Experiment 7(a)

Student Name: Punit Yadav

UID: 22BCS15444

Branch: CSE

Section/Group: KRG_IOT_3 A

Semester: 5

Date of Performance: 26-Sep-2024

Subject Name: Advanced Programming

Subject Code: 22CSP-314

1. Title: BFS Short Reach.

2. Aim:

Consider an undirected graph where each edge weighs 6 units. Each of the nodes is labelled consecutively from 1 to n. You will be given a number of queries. For each query, you will be given a list of edges describing an undirected graph. After you create a representation of the graph, you must determine and report the shortest distance to each of the other nodes from a given starting position using the breadthfirst search algorithm (BFS). Return an array of distances from the start node in node number order. If a node is unreachable, return -1 for that node.

3. Objective:

The objective of this experiment is to implement an efficient method for calculating the shortest paths in an undirected graph using BreadthFirst Search (BFS). This involves representing the graph with an adjacency list, executing BFS from a given start node, and returning the shortest distances to all other nodes, with -1 indicating unreachable nodes.

4. Algorithm:

Step1: Graph Representation:

Create an adjacency list to represent the undirected graph from the given list of edges.

Step 2: Initialize BFS:

Create a distance array of size n + 1 (where n is the number of nodes) initialized to -1. Set distance[startNode] to 0 to indicate the start node.
Create a queue and enqueue the startNode.

Step 3: BFS Traversal:

While the queue is not empty:

- Dequeue the front node currentNode.
- For each neighbor neighbor of currentNode:
  - If distance[neighbor] is -1 (indicating it has not been visited):
  - Update distance[neighbor] to distance[currentNode] + 6 (distance from the start node via the edge).
  - Enqueue neighbor.

Step 3: Return Distances:

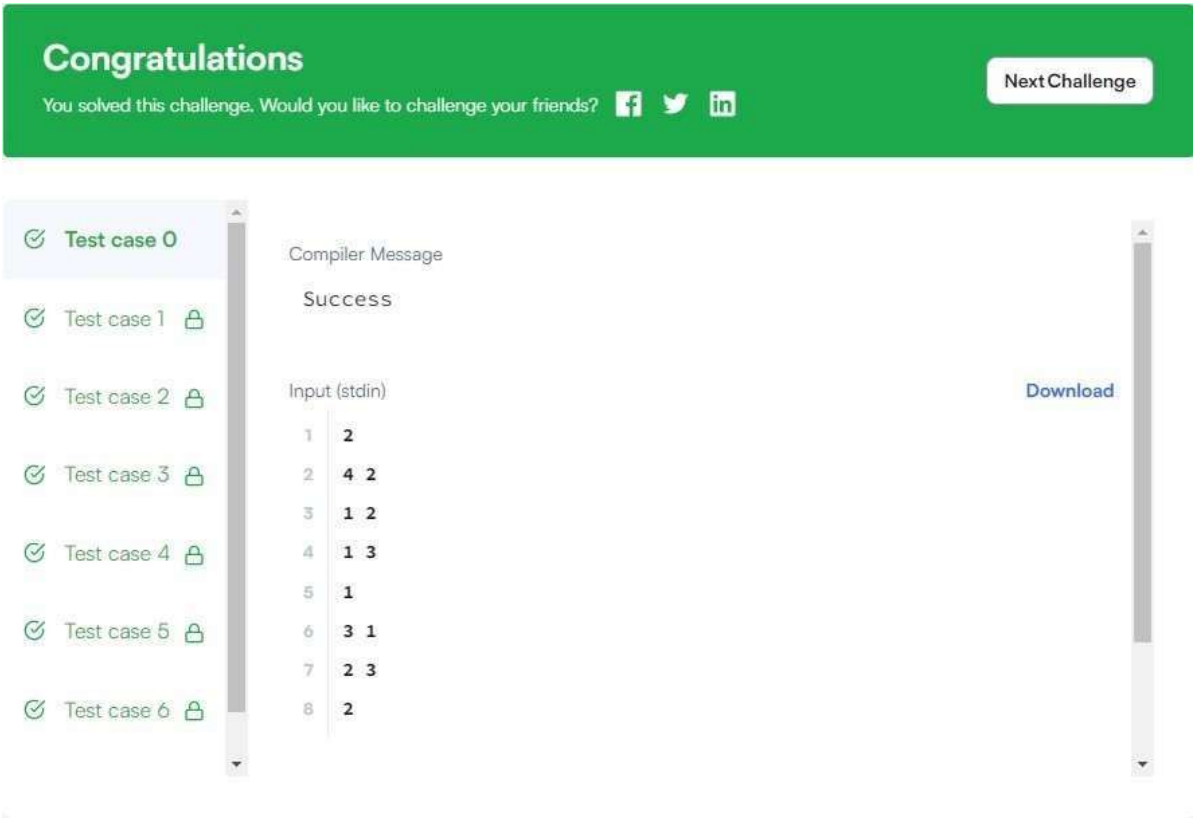
After BFS completes, return the distance array, which contains the shortest distances from the start node to each node.

5. Code:

```
#include <bits/stdc++.h>
using namespace std;

// Trim functions to remove leading/trailing spaces
string ltrim(const string &);
string rtrim(const string &);
vector<string> split(const string &);

// BFS function to calculate shortest paths
vector<int> bfs(int n, int m, vector<vector<int>> edges, int s) {
    // Create an adjacency list for the graph
    vector<vector<int>> adj(n + 1); // Nodes are 1-indexed
    for (const auto &edge : edges) {
        int u = edge[0], v = edge[1];
        adj[u].push_back(v);
        adj[v].push_back(u); // As the graph is undirected
    }

    // Distance vector to store the shortest distance from the start node
    vector<int> distance(n + 1, -1);
    queue<int> q;
    distance[s] = 0;
    q.push(s);

    // BFS to traverse the graph
    while (!q.empty()) {
        int node = q.front();
        q.pop();

        for (int neighbor : adj[node]) {
```

```
            if (distance[neighbor] == -1) { // If not visited
                distance[neighbor] = distance[node] + 6; // Unweighted graph: edge weight is6
                q.push(neighbor);
            }
        }
    }

    // Prepare the result excluding the start node
    vector<int> result;
    for (int i = 1; i <= n; i++) {
        if (i != s) {
            result.push_back(distance[i]);
        }
    }

    return result;
}

int main() {
    ofstream fout(getenv("OUTPUT_PATH"));

    string q_temp;
    getline(cin, q_temp);
    int q = stoi(ltrim(rtrim(q_temp))); // Read the number of test cases

    // Process each test case
    for (int q_itr = 0; q_itr < q; q_itr++) {
        string first_multiple_input_temp;
        getline(cin, first_multiple_input_temp);
        vector<string> first_multiple_input = split(rtrim(first_multiple_input_temp));

        int n = stoi(first_multiple_input[0]); // Number of nodes
        int m = stoi(first_multiple_input[1]); // Number of edges

        // Read the edges
        vector<vector<int>> edges(m);
        for (int i = 0; i < m; i++) {
```

6. Output:



7. Learning outcomes:

1. Graph Representation: Understand how to represent graphs using adjacency lists for efficient traversal.
2. BFS Algorithm: Learn to implement Breadth-First Search to compute shortest paths in unweighted graphs.
3. Queue Utilization: Use queues to explore nodes level by level in BFS.
4. Edge Case Handling: Handle unreachable nodes by marking them with a default value (e.g., -1).

Time Complexity: O(n+m)
Space Complexity: O(n+m)

Experiment 7(b)

Student Name: Punit Yadav

UID: 22BCS15444

Branch: CSE

Section/Group: KRG_IOT_3 A

Semester: 5

Date of Performance: 26-Sep-2024

Subject Name: Advanced Programming

Subject Code: 22CSP-314

1. **Title**: Quickest Way Up.

2. **Aim**:

Markov takes out his Snakes and Ladders game, stares at the board and wonders: "If I can always roll the die to whatever number I want, what would be the least number of rolls to reach the destination?"

Rules The game is played with a cubic die of 6 faces numbered 1 to 6.

- Starting from square 1, land on square 100 with the exact roll of the die. If moving the number rolled would place the player beyond square 100, no move is made.
- If a player lands at the base of a ladder, the player must climb the ladder. Ladders go up only.
- If a player lands at the mouth of a snake, the player must go down the snake and come out through the tail. Snakes go down only.

3. **Objective**:

The objective is to determine the minimum number of dice rolls needed to reach square 100, accounting for ladders that advance the player and snakes that send the player backward, while ensuring the player lands exactly on square 100.

4. **Algorithm**:

1. Graph Representation: Treat the board as a graph with 100 nodes (each square representing a node). Each node is connected to the next 6 nodes (dice rolls), with additional edges for ladders and snakes.
2. Initialize: Create an array moves [] where each index represents a square. For ladders and snakes, update moves[i] to point to the destination square (either the top of the ladder or the tail of the snake).
3. BFS Setup: Use Breadth-First Search (BFS) to explore the shortest path. Initialize a queue and start from square 1. Maintain a distance array dist[] to track the minimum number of dice rolls to reach each square.

4. BFS Execution:
   - For each square, check all possible dice rolls (1 to 6).
   - For each roll, move the player to the corresponding square and check for ladders or snakes to adjust the position.
   - If the new square hasn't been visited, update its distance and enqueue it.
5. Terminate: The BFS terminates when you reach square 100, and the result is the minimum number of rolls stored in the distance array.

