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)

그러므로 이제 그리스도 예수 안에 있는 자에게는 결코 정죄함이 없나니 이는 그리스도 예수 안에 있는 생명의 성령의 법이 죄와 사망의 법에서 너를 해방하였음이라 (롬5:1)
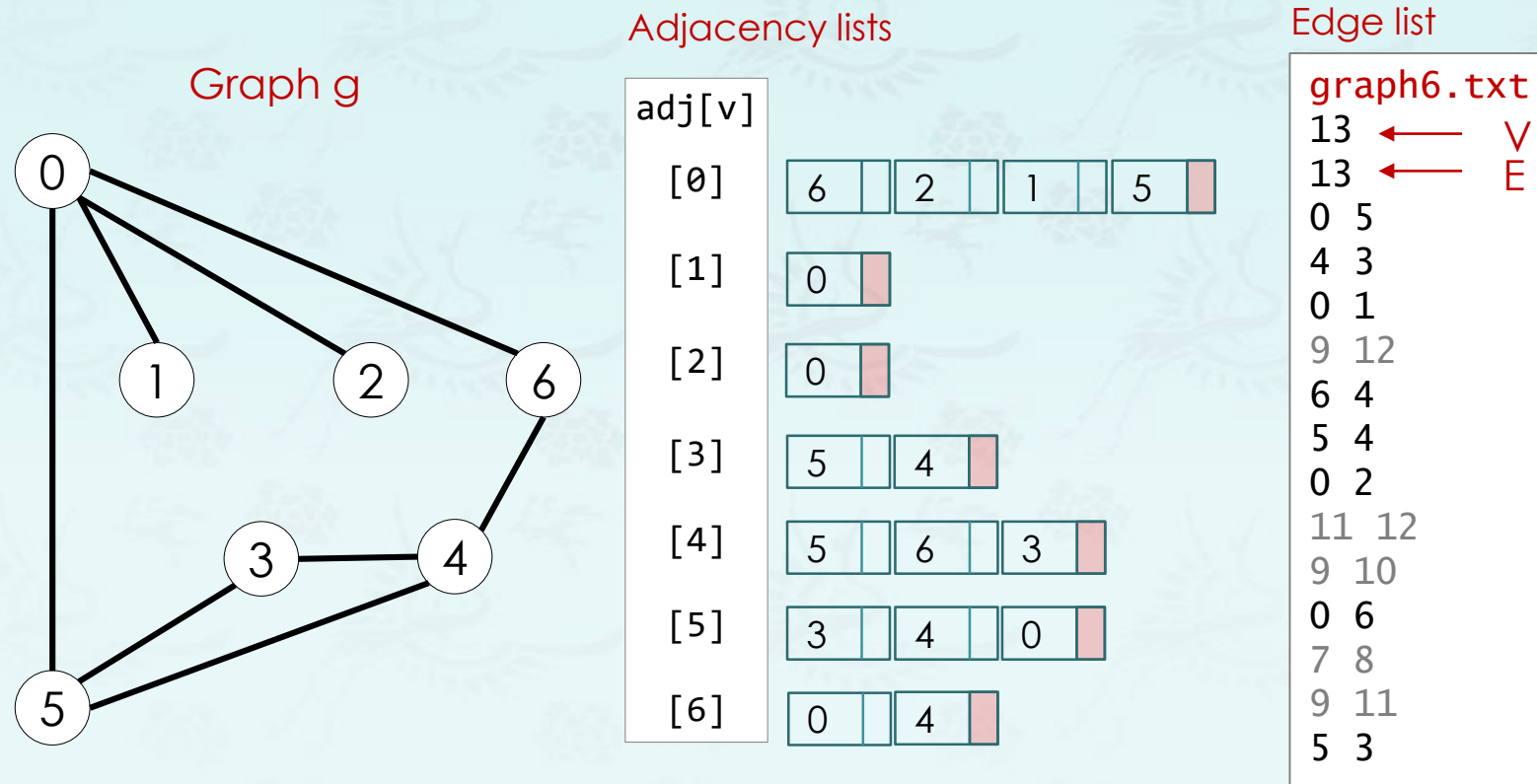
네가 만일 네 입으로 예수를 주로 시인하며 또 하나님께서 그를 죽은 자 가운데서 살리신 것을 네 마음에 믿으면 구원을 받으리라 사람이 마음으로 믿어 의에 이르고 입으로 시인하여 구원에 이르느니라 (롬10:9-10)

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

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

Adjacency list processing

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

Graph g

Adjacency lists

Edge list

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		

```
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
```

