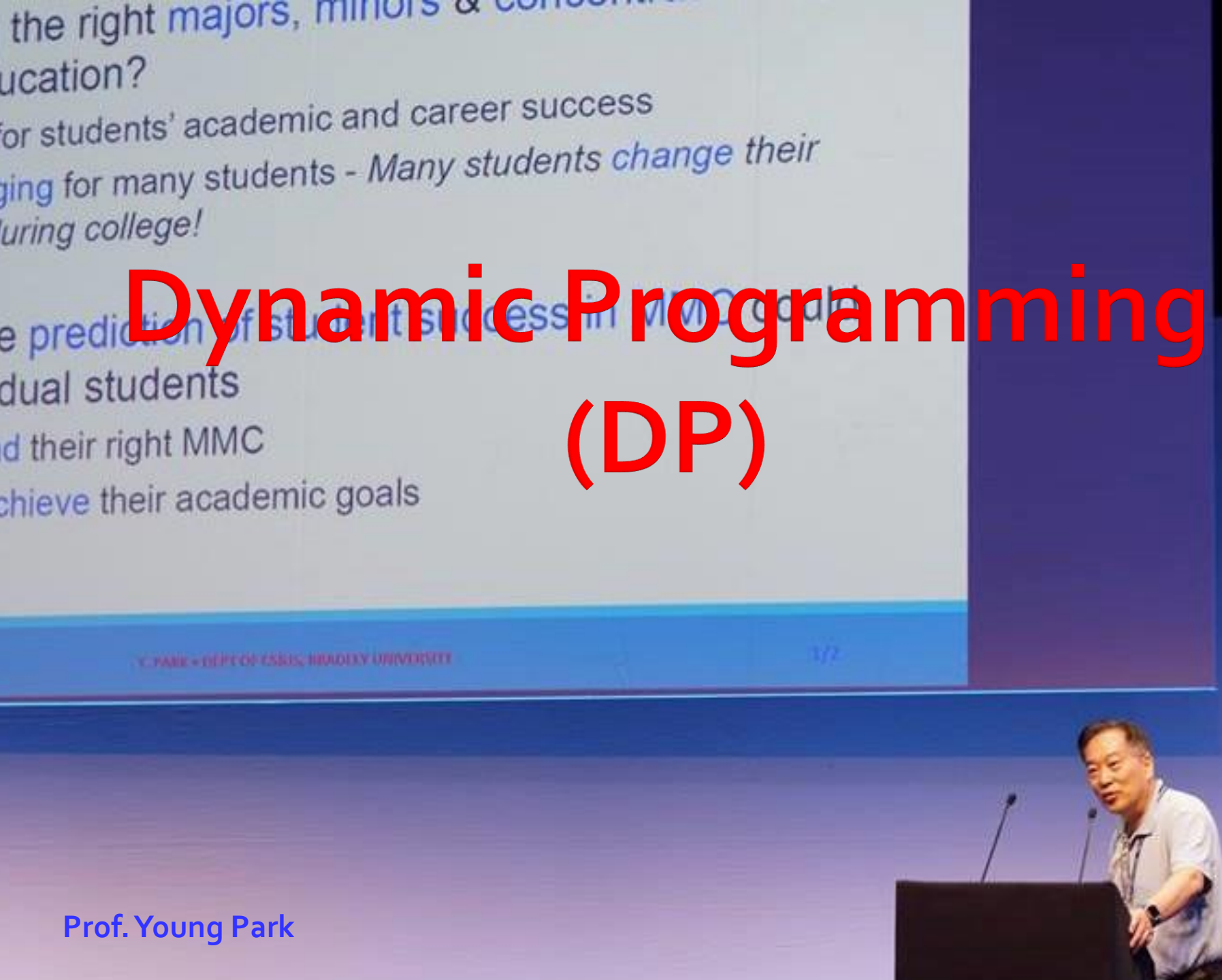
Dynamic Programming (DP)



Prof. Young Park

✓ When D&C Not Efficient?

Problems Solving via Divide-and-Conquer (D&C)

- **Binary Search**
- **Merge Sort**
- **Quick Sort**
- **Matrix Multiplication**

- N-th Fibonacci Term Computing

Computing N-th Fibonacci Term via D&C

- (0,) 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- **Algorithm 1.6** N-th Fibonacci Term (Recursive)
  - **Divide-and-Conquer**
  - **Top-down**
  - **Exponential time $O(2^N)$**
- Fib(n) = Fib(n – 1) + Fib(n – 2) for n >= 2;
- Fib(0) =0
- Fib(1) = 1
- Example: Fib(5)?
- 5

Computing N-th Fibonacci Term via D&C

- In problems where smaller instances are related,
  - A D&C algorithm often ends up repeatedly solving common instances.
  - The result is very inefficient!

- **Overlapping subproblems**

When D&C Not Efficient

- D&C algorithms should be avoided in the following two cases:
  - An instance of size *n* is divided into two or more instances each **almost of size *n***.
  - An instance of size *n* is divided into **almost *n* instances** of size *n/c*, where *c* is a constant.

Then, WHAT? **DP!**

QUIZ?

- D&C algorithms should be avoided in the following two cases?
  - An instance of size *n* is divided into two or more instances each **almost of size *n***.
  - An instance of size *n* is divided into **almost *n* instances** of size *n/c*, where *c* is a constant.

✓Dynamic Programming (DP)

- An approach/paradigm to designing algorithms!

Dynamic Programming (DP)

- Similar to the D&C approach
  - An instance of a problem is divided into smaller instances.
- But, in DP:
  - Solve the small instances first,
  - Store the results, and
  - Look them up when we need them instead of recomputing them. (memoization!)

Dynamic Programming

- A **bottom-up** approach since the solution is constructed from the bottom up in an array or sequence of arrays.
  - Memoization!
  - **DP= Recursion + Memoization**
    = Tabulation

Two Steps in Dynamic Programming (DP)

- First: **Establish a recursive property** that gives the solution to an instance of the problem.
- Second: **Solve** an instance of the problem **in a bottom-up fashion** by solving and storing smaller instances first.

✓ Dynamic Programming (DP)-based Algorithms

Dynamic Programming (DP)-based Algorithms

1. N-th Fibonacci Term via DP
2. Binomial Coefficient via DP
3. Floyd & Warshall's Algorithm - APSP via DP
4. Bellman & Ford's Algorithm – SSSP via DP
5. Chained Matrix Multiplication via DP
6. Traveling Salesperson Problem via DP
7. The 0-1 Knapsack Problem via DP

1. N-th Fibonacci Term

- (0,) 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

N-th Fibonacci Term via D&C

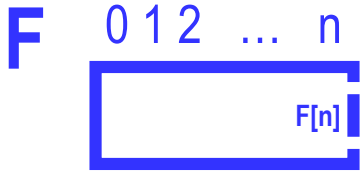
- (0,) 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- **Algorithm 1.6** N-th Fibonacci Term (Recursive)
  - **Divide-and-Conquer**
  - **Top-down**
  - **Exponential time $O(2^N)$**
- Fib(n) = Fib(n – 1) + Fib(n – 2) for n >= 2;
- Fib(0) =0
- Fib(1) = 1
- Example: Fib(5)?
- 5

N-th Fibonacci Term via DP

- A more efficient algorithm using DP!

N-th Fibonacci Term via DP

- (0,) 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- **Algorithm 1.7** N-th Fibonacci Term (Iterative)
  - **DP**
  - **Bottom-up**
  - **Linear-time O(n) algorithm!**
- Recursive Property
  - **The Array F**
    - F[i] = F[i – 1] + F[i – 2] for n >= 2;
    - F[0]=0
    - F[1] = 1
- Example: Fib(5)?
  - F[5]?
  - 5



QUIZ?

- N-th Fibonacci Term via DP?
  - Algorithm 1.7 nth Fibonacci Term (Iterative)
    - **DP**
    - Bottom-up
    - Linear-time algorithm!

N-th Fibonacci Term via DP Visualization

- ***N-th Fibonacci Term via DP Visualization***

2. The Binomial Coefficient

- **n choose k** for o<=k<=n

  C(n,k) = n! / k! (n-k)!

- Example: C(4,2)?
  - 6

- We cannot compute the binomial coefficient directly from this definition because *n*! is very large, even for moderate values of *n*.

The Binomial Coefficient via D&C

- We should eliminate the need to compute *n*! or *k*! by using **the recursive property**:

$$C(n,k) = C(n-1,k-1) + C(n-1,k) \quad \text{if } n > k \text{ and } k > 0$$
$$= 1 \quad \text{if } k = 0 \text{ or } n = k$$

- How?
  - Apply **D&C**!
  - Recursive calls!

The Binomial Coefficient via D&C

- **Algorithm 3.1** Binomial Coefficient using **Divide-and-Conquer**
- Example: C(4,2)?
  - 6
- Very inefficient - Factorial!

The Binomial Coefficient via DP

- A more efficient algorithm using DP!

The Binomial Coefficient via DP

- Idea?
- Step 1: Establish a recursive property.
- **The Array B**

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & \text{for } 0 < j < i \\ 1 & \text{for } j = 0 \text{ or } j = i \end{cases}$$

- Step 2: Solve an instance of the problem in a bottom-up fashion by computing values for B[i][j] starting from B[0][0].

