Graph

- Introduction
- Graph API
- Elementary Graph Operations
  - **DFS: Depth first search**
  - BFS: Breadth first search
  - CC: Connected Components

Major references:

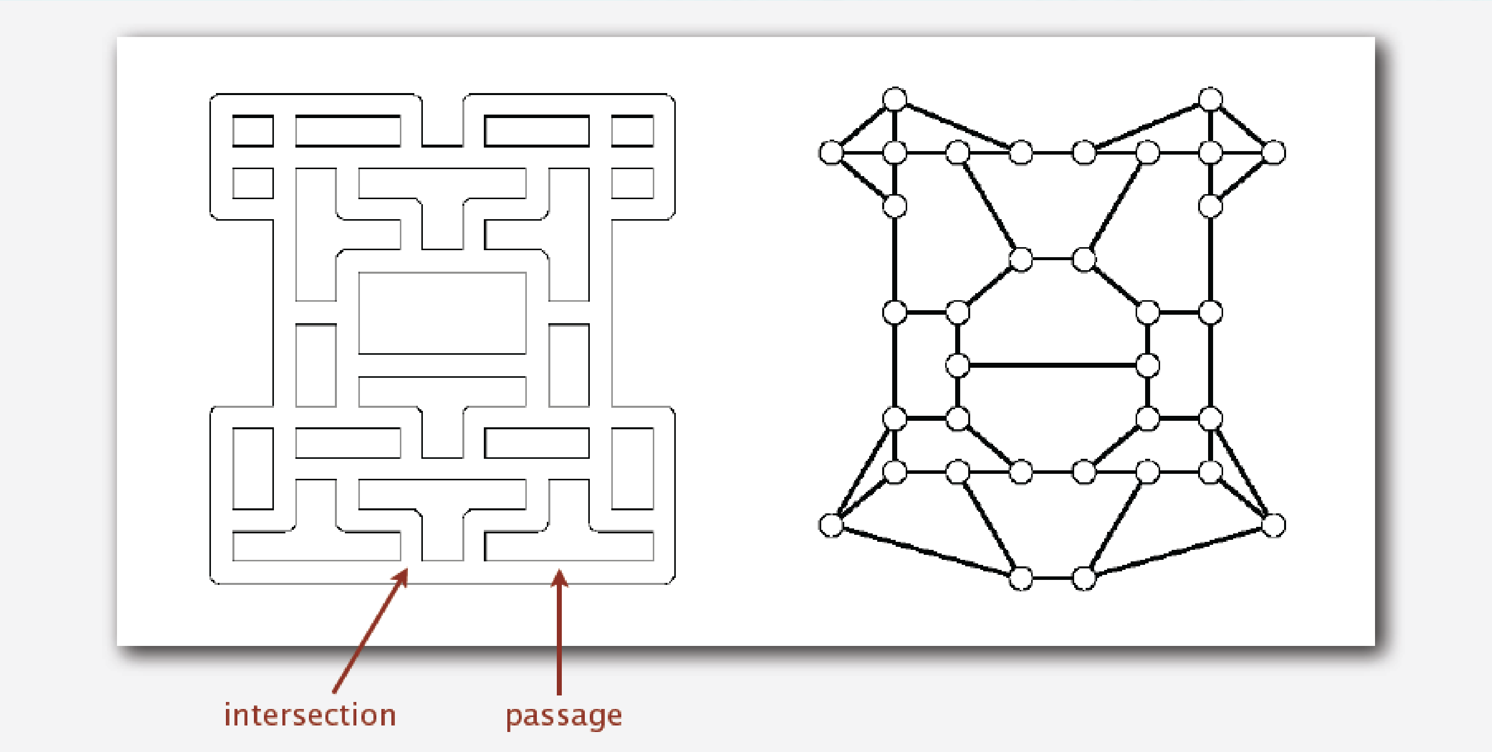
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

DFS: Depth-First Search

Algorithm:

- Vertex = intersection
- Edge = passage

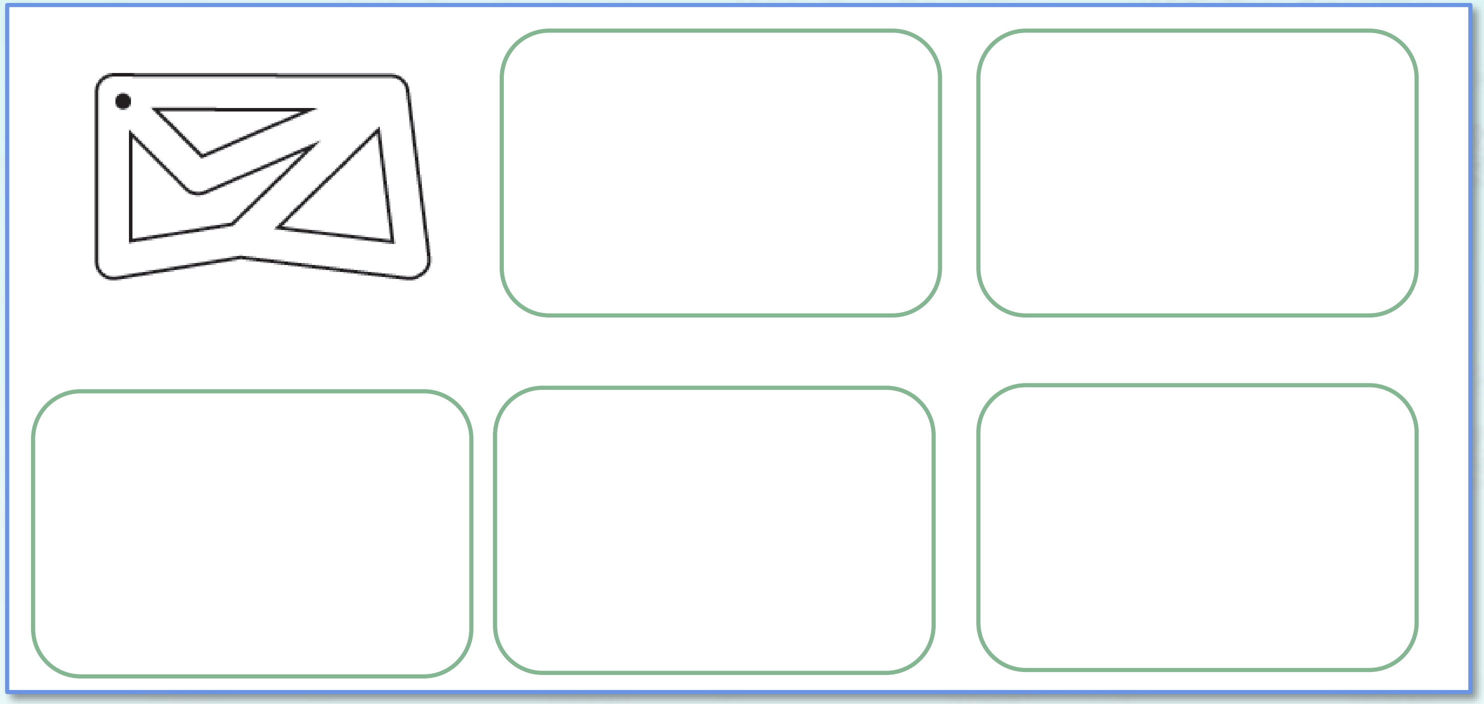
pacman



Maze Goal: Explore every intersection in the maze.

DFS: Depth-First Search

Maze graph:
- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options



Maze Goal: Explore every intersection in the maze.
Good Visualization: https://www.cs.usfca.edu/~galles/visualization/DFS.html

DFS: Depth-First Search

Maze graph:

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options

Theseus, a hero of Greek mythology, is best known for slaying a monster called the Minotaur. When Theseus entered the Labyrinth where the Minotaur lived, he took a ball of yarn to unwind and mark his route. Once he found the Minotaur and killed it, Theseus used the string to find his way out of the maze.

Read more: http://www.mythencyclopedia.com/Sp-Tl/Theseus.html#ixzz30wFO3ofe

Maze Goal: Explore every intersection in the maze.

DFS: Depth-First Search

Maze graph:

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options

Shannon and his famous electromechanical mouse *Theseus* (named after Theseus from Greek mythology) which he tried to have solve the maze in one of the first experiments in artificial intelligence.

The Las Vegas connection: Shannon and his wife Betty also used to go on weekends to Las Vegas with MIT mathematician Ed Thorp, and made very successful forays in blackjack using game theory.

Maze Goal: Explore every intersection in the maze.

DFS: Depth-First Search

Design pattern: Decouple graph data type
Idea: Mimic maze exploration

DFS (to visit a vertex v)

- **Mark v as visited.**
- **Recursively visit all unmarked vertices w adjacent to v.**

Typical applications:

- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

Challenge:

- How to implement?

DFS: Depth-First Search

Goal: Systematically search through a graph from graph processing

- **Create a graph object**
- **Pass the graph to a graph processing routine**
- **Query the graph-processing routine**

DFS: Depth-First Search

Goal: Systematically search through a graph from graph processing

- **Create a graph object**
- **Pass the graph to a graph processing routine**
- **Query the graph-processing routine**

public class Paths		
	`Paths(Graph G, int s)`	*find paths in G from source s*
`boolean`	`hasPathTo(int v)`	*is there a path from s to v?*
`Iterable<Integer>`	`pathTo(int v)`	*path from s to v; null if no such path*

```
Paths paths = new Paths(G, s);
for (int v = 0; v < G.V(); v++)
    if (paths.hasPathTo(v))
        StdOut.println(v);
```

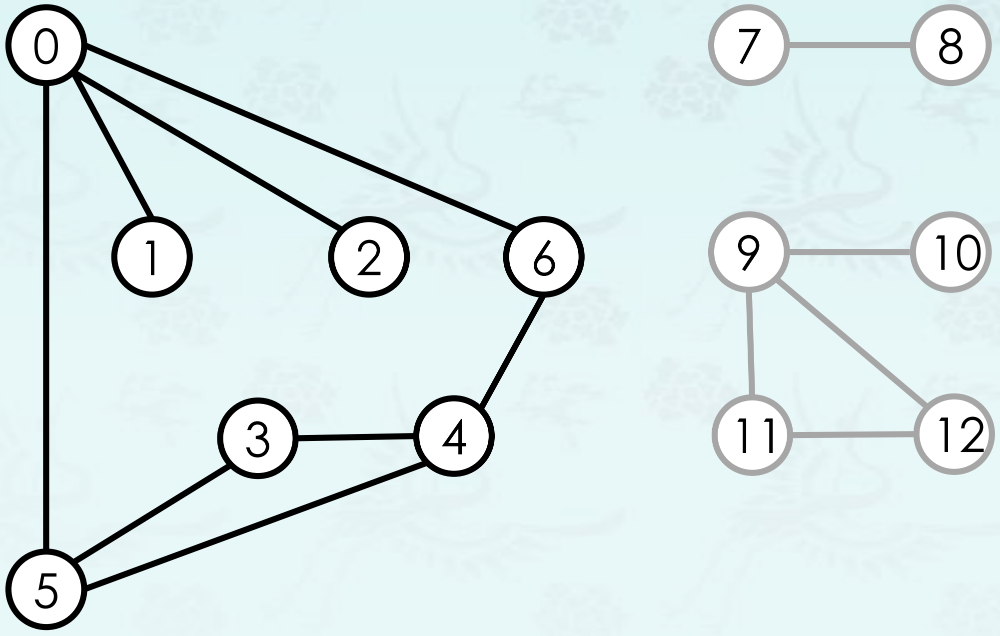
print all vertices connected to s

Graph – Coding

For each edge(v, w) in the list

- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeFromTo(g, v, w);   // add an edge from v to w.

엣지리스트로만은 힘들어서 인접행렬로 바꿔야뎀



V-E lists →

```
graph3.txt
13  ← V
13  ← E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

Graph g:

Challenge: build adjacency lists?

Graph Coding – graph.cpp

```
struct Graph {
  int V;              // number of vertices in the graph
  int E;              // number of edges in the graph
  gnode adj;          // an array of adjacency lists (or gnode pointers)

  Graph(int v = 0) { // constructs a graph with v vertices
    V = v;
    E = 0;

    // initialize each adjacency list as an empty list;
    for (int  i = 0; i < V; i++) {
        g->adj[i].next = nullptr;
        g->adj[i].item = i;
    }
  }
  ~Graph() {}
};
using graph = Graph *;
```

set each adj list nullptr

unused; but may store the size of degree.

Graph Coding

```
// add an edge to an undirected graph

void addEdgeFromTo(graph g, int v, int w) {
  // add an edge from v to w.
  // A new vertex is added to the adjacency list of v.
  // The vertex is added at the beginning

