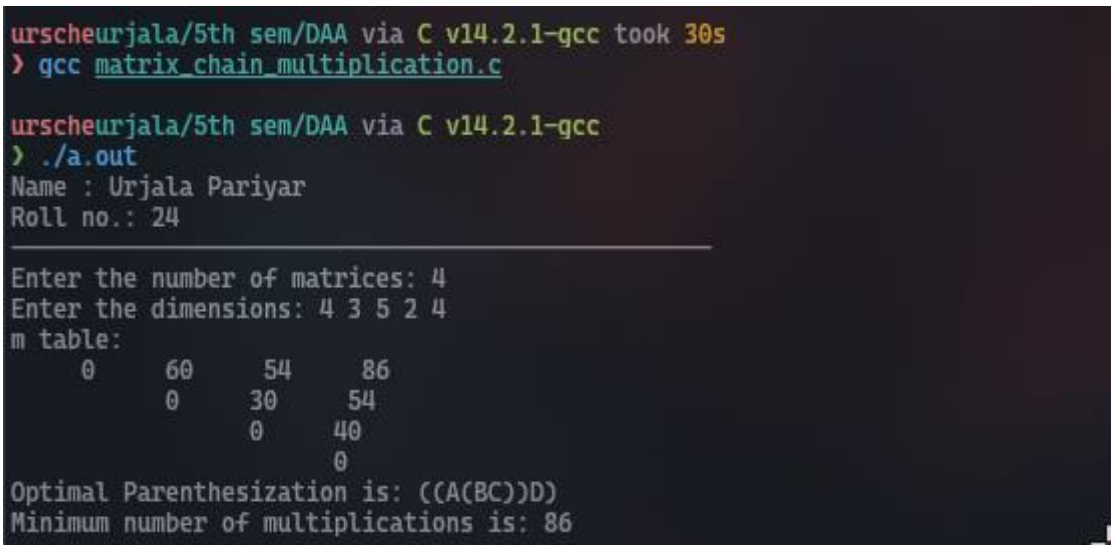
Implementation of Matrix Chain Multiplication

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

void printOptimalParenthesis(int i, int j, int n, int* bracket, char* name) {
    if (i == j) {
        printf("%c", *name);
        (*name)++;
        return;
    }
    printf("(");
    printOptimalParenthesis(i, *((bracket + i * n) + j), n, bracket, name);
    printOptimalParenthesis(*((bracket + i * n) + j) + 1, j, n, bracket, name);

    printf(")");
}
int matrixChainOrder(int p[], int n) {
    int m[n][n];
    int bracket[n][n];

    for (int i = 1; i < n; i++)
        m[i][i] = 0;

    for (int l = 2; l < n; l++) {
        for (int i = 1; i < n - l + 1; i++) {
            int j = i + l - 1;
            m[i][j] = INT_MAX;

            for (int k = i; k < j; k++) {
                int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j]) {
                    m[i][j] = q;
                    bracket[i][j] = k;
                }
            }
        }
    }

    // Print the m table
    printf("m table:\n");
    for (int i = 1; i < n; i++) {
        for (int j = 1; j < n; j++) {
            if (j < i) {
                printf("      ");
            } else {
                printf("%6d ", m[i][j]);
```

```
            }
        }
        printf("\n");
    }

    // Print the optimal parenthesis order
    char name = 'A';
    printf("Optimal Parenthesization is: ");
    printOptimalParenthesis(1, n - 1, n, (int*)bracket, &name);
    printf("\n");

    return m[1][n - 1];
}

int main() {
    int n;

    // Input the number of matrices
    printf("Enter the number of matrices: ");
    scanf("%d", &n);

    int p[n + 1];
    printf("Enter the dimensions: ");
    for (int i = 0; i <= n; i++) {
        scanf("%d", &p[i]);
    }

    int minCost = matrixChainOrder(p, n + 1);

    printf("Minimum number of multiplications is: %d\n", minCost);

    return 0;
}
```

OUTPUT:

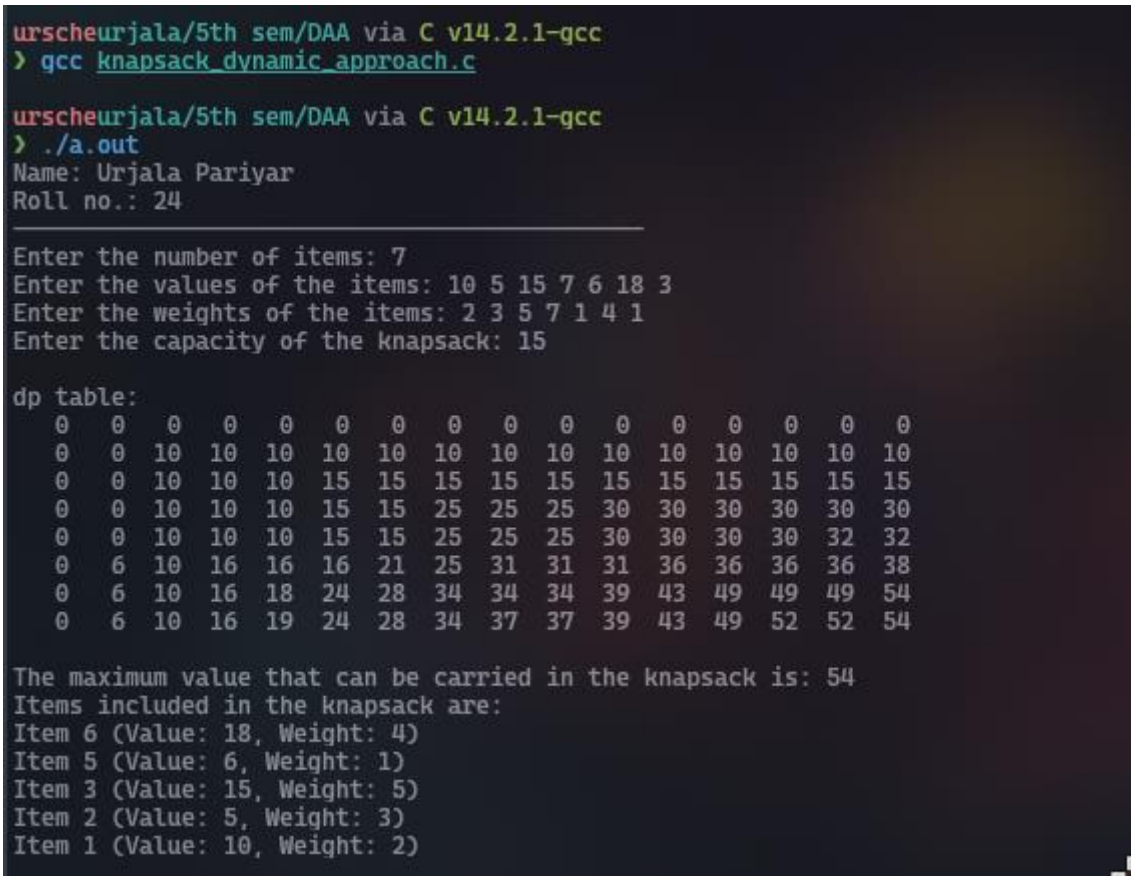


Implementation of 0/1 Knapsack Problem using dynamic approach

