**Breadth-First Search (BFS)**

**Input:** Unweighted directed or undirected graph $G = (V, E)$ with vertex set $V$ and edge set $E$, and a source vertex $s \in G.V$. For each $v \in V$, the adjacency list of $v$ is $G.Adj[v]$.

**Output:** For all $v \in G[V]$, $v.d$ is set to the shortest distance (in terms of the number of edges) from $s$ to $v$. Also, $v.\pi$ pointers form a breadth-first tree rooted at $s$ that contains all vertices reachable from $s$.

```
BFS ( G, s )
1.   for each vertex u ∈ G.V \ {s} do
2.     u.color ← WHITE, u.d ← ∞, u.π ← NIL
3.   s.color ← GRAY, s.d ← 0, s.π ← NIL
4.   Queue Q ← ∅
5.   ENQUEUE( Q, s )
6.   while Q ≠ ∅ do
7.     u ← DEQUEUE( Q )
8.     for each v ∈ G.Adj[u] do
9.       if v.color = WHITE then
10.          v.color ← GRAY, v.d ← u.d + 1, v.π ← u
11.           ENQUEUE( Q, v )
12.     u.color ← BLACK
```
Breadth-First Search (BFS)

ENQUEUE (Q, s)

\[ Q \begin{array}{c} s \\ 0 \end{array} \]

\[
\begin{array}{c}
\infty \\
v \\
\infty \\
w \\
\infty \\
x \\
\infty \\
y \\
\infty \\
r \\
0 \\
\infty \\
s \\
\infty \\
t \\
\infty \\
u
\end{array}
\]
Breadth-First Search (BFS)

**DEQUEUE** \((Q) \rightarrow s\)

**ENQUEUE** \((Q, w), \, ENQUEUE\, (Q, r)\)

\[
\begin{array}{c}
\text{Q} \\
\begin{array}{cc}
w & r \\
1 & 1 \\
\end{array}
\end{array}
\]
Breadth-First Search (BFS)

\[\text{DEQUEUE (} Q \text{)} \rightarrow w\]

\[\text{ENQUEUE (} Q, t \text{), ENQUEUE (} Q, x \text{)}\]

\[
\begin{array}{cccc}
1 & 2 & \infty \\
\infty & 0 & 2 \\
\infty & \infty & \infty \\
\end{array}
\]

\[Q = \begin{bmatrix}
1 & 2 & 2 \\
\end{bmatrix}\]
**Breadth-First Search (BFS)**

**DEQUEUE** \((Q) \rightarrow r\)

**ENQUEUE** \((Q, x), (Q, v)\)

\[
\begin{array}{c}
\text{r} & \text{s} & \text{t} & \text{u} \\
1 & 0 & 2 & \infty \\
\end{array}
\]

\[
\begin{array}{c}
v & w & x & y \\
2 & 1 & 2 & \infty \\
\end{array}
\]

**Queue**

\[
\begin{array}{c}
Q \quad t & x & v \\
2 & 2 & 2 \\
\end{array}
\]
Breadth-First Search (BFS)

**DEQUEUE** \( (Q) \rightarrow t \)

**ENQUEUE** \( (Q, u) \)

\[
\begin{align*}
&\text{\textcolor{green}{DEQUEUE}} (Q) \rightarrow t \\
&\text{\textcolor{green}{ENQUEUE}} (Q, u)
\end{align*}
\]
Breadth-First Search (BFS)

**DEQUEUE (Q) → x**

**ENQUEUE (Q, y)**

Diagram:

```
  1 ——— 2 ——— 3
  |     |     |
v     w     y
  0 ——— 2
  |     |
s     x
  |     |
  r ——— 2
```

Queue:

```
  Q  |   v  |   u  |   y  |
     2 | 3    | 3    |
```
Breadth-First Search (BFS)

**Dequeue (Q) → v**

Q \[ \begin{matrix} u & y \\ 3 & 3 \end{matrix} \]
Breadth-First Search (BFS)

**Deque (Q) → u**

Diagram of a connected graph with nodes labeled from 0 to 4, and edges connecting them. The queue `Q` contains `y` and `3`. The node `u` is dequeued from the queue.
Breadth-First Search (BFS)

\[ \text{DEQUEUE}(Q) \rightarrow y \]

\[ Q \rightarrow \emptyset \]
**Breadth-First Search (BFS)**

<table>
<thead>
<tr>
<th>BFS ( G, s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( \text{for each vertex } u \in G.V \setminus {s} ) do</td>
</tr>
<tr>
<td>2. ( u.color \leftarrow \text{WHITE}, \ u.d \leftarrow \infty, \ u.\pi \leftarrow \text{NIL} )</td>
</tr>
<tr>
<td>3. ( s.color \leftarrow \text{GRAY}, \ s.d \leftarrow 0, \ s.\pi \leftarrow \text{NIL} )</td>
</tr>
<tr>
<td>4. Queue ( Q \leftarrow \emptyset )</td>
</tr>
<tr>
<td>5. ( \text{ENQUEUE}(Q,s) )</td>
</tr>
<tr>
<td>6. ( \text{while } Q \neq \emptyset ) do</td>
</tr>
<tr>
<td>7. ( u \leftarrow \text{DEQUEUE}(Q) )</td>
</tr>
<tr>
<td>8. ( \text{for each } v \in G.Adj[u] ) do</td>
</tr>
<tr>
<td>9. ( \text{if } v.color = \text{WHITE} ) then</td>
</tr>
<tr>
<td>10. ( v.color \leftarrow \text{GRAY}, \ v.d \leftarrow u.d + 1, \ v.\pi \leftarrow u )</td>
</tr>
<tr>
<td>11. ( \text{ENQUEUE}(Q,v) )</td>
</tr>
<tr>
<td>12. ( u.color \leftarrow \text{BLACK} )</td>
</tr>
</tbody>
</table>

Let \( n = |G.V| \) and \( m = |G.E| \)

**Time spent**
- initializing = \( \Theta(n) \)
- enqueueing / dequeuing = \( \Theta(n) \)
- scanning the adjacency lists = \( \Theta(\sum_{v \in G.V}|G.Adj[v]|) \) = \( \Theta(m) \)

\( \therefore \) Total cost = \( \Theta(m + n) \)
Depth-First Search (DFS)

**Input:** Unweighted directed or undirected graph \( G = (V, E) \) with vertex set \( V \) and edge set \( E \). For each \( v \in V \), the adjacency list of \( v \) is \( G.\text{Adj}[v] \).

