Name.: Suryansh Sandeep Kumar Singh
Roll No.: S1951122
PRN: 72018560C
DIV: B (Batch B2)

Assignment No : 1

Title : Fractional knapsack using Greedy algorithm and 0/1 knapsack using dynamic programming

Problem Statement: Write a program to implement Fractional knapsack using Greedy algorithm and 0/1 knapsack using dynamic programming. Show that Greedy strategy does not necessarily yield an optimal solution over a dynamic programming approach.

Objective: Greedy strategy does not necessarily yield an optimal solution over a dynamic programming approach.

Theory:

1) Fractional knapsack using Greedy algorithm

Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In **Fractional Knapsack**, we can break items for maximizing the total value of knapsack. This problem in which we can break an item is also called the fractional knapsack problem.

A **brute-force solution** would be to try all possible subset with all different fraction but that will be too much time taking.

An **efficient solution** is to use Greedy approach. The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.
A simple code with our own comparison function can be written as follows, please see sort function more closely, the third argument to sort function is our comparison function which sorts the item according to value/weight ratio in non-decreasing order.
After sorting we need to loop over these items and add them in our knapsack satisfying above-mentioned criteria.

Input :
Items as (value, weight) pairs
arr[] = {{60, 10}, {100, 20}, {120, 30}}
Knapsack Capacity, W = 50;
Output :
Maximum possible value = 240
By taking full items of 10 kg, 20 kg and
2/3rd of last item of 30 kg

Below is the implementation of the above idea:

Code :

```
#include <stdio.h>

void main()

{

    int capacity = 50;

    int value[] = {60, 100, 120};

    int weight[] = {10, 20, 30};

    int no_items = 3, cur_weight, item;

    int used[10];

    float total_profit;

    int i;

    for (i = 0; i < no_items; ++i)

        used[i] = 0;

    cur_weight = capacity;

    while (cur_weight > 0)

    {

        item = -1;

        for (i = 0; i < no_items; ++i)
            if ((used[i] == 0) && ((item == -1) || ((float)value[i] / weight[i] > (float)value[item] /
weight[item])))

                item = i;

        used[item] = 1;

        cur_weight -= weight[item];

        total_profit += value[item];

        if (cur_weight >= 0)

            printf("Added object %d (%d Rs., %dKg) completely in the bag. Space left: %d.\n", item
+ 1, value[item], weight[item], cur_weight);

        else
        {

            int item_percent = (int)((1 + (float)cur_weight / weight[item]) * 100);

            printf("Added %d%% (%d Rs., %dKg) of object %d in the bag.\n",
item_percent,value[item], weight[item], item + 1);

            total_profit -= value[item];

            total_profit += (1 + (float)cur_weight / weight[item]) * value[item];

        }

    }

    printf("Filled the bag with objects worth %.2f Rs.\n", total_profit);

}
```

Theory :

2) 0/1 knapsack using dynamic programming

In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach. In a DP[][] table let's consider all the possible weights from '1' to 'W' as the columns and weights that can be kept as the rows.
The state DP[i][j] will denote maximum value of 'j-weight' considering all values from '1 to ith'. So if we consider 'wi' (weight in 'ith' row) we can fill it in all columns which have 'weight values > wi'. Now two possibilities can take place:

- Fill 'wi' in the given column.
- Do not fill 'wi' in the given column.

Now we have to take a maximum of these two possibilities, formally if we do not fill 'ith' weight in 'jth' column then DP[i][j] state will be same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of 'wi'+ value of the column weighing 'j-wi' in the previous row. So we take the maximum of these two possibilities to fill the current state. This visualization will make the concept clear:

Let weight elements = {1, 2, 3}
Let weight values = {10, 15, 40}
Capacity=6

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0						

Explanation:

For filling 'weight = 2' we come
across 'j = 3' in which
we take maximum of
(10, 15 + DP[1][3-2]) = 25
 | |
'2' '2 filled'
not filled

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

Explanation:

For filling 'weight=3',
we come across 'j=4' in which
we take maximum of (25, 40 + DP[2][4-3])
= 50

For filling 'weight=3'
we come across 'j=5' in which
we take maximum of (25, 40 + DP[2][5-3])
= 55

For filling 'weight=3'
we come across 'j=6' in which
we take maximum of (25, 40 + DP[2][6-3])
= 65

Complexity Analysis:

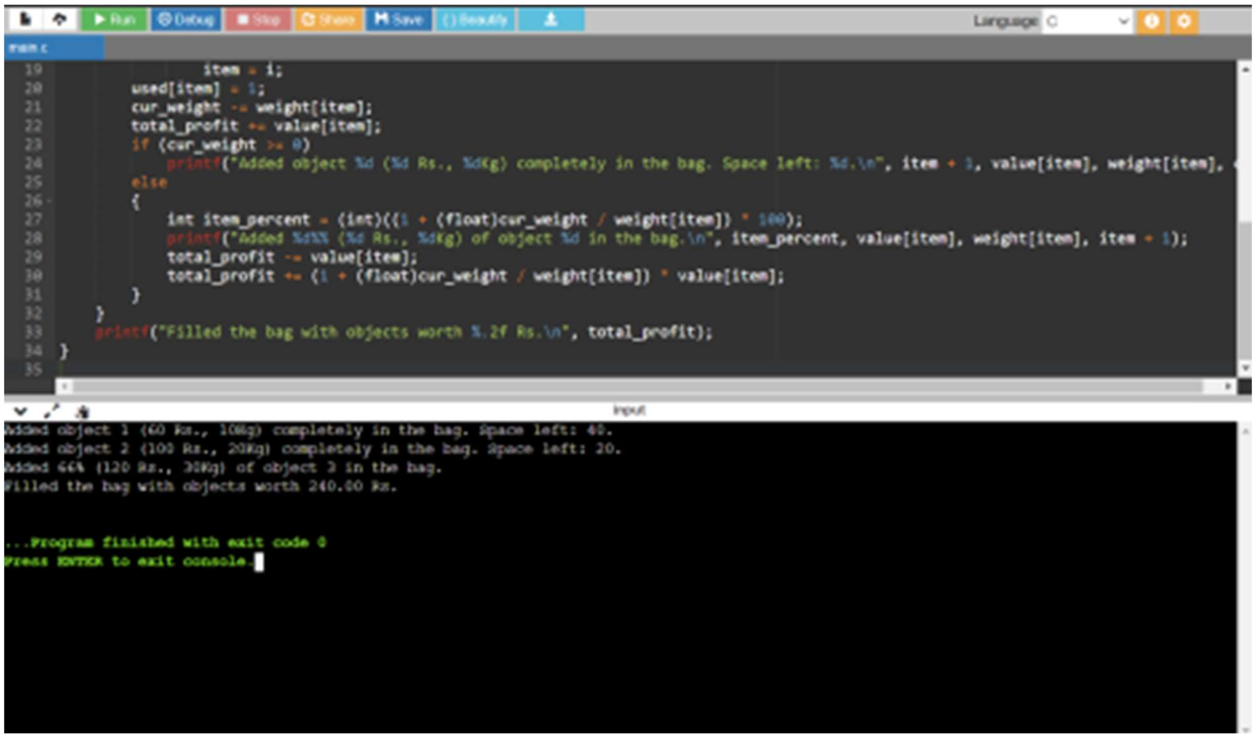
- **Time Complexity:** O(N*W).
  where 'N' is the number of weight element and 'W' is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.
- **Auxiliary Space:** O(N*W).
  The use of 2-D array of size 'N*W'.

Code :

```
/* A Naive recursive implementation

of 0-1 Knapsack problem */

#include <stdio.h>
```

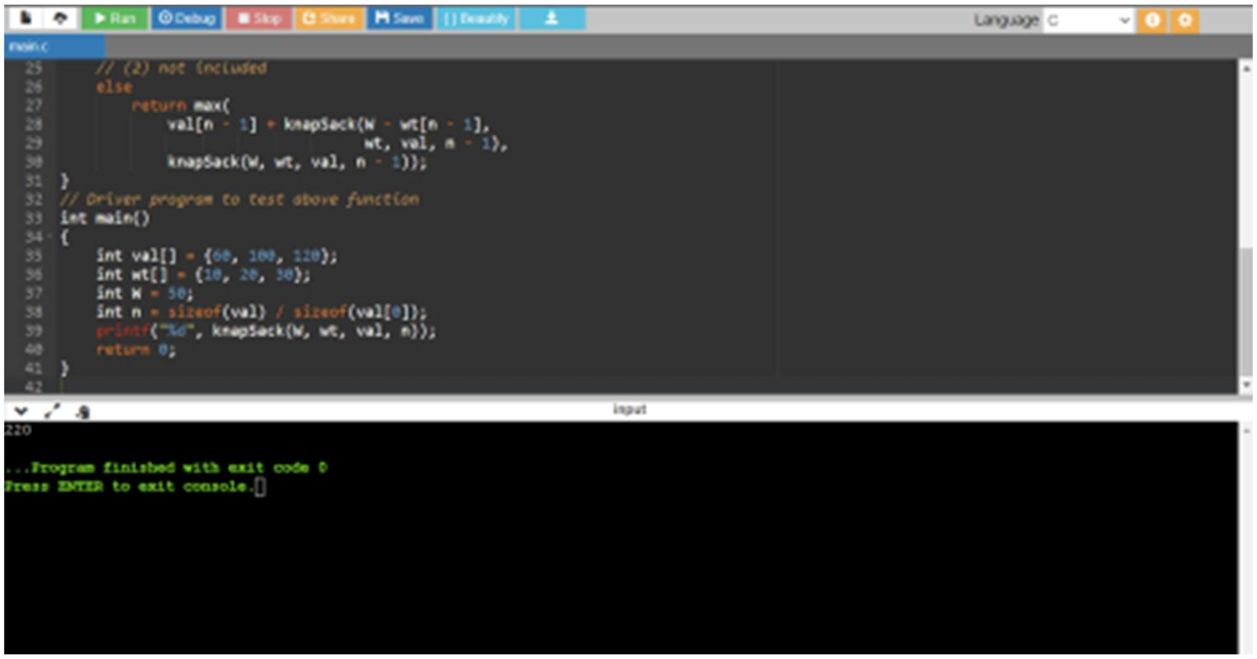
```
// A utility function that returns

// maximum of two integers

int max(int a, int b) { return (a > b) ? a : b; }



// Returns the maximum value that can be

// put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int

 n) {

        // Base Case

        if (n == 0 || W == 0)

                return 0;



        // If weight of the nth item is more than
        // Knapsack capacity W, then this item

        cannot // be included in the optimal solution

        if (wt[n - 1] > W)

                return knapSack(W, wt, val, n - 1);



        // Return the maximum of two cases:

        // (1) nth item included

        // (2) not included

        else

                return max(

                        val[n - 1]

                                + knapSack(W - wt[n - 1],

                                                wt, val, n - 1),

                        knapSack(W, wt, val, n - 1));
}

// Driver program to test above function

int main()

 {

        int val[] = { 60, 100, 120 };

        int wt[] = { 10, 20, 30 };

        int W = 50;

        int n = sizeof(val) / sizeof(val[0]);
        printf("%d", knapSack(W, wt, val, n));

        return 0;

}
```

Conclusion : We studied the Greedy strategy does not necessarily yield an optimal solution over a dynamic programming approach.



```
                item = i;
            used[item] = 1;
            cur_weight -= weight[item];
            total_profit += value[item];
            if (cur_weight >= 0)
                printf("Added object %d (%d Rs., %dKg) completely in the bag. Space left: %d.\n", item + 1, value[item], weight[item], 
            else
            {
                int item_percent = (int)((1 + (float)cur_weight / weight[item]) * 100);
                printf("Added %d%% (%d Rs., %dKg) of object %d in the bag.\n", item_percent, value[item], weight[item], item + 1);
                total_profit -= value[item];
                total_profit += (1 + (float)cur_weight / weight[item]) * value[item];
            }
        }
        printf("Filled the bag with objects worth %.2f Rs.\n", total_profit);
}
```

```
Added object 1 (60 Rs., 10Kg) completely in the bag. Space left: 40.
Added object 2 (100 Rs., 20Kg) completely in the bag. Space left: 20.
Added 66% (120 Rs., 30Kg) of object 3 in the bag.
Filled the bag with objects worth 240.00 Rs.


...Program finished with exit code 0
Press ENTER to exit console.
```



```
    // (2) not included
    else
        return max(
            val[n - 1] + knapSack(W - wt[n - 1],
                                   wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}
// Driver program to test above function
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

```
220

...Program finished with exit code 0
Press ENTER to exit console.
```

Assignment No : 2

Title : Bellman Ford Algorithm using Dynamic Programming

Problem Statement: Write a program to implement Bellman-Ford Algorithm using Dynamic Programming and verify the time complexity.

Objective: To understand the use of Bellman-Ford algorithm, for finding shortest path and implement.

Theory:

Bellman-Ford Algorithm:

Dijkstra and Bellman-Ford Algorithms used to find out single source shortest paths. i.e. there is a source node, from that node we have to find shortest distance to every other node. Dijkstra algorithm fails when graph has negative weight cycle. But Bellman-Ford Algorithm won't fail even, the graph has negative edge cycle. If there any negative edge cycle it will detect and say there is negative edge cycle. If not it will give answer to given problem.

Bellman-Ford Algorithm will work on logic that, **if graph has n nodes, then shortest path never contain more than n-1 edges**. This is exactly what Bellman-Ford do. It is enough to relax each edge (v-1) times to find shortest path. But to find whether there is negative cycle or not we again do one more relaxation. If we get less distance in nth relaxation we can say that there is negative edge cycle. Reason for this is negative value added and distance get reduced. **Relaxing edge**
In algorithm and code below we use this term Relaxing edge.

Relaxing edge is an operation performed on an edge (u, v) . when,

d(u) > d(v) + Cost(u,v)

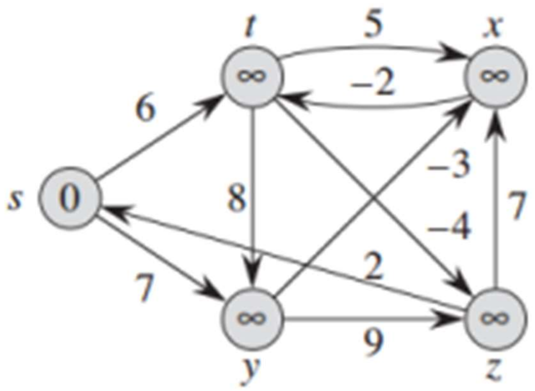
Here d(u) means distance of u. If already known distance to "u" is greater than the path from "s" to “v” and “v” to “u” we update that d(u) value with d(v) + cost(u,v).

Algorithm and Time Complexity

```
Bellman-Ford (G,w,S){ //G is graph given, W is weight matrix, S is source vertex (starting
vertex)
    Initialize single source (G,S) //means initially distance to every node is infinity except to
source. Source is 0 (zero). This will take O(v) time

  For i=1 to |G.V| -1 //Runs (v-1) times
     For each edge (G,V)€ G.E // E times
        Relax(u,v,w) //O(1) time

  For each edge (G,V) € G.E
        If (v.d > u.d + w(u,v)) //The idea behind this is we are relaxing edges nth time if we found
more shortest path than (n-1)th level, we can say that graph is having negative edge cycle and
detected.
        Return False

  return true
}
```

Finally time complexity is (v-1) (E) O(1) = O(VE)

Example Problem



This is the given directed graph.

(s,t) = 6 (y,x) = -3

(s,y)= 7 (y,z) = 9

(t,y) = 8 (x,t) = -2

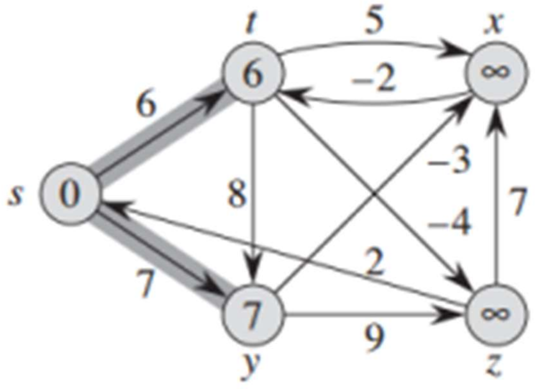
(t,z) = -4 (z,x) = 7

(t,x) = 5 (z,s) = 2

Above we can see using vertex "S" as source (Setting distance as 0), we initialize all other distance as infinity.

S T X Y Z distance 0 ∞ ∞ ∞ ∞ **Path** – – – – –

Table and Image explanation: This table, 2nd row shows distance from source to that particular node ad 3rd row shows to reach that node what is the node we visited recently. This path we can see in the image also.
Note: In each iteration, iteration "n" means it contains the path at most "n" edges. And while we are doing iteration "n" we must follow the graph which we obtained in iteration "n-1". **Iteration 1:** edge (s,t) and (z,y) relaxed and updating the distances to t and y.

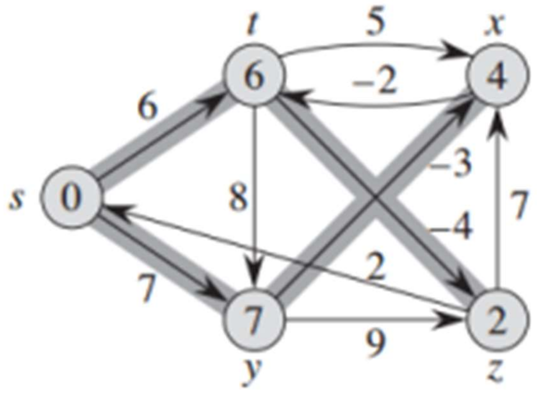


S T X Y Z

distance 0 6 ∞ 7 ∞

Path – S – S –

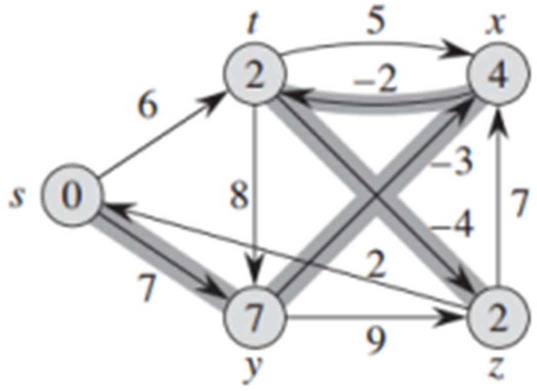
Iteration 2 : edge (t,z) and (y,x) relaxed and x and z values are updated.



S T X Y Z
distance 0 6 4 7 2

Path – S Y S T

Iteration 3: Value of t updated by relaxing (x,t)

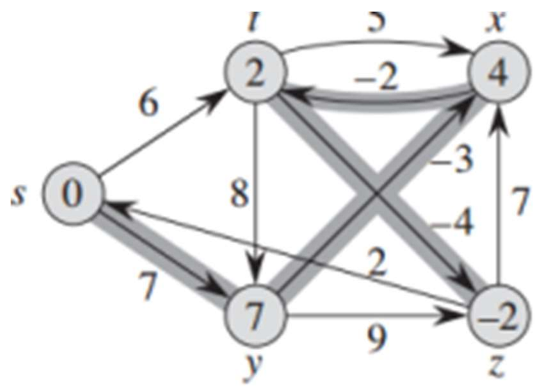


S T X Y Z

distance 0 2 4 7 2

Path – X Y S T

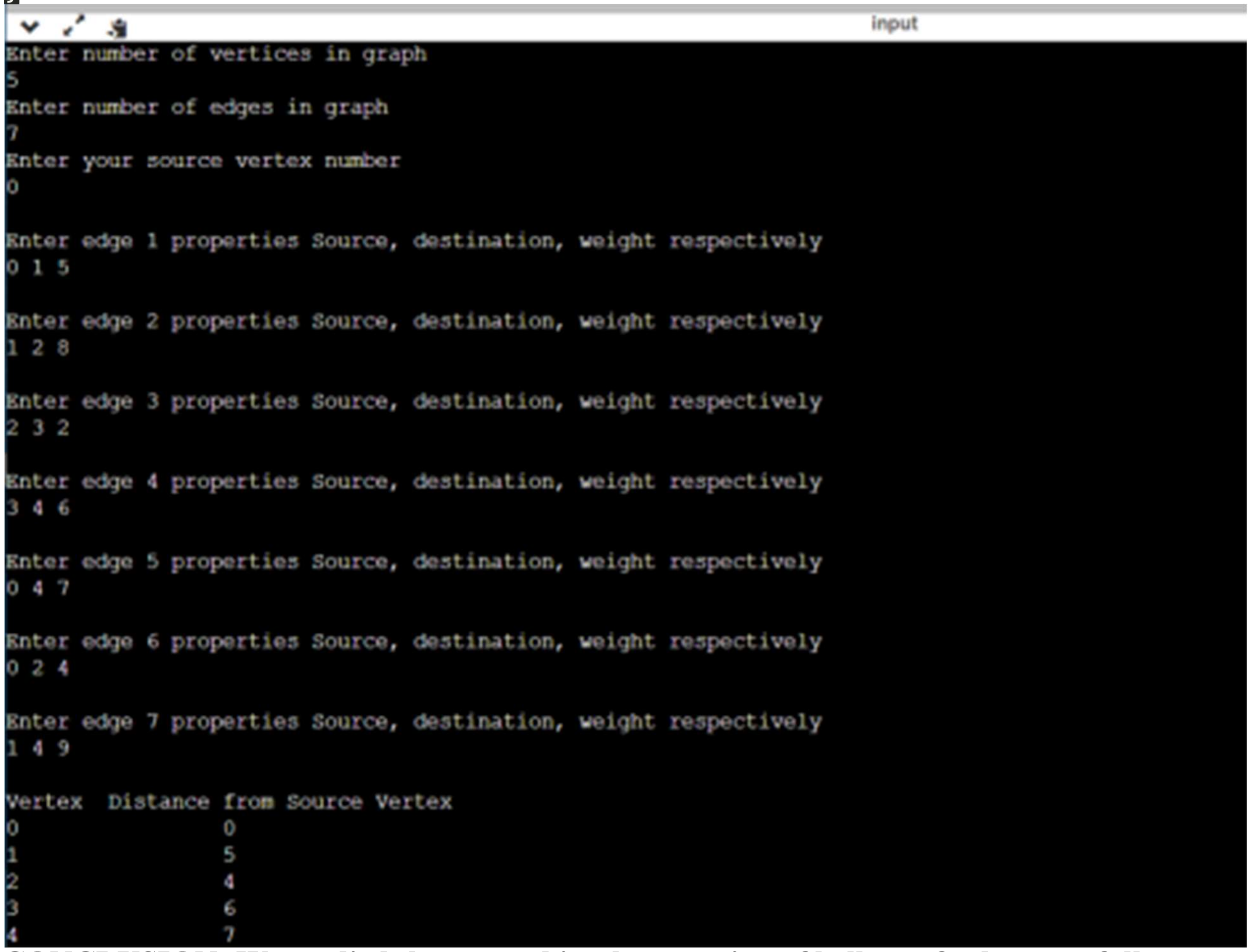
Iteration 4: Value of z updated by relaxing edge (t,z)



Until now 4 iterations completed and shortest path found to every node form source node. Now we have to do one more iteration to find whether there exists negative edge cycle or not. When we do this nth (5th here) relaxation if we found less distance to any vertex from any other path we can say that there is negative edge cycle. Here we can relax any edge to graph which obtained
in iteration 4and we can observe that there is no chance to change those values. So we can confirm that there is no negative edge cycle in this graph.

Code :

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

struct Edge
{
        int source, destination, weight;
};

struct Graph
{
    int V, E;

    struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph));

    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

void FinalSolution(int dist[], int n)
{
    printf("\nVertex\tDistance from Source Vertex\n");
    int i;
    for (i = 0; i < n; ++i){
        printf("%d \t\t %d\n", i, dist[i]);
    }
}

void BellmanFord(struct Graph* graph, int source)
{
    int V = graph->V;

    int E = graph->E;

    int StoreDistance[V];

    int i,j;


    for (i = 0; i < V; i++)
        StoreDistance[i] = INT_MAX;

    StoreDistance[source] = 0;

    for (i = 1; i <= V-1; i++)
    {
        for (j = 0; j < E; j++)
        {
            int u = graph->edge[j].source;

            int v = graph->edge[j].destination;

            int weight = graph->edge[j].weight;

            if (StoreDistance[u] + weight < StoreDistance[v])
                StoreDistance[v] = StoreDistance[u] + weight;
        }
    }



    for (i = 0; i < E; i++)
    {
        int u = graph->edge[i].source;

        int v = graph->edge[i].destination;

        int weight = graph->edge[i].weight;

        if (StoreDistance[u] + weight < StoreDistance[v])
    printf("This graph contains negative edge cycle\n"); }

    FinalSolution(StoreDistance, V);

    return;
}
int main()
{
    int V,E,S;
    printf("Enter number of vertices in graph\n");
    scanf("%d",&V);

    printf("Enter number of edges in graph\n");
    scanf("%d",&E);

    printf("Enter your source vertex number\n");
    scanf("%d",&S);

    struct Graph* graph = createGraph(V, E);
    int i;
    for(i=0;i<E;i++){
        printf("\nEnter edge %d properties Source, destination, weight respective
ly\n",i+1);
        scanf("%d",&graph->edge[i].source);
        scanf("%d",&graph->edge[i].destination);
        scanf("%d",&graph->edge[i].weight);
    }

    BellmanFord(graph, S);

    return 0;
}
```



```
Enter number of vertices in graph
5
Enter number of edges in graph
7
Enter your source vertex number
0

Enter edge 1 properties Source, destination, weight respectively
0 1 5

Enter edge 2 properties Source, destination, weight respectively
1 2 8

Enter edge 3 properties Source, destination, weight respectively
2 3 2

Enter edge 4 properties Source, destination, weight respectively
3 4 6

Enter edge 5 properties Source, destination, weight respectively
0 4 7

Enter edge 6 properties Source, destination, weight respectively
0 2 4

Enter edge 7 properties Source, destination, weight respectively
1 4 9

Vertex  Distance from Source Vertex
0               0
1               5
2               4
3               6
4               7
```

CONCLUSION: We studied the use and implementation of bellman ford successfully.

Assignment No : 3

Problem Statement: - **Write a recursive program to find the solution of placing n queens on chessboard so that no two queens attack each other using Backtracking**

Objective: - In chess, a queen can move as far as she pleases, horizontally, vertically, or diagonally. A chess board has 8 rows and 8 columns. The standard 8 by 8 Queen's problem asks how to place 8 queens on an ordinary chess board so that none of them can hit any other in one move.

Theory: -

The **eight queens puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general ***n*-queens problem** of placing *n* queens on an *n*×*n* chessboard, where solutions exist for all natural numbers *n* with the exception of *n*=2 and *n*=3. The eight queens puzzle has 92 **distinct** solutions. If solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 **fundamental** solutions.
Here we are solving it for N queens in NxN chess board.

(Explain 5 queens problem with Diagram)

Approach:

- Create a solution matrix of the same structure as chess board.
- Whenever place a queen in the chess board, mark that particular cell in solution matrix.
- At the end print the solution matrix, the marked cells will show the positions of the queens in the chess board.

Algorithm

```
Place the queens column wise, start from the left most column
 1. If all queens are placed.
     1. return true and print the solution matrix.
 2. Else
     1. Try all the rows in the current column.
     2. Check if queen can be placed here safely if yes mark the current cell in solution
        matrix as 1 and try to solve the rest of the problem recursively.
     3. If placing the queen in above step leads to the solution return true.
     4. If placing the queen in above step does not lead to the solution , BACKTRACK,
        mark the current cell in solution matrix as 0 and return false.
 3. If all the rows are tried and nothing worked, return false and print NO SOLUTION.
```

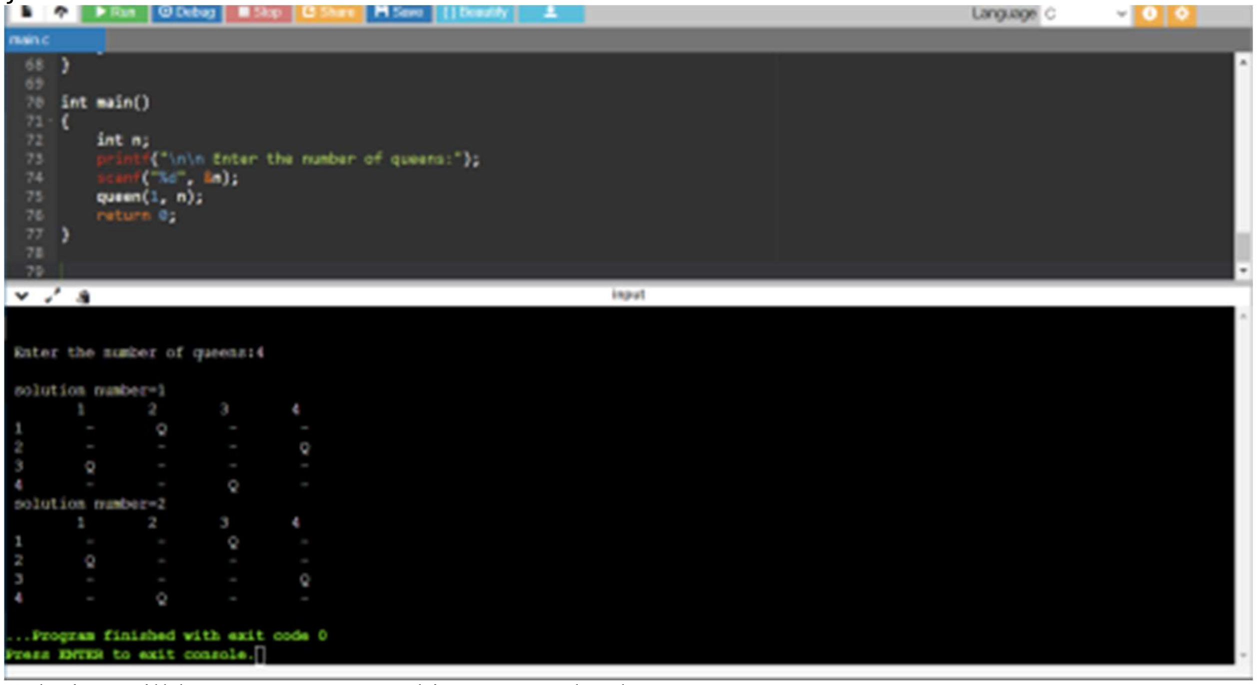
Better Solution: If you notice in solution matrix, at every row we have only one entry as 1 and rest of the entries are 0. Solution matrix takes O(N^2) space. We can reduce it to O(N). We will solve it by taking one dimensional array and consider solution[1] = 2 as "Queen at 1st row is placed at 2nd column

Input:
Enter the number of queens.

Code :

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

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

int a[30], count;


int place(int i, int j)
{
    int k;
    for (k = 1; k <= i - 1; k++)
    {
        if (a[k] == j)
        {
            return 0;
        }
        else if (abs(a[k] - j) == abs(k - i))
        {
            return 0;
        }
    }
    return 1;
}

void print_func(int n)
{
    int i, j, k;
    printf("\n solution number=%d\n", ++count);
    for (i = 1; i <= n; i++)
    {
        printf("\t%d", i);
    }
    for (i = 1; i <= n; i++)
    {
        printf("\n %d", i);
        for (j = 1; j <= n; j++)
        {
            if (a[i] == j)
            {
                printf("\t Q");
            }

            else
            {
                printf("\t -");
            }
        }
    }
}

void queen(int i, int n)
{
    int j, k;
    for (j = 1; j <= n; j++)
    {
        if (place(i, j))
        {
            a[i] = j;
            if (i == n)
            {
                print_func(n);
            }
            else
            {
                queen(i + 1, n);
            }
        }
    }
}


int main()
{
    int n;
    printf("\n\n Enter the number of queens:");
    scanf("%d", &n);
    queen(1, n);
    return 0;
}
```



```c
}

int main()
{
    int n;
    printf("\n\n Enter the number of queens:");
    scanf("%d", &n);
    queen(1, n);
    return 0;
}
```

```
Enter the number of queens:4

solution number=1
        1       2       3       4
1       -       Q       -       -
2       -       -       -       Q
3       Q       -       -       -
4       -       -       Q       -
solution number=2
        1       2       3       4
1       -       -       Q       -
2       Q       -       -       -
3       -       -       -       Q
4       -       Q       -       -

...Program finished with exit code 0
Press ENTER to exit console.
```

Solution will be queens arranged in rows and column

Assignment No : 4

***Problem Statement*:** Write a program to solve the travelling salesman problem and to print the path and the cost using Branch and Bound

<u>Objective</u>: To understand the concept of Branch and bound used in Travel Salesman Problem.

Theory:

Branch and Bound Solution

In Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node.

Note that the cost through a node includes two costs.

1) Cost of reaching the node from the root (When we reach a node, we have this cost computed)

2) Cost of reaching an answer from current node to a leaf (We compute a bound on this cost to decide whether to ignore subtree with this node or not).

- In cases of a **maximization problem**, an upper bound tells us the maximum possible solution if we follow the given node. For example in 0/1 knapsack we used Greedy approach to find an upper bound.
- In cases of a **minimization problem**, a lower bound tells us the minimum possible solution if we follow the given node. For example, in Job Assignment Problem, we get a lower bound by assigning least cost job to a worker.

In branch and bound, the challenging part is figuring out a way to compute a bound on best possible solution. Below is an idea used to compute bounds for Traveling salesman problem.

Cost of any tour can be written as below.

```
Cost of a tour T = (1/2) * ∑ (Sum of cost of two edges
                              adjacent to u and in the
                              tour T)
                   where u ∈ V
```

For every vertex u, if we consider two edges through it in T,

and sum their costs. The overall sum for all vertices would

be twice of cost of tour T (We have considered every edge

twice.)

(Sum of two tour edges adjacent to u) >= (sum of minimum weight

two edges adjacent to

u)

```
Cost of any tour >= 1/2) * ∑ (Sum of cost of two minimum
                              weight edges adjacent to u)
                   where u ∈ V
```

For example, consider the above shown graph. Below are minimum cost two edges adjacent to every node.

Node	Least cost edges	Total cost
0	(0, 1), (0, 2)	25
1	(0, 1), (1, 3)	35
2	(0, 2), (2, 3)	45
3	(0, 3), (1, 3)	45

Thus a lower bound on the cost of any tour =

1/2(25 + 35 + 45 + 45)

= 75

Refer this for one more example.

Now we have an idea about computation of lower bound. Let us see how to how to apply it state space search tree. We start enumerating all possible nodes (preferably in lexicographical order)

1. The Root Node: Without loss of generality, we assume we start at vertex “0” for which the lower bound has been calculated above.

Dealing with Level 2: The next level enumerates all possible vertices we can go to (keeping in mind that in any path a vertex has to occur only once) which are, 1, 2, 3… n (Note that the graph is complete). Consider we are calculating for vertex 1, Since we moved from 0 to 1, our tour has now included the edge 0-1. This allows us to make necessary changes in the lower bound of the root.

Lower Bound for vertex 1 =

Old lower bound - ((minimum edge cost of 0 +

minimum edge cost of 1) / 2)

+ (edge cost 0-1)

How does it work? To include edge 0-1, we add the edge cost of 0-1, and subtract an edge weight such that the lower bound remains as tight as possible which would be the sum of the minimum edges of 0 and 1 divided by 2. Clearly, the edge subtracted can’t be smaller than this.

Dealing with other levels: As we move on to the next level, we again enumerate all possible vertices. For the above case going further after 1, we check out for 2, 3, 4, …n. Consider lower bound for 2 as we moved from 1 to 1, we include the edge 1-2 to the tour and alter the new lower bound for this node.

Lower bound(2) =

Old lower bound - ((second minimum edge cost of 1 +

minimum edge cost of 2)/2)

+ edge cost 1-2)

Note: The only change in the formula is that this time we have included second minimum edge cost for 1, because the minimum edge cost has already been subtracted in previous level

Time Complexity: The worst case complexity of Branch and Bound remains same as that of the Brute Force clearly because in worst case, we may never get a chance to prune a node. Whereas, in practice it performs very well depending on the different instance of the TSP. The complexity also depends on the choice of the bounding function as they are the ones deciding how many nodes to be pruned.

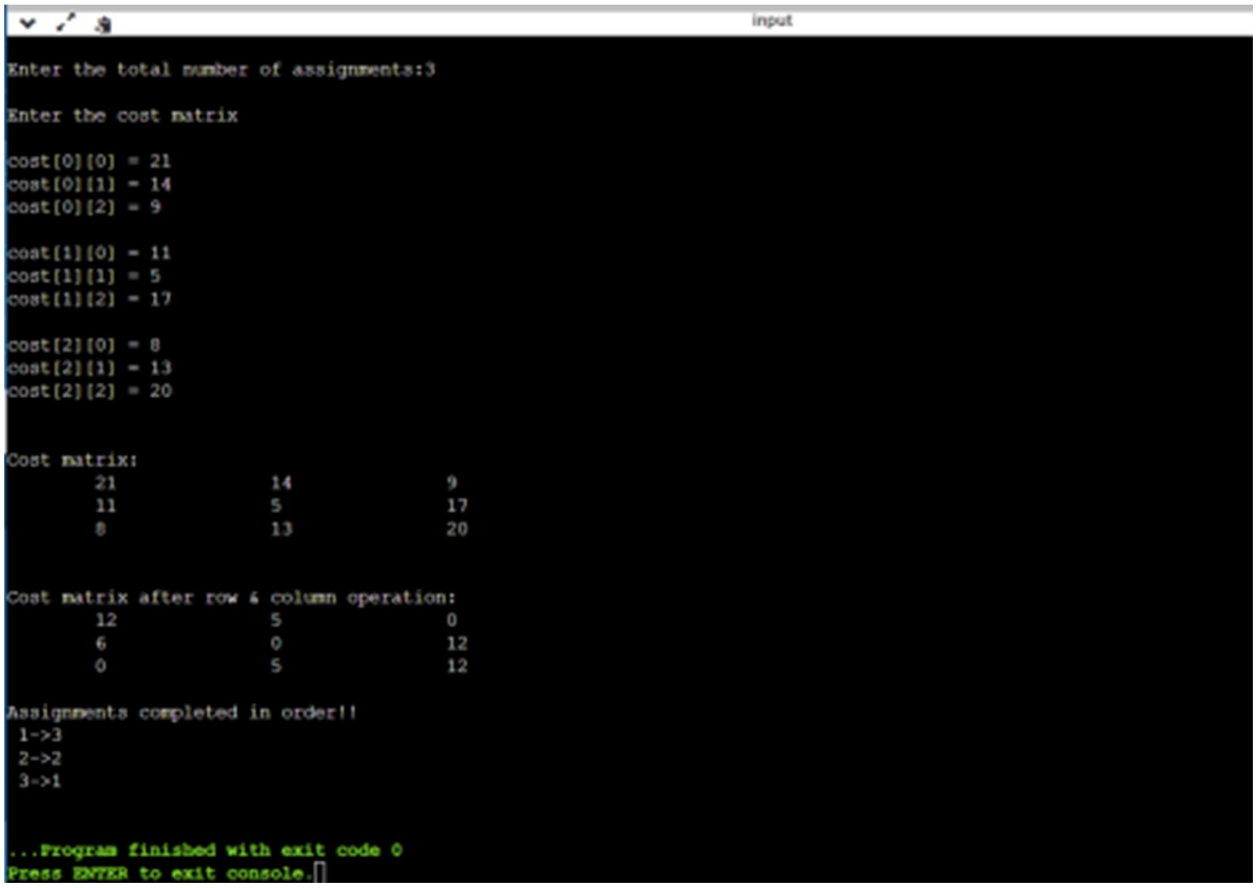
Code :

```
#include <stdio.h>
#include <conio.h>
int main()
{

    int
        cost[20][20],
        min, l, m, sr[20], sc[20], flag[20][20], i, j, k, rf[20], cf[20], n;

    int
        nrz[20],
        ncz[20], cn, a, noz, nrz1[20], ncz1[20], counter = 0;

    printf(
        "\n\nEnter the total number of assignments:");
    scanf(
        "%d", &n);

    /* Enter the cost matrix*/

    printf(
        "\nEnter the cost matrix\n");

    for (i = 0; i < n; i++)

    {

        printf(
            "\n");

        for (j = 0; j < n; j++)

        {

            printf(
                "cost[%d][%d] = ", i, j);

            scanf(
                "%d", &cost[i][j]);
        }
    }

    printf(
        "\n\n");
    /* Display the entered cost matrix*/

    printf(
        "Cost matrix:\n");

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < n; j++)

            printf(
                "\t%d\t", cost[i][j]);

        printf(
            "\n");
    }
    /* operation on rows*/

    for (i = 0; i < n; i++)

    {

        min = cost[i][0];

        /* find the minmum element in each row*/

        for (j = 0; j < n; j++)

        {

            if (min > cost[i][j])

                min = cost[i][j];
        }

        /*subtract the minimum element from each element of the respective
rows*/

        for (j = 0; j < n; j++)

            cost[i][j] = cost[i][j] - min;
    }

    /* operation on colums*/

    for (i = 0; i < n; i++)

    {

        min = cost[0][i];

        /* find the minimum element in each column*/

        for (j = 0; j < n; j++)
        {

            if (min > cost[j][i])

                min = cost[j][i];
        }
        /*subtract the minimum element from each element of the respective
columns*/

        for (j = 0; j < n; j++)

            cost[j][i] = cost[j][i] - min;
    }

    printf(
        "\n\n");

    printf(
        "Cost matrix after row & column operation:\n");

    for (i = 0; i < n; i++)

    {
        for (j = 0; j < n; j++)

            printf(
                "\t%d\t", cost[i][j]);

        printf(
            "\n");
    }

repeatx:;

    /*Draw minimum number of horizontal and vertical lines to cover all zeros
in

resulting matrix*/

    a = 0;
    noz = 0, min = 1000;

    for (i = 0; i < n; i++)

    {
        for (j = 0; j < n; j++)

            flag[i][j] = 0;
    }

    for (i = 0; i < n; i++)

    {
        cn = 0;
        for (j = 0; j < n; j++)
        {
            if (cost[i][j] == 0)

            {
                cn++;

                flag[i][j] = 1;
            }
        }

        nrz[i] = cn;

        noz = noz + cn;
    }

    for (i = 0; i < n; i++)

    {
        cn = 0;

        for (j = 0; j < n; j++)

        {
            if (cost[j][i] == 0)

            {
                cn++;

                flag[j][i] = 1;
            }
        }

        ncz[i] = cn;

        noz = noz + cn;
    }

    for (i = 0; i < n; i++)

    {
        nrz1[i] = nrz[i];

        ncz1[i] = ncz[i];
    }
    k = 0;

    while (nrz[k] != 0 || ncz[k] != 0)

    {

        for (i = 0; i < n; i++)

        {
            cn = 0;
            for (j = 0; j < n; j++)

            {
                if (flag[i][j] == 1)

                    cn++;

                nrz[i] = cn;
            }

            if (nrz[i] == 1)

            {
                for (j = 0; j < n; j++)

                {
                    if (flag[i][j] == 1)

                    {
                        flag[i][j] = 2;

                        for (k = 0; k < n; k++)

                        {
                            if (flag[k][j] == 1)

                                flag[k][j] = 0;
                        }
                    }
                }
            }
        }

        for (i = 0; i < n; i++)

        {
            cn = 0;
            for (j = 0; j < n; j++)

            {
                if (flag[j][i] == 1)

                    cn++;

                ncz[i] = cn;
            }

            if (ncz[i] == 1)

            {
                for (j = 0; j < n; j++)

                {
                    if (flag[j][i] == 1)

                    {
```

```
                        flag[j][i] = 2;

                        for (k = 0; k < n; k++)

                        {
                            if (flag[j][k] == 1)

                                flag[j][k] = 0;
                        }
                    }
                }
            }
        }

        k++;
    }

    for (i = 0; i < n; i++)

    {
        for (j = 0; j < n; j++)

        {
            if (flag[i][j] == 2)

                a++;
        }
    }
    /* If minimum number of lines, a is equal to the order of the matrix n
then

assignment can be optimally completed.*/

    if (a == n)

    {

        printf(
            "\nAssignments completed in order!!\n");

        /* Display the order in which assignments will be completed*/

        for (i = 0; i < n; i++)

        {
            for (j = 0; j < n; j++)

            {
                if (flag[i][j] == 2)

                    printf(
                        " %d->%d ", i + 1, j + 1);
            }

            printf(
                "\n");
        }

        getch();

        exit(0);
    }

    /* if order of matrix and number of lines is not same then its difficult
to

find the optimal solution.

Now determine the smallest uncovered element i.e. element not covered by
lines

and then subtract this minimum element from all uncovered elements and add
the

same elements at the intersection of horizontal and vertical lines.*/
    else

    {
        for (i = 0; i < n; i++)

        {
            rf[i] = 0, sr[i] = 0;

            cf[i] = 0, sc[i] = 0;
        }

        for (k = n; (k > 0 && noz != 0); k--)

        {
            for (i = 0; i < n; i++)

            {
                m = 0;

                for (j = 0; j < n; j++)

                {
                    if ((flag[i][j] == 4) && (cost[i][j] == 0))

                        m++;
                }

                sr[i] = m;
            }

            for (i = 0; i < n; i++)

            {
                if (nrz1[i] == k && nrz1[i] != sr[i])

                {
```

```
            rf[i] = 1;

            for (j = 0; j < n; j++)

            {
                if (cost[i][j] == 0)

                    flag[i][j] = 4;
            }

            noz = noz - k;
        }
    }

    for (i = 0; i < n; i++)

    {

        l = 0;

        for (j = 0; j < n; j++)

        {
            if ((flag[j][i] == 4) && (cost[j][i] == 0))

                l++;
        }

        sc[i] = l;
    }

    for (i = 0; i < n; i++)

    {
        if (ncz1[i] == k && ncz1[i] != sc[i])

        {
            cf[i] = 1;

            for (j = 0; j < n; j++)

            {
                if (cost[j][i] == 0)

                    flag[j][i] = 4;
            }

            noz = noz - k;
        }
    }

    for (i = 0; i < n; i++)

    {
        for (j = 0; j < n; j++)
```

```
                {
                    if (flag[i][j] != 3)
                    {
                        if (rf[i] == 1 && cf[j] == 1)

                        {
                            flag[i][j] = 3;

                            if (cost[i][j] == 0)

                                noz = noz + 1;
                        }
                    }
                }
            }
        }

        for (i = 0; i < n; i++)

        {
            for (j = 0; j < n; j++)

            {
                if (rf[i] != 1 && cf[j] != 1)

                {
                    if (min > cost[i][j])

                        min = cost[i][j];
                }
            }
        }

        for (i = 0; i < n; i++)

        {
            for (j = 0; j < n; j++)

            {
                if (rf[i] != 1 && cf[j] != 1)

                    cost[i][j] = cost[i][j] - min;
            }
        }

        for (i = 0; i < n; i++)

        {
            for (j = 0; j < n; j++)
            {
                if (flag[i][j] == 3)

                    cost[i][j] = cost[i][j] + min;
            }
        }
    }
    printf(
        "\n\n");

    if (counter < 10)

    {

        counter = counter + 1;

        printf(
            "\n\nIntermediate Matrix: \n");

        for (i = 0; i < n; i++)

        {

            for (j = 0; j < n; j++)

                printf(
                    "\t%d\t", cost[i][j]);

            printf(
                "\n");
        }
    }

    else

    {

        printf(
            "\n\nOptimal solution to given problem is not possible");

        getch();

        return 0;
    }

    goto repeatx;
}
```



```
Enter the total number of assignments:3

Enter the cost matrix

cost[0][0] = 21
cost[0][1] = 14
cost[0][2] = 9

cost[1][0] = 11
cost[1][1] = 5
cost[1][2] = 17

cost[2][0] = 8
cost[2][1] = 13
cost[2][2] = 20


Cost matrix:
        21              14              9
        11              5               17
        8               13              20


Cost matrix after row & column operation:
        12              5               0
        6               0               12
        0               5               12

Assignments completed in order!!
 1->3
 2->2
 3->1


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: We successfully studied the use of Branch and Bound for Travelling Salesman Problem.