are two strings?

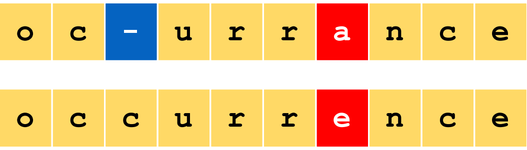
- `ocurrance`
- `occurrence`



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

6 mismatches, 1 gap

Clearly a better matching

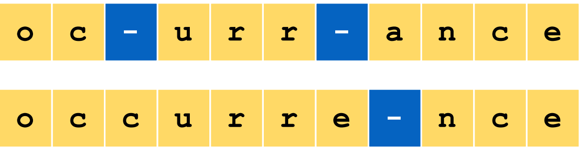


o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

Maybe a better matching

- depends on cost of gaps vs mismatches



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

0 mismatches, 3 gaps

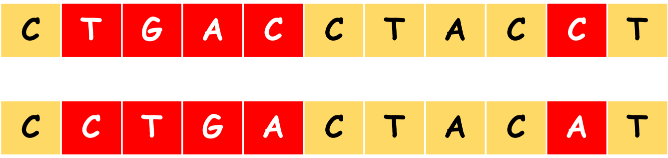
Edit Distance

Applications:

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

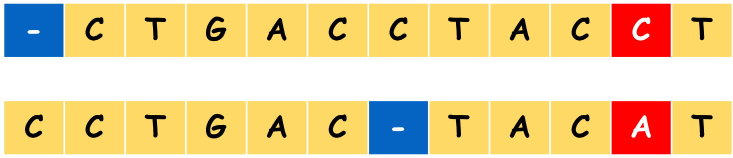
Edit distance: [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq} if symbol p is replaced by symbol q.
- **Cost** = gap penalties + 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

Sequence Alignment:

Given: Two strings $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$

Find: "Alignment" of X and Y of minimum edit cost.

Defn: An alignment M of X and Y is a set of ordered pairs x_i-y_j s.t. each symbol of X and Y occurs in at most one pair with no "crossing pairs".

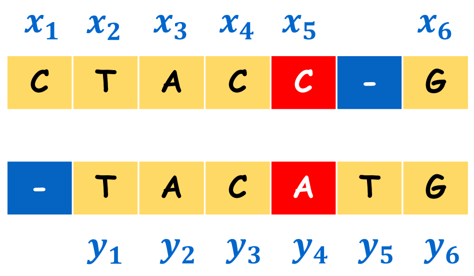
The pairs x_i-y_j and $x_{i'}$-$y_{j'}$ cross iff $i < i'$ but $j > j'$.

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

Note: if $x_i = y_j$ then $\alpha_{x_iy_j} = 0$

Example:

CTACCG vs TACATG



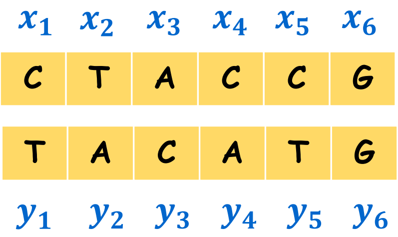
x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	-	G
-	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

$M = \{x_2\text{-}y_1, x_3\text{-}y_2, x_4\text{-}y_3, x_5\text{-}y_4, x_6\text{-}y_6\}$

Edit Distance – Four Steps

1. Formulate the answer with a recursive structure
    - What are the options for the last choice?
    - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
    - Figure out the possible values of all parameters in the recursive calls.
    - How many subproblems (options for last choice) are there?
    - What are the parameters needed to identify each?
    - How many different values could there be per parameter?
3. Specify an order of evaluation.
    - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
    - With this step: a “Bottom-up” (iterative) algorithm
    - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
    - Is it possible to reuse some memory locations?

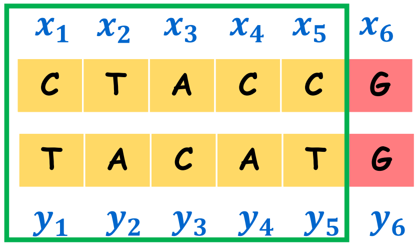
Step 1: Identify Recursive Structure



Consider the last two indices x_i and y_j

Options for what to do with them:

Option 1:
Match them

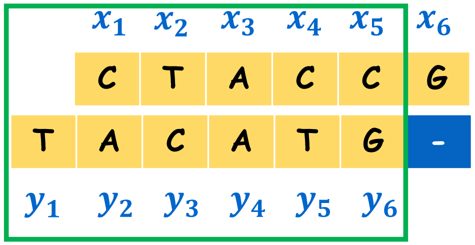


We use up one index from x and y

Accrue a mismatch penalty

$$OPT(i-1, j-1) + \alpha_{x_i y_j}$$

Option 2:
Don't match x_i

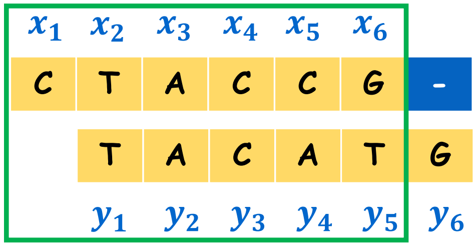


We use up one index from x only

Accrue a gap penalty

$$OPT(i-1, j) + \delta$$

Option 3:
Don't match y_i



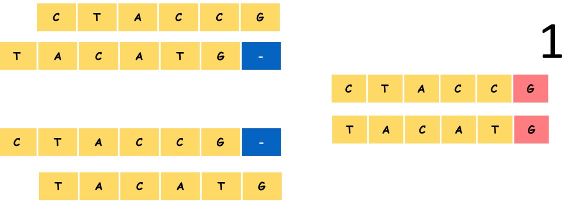
We use up one index from y only

Accrue a gap penalty

$$OPT(i, j-1) + \delta$$

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

Edit Distance – Four Steps



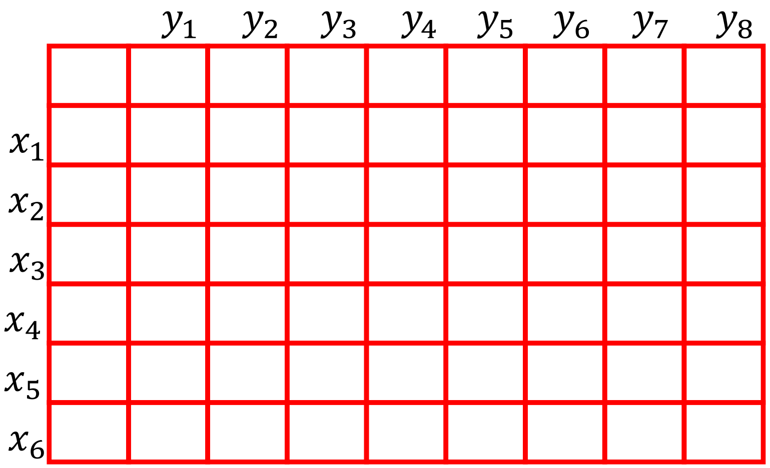
1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Step 2: Identify Memory Structure

x_1	x_2	x_3	x_4	x_5	x_6
C	T	A	C	C	G
T	A	C	A	T	G
y_1	y_2	y_3	y_4	y_5	y_6

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

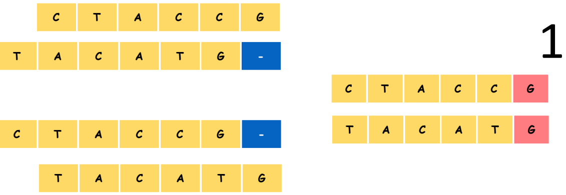
- How many parameters?
  - 2
- What does each represent?
  - The number of items in each sequence
- How many different values?
  - Length of sequence x for i
  - Length of sequence y for j
  - $n \cdot m$ overall

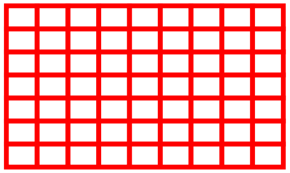


Top-Down Sequence Alignment

```
**align**($i, j$):
    if OPT[$i$][$j$] not blank:    // Check if we've solved this already
        return OPT[$i$][$j$]
    if $i \cdot j == 0$:    // Check if this is a base case
        solution = $(i + j) \cdot \delta$
        OPT[$i$][$j$] = solution    // Always save your solution before returning
        return solution
    match = align($i - 1, j - 1$) // solve each subproblem
    gapx = align($i - 1, j$) // solve each subproblem
    gapy = align($j, i - 1$) // solve each subproblem
    solution = min(match + $\alpha_{x_i y_j}$, gapx + $\delta$, gapy+ $\delta$) // Pick the subproblem to use
     OPT[$i$][$j$] = solution    // Always save your solution before returning
    return solution
```

