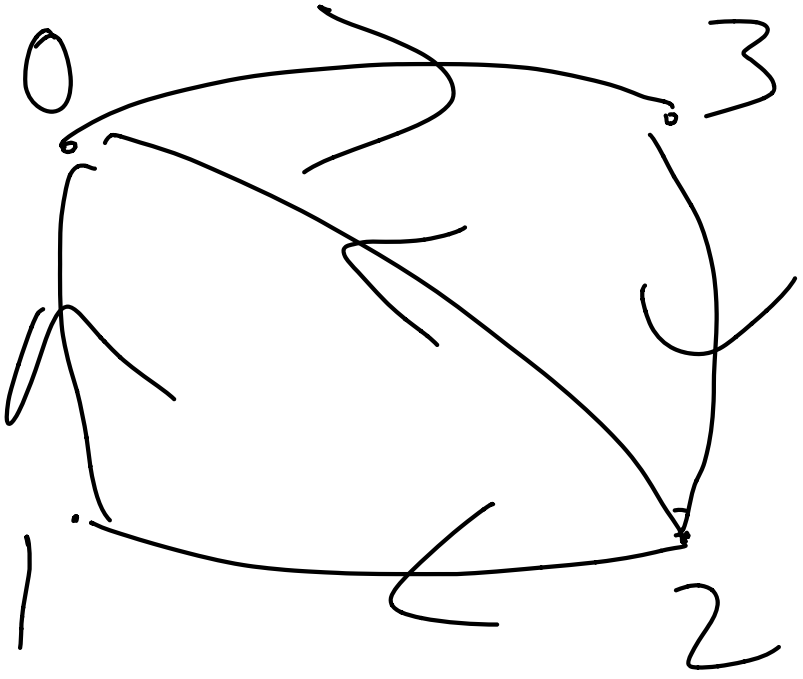
Lecture 12

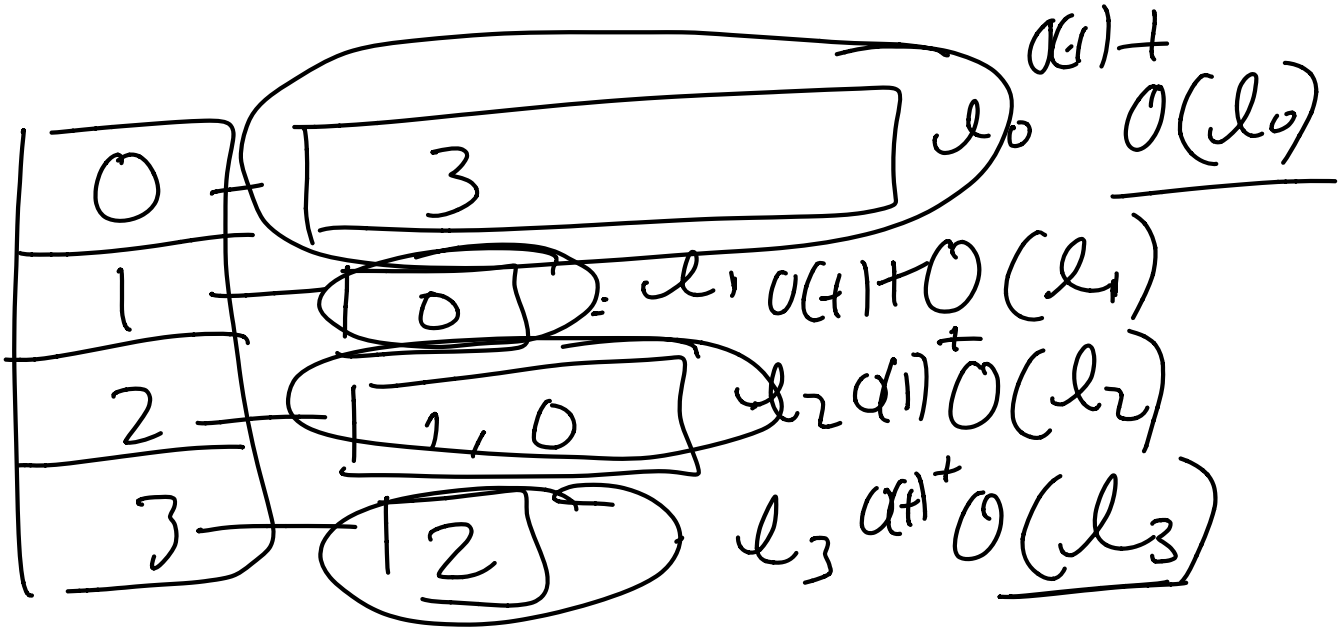
HW11

Problem 1

```
(: count-degree : Graph Integer
   -> Integer)
(define (count-degree g d)
  (vector-count (lambda ([nbrs : (Listof
    Vertex)]) (= d (length nbrs)))
   (graph-adj g)))
```

