1. Implement 0/1 Knapsack algorithm.

```
#include <stdio.h>
int max(int a, int b) {
   return (a > b) ? a : b;
}
int knapsack(int max_weight, int weights[], int values[], int num_items) {
   int dp[num_items + 1][max_weight + 1];
   for (int i = 0; i<= num_items; i++) {
      for (int w = 0; w <= max_weight; w++) {
         if (i == 0 || w == 0)
dp[i][w] = 0;
         else if (weights[i - 1] <= w)
dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
         else
dp[i][w] = dp[i - 1][w];
      }
   }
   return dp[num_items][max_weight];
}
int main() {
   int num_items;
printf("Enter the number of items: ");
scanf("%d", &num_items);
   int weights[num_items];
   int values[num_items];
printf("Enter the weight and value of each item:\n");
   for (int i = 0; i<num_items; i++) {
printf("Item %d:\n", i + 1);
scanf("%d", &weights[i]);
scanf("%d", &values[i]);
   }
   int max_weight;
printf("Enter the maximum weight: ");
scanf("%d", &max_weight);
   int max_value = knapsack(max_weight, weights, values, num_items);
printf("Maximum value: %d\n", max_value);
   return 0;
}
```

2. Alice and Bob:

```
#include <stdio.h>
#include <stdbool.h>
bool canWin(int n) {
    if (n <= 1) {
        return false;
    }
    for (int x = 1; x < n; x++) {
        if (n % x == 0) {
            if (!canWin(n - x)) {
                return true;
            }
        }
    }
    return false;
}
int main() {
    int n;
printf("Enter the initial number: ");
scanf("%d", &n);
    bool aliceWins = canWin(n);
    if (aliceWins) {
printf("true");
    } else {
printf("false");
    }
    return 0;
}
```

3. Implement Matrix Chain multiplication algorithm with top-down approach.

```
#include <stdio.h>
#include <limits.h>
#define MAX_SIZE 100
int matrixChainMultiplication(int dimensions[], int i, int j, int dp[][MAX_SIZE]) {
  if (i == j) {
    return 0;
  }
  if (dp[i][j] != -1) {
    return dp[i][j];
  }
dp[i][j] = INT_MAX;
  for (int k = i; k < j; k++) {
    int cost = matrixChainMultiplication(dimensions, i, k, dp) +
matrixChainMultiplication(dimensions, k + 1, j, dp) +
          dimensions[i - 1] * dimensions[k] * dimensions[j];
    if (cost <dp[i][j]) {
dp[i][j] = cost;
    }
  }
  return dp[i][j];
}
int main() {
  int numMatrices;
printf("Enter the number of matrices: ");
scanf("%d", &numMatrices);
  int dimensions[numMatrices + 1];
printf("Enter the dimensions of the matrices:\n");
  for (int i = 0; i<= numMatrices; i++) {
scanf("%d", &dimensions[i]);
  }
  int dp[MAX_SIZE][MAX_SIZE];
  for (int i = 0; i< MAX_SIZE; i++) {
    for (int j = 0; j < MAX_SIZE; j++) {
dp[i][j] = -1;
    }
  }
  int minimumCost = matrixChainMultiplication(dimensions, 1, numMatrices, dp);
printf("Minimum number of multiplications: %d\n", minimumCost);
  return 0;
}
```

4. Write a program to find minimum change to return when unlimited number of denominations are available using Dynamic programming.

```
#include <stdio.h>
#include <limits.h>
int minCoins(int coins[], int numCoins, int amount) {
   int dp[amount + 1];
dp[0] = 0;
   for (int i = 1; i<= amount; i++) {
dp[i] = INT_MAX;
   }
   for (int i = 1; i<= amount; i++) {
      for (int j = 0; j <numCoins; j++) {
         if (coins[j] <= i) {
            int subproblem = dp[i - coins[j]];
            if (subproblem != INT_MAX && subproblem + 1 <dp[i]) {
dp[i] = subproblem + 1;
            }
         }
      }
   }
   return dp[amount];
}
int main() {
   int numCoins;
printf("Enter the number of coin denominations: ");
scanf("%d", &numCoins);
   int coins[numCoins];
printf("Enter the coin denominations:\n");
   for (int i = 0; i<numCoins; i++) {
scanf("%d", &coins[i]);
   }
   int amount;
printf("Enter the amount for which to make change: ");
scanf("%d", &amount);
   int minNumCoins = minCoins(coins, numCoins, amount);
printf("Minimum number of coins needed: %d\n", minNumCoins);
   return 0;
}
```

