Graphs

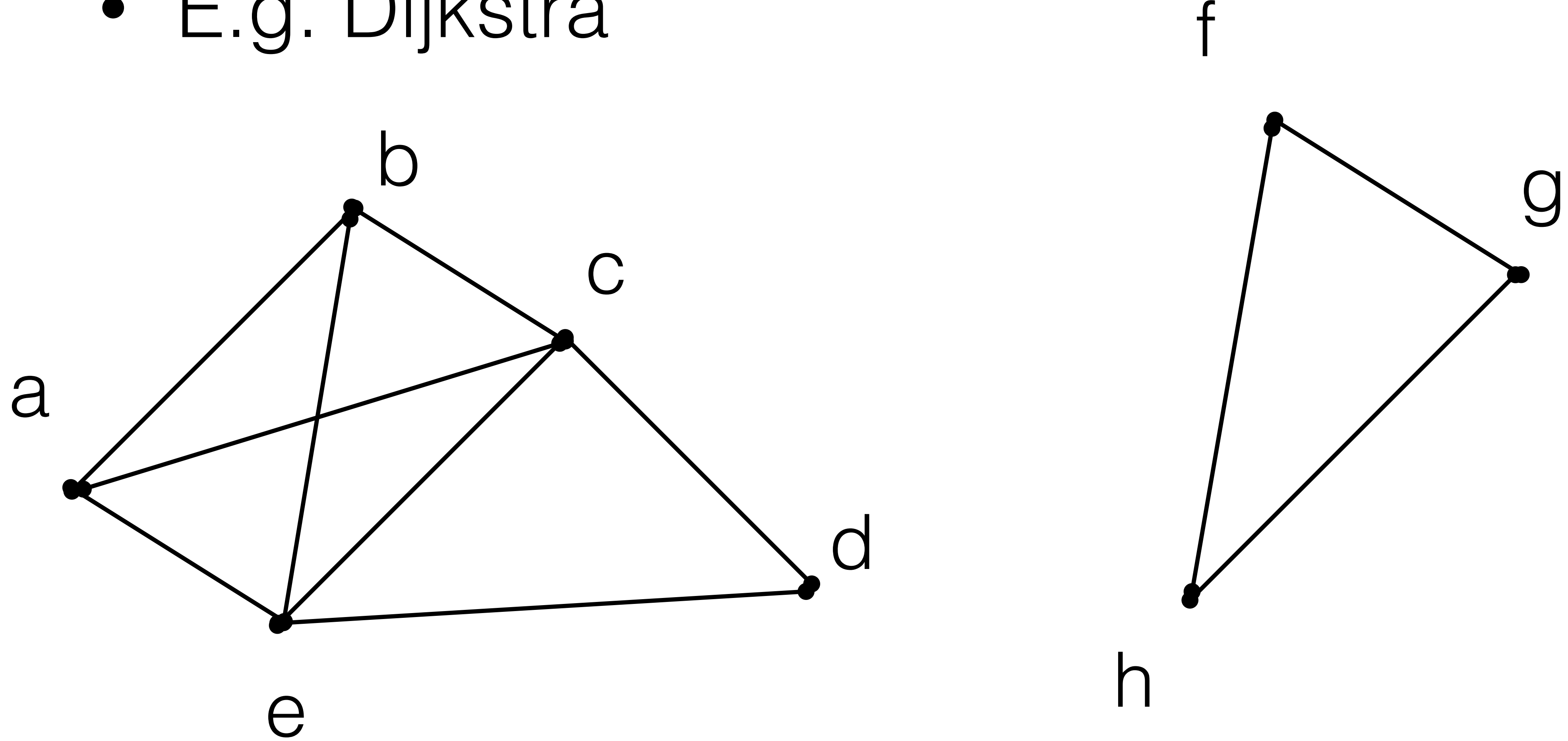
Lecture17

Two techniques for algorithm design

- We have seen recursion techniques so far
- Next few weeks, we will see graph algorithms
- E.g. Dijkstra…

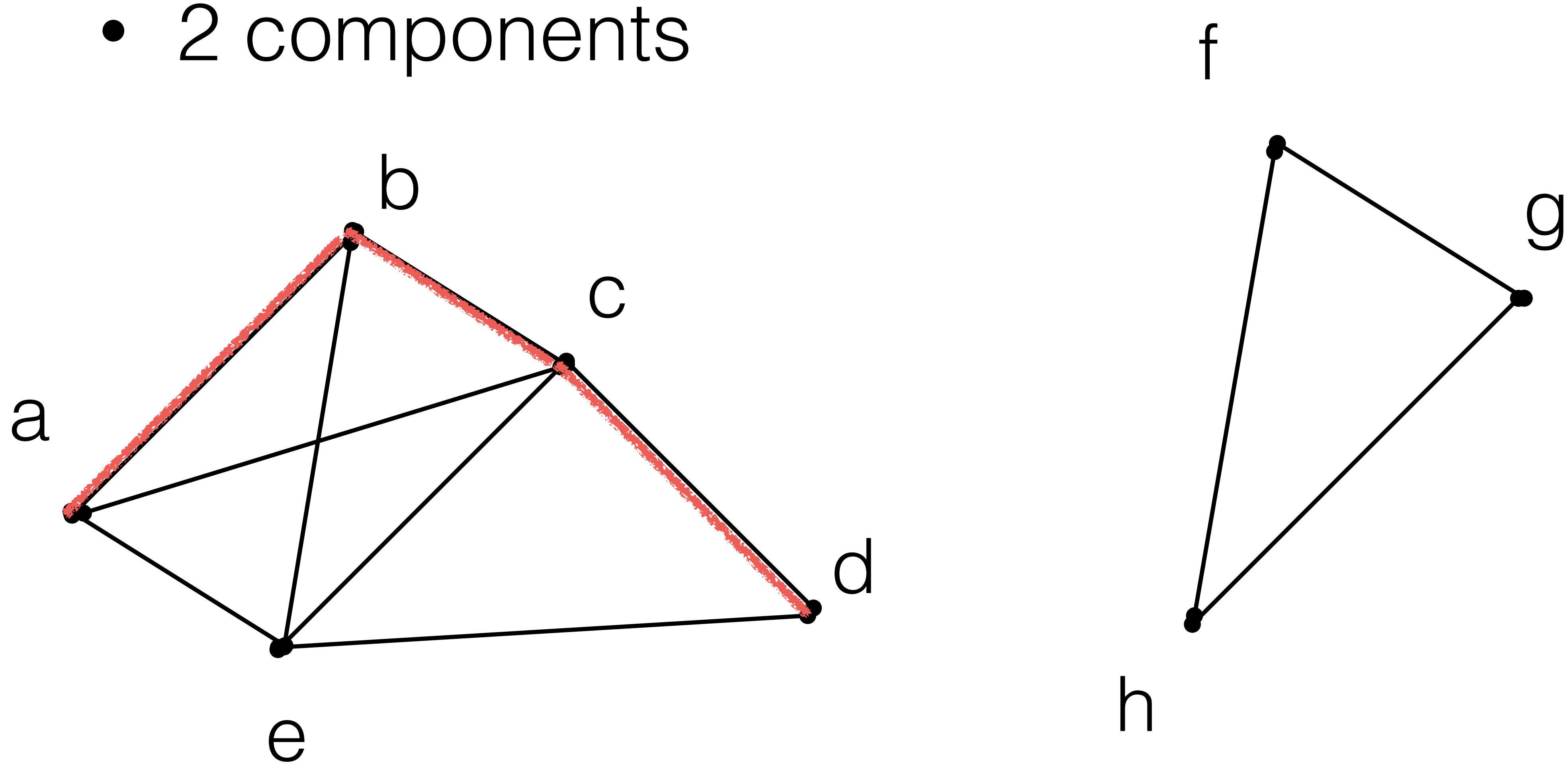
Two techniques of algorithm design

- We have seen recursion techniques so far
- Next few weeks, we will see graph algorithms
- E.g. Dijkstra



