**30 DAYS CODING CHALLENGE:**

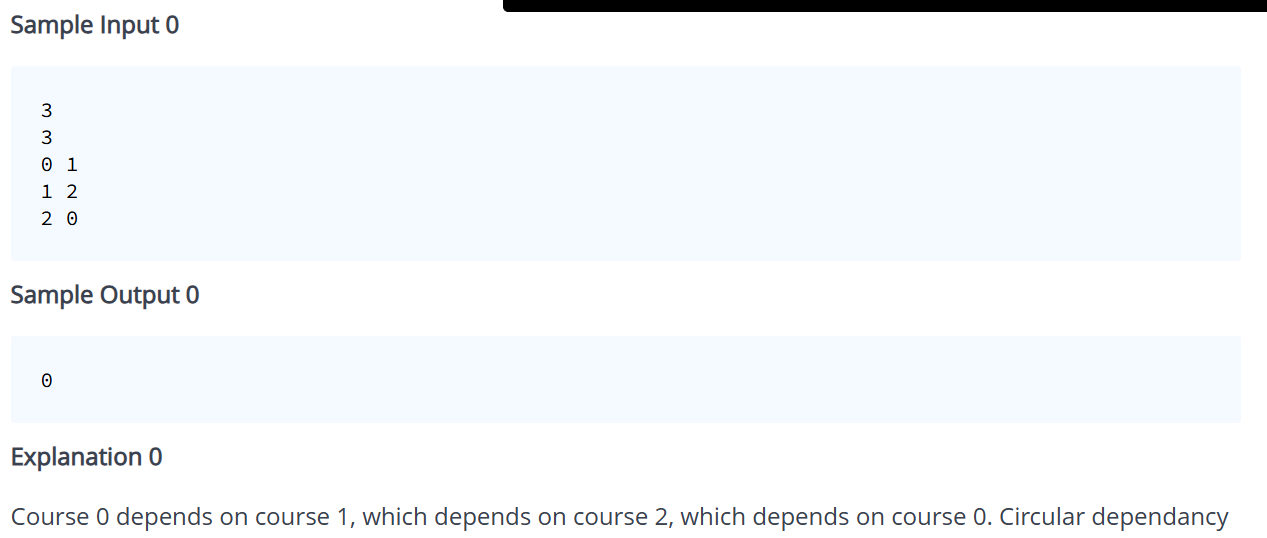
**DAY 24:**

**COURSE SCHEDULE 5:**

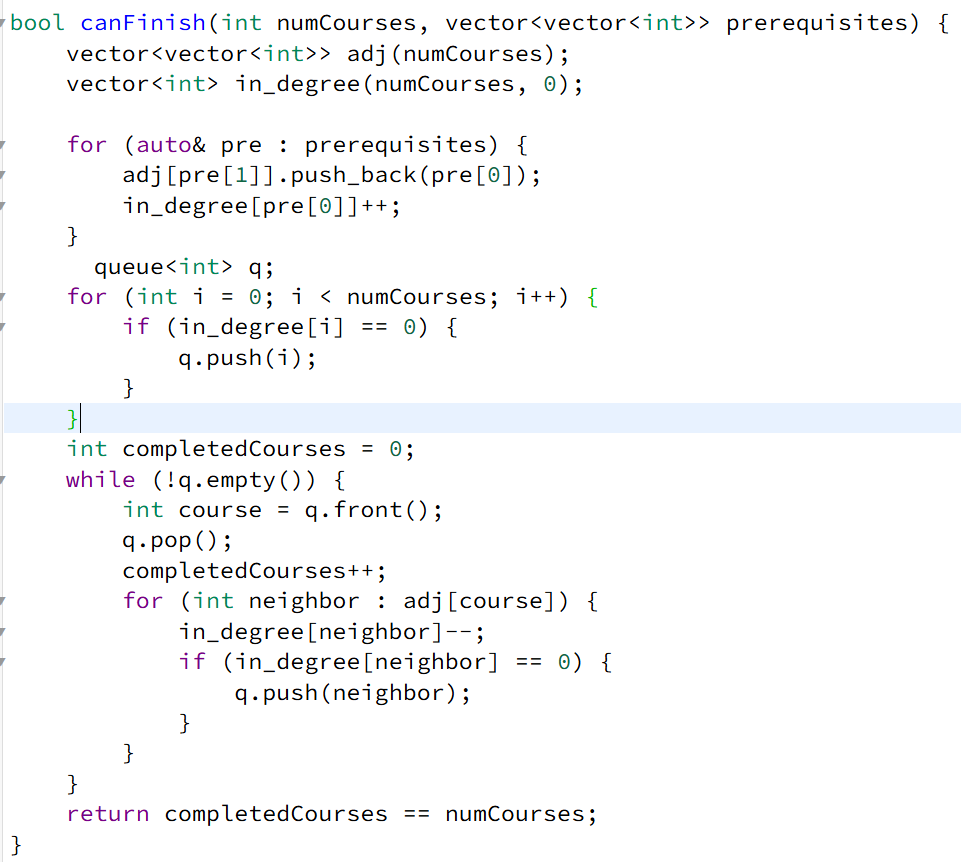
There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.



