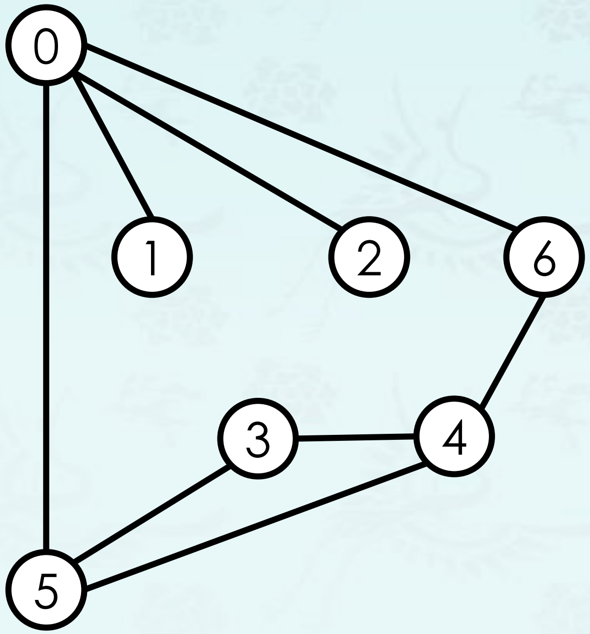
Graph

- Adjacency list processing
- Graph API - Implementation
  - **Cycle**
  - Bipartite

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

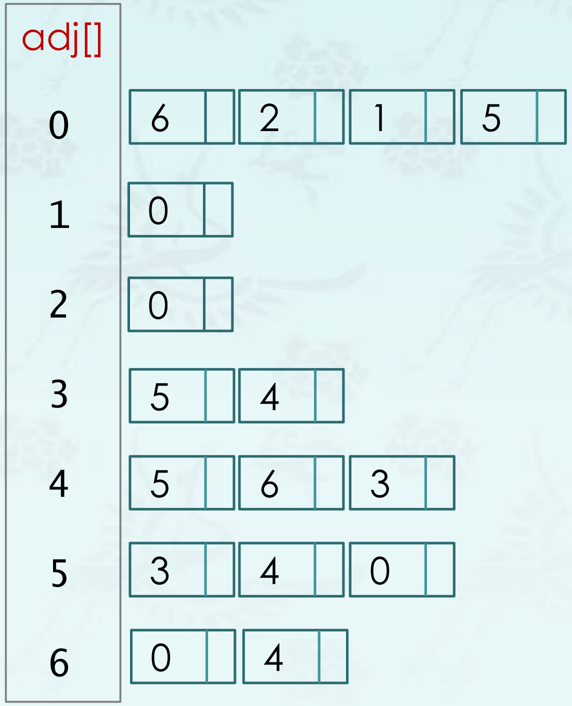
Adjacency list processing

Challenge: How to process adj[v] and its vertices:



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
```

Adjacency list processing

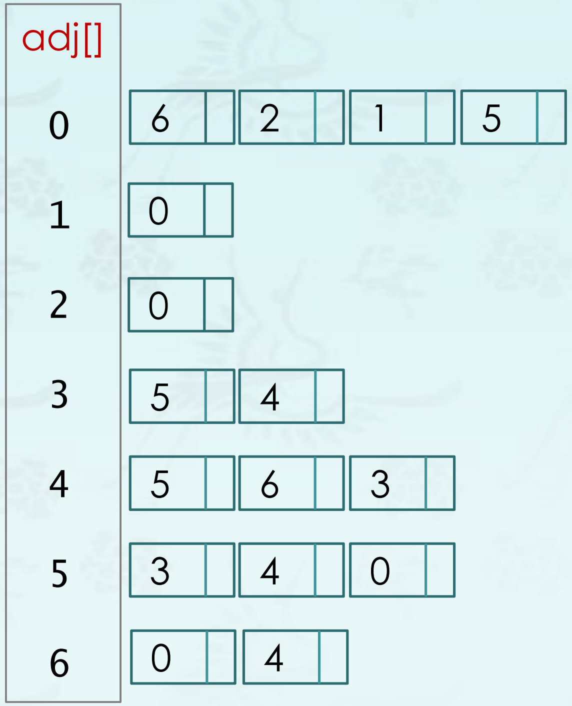
Challenge: How to process adj[v] and its vertices:

```
// print the adjacency list of graph
void print_adjlist(graph g) {

  cout << "\n\tAdjacency-list: \n";
  for (int v = 0; v < V(g); ++v) {
    cout << "\tV[" << v << "]: ";
    gnode w = g->adj[v].next;
    while (w) {
        ~~

    }
    cout << endl;
  }
}
```

Adjacency lists



Adjacency list processing

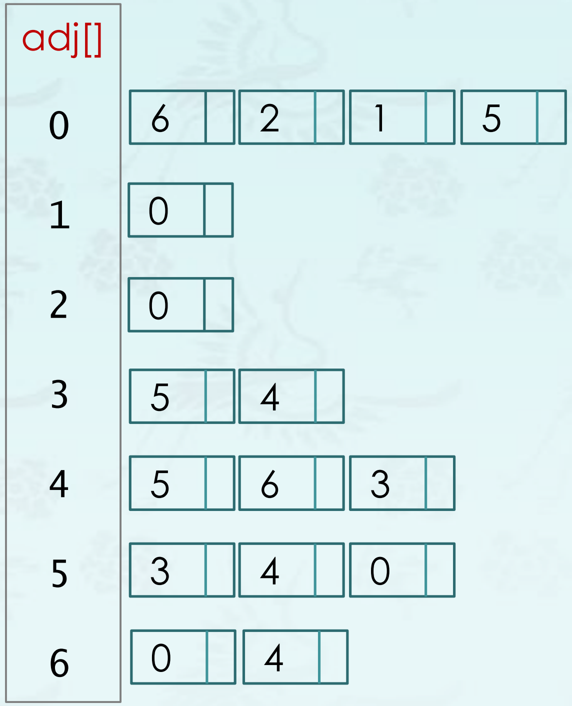
Challenge: How to process adj[v] and its vertices:

```
// print the adjacency list of graph
void print_adjlist(graph g) {

  cout << "\n\tAdjacency-list: \n";
  for (int v = 0; v < V(g); v++) {
    cout << "\tV[" << v << "]: ";
    for (gnode w = g->adj[v].next; w; w = w->next) {

       ~~
    }
}
```

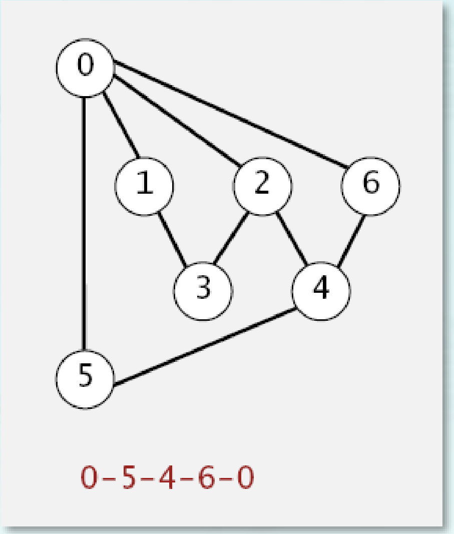
Adjacency lists



Cycle detection using depth-first search

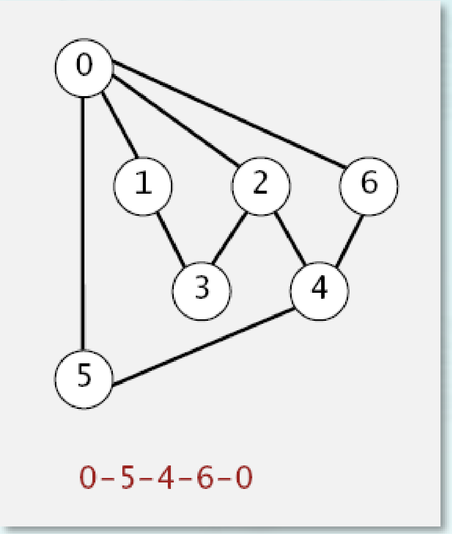
Problem: Find a cycle.

How difficult?



Cycle detection using depth-first search

Problem: Find a cycle.

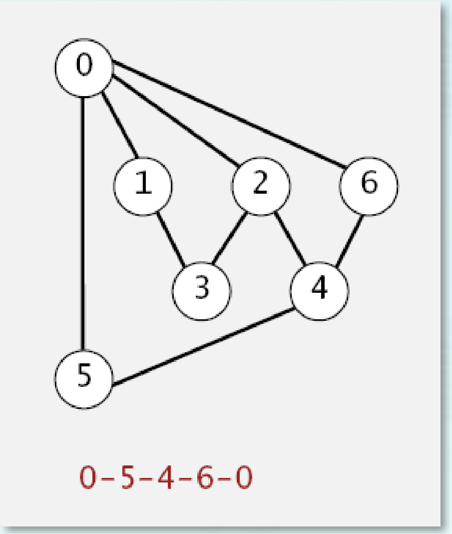


How difficult?

1. Any programmer could do it.
2. Typical diligent algorithms student could do it.
3. Hire an expert.
4. Intractable.
5. No one knows.
6. Impossible.

Cycle detection using depth-first search

Problem: Find a cycle.



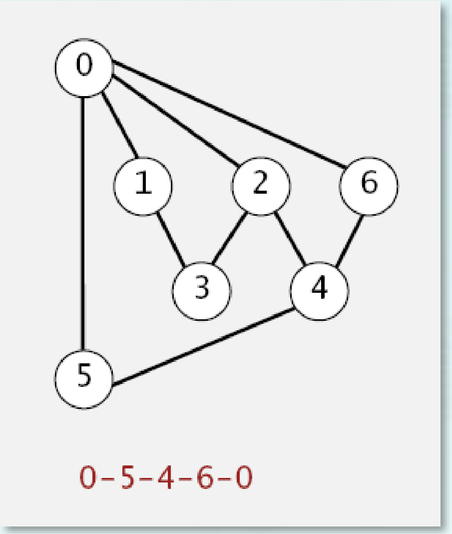
How difficult?

1. Any programmer could do it.
2. Typical diligent algorithms student could do it.
3. Hire an expert.
4. Intractable.
5. No one knows.
6. Impossible.

simple DFS-based solution

Cycle detection using depth-first search

Problem: Find a cycle.

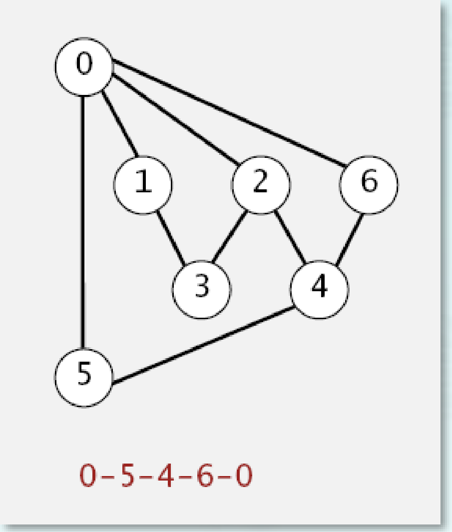


- A *cycle* is a path (with at least one edge) whose first and last vertices are the same.
- A *simple cycle* is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices).

Cycle detection using depth-first search

Challenge - Cycle detection: Is a given graph cyclic?

Implementation: Use depth-first search to determine whether a graph has a cycle, and if so return one.
It takes time proportional to V + E in the worst case.

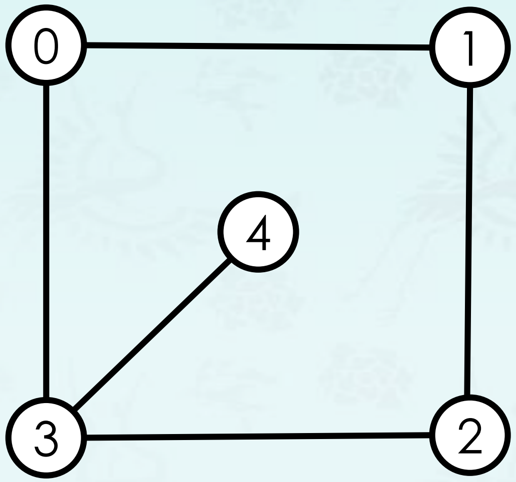


- A *cycle* is a path (with at least one edge) whose first and last vertices are the same.
- A *simple cycle* is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices).

Cycle detection **example** using depth-first search

To visit a vertex v:

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



Graph g:

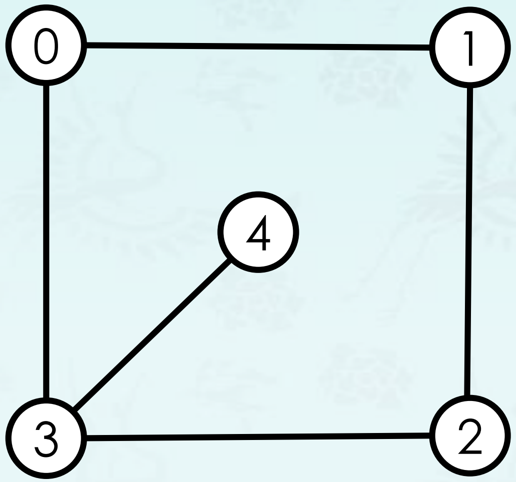
```
graph6.txt
5    ← V
5    ← E
0 1
0 3
1 2
2 3
3 4
```

Challenge: build adjacency lists?

Cycle detection **example** using depth-first search

To visit a vertex v:

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



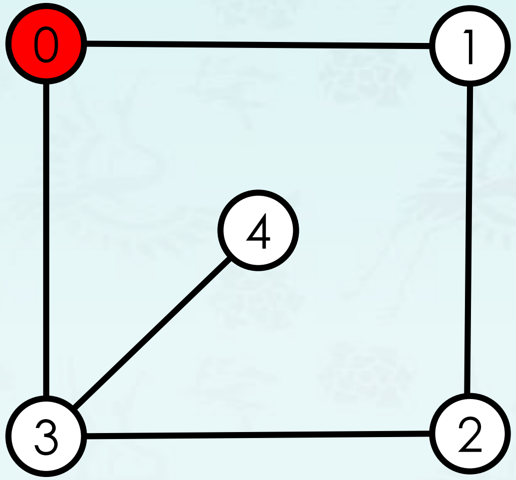

