Tutorial 12

ZHANG Xinyun

- Graph
- Disjoint set

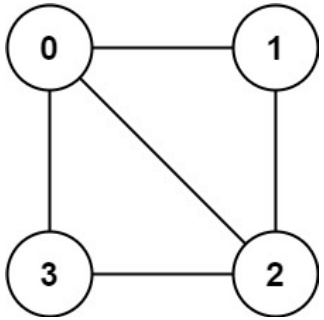
Exercise 1

Question: There is an **undirected** graph with *n* nodes. You are given a 2D array *graph*, where *graph[u]* is an array of nodes that node *u* is adjacent to. More formally, for each *v* in *graph[u]*, there is an undirected edge between node *u* and node *v*. The graph has the following properties:

- There are no self-edges (*graph[u]* does not contain *u*).
- There are no parallel edges (*graph[u]* without duplicate values).
- If *v* is in *graph[u]*, then *u* is in *graph[v]* (the graph is undirected).
- The graph may not be connected, meaning there may be two nodes *u* and *v* such that there is no path between them.

A graph is **bipartite** if the nodes can be partitioned into two independent sets A and B, and every edge contains a node in A and a node in B. Return true *if and only if it is **bipartite***.

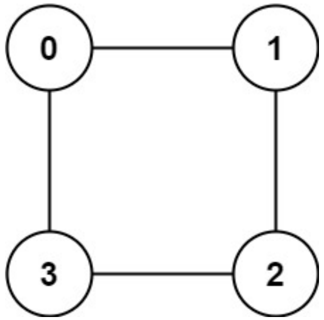
Exercise 1



Example 1:

Input: graph = [[1,2,3],[0,2],[0,1,3],[0,2]]

Output: false



Example 2:

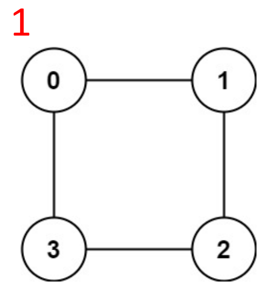
Input: graph = [[1,3],[0,2],[1,3],[0,2]]
Output: true (we have [0,2], [1,3])

Exercise 1

```
typedef struct DigraphCDT *DigraphADT;

// We use Version 1.0 here. For
simplicity, you may assume the
graph contains less than 10 nodes.
struct DigraphCDT {int A[10][10];};


bool isBipartite(DigraphADT G, int graphSize)
{
    // Please type your code here

}
```

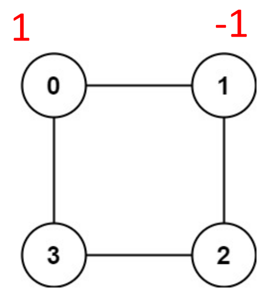
Exercise 1

We can consider this problem as a coloring problem.
For nodes in set A we can use one color and for nodes in set B we can use another one.
If we find no contradictory during the coloring process then this graph is bipartite.



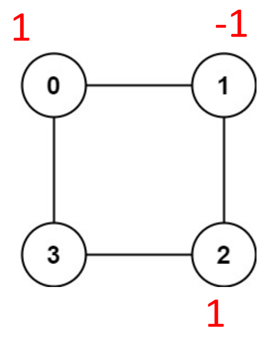
Exercise 1

We can consider this problem as a coloring problem.
For nodes in set A we can use one color and for nodes in set B we can use another one.
If we find no contradictory during the coloring process then this graph is bipartite.



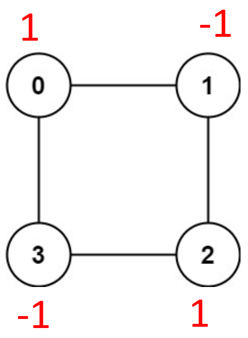
Exercise 1

We can consider this problem as a coloring problem.
For nodes in set A we can use one color and for nodes in set B we can use another one.
If we find no contradictory during the coloring process then this graph is bipartite.



