DSC 40B

Lecture 17 : Graph representations

Adjacency Matrices

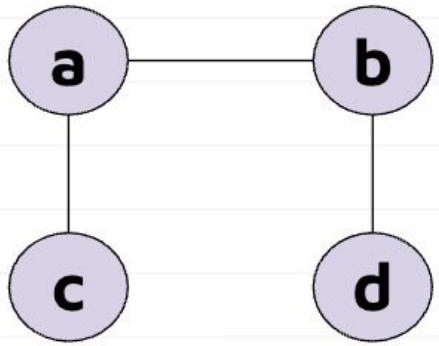
Representations

- How do we store a graph in a computer's memory?
- **Three approaches:**
  - Adjacency matrices.
  - Adjacency lists.
  - "Dictionary of sets"

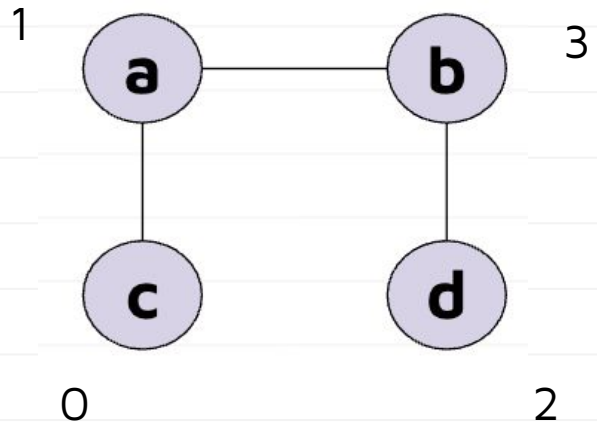
Adjacency Matrices

- Assume nodes are numbered 0, 1, …, $|V| - 1$
- Allocate a $|V| \times |V|$ (Numpy) array
- Fill array as follows:
  - `arr[i,j] = 1 if` $(i, j) \in E$
  - `arr[i,j] = 0 if` $(i, j) \notin E$

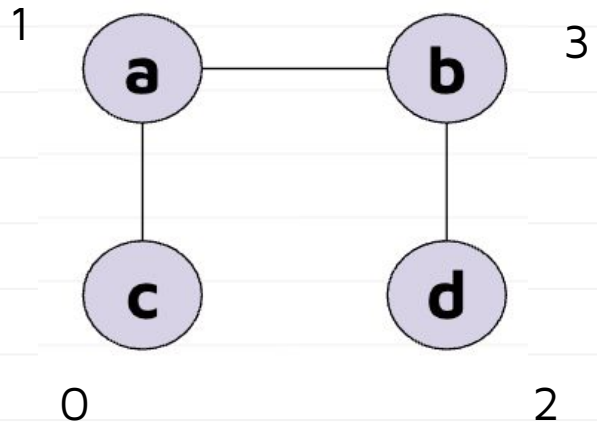
Example



Example

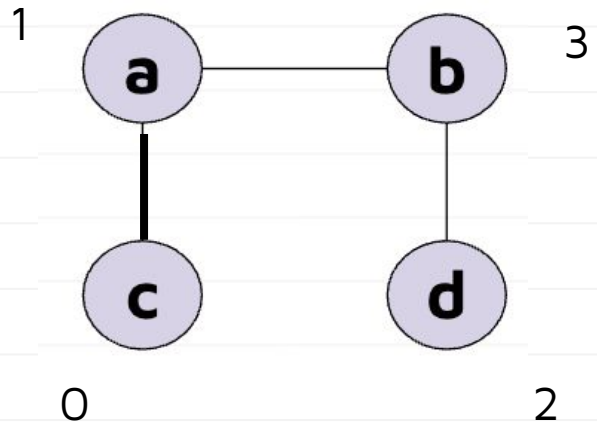


Example



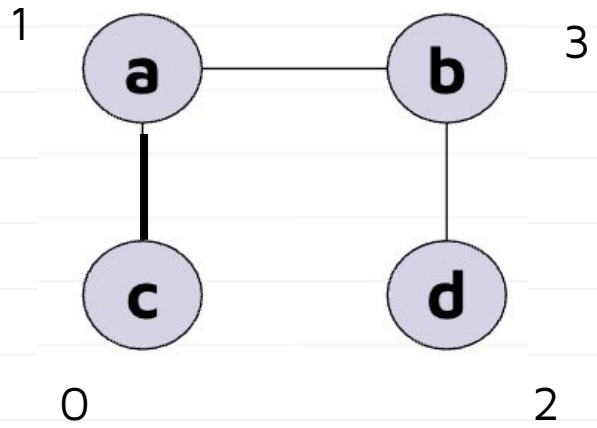
	0	1	2	3
0				
1				
2				
3				

Example



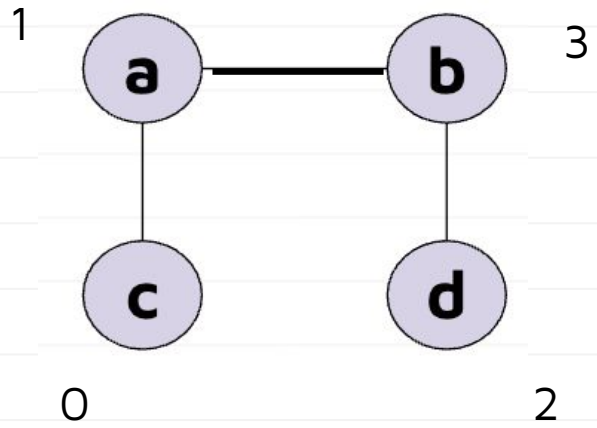
	0	1	2	3
0		**1**		
1				
2				
3				

Example



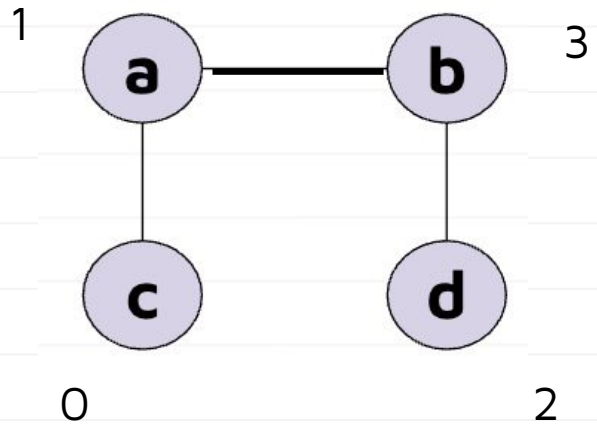
	0	1	2	3
0		**1**		
1	**1**			
2				
3				

Example



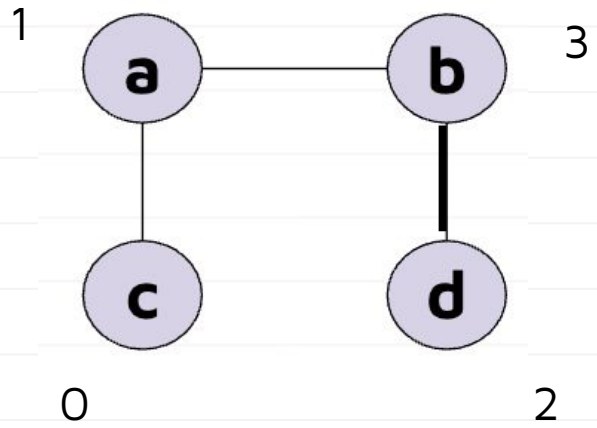
	0	1	2	3
0		**1**		
1	**1**			**1**
2				
3				