Edit Distance – Four Steps

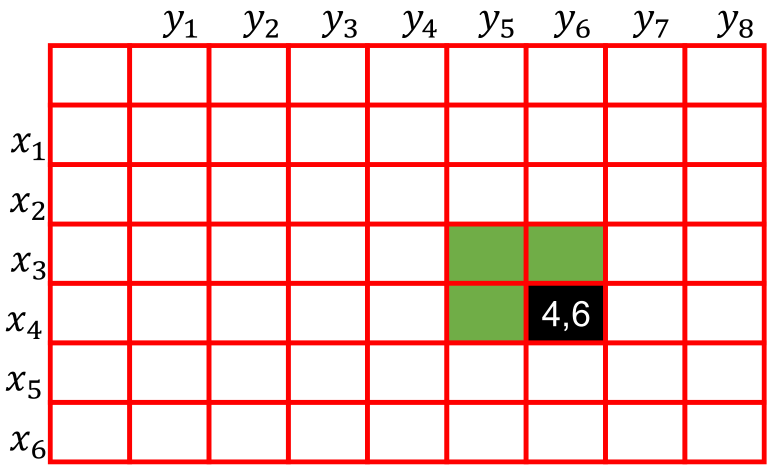




1. Formulate the answer with a recursive structure
    - What are the options for the last choice?
    - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
    - Figure out the possible values of all parameters in the recursive calls.
    - How many subproblems (options for last choice) are there?
    - What are the parameters needed to identify each?
    - How many different values could there be per parameter?
3. Specify an order of evaluation.
    - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
    - With this step: a “Bottom-up” (iterative) algorithm
    - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
    - Is it possible to reuse some memory locations?

Step 3: Identify Order of Evaluation

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

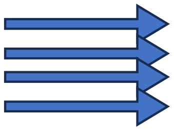


Each index depends on 3 others:

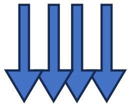
1. The one above it: $(i-1,j)$
2. The one to its left: $(i,j-1)$
3. The one to it's upper left: $(i-1,j-1)$

Any of these orders will work:

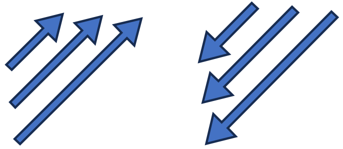
- Top-to-bottom, then left-to-right



- Left-to-right, then top-to-bottom



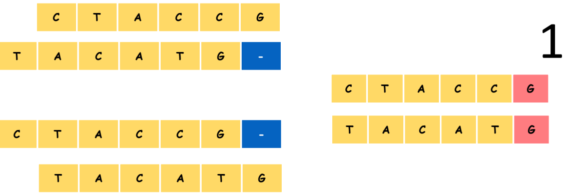
- Diagonally



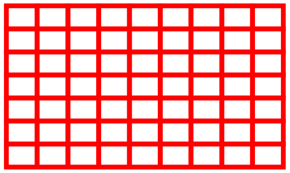
Bottom-Up Sequence Alignment

```
align($x, y$):
    for $i = 0$ up to $n$:
        OPT[$i$][0] = 0   // Solve and save base cases
    for $j = 0$ up to $m$:
        OPT[0][$j$] = 0   // Solve and save base cases
    for $i = 1$ up to $n$:
        for $j = 1$ up to $m$ :
            match = OPT[$i-1$][$j-1$] // solve each subproblem
            gapx = OPT[$i$][$j-1$] // solve each subproblem
            gapy = OPT[$i-1$][$j$] // solve each subproblem
            solution = min(match + $\alpha_{x_i y_j}$, gapx + $\delta$, gapy+ $\delta$) // pick solution
            OPT[$i$][$j$] = solution // save solution
    return OPT[$n$][$m$]
```

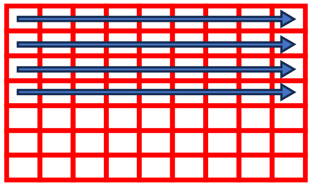
Edit Distance – Four Steps



1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?



2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?



3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Example run with *AGACATTG* and *GAGTTA*: $\boldsymbol{\delta} = \boldsymbol{\alpha}_{\text{mis}} = \mathbf{1}$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1								
A	2								
G	3								
T	4								
T	5								
A	6								

Example run with AGACATTG and GAGTTA: $\boldsymbol{\delta = \alpha_{\mathrm{mis}} = 1}$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2								
G	3								
T	4								
T	5								
A	6								

Example run with **AGACATTG** and **GAGTTA**: $\boldsymbol{\delta} = \boldsymbol{\alpha}_{\text{mis}} = \mathbf{1}$

		A	**G**	**A**	**C**	**A**	**T**	**T**	**G**
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1					
G	3								
T	4								
T	5								
A	6								

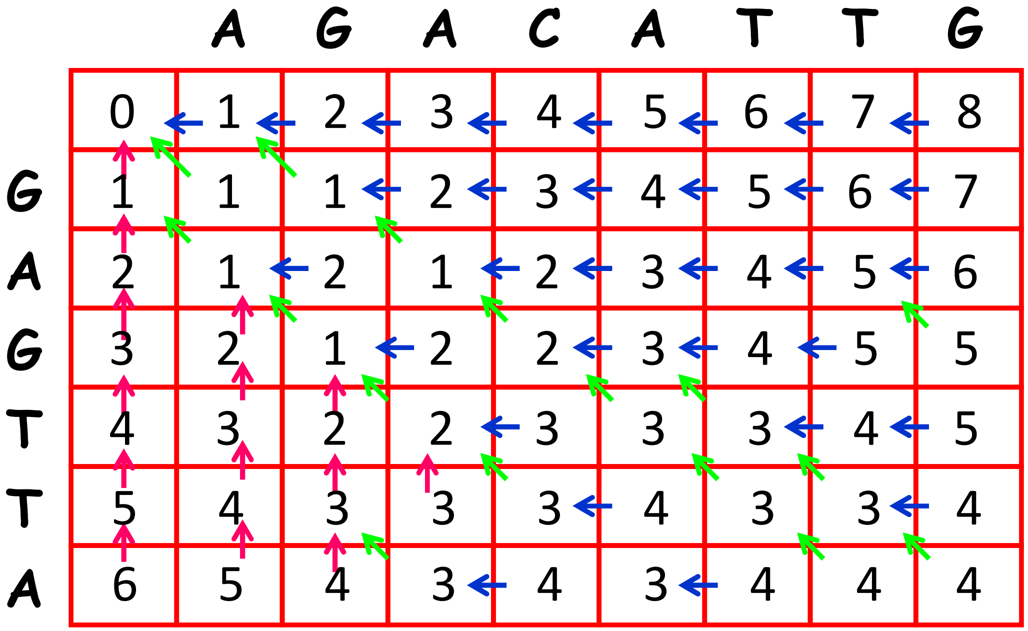
Example run with *AGACATTG* and *GAGTTA*: $\boldsymbol{\delta = \alpha_{\text{mis}} = 1}$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4								
T	5								
A	6								

Example run with **AGACATTG** and **GAGTTA**: $\boldsymbol{\delta = \alpha_{\text{mis}} = 1}$

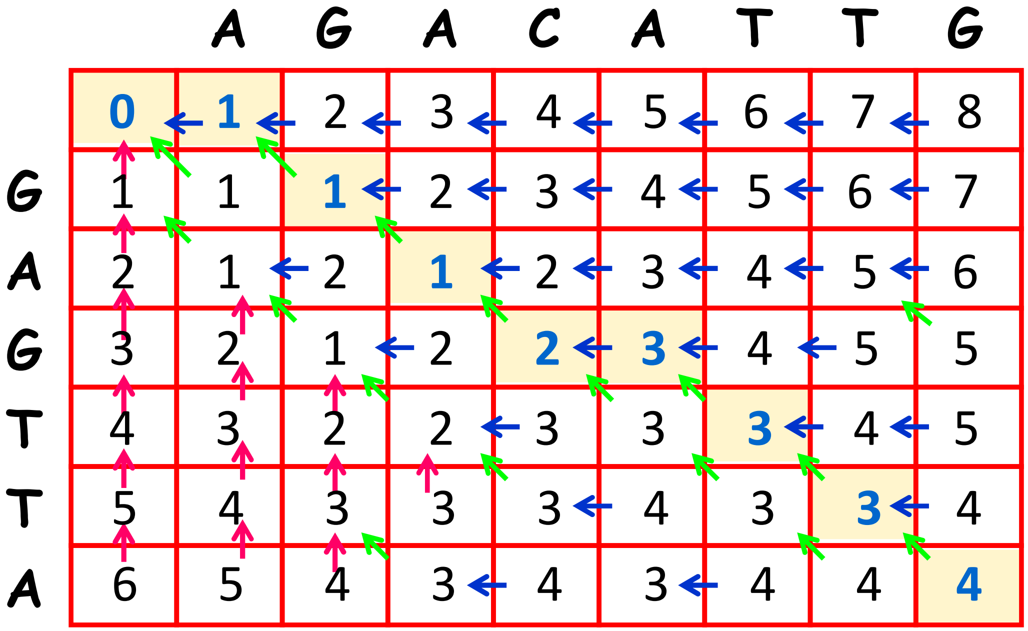
		A	**G**	**A**	**C**	**A**	**T**	**T**	**G**
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4	3	2	2	3	3	3	4	5
T	5	4	3	3	3	4	3	3	4
A	6	5	4	3	4	3	4	4	4

