Name Chirag Rana
SE COMPS
2018130043
C batch

EXPERIMENT 6

AIM: To implement Kruskal’s algorithm using Greedy Method.

THEORY:

1] Greedy algorithm:
A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

2] How do you decide which choice is optimal?

Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

3] How to create a Greedy Algorithm?

Being a very busy person, you have exactly T time to do some interesting things and you want to do maximum such things.

You are given an array A of integers, where each element indicates the time a thing takes for completion. You want to calculate the maximum number of things that you can do in the limited time that you have.

This is a simple Greedy-algorithm problem. In each iteration, you have to greedily select the things which will take the minimum amount of time to complete while maintaining two variables currentTime and numberOfThings. To complete the calculation, you must:

1. Sort the array A in a non-decreasing order.
2. Select each to-do item one-by-one.

3. Add the time that it will take to complete that to-do item into currentTime.
4. Add one to numberOfThings.

Repeat this as long as the currentTime is less than or equal to T.

4] Greedy algorithms have some advantages and disadvantages:

1. It is quite easy to come up with a greedy algorithm (or even multiple greedy algorithms) for a problem.
2. Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.
3. The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

5] Kruskal’s Minimum Spanning Tree Algorithm using Greedy approach:

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph. Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

What is Minimum Spanning Tree?

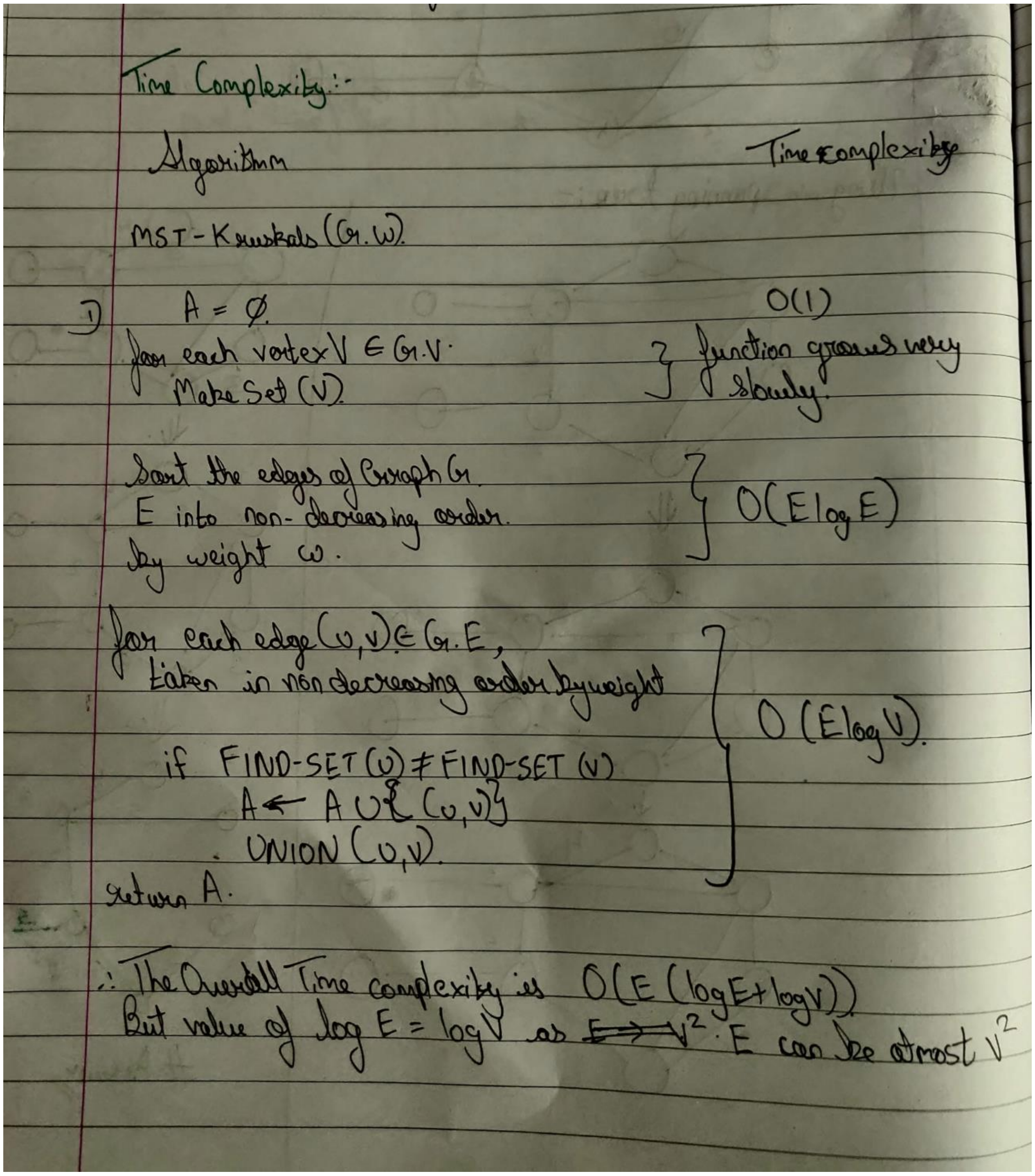
Given a connected and undirected graph, a spanning tree of that graph is a

subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

ALGORITHM:

```
algorithm Kruskal(G) is
    A := ∅
    for each v ∈ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v), increasing do
        if FIND-SET(u) ≠ FIND-SET(v) then
            A := A ∪ {(u, v)}
            UNION(FIND-SET(u), FIND-SET(v))
    return A
```

TIME-COMPLEXITY:

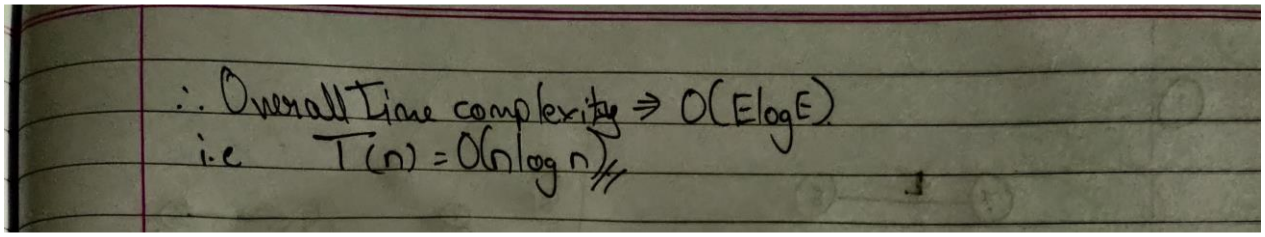


Time Complexity:-

```
Algorithm                                         Time complexity

MST-Kruskals (G.W).

1)   A = ∅.                                       O(1)
for each vertex V ∈ G.V.                        } function grows very
    MakeSet (V).                                    slowly.

Sort the edges of Graph G                       }
E into non-decreasing order.                    } O(E log E)
by weight w.                                    }

for each edge (u,v) ∈ G.E,                      }
  taken in non decreasing order by weight       }
                                                } O (E log V).
    if FIND-SET (u) ≠ FIND-SET (v)              }
        A ← A ∪ {(u,v)}                         }
        UNION (u,v).                            }
return A.
```

