Graphs

I already discussed graphs in lecture 11.

Graph: (V, E)

V: Set of vertices

E: Set of edges.

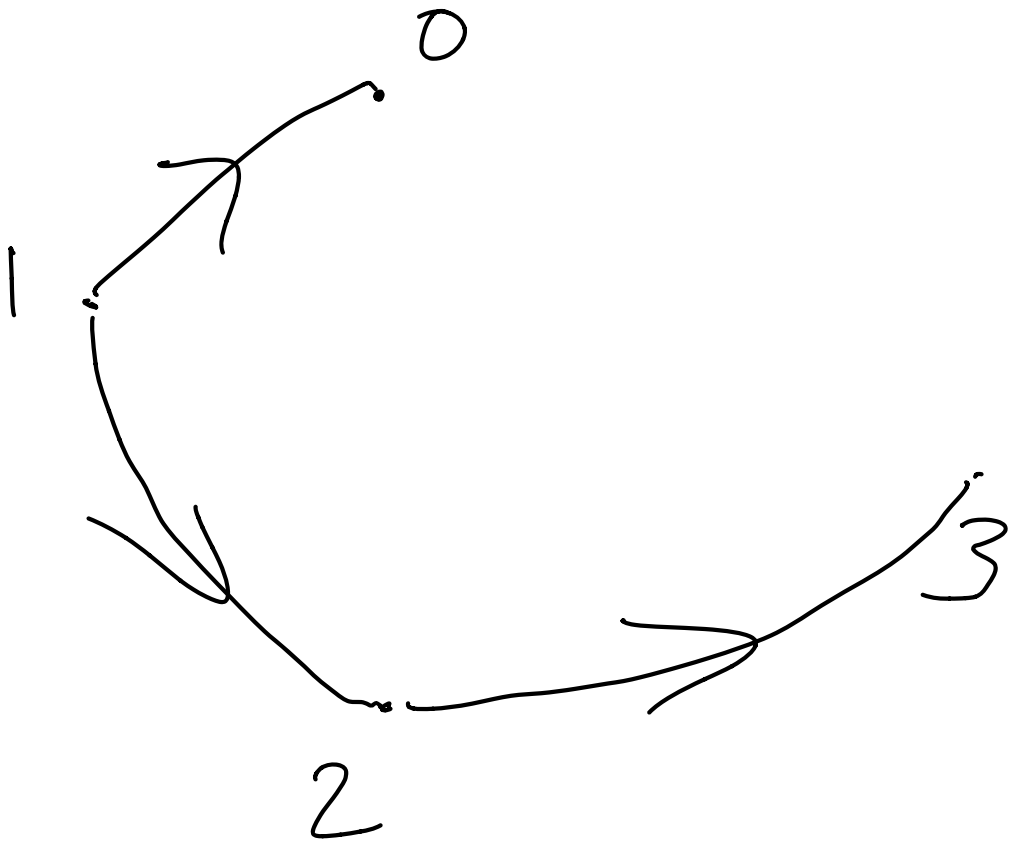
$V = \{0, 1, 2, 3\}$

$E = \{(1,2), (2,3), (1,0)\}$

$(1,2) \neq (2,1)$

order matters

Represent this pictorially



Undirected version

