# Graph

📌 **Problem Statement:**

Given an undirected graph, determine if the graph contains **a cycle**.

📥 **Example:**

```
Input:
Edges = [[0,1], [1,2], [2,0], [1,3]]
Number of vertices = 4

Output:
true

Explanation:
There is a cycle 0-1-2-0
```

## 🔁 Brute Force:

- Check every possible path for a cycle.
- Inefficient for large graphs.

## 🚀 Optimal Solutions:

Two common methods:

1. **DFS (Depth-First Search)**
   - Use a recursive DFS.
   - Track the parent of each node to avoid trivial cycle detection.
2. **Union-Find (Disjoint Set Union - DSU)**
   - Efficient for detecting cycles when adding edges.
   - If two vertices belong to the same set, adding an edge creates a cycle.

## ✅ Java Code Using DFS:

```
import java.util.*;

public class CycleDetectionDFS {
    private static boolean dfs(int node, int parent, List<List<Integer>> adj,
boolean[] visited) {
        visited[node] = true;
```

```
        for (int neighbor : adj.get(node)) {
            if (!visited[neighbor]) {
                if (dfs(neighbor, node, adj, visited)) return true;
            } else if (neighbor != parent) {
                // Found a cycle
                return true;
            }
        }
        return false;
    }

    public static boolean hasCycle(int n, int[][] edges) {
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < n; i++) adj.add(new ArrayList<>());

        for (int[] edge : edges) {
            adj.get(edge[0]).add(edge[1]);
            adj.get(edge[1]).add(edge[0]);
        }

        boolean[] visited = new boolean[n];
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {
                if (dfs(i, -1, adj, visited)) return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        int n = 4;
        int[][] edges = {{0,1}, {1,2}, {2,0}, {1,3}};
        System.out.println(hasCycle(n, edges));  // Output: true
    }
}
```

### ✅ Java Code Using Union-Find:

```
public class CycleDetectionUnionFind {
    static int find(int[] parent, int i) {
        if (parent[i] == -1) return i;
        return parent[i] = find(parent, parent[i]);
    }
```

```java
    static boolean union(int[] parent, int x, int y) {
        int s1 = find(parent, x);
        int s2 = find(parent, y);

        if (s1 == s2) return true;  // cycle detected

        parent[s1] = s2;
        return false;
    }

    public static boolean hasCycle(int n, int[][] edges) {
        int[] parent = new int[n];
        Arrays.fill(parent, -1);

        for (int[] edge : edges) {
            if (union(parent, edge[0], edge[1])) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        int n = 4;
        int[][] edges = {{0,1}, {1,2}, {2,0}, {1,3}};
        System.out.println(hasCycle(n, edges));  // Output: true
    }
}
```

✅ Output:

```
true
```

📌 **Problem Statement:**

A country has `n` cities and `m` bidirectional roads. The government wants every city to have access to a library.

- They can build a library in a city.
- Or repair roads to connect cities to a city with a library.

Find the **minimum cost** to ensure every city can access a library.

---

📥 **Example:**

```
Input:
n = 3, m = 3
c_lib = 2, c_road = 1
roads = [[1,2], [3,1], [2,3]]

Output: 4

Explanation:
Build 1 library + repair 2 roads = 2 + 2*1 = 4
```

### 🔁 Brute Force:

- Try building a library in every city (costly).
- Repair every road (too expensive).

---

### 🚀 Optimal Approach:

- If `c_lib <= c_road`, build a library in every city (cost = n * c_lib).
- Else:
  - Find connected components in graph.
  - For each component:
    - Build 1 library
    - Repair roads to connect others in component (component_size - 1 roads)
  - Cost = sum of these.