∴ The Overall Time complexity is $O(E(\log E + \log V))$

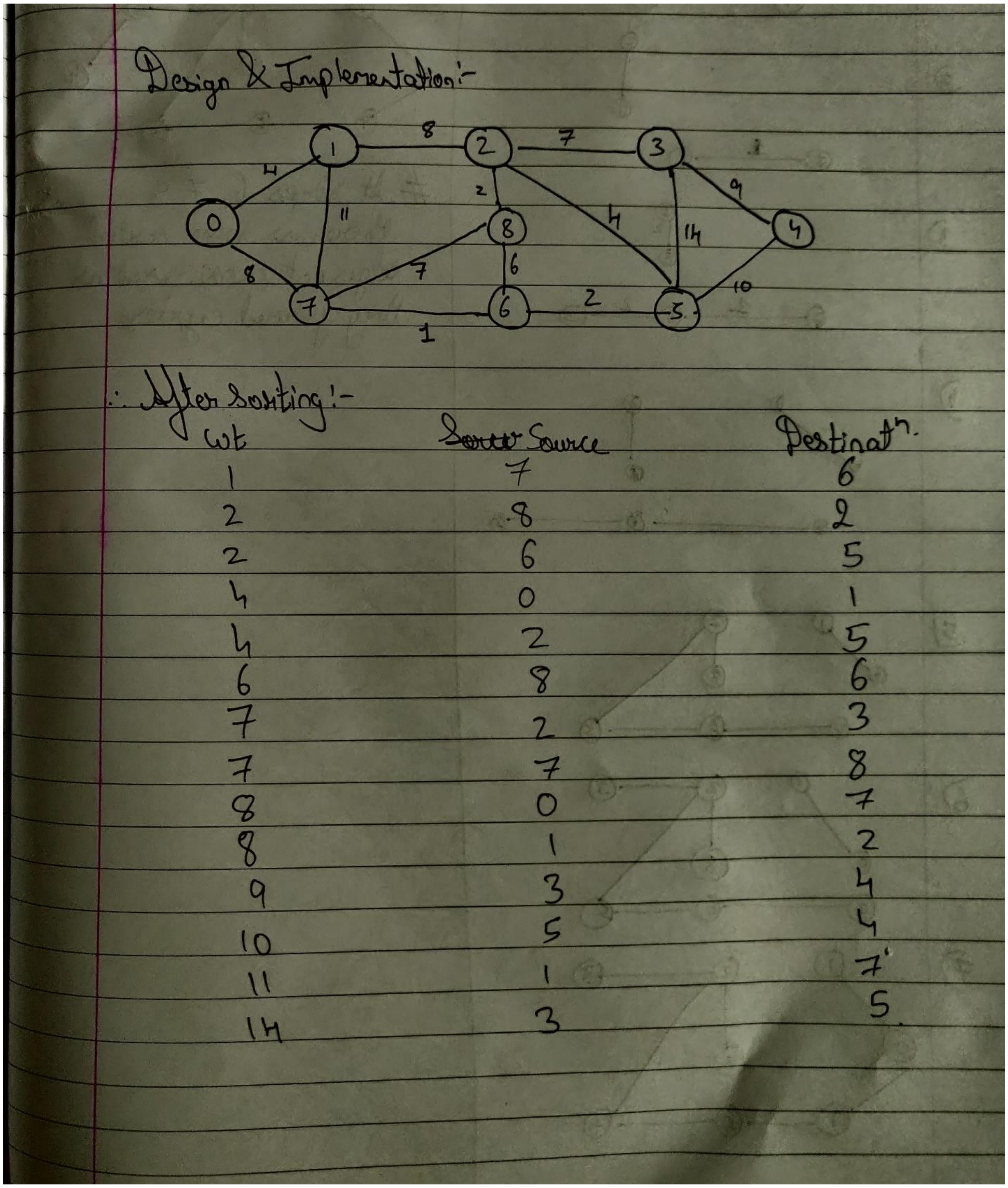
But value of $\log E = \log V$ as ~~$E \Rightarrow$~~ V^2. E can be atmost V^2



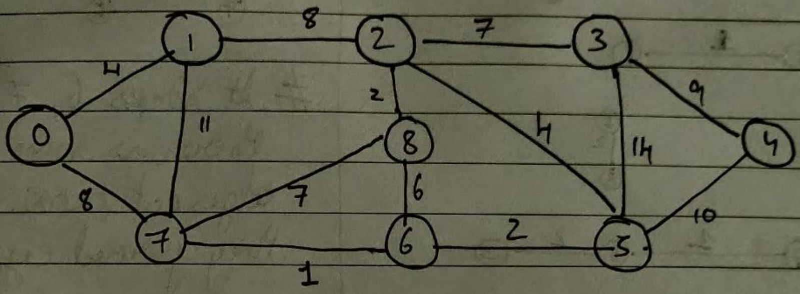
∴ Overall Time complexity ⇒ $O(E \log E)$

i.e $T(n) = O(n \log n)$

DESIGN AND IMPLEMENTATION:

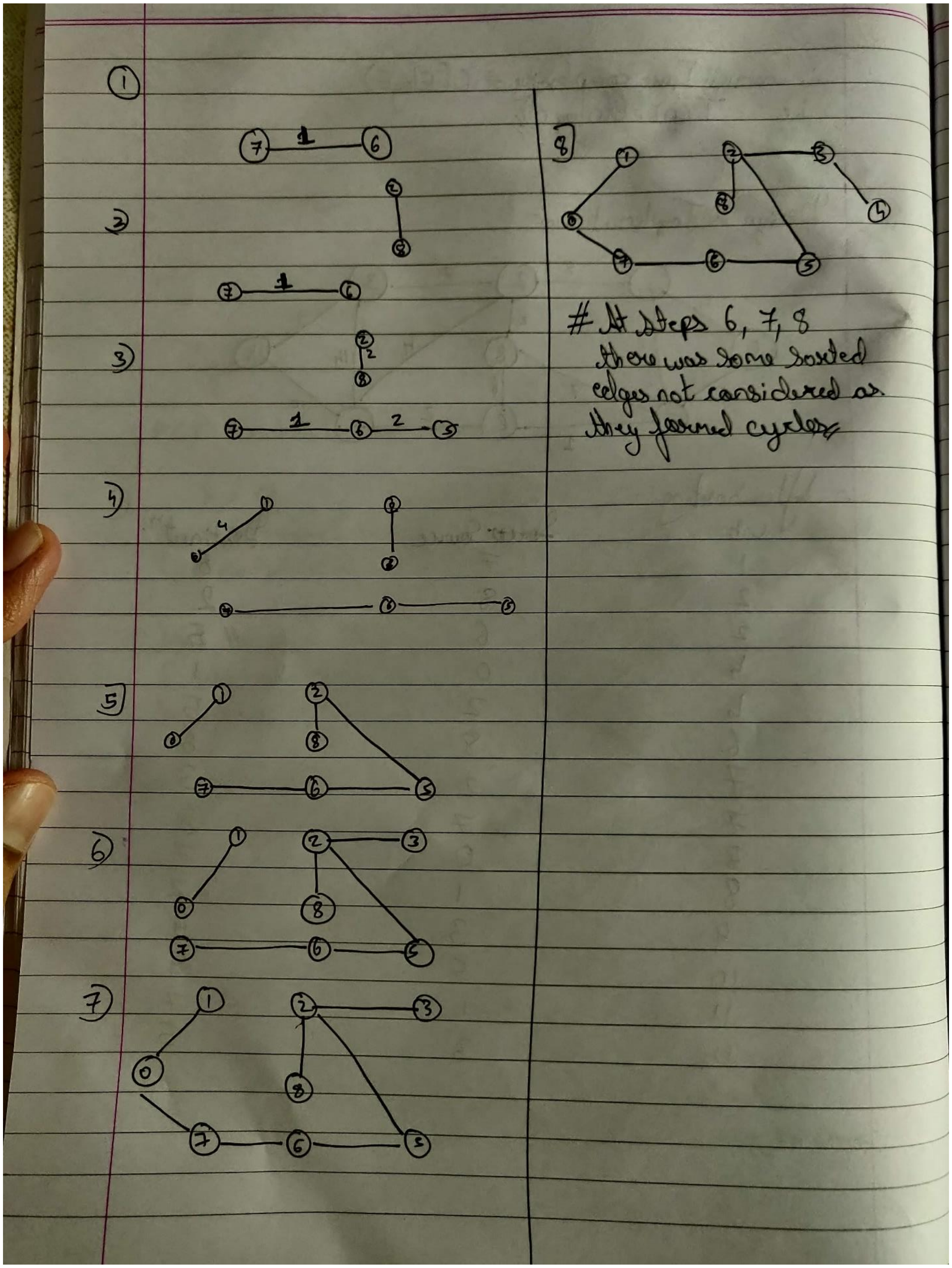


Design & Implementation:-



∴ After Sorting:-

Wt	~~Sorce~~ Source	Destinat^n.
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5