```
graph6.txt
5    V
5    E
0 1
0 3
1 2
2 3
3 4
```

Graph g:

Cycle detection **example** using depth-first search

To visit a vertex v:

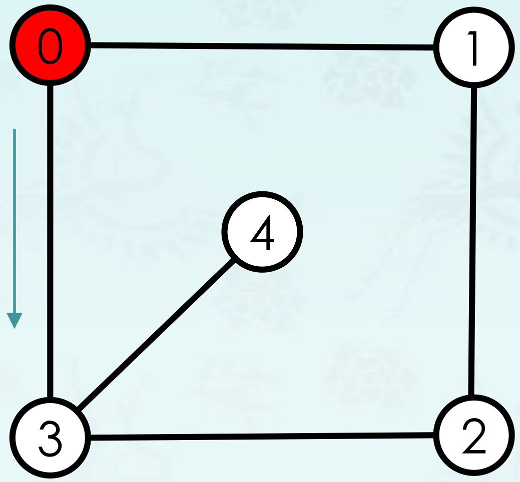
- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

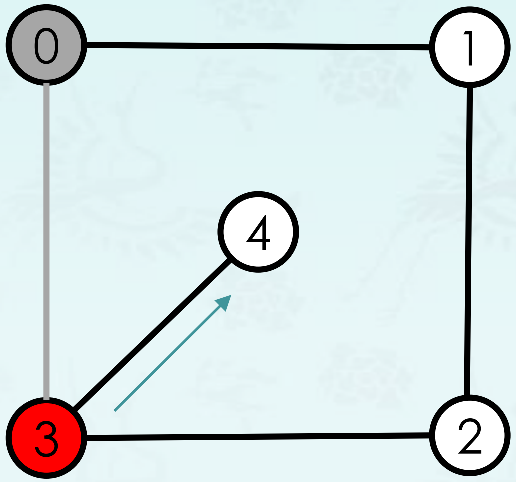
visit 0: check 3, **check 1**

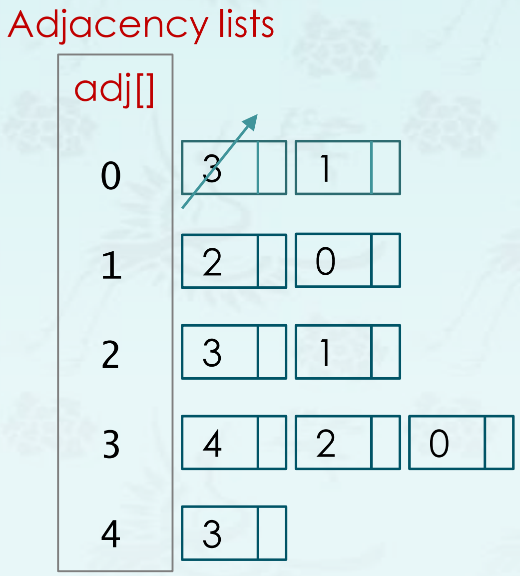
DFS: 0



visit 0: check 3, **check 1**

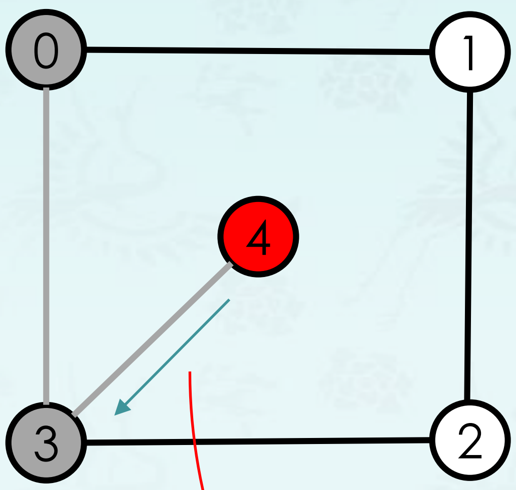
DFS: 0



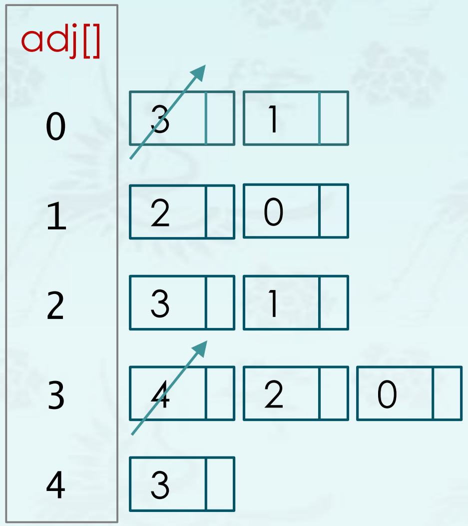


visit 3: check 4, **check 2, check 0**

DFS: 0 3



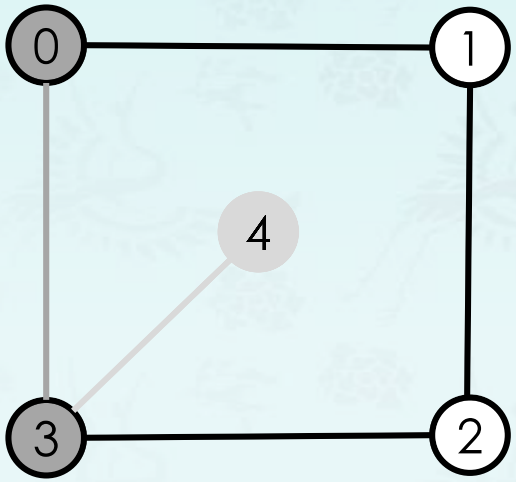
Adjacency lists

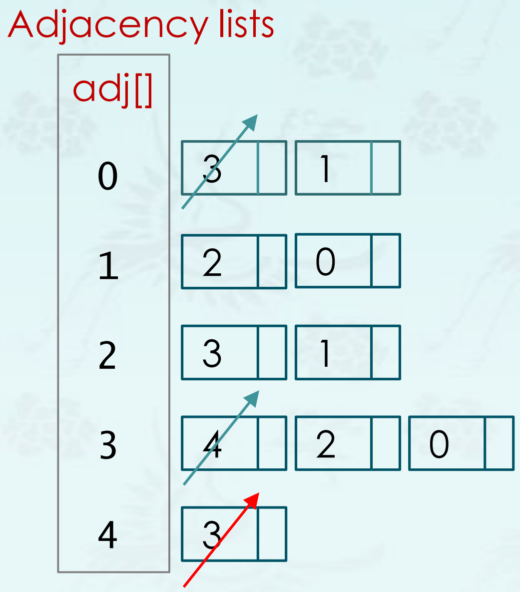


This is not cycle.
When you look for a cycle,
disregard the back edge leading to the previous vertex.

visit 4: check 3

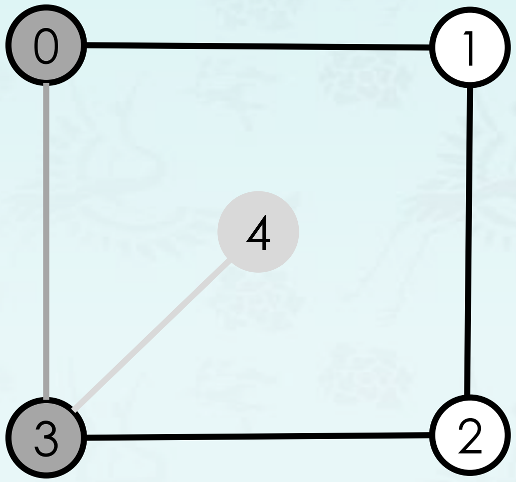
DFS: 0 3 4



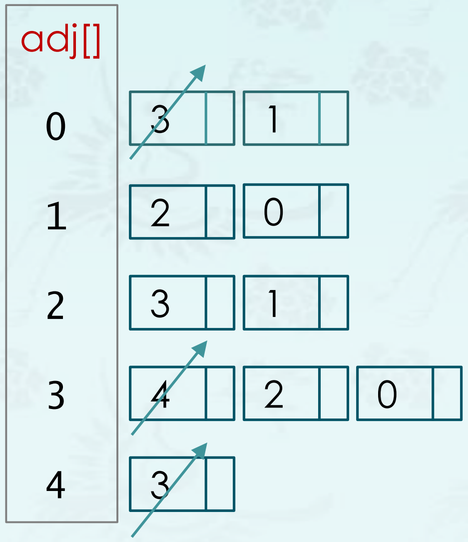


4 done

DFS: 0 3 4

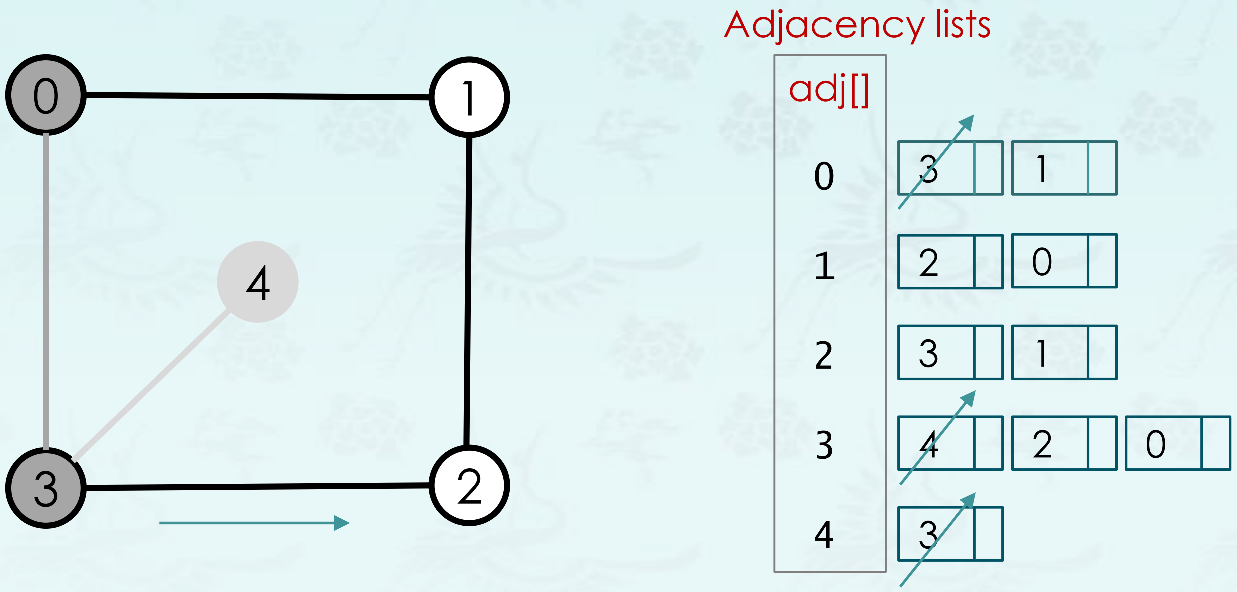


Adjacency lists



visit 3: check 4, **check 2**, **check 0**

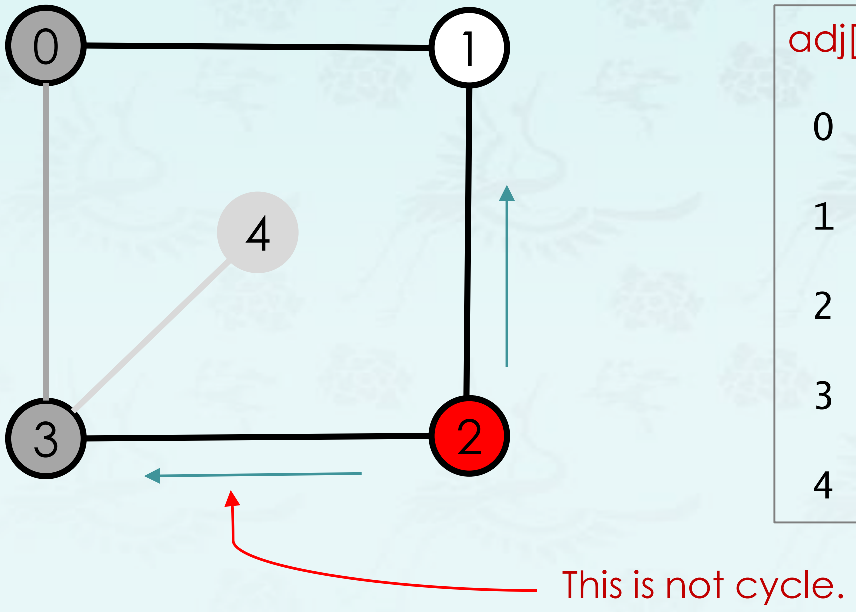
DFS: 0 3 4

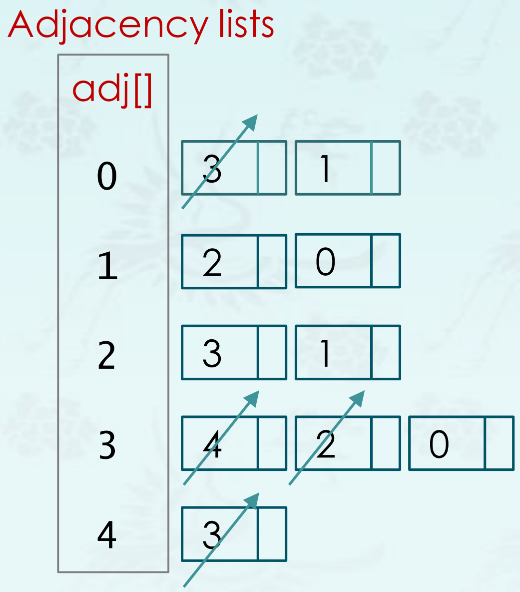


4	2	0

visit 3: check 4, **check 2**, **check 0**

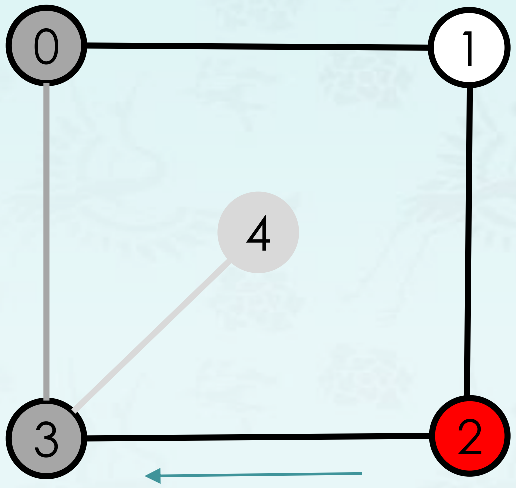
DFS: 0 3 4

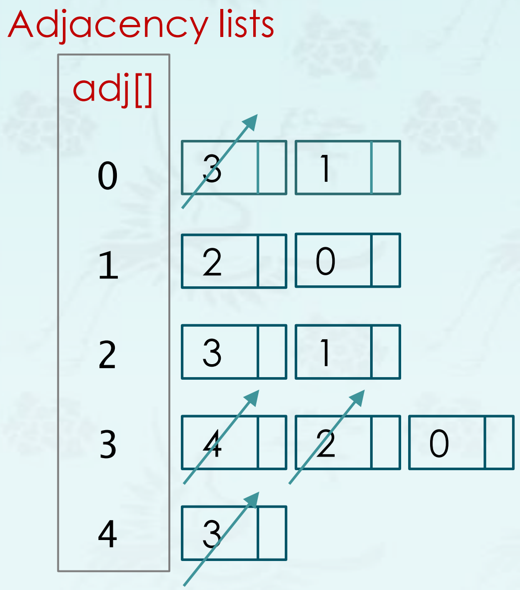




visit 2: **check 3, check 1**