Time complexity





$$O(1)+O(l_0)$$
$$+O(1)+O(l_1)+O(1)+O(l_2)$$
$$+O(1)+O(l_3)$$

$$l_0 = \underline{\underline{d_0}}$$

$O(n) + O(d_0 + d_1 + d_2 + d_3)$

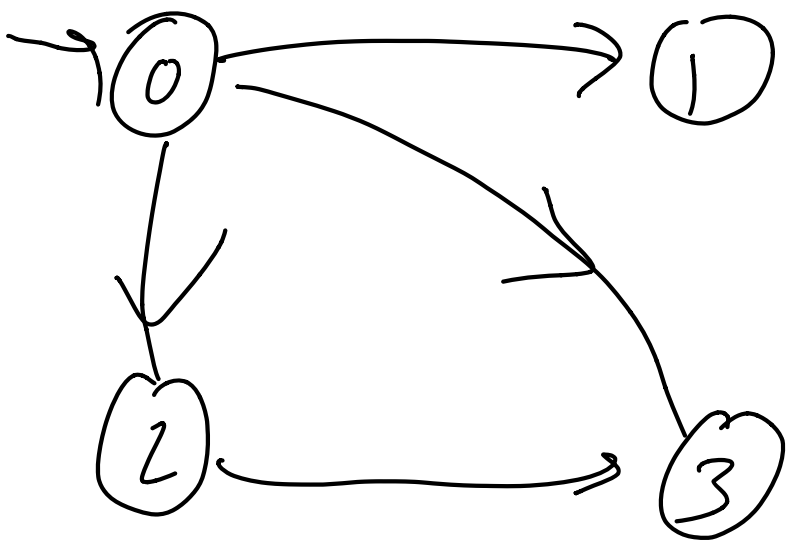
$d_0 + d_1 + d_2 + d_3$ = "num of edges in a directed graph"

$O(n) + O(m)$

$= O(n+m)$

Depth-First-Search

You can store information in these nodes



Every node in your can represent web page

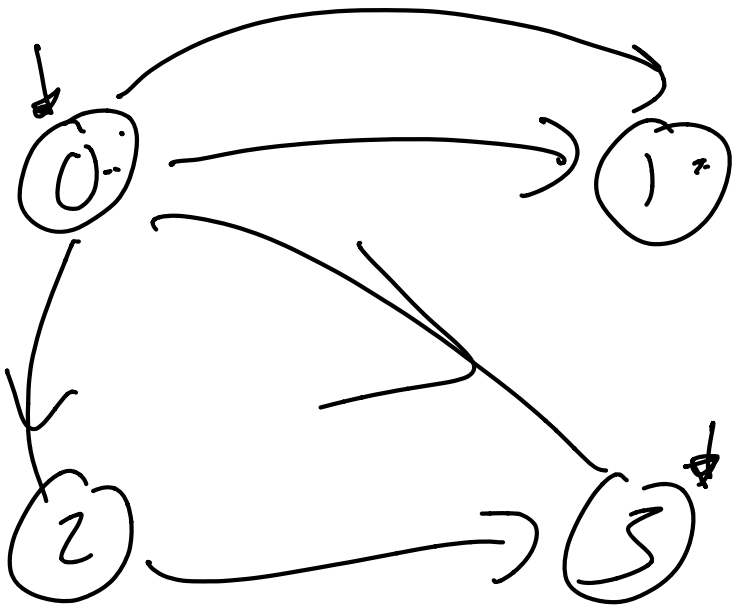
Arrow will represent a link from webpage 1 to webpage 2.

How will you visit every node in a graph.

BFS (Breadth-First-Search)

DFS (Depth-First-Search)

Naturally recursive



0 → [1, 2, 3] , [2, 1]

1 → [ ]