5. Code:

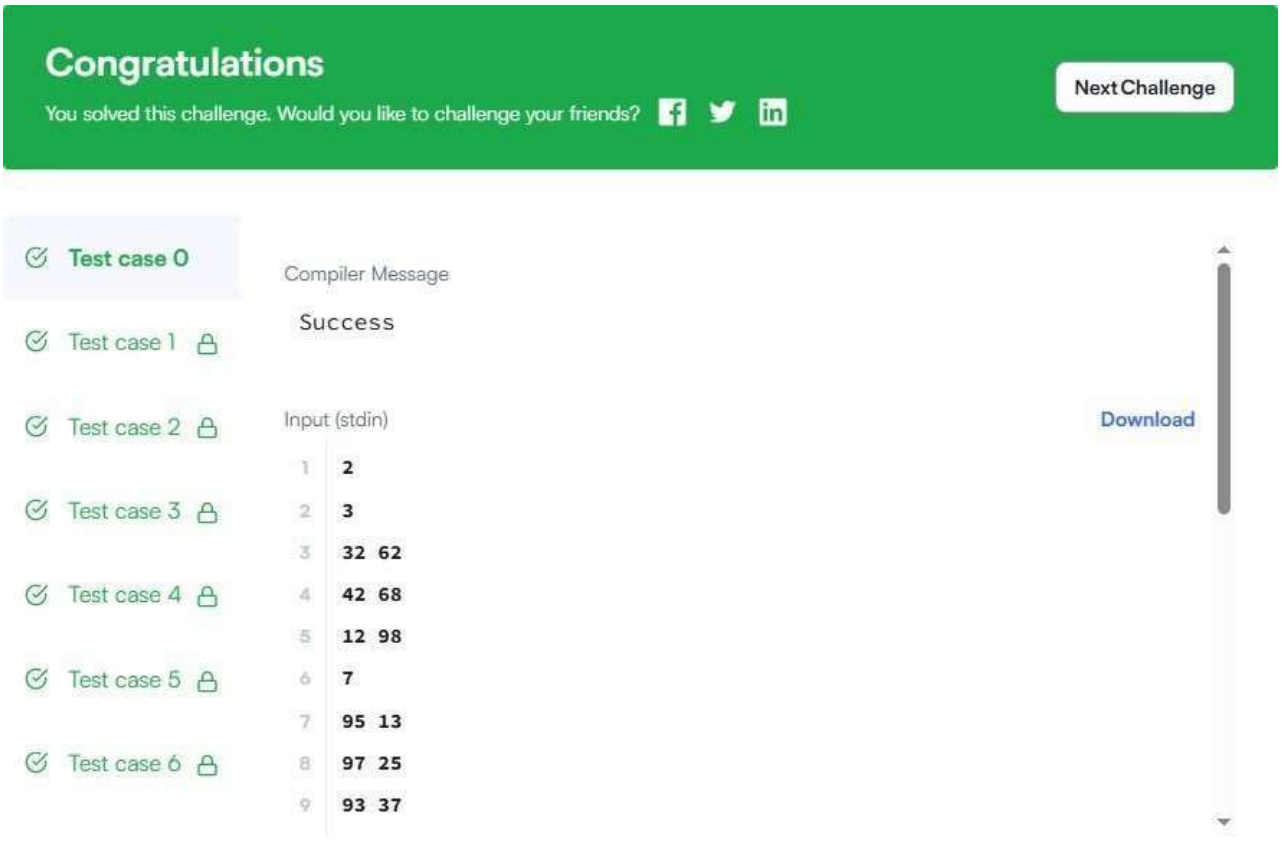
```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

const int N = 104;
const int INF = 100000000;

int main()
{
   int t;
   cin >> t;
   for (int k = 1; k <= t; ++k)
   {
      vector<int> graph(N, 0); // Graph stores ladders and snakes
      vector<bool> mark(N, false);  // Mark to track visited positions
      int n, m;

      // Input ladders
      cin >> n;
      for (int i = 0; i < n; ++i)
      {
         int a, b;
         cin >> a >> b;
         graph[a] = b; // Ladder from a to b
      }

      // Input snakes
      cin >> m;
      for (int i = 0; i < m; ++i)
      {
         int a, b;
```

```
    cin >> a >> b;
    graph[a] = b; // Snake from a to b
}

queue<pair<int, int>> q;  // Pair of (current position, number of moves)
int ans = INF;

// Start from position 1 with 0 moves
q.push(make_pair(1, 0));
mark[1] = true;

while (!q.empty())
{
    pair<int, int> p = q.front();
    q.pop();

    if (p.first == 100)  // Reached position 100
    {
        ans = p.second;
        break;
    }

    // Check for all possible moves (1 to 6)
    for (int i = 1; i <= 6; ++i)
    {
        int x = p.first + i;
        if (x > 100)
            continue;

        if (!mark[x])
        {
            mark[x] = true;

            // If there's no ladder or snake, move to next position
            if (graph[x] == 0)
                q.push(make_pair(x, p.second + 1));
            else
            {
                // Move to the end of the ladder/snake
                x = graph[x];
                mark[x]  = true;
                q.push(make_pair(x, p.second + 1));
```

Discover. Learn. Empower.

```
                }
            }
        }
    }

    // Output the result
    if (ans == INF)
        cout << -1 << endl; // No possible way to reach 100
    else
        cout << ans << endl;  // Minimum moves to reach 100
  }
}
```

6. Output:



7. Learning outcomes:

1. Graph Representation: Learn to model a game board as a graph, where each node represents a square and edges represent possible dice rolls and game mechanics like ladders and snakes.

2. BFS Algorithm: Understand how Breadth-First Search (BFS) can be used to find the shortest path in an unweighted graph, which is essential for solving problems involving minimum steps or moves.

3. Handling Game Mechanics: Gain experience in incorporating specific game rules (e.g., ladders and snakes) into the graph traversal to adapt general algorithms to problem-specific constraints.

Time Complexity: O(1)
Space Complexity: O(1)

Experiment 6 (a)

Student Name: Punit Yadav **UID:** 22BCS15444

Branch: CSE **Section/Group:** KRG_IOT_3 A

Semester: 5 **Date of Performance:** 19-Sep-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: - You are given a tree (a simple connected graph with no cycles). Find the maximum number of edges you can remove from the tree to get a forest such that each connected component of the forest contains an even number of nodes.

2. Objective:

The objective of this experiment is to determine how many ways you can remove an edge from a tree to create two separate subtrees such that both resulting subtrees have an even number of nodes.

3. Implementation/Code:

```
#include <bits/stdc++.h>
using namespace std;

// Function to compute the number of removable edges
int evenForest(int t_nodes, int t_edges, vector<int>& t_from, vector<int>& t_to) {
    // Create an adjacency list
    unordered_map<int, list<int>> adjacencyList;
    for (int i = 1; i <= t_nodes; i++) {
        adjacencyList[i] = list<int>();
    }
```

```
// Build the adjacency list based on the edges
for (int i = 0; i < t_edges; i++) {
    int u = t_from[i];
    int v = t_to[i];
    adjacencyList[u].push_back(v);
    adjacencyList[v].push_back(u);
}

// Array to store the size of each subtree
vector<int> subtreeSize(t_nodes + 1, 0);
vector<bool> visited(t_nodes + 1, false);

// Start DFS from node 1 (root)
auto dfs = [&](int node, auto& dfs_ref) -> void {
    visited[node] = true;
    subtreeSize[node] = 1;

    for (int neighbor : adjacencyList[node]) {
        if (!visited[neighbor]) {
            dfs_ref(neighbor, dfs_ref);
            subtreeSize[node] += subtreeSize[neighbor];
        }
    }
};

dfs(1, dfs);
// Count the number of edges that can be removed
int removableEdges = 0;
```

```
    for (int i = 2; i <= t_nodes; i++) {
        if (subtreeSize[i] % 2 == 0) {
            removableEdges++;
        }
    }
    return removableEdges;
}
int main() {
    // Example test case
    vector<int> t_from = {1, 1, 3, 3, 4, 6, 6, 8, 8};
    vector<int> t_to = {3, 2, 6, 4, 5, 8, 7, 9, 10};
    cout << evenForest(10, 9, t_from, t_to) << endl;
    return 0;
}
```

4. Output:

Experiment 6 (b)

Student Name: Punit Yadav **UID:** 22BCS15444

Branch: CSE **Section/Group:** KRG_IOT_3 A

Semester: 5 **Date of Performance:** 19-Sep-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: - You are given a pointer to the root of a binary search tree and values to be inserted into the tree. Insert the values into their appropriate position in the binary search tree and return the root of the updated binary tree. You just have to complete the function.

2. Objective:

The objective of this experiment is to implement a function in Java that determines the minimum number of characters needed to make a given password "strong" according to specific criteria.

3. Code:

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

// Definition of the Node structure
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr), right(nullptr) {}
```

```
};

// Function to insert a node in the BST
Node* insert(Node* root, int data) {
    // If the tree is empty, create a new node and return it
    if (root == nullptr) {
        return new Node(data);
    }

    // Otherwise, recur down the tree and insert the node
    if (data <= root->data) {
        root->left = insert(root->left, data);
    } else {
        root->right = insert(root->right, data);
    }

    // Return the (unchanged) node pointer
    return root;
}

// Helper function to perform in-order traversal (for testing)
void inorder(Node* root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}
```