Example run with *AGACATTG* and *GAGTTA*: $\boldsymbol{\delta = \alpha_{\mathrm{mis}} = 1}$



		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4	3	2	2	3	3	3	4	5
T	5	4	3	3	3	4	3	3	4
A	6	5	4	3	4	3	4	4	4

Example run with *AGACATTG* and *GAGTTA*: $\boldsymbol{\delta = \alpha_{\text{mis}} = 1}$



		A	G	A	C	A	T	T	G
	0	**1**	2	3	4	5	6	7	8
G	1	1	**1**	2	3	4	5	6	7
A	2	1	2	**1**	2	3	4	5	6
G	3	2	1	2	**2**	**3**	4	5	5
T	4	3	2	2	3	3	**3**	4	5
T	5	4	3	3	3	4	3	**3**	4
A	6	5	4	3	4	3	4	4	**4**

Optimal Alignment

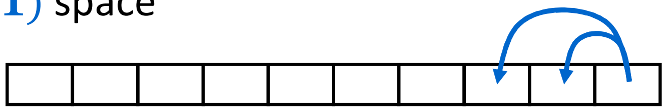
A G A C A T T G

_ G A G _ T T A

Dynamic Programming Patterns

Fibonacci pattern:

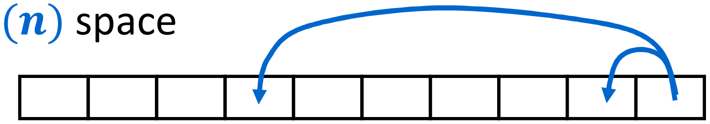
- 1-D, $O(\mathbf{1})$ immediately prior
- $O(\mathbf{1})$ space



$O(\boldsymbol{n})$

Weighted interval scheduling pattern:

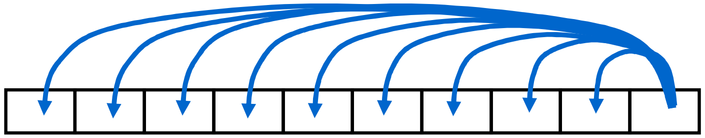
- 1-D, $O(\mathbf{1})$ arbitrary prior
- $O(\boldsymbol{n})$ space



$O(\boldsymbol{n})$

Longest increasing subsequence pattern:

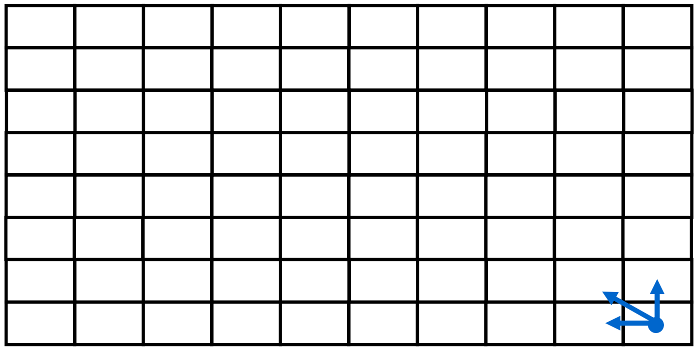
- 1-D, all $\boldsymbol{n}-\mathbf{1}$ prior
- $O(\boldsymbol{n})$ space



$O(\boldsymbol{n}^{\mathbf{2}})$

Alignment pattern:

- 2-D, $O(\mathbf{1})$ in previous row, above and arbitrary prior
- $O(\boldsymbol{n}\cdot\boldsymbol{m})$ space



$O(\boldsymbol{nm})$

Single-source shortest paths (332)

Given: an (un)directed graph $\boldsymbol{G = (V, E)}$ with each edge $\boldsymbol{e}$ having a non-negative weight $\boldsymbol{w(e)}$ and a vertex $\boldsymbol{s}$

Find: (length of) shortest paths from $\boldsymbol{s}$ to each vertex in $\boldsymbol{G}$

Single-source shortest paths (Today)

Given: an (un)directed graph $\boldsymbol{G = (V, E)}$ with each edge $\boldsymbol{e}$ having a ~~non-negative~~ weight $\boldsymbol{w(e)}$ and a vertex $\boldsymbol{s}$

Find: (length of) shortest paths from $\boldsymbol{s}$ to each vertex in $\boldsymbol{G}$

Dijkstra’s Algorithm

- Maintain a set $\boldsymbol{S}$ of vertices whose shortest paths are known
  - initially $\boldsymbol{S} = \{\boldsymbol{s}\}$
- Maintaining current best lengths of paths that *only go through* $\boldsymbol{S}$ to each of the vertices in $\boldsymbol{G}$
  - path-lengths to elements of $\boldsymbol{S}$ will be right, to $\boldsymbol{V} \setminus \boldsymbol{S}$ they might not be right
- Repeatedly add vertex $\boldsymbol{v}$ to $\boldsymbol{S}$ that has the shortest path-length of any vertex in $\boldsymbol{V} \setminus \boldsymbol{S}$
  - update path lengths based on new paths through $\boldsymbol{v}$

