CODE:

```
#include <iostream>
#include <cmath>

bool isPrime

(int num) {
    if (num <= 1) {
        return false;
    }

    int sqrtNum = sqrt(num);
    for (int i = 2; i <= sqrtNum; ++i) {
        if (num % i == 0) {
            return false;
        }
    }

    return true;
}

int main() {
    std::cout << "Prime numbers between 1 and 100:" << std::endl;
    for (int i = 1; i <= 100; ++i) {
        if (isPrime(i)) {
            std::cout << i << " ";
        }
    }
    std::cout << std::endl;

    int num;
    std::cout << "Enter a number to check if it's prime: ";
    std::cin >> num;

    if (isPrime(num)) {
        std::cout << num << " is a prime number." << std::endl;
    } else {
        std::cout << num << " is not a prime number." << std::endl;
    }

    return 0;
}
```

OUTCOMES:

Enter a number: 5
5 is a prime number.

Prime numbers between 1 and 100 are:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

CODE :

```
#include <stdio.h>
#include <string.h>

// Function to reverse a string
void reverseString(char str[]) {
   int length = strlen(str);
   int start = 0;
   int end = length - 1;

   while (start < end) {
      char temp = str[start];
      str[start] = str[end];
      str[end] = temp;

      start++;
      end--;
   }
}

// Function to reverse an array
void reverseArray(int arr[], int size) {
   int start = 0;
   int end = size - 1;

   while (start < end) {
      int temp = arr[start];
      arr[start] = arr[end];
      arr[end] = temp;

      start++;
      end--;
   }
}

// Function to concatenate two strings
void concatenateStrings(char str1[], char str2[]) {
   strcat(str1, str2);
}
```

```
int main() {
    // Reverse a string
    char string[] = "Hello, World!";
    printf("Original string: %s\n", string);
    reverseString(string);
    printf("Reversed string: %s\n", string);

    // Reverse an array
    int array[] = {1, 2, 3, 4, 5};
    int size = sizeof(array) / sizeof(array[0]);
    printf("\nOriginal array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    reverseArray(array, size);
    printf("\nReversed array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }

    // Concatenate two strings
    char str1[100] = "Hello";
    char str2[] = " World!";
    printf("\n\nString 1: %s\n", str1);
    printf("String 2: %s\n", str2);
    concatenateStrings(str1, str2);
    printf("Concatenated string: %s\n", str1);

    return 0;
}
```

OUTCOMES:

```
Original string: Hello, World!
Reversed string: !dlroW ,olleH

Original array: 1 2 3 4 5
Reversed array: 5 4 3 2 1

String 1: Hello
String 2:  World!
Concatenated string: Hello World!
```

CODE:

```
 #include <studio.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to perform heapify on a subtree rooted at index i
void heapify(int arr[], int n, int i) {
    int largest = i;       // Initialize largest as root
    int left = 2 * i + 1;  // Left child
    int right = 2 * i + 2; // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than current largest
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        swap(&arr[i], &arr[largest]);

        // Recursively heapify the affected subtree
        heapify(arr, n, largest);
    }
}

// Heap Sort function
void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from the heap one by one
```

```
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to the end
        swap(&arr[0], &arr[i]);

        // Heapify the reduced heap
        heapify(arr, i, 0);
    }
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);
    heapSort(arr, n);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;
}
```

OUTCOMES:

Original array: 12 11 13 5 6 7

Sorted array: 5 6 7 11 12 13

CODE ;

```
: #include <stdio.h>

#define N 8

void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%2d ", board[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

int isSafe(int board[N][N], int row, int col) {
    int i, j;

    // Check if there is a queen in the same row
    for (i = 0; i < col; i++) {
        if (board[row][i])
            return 0;
    }

    // Check if there is a queen in the upper left diagonal
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {

        if (board[i][j])
            return 0;
    }

    // Check if there is a queen in the lower left diagonal
    for (i = row, j = col; j >= 0 && i < N; i++, j--) {
        if (board[i][j])
            return 0;
    }

    return 1;
}

int solveNQueenUtil(int board[N][N], int col) {
    if (col >= N) {
        printSolution(board);
        return 1;
    }

    int res = 0;
```

```
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            res += solveNQueenUtil(board, col + 1);
            board[i][col] = 0;
        }
    }

    return res;
}

void solveNQueen(int n) {
    int board[N][N] = {0};
    int count = solveNQueenUtil(board, 0);

    printf("Total solutions for %d-Queen: %d\n", n, count);
}

int main() {
    printf("Solutions for 4-Queen Problem:\n");
    solveNQueen(4);

    printf("Solutions for 8-Queen Problem:\n");
    solveNQueen(8);

    return 0;
}
```