Graphs

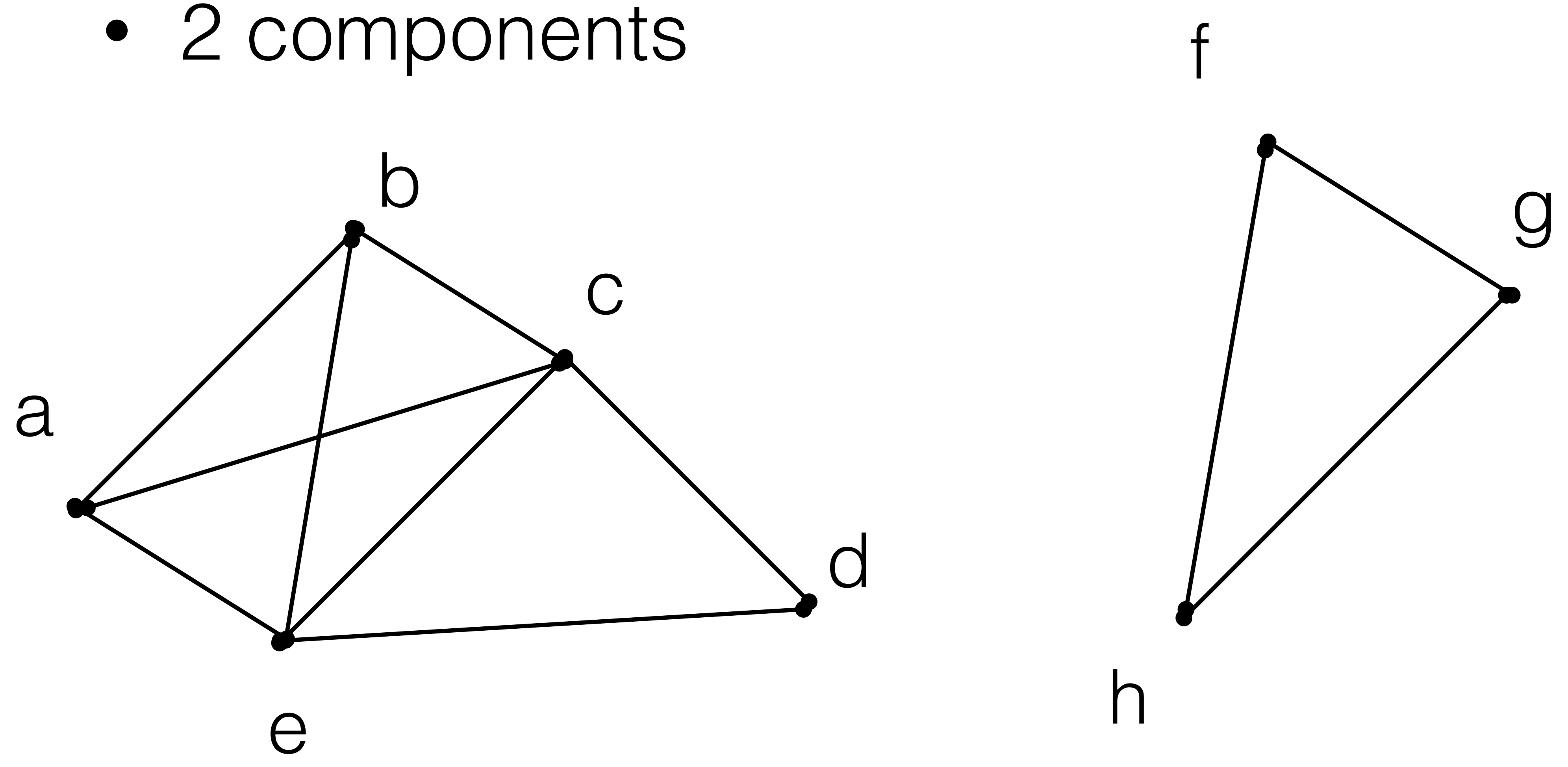
- 8 vertices
- 10 edges
- 2 components



Graph Representation

- 8 vertices
- 10 edges
- 2 components

Just a representation of the graph!



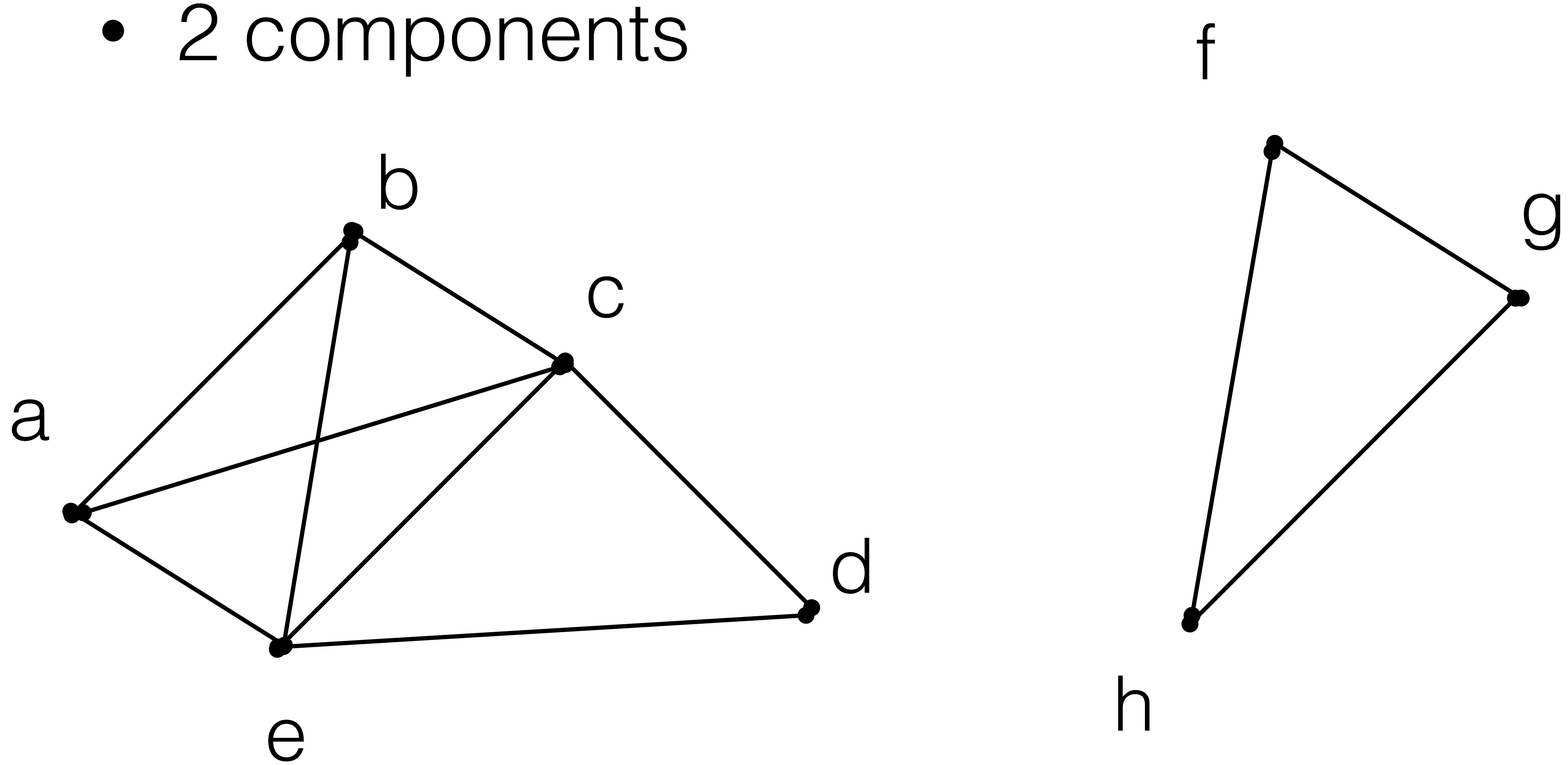
Graph Definition

- A (simple) graph is
  - Non empty finite set V, called vertex set
  - Set E of pairs of vertices, called edges.
  - Undirected (u,v)={u,v}
  - Directed (u,v) = u → v

Graph Representation

- 8 vertices
- 10 edges
- 2 components

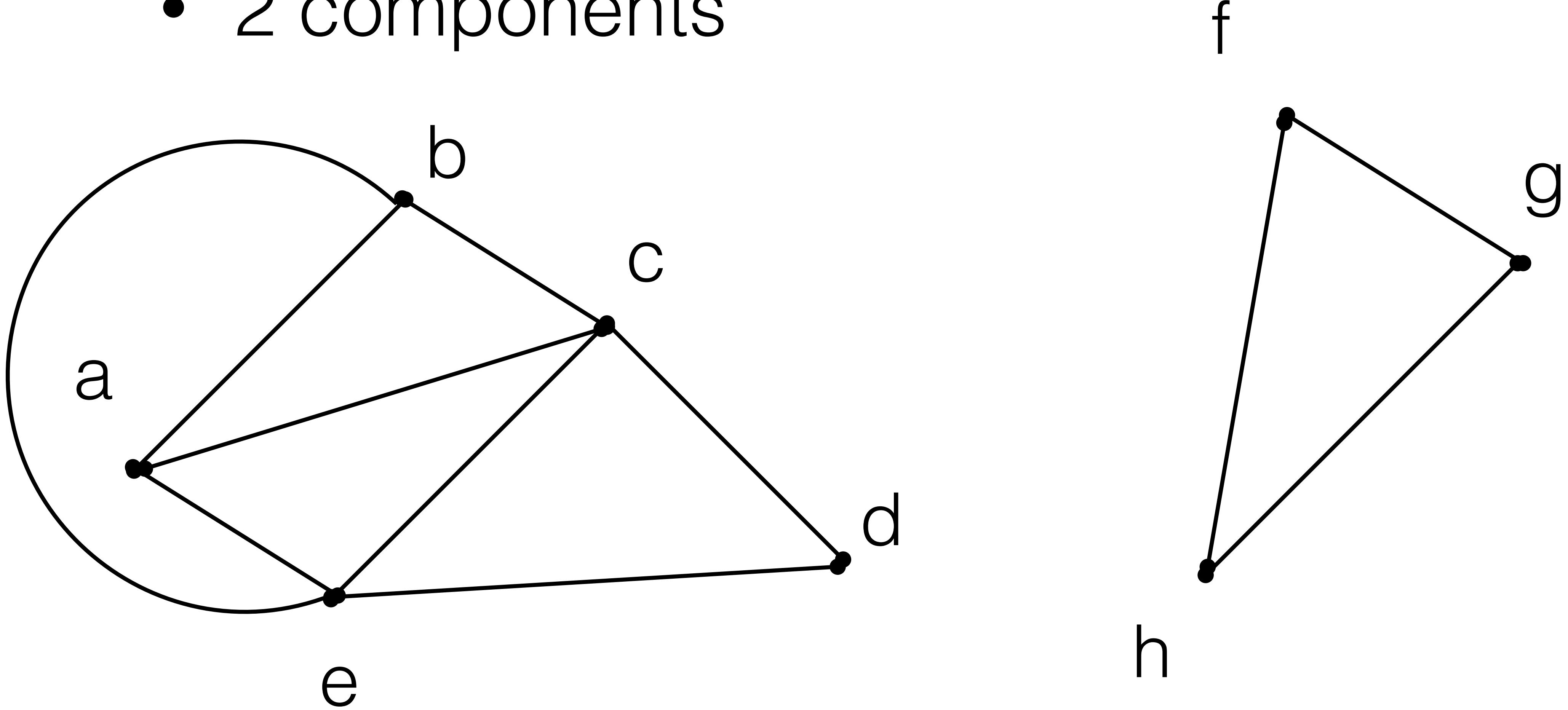
Just a representation of the graph!