5. Implement the LCS problem using dynamic programming

```
#include <stdio.h>
#include <string.h>
#define MAX_SIZE 100
int max(int a, int b) {
    return (a > b) ? a : b;
}
int lcs(char* str1, char* str2, int len1, int len2) {
    int dp[MAX_SIZE + 1][MAX_SIZE + 1];
    for (int i = 0; i<= len1; i++) {
        for (int j = 0; j <= len2; j++) {
            if (i == 0 || j == 0)
dp[i][j] = 0;
            else if (str1[i - 1] == str2[j - 1])
dp[i][j] = dp[i - 1][j - 1] + 1;
            else
dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
    return dp[len1][len2];
}
int main() {
    char str1[MAX_SIZE];
    char str2[MAX_SIZE];
printf("Enter the first string: ");
scanf("%s", str1);
printf("Enter the second string: ");
scanf("%s", str2);
    int len1 = strlen(str1);
    int len2 = strlen(str2);
    int lcsLength = lcs(str1, str2, len1, len2);
printf("Length of Longest Common Subsequence: %d\n", lcsLength);
    return 0;
}
```

6. Implement a program to find longest increasing subsequence.

```
#include <stdio.h>
#include <stdlib.h>
int lis(int arr[], int n) {
   int* dp = (int*)malloc(sizeof(int) * n);
   int maxLen = 0;
   for (int i = 0; i< n; i++) {
dp[i] = 1;
   }
   for (int i = 1; i< n; i++) {
      for (int j = 0; j <i; j++) {
         if (arr[i] >arr[j] &&dp[i] <dp[j] + 1) {
dp[i] = dp[j] + 1;
         }
      }
   }
   for (int i = 0; i< n; i++) {
      if (dp[i] >maxLen) {
maxLen = dp[i];
      }
   }
   free(dp);
   return maxLen;
}
int main() {
   int n;
printf("Enter the number of elements in the array: ");
scanf("%d", &n);
   int arr[n];
printf("Enter the elements of the array:\n");
   for (int i = 0; i< n; i++) {
scanf("%d", &arr[i]);
   }
   int length = lis(arr, n);
printf("Length of Longest Increasing Subsequence: %d\n", length);
   return 0;
}
```

7. Implement Matrix Chain multiplication algorithm with bottom-up approach.

```
#include <stdio.h>
#include <limits.h>
#define MAX_SIZE 100
int min(int a, int b) {
   return (a < b) ? a : b;
}
int matrixChainMultiplication(int dimensions[], int numMatrices) {
   int dp[MAX_SIZE][MAX_SIZE];
   for (int i = 0; i<= numMatrices; i++) {
dp[i][i] = 0;
   }
   for (int length = 2; length <= numMatrices; length++) {
      for (int i = 1; i<= numMatrices - length + 1; i++) {
         int j = i + length - 1;
dp[i][j] = INT_MAX;
         for (int k = i; k < j; k++) {
            int cost = dp[i][k] + dp[k + 1][j] +
                     dimensions[i - 1] * dimensions[k] * dimensions[j];
dp[i][j] = min(dp[i][j], cost);
         }
      }
   }
   return dp[1][numMatrices];
}
int main() {
   int numMatrices;
printf("Enter the number of matrices: ");
scanf("%d", &numMatrices);
   int dimensions[numMatrices + 1];
printf("Enter the dimensions of the matrices:\n");
   for (int i = 0; i<= numMatrices; i++) {
scanf("%d", &dimensions[i]);
   }
   int minimumCost = matrixChainMultiplication(dimensions, numMatrices);
printf("Minimum number of multiplications: %d\n", minimumCost);
   return 0;
}
```

