Unit 3

1.floyd warshall:

```
#include <iostream>

#include <vector>

#include <climits>  // For INT_MAX

// Function to implement Floyd-Warshall algorithm

void floydWarshall(std::vector<std::vector<int>>& graph, int V) {

   // dist[][] will be the output matrix that will hold the shortest distance
between every pair of vertices

   std::vector<std::vector<int>> dist = graph;

   // Apply Floyd-Warshall algorithm

   for (int k = 0; k < V; ++k) {

      for (int i = 0; i < V; ++i) {

         for (int j = 0; j < V; ++j) {

            if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX && dist[i][j] >
dist[i][k] + dist[k][j]) {

               dist[i][j] = dist[i][k] + dist[k][j];

            }

         }

      }

   }

   // Print the shortest distance matrix

   std::cout << "Shortest distance matrix:" << std::endl;

   for (int i = 0; i < V; ++i) {

      for (int j = 0; j < V; ++j) {

         if (dist[i][j] == INT_MAX)

            std::cout << "INF ";
```

```
            else
                std::cout << dist[i][j] << " ";
        }
        std::cout << std::endl;
    }
}
int main() {
    int V;  // Number of vertices
    std::cout << "Enter the number of vertices: ";
    std::cin >> V;
    // Initialize the graph with distances
    std::vector<std::vector<int>> graph(V, std::vector<int>(V, INT_MAX));
    std::cout << "Enter the adjacency matrix (use " << INT_MAX << " for no
edge):" << std::endl;
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            std::cin >> graph[i][j];
        }
    }
    // Run Floyd-Warshall algorithm
    floydWarshall(graph, V);
    return 0;
}
```

2.coin changing:

```
#include <iostream>
#include <vector>
#include <algorithm> // For std::min
```

```
// Function to find the minimum number of coins for the given amount
int coinChange(std::vector<int>& coins, int amount) {
   // Create a DP array and initialize it with a large number (representing
infinity)
   std::vector<int> dp(amount + 1, amount + 1);
   dp[0] = 0;  // Base case: no coins needed to make amount 0

   // Loop through each amount from 1 to 'amount'
   for (int i = 1; i <= amount; ++i) {
      // Loop through each coin denomination
      for (int coin : coins) {
         if (i - coin >= 0) {
            dp[i] = std::min(dp[i], dp[i - coin] + 1);
         }
      }
   }

   // If dp[amount] is still a large number, return -1 (indicating no solution)
   return dp[amount] > amount ? -1 : dp[amount];
}

int main() {
   int amount, n;
   std::cout << "Enter the amount: ";
   std::cin >> amount;
```

```
    std::cout << "Enter the number of coin denominations: ";
    std::cin >> n;

    std::vector<int> coins(n);
    std::cout << "Enter the coin denominations: ";
    for (int i = 0; i < n; ++i) {
        std::cin >> coins[i];
    }

    int result = coinChange(coins, amount);
    if (result == -1) {
        std::cout << "It's not possible to make the given amount with the provided
coins." << std::endl;
    } else {
        std::cout << "Minimum coins required: " << result << std::endl;
    }

    return 0;
}
```

3.friendspairing:

```
// C++ program for solution of
// friends pairing problem
#include <bits/stdc++.h>
using namespace std;

// Returns count of ways n people
// can remain single or paired up.
```

```
int countFriendsPairings(int n)
{
	int dp[n + 1];

	// Filling dp[] in bottom-up manner using
	// recursive formula explained above.
	for (int i = 0; i <= n; i++) {
		if (i <= 2)
			dp[i] = i;
		else
			dp[i] = dp[i - 1] + (i - 1) * dp[i - 2];
	}

	return dp[n];
}

// Driver code
int main()
{
	int n = 4;
	cout << countFriendsPairings(n) << endl;
	return 0;
}
```