  gnode node = new Gnode(w, g->adj[v].next);
  g->E++;
}

// add an edge to an undirected graph
void addEdge(graph g, int v, int w) {
  addEdgeFromTo(g, v, w); // add an edge from v to w.
  addEdgeFromTo(g, w, v); // if graph is undirected, add both
}
```

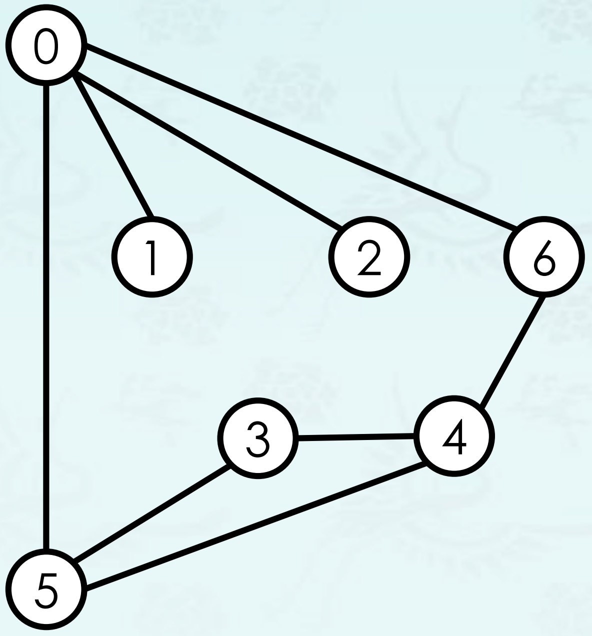
instantiate a node w insert it at **the front of** adjacency list[v]

add an edge for undirected graph

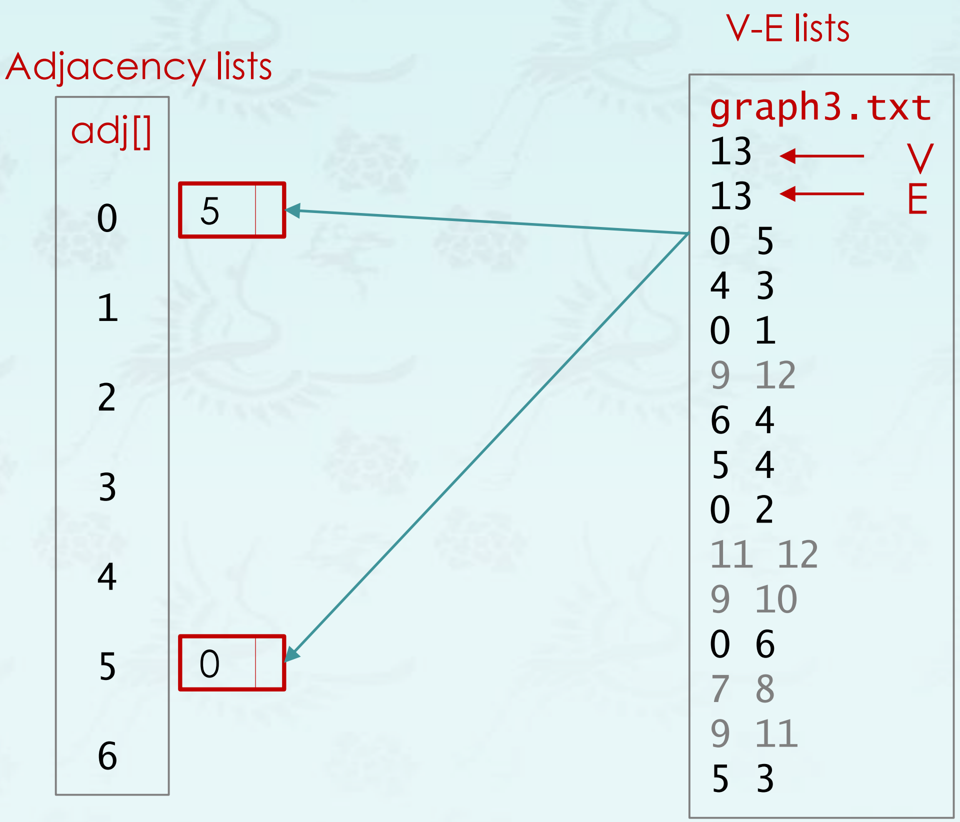
Graph – Build Adjacency list

For each edge(v, w) in the list

- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeFromTo(g, v, w);   // add an edge from v to w.



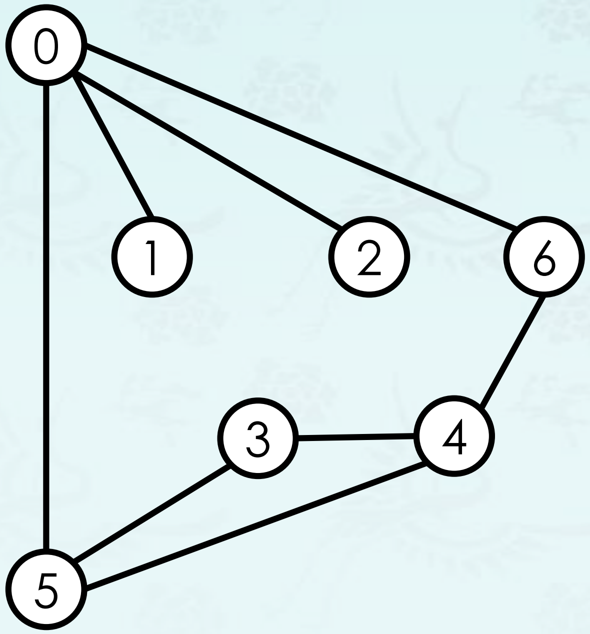
Graph g



Graph – Build Adjacency list

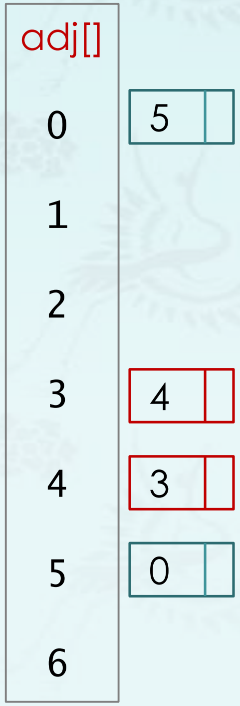
For each edge(v, w) in the list

- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeFromTo(g, v, w);   // add an edge from v to w.



Graph g

Adjacency lists



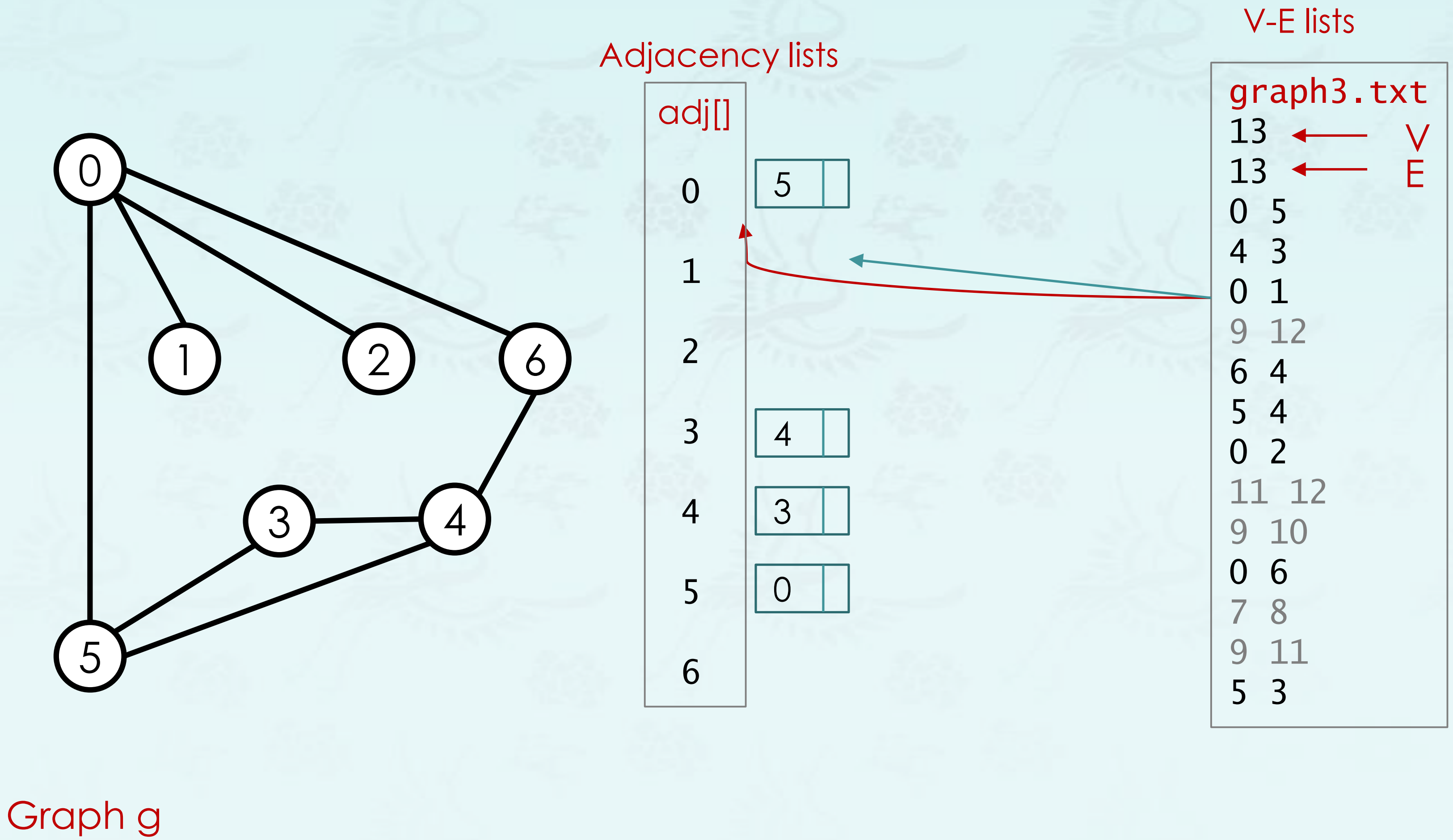
V-E lists

```
graph3.txt
13   ← V
13   ← E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

Graph – Build Adjacency list

For each edge(v, w) in the list

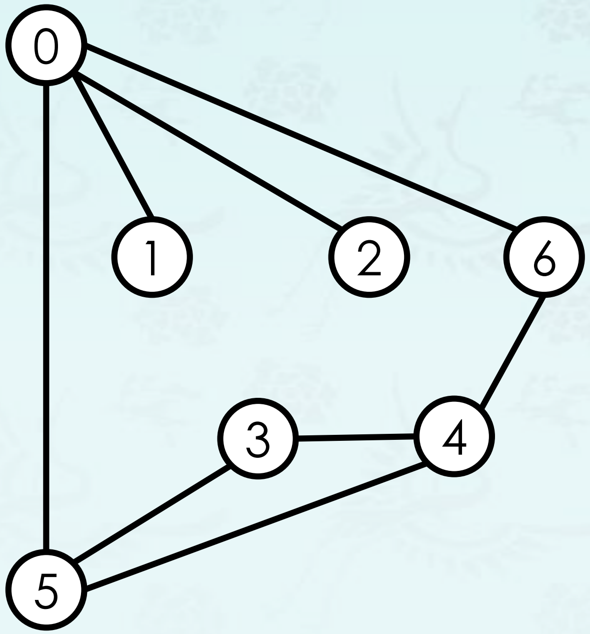
- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeFromTo(g, v, w);   // add an edge from v to w.



Graph – Build Adjacency list

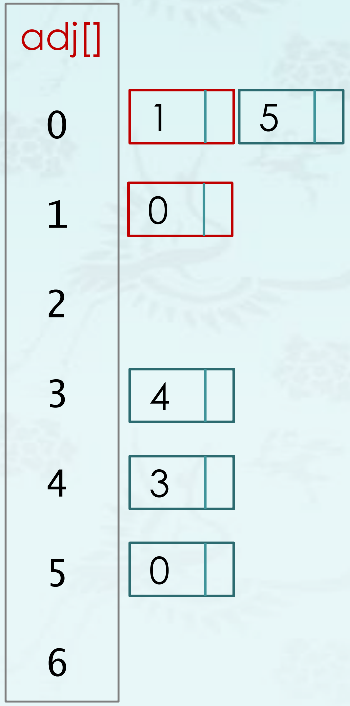
For each edge(v, w) in the list

- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeFromTo(g, v, w); // add an edge from v to w.



Graph g

Adjacency lists



V-E lists

```
graph3.txt
13   <-- V
13   <-- E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

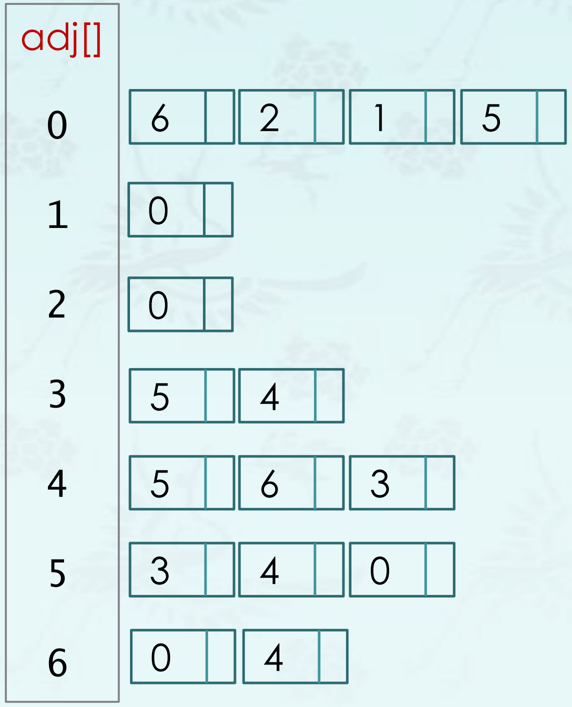
Graph – Build Adjacency list

For each edge(v, w) in the list

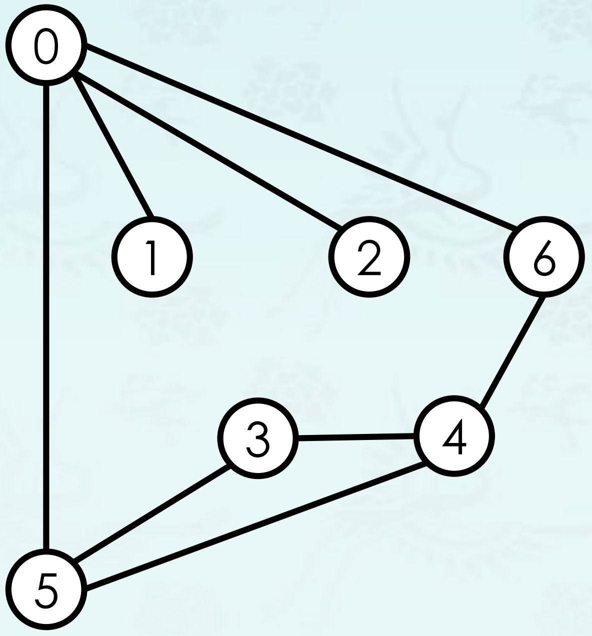
- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeFromTo(g, v, w);   // add an edge from v to w.

V-E lists

Adjacency lists




```
graph3.txt
13  ← V
13  ← E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```



Graph g

DFS: Depth-First Search Demo

To visit a vertex v:

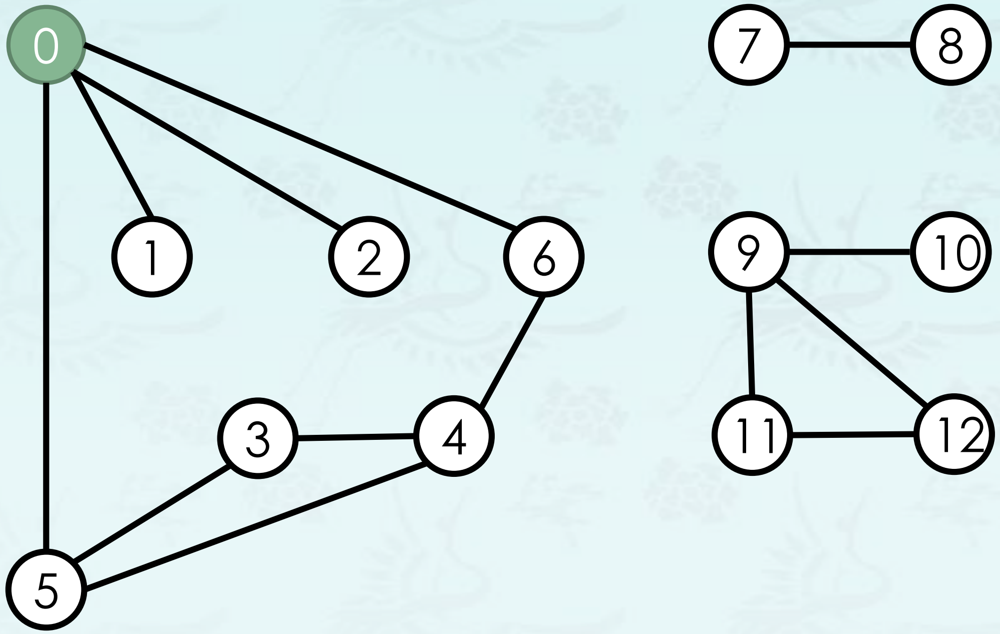
- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