**Output:** For each \( v \in G[V] \), \( v.d \) is set to the time when \( v \) was first discovered and \( v.f \) is set to the time when \( v \)’s adjacency list has been examined completely. Also, \( v.\pi \) pointers form a breadth-first tree rooted at \( s \) that contains all vertices reachable from \( s \).

\[
\text{DFS} \ ( G ) \\
1. \quad \text{for each vertex } u \in G.V \ \text{do} \\
2. \quad \ \ u.\text{color} \leftarrow \text{WHITE}, \ u.\pi \leftarrow \text{NIL} \\
3. \quad \text{time} \leftarrow 0 \\
4. \quad \text{for each } u \in G.V \ \text{do} \\
5. \quad \quad \text{if } u.\text{color} = \text{WHITE} \ \text{then} \\
6. \quad \quad \quad \text{DFS-Visit}( G, u ) \\
\]

\[
\text{DFS-Visit} \ ( G, u ) \\
1. \quad \ \text{time} \leftarrow \text{time} + 1 \\
2. \quad \ u.d \leftarrow \text{time} \\
3. \quad \ u.\text{color} \leftarrow \text{GRAY} \\
4. \quad \quad \text{for each } v \in G.\text{Adj}[u] \ \text{do} \\
5. \quad \quad \quad \text{if } v.\text{color} = \text{WHITE} \ \text{then} \\
6. \quad \quad \quad \quad \ v.\pi \leftarrow u \\
7. \quad \quad \quad \text{DFS-Visit}( G, v ) \\
8. \quad \ u.\text{color} \leftarrow \text{BLACK} \\
9. \quad \ \text{time} \leftarrow \text{time} + 1 \\
10. \quad \ u.f \leftarrow \text{time} \]

13
Depth-First Search (DFS)
Depth-First Search (DFS)

**Tree Edge (T):** These are edges in the depth-first forest $G_\pi$. Edge $(u, v)$ is a tree edge if $v$ was first discovered by exploring that edge. In the example above, we will make all tree edges green and thick.
Depth-First Search (DFS)
Depth-First Search (DFS)
Back Edge (B): A back edge goes from a vertex to its ancestor in a depth-first tree. Self-loops are also considered back edges.
Depth-First Search (DFS)
Depth-First Search (DFS)

\[\begin{align*}
&\text{Depth-First Search (DFS)} \\
&\text{Diagram showing nodes and edges}\end{align*}\]
Depth-First Search (DFS)
Depth-First Search (DFS)

Forward Edge (F): A forward edge is a nontree edge that connects a vertex to a descendant in a depth-first tree.
Depth-First Search (DFS)
Depth-First Search (DFS)
**Cross Edge (C):** If a non-tree edge is neither a back edge nor a forward edge then it’s a cross edge. Cross edges can go between vertices in the same depth-first tree or in different depth-first trees.
Depth-First Search (DFS)
Depth-First Search (DFS)
Depth-First Search (DFS)
Depth-First Search (DFS)
Depth-First Search (DFS)

**DFS (G)**
1. for each vertex \( u \in G. V \) do
2. \( u.\text{color} \leftarrow \text{WHITE}, \ u.\pi \leftarrow \text{NIL} \)
3. \( \text{time} \leftarrow 0 \)
4. for each \( u \in G. V \) do
5. \( \text{if} \ u.\text{color} = \text{WHITE} \text{ then} \)
6. \( \text{DFS-Visit}(G, u) \)

**DFS-Visit (G, u)**
1. \( \text{time} \leftarrow \text{time} + 1 \)
2. \( u.d \leftarrow \text{time} \)
3. \( u.\text{color} \leftarrow \text{GRAY} \)
4. for each \( v \in G.\text{Adj}[u] \) do
5. \( \text{if} \ v.\text{color} = \text{WHITE} \text{ then} \)
6. \( v.\pi \leftarrow u \)
7. \( \text{DFS-Visit}(G, v) \)
8. \( u.\text{color} \leftarrow \text{BLACK} \)
9. \( \text{time} \leftarrow \text{time} + 1 \)
10. \( u.f \leftarrow \text{time} \)

Let \( n = |G. V| \) and \( m = |G. E| \)

**Time spent**
- in **DFS** (exclusive of calls to **DFS-Visit**) = \( \Theta(n) \)
- in **DFS-Visit** scanning the adjacency lists = \( \Theta(\sum_{v \in G.V} |G.\text{Adj}[v]|) \)
  = \( \Theta(m) \)

\[ \therefore \text{Total cost} = \Theta(m + n) \]
A **topological sort** of a DAG (i.e., directed acyclic graph) $G = (V, E)$ is a linear ordering of all its vertices such that if $G$ contains an edge $(u, v)$, then $u$ appears before $v$ in the ordering.

We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.
**Topological Sort**

**TOPOLOGICAL-SORT ( G )**

1. call DFS ( G ) to compute the finish times \( v.f \) for each vertex \( v \in G.V \)
2. as each vertex is finished, insert it into the front of a linked list
3. return the linked list of vertices

```
socks 17/18
undershorts 11/16
pants 12/15
shirt 13/14
watch 9/10
shoes 13/14
socks 17/18
undershorts 11/16
pants 12/15
shirt 13/14
watch 9/10
```
Strongly Connected Components

A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u$ and $v$ in $C$, we have both $u \rightarrow v$ and $v \rightarrow u$; that is, vertices $u$ and $v$ are reachable from each other.
A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u$ and $v$ in $C$, we have both $u \sim v$ and $v \sim u$; that is, vertices $u$ and $v$ are reachable from each other.

![Graph Diagram]

The diagram illustrates a directed graph with vertices labeled $a, b, c, d, e, f, g, h$ and edges connecting them to form various strongly connected components.
A *strongly connected component* of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u$ and $v$ in $C$, we have both $u \rightarrow v$ and $v \rightarrow u$; that is, vertices $u$ and $v$ are reachable from each other.
Strongly Connected Components

**STRONGLY-CONNECTED-COMPONENTS** *(G)*

1. call *DFS* *(G)* to compute the finish times *v.f* for each vertex *v ∈ G.V*
2. compute *G^T*
3. call *DFS* *(G^T)*, but in the main loop of *DFS*, consider the vertices in order of decreasing *v.f* (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component
**Strongly Connected Components**

\[
\text{STRONGLY-CONNECTED-COMPONENTS}(G)
\]

1. call \(\text{DFS}(G)\) to compute the finish times \(v.f\) for each vertex \(v \in G.V\)
2. compute \(G^T\)
3. call \(\text{DFS}(G^T)\), but in the main loop of \(\text{DFS}\), consider the vertices in order of decreasing \(v.f\) (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component
Strongly Connected Components

**STRONGLY-CONNECTED-COMPONENTS (G)**

1. call **DFS (G)** to compute the finish times \(v.f\) for each vertex \(v \in G.V\)
2. compute \(G^T\)
3. call **DFS (G^T)**, but in the main loop of **DFS**, consider the vertices in order of decreasing \(v.f\) (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component
Strongly Connected Components

**STRONGLY-CONNECTED-COMPONENTS (G)**

1. call DFS (G) to compute the finish times v.f for each vertex v ∈ G.V
2. compute \( G^T \)
3. call DFS (\( G^T \)), but in the main loop of DFS, consider the vertices in order of decreasing v.f (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