Example



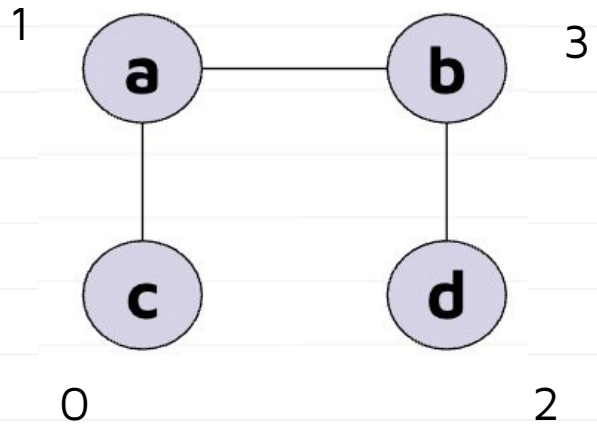
	0	1	2	3
0		**1**		
1	**1**			**1**
2				
3		**1**		

Example



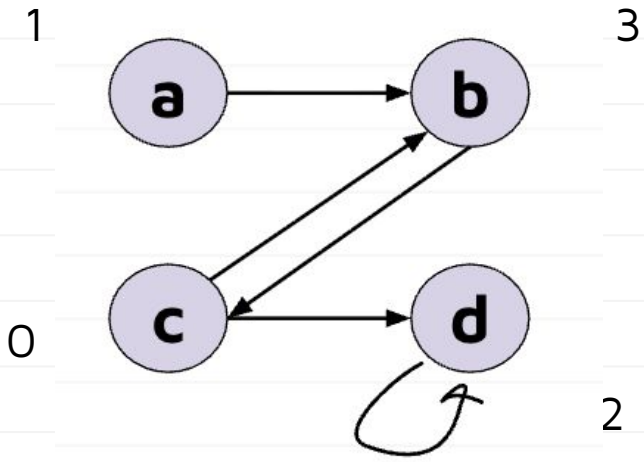
	0	1	2	3
0		**1**		
1	**1**			**1**
2				**1**
3		**1**	**1**	

Example



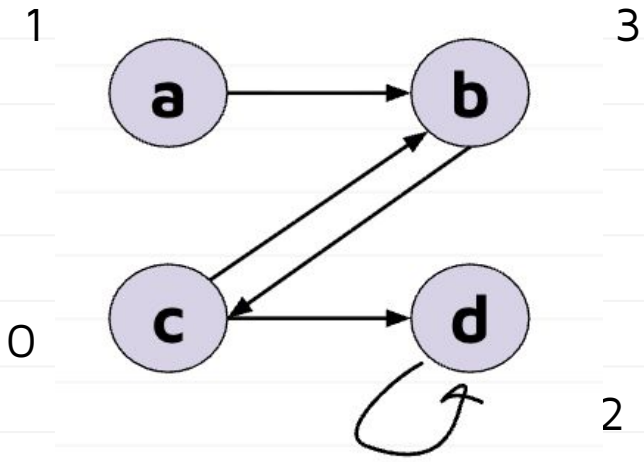
	0	1	2	3
0	**0**	**1**	**0**	**0**
1	**1**	**0**	**0**	**1**
2	**0**	**0**	**0**	**1**
3	**0**	**1**	**1**	**0**

Example



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

Example

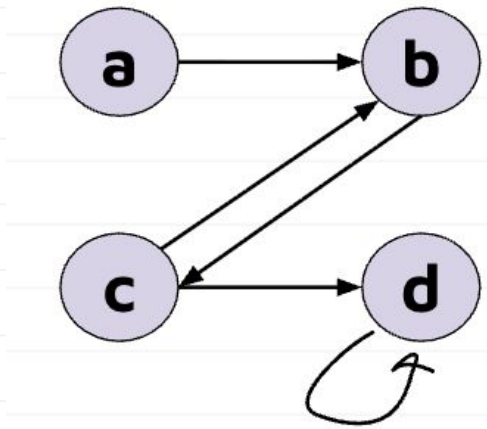


	0	1	2	3
0				**1**
1				**1**
2				
3				

Example

1

3

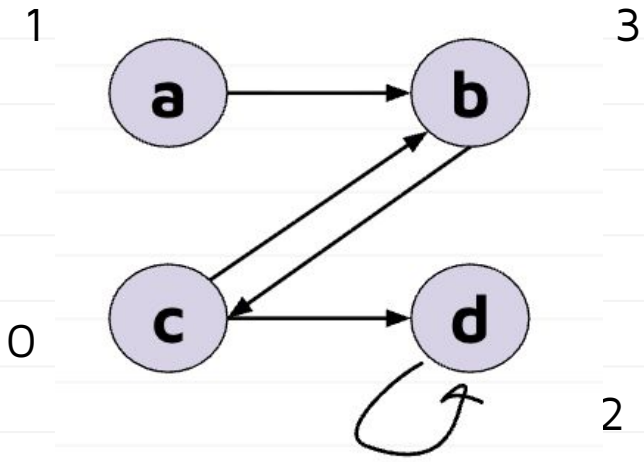


0

2

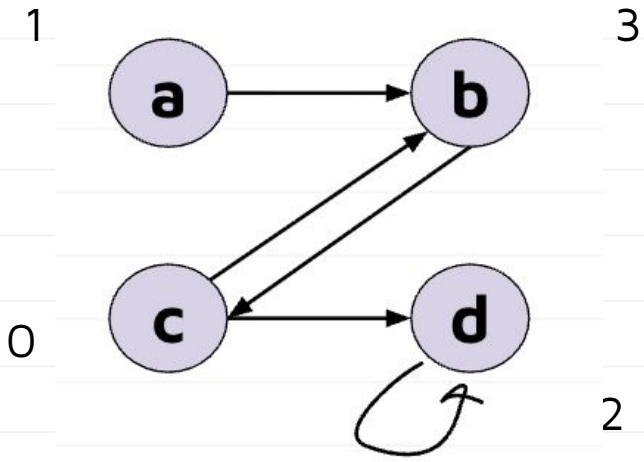
	0	1	2	3
0			**1**	**1**
1				**1**
2				
3				

Example



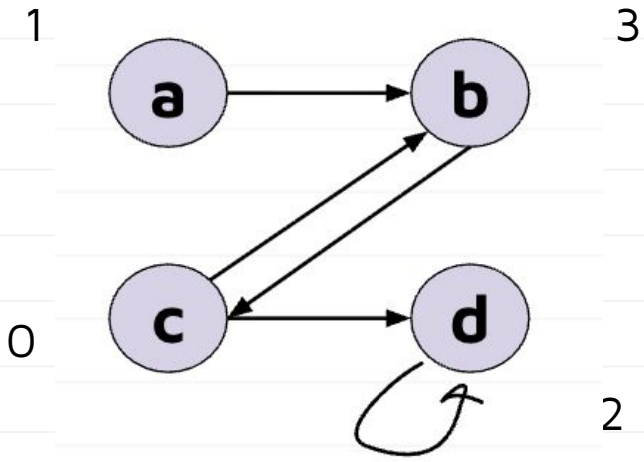
	0	1	2	3
0			**1**	**1**
1				**1**
2				
3	**1**			

Example



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

Example



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

Observations

- If G is **undirected**, matrix is *symmetric*.
- If G is **directed**, matrix *may not* be symmetric.