```
#include <stdio.h>
// Function to return the maximum of two integers
int max(int a, int b) {
   return (a > b) ? a : b;
}
// Function to solve the 0/1 Knapsack problem using dynamic programming
int knapsack(int W, int wt[], int val[], int n) {
   int dp[n + 1][W + 1];
   // Build table dp[][] in a bottom-up manner
   for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= W; w++) {
         if (i == 0 || w == 0) {
            dp[i][w] = 0;  // Base case: no items or zero capacity
         } else if (wt[i - 1] <= w) {
            // Include the item or exclude it
            dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);
         } else {
            dp[i][w] = dp[i - 1][w];  // Exclude the item
         }
      }
   }
   // Print the dp table
   printf("\ndp table:\n");
   for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= W; w++) {
         printf("%4d", dp[i][w]);
      }
      printf("\n");
   }
   // Find out which items are included in the optimal solution
   int res = dp[n][W];
   printf("\nThe maximum value that can be carried in the knapsack is: %d\n", res);
   int w = W;
   printf("Items included in the knapsack are:\n");
   for (int i = n; i > 0 && res > 0; i--) {
      if (res != dp[i - 1][w]) {
         // This item is included.
         printf("Item %d (Value: %d, Weight: %d)\n", i, val[i - 1], wt[i - 1]);

         // Since this item is included, its value is deducted
         res -= val[i - 1];
         w -= wt[i - 1];
      }
```

```
    }
    return dp[n][W];
}
int main() {
    int n, W;
    // Input number of items
    printf("Enter the number of items: ");
    scanf("%d", &n);
    int val[n], wt[n];
    // Input values and weights of items
    printf("Enter the values of the items: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &val[i]);
    }
    printf("Enter the weights of the items: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &wt[i]);
    }
    // Input the capacity of the knapsack
    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &W);
    // Calculate the maximum value that can be carried
    knapsack(W, wt, val, n);
    return 0;
}
```

OUTPUT

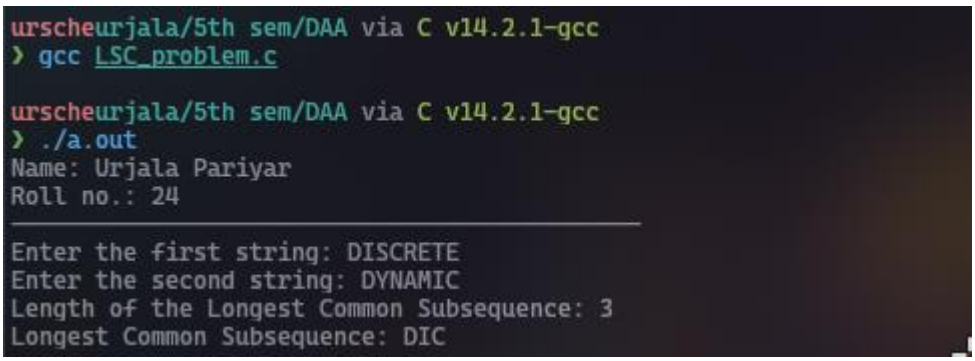

urscheurjala/5th sem/DAA via C v14.2.1-gcc
❯ gcc knapsack_dynamic_approach.c
urscheurjala/5th sem/DAA via C v14.2.1-gcc
❯ ./a.out
Name: Urjala Pariyar
Roll no.: 24
Enter the number of items: 7
Enter the values of the items: 10 5 15 7 6 18 3
Enter the weights of the items: 2 3 5 7 1 4 1
Enter the capacity of the knapsack: 15
dp table:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 10 10 10 10 10 10 10 10 10 10 10 10 10 10
0 0 10 10 10 15 15 15 15 15 15 15 15 15 15 15
0 0 10 10 10 15 15 25 25 25 30 30 30 30 30 30
0 0 10 10 10 15 15 25 25 25 30 30 30 30 32 32
0 6 10 16 16 16 21 25 31 31 31 36 36 36 36 38
0 6 10 16 18 24 28 34 34 34 39 43 49 49 49 54
0 6 10 16 19 24 28 34 37 37 39 43 49 52 52 54
The maximum value that can be carried in the knapsack is: 54
Items included in the knapsack are:
Item 6 (Value: 18, Weight: 4)
Item 5 (Value: 6, Weight: 1)
Item 3 (Value: 15, Weight: 5)
Item 2 (Value: 5, Weight: 3)
Item 1 (Value: 10, Weight: 2)

Implementation of LCS Problem

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

// Function to find the length of the longest common subsequence and print the LCS
void lcsAndPrint(char *X, char *Y, int m, int n) {
    int L[m + 1][n + 1];
    int i, j;

    // Building the LCS table in bottom-up manner
    for (i = 0; i <= m; i++) {
        for (j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                L[i][j] = 0;
            } else if (X[i - 1] == Y[j - 1]) {
                L[i][j] = L[i - 1][j - 1] + 1;
            } else {
                L[i][j] = (L[i - 1][j] > L[i][j - 1]) ? L[i - 1][j] : L[i][j - 1];
            }
        }
    }

    // L[m][n] contains the length of the LCS for X[0..m-1], Y[0..n-1]
    int length = L[m][n];
    printf("Length of the Longest Common Subsequence: %d\n", length);

    // Following the table to print the LCS
    int index = length;
    char lcs[index + 1];
    lcs[index] = '\0';  // Set the terminating character

    i = m;
    j = n;
    while (i > 0 && j > 0) {
        if (X[i - 1] == Y[j - 1]) {
            lcs[index - 1] = X[i - 1];
            i--;
            j--;
            index--;
        } else if (L[i - 1][j] > L[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }
```

```
    printf("Longest Common Subsequence: %s\n", lcs);
}

int main() {
    char X[100], Y[100];

    // Input strings from the user
    printf("Enter the first string: ");
    scanf("%s", X);
    printf("Enter the second string: ");
    scanf("%s", Y);

    int m = strlen(X);
    int n = strlen(Y);

    // Call the LCS function
    lcsAndPrint(X, Y, m, n);

    return 0;
}
```