![Diagram of a graph with vertices and edges](image)
**Strongly Connected Components**

**STRONGLY-CONNECTED-COMPONENTS** \((G)\)

1. call **DFS** \((G)\) to compute the finish times \(v.f\) for each vertex \(v \in G.V\)
2. compute \(G^T\)
3. call **DFS** \((G^T)\), but in the main loop of **DFS**, consider the vertices in order of decreasing \(v.f\) (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component
Strongly Connected Components

**STRONGLY-CONNECTED-COMPONENTS (G)**

1. call \textit{DFS} (G) to compute the finish times \(v.f\) for each vertex \(v \in G.V\)
2. compute \(G^T\)
3. call \textit{DFS} (\(G^T\)), but in the main loop of \textit{DFS}, consider the vertices in order of decreasing \(v.f\) (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component
Strongly Connected Components

**STRONGLY-CONNECTED-COMPONENTS** (*G*)

1. call **DFS** (*G*) to compute the finish times *v.f* for each vertex *v* ∈ *G.V*
2. compute *G^T*
3. call **DFS** (*G^T*), but in the main loop of **DFS**, consider the vertices in order of decreasing *v.f* (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component
The Single-Source Shortest Paths (SSSP) Problem

We are given a weighted, directed graph $G = (V, E)$ with vertex set $V$ and edge set $E$, and a weight function $w$ such that for each edge $(u, v) \in E$, $w(u, v)$ represents its weight.

We are also given a source vertex $s \in V$.

Our goal is to find a shortest path (i.e., a path of the smallest total edge weight) from $s$ to each vertex $v \in V$. 
SSSP: Relaxation

**INITIALIZE-SINGLE-SOURCE** \( (G = (V, E), s) \)

1. for each vertex \( v \in G.V \) do
2. \( v.d \leftarrow \infty \)
3. \( v.\pi \leftarrow NIL \)
4. \( s.d \leftarrow 0 \)

**RELAX** \( (u, v, w) \)

1. if \( u.d + w(u, v) < v.d \) then
2. \( v.d \leftarrow u.d + w(u, v) \)
3. \( v.\pi \leftarrow u \)
SSSP: Properties of Shortest Paths and Relaxation

The **weight** \( w(p) \) of path \( p = \langle v_0, v_1, \ldots, v_k \rangle \) is the sum of the weights of its constituent edges:

\[
    w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)
\]

We define the **shortest-path weight** \( \delta(u, v) \) from \( u \) to \( v \) by

\[
    \delta(u, v) = \begin{cases} 
    \min\{w(p) : p \text{ is } u \sim v\}, & \text{if there is a path from } u \text{ to } v, \\
    \infty, & \text{otherwise.}
    \end{cases}
\]

A **shortest path** from vertex \( u \) to vertex \( v \) is then defined as any path \( p \) with weight \( w(p) = \delta(u, v) \).
SSSP: Properties of Shortest Paths and Relaxation

**Triangle inequality** (Lemma 24.10 of CLRS)
For any edge \((u, v) \in E\), we have \(\delta(s, v) \leq \delta(s, u) + w(u, v)\).

**Upper-bound inequality** (Lemma 24.11 of CLRS)
We always have \(v.d \geq \delta(s, v)\) for all vertices \(v \in V\), and once \(v.d\) achieves the value \(\delta(u, v)\), it never changes.

**No-path property** (Corollary 24.12 of CLRS)
If there is no path from \(s\) to \(v\), then we always have \(v.d = \delta(s, v) = \infty\).

**Convergence property** (Lemma 24.14 of CLRS)
If \(s \sim u \rightarrow v\) is a shortest path in \(G\) for some \(u, v \in V\), and if \(u.d = \delta(s, u)\) at any time prior to relaxing edge \((u, v)\), then \(v.d = \delta(s, v)\) at all times afterward.
Path-relaxation property (Lemma 24.15 of CLRS)
If \( p = \langle v_0, v_1, \ldots, v_k \rangle \) is a shortest path from \( s = v_0 \) to \( v_k \), and we relax the edges of \( p \) in the order \((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\), then \( v_k \cdot d = \delta(s, v_k) \). This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations on the edges of \( p \).

Predecessor-subgraph property (Lemma 24.17 of CLRS)
Once \( v \cdot d = \delta(s, v) \) for all \( v \in V \), the predecessor subgraph is a shortest-paths tree rooted at \( s \).
Dijkstra’s SSSP Algorithm with a Min-Heap

( SSSP: Single-Source Shortest Paths )

Since we already discussed Dijkstra’s SSSP algorithm when we talked about greedy algorithms, we will skip over it in this lecture.

**Input:** Weighted graph $G = (V, E)$ with vertex set $V$ and edge set $E$, a non-negative weight function $w$, and a source vertex $s \in G[V]$.

**Output:** For all $v \in G[V]$, $v.d$ is set to the shortest distance from $s$ to $v$.

```
Dijkstra-SSSP( G = (V, E), w, s )
1. for each vertex v \in G.V do
2.     v.d \leftarrow \infty
3.     v.\pi \leftarrow NIL
4.     s.d \leftarrow 0
5.     Min-Heap Q \leftarrow \emptyset
6.     for each vertex v \in G.V do
7.         INSERT( Q, v )
8.     while Q \neq \emptyset do
9.         u \leftarrow EXTRACT-MIN( Q )
10.        for each (u, v) \in G.E do
11.            if u.d + w(u, v) < v.d then
12.                v.d \leftarrow u.d + w(u, v)
13.                v.\pi \leftarrow u
14.                DECREASE-KEY( Q, v, u.d + w(u, v) )
```

Let $n = |G[V]|$ and $m = |G[E]|$

Worst-case running time:

Using a binary min-heap

$= O((m + n) \log n)$

Using a Fibonacci heap

$= O(m + n \log n)$
Dijkstra’s SSSP Algorithm with a Min-Heap
( SSSP: Single-Source Shortest Paths )

Input: Weighted graph $G = (V, E)$ with vertex set $V$ and edge set $E$, a non-negative weight function $w$, and a source vertex $s \in G[V]$.

Output: For all $v \in G[V]$, $v.d$ is set to the shortest distance from $s$ to $v$.

Let $n = |G[V]|$ and $m = |G[E]|$

Worst-case running time:

Using a binary min-heap

$= O((m + n) \log n)$

Using a Fibonacci heap

$= O(m + n \log n)$
The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

Input: Weighted graph \( G = (V, E) \) with vertex set \( V \) and edge set \( E \), a weight function \( w \), and a source vertex \( s \in G[V] \). Negative-weight edges are allowed (unlike Dijkstra’s SSSP algorithm).