```
int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    // Print the in-order traversal of the tree
    inorder(root);  // Output: 20 30 40 50 60 70 80

    return 0;
}
```

4. Output:

5. Learning Outcome

i. Enhance problem-solving skills by breaking down a complex problem into smaller, more manageable components (like node insertion and traversal)

ii. Apply concepts of graph theory to solve problems involving trees and subtrees.

iii. Develop algorithms to process and analyze tree structures, specifically focusing on subtree properties and even node counts.

iv. Gain experience in designing recursive functions for common tree operations such as insertion and traversal.

v. Learn the properties of BSTs and how to maintain these properties during insertion operations.

Experiment 9(a)

Student Name: Punit Yadav **UID:** 22BCS15444

Branch: CSE **Section:** 22BCS_KRG_IOT_3 A

Semester: 5 **Date of Performance:** 17-Oct-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: Given a partially filled 9 x 9 sudoku board, you need to output a solution to it. Input-The first line contains N which is the number of Sudoku puzzles. N Sudoku puzzles follow. Each one contains 9 lines with each line containing 9 space separated integers. A 0 indicates an unfilled square, and an integer between 1 and 9 denotes a filled square with that value. Output-Output to STDOUT the completely filled Sudoku board for each Sudoku. If there are multiple solutions, you can output any of the one.

2. Objective:

The objective is to solve and output the completed solution of N given 9x9 partially filled Sudoku puzzles, filling in the unfilled squares while ensuring all Sudoku rules are followed.

3. Implementation/Code:

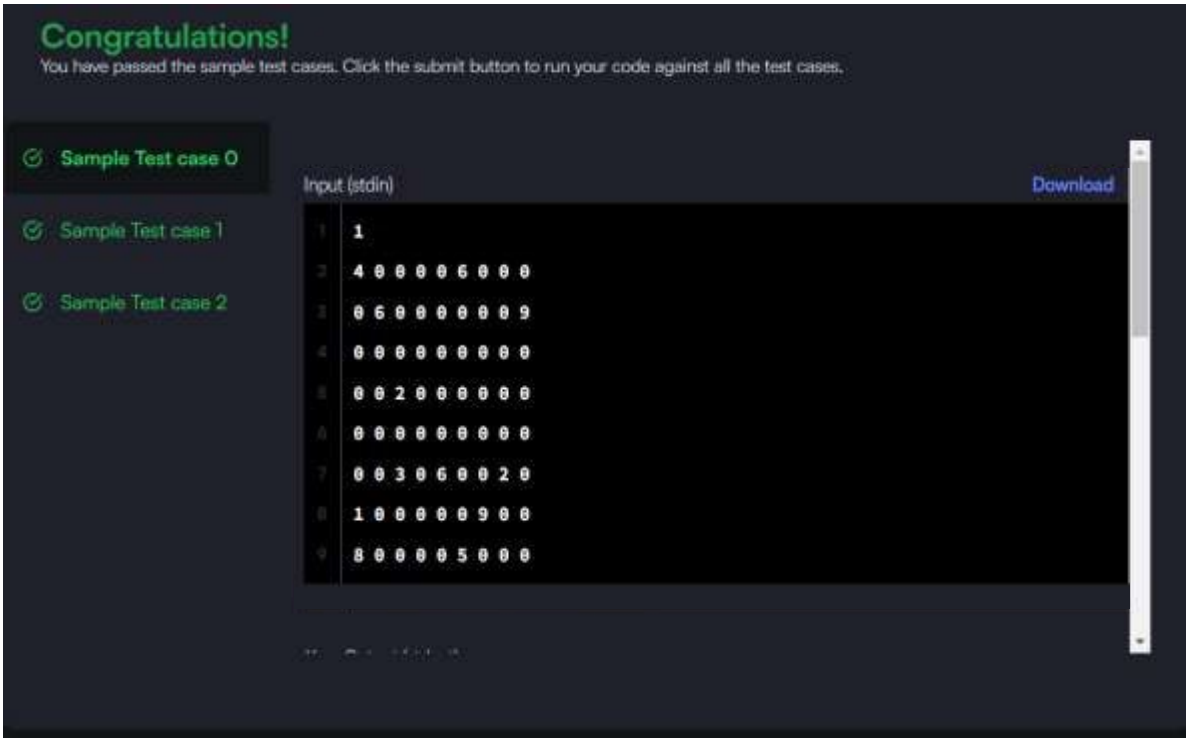
```
#include<iostream>

#include<vector>

#include<set>

#include<math.h>


using namespace std;
```

```
int getSector(int row, int col) {
    return (row/3)*3+(col/3);
}

bool valid(int grid[9][9], int row, int col) {
    for (int i = 0; i < 9; i++) {
        if (grid[row][i] == grid[row][col] && i != col) return false;
        if (grid[i][col] == grid[row][col] && i != row) return false;
    }
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (getSector(row, col) == getSector(i, j) && row != i && col != j && grid[i][j] ==
grid[row][col]) return false;
        }
    }

    return true;
}

/* Head ends here */
void sudoku_solve(int grid[9][9],set<int> set) {
    //your logic here
    int forward = 1;
    for (int idx = 0; idx < 81;) {
        int row = floor(idx/9);
```

```
            int col = fmod(idx,9);
            if (set.count(idx)>0) {
              idx += 1*forward;
            } else {
               grid[row][col]++;
               if (grid[row][col] == 10) {
                  grid[row][col] = 0;
                  forward = -1;
                  idx--;
               }
               if (valid(grid, row, col)) {
                  forward = 1;
                  idx++;
               }
            }
         }
         for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
               cout << grid[i][j] << " ";
            }
            cout << endl;
         }
      }
```

```
/* Tail starts here */
int main() {
    int n, board[9][9];
    set<int> set;
    cin >> n;
    for(int i=0;i<n;i++) {
        for(int j=0;j<9;j++) {
            for(int k=0;k<9;k++) {
                board[j][k] = 0;
                cin >> board[j][k];
                if ( board[j][k] != 0 ) set.insert(j*9+k);
            }
        }
        sudoku_solve(board,set);
    }
    return 0;
}
```

4. Output:


Congratulations!
You have passed the sample test cases. Click the submit button to run your code against all the test cases.
Sample Test case 0
Sample Test case 1
Sample Test case 2
Input (stdin)
Download
1
4 0 0 0 0 6 0 0 0
0 6 0 0 0 0 0 0 9
0 0 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 3 0 6 0 0 2 0
1 0 0 0 0 0 9 0 0
8 0 0 0 0 5 0 0 0

Experiment 9(b)

Student Name: Punit Yadav **UID:** 22BCS15444

Branch: CSE **Section:** 22BCS_KRG_IOT_3 A

Semester: 5 **Date of Performance:** 17-Oct-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: A 10*10 Crossword grid is provided to you, along with a set of words (or names of places) which need to be filled into the grid. Cells are marked either + or -. Cells marked with a - are to be filled with the word list.

2. Objective:

To fill a 10x10 crossword grid, where cells marked with '-' can be filled with a set of provided words (or names of places), ensuring the words fit correctly either horizontally or vertically.

3. Code:

```
#include<iostream>
#include<vector>
#include<string>
using namespace std;
// Function to check if a word can be placed horizontally
bool canPlaceHorizontally(vector<string>& grid, string word, int row, int col) {
   if (col + word.size() > 10)
      return false;
   for (int i = 0; i < word.size(); i++) {
      if (grid[row][col + i] != '-' && grid[row][col + i] != word[i])
         return false;
   }
```

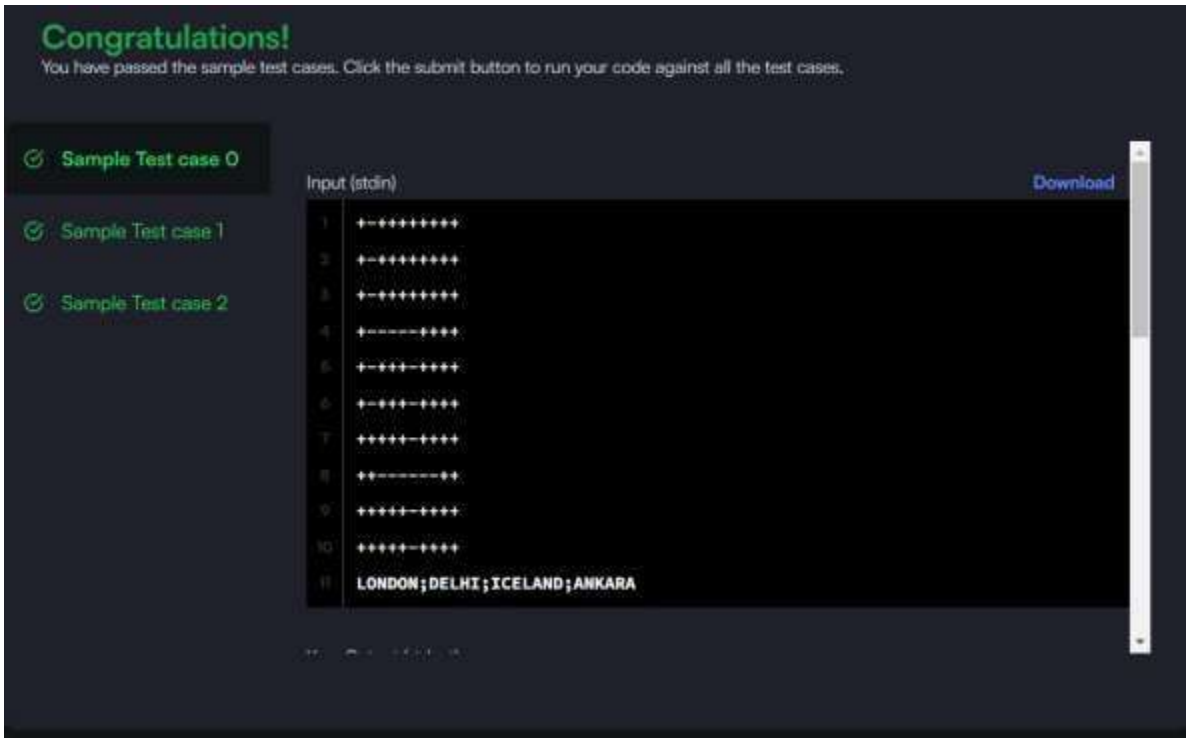
```
    return true;
}

// Function to place a word horizontally in the grid
void placeHorizontally(vector<string>& grid, string word, int row, int col,
vector<bool>& placed) {
    for (int i = 0; i < word.size(); i++) {
        if (grid[row][col + i] == '-') {
            grid[row][col + i] = word[i];
            placed[i] = true;
        }
    }
}

// Function to remove a horizontally placed word
void unplaceHorizontally(vector<string>& grid, string word, int row, int col,
vector<bool>& placed) {
    for (int i = 0; i < word.size(); i++) {
        if (placed[i])
            grid[row][col + i] = '-';
    }
}

// Function to check if a word can be placed vertically
bool canPlaceVertically(vector<string>& grid, string word, int row, int col) {
    if (row + word.size() > 10)
        return false;
    for (int i = 0; i < word.size(); i++) {
        if (grid[row + i][col] != '-' && grid[row + i][col] != word[i])
            return false;
    }
```