Graph Representation

- 8 vertices
- 10 edges
- 2 components

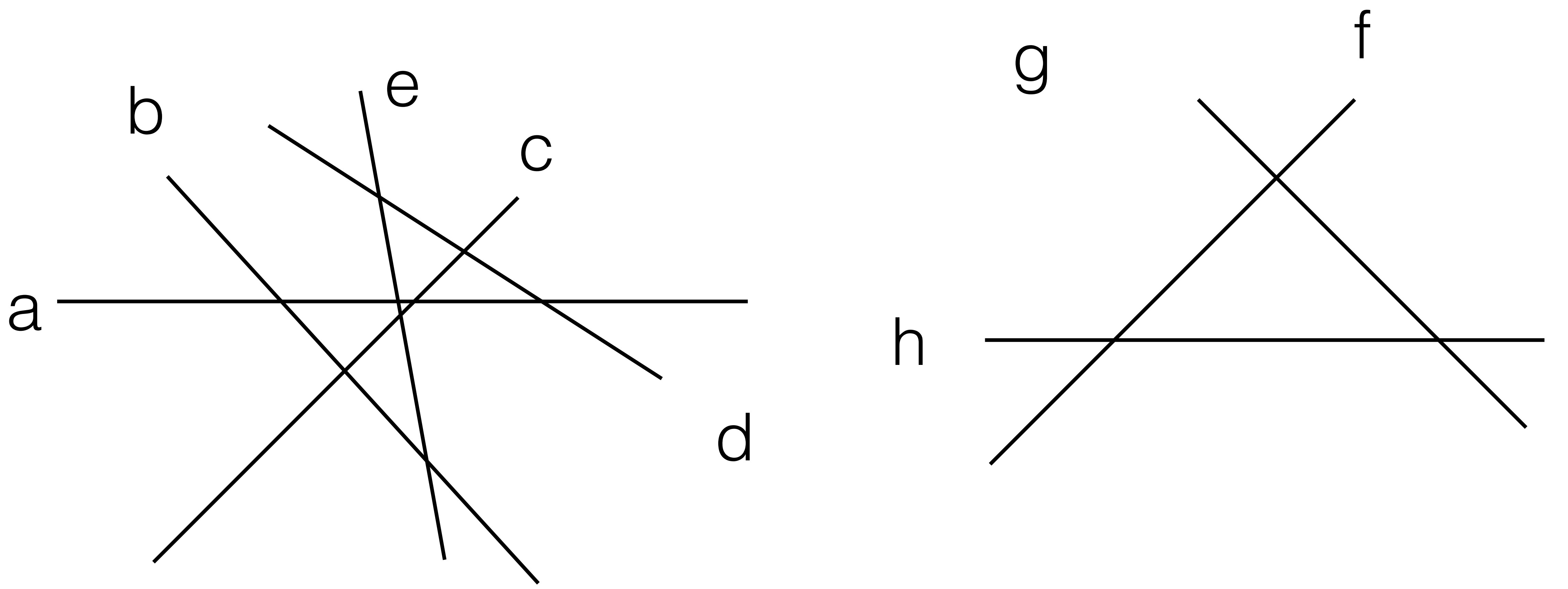
Planar Graph! independent of the representation