Adjacency list processing

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

Adjacency lists

```
// 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;
  }
}
```
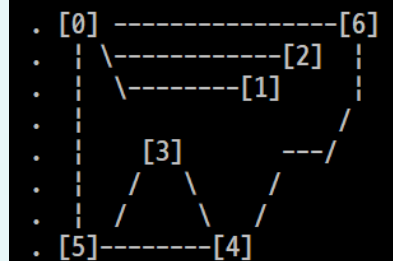
adj[v]

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

```
Adjacency-list:
V[0]: 6 2 1 5
V[1]: 0
V[2]: 0
V[3]: 5 4
V[4]: 5 6 3
V[5]: 3 4 0
V[6]: 0 4
V[7]: 8
V[8]: 7
V[9]: 11 10 12
V[10]: 9
V[11]: 9 12
V[12]: 11 9

. [0] --------------[6]
.  | \-----------[2]  |
.  |  \-------[1]     |
.  |                 /
.  |     [3]      ---/
.  |    /  \    /
.  |   /    \  /
.  | /      \ /
. [5]-------[4]
```

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

4

Adjacency list processing

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

Adjacency lists

```
// 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) {

      ~~
    }
  }
}
```
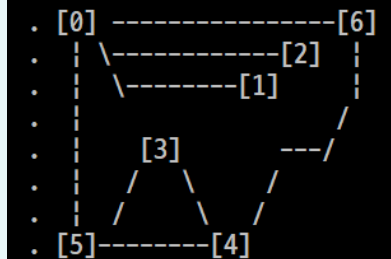
adj[v]

[0] 6 2 1 5

[1] 0

[2] 0

[3] 5 4

[4] 5 6 3

[5] 3 4 0

[6] 0 4

```
Adjacency-list:
V[0]: 6 2 1 5
V[1]: 0
V[2]: 0
V[3]: 5 4
V[4]: 5 6 3
V[5]: 3 4 0
V[6]: 0 4
V[7]: 8
V[8]: 7
V[9]: 11 10 12
V[10]: 9
V[11]: 9 12
V[12]: 11 9

. [0] ---------------[6]
.  | \-----------[2]  |
.  |   \-------[1]    |
.  |                 /
.  |     [3]      ---/
.  |    /  \    /
.  |  /     \  /
. [5]-------[4]
```

Use this style of using **for-loop** when you do the problem set.

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

5

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.
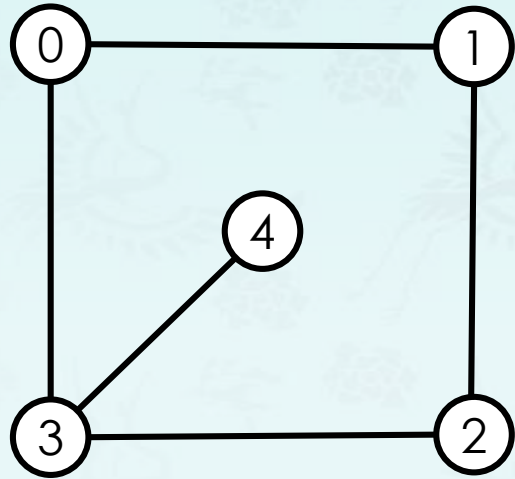5. No one knows.
6. Impossible.

simple DFS-based solution

0-5-4-6-0
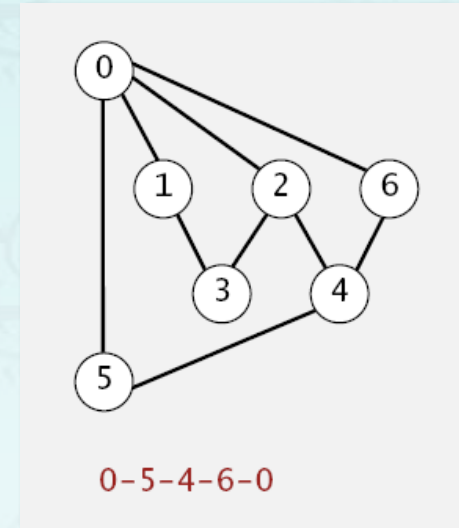
Cycle detection example using depth-first search

To visit a vertex v:

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

Adjacency lists

Graph g:

Challenge: build adjacency lists?

Graph-processing challenge
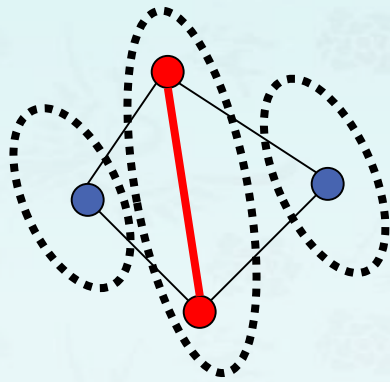
Problem: Is a graph bipartite (or bigraph)?