The Array B

B

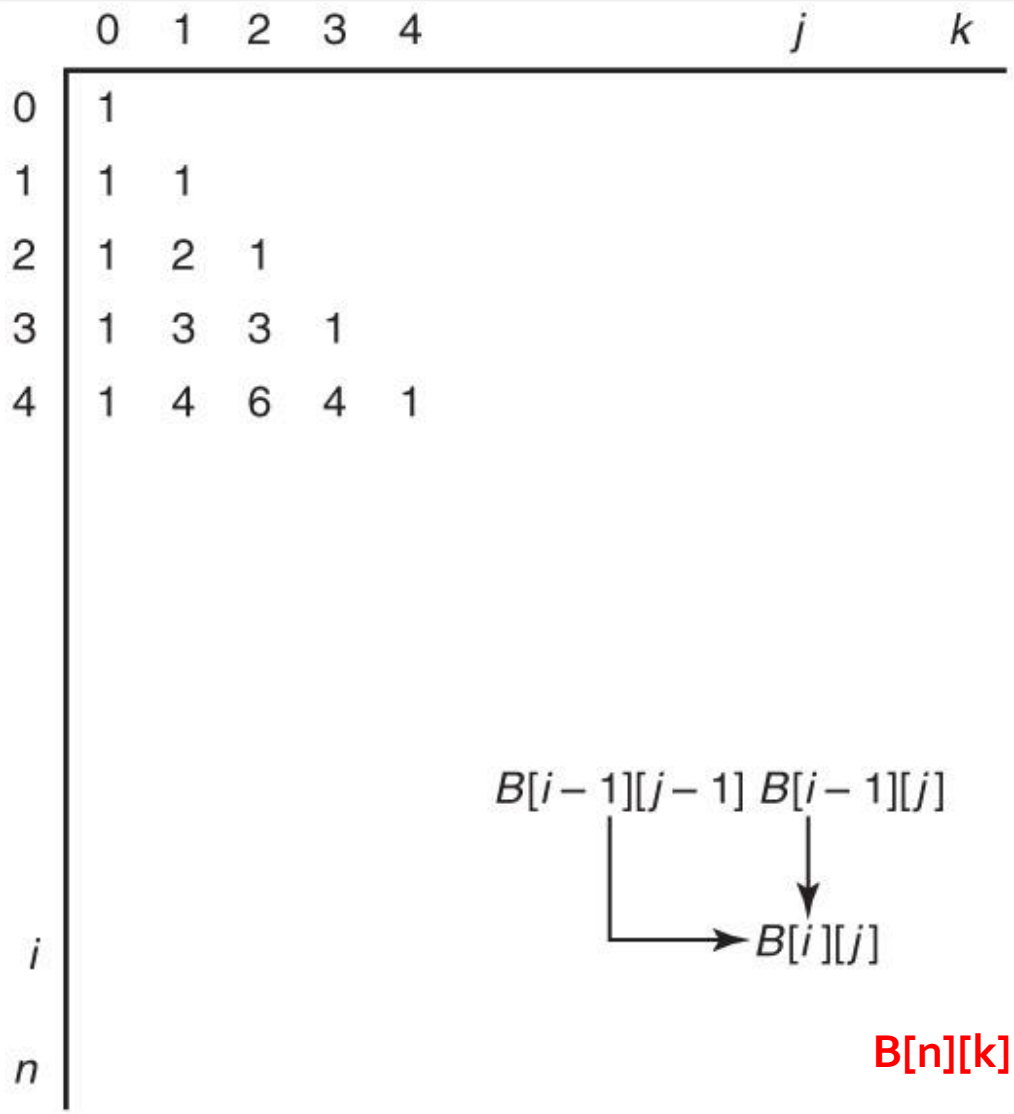


Figure 3.1

B[n][k]

The Binomial Coefficient via DP

- Example: C(4,2)?
  - B[4][2]?
  - 6
  - Example 3.1

The Binomial Coefficient via DP

- **Algorithm 3.2** Binomial Coefficient using **Dynamic Programming**
- Analysis of Algorithm 3.2
  - **O(nk)**
  - **$O(n^2)$**

DP vs D&C

- By using DP instead of D&C, we have developed a much more efficient algorithm!
- In DP, we use the recursive property to iteratively solve the instances in sequence, starting with the smallest instance.
  - In this way we solve the smallest instance just once.
- DP is a good technique to try when D&C leads to an inefficient algorithm!

3. The All-Pairs Shortest Paths Problem

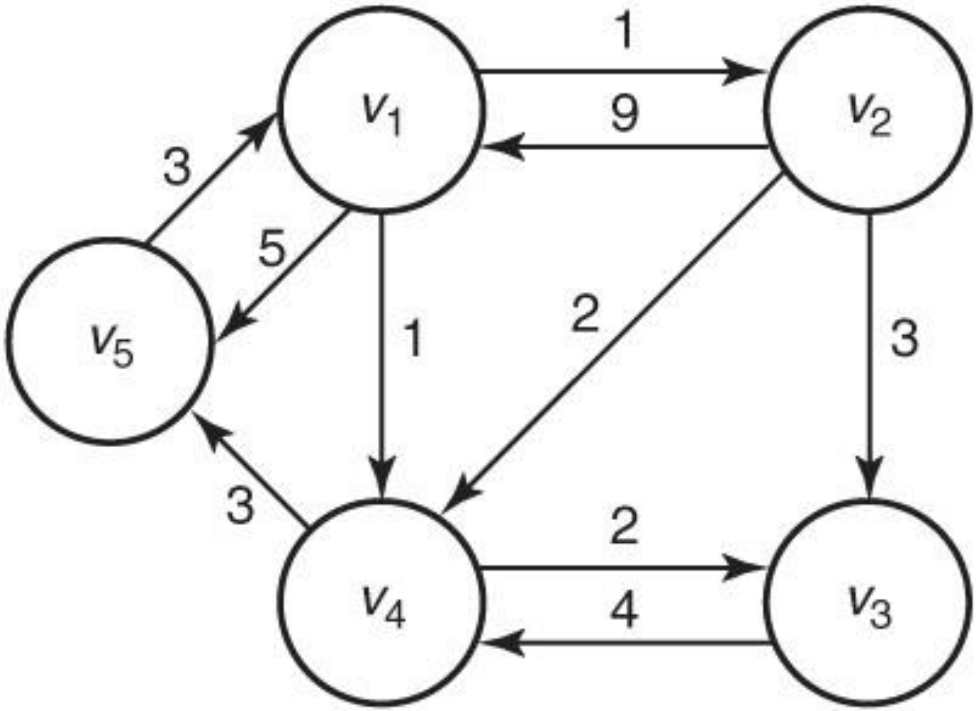
- A problem that has many applications is finding the shortest path from each vertex to all the other vertices.
- The Shortest Paths problem is an **optimization problem**.
  - There can be more than one candidate solution to an instance of an optimization problem.
  - Each candidate solution has a value associated with it.
  - A solution to the instance is any candidate solution that has an optimal value.

The All-Pairs Shortest Paths Problem

- Given a weighted, directed graph G=(V,E) with weight function W mapping edges to **nonnegative or negative weights**, **but** there is **no negative cycle**
- Find a shortest path from u to v for every pair of vertices u and v in V.
  - The **All-Pairs Shortest Paths** (APSP) Problem

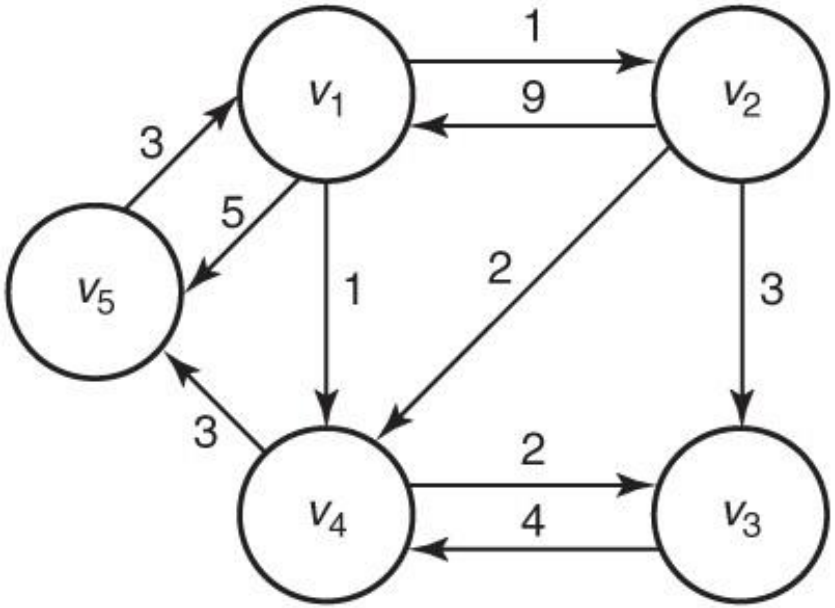
Example: An Edge-Weighted, Directed Graph

G:



Example: Adjacency Matrix Representation

G:



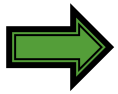
	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

From Adjacency Matrix to Path Length Array D

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W



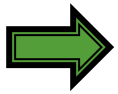
	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

From Adjacency Matrix to Path Array P

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W



	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

P

The All-Pairs Shortest Paths Problem

- The **brute-force** algorithm is to consider all possible paths and take the shortest.
- This is a very inefficient method – factorial!

APSP via DP

- A more efficient algorithm using DP!

Floyd's Algorithm - APSP via DP

- Using DP, we can create a cubic-time $O(n^3)$ algorithm!
  - First, we develop an algorithm that determines the lengths of only the shortest paths.
  - After that, we modify it to produce shortest paths as well.
- **Floyd's (&Warshall's) Algorithm**

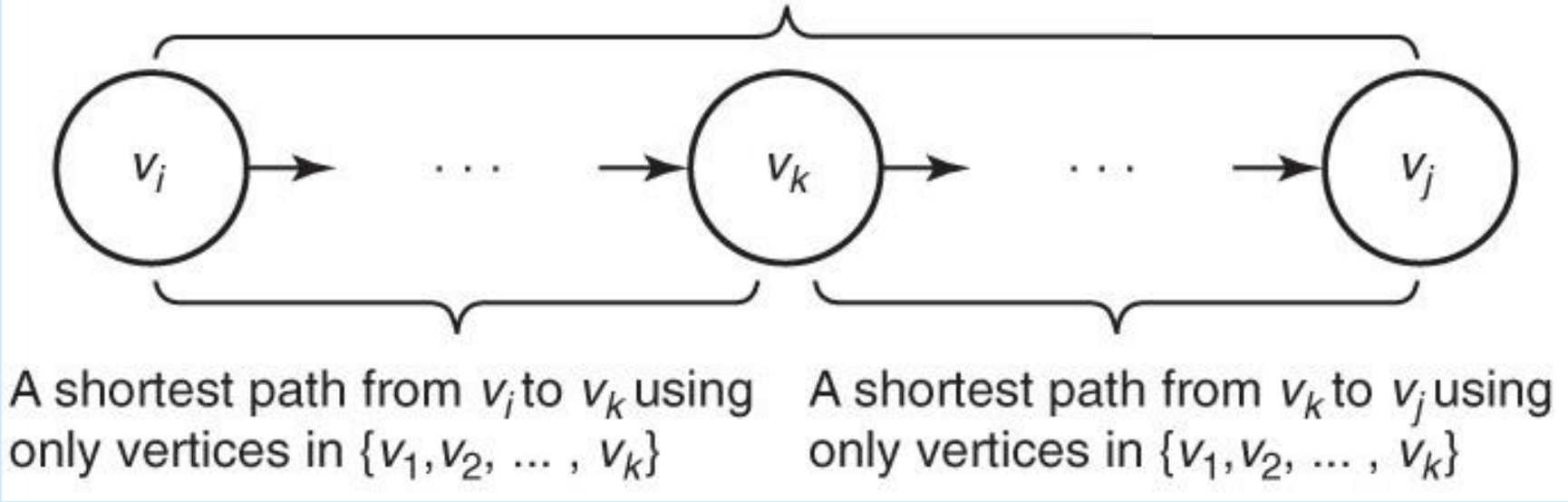
Floyd & Warshall's Algorithm

- Idea?
  - V = { 1, 2, …, **|V|=n** }
  - **D^K [i][j]** = The weighted length of the shortest path from the vertex i to the vertex j & The shortest path contains **no intermediate vertex of index greater than K**.
    - D^1 [i][j]
    - D^2 [i][j]
    - .
    - .
    - .
    - **$D^{|V|=n}$ [i][j]**

Floyd & Warshall's Algorithm

- Establish a recursive property.

$$D^{K}[i][j] = \min\left(D^{K-1}[i][j],\ D^{K-1}[i][k] + D^{K-1}[k][j]\right)$$



Floyd & Warshall's Algorithm

- Initially,
  - **$D^{initial}$ [i][j]** = The weighted length of the shortest path from the vertex i to the vertex j & The shortest path contains no intermediate vertex of index greater than k.
    - weight(i,j) =
      - 0 if i=j
      - The weight of the directed edge (i,j) if (i,j) in E
      - ∞ otherwise

Floyd & Warshall's Algorithm

$D^{initial(0)}$

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

Floyd & Warshall's Algorithm

- $D^{initial}$ [i][j]
- D^{1} [i][j]
- D^{2} [i][j]
- .
- .
- .
- $D^{|V|=n}$ [i][j]

Floyd & Warshall's Algorithm

- We can compute D^K from D^{K-1}!

$$D^K[i][j] = \min (D^{K-1}[i][j] , D^{K-1}[i][\kappa] + D^{K-1}[\kappa][j])$$

$$D^K[i][j] = \min (D^{K-1}[i][j] , D^{K-1}[i][\kappa] + D^{K-1}[\kappa][j])$$

QUIZ?