Graph Representation

- 8 vertices
- 10 edges
- 2 components

Another representation:
Intersection graph



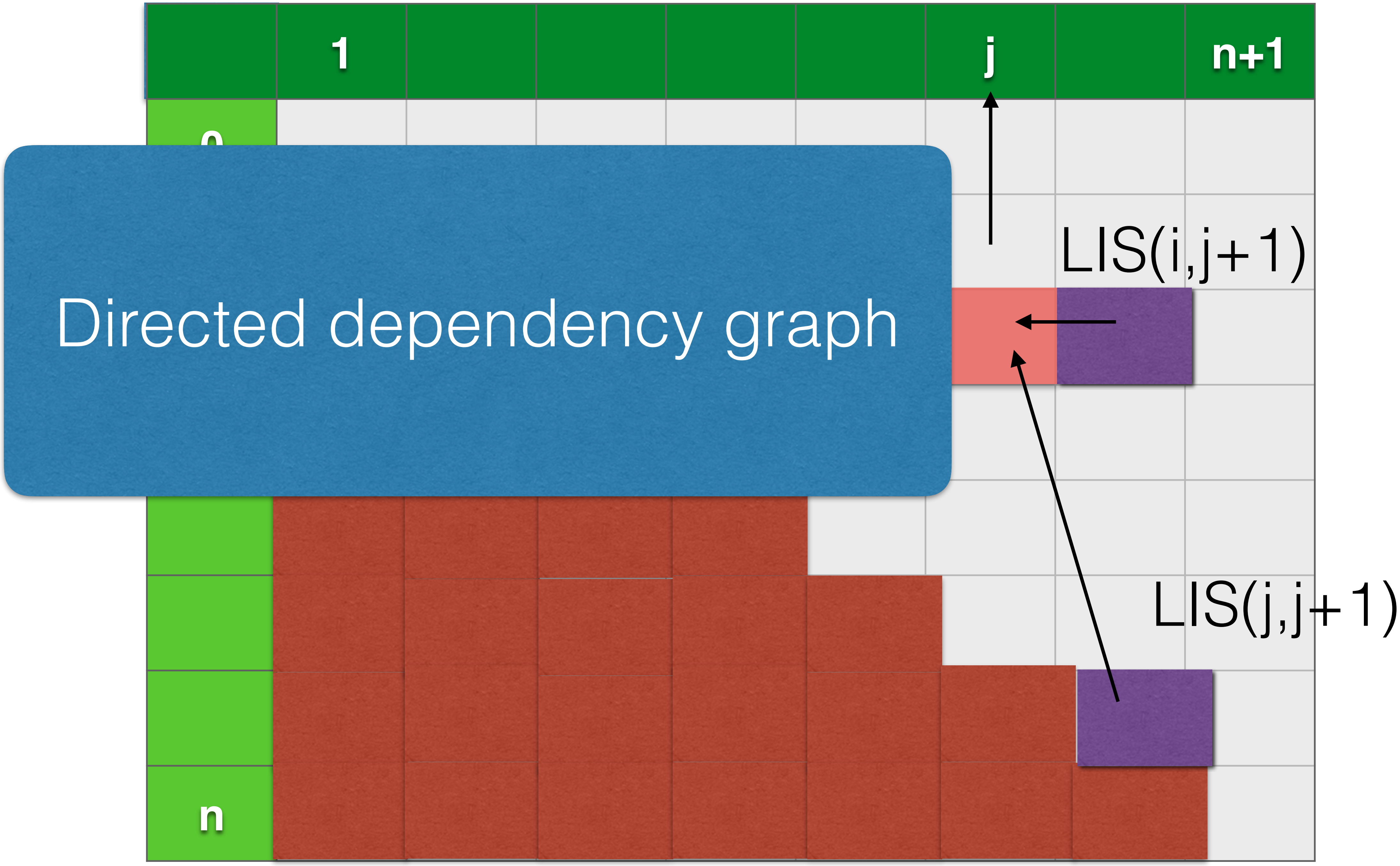
Graph Representation

- 8 vertices
- 10 edges
- 2 components

Another representation: Intervals

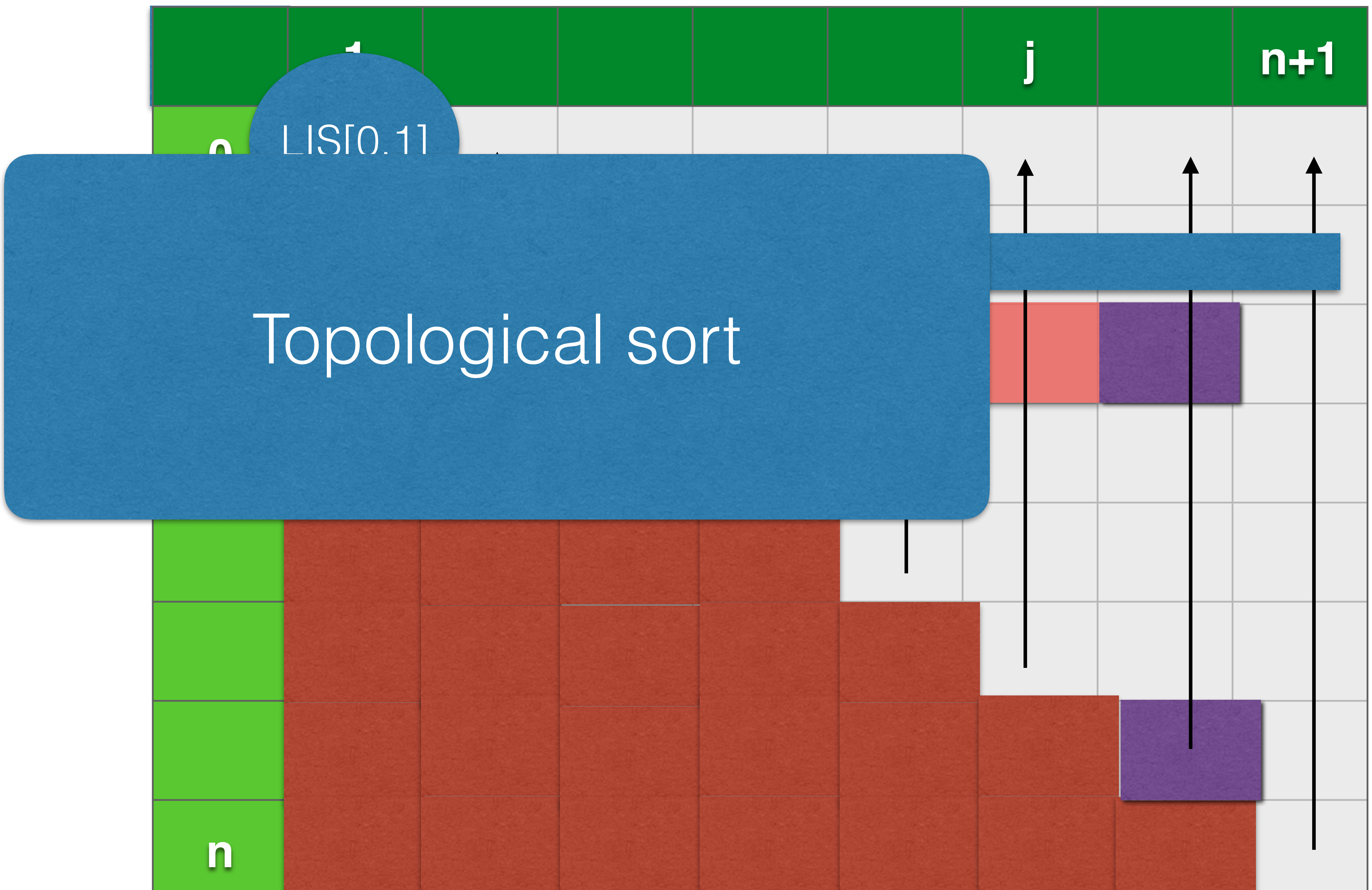
For i<j

$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j, j+1)\} & \text{otherwise} \end{cases}$$

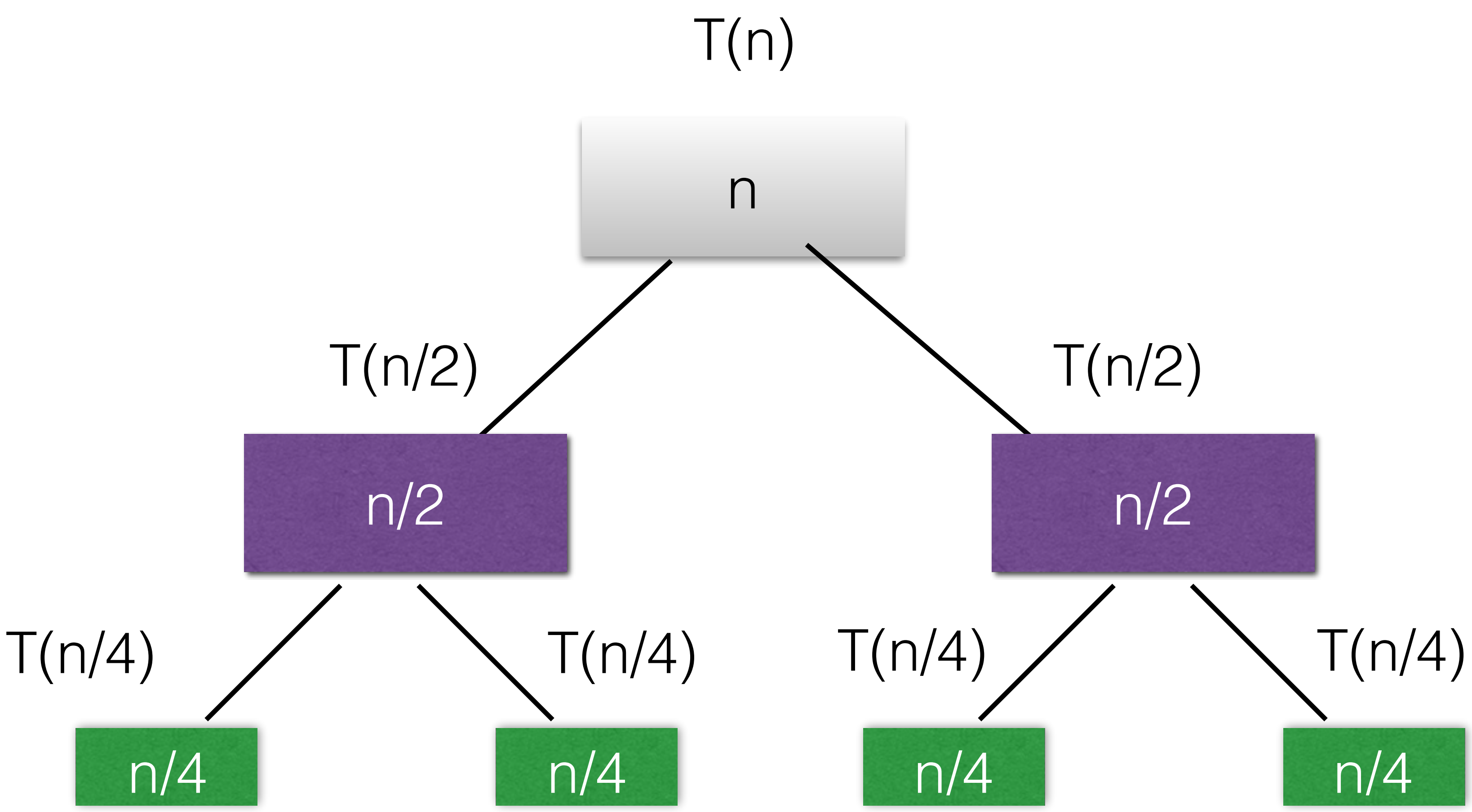


For i<j

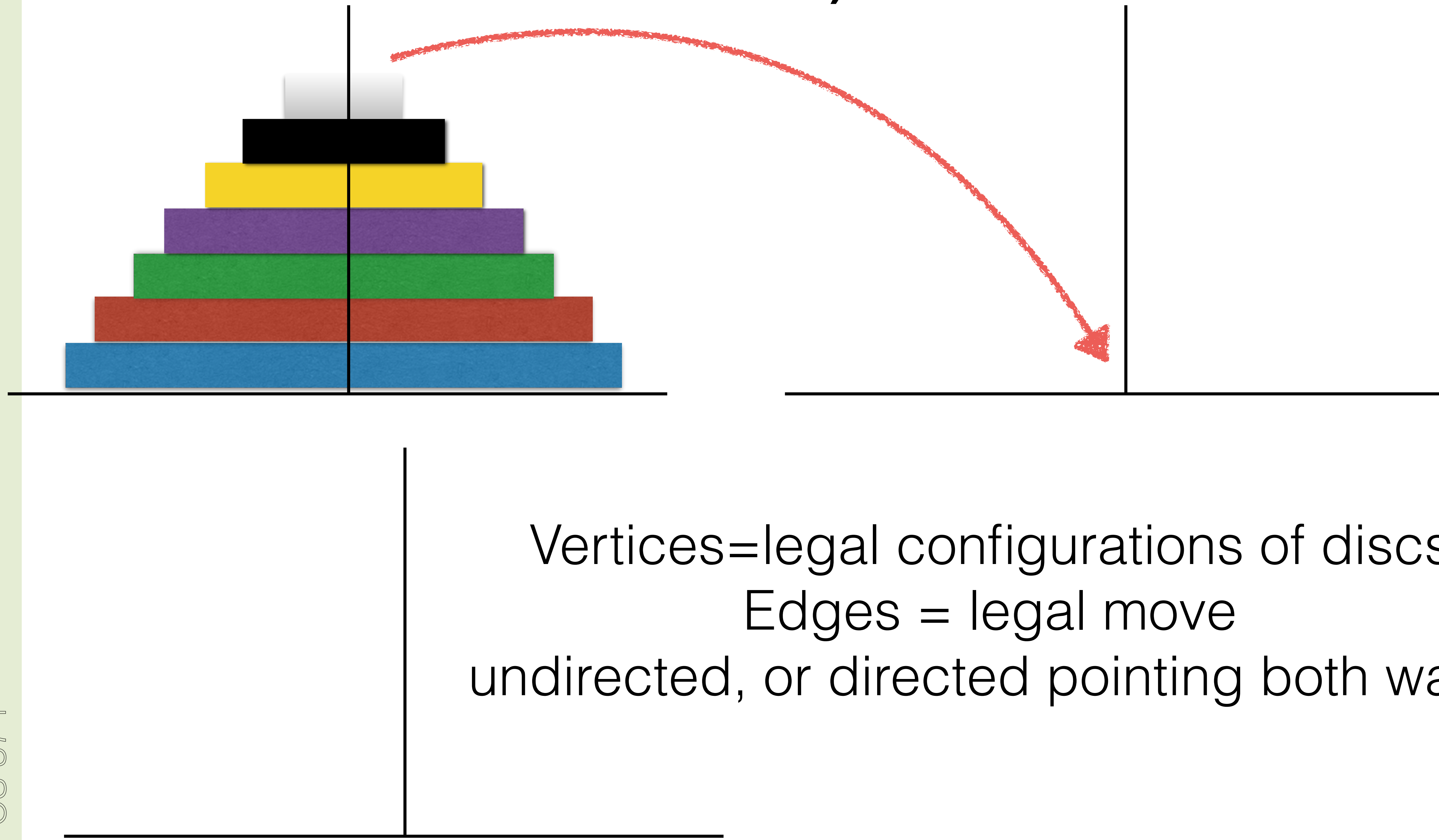
$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i,j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j,j+1)\} & \text{otherwise} \end{cases}$$