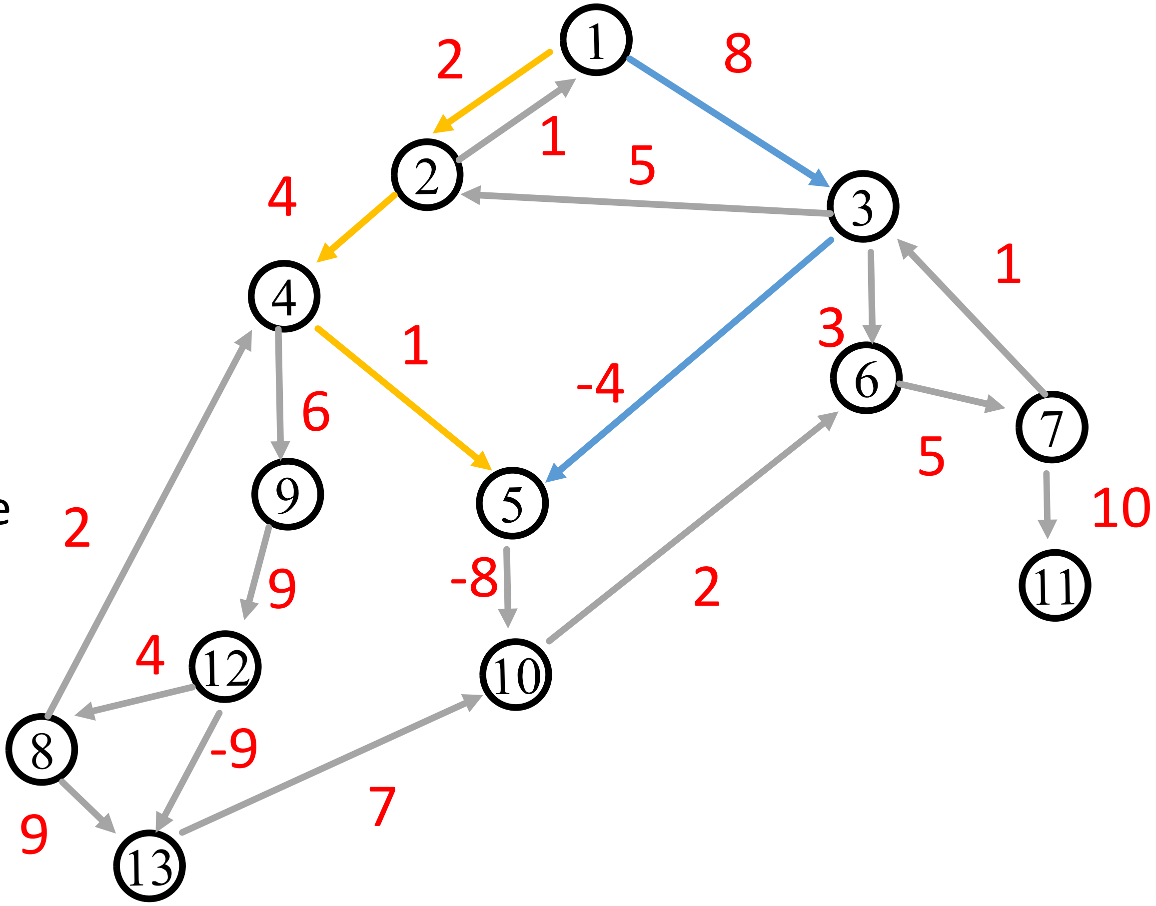
Directed Graph with Negative Weights

Dijkstra's algorithm would not find the shortest path from 1 to 5

Shortest path is 1,3,5 which has cost 4

The path 1,2,4,5 has cost 7

Dijkstra's will "lock in" *that* path before processing node 3



Negative Cycles

There's an issue when a graph has negative cost cycles

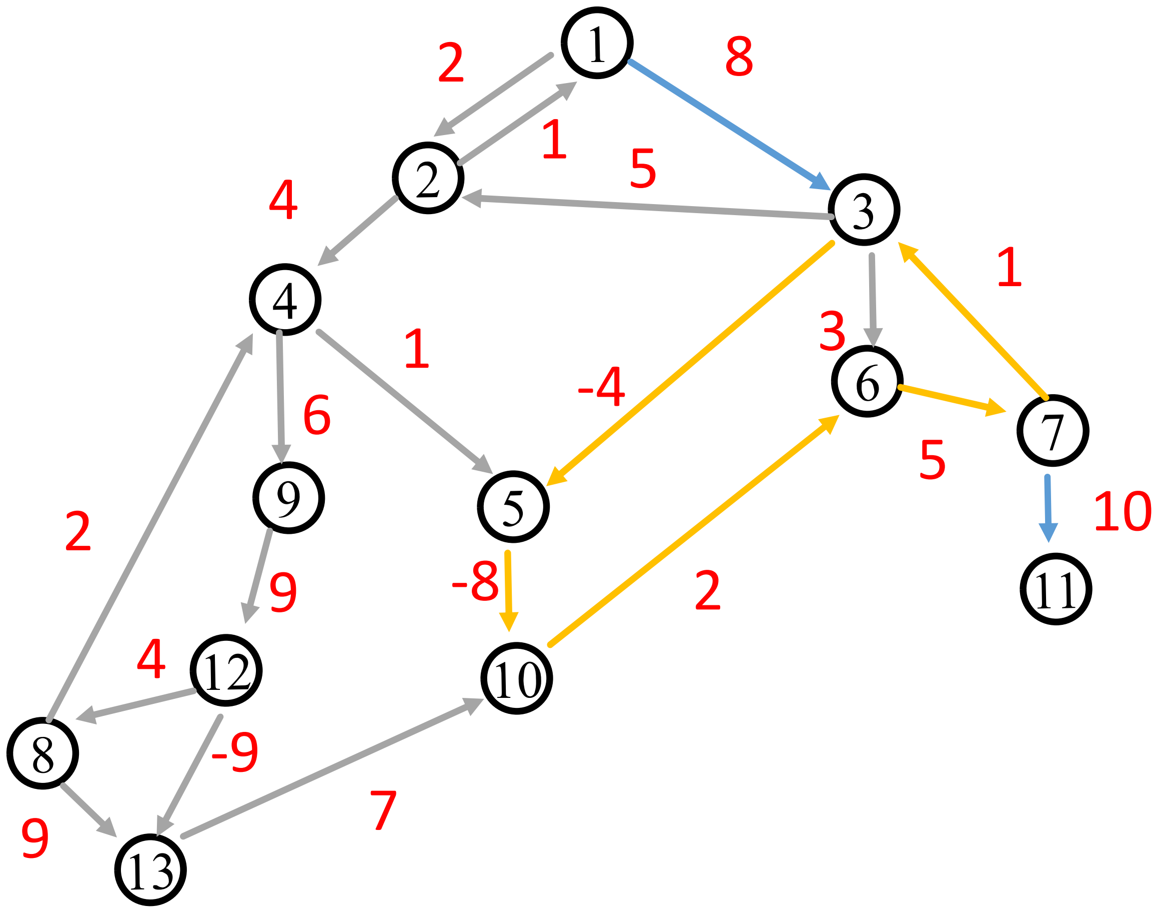
The shortest simple path to node 11 is 1,3,5,10,6,7,11 which has cost 13

The cycle 3,5,8,10,6,7,3 has cost -4

The (non-simple) path 1,3,5,8,10,6,7,3,5,10,6,7,11 has cost 9

Taking the cycle twice gives cost 5

No shortest path exists!



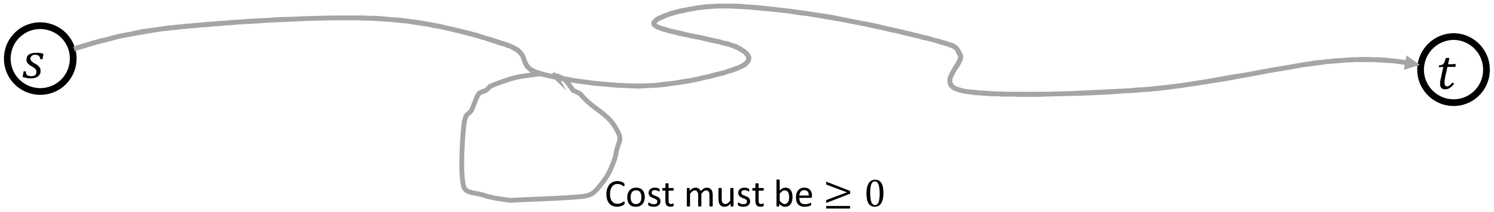
Observations

Claim: A simple path has at most $|V| - 1$ edges

Justication: Pigeon-hole principle. If we have $\geq |V|$ edges then we have used at least one node at least twice

Claim: If a graph has no negative weight cycles then any shortest path must be simple

Justication: If some shortest path was not simple then there is a repeated node. The cycle involving that repeated node must have weight ≥ 0. Removing that cycle from the path can't make it worse



Cost must be ≥ 0

Single-source shortest paths, with negative edge weights

Given: an (un)directed graph $\boldsymbol{G = (V, E)}$ with each edge $\boldsymbol{e}$ having a weight $\boldsymbol{w(e)}$ and a vertex $\boldsymbol{s}$

Find: (length of) shortest paths from $\boldsymbol{s}$ to each vertex in $\boldsymbol{G}$, **or else indicate that there is a negative-cost cycle**

Called the Bellman-Ford algorithm

(The original DP algorithm!)

(Also, the original shortest path algorithm!)

Bellman Ford– Four Steps

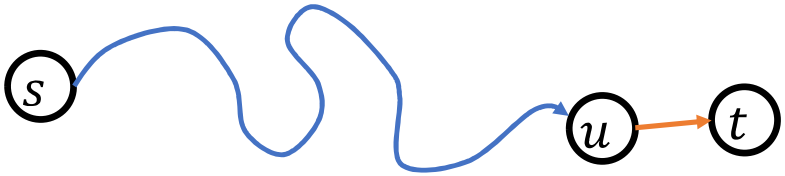
1. Formulate the answer with a recursive structure
    - What are the options for the last choice?
    - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
    - Figure out the possible values of all parameters in the recursive calls.
    - How many subproblems (options for last choice) are there?
    - What are the parameters needed to identify each?
    - How many different values could there be per parameter?
3. Specify an order of evaluation.
    - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
    - With this step: a “Bottom-up” (iterative) algorithm
    - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
    - Is it possible to reuse some memory locations?