Unit 4

1.Sieve of Sundaram:

```
#include <iostream>
```

```
#include <vector>
#include <cmath>

void sieveOfSundaram(int N) {
   // Calculate the maximum value for i and j
   int n = (N - 1) / 2;

   // Create a boolean array and initialize all values as true
   std::vector<bool> marked(n + 1, false);

   // Mark numbers of the form i + j + 2ij
   for (int i = 1; i <= n; ++i) {
      for (int j = i; (i + j + 2 * i * j) <= n; ++j) {
         marked[i + j + 2 * i * j] = true;
      }
   }

   // Print all primes using the information in 'marked'
   std::cout << "Prime numbers less than " << 2 * N + 2 << " are: " << std::endl;

   // 2 is a prime number
   if (N >= 2) {
      std::cout << 2 << " ";
   }

   // All numbers of the form 2i + 1 are prime, where 'i' is unmarked
```

```
    for (int i = 1; i <= n; ++i) {
        if (!marked[i]) {
            std::cout << 2 * i + 1 << " ";
        }
    }
    std::cout << std::endl;
}

int main() {
    int N;
    std::cout << "Enter the value of N: ";
    std::cin >> N;

    sieveOfSundaram(N);

    return 0;
}
```

2.Activity selection:

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

struct Activity {
    int start;
    int end;
};
```

```
// Function to compare two activities based on their end times
bool compare(Activity a, Activity b) {
    return a.end < b.end;
}

// Function to perform the Activity Selection
void activitySelection(std::vector<Activity>& activities) {
    // Sort activities by end time
    std::sort(activities.begin(), activities.end(), compare);

    // Select the first activity
    int n = activities.size();
    int lastSelected = 0;  // Index of the last selected activity
    std::cout << "Selected activities: \n";
    std::cout << "Activity 1: (" << activities[lastSelected].start << ", " <<
activities[lastSelected].end << ")\n";

    // Loop through the rest of the activities
    for (int i = 1; i < n; ++i) {
        // If the start time of the current activity is greater than or equal to the
end time of the last selected activity
        if (activities[i].start >= activities[lastSelected].end) {
            std::cout << "Activity " << i + 1 << ": (" << activities[i].start << ", " <<
activities[i].end << ")\n";
            lastSelected = i;  // Update the index of the last selected activity
        }
```

```
    }
}

int main() {
    int n;
    std::cout << "Enter the number of activities: ";
    std::cin >> n;

    std::vector<Activity> activities(n);
    std::cout << "Enter the start and end times of the activities (start end): \n";

    for (int i = 0; i < n; ++i) {
        std::cin >> activities[i].start >> activities[i].end;
    }

    // Perform activity selection
    activitySelection(activities);

    return 0;
}
```

3.mice problem:

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

```cpp
int assignMiceToHoles(std::vector<int>& mice, std::vector<int>& holes) {
    // Sort the positions of mice and holes
    std::sort(mice.begin(), mice.end());
    std::sort(holes.begin(), holes.end());

    // Initialize the variable to track the maximum distance a mouse has to travel
    int maxDistance = 0;

    // Pair each mouse with a hole and calculate the distance
    for (int i = 0; i < mice.size(); ++i) {
        int distance = std::abs(mice[i] - holes[i]);
        maxDistance = std::max(maxDistance, distance);
    }

    return maxDistance;
}

int main() {
    int n;
    std::cout << "Enter the number of mice and holes: ";
    std::cin >> n;

    std::vector<int> mice(n), holes(n);

    std::cout << "Enter the positions of the mice: ";
    for (int i = 0; i < n; ++i) {
```

```
        std::cin >> mice[i];
    }

    std::cout << "Enter the positions of the holes: ";
    for (int i = 0; i < n; ++i) {
        std::cin >> holes[i];
    }

    int result = assignMiceToHoles(mice, holes);
    std::cout << "The minimum time required for all mice to enter a hole is: " <<
result << std::endl;