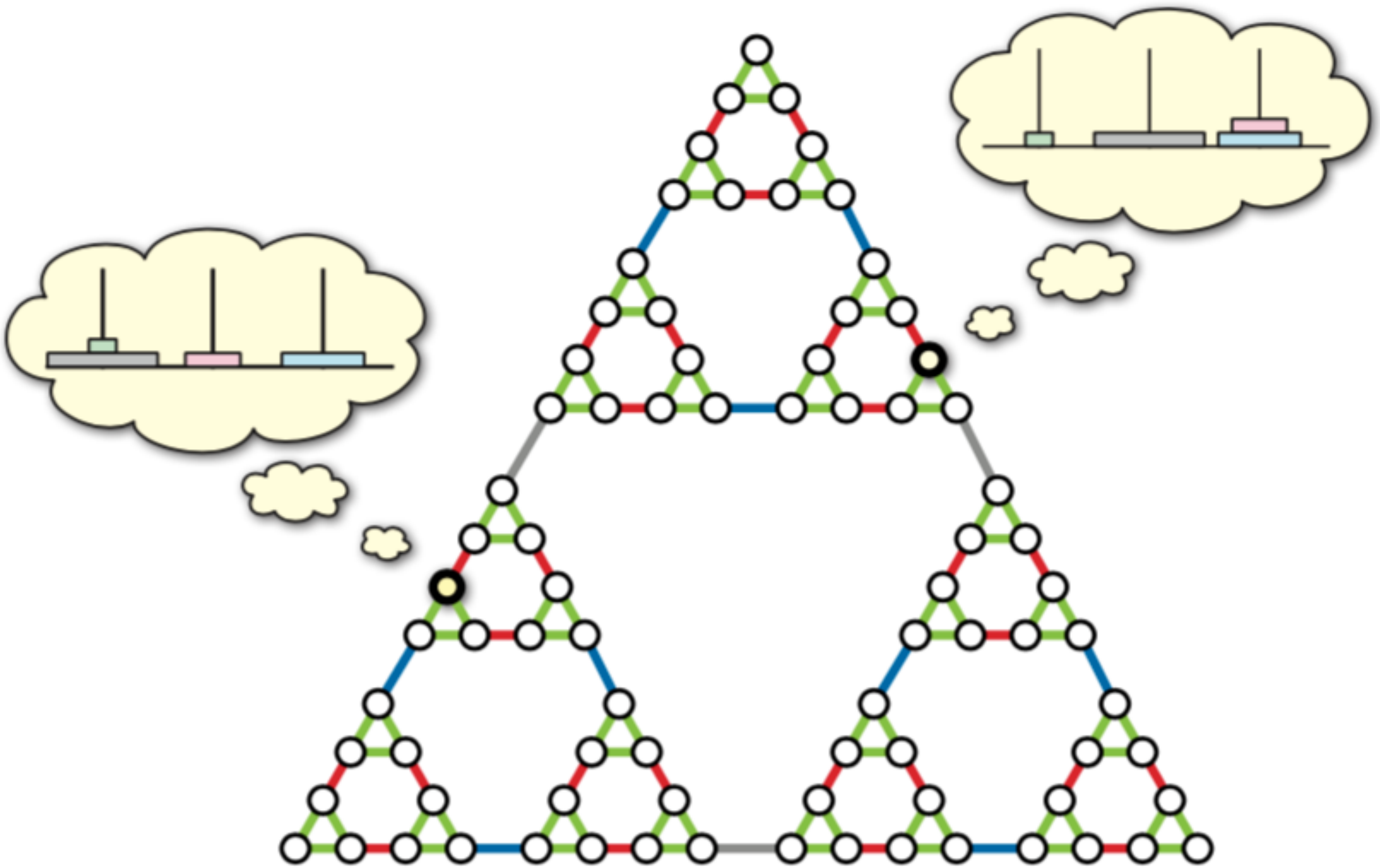
Recursion Tree (of Mergesort)



Configuration graph (Tower of Hanoi)

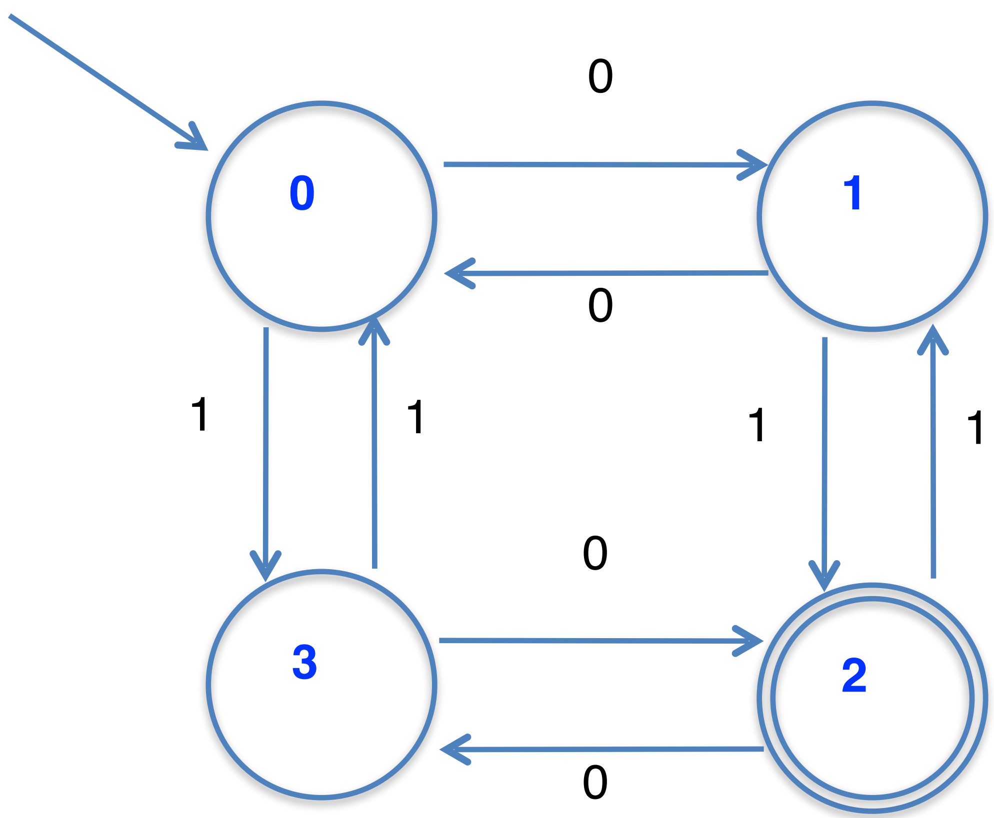


Vertices=legal configurations of discs
Edges = legal move
undirected, or directed pointing both ways



The configuration graph of the 4-disk Tower of Hanoi.