**SOURCE CODE:**

****

**DRY RUN:**

**Example Input**

* numCourses = 4
* prerequisites = {{1, 0}, {2, 1}, {3, 2}}

**Step 1: Build Graph & Compute In-degree**

* **Adjacency List:**
  + 0 → 1
  + 1 → 2
  + 2 → 3
* **In-degree Array:**
  + in\_degree = [0, 1, 1, 1]

**Step 2: Initialize Queue**

* Add courses with in-degree = 0 to the queue.
  + queue = [0] (only Course 0)

**Step 3: Process Courses in BFS**

1. **Process Course 0**
   * Queue: [0] → Remove 0
   * Update in\_degree = [0, 0, 1, 1]
   * Add 1 to queue → queue = [1]
2. **Process Course 1**
   * Queue: [1] → Remove 1
   * Update in\_degree = [0, 0, 0, 1]
   * Add 2 to queue → queue = [2]
3. **Process Course 2**
   * Queue: [2] → Remove 2
   * Update in\_degree = [0, 0, 0, 0]
   * Add 3 to queue → queue = [3]
4. **Process Course 3**
   * Queue: [3] → Remove 3
   * No more updates needed
   * Queue becomes empty

**Step 4: Check Completion**

* All numCourses processed → **Return true**

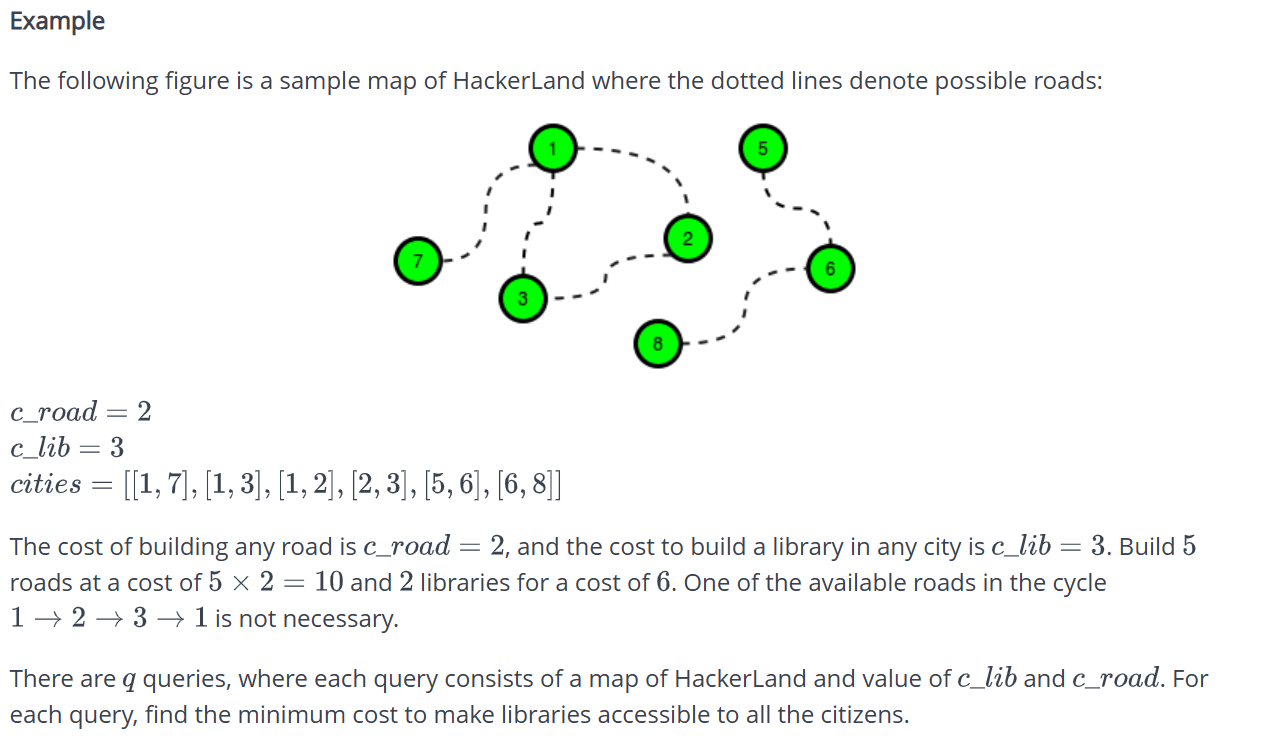
**DAY 25:**

**ROADS AND LIBRARIES:**

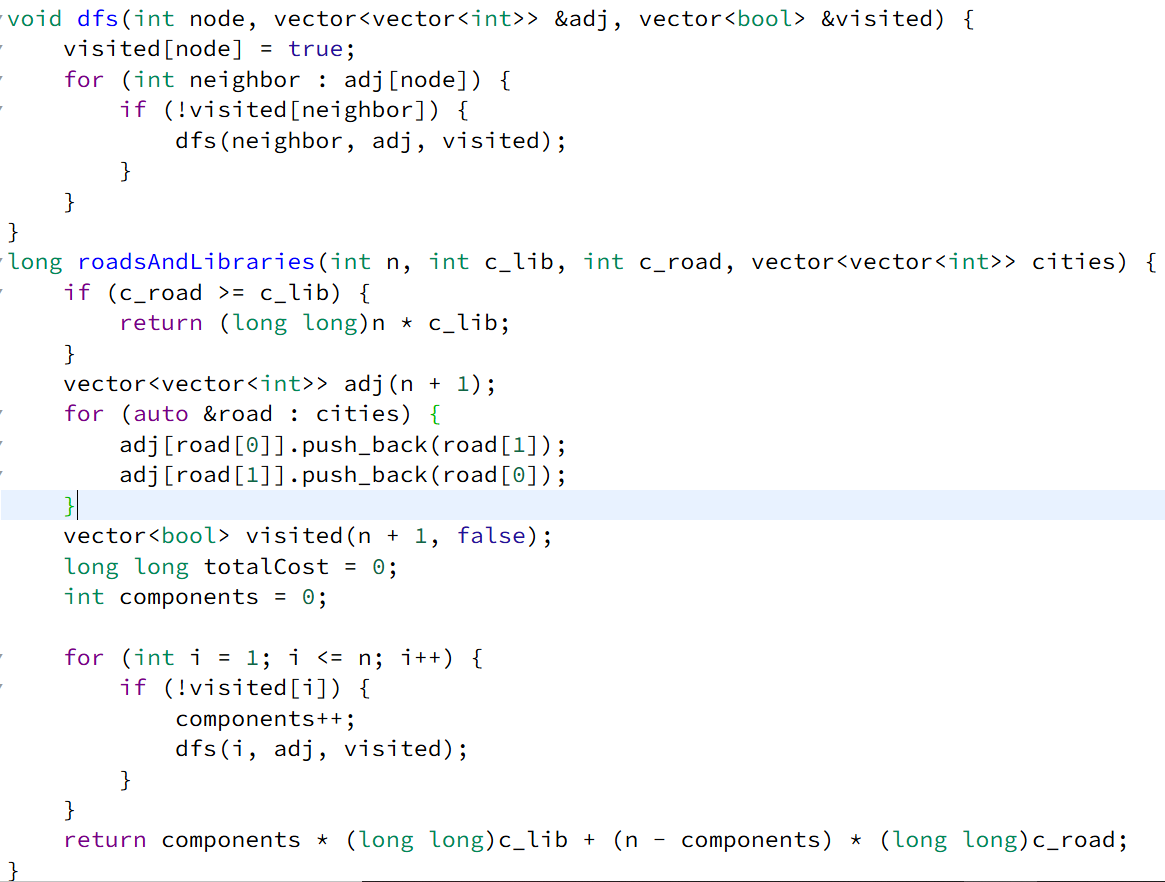
Determine the minimum cost to provide library access to all citizens of HackerLand. There are  cities numbered from  to . Currently there are no libraries and the cities are not connected. Bidirectional roads may be built between any city pair listed in . A citizen has access to a library if:

Their city contains a library.

They can travel by road from their city to a city containing a library.



**Source Code:**

****

**DRY RUN:**

**Example:**

n = 6, c\_lib = 2, c\_road = 1

cities = {{1, 2}, {1, 3}, {4, 5}, {5, 6}}

1. **Check Cost Condition:**
   * Since c\_road (1) < c\_lib (2), we proceed with roads and libraries.
2. **Build Adjacency List:**