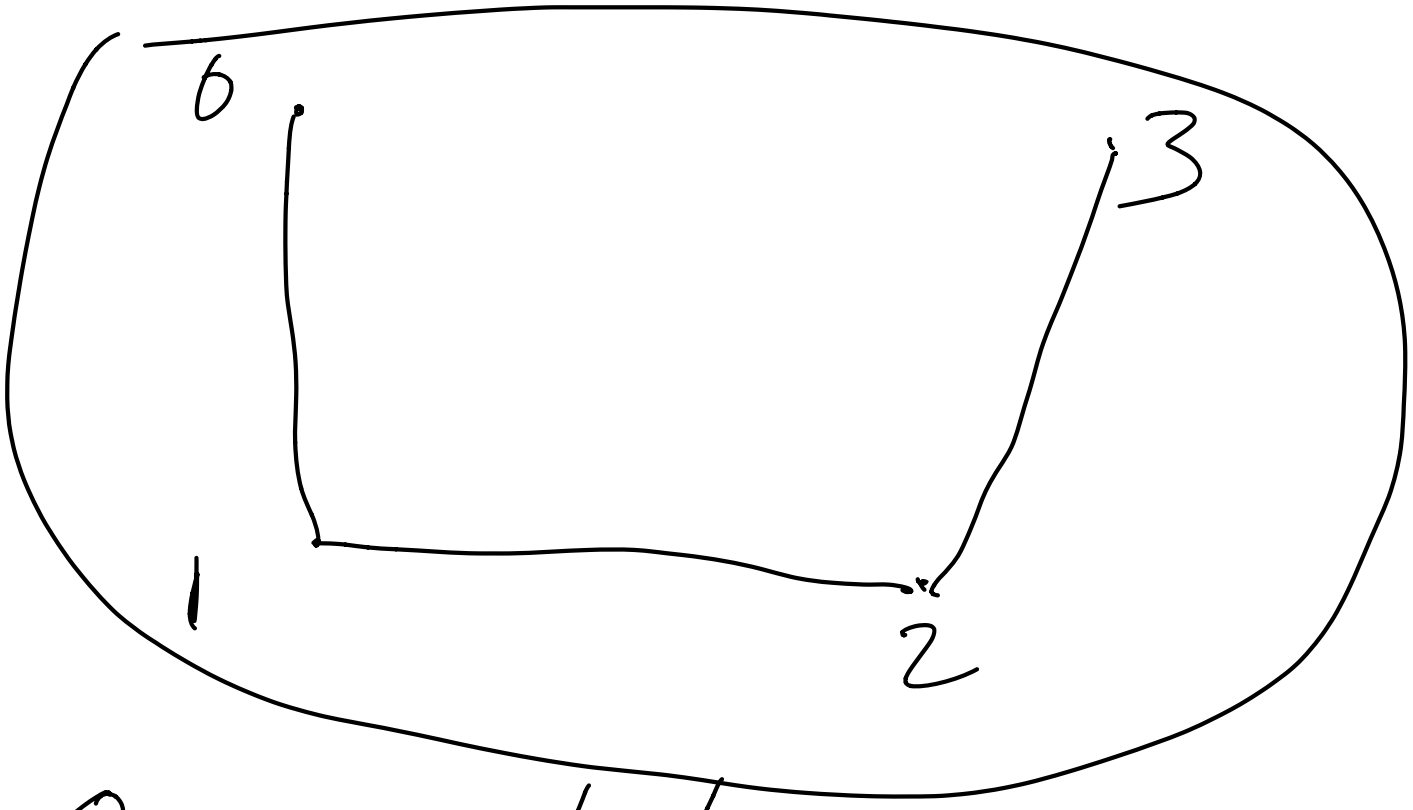
$V = \{0, 1, 2, 3\}$

$E = \{ \{1,2\}, \{2,3\}, \{1,0\} \}$

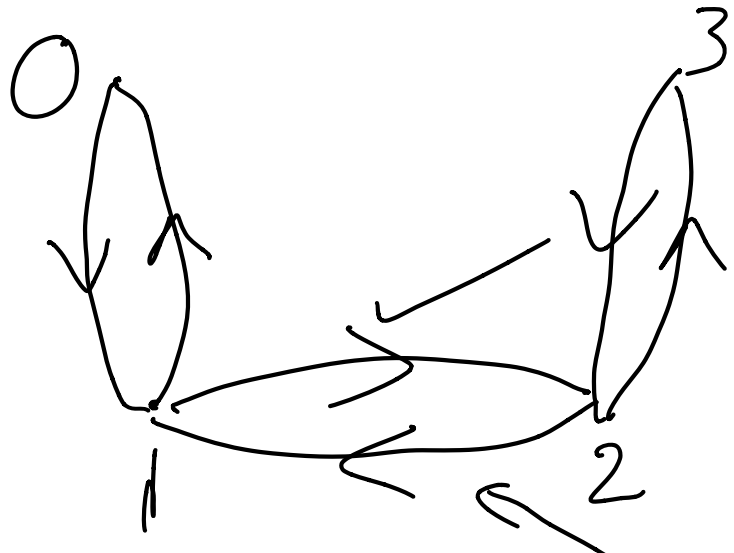
set

order doesn't matter.

$\{1,2\} = \{2,1\}$



One important every undirected graph can be though of a directed graph.