OUTPUT:

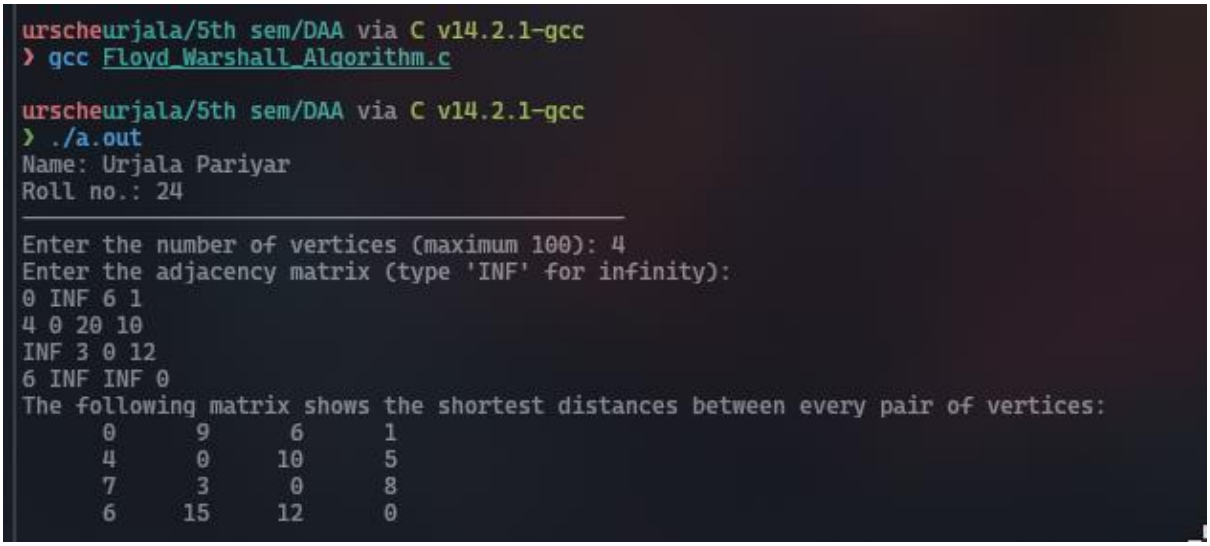


Implementation of Floyd Warshall Algorithm

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

#define MAX 100  // Define a maximum number of vertices
#define INF INT_MAX  // Define INF as INT_MAX

// Function to print the solution matrix
void printSolution(int dist[MAX][MAX], int V) {
    printf("The following matrix shows the shortest distances between every pair of vertices:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// Function to implement Floyd-Warshall algorithm
void floydWarshall(int graph[MAX][MAX], int V) {
    int dist[MAX][MAX];  // dist[i][j] will hold the shortest distance from i to j

    // Initialize the solution matrix same as input graph matrix.
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
        }
    }

    // Add all vertices one by one to the set of intermediate vertices.
    for (int k = 0; k < V; k++) {
        // Pick all vertices as source one by one
        for (int i = 0; i < V; i++) {
            // Pick all vertices as destination for the above picked source
            for (int j = 0; j < V; j++) {
                // If vertex k is on the shortest path from i to j, then update the value of dist[i][j]
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
```

```
    printSolution(dist, V);
}

int main() {
    int V;
    printf("Enter the number of vertices (maximum %d): ", MAX);
    scanf("%d", &V);

    if (V > MAX) {
        printf("Error: The number of vertices exceeds the maximum allowed (%d).\n", MAX);
        return 1;
    }
    int graph[MAX][MAX];
    printf("Enter the adjacency matrix (type 'INF' for infinity):\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            char input[10];
            scanf("%s", input);

