ITP20001/ECE20010 Data Structures

Chapter 6

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

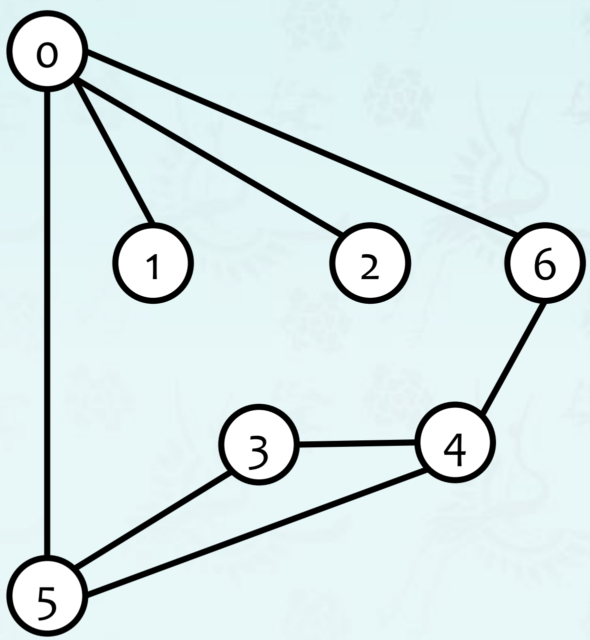
Major references:

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

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

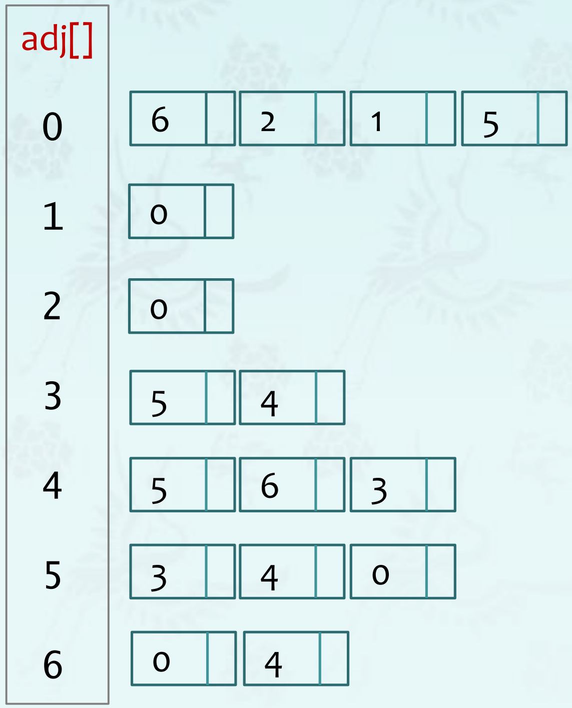
Adjacency list processing

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



Graph g

Adjacency lists



V-E lists

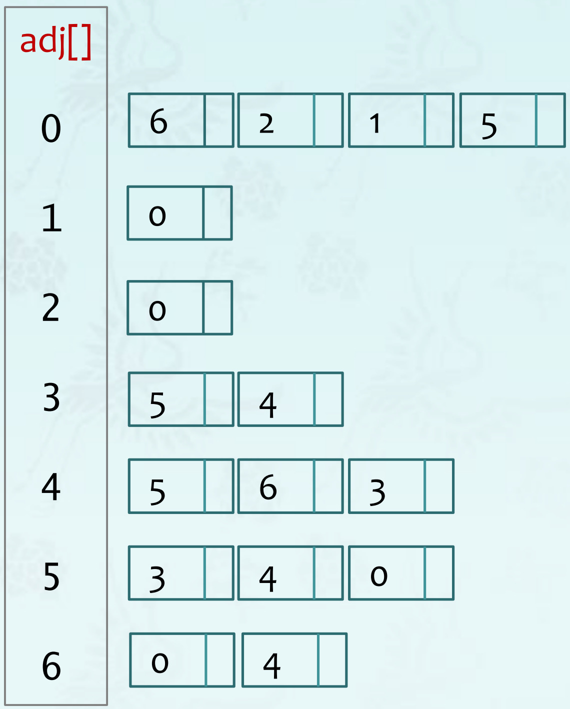
```
myG.txt
13      V
13      E
0  5
4  3
0  1
9  12
6  4
5  4
0  2
11  12
9  10
0  6
7  8
9  11
5  3
```

Adjacency list processing

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

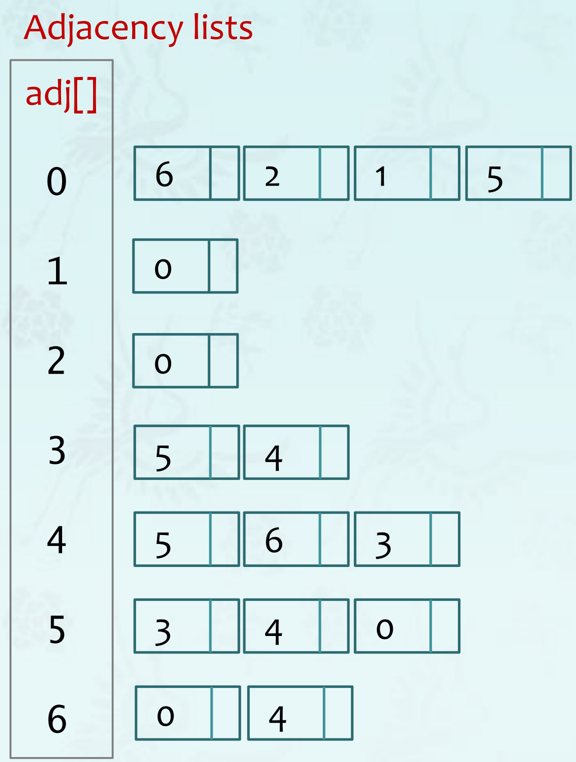
```
// print the adjacency list of graph
void printAdjList(graph g) {

    for (int v = 0; v < V(g); ++v) {
        gnode curr = g->adj[v].next;
        printf(" V[%d]: ", v);

        while (curr) {

        ~~

        }
        printf("\n");
    }
}
```

Adjacency lists



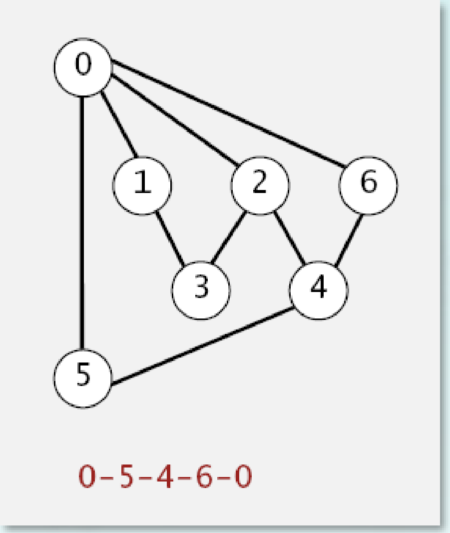
Challenge: How to process adj[v] and its vertices:

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

  for (int v = 0; v < V(g); v++) {
    printf(" V[%d]: ", v);

      for (gnode w = g->adj[v].next;    ~~ )
      {

       ~~


      }
}
```



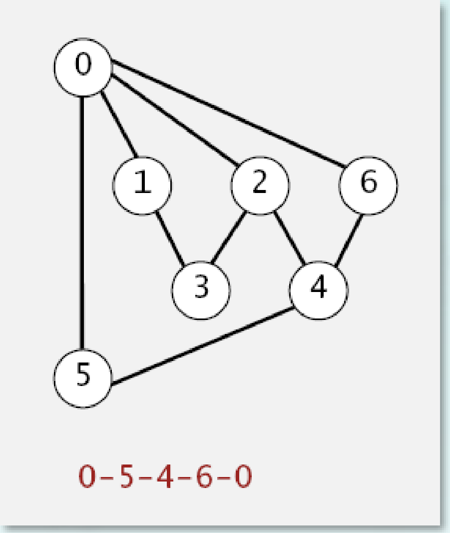
Cycle detection using depth-first search

Problem: Find a cycle.



Cycle detection using depth-first search

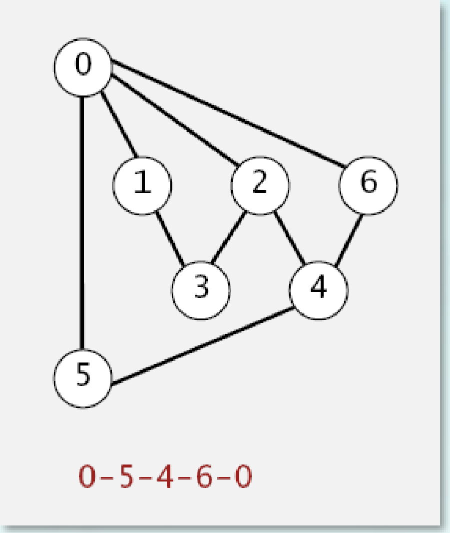
Problem: Find a cycle.



How difficult?

Cycle detection using depth-first search

Problem: Find a cycle.

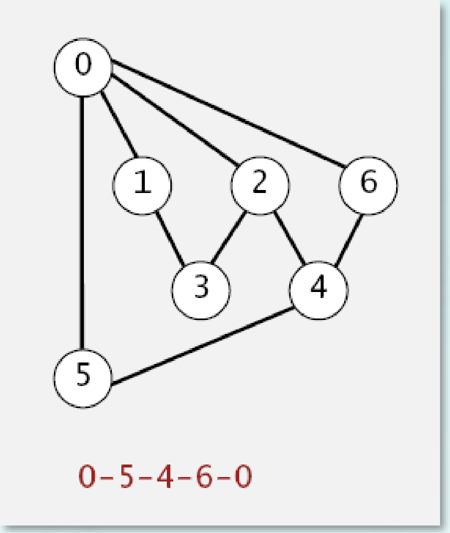


How difficult?

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.



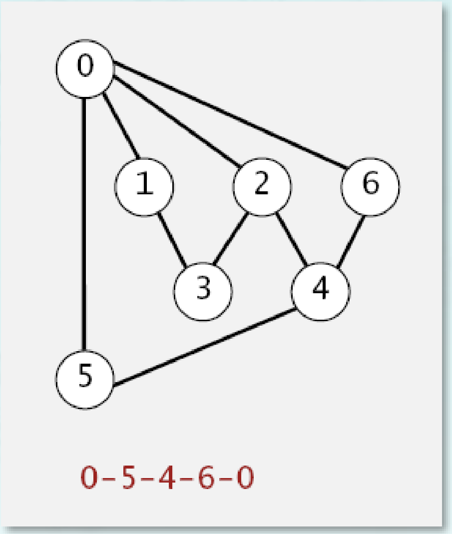
How difficult?

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.

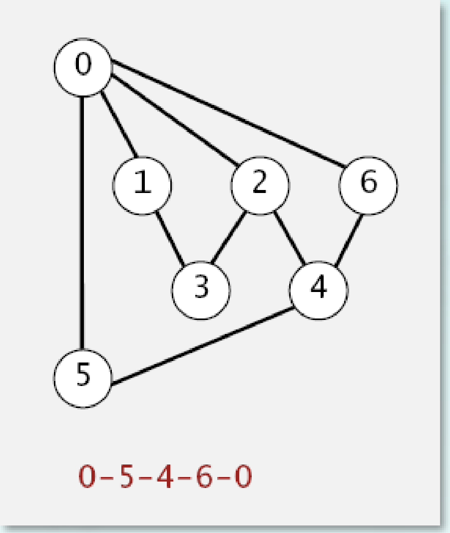


- A *cycle* is a path (with at least one edge) whose first and last vertices are the same.
- A *simple cycle* is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices).

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.

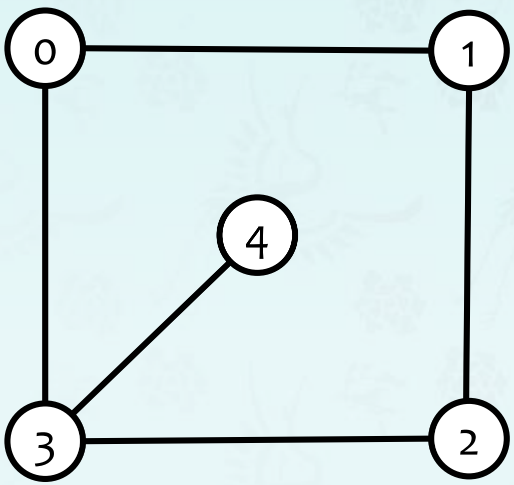


- A *cycle* is a path (with at least one edge) whose first and last vertices are the same.
- A *simple cycle* is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices).

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.



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

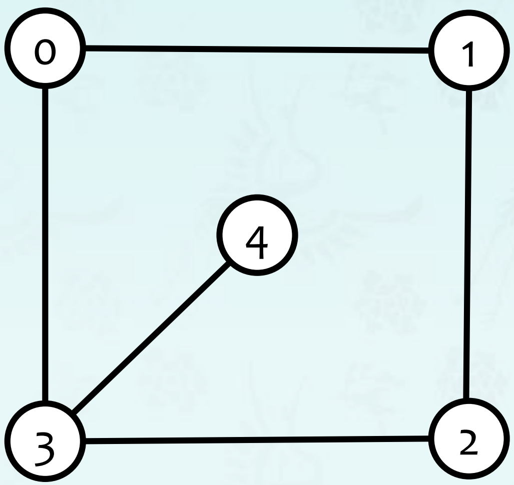
Graph g:

Challenge: build adjacency lists?

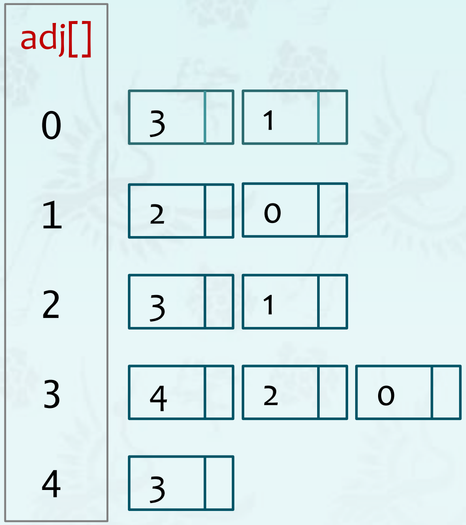
Cycle detection using depth-first search

To visit a vertex v:

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



Adjacency lists



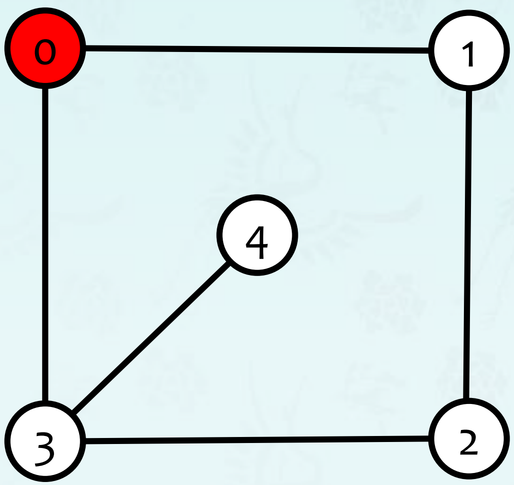