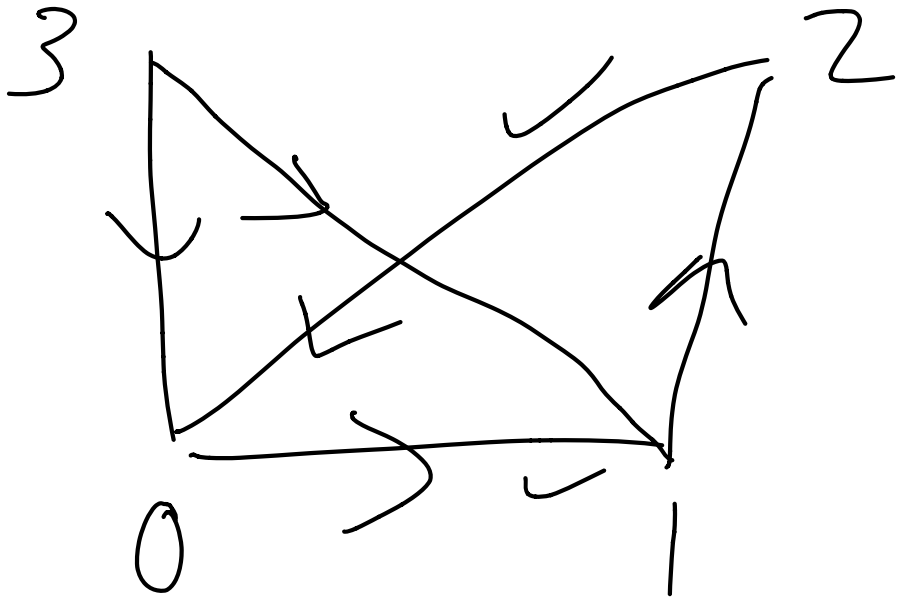


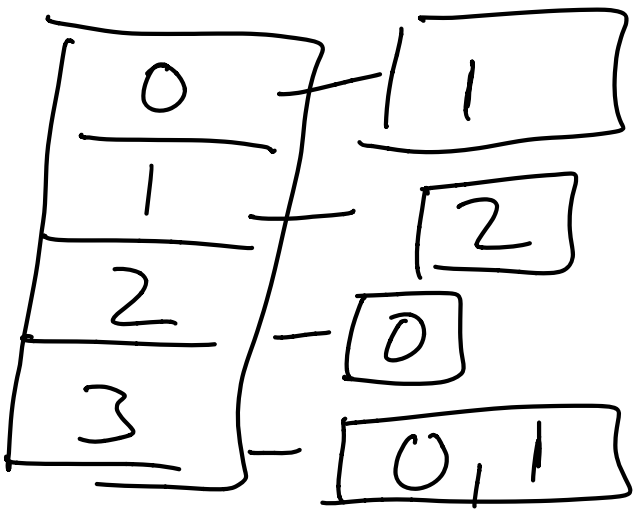
$V = \{0, 1, 2, 3\}$

$E = \{(0,1), (1,0), (1,2), (2,1), (3,2), (2,3)\}$.

That's we will focus on directed graphs.

We represent graphs using adjacency list.

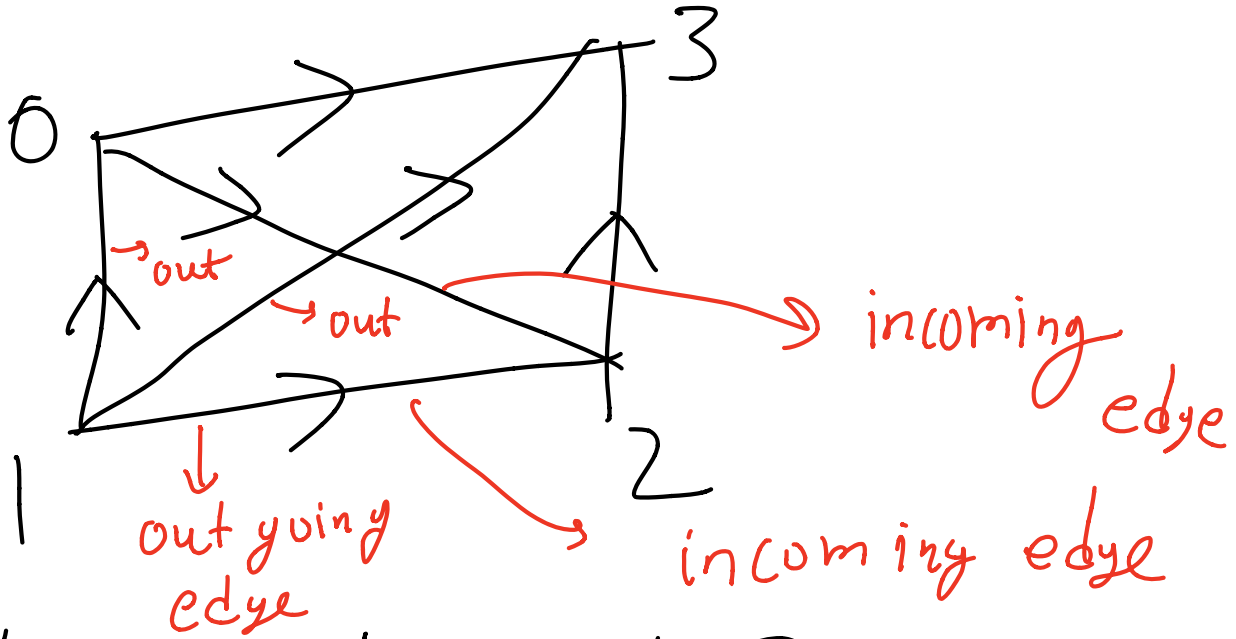




Degree

in-degree

in-degree of a vertex is the number of incoming edges.



What is in-degree of 2 = 2

What is the in-degree of 1 = 0

out-degree

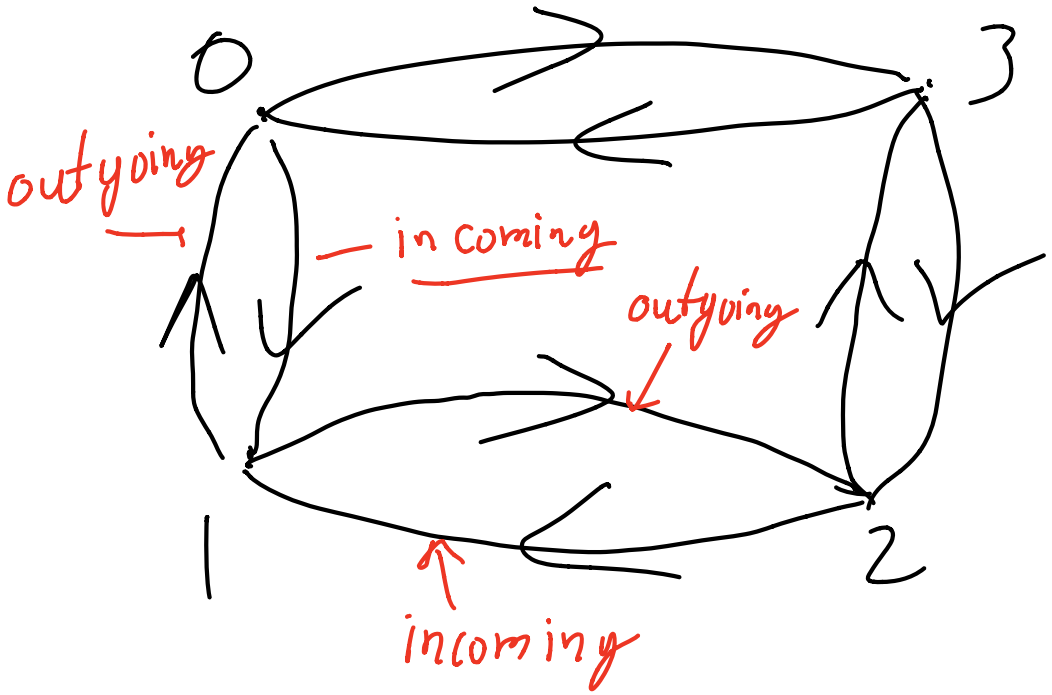
Out-degree of a vertic is number of outgoing edge.