- Compute D^K from D^{K-1} ?

$$D^K[i][j] = \min (D^{K-1}[i][j] , D^{K-1}[i][K] + D^{K-1}[K][j])$$

Floyd & Warshall's Algorithm

- Example 3.2
  - $D^{(5)}[5][3]$?

Floyd & Warshall's Algorithm

- Example 3.3
- Compute
  - $D^{(1)}$
  - $D^{(2)}$
  - $D^{(3)}$
  - $D^{(4)}$
  - $D^{(5)}$

Floyd & Warshall's Algorithm

$D^{(5)}$

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

Floyd & Warshall's Algorithm-Analysis

- **Algorithm 3.3 Floyd's Algorithm for APSP**
  - Just lengths of the shortest paths for now!
  - The shortest paths length array D
- Analysis of Algorithm 3.3 Floyd's Algorithm for APSP
  - **$O(|V|^3)$**

Floyd & Warshall's Algorithm– Shortest Paths

- **Algorithm 3.4 Floyd's Algorithm for APSP**
  - The shortest paths length array D
  - The path array P to produce shortest paths!
  - **P[i][j] = Highest index of an intermediate vertex on the shortest path from I to j & o if no intermediate vertex**
  - **O(|V|³)**
  - Figure 3.5

Floyd & Warshall's Algorithm

- $P^{initial}[i][j]$
- $P^{1}[i][j]$
- $P^{2}[i][j]$
- .
- .
- .
- $P^{|V|=n}[i][j]$

$$D^{K}[i][j] = \min(D^{K-1}[i][j], D^{K-1}[i][k] + D^{K-1}[k][j])$$

Floyd & Warshall's Algorithm

- The Path Array P:

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

Floyd & Warshall's Algorithm

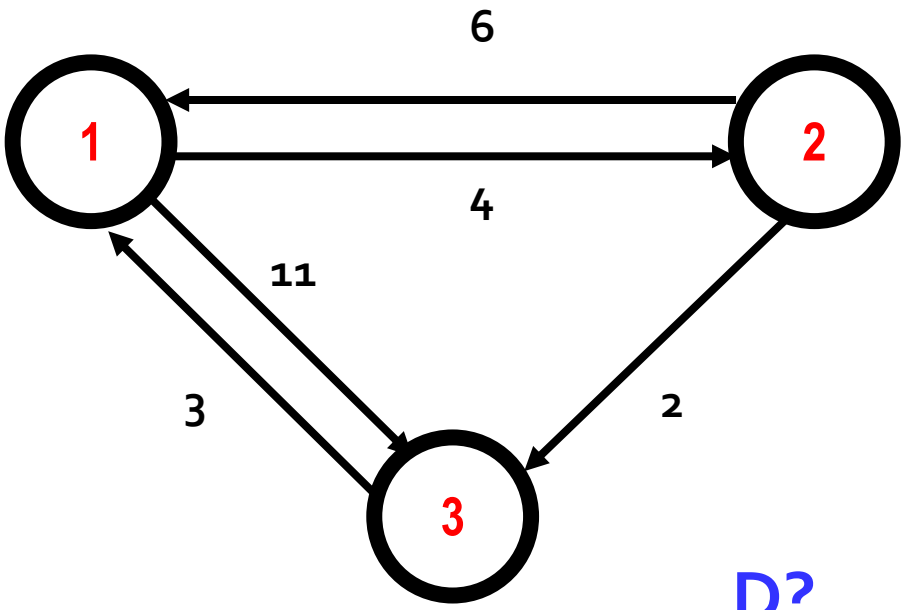
- **Algorithm 3.5 Print Shortest Path**
  - Print the shortest path using the path array
- Analysis of Algorithm 3.5 Print Shortest Path
  - **O(|V|)**

Floyd & Warshall's Algorithm

- Path(v5,v3) => v5 v1 v4 v3

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

Example: APSP by Floyd & Warshall's Algorithm



W

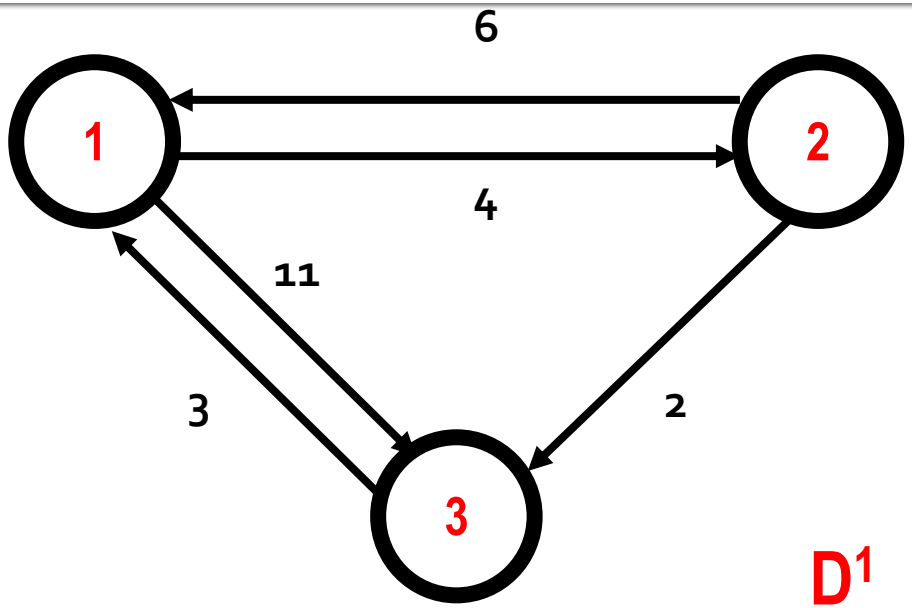
	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

D?

P?

Shortest paths?

