9. Write a program to solve 0/1 Knapsack problem Using Dynamic Programming

```
#include <stdio.h>
int max(int a, int b) {
    return (a > b) ? a : b;
}
void knapsackDP(int W, int wt[], int val[], int n) {
    int i, w;
    int dp[n + 1][W + 1];
  // Build table dp[][]
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (wt[i - 1] <= w)
                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];
```

```
        }
    }
printf("Maximum value: %d\n", dp[n][W]);
}
int main() {
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
     knapsackDP(W, wt, val, n);
    return 0;
}
```

Output:

```
Maximum value: 220
```

10. Implement N-Queens Problem Using Backtracking.

```
#include <stdio.h>
#define N 4
```

```
void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}
int isSafe(int board[N][N], int row, int col) {
    int i, j;
     for (i = 0; i < col; i++)
        if (board[row][i])
            return 0;
     for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return 0;
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return 0;
 return 1;
}
```

```
int solveNQUtil(int board[N][N], int col) {
    if (col >= N)
        return 1;
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
 if (solveNQUtil(board, col + 1))
                return 1;
board[i][col] = 0;
        }
    }
 return 0;
}
int solveNQ() {
    int board[N][N] = { {0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0} };

    if (solveNQUtil(board, 0) == 0) {
```

```
        printf("Solution does not exist");
        return 0;
    }
 printSolution(board);
    return 1;
}
int main() {
    solveNQ();
    return 0;
}
```

Output:

```
0  0  1  0
 1  0  0  0
 0  0  0  1
 0  1  0  0
```

11. Use Backtracking strategy to solve 0/1 Knapsack problem.

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

```
int max(int a, int b) {
    return (a > b) ? a : b;
}
int knapsackBacktracking(int W, int wt[], int val[], int n)
{
    if (n == 0 || W == 0)
        return 0;
 if (wt[n - 1] > W)
        return knapsackBacktracking(W, wt, val, n - 1);
 return max(val[n - 1] + knapsackBacktracking(W - wt[n
- 1], wt, val, n - 1),
               knapsackBacktracking(W, wt, val, n - 1));
}
int main() {
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
 printf("Maximum value: %d\n",
knapsackBacktracking(W, wt, val, n));
    return 0;
```

```
}
```

Output:

Maximum value: 220

12. Implement Travelling Sales Person problem using Branch and Bound approach.

```
#include <stdio.h>
#include <limits.h>
#define N 4
int tsp(int graph[][N], int pos, int visited, int dp[][1 <<
N]) {
    if (visited == (1 << N) - 1)
        return graph[pos][0];
 if (dp[pos][visited] != -1)
        return dp[pos][visited];
 int answer = INT_MAX;
 for (int i = 0; i < N; i++) {
        if ((visited & (1 << i)) == 0) {
```

```
            int newAnswer = graph[pos][i] + tsp(graph, i,
visited | (1 << i), dp);
            answer = (answer < newAnswer) ? answer :
newAnswer;
        }
    }
 return dp[pos][visited] = answer;
}
int main() {
    int graph[N][N] = { {0, 10, 15, 20},
                        {10, 0, 35, 25},
                        {15, 35, 0, 30},
                        {20, 25, 30, 0} };
 int dp[N][1 << N];
    for (int i = 0; i < N; i++)
        for (int j = 0; j < (1 << N); j++)
            dp[i][j] = -1;
 printf("Minimum cost: %d\n", tsp(graph, 0, 1, dp));
    return 0;
}
```

Output:

Minimum cost: 80