Identifying Recursive Structure – False Start

Consider the shortest path from s to t

This shortest path is composed of:

- The shortest path from s to the second-to-last node (call it u)
- The edge (u, t)



$OPT(t)$ = The cost of the shortest path from s to t

$$OPT(t) = \begin{cases} 0 & \text{if } s = t \\ \min\limits_{u \in V}\{OPT(u) + w(u,t)\} & \text{o.w.} \end{cases}$$

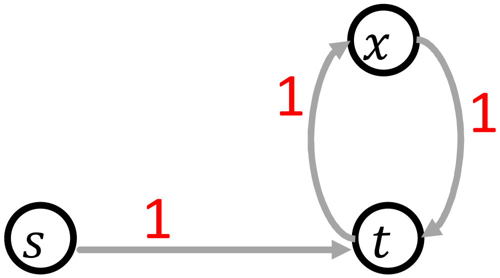
Where $w(u, t)$ is the weight of the edge from u to t if it exists and ∞ if not.

So…What's wrong with this?

$OPT(t)$ = The cost of the shortest path from s to t

$$OPT(t) = \begin{cases} 0 & \text{if } s = t \\ \min\limits_{u \in V}\{OPT(u) + w(u,t)\} & \text{o.w.} \end{cases}$$

Where $w(u,t)$ is the weight of the edge from u to t if it exists and ∞ if not.



$$OPT(t) = \min \begin{cases} \boldsymbol{OPT(x) + 1} \\ OPT(s) + 1 \end{cases}$$

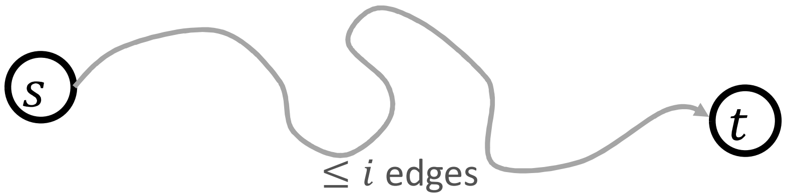
$$OPT(x) = \min \begin{cases} \boldsymbol{OPT(t) + 1} \\ OPT(s) + \infty \end{cases}$$

$$OPT(s) = 0$$

We never reach a base case!

Identifying Recursive Structure – Correctly!

Suppose the shortest path from s to t has ***i* or fewer edges**



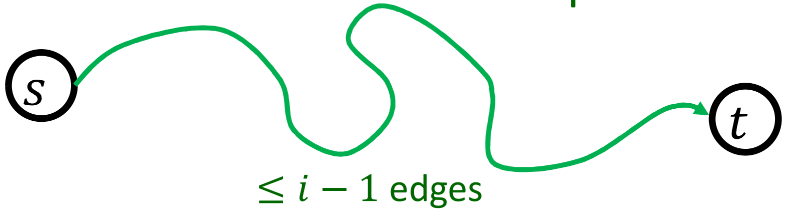
$OPT(i, t)$ = the weight of the shortest path from s to t with at most i edges

This shortest path will be one of these:

Option 1: the shortest path from s to some u with **$i - 1$ or fewer edges**, plus the edge (u, t)

$$\min_{u \in V}\{OPT(i-1, u) + w(u, t)\}$$

Option 2: the same as shortest path from s to t with **$i - 1$ or fewer edges**



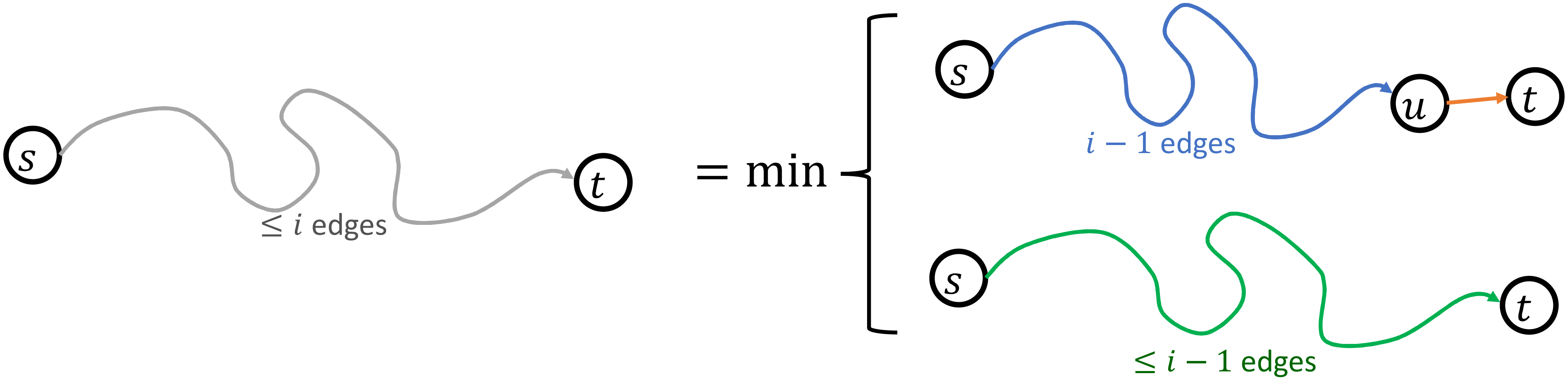
$$OPT(i-1, t)$$

Final Recursive Structure

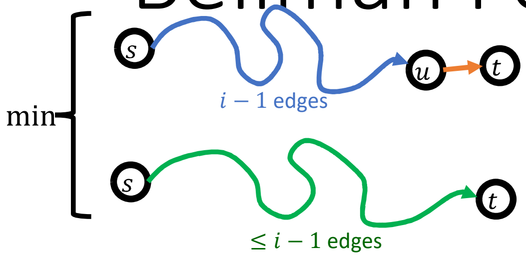
$OPT(i, t)$ = the weight of the shortest path from s to t with at most i edges

$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min\limits_{u \in V}\{OPT(i-1,u) + w(u,t)\} \\ OPT(i-1,t) \end{cases} \end{cases}$$

Where $w(u, t)$ is the weight of the edge from u to t if it exists and ∞ if not.



Bellman Ford– Four Steps



1. Formulate the answer with a recursive structure
    - What are the options for the last choice?
    - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
    - Figure out the possible values of all parameters in the recursive calls.
    - How many subproblems (options for last choice) are there?
    - What are the parameters needed to identify each?
    - How many different values could there be per parameter?
3. Specify an order of evaluation.
    - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
    - With this step: a “Bottom-up” (iterative) algorithm
    - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
    - Is it possible to reuse some memory locations?

Identifying the Memory Structure

$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min\limits_{u \in V}\{OPT(i-1,u) + w(u,t)\} \\ OPT(i-1,t) \end{cases} \end{cases}$$

- How many parameters?
  - 2
- What does each represent?
  - i: the length of the path
  - t: a node
- How many different values?
  - i: $|V|$ (from length 0 up to $|V| - 1$ if the path is simple)
  - t: $|V|$ (one value per node)

	1	2	3	4	5	6	7
0							
1							
2							
3							
4							
5							
6							

Top-Down Bellman-Ford

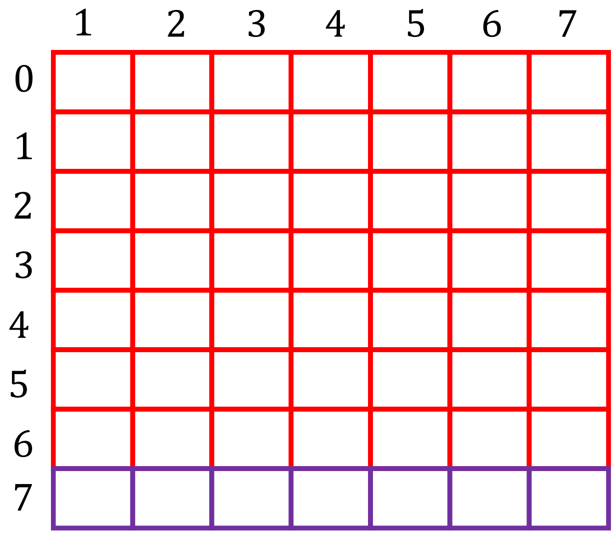
This algorithm correctly finds shortest paths when there are no negative-cost cycles
How can we check for negative cost cycles?

```
BF($i, t$):
    if OPT[$i$][$t$] not blank:    // Check if we've solved this already
        return OPT[$i$][$j$]
    if $i == 0$:    // Check if this is a base case
        solution = 0 ? $t == s$ : ∞
        OPT[$i$][$t$] = solution    // Always save your solution before returning
        return solution
    solution = ∞
    for each $u \in V$:
        solution = min(solution, BF($i - 1, u$) +$w(u, t)$) // solve each subproblem, pick which to use
    solution = min(solution, BF($i - 1,t$)) // solve each subproblem, pick which to use
    OPT[$i$][$t$] = solution    // Always save your solution before returning
    return solution
```