    return 0;
}
```

Unit 5

1.Knight tour problem:

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

// Size of the chessboard
#define N 8

// Knight's possible moves
int dx[] = {2, 1, -1, -2, -2, -1, 1, 2};
int dy[] = {1, 2, 2, 1, -1, -2, -2, -1};
```

```
// Function to check if a move is valid
bool isSafe(int x, int y, vector<vector<int>>& board) {
   return (x >= 0 && y >= 0 && x < N && y < N && board[x][y] == -1);
}

// Function to print the chessboard
void printBoard(const vector<vector<int>>& board) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         cout << board[i][j] << " ";
      }
      cout << endl;
   }
}

// Backtracking function to solve the Knight's Tour problem
bool knightTour(int x, int y, int moveCount, vector<vector<int>>& board) {
   // If all squares are visited, return true
   if (moveCount == N * N)
      return true;

   // Try all the possible moves for the knight
   for (int i = 0; i < 8; i++) {
      int newX = x + dx[i];
      int newY = y + dy[i];
```

```
        if (isSafe(newX, newY, board)) {
            board[newX][newY] = moveCount;
            if (knightTour(newX, newY, moveCount + 1, board)) {
                return true;
            }
            // Backtrack
            board[newX][newY] = -1;
        }
    }

    return false; // No valid move found
}

int main() {
    // Initialize the chessboard with -1 (indicating unvisited squares)
    vector<vector<int>> board(N, vector<int>(N, -1));

    // Starting position (usually top-left corner)
    int startX = 0, startY = 0;

    // Mark the starting position
    board[startX][startY] = 0;

    // Start the knight's tour from the initial position
    if (knightTour(startX, startY, 1, board)) {
        // Print the solution
```

```
        cout << "Solution found:\n";
        printBoard(board);
    } else {
        cout << "Solution does not exist\n";
    }

    return 0;
}
```

2.subset sum problem:

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

// Function to solve the Subset Sum Problem using backtracking
bool isSubsetSum(const vector<int>& set, int n, int target) {
    // Base cases
    if (target == 0) {
        return true;  // We found a subset that sums to the target
    }
    if (n == 0 && target != 0) {
        return false;  // No elements left and target is not 0, so return false
    }

    // If last element is greater than target, we can't include it
    if (set[n - 1] > target) {
        return isSubsetSum(set, n - 1, target);  // Exclude last element
```

```
    }

    // Check two possibilities:
    // 1. Include the last element in the subset
    // 2. Exclude the last element from the subset
    return isSubsetSum(set, n - 1, target) || isSubsetSum(set, n - 1, target - set[n
- 1]);
}

int main() {
    int n, target;

    // Read the number of elements and the target sum
    cout << "Enter the number of elements in the set: ";
    cin >> n;

    vector<int> set(n);

    cout << "Enter the elements of the set: ";
    for (int i = 0; i < n; i++) {
        cin >> set[i];
    }

    cout << "Enter the target sum: ";
    cin >> target;

    // Call the isSubsetSum function and print the result
```

```
    if (isSubsetSum(set, n, target)) {
        cout << "There is a subset with the given sum." << endl;
    } else {
        cout << "No subset with the given sum exists." << endl;
    }

    return 0;
}
```

Unit 2

1.Travelling Salesman:

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

int tsp(const std::vector<std::vector<int>>& graph, std::vector<bool>& visited, int current, int n, int count, int cost, int& minCost) {
    if (count == n && graph[current][0]) {
        minCost = std::min(minCost, cost + graph[current][0]);
        return minCost;
    }

    for (int i = 0; i < n; i++) {
        if (!visited[i] && graph[current][i]) {
            visited[i] = true;
            tsp(graph, visited, i, n, count + 1, cost + graph[current][i], minCost);
            visited[i] = false;
        }
```

```
    }
    return minCost;
}

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

    std::vector<std::vector<int>> graph(n, std::vector<int>(n));
    std::cout << "Enter the distance matrix:\n";
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            std::cin >> graph[i][j];

    std::vector<bool> visited(n, false);
    visited[0] = true;

    int minCost = INT_MAX;
    std::cout << "Minimum cost: " << tsp(graph, visited, 0, n, 1, 0, minCost) <<
std::endl;
    return 0;
}
```

2.Assignment problem:

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

```cpp
#include <limits.h>

// Function to calculate the cost for a specific assignment
int calculateCost(const std::vector<std::vector<int>>& costMatrix, const
std::vector<int>& assignment) {
    int totalCost = 0;
    for (int i = 0; i < assignment.size(); ++i) {
        totalCost += costMatrix[i][assignment[i]];
    }
    return totalCost;
}

// Brute force solution to the Assignment Problem
int solveAssignmentProblem(const std::vector<std::vector<int>>& costMatrix) {
    int n = costMatrix.size();
    std::vector<int> assignment(n); // Stores task indices
    for (int i = 0; i < n; ++i) {
        assignment[i] = i;
    }

    int minCost = INT_MAX;
    std::vector<int> bestAssignment;

    // Generate all permutations of tasks
    do {
        int currentCost = calculateCost(costMatrix, assignment);
        if (currentCost < minCost) {
```

```
            minCost = currentCost;
            bestAssignment = assignment;
        }
    } while (std::next_permutation(assignment.begin(), assignment.end()));

    // Output the best assignment
    std::cout << "Best Assignment:\n";
    for (int i = 0; i < bestAssignment.size(); ++i) {
        std::cout << "Agent " << i + 1 << " -> Task " << bestAssignment[i] + 1 <<
"\n";
    }

    return minCost;
}

int main() {
    int n;
    std::cout << "Enter the number of agents/tasks: ";
    std::cin >> n;

    std::vector<std::vector<int>> costMatrix(n, std::vector<int>(n));
    std::cout << "Enter the cost matrix (row-wise):\n";
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            std::cin >> costMatrix[i][j];
        }
    }
```

```
    int minCost = solveAssignmentProblem(costMatrix);
    std::cout << "Minimum Cost: " << minCost << std::endl;

    return 0;
}
```

Unit 1

1.Rabin Karp:

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

const int d = 256; // Number of characters in the input alphabet
const int q = 101; // A prime number for modulo

void rabinKarp(const std::string& text, const std::string& pattern) {
    int n = text.length();
    int m = pattern.length();
    int p = 0;  // Hash value for the pattern
    int t = 0;  // Hash value for the text
    int h = 1;

    // Calculate h = d^(m-1) % q
    for (int i = 0; i < m - 1; ++i) {
        h = (h * d) % q;
    }
```

```
// Calculate initial hash values for pattern and text
for (int i = 0; i < m; ++i) {
    p = (d * p + pattern[i]) % q;
    t = (d * t + text[i]) % q;
}

// Slide the pattern over the text
for (int i = 0; i <= n - m; ++i) {
    // Check hash values
    if (p == t) {
        // Verify characters
        bool match = true;
        for (int j = 0; j < m; ++j) {
            if (text[i + j] != pattern[j]) {
                match = false;
                break;
            }
        }
        if (match) {
            std::cout << "Pattern found at index " << i << std::endl;
        }
    }

    // Calculate hash for next window
    if (i < n - m) {
```

```
            t = (d * (t - text[i] * h) + text[i + m]) % q;
            if (t < 0) t += q; // Ensure positive hash value
        }
    }
}

int main() {
    std::string text, pattern;
    std::cout << "Enter the text: ";
    std::cin >> text;
    std::cout << "Enter the pattern: ";
    std::cin >> pattern;

    rabinKarp(text, pattern);
    return 0;
}
```

2.Kmp:

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

// Function to compute the LPS array
void computeLPS(const std::string& pattern, std::vector<int>& lps) {
    int m = pattern.length();
    int len = 0; // Length of the previous longest prefix suffix
    lps[0] = 0;  // LPS[0] is always 0
```

```
    int i = 1;

    while (i < m) {
        if (pattern[i] == pattern[len]) {
            ++len;
            lps[i] = len;
            ++i;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                ++i;
            }
        }
    }
}

// Function to perform KMP pattern matching
void KMP(const std::string& text, const std::string& pattern) {
    int n = text.length();
    int m = pattern.length();
    std::vector<int> lps(m);

    // Precompute the LPS array
    computeLPS(pattern, lps);
```

```
    int i = 0; // Index for text
    int j = 0; // Index for pattern

    while (i < n) {
        if (text[i] == pattern[j]) {
            ++i;
            ++j;
        }

        if (j == m) {
            std::cout << "Pattern found at index " << i - j << std::endl;
            j = lps[j - 1];
        } else if (i < n && text[i] != pattern[j]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                ++i;
            }
        }
    }
}

int main() {
    std::string text, pattern;
    std::cout << "Enter the text: ";
```

```
    std::cin >> text;
    std::cout << "Enter the pattern: ";
    std::cin >> pattern;

    KMP(text, pattern);
    return 0;
}
```