What is out-degree of 1 = 3.

Sometimes, degree of a vertex

they assume the graph is undirected.

in-degree of any vertex
= out-degree of that vertex.
= degree of that vertex
(HW 11)



This is an undirected graph.

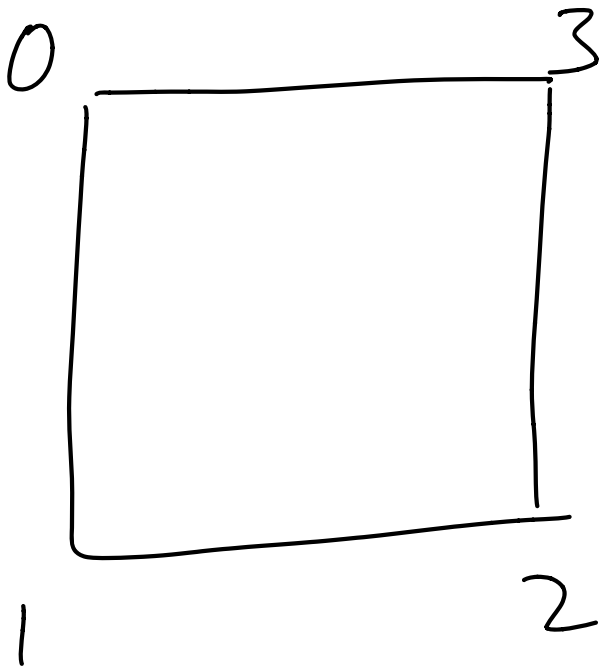
in-degree of 1 = 2

out-degree of 1 = 2

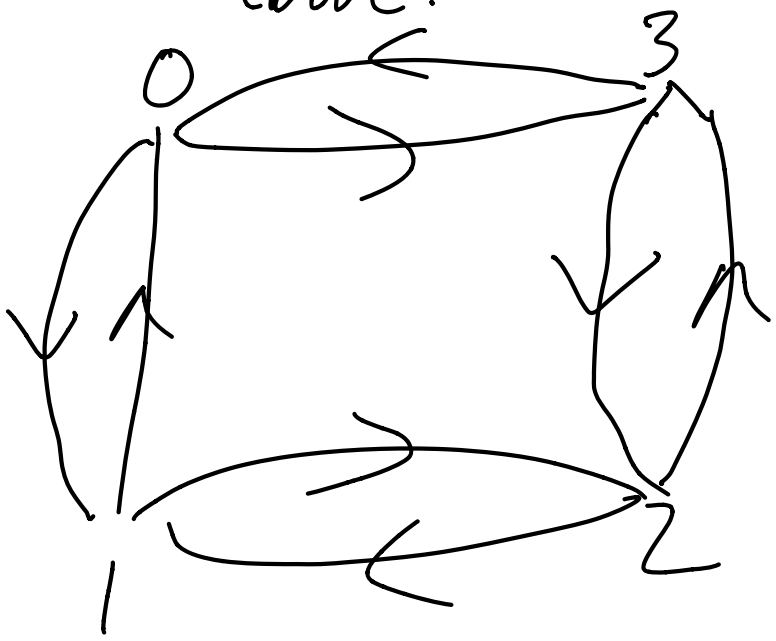
Represent graph in racket.

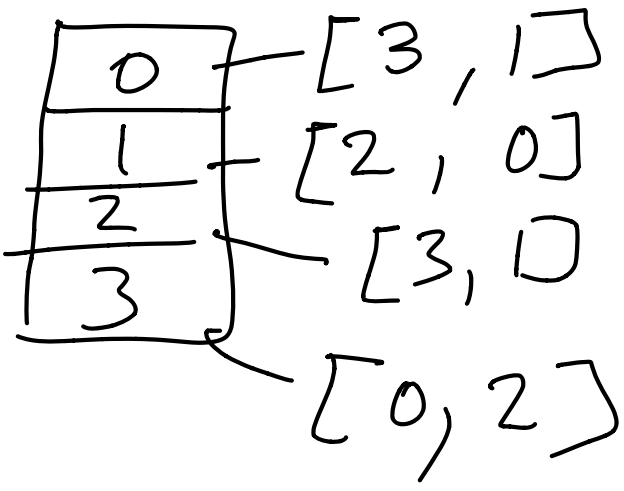
```
(define-type Vertex Integer)
(define-struct Graph
  ([n : Integer]
   [adj : (Vectorof (Listof Vertex))]))
```

→ Number of vertices in the graph



Directed version of the above.