Checking for Negative Cycles

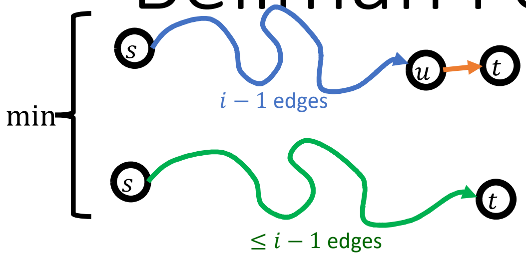
$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min\limits_{u \in V}\{OPT(i-1,u) + w(u,t)\} \\ OPT(i-1,t) \end{cases} \end{cases}$$

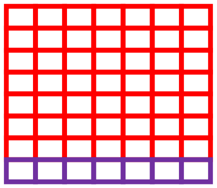
- How many parameters?
  - 2
- What does each represent?
  - i: the length of the path
  - t: a node
- How many different values?
  - **i: $|V|$+1**
    - **a path of $|V|$ edges is not simple, so if any $|V|$-edge path is shorter than one with fewer edges, there must be a negative cycle!**
  - t: $|V|$ (one value per node)



	1	2	3	4	5	6	7
0							
1							
2							
3							
4							
5							
6							
7							

Bellman Ford– Four Steps





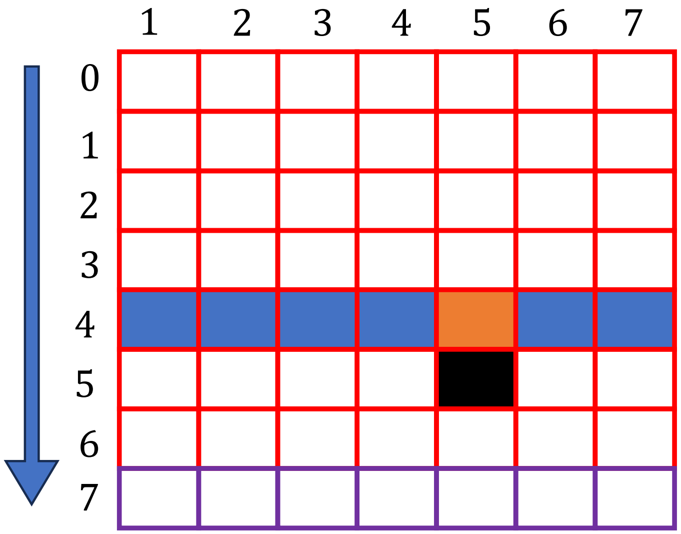
1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Order of Evaluations

$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min_{u \in V} \{ OPT(i-1,u) + w(u,t) \} \\ OPT(i-1,t) \end{cases} \end{cases}$$

Each cell depends on every value in the previous row

Solve in order of i



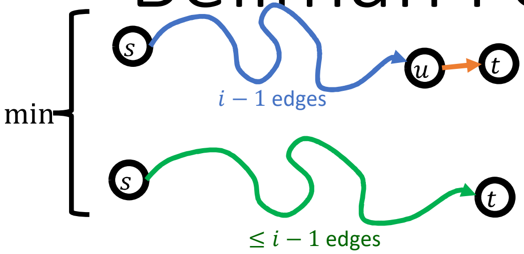
Bottom-Up Bellman-Ford

```
**BF**($s, t$):
    OPT[0][$s$] = 0   // Solve and save base cases
    for $u \in V \setminus \{s\}$:
        OPT[0][$u$] = ∞   // Solve and save base cases
    for $i = 0$ up to $|V|$:
        for $u \in V$:
            for $v \in \mathrm{neighbors}(u)$:
                $OPT[i][u] = \min(OPT[i][u], OPT[i-1][\ v])$ // solve and pick
            $OPT[i][u] = \min(OPT[i][u], OPT[i-1][u])$ // solve and pick
    for $u \in V$:
        if $OPT[|V|][u] < OPT[|V|-1][u]$: // check for negative cycles
            return "negative cycle"
    return OPT[$s$][$t$] // return the final answer
```

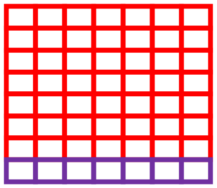
Bottom-Up Bellman-Ford

```
BF($s, t$):
    OPT[0][$s$] = 0   // Solve and save base cases   $\Theta(1)$
    for $u \in V \setminus \{s\}$:
        OPT[0][$u$] = ∞   // Solve and save base cases   $\Theta(|V|)$
    for $i = 0$ up to $|V|$:
        for $u \in V$:
            for $v \in \text{neighbors}(u)$:   $\Theta(|V||E|)$
                $OPT[i][u] = \min(OPT[i][u], OPT[i-1][v])$ // solve and pick
            $OPT[i][u] = \min(OPT[i][u], OPT[i-1][u])$ // solve and pick
    for $u \in V$:
        if $OPT[|V|][u] < OPT[|V|-1][u]$: // check for negative cycles   $\Theta(|V|)$
            return “negative cycle”
    return OPT[$s$][$t$] // return the final answer   $\Theta(1)$
```

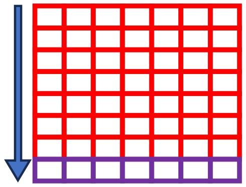
Bellman Ford– Four Steps



1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?



2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?



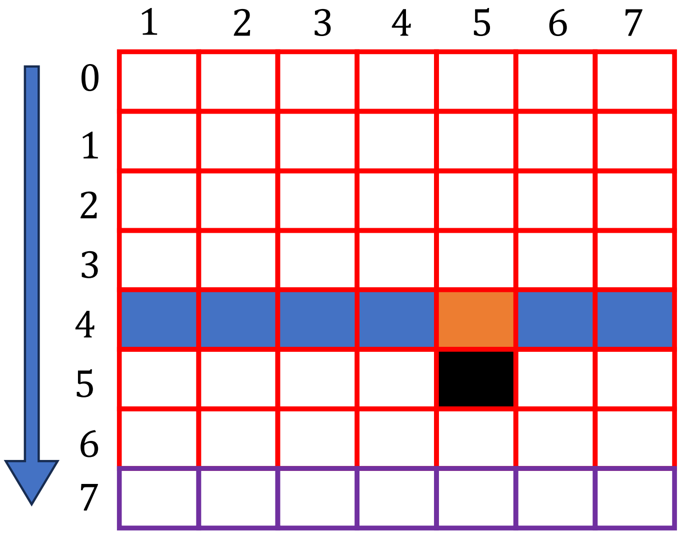
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Order of Evaluations

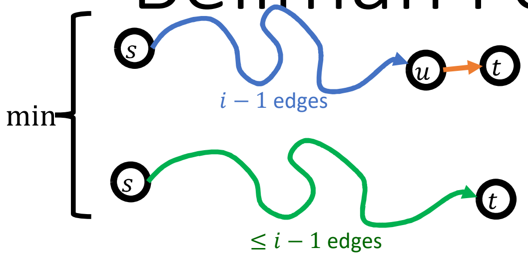
$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min_{u \in V} \{ \color{blue}{OPT(i-1,u)} + \color{orange}{w(u,t)} \} \\ \color{green}{OPT(i-1,t)} \end{cases} \end{cases}$$

Each cell depends *only* on values in the previous row

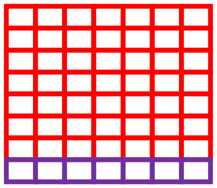
We only need two rows!



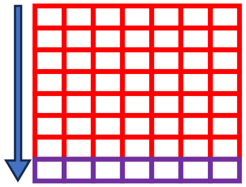
Bellman Ford– Four Steps



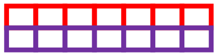
1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?



2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?