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

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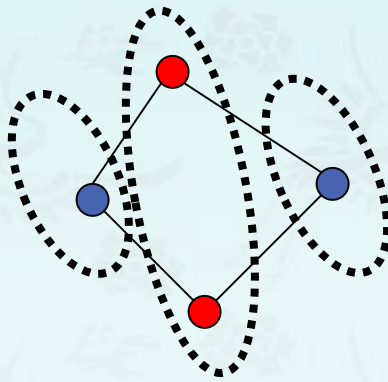4. Intractable.
5. No one knows.
6. Impossible.

Graph-processing challenge

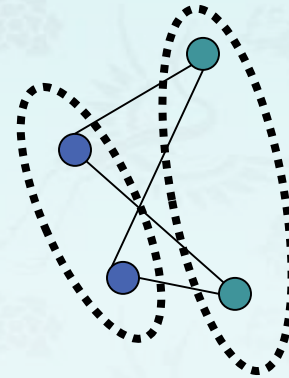**Problem:** Is a graph bipartite (or bigraph)?

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

non bipartite　　　bipartite　　　bipartite
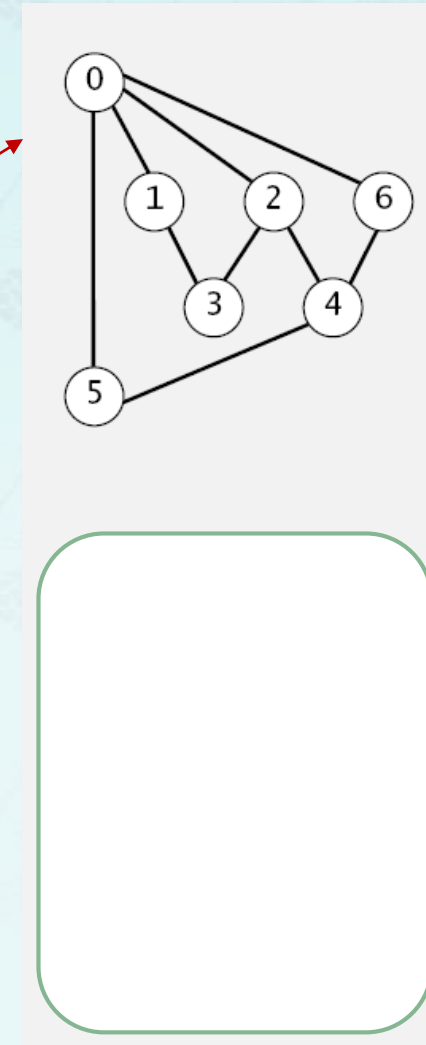
Graph-processing challenge

Problem: Is a graph bipartite (or bigraph)?

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

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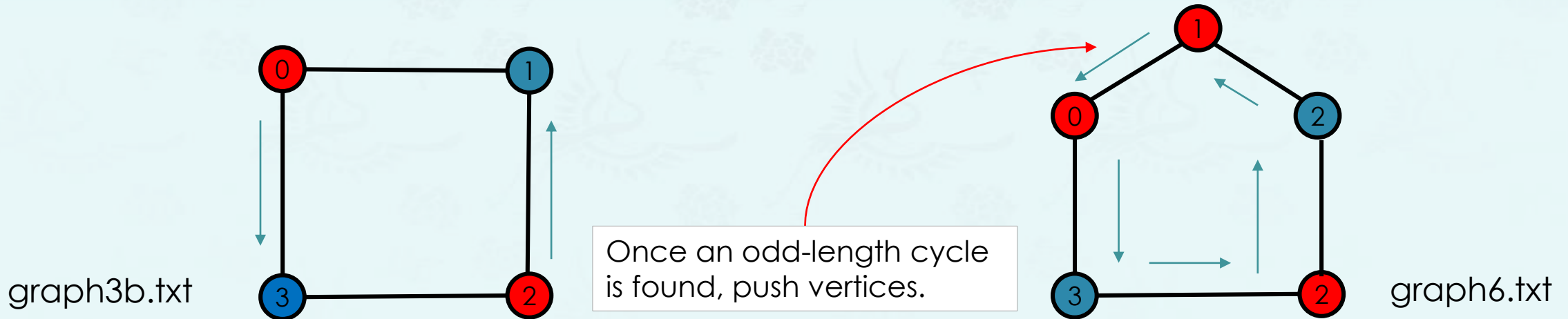5. No one knows.
6. Impossible.

a bigraph ?

Problem: Is a graph bipartite (or bigraph)?