DFS: Depth-First Search Demo

To visit a vertex v:

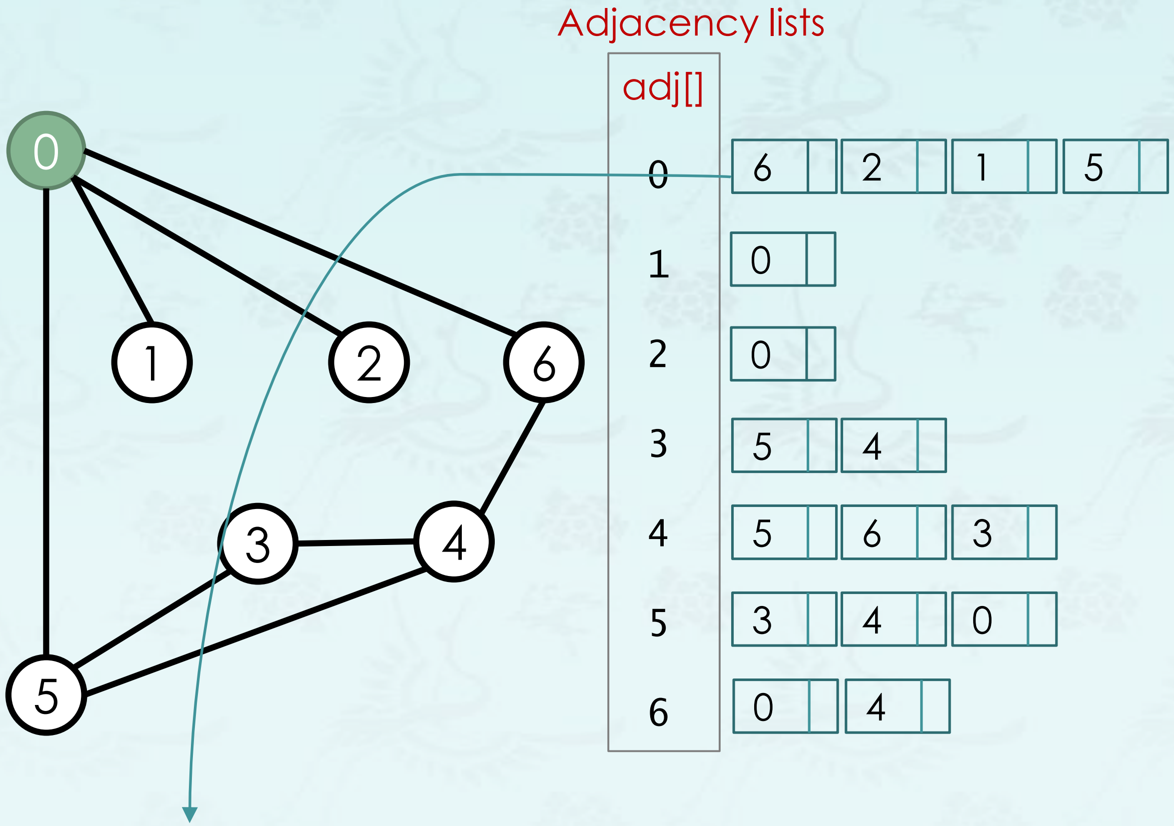
DFS BFS 이거 시험에 안나올수가 없어 디에프에스만해도 시험문제 2개

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

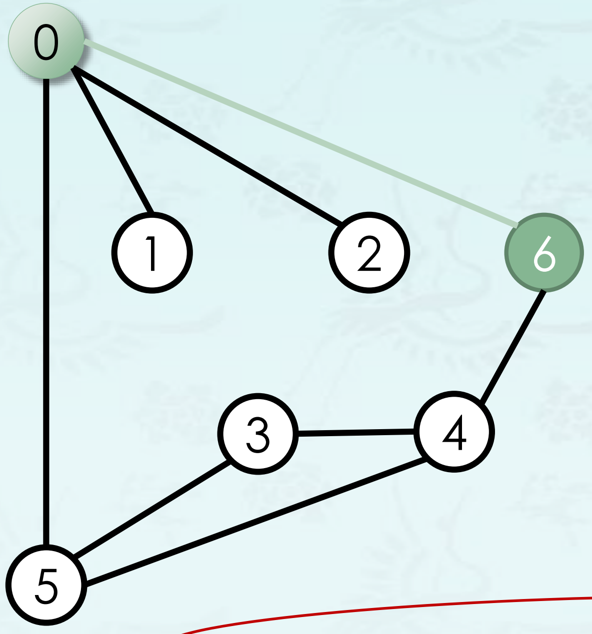
visit 0: **Which one first?**



v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: **check 6**, **check 2, check 1, and check 5**

DFS 0



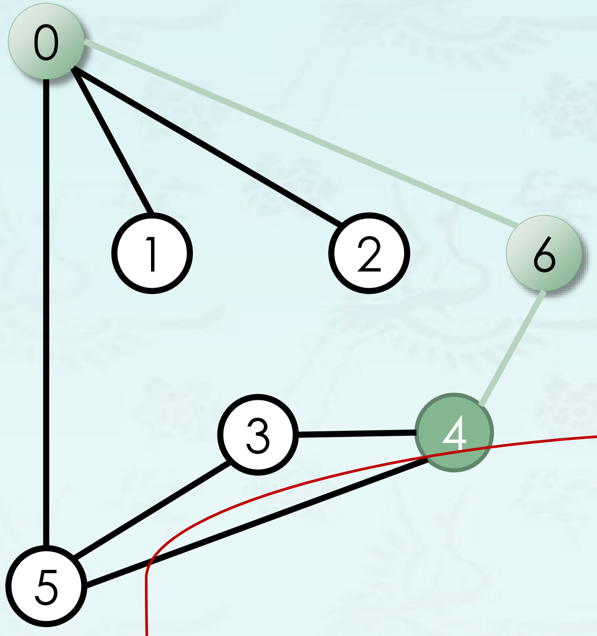
Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	~~0~~	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 6: **~~check 0~~**, **check 4**

DFS 0 6

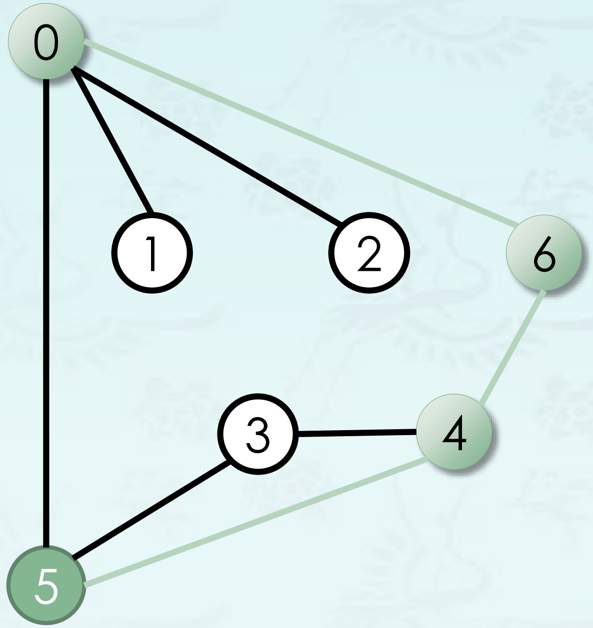


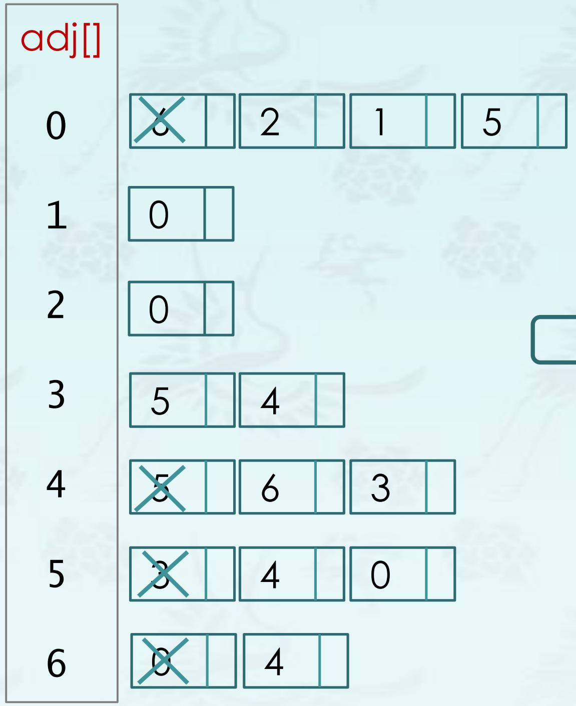
adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	~~0~~	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	F	–
4	T	6
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 4: **check 5**, **check 6**, **check 3**

DFS 0 6 4

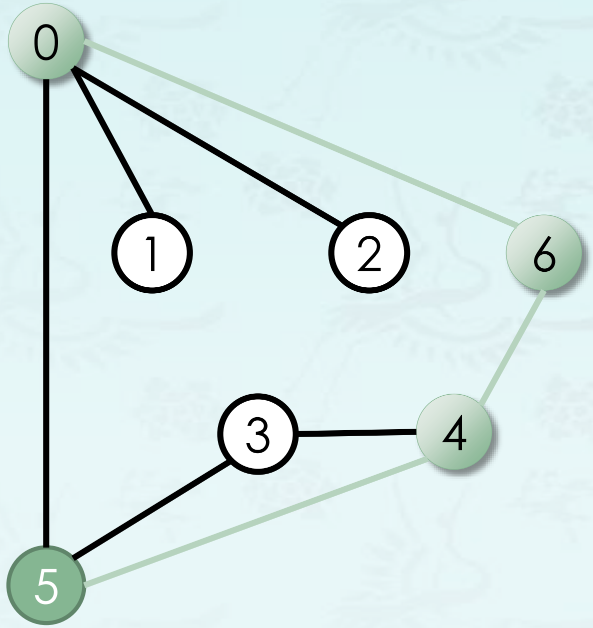




v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	F	–
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 5: **check 3**, **check 4**, **check 0**

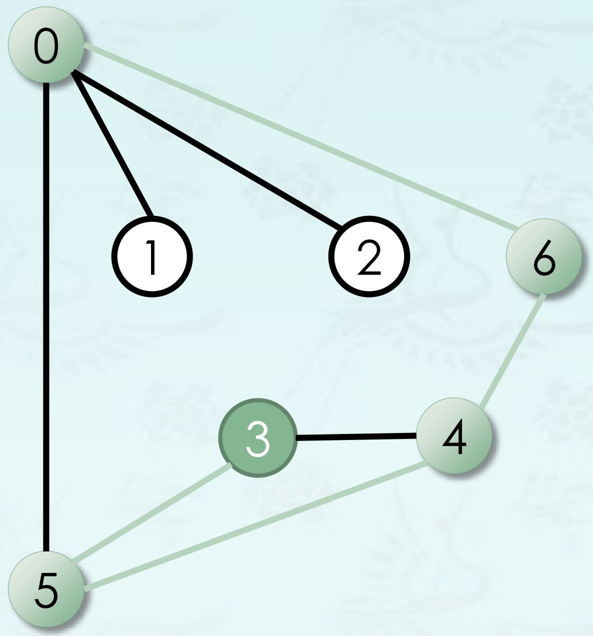
DFS 0 6 4 5



adj[]				
0	6	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	0	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	F	–
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 5: **check 3**, **check 4**, **check 0**

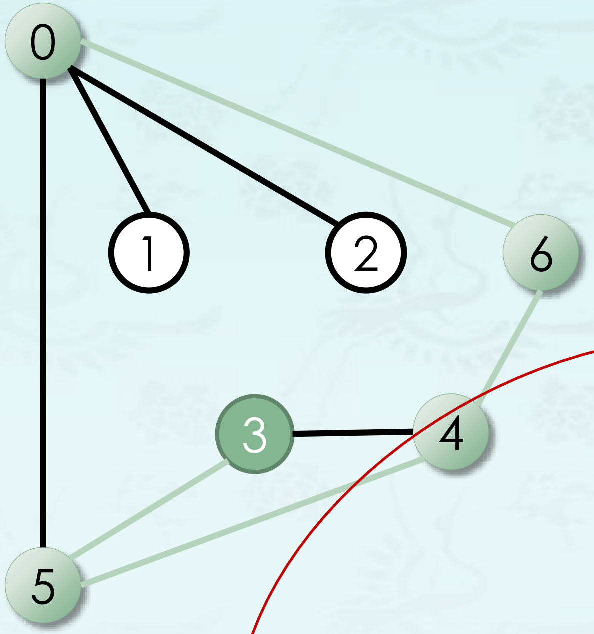


adj[]				
0	6	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	0	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 5: **check 3**, **check 4**, **check 0**

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University



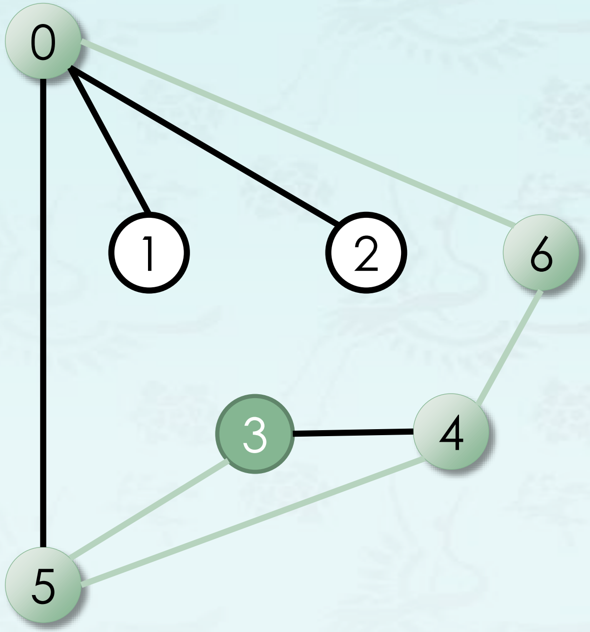
Adjacency lists

adj[]				
0	6	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	0	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 3: **check 5**, **check 4**

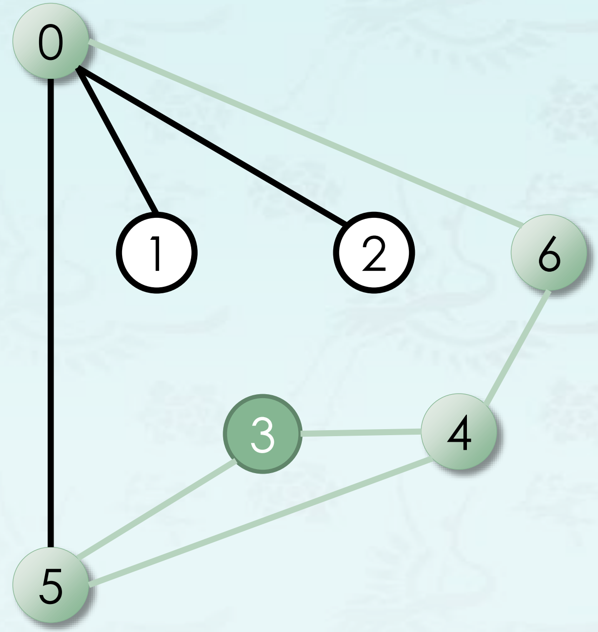
DFS 0 6 4 5 3



adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	4		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