3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm

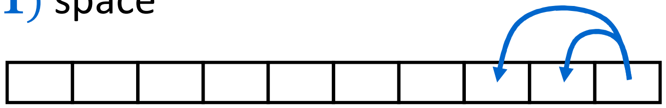


4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Dynamic Programming Patterns

Fibonacci pattern:

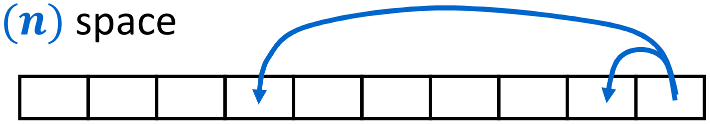
- 1-D, $O(\mathbf{1})$ immediately prior
- $O(\mathbf{1})$ space



$O(\boldsymbol{n})$

Weighted interval scheduling pattern:

- 1-D, $O(\mathbf{1})$ arbitrary prior
- $O(\boldsymbol{n})$ space

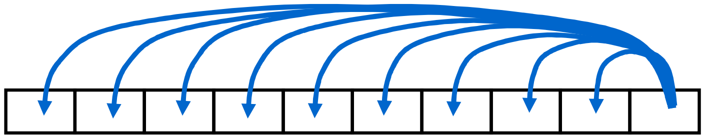


$O(\boldsymbol{n})$

Longest increasing subsequence pattern:

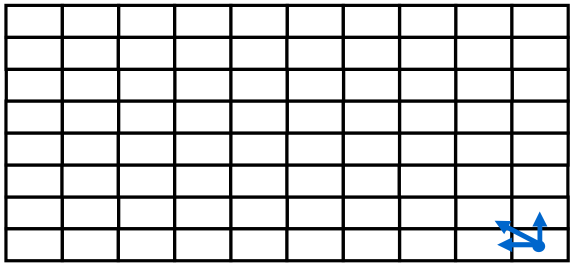
- 1-D, all $\boldsymbol{n}-\mathbf{1}$ prior
- $O(\boldsymbol{n})$ space

$O(\boldsymbol{n^2})$



Alignment pattern:

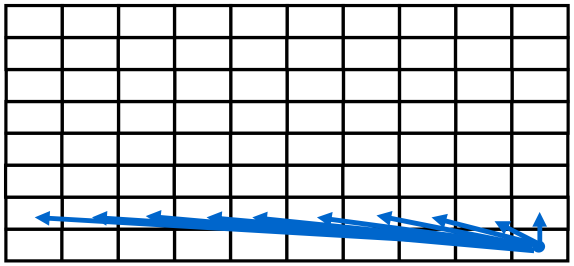
- 2-D, $O(\mathbf{1})$ in previous row, above, left, diagonal
- $O(\boldsymbol{n}\cdot\boldsymbol{m})$ space



$O(\boldsymbol{nm})$

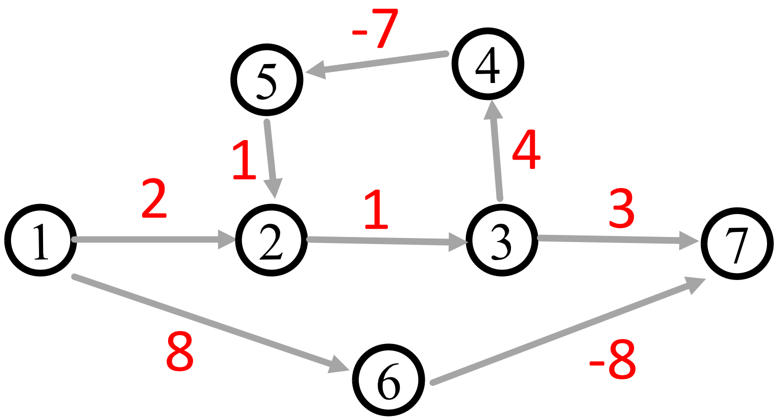
Bellman Ford pattern:

- 2-D, $O(\mathbf{|V|})$ in previous row,
- $O(|V|)$ space

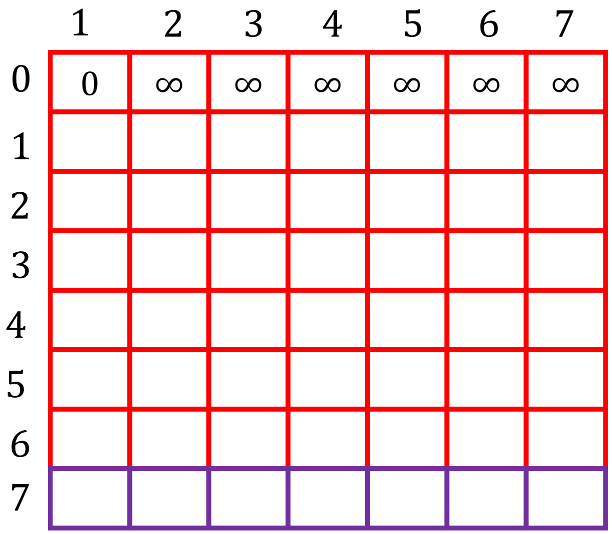


$O(|V||E|)$

Example Execution

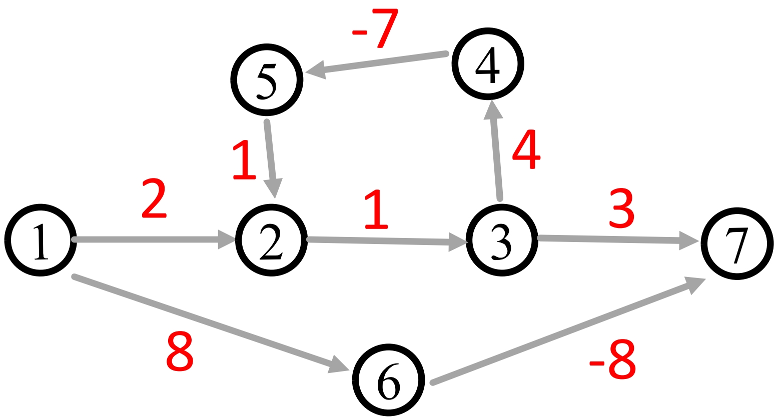


$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min_{u \in V} \{ OPT(i-1,u) + w(u,t) \} \\ OPT(i-1,t) \end{cases} \end{cases}$$

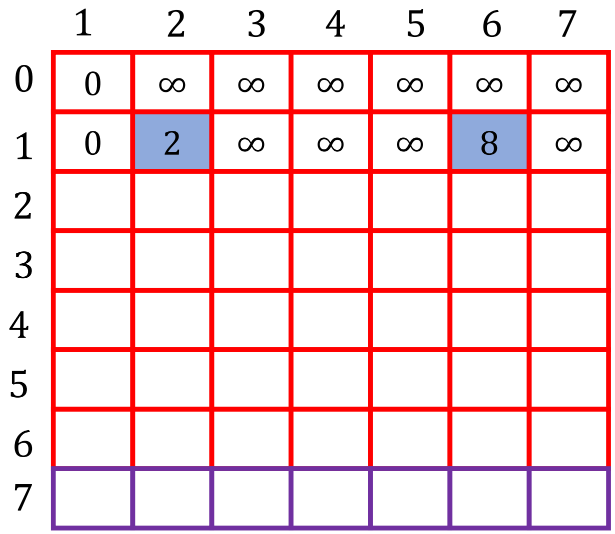


	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞
1							
2							
3							
4							
5							
6							
7							

Example Execution

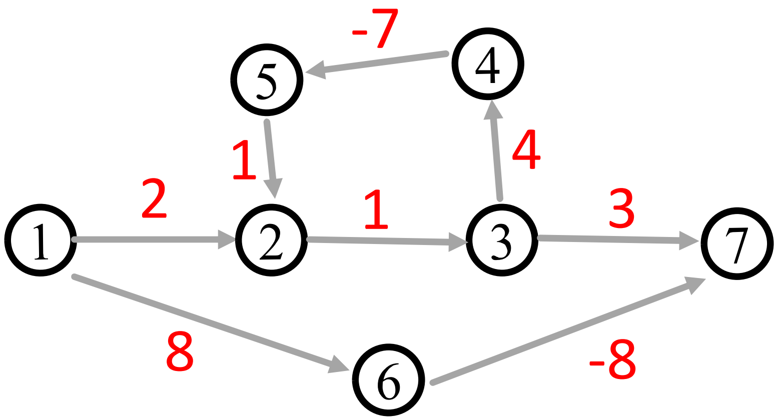


$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min\limits_{u \in V} \{OPT(i-1,u) + w(u,t)\} \\ OPT(i-1,t) \end{cases} \end{cases}$$



	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞
1	0	2	∞	∞	∞	8	∞
2							
3							
4							
5							
6							
7							

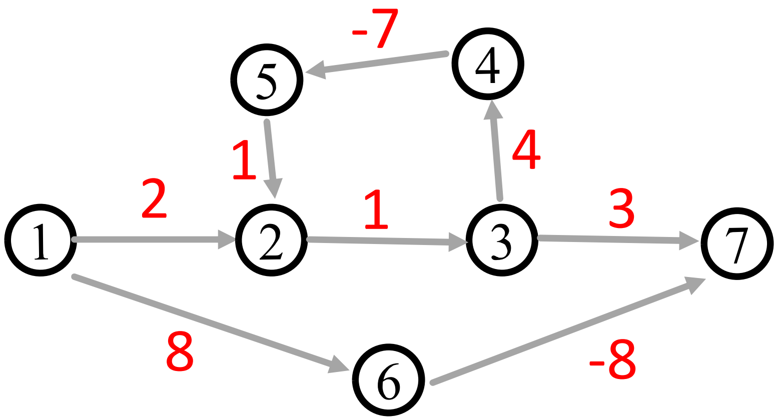
Example Execution



$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min_{u \in V} \{OPT(i-1,u) + w(u,t)\} \\ OPT(i-1,t) \end{cases} \end{cases}$$