DFA as graph

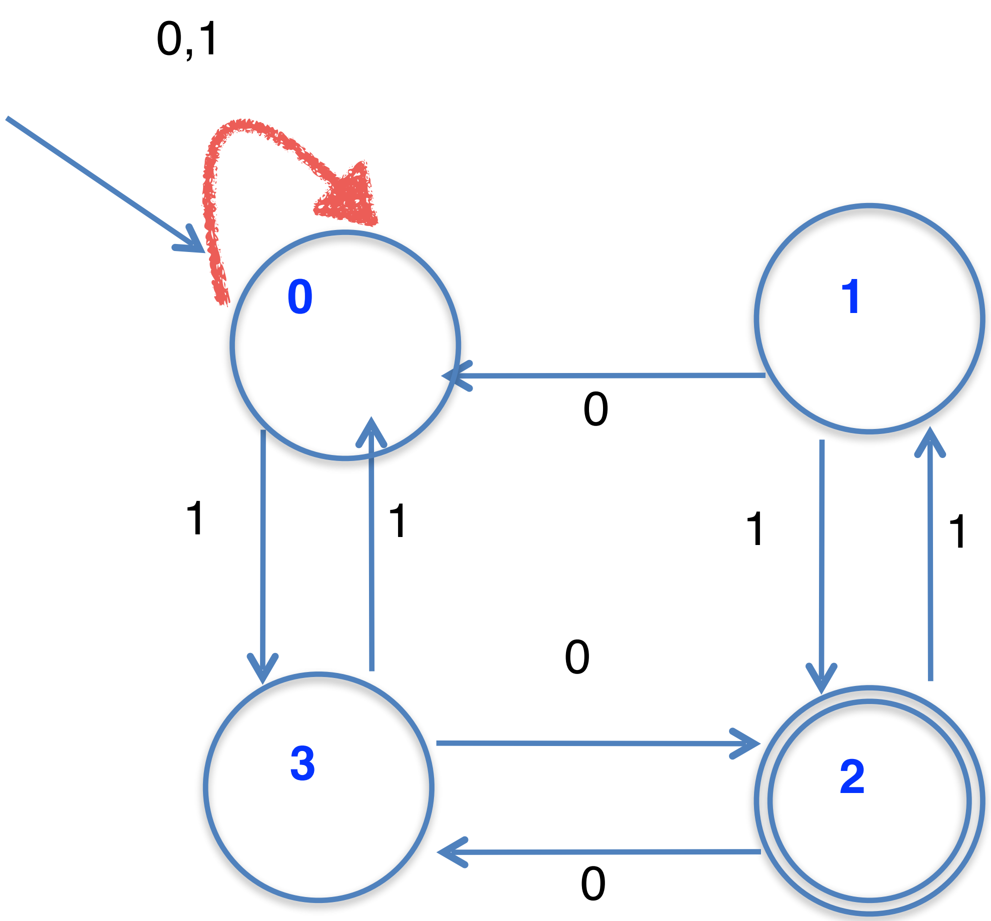


Labeled graph,
With conditions

q	$\delta[q, 0]$	$\delta[q, 1]$	$A[q]$
0	0	1	TRUE
1	2	3	FALSE
2	4	0	FALSE
3	1	2	FALSE
4	3	4	FALSE

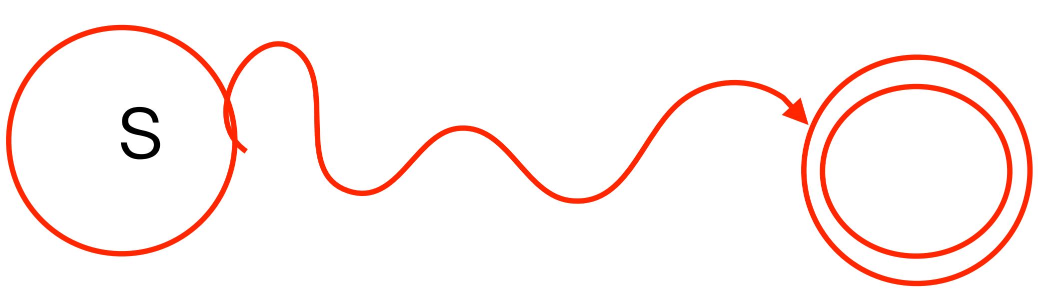
lookup table from transition function is data structure

NFA as graph

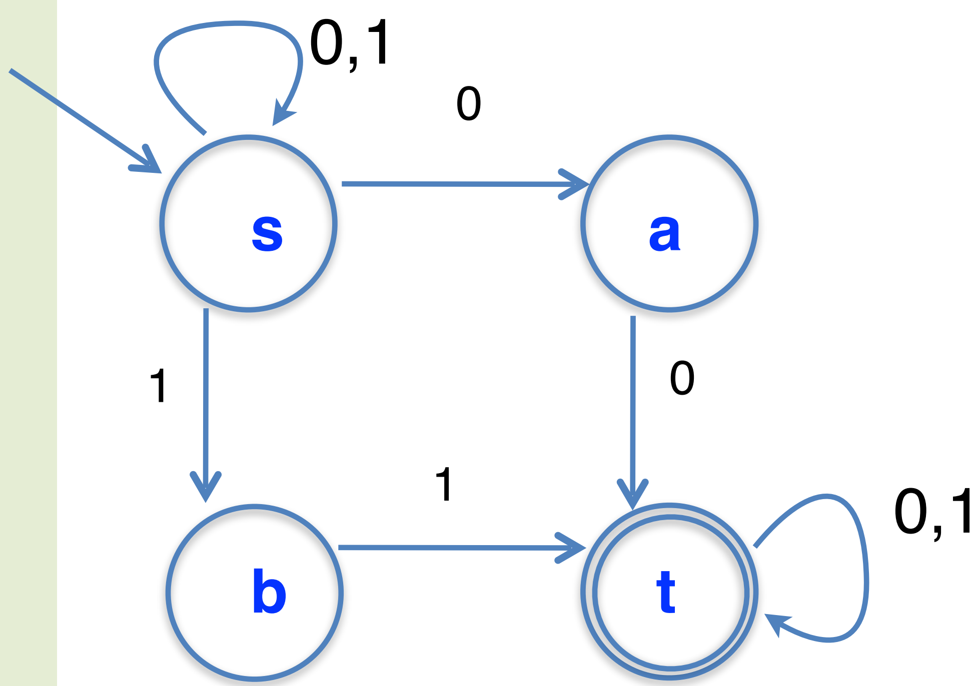


Labeled graph,
Without conditions

How to determine if NFA accepts anything?
Can s reach an accepting state?
DFS, reachability



NFA to DFA (subset construction)



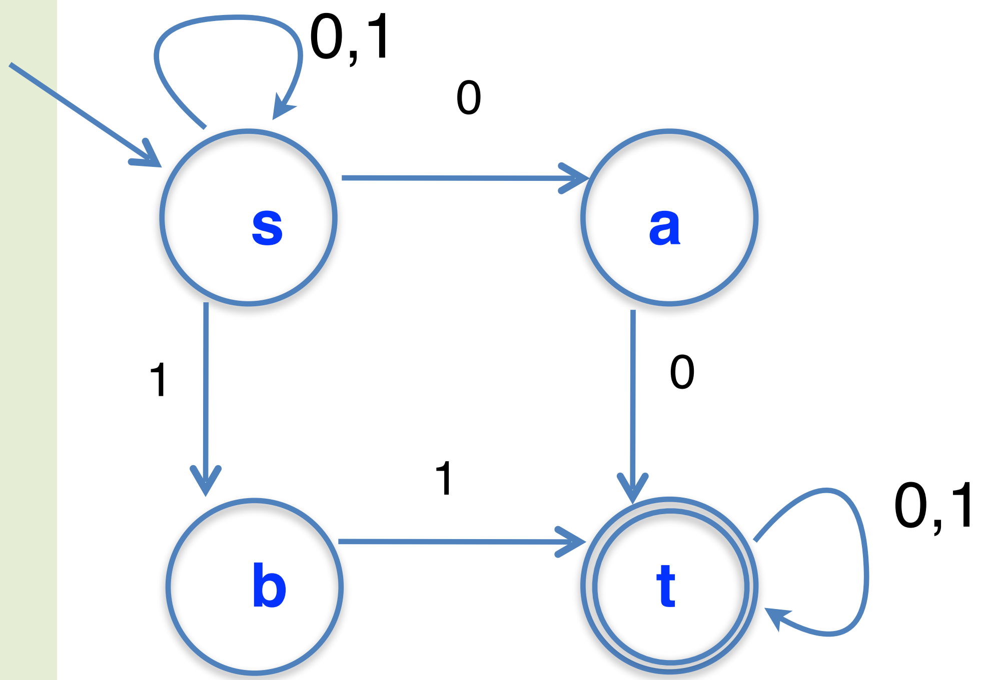
Some times the graph given is not the right graph!

V= 2^Q

E ={A → B | for all u in A there is

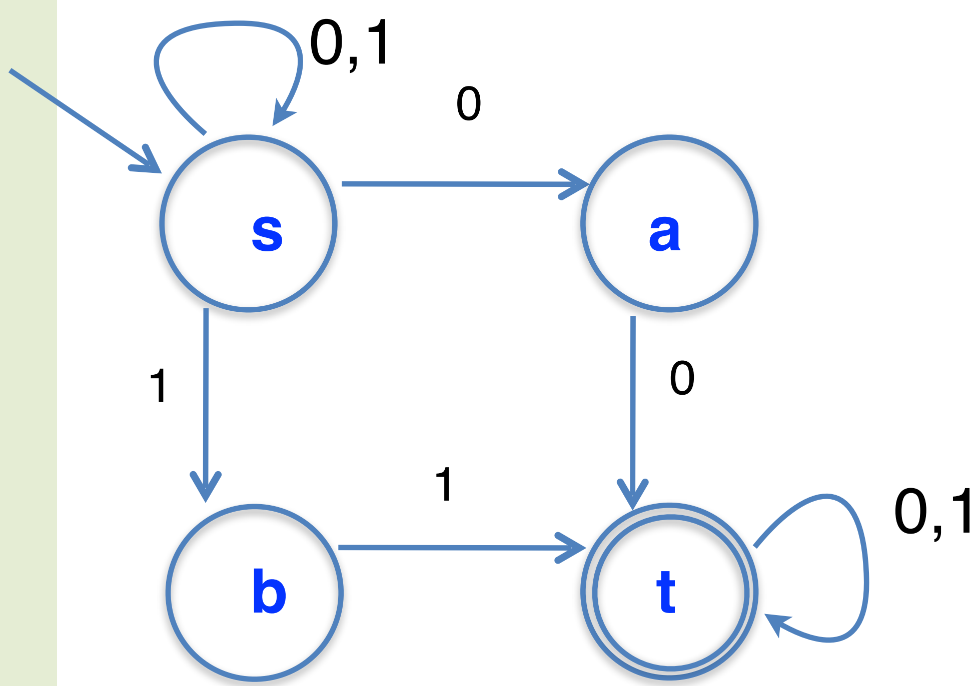
v in B such that u →v}

This is a 16 node graph!



P	ε	$\delta'(P,0)$	$\delta'(P,1)$	$q' \in A'$
s	s	as	bs	No
as	as	ats	bs	No
bs	bs	as	bts	No
ats	ats	ats	bts	Yes
bts	bts	ats	bts	Yes

NFA to DFA (subset construction)



Some times the graph given is not the right graph!

$V = 2^Q$

E ={A → B | for all u in A there is

v in B such that u →v}

This is a 16 node graph!

Incremental subset construction was running BFS in the DFA, though we were only explicitly given the NFA graph

Graph Boilerplate

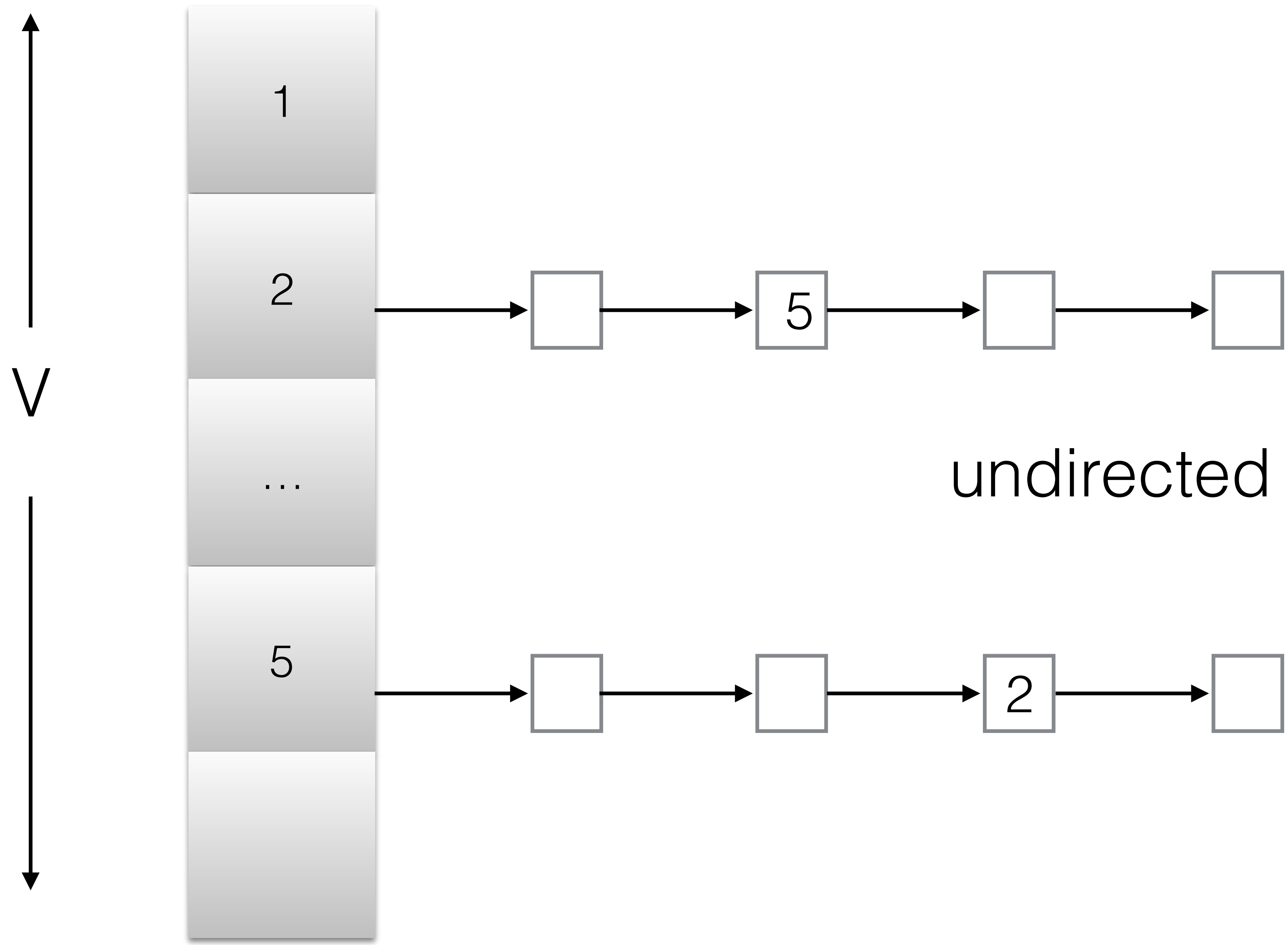
- When I design algorithm on a graph:
- V =?
- E=?
- Problem
- Algorithm
- Running time in terms of original input

Graph Algorithms

- "Given a graph G(V,E), do …"
- What does that mean?
- How to represent a graph ? (string is represented by array etc..)
- Two standard representations

Adjacency Lists

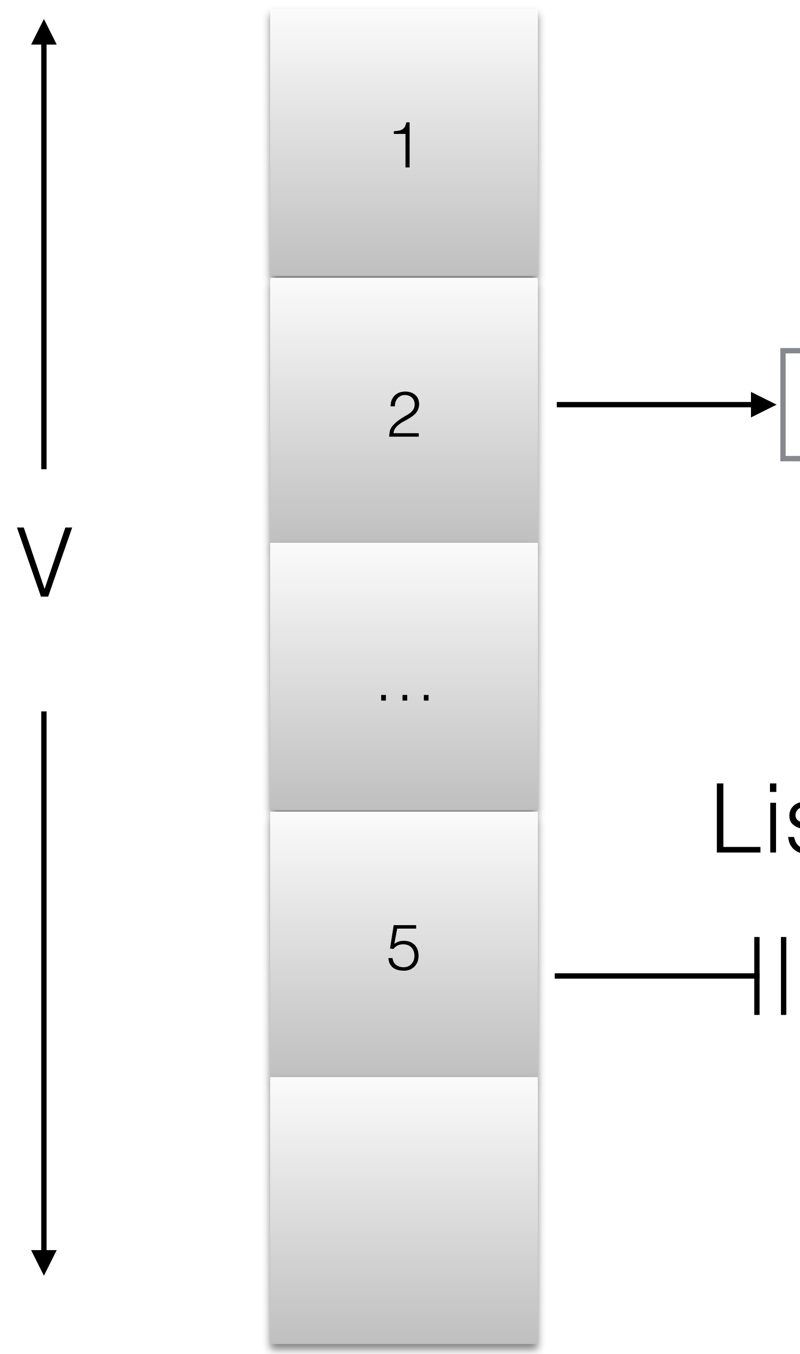
- Adjacency List (= Array of lists)