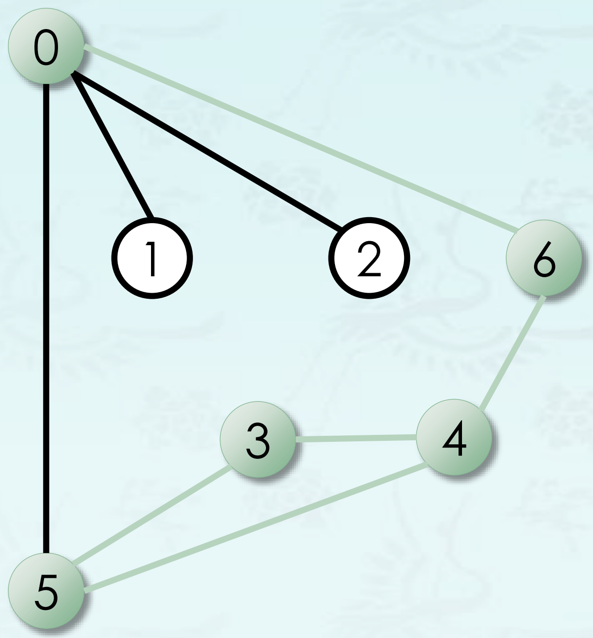
visit 3: ~~**check 5**~~, **check 4**



adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	4		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

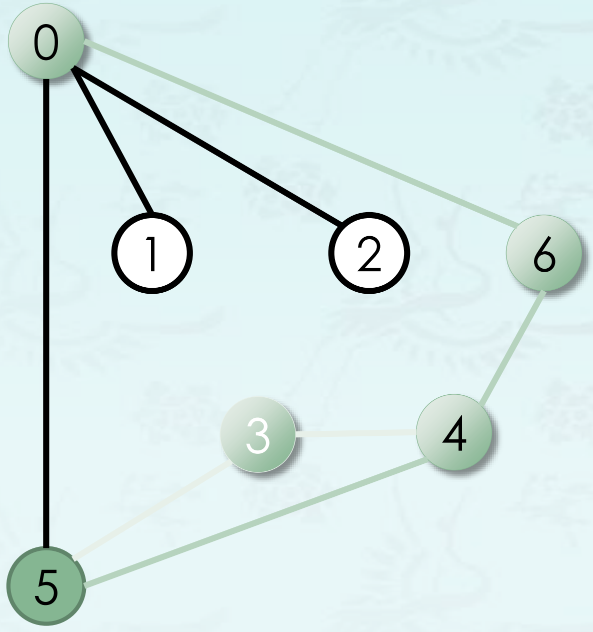
visit 3: check 5, ~~check 4~~



adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

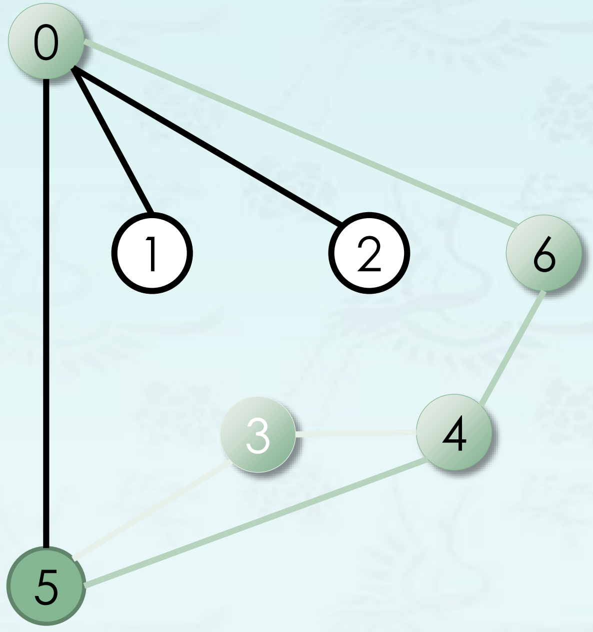
visit 3: check 5, check 4



adj[]	
0	~~6~~ → 2 → 1 → 5
1	0
2	0
3	~~5~~ → ~~4~~
4	~~5~~ → 6 → 3
5	~~3~~ → 4 → 0
6	~~0~~ → 4

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

3 done: What's the next?

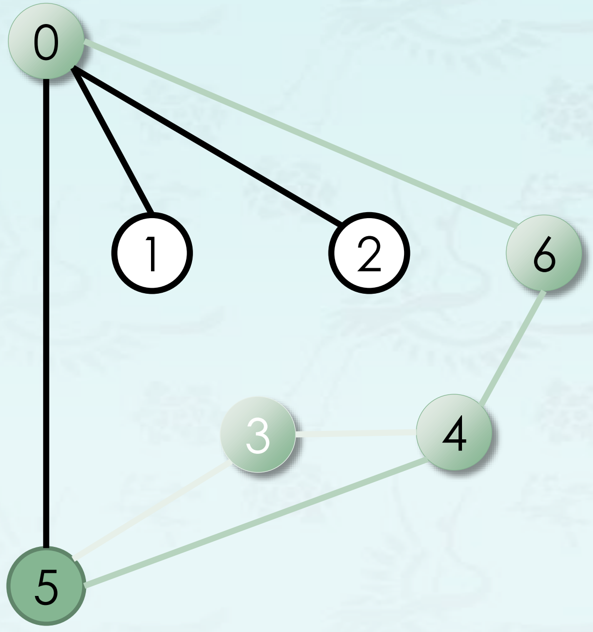


adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

3 done: What's the next? **Backtrack!**

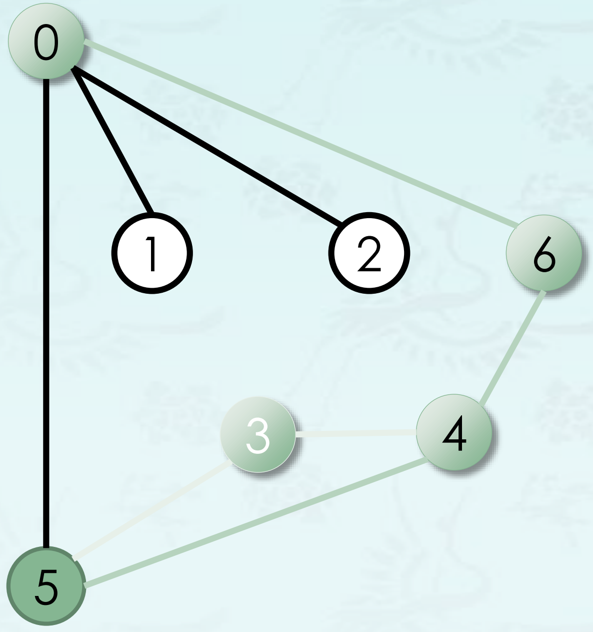
Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

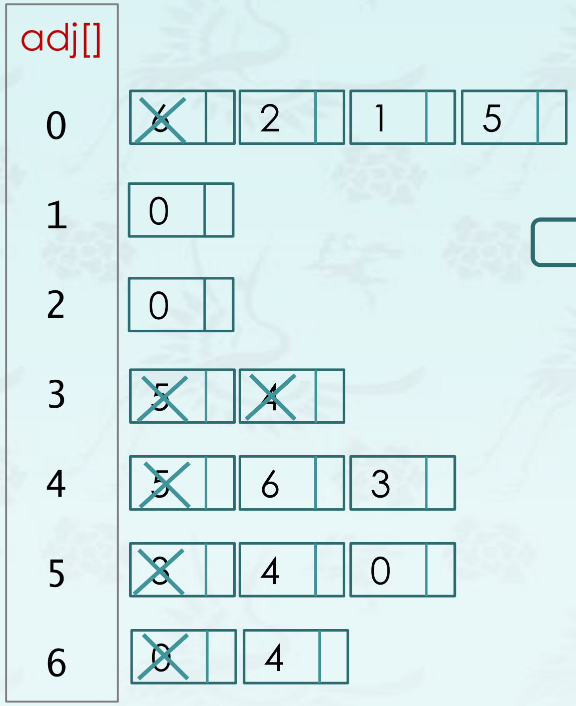


adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

3 done: What's the next? **Backtrack!**
How to?

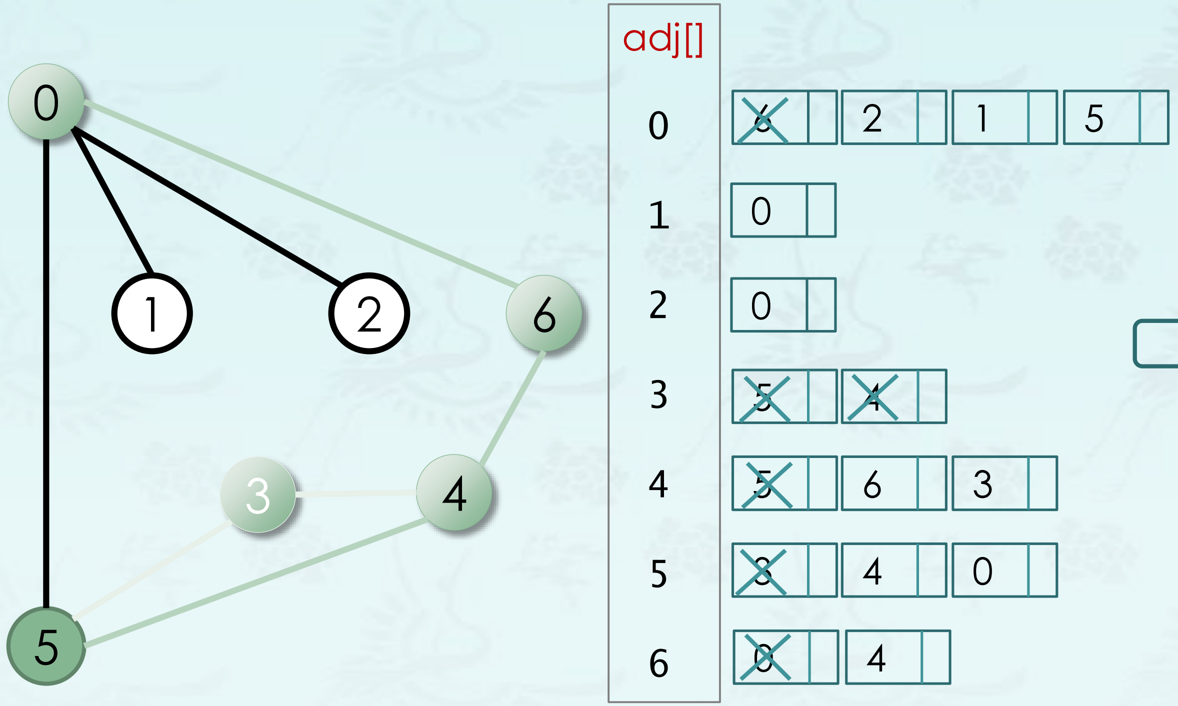




v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

3 done: What's the next? **Backtrack!**

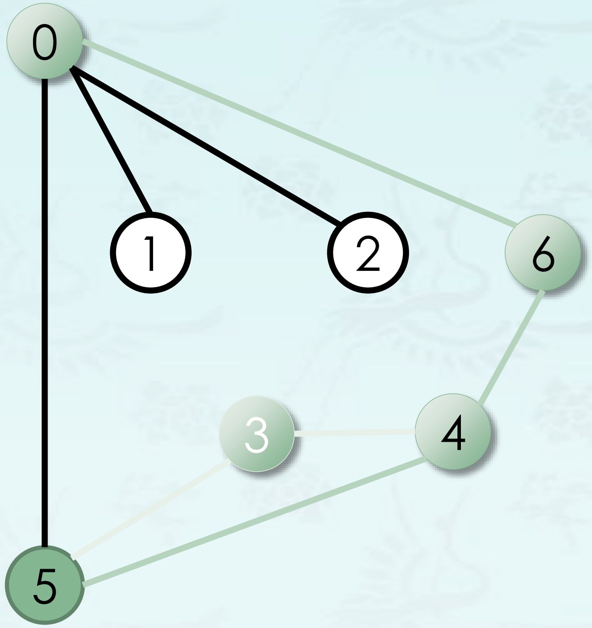
How to? **Use parent[v]** parent[3] = 5



v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done

visit 5: check 3, **check 4**, **check 0**

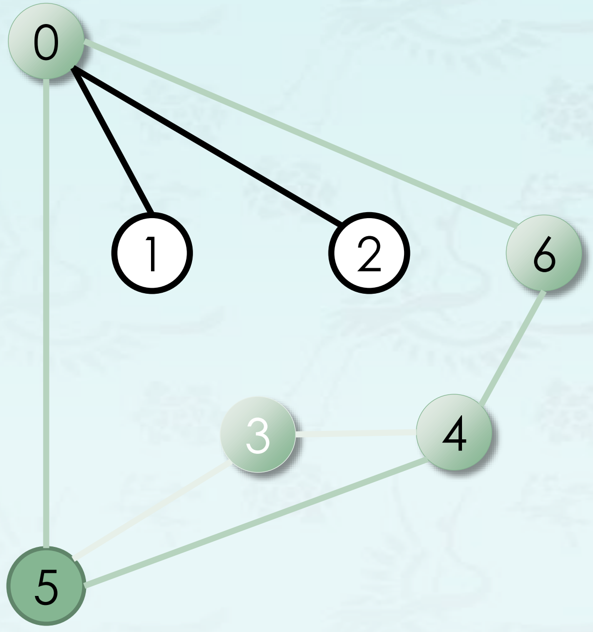


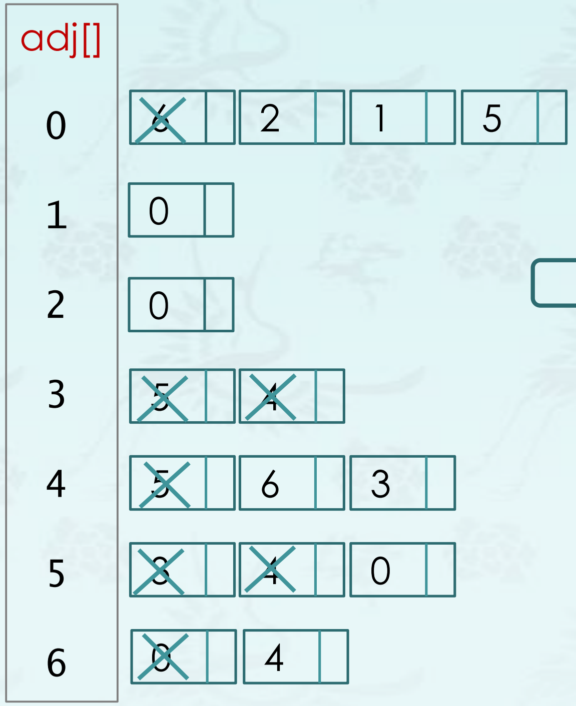
adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done

visit 5: check 3, **check 4**, **check 0**

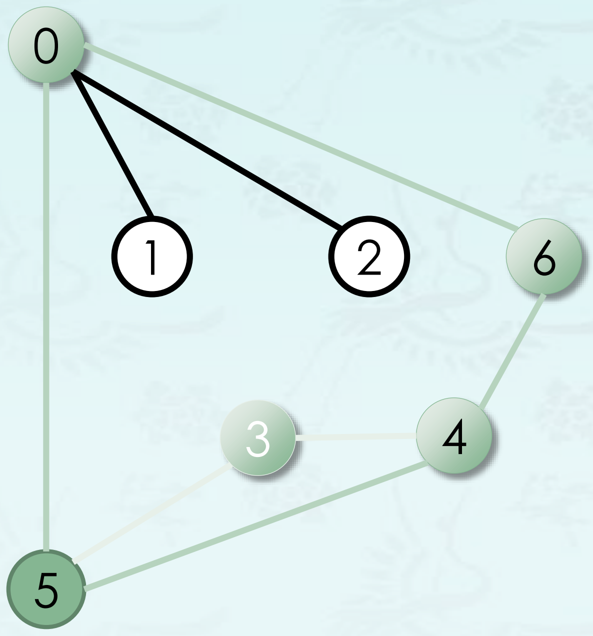




v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done

visit 5: ~~check 3~~, ~~check 4~~, check 0



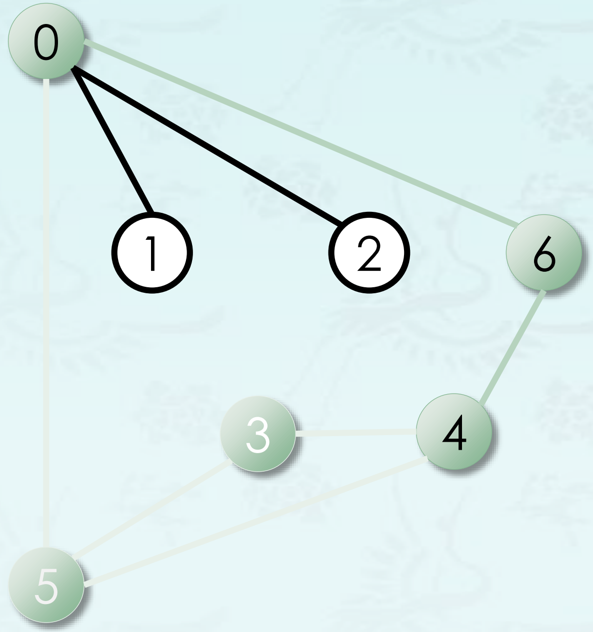
adj[]				
0	6	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	0	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done

visit 5: check 3, check 4, **check 0**

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

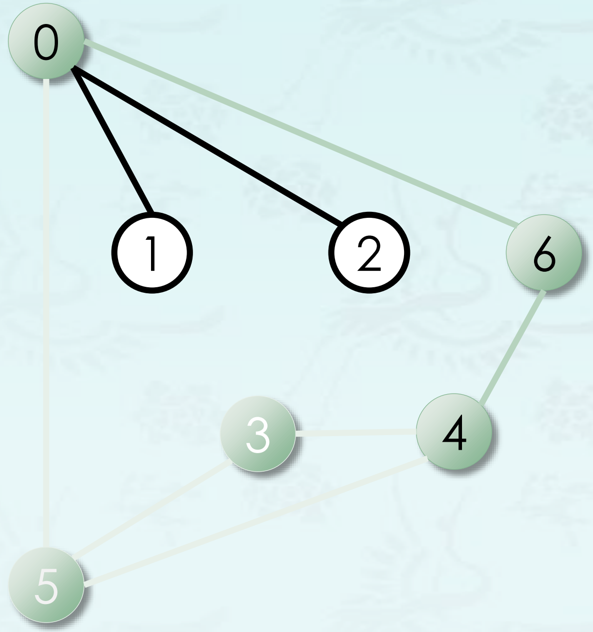


adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

5 done

What's the next? **Backtrack!**
How to?



adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	4		

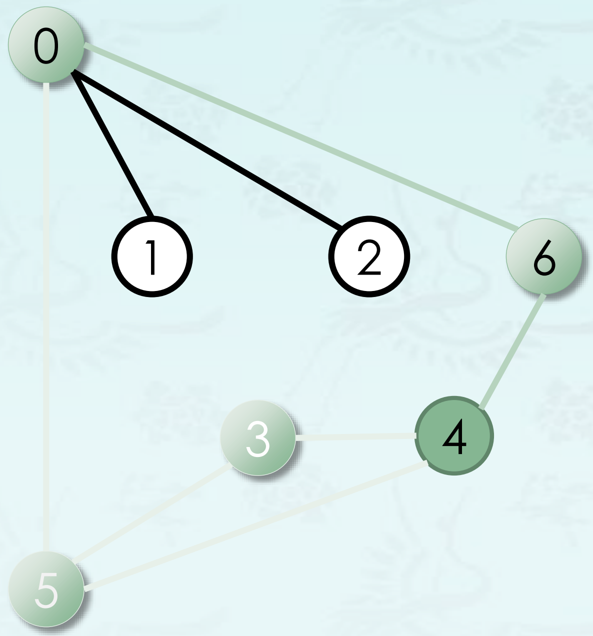
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

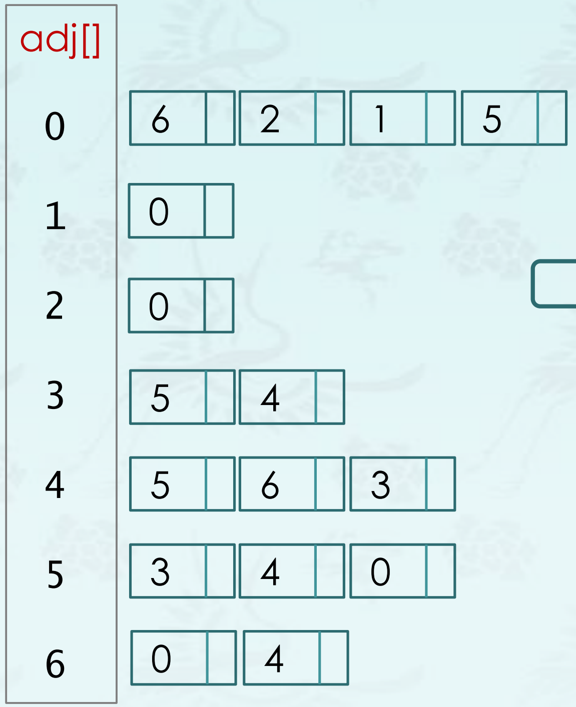
5 done

What's the next? **Backtrack!**

How to? **Use parent[v]**

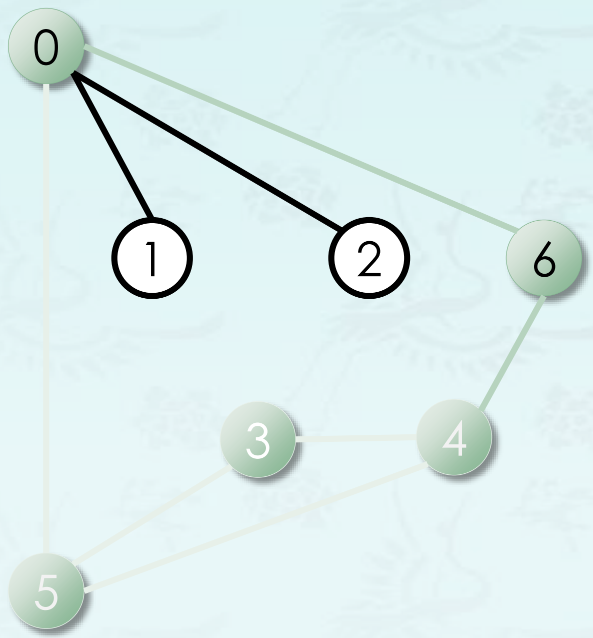
parent[5] = 4





v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

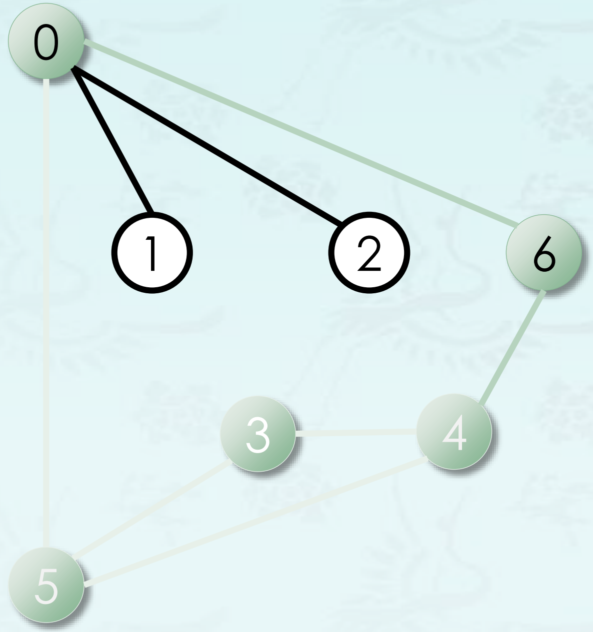
visit 4: check 5, **check 6**, **check 3**

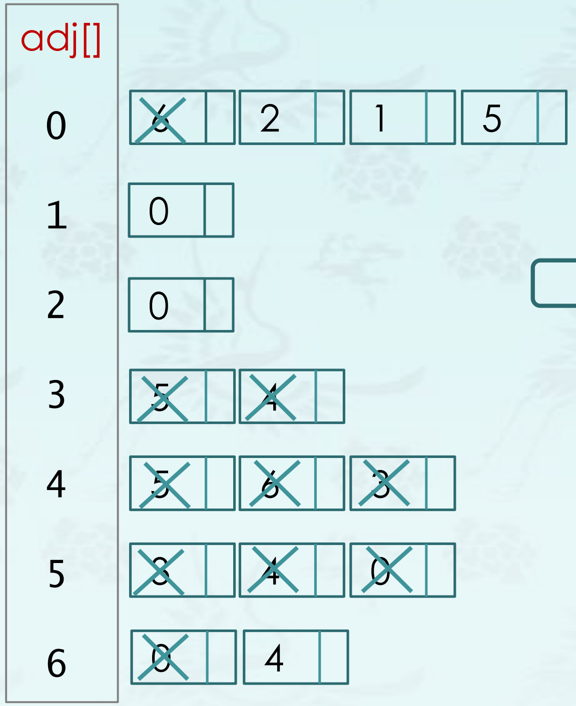


adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	~~6~~	~~3~~	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

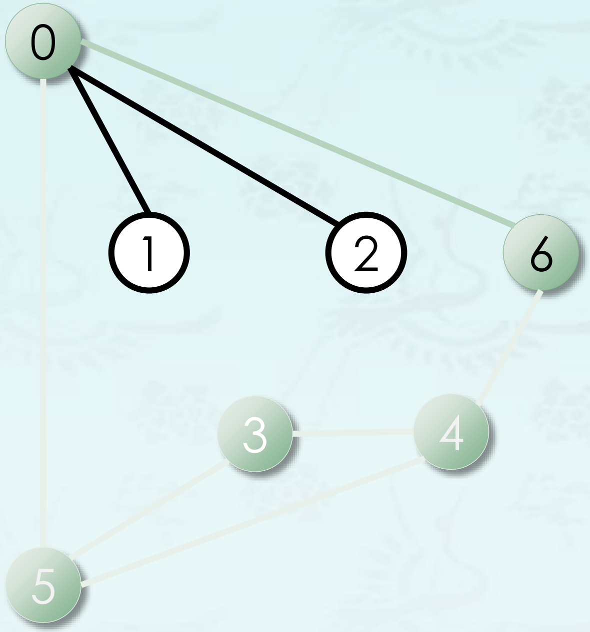
visit 4: check 5, check 6, **check 3** 4 done

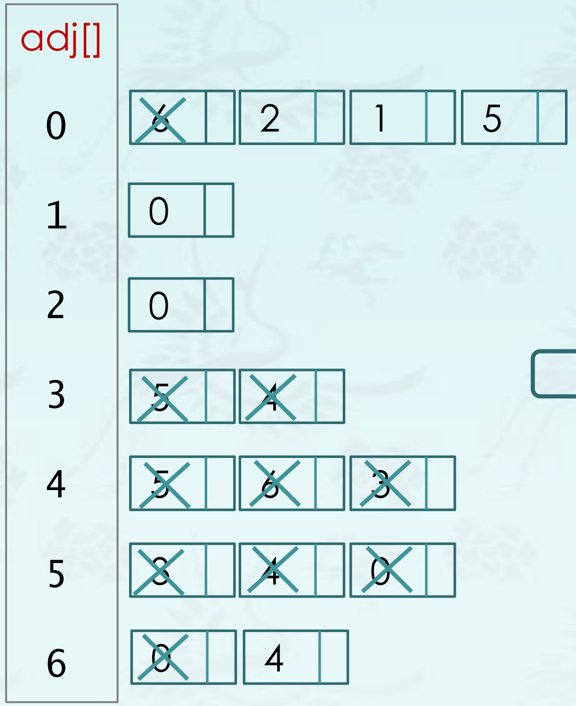




v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

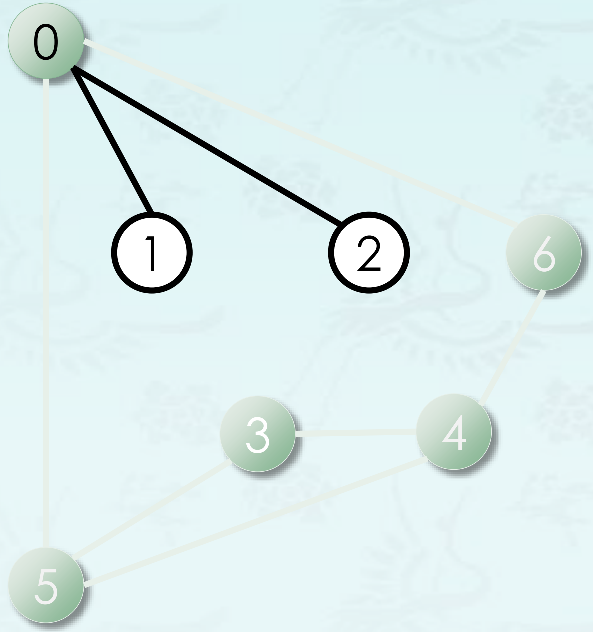
visit 4: **check 5**, **check 6**, **check 3** 4 done **Backtrack!** **parent[4] = 6**