            if (strcmp(input, "INF") == 0) {
                graph[i][j] = INF;
            } else {
                graph[i][j] = atoi(input);
            }
        }
    }
    // Run Floyd-Warshall algorithm
    floydWarshall(graph, V);
    return 0;
}
```

OUTPUT:

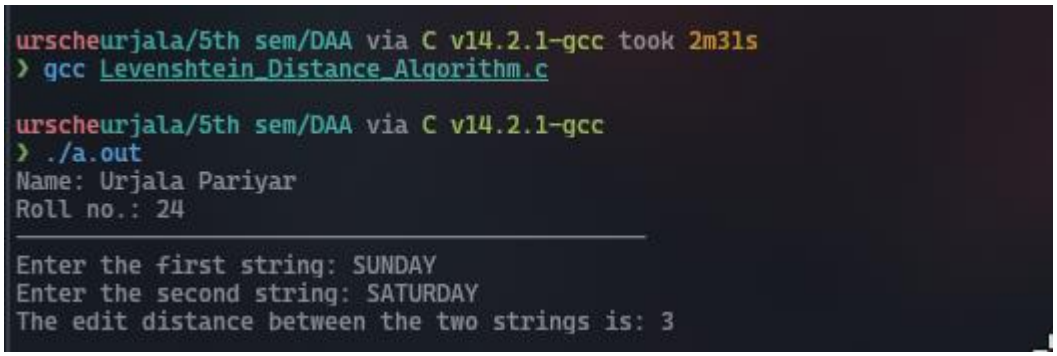


Implementation of Levenshtein Distance Algorithm

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

// Function to find the minimum of three numbers
int min(int x, int y, int z) {
   return x < y ? (x < z ? x : z) : (y < z ? y : z);
}

// Function to compute the edit distance between two strings
int editDistance(char *str1, char *str2, int m, int n) {
   // Create a table to store results of subproblems
   int dp[m + 1][n + 1];

   // Fill dp[][] in bottom up manner
   for (int i = 0; i <= m; i++) {
      for (int j = 0; j <= n; j++) {
         // If first string is empty, the only option is to insert all characters of the second string
         if (i == 0) {
            dp[i][j] = j; // Insert all j characters of str2
         }
         // If second string is empty, the only option is to remove all characters of the first string
         else if (j == 0) {
            dp[i][j] = i; // Remove all i characters of str1
         }
         // If last characters are the same, ignore the last character and recur for the remaining substring
         else if (str1[i - 1] == str2[j - 1]) {
            dp[i][j] = dp[i - 1][j - 1];
         }
         // If last characters are different, consider all possibilities and find the minimum
         else {
            dp[i][j] = 1 + min(dp[i][j - 1],    // Insert
                               dp[i - 1][j],    // Remove
                               dp[i - 1][j - 1] // Replace
                               );
         }
      }
   }

   return dp[m][n];
}

int main() {
printf("Name: Urjala Pariyar\n");
   printf("Roll no.: 24\n");
```

```
    printf("-----------------------------------------\n");
    char str1[100], str2[100];

    // Input strings
    printf("Enter the first string: ");
    scanf("%s", str1);
    printf("Enter the second string: ");
    scanf("%s", str2);

    int m = strlen(str1);
    int n = strlen(str2);

    // Calculate edit distance
    int result = editDistance(str1, str2, m, n);

    // Output the result
    printf("The edit distance between the two strings is: %d\n", result);

    return 0;
}
```

OUTPUT:



Implementation of TSP Problem

```
#include <stdio.h>
#include <limits.h>
#include <string.h> // For memset
#define MAX_N 15 // Maximum number of cities

int n; // Number of cities
int cost[MAX_N][MAX_N]; // Cost matrix
int dp[1 << MAX_N][MAX_N]; // Dynamic programming table
int path[1 << MAX_N][MAX_N]; // To store the path
// Helper function to get the minimum of two integers
int min(int a, int b) {
    return a < b ? a : b;
}
// Recursive function to solve the TSP problem
int tsp(int mask, int pos) {
    if (mask == (1 << n) - 1) {
        return cost[pos][0]; // Return to the starting city
    }
    if (dp[mask][pos] != -1) {
        return dp[mask][pos];
    }
    int ans = INT_MAX;
    int best_next_city = -1;

    for (int next = 0; next < n; next++) {
        if ((mask & (1 << next)) == 0) { // If city 'next' is not visited
            int newCost = cost[pos][next] + tsp(mask | (1 << next), next);
            if (newCost < ans) {
                ans = newCost;
                best_next_city = next;
            }
        }
    }
    path[mask][pos] = best_next_city; // Store the best path
    return dp[mask][pos] = ans;
}
// Function to print the path taken
void printPath() {
    int mask = 1; // Starting with city 0
    int pos = 0;  // Starting from city 0
    printf("Path: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", pos + 1); // Print the current city (1-based index)
        int next_city = path[mask][pos];
        mask |= (1 << next_city); // Mark the next city as visited
```

```
        pos = next_city; // Move to the next city
    }
    printf("%d\n", 1); // Return to the starting city
}
int main() {
printf("Name: Urjala Pariyar\n");
    printf("Roll no.: 24\n");
    printf("-----------------------------------------\n");
    printf("Enter the number of cities: ");
    scanf("%d", &n);
    if (n > MAX_N) {
        printf("Number of cities exceeds maximum allowed (%d).\n", MAX_N);
        return 1;
    }
    printf("Enter the cost matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] < 0) {
                printf("Cost cannot be negative.\n");
                return 1;
            }
        }
    }
    memset(dp, -1, sizeof(dp));
    memset(path, -1, sizeof(path));
    int minCost = tsp(1, 0); // Start from city 0
    printPath(); // Print the path taken
    printf("Minimum cost of the TSP: %d\n", minCost);
    return 0;
}
```

OUTPUT:

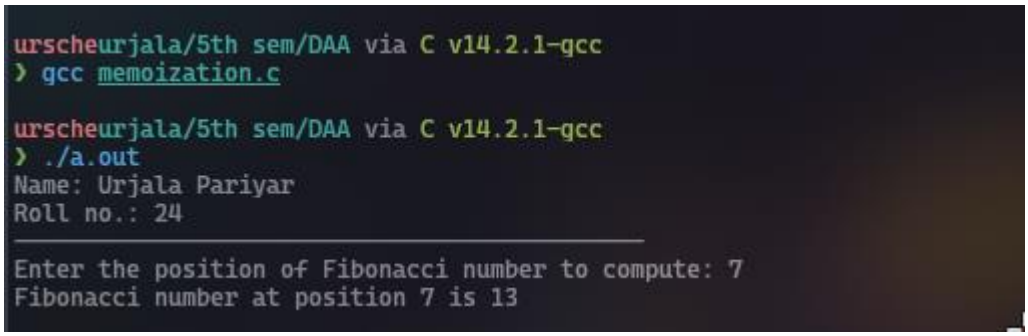
```
urscheurjala/5th sem/DAA via C v14.2.1-gcc took 27s
❯ gcc TSP_problem.c

urscheurjala/5th sem/DAA via C v14.2.1-gcc
❯ ./a.out
Name: Urjala Pariyar
Roll no.: 24
-----------------------------------------
Enter the number of cities: 4
Enter the cost matrix:
0 4 1 3
4 0 2 1
1 2 0 5
3 1 5 0
Path: 1 3 2 4 1
Minimum cost of the TSP: 7
```