Output: Returns FALSE if a negative-weight cycle is reachable from \( s \), otherwise returns TRUE and for all \( v \in G[V] \), sets \( v.d \) to the shortest distance from \( s \) to \( v \).

\[
\begin{align*}
\text{INITIALIZE-SINGLE-SOURCE} & ( G = (V, E), \ s ) \\
1. & \text{ for each vertex } v \in G.V \text{ do} \\
2. & \quad v.d \leftarrow \infty \\
3. & \quad v.\pi \leftarrow \text{NIL} \\
4. & \quad s.d \leftarrow 0 \\
\end{align*}
\]

\[
\begin{align*}
\text{RELAX} & ( u, v, w ) \\
1. & \text{ if } u.d + w(u,v) < v.d \text{ then} \\
2. & \quad v.d \leftarrow u.d + w(u,v) \\
3. & \quad v.\pi \leftarrow u \\
\end{align*}
\]

\[
\begin{align*}
\text{BELLMAN-FORD} & ( G = (V, E), w, \ s ) \\
1. & \text{ INITIALIZE-SINGLE-SOURCE}( G, s ) \\
2. & \text{ for } i \leftarrow 1 \text{ to } |G.V| - 1 \text{ do} \\
3. & \quad \text{ for each } (u,v) \in G.E \text{ do} \\
4. & \quad \quad \text{ RELAX}( u, v, w ) \\
5. & \quad \text{ for each } (u,v) \in G.E \text{ do} \\
6. & \quad \quad \text{ if } u.d + w(u,v) < v.d \text{ then} \\
7. & \quad \quad \quad \text{return FALSE} \\
8. & \quad \quad \text{return TRUE} \\
\end{align*}
\]
The Bellman-Ford (SSSP) Algorithm
( SSSP: Single-Source Shortest Paths )

Initial State (with initial tentative distances)
The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

Iteration 1

Graph showing the shortest path from source node s to all other nodes.
The Bellman-Ford (SSSP) Algorithm
( SSSP: Single-Source Shortest Paths )

Iteration 2
The Bellman-Ford (SSSP) Algorithm (SSSP: Single-Source Shortest Paths)

Iteration 3
The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

Iteration 4
The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

Done!

Diagram:
- Vertices: s, t, x, y, z
- Edges: s→t (6), t→y (8), y→z (7), z→x (9), x→t (5), t→x (−2), s→y (−3), y→s (−4)
- Source vertex: s
- Shortest path lengths:
  - s→t = 14
  - s→y = 15
  - s→z = 20
  - s→x = 17

56
The Bellman-Ford (SSSP) Algorithm
( SSSP: Single-Source Shortest Paths )

\[ \text{INITIALIZE-SINGLE-SOURCE} \ ( G = (V, E), \ s ) \]
1. for each vertex \( v \in G.V \) do
2. \( v.d \leftarrow \infty \)
3. \( v.\pi \leftarrow \text{NIL} \)
4. \( s.d \leftarrow 0 \)

\[ \text{RELAX} \ ( u, v, w ) \]
1. if \( u.d + w(u,v) < v.d \) then
2. \( v.d \leftarrow u.d + w(u,v) \)
3. \( v.\pi \leftarrow u \)

Let \( n = |V| \) and \( m = |E| \)

Time taken by: Line 1: \( \Theta(n) \)
Lines 2 – 4: \( \Theta(mn) \)
Lines 5 – 7: \( \Theta(m) \)

Total time: \( \Theta(mn) \)
Correctness of the Bellman-Ford Algorithm

**Lemma 24.2 (CLRS):** Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w: E \rightarrow \mathbb{R}$, and suppose $G$ contains no negative-weight cycles reachable from $s$. Then, after the $|V| - 1$ iterations of the for loop of lines 2–4 of **Bellman-Ford**, we have $v \cdot d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

**Proof:** The proof is based on the *path-relaxation property*. Consider any $v \in G.V$ reachable from $s$, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from $s$ to $v$. Because shortest paths are simple, $p$ has at most $|V| - 1$ edges, and so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations of the for loop of lines 2–4 relaxes all $|E|$ edges. Among the edges relaxed in the $i^{th}$ iteration, for $i = 1, 2, \ldots, k$, is $(v_{i-1}, v_i)$. By the path-relaxation property, therefore, $v \cdot d = v_k \cdot d = \delta(s, v_k) = \delta(s, v)$. 

---

58
**Correctness of the Bellman-Ford Algorithm**

**Corollary 24.3 (CLRS):** Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w: E \rightarrow \mathbb{R}$, and suppose $G$ contains no negative-weight cycles reachable from $s$. Then, for each $v \in V$, there is a path from $s$ to $v$ if and only if **Bellman-Ford** terminates with $v$. $d < \infty$ when it is run on $G$. 
**Correctness of the Bellman-Ford Algorithm**

**THEOREM 24.4 (CLRS):** Let $\text{BELLMAN-FORD}$ be run on a weighted, directed graph $G = (V, E)$ with source $s$ and weight function $w: E \rightarrow \mathbb{R}$. If $G$ contains no negative-weight cycles reachable from $s$, then the algorithm returns $\text{TRUE}$, we have $v. d = \delta(s, v)$ for all $v \in V$, and the predecessor subgraph $G_\pi$ is a shortest-paths tree rooted at $s$. If $G$ does contain a negative-weight cycle reachable from $s$, then the algorithm returns $\text{FALSE}$. 
Correctness of the Bellman-Ford Algorithm

Proof of Theorem 24.4: Two cases:

*G contains no negative-weight cycles reachable from s:*

If \( v \in G \cdot V \) is reachable from \( s \) then according to Lemma 24.2 we have \( v \cdot d = \delta(s, v) \) at termination. Otherwise, \( v \cdot d = \delta(s, v) = \infty \) follows from the \textit{no-path property}.

The \textit{predecessor-subgraph property}, along with \( v \cdot d = \delta(s, v) \), implies that \( G_\pi \) is a shortest-paths tree.

Now, since at termination, for all edges \((u, v) \in G \cdot E\), we have, \( v \cdot d = \delta(s, v) \) and \( u \cdot d = \delta(s, u) \), then by \textit{triangle inequality}:

\[
v \cdot d = \delta(s, v) \leq \delta(s, u) + w(u, v) = u \cdot d + w(u, v).
\]

So, none of the tests in line 6 causes \textit{Bellman-Ford} to return FALSE. Therefore, it returns TRUE.
Correctness of the Bellman-Ford Algorithm

**Proof of Theorem 24.4 (Continued):**

$G$ contains a negative-weight cycle reachable from $s$:

Let $c = \langle v_0, v_1, \ldots, v_k \rangle$ be the cycle, where $v_0 = v_k$. Then

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0.$$ 

Assume for the sake of contradiction that *Bellman-Ford* returns *True*. Then $v_i \cdot d \leq v_{i-1} \cdot d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. Thus,

$$\sum_{i=1}^{k} v_i \cdot d \leq \sum_{i=1}^{k} (v_{i-1} \cdot d + w(v_{i-1}, v_i)) = \sum_{i=1}^{k} v_{i-1} \cdot d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

But $\sum_{i=1}^{k} v_i \cdot d = \sum_{i=1}^{k} v_{i-1} \cdot d$, and by Corollary 24.3, each $v_i \cdot d$ is finite. Thus, $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$, which contradicts our initial assumption that $c = \langle v_0, v_1, \ldots, v_k \rangle$ is a negative-weight cycle.
SSSP in Directed Acyclic Graphs (DAGs) (SSSP: Single-Source Shortest Paths)

**Input:** Weighted DAG $G = (V, E)$ with vertex set $V$ and edge set $E$, a weight function $w$, and a source vertex $s \in G[V]$. Negative-weight edges are allowed (unlike Dijkstra’s SSSP algorithm).