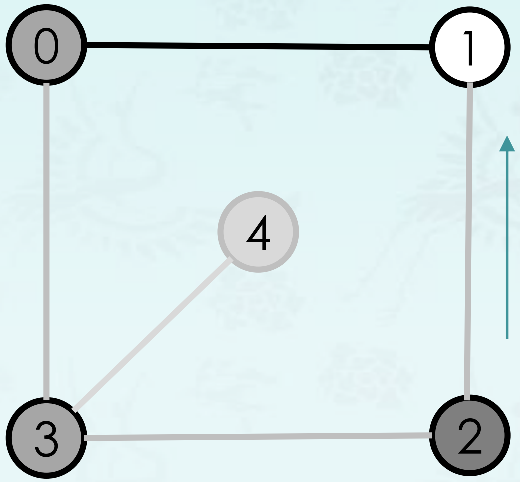
DFS: 0 3 4 2



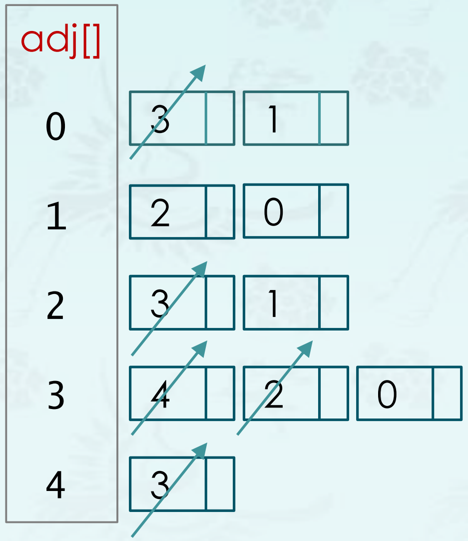


visit 2: **check 3,** **check 1**

DFS: 0 3 4 2



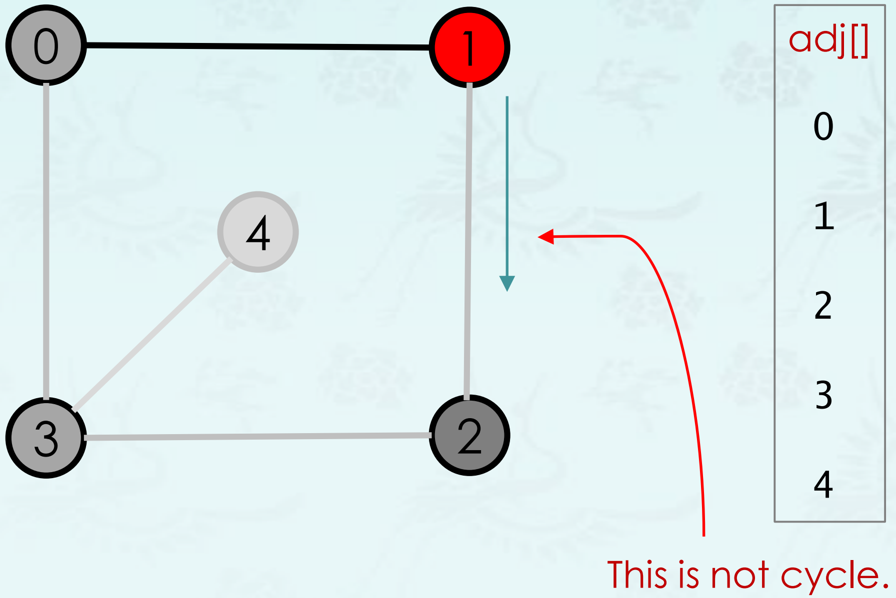
Adjacency lists



visit 2: check 3, **check 1**

DFS: 0 3 4 2

Cycle detection **example** using depth-first search



Adjacency lists

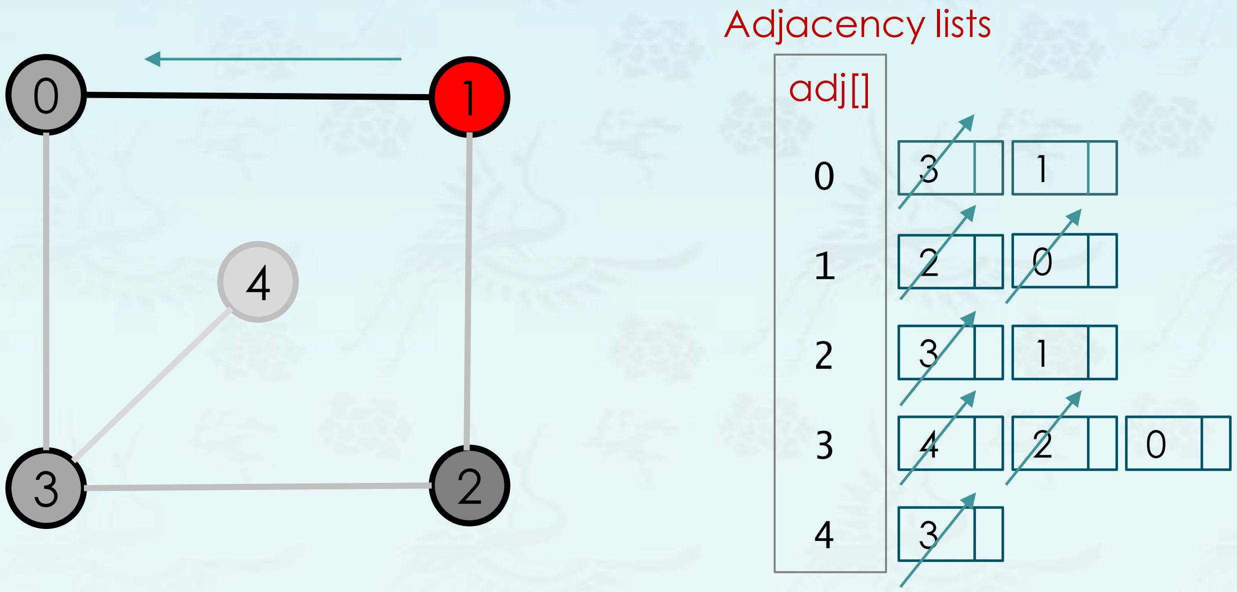
adj[]			
0	~~3~~	1	
1	~~2~~	~~0~~	
2	~~3~~	1	
3	~~4~~	~~2~~	0
4	~~3~~		

2	0

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

DFS: 0 3 4 2 1

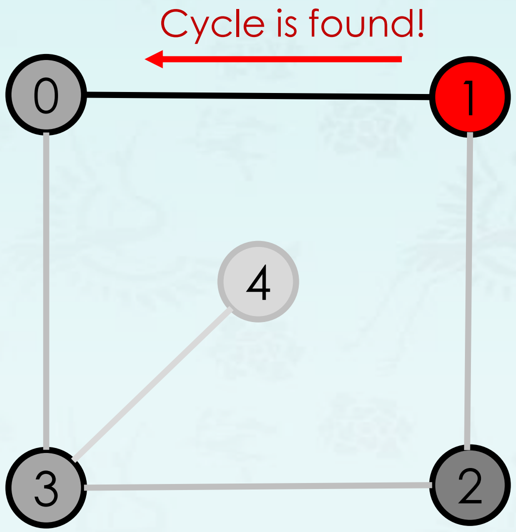
Cycle detection **example** using depth-first search



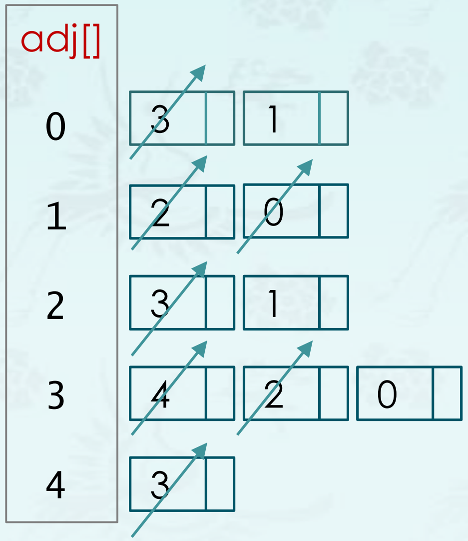
visit 1: check 2, **check 0**

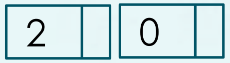
DFS: 0 3 4 2 1

Cycle detection **example** using depth-first search



Adjacency lists

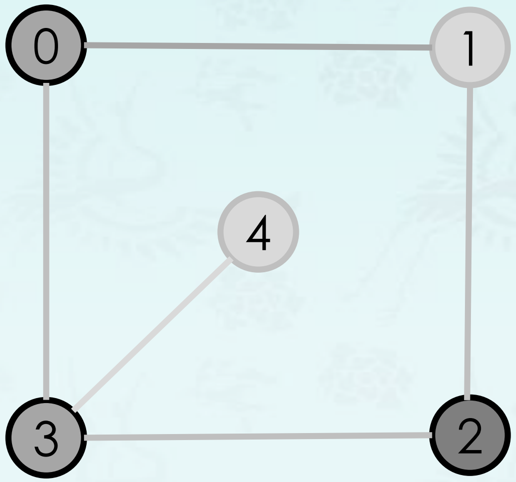


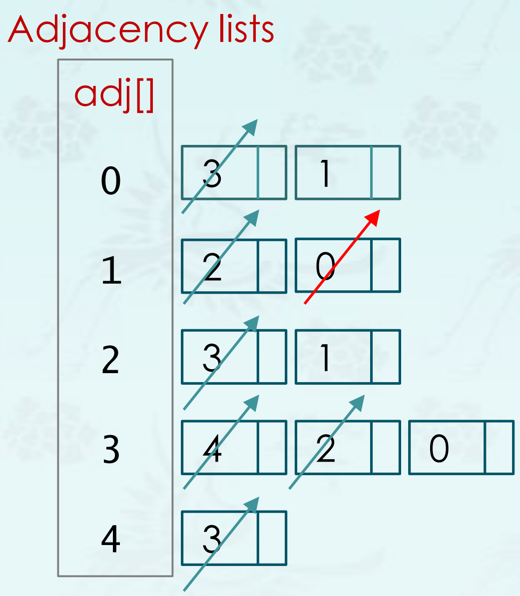


visit 1: check 2, **check 0**

DFS: 0 3 4 2 1

Cycle detection **example** using depth-first search

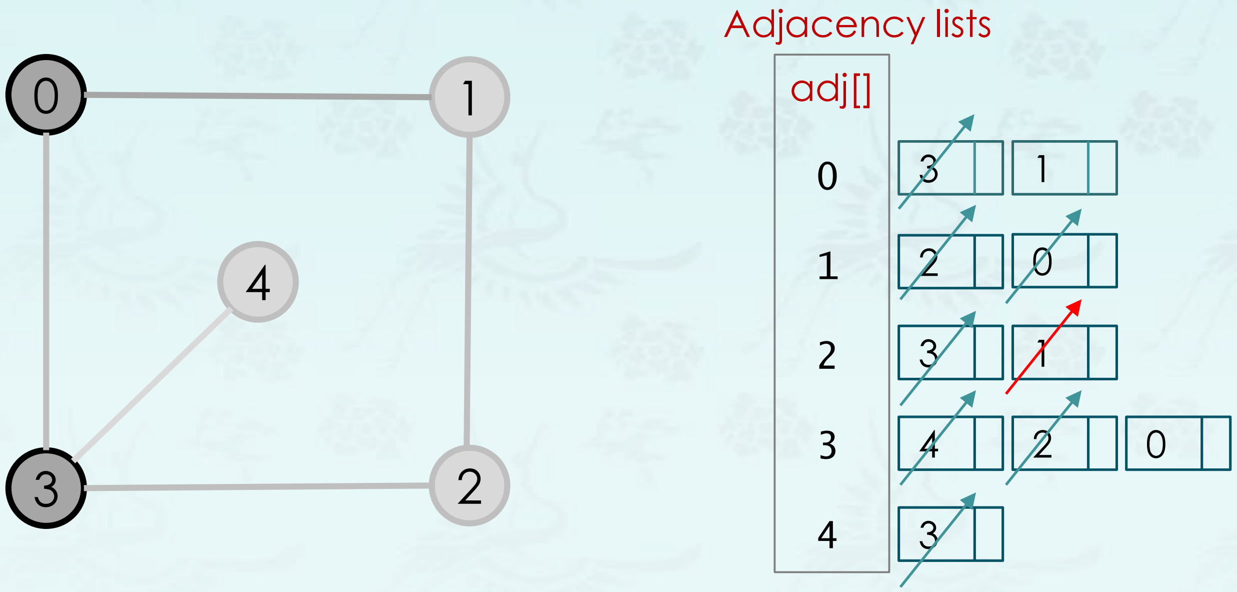




1 done

DFS: 0 3 4 2 1

Cycle detection **example** using depth-first search

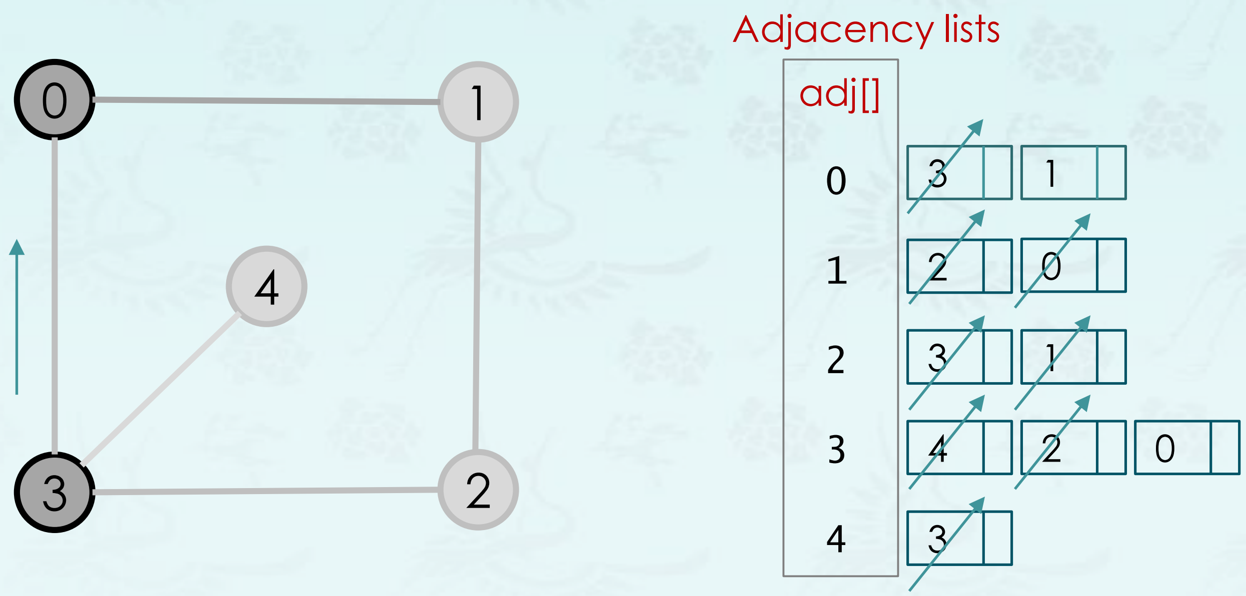


2 done

Once 1 done, 2 is done;
since it was recurred from “visit2: check 3, check 1”

DFS: 0 3 4 2 1