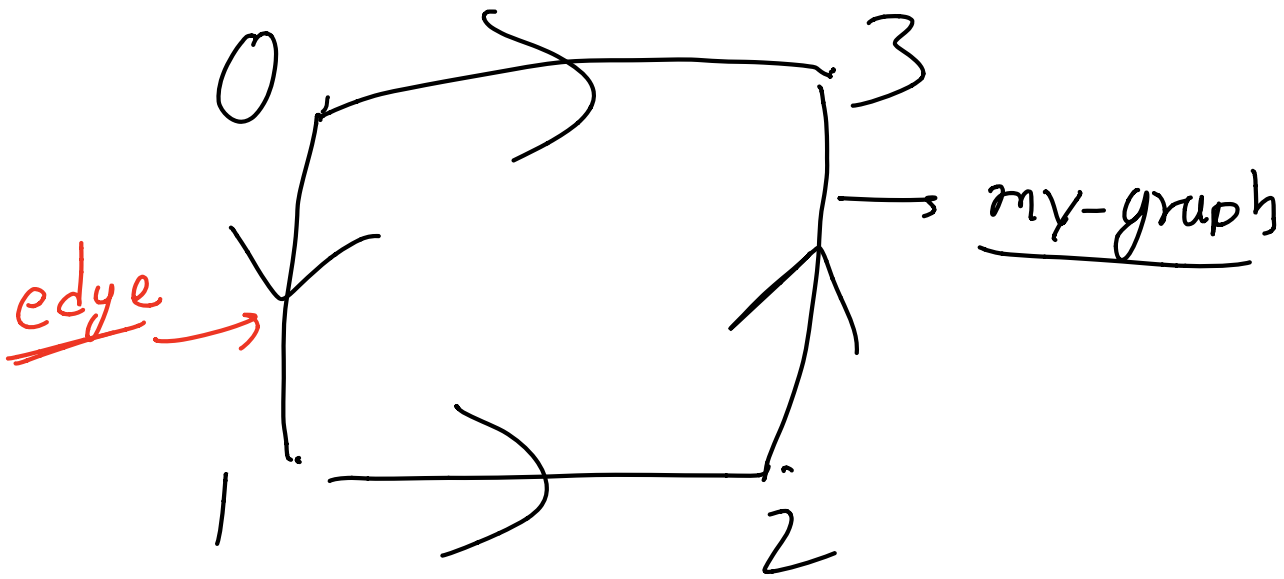





```
(: my-graph : Graph)
(define my-graph (Graph 4
  (vector (list 3 1) (list 2 0)
          (list 3 1) (list 0 2))))
```

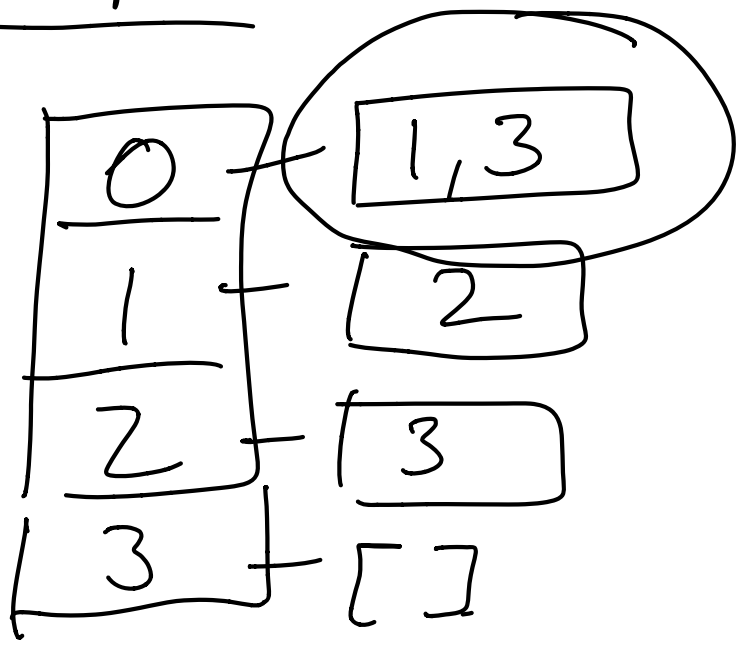
Already did get-nbrs in lec11

Adjacent:
Write a function that takes a graph and two vertices a and b and checks if there is an edge from a to b.



(adjacent? my-graph 0 1) ⇒
True.

Step 1



Step 1:

get-nbrs of 0.

⇒ 1,3

Check if 1 is is 1,3

⇒ True

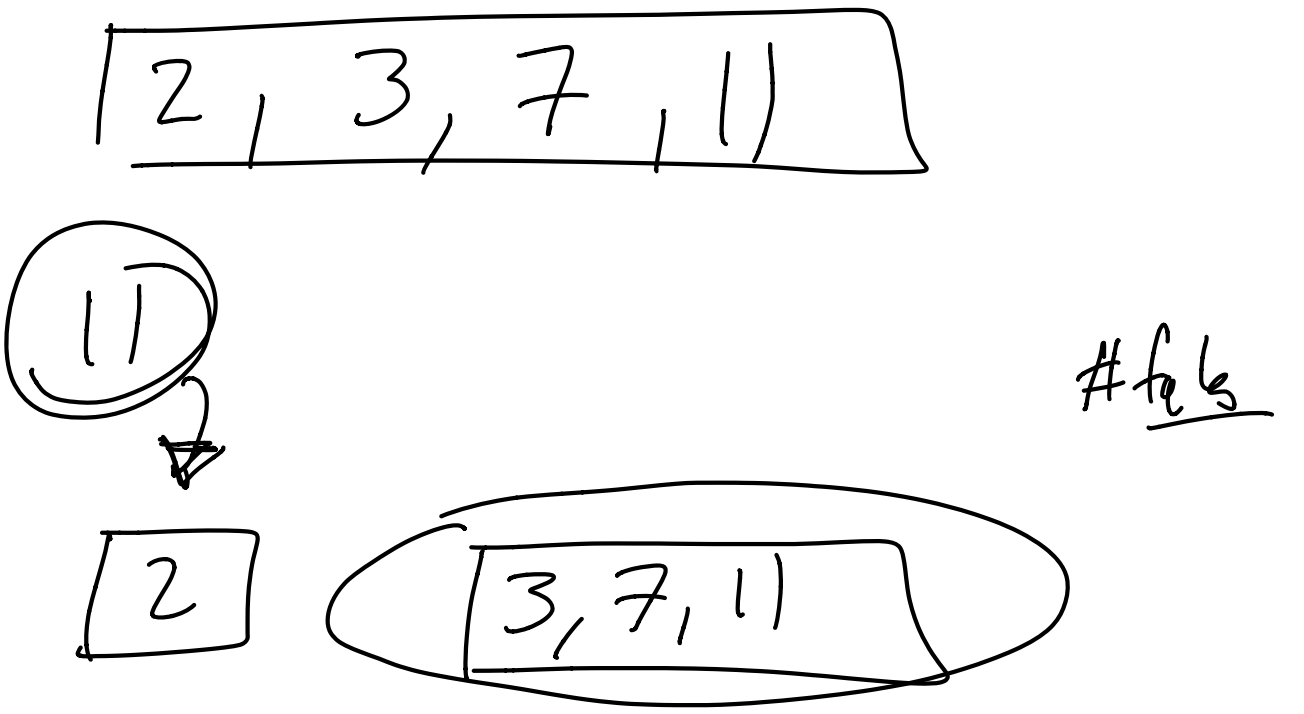
Write a function which takes a list as input and an integer as input and checks if

that integer is in the list!

[3, 4, 5, 6.] → my-list

5 (in-list? my-list 5) ⇒ true

```
(: in-list? : (Listof Vertex)
              Vertex → Boolean)
(define (in-list? vtxs v)
  (cond
   [(empty? vtxs) #f]
   [else (or (= v (first vtxs))
             (in-list? (rest vtxs) v))]))
```



#fds

(: adjacent? : Graph Vertex Vertex → Boolean)

(define (adjacent? g u v)

(in-list? (get-nbrs g u) v))

This will get me

the neighbours of u.

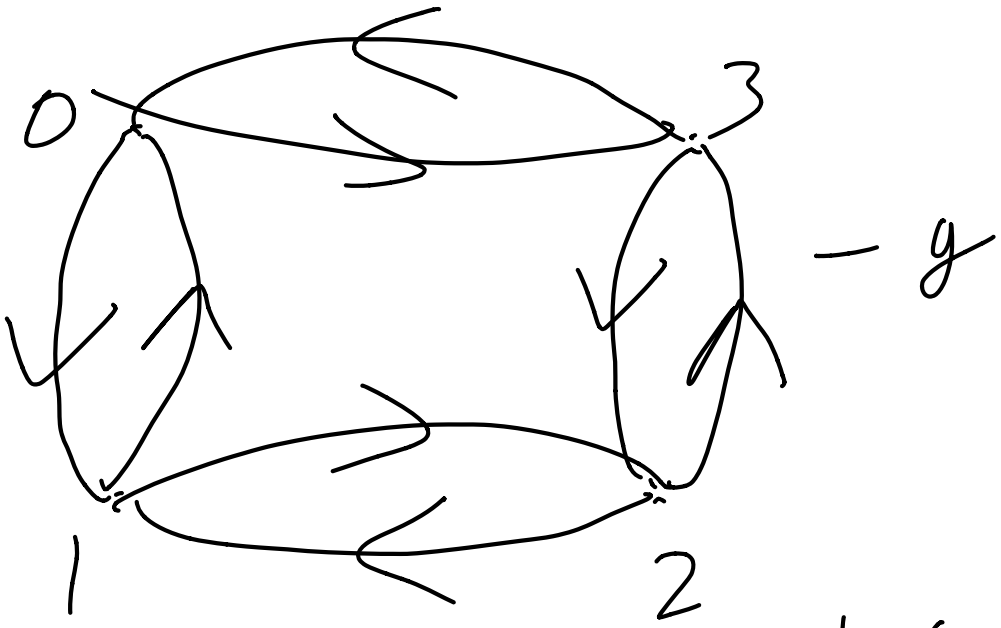
list

Example

Write a function degree which takes a graph and vertex as input and outputs degree of this vertex

degree

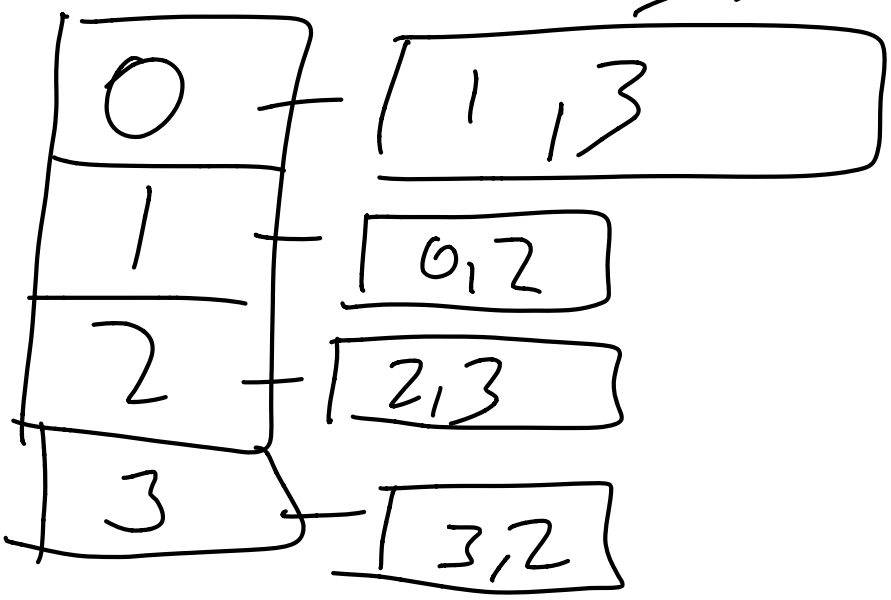
```
(: degree : Graph Vertex -> Integer)

(length (get-nbrs g v)))
```



$- g$

output (yet-nbrs g u)

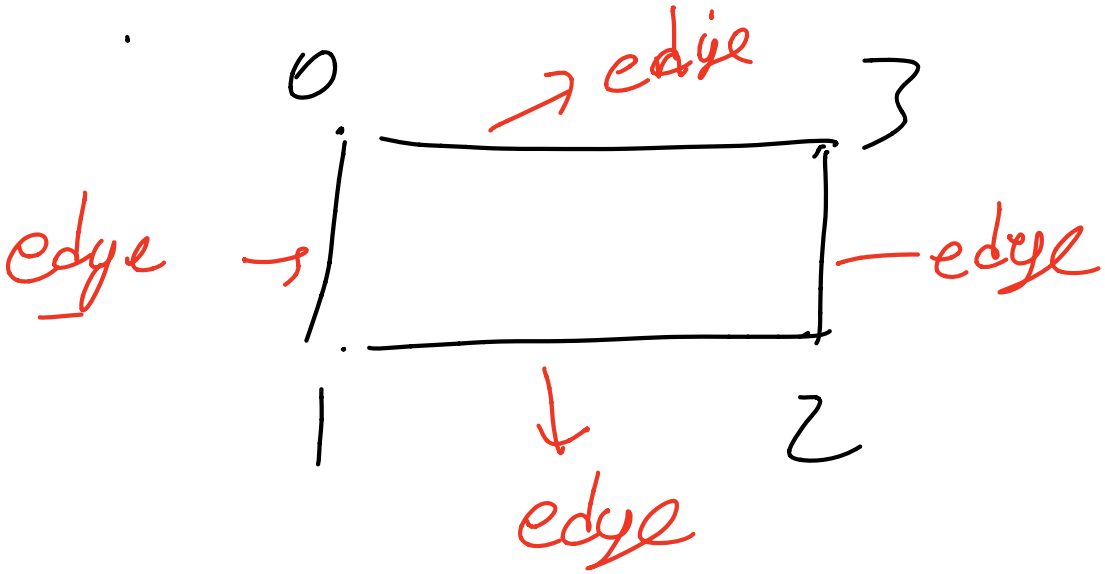
$u = 0$



degree of 0

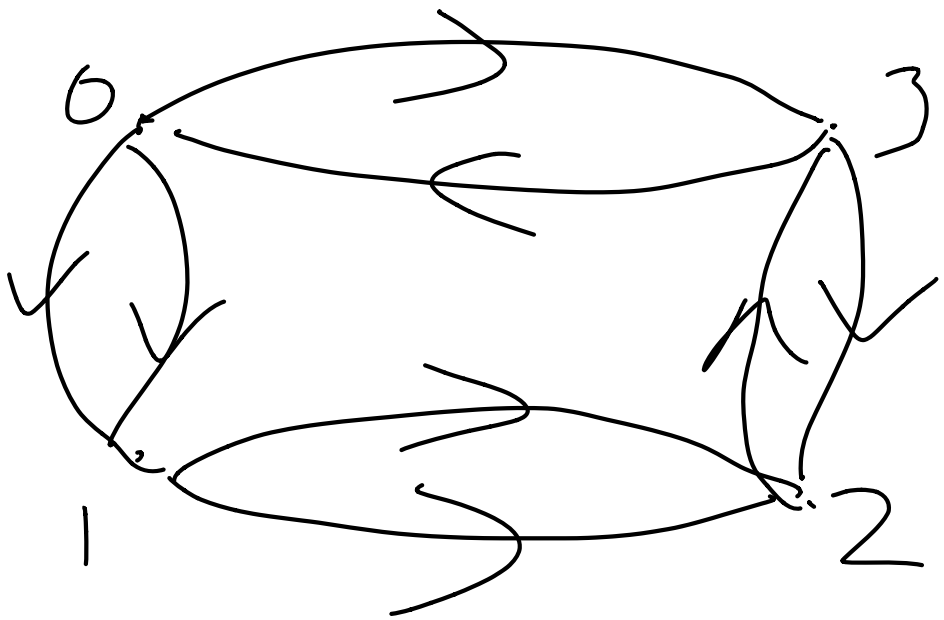
(len [1,3]) = 2.

Num of edges



Num of edges
= 4

Undirected version

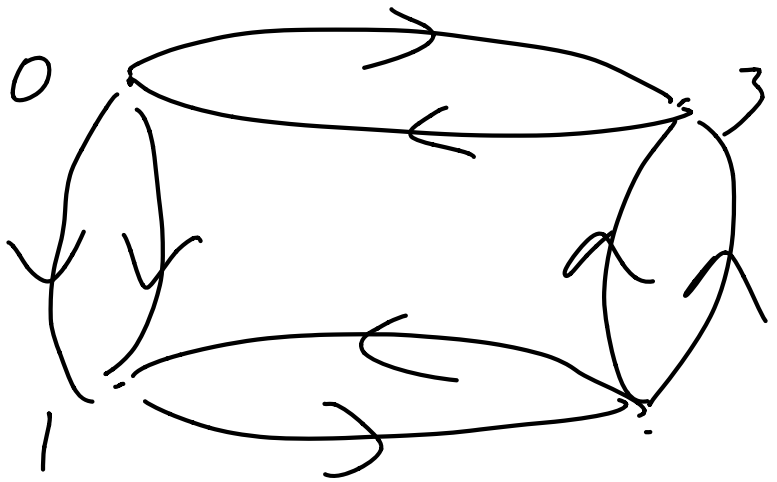


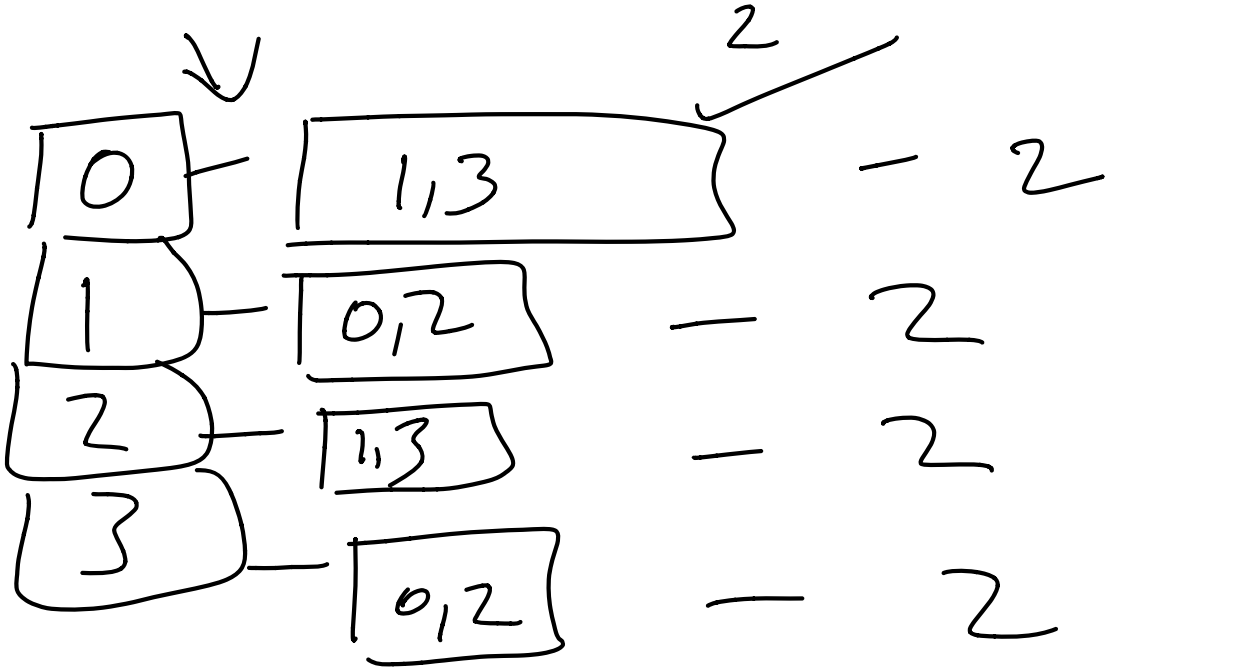
Num of edges
= 8

Num of edges in directed version
= 2 * Num of edges in

undirected version.

Write a function that compute number of edges in a directed graph.





```
(: num-edges : Graph → Integer)
(define (num-edges g)
```

```
(vector-map length
            (graph-adj g))
```

This will not work

length is polymorphic

You can define your lambda

```
(inst length vertex)
```

```
(: num-edges : Graph → Integer)
(define (num-edges g)
  (foldr + 0
    (vector->list (vector-map length
                              (graph-adj g)))))
```

function

Remark

vector-map will output a vector

foldr doesn't work on vectors

So convert vector to a list

by vector->list

Instantiate length
(inst length int)

This is to prevent type-checks errors.

dfs check if a graph is connected.