Example: APSP by Floyd & Warshall's Algorithm



$D^{initial}$

	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

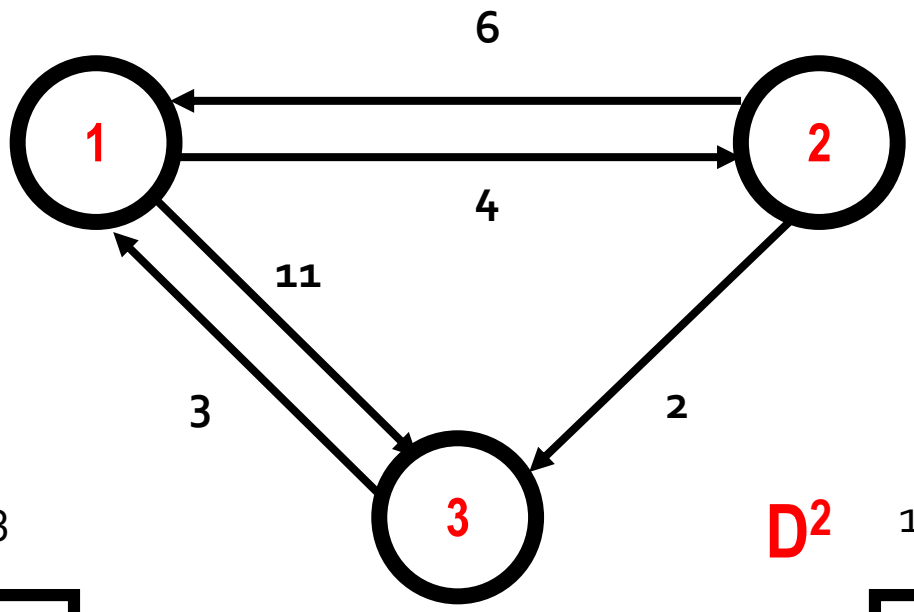
D^1

	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

P^1

	1	2	3
1	0	0	0
2	0	0	0
3	0	1	0

Example: APSP by Floyd & Warshall's Algorithm



D^1

	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

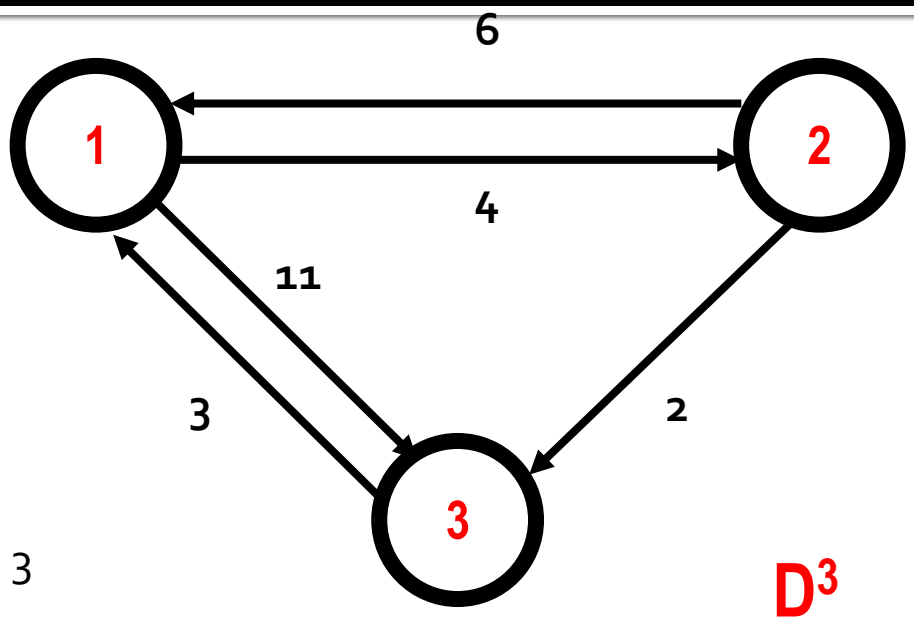
D^2

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

P^2

	1	2	3
1	0	0	2
2	0	0	0
3	0	1	0

Example: APSP by Floyd & Warshall's Algorithm



D^2

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

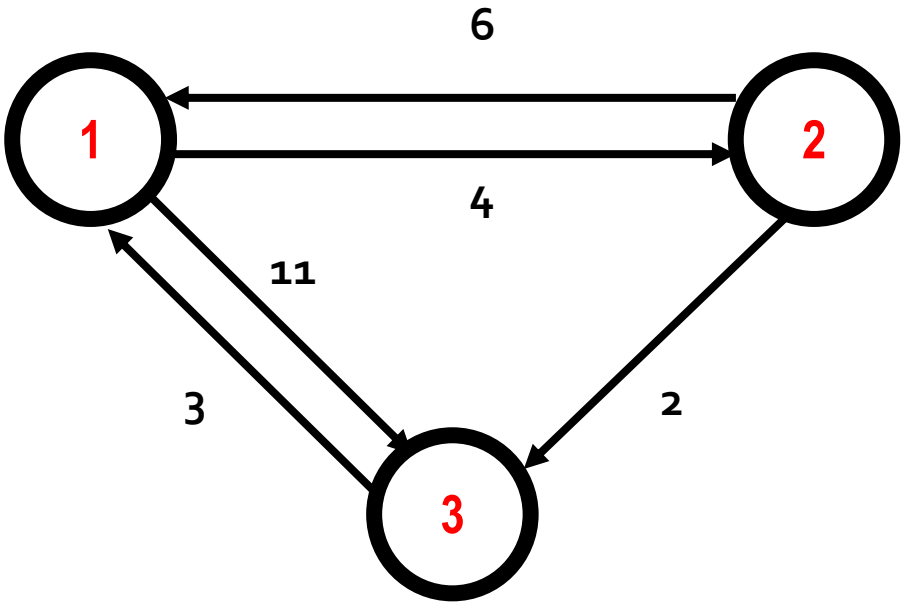
D^3

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

P^3

	1	2	3
1	0	0	2
2	3	0	0
3	0	1	0

Example: APSP by Floyd & Warshall's Algorithm



$D^{initial}$

	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

D^{1}

	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

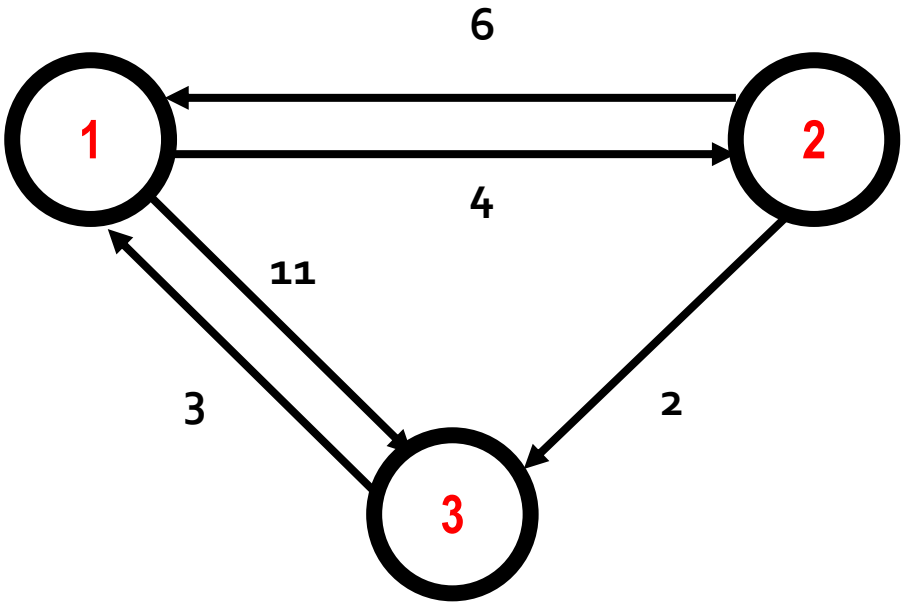
D^{2}

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

D^{3}

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

Example: APSP by Floyd & Warshall's Algorithm



$P^{initial}$

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0

P^{1}

	1	2	3
1	0	0	0
2	0	0	0
3	0	1	0

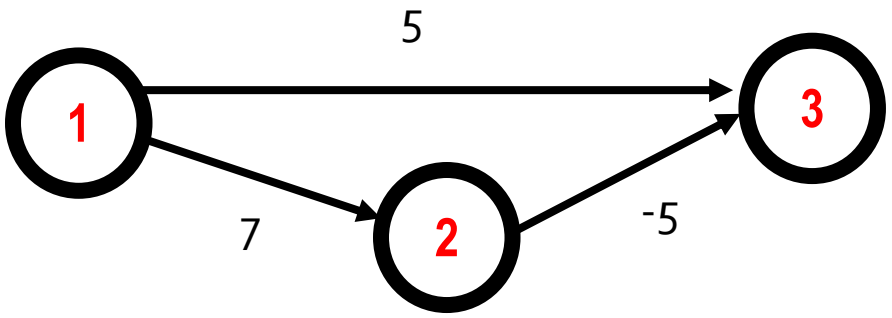
P^{2}

	1	2	3
1	0	0	2
2	0	0	0
3	0	1	0

P^{3}

	1	2	3
1	0	0	2
2	3	0	0
3	0	1	0

QUIZ? APSP by Floyd & Warshall's Algorithm

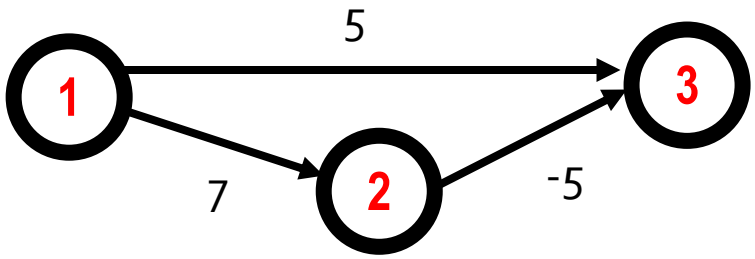


D?

P?

Shortest paths?

▶ QUIZ: APSP by Floyd & Warshall's Algorithm



$P^{initial}$

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0

P^{1}

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0

P^{2}

	1	2	3
1	0	0	2
2	0	0	0
3	0	0	0

P^{3}

	1	2	3
1	0	0	2
2	0	0	0
3	0	0	0

$D^{initial}$

	1	2	3
1	0	7	5
2	∞	0	-5
3	∞	∞	0

D^{1}

	1	2	3
1	0	7	5
2	∞	0	-5
3	∞	∞	0

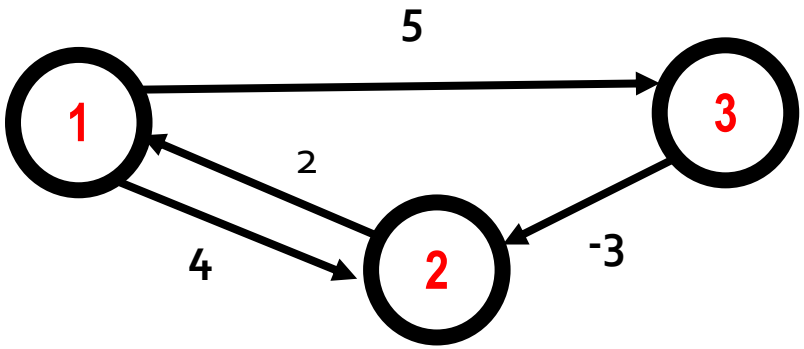
D^{2}

	1	2	3
1	0	7	2
2	∞	0	-5
3	∞	∞	0

D^{3}

	1	2	3
1	0	7	2
2	∞	0	-5
3	∞	∞	0

▶ QUIZ? APSP by Floyd & Warshall's Algorithm

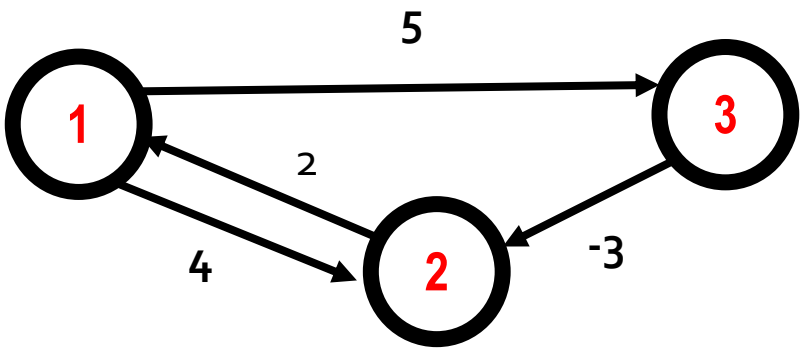


D?

P?

Shortest paths?

▶ QUIZ: APSP by Floyd & Warshall's Algorithm



$D^{initial}$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

D^{3}

	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

$P^{initial}$

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0

P^{3}

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

Floyd & Warshall's APSP Algorithm via DP Visualization

- ***Floyd & Warshall's APSP Algorithm via DP Visualization***

✓ DP for Optimization Problems

DP and Optimization Problems

- Floyd & Warshall's Algorithm - APSP via DP
- Bellman & Ford's Algorithm – SSSP via DP
- Chained Matrix Multiplication via DP
- Traveling Salesperson Problem via DP
- The 0-1 Knapsack Problem vis DP

DP and Optimization Problems

- Although it may seem that any optimization problem can be solved using dynamic programming, it is not true.

Principle of Optimality

- The **principle of optimality**:
  - An **optimal solution** to an instance of a problem always contains **optimal solutions** to **all subinstances**.

or

- **Optimal Substructure Property**

Principle of Optimality for DP

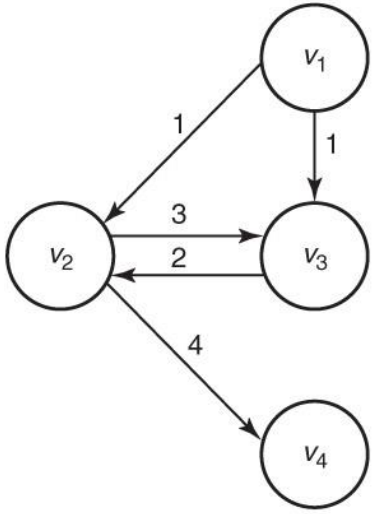
- The principle of optimality must apply in the optimization problem **to use DP**!
- The **shortest paths problem** apply!
  - If v_k is a node on an optimal path from v_i to v_j then **the sub-paths v_i to v_k and v_k to v_j are also optimal paths**.

Principle of Optimality

- The principle of optimality does not apply in every problem!

QUIZ? Longest Path Problem

- Example 3.4 The **longest paths problem**
  - The longest path **V1 -> V4**?



  - The principle of optimality does not apply! **No** DP!
  - Actually, **NP-complete problem**!