	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞
1	0	2	∞	∞	∞	8	∞
2	0	2	3	∞	∞	8	0
3							
4							
5							
6							
7							

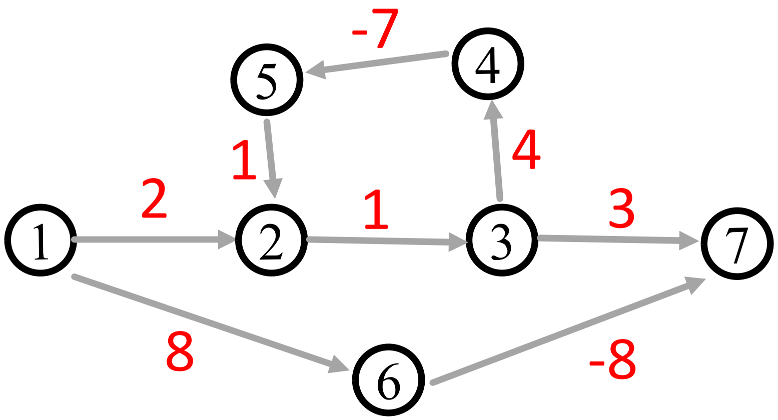
Example Execution



$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min\limits_{u \in V} \{ OPT(i-1,u) + w(u,t) \} \\ OPT(i-1,t) \end{cases} \end{cases}$$

	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞
1	0	2	∞	∞	∞	8	∞
2	0	2	3	∞	∞	8	0
3	0	2	3	7	∞	8	0
4							
5							
6							
7							

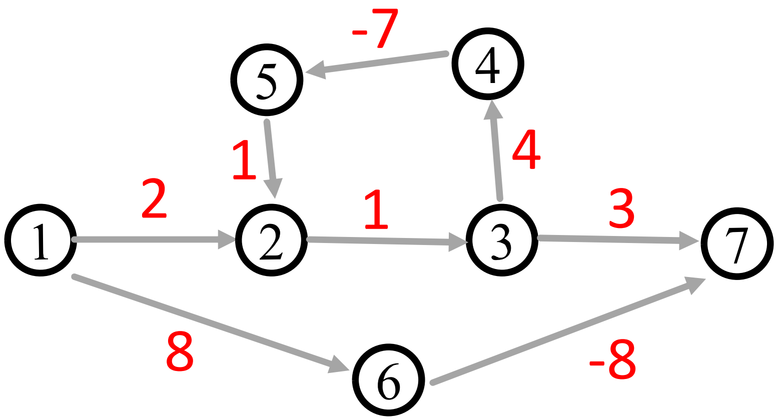
Example Execution



$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min\limits_{u \in V}\{OPT(i-1,u) + w(u,t)\} \\ OPT(i-1,t) \end{cases} \end{cases}$$

	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞
1	0	2	∞	∞	∞	8	∞
2	0	2	3	∞	∞	8	0
3	0	2	3	7	∞	8	0
4	0	2	3	7	0	8	0
5							
6							
7							

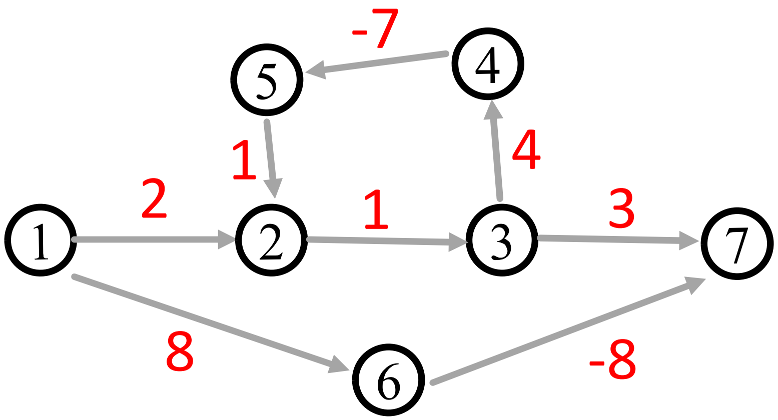
Example Execution



$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min\limits_{u \in V} \{OPT(i-1,u) + w(u,t)\} \\ OPT(i-1,t) \end{cases} \end{cases}$$

	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞
1	0	2	∞	∞	∞	8	∞
2	0	2	3	∞	∞	8	0
3	0	2	3	7	∞	8	0
4	0	2	3	7	0	8	0
5	0	1	3	7	0	8	0
6							
7							

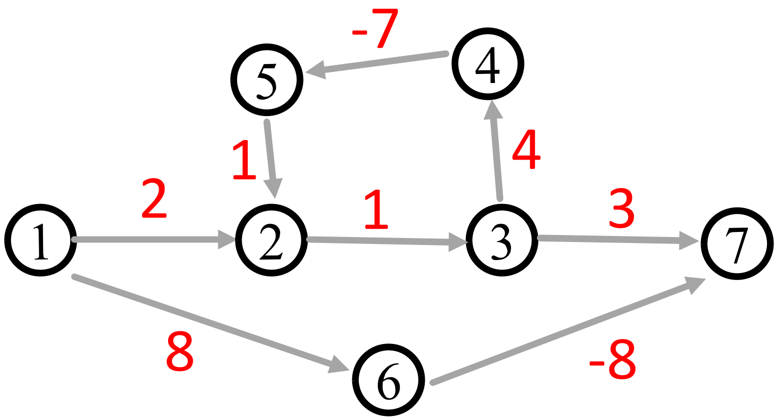
Example Execution



$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min\limits_{u \in V} \{OPT(i-1,u) + w(u,t)\} \\ OPT(i-1,t) \end{cases} \end{cases}$$

	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞
1	0	2	∞	∞	∞	8	∞
2	0	2	3	∞	∞	8	0
3	0	2	3	7	∞	8	0
4	0	2	3	7	0	8	0
5	0	1	3	7	0	8	0
6	0	1	2	7	0	8	0
7							

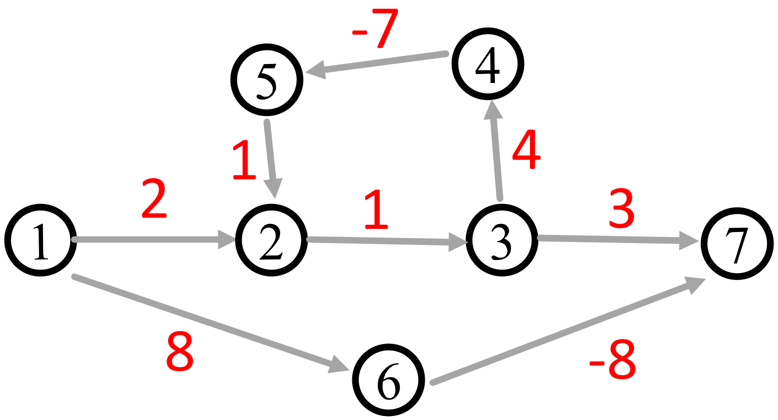
Example Execution



$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min\limits_{u \in V} \{OPT(i-1,u) + w(u,t)\} \\ OPT(i-1,t) \end{cases} \end{cases}$$

	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞
1	0	2	∞	∞	∞	8	∞
2	0	2	3	∞	∞	8	0
3	0	2	3	7	∞	8	0
4	0	2	3	7	0	8	0
5	0	1	3	7	0	8	0
6	0	1	2	7	0	8	0
7	0	1	2	3	0	8	0

Example Execution



$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min\limits_{u \in V} \{ OPT(i-1,u) + w(u,t) \} \\ OPT(i-1,t) \end{cases} \end{cases}$$

	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞
1	0	2	∞	∞	∞	8	∞
2	0	2	3	∞	∞	8	0
3	0	2	3	7	∞	8	0
4	0	2	3	7	0	8	0
5	0	1	3	7	0	8	0
6	0	1	2	7	0	8	0
7	0	1	2	3	0	8	0

Negative Cycle Found!