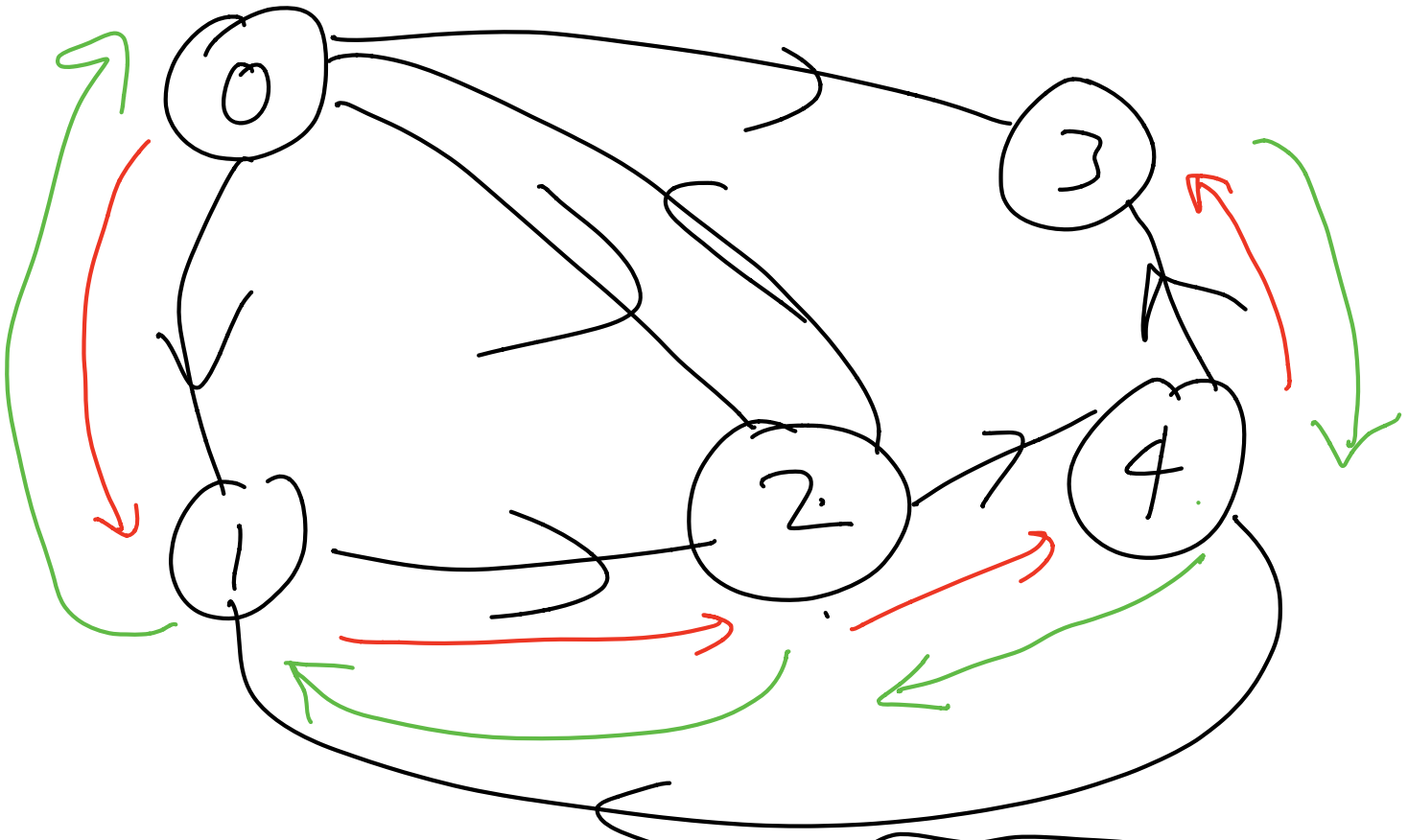
2 → [3]

3 → [ ]

0, 1, 2, 1

0, 1, 2, 3

visited boolean



0 - [1, 2, 3]
1 - [2]
2 - [0, 4]
3 - []
4 - [3, 1]

1 [3, 2, 1]