Time Complexity

operation	code	time
Edge query	`adj[i, j] == 1`	$\Theta(1)$
Degree(i)	`np.sum(adj[i,:])`	$\Theta(\|V\|)$

Space Requirements

- Uses $|V|^2$ bits, even if there are very few edges.
- But most real-world graphs are **sparse**.
  - They contain many *fewer* edges than possible.

Example: Facebook

- Facebook has 2 billion users. (V)

$(2 \times 10^9)^2 = 4 \times 10^{18}$ bits

= 500 petabits

≈ 6500 years of video at 1080p

≈ 60 copies of the internet as it was in 2000

Adjacency Matrices and Math

- Adjacency matrices are useful mathematically.
- **Example**: (i, j) entry of A^2 gives number of hops of length 2 between i and j.

Adjacency Lists

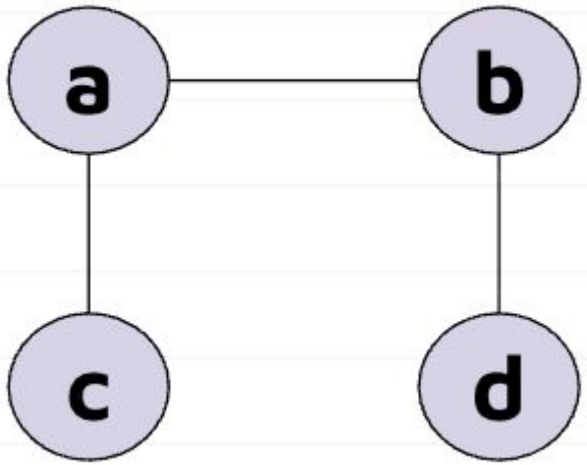
What's Wrong with Adjacency Matrices?

- Requires $\Theta(|V|^2)$ storage.
- Even if the graph has no edges.
- **Idea**: only store the edges that *exist*.

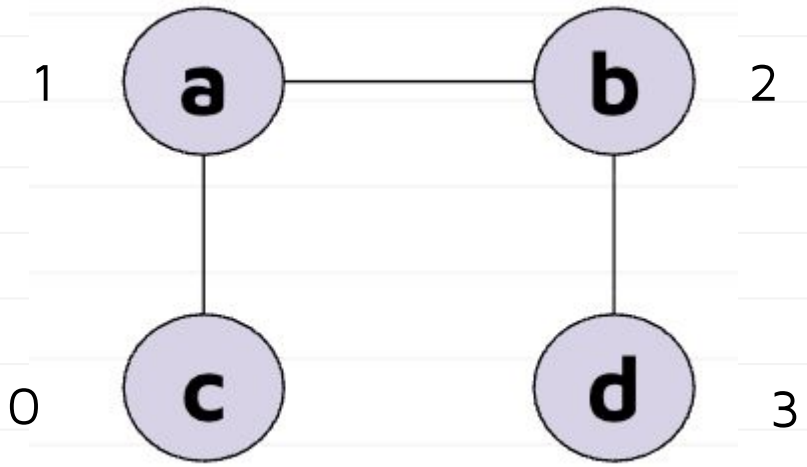
Adjacency Lists

- Create a list `adj` containing $|V|$ lists.
- `adj[i]` is list containing the neighbors of node i.

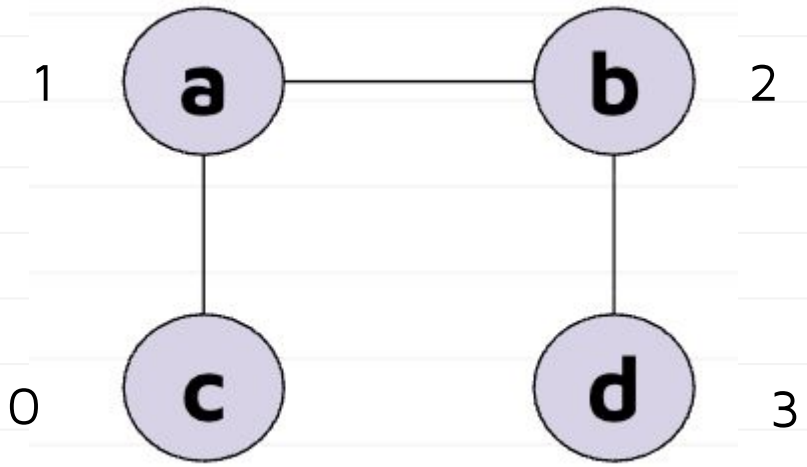
Example



Example

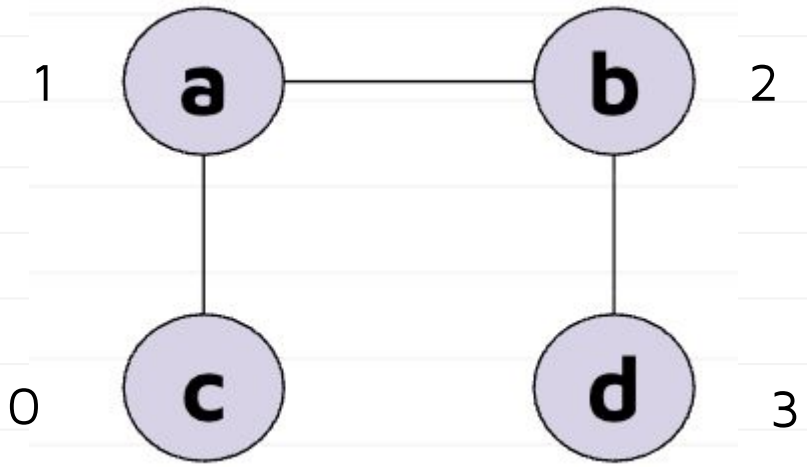


Example



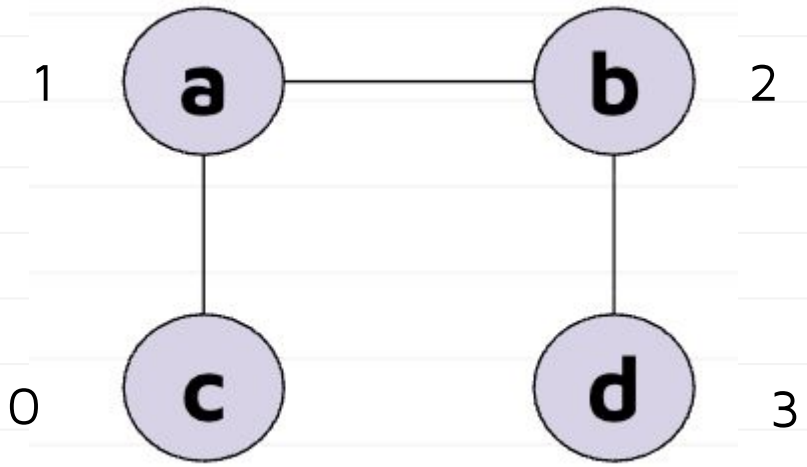

```
adj = [

        [               ],

        [               ],

        [               ],

        [               ]

    ]
```

Example



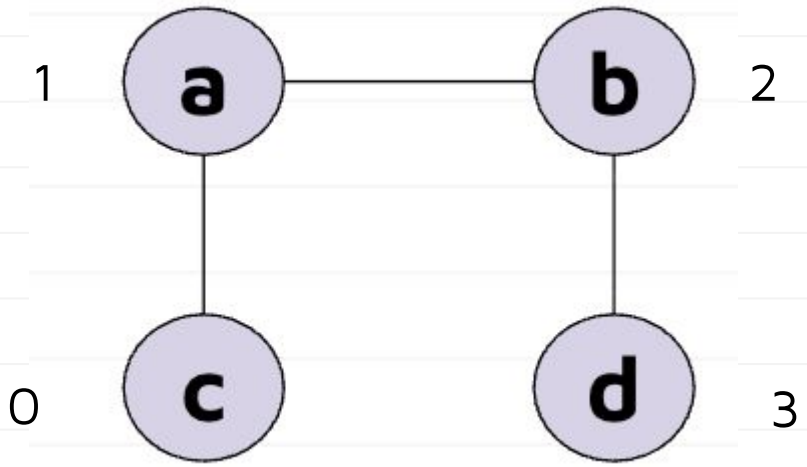

```
adj = [

    #0  [                ],

    #1  [                ],

    #2  [                ],

    #3  [                ]

     ]
```

`adj[i]` is list containing the *neighbors* of node *i*.