OUTCOME:

```
0 0 1 0 0 0 0 0            | 0 1 0 0
0 0 0 0 0 1 0 0            | 0 0 0 1   ← 4 Queen
0 0 0 1 0 0 0 0            | 1 0 0 0
0 1 0 0 0 0 0 0            | 0 0 1 0
0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0   ← 8 Queen
1 0 0 0 0 0 0 0
```

CODE:

```
#include <stdio.h>

// Function to calculate the GCD of two numbers
int gcd(int a, int b) {
    // Base case: If b is 0, the GCD is a
    if (b == 0) {
        return a;
    }

    // Recursive case: Calculate GCD using Euclidean algorithm
    return gcd(b, a % b);
}

int main() {
    int num1, num2;

    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);

    // Calculate the GCD
    int result = gcd(num1, num2);

    printf("The GCD of %d and %d is: %d\n", num1, num2, result);

    return 0;
}
```

OUTCOME :

Enter two numbers: 8 10

The GCD of 8 and 10 is: 2

CODE :

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

using namespace std;

// Graph class
class Graph {
  int numVertices; // Number of vertices

  // Adjacency list representation
  vector<vector<int>> adjList;

public:
  // Constructor
  Graph(int vertices) {
    numVertices = vertices;
    adjList.resize(numVertices);
  }

  // Add edge to the graph
  void addEdge(int source, int destination) {
    adjList[source].push_back(destination);
  }

  // Depth-First Search traversal
  void DFS(int startVertex) {
    // Visited array to keep track of visited vertices
    vector<bool> visited(numVertices, false);

    // Create a stack for DFS
    stack<int> stack;

    // Push the start vertex into the stack
    stack.push(startVertex);

    // Run DFS until the stack is empty
    while (!stack.empty()) {
      // Pop a vertex from the stack
      int currentVertex = stack.top();
      stack.pop();

      // Process the current vertex if it hasn't been visited
      if (!visited[currentVertex]) {
        cout << currentVertex << " ";
        visited[currentVertex] = true;
```

```
            }

            // Get all adjacent vertices of the current vertex
            vector<int> neighbors = adjList[currentVertex];

            // Push unvisited neighbors into the stack
            for (int neighbor : neighbors) {
                if (!visited[neighbor]) {
                    stack.push(neighbor);
                }
            }
        }
    }
};

// Main function
int main() {
    // Create a graph
    Graph graph(6);

    // Add edges
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(2, 5);

    // Perform DFS traversal starting from vertex 0
    cout << "DFS traversal starting from vertex 0: ";
    graph.DFS(0);

    return 0;
}
```

OUTCOME :

DFS traversal starting from vertex 0: 0 2 5 4 1 3

CODE:

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

using namespace std;

// Graph class
class Graph {
   int numVertices; // Number of vertices

   // Adjacency list representation
   vector<vector<int>> adjList;

public:
   // Constructor
   Graph(int vertices) {
      numVertices = vertices;
      adjList.resize(numVertices);
   }

   // Add edge to the graph
   void addEdge(int source, int destination) {
      adjList[source].push_back(destination);
   }

   // Breadth-First Search traversal
   void BFS(int startVertex) {
      // Visited array to keep track of visited vertices
      vector<bool> visited(numVertices, false);

      // Create a queue for BFS
      queue<int> queue;

      // Mark the start vertex as visited and enqueue it
      visited[startVertex] = true;
      queue.push(startVertex);

      // Run BFS until the queue is empty
      while (!queue.empty()) {
         // Dequeue a vertex from the queue
```

```
            int currentVertex = queue.front();
            queue.pop();

            cout << currentVertex << " ";

            // Get all adjacent vertices of the current vertex
            vector<int> neighbors = adjList[currentVertex];

            // Enqueue unvisited neighbors and mark them as visited
            for (int neighbor : neighbors) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.push(neighbor);
                }
            }
        }
    }
};

// Main function
int main() {
    // Create a graph
    Graph graph(6);

    // Add edges
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(2, 5);

    // Perform BFS traversal starting from vertex 0
    cout << "BFS traversal starting from vertex 0: ";
    graph.BFS(0);

    return 0;
}
```