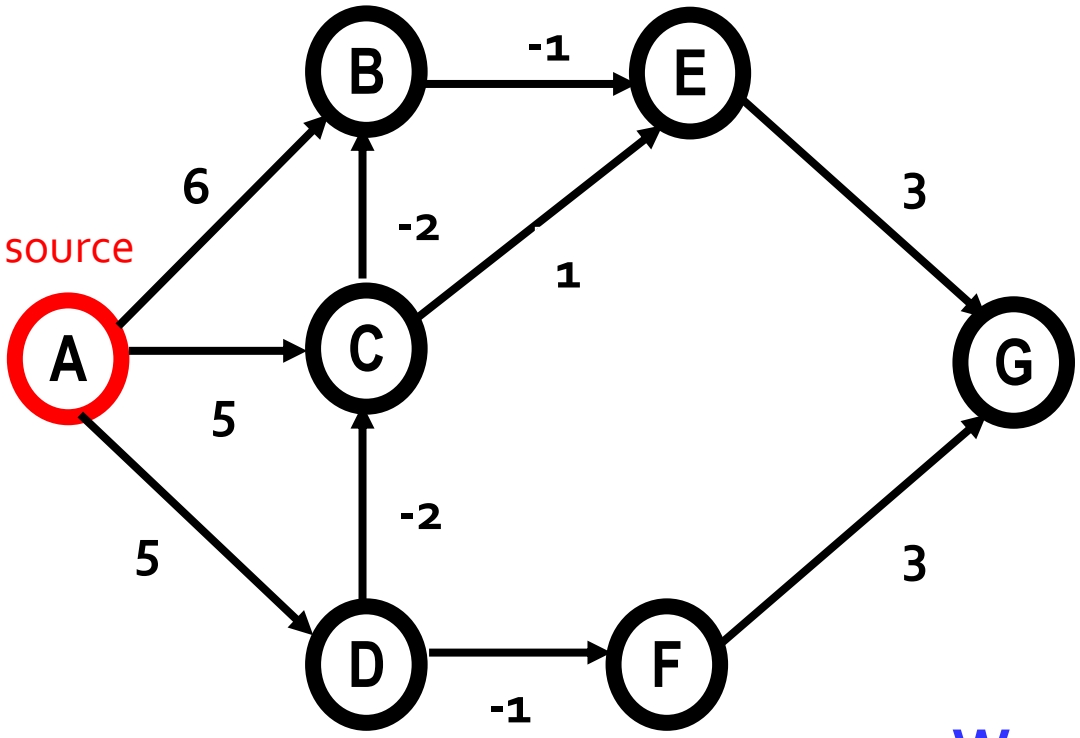
QUIZ?

- DP for The longest paths problem?

4. The Single-Source Shortest-Paths Problem with Negative Edges but No Negative Cycles

- Problem:
  - Given a weighted, directed graph G=(V,E) with weight function W mapping edges to real valued weights that may contain **negative edges** **but** there is **no negative cycle**,
  - Find a shortest path from a given source vertex s to every vertex v in V.
  - The single-source/all-destination shortest-paths problem!

Example: SSSP by Bellman & Ford's Algorithm



W

Distance?
Path?
Shortest paths?

The Single Source Shortest Paths Problem

- The **brute-force** algorithm is to consider all possible paths and take the shortest.
- This is a very inefficient method – factorial!

SSSP via DP

- A more efficient algorithm using DP!

The Single-Source Shortest-Paths with Negative Edges but No Negative Cycles Algorithm

- **Bellman & Ford's algorithm**
- **DP** (Dynamic Programming)!

Bellman & Ford's Algorithm

- Observation:
  - If there is no cycle of negative weight in a edge-weighted directed graph G = (V,E) then
  - There exists a shortest path between any two vertices that has **at most |V|-1 edges**!

Bellman & Ford's Algorithm

- Idea?
  - **Distance^{K} [v]** = The weighted length of the shortest path from the source vertex s to v & The shortest path contains **at most K edges**.
    - Distance^{1} [v]
    - Distance^{2} [v]
    - Distance^{3} [v]
    - .
    - .
    - .
    - **Distance$^{|V|-1}$ [v]**

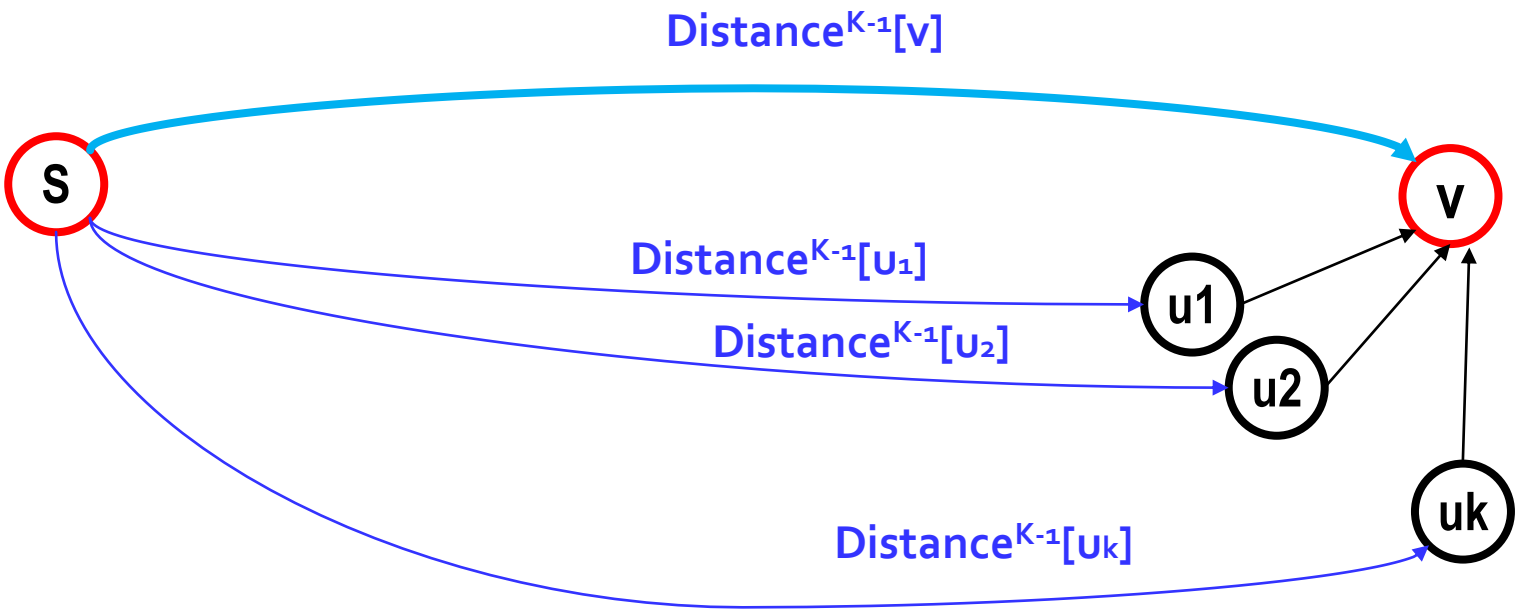
Bellman & Ford's Algorithm

- For each vertex v, maintains
  - **Distance^K[v]** = The weighted length of the shortest path from the source vertex s to v & The shortest path contains at most K edges.
  - **Path[v]** = The predecessor of the vertex v on the shortest path from the source vertex s to v.

Bellman & Ford's Algorithm

- Establish a **recursive property**.

Distance ^{K}[v] ?



Bellman & Ford's Algorithm

- **Distance ^{K}[v] ?**
  - For each vertex **v** with an edge **(u,v)**,
    - Compute Distance $^{K-1}$[u] + weight (u,v).
    - Select the vertex **u'** whose Distance $^{K-1}$[u] + weight (u,v) is the smallest.
  - If **Distance^{K-1}[v]** > Distance^{K-1}[u'] + weight (u',v)) then
    - **Distance ^{K}[v]** = Distance $^{K-1}$[u'] + weight (u',v)
    - **Path ^{K}[v]** = **u'**

Bellman & Ford's Algorithm

- The **recursive property**.

$$
\begin{aligned}
&\text{Distance}^{K}[v] = \\
&\min \\
&(\\
&\quad \text{Distance}^{K-1}[v], \\
&\quad \\
&\quad \min\,(\text{Distance}^{K-1}[u] + \text{weight}\,(u,v) \\
&\qquad \text{for all } u \text{ with edge } (u,v)) \\
&)
\end{aligned}
$$

Bellman & Ford's Algorithm

- We can compute **Distance^{K}[v]** from **Distance^{K-1}[v]!**

$$\text{Distance}^{K}[v] = \min\left(\text{Distance}^{K-1}[v],\ \min\left(\text{Distance}^{K-1}[u] + \text{weight}(u,v) \text{ for all } u \text{ with edge } (u,v)\right)\right)$$

Bellman & Ford's Algorithm

- Initially,
  - $Distance^{initial}[v]$
    - 0 if s=v
    - ∞ otherwise
  - $Path^{initial}[v]$ = unknown for all v in V.

Bellman & Ford's Algorithm

- $Distance^{initial}$ [v]
- $Distance^{1}$ [v]
- $Distance^{2}$ [v]
- $Distance^{3}$ [v]
- .
- .
- .
- $Distance^{|V|-1}$ [v]