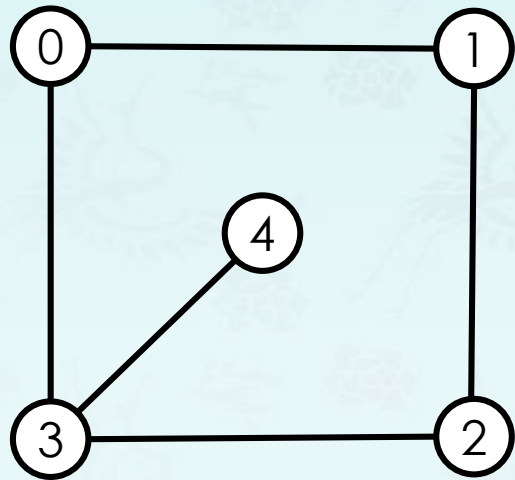**Solution: Two-colorability**

- 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.

graph3b.txt

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

graph6.txt

Bigraph two-colorability

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

Adjacency lists

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

graph3b.txt
```
5       V
5       E
0 1
0 3
1 2
2 3
3 4
```

Graph g:

Bigraph two-colorability

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

Adjacency lists

```
adj[v]
  [0]    3    1
  [1]    2    0
  [2]    3    1
  [3]    4    2    0
  [4]    3
```
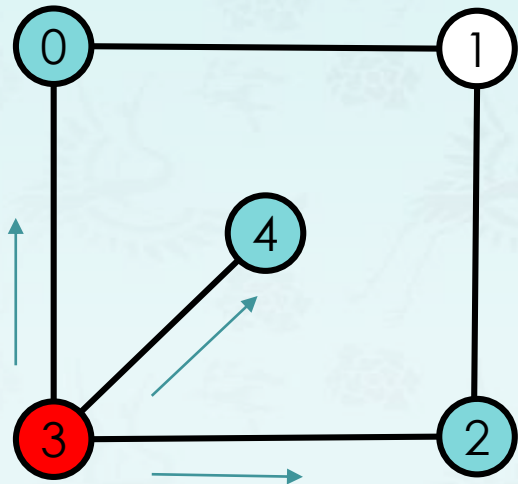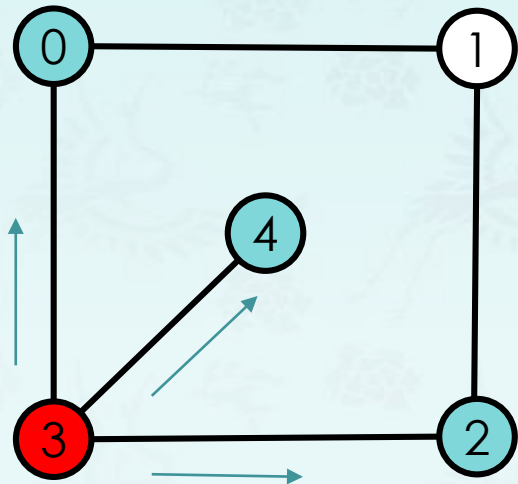
```
graph3b.txt
5        V
5        E
0  1
0  3
1  2
2  3
3  4
```

Graph g:

Bigraph two-colorability

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

Adjacency lists

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

graph3b.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

Graph g:

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

52

```
// helper functions to check two-colorability for bigraph
void init2colorability(graph g) {
  for (int v = 0; v < V(g); v++) g->marked[v] = false;
  g->color[0] = BLACK;            // set starting at v=0, BLACK=0, WHITE=1
}
```

```
// returns true if it is two-colorable, false otherwise
// assuming the graph is colored properly saved in g->color[].
bool check2colorability(graph g) {
  for (int v = 0; v < V(g); v++) {
    for (gnode w = g->adj[v].next; w; w = w->next) {
      // if v and w are the same color, then return false
      if (g->color[v] == g->color[w->item]) return false;
    }
  }
  return true;
}
```

Adjacency lists

adj[v]

[0] 3 1

[1] 2 0

[2] 3 1

[3] 4 2 0

[4] 3

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

53

Graph-processing challenge 2 – bigraph two-colorability

```
// helper functions to check two-colorability for bigraph
void init2colorability(graph g) {
  for (int v = 0; v < V(g); v++) g->marked[v] = false;
  g->color[0] = BLACK;              // set starting at v=0, BLACK=0, WHITE=1
}
```

```
// returns true if it is two-colorable, false otherwise
// assuming the graph is colored properly saved in g->color[].
bool check2colorability(graph g) {
  for (int v = 0; v < V(g); v++) {
    for (gnode w = g->adj[v].next; w; w = w->next) {
      // if v and w are the same color, then return false
      if (g->color[v] == g->color[w->item]) return false;
    }
  }
  return true;
}
```

```
// runs two-coloring using BFS
bool bigraphBFS2Coloring(graph g) {
    if (empty(g)) return true;

    init2colorability(g);
    BFS2Coloring(g);
    return check2colorability(g);
}
```