```
    return true;
}

// Function to place a word vertically in the grid
void placeVertically(vector<string>& grid, string word, int row, int col, vector<bool>&
placed) {
    for (int i = 0; i < word.size(); i++) {
        if (grid[row + i][col] == '-') {
            grid[row + i][col] = word[i];
            placed[i] = true;
        }
    }
}

// Function to remove a vertically placed word
void unplaceVertically(vector<string>& grid, string word, int row, int col, vector<bool>&
placed) {
    for (int i = 0; i < word.size(); i++) {
        if (placed[i])
            grid[row + i][col] = '-';
    }
}

// Backtracking function to solve the crossword puzzle
bool solveCrossword(vector<string>& grid, vector<string>& words, int index) {
    if (index == words.size())
        return true;

    string word = words[index];

    for (int i = 0; i < 10; i++) {
```

```
            for (int j = 0; j < 10; j++) {
                // Try placing the word horizontally
                if (canPlaceHorizontally(grid, word, i, j)) {
                    vector<bool> placed(word.size(), false);
                    placeHorizontally(grid, word, i, j, placed);

                    if (solveCrossword(grid, words, index + 1))
                        return true;

                    unplaceHorizontally(grid, word, i, j, placed);
                }

                // Try placing the word vertically
                if (canPlaceVertically(grid, word, i, j)) {
                    vector<bool> placed(word.size(), false);
                    placeVertically(grid, word, i, j, placed);

                    if (solveCrossword(grid, words, index + 1))
                        return true;

                    unplaceVertically(grid, word, i, j, placed);
                }
            }
        }

        return false;
    }
    int main() {
        vector<string> grid(10);

        for (int i = 0; i < 10; i++) {
```

```
        cin >> grid[i];
    }

    string words_input;
    cin >> words_input;

    vector<string> words;
    string word = "";
    for (char c : words_input) {
        if (c == ';') {
            words.push_back(word);
            word = "";
        } else {
            word += c;
        }
    }
    if (!word.empty()) words.push_back(word);

    solveCrossword(grid, words, 0);

    // Output the filled grid
    for (int i = 0; i < 10; i++) {
        cout << grid[i] << endl;
    }

    return 0;
}
```

4. Output:



5. Learning Outcome

1. Gained knowledge of how backtracking algorithms can be used to solve constraint satisfaction problems, like placing words in a crossword puzzle.
2. Developed the ability to break down a complex problem into smaller subproblems and solve them using recursive functions.
3. Acquired skills in working with 2D grids or matrices by navigating through the grid and managing row/column placements effectively.

Experiment 8(a)

Student Name: Punit Yadav
UID: 22BCS15444
Branch: CSE
Section/Group: KRG_IOT_3 A
Semester: 5
Date of Performance: 03-Oct-2024
Subject Name: Advanced Programming
Subject Code: 22CSP-314

1. **Aim:**

Problem Statement:

Alice is a kindergarten teacher. She wants to give some candies to the children in her class. All the children sit in a line and each of them has a rating score according to his or her performance in the class. Alice wants to give at least 1 candy to each child. If two children sit next to each other, then the one with the higher rating must get more candies. Alice wants to minimize the total number of candies she must buy.

2. **Objective:**

The objective is to determine the minimum number of candies Alice needs to distribute to children seated in a line based on their rating scores. Each child must receive at least one candy, and children with higher ratings than their adjacent neighbours must receive more candies than those neighbours.

3. **Implementation/Code:**

```
#include <iostream>
#include <vector>
#include <numeric>
#include <iomanip>

using namespace std;

class Solution {
public:
    static long long candies(int n, vector<int>& arr) {
        if (n == 0) return 0; // Edge case for empty array

        vector<int> cache(n, 1); // Initialize cache with 1 for each child

        // First pass: left to right
        for (int i = 1; i < n; i++) {
```

```
            if (arr[i] > arr[i - 1]) {
                cache[i] = cache[i - 1] + 1; // Give more candies than the left child
            }
        }

        // Second pass: right to left
        for (int i = n - 2; i >= 0; i--) {
            if (arr[i] > arr[i + 1]) {
                // Ensure the current child has more candies than the right child
                if (cache[i] <= cache[i + 1]) {
                    cache[i] = cache[i + 1] + 1;
                }
            }
        }

        // Calculate the total number of candies
        long long sum = accumulate(cache.begin(), cache.end(), 0LL); // Use accumulate to
sum up

        return sum;
    }
};

int main() {
    // Example usage
    vector<int> ratings = {1, 0, 2};
    long long totalCandies = Solution::candies(ratings.size(), ratings);
    cout << "Total candies required: " << totalCandies << endl;
    return 0;
}
```

4. Output:


Test case 0
Test case 1
Test case 2
Test case 3
Test case 4
Test case 5
Test case 6
Compiler Message
Success
Input (stdin)
Download
3
1
2
2
Expected Output
Download
4

Experiment 8(b)

Student Name: Punit Yadav **UID:** 22BCS15444

Branch: CSE **Section/Group:** KRG_IOT_3 A

Semester: 5 **Date of Performance:** 03-Oct-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement:

Marc loves cupcakes, but he also likes to stay fit. Each cupcake has a calorie count, and Marc can walk a distance to expend those calories. If Marc has eaten j cupcakes so far, after eating a cupcake with c calories he must walk at least 2j *cmiles to maintain his weight.

2. Objective:

The objective is to calculate the minimum total distance Marc needs to walk to offset the calories from the cupcakes he has eaten. Given that Marc must walk an increasing distance proportional to the number of cupcakes consumed, the goal is to determine the total walking distance required to balance out the calorie intake from all cupcakes.

3. Code:

```
class Result {

   public static long marcsCakewalk(List<Integer> calorie) {
   // Write your code here
    Collections.sort(calorie, Collections.reverseOrder());

      long totalCandies = 0;        for (int i = 0; i < calorie.size(); i++) {
totalCandies += (1L << i) * calorie.get(i);
      }

      return totalCandies;
```

```
    }

}
```

4. Output:

5. Learning Outcome

- Learn to solve optimization problems where constraints involve comparative values between adjacent elements.
- Understand how to implement a two-pass algorithm to achieve the optimal solution in such scenarios.
- learn how to apply cumulative cost calculations in scenarios involving variable constraints.
- They will understand how to optimize and calculate required values based on a progressive increase in factors, such as the distance Marc must walk per cupcake.
- This exercise will enhance problem-solving skills in handling scenarios with incremental constraints and applying mathematical calculations effectively.

Experiment 5(a)

Student Name: Punit Yadav **UID:** 22BCS15444

Branch: CSE **Section:** 22BCS_KRG_IOT_3 A

Semester: 5 **Date of Performance:** 12-Sep-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: - There is a sequence of words in CamelCase as a string of letters s, having the following properties:

- It is a concatenation of one or more words consisting of English letters.
- All letters in the first word are lowercase.
- For each of the subsequent words, the first letter is uppercase and rest of the letters are lowercase. Given, determine the number of words in s.

2. Objective:

The objective of this experiment is to implement a function in Java that counts the number of words in a camel case formatted string.

3. Implementation/Code:

```
class Result {
    public static String pangrams(String s) {
    // Write your code
here    s =
s.toLowerCase();

    // Create a boolean array to track presence of each letter (26 letters in
alphabet)    boolean[] seen = new boolean[26];
```

```
        // Iterate over each character in the string
    for (char c : s.toCharArray()) {            //
    Check if character is a letter            if (c >=
    'a' && c <= 'z') {                // Mark the
    character as seen                seen[c - 'a'] = true;
            }
        }

        // Check if all letters have been
    seen          for (boolean letterSeen : seen)
    {             if (!letterSeen) {
    return "not pangram";
            }
        }

        return "pangram";
    }
}
```

4. Output:

Experiment 5(b)

Student Name: Punit Yadav **UID:** 22BCS15444

Branch: CSE **Section:** 22BCS_KRG_IOT_3 A

Semester: 5 **Date of Performance:** 12-Sep-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: - Louise joined a social networking site to stay in touch with her friends. The signup page required her to input a name and a password. However, the password must be strong. The website considers a password to be strong if it satisfies the following criteria:

- Its length is at least.
- It contains at least one digit.
- It contains at least one lowercase English character.
- It contains at least one uppercase English character.
- It contains at least one special character. The special characters are: !@#$%^&*()-+.

2. Objective:

The objective of this experiment is to implement a function in Java that determines the minimum number of characters needed to make a given password "strong" according to specific criteria.

3. Code:

```
class Result {

    public static int minimumNumber(int n, String password) {
    // Return the minimum number of characters to make the password strong
```

```
int count = 0;
      // Flags to check for each type of character
boolean hasDigit = false;
boolean hasLowerCase = false;
boolean hasUpperCase = false;
boolean hasSpecialChar = false;

      // Special characters set
      String specialCharacters = "!@#$%^&*()-+";
// Check each character in the password        for
(char ch : password.toCharArray()) {           if
(Character.isDigit(ch)) {              hasDigit = true;
        } else if (Character.isLowerCase(ch)) {
hasLowerCase = true;
        } else if (Character.isUpperCase(ch)) {
           hasUpperCase = true;
        } else if (specialCharacters.contains(Character.toString(ch))) {
hasSpecialChar = true;
        }
      }
      // Count missing character types
if (!hasDigit) count++;
if (!hasLowerCase) count++;
if (!hasUpperCase) count++;
if (!hasSpecialChar) count++;
int additionalLength = Math.max(6 - n, 0);
additional length needed
```