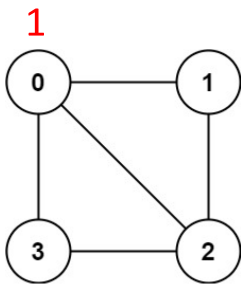
Exercise 1

We can consider this problem as a coloring problem.
For nodes in set A we can use one color and for nodes in set B we can use another one.
If we find no contradictory during the coloring process then this graph is bipartite.



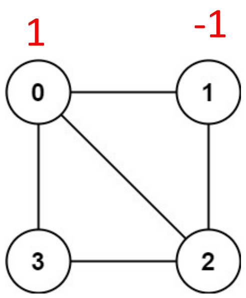
Exercise 1

We can consider this problem as a coloring problem.
For nodes in set A we can use one color and for nodes in set B we can use another one.
If we find no contradictory during the coloring process then this graph is bipartite.



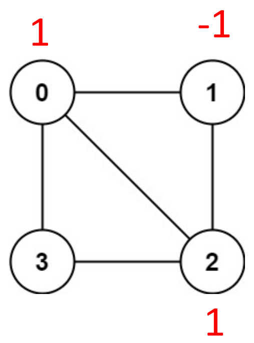
Exercise 1

We can consider this problem as a coloring problem.
For nodes in set A we can use one color and for nodes in set B we can use another one.
If we find no contradictory during the coloring process then this graph is bipartite.



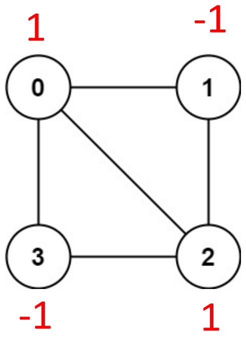
Exercise 1

We can consider this problem as a coloring problem.
For nodes in set A we can use one color and for nodes in set B we can use another one.
If we find no contradictory during the coloring process then this graph is bipartite.



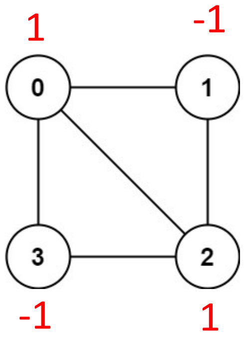
Exercise 1

We can consider this problem as a coloring problem.
For nodes in set A we can use one color and for nodes in set B we can use another one.
If we find no contradictory during the coloring process then this graph is bipartite.



Exercise 1

We can consider this problem as a coloring problem.
For nodes in set A we can use one color and for nodes in set B we can use another one.
If we find no contradictory during the coloring process then this graph is bipartite.



-1 Contradictory found!

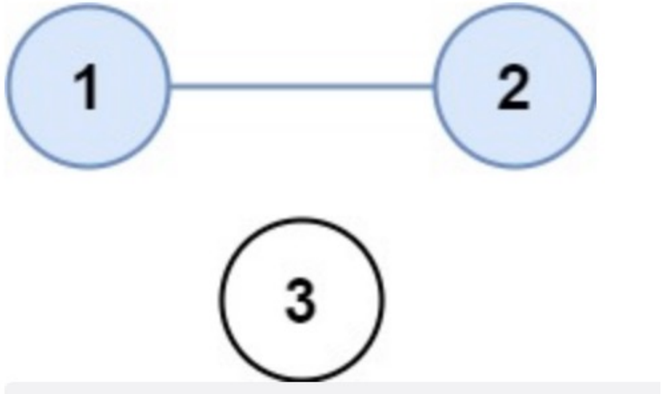
Exercise 1

```
bool isBipartite(DigraphADT G, int garphSize) {
    int* color = (int*) malloc(garphSize*sizeof(int));
    for(int k=0;k<garphsize;k++){
        if(color[k]!=0) continue;
        color[k]=-1;
        for(int i=0; i<graphSize; i++){
            if(G->A[k][i] && !helper(G, garphsize, i, color, k))
                return false;
        }
    }
    return true;
}
```