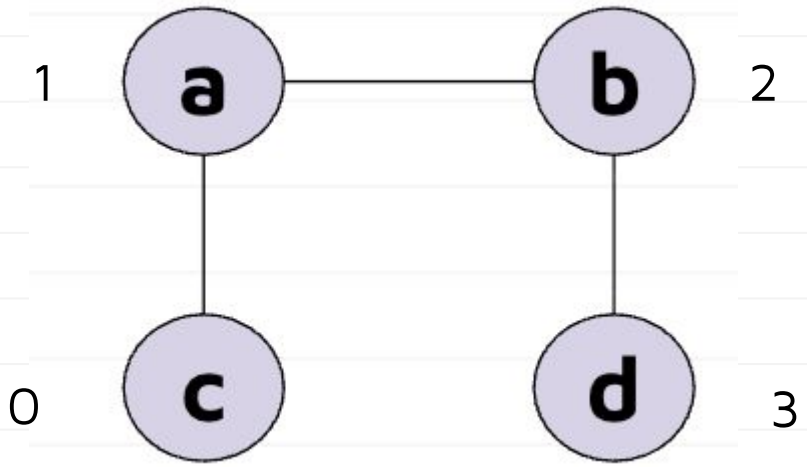

```
adj = [

    #0  [       ?       ],

    #1  [       ?       ],

    #2  [       ?       ],

    #3 [        ?       ]

     ]
```

`adj[i]` is list containing the *neighbors* of node *i*.



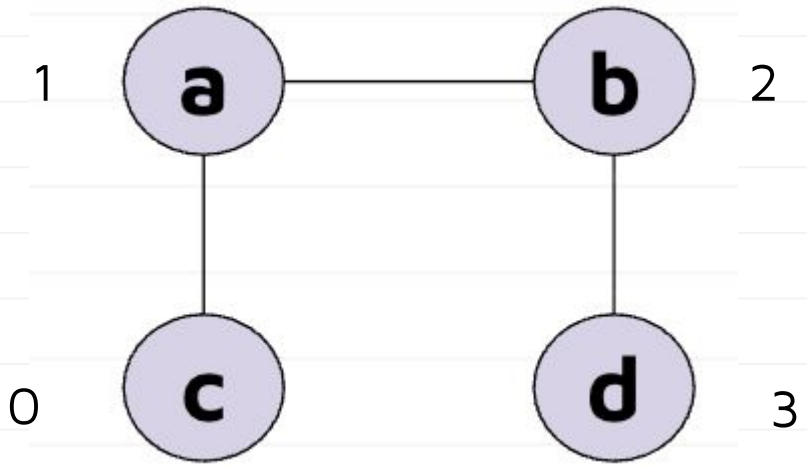

```
adj = [

    #0   [ 1 ],

    #1   [       ?       ],

    #2   [       ?       ],

     #3  [       ?        ]

      ]
```

`adj[i]` is list containing the *neighbors* of node *i*.



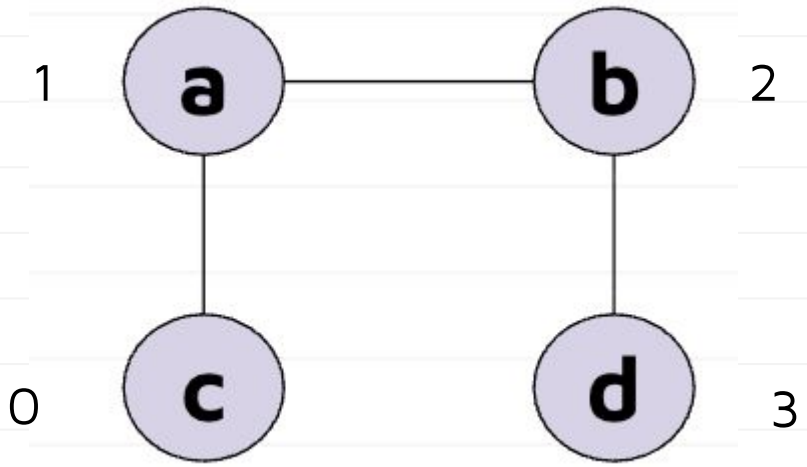

```
adj = [

    #0   [ 1 ],

     #1   [ 0, 2 ],

     #2   [        ?        ],

      #3  [        ?        ]

       ]
```

`adj[i]` is list containing the *neighbors* of node *i*.



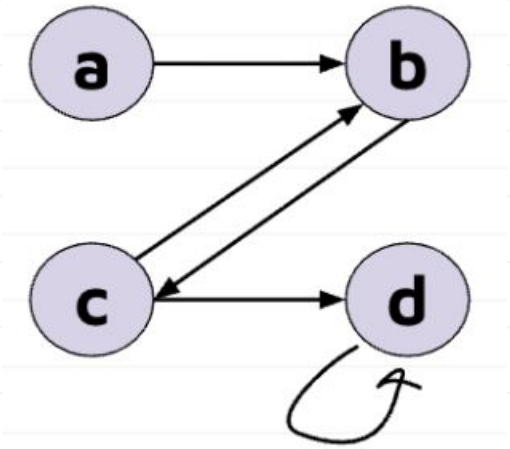

```
adj = [

    #0   [ 1 ],

     #1   [ 0, 2 ],

     #2   [ 1, 3 ],

      #3  [        ?        ]

       ]
```

`adj[i]` is list containing the *neighbors* of node *i*.

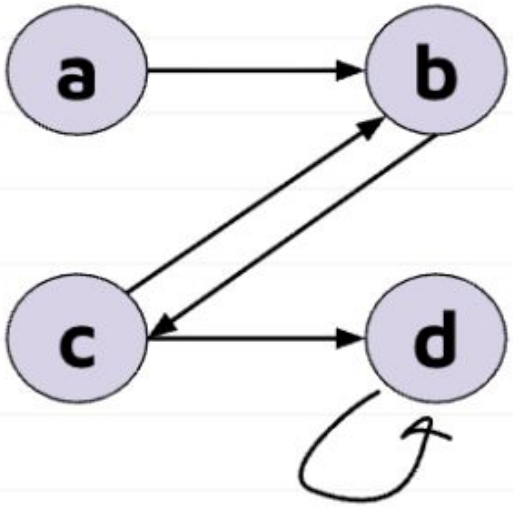



```
adj = [

    #0   [ 1 ],

     #1   [ 0, 2 ],

     #2   [ 1, 3 ],

      #3  [ 2 ]

       ]
```

Example



Example



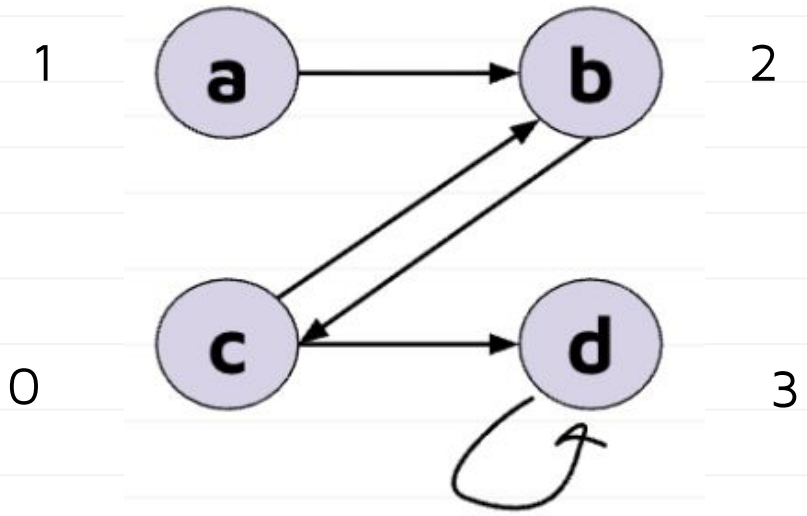
1

2

0

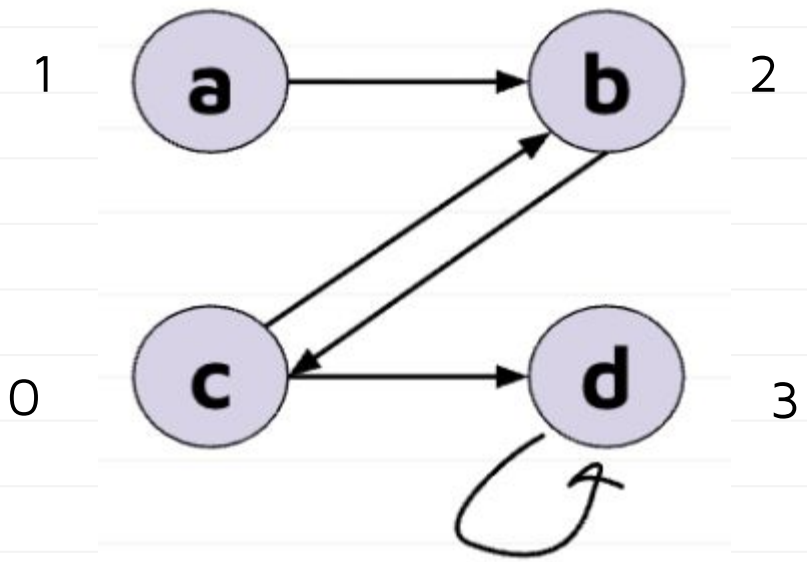
3

Example



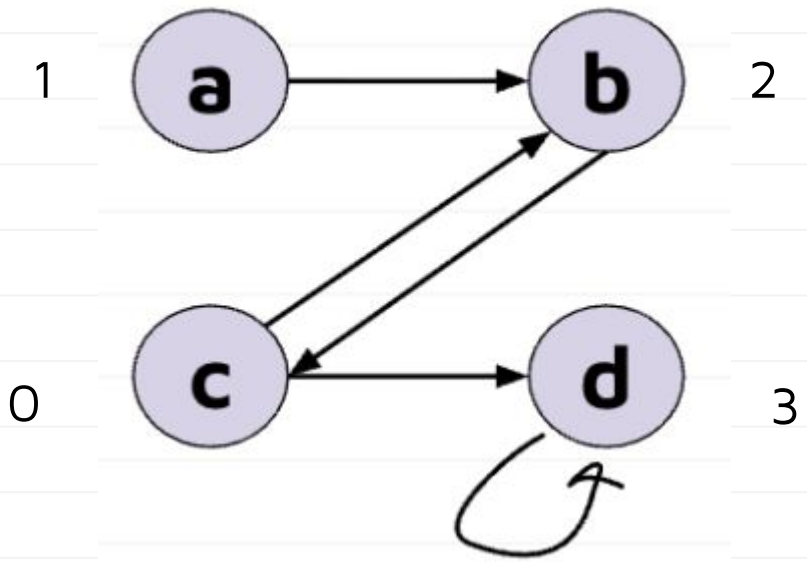

```
adj = [

        [               ],

        [               ],

        [               ],

        [               ]

    ]
```

Example



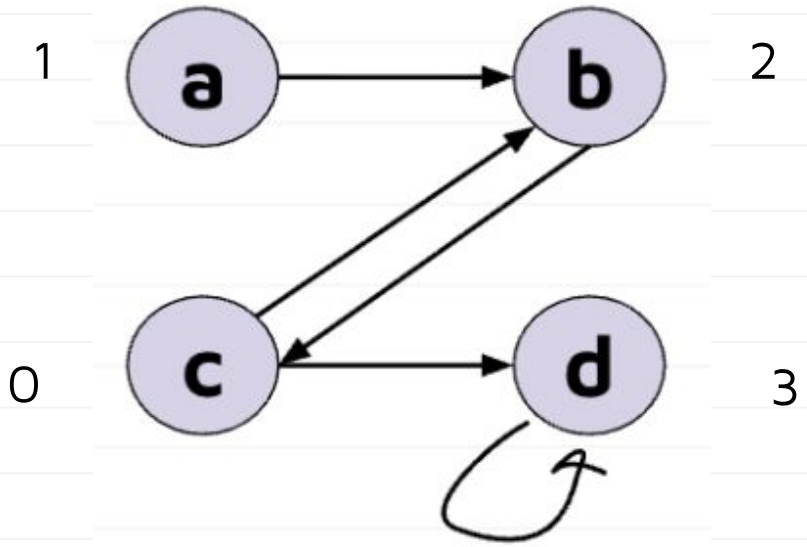

```
adj = [

    #0  [                ],

     #1  [                ],

     #2  [                ],

      #3 [                 ]

       ]
```

`adj[i]` is list containing the ***neighbors*** of node i.



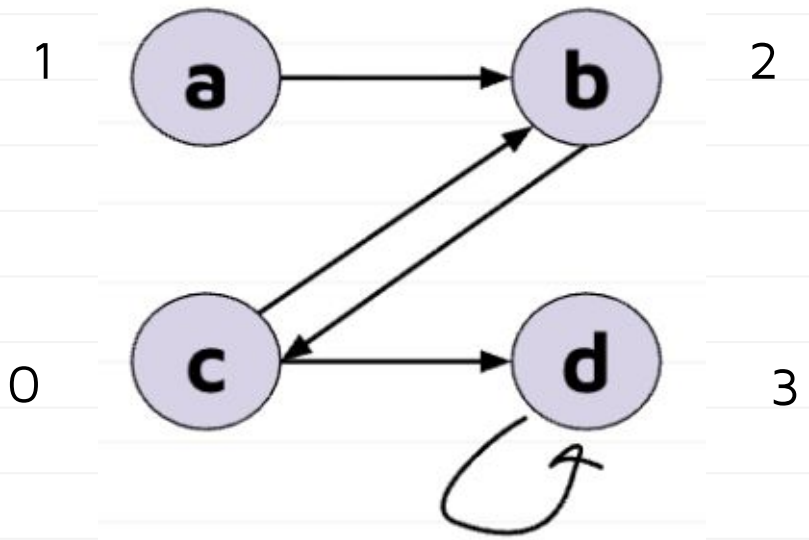

```
adj = [

    #0  [       ?       ],

    #1  [       ?       ],

    #2  [       ?       ],

    #3  [       ?       ]

    ]
```

`adj[i]` is list containing the *neighbors* of node *i*.



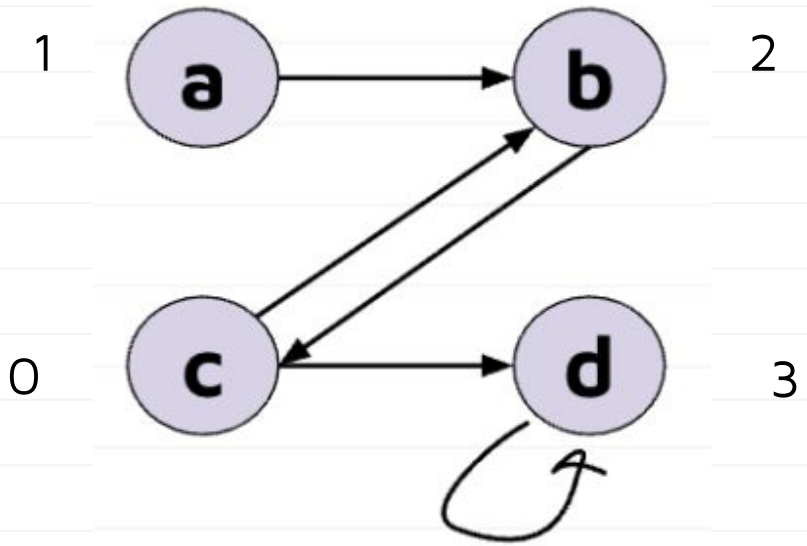

```
adj = [

    #0   [ 2, 3 ],

     #1   [       ?       ],

     #2   [       ?       ],

      #3  [       ?       ]

      ]
```

`adj[i]` is list containing the *neighbors* of node *i*.



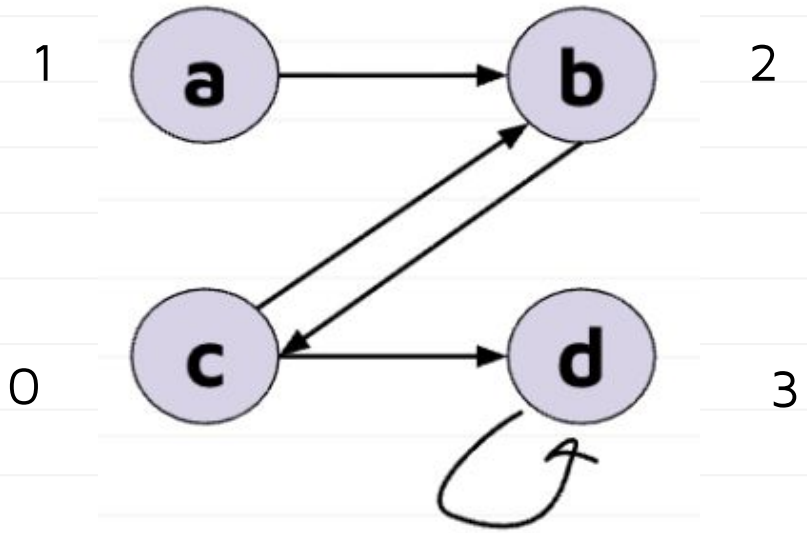

```
adj = [

    #0   [ 2, 3 ],

    #1   [ 2 ],

    #2   [       ?       ],

    #3  [       ?       ]

     ]
```

`adj[i]` is list containing the *neighbors* of node *i*.




```
adj = [

    #0   [ 2, 3 ],

    #1   [ 2 ],

    #2   [ 0 ],

    #3  [        ?        ]

     ]
```

`adj[i]` is list containing the *neighbors* of node *i*.




```
adj = [

    #0   [ 2, 3 ],

     #1   [ 2 ],

     #2   [ 0 ],

      #3  [ 3 ]

       ]
```

Observations

- If G is **undirected**, each edge appears **twice**.
- If G is **directed**, each edge appears **once**.

Time Complexity

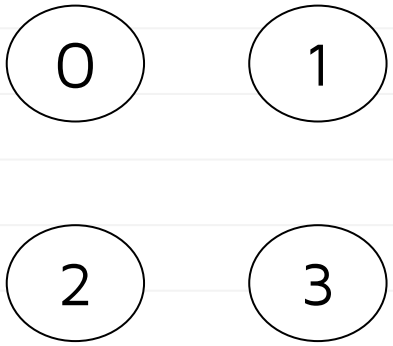
operation	code	time
Edge query	`j in adj[i, j]`	Θ(degree(i))
degree(i)	`len(adj[i])`	Θ(1)

Exercise 1: Build a graph from the list

```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

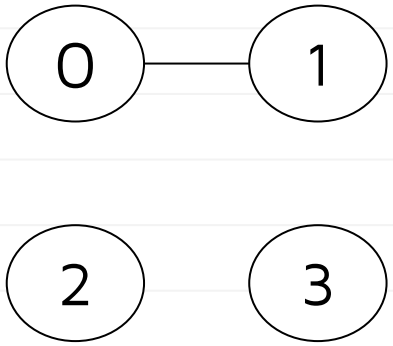
Exercise 1: Build a graph from the list




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

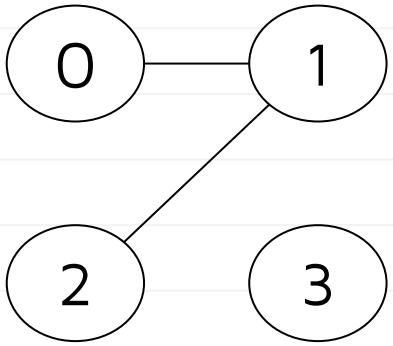
Exercise 1: Build a graph from the list




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

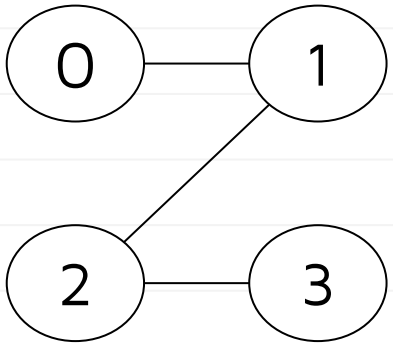
Exercise 1: Build a graph from the list




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

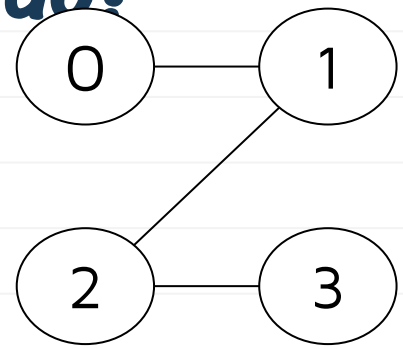
Exercise 1: Build a graph from the list




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

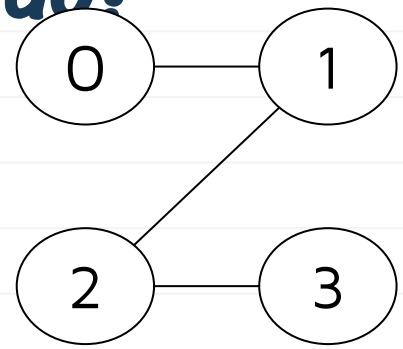
```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

Exercise 2: What does the following code do?




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]

for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

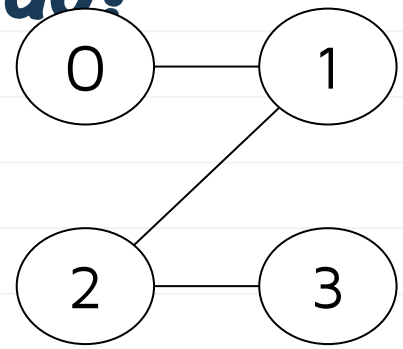
Exercise 2: What does the following code do?




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]

for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

Prints all edges

Exercise 2: What does the following code do?



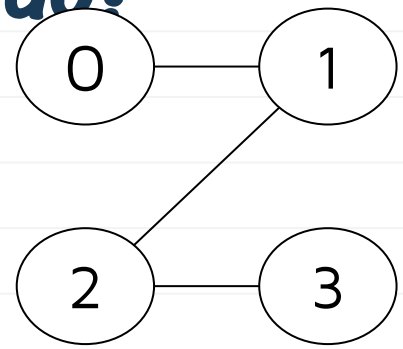

```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

(0, 1)

Prints all edges

Exercise 2: What does the following code do?

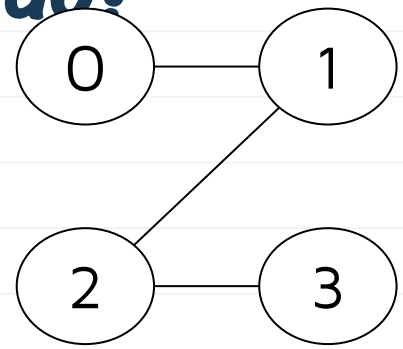



```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]

for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

Prints all edges

(0, 1)
(1, 0)
(1, 2)

Exercise 2: What does the following code do?

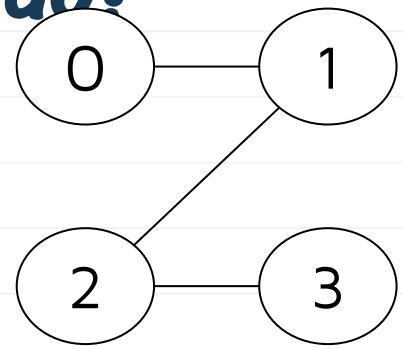



```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]

for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

Prints all edges

(0, 1)
(1, 0)
(1, 2)
(2, 1)
(2, 3)

Exercise 2: What does the following code do?

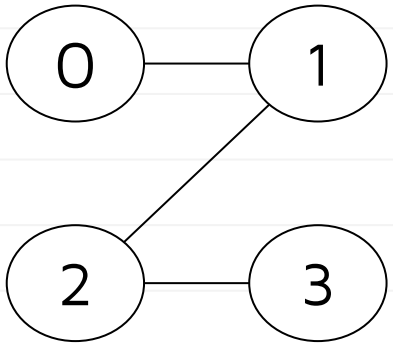



```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]

for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

Prints all edges

```
(0, 1)
(1, 0)
(1, 2)
(2, 1)
(2, 3)
(3, 2)
```

Exercise 3: Complexity




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

How many times is the print statement executed in terms of $|V|$ and $|E|$ (if undirected)?

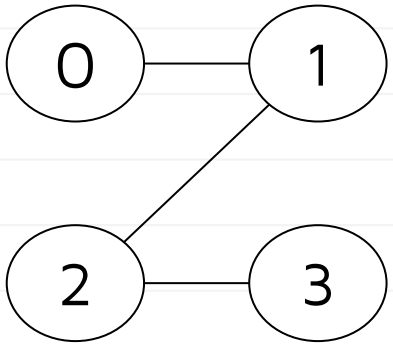
A: |E| + |V|

C: 2 * |E|

B: |E| * |V|

D: |E| + |V| - 1

Exercise 3: Complexity




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

How many times is the print statement executed in terms of $|V|$ and $|E|$ (if undirected)?

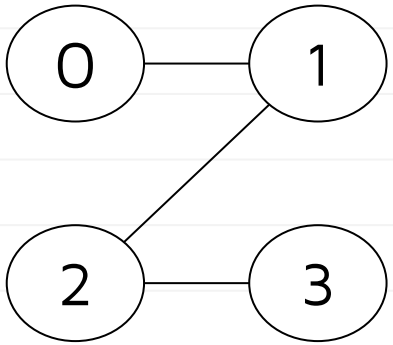
A: |E| + |V|

C: 2 * |E|

B: |E| * |V|

D: |E| + |V| - 1

Exercise 3: Complexity



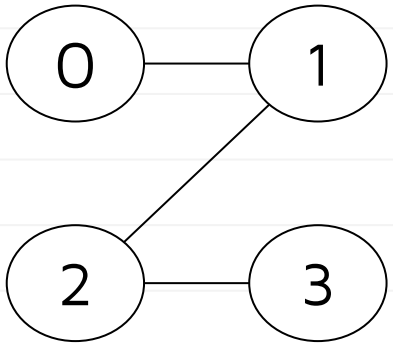

```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

How many times is the print statement executed in terms of $|V|$ and $|E|$ (if directed)?

A: |E|

Exercise 4: Complexity




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

What is the time complexity in terms of $|V|$ and $|E|$?

A: Θ (|E| + |V|)

C: Θ (|E|)

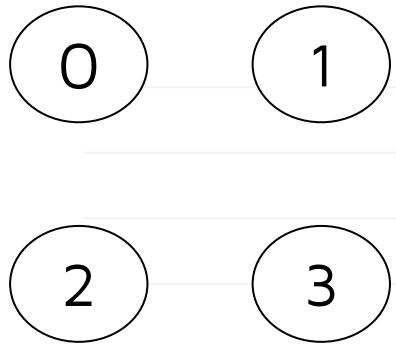
B: Θ (|E| * |V|)

D: Θ (|V|)

Exercise 5:

```
adj = [
    [],
    [],
    [],
    []
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```



What is the time complexity for this case?

A: Θ (|E| + |V|)

C: Θ (|E|)

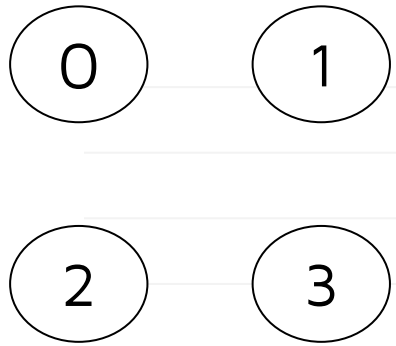
B: Θ (|E| * |V|)

D: Θ (|V|)

Exercise 5:

```
adj = [
    [],
    [],
    [],
    []
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```



What is the time complexity for this case?

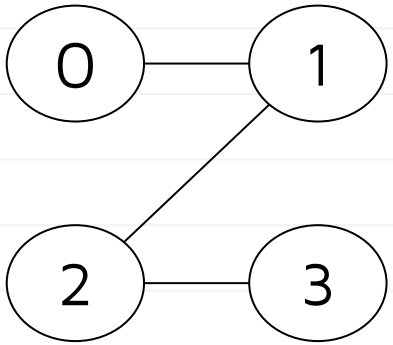
A: Θ (|E| + |V|)

C: Θ (|E|)

B: Θ (|E| * |V|)

D: **Θ (|V|)**

Exercise 4: Complexity