Graph-processing challenge 2 – bigraph two-colorability

```
// runs two-coloring using BFS - no recursion
void BFS2Coloring(graph g) {
  queue<int> que;
  int v = 0;
  que.push(v);

  // while (!que.empty()) {

    cout << "your code here \n";

  // }

}
```

```
// runs two-coloring using BFS
bool bigraphBFS2Coloring(graph g) {
  if (empty(g)) return true;

  init2colorability(g);
  BFS2Coloring(g);
  return check2colorability(g);
}
```

Graph-processing challenge 2 – bigraph two-colorability

```
// helper functions to check two-colorability for bigraph
void init2colorability(graph g) {
  for (int v = 0; v < V(g); v++) g->marked[v] = false;
  g->color[0] = BLACK;            // set starting at v=0, BLACK=0, WHITE=1
}
```

```
// returns true if it is two-colorable, false otherwise
// assuming the graph is colored properly saved in g->color[].
bool check2colorability(graph g) {
  for (int v = 0; v < V(g); v++) {
    for (gnode w = g->adj[v].next; w; w = w->next) {
      // if v and w are the same color, then return false
      if (g->color[v] == g->color[w->item]) return false;
    }
  }
  return true;
}
```

```
// runs two-coloring using DFS
bool bigraphDFS2Coloring(graph g) {
    if (empty(g)) return true;

    init2colorability(g);
    DFS2Coloring(g, 0);
    return check2colorability(g);
} // sometimes it does not work
```

Graph-processing challenge 2 – bigraph two-colorability

```cpp
// runs two-coloring using DFS recursively
void DFS2Coloring(graph g, int v) {        // DFS
  g->marked[v] = true;                     // v is visited now

  cout << "your code here (recursion)\n";

}
```

```cpp
// runs two-coloring using DFS
bool bigraphDFS2Coloring(graph g) {
  if (empty(g)) return true;

  init2colorability(g);
  DFS2Coloring(g, 0);
  return check2colorability(g);
} // sometimes it does not work
```
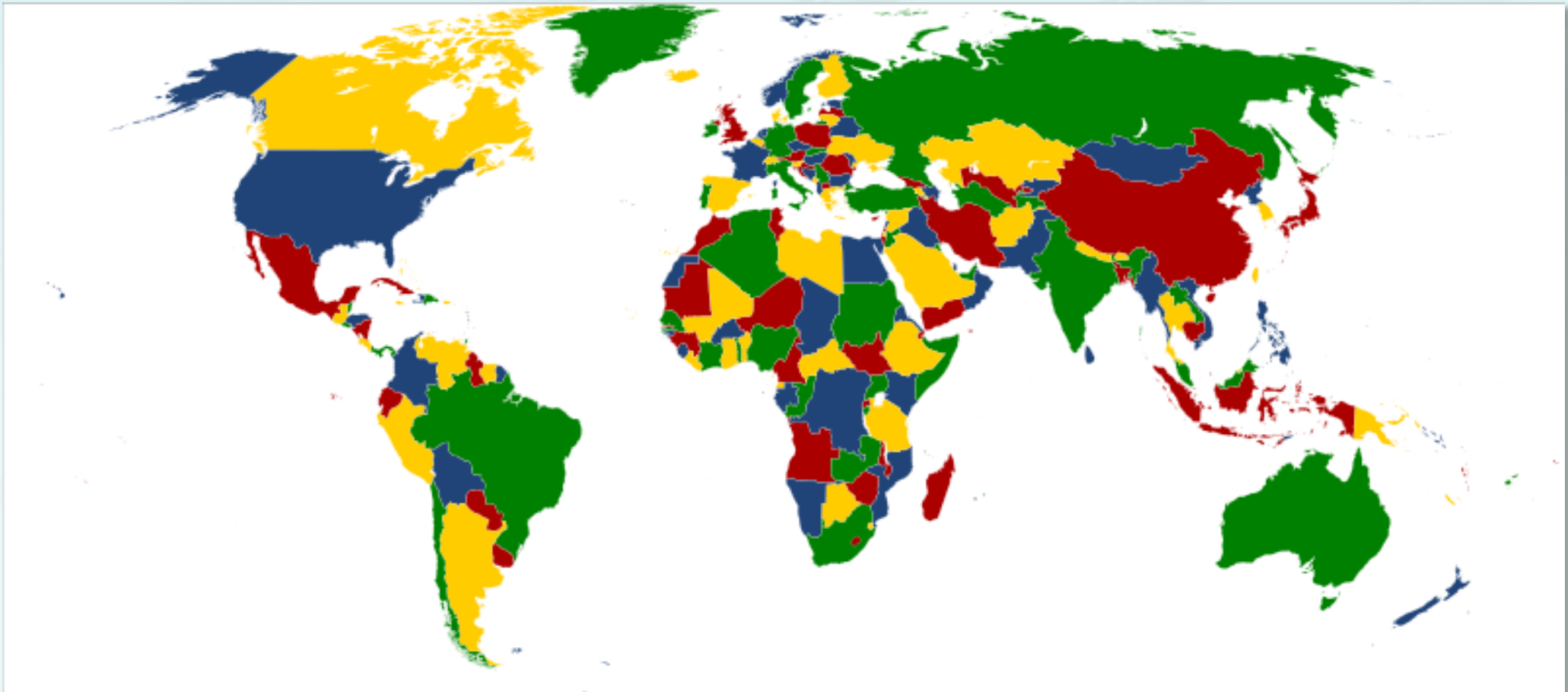
Graph-processing challenge 2 – **Graph Coloring Case Study**