**Output:** For all $v \in G[V]$, sets $v.d$ to the shortest distance from $s$ to $v$.

---

### INITIALIZE-SINGLE-SOURCE ($G = (V, E), s$)

1. *for* each vertex $v \in G.V$ *do*
2. $v.d \leftarrow \infty$
3. $v.\pi \leftarrow \text{NIL}$
4. $s.d \leftarrow 0$

### RELAX ($u$, $v$, $w$)

1. *if* $u.d + w(u,v) < v.d$ *then*
2. $v.d \leftarrow u.d + w(u,v)$
3. $v.\pi \leftarrow u$

### DAG-SHORTEST-PATHS ($G = (V, E), w, s$)

1. topologically sort the vertices of $G$
2. INITIALIZE-SINGLE-SOURCE($G, s$)
3. *for* each $v \in V.G$ taken in topologically sorted order *do*
4. *for* each $(u,v) \in G.E$ *do*
5. RELAX($u,v,w$)
SSSP in Directed Acyclic Graphs (DAGs) (SSSP: Single-Source Shortest Paths)

Given DAG

```
SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

Given DAG

Gráfico de DAG con pesos en las aristas.
```
SSSP in Directed Acyclic Graphs (DAGs)  
(SSSP: Single-Source Shortest Paths)

After Topological Sorting (with initial tentative distances)
SSSP in Directed Acyclic Graphs (DAGs)
( SSSP: Single-Source Shortest Paths )

After Iteration 1
SSSP in Directed Acyclic Graphs (DAGs)  
( SSSP: Single-Source Shortest Paths )

After Iteration 2

![Graph diagram showing shortest paths after iteration 2.](Image)
SSSP in Directed Acyclic Graphs (DAGs)
( SSSP: Single-Source Shortest Paths )

After Iteration 3
SSSP in Directed Acyclic Graphs (DAGs)  
(SSSP: Single-Source Shortest Paths)

After Iteration 4
SSSP in Directed Acyclic Graphs (DAGs) (SSSP: Single-Source Shortest Paths)

After Iteration 5
SSSP in Directed Acyclic Graphs (DAGs) (SSSP: Single-Source Shortest Paths)

Done!
SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

**INITIALIZE-SINGLE-SOURCE** \((G = (V, E), s)\)
1. for each vertex \(v \in G.V\) do
2. \(v.d \leftarrow \infty\)
3. \(v.\pi \leftarrow NIL\)
4. \(s.d \leftarrow 0\)

**RELAX** \((u, v, w)\)
1. if \(u.d + w(u,v) < v.d\) then
2. \(v.d \leftarrow u.d + w(u,v)\)
3. \(v.\pi \leftarrow u\)

**DAG-SHORTEST-PATHS** \((G = (V, E), w, s)\)
1. topologically sort the vertices of \(G\)
2. **INITIALIZE-SINGLE-SOURCE** \((G, s)\)
3. for each \(v \in V.G\) taken in topologically sorted order do
4. for each \((u,v) \in G.E\) do
5. **RELAX** \((u,v,w)\)

Let \(n = |V|\) and \(m = |E|\)

Time taken by:

- Line 1: \(\Theta(n + m)\)
- Line 2: \(\Theta(n)\)
- Lines 3 – 5: \(\Theta(m)\)

Total time: \(\Theta(n + m)\)
Correctness of DAG-Shortest-Paths

**Theorem 24.5 (CLRS):** If a weighted, directed graph $G = (V, E)$ has a source vertex $s$ and no cycles, then at the termination of the DAG-Shortest-Paths procedure, $v.d = \delta(s, v)$ for all vertices $v \in G.V$, and the predecessor subgraph $G_\pi$ is a shortest-paths tree.

**Proof:** Consider any $v \in G.V$.

If $v$ is not reachable from $s$ then $v.d = \delta(s, v) = \infty$ follows from the no-path property.

If $v$ is reachable from $s$, and let $p = \langle v_0, v_1, ..., v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from $s$ to $v$. Since we process the vertices in topological order, we relax the edges on $p$ in the order $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. The path-relaxation property implies that $v_i.d = \delta(s, v_i)$ at termination for $i = 1, 2, ..., k$.

By the predecessor-subgraph property, $G_\pi$ is a shortest-paths tree.
Correctness of DAG-Shortest-Paths

**Theorem 24.5 (CLRS):** If a weighted, directed graph $G = (V, E)$ has a source vertex $s$ and no cycles, then at the termination of the **DAG-Shortest-Paths** procedure, $v. d = \delta(s, v)$ for all vertices $v \in G.V$, and the predecessor subgraph $G_\pi$ is a shortest-paths tree.

**Proof:** Consider any $v \in G.V$.

If $v$ is not reachable from $s$ then $v. d = \delta(s, v) = \infty$ follows from the *no-path property*.

If $v$ is reachable from $s$, and let $p = \langle v_0, v_1, ..., v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from $s$ to $v$. Since we process the vertices in topological order, we relax the edges on $p$ in the order $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. The *path-relaxation property* implies that $v_i. d = \delta(s, v_i)$ at termination for $i = 1,2, ..., k$.

By the *predecessor-subgraph property*, $G_\pi$ is a shortest-paths tree.
**Correctness of DAG-Shorest-Paths**

**Theorem 24.5 (CLRS):** If a weighted, directed graph $G = (V, E)$ has a source vertex $s$ and no cycles, then at the termination of the **DAG-Shortest-Paths** procedure, $v.d = \delta(s, v)$ for all vertices $v \in G.V$, and the predecessor subgraph $G_\pi$ is a shortest-paths tree.

**Proof:** Consider any $v \in G.V$.

If $v$ is not reachable from $s$ then $v.d = \delta(s, v) = \infty$ follows from the **no-path property**.

If $v$ is reachable from $s$, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from $s$ to $v$. Since we process the vertices in topological order, we relax the edges on $p$ in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. The **path-relaxation property** implies that $v_i.d = \delta(s, v_i)$ at termination for $i = 1, 2, \ldots, k$.

By the **predecessor-subgraph property**, $G_\pi$ is a shortest-paths tree.
The All-Pairs Shortest Paths (APSP) Problem

We are given a weighted, directed graph $G = (V, E)$ with vertex set $V$ and edge set $E$, and a weight function $w$ such that for each edge $(u, v) \in E$, $w(u, v)$ represents its weight.