8. Write a program to find subset sum by using Dynamic programming

```c
#include <stdio.h>
#include <stdbool.h>
#define MAX_SIZE 100
bool subsetSum(int set[], int n, int sum) {
    bool dp[MAX_SIZE + 1][MAX_SIZE + 1];
    for (int i = 0; i<= n; i++) {
dp[i][0] = true;
    }
    for (int j = 1; j <= sum; j++) {
dp[0][j] = false;
    }
    for (int i = 1; i<= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (set[i - 1] > j) {
dp[i][j] = dp[i - 1][j];
            }
            else {
dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]];
            }
        }
    }
    return dp[n][sum];
}
int main() {
    int n;
printf("Enter the number of elements in the set: ");
scanf("%d", &n);
    int set[n];
printf("Enter the elements of the set:\n");
    for (int i = 0; i< n; i++) {
scanf("%d", &set[i]);
    }
    int sum;
printf("Enter the target sum: ");
scanf("%d", &sum);
    bool exists = subsetSum(set, n, sum);
    if (exists) {
printf("Subset with the given sum exists.\n");
    } else {
printf("No subset with the given sum exists.\n");
    }
    return 0;
}
```

9. Implement of N-queens problem with Back tracking.

```c
#include <stdio.h>
#include <stdbool.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("%c ", board[i][j] ? 'Q' : '.');
        }
printf("\n");
    }
printf("\n");
}
bool isSafe(int board[N][N], int row, int col) {
    int i, j;
    for (i = 0; i< col; i++) {
        if (board[row][i])
            return false;
    }
    for (i = row, j = col; i>= 0 && j >= 0; i--, j--) {
        if (board[i][j])
            return false;
    }
    for (i = row, j = col; i< N && j >= 0; i++, j--) {
        if (board[i][j])
            return false;
    }
    return true;
}
bool solveNQueensUtil(int board[N][N], int col) {
    if (col == N) {
printSolution(board);
        return true;
    }
    bool res = false;
    for (int i = 0; i< N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            res = solveNQueensUtil(board, col + 1) || res;
            board[i][col] = 0;
        }
    }
    return res;
}
void solveNQueens() {
    int board[N][N] = {0};
    if (!solveNQueensUtil(board, 0)) {
printf("No solution found.\n");
    }
}
int main() {
solveNQueens();
    return 0;
}
```

10.Implement Sum of subsets problem by using Backtracking

```
#include <stdio.h>
#include <stdbool.h>
void printSubset(int set[], int subset[], int n) {
printf("Subset: ");
    for (int i = 0; i< n; i++) {
        if (subset[i])
printf("%d ", set[i]);
    }
printf("\n");
}
void subsetSumUtil(int set[], int subset[], int n, int sum, int currentSum, int index) {
    if (currentSum == sum) {
printSubset(set, subset, n);
        return;
    }
    if (index == n)
        return;
    if (currentSum + set[index] <= sum) {
        subset[index] = 1;
subsetSumUtil(set, subset, n, sum, currentSum + set[index], index + 1);
        subset[index] = 0;
    }
subsetSumUtil(set, subset, n, sum, currentSum, index + 1);
}
void subsetSum(int set[], int n, int sum) {
    int subset[n];
subsetSumUtil(set, subset, n, sum, 0, 0);
}
int main() {
    int n;
printf("Enter the number of elements in the set: ");
scanf("%d", &n);
    int set[n];
printf("Enter the elements of the set:\n");
    for (int i = 0; i< n; i++) {
scanf("%d", &set[i]);
    }
    int sum;
printf("Enter the target sum: ");
scanf("%d", &sum);
subsetSum(set, n, sum);
    return 0;
}
```

11. Implement Graph coloring problem with back tracking.

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 20
bool isSafe(int v, int graph[MAX_VERTICES][MAX_VERTICES], int vertices[], int color, int n) {
    for (int i = 0; i< n; i++) {
        if (graph[v][i] && color == vertices[i])
            return false;
    }
    return true;
}
bool graphColoringUtil(int graph[MAX_VERTICES][MAX_VERTICES], int m, int vertices[], int v, int n) {
    if (v == n)
        return true;
    for (int color = 1; color <= m; color++) {
        if (isSafe(v, graph, vertices, color, n)) {
            vertices[v] = color;
            if (graphColoringUtil(graph, m, vertices, v + 1, n))
                return true;
            vertices[v] = 0;
        }
    }
    return false;
}
void graphColoring(int graph[MAX_VERTICES][MAX_VERTICES], int m, int n) {
    int vertices[MAX_VERTICES] = {0};
    if (graphColoringUtil(graph, m, vertices, 0, n)) {
printf("Graph can be colored using at most %d colors.\n", m);
printf("Coloring: ");
        for (int i = 0; i< n; i++) {
printf("%d ", vertices[i]);
        }
printf("\n");
    } else {
printf("Graph cannot be colored using %d colors.\n", m);
    }
}
int main() {
    int n, m;
printf("Enter the number of vertices in the graph: ");
scanf("%d", &n);
printf("Enter the adjacency matrix of the graph:\n");
    int graph[MAX_VERTICES][MAX_VERTICES];
    for (int i = 0; i< n; i++) {
        for (int j = 0; j < n; j++) {
scanf("%d", &graph[i][j]);
        }
    }
printf("Enter the number of colors available: ");
scanf("%d", &m);
graphColoring(graph, m, n);
    return 0;
}
```

12. Implement a program to find Hamiltonian cycle from a given graph

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 20
void printSolution(int path[], int n) {
printf("Hamiltonian Cycle: ");
    for (int i = 0; i< n; i++) {
printf("%d ", path[i]);
    }
printf("%d\n", path[0]);
}
bool isSafe(int v, int graph[MAX_VERTICES][MAX_VERTICES], int path[], int pos, int n) {
    if (graph[path[pos - 1]][v] == 0)
        return false;
    for (int i = 0; i< pos; i++) {
        if (path[i] == v)
            return false;
    }
    return true;
}
bool hamiltonianCycleUtil(int graph[MAX_VERTICES][MAX_VERTICES], int path[], int pos, int n) {
    if (pos == n) {
        if (graph[path[pos - 1]][path[0]] == 1)
            return true;
        else
            return false;
    }
    for (int v = 1; v < n; v++) {
        if (isSafe(v, graph, path, pos, n)) {
            path[pos] = v;
            if (hamiltonianCycleUtil(graph, path, pos + 1, n))
                return true;
            path[pos] = -1;  // Backtrack
        }
    }
    return false;
}
void hamiltonianCycle(int graph[MAX_VERTICES][MAX_VERTICES], int n) {
    int path[MAX_VERTICES];
    for (int i = 0; i< n; i++) {
        path[i] = -1;
    }
    path[0] = 0;
    if (hamiltonianCycleUtil(graph, path, 1, n)) {
printSolution(path, n);
    } else {
printf("No Hamiltonian Cycle found.\n");
    }
}
int main() {
    int n;
printf("Enter the number of vertices in the graph: ");
scanf("%d", &n);
printf("Enter the adjacency matrix of the graph:\n");
    int graph[MAX_VERTICES][MAX_VERTICES];
    for (int i = 0; i< n; i++) {
        for (int j = 0; j < n; j++) {
scanf("%d", &graph[i][j]);
        }
    } hamiltonianCycle(graph, n); return 0;}
```

13.Implement TCS by branch and bound:

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_N 10
int n;
int graph[MAX_N][MAX_N];
int minCost = INT_MAX;
int bestPath[MAX_N];
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int calculatePathCost(int path[]) {
    int cost = 0;
    for (int i = 0; i< n - 1; i++) {
        cost += graph[path[i]][path[i + 1]];
    }
    cost += graph[path[n - 1]][path[0]];
    return cost;
}
void TSPBranchAndBoundUtil(int path[], bool visited[], int level, int cost) {
    if (level == n) {
        int currentCost = cost + graph[path[level - 1]][path[0]];
        if (currentCost<minCost) {
minCost = currentCost;
            for (int i = 0; i< n; i++) {
bestPath[i] = path[i];
            }
        }
        return;
    }
    for (int i = 0; i< n; i++) {
        if (!visited[i]) {
            path[level] = i;
            visited[i] = true;
            int newCost = cost + graph[path[level - 1]][i];
            int lowerBound = 0;
            for (int j = 0; j < n; j++) {
                if (!visited[j]) {
                    int minEdgeCost = INT_MAX;
                    for (int k = 0; k < n; k++) {
                        if (graph[j][k] <minEdgeCost&& j != k) {
minEdgeCost = graph[j][k];
                        }
                    }
lowerBound += minEdgeCost;
                }
            }
            if (newCost + lowerBound<minCost) {
TSPBranchAndBoundUtil(path, visited, level + 1, newCost);
            }
            visited[i] = false;
        }
    }
}
void TSPBranchAndBound(int startingCity) {
    int path[MAX_N];
    bool visited[MAX_N];
    for (int i = 0; i< n; i++) {
        visited[i] = false;
    }
    path[0] = startingCity;
    visited[startingCity] = true;
TSPBranchAndBoundUtil(path, visited, 1, 0);
printf("Optimal TSP Path: ");
    for (int i = 0; i< n; i++) {
printf("%d ", bestPath[i]);
    }
printf("%d\n", bestPath[0]);
printf("Optimal TSP Cost: %d\n", minCost);
}
int main() {
printf("Enter the number of cities: ");
scanf("%d", &n);
printf("Enter the adjacency matrix of distances between cities:\n");
    for (int i = 0; i< n; i++) {
        for (int j = 0; j < n; j++) {
scanf("%d", &graph[i][j]);
        }
    }
    int startingCity;
printf("Enter the starting city (0-%d): ", n - 1);
scanf("%d", &startingCity);
TSPBranchAndBound(startingCity);
    return 0;
}
```

14.Implement the Dijkstra's single source shortest paths algorithm.

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_VERTICES 20
int graph[MAX_VERTICES][MAX_VERTICES];
int n;
void dijkstra(int source) {
    int dist[MAX_VERTICES];
    bool visited[MAX_VERTICES];
    int parent[MAX_VERTICES];

    for (int i = 0; i< n; i++) {
dist[i] = INT_MAX;
        visited[i] = false;
        parent[i] = -1;
    }
dist[source] = 0;
    for (int count = 0; count < n - 1; count++) {
        int minDist = INT_MAX, minDistVertex;
        for (int v = 0; v < n; v++) {
            if (!visited[v] &&dist[v] <minDist) {
minDist = dist[v];
minDistVertex = v;
            }
        }
        visited[minDistVertex] = true;
        for (int v = 0; v < n; v++) {
            if (!visited[v] && graph[minDistVertex][v] != 0 &&dist[minDistVertex] != INT_MAX &&
dist[minDistVertex] + graph[minDistVertex][v] <dist[v]) {
dist[v] = dist[minDistVertex] + graph[minDistVertex][v];
                parent[v] = minDistVertex;
            }
        }
    }
printf("Vertex\tDistance\tPath\n");
    for (int v = 0; v < n; v++) {
printf("%d\t%d\t\t%d", v, dist[v], v);
        int p = parent[v];
        while (p != -1) {
printf(" <- %d", p);
            p = parent[p];
        }
printf("\n");
    }
}
int main() {
printf("Enter the number of vertices in the graph: ");
scanf("%d", &n);
printf("Enter the adjacency matrix of the graph (0 for no edge, positive weight for edge):\n");
    for (int i = 0; i< n; i++) {
        for (int j = 0; j < n; j++) {
scanf("%d", &graph[i][j]);
        }
    }
    int source;
printf("Enter the source vertex: ");
scanf("%d", &source);
dijkstra(source);
    return 0;
}
```