Implementation of Memoization to calculate fibonacci

```
#include <stdio.h>

#define MAX 1000  // Maximum size of the memoization array

// Memoization table to store computed Fibonacci values
int memo[MAX];

// Function to initialize the memoization table
void initializeMemo() {
    for (int i = 0; i < MAX; i++) {
        memo[i] = -1;  // -1 indicates that the value is not computed yet
    }
}

// Recursive function to compute Fibonacci numbers using memoization
int fibonacci(int n) {
    // Base cases
    if (n <= 1) {
        return n;
    }

    // Check if the value is already computed
    if (memo[n] != -1) {
        return memo[n];
    }

    // Compute and store the value
    memo[n] = fibonacci(n - 1) + fibonacci(n - 2);
    return memo[n];
}

int main() {
printf("Name: Urjala Pariyar\n");
    printf("Roll no.: 24\n");
    printf("-----------------------------------------\n");
    int n;
    printf("Enter the position of Fibonacci number to compute: ");
    scanf("%d", &n);

    // Check for valid input
    if (n < 0 || n >= MAX) {
        printf("Position must be between 0 and %d.\n", MAX - 1);
        return 1;
    }
```

```
    // Initialize memoization table
    initializeMemo();

    // Compute and print the Fibonacci number
    printf("Fibonacci number at position %d is %d\n", n, fibonacci(n));

    return 0;
}
```

OUTPUT:

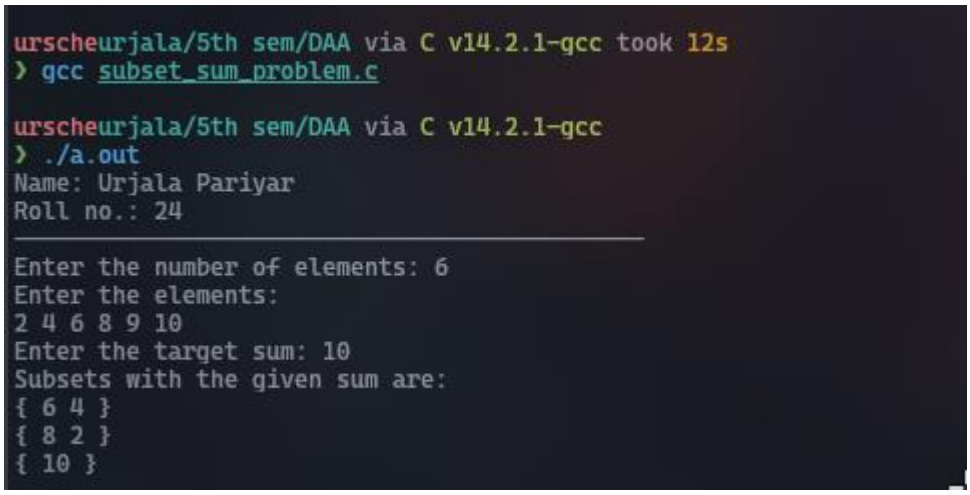


Implementation of Subset Sum Problem using Backtracking

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 100

// Function to print the subset
void printSubset(int subset[], int subsetSize) {
   printf("{ ");
   for (int i = 0; i < subsetSize; i++) {
      printf("%d ", subset[i]);
   }
   printf("}\n");
}

// Recursive function to find all subsets with the given sum
void findSubsetsWithSum(int arr[], int n, int targetSum, int subset[], int subsetSize) {
   // Base case: if targetSum is 0, then we found a subset
   if (targetSum == 0) {
      printSubset(subset, subsetSize);
      return;
   }

   // Base case: no elements left or targetSum is negative
   if (n == 0 || targetSum < 0) {
      return;
   }

   // Exclude the last element and try further
   findSubsetsWithSum(arr, n - 1, targetSum, subset, subsetSize);

   // Include the last element
   subset[subsetSize] = arr[n - 1];
   findSubsetsWithSum(arr, n - 1, targetSum - arr[n - 1], subset, subsetSize + 1);
}

int main() {
printf("Name: Urjala Pariyar\n");
   printf("Roll no.: 24\n");
   printf("-----------------------------------------\n");
   int n, targetSum;
   int arr[MAX], subset[MAX];

   // Input the number of elements
   printf("Enter the number of elements: ");
   scanf("%d", &n);
```

```
    // Input the elements of the array
    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Input the target sum
    printf("Enter the target sum: ");
    scanf("%d", &targetSum);

    // Initialize the subset array
    for (int i = 0; i < MAX; i++) {
        subset[i] = 0;
    }

    // Call the function to find all subsets with the given sum
    printf("Subsets with the given sum are:\n");
    findSubsetsWithSum(arr, n, targetSum, subset, 0);

    return 0;
}
```

OUTPUT:



Implementation of 0/1 Knapsack Problem using Backtracking

```
#include <stdio.h>

#define MAX_ITEMS 100

// Function to calculate the maximum value and track included items
void knapsack(int weights[], int values[], int n, int capacity, int currentIndex, int currentWeight, int
currentValue, int *maxValue, int includedItems[], int currentItems[]) {
   // Base case: If current weight exceeds capacity, return
   if (currentWeight > capacity) {
      return;
   }

   // Update the maximum value if the current value is higher
   if (currentValue > *maxValue) {
      *maxValue = currentValue;

      // Copy the current items to includedItems
      for (int i = 0; i < n; i++) {
         includedItems[i] = currentItems[i];
      }
   }

   // Try including the next item
   for (int i = currentIndex; i < n; i++) {
      currentItems[i] = 1; // Include item i
      knapsack(weights, values, n, capacity, i + 1, currentWeight + weights[i], currentValue +
values[i], maxValue, includedItems, currentItems);
      currentItems[i] = 0; // Exclude item i
   }
}

int main() {
printf("Name: Urjala Pariyar\n");
   printf("Roll no.: 24\n");
   printf("-----------------------------------------------\n");
   int n, capacity;
   int weights[MAX_ITEMS], values[MAX_ITEMS];
   int includedItems[MAX_ITEMS] = {0}; // To track included items
   int currentItems[MAX_ITEMS] = {0};  // To track current items in recursion

   // Input the number of items
   printf("Enter the number of items: ");
   scanf("%d", &n);

