

# Data Structures

## Chapter 7: Graph

1. Introduction
  - Terminology, Representation, ADT
2. **Basic Operations**
  - **DFS**, CC, BFS, Processing
3. Digraph and Applications
4. Minimum Spanning Tree(MST)



**죄의 삯은 사망이요 하나님의 은사는 그리스도 예수 우리 주 안에 있는 영생이니라 (로마서 6:23)**

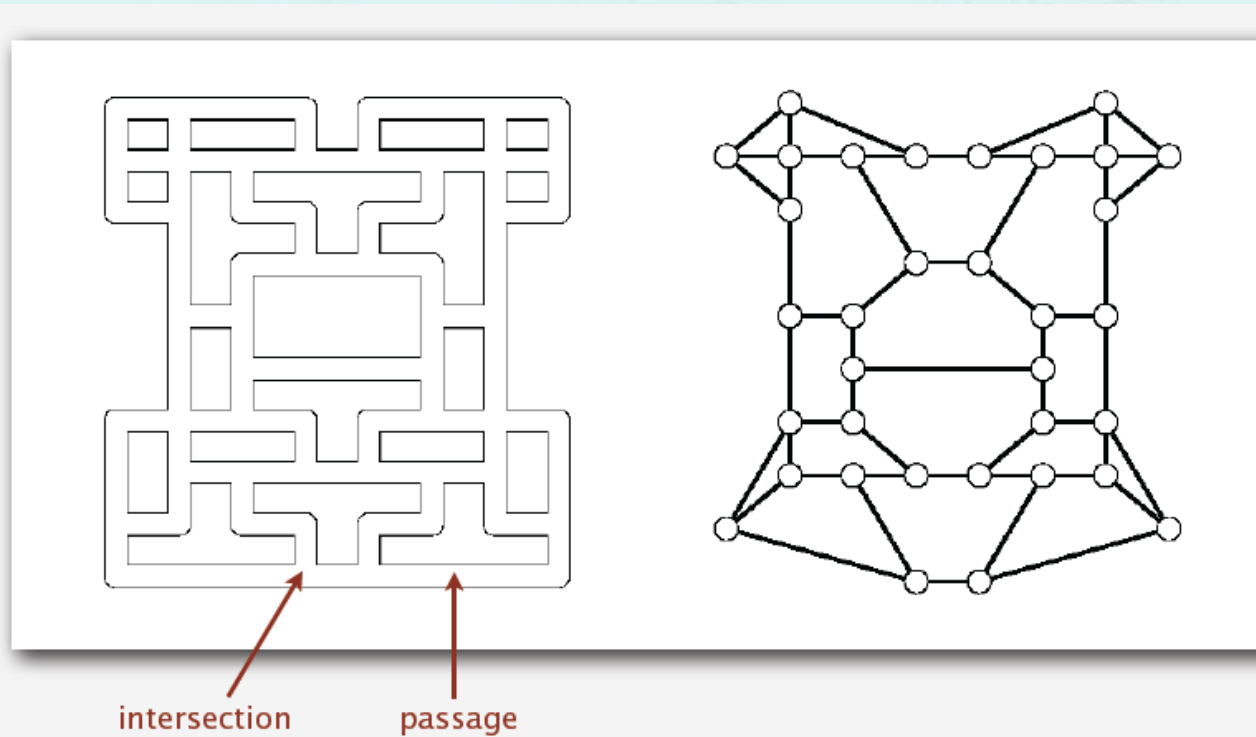
**모든 사람이 죄를 범하였으매 하나님의 영광에 이르지 못하더니 그리스도 예수 안에 있는 속량으로 말미암아 하나님의 은혜로 값없이 의롭다 하심을 얻은 자 되었느니라 (로마서 3:23-24)**

# DFS: Depth-First Search

## Algorithm:

- Vertex = intersection
- Edge = passage

pacman

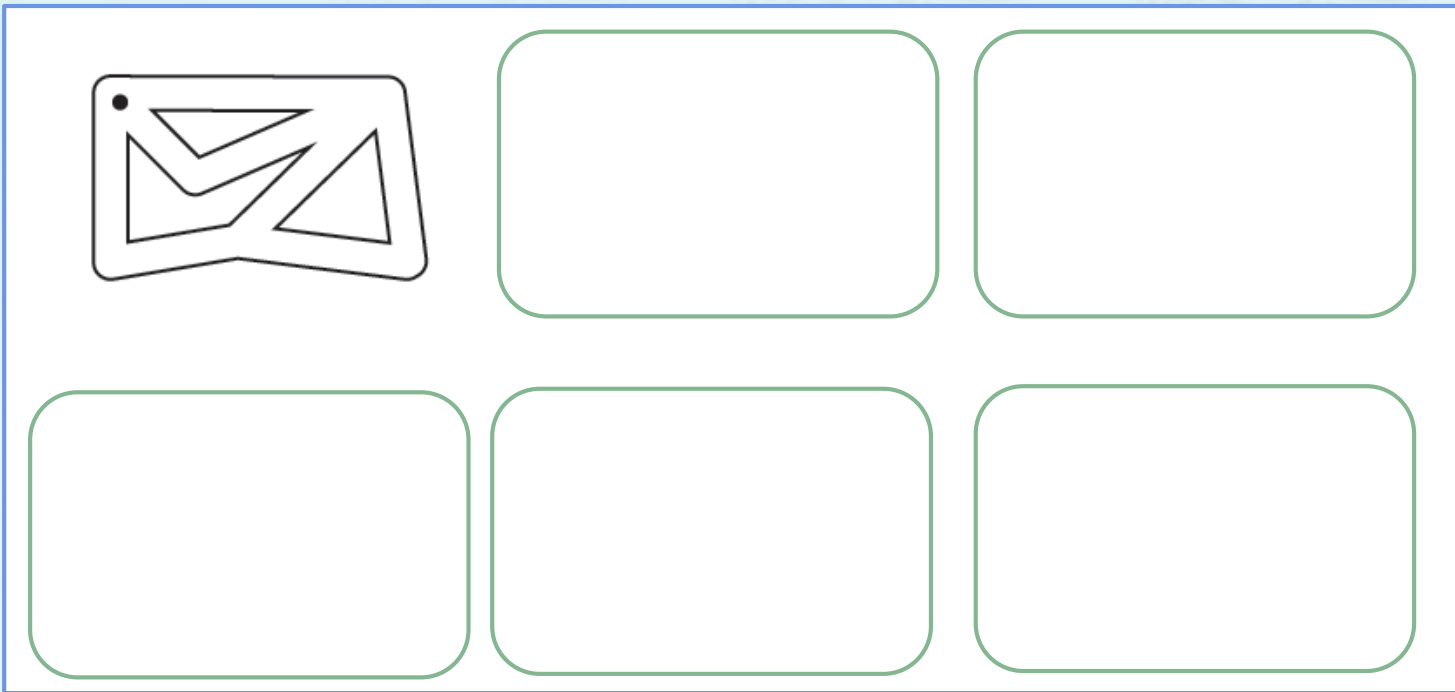


**Maze Goal:** Explore every intersection in the maze.

# 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.

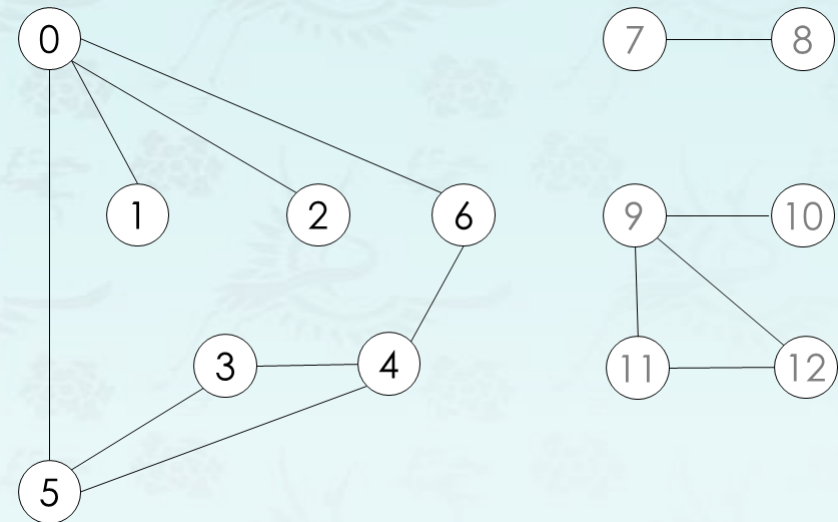
**Maze Goal:** Explore every intersection in the maze.

# 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**
  - path from  $v$  to  $w$
  - distance from  $v$  to  $w$
  - connected
  - bipartite
  - cyclic



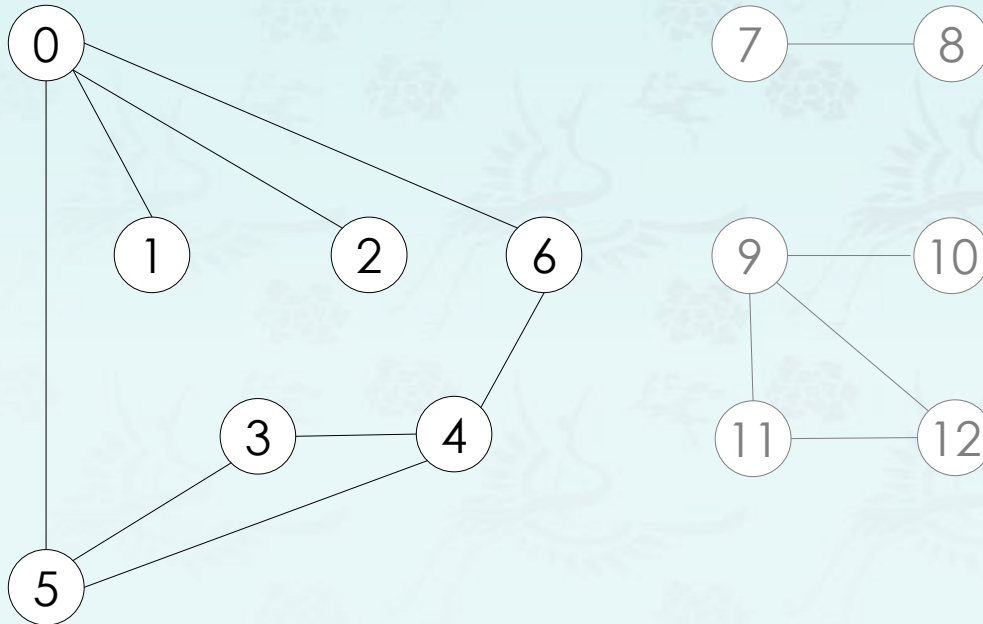
# Graph – Build Adjacency list

- For each edge( $v, w$ ) in the list
- Insert front each vertex both ( $\text{adj}[v], w$ ) and ( $\text{adj}[w], v$ )  
`addEdgeFromTo(g, v, w); // from v to w.`

Edge list

graph6.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:

Adjacency lists

adj[v]

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

**Challenge:** build adjacency lists?

# Graph – ADT

```
// a structure to represent an adjacency list of vertices
struct Gnode {
    int    item;
    Gnode* next;
    Gnode (int i, Gnode *p = nullptr) {
        item = i;  next = p;
    }
    ~Gnode() {}
};

using gnode = Gnode *;
```

adjacent vertices using a singly **linked list**

next vertex to link if any.

# Graph ADT – graph.h

```
struct Graph {  
    int V;           // N vertices  
    int E;           // N edges  
    gnode adj;       // array of linked lists of vertices  
    Graph(int v = 0) { // constructs a graph with v vertices  
        V = v;  
        E = 0;  
        adj = new (nothrow) Gnode[v];  
        assert(adj != nullptr);  
  
        for (int i = 0; i < v; i++) { // initialize adj list as empty;  
            adj[i].next = nullptr; ← set each adj list nullptr  
            adj[i].item = i;         ← to begin with  
        }                          ← unused;  
    }                               but may store the degree of vertex i.  
    ~Graph() {}  
};  
using graph = Graph *;
```

```
graph g = new Graph(v);  
for (int i = 0; i < E; i++)  
    addEdge(g, from[i], to[i]);
```

## Graph ADT – graph.cpp

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

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

With a bug

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


## Graph ADT – graph.cpp

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

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

```
} With a bug
```

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



instantiate a node w and  
make a link with vertex v.  
what is wrong?



## Graph ADT – graph.cpp

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

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

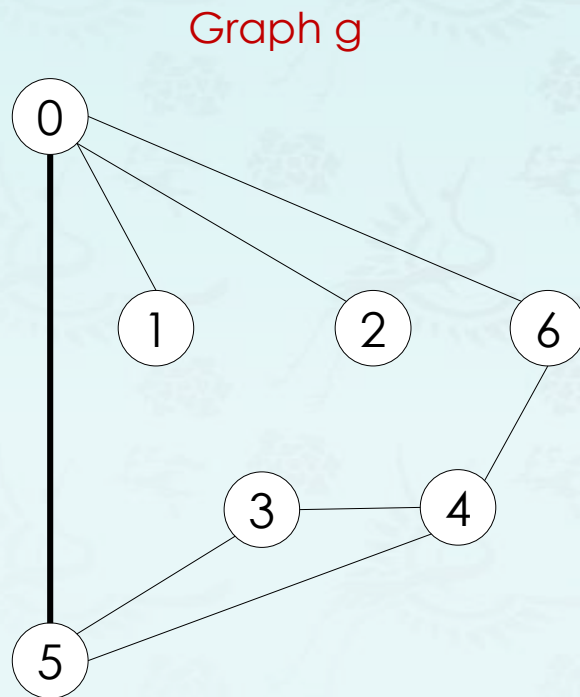
instantiate a node w and  
insert it at **the front of** adj[v]

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

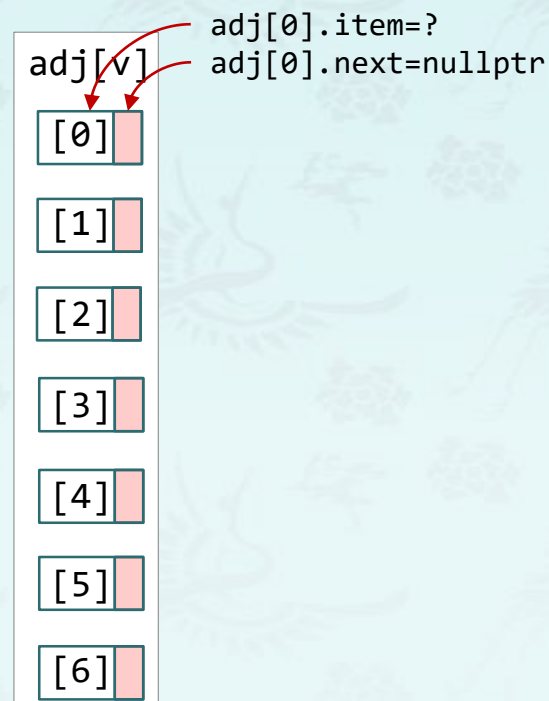
# Graph – Build Adjacency list

```
void addEdgeFromTo(graph g, int v, int w) {  
    gnode node = new Gnode(w, g->adj[v].next);  
    g->adj[v].next = node;  
    g->E++;  
}
```

```
struct Graph {  
    int V, E;  
    gnode adj;  
    Graph(int v = 0) {  
        V = v;  
        E = 0;  
        adj = new (nothrow) Gnode[v];  
        assert(adj != nullptr);  
  
        for (int i=0; i<v; i++)  
            adj[i].next = nullptr;  
    }  
    ~Graph() {}  
};  
using graph = Graph *;
```



Adjacency lists



Edge list

graph6.txt

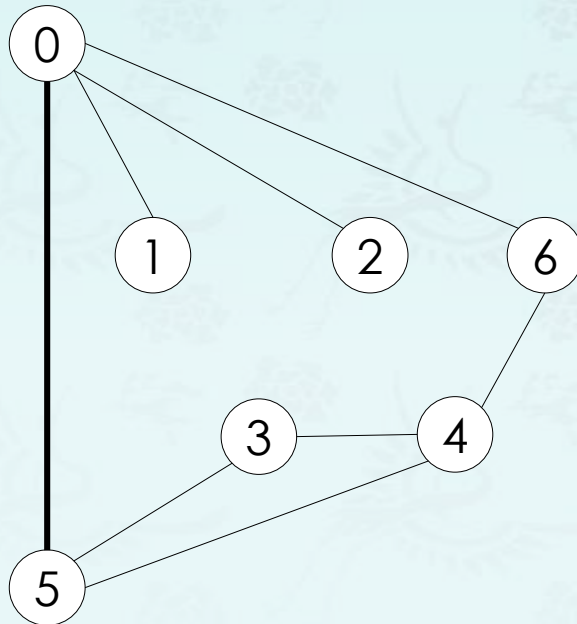
13	←	V
13	←	E
0 5		
4 3		
0 1		
9 12		
6 4		
5 4		
0 2		
11 12		
9 10		
0 6		
7 8		
9 11		
5 3		

# Graph – Build Adjacency list

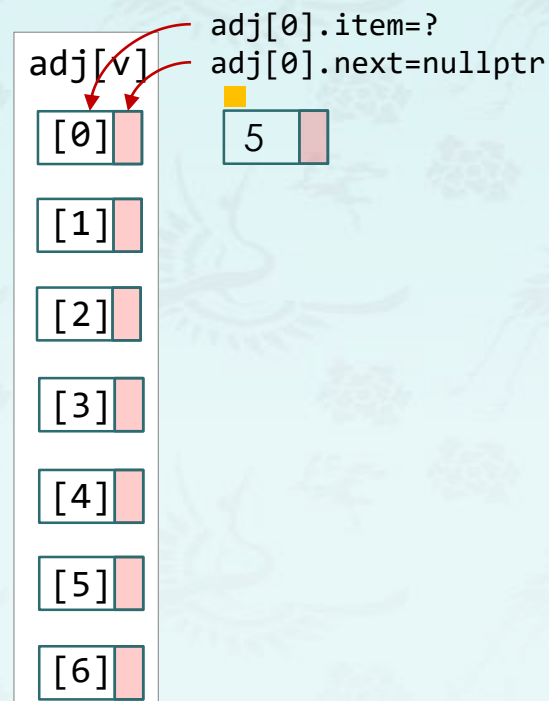
```
void addEdgeFromTo(graph g, int v, int w) {  
    gnode node = new Gnode(w, g->adj[v].next);  
    g->adj[v].next = node;  
    g->E++;  
}
```



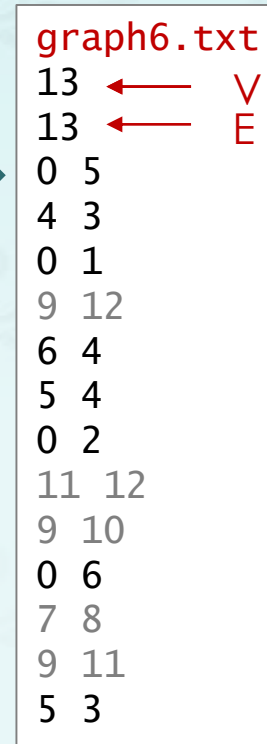
Graph g



Adjacency lists



Edge list

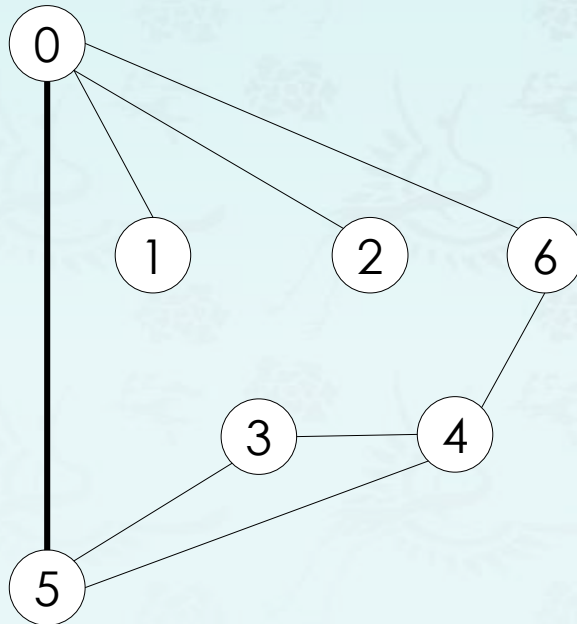


# Graph – Build Adjacency list

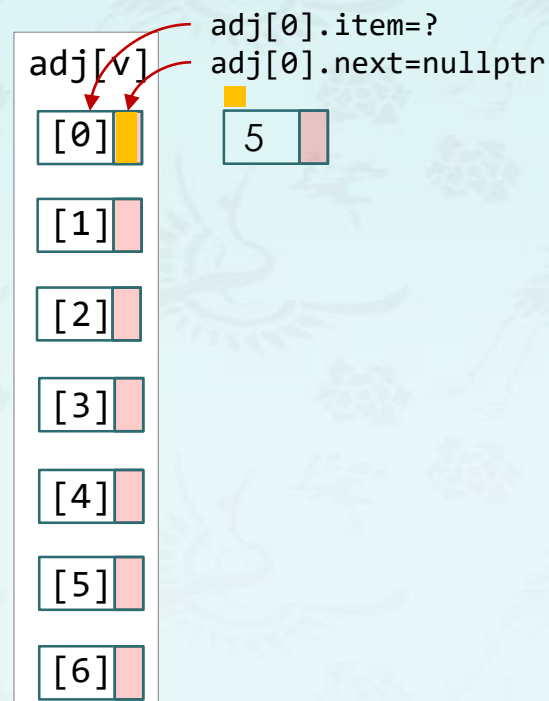
```
void addEdgeFromTo(graph g, int v, int w) {  
    gnode node = new Gnode(w, g->adj[v].next);  
    g->adj[v].next = node;  
    g->E++;  
}
```



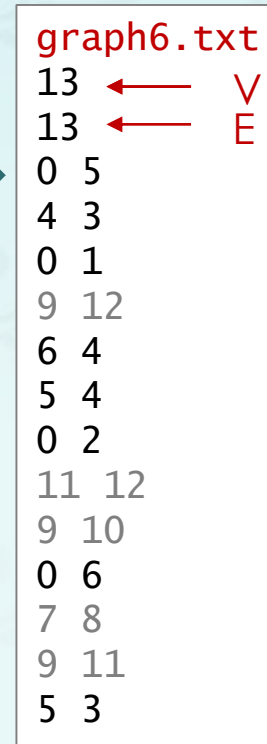
Graph g



Adjacency lists

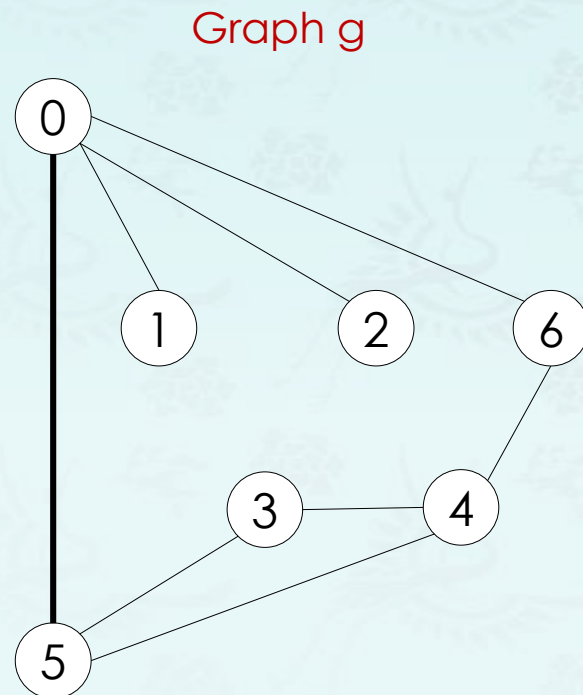


Edge list



# Graph – Build Adjacency list

```
void addEdgeFromTo(graph g, int v, int w) {  
    gnode node = new Gnode(w, g->adj[v].next);  
    g->adj[v].next = node;  
    g->E++;  
}
```



Adjacency lists



Edge list

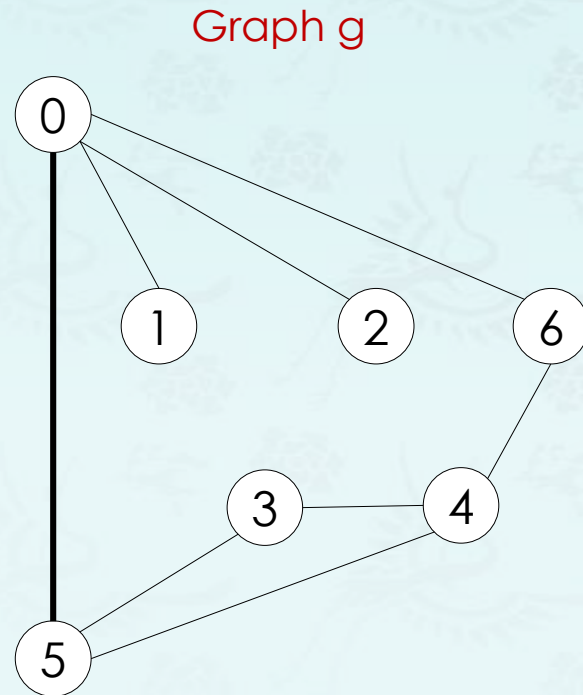
graph6.txt

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

```
void addEdgeFromTo(graph g, int v, int w) {  
    gnode node = new Gnode(w, g->adj[v].next);  
    g->adj[v].next = node;  
    g->E++;  
}
```



Adjacency lists



Edge list

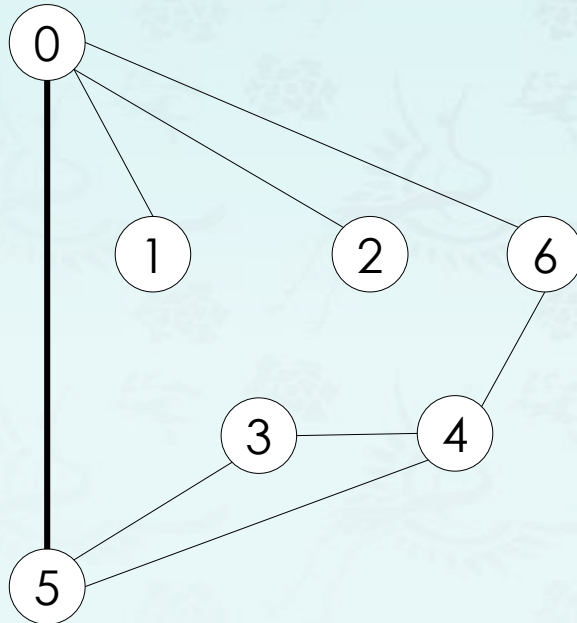
graph6.txt

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

```
void addEdgeFromTo(graph g, int v, int w) {  
    gnode node = new Gnode(w, g->adj[v].next);  
    g->adj[v].next = node;  
    g->E++;  
}
```

Graph g



Adjacency lists

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

Edge list

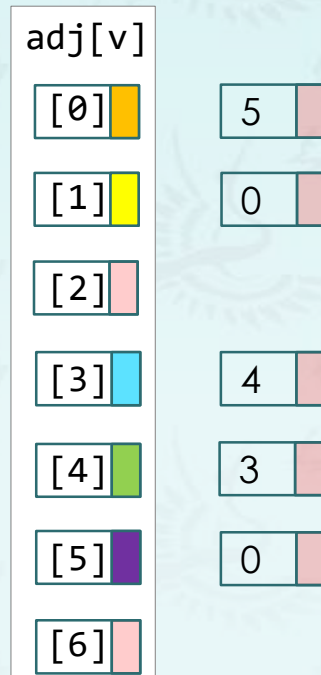
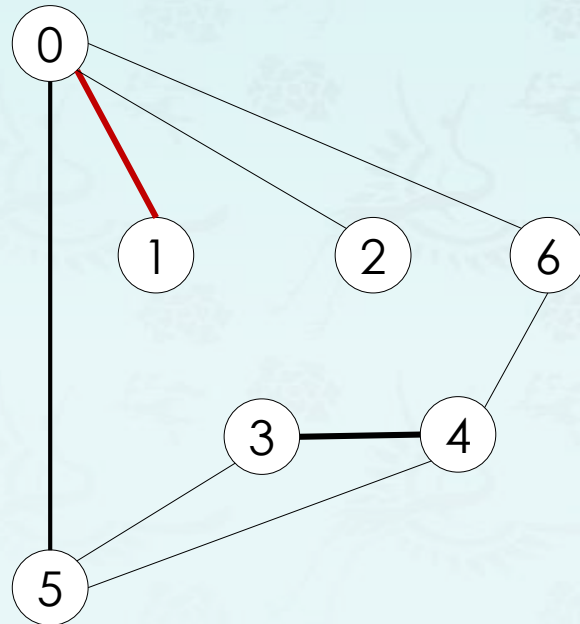
graph6.txt

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

```
void addEdgeFromTo(graph g, int v, int w) {
    gnode node = new Gnode(w, g->adj[v].next);
    g->adj[v].next = node;
    g->E++;
}
```

Graph g



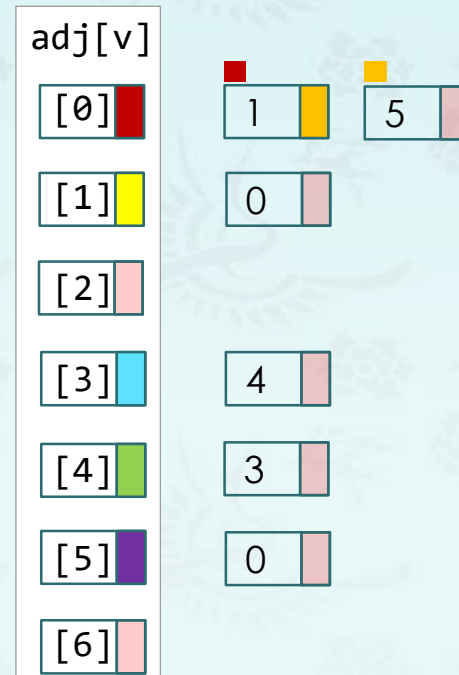
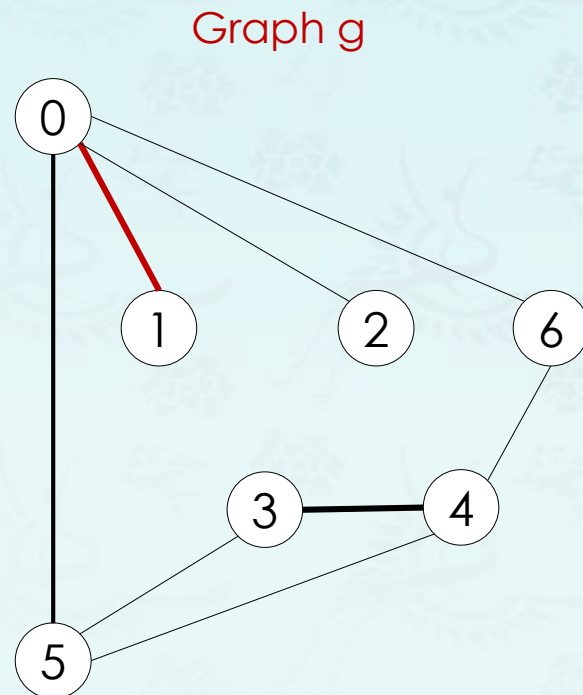
Edge list

graph6.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

```
void addEdgeFromTo(graph g, int v, int w) {  
    gnode node = new Gnode(w, g->adj[v].next);  
    g->adj[v].next = node;  
    g->E++;  
}
```



Edge list

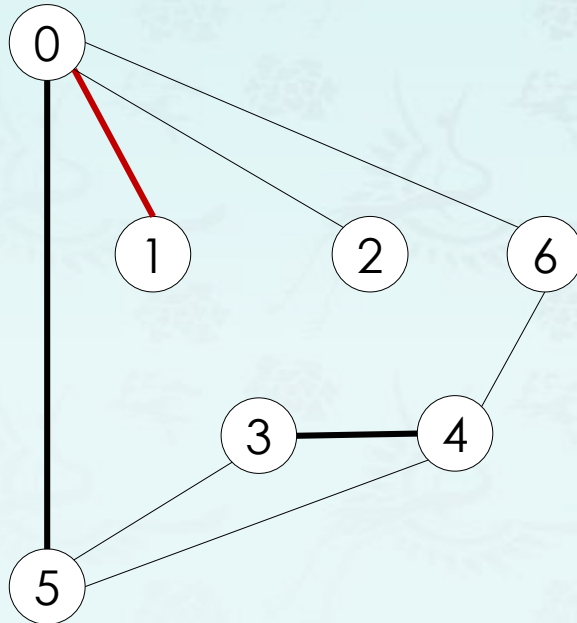
graph6.txt

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

```
void addEdgeFromTo(graph g, int v, int w) {  
    gnode node = new Gnode(w, g->adj[v].next);  
    g->adj[v].next = node;  
    g->E++;  
}
```

Graph g



Adjacency lists

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

Edge list

graph6.txt

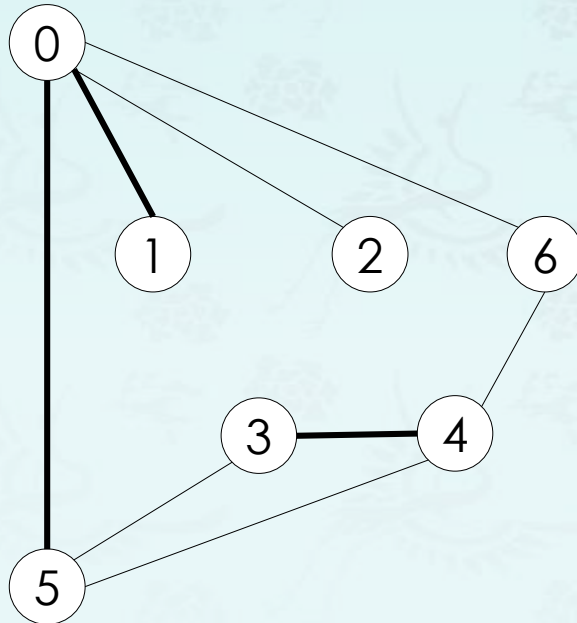
13	← V
13	← E
0 5	
4 3	
0 1	
9 12	
6 4	
5 4	
0 2	
11 12	
9 10	
0 6	
7 8	
9 11	
5 3	



# Graph – Build Adjacency list

```
void addEdgeFromTo(graph g, int v, int w) {  
    gnode node = new Gnode(w, g->adj[v].next);  
    g->adj[v].next = node;  
    g->E++;  
}
```

Graph g



Adjacency lists

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

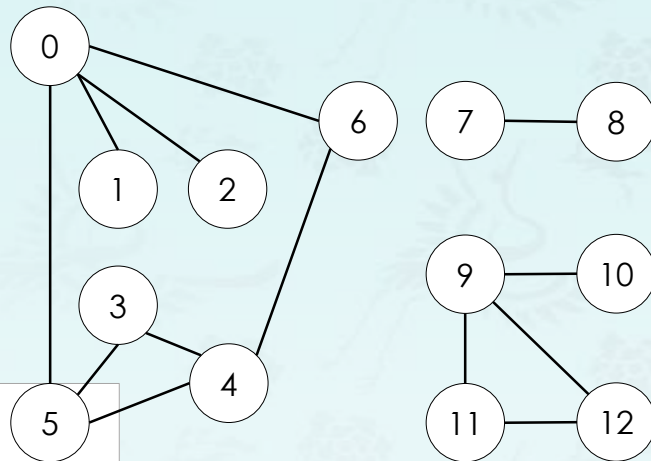
Edge list

graph6.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	

# Warming-up: degree()

```
struct Graph {  
    int V, E;  
    gnode adj;  
    Graph(int v = 0) {  
        V = v;  
        E = 0;  
        adj = new (nothrow) Gnode[v];  
        assert(adj != nullptr);  
  
        for (int i=0; i<v; i++)  
            adj[i].next = nullptr;  
    }  
    ~Graph() {}  
};  
using graph = Graph *;
```



Adjacency lists

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

Edge list

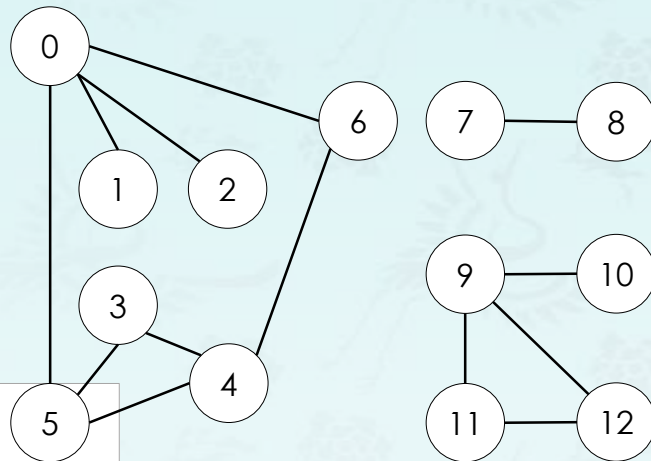
graph6.txt

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	

# Warming-up: degree()

```
int degree(graph g, int v) {  
    int deg = 0;  
    for (gnode w = g->adj[v].next; w != nullptr; w = w->next)  
        deg++;  
    return deg;  
}
```

```
struct Graph {  
    int V, E;  
    gnode adj;  
    Graph(int v = 0) {  
        V = v;  
        E = 0;  
        adj = new (nothrow) Gnode[v];  
        assert(adj != nullptr);  
  
        for (int i=0; i<v; i++)  
            adj[i].next = nullptr;  
    }  
    ~Graph() {}  
};  
using graph = Graph *;
```



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

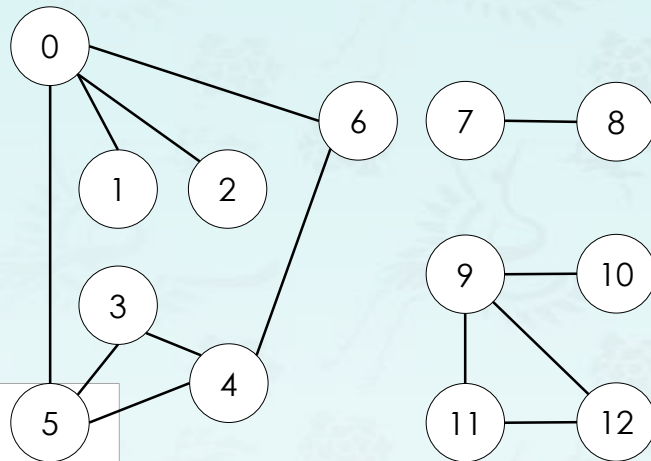
Edge list

graph6.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	

# Warming-up: degree()

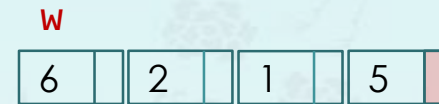
```
int degree(graph g) {  
    int deg = 0;  
    for (int v = 0; v < V(g); v++)  
        deg = max(degree(g, v), deg);  
    return deg;  
}
```



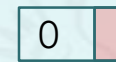
```
struct Graph {  
    int V, E;  
    gnode adj;  
    Graph(int v = 0) {  
        V = v;  
        E = 0;  
        adj = new (nothrow) Gnode[v];  
        assert(adj != nullptr);  
  
        for (int i=0; i<v; i++)  
            adj[i].next = nullptr;  
    }  
    ~Graph() {}  
};  
using graph = Graph *;
```

adj[v]

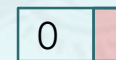
[0]



[1]



[2]



[3]



[4]



[5]



[6]



Edge list

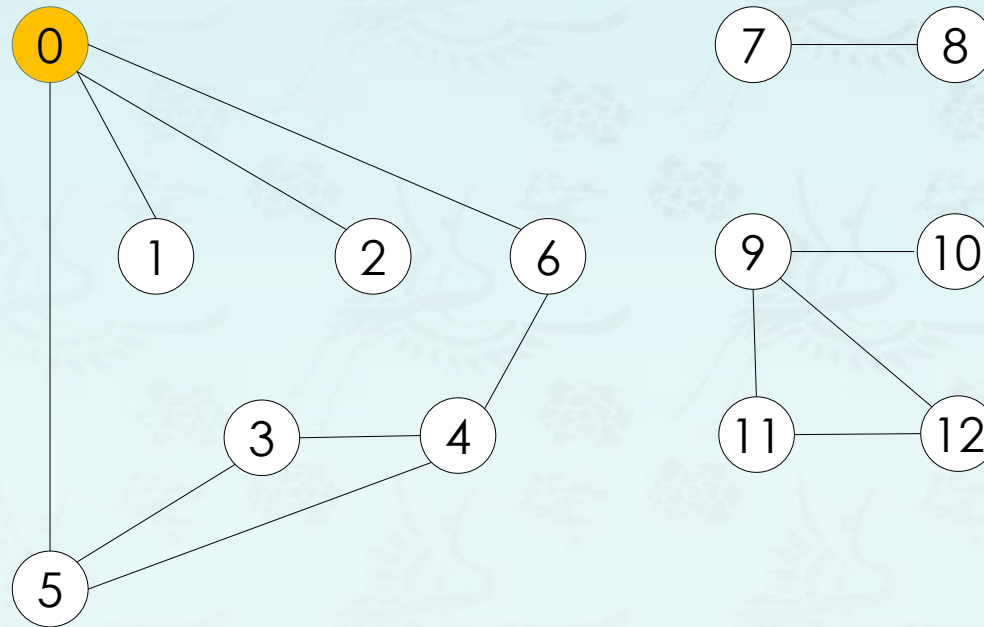
graph6.txt

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

# DFS: Depth-First Search Demo

## To visit a vertex $v$ :

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

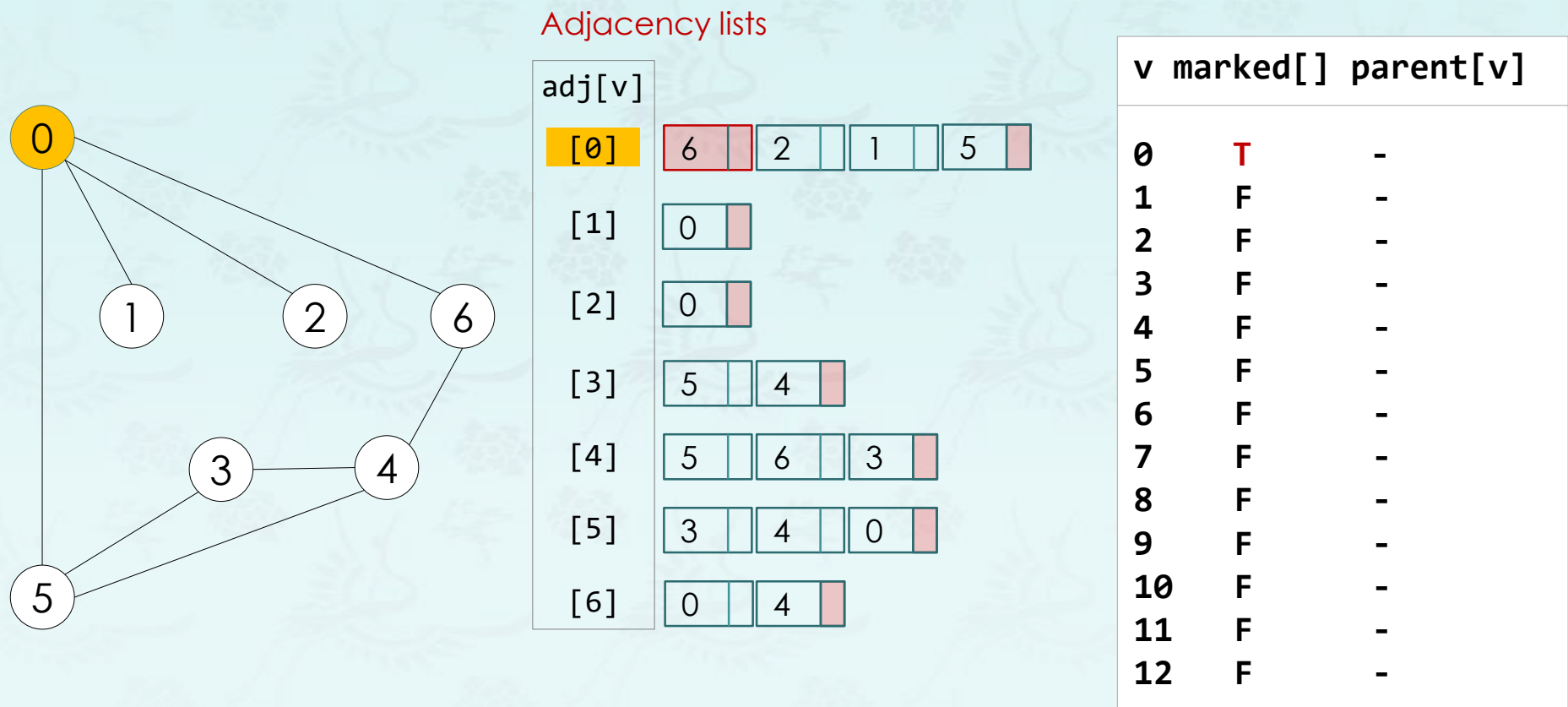


v	marked[]	parent[v]
0	<b>T</b>	-
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

**Which one first?** visit 0:



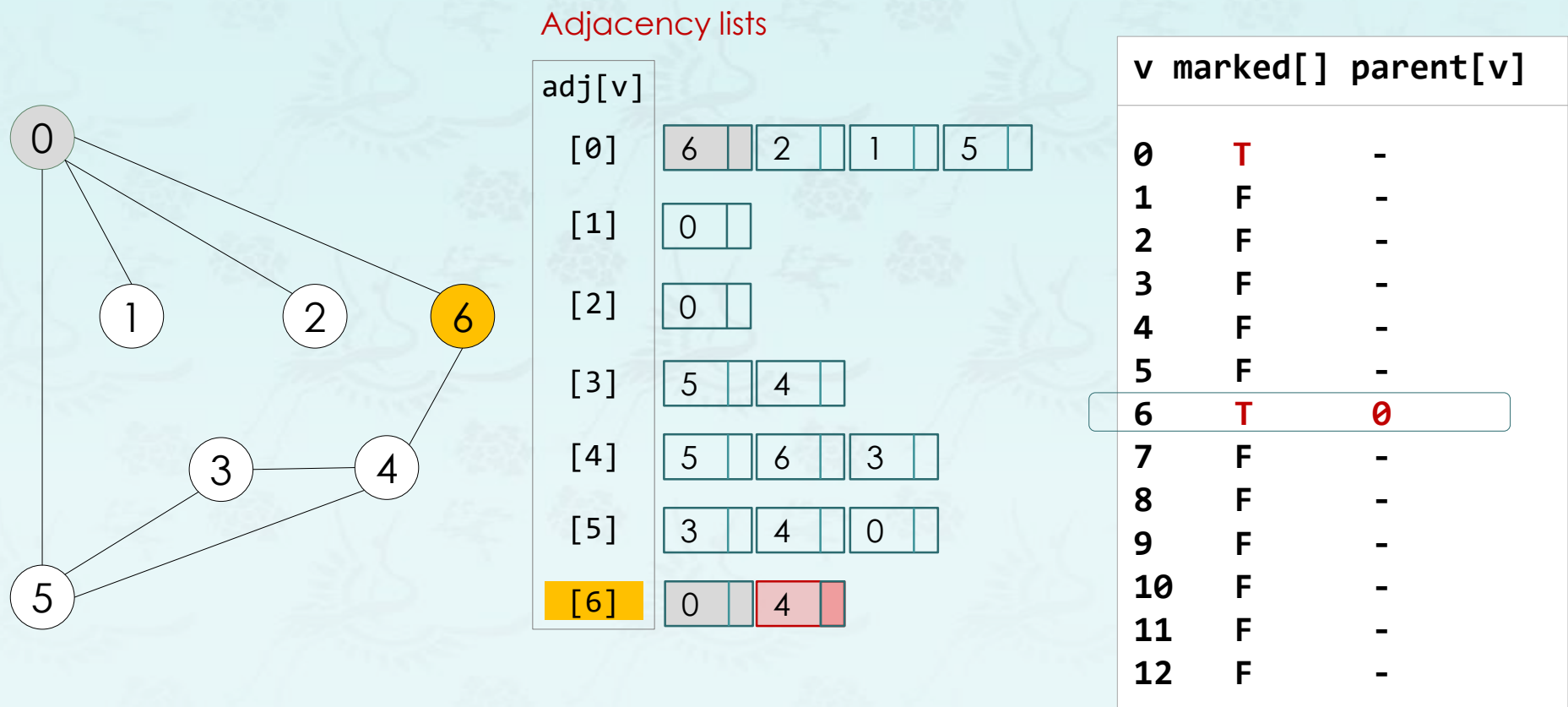
# DFS: Depth-First Search Demo



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

DFS 0

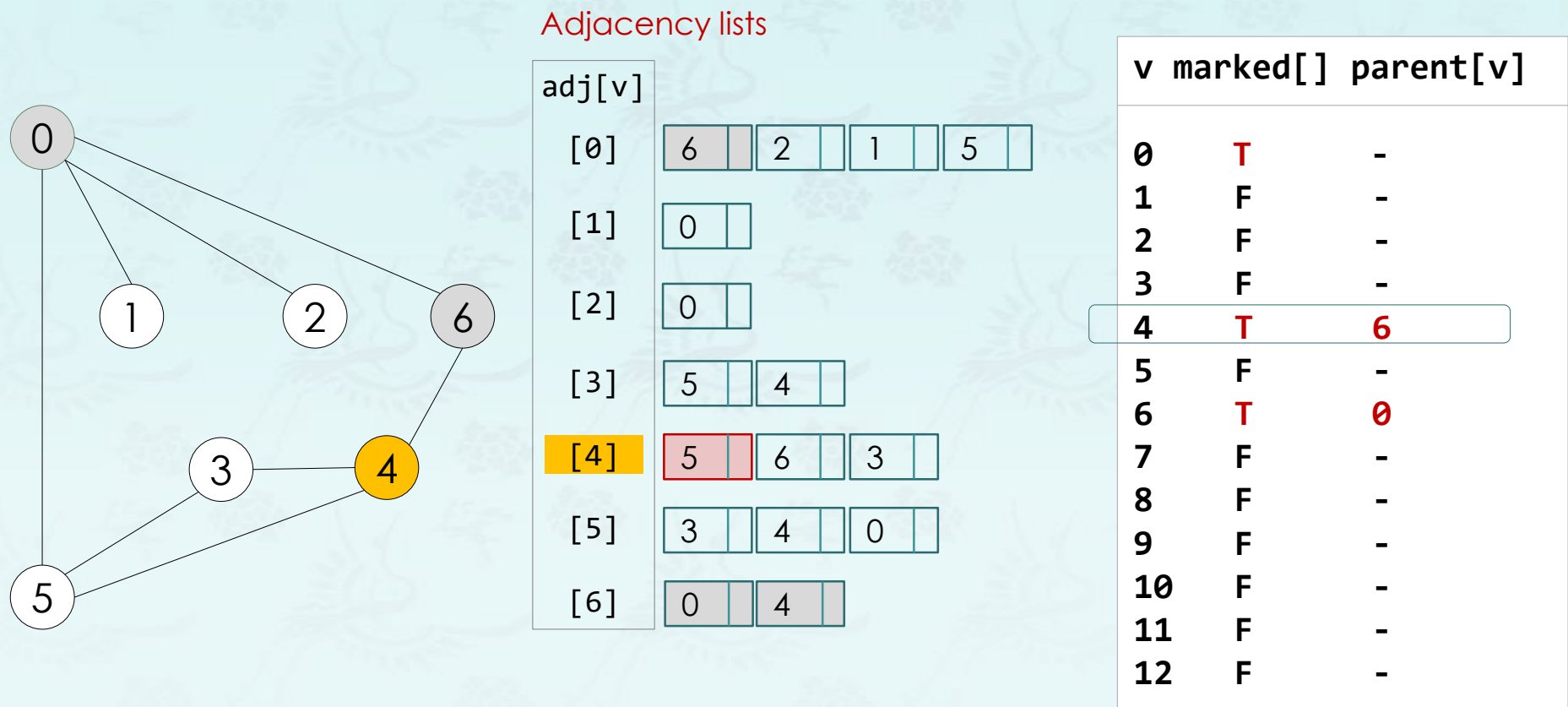
# DFS: Depth-First Search Demo



visit 6: check 0, check 4

DFS 0 6

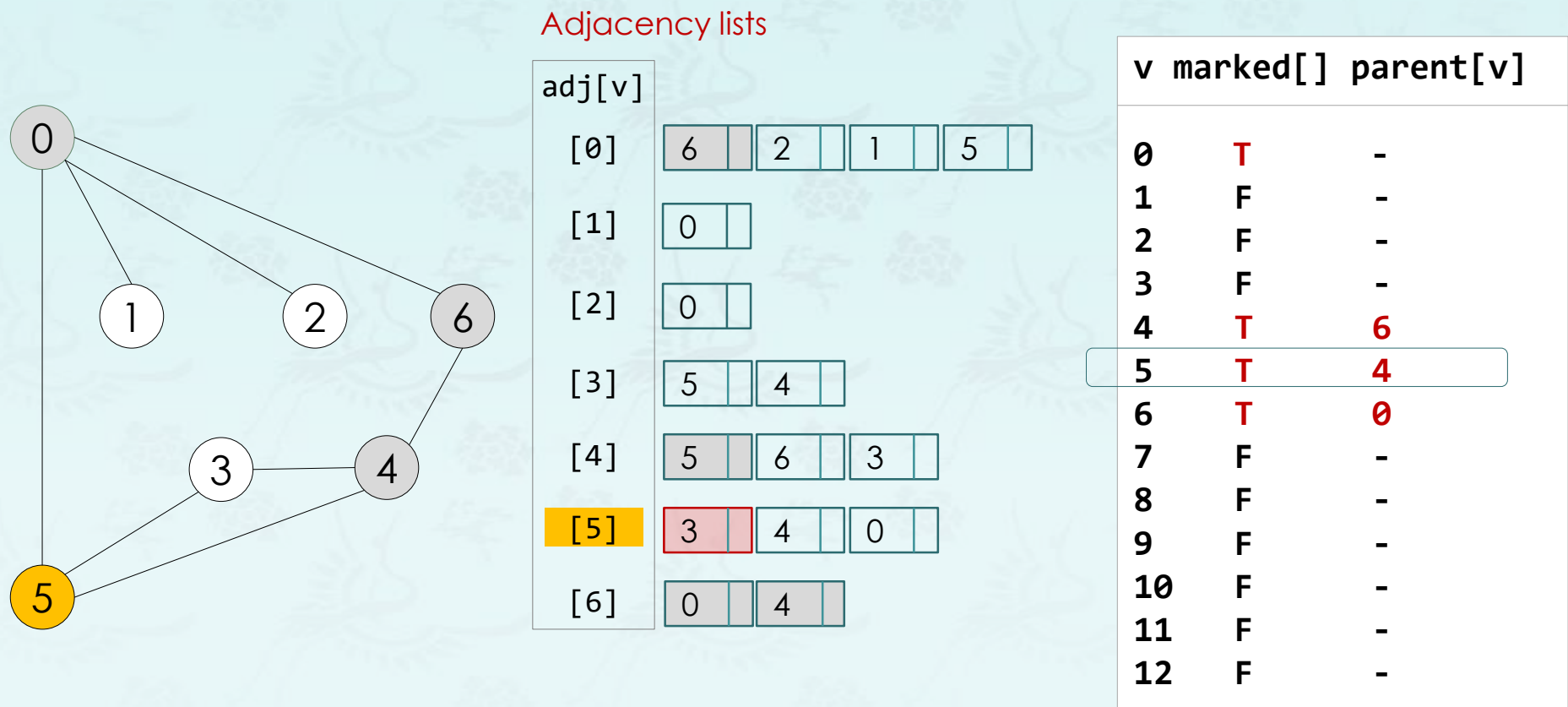
# DFS: Depth-First Search Demo



visit 4: check 5, check 6, check 3

DFS 0 6 4

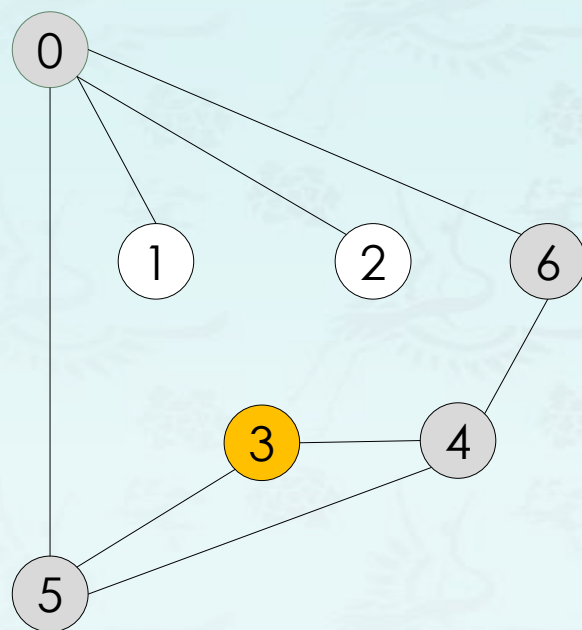
# DFS: Depth-First Search Demo



visit 5: check 3, check 4, check 0

DFS 0 6 4 5

# DFS: Depth-First Search Demo



Adjacency lists

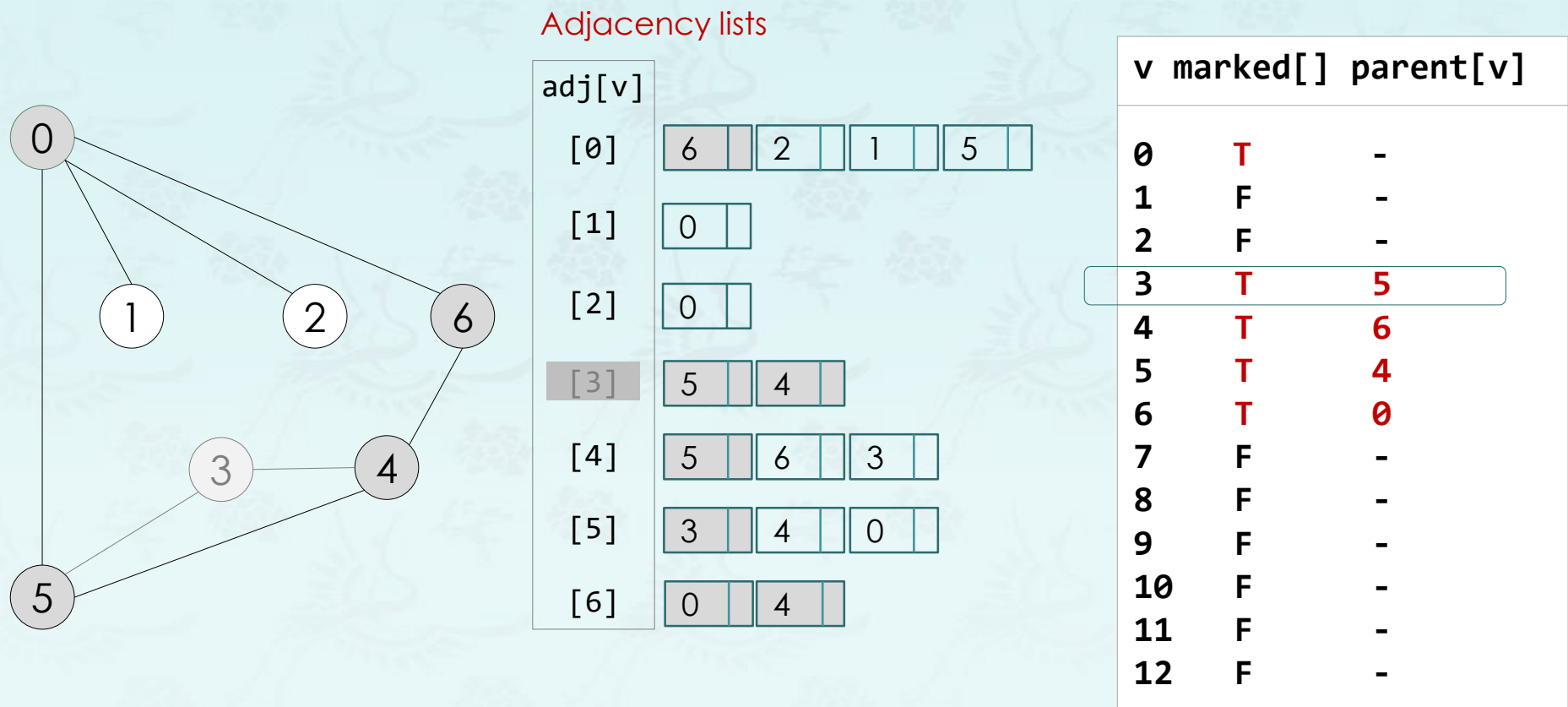
adj[v]	
[0]	6 2 1 5
[1]	0
[2]	0
[3]	5 4
[4]	5 6 3
[5]	3 4 0
[6]	0 4

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

# DFS: Depth-First Search Demo

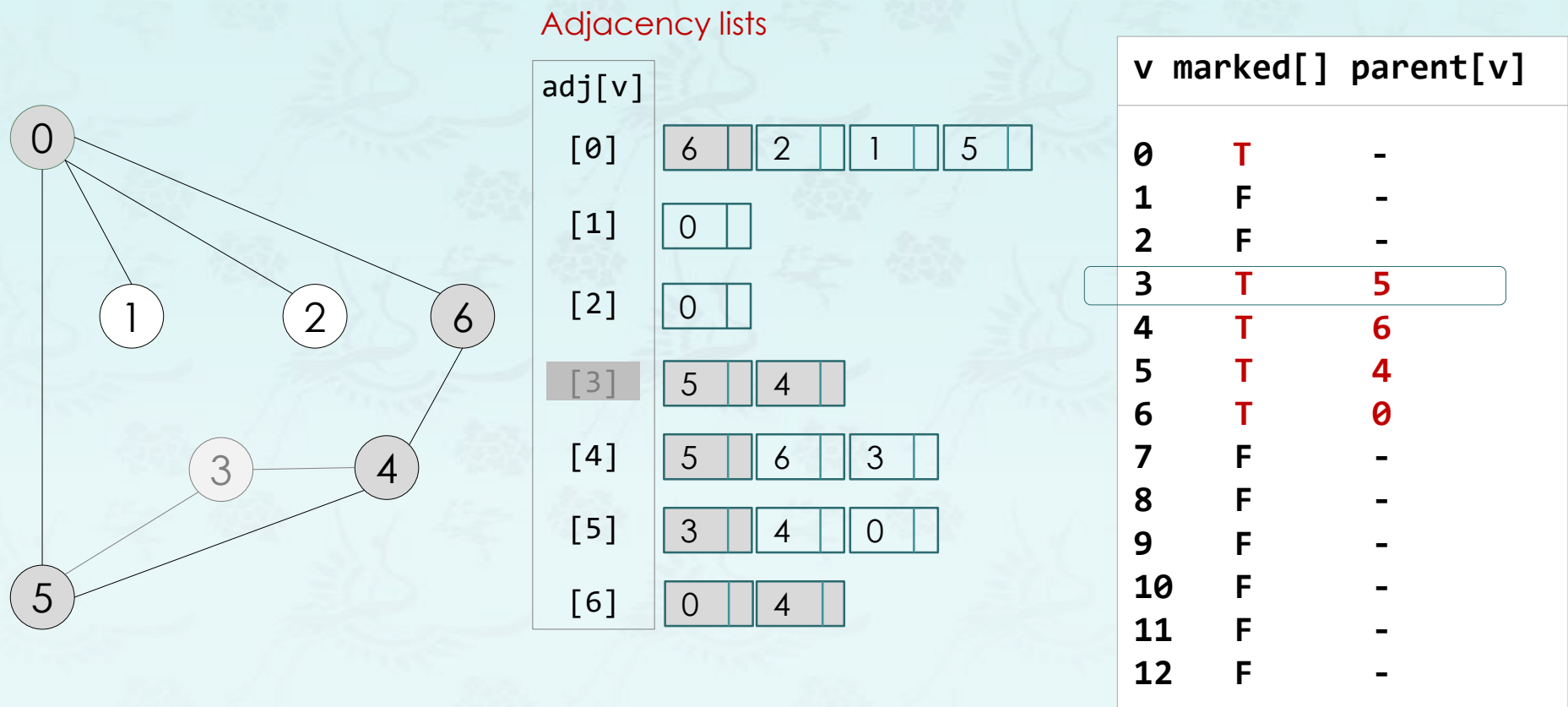


visit 3: check 5, check 4

3 done:

DFS 0 6 4 5 3

# DFS: Depth-First Search Demo



visit 3: check 5, check 4

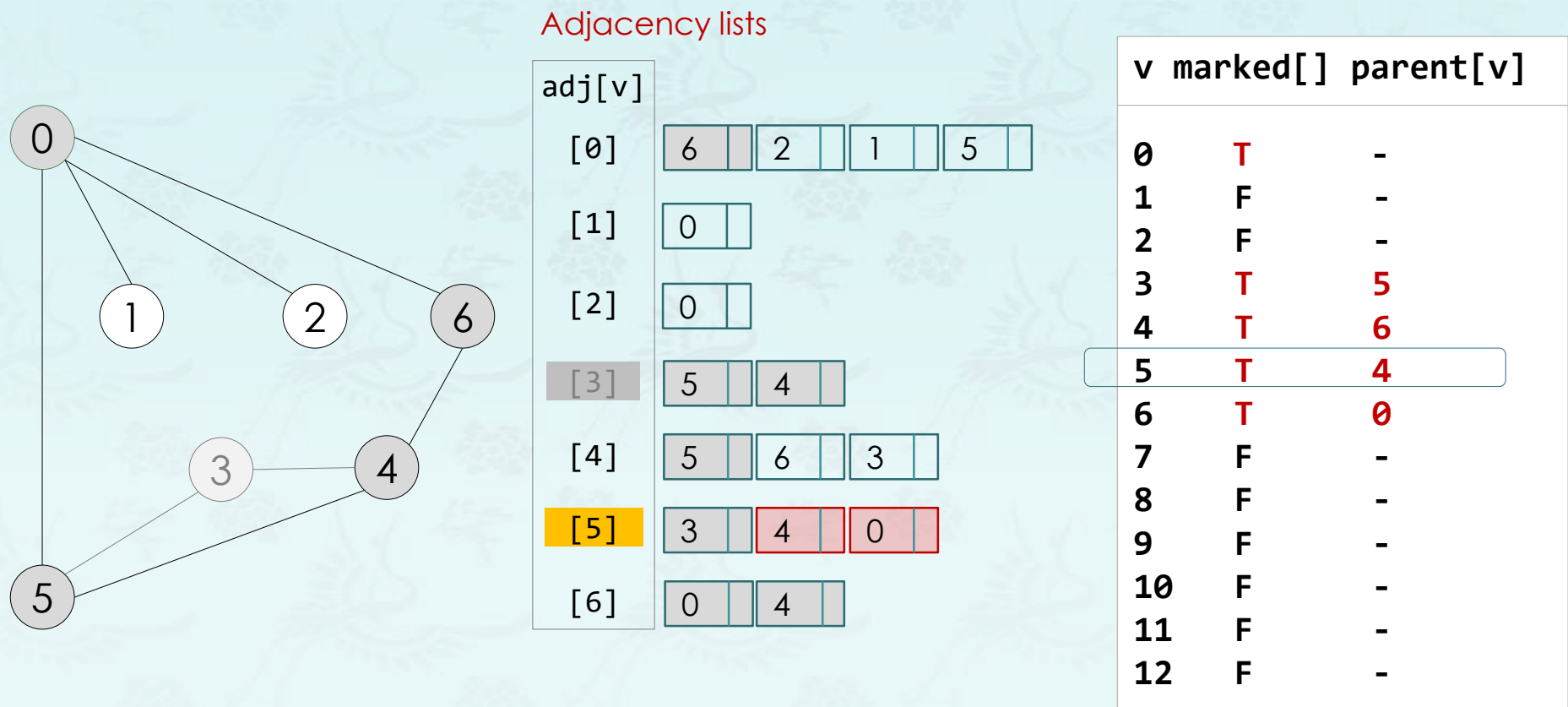
3 done:

DFS 0 6 4 5 3

What's next? **Backtrack!** How?  
Use parent[v] → **parent[3] = 5**



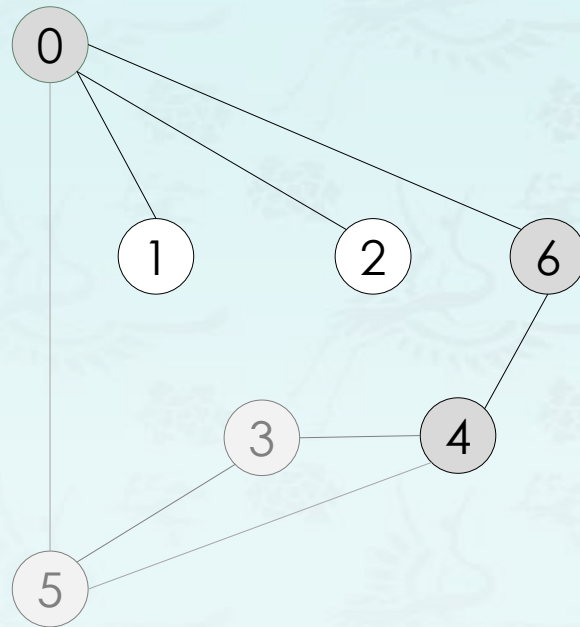
# DFS: Depth-First Search Demo



visit 5: check 3, check 4, check 0

DFS 0 6 4 5 3

# DFS: Depth-First Search Demo



Adjacency lists

adj[v]	
[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	-

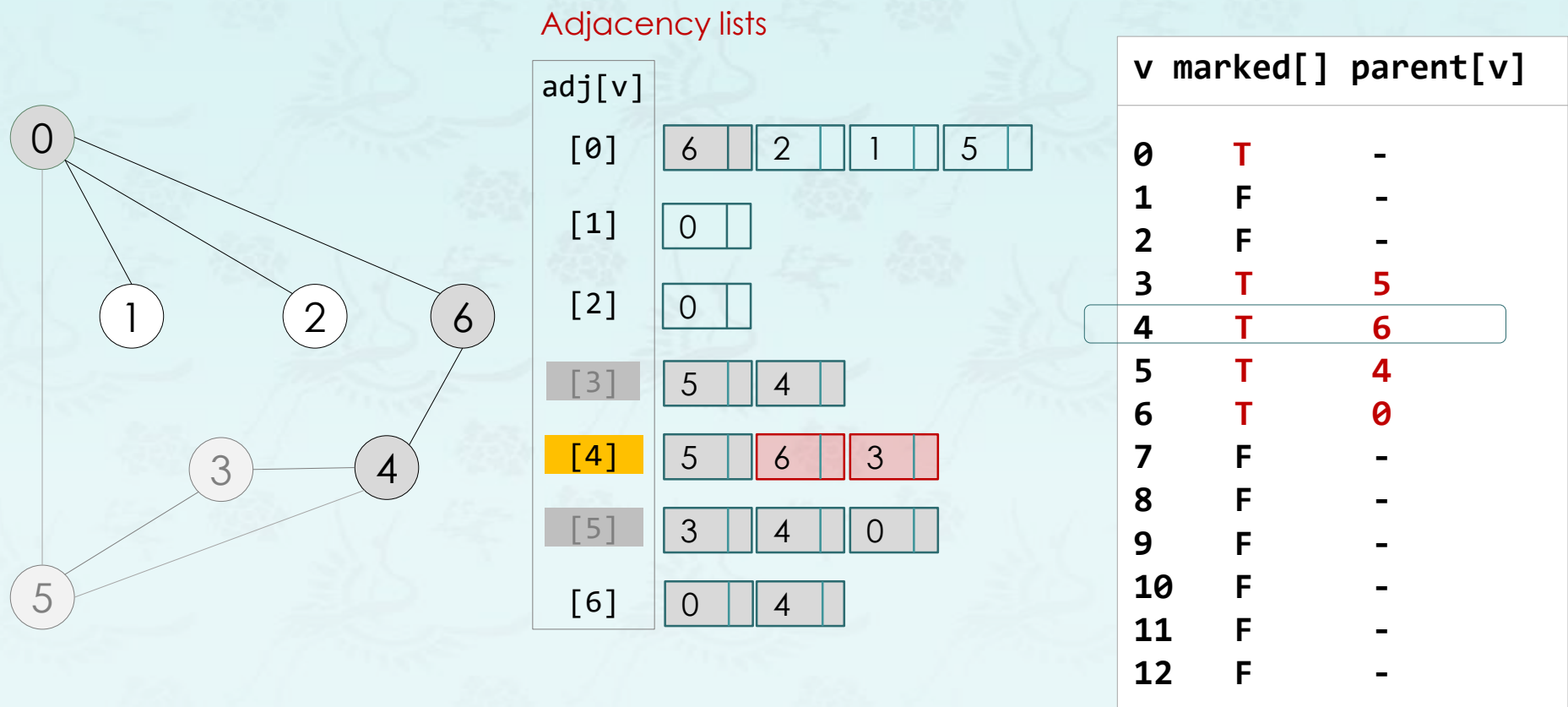
visit 5: check 3, check 4, check 0

5 done:

DFS 0 6 4 5 3

What's next? **Backtrack!** How to?  
Use parent[v] → **parent[5] = 4**

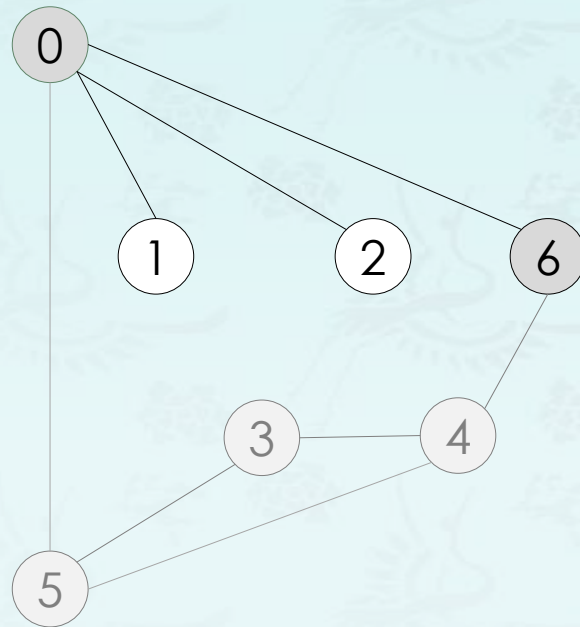
# DFS: Depth-First Search Demo



visit 4: check 5, check 6, check 3

DFS 0 6 4 5 3

# DFS: Depth-First Search Demo



Adjacency lists

adj[v]	
[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	-

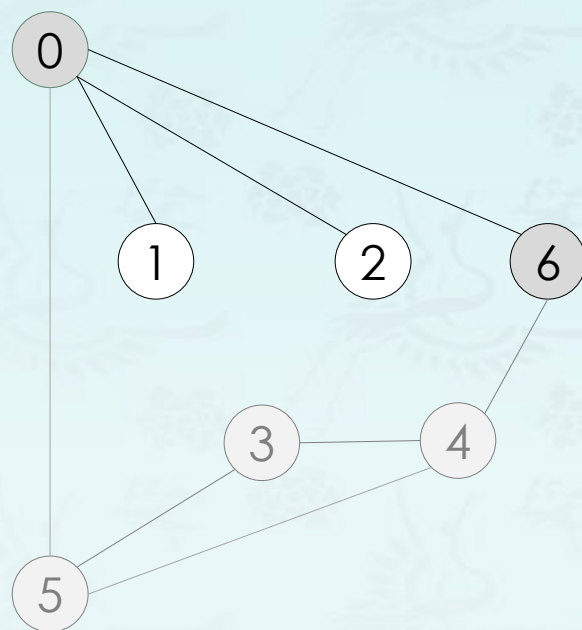
visit 4: check 5, check 6, check 3

4 done:

DFS 0 6 4 5 3

What's next? **Backtrack!** How to?  
Use parent[v] → **parent[4] = 6**

# DFS: Depth-First Search Demo



Adjacency lists

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

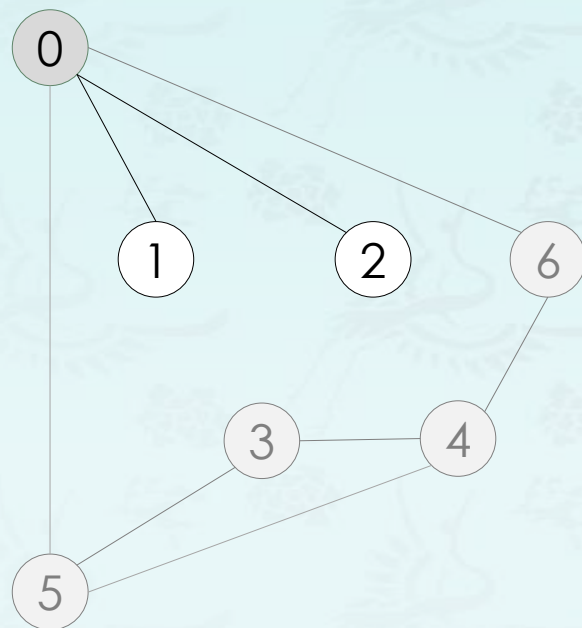
v marked[] parent[v]

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

DFS 0 6 4 5 3

# DFS: Depth-First Search Demo



Adjacency lists

adj[v]	
[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	-

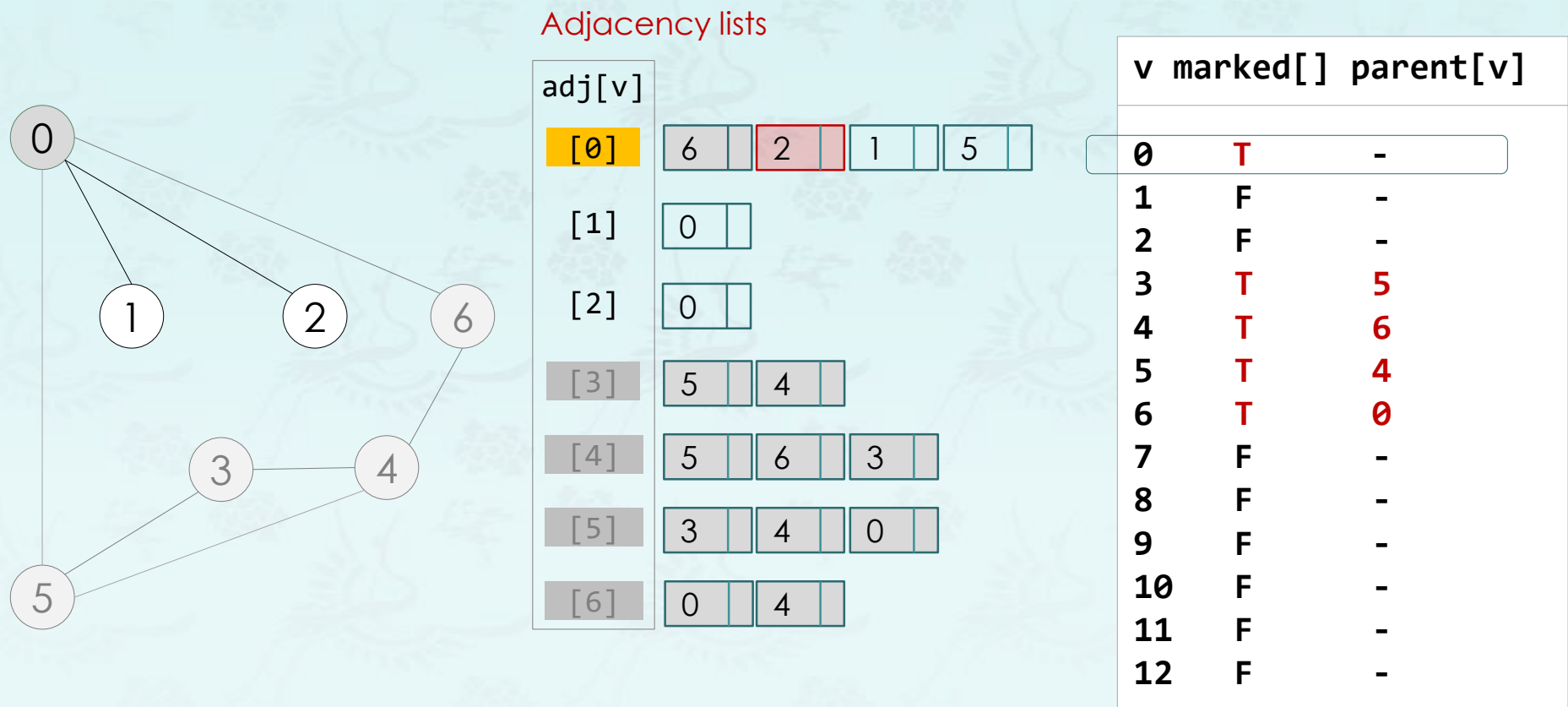
visit 6: check 0, check 4

6 done:

DFS 0 6 4 5 3

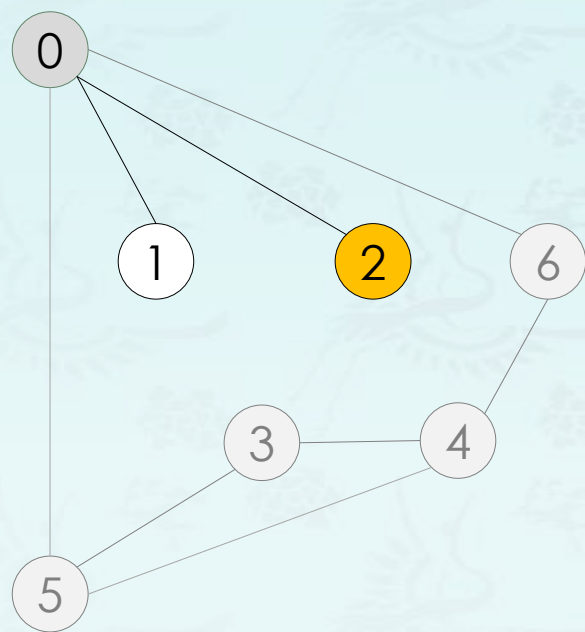
What's next? Backtrack!  
parent[6] = 0

# DFS: Depth-First Search Demo





# DFS: Depth-First Search Demo



Adjacency lists

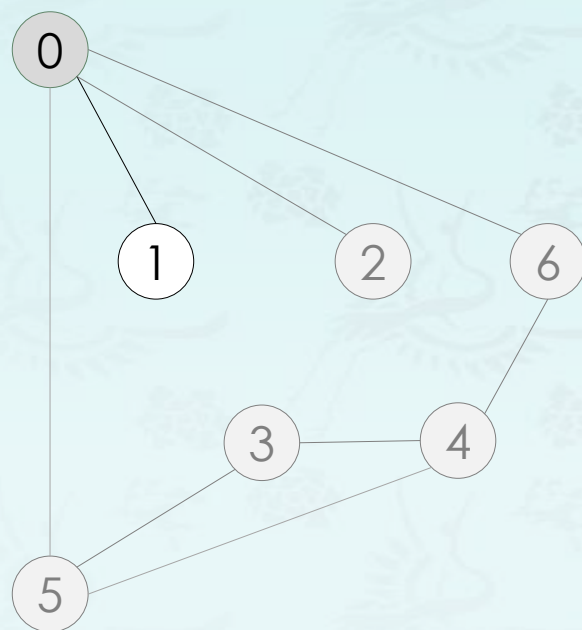
adj[v]	
[0]	6 2 1 5
[1]	0
[2]	0
[3]	5 4
[4]	5 6 3
[5]	3 4 0
[6]	0 4

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

# DFS: Depth-First Search Demo



Adjacency lists

adj[v]	
[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	-

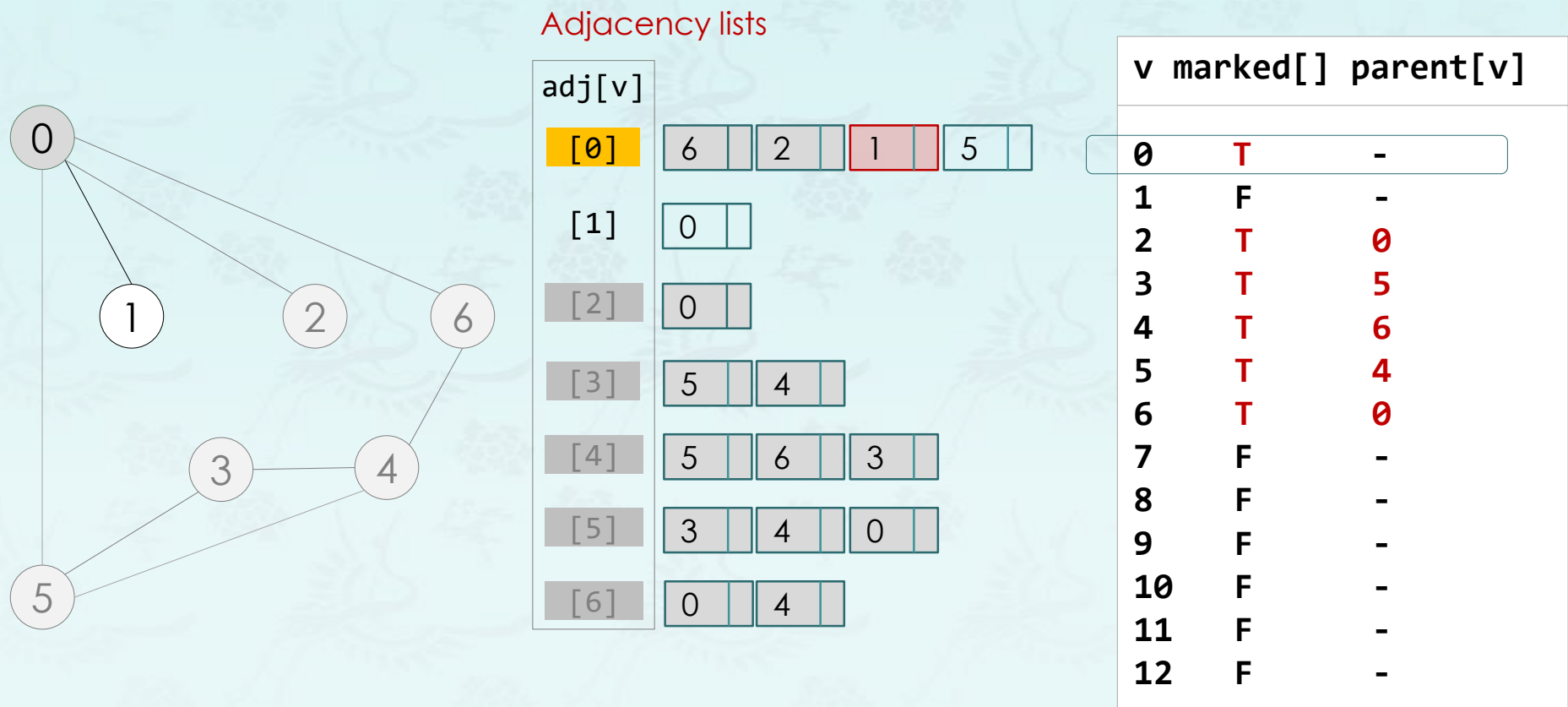
visit 2: check 0

DFS 0 6 4 5 3 2

2 done:

What's next? Backtrack!  
parent[2] = 0

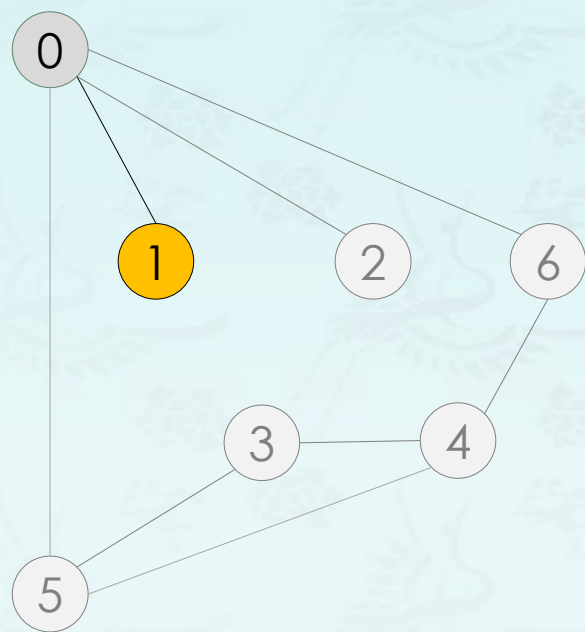
# DFS: Depth-First Search Demo



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

**DFS 0 6 4 5 3 2**

# DFS: Depth-First Search Demo



Adjacency lists

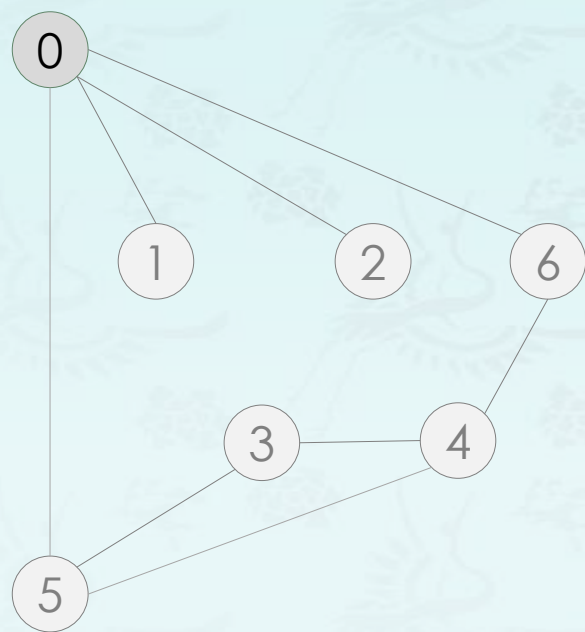
adj[v]	
[0]	6 2 1 5
[1]	0
[2]	0
[3]	5 4
[4]	5 6 3
[5]	3 4 0
[6]	0 4

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

DFS 0 6 4 5 3 2 1

# DFS: Depth-First Search Demo



Adjacency lists

adj[v]	
[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

DFS 0 6 4 5 3 2 1

1 done:

What's next? Backtrack!  
parent[1] = 0

# DFS: Depth-First Search Demo



visit 1: check 6, check 2, check 1, check 5

DFS 0 6 4 5 3 2 1

# DFS: Depth-First Search Demo



visit 1: check 6, check 2, check 1, check 5

0 done:

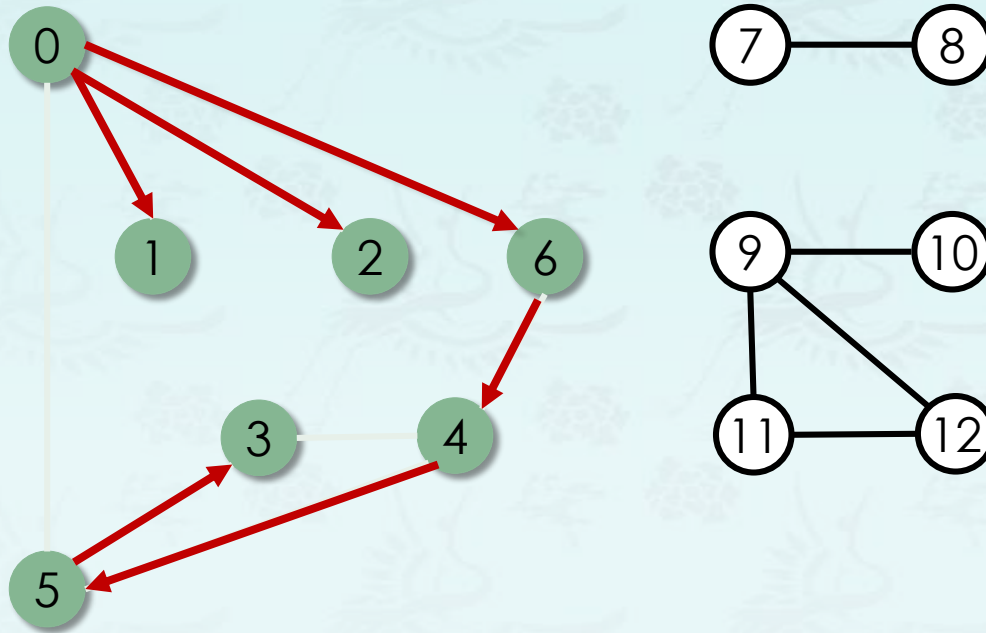
DFS 0 6 4 5 3 2 1



# DFS: Depth-First Search Demo

## To visit a vertex $v$ :

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



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

- found vertices reachable from 0
- build a data structure **parent[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 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 vertices.
- **int[] parent** to keep tree of paths.  
( $\text{parent}[w] == v$ ) means that edge  $v$ - $w$  taken to visit  $w$  for first time

# DFS: Depth-First Search Coding

```
// runs DFS for at vertex v recursively.
// Only que, g->marked[v] and g->parentDFS[] are updated here.
void DFS(graph g, int v, queue<int>& que) {
    g->marked[v] = true;           // visited
    que.push(v);                  // save the path

    for (gnode w = g->adj[v].next; w; w = w->next) {
        cout << "your code here (recursion) \n";
        - only for unmarked vertex, do the followings:
        - invoke DFS(g, ..., ...);
        - set g->parentDFS[...] = ...;
    }
}
```

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

# DFS: Depth-First Search Coding

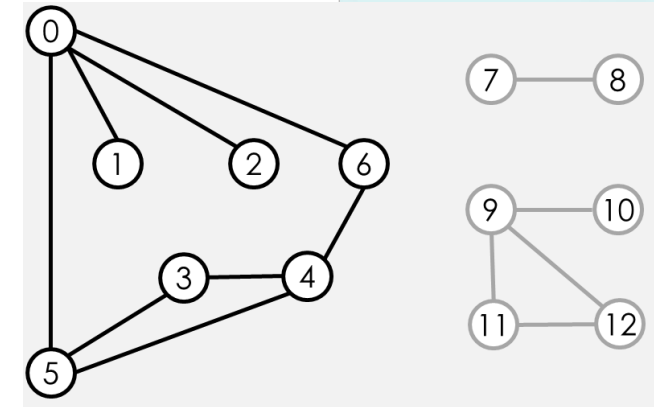
```
// runs DFS for all components and produces DFS0[], CCID[] & parentDFS[].
```

```
void DFS_CCs(graph g) {  
    if (empty(g)) return;  
    for (int i = 0; i < V(g); i++) {  
        g->marked[i] = false;  
        g->parentDFS[i] = -1;  
        g->CCID[i] = 0;  
    }  
}
```

```
queue<int> que;
```

```
DFS(g, 0, que);  
setDFS0(g, 0, que);
```

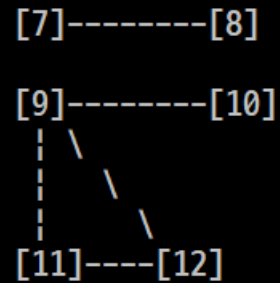
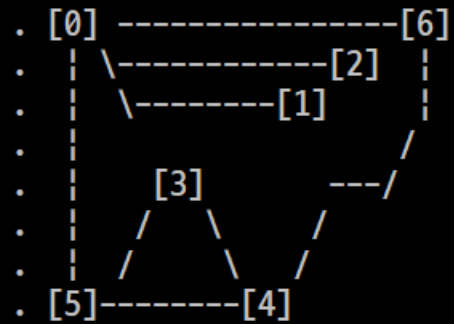
It works only for the 1<sup>st</sup> CC at 0.  
Make it work for all CC's.



```
g->DFSv = {};  
}
```

```
// runs DFS for at vertex v recursively.  
// Only que, g->marked[v] and g->parentDFS[] are updated here.  
void DFS(graph g, int v, queue<int>& que) {  
    g->marked[v] = true;           // visited  
    que.push(v);                  // save the path  
    for (gnode w = g->adj[v].next; w; w = w->next) {  
        cout << "your code here (recursion) \n";  
    }  
}
```

# DFS: Depth-First Search Coding



```

vertex[0..12] =  0  1  2  3  4  5  6  7  8  9 10 11 12
color[0..12] =  0  0  0  0  0  0  0  0  0  0  0  0  0

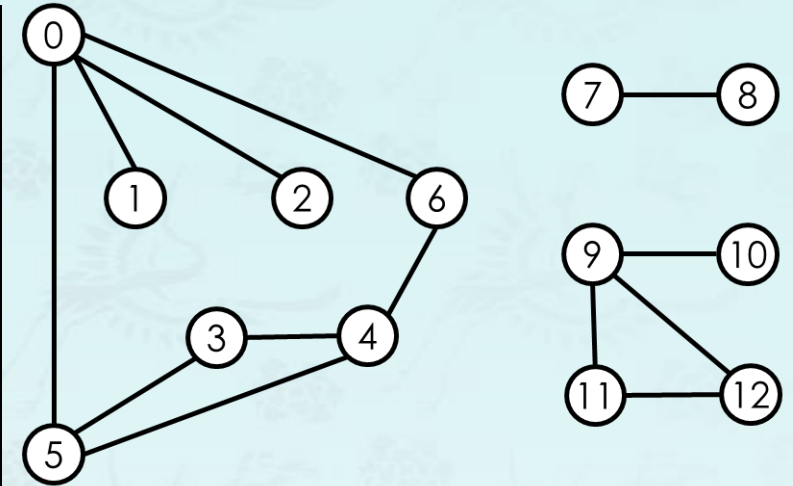
DFS0[0..12] =  0  6  4  5  3  2  1  7  8  9 11 12 10
CCID[0..12] =  1  1  1  1  1  1  1  2  2  3  3  3  3
DFS parent[0..12] = -1  0  0  5  6  4  0 -1  7 -1  9  9 11
BFS0[0..12] =  0  6  2  1  5  4  3  7  8  9 11 10 12
DistTo[0..12] =  0  1  1  2  2  1  1  0  1  0  1  1  1
BFS parent[0..12] = -1  0  0  5  6  0  0 -1  7 -1  9  9  9

```

```

Graph [Graph][Tablet]  file:graph3.txt V:13 E:26 CCs:3 Deg:4
n - new graph file      x - connected(v,w)
d - DFS(v=0)            e - distance(v,w)
b - BFS(v=0)            p - path(v,w)
c - cyclic(v=0)?        m - print mode[adjList/graph]
t - bigraph(v=0)?       a - bigraph using adj-list coloring
Command(q to quit):

```



## 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	-

# 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**.

```
// returns a path from v to w using the DFS result or parentDFS[].
// 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;
    for (int i = 0; i < V(g); i++) {
        g->marked[i] = false;
        g->parentDFS[i] = -1;
    }
    queue<int> q;
    DFS(g, v, q);    // DFS at v, starting vertex
    g->DFSv = q;      // DFS result at v
    path = {};
    cout << "your code here\n"; // push v to w path to the stack path
}
```

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	-

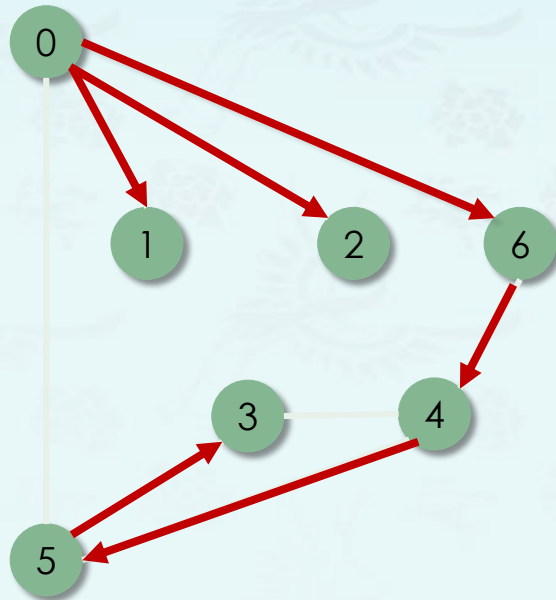


# DFS: Depth-First Search Properties

- **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**.

- What is the path from vertex 0 to vertex 3?
- In this case, what is in the stack when `parent[]` returns?

```
void DFSpath(graph g, int v, int w, stack<int>& path)
```

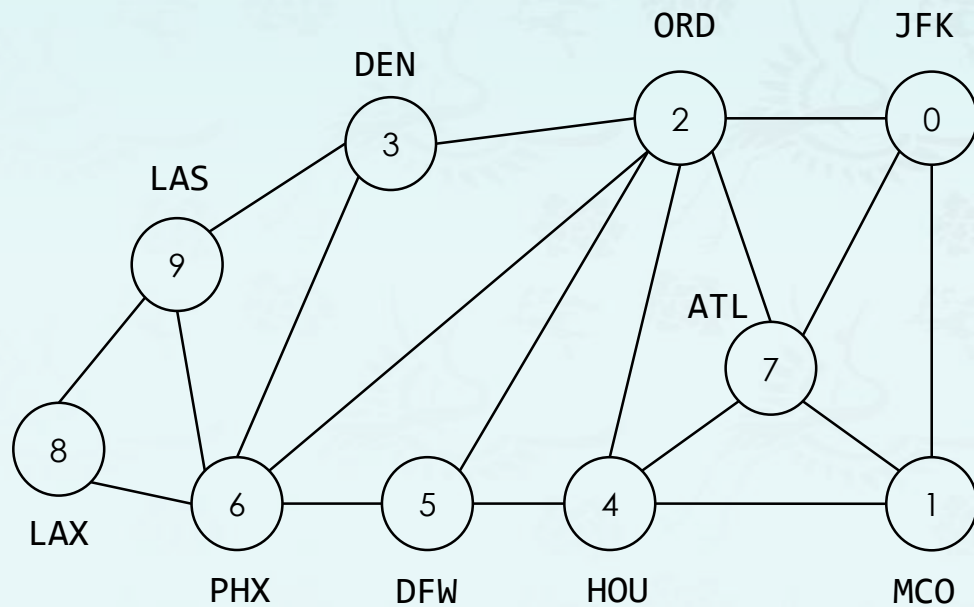


	v	marked[]	parent[v]
v →	0	T	-
	1	T	0
	2	T	0
	3	T	5
w →	4	T	6
	5	T	4
	6	T	0
	7	F	-
	8	F	-
	9	F	-
	10	F	-
	11	F	-
	12	F	-

# DFS/BFS – Exercise

To visit a vertex  $v$ :

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



Graph  $g$ :

routes.txt  
edge list

```
10
18
0 1
2 3
2 4
5 6
0 7
0 2
2 5
2 6
```

Adjacency list

```
0: 2 7 1
1: 4 7 0
2: 7 6 5 0 4 3
3: 9 6 2
4: 5 1 7 2
5: 4 2 6
6: 8 9 3 2 5
7: 1 2 4 0
8: 9 6
9: 8 6 3
```

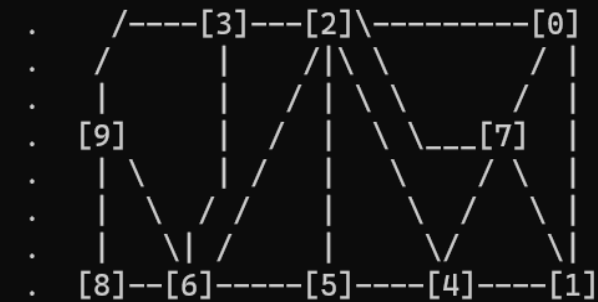
$v$   $w$

## DFS/BFS – Exercise

1. Get familiar with using graphx.exe and using ~.txt graph files provided in pset.
2. Create an **routes.txt** such that it generates the results as shown in the screen capture. Run DFS/BFS and get familiar with the results, its meaning and menu items.

Adjacency-list:

```
V[0]: 2 7 1
V[1]: 4 7 0
V[2]: 7 6 5 0 4 3
V[3]: 9 6 2
V[4]: 5 1 7 2
V[5]: 4 2 6
V[6]: 8 9 3 2 5
V[7]: 1 2 4 0
V[8]: 9 6
V[9]: 8 6 3
```



vertex[0..9] =	0	1	2	3	4	5	6	7	8	9
color[0..9] =	0	0	0	0	0	0	0	0	0	0
DFS0[0..9] =	0	2	7	1	4	5	6	8	9	3
CCID[0..9] =	1	1	1	1	1	1	1	1	1	1
DFS parent[0..9] =	-1	7	0	9	1	4	5	2	6	8
BFS0[0..9] =	0	2	7	1	6	5	4	3	8	9
DistTo[0..9] =	0	1	1	2	2	2	2	1	3	3
BFS parent[0..9] =	-1	0	0	2	2	2	2	0	6	6

Graph [AdjList][Graph][Tablet] file:routes.txt V:10 E:36 CCs

n - new graph file x - connected(v,w)  
d - DFS(v=0) e - distance(v,w)  
b - BFS(v=0) p - path(v,w)  
m - print mode[adjList/graph]  
Command(q to quit):

# Data Structures

## Chapter 7: Graph

1. Introduction
  - Terminology, Representation, ADT
2. Basic Operations
  - **DFS**, CC, BFS, Processing
3. Digraph and Applications
4. Minimum Spanning Tree(MST)