Bellman & Ford's Algorithm

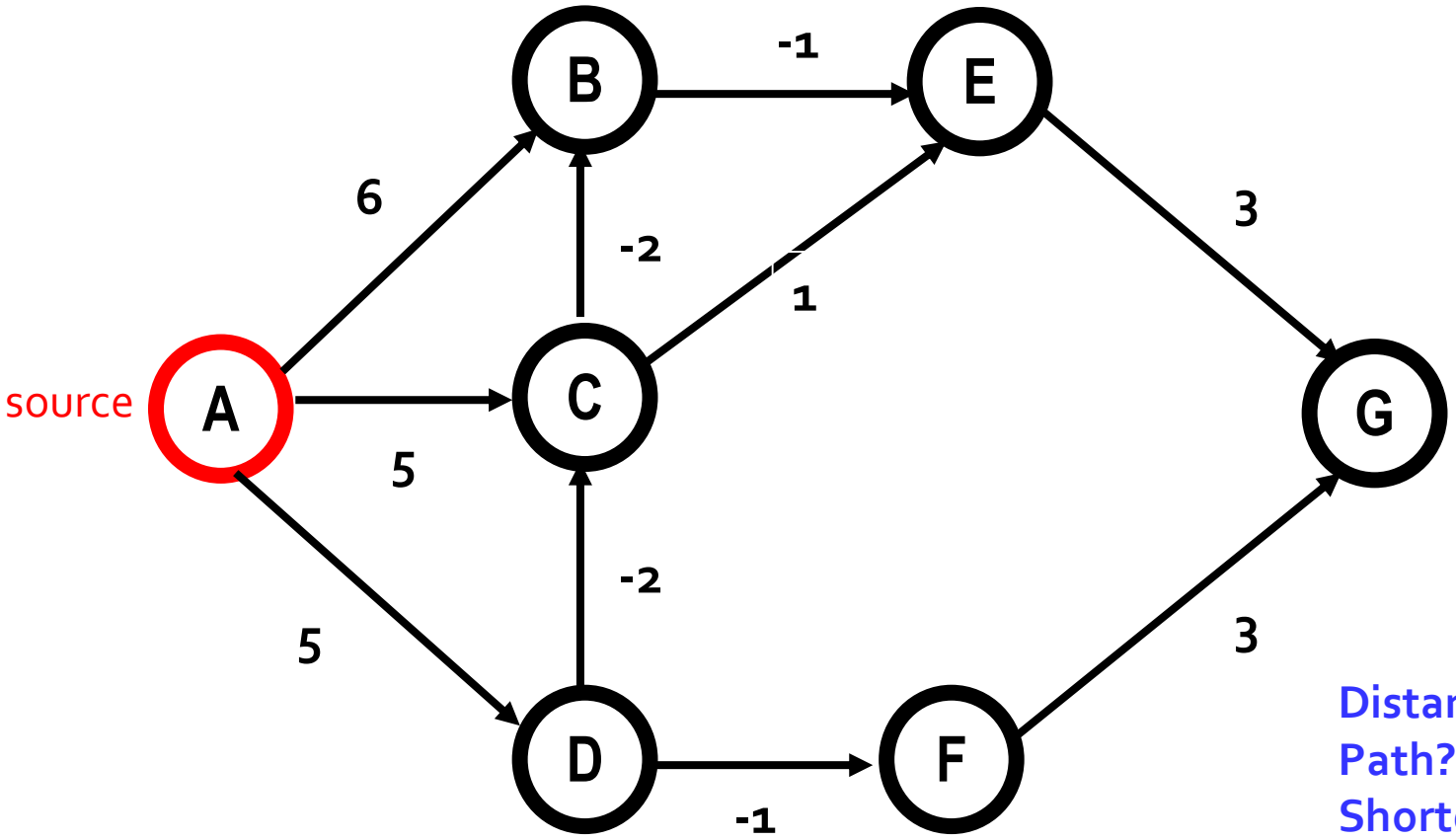
- $Path^{initial}$ [v]
- $Path^{1}$ [v]
- $Path^{2}$ [v]
- $Path^{3}$ [v]
- .
- .
- .
- $Path^{|V|-1}$ [v]

▶ QUIZ? Bellman & Ford's Algorithm

- Compute **Distance^K** from **Distance^{K-1}** ?

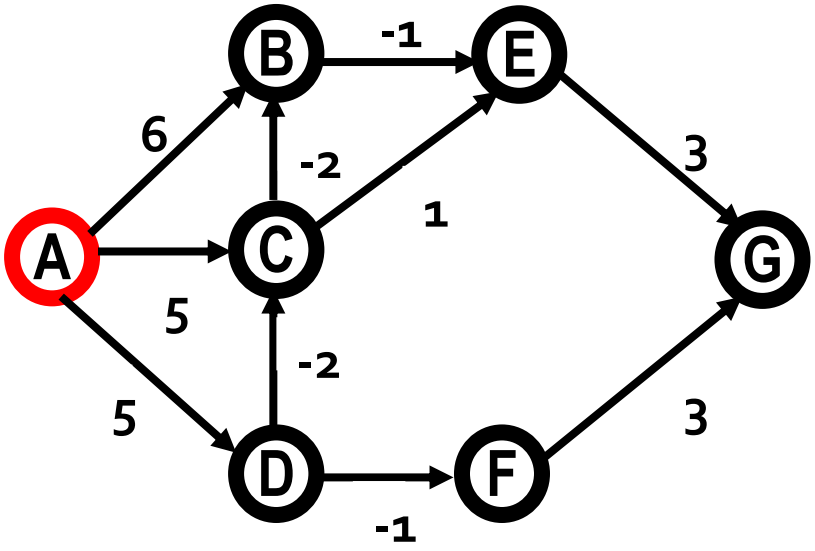
$$\text{Distance}^K[v] = \min\Big(\text{Distance}^{K-1}[v],\ \min\big(\text{Distance}^{K-1}[u] + \text{weight}(u,v) \text{ for all } u \text{ with edge } (u,v)\big)\Big)$$

Example: SSSP by Bellman & Ford's Algorithm



Distance?
Path?
Shortest paths?

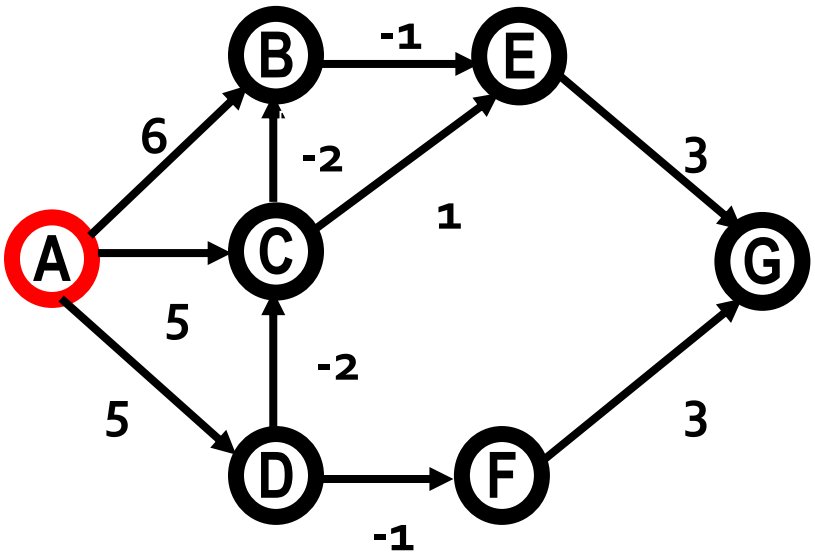
Example: SSSP by Bellman & Ford's Algorithm



|V| = 7

		A	B	C	D	E	F	G
k=0	D^0	0	∞	∞	∞	∞	∞	∞
k=1	D^1	0	6	5	5	∞	∞	∞
k=2	D^2	0	3	3	5	5	4	∞

Example: SSSP by Bellman & Ford's Algorithm



SPT(Shortest Path Tree)

|V| = 7

		A	B	C	D	E	F	G
k=1	D^1	0	6	5	5	∞	∞	∞
k=2	D^2	0	**3**	**3**	5	**5**	**4**	∞
k=3	D^3	0	**1**	3	5	**2**	4	**7**
k=4	D^4	0	1	3	5	**0**	4	**5**
k=5	D^5	0	1	3	5	0	4	**3**
k=6	D^6	0	1	3	5	0	4	3
k=6	P^6	-	C	D	A	B	D	E

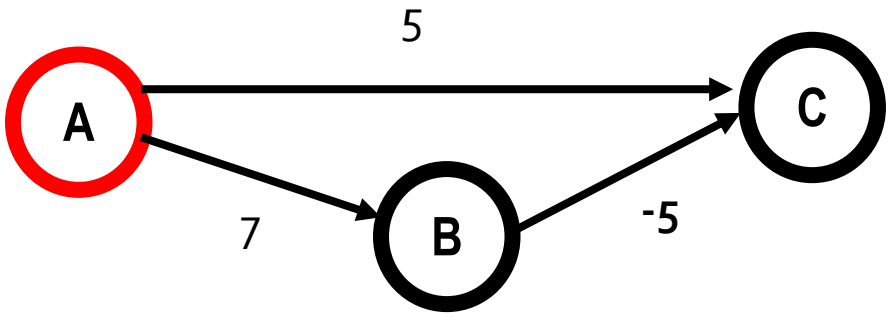
Bellman & Ford's - Algorithm

```
for all v in V
    Distance[v]= infinity; Path[v] = nil;
// Relaxation
for i := 1 to |V| - 1
  for each (u,v) in a specific order in E
    if Distance[v] > Distance[u] + weight (u,v)) then
      Distance [v] = Distance [u] + weight (u,v);
      Path[v] = u;
```

Bellman & Ford's - Algorithm

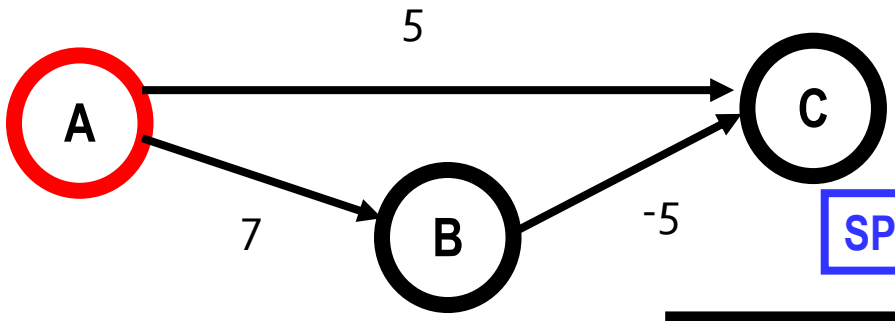
- Time complexity
  - **O(|V||E|)**

▶QUIZ? Bellman & Ford's Algorithm



Distance?
Path?
Shortest paths?

QUIZ: Bellman & Ford's Algorithm



SPT(Shortest Path Tree)

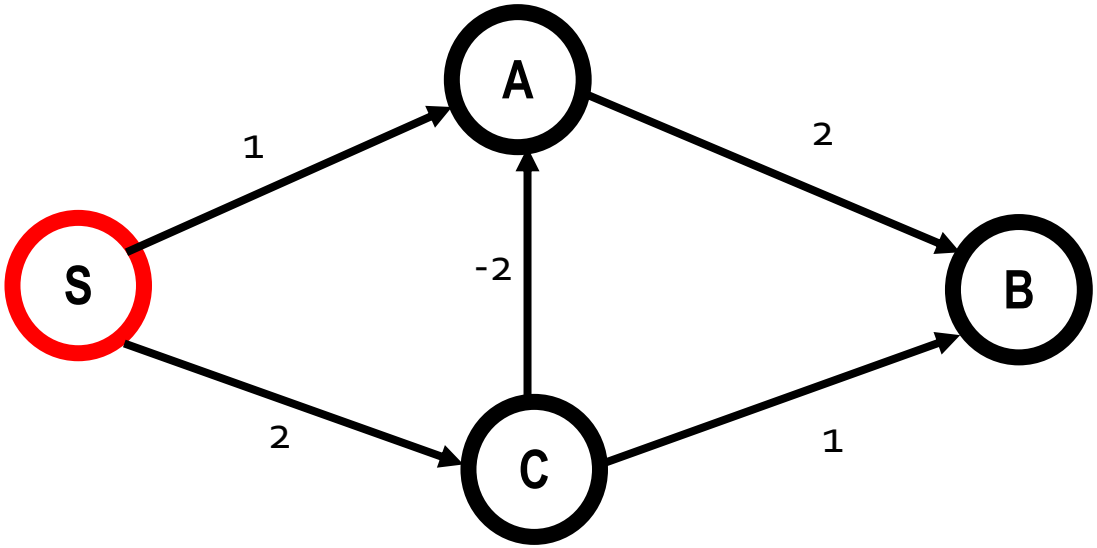
Shortest Paths from A:
A-B (7)
A-B-C (2)