adj[1] -> {2, 3}

adj[2] -> {1}

adj[3] -> {1}

adj[4] -> {5}

adj[5] -> {4, 6}

adj[6] -> {5}

1. **Initialize visited array:**

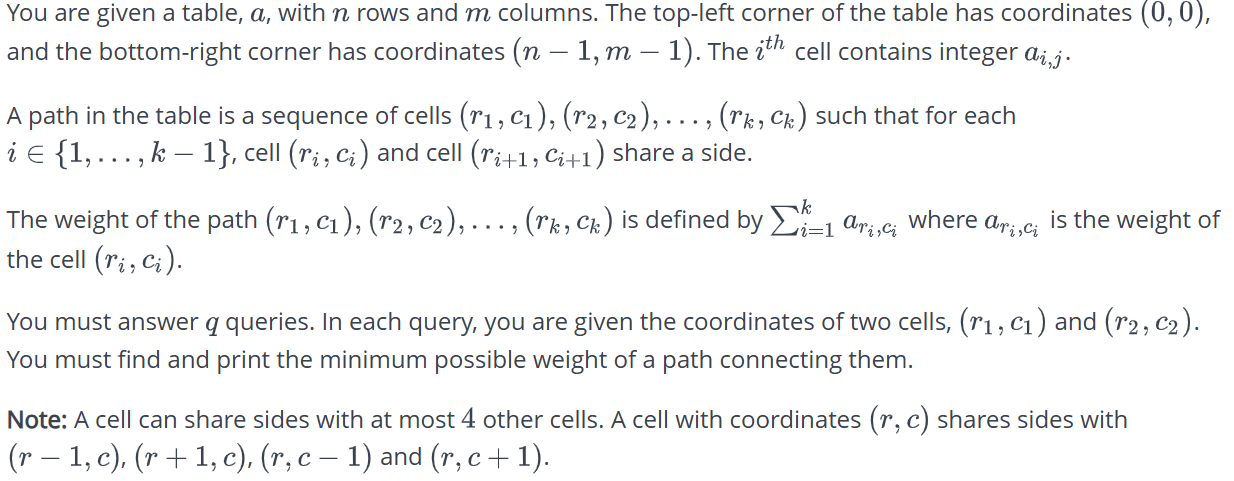
visited = {false, false, false, false, false, false, false} (7 elements, as indexing starts from 1)

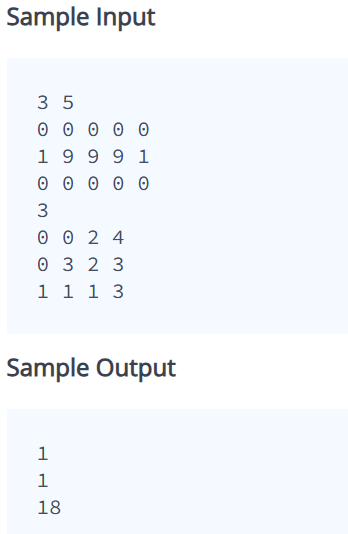
1. **Find Connected Components using DFS:**
   * Start DFS from **node 1** → Visits {1, 2, 3} → **First component found.**
   * Start DFS from **node 4** → Visits {4, 5, 6} → **Second component found.**
2. **Compute Cost:**
   * components = 2 (Libraries needed)
   * roads needed = n - components = 6 - 2 = 4
   * totalCost = (2 \* 2) + (4 \* 1) = 4 + 4 = 8

### ****Final Output:**** 8

**DAY 26:**

**FIND THE PATH:**

****

****

**SOURCE CODE:**

typedef pair<int, pair<int, int>> Node;

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

int dijkstra(vector<vector<int>>& grid, int r1, int c1, int r2, int c2) {

int n = grid.size(), m = grid[0].size();

vector<vector<int>> dist(n, vector<int>(m, INT\_MAX));

priority\_queue<Node, vector<Node>, greater<Node>> pq;

dist[r1][c1] = grid[r1][c1];

pq.push({dist[r1][c1], {r1, c1}});

while (!pq.empty()) {

auto [curr\_w, pos] = pq.top();

pq.pop();

int r = pos.first, c = pos.second;

if (r == r2 && c == c2) return curr\_w;

for (auto& d : directions) {

int nr = r + d[0], nc = c + d[1];

if (nr >= 0 && nr < n && nc >= 0 && nc < m) {

int new\_w = curr\_w + grid[nr][nc];

if (new\_w < dist[nr][nc]) {

dist[nr][nc] = new\_w;

pq.push({new\_w, {nr, nc}});

}

}

}

}

return -1;

}

vector<int> shortestPath(vector<vector<int>> a, vector<vector<int>> queries) {

vector<int> results;

for (auto& q : queries) {

int r1 = q[0], c1 = q[1], r2 = q[2], c2 = q[3];

results.push\_back(dijkstra(a, r1, c1, r2, c2));

}

return results;