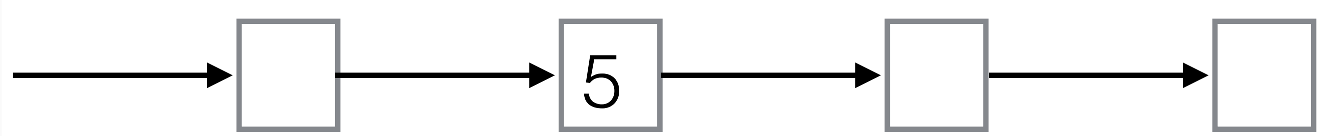


undirected graph

Adjacency Lists

- Adjacency List (= Array of lists)





O(|V|+|E|) space

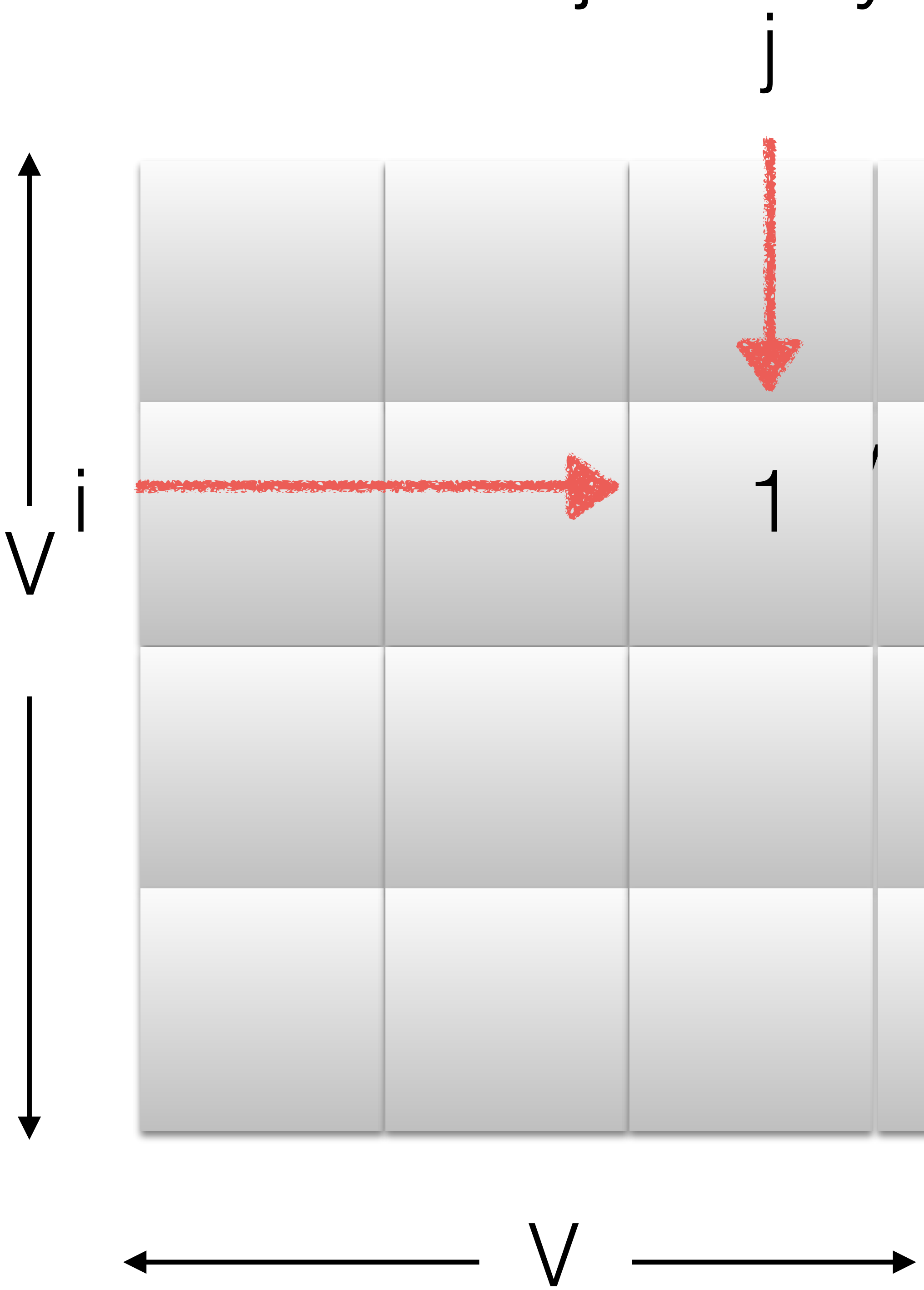
Why Adjacecny, lists?
Acces each node in O(1) time
List edges at each node in O(1) time each
Insert edge

Hard: is (u,v) in E?
More efficient data structure?
Why use linked lists?

Adjacency Matrix

- Adjacency matrix

Why use those at all?



A(i,j)=1 if (i,j) edge
0 otherwise

O(1) time to decide if (u,v) edge
Always O(n^2) space!

O(V) time to list all the neighbors of a vertex u even though there are only constant number of edges!

	Adjacency matrix	Standard adjacency list (linked lists)	Adjacency list (hash tables)
Space	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(V+E)$
Time to test if $uv \in E$	$O(1)$	$O(1+\min\{\deg(u), \deg(v)\}) = O(V)$	$O(1)$
Time to test if $u \to v \in E$	$O(1)$	$O(1+\deg(u)) = O(V)$	$O(1)$
Time to list the neighbors of v	$O(V)$	$O(1+\deg(v))$	$O(1+\deg(v))$
Time to list all edges	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(V+E)$
Time to add edge uv	$O(1)$	$O(1)$	$O(1)^*$
Time to delete edge uv	$O(1)$	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$

How to traverse a graph?

- Traversal in general: e.g. you have a data structure with pointers and you want to print it out once

- How to traverse a graph?

```
RecursiveDFS(v):
    if v is unmarked
        mark v
        for each edge vw
            RecursiveDFS(w)
```

How to traverse a graph?

```
RecursiveDFS(v):
  if v is unmarked
    mark v
    for each edge vw
      RecursiveDFS(w)
```

O(|V|+|E|) time

```
IterativeDFS(s):
  Push(s)
  while the stack is not empty
    v ← Pop
    if v is unmarked
      mark v
      for each edge vw
        Push(w)
```

How to traverse a graph?

```
TRAVERSE(s):
    put s into the bag
    while the bag is not empty
        take v from the bag
        if v is unmarked
            mark v
            for each edge vw
                put w into the bag
```

stack =LIFO (DFS)
Queue = FIFO (BFS)
Priority Queue = lightest out
Random, etc

Whatever First Search

TRAVERSE(s):
  put $(\emptyset, s)$ in bag
  while the bag is not empty
    take (p, v) from the bag $(\star)$
    if v is unmarked
      mark v
      $parent(v) \leftarrow p$
      for each edge vw $(\dagger)$
        put (v, w) into the bag $(\star\star)$

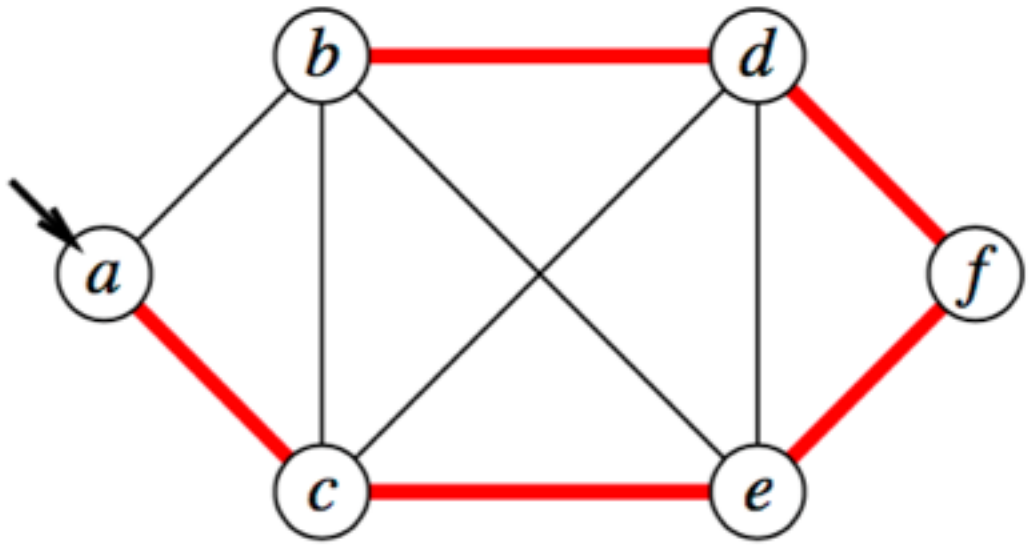
stack =LIFO (DFS)
Queue = FIFO (BFS)
Priority Queue = lightest out
Random, etc

Whatever First Search

Traverse(s) marks every vertex in a connected graph exactly once, and the set of pairs (v, parent(v)) with parent(v) not empty, defines a spanning tree of the graph

BFS tree has shortest paths from s!

DFS tree



BFS tree