Starting vertex is 0

Visited

F	F	F	F	F
0	1	2	3	4

Step 1

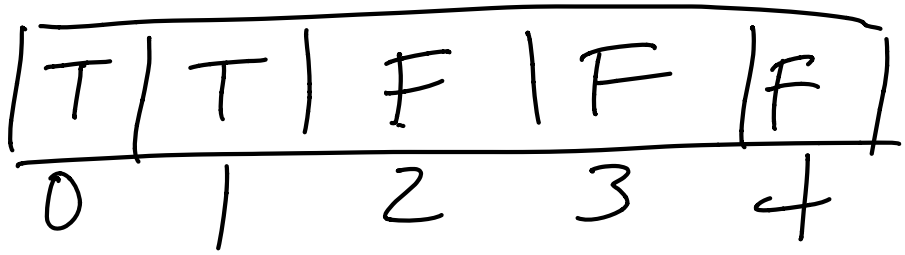
Visit node 0

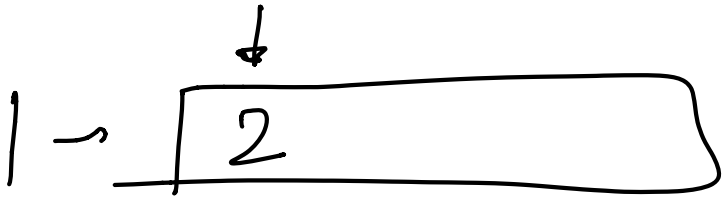
T	F	F	F	F
0	1	2	3	4

0 → 1, 2, 3

Step 2

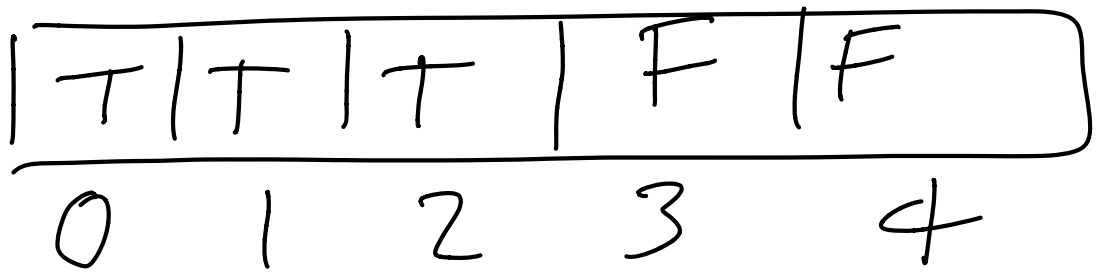
visit node 1

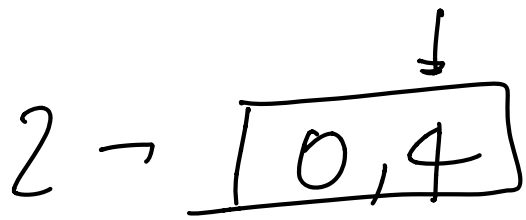




Step 3

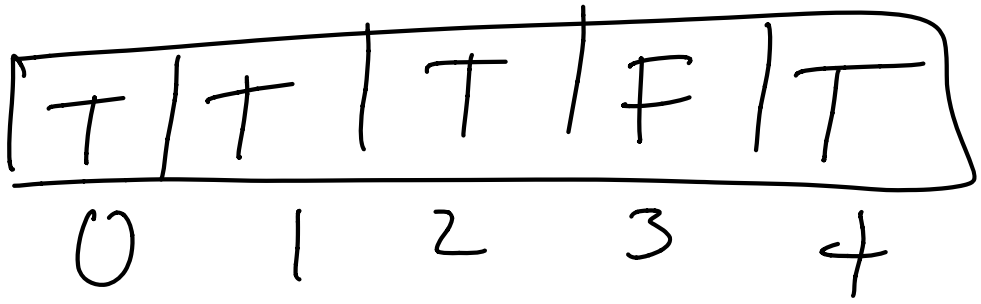
visit node 2

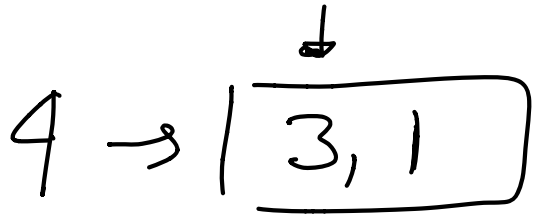




Step 4

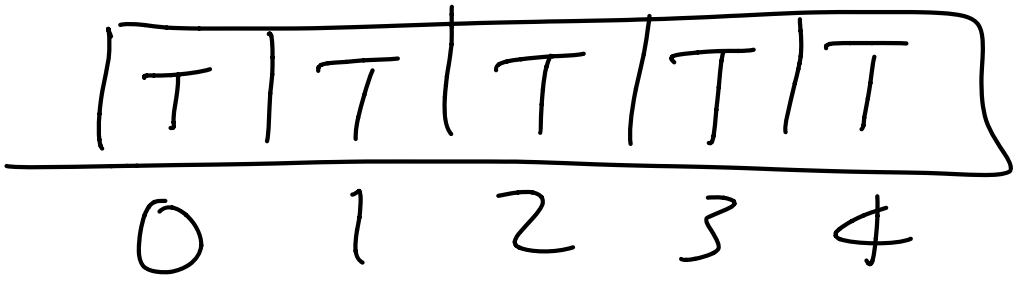
visit node 4



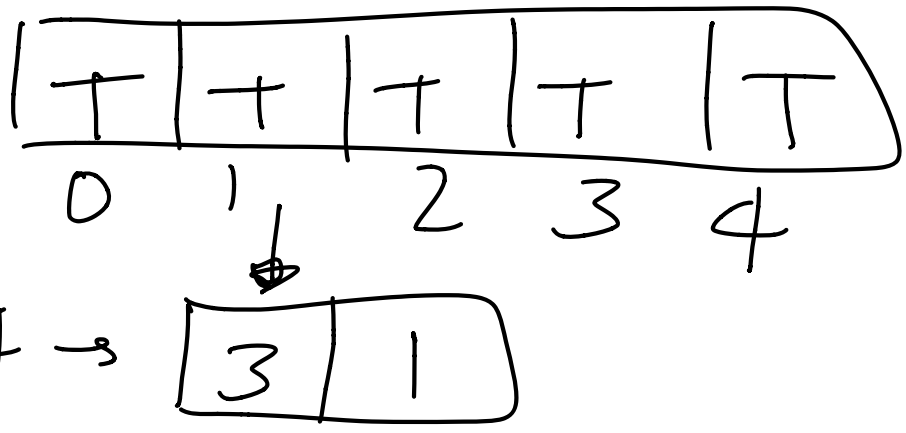