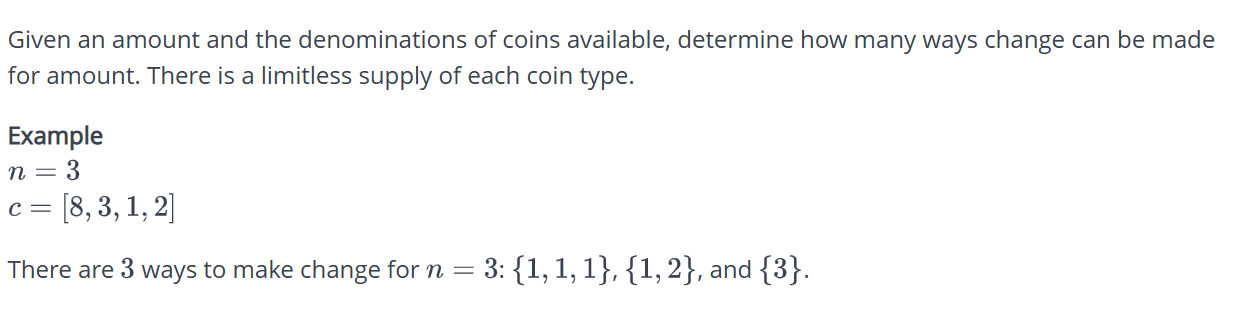
}

**DRY RUN:**

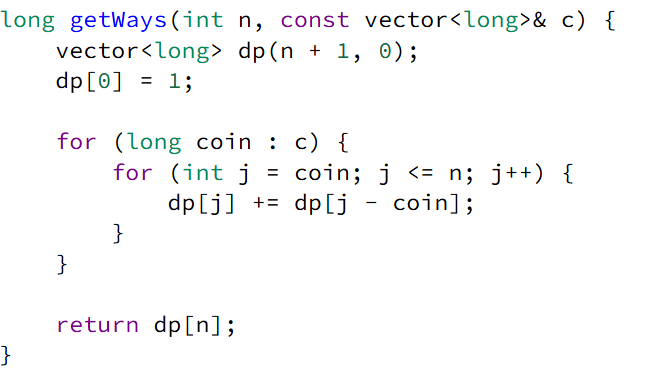
**Example:**

**DAY 27:**

**THE COIN CHANGE PROBLEM:**

****

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

#### ****Step 1: Initialization****

* Given **n = 5**, **coins = {1, 2, 5}**
* Create dp array of size n + 1: [0, 0, 0, 0, 0, 0]
* Set dp[0] = 1 (Only one way to make amount 0, by using no coins)

### ****Processing Coins One by One****

#### ****Step 2: Processing Coin**** 1

* Iterate from j = 1 to 5:
  + dp[1] += dp[0] → dp[1] = 1
  + dp[2] += dp[1] → dp[2] = 1
  + dp[3] += dp[2] → dp[3] = 1
  + dp[4] += dp[3] → dp[4] = 1
  + dp[5] += dp[4] → dp[5] = 1
* **Updated dp array**: [1, 1, 1, 1, 1, 1]

#### ****Step 3: Processing Coin**** 2

* Iterate from j = 2 to 5:
  + dp[2] += dp[0] → dp[2] = 2
  + dp[3] += dp[1] → dp[3] = 2
  + dp[4] += dp[2] → dp[4] = 3
  + dp[5] += dp[3] → dp[5] = 3
* **Updated dp array**: [1, 1, 2, 2, 3, 3]

#### ****Step 4: Processing Coin**** 5

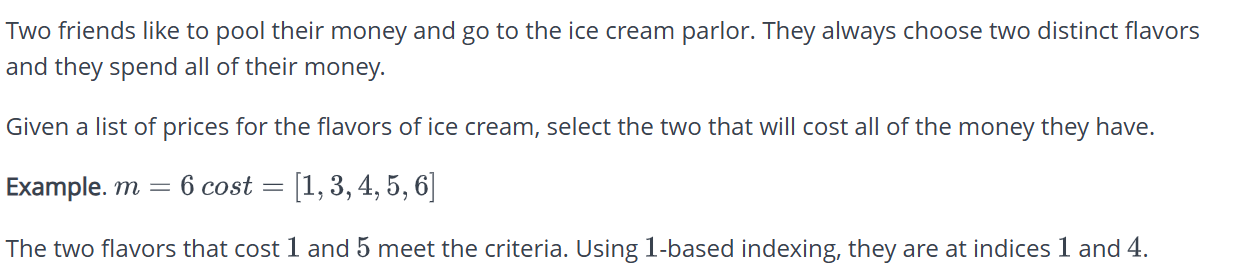
* Iterate from j = 5:
  + dp[5] += dp[0] → dp[5] = 4
* **Updated dp array**: [1, 1, 2, 2, 3, 4]

**OUTPUT:**

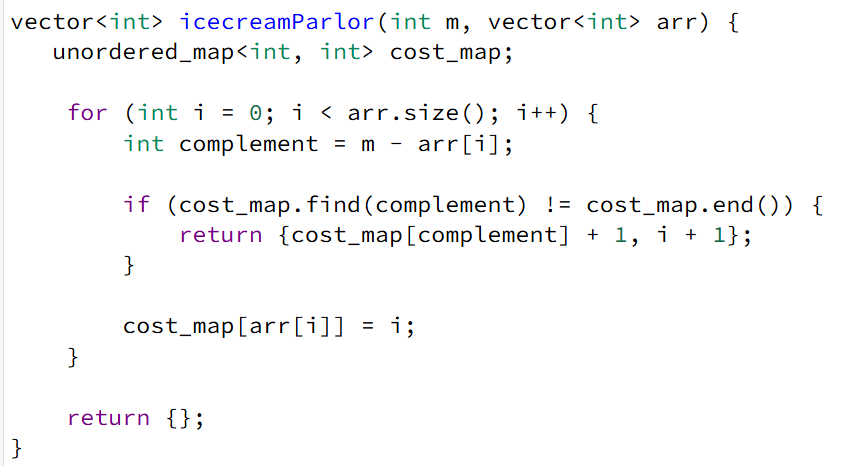
* dp[5] = 4, so the total ways to make change for 5 using {1, 2, 5} is **4**.

**DAY 28:**

**ICECREAM PARLOR:**

****

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

m = 6

arr = [1, 3, 4, 5, 6]

### ****Initialization****

* cost\_map = {} (Empty unordered map)
* m = 6
* arr = [1, 3, 4, 5, 6]

### ****Iteration 1 (i = 0)****

* **Current** arr[i] = 1
* **Complement =** m - arr[i] = 6 - 1 = 5
* **Check if** 5 **exists in** cost\_map → **No**
* **Store** 1 **in** cost\_map

cost\_map = {1: 0}

### ****Iteration 2 (i = 1)****

* **Current arr[i] = 3**
* **Complement = 6 - 3 = 3**
* **Check if 3 exists in cost\_map** → **No**
* **Store 3 in cost\_map**

cost\_map = {1: 0, 3: 1}

### ****Iteration 3 (i = 2)****

* **Current arr[i] = 4**
* **Complement = 6 - 4 = 2**
* **Check if 2 exists in cost\_map** → **No**
* **Store 4 in cost\_map**

cost\_map = {1: 0, 3: 1, 4: 2}

### ****Iteration 4 (i = 3)****

* **Current arr[i] = 5**
* **Complement = 6 - 5 = 1**
* **Check if 1 exists in cost\_map** → **Yes! (1 is found at index 0)**
* **Return indices (1-based index)**:
  + Stored index of 1 is 0 (from cost\_map).
  + Current index of 5 is 3.
  + **Return** {0+1, 3+1} → {1, 4}.

### ****Final Output****

1 4

**DAY 29:**

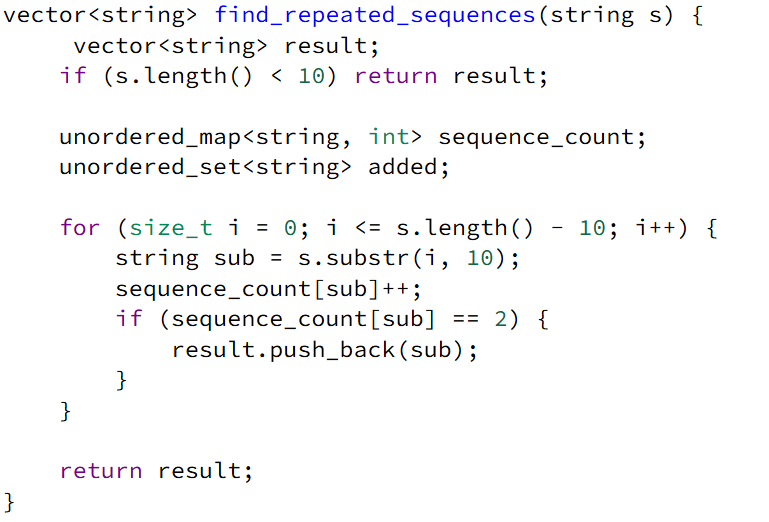
**REPEATED DNA SEQUENCES:**

The DNA sequence is composed of a series of nucleotides abbreviated as 'A', 'C', 'G', and 'T'.

For example, "ACGAATTCCG" is a DNA sequence. When studying DNA, it is useful to identify repeated sequences within the DNA.

Given a string s that represents a DNA sequence, return all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule. Return the answer in order of the substrings as they appear. If length of s is less than 10, return an empty array

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"

### ****Initialization****

* sequence\_count → empty {}
* result → empty []

### ****Iteration Over the String****

We extract 10-letter substrings and update sequence\_count:

1. **i = 0 → Substring = "AAAAACCCCC"**
   * sequence\_count["AAAAACCCCC"] = 1
   * Not added to result (appeared once).
2. **i = 1 → Substring = "AAAACCCCCA"**
   * sequence\_count["AAAACCCCCA"] = 1
   * Not added to result.
3. **i = 2 → Substring = "AAACCCCCAA"**
   * sequence\_count["AAACCCCCAA"] = 1
   * Not added to result.
4. **i = 3 → Substring = "AACCCCCAAA"**
   * sequence\_count["AACCCCCAAA"] = 1
   * Not added to result.
5. **i = 4 → Substring = "ACCCCCAAAA"**
   * sequence\_count["ACCCCCAAAA"] = 1
   * Not added to result.
6. **i = 5 → Substring = "CCCCCAAAAA"**
   * sequence\_count["CCCCCAAAAA"] = 1
   * Not added to result.
7. **i = 6 → Substring = "CCCCAAAAAC"**
   * sequence\_count["CCCCAAAAAC"] = 1
   * Not added to result.
8. **i = 7 → Substring = "CCCAAAAACC"**
   * sequence\_count["CCCAAAAACC"] = 1
   * Not added to result.
9. **i = 8 → Substring = "CCAAAAACCC"**
   * sequence\_count["CCAAAAACCC"] = 1
   * Not added to result.
10. **i = 9 → Substring = "CAAAAACCCC"**

* sequence\_count["CAAAAACCCC"] = 1
* Not added to result.

