NAME:	Shubham Solanki
UID:	2022301015
SUBJECT	Design and Analysis of Algorithms
EXPERIMENT NO:	06
AIM:	To implement Single source shortest path
Algorithm:	**Bellman Ford Algorithm** function bellmanFordAlgorithm(G, s) //G is the graph and s is the source vertex for each vertex V in G dist[V] <- infinite // dist is distance prev[V] <- NULL // prev is previous dist[s] <- 0 for each vertex V in G for each edge (u,v) in G temporaryDist <- dist[u] + edgeweight(u, v) if temporaryDist < dist[v] dist[v] <- temporaryDist prev[v] <- u for each edge (U,V) in G If dist[U] + edgeweight(U, V) < dist[V} Error: Negative Cycle Exists

	return dist[], previ[] **Djikstra Algorithm** 1. function Dijkstra(Graph, source): 2. for each vertex v in Graph.Vertices: 3. dist[v] ← INFINITY 4. prev[v] ← UNDEFINED 5. add v to Q 6. dist[source] ← 0 7. while Q is not empty: 8. u ← vertex in Q with min dist[u] 9. remove u from Q 10. for each neighbor v of u still in Q: 11. alt ← dist[u] + Graph.Edges(u, v) 12. if alt < dist[v]: 13. dist[v] ← alt 14. prev[v] ← u 15. return dist[], prev[]

Code Part 1:	**A weighted, directed graph in which edge weights may be negative G=(V; E) with source s (Bellman-Ford)**
	
	Source Code
	
	#include<bits/stdc++.h>
	using namespace std;
	
	int V;
	void printSolution(int dist[])
	{
	cout << "Vertex \t Distance from Source" << endl;
	for (int i = 0; i < V; i++)
	cout << i << " \t\t\t\t" << dist[i] << endl;
	}
	
	void BellmanFord(int ** graph,int src, vector<pair<int,int>> edges){

```
        int dist[V];

  for(int i=0;i<V;i++){
        dist[V]=INT_MAX;
  }
        dist[src]=0;

  for(int it=1;it<=V-1;it++){

        for(int i=0;i<edges.size();i++){
        int u=edges[i].first;
        int v=edges[i].second;

        if(dist[u]!=INT_MAX && dist[u]+graph[u][v]<dist[v]){
        dist[v]=dist[u]+graph[u][v];
        }

        }
  }

  for (int i = 0; i < edges.size(); i++) {
        int u=edges[i].first;
        int v=edges[i].second;
        int weight = graph[u][v];
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
        printf("Graph contains negative weight cycle");
        return;
        }
  }

  printSolution(dist);

}


int main(){

  cout<<"Enter the number of vertices :";
```

```
 cin>>V;

 int **graph=new int*[V];
        for(int i=0;i<V;i++)
 {
        graph[i]=new int[V];
 }

 for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
        graph[i][j]=INT_MAX;
        }
 }

 cout<<"Enter the number of edges :";
 int e; cin >> e;

 vector<pair<int,int>> edges;

 for(int i=0;i<e;i++){

        cout<<"\nEnter the Vertices of the edge "<<i<<" :";
        int a,b,w;
        cin>>a>>b;
        a--;b--;
        edges.push_back(make_pair(a,b));


        cout<<"Enter the Weight of the edge "<<i<<" :";
        cin>>w;

        graph[a][b]=w;

 }

 BellmanFord(graph,0,edges);

        return 0;
}
```

Output:

```
Enter the number of vertices :5
Enter the number of edges :6

Enter the Vertices of the edge 0 :1 2
Enter the Weight of the edge 0 :5

Enter the Vertices of the edge 1 :1 5
Enter the Weight of the edge 1 :1

Enter the Vertices of the edge 2 :1 4
Enter the Weight of the edge 2 :4

Enter the Vertices of the edge 3 :2 3
Enter the Weight of the edge 3 :-2

Enter the Vertices of the edge 4 :3 4
Enter the Weight of the edge 4 :3

Enter the Vertices of the edge 5 :4 5
Enter the Weight of the edge 5 :5
Vertex   Distance from Source
1                              0
2                              5
3                              3
```

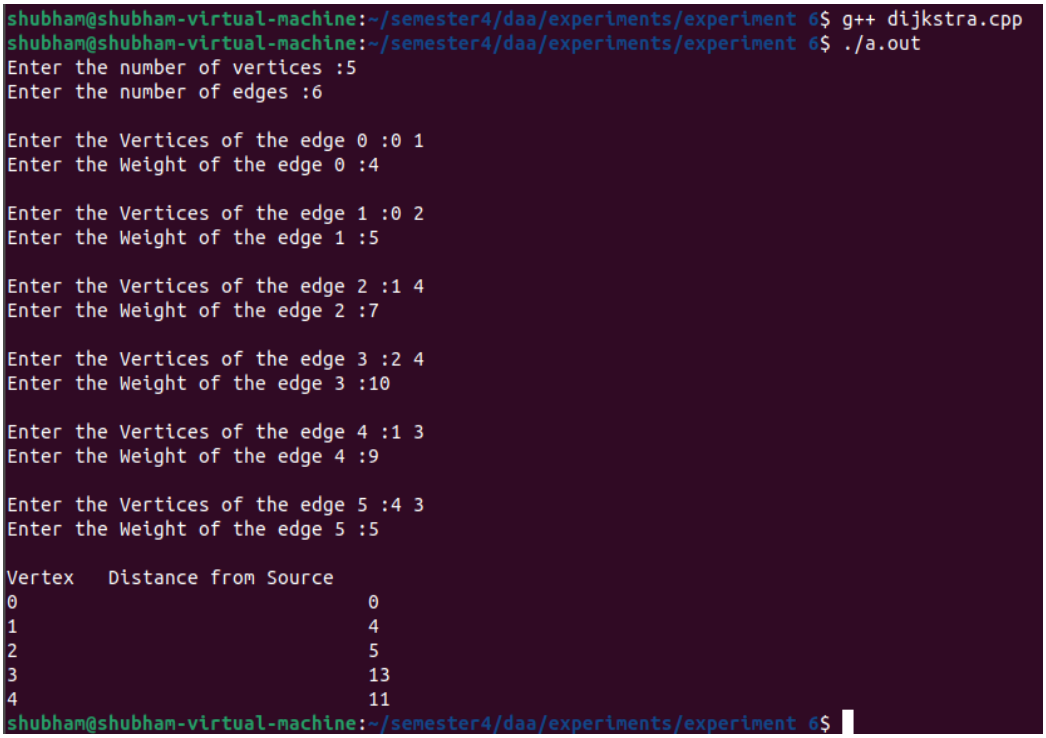
```
Enter the Vertices of the edge 2 :1 4
Enter the Weight of the edge 2 :4

Enter the Vertices of the edge 3 :2 3
Enter the Weight of the edge 3 :-2

Enter the Vertices of the edge 4 :3 4
Enter the Weight of the edge 4 :3

Enter the Vertices of the edge 5 :4 5
Enter the Weight of the edge 5 :5
Vertex   Distance from Source
1                            0
2                            5
3                            3
4                            4
5                            1
```

Code Part 2:	**A weighted, directed graph G=(V; E) for the case in which all edge weights are nonnegative with source s (Dijkstra)** **Source Code**

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

int V;



int minDistance(int distance[],bool sptSet[]){

   int minDist=INT_MAX;
   int minVertex=0;

   for(int i=0;i<V;i++){
      if(sptSet[i]==false && distance[i]<=minDist){
         minDist=distance[i];
         minVertex=i;
      }
   }
   return minVertex;
}

void printSolution(int dist[])
{
   cout << "\nVertex \t Distance from Source" << endl;
   for (int i = 0; i < V; i++)
      cout << i << " \t\t\t\t" << dist[i] << endl;
}



void dijkstra(int **graph, int src)
{
   int dist[V];

   bool sptSet[V];
```

```
    for (int i = 0; i < V; i++){

        dist[i] = INT_MAX;
        sptSet[i] = false;  // All s=distance initialised to INF

    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {

        int u = minDistance(dist, sptSet); //u is vertex with min distance

        sptSet[u]=true; // u included

        // Update dist value of the adjacent vertices of the picked vertex.
        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an edge from
u to v,
            // and total weight of path from src to  v through u is smaller
than current value of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX &&
dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist);
}

int main(){

    cout<<"Enter the number of vertices :";
    cin>>V;

    int **graph=new int*[V];
    for(int i=0;i<V;i++)
```

```
    {
        graph[i]=new int[V];
    }

    for(int i=0;i<V;i++){

      for(int j=0;j<V;j++){

            graph[i][j]=0;

        }
    }

    cout<<"Enter the number of edges :";
    int e; cin >> e;

    for(int i=0;i<e;i++){

        cout<<"\nEnter the Vertices of the edge "<<i<<" :";
        int a,b,w;

        cin>>a>>b;
        cout<<"Enter the Weight of the edge "<<i<<" :";

        cin>>w;

        graph[a][b]=w;

        graph[b][a]=w;

    }

    dijkstra(graph,0);

    return 0;

}
```

Output 2:	
Conclusion:	Thus we have implemented Bellman Ford and Djikstra Algorithm to find the shortest path between two nodes in a graph