```
        return Math.max(count, additionalLength);
    }
```

4. Output:

5. Learning Outcome

1. Learn to efficiently process strings and manipulate individual characters
2. Learn to work with strings in Java, including iterating over characters, checking character properties, and applying conditions based on those properties.
3. Understand basic password strength requirements, which is essential for developing secure applications.
4. Develop skills in analyzing strings and manipulating characters in Java, which is critical in many programming tasks.

Experiment 7(a)

Student Name: Mukul Dagar

UID: 22BCS15436

Branch: CSE

Section/Group: KRG_IOT_3 A

Semester: 5

Date of Performance: 26-Sep-2024

Subject Name: Advanced Programming

Subject Code: 22CSP-314

1. Title: BFS Short Reach.

2. Aim:

Consider an undirected graph where each edge weighs 6 units. Each of the nodes is labelled consecutively from 1 to n. You will be given a number of queries. For each query, you will be given a list of edges describing an undirected graph. After you create a representation of the graph, you must determine and report the shortest distance to each of the other nodes from a given starting position using the breadthfirst search algorithm (BFS). Return an array of distances from the start node in node number order. If a node is unreachable, return -1 for that node.

3. Objective:

The objective of this experiment is to implement an efficient method for calculating the shortest paths in an undirected graph using BreadthFirst Search (BFS). This involves representing the graph with an adjacency list, executing BFS from a given start node, and returning the shortest distances to all other nodes, with -1 indicating unreachable nodes.

4. Algorithm:

Step1: Graph Representation:

Create an adjacency list to represent the undirected graph from the given list of edges.

Step 2: Initialize BFS:

Create a distance array of size n + 1 (where n is the number of nodes) initialized to -1. Set distance[startNode] to 0 to indicate the start node.
Create a queue and enqueue the startNode.

Step 3: BFS Traversal:

While the queue is not empty:

- Dequeue the front node currentNode.
- For each neighbor neighbor of currentNode:
  - If distance[neighbor] is -1 (indicating it has not been visited):
  - Update distance[neighbor] to distance[currentNode] + 6 (distance from the start node via the edge).
  - Enqueue neighbor.

Step 3: Return Distances:

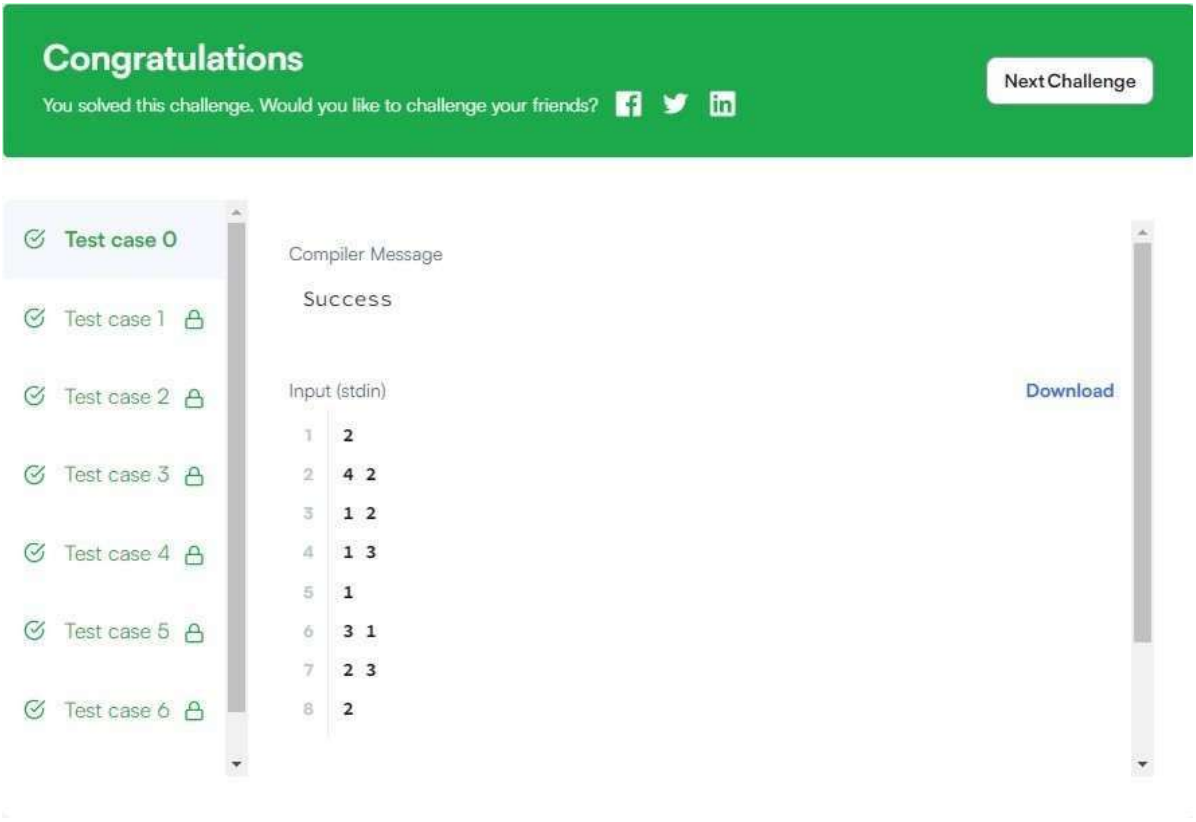
After BFS completes, return the distance array, which contains the shortest distances from the start node to each node.

5. Code:

```
#include <bits/stdc++.h>
using namespace std;

// Trim functions to remove leading/trailing spaces
string ltrim(const string &);
string rtrim(const string &);
vector<string> split(const string &);

// BFS function to calculate shortest paths
vector<int> bfs(int n, int m, vector<vector<int>> edges, int s) {
    // Create an adjacency list for the graph
    vector<vector<int>> adj(n + 1); // Nodes are 1-indexed
    for (const auto &edge : edges) {
        int u = edge[0], v = edge[1];
        adj[u].push_back(v);
        adj[v].push_back(u); // As the graph is undirected
    }

    // Distance vector to store the shortest distance from the start node
    vector<int> distance(n + 1, -1);
    queue<int> q;
    distance[s] = 0;
    q.push(s);

    // BFS to traverse the graph
    while (!q.empty()) {
        int node = q.front();
        q.pop();

        for (int neighbor : adj[node]) {
```

```
            if (distance[neighbor] == -1) { // If not visited
                distance[neighbor] = distance[node] + 6; // Unweighted graph: edge weight is6
                q.push(neighbor);
            }
        }
    }

    // Prepare the result excluding the start node
    vector<int> result;
    for (int i = 1; i <= n; i++) {
        if (i != s) {
            result.push_back(distance[i]);
        }
    }

    return result;
}

int main() {
    ofstream fout(getenv("OUTPUT_PATH"));

    string q_temp;
    getline(cin, q_temp);
    int q = stoi(ltrim(rtrim(q_temp))); // Read the number of test cases

    // Process each test case
    for (int q_itr = 0; q_itr < q; q_itr++) {
        string first_multiple_input_temp;
        getline(cin, first_multiple_input_temp);
        vector<string> first_multiple_input = split(rtrim(first_multiple_input_temp));

        int n = stoi(first_multiple_input[0]); // Number of nodes
        int m = stoi(first_multiple_input[1]); // Number of edges

        // Read the edges
        vector<vector<int>> edges(m);
        for (int i = 0; i < m; i++) {
```

6. Output:



7. Learning outcomes:

1. Graph Representation: Understand how to represent graphs using adjacency lists for efficient traversal.
2. BFS Algorithm: Learn to implement Breadth-First Search to compute shortest paths in unweighted graphs.
3. Queue Utilization: Use queues to explore nodes level by level in BFS.
4. Edge Case Handling: Handle unreachable nodes by marking them with a default value (e.g., -1).

Time Complexity: O(n+m)
Space Complexity: O(n+m)

Experiment 7(b)

Student Name: Mukul Dagar

UID: 22BCS15436

Branch: CSE

Section/Group: KRG_IOT_3 A

Semester: 5

Date of Performance: 26-Sep-2024

Subject Name: Advanced Programming

Subject Code: 22CSP-314

1. **Title**: Quickest Way Up.

2. **Aim**:

Markov takes out his Snakes and Ladders game, stares at the board and wonders: "If I can always roll the die to whatever number I want, what would be the least number of rolls to reach the destination?"

Rules The game is played with a cubic die of 6 faces numbered 1 to 6.

- Starting from square 1, land on square 100 with the exact roll of the die. If moving the number rolled would place the player beyond square 100, no move is made.
- If a player lands at the base of a ladder, the player must climb the ladder. Ladders go up only.
- If a player lands at the mouth of a snake, the player must go down the snake and come out through the tail. Snakes go down only.

3. **Objective**:

The objective is to determine the minimum number of dice rolls needed to reach square 100, accounting for ladders that advance the player and snakes that send the player backward, while ensuring the player lands exactly on square 100.

4. **Algorithm**:

1. Graph Representation: Treat the board as a graph with 100 nodes (each square representing a node). Each node is connected to the next 6 nodes (dice rolls), with additional edges for ladders and snakes.
2. Initialize: Create an array moves [] where each index represents a square. For ladders and snakes, update moves[i] to point to the destination square (either the top of the ladder or the tail of the snake).
3. BFS Setup: Use Breadth-First Search (BFS) to explore the shortest path. Initialize a queue and start from square 1. Maintain a distance array dist[] to track the minimum number of dice rolls to reach each square.

4. BFS Execution:
   - For each square, check all possible dice rolls (1 to 6).
   - For each roll, move the player to the corresponding square and check for ladders or snakes to adjust the position.
   - If the new square hasn't been visited, update its distance and enqueue it.
5. Terminate: The BFS terminates when you reach square 100, and the result is the minimum number of rolls stored in the distance array.

5. Code:

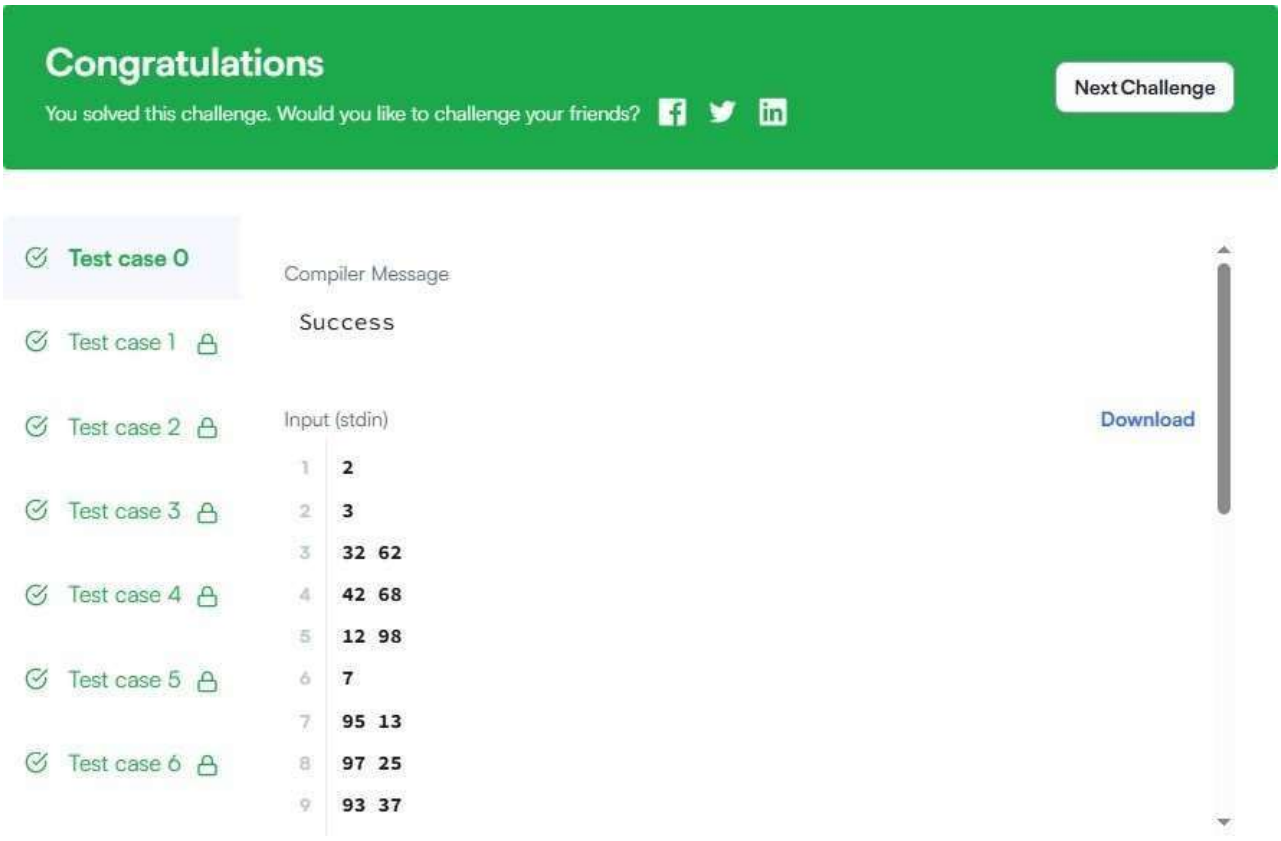
```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