1. **i = 10 → Substring = "AAAAACCCCC"**

* sequence\_count["AAAAACCCCC"] = 2 (Repeated!)
* Added to result: ["AAAAACCCCC"].

1. **i = 11 → Substring = "AAAACCCCCC"**

* sequence\_count["AAAACCCCCC"] = 1
* Not added to result.

1. **i = 12 → Substring = "AAACCCCCCA"**

* sequence\_count["AAACCCCCCA"] = 1
* Not added to result.

1. **i = 13 → Substring = "AACCCCCCAA"**

* sequence\_count["AACCCCCCAA"] = 1
* Not added to result.

1. **i = 14 → Substring = "ACCCCCCAAA"**

* sequence\_count["ACCCCCCAAA"] = 1
* Not added to result.

1. **i = 15 → Substring = "CCCCCCAAAA"**

* sequence\_count["CCCCCCAAAA"] = 1
* Not added to result.

1. **i = 16 → Substring = "CCCCCAAAAA"**

* sequence\_count["CCCCCAAAAA"] = 2 (Repeated!)
* Added to result: ["AAAAACCCCC", "CCCCCAAAAA"].

1. **i = 17 → Substring = "CCCCAAAAAG"**

* sequence\_count["CCCCAAAAAG"] = 1
* Not added to result.

1. **i = 18 → Substring = "CCCAAAAAGG"**

* sequence\_count["CCCAAAAAGG"] = 1
* Not added to result.

1. **i = 19 → Substring = "CCAAAAAGGG"**

* sequence\_count["CCAAAAAGGG"] = 1
* Not added to result.

1. **i = 20 → Substring = "CAAAAAGGGT"**

* sequence\_count["CAAAAAGGGT"] = 1
* Not added to result.

1. **i = 21 → Substring = "AAAAAGGGTT"**

* sequence\_count["AAAAAGGGTT"] = 1
* Not added to result.

1. **i = 22 → Substring = "AAAAGGGTTT"**

* sequence\_count["AAAAGGGTTT"] = 1
* Not added to result.