Cycle detection **example** using depth-first search

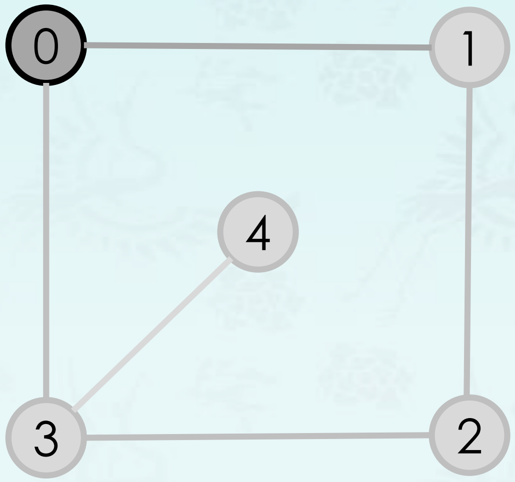


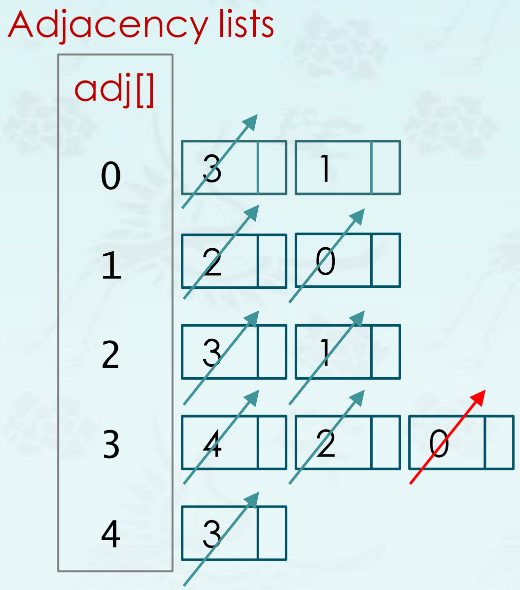
4	2	0

visit 3: check 4, **check 2**, **check 0**

DFS: 0 3 4 2 1

Cycle detection **example** using depth-first search

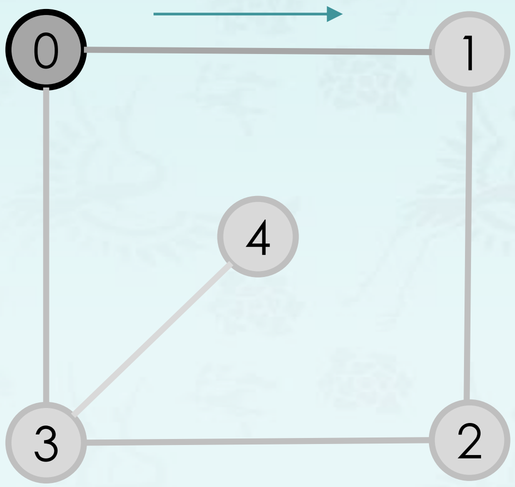


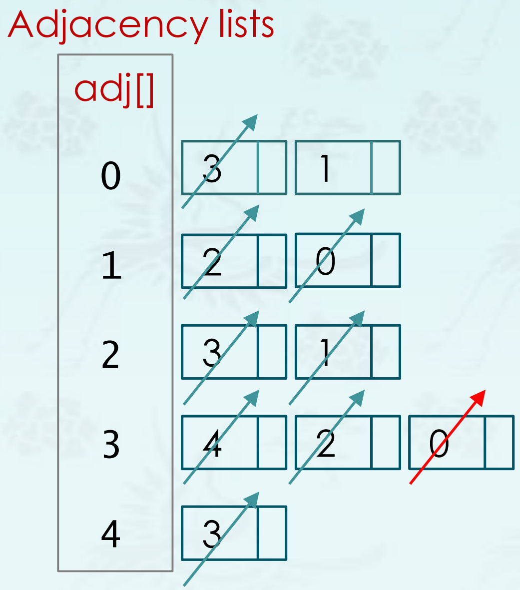


3 done

DFS: 0 3 4 2 1

Cycle detection **example** using depth-first search

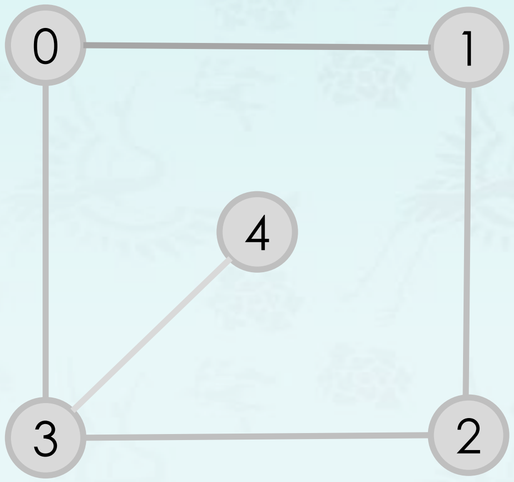




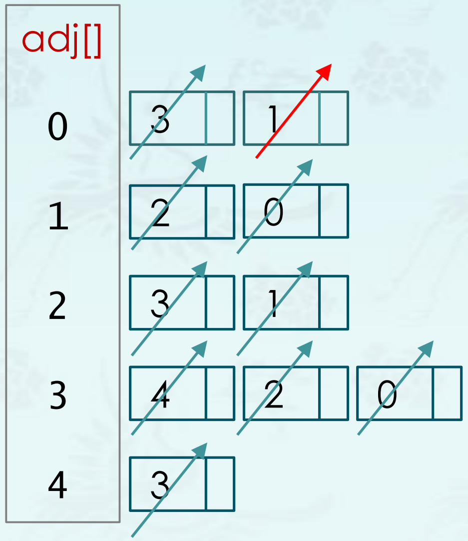
visit 0: check 3, **check 1**

DFS: 0 3 4 2 1

Cycle detection **example** using depth-first search



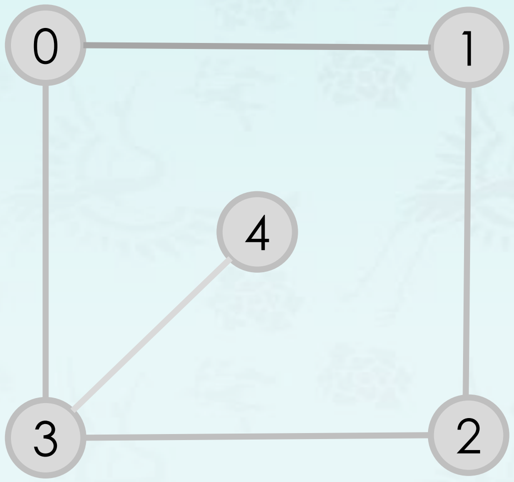
Adjacency lists

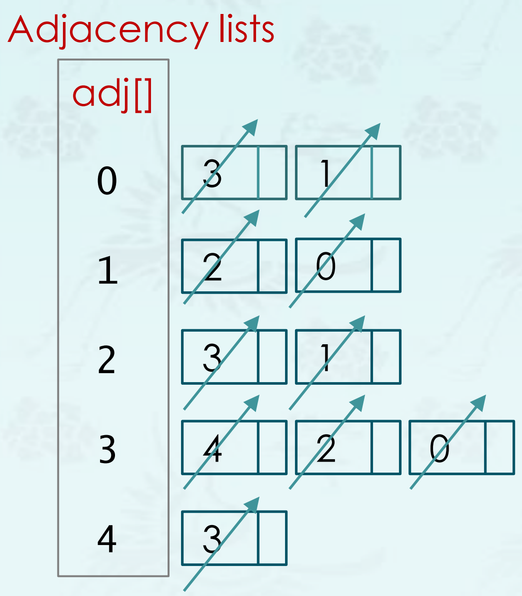


0 done

DFS: 0 3 4 2 1

Cycle detection **example** using depth-first search

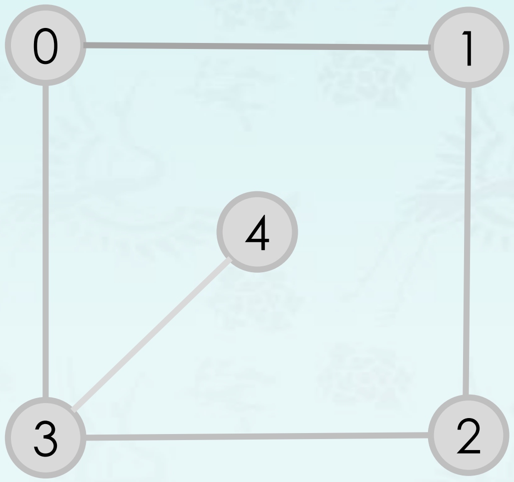


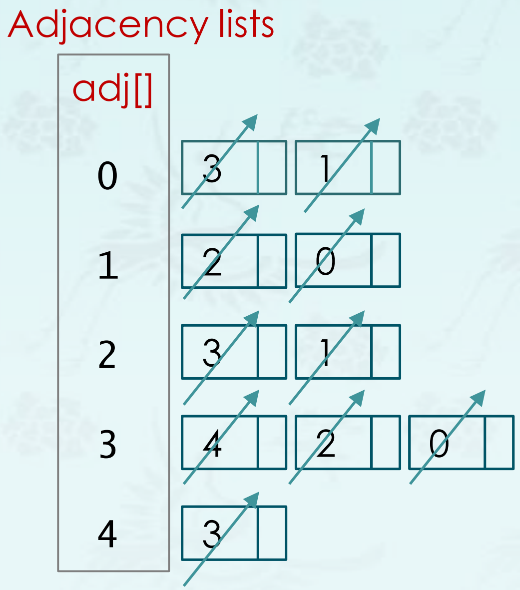


visit(0)
check(3)
visit(3)
check(4)
visit(4)
check(3)
4 done
check(2)
visit(2)
check(3)
check(1)
visit(1)
check(2)
check(0)
1 done
2 done
check(0)
3 done
check(1)
0 done

DFS: 0 3 4 2 1

Cycle detection **example** using depth-first search





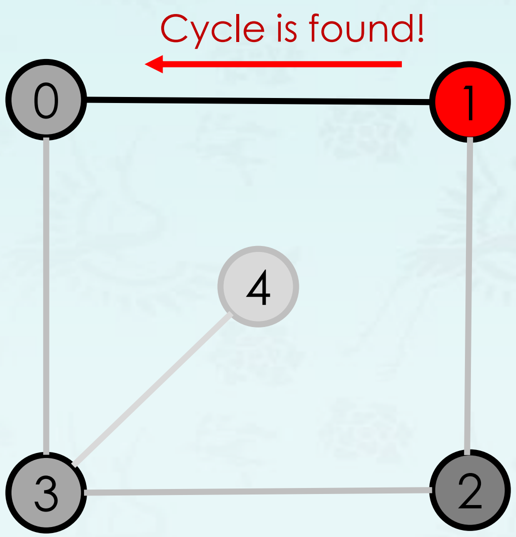
v	marked[]	parent[v]
0	T	-1
1	T	2
2	T	3
3	T	0
4	T	3

DFS: 0 3 4 2 1

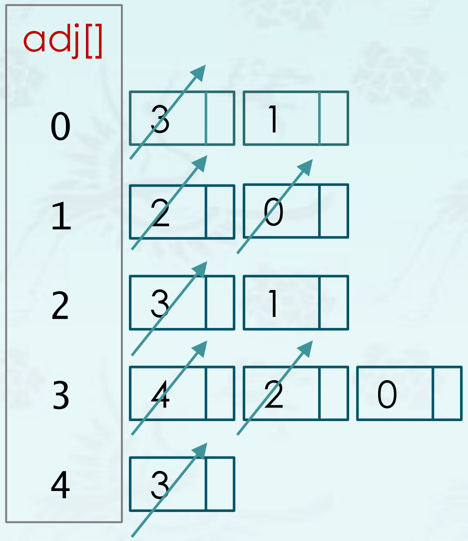