   // Input the weights and values of the items
```

```
    printf("Enter the weights of the items:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &weights[i]);
    }

    printf("Enter the values of the items:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &values[i]);
    }

    // Input the capacity of the knapsack
    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);

    int maxValue = 0;

    // Call the knapsack function
    knapsack(weights, values, n, capacity, 0, 0, 0, &maxValue, includedItems, currentItems);

    printf("The maximum value that can be obtained is: %d\n", maxValue);

    // Print the included items
    printf("Items included in the knapsack:\n");
    for (int i = 0; i < n; i++) {
        if (includedItems[i]) {
            printf("Item %d (Weight: %d, Value: %d)\n", i + 1, weights[i], values[i]);
        }
    }

    return 0;
}
```

OUTPUT:

```
urscheurjala/5th sem/DAA via C v14.2.1-gcc took 47s
❯ gcc knapsack_backtrack.c

urscheurjala/5th sem/DAA via C v14.2.1-gcc
❯ ./a.out
Name: Urjala Pariyar
Roll no.: 24
-----------------------------------------------
Enter the number of items: 6
Enter the weights of the items:
2 3 5 7 1 4
Enter the values of the items:
10 5 15 7 6 18
Enter the capacity of the knapsack: 15
The maximum value that can be obtained is: 54
Items included in the knapsack:
Item 1 (Weight: 2, Value: 10)
Item 2 (Weight: 3, Value: 5)
Item 3 (Weight: 5, Value: 15)
Item 5 (Weight: 1, Value: 6)
Item 6 (Weight: 4, Value: 18)
```

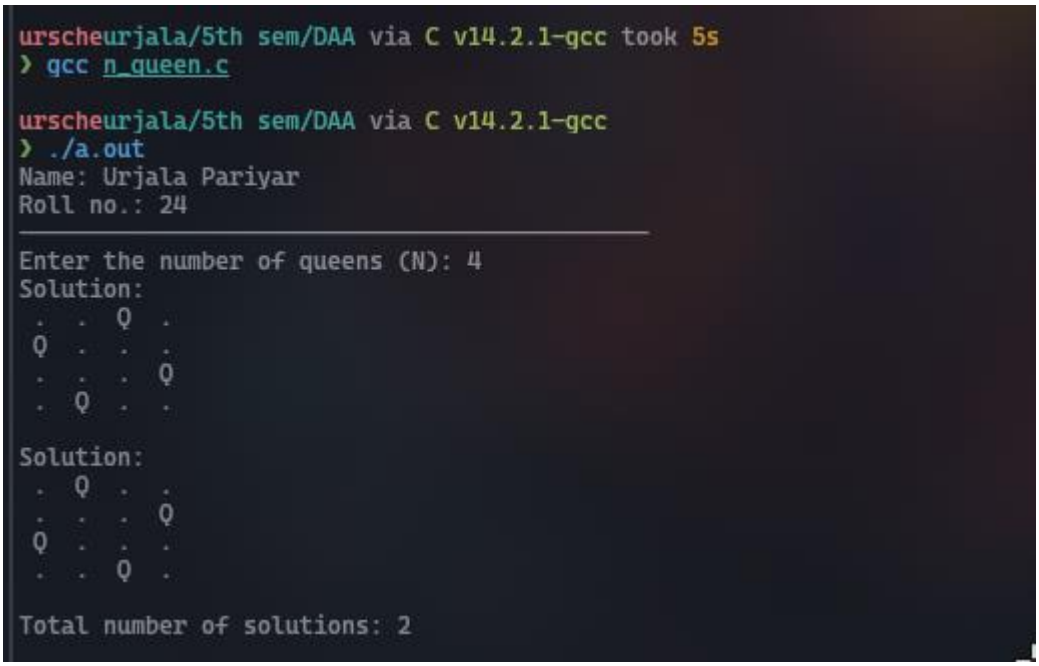
Implementation of N-Queen Problem using bactracking

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

#define MAX_N 15  // Define the maximum size of the board

// Function to print the chessboard with queens placed
void printSolution(int board[MAX_N][MAX_N], int N) {
    printf("Solution:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i][j] == 1) {
                printf(" Q ");
            } else {
                printf(" . ");
            }
        }
        printf("\n");
    }
    printf("\n");
}

// Function to check if a queen can be placed at board[row][col]
bool isSafe(int board[MAX_N][MAX_N], int row, int col, int N) {
    // Check this row on the left side
    for (int i = 0; i < col; i++) {
        if (board[row][i] == 1) {
            return false;
        }
    }

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

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

    return true;
```

```
}

// Recursive function to solve the N-Queens problem and collect all solutions
void solveNQueens(int board[MAX_N][MAX_N], int col, int N, int *solutionCount) {
    // Base case: If all queens are placed
    if (col >= N) {
        printSolution(board, N);
        (*solutionCount)++;
        return;
    }

    // Try placing this queen in all rows in the current column
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col, N)) {
            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place the rest of the queens
            solveNQueens(board, col + 1, N, solutionCount);

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

int main() {
printf("Name: Urjala Pariyar\n");
    printf("Roll no.: 24\n");
    printf("-----------------------------------------\n");
    int N;
    int board[MAX_N][MAX_N] = {0}; // Initialize the board with 0s
    int solutionCount = 0; // To count the number of solutions

    // Input the size of the board (N)
    printf("Enter the number of queens (N): ");
    scanf("%d", &N);

    // Check if the size is within the acceptable range
    if (N <= 0 || N > MAX_N) {
        printf("The number of queens must be between 1 and %d.\n", MAX_N);
        return 1;
    }

    // Solve the N-Queens problem and count all solutions
    solveNQueens(board, 0, N, &solutionCount);

    if (solutionCount == 0) {
```

```
        printf("No solution exists for N = %d.\n", N);
    } else {
        printf("Total number of solutions: %d\n", solutionCount);
    }
    return 0;
}
```