```
myG.txt
5        V
5        E
0 1
0 3
1 2
2 3
3 4
```

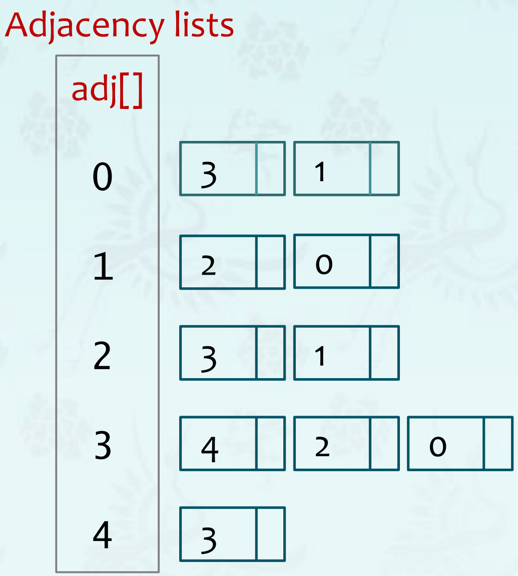
Graph g:

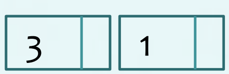
Cycle detection using depth-first search

To visit a vertex v:

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







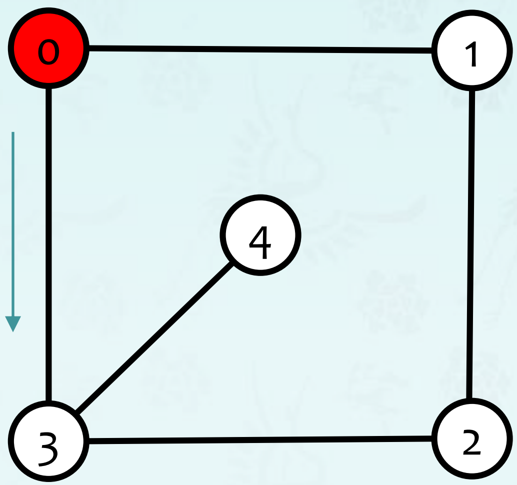
visit 0: check 3, check 1

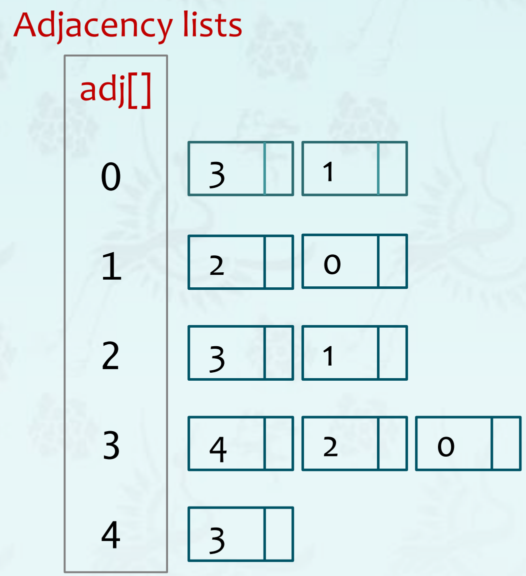
DFS: 0

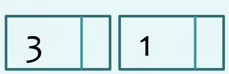
Cycle detection using depth-first search

To visit a vertex v:

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







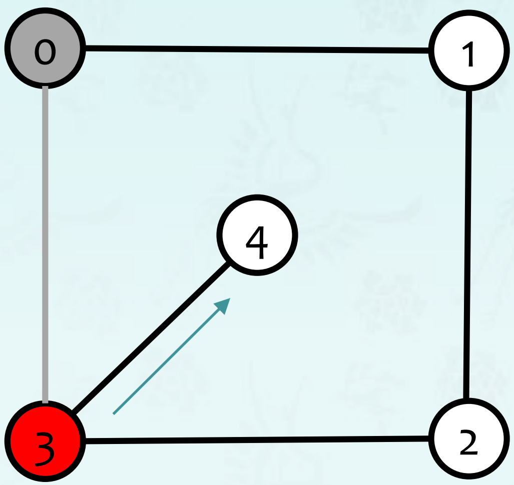
visit 0: check 3, check 1

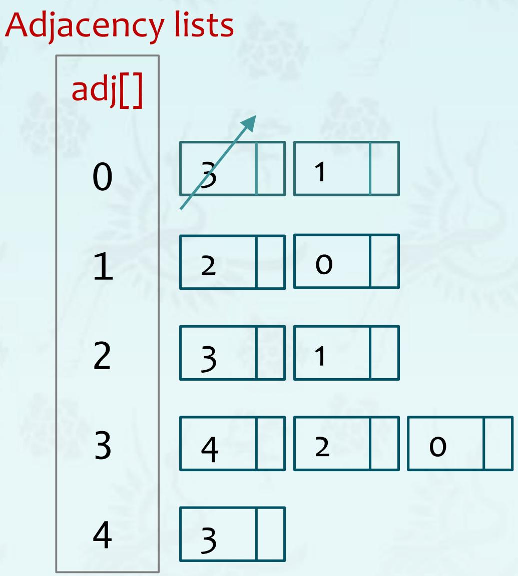
DFS: 0

Cycle detection using depth-first search

To visit a vertex v:

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





4	2	0

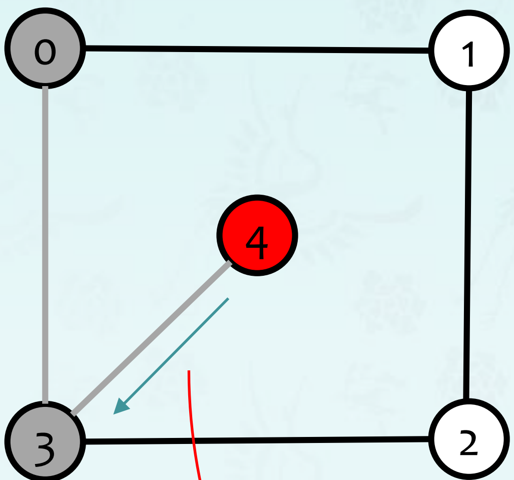
visit 3: check 4, check 2, check 0

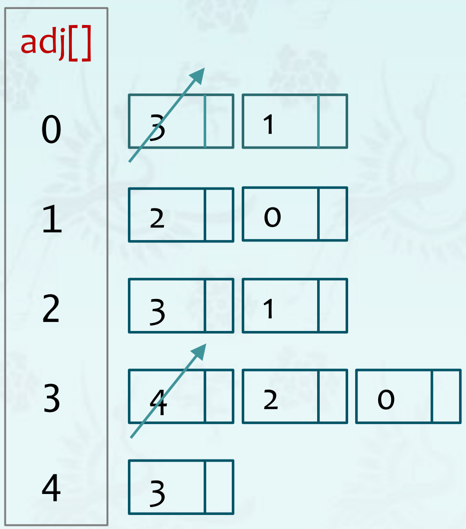
DFS: 0 3

Cycle detection using depth-first search

To visit a vertex v:

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





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

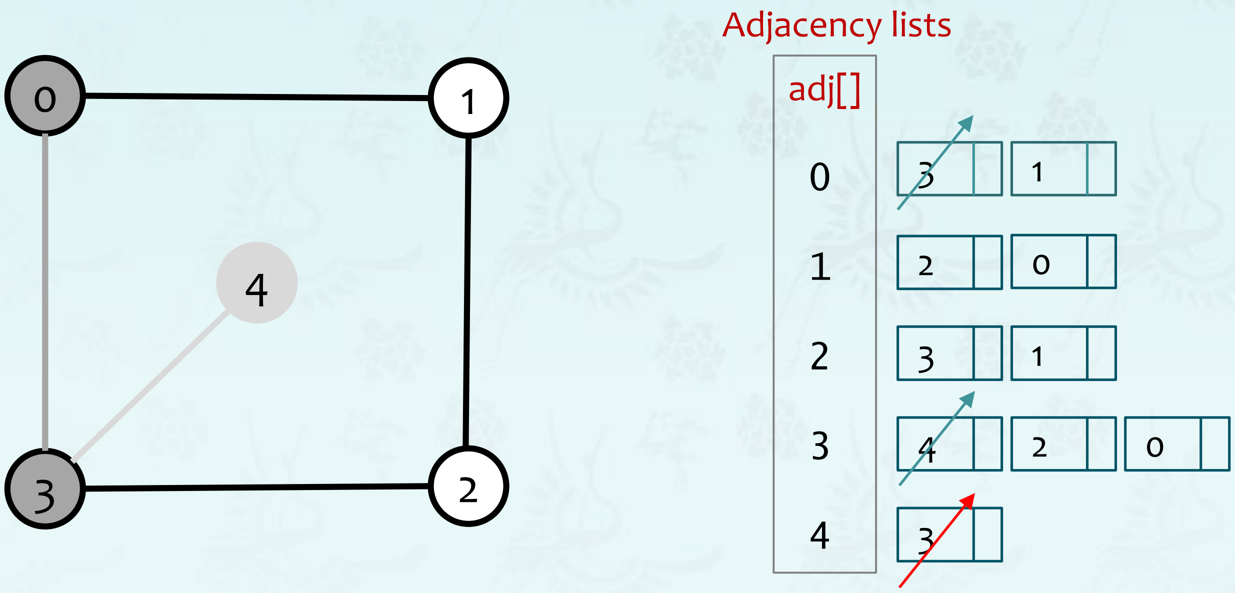
visit 4: check 3

DFS: 0 3 4

Cycle detection using depth-first search

To visit a vertex v:

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



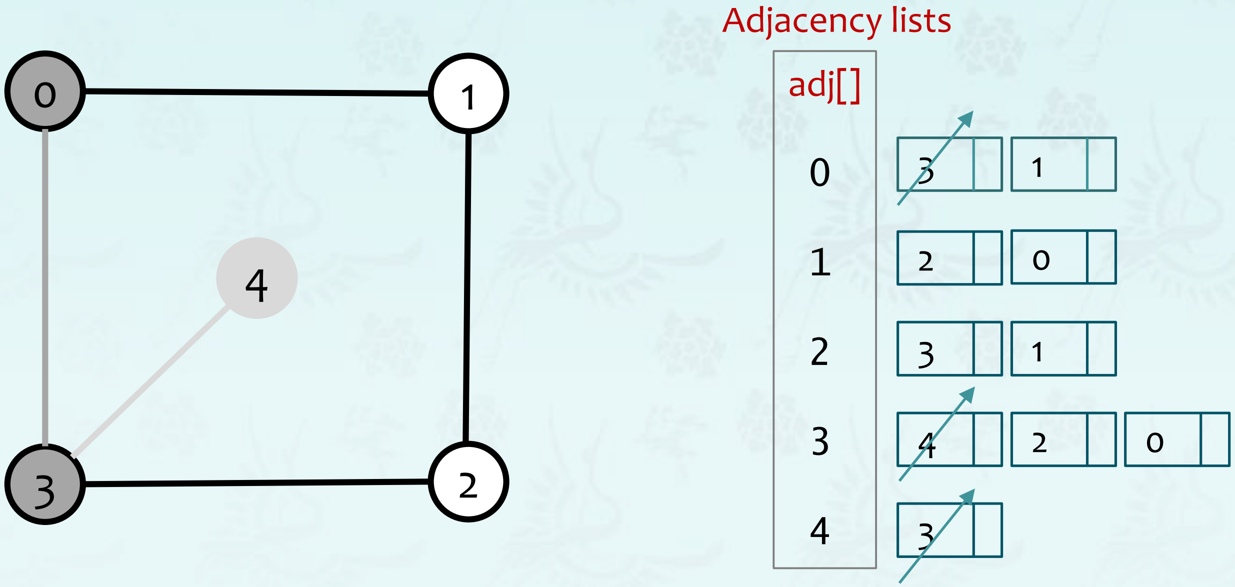
4 done

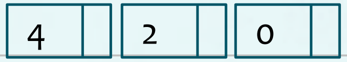
DFS: 0 3 4

Cycle detection using depth-first search

To visit a vertex v:

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





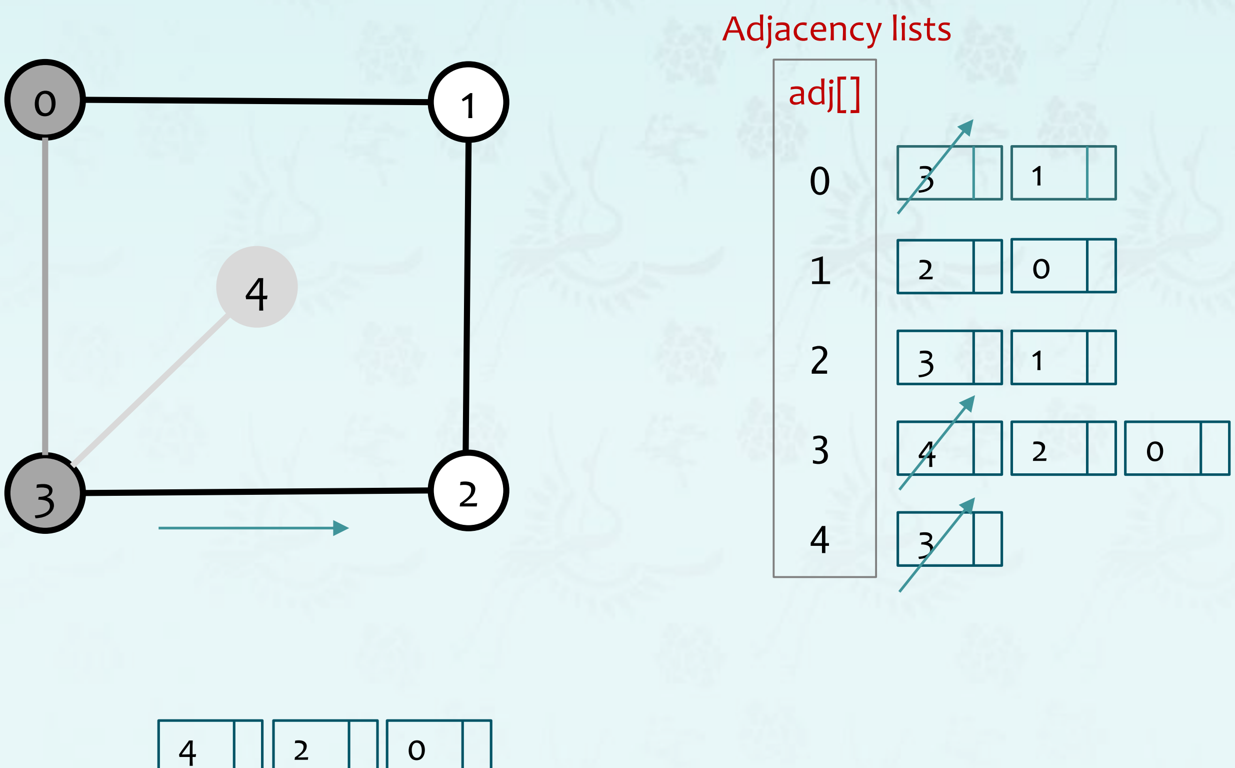
visit 3: check 4, **check 2**, check 0

DFS: 0 3 4

Cycle detection using depth-first search

To visit a vertex v:

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



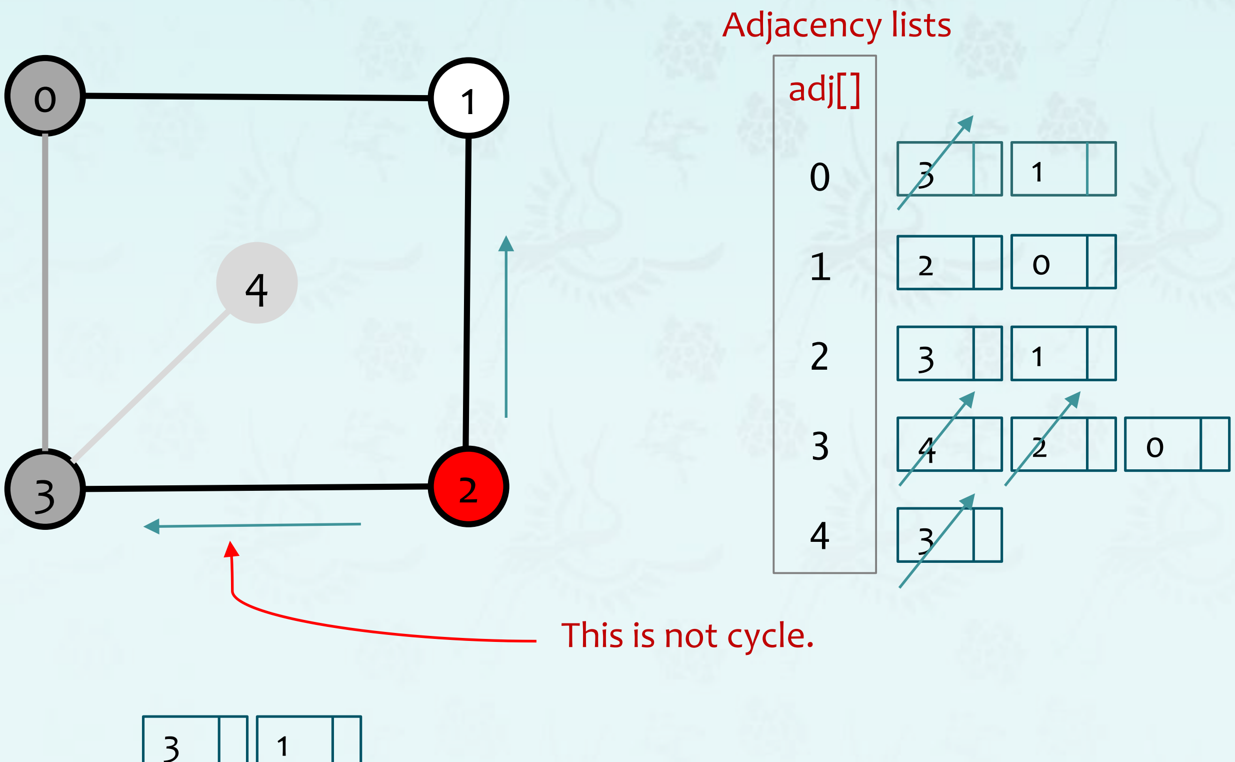
visit 3: check 4, **check 2**, check 0

DFS: 0 3 4

Cycle detection using depth-first search

To visit a vertex v:

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



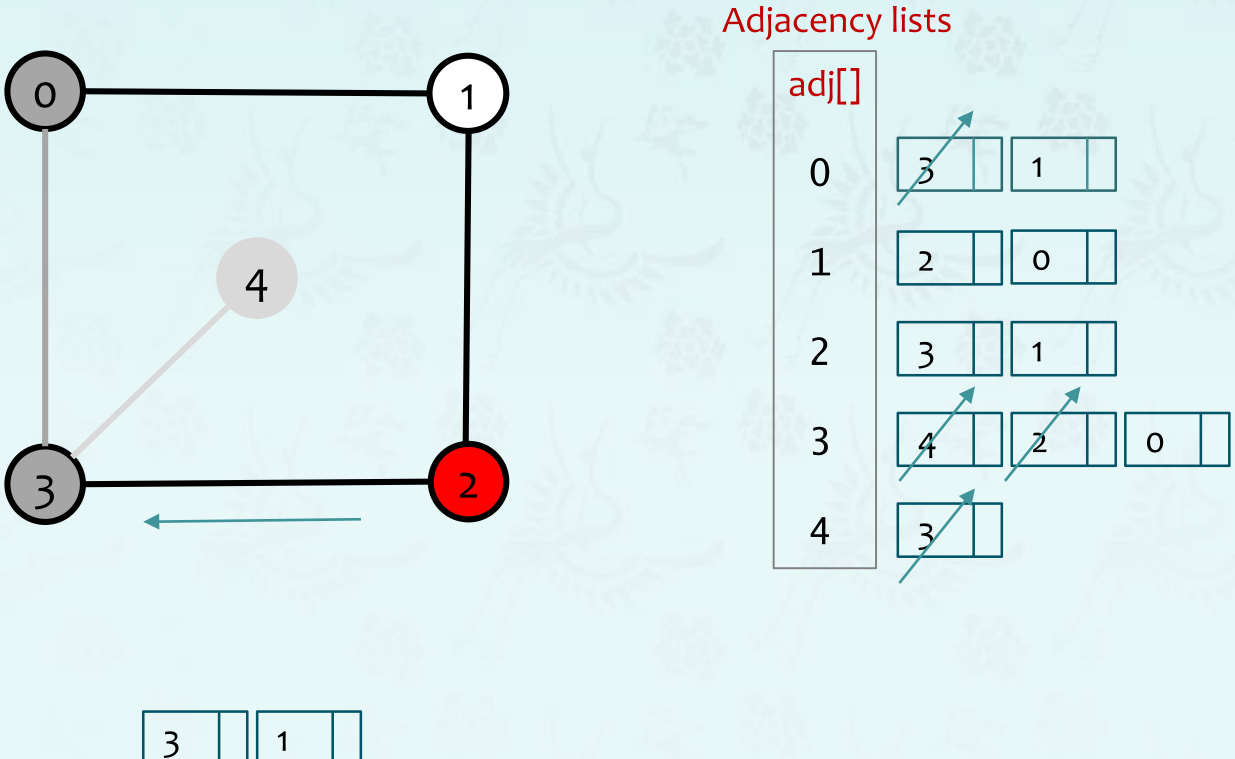
visit 2: **check 3,** check 1

DFS: 0 3 4 2

Cycle detection using depth-first search

To visit a vertex v:

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



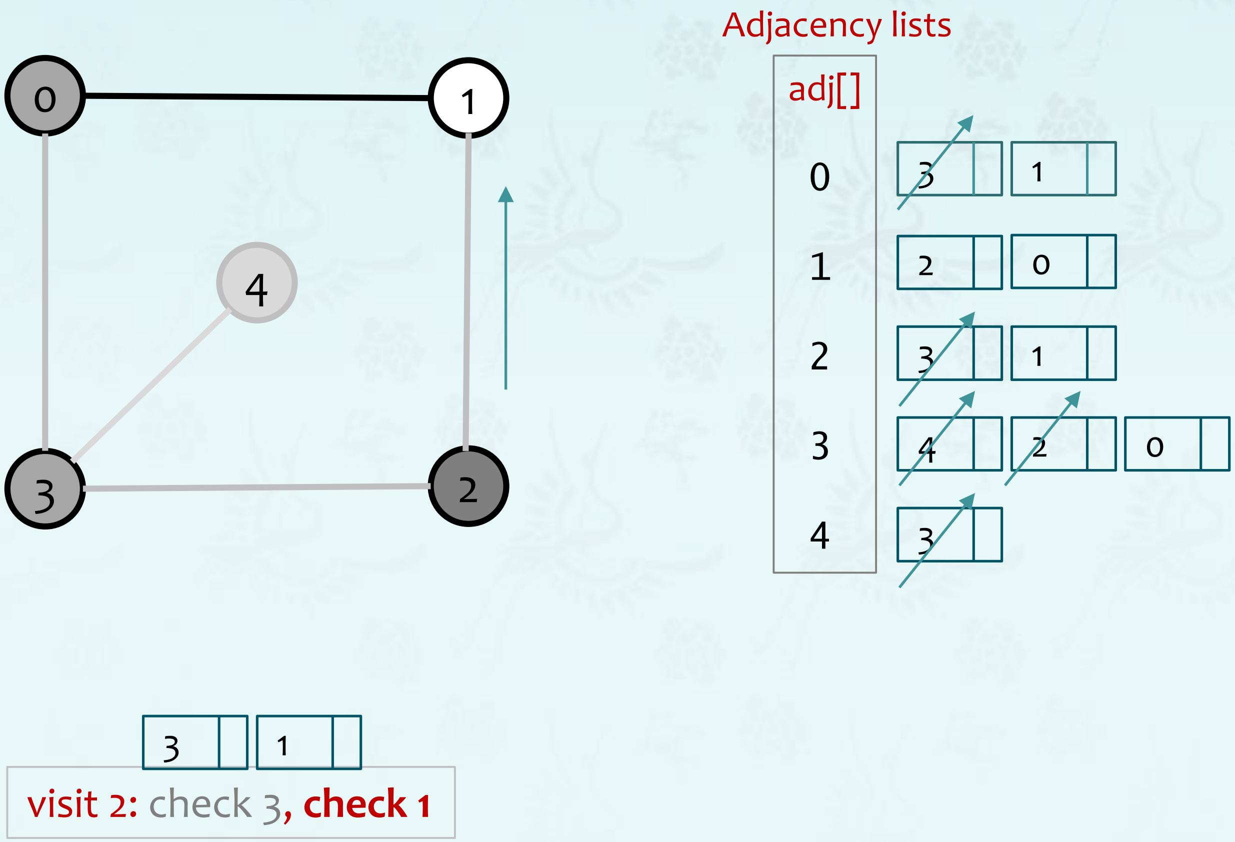
visit 2: **check 3,** check 1

DFS: 0 3 4 2

Cycle detection using depth-first search

To visit a vertex v:

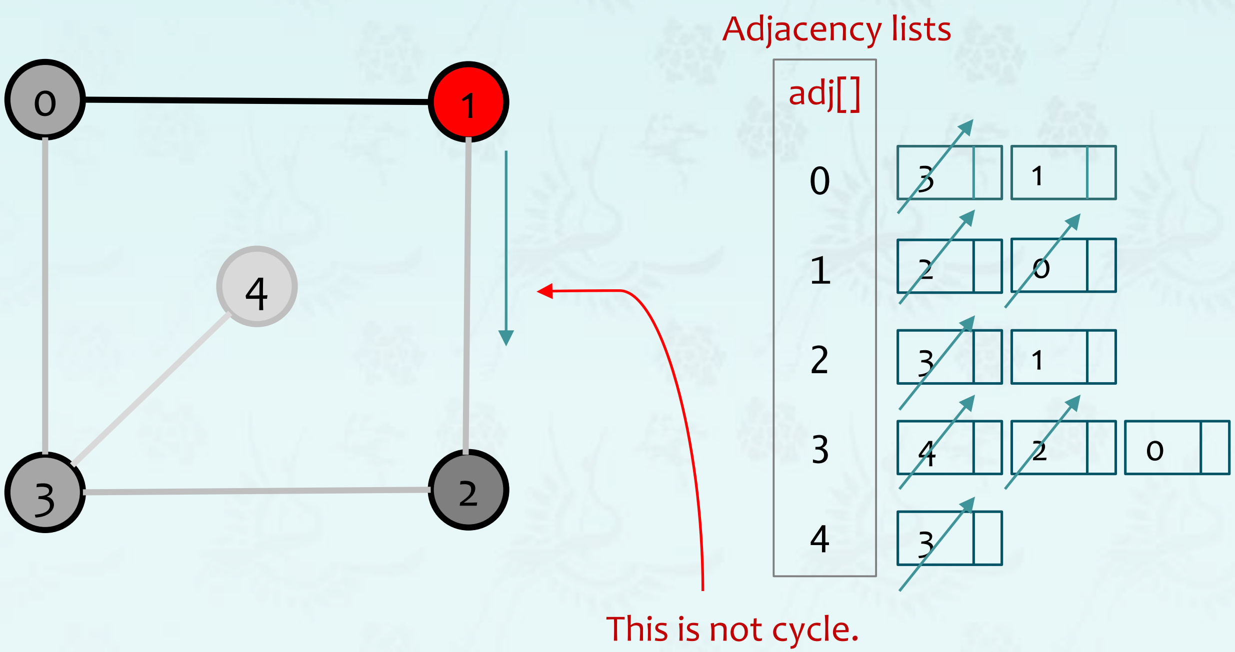
- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Cycle detection using depth-first search

To visit a vertex v:

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



2 0

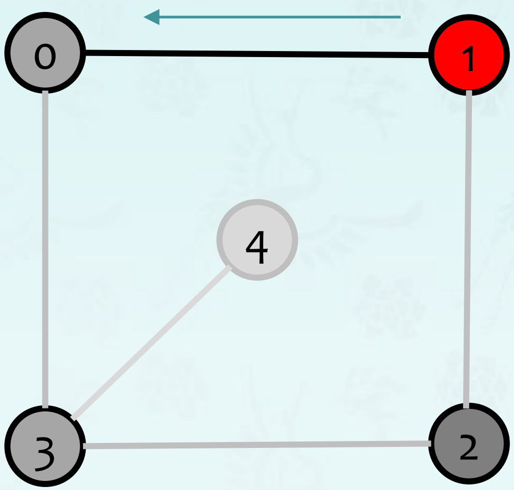
visit 1: check 2, check 0

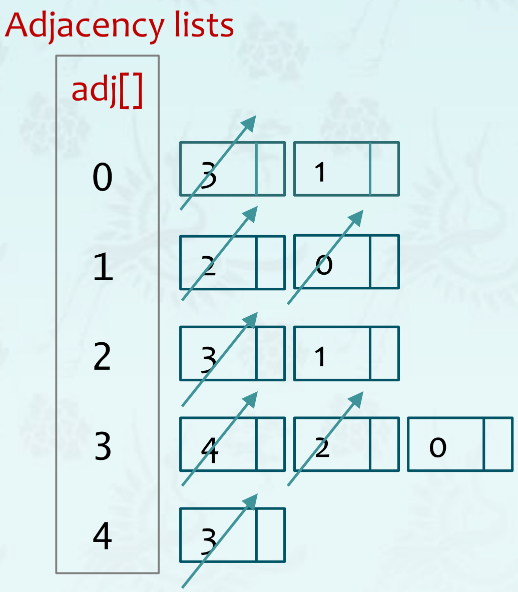
DFS: 0 3 4 2 1

Cycle detection using depth-first search

To visit a vertex v:

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





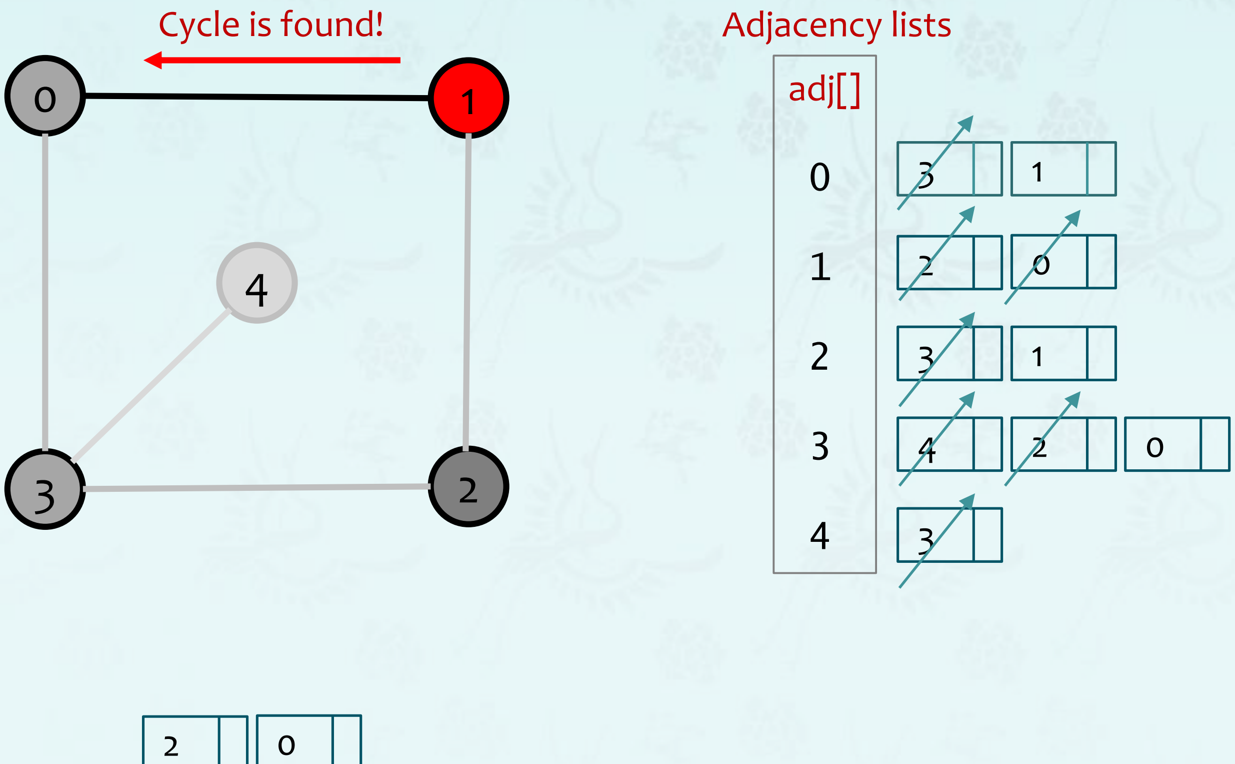
visit 1: check 2, **check 0**

DFS: 0 3 4 2 1

Cycle detection using depth-first search

To visit a vertex v:

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



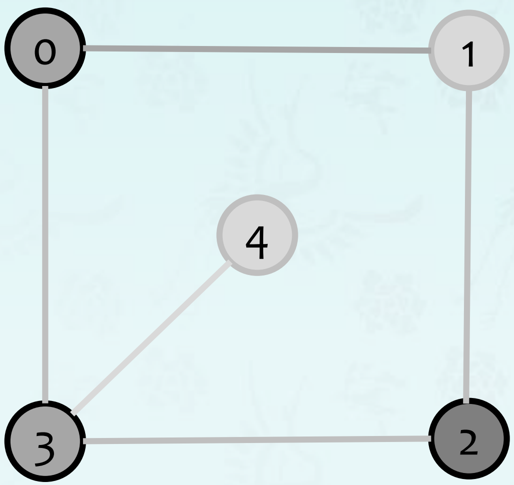
visit 1: check 2, **check 0**

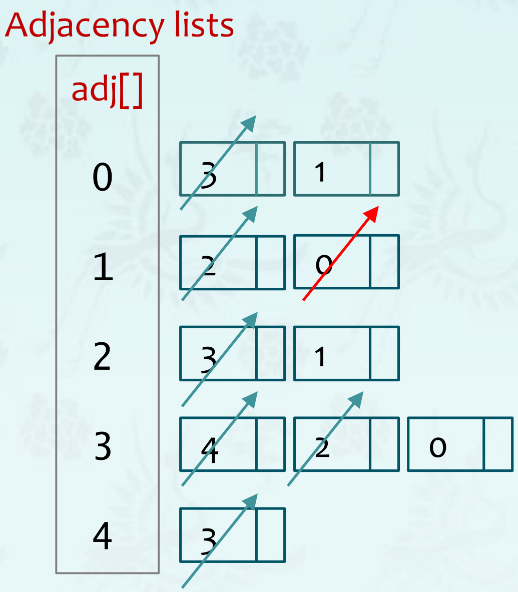
DFS: 0 3 4 2 1

Cycle detection using depth-first search

To visit a vertex v:

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





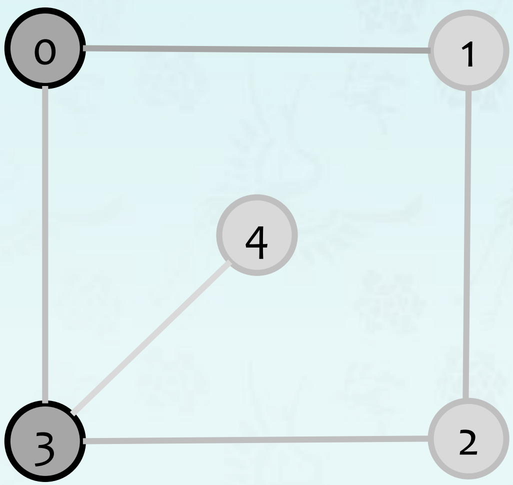
1 done

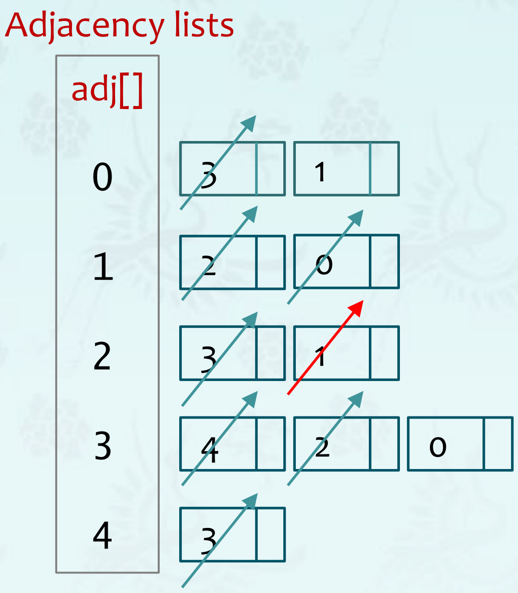
DFS: 0 3 4 2 1

Cycle detection using depth-first search

To visit a vertex v:

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





2 done

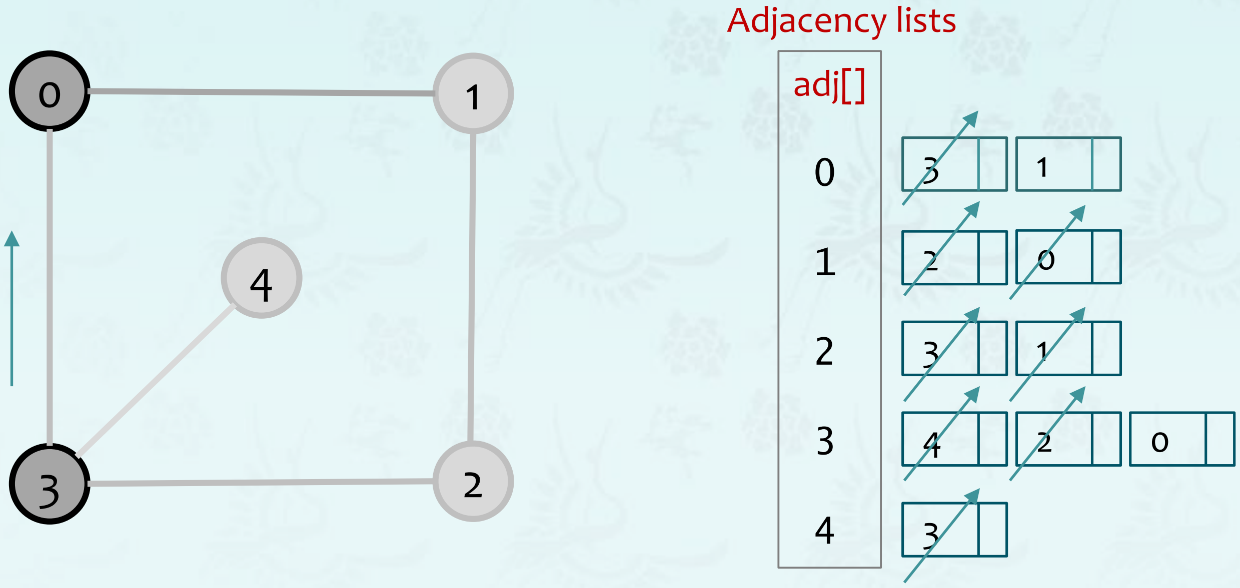
Once 1 done, 2 is done;
since it was recurred from "visit2: check 3, check 1"

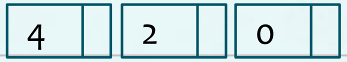
DFS: 0 3 4 2 1

Cycle detection using depth-first search

To visit a vertex v:

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





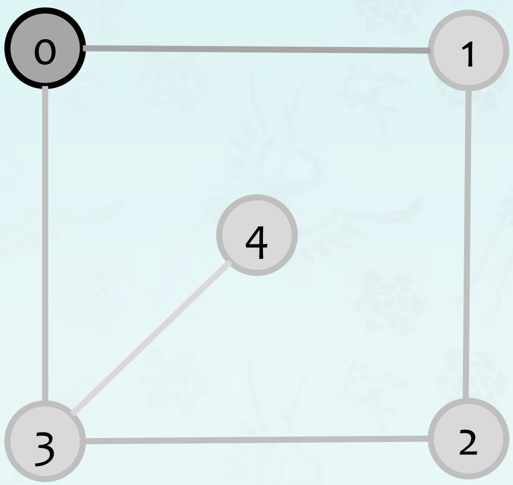
visit 3: check 4, **check 2**, **check 0**

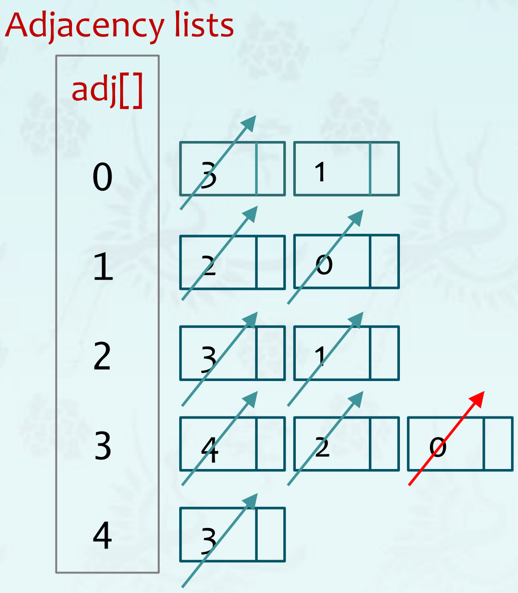
DFS: 0 3 4 2 1

Cycle detection using depth-first search

To visit a vertex v:

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





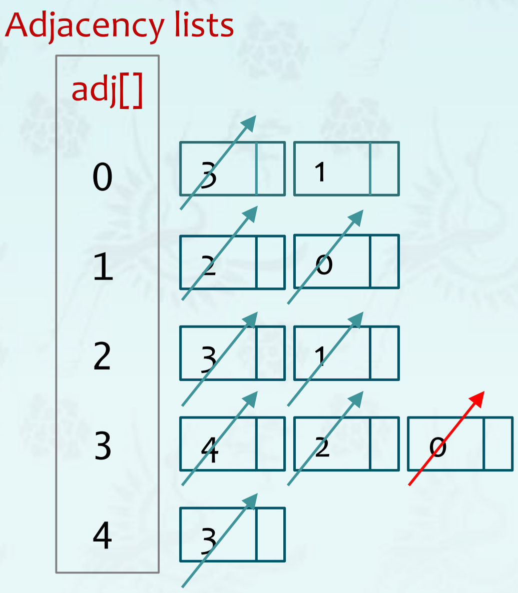
3 done

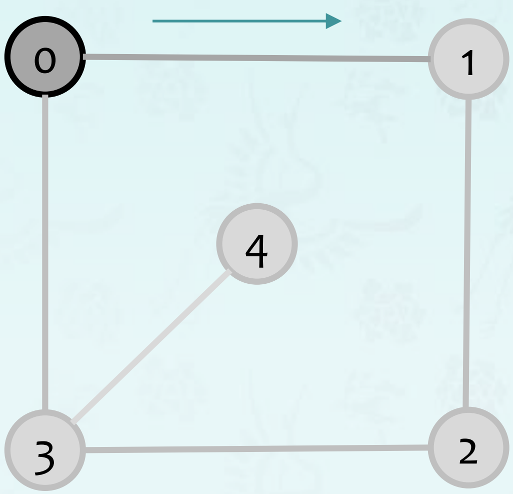
DFS: 0 3 4 2 1

Cycle detection using depth-first search

To visit a vertex v:

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





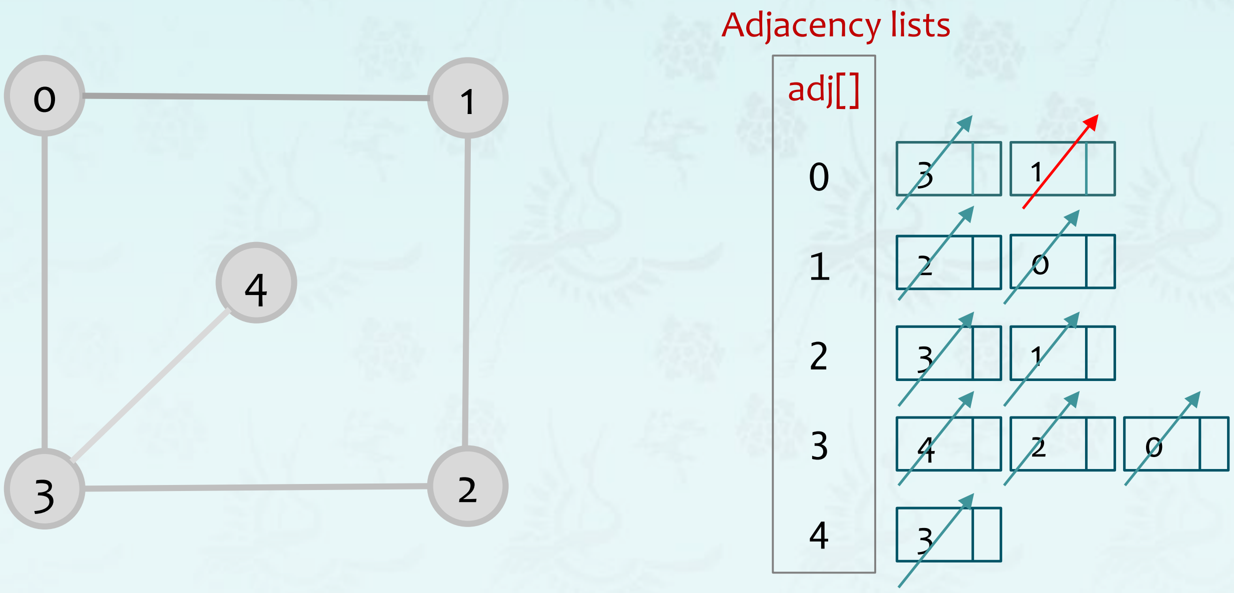
visit 0: check 3, **check 1**

DFS: 0 3 4 2 1

Cycle detection using depth-first search

To visit a vertex v:

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



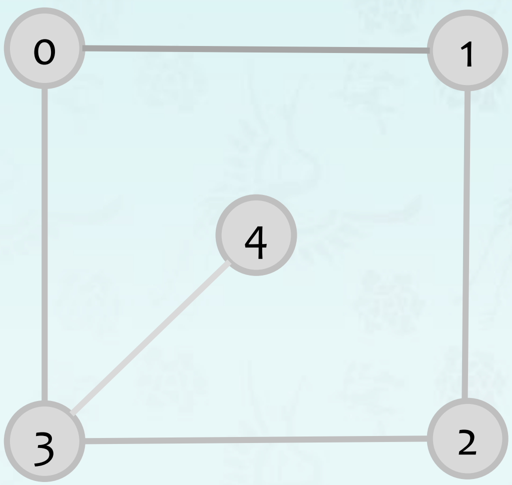
0 done

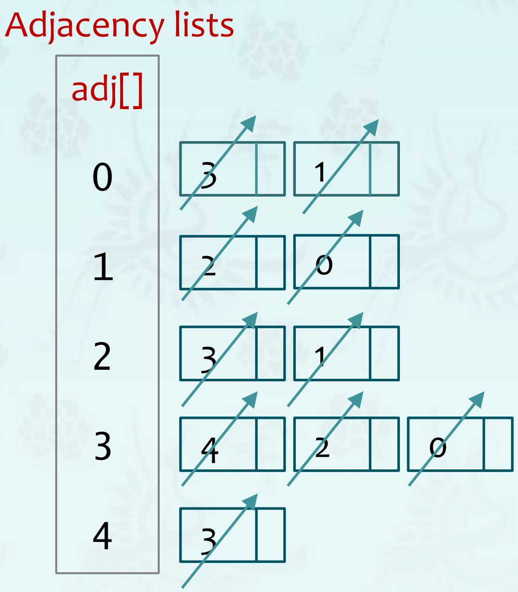
DFS: 0 3 4 2 1

Cycle detection using depth-first search

To visit a vertex v:

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





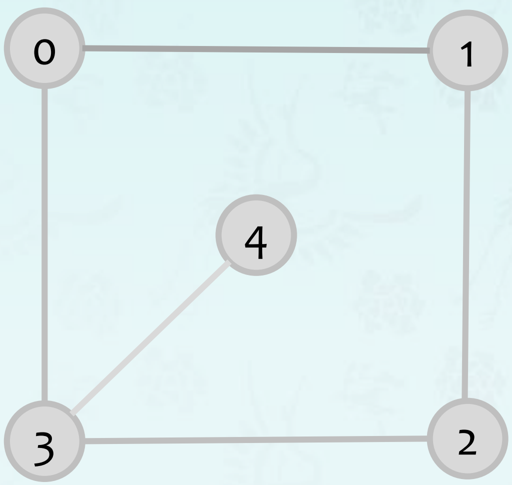
visit(0)
check(3)
visit(3)
check(4)
visit(4)
check(3)
4 done
check(2)
visit(2)
check(3)
check(1)
visit(1)
check(2)
check(0)
1 done
2 done
check(0)
3 done
check(1)
0 done

DFS: 0 3 4 2 1

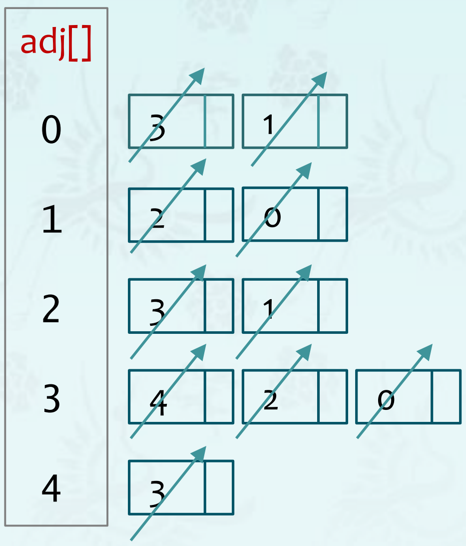
Cycle detection using depth-first search

To visit a vertex v:

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



Adjacency lists

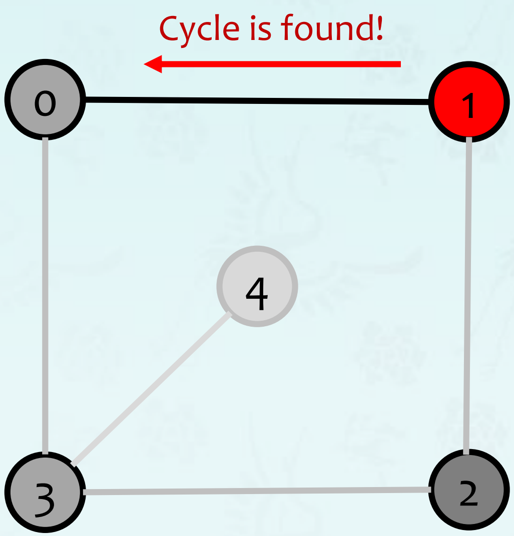


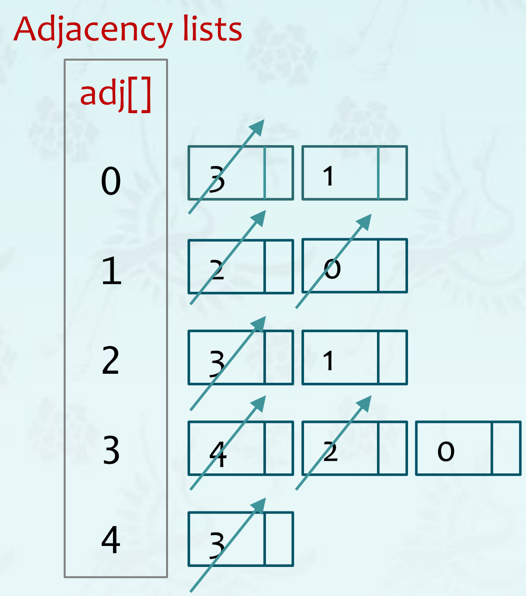
v	marked[]	parent[v]
0	T	-1
1	T	2
2	T	3
3	T	0
4	T	3

DFS: 0 3 4 2 1

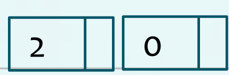
Cycle detection using depth-first search

Cycle is found:





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



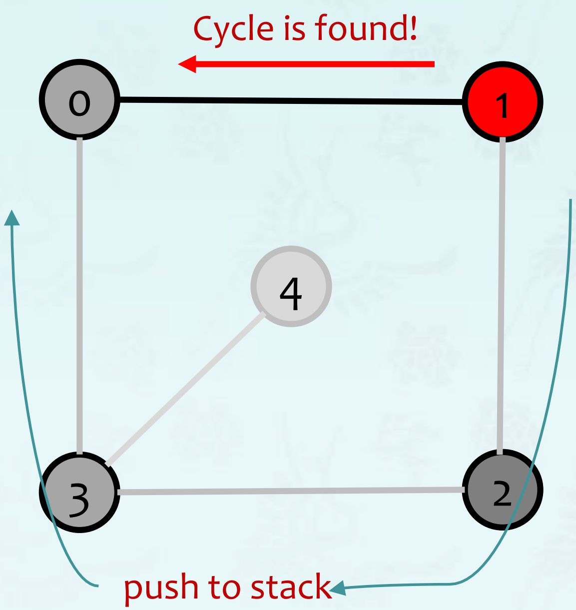
visit 1: check 2, **check 0**

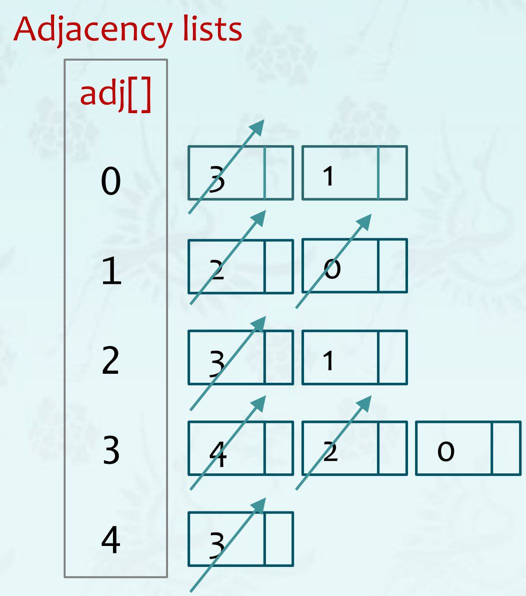
Cycle detection using depth-first search

starting at itself

Cycle is found:

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





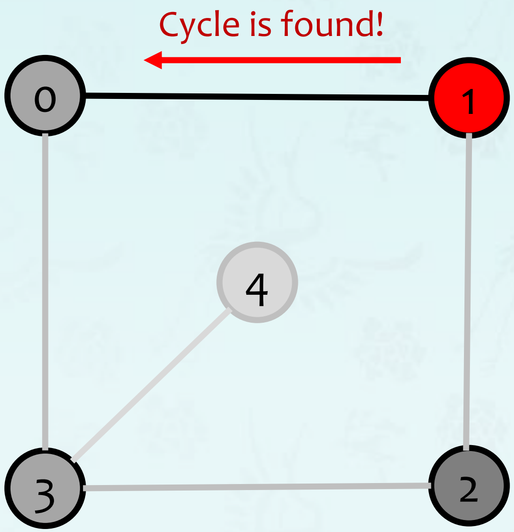
v	marked[]	parent[v]
0	T	-1
1	T	2
2	T	3
3	T	0
4	T	3

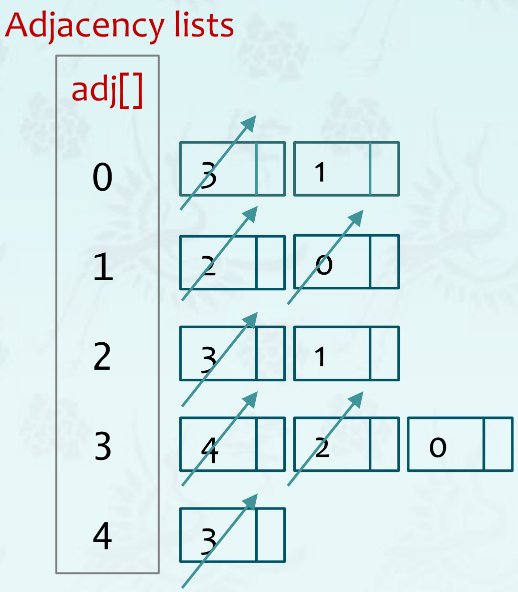
stack top

stack: 3, 2, 1

Cycle is found:

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





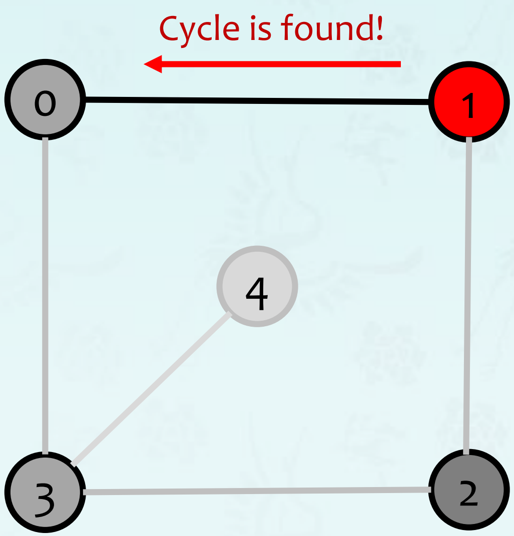
v	marked[]	parent[v]
0	T	-1
1	T	2
2	T	3
3	T	0
4	T	3

stack top

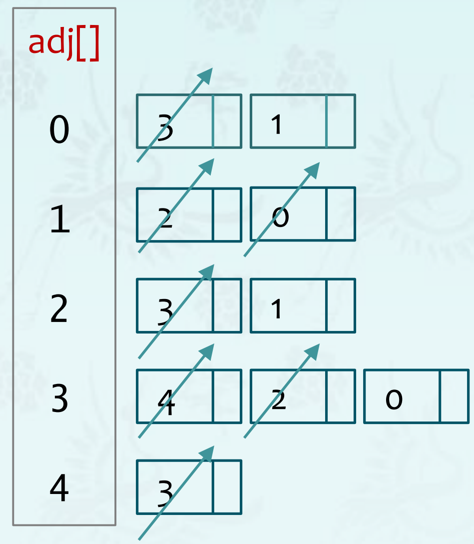
stack: 0, 3, 2, 1 → …. and add myself(1) stack top to make a cycle → stack: 1, 0, 3, 2, 1

Cycle is found:

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



Adjacency lists



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

stack top

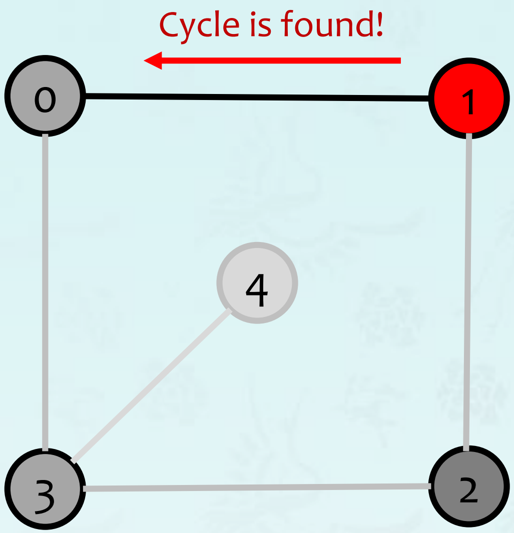
stack: 1, 0, 3, 2, 1

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.

pStack cycle(graph g) {
    g->cycle = NULL;
    if (hasSelfLoop(g)) || (hasParallelEdges(g)) return g->cycle;

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

    for (int v = 0; v < V(g); v++) {
        if ( ! g->marked[v] )    // visit every vertex if not marked.
            cycleDFS(g,  -1,  v);
    }
}
```

why? stay tuned.

Cycle detection using DFS implementation



(pStack) g->cycle : 1 0 3 2 1

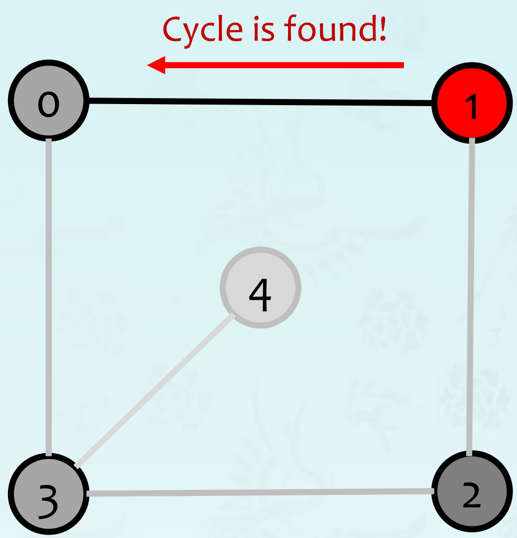
stack top

pop( )

Output:

```
void testGraph() {
    graph g = newGraph(5);
    addEdge(g, 0, 1);    addEdge(g, 0, 3);
    addEdge(g, 1, 2);    addEdge(g, 2, 3);
    addEdge(g, 3, 4);
    printAdjList(g);

    pStack s = cycle(g);

    if (s) {
        printf("There is a cycle: ");
        while (sizeStack(s))
            printf("%d ", pop(s));
        printf("\n");
    }
    else
        printf("This graph is acyclic.\n");
    // do something more ?
    freeGraph(g);
}
```

Cycle detection using DFS implementation



(pStack) g->cycle :

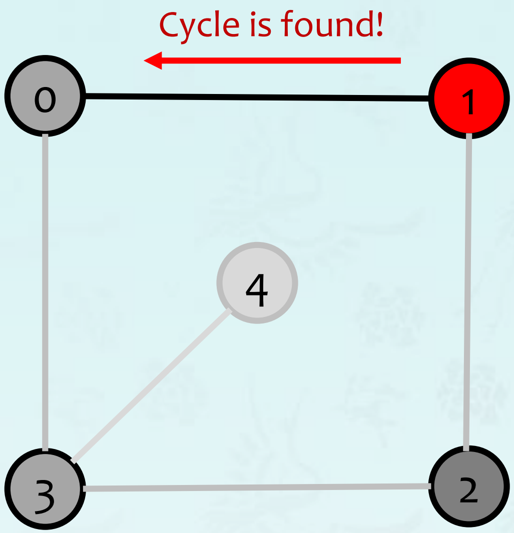
stack top

pop( )

Output: 1 0 3 2 1

```
void testGraph() {
    graph g = newGraph(5);
    addEdge(g, 0, 1);    addEdge(g, 0, 3);
    addEdge(g, 1, 2);    addEdge(g, 2, 3);
    addEdge(g, 3, 4);
    printAdjList(g);

    pStack s = cycle(g);

    if (s) {
        printf("There is a cycle: ");
        while (sizeStack(s))
            printf("%d ", pop(s));
        printf("\n");
    }
    else
        printf("This graph is acyclic.\n");
    // do something more ?
    freeGraph(g);
}
```

Cycle detection using DFS implementation




```
Key pop(pStack s) {
    pNode t = s->top;
    if (t == NULL) return NULL;

    Key key = t->key;
    s->top = t->next;
    free(t);

    s->size--;
    return key;
}
```

(pStack) g->cycle :

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.

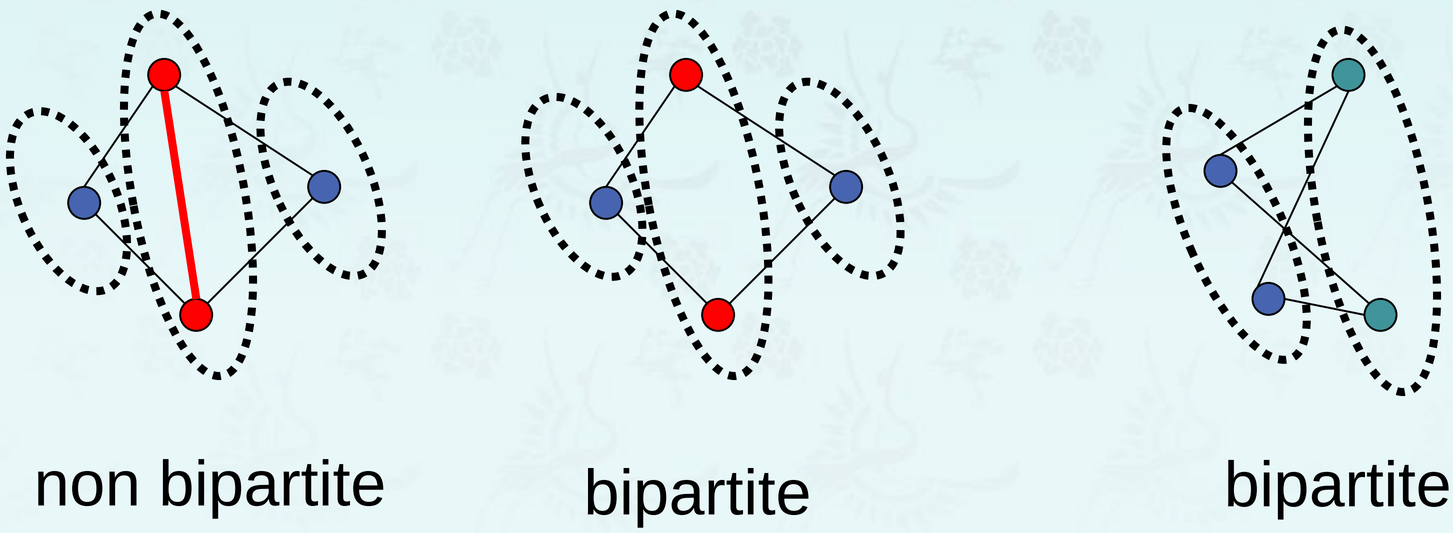
How difficult?

- 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.

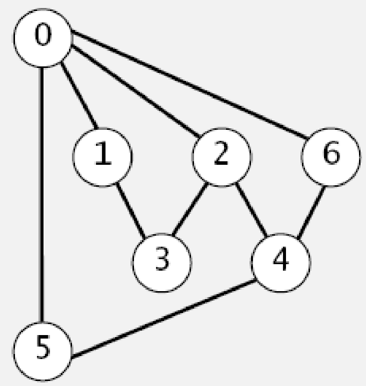


Graph-processing challenge 1 – Review

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 adjacent.

a bigraph ?



How difficult?

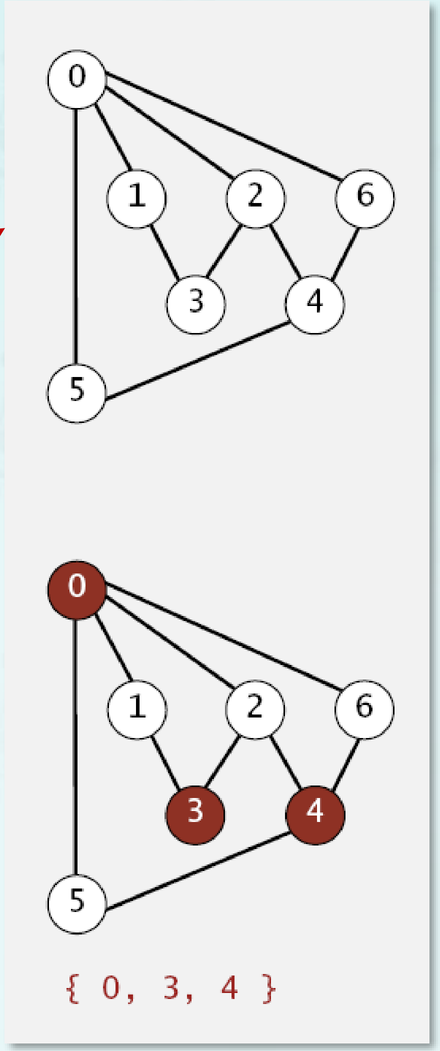
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

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 adjacent.

a bigraph ?

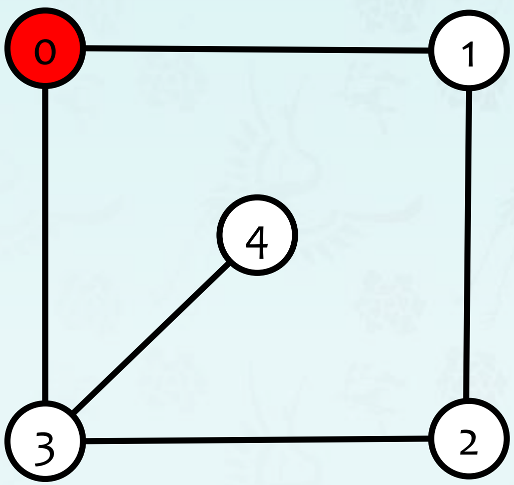


How difficult?

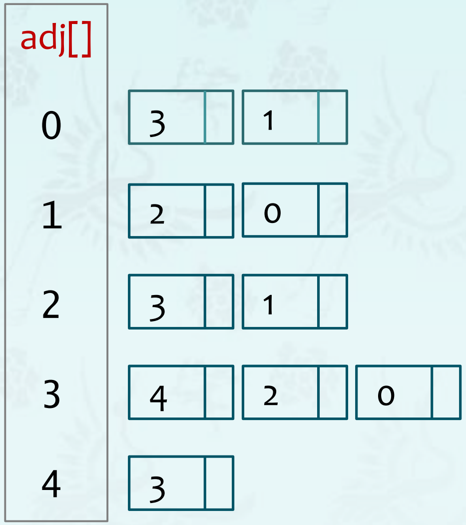
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

simple DFS or BFS-based solution

Problem: Is a graph bipartite (or bigraph)?



Adjacency lists



visit 0: check 3, check 1

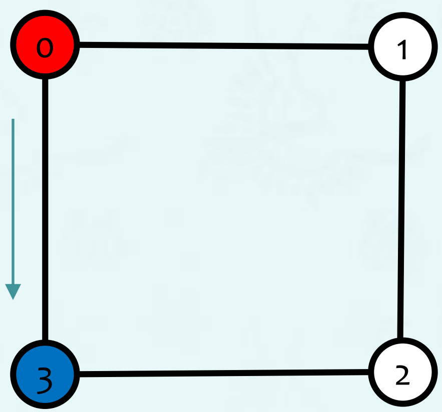
Is a graph bipartite?

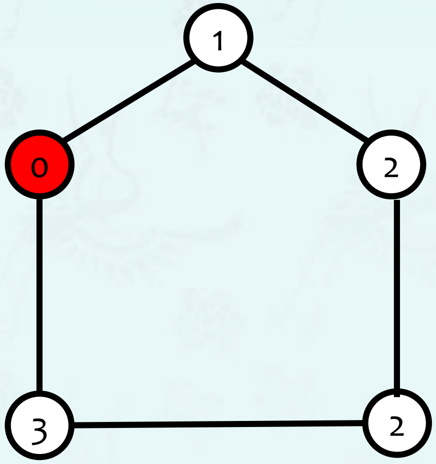
Problem: Is a graph bipartite (or bigraph)?

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.





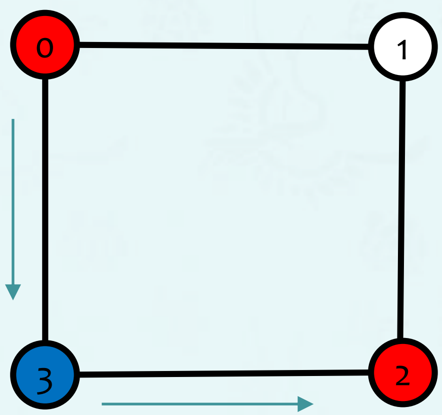
Is a graph bipartite?

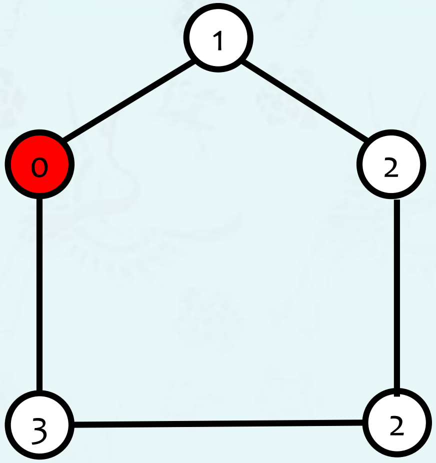
Problem: Is a graph bipartite (or bigraph)?

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.



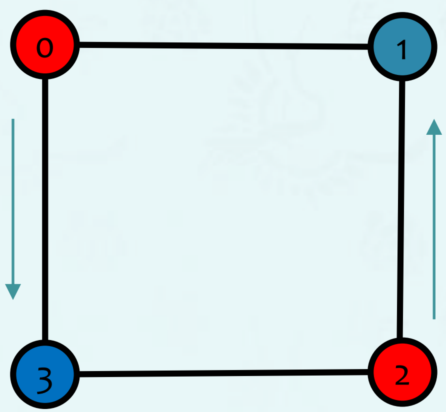


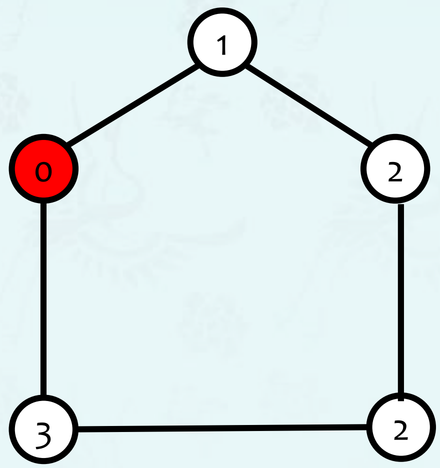
Is a graph bipartite?

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.

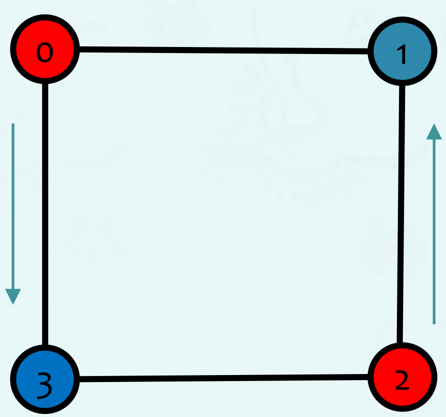




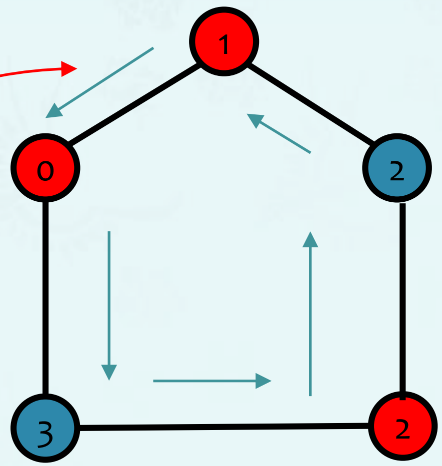
Is a graph bipartite?

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.



Once an odd-length cycle is found, push vertices.



Is a graph bipartite?

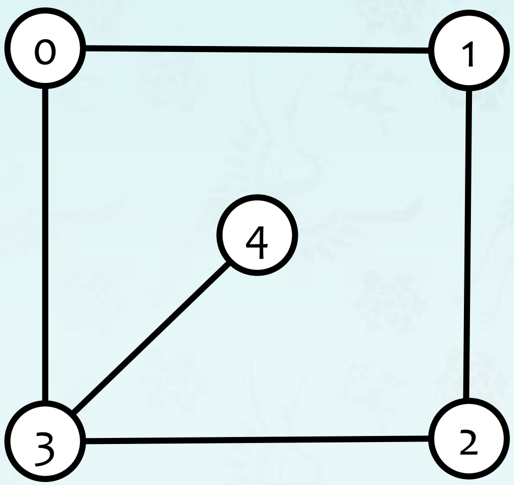
```
// determines whether an undirected graph is bipartite and
// returns g->stack with cyclic vertices pushed if there is a cycle.
void bipartite(graph g) {

    g->cycle = NULL;
    for (int i = 0; i < V(g); i++) {
        g->marked[i] = false;
        g->color[i]  = BLACK;          // BLACK=0, WHITE=1
        g->parent[i] = -1;
    }

    for (int v = 0; v < V(g); v++) {
        if (!g->marked[v]) {
            bipartiteBFS(g, v);
            if (g->cycle != NULL) {
                return g->cycle;  // found 1st cycle
            }
        }
    }
}
```

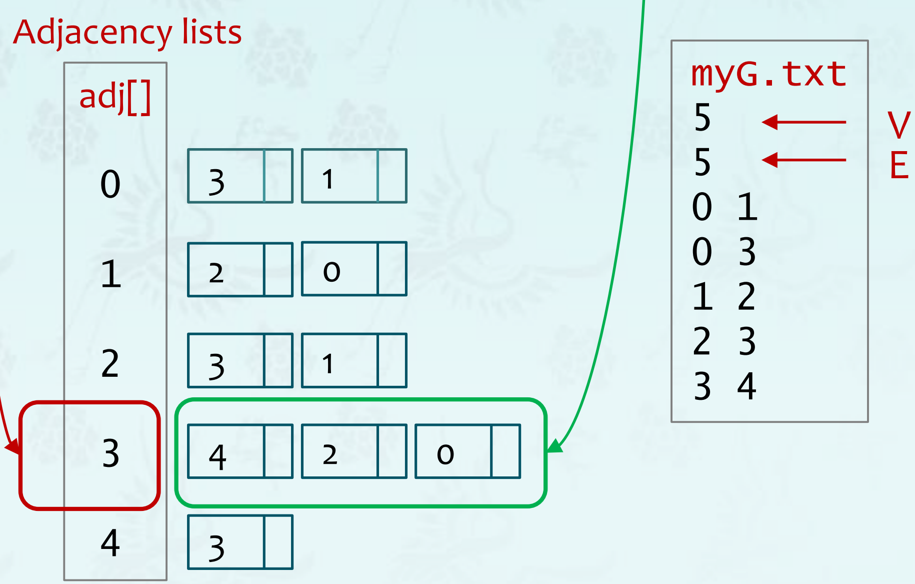
Is a graph bipartite?

```
// Recursive DFS does the work
void bipartiteBFS(graph g, int v) {
   g->marked[v] = true;                          // printf("%d ", v);// visiting node
   for (gnode w = g->adj[v].next; w; w = w->next) {
      if (g->cycle != NULL) return;              // short circuit if odd-length cycle found

      if (!g->marked[w->item]) {                 // found uncolored vertex, so recur
                                                 // keep this info in Graph(save it edgetToBFS[])
                                                 //  switch the color and save
                                                 //  invoke bipartiteBFS()
      }
      // 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)
      }
   }
}
```

verify that a graph is bipartite if it is two-colored.

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

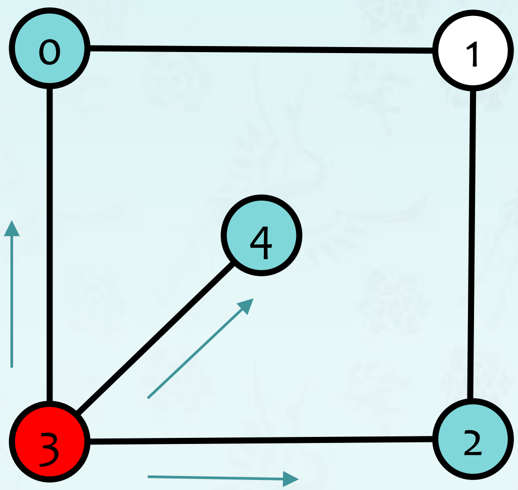


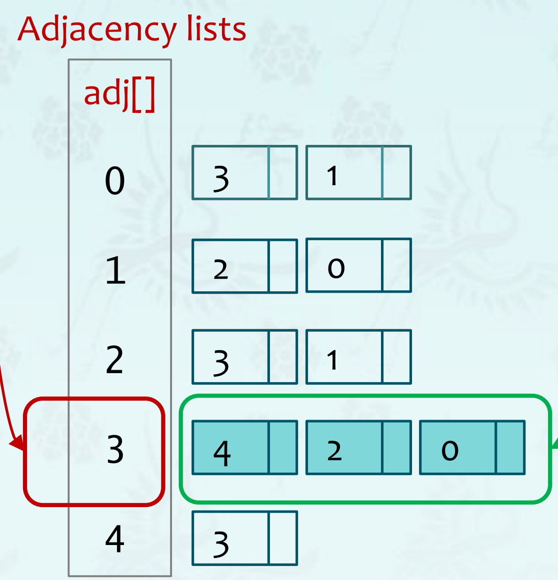


Graph g:

verify that a graph is bipartite if it is two-colored.

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:

Is a graph bipartite?

```
// verify that adj[v]'s color should be different from its adj[v]'s list vertices
// if it is bipartite.

bool bipartiteVerify(graph g) {

    for (int v = 0; v < V(g); v++) {









    }
    return true;
}
```