Cycle detection **example** using depth-first search

Cycle is found:

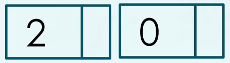
- **push path**



Adjacency lists



v	marked[]	parent[v]
0	T	-1
1	T	2
2	T	3
3	T	0
4	T	3

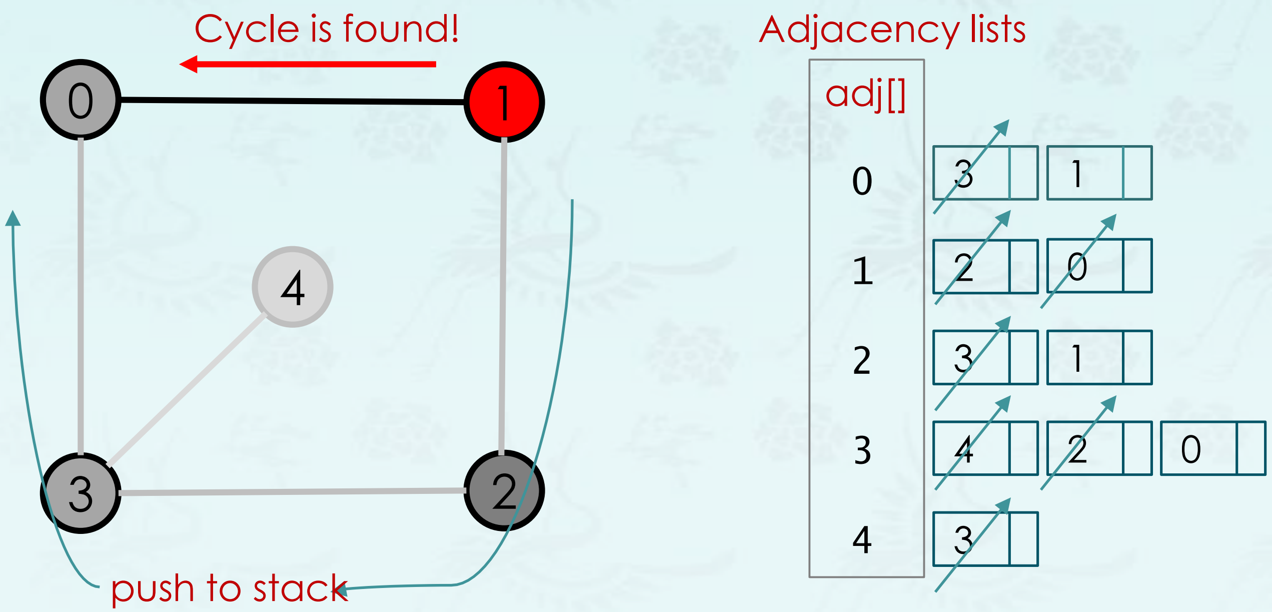


visit 1: check 2, **check 0**

Cycle detection **example** using depth-first search

Cycle is found: starting at itself

- **push path (1, 2, 3 or retrace back parent[] until you hit 0)**



v	marked[]	parent[v]
0	T	-1
1	T	2
2	T	3
3	T	0
4	T	3

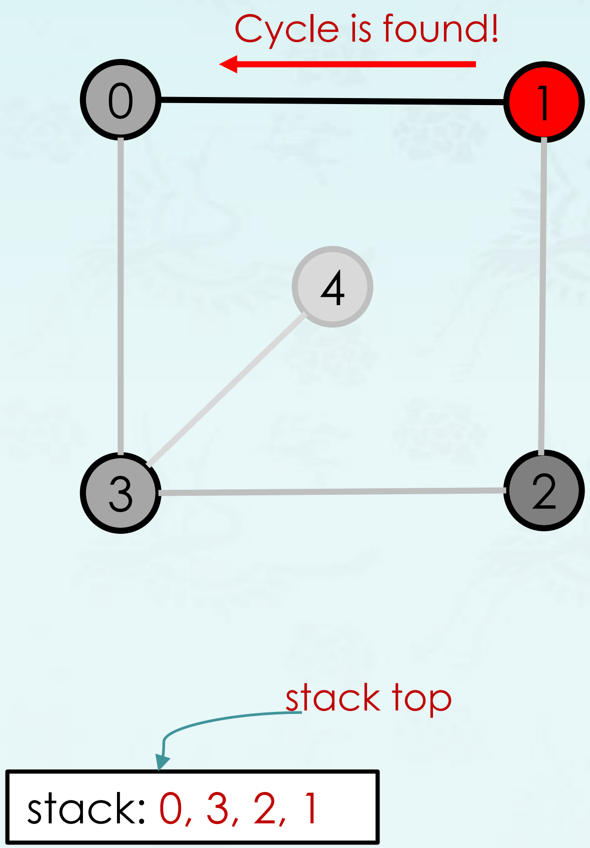
stack top

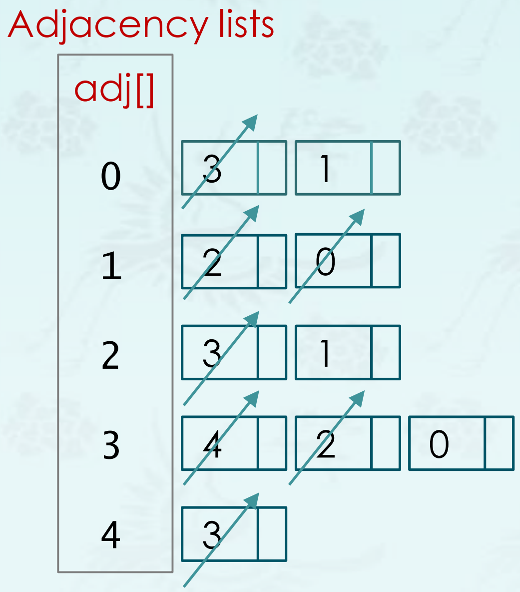
stack: 3, 2, 1

Cycle detection example using depth-first search

Cycle is found:

- **push path (1, 2, 3 or retrace back parent[] until you hit 0)**
- **push 0**





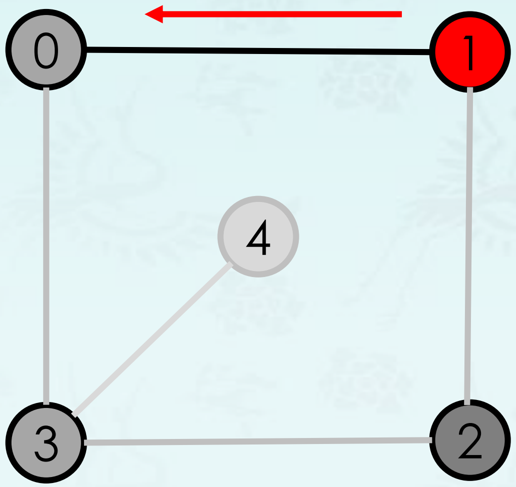
v	marked[]	parent[v]
0	T	-1
1	T	2
2	T	3
3	T	0
4	T	3

Cycle detection **example** using depth-first search

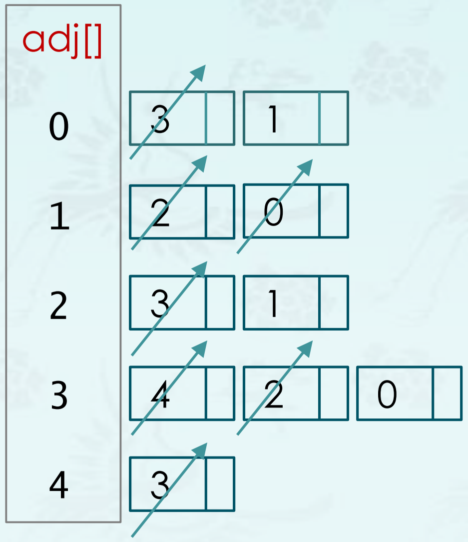
Cycle is found:

- **push path (1, 2, 3 or retrace back parent[] until you hit 0)**
- **push 0**
- **push 1 (to complete the cycle)**

Cycle is found!



Adjacency lists



v	marked[]	parent[v]
0	T	-1
1	T	2
2	T	3
3	T	0
4	T	3

stack top

stack: 0, 3, 2, 1

.... and add myself(1) to make a cycle

stack: 1, 0, 3, 2, 1

Cycle detection using DFS implementation