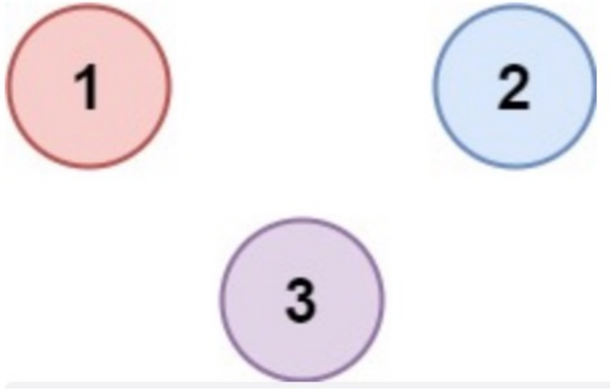
```
bool helper(DigraphADT G, int garphsize, int index, int* color, int front){
    if(color[front]==color[index])
        return false;
    else if(color[index]==0)
        color[index]=-color[front];
    else return true;
    for(int i=0; i<graphSize; i++){
        if(G->A[index][i] && !helper(G, garphSize, i, color, index))
            return false;
    }
    return true;
}
```

Exercise 2

There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city c.
A province is a group of directly or indirectly connected cities and no other cities outside of the group.
You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the i-th city and the j-th city are directly connected, and isConnected[i][j] = 0 otherwise.
Return the total number of provinces.



Output:2

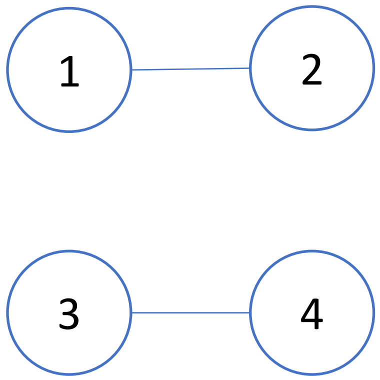


Output: 3

Exercise 2

Use DisjointSet to solve this problem.
First, we traverse the nodes and union them according to the connections.
Then, we count the number of sets.

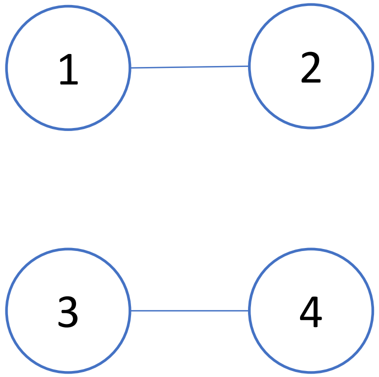
Exercise 2



Disjoint set

0	0	0	0

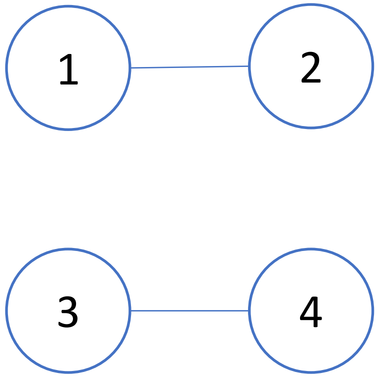
Exercise 2



Disjoint set

0	1	0	0

Exercise 2



Disjoint set

0	1	0	3

Exercise 2

```
int findProvinceNum(int** isConnected, int isConnectedSize){
    DisjSetADT set = NewDisjointSet();
    int numProvince = 0;
    for(int i=0; i < isConnectedSize; i++){
        for(int j=0; j < isConnectedSize; j++){
            if(isConnected[i][j])
                SetUnion(set, i, j);
        }
    }
    int count[isConnectedSize];
    for(int i=0; i<isConnectedSize; i++)
        count[i] = 0;
    for(int i=0; i< isConnectedSize; i++){
        int root = Find(i, set);
        if(count[root] == 0){
            count[root] = 1;
            numProvince += 1;
        }
    }
    return numProvince;
}
```