v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

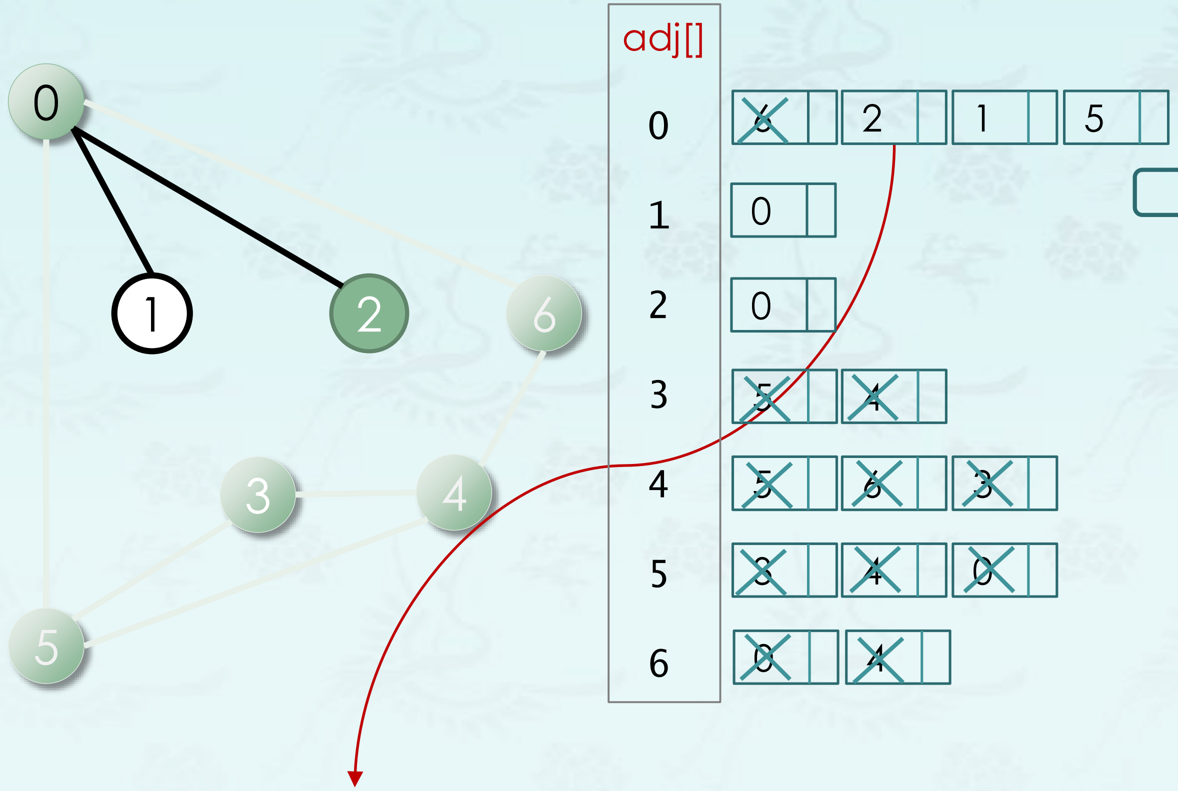
visit 6: **check 0**, **check 4**



adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	~~6~~	~~3~~	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	~~4~~		

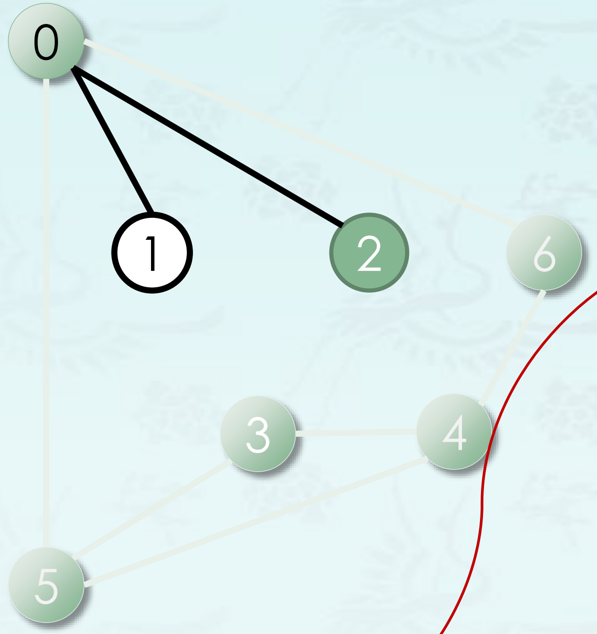
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 6: check 0, check 4 **done 6** **Backtrack!** **parent[6] = 0**



v	marked[]	parent[v]
0	T	–
1	F	–
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: check 6, **check 2**, check 1, and check 5

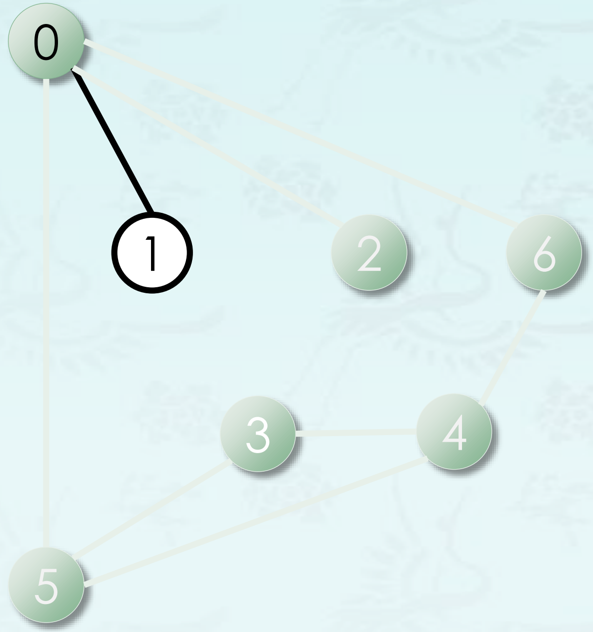


adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	~~6~~	~~3~~	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	~~4~~		

v	marked[]	parent[v]
0	T	–
1	F	–
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 2: check 0

DFS: 0 6 4 5 3 2

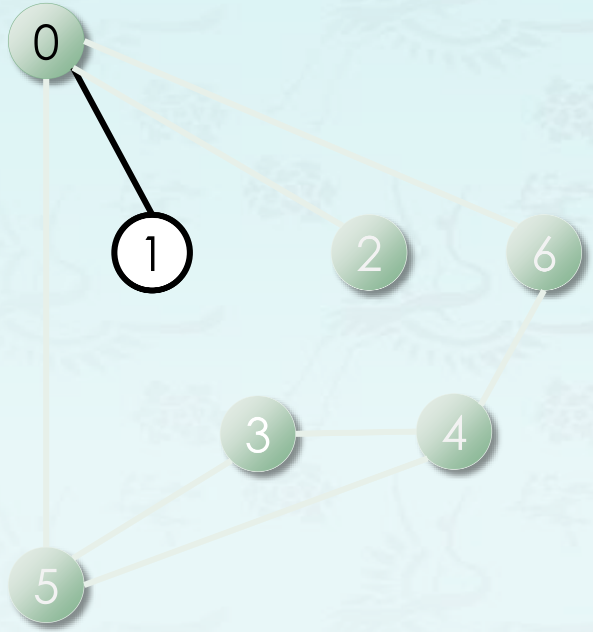


adj[]

0	~~6~~	~~2~~	1	5
1	0			
2	~~0~~			
3	~~5~~	~~4~~		
4	~~5~~	~~6~~	~~3~~	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	~~4~~		

v	marked[]	parent[v]
0	T	–
1	F	–
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

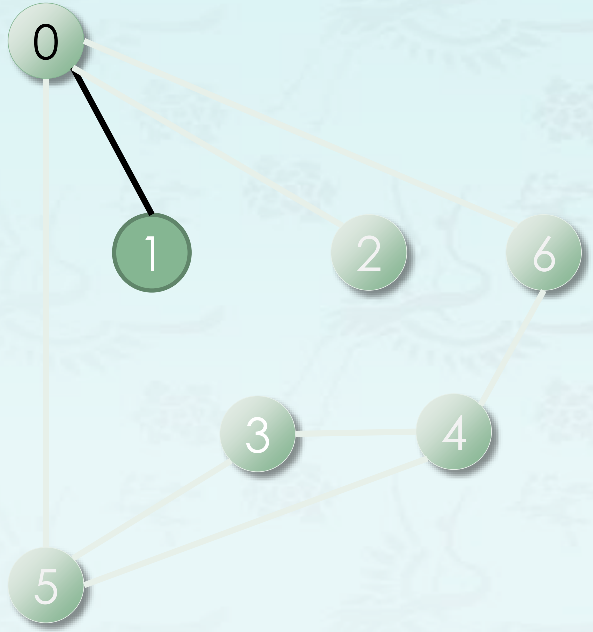
visit 2: check 0 **2 done** **Backtrack!** **parent[2] = 0**



adj[]				
0	~~6~~	~~2~~	1	5
1	0			
2	~~0~~			
3	~~5~~	~~4~~		
4	~~5~~	~~6~~	~~3~~	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	~~4~~		

v	marked[]	parent[v]
0	T	–
1	F	–
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: check 6, check 2, **check 1**, and check 5



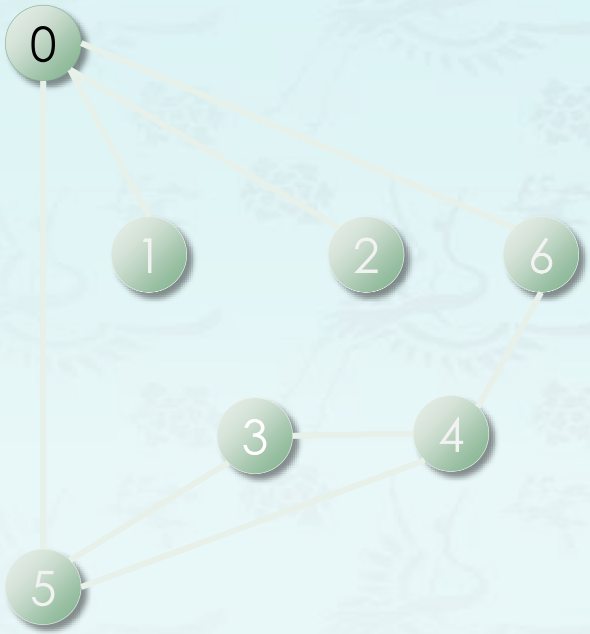
adj[]				
0	~~6~~	~~2~~	~~1~~	5
1	~~0~~			
2	~~0~~			
3	~~5~~	~~4~~		
4	~~5~~	~~6~~	~~3~~	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	~~4~~		

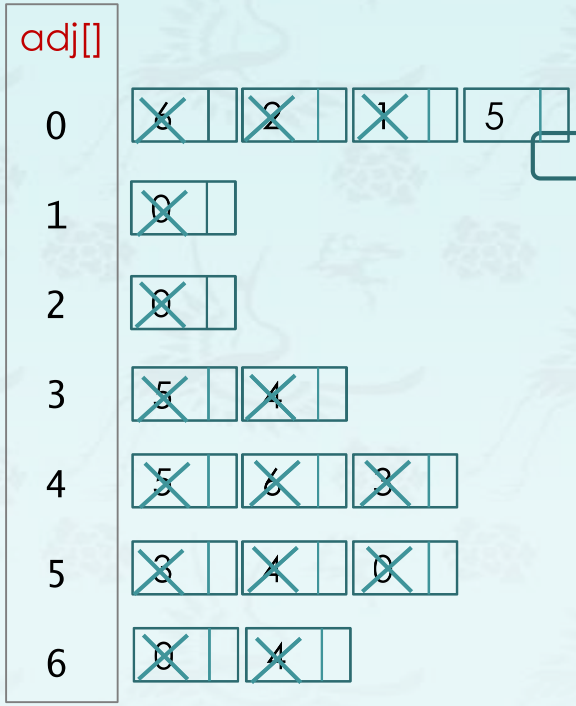
v	marked[]	parent[v]
0	T	–
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 1: check 0

1 done **Backtrack!** **parent[1] = 0**

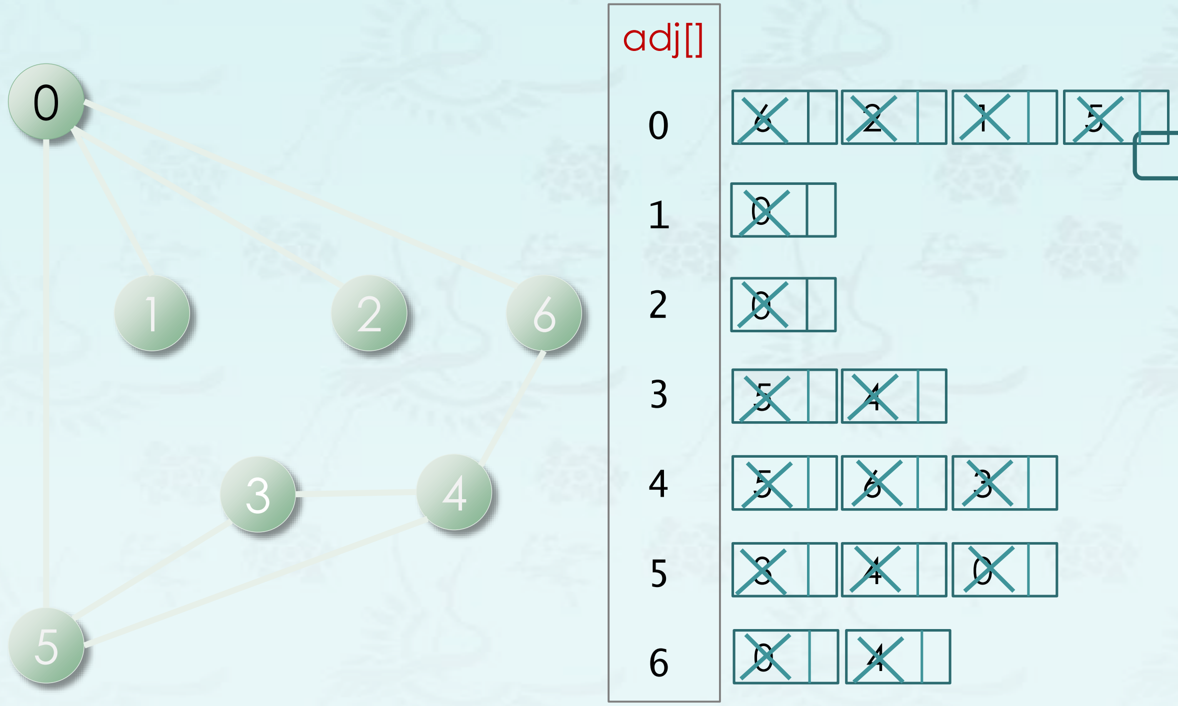
DFS: 0 6 4 5 3 2 1





v	marked[]	parent[v]
0	T	–
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: **check 6, check 2, check 1, and check 5**



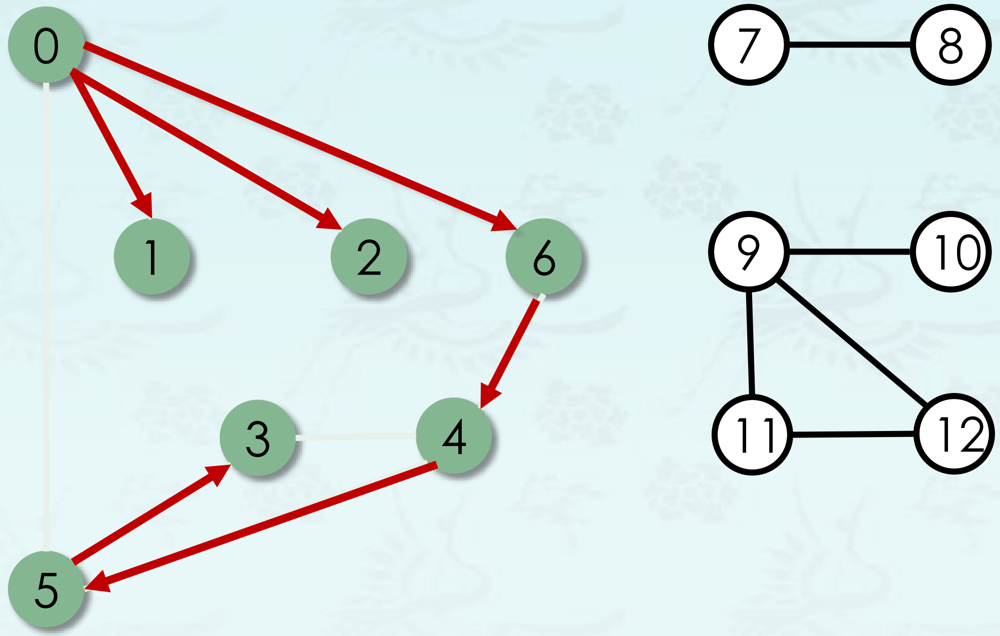
v	marked[]	parent[v]
0	T	–
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: **check 6, check 2, check 1, and check 5**
0 done

DFS: Depth-First Search Demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



v	marked[]	parent[v]
0	T	–
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

DFS Output: DFS: 0 6 4 5 3 2 1

- found vertices reachable from 0
- build a data structure **parent[v]**

DFS: Depth-First Search Demo

Goal: Find all vertices connected to s (and a corresponding path).
Idea: Mimic maze exploration

Algorithm:

- Use recursion (ball of string).
- Mark each visited vertex (and keep track of edge taken to visit it).
- Return (retrace steps) when no unvisited options.

Data Structures:

- `Boolean[] marked` to mark visited vetices.
- `int[] parent` to keep tree of paths.
  (`parent[w] == v`) means that edge v-w taken to visit w for first time

DFS: Depth-First Search Coding

```
// DFS - find vertices connected to v
void DFS(graph g, int v, queue<int>& que) {
  if (empty(g)) return;
  for (int i = 0; i < V(g); i++)
    g->marked[i] = false;
  _DFS(g, v, que);   // recursive _DFS at v
  g->DFSv = que;     // save result at DFSv at v
}
```

DFS: Depth-First Search Coding

```
// DFS - find vertices connected to v
void DFS(graph g, int v, queue<int>& que) {
  if (empty(g)) return;
  for (int i = 0; i < V(g); i++)
    g->marked[i] = false;
  _DFS(g, v, que);    // recursive _DFS at
  g->DFSv = que;     // save result at DF
}
```

```
// Recursive _DFS does the work
void _DFS(graph g, int v, queue<int>& que) {
  g->marked[v] = true;      'v' current visiting vertex
  que.push(v);              // save the path

  for (gnode w = g->adj[v].next; w; w = w->next) {
    if (!g->marked[w->item]) {
      _DFS(g, w->item, que);
      g->parentDFS[w->item] = v;
    }
  }
}
```

keep where it reached from

DFS: Depth-First Search Properties

Proposition: After DFS, can find vertices connected to s in constant time and can find a path to s (if one exists) in time proportional to its length.

Proof: parent[] is parent-link representation of a tree rooted at s.

v	marked[]	parent[v]
0	T	-
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

```
// returns a path from v to w using the result of DFS's parent[].
// It has to use a stack to retrace the path back to the source.
// Once the client(caller) gets a stack returned,
void DFSpath(graph g, int v, int w, stack<int>& path) {
  if (empty(g)) return;

  // DFS at v, starting vertex
  queue<int> q;
  DFS(g, v, q);

  path = {};
  for (int x = w; x != v; x = g->parentDFS[x])
    path.push(x);

  path.push(v);
}
```

DFS: Depth-First Search Coding

Proposition: After DFS, can find vertices connected to s in constant time and can find a path to s (if one exists) in time proportional to its length.

Proof: parent[] is parent-link representation of a tree rooted at s.

What is the path from vertex 0 to vertex 3?
In this case, what is in the stack when parnet() returns?

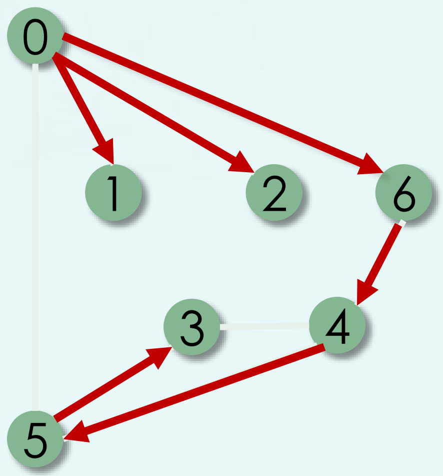
```
// returns a path from v to w using the result of DFS's parent[].
// It has to use a stack to retrace the path back to the source.
// Once the client(caller) gets a stack returned,
void DFSpath(graph g, int v, int w, stack<int>& path) {
  if (empty(g)) return;

  // DFS at v, starting vertex
  queue<int> q;
  DFS(g, v, q);

  path = {};
  for (int x = w; x != v; x = g->parentDFS[x])
    path.push(x);

  path.push(v);
}
```

v	marked[]	parent[v]
0	T	-
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-



DFS: Depth-First Search Coding

Proposition: After DFS, can find vertices connected to s in constant time and can find a path to s (if one exists) in time proportional to its length.

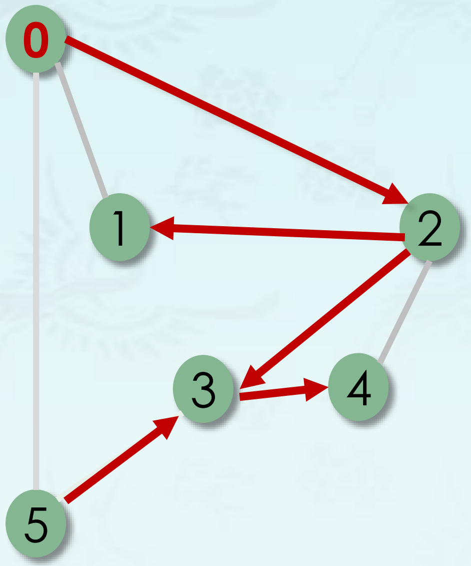
Proof: parent[] is parent-link representation of a tree rooted at s.

```
// returns a path from v to w using the result of DFS's parent[].
// It has to use a stack to retrace the path back to the source.
// Once the client(caller) gets a stack returned,
void DFSpath(graph g, int v, int w, stack<int>& path) {
  if (empty(g)) return;

  // DFS at v, starting vertex
  queue<int> q;
  DFS(g, v, q);

  path = {};
  for (int x = w; x != v; x = g->parentDFS[x])
    path.push(x);

  path.push(v);
}
```



parent[]

0	
1	2
2	0
3	2
4	3
5	3

a wrong edge between ?

Graph

- Introduction
- Graph API
- Elementary Graph Operations
  - **DFS: Depth first search**
  - BFS: Breadth first search
  - CC: Connected Components

Major references:
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Prof. Youngsup Kim, idebtor@handong.edu, 2014 Data Structures, CSEE Dept., Handong Global University

Graph

- Introduction
- Graph API
- Elementary Graph Operations
  - DFS: Depth first search
  - BFS: Breadth first search
  - **CC: Connected Components**

Major references:
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Prof. Youngsup Kim, idebtor@handong.edu, 2014 Data Structures, CSEE Dept., Handong Global University

Connectivity Queries

Def.: Vertices v and w are connected if there is a path between them.

Goal: Preprocess graph to answer queries of the form "*is v connected to w?*" in constant time.

public class CC		
	CC(Graph G)	*find connected components in G*
boolean	connected(int v, int w)	*are v and w connected?*
int	count()	*number of connected components*
int	id(int v)	*component identifier for v*

Depth-first search? Yes …

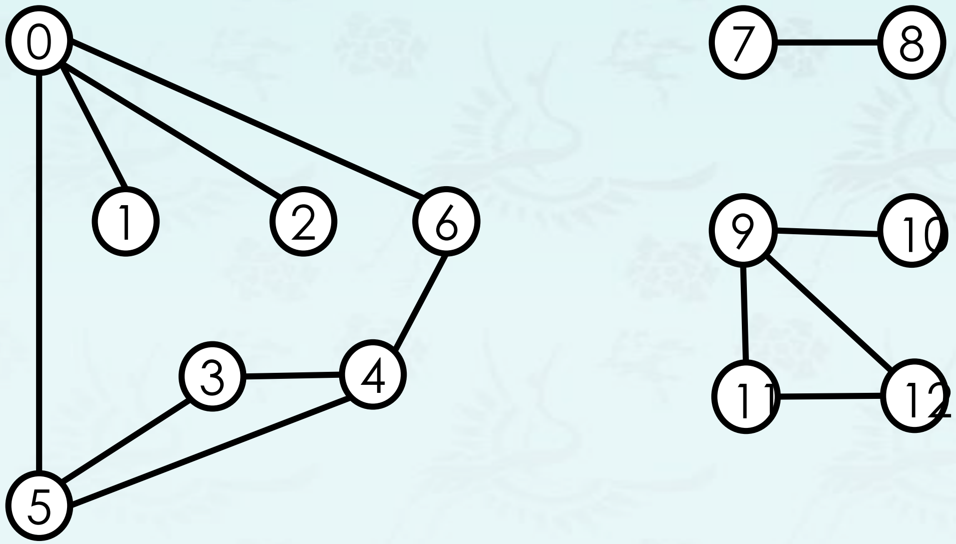
Connected Components

The relation "is connected to" is equivalence relation:

Reflexive: v is connected to v.

Symmetric: if v is connected to w, then w is connected v.

Transitive: if v connected to w and w connected to x, then v connected to x



Connected Components

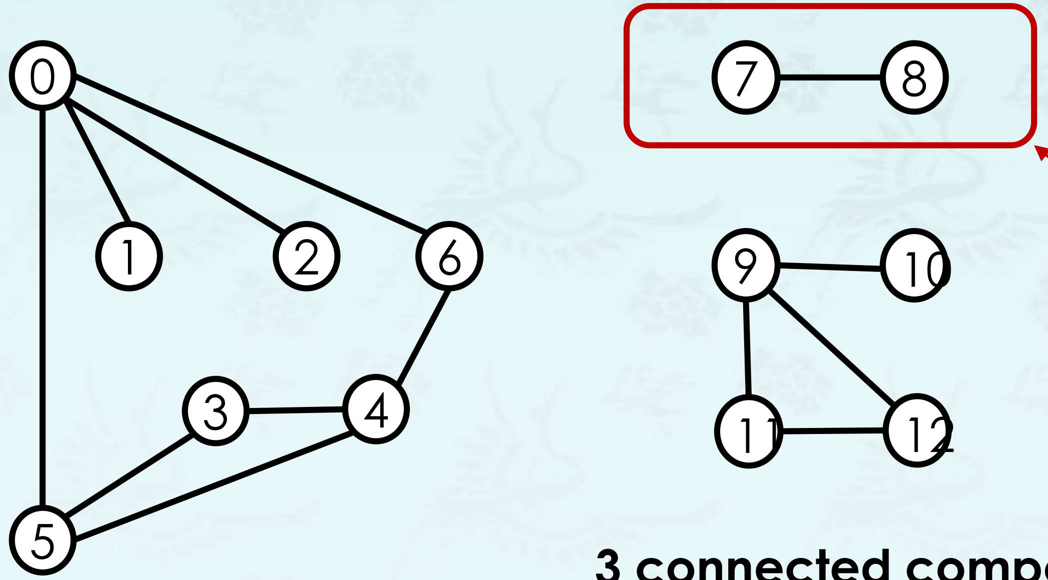
The relation "is connected to" is equivalence relation:

Reflexive: v is connected to v.

Symmetric: if v is connected to w, then w is connected v.

Transitive: if v connected to w and w connected to x, then v connected to x

Def.: A connected component is a maximal set of connected vertices.



3 connected components

v	id[v]
0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	2
8	2
9	3
10	3
11	3
12	3

Connected Components

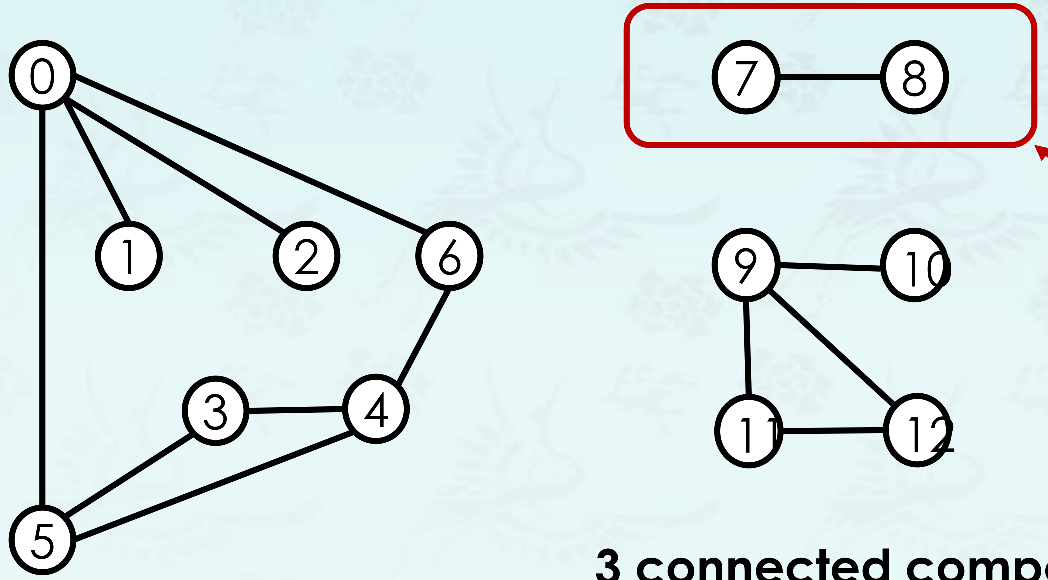
The relation "is connected to" is equivalence relation:

Reflexive: v is connected to v.

Symmetric: if v is connected to w, then w is connected v.

Transitive: if v connected to w and w connected to x, then v connected to x

Def.: A connected component is a maximal set of connected vertices.



3 connected components

v	id[v]
0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	2
8	2
9	3
10	3
11	3
12	3

Remark: Given connected components, can answer queries in constant time.

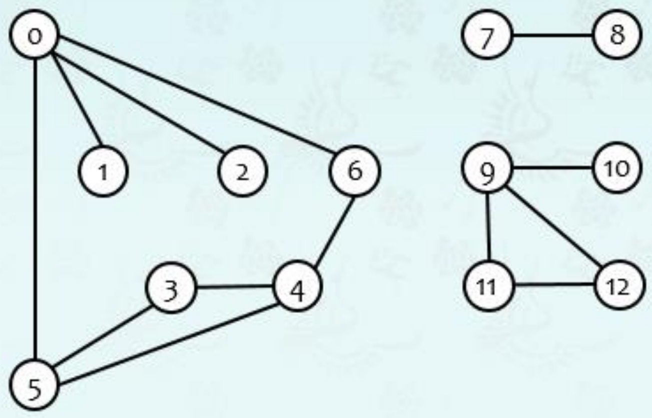
Connected Components

Goal: Partition vertices into connected components.

Connected components

Initialize all vertices v as unmarked.

For each unmarked vertex v, run DFS to identify all vertices discovered as part of the same component.

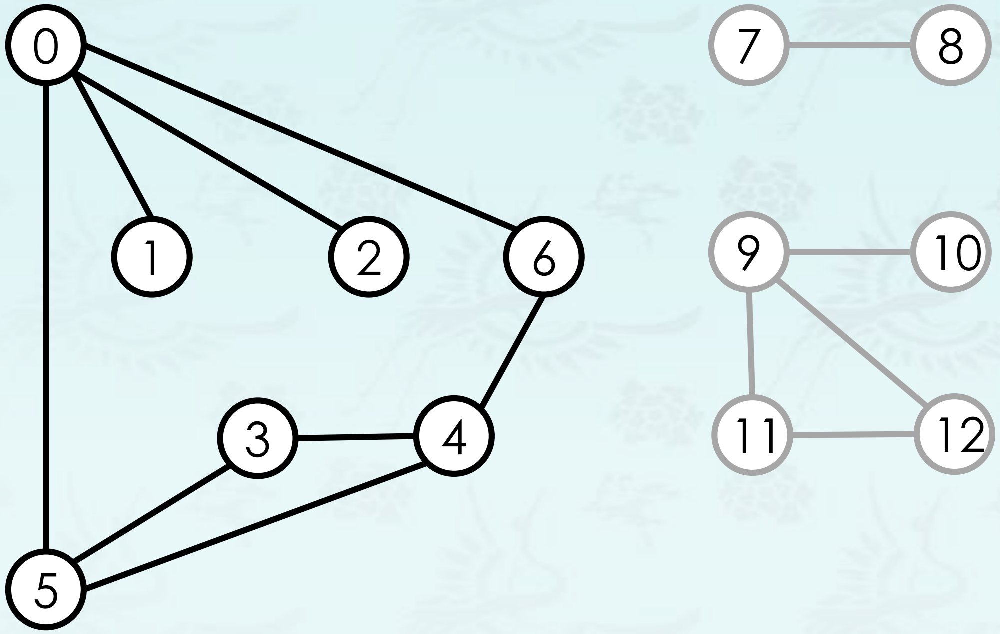



```
graph3.txt
13
13
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

Connected Components

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



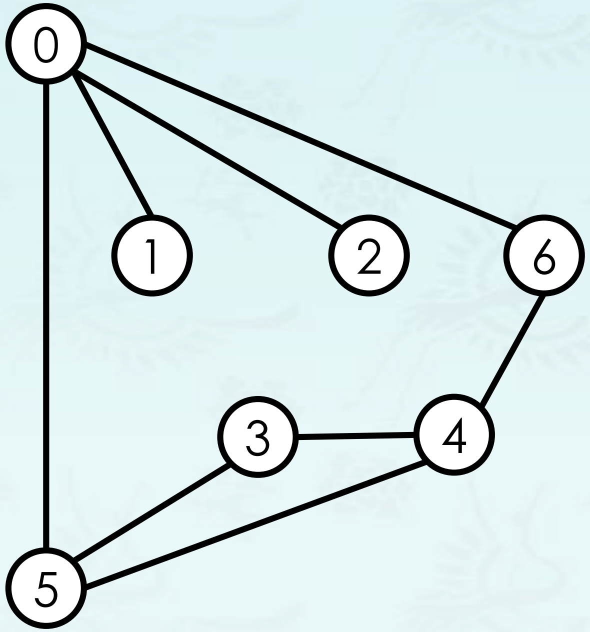
V-E lists →

```
graph3.txt
13  ← V
13  ← E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

Graph g:

Challenge: build adjacency lists?

Connected Components



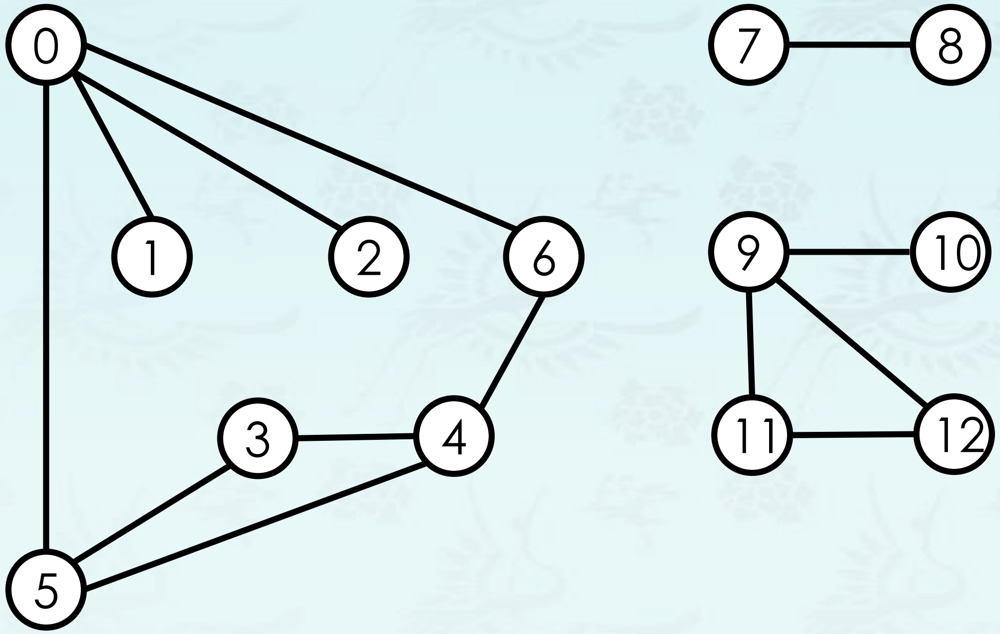
Graph g

Adjacency lists

adj[]				
0	6	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	0	4		

V-E lists

```
graph3.txt
13  ←  V
13  ←  E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

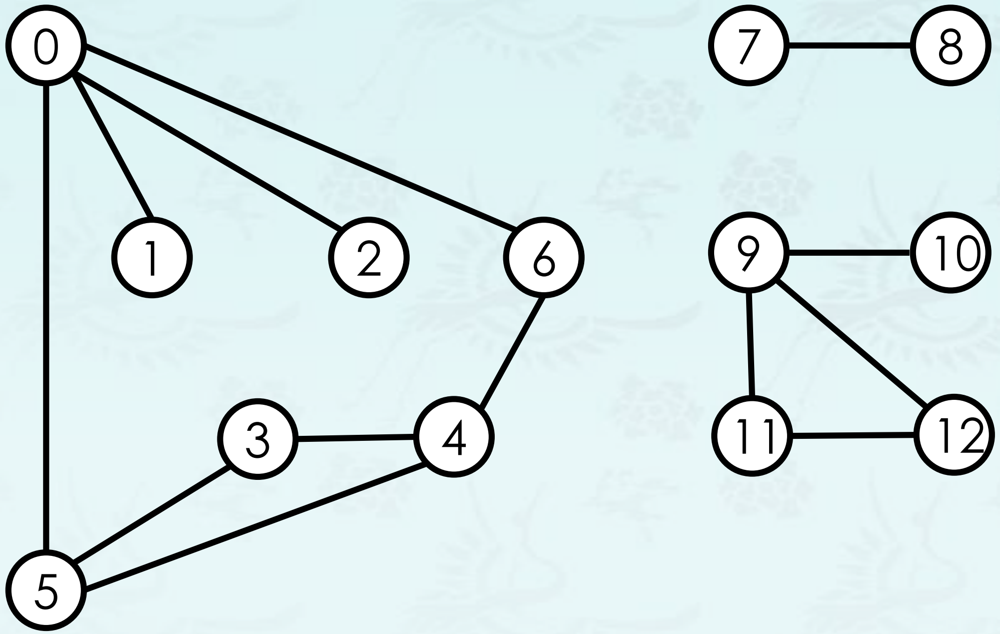


V-E lists → **graph3.txt**

```
13   ← V
13   ← E
0  5
4  3
0  1
9  12
6  4
5  4
0  2
11  12
9  10
0  6
7  8
9  11
5  3
```

Graph g:

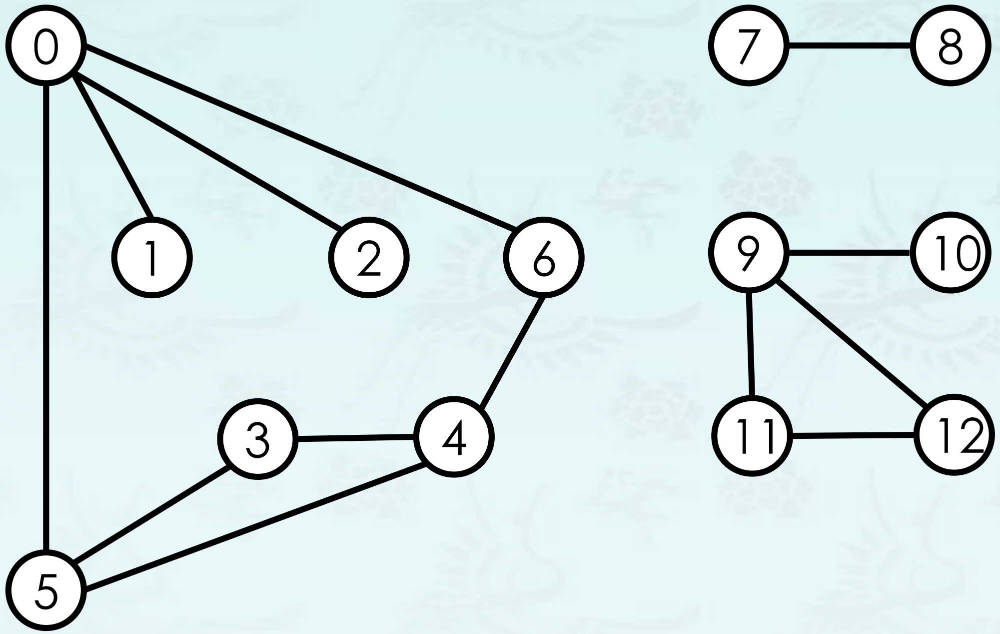
Connected Components



v	marked[]	id[]
0	F	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Graph g:

Connected Components



v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

Done:

Connected Components – Coding

```
// returns true if v and w are connected.
bool connected(graph g, int v, int w) {
  if (empty(g)) return true;

  queue<int> q;
  DFS(g, v, q);

  return g->CCID[v] == g->CCID[w];
}
```

```
// returns number of connected components.
int nCCs(graph g) {
  int id = g->CCID[0];
  int count = 1;
  for (int i = 0; i < V(g); i++)
    if (id != g->CCID[i]) {
      id = g->CCID[i];
      count++;
    }
  return id == 0 ? 0 : count;
}
```

Graph

- Introduction
- Graph API
- Elementary Graph Operations
  - DFS: Depth first search
  - BFS: Breadth first search
  - **CC: Connected Components**
- pset – graph.cpp
  - **implement DFS, BFS, CC and others**

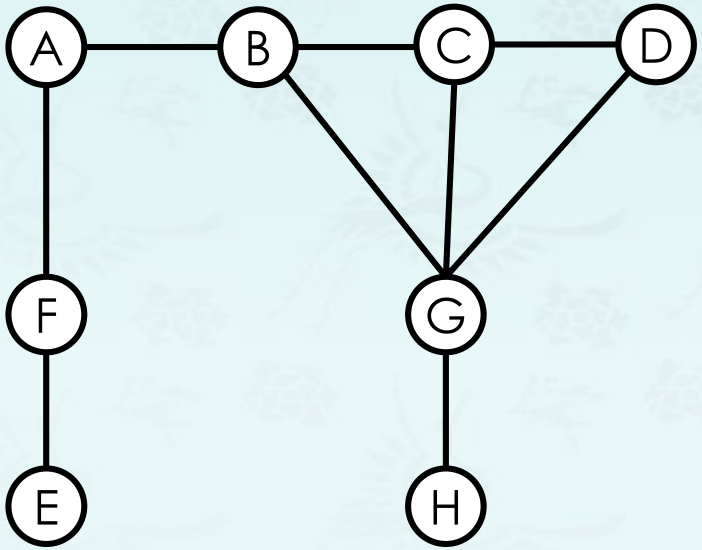
Major references:
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Prof. Youngsup Kim, idebtor@handong.edu, 2014 Data Structures, CSEE Dept., Handong Global University

DFS - Exercise

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



adjacent list

A: B F
B: G C A
C: D G B
D: C G
E: F
F: E A
G: H B C D
H: G

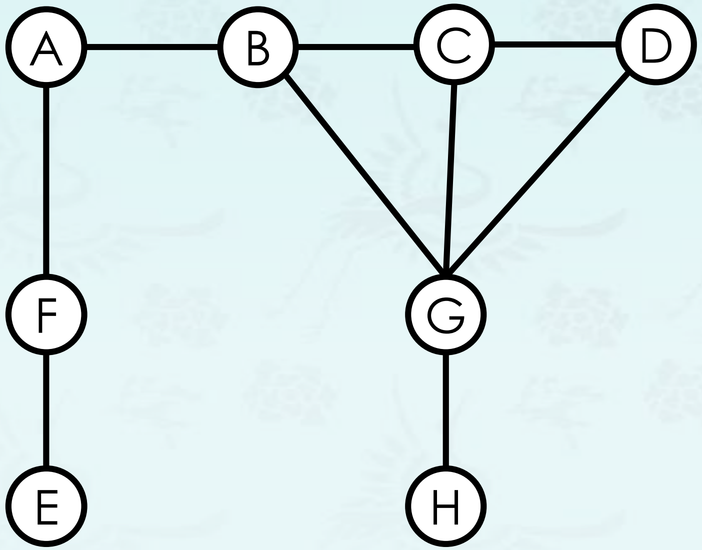
Graph g:

Hint: A B…?…F E

DFS - Exercise

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Graph g:

adjacent list

A: B F

B: G C A

C: D G B

D: C G

E: F

F: E A

G: H B C D

H: G

Hint: A B G H C D F E

```
dfs(A)
 dfs(B)
  dfs(G)
   dfs(H)
    check G
   H done
   check B
   dfs(C)
    dfs(D)
     check C
     check G
    D done
    check G
    check B
   C done
   check D
  G done
  check C
  check A
 B done
 dfs(F)
  dfs(E)
   check F
  E done
  check A
 F done
A done
check B
check C
check D
check E
check F
check G
check H
```