Step 5

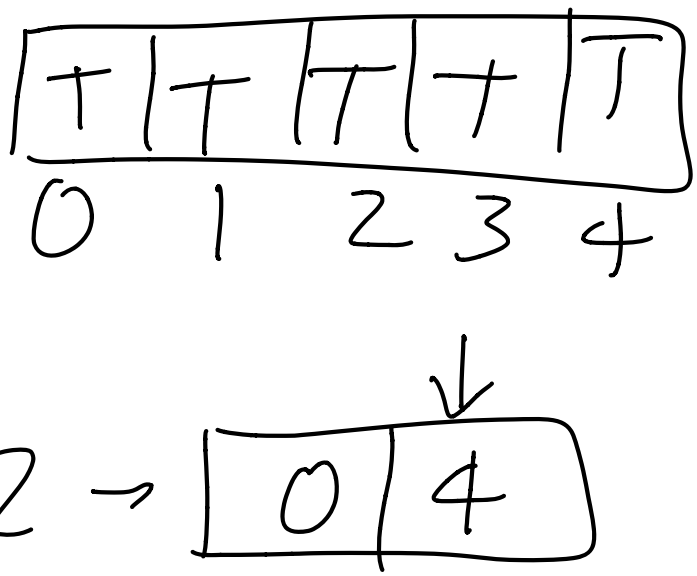
Visit node 3



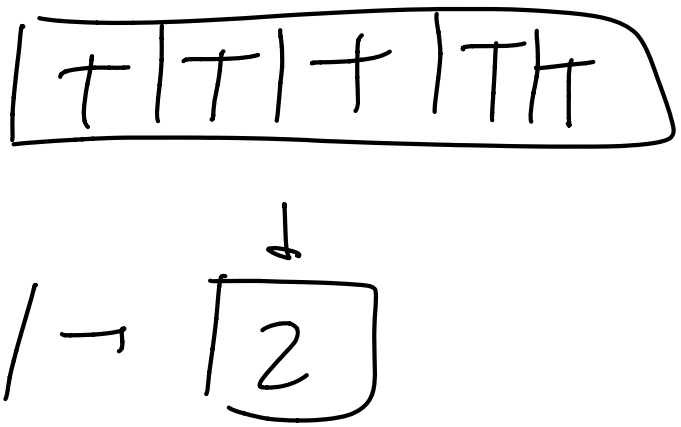
Step 6



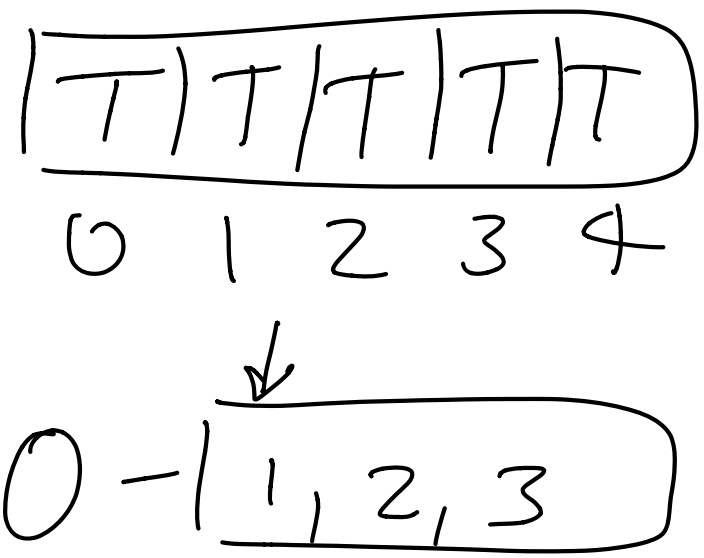
Step 7



Step 8



Step 9



0 finished

DFS ends

Recursion

```
dfs(v)
    visited[v] = True
    for every neighbor of v, u
        if u is not visited
            dfs(u)
```

```
(define-type Vertex Integer)

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

starting vertex
↓

```
(: dfs! : Graph Vertex (Vectorof
   Boolean) -> Void)