OUTCOME :

BFS traversal starting from vertex 0: 0 1 2 3 4 5

CODE:

```
#include <iostream>
 #include<bits/stdc++.h>
 #include <cstring>
 using namespace std;
 // number of vertices in graph
 #define V 7
 // create a 2d array of size 7x7
 //for adjacency matrix to represent graph
 int main () {
   // create a 2d array of size 7x7
 //for adjacency matrix to represent graph
   int G[V][V] = {
   {0,28,0,0,0,10,0},
 {28,0,16,0,0,0,14},
 {0,16,0,12,0,0,0},
 {0,0,12,22,0,18},
 {0,0,0,22,0,25,24},
 {10,0,0,0,25,0,0},
 {0,14,0,18,24,0,0}
 };
   int edge;            // number of edge
   // create an array to check visited vertex
   int visit[V];
   //initialise the visit array to false
  for(int i=0;i<V;i++){
    visit[i]=false;
}
  // set number of edge to 0
   edge = 0;
   // the number of edges in minimum spanning tree will be
   // always less than (V -1), where V is the number of vertices in
   //graph
   // choose 0th vertex and make it true
   visit[0] = true;
   int x;            //  row number
   int y;            //  col number
   // print for edge and weight
   cout << "Edge" << " : " << "Weight";
   cout << endl;
   while (edge < V - 1) {//in spanning tree consist the V-1 number of edges
```

```
  //For every vertex in the set S, find the all adjacent vertices
  // , calculate the distance from the vertex selected.
  // if the vertex is already visited, discard it otherwise
  //choose another vertex nearest to selected vertex.
    int min = INT_MAX;
    x = 0;
    y = 0;
    for (int i = 0; i < V; i++) {
     if (visit[i]) {
        for (int j = 0; j < V; j++) {
          if (!visit[j] && G[i][j]) { // not in selected and there is an edge
             if (min > G[i][j]) {
                min = G[i][j];
                x = i;
                y = j;
             }
          }
        }
      }
    }
    cout << x <<  " ---> " << y << " :  " << G[x][y];
    cout << endl;
    visit[y] = true;
    edge++;
   }
  return 0;
 }
```

OUTCOME:

Minimum Spanning Tree:
0 – 3
3 - 1
1 - 2
1 – 4
4 – 6
3 - 5

CODE:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

const int INF = 1e9;

// Function to calculate the distance between two points
double calcDistance(pair<int, int> p1, pair<int, int> p2) {
    int dx = p1.first - p2.first;
    int dy = p1.second - p2.second;
    return sqrt(dx * dx + dy * dy);
}

// Function to solve the TSP using a brute-force approach
double tspBruteForce(vector<pair<int, int>>& points, int n, int start, int current, int mask,
vector<vector<double>>& dp) {
    // If all cities have been visited, return the distance from the current city to the starting city
    if (mask == (1 << n) - 1) {
        return calcDistance(points[current], points[start]);
    }

    // If the subproblem has already been solved, return the precalculated value
    if (dp[current][mask] != -1) {
        return dp[current][mask];
    }

    double ans = INF;

    // Try visiting all unvisited cities
    for (int i = 0; i < n; i++) {
        if ((mask & (1 << i)) == 0) {
            int newMask = mask | (1 << i);
            double distance = calcDistance(points[current], points[i]);
            double temp = distance + tspBruteForce(points, n, start, i, newMask, dp);
            ans = min(ans, temp);
        }
    }

    // Store the result in the DP table
    dp[current][mask] = ans;

    return ans;
}
```

```
// Function to solve the TSP using a brute-force approach
double solveTSP(vector<pair<int, int>>& points) {
    int n = points.size();
    vector<vector<double>> dp(n, vector<double>(1 << n, -1));

    // Start from the first city
    int start = 0;

    // Call the recursive TSP function
    double result = tspBruteForce(points, n, start, start, 1 << start, dp);

    return result;
}

int main() {
    int n; // Number of cities
    cout << "Enter the number of cities: ";
    cin >> n;

    vector<pair<int, int>> points(n); // Coordinates of cities

    // Read the coordinates of the cities
    cout << "Enter the coordinates of the cities:" << endl;
    for (int i = 0; i < n; i++) {
        int x, y;
        cin >> x >> y;
        points[i] = make_pair(x, y);
    }

    // Solve the TSP problem
    double distance = solveTSP(points);

    // Display the result
    cout << "Shortest distance for TSP: " << distance << endl;

    return 0;
}
```

OUTPUT:

Minimum Distance: 80
Path: 0 1 3 2 0

CODE :

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

using namespace std;

class Graph {
private:
    int N; // No. of nodes
    vector<vector<int>> adjList; // Adjacency List

public:
    Graph(int n) {
        N = n;
        adjList.resize(n);
    }

    void addEdge(int x, int y) {
        adjList[x].push_back(y);
        adjList[y].push_back(x);
    }

    void findChromaticNumber(const vector<int>& color) {
        unordered_set<int> colorSet;
        for (int c : color) {
            colorSet.insert(c);
        }
        int chromaticNo = colorSet.size();
        cout << "The chromatic number of the graph is: " << chromaticNo << endl;
    }
```

```
    void greedyColorNodes() {
        vector<int> res(N, -1); // Initializing all vertices as unassigned
        res[0] = 0; // Assigning the first color to the first vertex
        vector<bool> avail(N, true); // Availability of colors

        // Assign colors to the remaining N - 1 nodes
        for (int n = 1; n < N; n++) {
            for (int neighbor : adjList[n]) {
                if (res[neighbor] != -1)
                    avail[res[neighbor]] = false;
            }

            int clr;
            for (clr = 0; clr < N; clr++) {
                if (avail[clr])
                    break;
            }

            res[n] = clr; // Assigning the found color
            fill(avail.begin(), avail.end(), true); // Resetting the availability array
        }

        // Printing the result
        for (int n = 0; n < N; n++) {
            cout << "Node " << n << " ---> Color - " << res[n] << endl;
        }

        // Finding the chromatic number of the graph
        findChromaticNumber(res);
    }
};
```

```
int main() {
    // Creating a graph with 5 nodes
    Graph graph1(5);

    // Adding edges between nodes
    graph1.addEdge(0, 1);
    graph1.addEdge(0, 2);
    graph1.addEdge(1, 2);
    graph1.addEdge(1, 3);
    graph1.addEdge(2, 3);
    graph1.addEdge(3, 4);

    cout << "Coloring of the graph 1 is: " << endl;

    // Coloring the nodes
    graph1.greedyColorNodes();

    cout << endl;

    // Creating a graph with 4 nodes
    Graph graph2(4);

    // Adding edges between nodes
    graph2.addEdge(0, 1);
    graph2.addEdge(0, 2);
    graph2.addEdge(1, 3);
    graph2.addEdge(2, 3);

    cout << "Coloring of the graph 2 is: " << endl;

    // Coloring the nodes
    graph2.greedyColorNodes();
```

```
    return 0;
}
```

OUTCOME:

Coloring of the graph 1 is:
Node 0 ---> Color - 0
Node 1 ---> Color - 1
Node 2 ---> Color - 2
Node 3 ---> Color - 0
Node 4 ---> Color - 1
The chromatic number of the graph is: 3

Coloring of the graph 2 is:
Node 0 ---> Color - 0
Node 1 ---> Color - 1
Node 2 ---> Color - 1
Node 3 ---> Color - 0
The chromatic number of the graph is: 2