const int N = 104;
const int INF = 100000000;

int main()
{
   int t;
   cin >> t;
   for (int k = 1; k <= t; ++k)
   {
      vector<int> graph(N, 0); // Graph stores ladders and snakes
      vector<bool> mark(N, false);  // Mark to track visited positions
      int n, m;

      // Input ladders
      cin >> n;
      for (int i = 0; i < n; ++i)
      {
         int a, b;
         cin >> a >> b;
         graph[a] = b; // Ladder from a to b
      }

      // Input snakes
      cin >> m;
      for (int i = 0; i < m; ++i)
      {
         int a, b;
```

```
    cin >> a >> b;
    graph[a] = b; // Snake from a to b
}

queue<pair<int, int>> q;  // Pair of (current position, number of moves)
int ans = INF;

// Start from position 1 with 0 moves
q.push(make_pair(1, 0));
mark[1] = true;

while (!q.empty())
{
    pair<int, int> p = q.front();
    q.pop();

    if (p.first == 100)  // Reached position 100
    {
        ans = p.second;
        break;
    }

    // Check for all possible moves (1 to 6)
    for (int i = 1; i <= 6; ++i)
    {
        int x = p.first + i;
        if (x > 100)
            continue;

        if (!mark[x])
        {
            mark[x] = true;

            // If there's no ladder or snake, move to next position
            if (graph[x] == 0)
                q.push(make_pair(x, p.second + 1));
            else
            {
                // Move to the end of the ladder/snake
                x = graph[x];
                mark[x]  = true;
                q.push(make_pair(x, p.second + 1));
```

Discover. Learn. Empower.

```
                }
            }
        }
    }

    // Output the result
    if (ans == INF)
        cout << -1 << endl; // No possible way to reach 100
    else
        cout << ans << endl;  // Minimum moves to reach 100
  }
}
```

6. Output:



7. Learning outcomes:

1. Graph Representation: Learn to model a game board as a graph, where each node represents a square and edges represent possible dice rolls and game mechanics like ladders and snakes.

2. BFS Algorithm: Understand how Breadth-First Search (BFS) can be used to find the shortest path in an unweighted graph, which is essential for solving problems involving minimum steps or moves.

3. Handling Game Mechanics: Gain experience in incorporating specific game rules (e.g., ladders and snakes) into the graph traversal to adapt general algorithms to problem-specific constraints.

Time Complexity: O(1)
Space Complexity: O(1)

Experiment 8(a)

Student Name: Mukul Dagar **UID:** 22BCS15436

Branch: CSE **Section/Group:** KRG_IOT_3 A

Semester: 5 **Date of Performance:** 03-Oct-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. **Aim:**

   Problem Statement:

   Alice is a kindergarten teacher. She wants to give some candies to the children in her class. All the children sit in a line and each of them has a rating score according to his or her performance in the class. Alice wants to give at least 1 candy to each child. If two children sit next to each other, then the one with the higher rating must get more candies. Alice wants to minimize the total number of candies she must buy.

2. **Objective:**

   The objective is to determine the minimum number of candies Alice needs to distribute to children seated in a line based on their rating scores. Each child must receive at least one candy, and children with higher ratings than their adjacent neighbours must receive more candies than those neighbours.

3. **Implementation/Code:**

```
#include <iostream>
#include <vector>
#include <numeric>
#include <iomanip>

using namespace std;

class Solution {
public:
    static long long candies(int n, vector<int>& arr) {
        if (n == 0) return 0; // Edge case for empty array

        vector<int> cache(n, 1); // Initialize cache with 1 for each child

        // First pass: left to right
        for (int i = 1; i < n; i++) {
```

```
            if (arr[i] > arr[i - 1]) {
                cache[i] = cache[i - 1] + 1; // Give more candies than the left child
            }
        }

        // Second pass: right to left
        for (int i = n - 2; i >= 0; i--) {
            if (arr[i] > arr[i + 1]) {
                // Ensure the current child has more candies than the right child
                if (cache[i] <= cache[i + 1]) {
                    cache[i] = cache[i + 1] + 1;
                }
            }
        }

        // Calculate the total number of candies
        long long sum = accumulate(cache.begin(), cache.end(), 0LL); // Use accumulate to
sum up

        return sum;
    }
};

int main() {
    // Example usage
    vector<int> ratings = {1, 0, 2};
    long long totalCandies = Solution::candies(ratings.size(), ratings);
    cout << "Total candies required: " << totalCandies << endl;
    return 0;
}
```

4. Output:


Test case 0
Test case 1
Test case 2
Test case 3
Test case 4
Test case 5
Test case 6
Compiler Message
Success
Input (stdin)
Download
3
1
2
2
Expected Output
Download
4

Experiment 8(b)

Student Name: Mukul Dagar **UID:** 22BCS15436

Branch: CSE **Section/Group:** KRG_IOT_3 A

Semester: 5 **Date of Performance:** 03-Oct-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement:

Marc loves cupcakes, but he also likes to stay fit. Each cupcake has a calorie count, and Marc can walk a distance to expend those calories. If Marc has eaten j cupcakes so far, after eating a cupcake with c calories he must walk at least 2j *cmiles to maintain his weight.

2. Objective:

The objective is to calculate the minimum total distance Marc needs to walk to offset the calories from the cupcakes he has eaten. Given that Marc must walk an increasing distance proportional to the number of cupcakes consumed, the goal is to determine the total walking distance required to balance out the calorie intake from all cupcakes.

3. Code:

```
class Result {

   public static long marcsCakewalk(List<Integer> calorie) {
   // Write your code here
    Collections.sort(calorie, Collections.reverseOrder());

      long totalCandies = 0;        for (int i = 0; i < calorie.size(); i++) {
totalCandies += (1L << i) * calorie.get(i);
      }

      return totalCandies;
```

```
    }

}
```

4. Output:

5. Learning Outcome

- Learn to solve optimization problems where constraints involve comparative values between adjacent elements.
- Understand how to implement a two-pass algorithm to achieve the optimal solution in such scenarios.
- learn how to apply cumulative cost calculations in scenarios involving variable constraints.
- They will understand how to optimize and calculate required values based on a progressive increase in factors, such as the distance Marc must walk per cupcake.
- This exercise will enhance problem-solving skills in handling scenarios with incremental constraints and applying mathematical calculations effectively.

Experiment 6 (a)

Student Name: Mukul Dagar **UID:** 22BCS15436

Branch: CSE **Section/Group:** KRG_IOT_3 A

Semester: 5 **Date of Performance:** 19-Sep-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: - You are given a tree (a simple connected graph with no cycles). Find the maximum number of edges you can remove from the tree to get a forest such that each connected component of the forest contains an even number of nodes.

2. Objective:

The objective of this experiment is to determine how many ways you can remove an edge from a tree to create two separate subtrees such that both resulting subtrees have an even number of nodes.

3. Implementation/Code:

```
#include <bits/stdc++.h>
using namespace std;

// Function to compute the number of removable edges
int evenForest(int t_nodes, int t_edges, vector<int>& t_from, vector<int>& t_to) {
    // Create an adjacency list
    unordered_map<int, list<int>> adjacencyList;
    for (int i = 1; i <= t_nodes; i++) {
        adjacencyList[i] = list<int>();
    }
```

```
// Build the adjacency list based on the edges
for (int i = 0; i < t_edges; i++) {
    int u = t_from[i];
    int v = t_to[i];
    adjacencyList[u].push_back(v);
    adjacencyList[v].push_back(u);
}

// Array to store the size of each subtree
vector<int> subtreeSize(t_nodes + 1, 0);
vector<bool> visited(t_nodes + 1, false);

// Start DFS from node 1 (root)
auto dfs = [&](int node, auto& dfs_ref) -> void {
    visited[node] = true;
    subtreeSize[node] = 1;

    for (int neighbor : adjacencyList[node]) {
        if (!visited[neighbor]) {
            dfs_ref(neighbor, dfs_ref);
            subtreeSize[node] += subtreeSize[neighbor];
        }
    }
};

dfs(1, dfs);
// Count the number of edges that can be removed
int removableEdges = 0;
```

```
    for (int i = 2; i <= t_nodes; i++) {
        if (subtreeSize[i] % 2 == 0) {
            removableEdges++;
        }
    }
    return removableEdges;
}
int main() {
    // Example test case
    vector<int> t_from = {1, 1, 3, 3, 4, 6, 6, 8, 8};
    vector<int> t_to = {3, 2, 6, 4, 5, 8, 7, 9, 10};
    cout << evenForest(10, 9, t_from, t_to) << endl;
    return 0;
}
```

4. Output:

Experiment 6 (b)

Student Name: Mukul Dagar **UID:** 22BCS15436

Branch: CSE **Section/Group:** KRG_IOT_3 A

Semester: 5 **Date of Performance:** 19-Sep-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: - You are given a pointer to the root of a binary search tree and values to be inserted into the tree. Insert the values into their appropriate position in the binary search tree and return the root of the updated binary tree. You just have to complete the function.

2. Objective:

The objective of this experiment is to implement a function in Java that determines the minimum number of characters needed to make a given password "strong" according to specific criteria.

3. Code:

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

// Definition of the Node structure
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr), right(nullptr) {}
```

```
};

// Function to insert a node in the BST
Node* insert(Node* root, int data) {
    // If the tree is empty, create a new node and return it
    if (root == nullptr) {
        return new Node(data);
    }

    // Otherwise, recur down the tree and insert the node
    if (data <= root->data) {
        root->left = insert(root->left, data);
    } else {
        root->right = insert(root->right, data);
    }

    // Return the (unchanged) node pointer
    return root;
}

// Helper function to perform in-order traversal (for testing)
void inorder(Node* root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}
```

```
int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    // Print the in-order traversal of the tree
    inorder(root);  // Output: 20 30 40 50 60 70 80

    return 0;
}
```

4. Output:

5. Learning Outcome

i. Enhance problem-solving skills by breaking down a complex problem into smaller, more manageable components (like node insertion and traversal)

ii. Apply concepts of graph theory to solve problems involving trees and subtrees.

iii. Develop algorithms to process and analyze tree structures, specifically focusing on subtree properties and even node counts.

iv. Gain experience in designing recursive functions for common tree operations such as insertion and traversal.

v. Learn the properties of BSTs and how to maintain these properties during insertion operations.

Experiment 5(a)

Student Name: Mukul Dagar **UID:** 22BCS15436

Branch: CSE **Section:** 22BCS_KRG_IOT_3 A

Semester: 5 **Date of Performance:** 12-Sep-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: - There is a sequence of words in CamelCase as a string of letters s, having the following properties:

- It is a concatenation of one or more words consisting of English letters.
- All letters in the first word are lowercase.
- For each of the subsequent words, the first letter is uppercase and rest of the letters are lowercase. Given, determine the number of words in s.

2. Objective:

The objective of this experiment is to implement a function in Java that counts the number of words in a camel case formatted string.

3. Implementation/Code:

```
class Result {
    public static String pangrams(String s) {
    // Write your code
here    s =
s.toLowerCase();

    // Create a boolean array to track presence of each letter (26 letters in
alphabet)    boolean[] seen = new boolean[26];
```

```
        // Iterate over each character in the string
    for (char c : s.toCharArray()) {            //
    Check if character is a letter            if (c >=
    'a' && c <= 'z') {                // Mark the
    character as seen                seen[c - 'a'] = true;
            }
        }

        // Check if all letters have been
    seen          for (boolean letterSeen : seen)
    {             if (!letterSeen) {
    return "not pangram";
            }
        }

        return "pangram";
    }
}
```

4. Output:

Experiment 5(b)

Student Name: Mukul Dagar **UID:** 22BCS15436

Branch: CSE **Section:** 22BCS_KRG_IOT_3 A

Semester: 5 **Date of Performance:** 12-Sep-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: - Louise joined a social networking site to stay in touch with her friends. The signup page required her to input a name and a password. However, the password must be strong. The website considers a password to be strong if it satisfies the following criteria:

- Its length is at least.
- It contains at least one digit.
- It contains at least one lowercase English character.
- It contains at least one uppercase English character.
- It contains at least one special character. The special characters are: !@#$%^&*()-+.

2. Objective:

The objective of this experiment is to implement a function in Java that determines the minimum number of characters needed to make a given password "strong" according to specific criteria.

3. Code:

```
class Result {

    public static int minimumNumber(int n, String password) {
    // Return the minimum number of characters to make the password strong
```

```
int count = 0;
      // Flags to check for each type of character
boolean hasDigit = false;
boolean hasLowerCase = false;
boolean hasUpperCase = false;
boolean hasSpecialChar = false;

      // Special characters set
      String specialCharacters = "!@#$%^&*()-+";
// Check each character in the password        for
(char ch : password.toCharArray()) {           if
(Character.isDigit(ch)) {              hasDigit = true;
        } else if (Character.isLowerCase(ch)) {
hasLowerCase = true;
        } else if (Character.isUpperCase(ch)) {
           hasUpperCase = true;
        } else if (specialCharacters.contains(Character.toString(ch))) {
hasSpecialChar = true;
        }
      }
      // Count missing character types
if (!hasDigit) count++;
if (!hasLowerCase) count++;
if (!hasUpperCase) count++;
if (!hasSpecialChar) count++;
int additionalLength = Math.max(6 - n, 0);
additional length needed
```

```
        return Math.max(count, additionalLength);
    }
```

4. Output:

5. Learning Outcome

1. Learn to efficiently process strings and manipulate individual characters
2. Learn to work with strings in Java, including iterating over characters, checking character properties, and applying conditions based on those properties.
3. Understand basic password strength requirements, which is essential for developing secure applications.
4. Develop skills in analyzing strings and manipulating characters in Java, which is critical in many programming tasks.

Experiment 9(a)

Student Name: Mukul Dagar **UID:** 22BCS15436

Branch: CSE **Section:** 22BCS_KRG_IOT_3 A

Semester: 5 **Date of Performance:** 17-Oct-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: Given a partially filled 9 x 9 sudoku board, you need to output a solution to it. Input-The first line contains N which is the number of Sudoku puzzles. N Sudoku puzzles follow. Each one contains 9 lines with each line containing 9 space separated integers. A 0 indicates an unfilled square, and an integer between 1 and 9 denotes a filled square with that value. Output-Output to STDOUT the completely filled Sudoku board for each Sudoku. If there are multiple solutions, you can output any of the one.

2. Objective:

The objective is to solve and output the completed solution of N given 9x9 partially filled Sudoku puzzles, filling in the unfilled squares while ensuring all Sudoku rules are followed.

3. Implementation/Code:

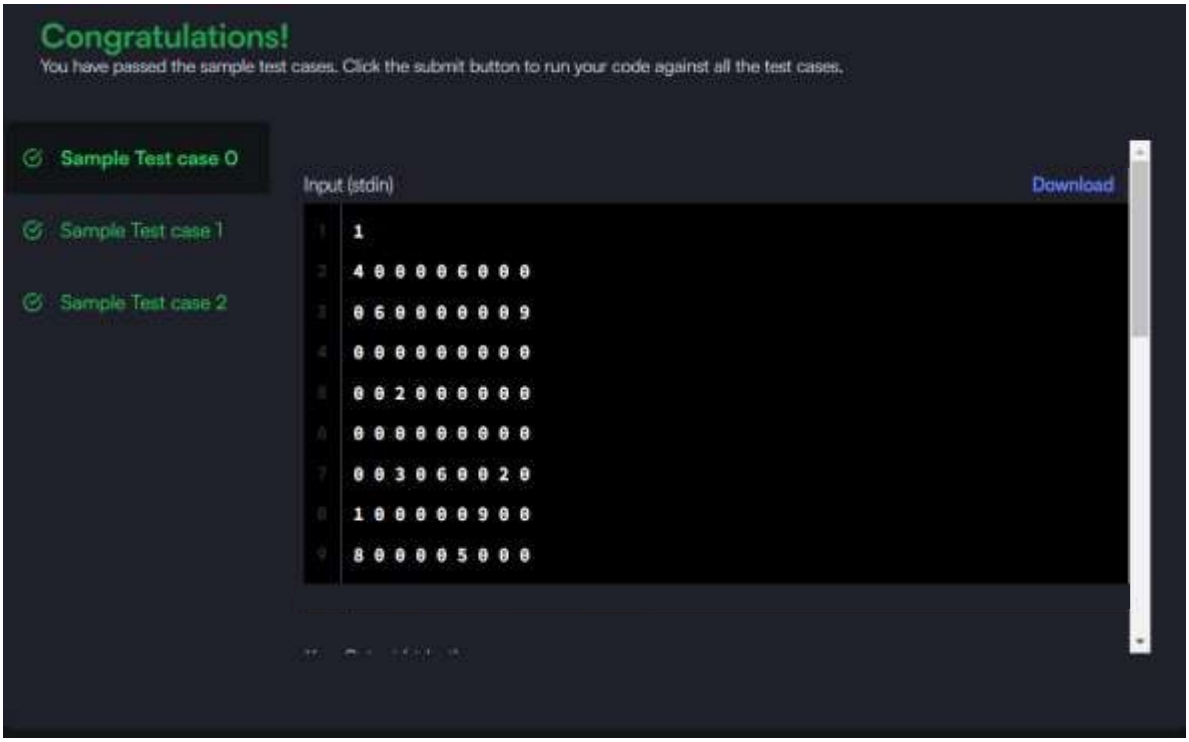
```
#include<iostream>

#include<vector>

#include<set>

#include<math.h>


using namespace std;
```

```
int getSector(int row, int col) {
    return (row/3)*3+(col/3);
}

bool valid(int grid[9][9], int row, int col) {
    for (int i = 0; i < 9; i++) {
        if (grid[row][i] == grid[row][col] && i != col) return false;
        if (grid[i][col] == grid[row][col] && i != row) return false;
    }
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (getSector(row, col) == getSector(i, j) && row != i && col != j && grid[i][j] ==
grid[row][col]) return false;
        }
    }

    return true;
}

/* Head ends here */
void sudoku_solve(int grid[9][9],set<int> set) {
    //your logic here
    int forward = 1;
    for (int idx = 0; idx < 81;) {
        int row = floor(idx/9);
```

```
            int col = fmod(idx,9);
            if (set.count(idx)>0) {
              idx += 1*forward;
            } else {
               grid[row][col]++;
               if (grid[row][col] == 10) {
                  grid[row][col] = 0;
                  forward = -1;
                  idx--;
               }
               if (valid(grid, row, col)) {
                  forward = 1;
                  idx++;
               }
            }
         }
         for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
               cout << grid[i][j] << " ";
            }
            cout << endl;
         }
      }
```

```
/* Tail starts here */
int main() {
    int n, board[9][9];
    set<int> set;
    cin >> n;
    for(int i=0;i<n;i++) {
        for(int j=0;j<9;j++) {
            for(int k=0;k<9;k++) {
                board[j][k] = 0;
                cin >> board[j][k];
                if ( board[j][k] != 0 ) set.insert(j*9+k);
            }
        }
        sudoku_solve(board,set);
    }
    return 0;
}
```

4. Output:


Congratulations!
You have passed the sample test cases. Click the submit button to run your code against all the test cases.
Sample Test case 0
Sample Test case 1
Sample Test case 2
Input (stdin)
Download
1
4 0 0 0 0 6 0 0 0
0 6 0 0 0 0 0 0 9
0 0 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 3 0 6 0 0 2 0
1 0 0 0 0 0 9 0 0
8 0 0 0 0 5 0 0 0

Experiment 9(b)

Student Name: Mukul Dagar **UID:** 22BCS15436

Branch: CSE **Section:** 22BCS_KRG_IOT_3 A

Semester: 5 **Date of Performance:** 17-Oct-2024

Subject Name: Advanced Programming **Subject Code:** 22CSP-314

1. Aim:

Problem Statement: A 10*10 Crossword grid is provided to you, along with a set of words (or names of places) which need to be filled into the grid. Cells are marked either + or -. Cells marked with a - are to be filled with the word list.

2. Objective:

To fill a 10x10 crossword grid, where cells marked with '-' can be filled with a set of provided words (or names of places), ensuring the words fit correctly either horizontally or vertically.

3. Code:

```
#include<iostream>
#include<vector>
#include<string>
using namespace std;
// Function to check if a word can be placed horizontally
bool canPlaceHorizontally(vector<string>& grid, string word, int row, int col) {
   if (col + word.size() > 10)
      return false;
   for (int i = 0; i < word.size(); i++) {
      if (grid[row][col + i] != '-' && grid[row][col + i] != word[i])
         return false;
   }
```

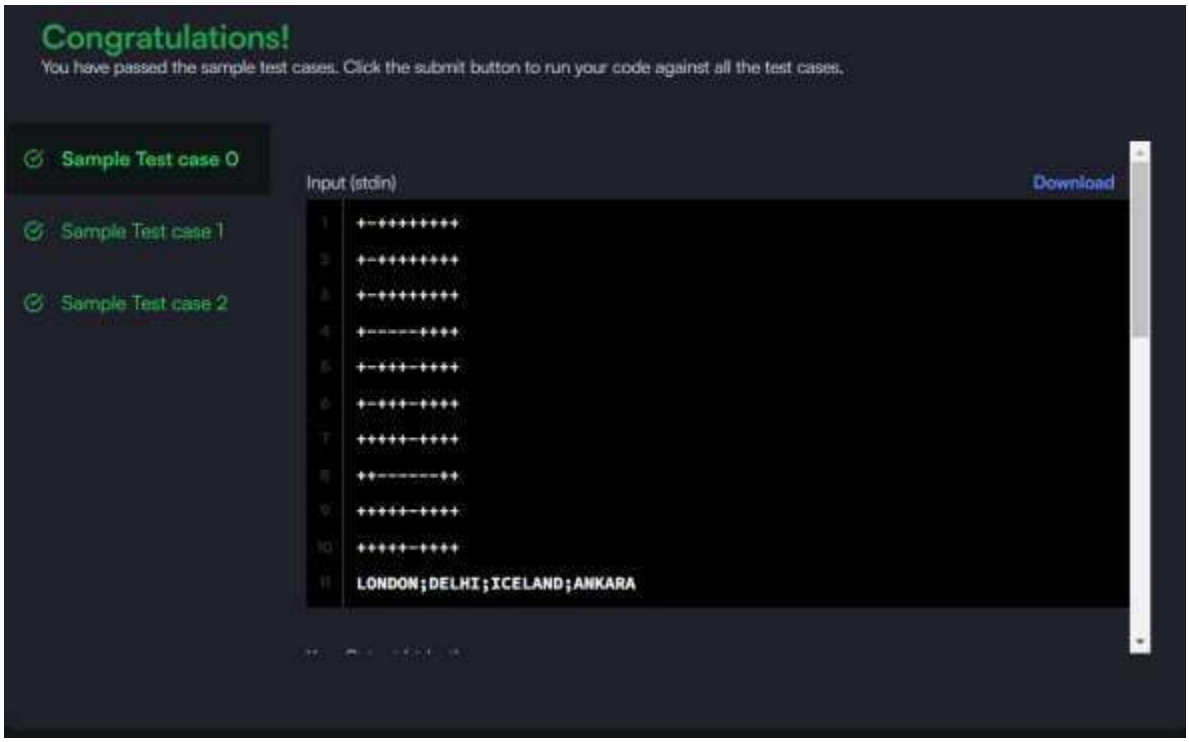
```
    return true;
}

// Function to place a word horizontally in the grid
void placeHorizontally(vector<string>& grid, string word, int row, int col,
vector<bool>& placed) {
    for (int i = 0; i < word.size(); i++) {
        if (grid[row][col + i] == '-') {
            grid[row][col + i] = word[i];
            placed[i] = true;
        }
    }
}

// Function to remove a horizontally placed word
void unplaceHorizontally(vector<string>& grid, string word, int row, int col,
vector<bool>& placed) {
    for (int i = 0; i < word.size(); i++) {
        if (placed[i])
            grid[row][col + i] = '-';
    }
}

// Function to check if a word can be placed vertically
bool canPlaceVertically(vector<string>& grid, string word, int row, int col) {
    if (row + word.size() > 10)
        return false;
    for (int i = 0; i < word.size(); i++) {
        if (grid[row + i][col] != '-' && grid[row + i][col] != word[i])
            return false;
    }
```

```
    return true;
}

// Function to place a word vertically in the grid
void placeVertically(vector<string>& grid, string word, int row, int col, vector<bool>&
placed) {
    for (int i = 0; i < word.size(); i++) {
        if (grid[row + i][col] == '-') {
            grid[row + i][col] = word[i];
            placed[i] = true;
        }
    }
}

// Function to remove a vertically placed word
void unplaceVertically(vector<string>& grid, string word, int row, int col, vector<bool>&
placed) {
    for (int i = 0; i < word.size(); i++) {
        if (placed[i])
            grid[row + i][col] = '-';
    }
}

// Backtracking function to solve the crossword puzzle
bool solveCrossword(vector<string>& grid, vector<string>& words, int index) {
    if (index == words.size())
        return true;

    string word = words[index];

    for (int i = 0; i < 10; i++) {
```

```
            for (int j = 0; j < 10; j++) {
                // Try placing the word horizontally
                if (canPlaceHorizontally(grid, word, i, j)) {
                    vector<bool> placed(word.size(), false);
                    placeHorizontally(grid, word, i, j, placed);

                    if (solveCrossword(grid, words, index + 1))
                        return true;

                    unplaceHorizontally(grid, word, i, j, placed);
                }

                // Try placing the word vertically
                if (canPlaceVertically(grid, word, i, j)) {
                    vector<bool> placed(word.size(), false);
                    placeVertically(grid, word, i, j, placed);

                    if (solveCrossword(grid, words, index + 1))
                        return true;

                    unplaceVertically(grid, word, i, j, placed);
                }
            }
        }

        return false;
    }
    int main() {
        vector<string> grid(10);

        for (int i = 0; i < 10; i++) {
```

```
        cin >> grid[i];
    }

    string words_input;
    cin >> words_input;

    vector<string> words;
    string word = "";
    for (char c : words_input) {
        if (c == ';') {
            words.push_back(word);
            word = "";
        } else {
            word += c;
        }
    }
    if (!word.empty()) words.push_back(word);

    solveCrossword(grid, words, 0);

    // Output the filled grid
    for (int i = 0; i < 10; i++) {
        cout << grid[i] << endl;
    }

    return 0;
}
```

4. Output:



5. Learning Outcome

1. Gained knowledge of how backtracking algorithms can be used to solve constraint satisfaction problems, like placing words in a crossword puzzle.
2. Developed the ability to break down a complex problem into smaller subproblems and solve them using recursive functions.
3. Acquired skills in working with 2D grids or matrices by navigating through the grid and managing row/column placements effectively.