15. Implement 0/1 knapsack by branch and bound.

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_ITEMS 100
typedef struct {
    int weight;
    int value;
} Item;
int maxProfit = 0;
int bestItems[MAX_ITEMS];
int numItems;
int capacity;
void branchAndBoundKnapsack(Item items[], int level, int currentWeight, int currentValue, bool selected[]) {
    if (currentWeight> capacity) {
        return;
    }
    if (currentValue>maxProfit) {
maxProfit = currentValue;
        for (int i = 0; i<numItems; i++) {
bestItems[i] = selected[i];
        }
    }
    if (level == numItems) {
        return;
    }
    double bound = currentValue;
    int remainingWeight = capacity - currentWeight;
    int i = level;
    while (i<numItems&&remainingWeight> 0) {
        if (items[i].weight <= remainingWeight) {
            bound += items[i].value;
remainingWeight -= items[i].weight;
        } else {
            bound += (double)items[i].value / items[i].weight * remainingWeight;
remainingWeight = 0;
        }
i++;
    }
    if (bound <= maxProfit) {
        return;
    }
    selected[level] = true;
branchAndBoundKnapsack(items, level + 1, currentWeight + items[level].weight, currentValue +
items[level].value, selected);
    selected[level] = false;
branchAndBoundKnapsack(items, level + 1, currentWeight, currentValue, selected);
}
void knapsack(Item items[], int num, int cap) {
numItems = num;
    capacity = cap;
    bool selected[MAX_ITEMS] = { false };
branchAndBoundKnapsack(items, 0, 0, 0, selected);
printf("Optimal Items: ");
    for (int i = 0; i<numItems; i++) {
        if (bestItems[i]) {
printf("%d ", i);
        }
    }
printf("\n");
printf("Optimal Profit: %d\n", maxProfit);
}
int main() {
    int num, cap;
printf("Enter the number of items: ");
scanf("%d", &num);
printf("Enter the capacity of the knapsack: ");
scanf("%d", &cap);
    Item items[MAX_ITEMS];
printf("Enter the weight and value of each item:\n");
    for (int i = 0; i<num; i++) {
scanf("%d %d", &items[i].weight, &items[i].value);
    }
    knapsack(items, num, cap);
    return 0;
}
```

16. Implement Bellman ford single source shortest paths algorithm

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_VERTICES 20
#define MAX_EDGES 100
typedef struct {
    int source;
    int destination;
    int weight;
} Edge;
int numVertices, numEdges;
Edge edges[MAX_EDGES];
int distances[MAX_VERTICES];
void bellmanFord(int source) {
    for (int i = 0; i<numVertices; i++) {
        if (i == source) {
            distances[i] = 0;
        } else {
            distances[i] = INT_MAX;
        }
    }
    for (int i = 0; i<numVertices - 1; i++) {
        for (int j = 0; j <numEdges; j++) {
            int u = edges[j].source;
            int v = edges[j].destination;
            int weight = edges[j].weight;
            if (distances[u] != INT_MAX && distances[u] + weight < distances[v]) {
                distances[v] = distances[u] + weight;
            }
        }
    }
    for (int i = 0; i<numEdges; i++) {
        int u = edges[i].source;
        int v = edges[i].destination;
        int weight = edges[i].weight;
        if (distances[u] != INT_MAX && distances[u] + weight < distances[v]) {
printf("Negative-weight cycle detected. The graph contains a negative-weight cycle.\n");
            return;
        }
    }
printf("Vertex\tDistance from Source\n");
    for (int i = 0; i<numVertices; i++) {
printf("%d\t%d\n", i, distances[i]);
    }
}
int main() {
printf("Enter the number of vertices: ");
scanf("%d", &numVertices);
printf("Enter the number of edges: ");
scanf("%d", &numEdges);
printf("Enter the edges (source, destination, weight):\n");
    for (int i = 0; i<numEdges; i++) {
scanf("%d %d %d", &edges[i].source, &edges[i].destination, &edges[i].weight);
    }
    int source;
printf("Enter the source vertex: ");
scanf("%d", &source);
bellmanFord(source); return 0;}
```