```
adj = [
    [1],
    [0, 2],
    [1, 3],
    [2]
]
```

```
for u in range(len(adj)):
    for v in adj[u]:
        print(f"({u}, {v})")
```

What is the time complexity in terms of $|V|$ and $|E|$?

A: **Θ (|E| + |V|)**

C: Θ (|E|)

B: Θ (|E| * |V|)

D: Θ (|V|)

Looping Over Edges

- Looping over all edges in this way takes $\Theta(V + E)$ time.
- In aggregate, the print statement is executed:
  - $2|E|$ times if graph is undirected.
  - $|E|$ times if graph is directed.
- This is called an aggregate analysis.

Space Requirements

- Need $\Theta(|V|)$ space for **outer list**.
- Plus $\Theta(|E|)$ space for **inner lists**.
- **In total**: $\Theta(|V| + |E|)$ space.

Example: Facebook

- Facebook has 2 billion users, 400 billion friendships.
- If each edge requires 32 bits: (2 bits × 200 × (2 billion))

  = 64 × 400 × 109 bits

  = 3.2 terabytes

  = 0.04 years of HD video

Dictionary of Sets

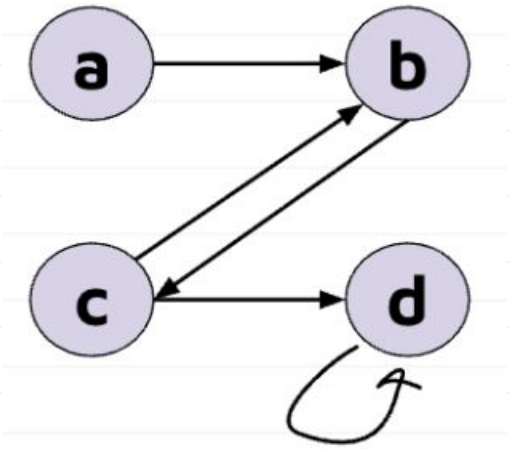
Trade Offs

- **Adjacency matrix**: fast edge query, lots of space.
- **Adjacency list**: slower edge query, space efficient.
- Can we have the best of both?

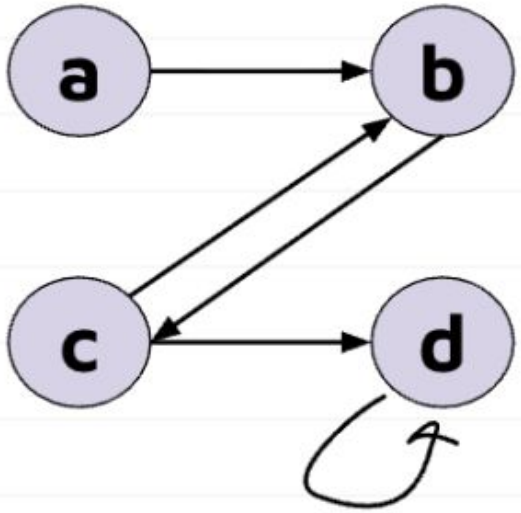
Idea

- Use hash tables.
- Replace inner edge lists by **set**s.
- Replace outer list with **dict**.
  - Doesn’t speed things up, but allows nodes to have arbitrary labels.

Example

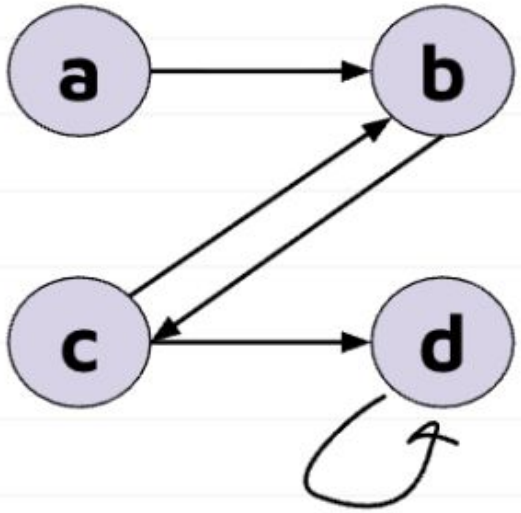


Example



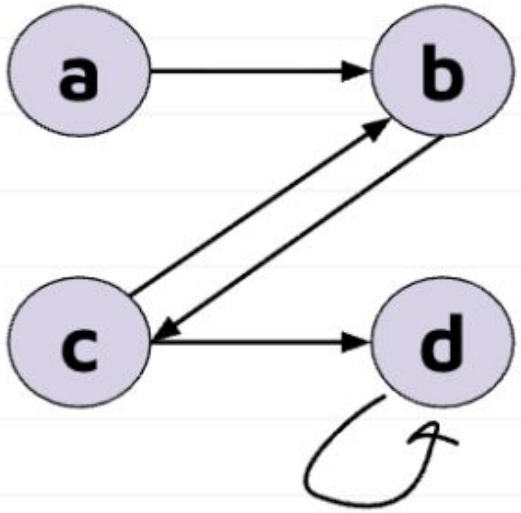

```
adj = {

        ‘a’:

        ‘b’:

        ‘c’:

        ‘d’:

    }
```

Example



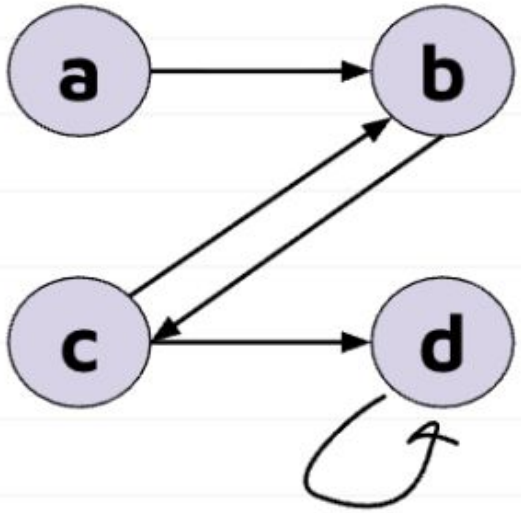

```
adj = {

        'a': {'b'},

        'b':

        'c':

        'd':

    }
```

Example



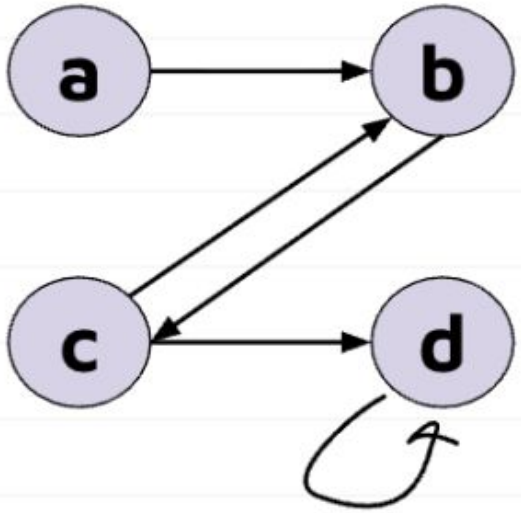

```
adj = {

        'a': {'b'},

        'b': {'c'},

        'c':

        'd':

    }
```

Example




```
adj = {

        'a': {'b'},

        'b': {'c'},

        'c': {'b', 'd'},

        'd':

    }
```

Example




```
adj = {

        'a': {'b'},

        'b': {'c'},

        'c': {'b', 'd'},

        'd': {'d'}

    }
```

Time Complexity

operation	code	time
Edge query	`j in adj[i, j]`	Θ(1) expected
Degree(i)	`len(adj[i])`	Θ(1) on average

Space Requirements

- Requires only $\Theta(V + E)$.
- But there is overhead to using hash tables.

Dict-of-sets implementation

- Install with `pip install dsc40graph`
- Import with `import dsc40graph`
- **Docs**: `https://eldridgejm.github.io/dsc40graph/`
- **Source code**: `https://github.com/eldridgejm/dsc40graph`
- Will be used in HW coding problems.
- Real world: networkX

Thank you!

Do you have any questions?

CampusWire!