|V| = 3

		A	B	C
k=0	Distance	0	∞	∞
k=1	Distance	0	7	5
k=2	Distance	0	7	2

k=0	Path	-	-	-
k=1	Path	-	A	A
k=2	Path	-	A	B

▶QUIZ? Bellman & Ford's Algorithm

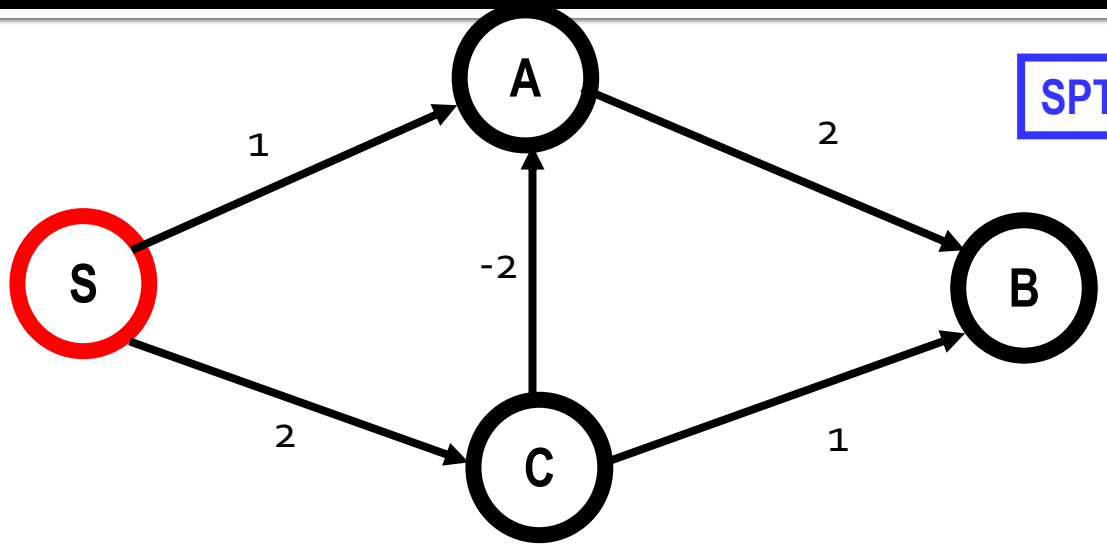


Distance?

Path?

Shortest paths?

▶QUIZ: Bellman & Ford's Algorithm



SPT(Shortest Path Tree)

Shortest Paths from S:
S-C-A (0)
S-C-A-B (2)
S-C (2)

|V| = 4

		S	A	B	C
k=0	Distance	0	∞	∞	∞
k=1	Distance	0	1	∞	2
k=2	Distance	0	0	3	2
k=3	Distance	0	0	2	2

		S	A	B	C
k=0	Path	-	-	-	-
k=1	Path	-	S	-	S
k=2	Path	-	C	A	S
k=3	Path	-	C	A	S

5. Chained Matrix Multiplication

- Standard matrix multiplication:
  A 2 X 3 matrix times a 3 X 4 matrix is a 2 X 4 matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} * \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

- How many multiplications?
  - 3 multiplications per each entry & 8 entries
  - Total 24

Chained Matrix Multiplication

- i*j matrix times j*k matrix needs i*j*k multiplications!

- Matrix multiplication is an **associative** operation, meaning that the order in which we multiply does not matter.

Chained Matrix Multiplication

- An **optimal order** for multiplying ***4*** matrices?

A	*	B	*	C	*	D
20 X 2		2 X 30		30 X 12		12 X 8

- How many different orders?
  - *5*

Chained Matrix Multiplication

A	*	B	*	C	*	D
20 X 2		2 X 30		30 X 12		12 X 8

- A(B (CD)) requires 3680 multiplications. 30 * 12 * 8 + 2 * 30 * 8 + 20 * 2 *8 = 3, 680
- (AB) (CD) requires 888omultiplications. 20* 2 * 30 + 30 * 12 * 8 + 20 * 30 *8 = 8, 880
- A((BC)D) requires 1232 multiplications. 2* 30 * 12 + 2 *12 * 8 + 20 * 2 * 8 = 1, 232
- ((AB)C)D requires 10320 multiplications. 20 * 2 * 30 + 20 * 30 * 12 + 20 * 12 * 8 = 10, 320
- (A(BC))D requires 3120 multiplications. 2 * 30 * 12 + 20 * 2 * 12 + 20 * 12 * 8 = 3, 120

Chained Matrix Multiplication

- The goal is to develop an optimal order for multiplying ***n*** matrices A1, A2, … , An:

  A1 x A2 x … x An

Chained Matrix Multiplication

- The **brute-force** algorithm is to consider all possible orders and take the minimum.
- This is a very inefficient method – exponential!

- Idea?

Chained Matrix Multiplication via DP

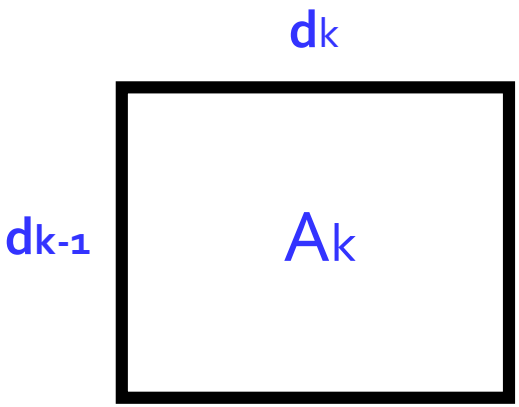
- A more efficient algorithm using DP!

Chained Matrix Multiplication via DP

- The principle of optimality applies.
- Dynamic programming can be applied!

Chained Matrix Multiplication via DP

- **M[i,j]** = minimum number of multiplications needed to multiply A_i through A_j
  - A_1: $\mathbf{d_0} \times \mathbf{d_1}$
  - A_2: $\mathbf{d_1} \times \mathbf{d_2}$
  - A_k: $\mathbf{d_{k-1}} \times \mathbf{d_k}$
  - A_n: $\mathbf{d_{n-1}} \times \mathbf{d_n}$

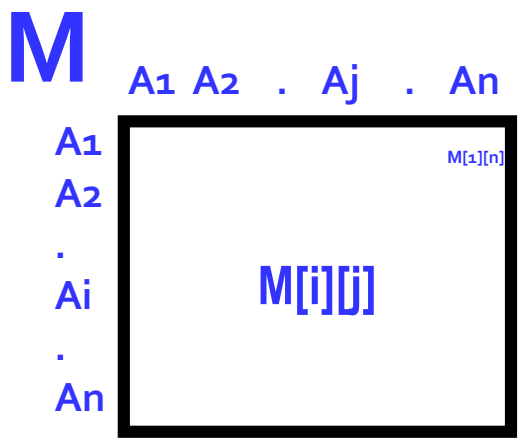


Chained Matrix Multiplication via DP

- Idea?
- A **recursive property** when multiplying *n* matrices?
  - For $1 =< i =< j =< n$ $\quad (A_i \ldots A_k)\,(A_{k+1} \ldots A_j)$
  - **$M[i][j] = \text{MIN}_{1 =< k =< j-1}$**
    $(M[i][k] + M[k+1][j] + d_{i-1}\, d_k\, d_j)$ if $i<j$
  - **$M[i][i] = 0$**

Chained Matrix Multiplication via DP

- **M[i,j]** = minimum number of multiplications needed to multiply A_i through A_j
  - A_1: $\mathbf{d_0} \times \mathbf{d_1}$
  - A_2: $\mathbf{d_1} \times \mathbf{d_2}$
  - A_k: $\mathbf{d_{k-1}} \times \mathbf{d_k}$
  - A_n: $\mathbf{d_{n-1}} \times \mathbf{d_n}$



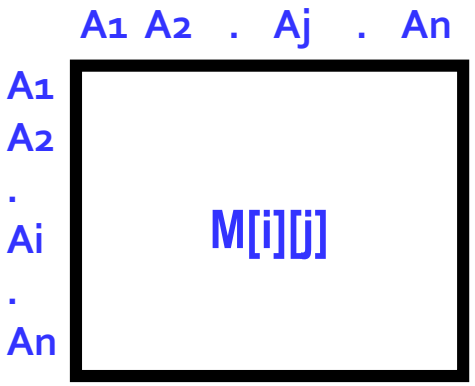
Chained Matrix Multiplication via DP

- **Algorithm 3.6** Minimum Multiplications & Order
  - Find **M[1][n]** for A1 x A2 x … x An.
  - **$O(n^3)$**
  - Space $O(n^2)$

- **Algorithm 3.7** Print Optimal Order
  - **O(n)**

▶ QUIZ? Chained Matrix Multiplication via DP

- A recursive property when multiplying *n* matrices?
- For 1 =< i =< j =< n



  - M[i][j] = MIN 1 =<k=< j-1
    (M[i][k] + M[k+1][j] + $d_{i-1}\, d_k\, d_j$) if i<j
  - M[i][i] = 0

6. Traveling Salesperson Problem (TSP)

- Suppose a salesperson is planning a sales trip.
- Each city is connected to some of the other cities by a road.
- To minimize travel time, we want to determine a shortest route (tour) that starts at the salesperson's home city, visits each of the cities once, and ends up at the home city.

Traveling Salesperson Problem

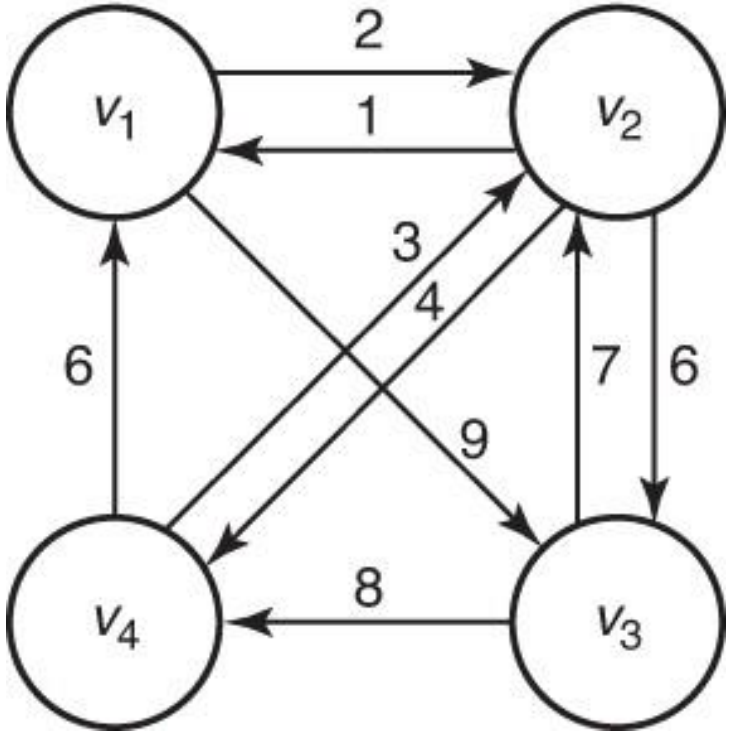
- An instance of this problem can be represented by a weighted directed graph.
- A tour (Hamiltonian circuit) is a path from a vertex to itself that passes through each of the other vertices only once.
- An optimal tour in a weighted, directed graph is such a path of minimum length.