3.manacher’s:

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

// Function to preprocess the string for handling even-length palindromes
std::string preprocess(const std::string& s) {
    std::string result = "@"; // Start sentinel
    for (char c : s) {
        result += "#" + std::string(1, c);
    }
    result += "#$"; // End sentinel
    return result;
}

// Manacher's Algorithm
std::string longestPalindromicSubstring(const std::string& s) {
    std::string t = preprocess(s);
    int n = t.length();
```

```
std::vector<int> p(n, 0);
int center = 0, right = 0;

for (int i = 1; i < n - 1; ++i) {
    int mirror = 2 * center - i; // Mirror index

    if (i < right) {
        p[i] = std::min(right - i, p[mirror]);
    }

    // Expand around the center
    while (t[i + p[i] + 1] == t[i - p[i] - 1]) {
        ++p[i];
    }

    // Update center and right boundary
    if (i + p[i] > right) {
        center = i;
        right = i + p[i];
    }
}

// Find the longest palindrome
int maxLength = 0, start = 0;
for (int i = 1; i < n - 1; ++i) {
    if (p[i] > maxLength) {
```

```
            maxLength = p[i];
            start = (i - maxLength) / 2;
        }
    }

    return s.substr(start, maxLength);
}

int main() {
    std::string str;
    std::cout << "Enter the string: ";
    std::cin >> str;

    std::string result = longestPalindromicSubstring(str);
    std::cout << "Longest palindromic substring: " << result << std::endl;

    return 0;
}
```

N queens problem:

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

// Function to check if the queen can be placed at (row, col)
bool isSafe(int row, int col, vector<vector<int>>& board, int N) {
    // Check column
```

```
    for (int i = 0; i < row; i++) {
        if (board[i][col] == 1) {
            return false;
        }
    }

    // Check upper-left diagonal
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    // Check upper-right diagonal
    for (int i = row, j = col; i >= 0 && j < N; i--, j++) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    return true;
}

// Backtracking function to solve N Queens problem
bool solveNQueens(vector<vector<int>>& board, int row, int N) {
    // If all queens are placed, return true
```

```
    if (row == N) {
        return true;
    }

    // Consider this row and try all columns
    for (int col = 0; col < N; col++) {
        // Check if it's safe to place the queen at (row, col)
        if (isSafe(row, col, board, N)) {
            // Place queen
            board[row][col] = 1;

            // Recur to place the rest of the queens
            if (solveNQueens(board, row + 1, N)) {
                return true;
            }

            // Backtrack: If placing queen in board[row][col] doesn't lead to a
solution, remove queen
            board[row][col] = 0;
        }
    }

    return false;  // No solution found
}

// Function to print the chessboard
void printBoard(const vector<vector<int>>& board, int N) {
```

```
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i][j] == 1)
                cout << "Q ";
            else
                cout << ". ";
        }
        cout << endl;
    }
}

int main() {
    int N;
    cout << "Enter the size of the board (N): ";
    cin >> N;

    // Initialize the chessboard with 0s
    vector<vector<int>> board(N, vector<int>(N, 0));

    // Try to solve the N Queens problem
    if (solveNQueens(board, 0, N)) {
        cout << "Solution to N Queens problem: \n";
        printBoard(board, N);
    } else {
        cout << "No solution exists." << endl;
    }
```

```
    return 0;
}
```