```
// finds a cycle in graph and returns a stack that has a list of vertices
// using DFS to find a cycle in the graph.
// The cycle() takes time proportional to V + E(in the worst case),
// where V is the number of vertices and E is the number of edges.

bool cyclic_at(graph g, int v, stack<int>& cy) {
  if (hasSelfLoop(g, cy) || hasParallelEdges(g, cy)) return true;

  for (int i = 0; i < V(g); i++) {
    g->marked[i] = false;
    g->parentDFS[i] = -1;
  }

  return DFScyclic(g, -1, v, cy); // u:vertex visited previously, v:visiting vertex
}
```

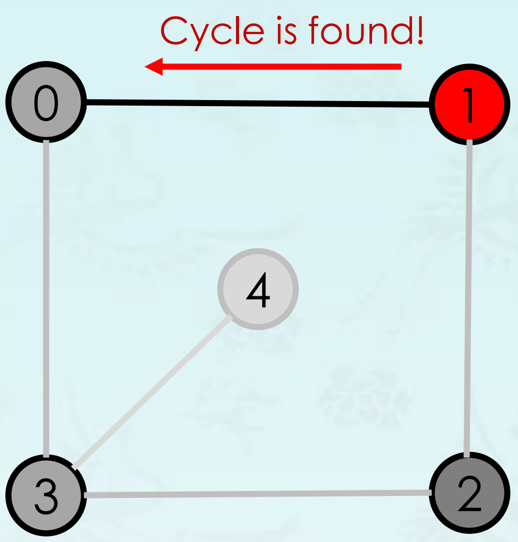
why? stay tuned.

Cycle detection using DFS implementation

```
// Recursive DFS does the work
// g: the gragph, u: vertex visited previously, v: visiting vertex
bool DFScyclic(graph g, int u, int v, stack<int>& cy){
    g->marked[v] = true;         // visit vertex v