Our goal is to find, for every pair of vertices $u, v \in G.V$, a shortest path (i.e., a path of the smallest total edge weight) from $u$ to $v$. 
The All-Pairs Shortest Paths (APSP) Problem

One can solve the APSP problem by running an SSSP algorithm $n = |G.V|$ times, once for each vertex as the source.

If all edge weights are nonnegative, one can use Dijkstra’s SSSP algorithm. Using a binary min-heap as the priority queue, one can solve the problem in $O(n(m + n) \log n)$ time, where $m = |G.E|$. Using a Fibonacci heap as the priority queue yields a running time of $O(n^2 \log n + mn)$.

If $G$ has negative-weight edges, then one can use the slower Bellman-Ford SSSP algorithm resulting in a running time of $O(mn^2)$ which is $O(n^4)$ for dense graphs.
The All-Pairs Shortest Paths (APSP) Problem

We assume that the edge-weights are given as an $n \times n$ adjacency matrix $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0, & \text{if } i = j, \\ \text{weight of directed edge } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

We allow negative-weight edges, but we assume for the time being that $G$ contains no negative-weight cycles.
**APSP: Extending SPs by One Edge at a Time**

Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex $i$ to vertex $j$ that contains at most $m$ edges. Then

$$l_{ij}^{(m)} = \begin{cases} 
0, & \text{if } m = 0 \text{ and } i = j, \\
\infty, & \text{if } m = 0 \text{ and } i \neq j, \\
\min_{1 \leq k \leq n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}, & \text{otherwise (i.e., } m > 0) .
\end{cases}$$

If $G$ has no negative-weight cycles, then for every pair of vertices $i$ and $j$ for which $\delta(i, j) < \infty$, there is a shortest path from $i$ to $j$ that is simple and thus contains at most $n - 1$ edges. A path from vertex $i$ to vertex $j$ with more than $n - 1$ edges cannot have lower weight than a shortest path from $i$ to $j$. Hence,

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \ldots.$$
Let $l^{(m)}_{ij}$ be the minimum weight of any path from vertex $i$ to vertex $j$ that contains at most $m$ edges. Then

$$l^{(m)}_{ij} = \begin{cases} 
0, & \text{if } m = 0 \text{ and } i = j, \\
\infty, & \text{if } m = 0 \text{ and } i \neq j, \\
\min_{1 \leq k \leq n} \left\{ l^{(m-1)}_{ik} + w_{kj} \right\}, & \text{otherwise (i.e., } m > 0). 
\end{cases}$$

If $G$ has no negative-weight cycles, then for every pair of vertices $i$ and $j$ for which $\delta(i, j) < \infty$, there is a shortest path from $i$ to $j$ that is simple and thus contains at most $n - 1$ edges. A path from vertex $i$ to vertex $j$ with more than $n - 1$ edges cannot have lower weight than a shortest path from $i$ to $j$. Hence,

$$\delta(i, j) = l^{(n-1)}_{ij} = l^{(n)}_{ij} = l^{(n+1)}_{ij} = \cdots.$$
**APSP: Extending SPs by One Edge at a Time**

**EXTEND-SHORTEST-PATHS (L, W)**

1. \( n \leftarrow L.\text{rows} \)
2. let \( L' = (l'_{ij}) \) be a new \( n \times n \) matrix
3. for \( i \leftarrow 1 \) to \( n \) do
4. for \( j \leftarrow 1 \) to \( n \) do
5. \( l'_{ij} \leftarrow \infty \)
6. for \( k \leftarrow 1 \) to \( n \) do
7. \( l'_{ij} \leftarrow \min(l'_{ij}, l'_{ik} + w_{kj}) \)
8. return \( L' \)

**SLOW-ALL-PAIRS-SHORTEST-PATHS (W)**

1. \( n \leftarrow W.\text{rows} \)
2. \( L^{(1)} \leftarrow W \)
3. for \( m \leftarrow 2 \) to \( n - 1 \) do
4. let \( L^{(m)} \) be a new \( n \times n \) matrix
5. \( L^{(m)} \leftarrow \text{EXTEND-SHORTEST-PATHS}(L^{(m-1)}, W) \)
6. return \( L^{(n-1)} \)
APSP: Extending SPs by One Edge at a Time

\[ W = \begin{pmatrix}
0 & 3 & 8 & \infty & -4 \\
\infty & 0 & \infty & 1 & 7 \\
\infty & 4 & 0 & \infty & \infty \\
2 & \infty & -5 & 0 & \infty \\
\infty & \infty & \infty & 6 & 0
\end{pmatrix} \quad \quad L^{(1)} = \begin{pmatrix}
0 & 3 & 8 & \infty & -4 \\
\infty & 0 & \infty & 1 & 7 \\
\infty & 4 & 0 & \infty & \infty \\
2 & \infty & -5 & 0 & \infty \\
\infty & \infty & \infty & 6 & 0
\end{pmatrix} \]
APSP: Extending SPs by One Edge at a Time

\[ L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \]

\[ L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix} \]
APSP: Extending SPs by One Edge at a Time

\[
L^{(2)} = \begin{pmatrix}
0 & 3 & 8 & 2 & -4 \\
3 & 0 & -4 & 1 & 7 \\
\infty & 4 & 0 & 5 & 11 \\
2 & -1 & -5 & 0 & -2 \\
8 & \infty & 1 & 6 & 0
\end{pmatrix} \quad L^{(3)} = \begin{pmatrix}
0 & 3 & -3 & 2 & -4 \\
3 & 0 & -4 & 1 & -1 \\
7 & 4 & 0 & 5 & 11 \\
2 & -1 & -5 & 0 & -2 \\
8 & 5 & 1 & 6 & 0
\end{pmatrix}
\]
APSP: Extending SPs by One Edge at a Time

\[ L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \]

\[ L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \]
Note the similarity between **EXTEND-SHORTEST-PATHS** and **SQUARE-MATRIX-MULTIPLY**:

**EXTEND-SHORTEST-PATHS** (L, W)
1. \( n \leftarrow L.\text{rows} \)
2. let \( L' = (l'_{ij}) \) be a new \( n \times n \) matrix
3. for \( i \leftarrow 1 \) to \( n \) do
4. for \( j \leftarrow 1 \) to \( n \) do
5. \( l'_{ij} \leftarrow \infty \)
6. for \( k \leftarrow 1 \) to \( n \) do
7. \( l'_{ij} \leftarrow \min(l'_{ij}, l'_{ik} + w_{kj}) \)
8. return \( L' \)

**SQUARE-MATRIX-MULTIPLY** (A, B)
1. \( n \leftarrow A.\text{rows} \)
2. let \( C = (c_{ij}) \) be a new \( n \times n \) matrix
3. for \( i \leftarrow 1 \) to \( n \) do
4. for \( j \leftarrow 1 \) to \( n \) do
5. \( c_{ij} \leftarrow 0 \)
6. for \( k \leftarrow 1 \) to \( n \) do
7. \( c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj} \)
8. return \( C \)

Both have the same \( \Theta(n^3) \) running time.
APSP: Extending SPs by One Edge at a Time

\[ \text{Running time} = \Theta(n^3) \]

**Extend-Shortest-Paths** \((L, W)\)

1. \( n \leftarrow L\).rows
2. let \( L' = (l'_{ij}) \) be a new \( n \times n \) matrix
3. for \( i \leftarrow 1 \) to \( n \) do
4.   for \( j \leftarrow 1 \) to \( n \) do
5.     \( l'_{ij} \leftarrow \infty \)
6.   for \( k \leftarrow 1 \) to \( n \) do
7.     \( l'_{ij} \leftarrow \min(l'_{ij}, l'_{ik} + w_{kj}) \)
8. return \( L' \)

**Slow-All-Pairs-Shortest-Paths** \((W)\)

1. \( n \leftarrow W\).rows
2. \( L^{(1)} \leftarrow W \)
3. for \( m \leftarrow 2 \) to \( n - 1 \) do
4.   let \( L^{(m)} \) be a new \( n \times n \) matrix
5. \( L^{(m)} \leftarrow \text{Extend-Shortest-Paths}(L^{(m-1)}, W) \)
6. return \( L^{(n-1)} \)

\[ \text{Running time} = n \times \Theta(n^3) = \Theta(n^4) \]
APSP: Extending SPs by Repeated Squaring

**Extend-Shortest-Paths** \((L, W)\)

1. \(n \leftarrow L\).rows
2. let \(L' = (l'_{ij})\) be a new \(n \times n\) matrix
3. \(\text{for } i \leftarrow 1 \text{ to } n \text{ do}\)
4. \(\text{for } j \leftarrow 1 \text{ to } n \text{ do}\)
5. \(l'_{ij} \leftarrow \infty\)
6. \(\text{for } k \leftarrow 1 \text{ to } n \text{ do}\)
7. \(l'_{ij} \leftarrow \min(l'_{ij}, l'_{ik} + w_{kj})\)
8. return \(L'\)

**Faster-All-Pairs-Shortest-Paths** \((W)\)

1. \(n \leftarrow W\).rows
2. \(L^{(1)} \leftarrow W\)
3. \(m \leftarrow 1\)
4. \(\text{while } m < n - 1 \text{ do}\)
5. let \(L^{(2m)}\) be a new \(n \times n\) matrix
6. \(L^{(2m)} \leftarrow \text{Extend-Shortest-Paths}(L^{(m)}, L^{(m)})\)
7. \(m \leftarrow 2m\)
8. return \(L^{(m)}\)
APSP: Extending SPs by Repeated Squaring

\textbf{Extend-Shortest-Paths} (L, W)

1. \( n \leftarrow L.\text{rows} \)
2. let \( L' = (l'_ij) \) be a new \( n \times n \) matrix
3. \textbf{for} \( i \leftarrow 1 \) \textbf{to} \( n \) \textbf{do}
4. \textbf{for} \( j \leftarrow 1 \) \textbf{to} \( n \) \textbf{do}
5. \( l'_{ij} \leftarrow \infty \)
6. \textbf{for} \( k \leftarrow 1 \) \textbf{to} \( n \) \textbf{do}
7. \( l'_{ij} \leftarrow \min(l'_{ij}, l'_{ik} + w_{kj}) \)
8. \textbf{return} \( L' \)

\textbf{Running time} \( = \Theta(n^3) \)

\textbf{Faster-All-Pairs-Shortest-Paths} (W)

1. \( n \leftarrow W.\text{rows} \)
2. \( L^{(1)} \leftarrow W \)
3. \( m \leftarrow 1 \)
4. \textbf{while} \( m < n - 1 \) \textbf{do}
5. let \( L^{(2m)} \) be a new \( n \times n \) matrix
6. \( L^{(2m)} \leftarrow \text{Extend-Shortest-Paths}(L^{(m)}, L^{(m)}) \)
7. \( m \leftarrow 2m \)
8. \textbf{return} \( L^{(m)} \)

\textbf{Running time} \( = \lfloor \log_2 (n - 1) \rfloor \times \Theta(n^3) = \Theta(n^3 \log n) \)
Let $d_{ij}^{(k)}$ be the minimum weight of any path from vertex $i$ to vertex $j$ for which all intermediate vertices are in $\{1,2,\ldots,k\}$. Then

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{if } k = 0, \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & \text{if } k \geq 1. \end{cases}$$

Then $D^{(n)} = (d_{ij}^{(n)})$ gives: $d_{ij}^{(n)} = \delta(i,j)$ for all $i,j \in G.V.$
APSP: Floyd-Warshall’s Algorithm

**FLOYD-WARSHALL** (W)

1. \( n \leftarrow W.\ rows \)
2. \( D^{(0)} \leftarrow W \)
3. \( \text{for } k \leftarrow 1 \text{ to } n \text{ do} \)
4. \( \text{let } D^{(k)} = \left( d_{ij}^{(k)} \right) \text{ be a new } n \times n \text{ matrix} \)
5. \( \text{for } i \leftarrow 1 \text{ to } n \text{ do} \)
6. \( \text{for } j \leftarrow 1 \text{ to } n \text{ do} \)
7. \( d_{ij}^{(k)} \leftarrow \min \left( d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) \)
8. \( \text{return } D^{(n)} \)
APSP: Floyd-Warshall with Predecessor Matrix

**FLOYD-WARSHALL ( W )**

1. $n \leftarrow W.rows$
2. $D^{(0)} \leftarrow W$
3. let $\Pi^{(0)} = (\pi_{ij}^{(0)})$ be a new $n \times n$ matrix
4. for $i \leftarrow 1$ to $n$ do
5.  for $j \leftarrow 1$ to $n$ do
6.    if $i = j$ or $w_{ij} = \infty$ then $\pi_{ij}^{(0)} \leftarrow NIL$
7.    else $\pi_{ij}^{(0)} \leftarrow i$
8. for $k \leftarrow 1$ to $n$ do
9.  let $D^{(k)} = (d_{ij}^{(k)})$ and $\Pi^{(k)} = (\pi_{ij}^{(k)})$ be new $n \times n$ matrices
10. for $i \leftarrow 1$ to $n$ do
11.   for $j \leftarrow 1$ to $n$ do
12.      if $d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ then $\pi_{ij}^{(k)} \leftarrow \pi_{ij}^{(k-1)}$
13.      else $\pi_{ij}^{(k)} \leftarrow \pi_{kj}^{(k-1)}$
14.      $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
15.   return $D^{(n)}$ and $\Pi^{(n)}$
**APSP: Floyd-Warshall with Predecessor Matrix**

```plaintext
Print-All-Pairs-Shortest-Path ( Π, i, j )

1. if i = j then
2. print i
3. elseif π_{i j} = NIL then
4. print "no path from" i "to" j "exists"
5. else Print-All-Pairs-Shortest-Path ( Π, i, π_{i j} )
6. print j
```
APSP: Floyd-Warshall with Predecessor Matrix

