Name: Rajshri Firke
Roll No: 307A035
Division: 1 (TEIT-1)
Batch: B

Assignment No: 1

Problem Statement:

Write a program to implement Fractional knapsack using Greedy algorithm and 0/1 knapsack using Dynamic programming.

Program:

Implementation of Fractional Knapsack

```
#include<iostream>
#include<algorithm>
#include<bits/stdc++.h>
using namespace std;

//Structure of the object and profit,weight corresponding to object
typedef struct obejct
{
   int profit;
   int weight;

   obejct(int profit,int weight)    //Parameterised constructor for object
   {
      this->profit=profit;
      this->weight=weight;
   }
}obj;   //Custom datatype for object

//Case-I:Greedy about profit
int cmp1(obj a,obj b)    //Compare function to return max profit
{
   int temp1,temp2;
   temp1=a.profit;
   temp2=b.profit;
```

```
    return temp1>temp2;
}
float ks1(obj a[],int k_capacity,int n)
{
    int curr_weight=0;        //Current weight in knapsack
    float total_profit=0.0;     //Total profit
    int r_capacity;           //Remaining capacity

    sort(a,a+n,cmp1);    //Sort function to sort object on basis of maximum profit

    //Going through all objects
    for(int i=0;i<n;i++)
    {
        //Add object in knapsack if weight of given object is less than knapsack
capacity
        //Add that object completely
        if(curr_weight+a[i].weight<=k_capacity)
        {
            curr_weight=curr_weight+a[i].weight;
            total_profit=total_profit+a[i].profit;
        }
        else
        {
            //If we can't add current object add fractional part of it
            r_capacity=k_capacity-curr_weight;        //calculating remaining
capacity of knapsack
            total_profit=total_profit+(float)(r_capacity*a[i].profit)/a[i].weight;
            break;
        }
    }
    return total_profit;     //Returning total final profit
}

//Case-II:Greedy about weight
int cmp2(obj a,obj b)   //Compare function to return min weight
{
    float temp1,temp2;
```

```
    temp1=a.weight;
    temp2=b.weight;
    return temp1<temp2;
}
float ks2(obj a[],int k_capacity,int n)
{
    int curr_weight=0;       //Current weight in knapsack
    float total_profit=0.0;    //Total profit
    int r_capacity;          //Remaining capacity

    sort(a,a+n,cmp2);   //Sort function to sort object on basis of minimum weight
    //Going through all objects
    for(int i=0;i<n;i++)
    {
        //Add object in knapsack if weight of given object is less than knapsack
capacity
        //Add that object completely
        if(curr_weight+a[i].weight<=k_capacity)
        {
            curr_weight=curr_weight+a[i].weight;
            total_profit=total_profit+a[i].profit;
        }
        else
        {
            //If we can't add current object add fractional part of it
            r_capacity=k_capacity-curr_weight;            //calculating remaining
capacity of knapsack
            total_profit=total_profit+(float)(r_capacity*a[i].profit)/a[i].weight;
            break;
        }
    }
    return total_profit;   //Returning total final profit
}

//Case-III:Greedy about Profit/Weight ratio
int cmp3(obj a,obj b)   //Compare function to return max P/W ratio
{
```

```
    float temp1,temp2;
    temp1=(float)a.profit/a.weight;
    temp2=(float)b.profit/b.weight;
    return temp1>temp2;
}
float ks3(obj a[],int k_capacity,int n)
{
    int curr_weight=0;        //Current weight in knapsack
    float total_profit=0.0;     //Total profit
    float r_capacity;           //Remaining capacity

    sort(a,a+n,cmp3);        //Sort function to sort object on basis of maximum
P/W ratio
    //Going through all objects
    for(int i=0;i<n;i++)
    {
        if(curr_weight+a[i].weight<=k_capacity)
        {
            //Add object in knapsack if weight of given object is less than knapsack
capacity
            //Add that object completely
            curr_weight=curr_weight+a[i].weight;
            total_profit=total_profit+a[i].profit;
        }
        else
        {
            //If we can't add current object add fractional part of it
            r_capacity=k_capacity-curr_weight;          //calculating remaining
capacity of knapsack
            total_profit=total_profit+(float)(r_capacity*a[i].profit)/a[i].weight;
            break;
        }
    }
    return total_profit;     //Returning total final profit
}

int main()
```

```
{
   int k_capacity=20;    //capacity of knapsack

   obj a[]={{25,18},{24,15},{15,10}};    //Profit and weight values as pairs

   //To find the size of array
   int size=sizeof(a)/sizeof(a[0]);

   cout<<"Case-I:Greedy about profit: "<<endl;
   cout<<"Total Profit:"<<ks1(a,k_capacity,size)<<endl;

   cout<<"Case-II:Greedy about weight: "<<endl;
   cout<<"Total profit:"<<ks2(a,k_capacity,size)<<endl;

   cout<<"Case-III:Greedy about Profit/Weight: "<<endl;
   cout<<"Total profit:"<<ks3(a,k_capacity,size)<<endl;

   return 0;
}
```

Output:
Case-I:Greedy about profit:
Total Profit:28.2
Case-II:Greedy about weight:
Total profit:31
Case-III:Greedy about Profit/Weight:
Total profit:31.5

Program:

Implementation of 0/1 knapsack using dynamic programming

```
#include<iostream>
#include<bits/stdc++.h>
```

```
using namespace std;

// Max function for comparing two numbers
int max(int a,int b)
{
   return (a>b)?a:b;
}

//0/1 Knapsack function
void knapsack(int p[],int w[],int n,int k_capacity)
{
   int i,j,total_profit;
   int a[n+1][k_capacity+1];     //2-d array for matrix implementation
   for(i=0;i<=n;i++)             //Loop for traversing row
   {
      for(j=0;j<=k_capacity;j++)//Loop for traversing columns
      {
         if(i==0 || j==0)
         {
            a[i][j]=0;        //Initialising first row and first column with zero
         }
         else if(w[i-1]<=j)    //If weight is greater than w[i-1] then use formula
         {
            a[i][j]=max(a[i-1][j],(a[i-1][j-w[i-1]]+p[i-1]));
         }
         else                  //Else copy the above value as it is
         {
            a[i][j]=a[i-1][j];
         }
      }
   }
   //The last box of matrix holds the total profit
   //a[n][k_capacity]=Total Profit
   int profit=a[n][k_capacity];
   cout<<"Total profit: "<<profit<<endl;

   cout<<"Matrix generated for Dynamic Programming: "<<endl;
```

```
    for(i=0;i<=n;i++)
    {
        for(j=0;j<=k_capacity;j++)
        {
                cout<<a[i][j]<<"\t";
                }
       cout<<endl;
        }
        cout<<endl;

    //For finding which item is included:
// either the result comes from the top
    // (a[i-1][j]) or from (p[i-1] + a[i-1]
    // [j-w[i-1]]) as in Knapsack table
    // If it comes from the latter one, it means
    // the item is included.
    for(i=n;i>0 && profit>0;i--)
    {
       if(profit==a[i-1][j])
       {
          cout<<"This item is not included "<<i<<" ->0"<<endl;
       }
       else
       {
          //This item is included
          cout<<"This item is included"<<i<<" ->1"<<endl;
          // Since this weight is included its
          // value is deducted
          profit=profit-p[i-1];
          w=w-w[i-1];
       }
    }
}


int main()
{
```

```
    int n,k_capacity;
    cout<<"Enter the number of objects: "<<endl;
    cin>>n;
    cout<<"Enter the capacity: "<<endl;
    cin>>k_capacity;

    int w[n];    //Weight array
    int p[n];    //Profit array

    cout<<"Enter the weights: "<<endl;
    for(int i=0;i<n;i++)
    {
        cin>>w[i];    //Accepting weights values
    }

    cout<<"Enter the profit: "<<endl;
    for(int i=0;i<n;i++)
    {
        cin>>p[i];    //Accepting profits values
    }

    knapsack(p,w,n,k_capacity);   //Function call for knapsack
    return 0;
}
```

Output:

```
Enter the number of objects:
5
Enter the capacity:
10
Enter the weights:
2  1 5 7 3
Enter the profit:
21 32 12 1 5
Total profit: 65
Matrix generated for Dynamic Programming:
0	0	0	0	0	0	0	0	0	0	0
```

0	0	21	21	21	21	21	21	21	21	21
0	32	32	53	53	53	53	53	53	53	53
0	32	32	53	53	53	53	53	65	65	65
0	32	32	53	53	53	53	53	65	65	65
0	32	32	53	53	53	58	58	65	65	65

This item is included5 ->1
This item is included4 ->1
This item is included3 ->1
This item is included2 ->1
This item is included1 ->1

Name: Rajshri Firke
Roll No: 307A035
Division: 1 (TEIT-1)
Batch: B

Assignment No: 2

Problem Statement:

Write a program to implement Bellman-Ford Algorithm using Dynamic programming and Verify the time Complexity.

Program:

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

//Structure for node of graph
struct node
{
	int u;
	int v;
	int wt;
	node(int u,int v,int wt)  //Constructor for creation of node
	{
		this->u=u;
		this->v=v;
		this->wt=wt;
	}
};

int main()
{
	int n,m;     //n=Number of vertices and m=Number of edges
	cout<<"Enter the number of vertices: "<<endl;
	cin>>n;
	cout<<"Enter the number of edges: "<<endl;
```

```
    cin>>m;

    //u=current vertex, v=next vertex, wt=distance between these two this two
vertices
    int u,v,wt;
    vector<node> edges;
    cout<<"Enter the source,vertex and weight: "<<endl;
    cout<<"\tu"<<"\tv"<<"\twt"<<endl;
    for(int i=0;i<m;i++)
    {
        cin>>u>>v>>wt;
        edges.push_back(node(u,v,wt));
    }

    int src;
    cout<<"Enter the source: "<<endl;  //Taking input i.e...source vertex
    cin>>src;

    int inf=100000;
    vector<int> dist(n,inf);    //At the beginnig all the vertices are having
weight(distnace) infinity
    dist[src]=0;                        //The distance(weight) of source to
source is 0

    for(int i=0;i<=n-1;i++)
    {
        //Loop for each relaxing each edge
for(auto it:edges)      //Here we are relaxing every edge(repeatedly) for |n-1|
times
        {
            if(dist[it.u] + it.wt < dist[it.v])
            {
                dist[it.v] = dist[it.u] + it.wt;
              }
        }
    }
```

```
	//Check for negative cycles
	//After relaxing each edge |n-1| times, if distance of any vertex is getting 
reduced
	//that means there is negatibe cycle
	//And otherwise if distances are remain same so there is no negative cycle
	int f=0;
	for(auto it:edges)
	{
		if(dist[it.u]+it.wt<dist[it.v])   //Checking condition
		{
			cout<<"Graph contains negative-weight cycle"<<endl;
			f=1;
			break;
		}
	}

	//If negative cycle is not found then
	//then print out vertex and its corresponding
	//distance with respect to source
	if(!f)
	{
		for(int i=1;i<=n;i++)
		{
			   cout<<"Corresponding vertex and its distance from 
source: "<<endl;
			cout<<i<<" "<<dist[i]<<endl;
		}
	}

	return 0;
}
```

Output:

Enter the number of vertices:
4
Enter the number of edges:

8
Enter the source,vertex and weight:
u v wt
1 2 -1
2 3 2
4 2 1
2 4 2
2 5 3
1 5 4
4 5 5
3 4 -3
Enter the source:
1
Corresponding vertex and its distance from source:
1 0
Corresponding vertex and its distance from source:
2 -1
Corresponding vertex and its distance from source:
3 1
Corresponding vertex and its distance from source:
4 -2

Name: Rajshri Firke
Roll No: 307A035
Division: 1 (TEIT-1)
Batch: B

Assignment No: 3

Problem Statement:

Write a recursive program to find the solution of placing n queens on a chessboard so that no queen takes each other

Program:

```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;

bool isSafe(int** board,int x,int y,int n) //x is for row, y is for column and n is number of queens
{                                   //int** is double pointer in stack
   //Checking if there is any safe position in columns else return false
   for(int row=0;row<x;row++)
   {
      if(board[row][y]==1)
      {
         return false;
      }
   }

   int row=x; //iterators
   int col=y;

   //Checking if there is any safe position in left upper diagonal else return false
   while(row>=0 && col>=0)
   {
      if(board[row][col]==1)
```

```
        {
            return false;
        }
        row--;
        col--;
    }

    row=x;
    col=y;

    //Checking if there is any safe position in right upper diagonal else return
false
    while(row>=0 && col<n)
    {
        if(board[row][col]==1)
        {
            return false;
        }
        row--;
        col++;
    }
    return true;   //Finally return true if positions are safe
}

bool nQueen(int** board,int x,int n)
{
    if(x>=n)
    {
return true;      //n queens are placed
    }
    //we are checking our safe positions for columns
    for(int col=0;col<n;col++)   //new col variable
    {
        if(isSafe(board,x,col,n))   //If isSafe() returning true place queen
        {
            board[x][col]=1;        //Queen is placed on board
```

```
            if(nQueen(board,x+1,n)) //Recursive call
            {
                return true;
            }
            board[x][col]=0;    //Backtracking
        }
    }
    return false;
}

int main()
{
    int n;
    cout<<"Enter Number of queens: "<<endl;
    cin>>n;

    int** board=new int*[n];      //Board of n*n

    for(int i=0;i<n;i++)
    {
        board[i]=new int[n];
        for(int j=0;j<n;j++)
        {
            board[i][j]=0;    //Initialise board with 0
        }
    }

    //Calling nQueen() function
    if(nQueen(board,0,n))
    {
        for(int i=0;i<n;i++)     //Printing board(n*n)
        {
            for(int j=0;j<n;j++)
            {
                cout<<"Solution for N-Queen: "<<endl;
                cout<<"1=Queen is place and 0=Box is empty "<<endl<<endl;
                cout<<board[i][j]<<" ";   //If 1 is present that means
```

```
            }                          //there is queen placed
            cout<<endl;
        }
    }
    return 0;
}
```

Output:

```
Enter Number of queens:
4
Solution for N-Queen:
1=Queen is place and 0=Box is empty
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
```

Name: Rajshri Firke
Roll No: 307A035
Division: 1 (TEIT-1)
Batch: B

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.

Program:

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;
const int N = 4;

// final_path[] stores the final solution ie, the
// path of the salesman.
int final_path[N+1];

// visited[] keeps track of the already visited nodes
// in a particular path
bool visited[N];

// Stores the final minimum weight of shortest tour.
int final_res = INT_MAX;

// Function to copy temporary solution to
// the final solution
void copyToFinal(int curr_path[])
{
	for (int i=0; i<N; i++)
		final_path[i] = curr_path[i];
	final_path[N] = curr_path[0];
```

```
}

// Function to find the minimum edge cost
// having an end at the vertex i
int firstMin(int adj[N][N], int i)
{
	int min = INT_MAX;
	for (int k=0; k<N; k++)
		if (adj[i][k]<min && i != k)
			min = adj[i][k];
	return min;
}

// function to find the second minimum edge cost
// having an end at the vertex i
int secondMin(int adj[N][N], int i)
{
	int first = INT_MAX, second = INT_MAX;
	for (int j=0; j<N; j++)
	{
		if (i == j)
			continue;

		if (adj[i][j] <= first)
		{
			second = first;
			first = adj[i][j];
}
		else if (adj[i][j] <= second &&
				adj[i][j] != first)
			second = adj[i][j];
	}
	return second;
}

// function that takes as arguments:
// curr_bound -> lower bound of the root node
```

```
// curr_weight-> stores the weight of the path so far
// level-> current level while moving in the search
//                   space tree
// curr_path[] -> where the solution is being stored which
//                            would later be copied to final_path[]
void TSPRec(int adj[N][N], int curr_bound, int curr_weight,
                              int level, int curr_path[])
{
          // base case is when we have reached level N which
          // means we have covered all the nodes once
          if (level==N)
          {
                    // check if there is an edge from last vertex in
                    // path back to the first vertex
                    if (adj[curr_path[level-1]][curr_path[0]] != 0)
                    {
                              // curr_res has the total weight of the
                              // solution we got
                              int curr_res = curr_weight +
                                                  adj[curr_path[level-1]][curr_path[0]];

                              // Update final result and final path if
                              // current result is better.
                              if (curr_res < final_res)
                              {
                                        copyToFinal(curr_path);
                                        final_res = curr_res;
                              }
                    }
                    return;
          }

          // for any other level iterate for all vertices to
          // build the search space tree recursively
          for (int i=0; i<N; i++)
          {
                    // Consider next vertex if it is not same (diagonal
```

```
                // entry in adjacency matrix and not visited
                // already)
                if (adj[curr_path[level-1]][i] != 0 &&
visited[i] == false)
                {
                        int temp = curr_bound;
                        curr_weight += adj[curr_path[level-1]][i];

                        // different computation of curr_bound for
                        // level 2 from the other levels
                        if (level==1)
                        curr_bound -= ((firstMin(adj, curr_path[level-1]) +
                                                        firstMin(adj, i))/2);
                        else
                        curr_bound -= ((secondMin(adj, curr_path[level-1]) +
                                                        firstMin(adj, i))/2);

                        // curr_bound + curr_weight is the actual lower bound
                        // for the node that we have arrived on
                        // If current lower bound < final_res, we need to explore
                        // the node further
                        if (curr_bound + curr_weight < final_res)
                        {
                                curr_path[level] = i;
                                visited[i] = true;

                                // call TSPRec for the next level
                                TSPRec(adj, curr_bound, curr_weight, level+1,
                                        curr_path);
                        }

                        // Else we have to prune the node by resetting
                        // all changes to curr_weight and curr_bound
                        curr_weight -= adj[curr_path[level-1]][i];
                        curr_bound = temp;

                        // Also reset the visited array
```

```
                        memset(visited, false, sizeof(visited));
                        for (int j=0; j<=level-1; j++)
                                visited[curr_path[j]] = true;
                }
        }
}

// This function sets up final_path[]
void TSP(int adj[N][N])
{
        int curr_path[N+1];

        // Calculate initial lower bound for the root node
        // using the formula 1/2 * (sum of first min +
        // second min) for all edges.
        // Also initialize the curr_path and visited array
int curr_bound = 0;
        memset(curr_path, -1, sizeof(curr_path));
        memset(visited, 0, sizeof(curr_path));

        // Compute initial bound
        for (int i=0; i<N; i++)
                curr_bound += (firstMin(adj, i) +
                                        secondMin(adj, i));

        // Rounding off the lower bound to an integer
        curr_bound = (curr_bound&1)? curr_bound/2 + 1 :
                                                        curr_bound/2;

        // We start at vertex 1 so the first vertex
        // in curr_path[] is 0
        visited[0] = true;
        curr_path[0] = 0;

        // Call to TSPRec for curr_weight equal to
        // 0 and level 1
        TSPRec(adj, curr_bound, 0, 1, curr_path);
```

```
}

// Driver code
int main()
{
	//Adjacency matrix for the given graph
	int adj[N][N] = { {0, 10, 15, 20},
		{10, 0, 25, 35},
		{20, 10, 0, 30},
		{10, 50, 12, 0}
	};

	TSP(adj);


	  cout<<"Minimum cost: "<<final_res;
	cout<<"Path Taken: "<<endl;

	for (int i=0; i<=N; i++)
		cout<<" ->"<<final_path[i];


	return 0;
}
```

Output:

```
Minimum cost: 52Path Taken:
 ->0 ->3 ->2 ->1 ->0
```