    for (gnode w = g->adj[v].next; w; w = w->next) {        // check all vertices in adj.list
      if (cy.size() > 0) return true;
      if (!g->marked[w->item]) {
            g->parent[w->item] = v;
            DFScyclic (g, v, w->item, cy);
      }
      // check for cycle (but disregard reverse of edge leading to v)
      else if (w->item != u) {
        // Now... a cycle is found
        // instantiate a stack
        // push all vertices that led us here or (1, 2, 3) - use for loop and g->parentDFS[]
        // push the last v or starting v of cycle (1, 2, 3, 0)
        // push the current visit v (1, 2, 3, 0, 1)
      }
    }
}
```

Cycle detection using DFS implementation



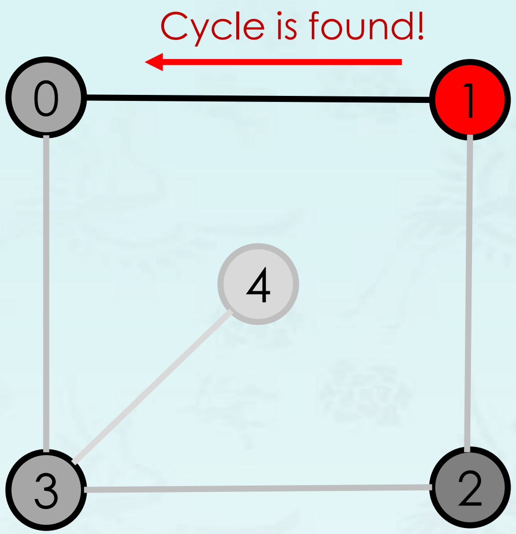
stack<int> cy : 1 0 3 2 1

stack top

top( ) & pop()

Output:

Cycle detection using DFS implementation



stack<int> cy : 1 0 3 2 1

stack top

top( ) & pop()

Output: 1 0 3 2 1

Graph-processing challenge 1 – Review

Problem: Is a graph bipartite (or bigraph)?

a set of graph vertices decomposed into two disjoint sets
such that no two graph vertices within the same set are adjacent.

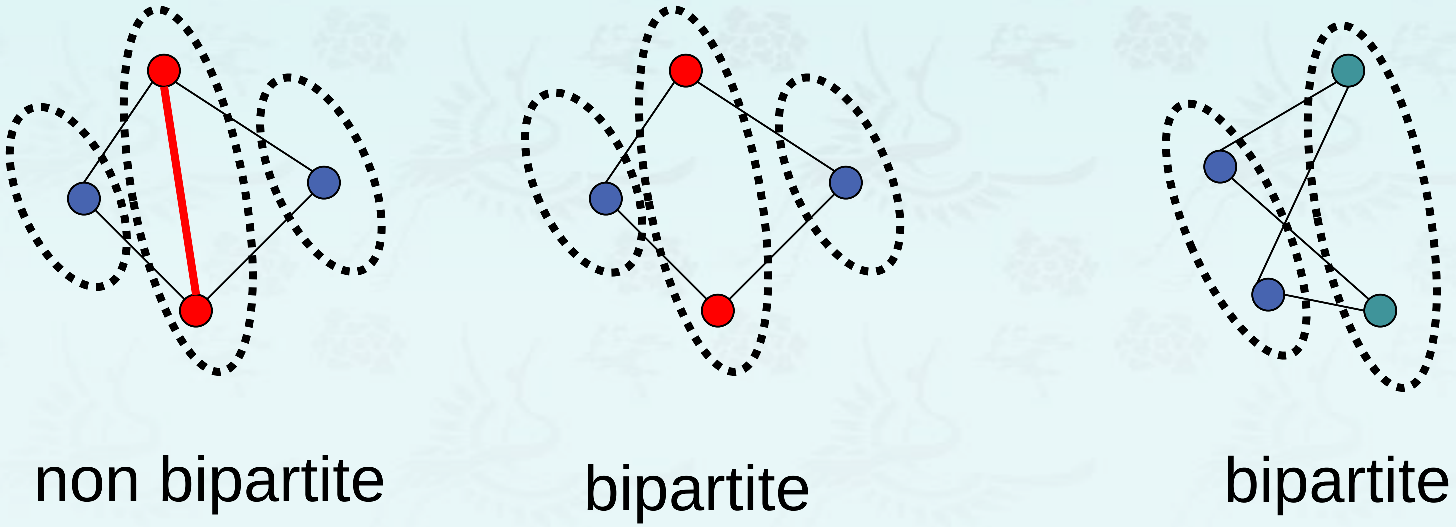
How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 1 – Review

Problem: Is a graph bipartite (or bigraph)?

a set of graph vertices decomposed into **two disjoint sets**
such that no two graph vertices within the same set are adjacent.

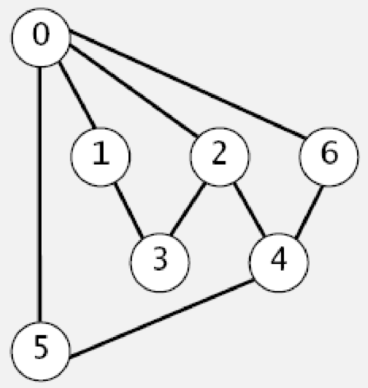


Graph-processing challenge 1 – Review

Problem: Is a graph bipartite (or bigraph)?

a set of graph vertices decomposed into two disjoint sets
such that no two graph vertices within the same set are adja

a bigraph ?



How difficult?

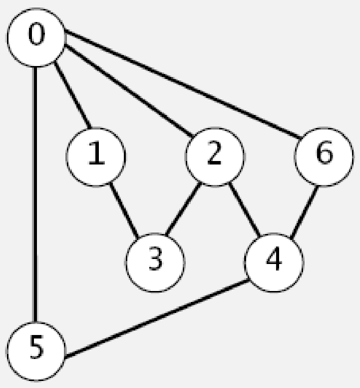
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 1 – Review

Problem: Is a graph bipartite (or bigraph)?

a set of graph vertices decomposed into two disjoint sets
such that no two graph vertices within the same set are adja

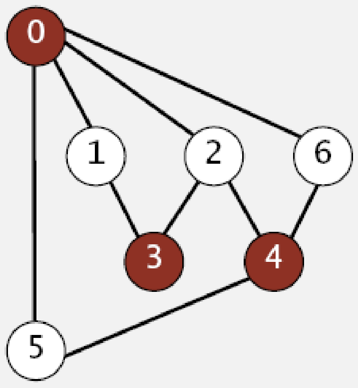
a bigraph ?



How difficult?

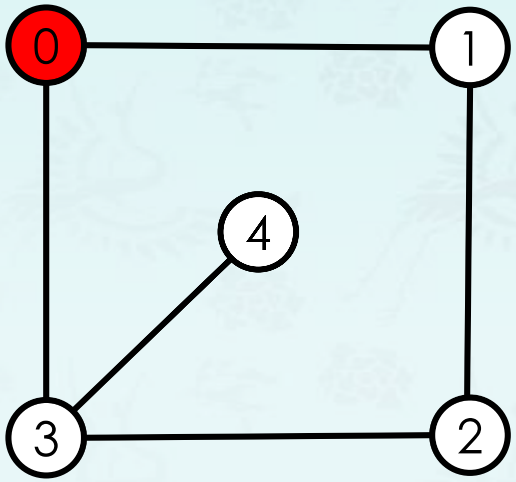
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

simple DFS or BFS-based solution



Graph-processing challenge 1 – bigraph

Problem: Is a graph bipartite (or bigraph)?



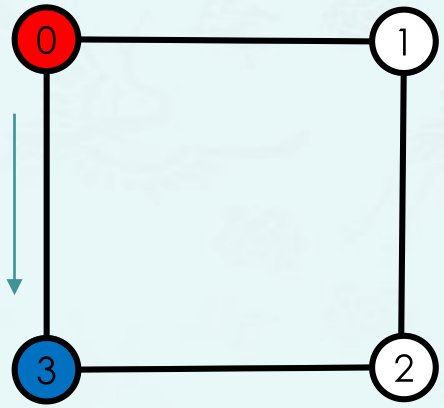
visit 0: check 3, **check 1**

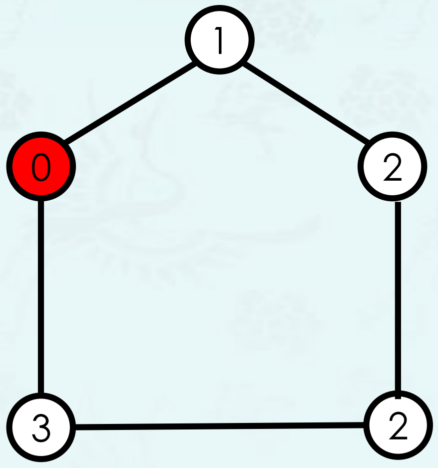
Graph-processing challenge 1 – bigraph

Problem: Is a graph bipartite (or bigraph)?

Solution: Two-colorability
The vertices of a given graph can be assigned one of two colors in such a way that no edge connects vertices of the same color.

Solution: It is called two-colorability. graphBipartite() uses depth-first search to determine whether or not a graph has a bipartition; if so, return one; if not, return an odd-length cycle. It takes time proportional to V + E in the worst case.



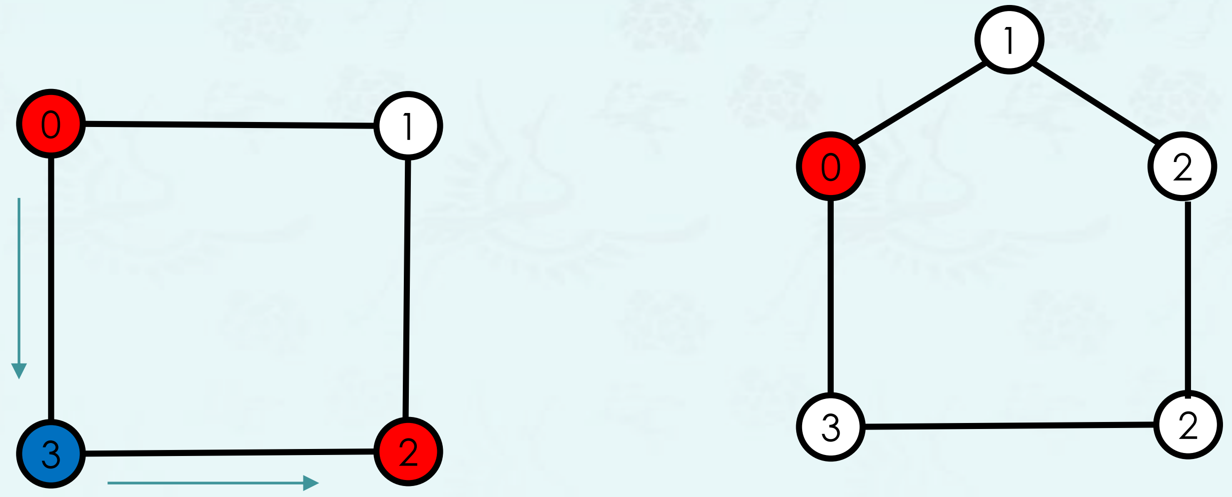


Graph-processing challenge 1 – bigraph

Problem: Is a graph bipartite (or bigraph)?

Solution: Two-colorability
The vertices of a given graph can be assigned one of two colors in such a way that no edge connects vertices of the same color.

Solution: It is called two-colorability. graphBipartite() uses depth-first search to determine whether or not a graph has a bipartition; if so, return one; if not, return an odd-length cycle. It takes time proportional to V + E in the worst case.

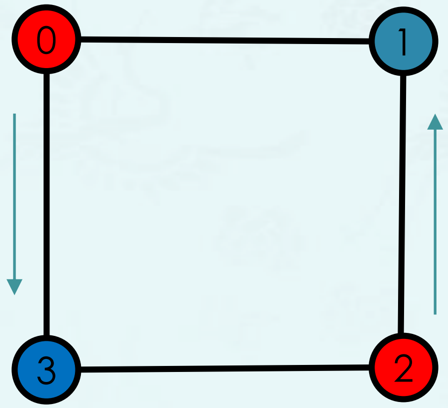


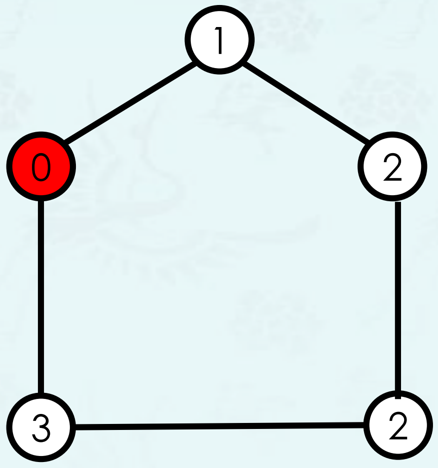
Graph-processing challenge 1 – bigraph

Problem: Is a graph bipartite (or bigraph)?

Solution: Two-colorability
The vertices of a given graph can be assigned one of two colors in such a way that no edge connects vertices of the same color.

Solution: It is called two-colorability. graphBipartite() uses depth-first search to determine whether or not a graph has a bipartition; if so, return one; if not, return an odd-length cycle. It takes time proportional to V + E in the worst case.



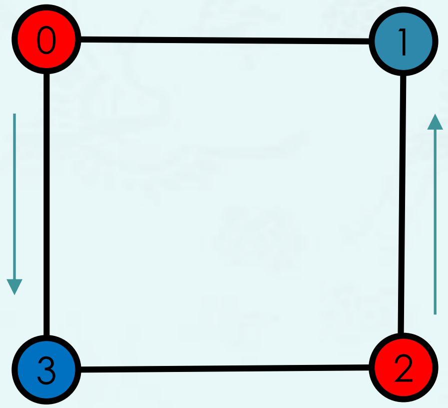


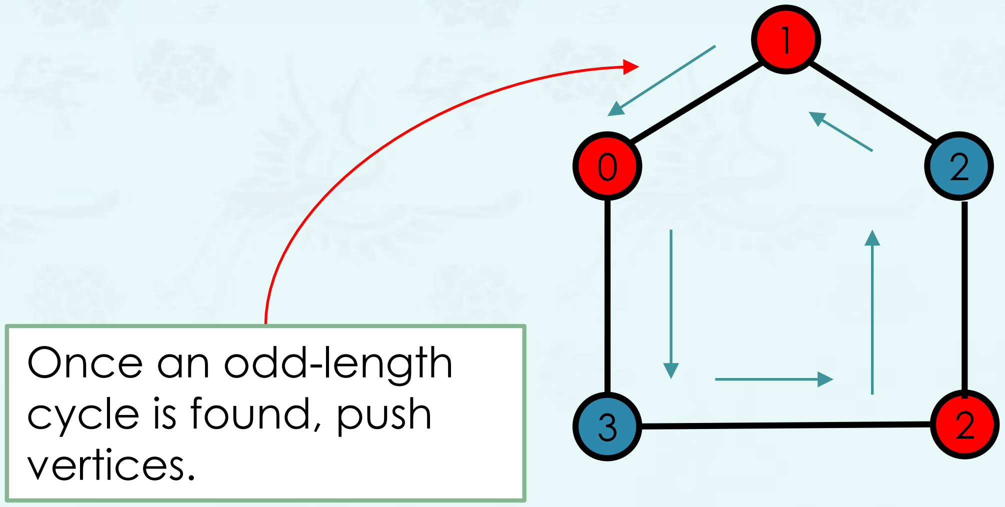
Graph-processing challenge 1 – bigraph

Problem: Is a graph bipartite (or bigraph)?

Solution: Two-colorability
The vertices of a given graph can be assigned one of two colors in such a way that no edge connects vertices of the same color.

Solution: bipartite() uses depth-first search to determine whether a graph has a bipartition or not; if not, return an odd-length cycle.
It takes time proportional to V + E in the worst case.





Graph-processing challenge 1 – bigraph coding

```
// determines whether or not an undirected graph is bigraph and
// finds either a bipartition or an odd length cycle.
// returns a stack with cyclic vertices pushed.
bool bigraph(graph g, stack<int>& cy) {
       if (empty(g)) return false;
       for (int i = 0; i < V(g); i++) {
              g->marked[i] = false;
              g->color[i] = BLACK;  // BLACK=0, WHITE=1
              g->parentDFS[i] = -1;   // needs info when backtrack the cycle.
       }
       cy = {};                                  // clear stack
       for (int v = 0; v < V(g); v++) {
              if (!g->marked[v]) {
                     if (!DFSbigraph(g, v, cy))
                            return false;  // found an odd-length cycle
              }
       }
       return true;
}
```

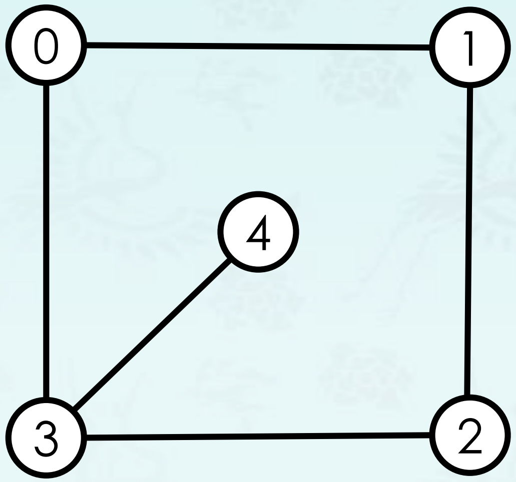
2

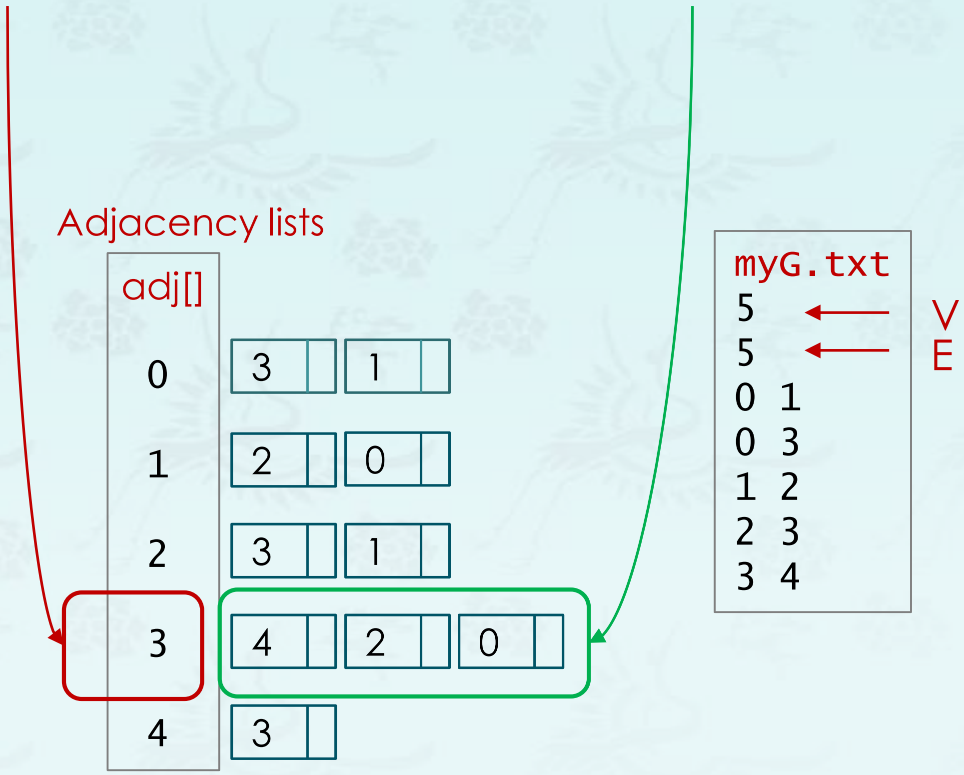
Graph-processing challenge 1 – bigraph coding

```
// Recursive DFS does the work
bool DFSbigraph(graph g, int v, stack<int>& cy) {
  g->marked[v] = true;
  for (gnode w = g->adj[v].next; w; w = w->next) {
    // short circuit if odd-length cycle found
    if (cy.size() > 0) return false; // found 1st cycle
    if (!g->marked[w->item]) {                       // found uncolored vertex, so recur
        g->parentDFS[w->item] = v;                   // keep it to backtrack the cycle.
        g->color[w->item] = !g->color[v];            // flip the color
        DPRINT(cout << " " << v << " Color:" << g->color[v] << ",";);
        DPRINT(cout << " " << w->item << " Color:" << g->color[w->item] << endl;);
        DFSbigraph(g, w->item, cy);
    }        // if v-w create an odd-length cycle, find it (push vertices and push them)
    else if (g->color[w->item] == g->color[v]) {
        //bipartite = false;
        // 1. instantiate a new stack and set it to g->cycle
        // 2. push w->item since first v = last v, duplicated
        // 3.  retrace g->parent[x] from v to w->item
        //       and push them to stack – need a for loop here.
        //  4. push w->item (to form a cycle)
    }
  }
}
```

Graph-processing challenge 1 – bigraph two-colorability coding

Solution: for every v, the color of adj[v] is different from those of adj[v]'s list vertices, if it is bipartite.

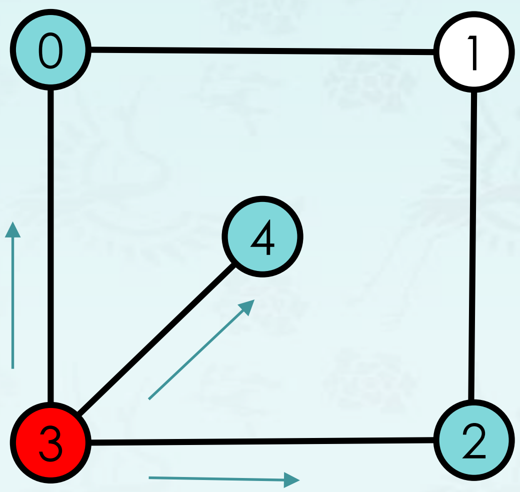


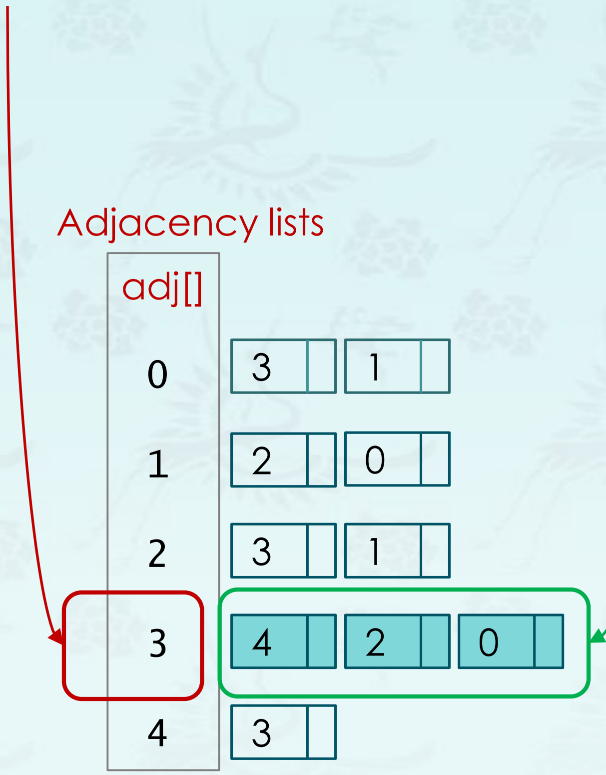


Graph g:

Graph-processing challenge 1 – bigraph two-colorability coding

Solution: for every v, the color of adj[v] is different from those of adj[v]'s list vertices, if it is bipartite.



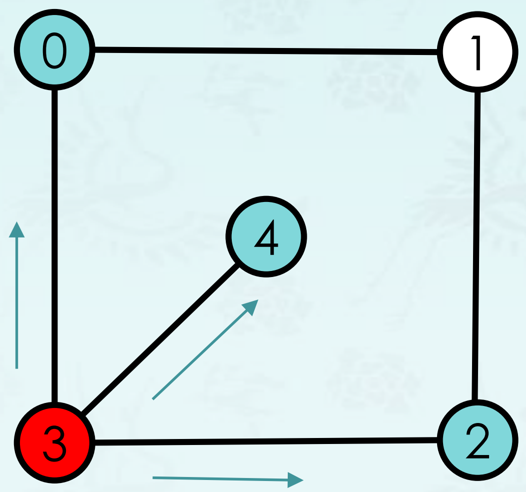



```
myG.txt
5   ← V
5   ← E
0 1
0 3
1 2
2 3
3 4
```

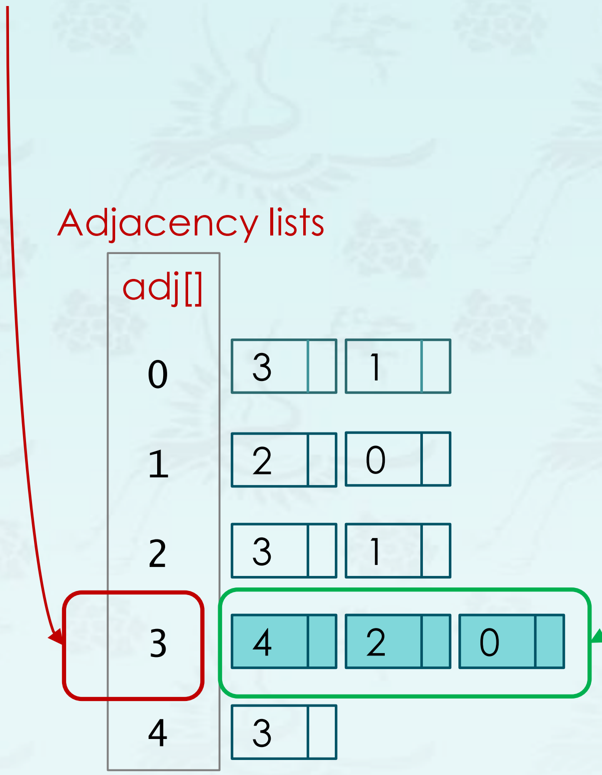
Graph g:

Graph-processing challenge 1 – bigraph two-colorability coding

Solution: for every v, the color of adj[v] is different from those of adj[v]'s list vertices, if it is bipartite.



Graph g:




```
myG.txt
5    ← V
5    ← E
0 1
0 3
1 2
2 3
3 4
```

v	marked[]	color[]
1	F	-1
2	F	-1
3	F	-1
4	F	-1
5	F	-1