- <u>Akamai</u> runs a network of thousands of servers and the servers are used to distribute content on Internet. They install a new software or update existing software's pretty much every week. The update cannot be deployed on every server at the same time, because the server may have to be taken down for the install. Also, the update should not be done one at a time, because it will take a lot of time. There are sets of servers that cannot be taken down together, because they have certain critical functions.

- This is a typical **scheduling application of graph coloring problem**. It turned out that 8 colors were good enough to color the graph of **75000** nodes.

- So they could install updates in 8 passes.

Graph-processing challenge 2 – **Graph Coloring Case Study**

- Given any separation of a plane into contiguous regions, producing a figure called map, no more than _____ colors are required to color the regions of the map so that no two adjacent regions have the same color.

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

59

Problem: Graph Coloring

- Given a graph G and K colors, assign a color to each node so adjacent nodes get different colors.
- The minimum value of color K which such a coloring exists is the Chromatic Number of G, $\chi(G)$

- **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.

A NP complete problem

Graph-processing challenge 3: Euler tour

▪ **Problem:** The Seven Bridge of Kőnigsberg. [Leonhard Euler 1736]

" …in Königsberg in Prussia, there is an island A, called the Kneiphof; the river which surrounds it is divided into two branches … and these branches are crossed by seven bridges. Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he could cross each bridge once and only once. "
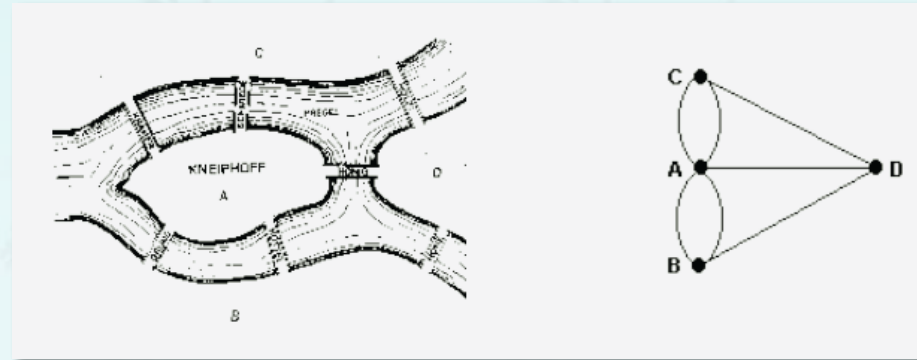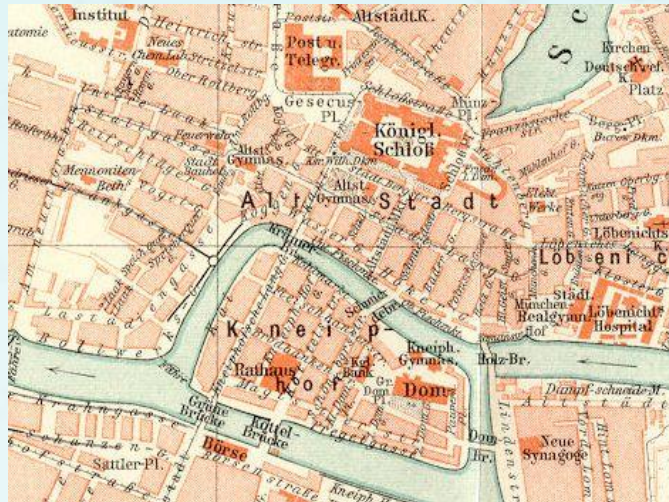