\[ D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \]

\[ \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix} \]
APSP: Floyd-Warshall with Predecessor Matrix

\[
D^{(0)} = \begin{pmatrix}
0 & 3 & 8 & \infty & -4 \\
\infty & 0 & \infty & 1 & 7 \\
\infty & 4 & 0 & \infty & \infty \\
2 & \infty & -5 & 0 & \infty \\
\infty & \infty & 8 & 6 & 0 \\
\end{pmatrix}
\]

\[
\Pi^{(0)} = \begin{pmatrix}
\text{NIL} & 1 & 1 & \text{NIL} & 1 \\
\text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\
\text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\
4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\
\text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \\
\end{pmatrix}
\]

\[
D^{(1)} = \begin{pmatrix}
0 & 3 & 8 & \infty & -4 \\
\infty & 0 & \infty & 1 & 7 \\
\infty & 4 & 0 & \infty & \infty \\
2 & 5 & -5 & 0 & -2 \\
\infty & \infty & 8 & 6 & 0 \\
\end{pmatrix}
\]

\[
\Pi^{(1)} = \begin{pmatrix}
\text{NIL} & 1 & 1 & \text{NIL} & 1 \\
\text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\
\text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\
4 & 1 & 4 & \text{NIL} & 1 \\
\text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \\
\end{pmatrix}
\]
**APSP: Floyd-Warshall with Predecessor Matrix**

\[ D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \]

\[ \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix} \]

\[ D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \]

\[ \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix} \]
### APSP: Floyd-Warshall with Predecessor Matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NIL</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>NIL</td>
<td>NIL</td>
<td>NIL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>NIL</td>
<td>3</td>
<td>NIL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>NIL</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>NIL</td>
<td>NIL</td>
<td>NIL</td>
<td>5</td>
<td>NIL</td>
</tr>
</tbody>
</table>

The initial distance matrix is:

$$ D^{(2)} = \begin{pmatrix}
0 & 3 & 8 & 4 & -4 \\
\infty & 0 & \infty & 1 & 7 \\
\infty & 4 & 0 & 5 & 11 \\
2 & 5 & -5 & 0 & -2 \\
\infty & \infty & \infty & 6 & 0
\end{pmatrix} $$

The predecessor matrix is:

$$ \Pi^{(2)} = \begin{pmatrix}
\infty & \infty & \infty & 6 & 0
\end{pmatrix} $$

The updated distance matrix is:

$$ D^{(3)} = \begin{pmatrix}
0 & 3 & 8 & 4 & -4 \\
\infty & 0 & \infty & 1 & 7 \\
\infty & 4 & 0 & 5 & 11 \\
2 & -1 & -5 & 0 & -2 \\
\infty & \infty & \infty & 6 & 0
\end{pmatrix} $$

The updated predecessor matrix is:

$$ \Pi^{(3)} = \begin{pmatrix}
\infty & \infty & \infty & 6 & 0
\end{pmatrix} $$
APSP: Floyd-Warshall with Predecessor Matrix

\[ D^{(3)} = \begin{pmatrix}
  0 & 3 & 8 & 4 & -4 \\
  \infty & 0 & \infty & 1 & 7 \\
  \infty & 4 & 0 & 5 & 11 \\
  2 & -1 & -5 & 0 & -2 \\
  \infty & \infty & \infty & 6 & 0 \\
\end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix}
  \text{NIL} & 1 & 1 & 2 & 1 \\
  \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\
  \text{NIL} & 3 & \text{NIL} & 2 & 2 \\
  4 & 3 & 4 & \text{NIL} & 1 \\
  \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \\
\end{pmatrix} \]

\[ D^{(4)} = \begin{pmatrix}
  0 & 3 & -1 & 4 & -4 \\
  3 & 0 & -4 & 1 & -1 \\
  7 & 4 & 0 & 5 & 3 \\
  2 & -1 & -5 & 0 & -2 \\
  8 & 5 & 1 & 6 & 0 \\
\end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix}
  \text{NIL} & 1 & 4 & 2 & 1 \\
  4 & \text{NIL} & 4 & 2 & 1 \\
  4 & 3 & \text{NIL} & 2 & 1 \\
  4 & 3 & 4 & \text{NIL} & 1 \\
  4 & 3 & 4 & 5 & \text{NIL} \\
\end{pmatrix} \]
APSP: Floyd-Warshall with Predecessor Matrix

\[ D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \]

\[ \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix} \]

\[ D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \]

\[ \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix} \]
APSP: Floyd-Warshall’s Algorithm

**FLOYD-WARSHALL (W)**

1. \( n \leftarrow W\). rows
2. \( D^{(0)} \leftarrow W \)
3. \( \text{for } k \leftarrow 1 \text{ to } n \text{ do} \)
4. \( \text{let } D^{(k)} = (d_{ij}^{(k)}) \text{ be a new } n \times n \text{ matrix} \)
5. \( \text{for } i \leftarrow 1 \text{ to } n \text{ do} \)
6. \( \text{for } j \leftarrow 1 \text{ to } n \text{ do} \)
7. \( d_{ij}^{(k)} \leftarrow \min \left( d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) \)
8. \( \text{return } D^{(n)} \)

Running Time = \( \Theta(n^3) \)
Space Complexity = \( \Theta(n^3) \)
APSP: Floyd-Warshall’s Algorithm

But $D^{(k)}$ depends only on $D^{(k-1)}$.

\[ FLOYD\text{-}WARSHALL\text{-}QUADRATIC\text{-}SPACE (W) \]

1. $n \leftarrow W.rows$
2. let $D^{(0)} = (d_{ij}^{(0)})$ and $D^{(1)} = (d_{ij}^{(1)})$ be new $n \times n$ matrices
3. $D^{(0)} \leftarrow W$
4. for $k \leftarrow 1$ to $n$ do
5.   for $i \leftarrow 1$ to $n$ do
6.     for $j \leftarrow 1$ to $n$ do
7.       $d_{ij}^{(1)} \leftarrow \min(d_{ij}^{(0)}, d_{ik}^{(0)} + d_{kj}^{(0)})$
8.     $D^{(0)} \leftarrow D^{(1)}$
9. return $D^{(0)}$

Running Time = $\Theta(n^3)$
Space Complexity = $\Theta(n^2)$
APSP: Floyd-Warshall’s Algorithm

Can be solved in-place!

```
FLOYD-WARSHALL-IN-PLACE ( W )
1. n ← W. rows
2. for k ← 1 to n do
3.     for i ← 1 to n do
4.         for j ← 1 to n do
5.             w_{ij} ← \min(w_{ij}, w_{ik} + w_{kj})
6.     return W
```

Running Time = $\Theta(n^3)$
Space Complexity = $\Theta(n^2)$