Bus Routes in C++

```cpp
#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;
                        }
```

Input:

```
routes = {
    {1, 2, 7},
    {3, 6, 7}
};
src = 1;
dest = 6;
```

🧠 High-Level Idea:

The code builds a graph where each **bus stop** connects to **bus routes**, then performs **BFS** starting from the source stop to find the **minimum number of buses** needed to reach the destination.

🔁 Dry Run Table (Iterative BFS)

Iteration	Level	Queue Contents	Current Stop	Bus Routes from Stop	New Stops Added to Queue	Bus Visited	Comments
Init	0	[1]	-	-	-	-	Start from stop 1
1	0	[1]	1	[0]	[2, 7]	{0}	Stop 1 is in route 0; enqueue 2, 7
2	1	[2, 7]	2	[0]	-	{0}	Bus 0 already visited
3	1	[7]	7	[0, 1]	[3, 6]	{0, 1}	Route 1 has 6 (destination!)
4	2	[3, 6]	3	[1]	-	{0, 1}	Already visited bus 1
5	2	[6]	6	[1]	-	{0, 1}	🎯 Destination reached

```
                    q.push(nextStop);

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

✅ Result:

The `level` when we reach stop `6` is **2**, but since levels are **incremented after each BFS layer**, and the first bus was taken at level 0:

☞ **Minimum buses required = 2**

🔚 Final Output:

```
2
```

Output:-
2

Coloring Border in C++

```cpp
#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;
```

Input:

```
grid = {
    {2, 1, 3, 4},
    {1, 2, 2, 2},
    {3, 2, 2, 2},
    {1, 2, 2, 2}
}
start = (1, 1)
color = 3
```

🧭 Initial Color at (1, 1): `2`

🔁 DFS Dry Run (Marking Border)

Step	Cell	Action	Count of Same Color Neighbors	Final Cell State
1	(1,1)	Mark `-2`, recurse	0 → Recursing neighbors	-2
2	(1,2)	Mark `-2`, recurse	0 → Recursing	-2
3	(1,3)	Mark `-2`, recurse	0 → Recursing	-2
4	(2,3)	Mark `-2`, recurse	0	-2
5	(2,2)	Mark `-2`, recurse	1	-2
6	(2,1)	Mark `-2`, recurse	2	-2
7	(3,1)	Mark `-2`, recurse	0	-2
8	(3,2)	Mark `-2`, recurse	1	-2
9	(3,3)	Mark `-2`, recurse	1	-2

Once recursion returns, it checks `count == 4`. If true, the cell is fully surrounded by the same component → restore it to `2`. Otherwise, it's on border → leave as `-2`.

Only cell (2,2) has all 4 neighbors of same component → reset to `2`.

🖌 Coloring Step:

- Any cell still marked as `-2` → set to

```
    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;
}
```

new `color = 3`

✅ Final Output Grid:

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

📄 Dry Run Summary Table (Key Points):

Cell	Was Visited	Final Value
(1,1)	✅	3
(1,2)	✅	3
(1,3)	✅	3
(2,1)	✅	3
(2,2)	✅	2
(2,3)	✅	3
(3,1)	✅	3
(3,2)	✅	3
(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++

```cpp
#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]) {
```

Core Concepts in the Code:

- Uses a **priority queue (min-heap)** to always pick the edge with the **least weight**.
- Starts from vertex `0`.
- Adds edge weights to the total MST weight only when visiting **unvisited vertices**.
- `vis[]` tracks visited vertices.

📊 Hardcoded Graph (7 vertices, 8 edges):

```
Edges:
{v1,v2,wt}
{0, 1, 10}
{1, 2, 10}
{2, 3, 10}
{0, 3, 40}
{3, 4, 2}
{4, 5, 3}
{5, 6, 3}
{4, 6, 8}
```

📋 Dry Run Table: Prim's MST

Step	Vertex Visited	Edge Added (from)	Weight Added	Total MST Weight	Priority Queue (next min weight edges)
1	0	- (start)	0	0	(1,10), (3,40)
2	1	0 → 1	10	10	(2,10), (3,40)
3	2	1 → 2	10	20	(3,10), (3,40)
4	3	2 → 3	10	30	(4,2), (3,40)
5	4	3 → 4	2	32	(5,3), (6,8), (3,40)
6	5	4 → 5	3	35	(6,3), (6,8), (3,40)

```
            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;
}
```

7	6	5 → 6	3	38	(6,8), (3,40) → both discarded (visited)

✅ MST Total Weight: 38

Even though there's a 40-weight edge from 0 to 3, we never pick it because we reach 3 through a cheaper path (0→1→2→3).

🖨 Output:

```
38
```

Output:-
38

Negative Wt Cycle Detection in C++

```cpp
#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;
}
```

Bellman-Ford Key Idea:

- Perform `n - 1` iterations relaxing all edges.
- Then **one more iteration** to see if **any distance still improves** → indicates a **negative cycle**.

📃 Input:

```
n = 3
edges = {
    {0, 1, -1},
    {1, 2, -4},
    {2, 0, 3}
}
```

🧮 Dry Run Table (Relaxation)

Initial `dist`:

```
[0, ∞, ∞]
```

🔁 Iteration 1:

Edge	Condition	Action	Updated dist
0 → 1 -1	$0 + (-1) < \infty$	dist[1] = -1	[0, -1, ∞]
1 → 2 -4	$-1 + (-4) < \infty$	dist[2] = -5	[0, -1, -5]
2 → 0 +3	$-5 + 3 = -2 < 0$	dist[0] = -2	[-2, -1, -5]

🔁 Iteration 2:

Edge	Condition	Action	Updated dist
0 → 1 -1	$-2 -1 = -3 < -1$	dist[1] = -3	[-2, -3, -5]
1 → 2 -4	$-3 -4 = -7 < -5$	dist[2] = -7	[-2, -3, -7]
2 → 0 +3	$-7 + 3 = -4 < -2$	dist[0] = -4	[-4, -3, -7]

🔁 Extra Iteration – Check for

Negative Cycle

Edge	Condition	Result
0 → 1 -1	-4 + (-1) = -5 < -3	✅ Negative cycle!

✅ Conclusion:

- A **negative weight cycle** exists.
- Specifically: `0 → 1 → 2 → 0` forms a cycle with total weight: `-1 + (-4) + 3 = -2`

🖨 Output:

```
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
```

Key Concepts:

- An **island** is a group of `1`s connected **horizontally or vertically**.
- Each island is converted into a **path string** (`psf`) using DFS with directional encoding (`u`, `r`, `d`, `l`, and `a` for backtracking).
- The `unordered_set` stores these path strings to count **unique island shapes**.

📥 Input Grid:

```
1 0 0
0 1 0
1 1 1
```

Key for DFS path string (`psf`):

- `x` → Start of island
- `u` → Up
- `r` → Right
- `d` → Down
- `l` → Left
- `a` → Backtrack (anchor)

📊 Dry Run Table:

Island #	Starting Cell	DFS Path (`psf`)	Shape Description	Is Unique?
1	(0, 0)	`xa`	Single cell	✅ Yes
2	(1, 1)	`xa`	Single cell	❌ No
3	(2, 0)	`xrraa`	Horizontal chain (L-shape)	✅ Yes

🧮 Final Set of Unique Island Shapes:

Shape Path

```
xa
xrraa
```

<pre>islands } 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; }</pre>	✅ **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;
}
```

📃 Input Grid:

	0	1	2	3
0	0	0	0	0
1	1	0	1	0
2	0	1	1	0
3	0	0	0	0

📊 Dry Run Table – Step-by-Step

🟡 Step 1: Mark boundary-connected 1s using DFS

Check all boundary cells and run DFS from any land (1) on the edge:

Cell	Is Boundary?	Is Land?	DFS Run?	Action
(0,x)/(x,0)/(3,x)/(x,3)	✅ Yes	Mixed	✅ If land	DFS removes (1,0) only

✅ Only **(1,0)** is a boundary land → DFS marks it and its connected land 0.

🔄 After DFS update, grid becomes:

	0	1	2	3
0	0	0	0	0
1	0	0	1	0
2	0	1	1	0
3	0	0	0	0

🟢 Step 2: Count remaining 1s (enclaves)

Cell	Value	Is Land (1)?	Count += 1?
(1,2)	1	✅	✅ (count=1)
(2,1)	1	✅	✅ (count=2)

Cell	Value	Is Land (1)?	Count += 1?
(2,2)	1	✅	✅ (count=3)

🧮 Total enclave land cells = **3**

✅ Final Output:

```
3
```

🔁 Summary Table:

Phase	Operation	Result
Boundary DFS	Remove all 1s connected to boundary	(1,0) set to 0
Enclave Counting	Count remaining 1s in the grid	3
Final Return Value	`numEnclaves()`	`3`

Output:-
3

Optimize water distribution in C++

```
#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}};
```

📜 Input:

- **Number of houses (`n`)** = `3`
- **Wells**: `[1, 2, 2]` → Cost to build wells at house 1, 2, 3
- **Pipes**:

```
[1, 2, 1]
[2, 3, 1]
```

🔧 Graph Construction (Adjacency List):

Node	Connections
0	(1,1), (2,2), (3,2)
1	(2,1), (0,1)
2	(1,1), (3,1), (0,2)
3	(2,1), (0,2)

🧮 Dry Run of Prim's Algorithm:

Step	Min Edge Picked (u→v, wt)	Added to MST	MST Cost	Visited Nodes	Heap Contents After Push
1	(0→0, 0)	0	0	{0}	(1,1), (2,2), (3,2)
2	(0→1, 1)	1	1	{0,1}	(2,2), (3,2), (2,1)
3	(1→2, 1)	2	2	{0,1,2}	(3,2), (2,2), (3,1)
4	(2→3, 1)	3	3	{0,1,2,3}	Remaining edges ignored (already visited nodes)

✅ All nodes visited.

✅ Final Output:

```
3
```

🎁 Explanation:

- Use **well at house 1**: cost 1

cout << minCostToSupplyWater(v, wells, pipes) << endl; return 0; }	• Use **pipe 1–2**: cost 1 • Use **pipe 2–3**: cost 1 ➞ **Total = 3** 🧠 This is cheaper than building all wells (1+2+2=5)
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 {};
```

You're given edges forming a graph. Initially, it's a tree (n nodes, n−1 edges), but one extra edge was added, forming a cycle.
Goal: Find the **redundant edge** forming the cycle.

📄 Input

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

🎁 Initial Setup

- Nodes: `1, 2, 3`
- `parent[] = [0, 1, 2, 3]` (0-index unused)
- `rank[] = [0, 1, 1, 1]`

🧮 Dry Run Table (Union-Find Process)

Step	Edge	Find(u)	Find(v)	Same Root?	Action	Updated `parent[]`	Updated `rank[]`
1	1-2	1	2	✖ No	Union(1, 2)	[0, 1, 1, 3]	[0, 2, 1, 1]
2	1-3	1	3	✖ No	Union(1, 3)	[0, 1, 1, 1]	[0, 2, 1, 1]
3	2-3	1	1	✔ Yes	**! Cycle found**	—	—

✔ Output

```
2 3
```

- Edge **{2, 3}** forms the cycle.
- It is **redundant**, and hence returned.

`` } int main() { // 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:- 3	