Euler tour: Is there a (general) cycle that uses each **edge** exactly once?
**Answer:** A connected graph is Eulerian iff all vertices have **even** degree.
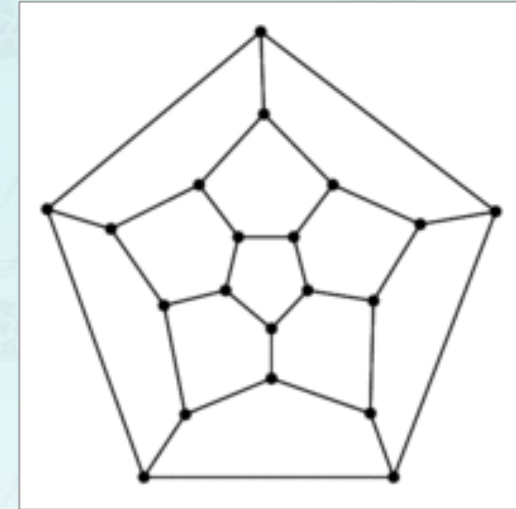
- **Problem:** Find a cycle that visits every **vertex exactly once**.
- **How difficult?** **Euler tour**:

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.

Eulerian tour
(classic graph-processing problem)

0-1
0-2
0-5
0-6
1-2
2-3
2-4
3-4
4-5
4-6

0-1-2-3-4-2-0-6-4-5-0
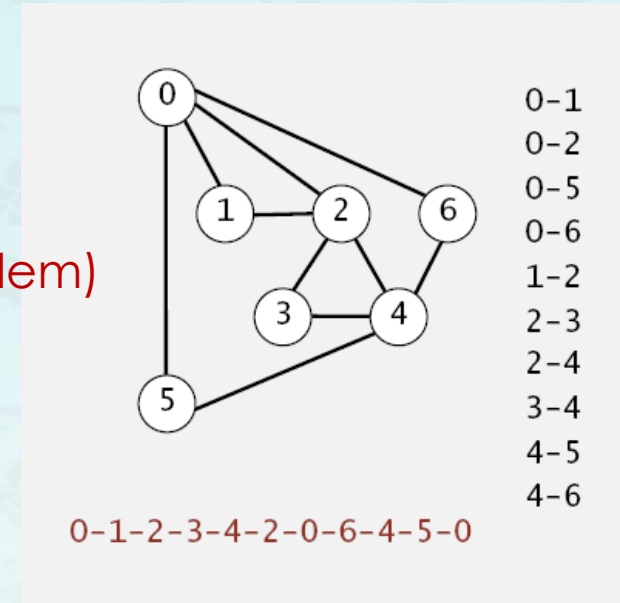
Graph-processing challenge 4: Hamilton tour

- **Problem:** Find a (general) cycle that uses every **edge exactly once**.
- **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.

Hamilton cycle
(classic NP-complete
problem)
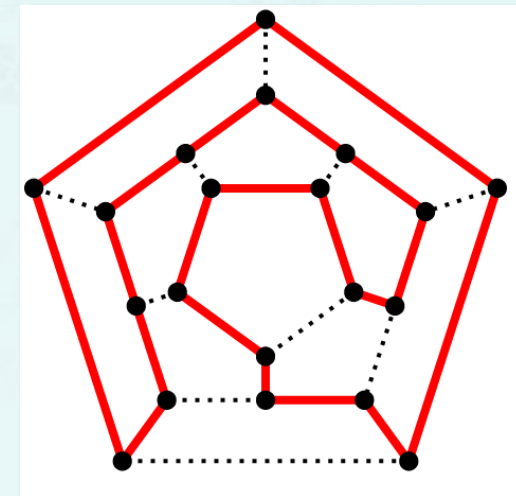
Graph-processing challenge 5: isomorphism

- **Problem:** Are **two graphs identical** except for vertex names?
- **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.