Traveling Salesperson Problem

- The **TSP** is to find an optimal tour (Hamiltonian circuit) in a weighted directed graph when at least one tour exists.

Traveling Salesperson Problem

What is the optimal tour from v1?



Traveling Salesperson Problem

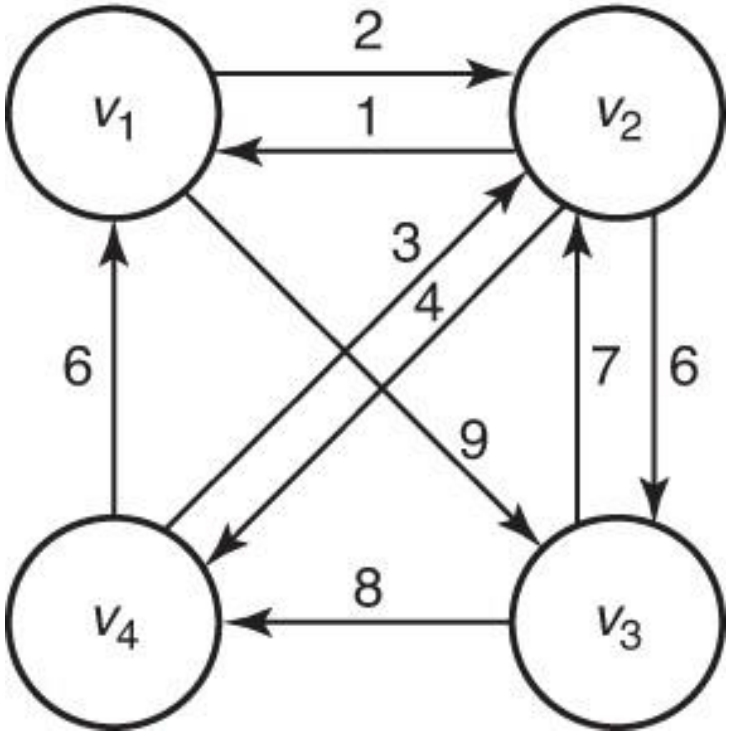


Figure 3.16: The optimal tour is [v1, v3, v4, v2, v1]

Traveling Salesperson Problem

- The **brute-force** algorithm is to consider all possible paths and take the optimal tour.
- This is a very inefficient method – factorial!

- Idea?

Traveling Salesperson Problem via DP

- A more efficient algorithm using DP!

Traveling Salesperson Problem via DP

- The principle of optimality applies.
- Dynamic programming can be applied!

Traveling Salesperson Problem via DP

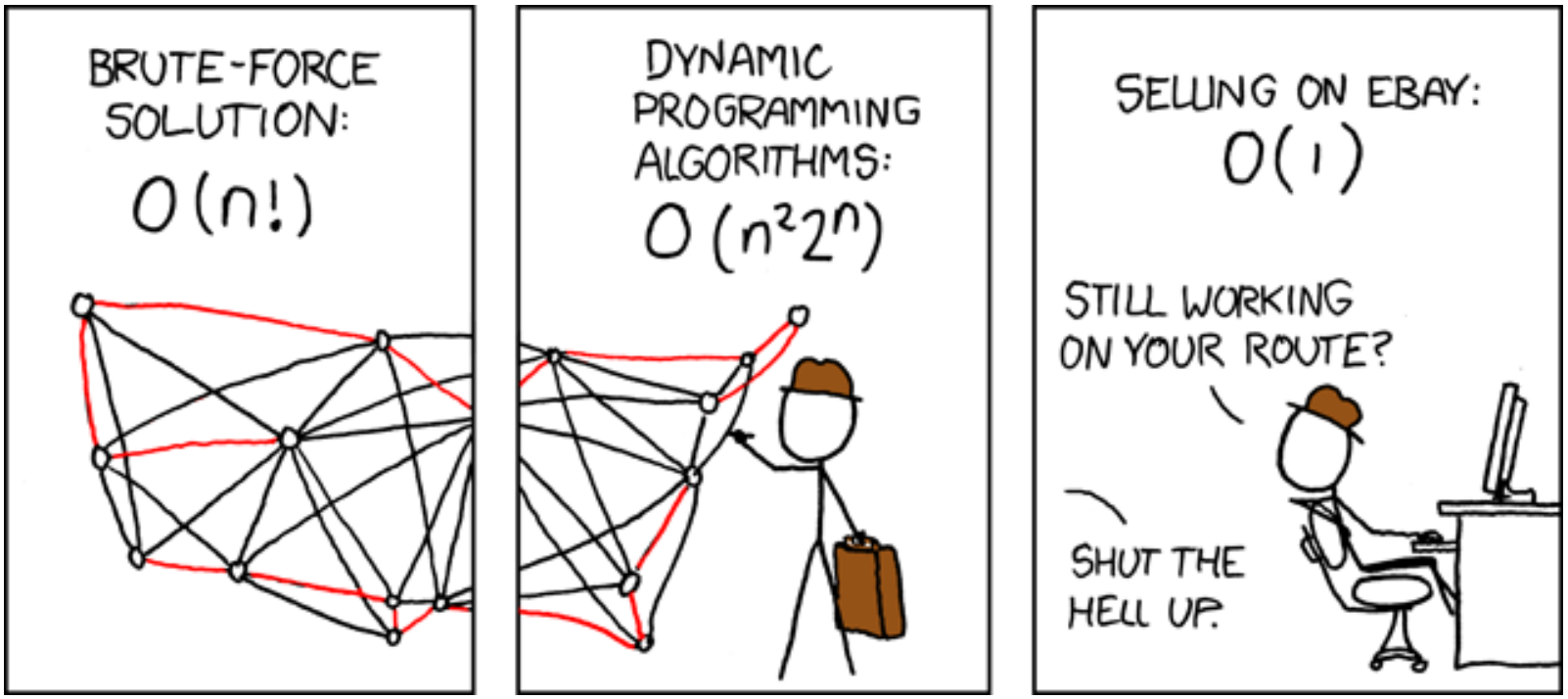
- **Algorithm 3.11** DP Algorithm for the TSP
  - **$O(n^2 2^n)$**
  - **BUT, no polynomial time!**
- Example 3.12

Traveling Salesperson Problem via DP

- No one has ever found an algorithm for the Traveling Salesperson problem whose worst-case time complexity is better than exponential!
- Yet, no one has ever proved that the algorithm is not possible!

- The Traveling Salesperson Problem is **hard**.
- **NP-complete!**

Traveling Salesperson Problem



QUIZ? TSP via DP

- What is the worst-case time complexity of the DP_based algorithm for TSP?

7. The 0-1 Knapsack Problem

- Given a knapsack (rucksack) with maximum weight *W* and a set *S* of *n* items, each with weight *w* and profit (value) *p*, i.e.,
  - $S = \{ (item_1, w_1, p_1), (item_2, w_2, p_2), \ldots, (item_n, w_n, p_n) \}$
- Determine the count of each item to include in the knapsack so that
  - The total weight is less than or equal to W and
  - The total profit (value) is **as large as possible**.
- For each item, only 1 copy is available & not allowed to break!

The 0-1 Knapsack Problem via DP

- We can apply **DP** to the 0-1 Knapsack problem!
  - **$O(2^n)$** where n = the number of items

- The 0-1 Knapsack Problem is **hard**.
- **NP-complete!**

✓ Dynamic Programming (DP)-based Algorithms Summary

1. N-th Fibonacci Term via DP
2. Binomial Coefficient via DP
3. Floyd & Warshall's Algorithm - APSP via DP
4. Bellman & Ford's Algorithm – SSSP via DP
5. Chained Matrix Multiplication via DP
6. Traveling Salesperson Problem via DP
7. The 0-1 Knapsack Problem via DP

Homework Assignment

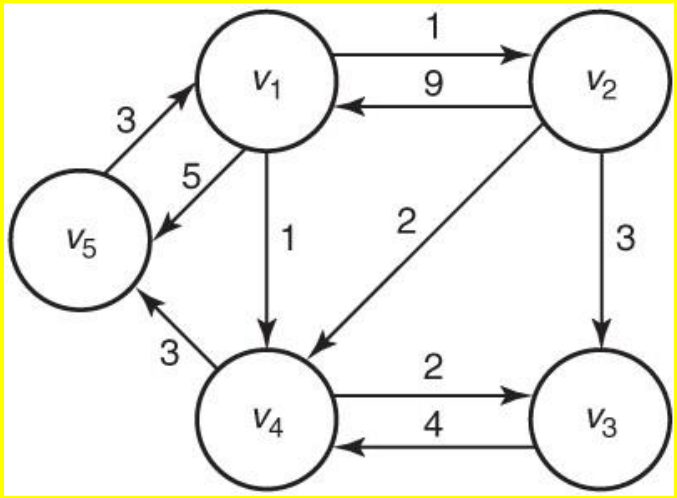
Homework Assignment?

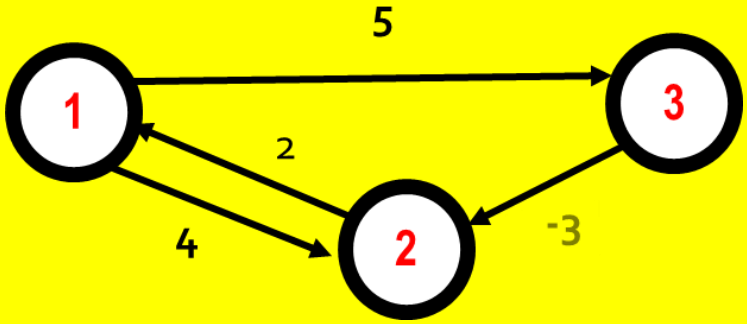
- Chapter 3: Exercise #5, #6 and #10

Homework Assignment?

- *Dynamic Programming:* Chapter 3: Exercise #8 (Dynamic Programming-based Floyd-Warshall Algorithm for the All-Pairs Shortest Path Problem)

Test Cases





✓ Textbook Readings

- Chapter 3:
  - 3.1
  - 3.2
  - 3.3
  - 3.4
  - 3.6

the right majors, minors &

ucation?

or students' academic and career success

ging for many students - *Many students change their*
uring college!

e prediction of student success in MMC could
dual students
d their right MMC
chieve their academic goals

END

Prof. Young Park