**OUTPUT:**

AAAAACCCCC

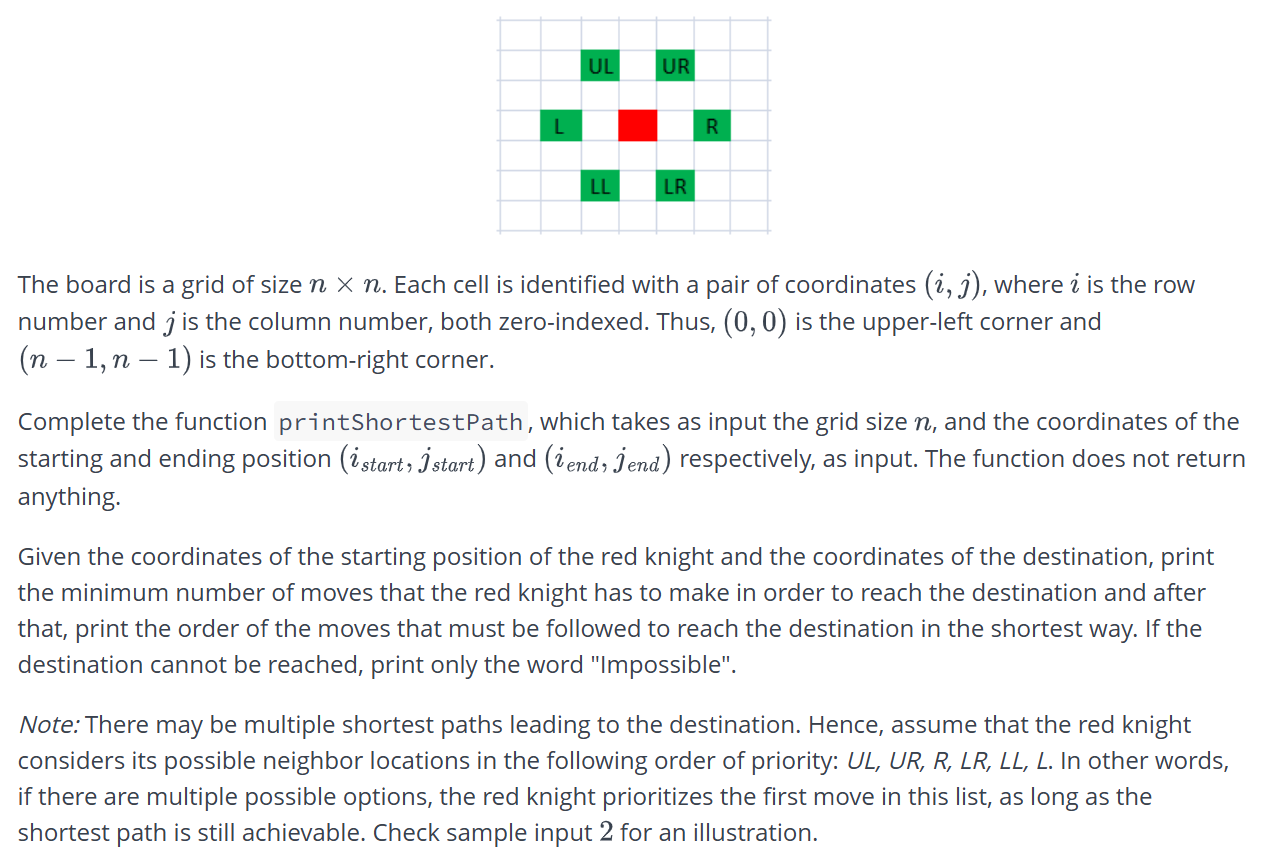
CCCCCAAAAA

**DAY 30:**

**RED KNIGHT’S SHORTEST PATH:**

In ordinary chess, the pieces are only of two colors, black and white. In our version of chess, we are including new pieces with unique movements. One of the most powerful pieces in this version is the *red knight*.

The red knight can move to six different positions based on its current position (UpperLeft, UpperRight, Right, LowerRight, LowerLeft, Left) as shown in the figure below.

****

**SOURCE CODE:**

const vector<pair<int, int>> moves = {{-2, -1}, {-2, 1}, {0, 2}, {2, 1}, {2, -1}, {0, -2}};

const vector<string> moveNames = {"UL", "UR", "R", "LR", "LL", "L"};

struct Node {

int x, y, dist;

vector<string> path;

};

void printShortestPath(int n, int i\_start, int j\_start, int i\_end, int j\_end) {

queue<Node> q;

vector<vector<bool>> visited(n, vector<bool>(n, false));

q.push({i\_start, j\_start, 0, {}});

visited[i\_start][j\_start] = true;

while (!q.empty()) {

Node current = q.front();

q.pop();

if (current.x == i\_end && current.y == j\_end) {

cout << current.dist << endl;

for (const string& move : current.path) {

cout << move << " ";

}

cout << endl;

return;

}

for (int i = 0; i < moves.size(); ++i) {

int newX = current.x + moves[i].first;

int newY = current.y + moves[i].second;

if (newX >= 0 && newX < n && newY >= 0 && newY < n && !visited[newX][newY]) {

visited[newX][newY] = true;

vector<string> newPath = current.path;

newPath.push\_back(moveNames[i]);

q.push({newX, newY, current.dist + 1, newPath});

}

}

}

cout << "Impossible" << endl;

}

**DRY RUN:**

### ****Initial Setup****

* Board size: 7 x 7
* Start position: (6,6)
* Destination: (0,1)
* Moves in priority order: UL, UR, R, LR, LL, L

### ****Iterations****

1. **Start at (6,6), distance = 0**
   * Possible moves: (4,5), (4,7), (6,8), (8,7), (8,5), (6,4)
   * Add (4,5), (4,7), and (6,4) to queue (valid within bounds)
2. **Move to (4,5), distance = 1**
   * Possible moves: (2,4), (2,6), (4,7), (6,6), (6,4), (4,3)
   * Add (2,4), (2,6), and (4,3) to queue
3. **Move to (4,7), distance = 1**
   * Possible moves: (2,6), (2,8), (4,9), (6,8), (6,6), (4,5)
   * Add (2,8)
4. **Move to (6,4), distance = 1**
   * Possible moves: (4,3), (4,5), (6,6), (8,5), (8,3), (6,2)
   * Add (6,2)
5. **Move to (2,4), distance = 2**
   * Possible moves: (0,3), (0,5), (2,6), (4,5), (4,3), (2,2)
   * Add (0,3), (0,5), and (2,2)
6. **Move to (2,6), distance = 2**
   * Possible moves: (0,5), (0,7), (2,8), (4,7), (4,5), (2,4)
   * Add (0,7)
7. **Move to (4,3), distance = 2**
   * Possible moves: (2,2), (2,4), (4,5), (6,4), (6,2), (4,1)
   * Add (4,1)
8. **Move to (0,3), distance = 3**
   * No valid moves within bounds
9. **Move to (0,5), distance = 3**
   * No valid moves within bounds
10. **Move to (4,1), distance = 3**
    * Possible moves: (2,0), (2,2), (4,3), (6,2), (6,0), (4,-1)
    * Add (2,0)
11. **Move to (2,0), distance = 4**
    * Possible moves: (0,-1), (0,1) ✅ Reached!

### ****Result****

* **Minimum moves:** 4
* **Path taken:** ["UL", "L", "L", "UL"]