---

### ✅ Java Code:

```
import java.util.*;

public class RoadsAndLibraries {
    static void dfs(int city, List<List<Integer>> adj, boolean[] visited) {
        visited[city] = true;
        for (int neighbor : adj.get(city)) {
            if (!visited[neighbor]) {
                dfs(neighbor, adj, visited);
            }
        }
    }

    public static long roadsAndLibraries(int n, int c_lib, int c_road, int[][]
roads) {
        if (c_lib <= c_road) {
            // Cheaper to build library in each city
```

```
            return (long) n * c_lib;
        }

        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i <= n; i++) adj.add(new ArrayList<>());

        for (int[] road : roads) {
            adj.get(road[0]).add(road[1]);
            adj.get(road[1]).add(road[0]);
        }

        boolean[] visited = new boolean[n + 1];
        long cost = 0;

        for (int city = 1; city <= n; city++) {
            if (!visited[city]) {
                dfs(city, adj, visited);
                // For each component, cost = 1 lib + roads to connect rest
                cost += c_lib + (long) (countComponentCities(adj, city, visited) -
1) * c_road;
            }
        }
        return cost;
    }

    static int countComponentCities(List<List<Integer>> adj, int start, boolean[]
visited) {
        // BFS to count cities in this component
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(start);
        int count = 0;

        boolean[] tempVisited = new boolean[visited.length];
        System.arraycopy(visited, 0, tempVisited, 0, visited.length);

        while (!queue.isEmpty()) {
            int curr = queue.poll();
            count++;
            for (int neighbor : adj.get(curr)) {
                if (!tempVisited[neighbor]) {
                    tempVisited[neighbor] = true;
                    queue.offer(neighbor);
                }
```

```
            }
        }
        return count;
    }

    public static void main(String[] args) {
        int n = 3, c_lib = 2, c_road = 1;
        int[][] roads = {{1,2}, {3,1}, {2,3}};
        System.out.println(roadsAndLibraries(n, c_lib, c_road, roads));  // Output:
4
    }
}
```

*Note:* The counting function here is redundant since DFS is already marking visited. To avoid confusion, let's replace with a better way.

---

### Revised Code for Counting Component Size:

Since DFS marks visited nodes, we can count component size during DFS.

```
import java.util.*;

public class RoadsAndLibraries {
    static int dfs(int city, List<List<Integer>> adj, boolean[] visited) {
        visited[city] = true;
        int count = 1;
        for (int neighbor : adj.get(city)) {
            if (!visited[neighbor]) {
                count += dfs(neighbor, adj, visited);
            }
        }
        return count;
    }

    public static long roadsAndLibraries(int n, int c_lib, int c_road, int[][]
roads) {
        if (c_lib <= c_road) {
            return (long) n * c_lib;
        }

        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i <= n; i++) adj.add(new ArrayList<>());

        for (int[] road : roads) {
```

```
            adj.get(road[0]).add(road[1]);
            adj.get(road[1]).add(road[0]);
        }

        boolean[] visited = new boolean[n + 1];
        long cost = 0;

        for (int city = 1; city <= n; city++) {
            if (!visited[city]) {
                int componentSize = dfs(city, adj, visited);
                cost += c_lib + (long) (componentSize - 1) * c_road;
            }
        }
        return cost;
    }

    public static void main(String[] args) {
        int n = 3, c_lib = 2, c_road = 1;
        int[][] roads = {{1,2}, {3,1}, {2,3}};
        System.out.println(roadsAndLibraries(n, c_lib, c_road, roads));  // Output:
4
    }
}
```

✅ Output:

```
4
```

## ✅ Graph Theory + Greedy – Goodland Electricity

---

### 📌 Problem Statement:

There are `n` cities arranged in a line, each city at a certain position. You need to supply electricity to all cities.

- You can place power plants in some cities.
- Each power plant supplies electricity to all cities within a range `k` on both sides (including itself).
- Find the **minimum number of power plants** needed to cover all cities.

---

### 📥 Example:

```
Input:
cities = [0, 1, 1, 0, 1]
k = 2
```

```
Output:
1

Explanation:
One power plant placed at city 2 (0-based indexing) covers cities 0 to 4.
```

### 🔁 Brute Force:

- Try placing a power plant in every possible city and check coverage.
- Very inefficient (exponential time).

---

### 🚀 Optimal Approach (Greedy):

- Traverse cities left to right.
- For current uncovered city, find the rightmost city within `k` that has a power plant.
- Place power plant there and cover all cities in range.
- Move to the next uncovered city beyond the range.

---

### ✅ Java Code:

```java
public class GoodlandElectricity {
    public static int minPowerPlants(int[] cities, int k) {
        int n = cities.length;
        int count = 0;
        int i = 0;

        while (i < n) {
            int loc = -1;

            // Find rightmost city within [i, i+k-1] that has power plant
(cities[j] == 1)
            int rightLimit = Math.min(i + k - 1, n - 1);
            for (int j = rightLimit; j >= i; j--) {
                if (cities[j] == 1) {
                    loc = j;
                    break;
                }
            }

            if (loc == -1) {
                // Cannot place power plant to cover city i
                return -1;
            }
```

```
            count++;

            // Next city to cover is loc + k
            i = loc + k;
        }

        return count;
    }

    public static void main(String[] args) {
        int[] cities = {0, 1, 1, 0, 1};
        int k = 2;
        System.out.println(minPowerPlants(cities, k));  // Output: 1
    }
}
```

✅ Output:

```
1
```