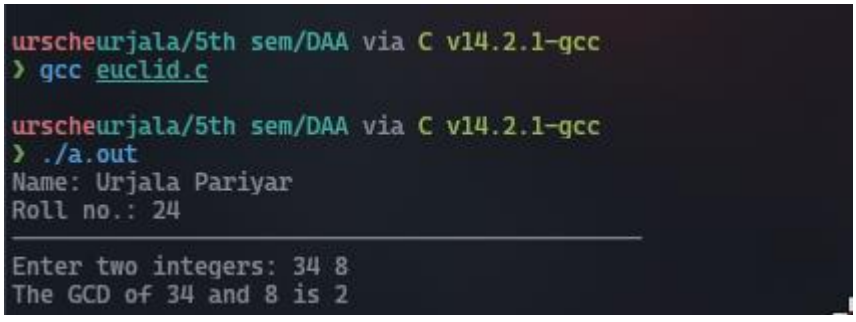
OUTPUT:



Implementation of Euclid's Algorithm

```
#include <stdio.h>

// Function to compute GCD using the Euclidean algorithm
int gcd(int a, int b) {
    // Ensure a is greater than or equal to b
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

int main() {
printf("Name: Urjala Pariyar\n");
    printf("Roll no.: 24\n");
    printf("-------------------------------------------\n");
    int a, b;

    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);

    printf("The GCD of %d and %d is %d\n", a, b, gcd(a, b));

    return 0;
}
```

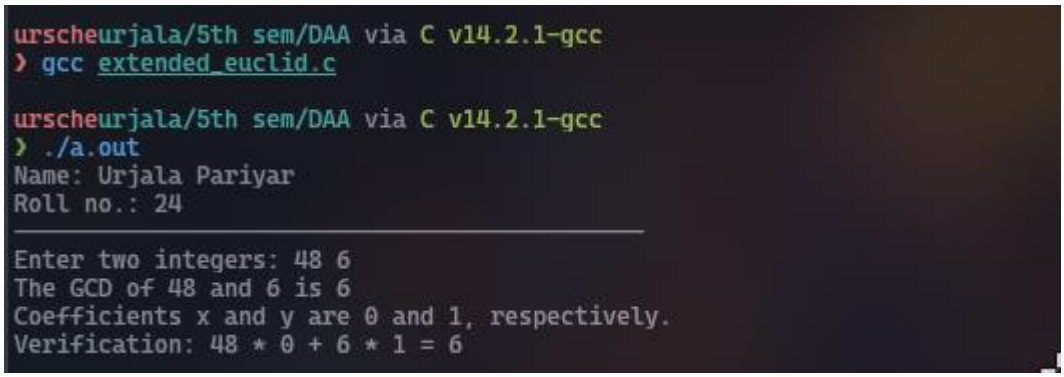
OUTPUT:



Implementation of Extended Euclid's Algorithm

```
#include <stdio.h>
// Function to perform the Extended Euclidean Algorithm
int extendedGCD(int a, int b, int *x, int *y) {
    // Base case: gcd(a, b) = a if b is 0
    if (b == 0) {
        *x = 1;
        *y = 0;
        return a;
    }
    int x1, y1;
    int gcd = extendedGCD(b, a % b, &x1, &y1);
    // Update x and y using results of recursive call
    *x = y1;
    *y = x1 - (a / b) * y1;
    return gcd;
}
int main() {
printf("Name: Urjala Pariyar\n");
    printf("Roll no.: 24\n");
    printf("-----------------------------------------\n");
    int a, b, x, y;
    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);
    int gcd = extendedGCD(a, b, &x, &y);
    printf("The GCD of %d and %d is %d\n", a, b, gcd);
    printf("Coefficients x and y are %d and %d, respectively.\n", x, y);
    printf("Verification: %d * %d + %d * %d = %d\n", a, x, b, y, gcd);
    return 0;
}
```

OUTPUT:

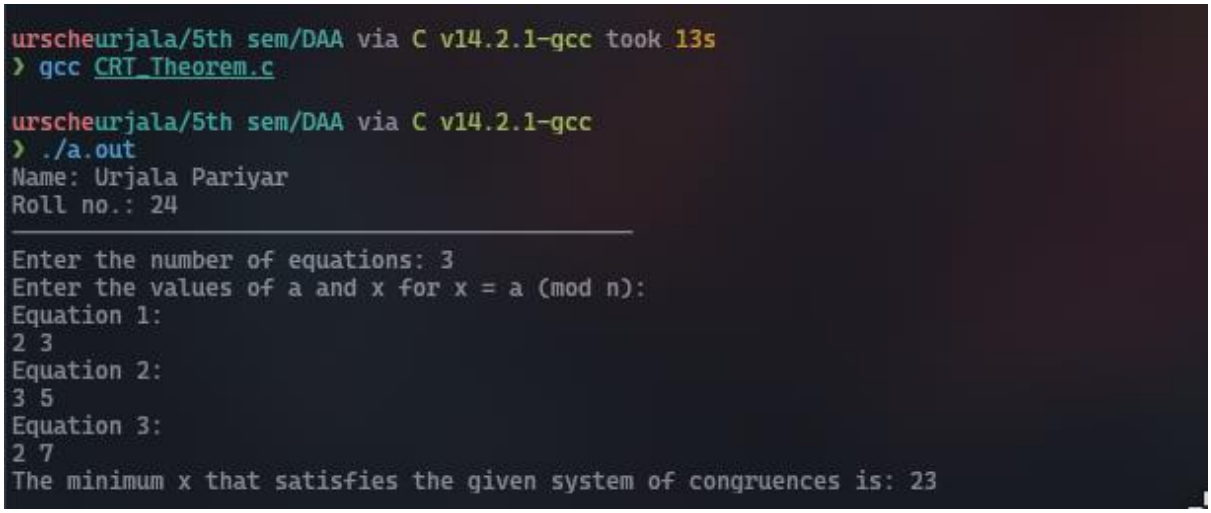


Implementation of CRT Theorem

```
#include <stdio.h>
// Function to find gcd of two numbers
int gcd(int a, int b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}
// Function to find the modular inverse using the Extended Euclidean algorithm
int modInverse(int a, int m) {
    int m0 = m, t, q;
    int x0 = 0, x1 = 1;
    if (m == 1)
        return 0;
    // Apply the extended Euclidean algorithm
    while (a > 1) {
        q = a / m;
        t = m;
        m = a % m;
        a = t;
        t = x0;

        x0 = x1 - q * x0;
        x1 = t;
    }
    // Make x1 positive
    if (x1 < 0)
        x1 += m0;

    return x1;
}

// Function to find the smallest x that satisfies the given system of congruences
int findMinX(int num[], int rem[], int k) {
    int prod = 1;
    int result = 0;
    // Calculate product of all numbers
    for (int i = 0; i < k; i++)
        prod *= num[i];
    // Apply the Chinese Remainder Theorem
    for (int i = 0; i < k; i++) {
        int pp = prod / num[i];
        result += rem[i] * modInverse(pp, num[i]) * pp;
    }

    return result % prod;
```

```
}

int main() {
printf("Name: Urjala Pariyar\n");
   printf("Roll no.: 24\n");
   printf("-----------------------------------------\n");
   int num[10], rem[10], k;

   // Input number of equations
   printf("Enter the number of equations: ");
   scanf("%d", &k);

   // Input values for num[i] and rem[i]
   printf("Enter the values of a and x for x = a (mod n):\n");
   for (int i = 0; i < k; i++) {
      printf("Equation %d:\n", i + 1);
      scanf("%d", &rem[i]);
      scanf("%d", &num[i]);

   }
   // Calculate and print the minimum x
   int x = findMinX(num, rem, k);
   printf("The minimum x that satisfies the given system of congruences is: %d\n", x);
   return 0;
}
```

OUTPUT:

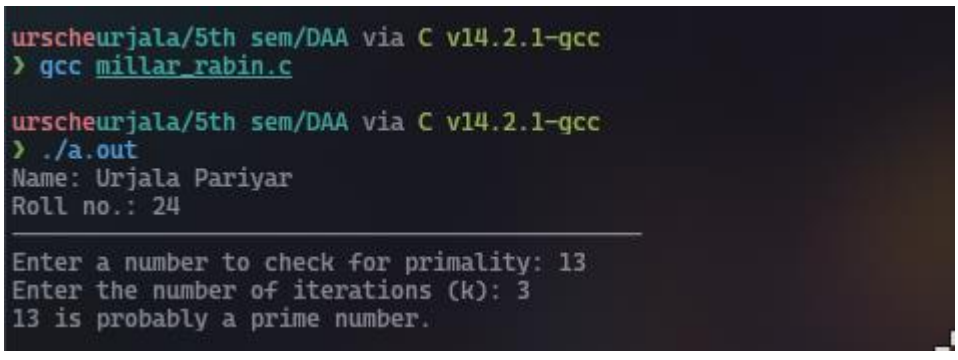


Implementation of Miller Rabin Primality Test Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h> // Include this for bool, true, false types

// Function to perform modular exponentiation
// It returns (base^exp) % mod
long long modularExponentiation(long long base, long long exp, long long mod) {
    long long result = 1;
    base = base % mod;

    while (exp > 0) {
        if (exp % 2 == 1) // If exp is odd, multiply base with the result
            result = (result * base) % mod;

        exp = exp >> 1;   // exp = exp / 2
        base = (base * base) % mod;
    }
    return result;
}
// Function to perform the Miller test
// This function returns false if n is composite and true if n is probably prime
bool millerTest(long long d, long long n) {
    // Pick a random number a in the range [2, n-2]
    long long a = 2 + rand() % (n - 4);

    // Compute a^d % n
    long long x = modularExponentiation(a, d, n);

    if (x == 1 || x == n - 1)
        return true;

    // Keep squaring x while one of the conditions is met
    while (d != n - 1) {
        x = (x * x) % n;
        d *= 2;

        if (x == 1)
            return false;
        if (x == n - 1)
            return true;
    }

    return false;
}
// Function to check if a number is prime using the Miller-Rabin test
```

```
bool isPrime(long long n, int k) {
    // Handle base cases
    if (n <= 1 || n == 4)
        return false;
    if (n <= 3)
        return true;

    // Find d such that n-1 = d * 2^r, where d is odd
    long long d = n - 1;
    while (d % 2 == 0)
        d /= 2;

    // Iterate k times
    for (int i = 0; i < k; i++)
        if (!millerTest(d, n))
            return false;

    return true;
}

int main() {
printf("Name: Urjala Pariyar\n");
    printf("Roll no.: 24\n");
    printf("-----------------------------------------------\n");
    long long n;
    int k;
    // Input the number to check for primality
    printf("Enter a number to check for primality: ");
    scanf("%lld", &n);
    // Input the number of iterations
    printf("Enter the number of iterations (k): ");
    scanf("%d", &k);
    // Perform the Miller-Rabin primality test
    if (isPrime(n, k))
        printf("%lld is probably a prime number.\n", n);
    else
        printf("%lld is not a prime number.\n", n);
    return 0;
}
```

OUTPUT:



Implementation of Vertex Cover problem using Approximation algorithm

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

#define MAX_VERTICES 100

// Function to find the Vertex Cover using the approximation algorithm
void vertexCover(int graph[MAX_VERTICES][MAX_VERTICES], int V) {
    bool visited[MAX_VERTICES] = {false};  // Array to mark visited vertices

    for (int u = 0; u < V; u++) {
        // If the vertex u is not yet visited
        if (!visited[u]) {
            // Check all adjacent vertices of u
            for (int v = 0; v < V; v++) {
                // If there's an edge from u to v and v is not visited
                if (graph[u][v] && !visited[v]) {
                    // Include both u and v in the vertex cover
                    visited[u] = true;
                    visited[v] = true;
                    break;  // Move to the next vertex u
                }
            }
        }
    }

    // Print the Vertex Cover
    printf("The Vertex Cover is: ");
    for (int i = 0; i < V; i++) {
        if (visited[i]) {
            printf("%d ", i);
        }
    }
    printf("\n");
}

int main() {
printf("Name: Urjala Pariyar\n");
    printf("Roll no.: 24\n");
    printf("-----------------------------------------------\n");
    int V;  // Number of vertices
    printf("Enter the number of vertices: ");
    scanf("%d", &V);

    int graph[MAX_VERTICES][MAX_VERTICES];
```

```
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    vertexCover(graph, V);
    return 0;
}
```

OUTPUT: