Bus Routes in C++

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>
#include <unordered_set>

using namespace std;

int numBusesToDestination(vector<vector<int>>&
routes, int S, int T) {
   int n = routes.size();
   unordered_map<int, vector<int>> map;

   // Building a map of bus stops to their respective bus
routes
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < routes[i].size(); ++j) {
         int busStopNo = routes[i][j];
         map[busStopNo].push_back(i);
      }
   }

   queue<int> q;
   unordered_set<int> busStopVisited;
   unordered_set<int> busVisited;
   int level = 0;
   q.push(S);
   busStopVisited.insert(S);

   // Performing BFS to find the minimum number of
buses
   while (!q.empty()) {
      int size = q.size();
      while (size-- > 0) {
         int currentStop = q.front();
         q.pop();
         if (currentStop == T) {
            return level;
         }

         if (map.find(currentStop) != map.end()) {
            vector<int>& buses = map[currentStop];
            for (int bus : buses) {
               if (busVisited.count(bus) > 0) {
                  continue;
               }

               vector<int>& busRoute = routes[bus];
               for (int nextStop : busRoute) {
                  if (busStopVisited.count(nextStop) > 0) {
                     continue;
                  }

                  q.push(nextStop);
                  busStopVisited.insert(nextStop);
               }
               busVisited.insert(bus);
            }
         }
      }
      ++level;
```

Input:

- Bus routes:

```
routes = {
    {1, 2, 7},    // Bus 0
    {3, 6, 7}     // Bus 1
}
```

- Source bus stop (`S = 1`)
- Destination bus stop (`T = 6`)

Step 1: Build the Map

The program constructs a map where each bus stop points to the buses that stop there. The map is:

```
map = {
    1: {0},
    2: {0},
    7: {0, 1},
    3: {1},
    6: {1}
}
```

Here:

- `1` is served by bus `0`.
- `7` is served by buses `0` and `1`.
- `6` is served by bus `1`, etc.

Step 2: BFS Initialization

- Queue `q` is initialized with the **source stop (`S = 1`)**: `q = {1}`.
- Visited sets:
  - `busStopVisited = {1}` (to track visited bus stops).
  - `busVisited = {}` (to track visited buses).
- `level = 0` (tracks the number of buses taken).

Step 3: BFS Process

Level 0:

```
    }
    return -1; // If destination is not reachable
}

int main() {
    // Hardcoded input values
    vector<vector<int>> routes = {
        {1, 2, 7},
        {3, 6, 7}
    };
    int src = 1; // source bus stop
    int dest = 6; // destination bus stop

    cout << numBusesToDestination(routes, src, dest)
<< endl;

    return 0;
}
```

- Queue size = 1 (contains `1`).
- Process bus stop `1`:
  - Stops at `1` are served by bus `0` (from `map`).
  - Bus `0` is not visited, so:
    - Add all stops from bus `0` (`{1, 2, 7}`) to the queue:
      - Add `2` to `q`.
      - Add `7` to `q`.
      - Mark stops `2` and `7` as visited (`busStopVisited = {1, 2, 7}`).
    - Mark bus `0` as visited (`busVisited = {0}`).
- **End of level 0:**
  - Queue: `q = {2, 7}`.
  - Increment `level = 1`.

Level 1:

- Queue size = 2 (contains `2, 7`).
- Process bus stop `2`:
  - Stops at `2` are served by bus `0`, which is already visited (`busVisited = {0}`).
  - Skip further processing for stop `2`.
- Process bus stop `7`:
  - Stops at `7` are served by buses `0` and `1` (from `map`).
  - Bus `0` is already visited.
  - Bus `1` is not visited, so:
    - Add all stops from bus `1` (`{3, 6, 7}`) to the queue:
      - Add `3` to `q`.
      - Add `6` to `q`.
      - Mark stops `3` and `6` as visited (`busStopVisited = {1, 2, 3, 6, 7}`).
    - Mark bus `1` as visited (`busVisited = {0, 1}`).
- **End of level 1:**
  - Queue: `q = {3, 6}`.

    - Increment `level = 2`.

Level 2:

- Queue size = 2 (contains `3, 6`).
- Process bus stop `3`:
    - Stops at `3` are served by bus `1`, which is already visited (`busVisited = {0, 1}`).
    - Skip further processing for stop `3`.
- Process bus stop `6`:
    - `6` is the destination (`T = 6`).
    - **Return `level = 2`.**

Output:

The minimum number of buses required to travel from stop `1` to stop `6` is:

Output:-
2

Coloring Border in C++

```
#include <iostream>
#include <vector>

using namespace std;

vector<vector<int>> dirs = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

void dfs(vector<vector<int>>& grid, int row, int col, int
clr) {
    grid[row][col] = -clr;
    int count = 0;

    for (auto dir : dirs) {
        int rowdash = row + dir[0];
        int coldash = col + dir[1];

        if (rowdash < 0 || coldash < 0 || rowdash >=
grid.size() || coldash >= grid[0].size() ||
abs(grid[rowdash][coldash]) != clr) {
            continue;
        }

        count++;

        if (grid[rowdash][coldash] == clr) {
            dfs(grid, rowdash, coldash, clr);
        }
    }

    if (count == 4) {
        grid[row][col] = clr;
    }
}

void coloring_border(vector<vector<int>>& grid, int row,
int col, int color) {
    dfs(grid, row, col, grid[row][col]);

    for (int i = 0; i < grid.size(); i++) {
        for (int j = 0; j < grid[0].size(); j++) {
            if (grid[i][j] < 0) {
                grid[i][j] = color;
            }
        }
    }
}

int main() {
    // Hardcoded input
    int m = 4;
    int n = 4;
    vector<vector<int>> arr = {
        {2, 1, 3, 4},
        {1, 2, 2, 2},
        {3, 2, 2, 2},
        {1, 2, 2, 2}
    };
    int row = 1;
    int col = 1;
    int color = 3;
```

Step-by-Step Dry Run:

Step 1: Call to coloring_border

- **Initial call:** coloring_border(arr, row=1, col=1, color=3).
- Call dfs(grid, row=1, col=1, clr=2) to mark the connected component and determine the border cells.

Step 2: DFS Traversal

The function dfs will:

1. Mark cells in the connected component as -2 (negate the value).
2. Identify border cells and keep them marked as -2. Interior cells are restored to clr (2) if they are surrounded by four other cells of the same value.

Traversal Process:

1. Start DFS at (1, 1):
   - Mark grid[1][1] = -2.
   - Neighboring cells:
     - (1, 2) → Proceed (same value 2).
     - (2, 1) → Proceed (same value 2).
     - Other directions lead to invalid cells or cells with a different value.
   - **Connected cells count so far:** 2.
2. Move to (1, 2):
   - Mark grid[1][2] = -2.
   - Neighboring cells:
     - (1, 3) → Proceed (same value 2).
     - (2, 2) → Proceed (same value 2).
   - **Connected cells count so far:** 4.
3. Move to (1, 3):
   - Mark grid[1][3] = -2.
   - Neighboring cells:
     - (2, 3) → Proceed (same value 2).
4. Continue this process until all connected cells are visited and marked as -2. For the input grid, all 2 values connected to (1, 1) are part of the connected component.

```
    coloring_border(arr, row, col, color);

    // Print the modified grid
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cout << arr[i][j] << "\t";
        }
        cout << endl;
    }

    return 0;
}
```

- **Marked Grid after DFS:**

  Copy code
  ```
  2  1  3  4
  1 -2 -2 -2
  3 -2 -2 -2
  1 -2 -2 -2
  ```

Step 3: Check for Border Cells

The DFS function identifies border cells:

- A cell is on the border if it:
    - Is not surrounded on all four sides by other cells of the same value.
- Border cells remain marked as -2.

Step 4: Update Colors

- Traverse the grid:
    - Any cell marked as -2 is changed to the new color (3).
    - Interior cells (those surrounded by the same value) are reverted to clr (2).
- **Final Grid:**

  ```
  2  1  3  4
  1  3  3  3
  3  3  3  3
  1  3  3  3
  ```

Output:-
```
2	1	3	4
1	3	3	3
3	3	2	3
1	3	3	3
```

Min Cost to collect all cities in C++

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

struct Edge {
    int v;
    int wt;

     Edge(int nbr, int weight) {
        this->v = nbr;
        this->wt = weight;
    }
};

struct CompareEdge {
    bool operator()(const Edge& e1, const Edge& e2) {
        return e1.wt > e2.wt; // Min-Heap based on edge
weight
    }
};

int main() {
    // Hardcoded input
    int vtces = 7;
    int edges = 8;
    vector<vector<Edge>> graph(vtces);

    // Hardcoded edges
    vector<vector<int>> hardcoded_edges = {
        {0, 1, 10},
        {1, 2, 10},
        {2, 3, 10},
        {0, 3, 40},
        {3, 4, 2},
        {4, 5, 3},
        {5, 6, 3},
        {4, 6, 8}
    };

    // Populating the graph with hardcoded edges
    for (auto& edge : hardcoded_edges) {
        int v1 = edge[0];
        int v2 = edge[1];
        int wt = edge[2];
        graph[v1].emplace_back(v2, wt);
        graph[v2].emplace_back(v1, wt);
    }

    int ans = 0;
    priority_queue<Edge, vector<Edge>, CompareEdge>
pq;
    vector<bool> vis(vtces, false);
    pq.push(Edge(0, 0)); // Start with any vertex (0 in this
case) with 0 weight

    while (!pq.empty()) {
        Edge rem = pq.top();
        pq.pop();

        if (vis[rem.v]) {
```

Step-by-Step Execution

Step 1: Populate the Graph

The adjacency list (graph) for the given input will look like this:

0: (1, 10), (3, 40)
1: (0, 10), (2, 10)
2: (1, 10), (3, 10)
3: (2, 10), (0, 40), (4, 2)
4: (3, 2), (5, 3), (6, 8)
5: (4, 3), (6, 3)
6: (5, 3), (4, 8)

Step 2: Initialize Priority Queue and Visited Array

- Start with vertex 0 and push an edge (0, 0) to the priority queue (pq).
- Initially:

  pq: [(0, 0)] (min-heap: weight 0, vertex 0)
  vis: [false, false, false, false, false, false, false]
  ans: 0

Step 3: Prim's Algorithm

- **Iteration 1:**
  - Pop (0, 0) from pq.
  - Add weight 0 to ans. Now ans = 0.
  - Mark vertex 0 as visited (vis[0] = true).
  - Push neighbors (1, 10) and (3, 40) to pq.

  pq: [(1, 10), (3, 40)]
  vis: [true, false, false, false, false, false, false]
  ans: 0

- **Iteration 2:**
  - Pop (1, 10) from pq.
  - Add weight 10 to ans. Now ans = 10.
  - Mark vertex 1 as visited (vis[1] = true).
  - Push neighbors (2, 10) to pq (skip (0, 10) because 0 is already visited).

  pq: [(2, 10), (3, 40)]
  vis: [true, true, false, false, false, false, false]

```
            continue;
        }
        vis[rem.v] = true;
        ans += rem.wt;

        for (Edge nbr : graph[rem.v]) {
            if (!vis[nbr.v]) {
                pq.push(nbr);
            }
        }
    }

    cout << ans << endl;

    return 0;
}
```

ans: 10

- **Iteration 3:**
    - Pop (2, 10) from pq.
    - Add weight 10 to ans. Now ans = 20.
    - Mark vertex 2 as visited (vis[2] = true).
    - Push neighbor (3, 10) to pq (skip (1, 10) because 1 is already visited).

  pq: [(3, 10), (3, 40)]
  vis: [true, true, true, false, false, false, false]
  ans: 20

- **Iteration 4:**
    - Pop (3, 10) from pq.
    - Add weight 10 to ans. Now ans = 30.
    - Mark vertex 3 as visited (vis[3] = true).
    - Push neighbors (4, 2) to pq (skip (2, 10) and (0, 40) because 2 and 0 are already visited).

  pq: [(3, 40), (4, 2)]
  vis: [true, true, true, true, false, false, false]
  ans: 30

- **Iteration 5:**
    - Pop (4, 2) from pq.
    - Add weight 2 to ans. Now ans = 32.
    - Mark vertex 4 as visited (vis[4] = true).
    - Push neighbors (5, 3) and (6, 8) to pq (skip (3, 2) because 3 is already visited).

  pq: [(3, 40), (5, 3), (6, 8)]
  vis: [true, true, true, true, true, false, false]
  ans: 32

- **Iteration 6:**
    - Pop (5, 3) from pq.
    - Add weight 3 to ans. Now ans = 35.
    - Mark vertex 5 as visited (vis[5] = true).
    - Push neighbor (6, 3) to pq (skip (4, 3) because 4 is already visited).

  pq: [(3, 6), (6, 8), (3, 40)]
  vis: [true, true, true, true, true, true, false]

ans: 35

- **Iteration 7:**
  - Pop (6, 3) from pq.
  - Add weight 3 to ans. Now ans = 38.
  - Mark vertex 6 as visited (vis[6] = true).
  - Skip pushing neighbors because all are already visited.

pq: [(3, 40), (6, 8)]
vis: [true, true, true, true, true, true, true]
ans: 38

Final MST Weight:

- All vertices are visited, and the MST weight is 38.

Output:-

```
38
```

Negative Wt Cycle Detection in C++

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

struct Edge {
   int u, v, weight;
};

bool isNegativeWeightCycle(int n, vector<Edge>& edges)
{
   vector<int> dist(n, INT_MAX);
   dist[0] = 0; // Starting from vertex 0

   // Relaxation process
   for (int i = 0; i < n - 1; ++i) {
      for (const auto& edge : edges) {
         if (dist[edge.u] != INT_MAX && dist[edge.u] + edge.weight < dist[edge.v]) {
            dist[edge.v] = dist[edge.u] + edge.weight;
         }
      }
   }

   // Checking for negative weight cycles
   for (const auto& edge : edges) {
      if (dist[edge.u] != INT_MAX && dist[edge.u] + edge.weight < dist[edge.v]) {
         return true; // Negative weight cycle detected
      }
   }

   return false; // No negative weight cycle found
}

int main() {
   // Hardcoded input
   int n = 3; // Number of vertices
   int m = 3; // Number of edges
   vector<Edge> edges = {{0, 1, -1}, {1, 2, -4}, {2, 0, 3}}; // Edges with (u, v, weight)

   if (isNegativeWeightCycle(n, edges)) {
      cout << "1\n"; // Negative weight cycle detected
   } else {
      cout << "0\n"; // No negative weight cycle found
   }

   return 0;
}
```

Step-by-Step Execution

Input Edges:

Edge 1: (0 → 1, weight = -1)
Edge 2: (1 → 2, weight = -4)
Edge 3: (2 → 0, weight = 3)

Initial State:

dist = [0, INT_MAX, INT_MAX]

Relaxation Process (n-1 = 2 times):

- **Iteration 1:**
  - Relax Edge (0 → 1, weight = -1):

    dist[1] = min(INT_MAX, dist[0] + (-1)) = min(INT_MAX, 0 + (-1)) = -1
    dist = [0, -1, INT_MAX]

  - Relax Edge (1 → 2, weight = -4):

    dist[2] = min(INT_MAX, dist[1] + (-4)) = min(INT_MAX, -1 + (-4)) = -5
    dist = [0, -1, -5]

  - Relax Edge (2 → 0, weight = 3):

    dist[0] = min(0, dist[2] + 3) = min(0, -5 + 3) = -2
    dist = [-2, -1, -5]

- **Iteration 2:**
  - Relax Edge (0 → 1, weight = -1):

    dist[1] = min(-1, dist[0] + (-1)) = min(-1, -2 + (-1)) = -3
    dist = [-2, -3, -5]

  - Relax Edge (1 → 2, weight = -4):

    dist[2] = min(-5, dist[1] + (-4)) = min(-5, -3 + (-4)) = -7
    dist = [-2, -3, -7]

  - Relax Edge (2 → 0, weight = 3):

    dist[0] = min(-2, dist[2] + 3) = min(-2, -7 + 3) = -4
    dist = [-4, -3, -7]

Check for Negative Weight Cycles:

- Try relaxing all edges once more:

	○ Relax Edge (0 → 1, weight = -1): dist[1] = min(-3, dist[0] + (-1)) = min(-3, -4 + (-1)) = -5 ○ At this point, the distance to vertex 1 changes, which means a negative weight cycle exists. **Output:** • Since a negative weight cycle is detected: 1
Output:- 1	

No of Distinct Island in C++

```
#include <iostream>
#include <vector>
#include <unordered_set>

using namespace std;

// Function prototypes
void dfs(vector<vector<int>>& arr, int row, int col,
string& psf);
int numDistinctIslands(vector<vector<int>>& arr);

// Depth-first search to mark all connected land cells of
an island
void dfs(vector<vector<int>>& arr, int row, int col,
string& psf) {
    arr[row][col] = 0; // Marking current cell as visited
    int n = arr.size();
    int m = arr[0].size();

    // Directions: up, right, down, left
    vector<pair<int, int>> dirs = {{-1, 0}, {0, 1}, {1, 0}, {0,
-1}};
    string dirStr = "urdl"; // Corresponding directions
characters

    for (int i = 0; i < 4; ++i) {
        int newRow = row + dirs[i].first;
        int newCol = col + dirs[i].second;
        if (newRow >= 0 && newRow < n && newCol >= 0
&& newCol < m && arr[newRow][newCol] == 1) {
            psf += dirStr[i]; // Append direction character to
path string
            dfs(arr, newRow, newCol, psf);
        }
    }
    psf += "a"; // Append anchor to indicate end of island
path
}

// Function to find number of distinct islands
int numDistinctIslands(vector<vector<int>>& arr) {
    int n = arr.size();
    if (n == 0) return 0;
    int m = arr[0].size();

    unordered_set<string> islands; // Set to store distinct
island paths

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (arr[i][j] == 1) {
                string psf = "x"; // Starting character to
represent new island
                dfs(arr, i, j, psf);
                islands.insert(psf); // Insert island path into set
            }
        }
    }

    return islands.size(); // Return the number of distinct
islands
```

Input:

Grid:
1 0 0
0 1 0
1 1 1

Execution Steps

Step 1: Initializing Variables

- The grid has 3 rows x 3 columns.
- An empty set islands is initialized to store distinct island shapes.

Step 2: Traversing the Grid

1. At (0, 0):
    - Start a DFS and encode the path:

      sql
      Copy code
      psf = "x" (start)
      Move down: psf = "xurda" (anchor added after exploring up, right, down, left)

    - Add "xurda" to islands.
2. At (1, 1):
    - Start a DFS and encode the path:

      arduino
      Copy code
      psf = "x" (start)
      No other cells connected to (1, 1): psf = "xurda" (isolated cell)

    - Add "xurda" to islands (already present).
3. At (2, 0):
    - Start a DFS and encode the path:

      sql
      Copy code
      psf = "x" (start)
      Move right: psf = "xrd" (connects (2, 0) → (2, 1))
      Move right again: psf = "xrdrr" (connects (2, 1) → (2, 2))
      Move up: psf = "xrdrru" (connects (2, 2) → (1, 2))
      Add anchors: psf = "xrdrruarrarrarr"

    - Add "xrdrruarrarrarr" to