(define (dfs! g v visited)
 (begin

  (vector-set! visited v #t)
 (local
  [
   (: explore-list! : (Listof Vertex)
                      -> Void)
   (define (explore-list! nbrs)
    (cond
     [(empty? nbrs) (void)]
[else (begin (if (vector-ref visited
                  (first nbrs))
```

First

(void)
(dfs! g (first nbrs)
visited))

else → (explore-list! (rest nbrs))]))
y
(explore-list! (get-nbrs g v)))

explore all the neighbors of v

To scan a list use recursion
scan the first element
and then recursively scan
rest.

Time complexity

Your input is a node, a graph

Input size : n, - number of vertices

m - number of edges

$T(n, m) =$

Recurrence relation will not really work here

DFS - check for every node

$O(n) + O(d_0 + d_1 + d_2 + \dots + d_{n-1})$.

$= m$

Number of nodes — Exploring the neighbours

$O(n+m)$ — informal reasoning

Let $T(n, m)$ be the time complexity of dfs.

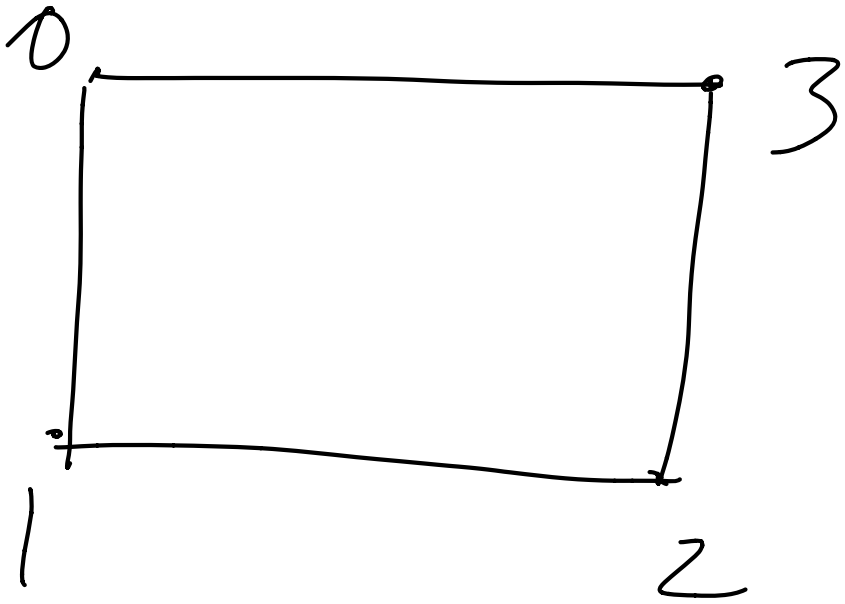
$T(n,m) \in O(n+m)$

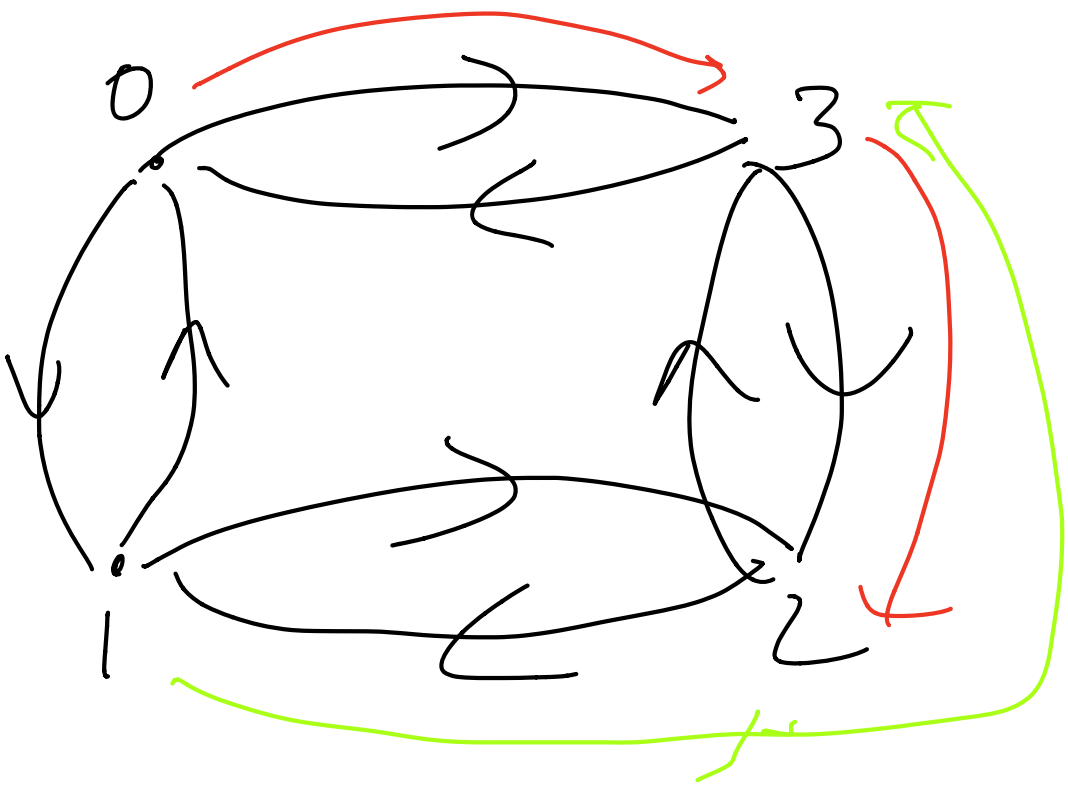
$f(n,m) = n+m$

$O(f(n,m))$

How to check if a graph is connected

Undirected graph

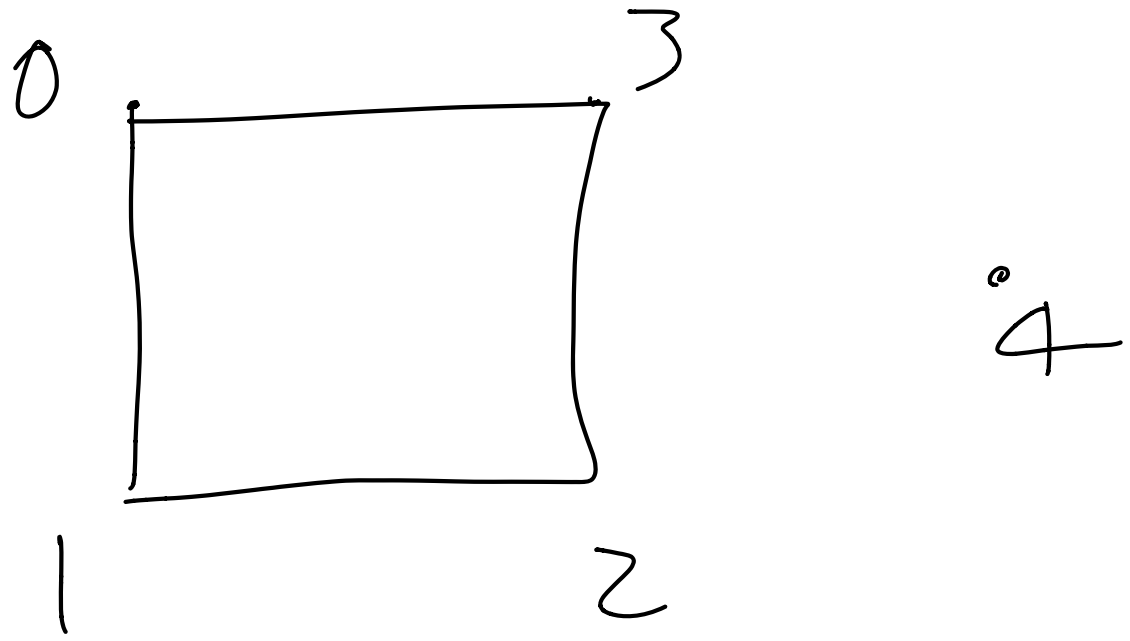


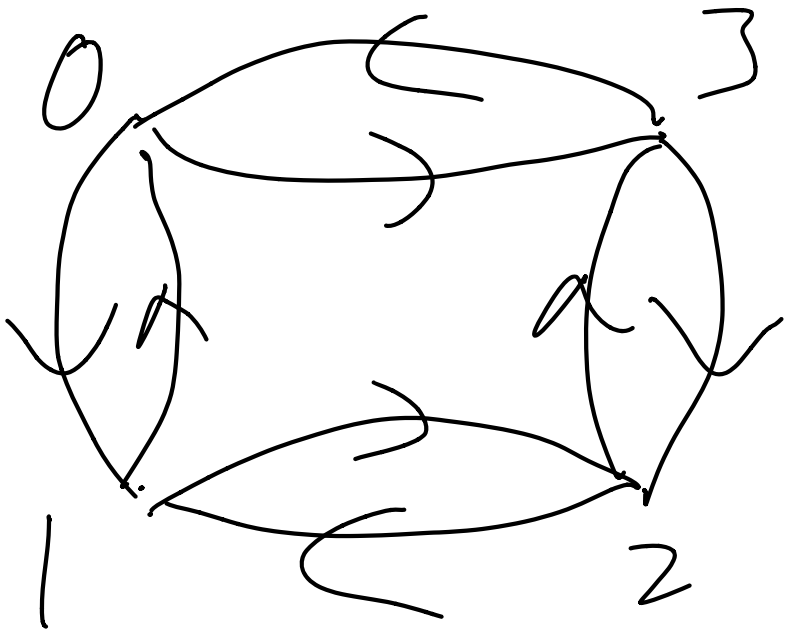


This graph is connected:

You can go from any vertex to any other vertex.

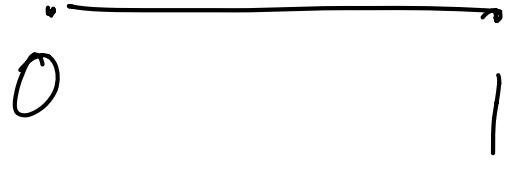
Disconnected graph

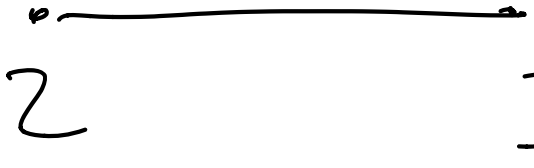


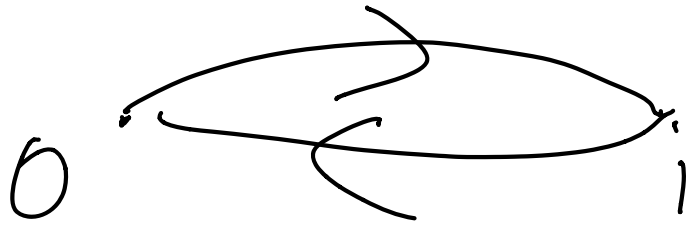


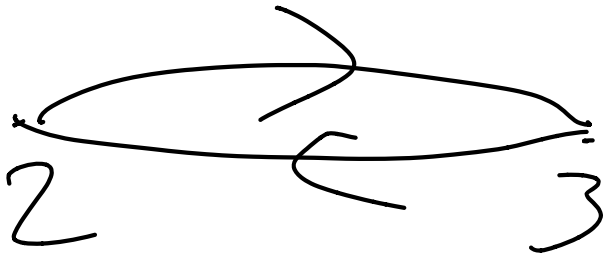
This is not connected.

There is no path from say 1 to 4.







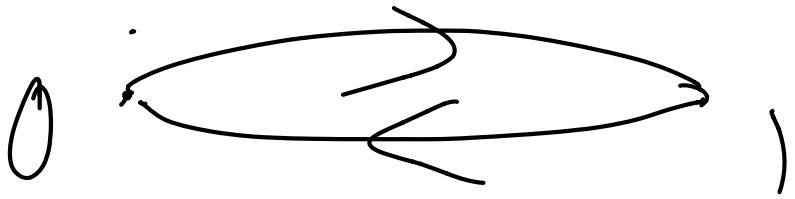


This not connected

We don't have a path from 0 to 2

How to check if a graph is connected?

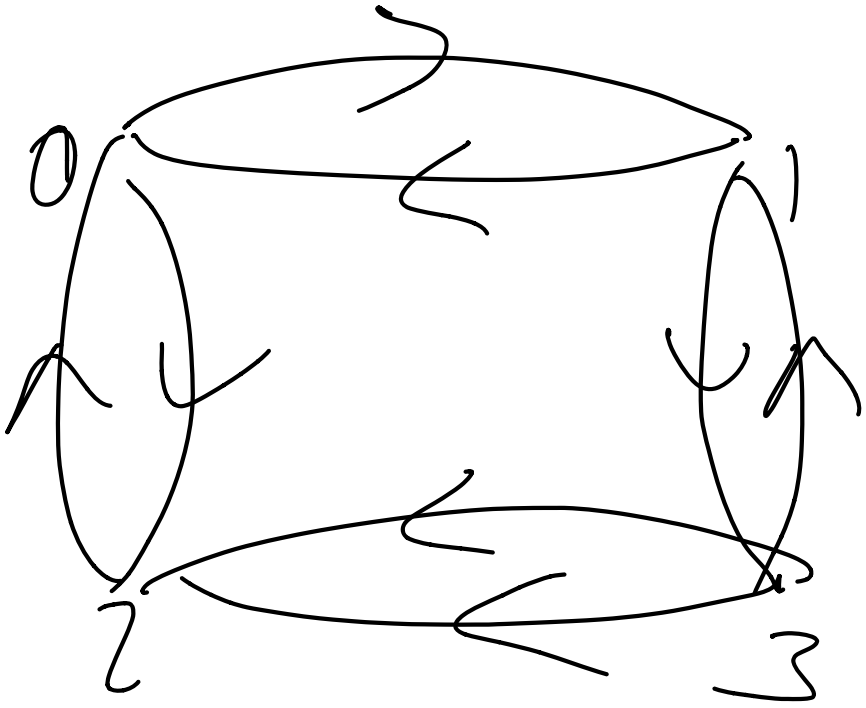
Do Dfs on that graph.



F	F	F	F
0	1	2	3

(dfs! on 0 visited)

T	T	F	F
0	1	2	3



→ g2.

Connected graph

visited vector

F	F	F	F
0	1	2	3

(dfs! g2 0 visited)

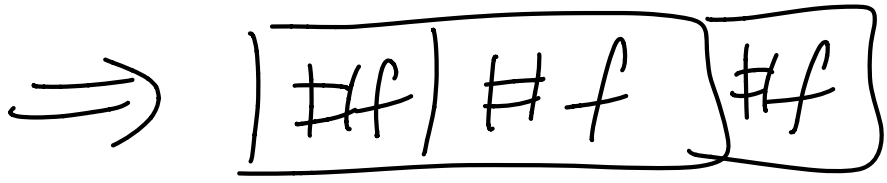
T	T	T	T
0	1	2	3

Write a function to check if a graph is connected

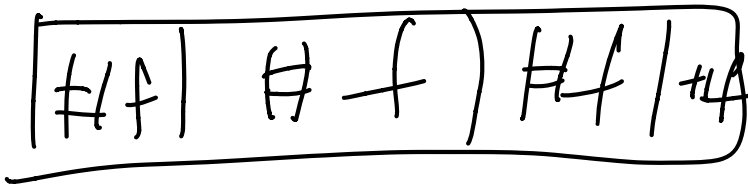
```
(: connected? : Graph → Boolean)
(define (connected? g)
  (local
    {(: visited : (Vectorof Boolean))
     (define visited (make-vector
                      (graph-n g) #f))}
    (begin (dfs1 g 0 visited)
           (andmap (lambda ([b : boolean])
                     b)
                   (vector->list visited)))))
```