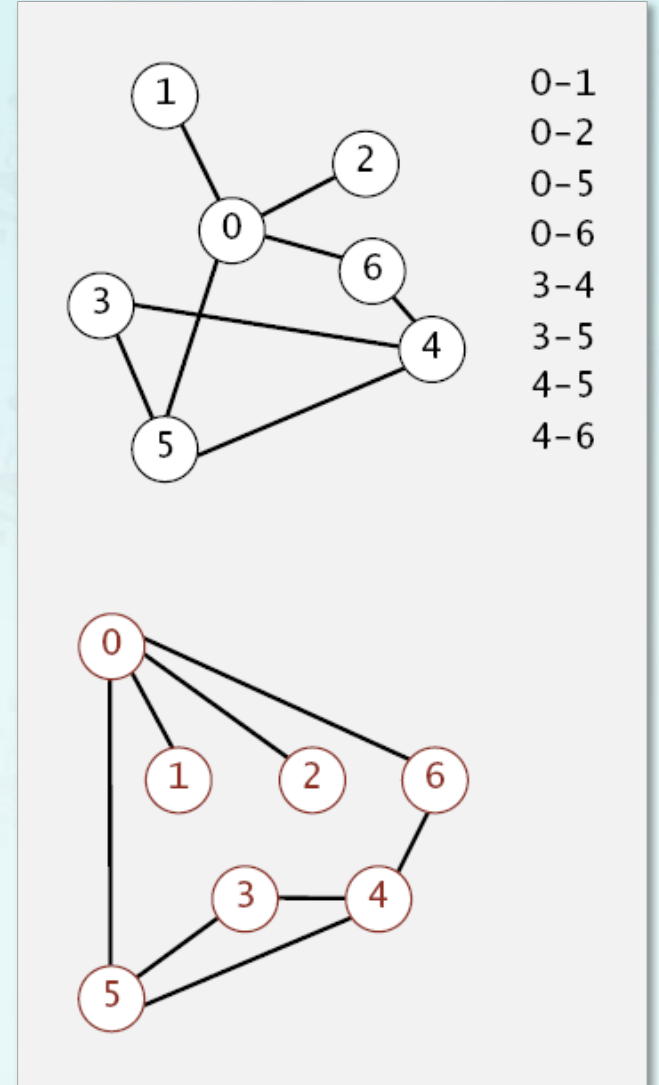graph **isomorphism** is longstanding open problem

| 0-1 |
| 0-2 |
| 0-5 |
| 0-6 |
| 3-4 |
| 3-5 |
| 4-5 |
| 4-6 |

| 0-4 |
| 0-5 |
| 0-6 |
| 1-4 |
| 1-5 |
| 2-4 |
| 3-4 |
| 5-6 |
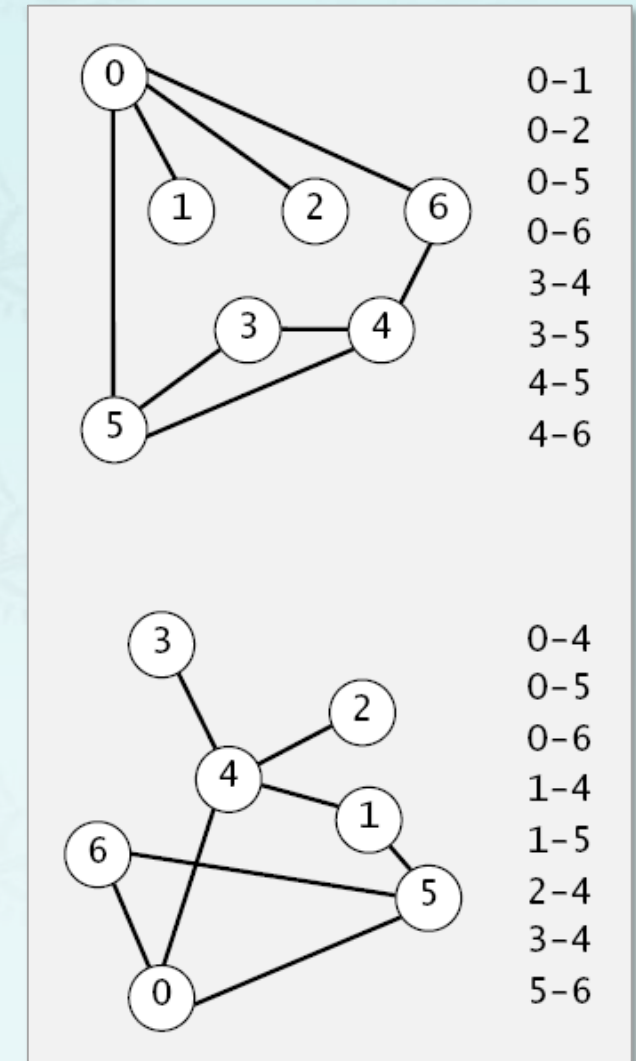
0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

Graph-processing challenge 6: planarity

- **Problem:** Lay out a graph in the plane **without crossing edges**?
- **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.

linear-time DFS-based planarity algorithm
discovered by Tarjan in 1970s
(too complicated for most practitioners)


```
0-1
0-2
0-5
0-6
3-4
3-5
4-5
4-6
```

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

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)