```
}

int main() {
    // Hardcoded input
    vector<vector<int>> arr = {
        {1, 0, 0},
        {0, 1, 0},
        {1, 1, 1}
    };

    // Calculating number of distinct islands
    cout << numDistinctIslands(arr) << endl;

    return 0;
}
```

islands.

Step 3: Count Distinct Islands

- The set islands contains:

  {"xurda", "xrdrruarrarrarr"}

- The size of the set is 2.

Output:

2

Output:-
2

No of enclaves in C++

```
#include <iostream>
#include <vector>

using namespace std;

void dfs(vector<vector<int>>& arr, int i, int j) {
   if (i < 0 || j < 0 || i >= arr.size() || j >= arr[0].size()
|| arr[i][j] == 0) {
       return;
   }
   arr[i][j] = 0;
   dfs(arr, i + 1, j);
   dfs(arr, i - 1, j);
   dfs(arr, i, j + 1);
   dfs(arr, i, j - 1);
}

int numEnclaves(vector<vector<int>>& arr) {
   int m = arr.size();
   int n = arr[0].size();

   // Marking connected components touching the
boundaries
   for (int i = 0; i < m; ++i) {
       for (int j = 0; j < n; ++j) {
           if ((i == 0 || j == 0 || i == m - 1 || j == n - 1) &&
arr[i][j] == 1) {
               dfs(arr, i, j);
           }
       }
   }

   // Counting remaining land cells
   int count = 0;
   for (int i = 0; i < m; ++i) {
       for (int j = 0; j < n; ++j) {
           if (arr[i][j] == 1) {
               ++count;
           }
       }
   }

   return count;
}

int main() {
   int m = 4, n = 4;
   vector<vector<int>> arr = {
       {0, 0, 0, 0},
       {1, 0, 1, 0},
       {0, 1, 1, 0},
       {0, 0, 0, 0}
   };

   int result = numEnclaves(arr);
   cout << result << endl;

   return 0;
}
```

Dry Run

Input Grid:

```
{
   {0, 0, 0, 0},
   {1, 0, 1, 0},
   {0, 1, 1, 0},
   {0, 0, 0, 0}
}
```

Step 1: DFS from Boundary Cells

- **Boundary cells:** We start by scanning the boundary cells (first and last rows, first and last columns). The boundary cells are:
  - Row 0: {0, 0, 0, 0}
  - Row 3: {0, 0, 0, 0}
  - Column 0: {1, 0, 0, 0}
  - Column 3: {0, 0, 0, 0}
- The boundary cells that are 1 (land) are:
  - (1, 0)

Step 2: Marking Land Cells Connected to Boundary

1. **DFS starting at (1, 0):**
   - Mark arr[1][0] as 0.
   - Explore its neighbors (down: (2, 0), left: out of bounds, right: (1, 1), up: (0, 0)).
   - No other connected land cells.

Step 3: Count Remaining Land Cells

After marking the connected land cells to the boundary, the grid looks like this:

```
{
   {0, 0, 0, 0},
   {0, 0, 1, 0},
   {0, 1, 1, 0},
   {0, 0, 0, 0}
}
```

Now, we count the remaining land cells (1) in the grid:

- (1, 2), (2, 1), and (2, 2) are the remaining land cells.

Final Answer:

The number of enclosed land cells is 3.

Output:

	3
Output:- 3	

Optimize water distribution in C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <utility>

using namespace std;

class Pair {
public:
    int vtx;
    int wt;
    Pair(int vtx, int wt) {
        this->vtx = vtx;
        this->wt = wt;
    }
    bool operator>(const Pair& other) const {
        return this->wt > other.wt;
    }
};

int minCostToSupplyWater(int n, vector<int>& wells,
vector<vector<int>>& pipes) {
    vector<vector<Pair>> graph(n + 1);
    for (const auto& pipe : pipes) {
        int u = pipe[0];
        int v = pipe[1];
        int wt = pipe[2];
        graph[u].emplace_back(v, wt);
        graph[v].emplace_back(u, wt);
    }
    for (int i = 1; i <= n; ++i) {
        graph[i].emplace_back(0, wells[i - 1]);
        graph[0].emplace_back(i, wells[i - 1]);
    }

    int ans = 0;
    priority_queue<Pair, vector<Pair>, greater<Pair>> pq;
    pq.emplace(0, 0);
    vector<bool> vis(n + 1, false);

    while (!pq.empty()) {
        Pair rem = pq.top();
        pq.pop();
        if (vis[rem.vtx]) continue;
        ans += rem.wt;
        vis[rem.vtx] = true;
        for (const Pair& nbr : graph[rem.vtx]) {
            if (!vis[nbr.vtx]) {
                pq.push(nbr);
            }
        }
    }
    return ans;
}

int main() {
    int v = 3, e = 2;
    vector<int> wells = {1, 2, 2};
    vector<vector<int>> pipes = {{1, 2, 1}, {2, 3, 1}};

    cout << minCostToSupplyWater(v, wells, pipes) <<
```

```cpp
int v = 3, e = 2;
vector<int> wells = {1, 2, 2};
vector<vector<int>> pipes = {{1, 2, 1}, {2, 3, 1}};
```

- v = 3: Number of houses (vertices).
- wells = {1, 2, 2}: The cost to build a well at house 1, 2, and 3.
- pipes = {{1, 2, 1}, {2, 3, 1}}: The pipes connecting houses, with respective costs.

Step 1: Construct the Graph

We begin by creating an adjacency list that represents the graph, including both the pipes and wells.

- **Graph Construction**:
    - Create an adjacency list graph with v + 1 = 4 nodes (including the virtual node 0).
    - Add edges for the pipes between houses:
        - Pipe from 1 to 2 with cost 1.
        - Pipe from 2 to 3 with cost 1.
    - Add edges for the wells:
        - Well for house 1 (cost 1), connect node 0 to node 1.
        - Well for house 2 (cost 2), connect node 0 to node 2.
        - Well for house 3 (cost 2), connect node 0 to node 3.
- **Graph Representation**:

    Node 0 (virtual node) → {(1, 1), (2, 2), (3, 2)}
    Node 1 → {(2, 1), (0, 1)}
    Node 2 → {(1, 1), (3, 1), (0, 2)}
    Node 3 → {(2, 1), (0, 2)}

Step 2: Prim's Algorithm with Min-Heap

We will use **Prim's Algorithm** to find the Minimum Spanning Tree (MST) with a priority queue (min-heap).

- **Priority Queue Initialization**: Start with node 0 (virtual node), which has no cost yet, so we push (0, 0) into the priority queue.

```
endl;

    return 0;
}
```

Priority Queue: [(0, 0)]

- **Step 3: First Iteration (start with node 0)**
  - **Pop from the priority queue**: (0, 0) is popped, meaning we're at the virtual node 0 with a cost of 0.
  - **Visit Node 0** and explore its neighbors (nodes 1, 2, 3):
    - Add the edges to the priority queue:
      - Edge (0 → 1, cost 1)
      - Edge (0 → 2, cost 2)
      - Edge (0 → 3, cost 2)

  After this step:

  Priority Queue: [(1, 1), (2, 2), (2, 3)]
  Visited nodes: [0]
  Total Cost: 0

- **Step 4: Second Iteration (pop node 1)**
  - **Pop from the priority queue**: (1, 1) is popped, meaning we're now at node 1 with a cost of 1.
  - **Visit Node 1** and explore its neighbors:
    - Node 0 has already been visited, so ignore.
    - Add edge (1 → 2, cost 1) to the priority queue.

  After this step:

  Priority Queue: [(1, 2), (2, 3), (1, 2)]
  Visited nodes: [0, 1]
  Total Cost: 1

- **Step 5: Third Iteration (pop node 2)**
  - **Pop from the priority queue**: (1, 2) is popped, meaning we're now at node 2 with a cost of 1.
  - **Visit Node 2** and explore its neighbors:
    - Node 1 has already been visited, so ignore.
    - Node 3 is unvisited, so add edge (2 → 3, cost 1) to the priority queue.
    - Node 0 has already been visited, so ignore.

  After this step:

  Priority Queue: [(1, 3), (2, 3), (1, 2)]

Visited nodes: [0, 1, 2]
Total Cost: 2

- **Step 6: Fourth Iteration (pop node 3)**
  - **Pop from the priority queue**: (1, 3) is popped, meaning we're now at node 3 with a cost of 1.
  - **Visit Node 3** and explore its neighbors:
    - Node 2 has already been visited, so ignore.
    - Node 0 has already been visited, so ignore.

  After this step:

  Priority Queue: [(2, 3)]
  Visited nodes: [0, 1, 2, 3]
  Total Cost: 3

- **Step 7: Termination**
  - The priority queue is empty, and all nodes are visited.
  - **Final Total Cost**: 3.

Final Output

The minimum cost to supply water is: 3

Output:-

```
3
```

Redundant Connection in C++

```cpp
#include <iostream>
#include <vector>

using namespace std;

class UnionFind {
public:
    vector<int> parent;
    vector<int> rank;

    UnionFind(int n) {
        parent.resize(n + 1);
        rank.resize(n + 1, 1);
        for (int i = 1; i <= n; ++i) {
            parent[i] = i;
        }
    }

    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // Path compression
        }
        return parent[x];
    }

    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};

vector<int>
findRedundantConnection(vector<vector<int>>& edges) {
    int n = edges.size();
    UnionFind uf(n);

    for (auto& edge : edges) {
        int u = edge[0];
        int v = edge[1];

        if (uf.find(u) == uf.find(v)) {
            return edge;  // This edge is a redundant connection
        }
        uf.unionSets(u, v);
    }
    return {};
}

int main() {
```

Dry Run

Input:

```
edges = {
    {1, 2},
    {1, 3},
    {2, 3}
}
```

Step-by-Step Execution:

- **Initialization**:
    - UnionFind:
        - parent = [1, 2, 3] (each node is its own parent initially)
        - rank = [1, 1, 1] (all ranks start as 1)

Processing Edges:

1. **Edge (1, 2)**:
    - find(1) = 1, find(2) = 2 → Different components.
    - Union the components:
        - Set parent[2] = 1.
        - Update rank[1] = 2.
    - Updated state:
        - parent = [1, 1, 3]
        - rank = [1, 2, 1].
2. **Edge (1, 3)**:
    - find(1) = 1, find(3) = 3 → Different components.
    - Union the components:
        - Set parent[3] = 1.
    - Updated state:
        - parent = [1, 1, 1]
        - rank = [1, 2, 1].
3. **Edge (2, 3)**:
    - find(2) = 1, find(3) = 1 → Same component (cycle detected).
    - Return the edge (2, 3) as the redundant connection.

```
    // Hardcoded input
    vector<vector<int>> edges = {
        {1, 2},
        {1, 3},
        {2, 3}
    };

    vector<int> result = findRedundantConnection(edges);
    cout << result[0] << " " << result[1] << endl;

    return 0;
}
```

Output:-

```
2 3
```