make-vector takes 2 inputs

(make-vector 3 #f)

⇒ 

(make-vector 4 #f)

⇒ 

And map - takes two inputs.

(A → Boolean) — f

list of A

(andmap f my-list)

my-list [a1, a2, a3, a4]

[f(a1), f(a2), f(a3), f(a4)]

⇒ (and f(a1) f(a2) f(a3)
f(a4))

Similarly there is a or-
does or.

For this function

[#t #f #t #f]

So f in our case should be the identity function which means output is same as input.

```
(andmap (lambda ([b : Boolean])
          b)
        (list #t #f #t))

(list #t #f #t)
⇒ #f
```

Time-complexity of connected.

$$\frac{O(n+m)}{= O(n+m)} + O(n)$$

Time-complexity of connected
= Time-complexity of dfs

+ Time-complexity of andmap.

Time-complexity of dfs
$= O(n+m)$

Tim-complexity of andmap
$= O(f \cdot n)$

where f is the Time
-complexity of our lambda.

The time complexity of lambda is constant $O(1)$.

So, time-complexity of andmap is $O(n)$.

So total time-complexity $= O(n+m) + O(n) = O(n+m)$.

Difference between selectors and constructors functions

```
(define-struct Point
  ([x : Integer]
```

[y : Integer]))

Point-x
Point-y
—— Selector functions

(Point 1 2) ⇒ Point with x value 1 and y value 2

↓
constructor function