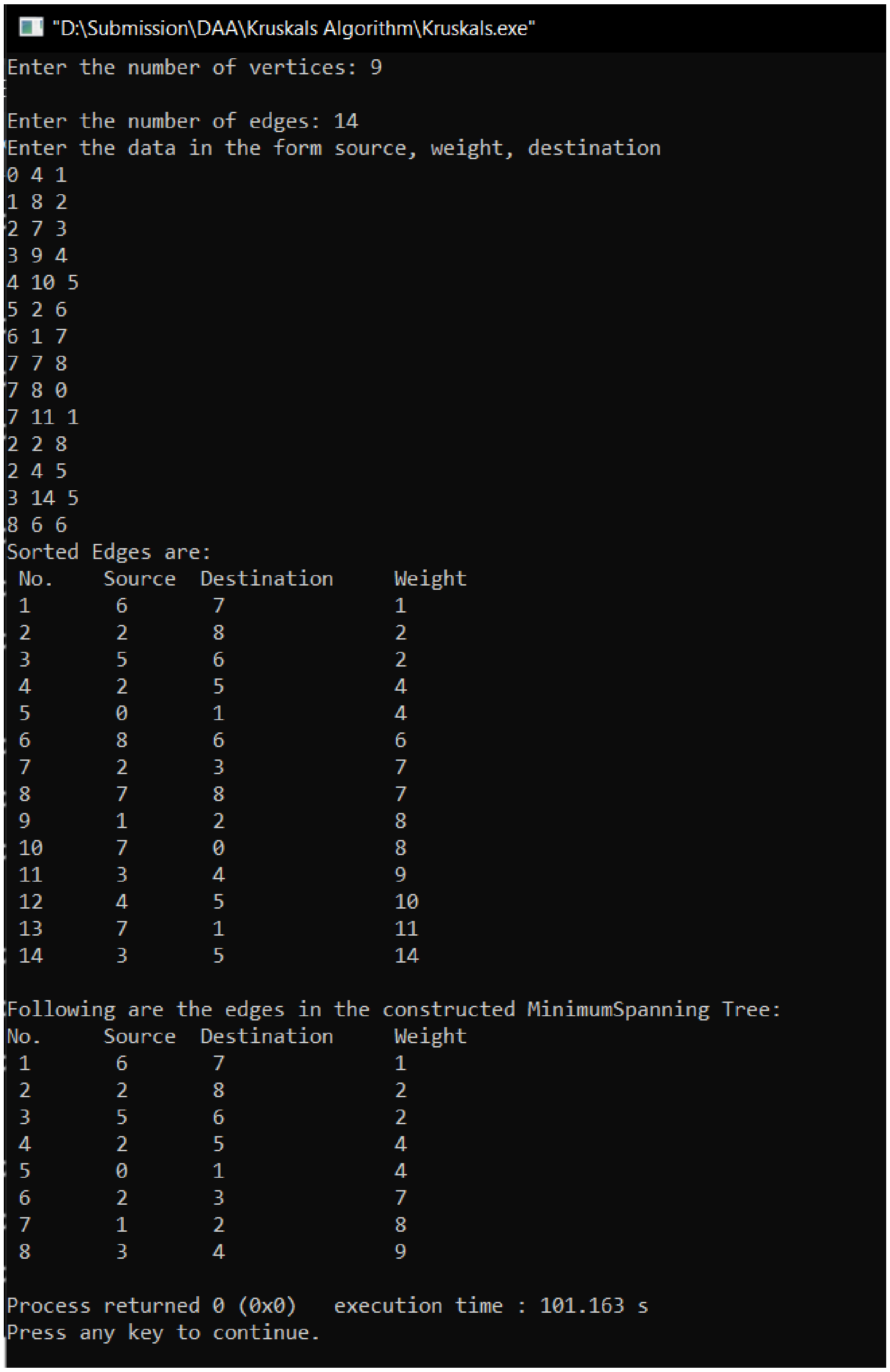
CODE:

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

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

int V;
int E;
struct Edge* edge[15];
struct Edge* solution[15];
int count=0;

void input()
{
 printf("\nEnter the number of vertices: ");
 scanf("%d",&V);

 printf("\nEnter the number of edges: ");
 scanf("%d",&E);
 //edge = (struct Edge*)malloc(E*sizeof(struct Edge));
 printf("Enter the data in the form source, weight, destination\n");
 for(int i=0;i<E;i++)
 {
     edge[i] = (struct Edge*)malloc(sizeof(struct Edge));
     scanf("%d",&edge[i]->source);
     scanf("%d",&edge[i]->weight);
     scanf("%d",&edge[i]->destination);
 }
}
void sort_edges()
{
 int min, min_index;
 for(int i=0; i<E-1; i++)
 {
     min=edge[i]->weight;
     min_index=i;
     for(int j=i;j<E;j++)
     {
         if(min>=edge[j]->weight)
         {
             min=edge[j]->weight;
             min_index=j;
         }
     }
     struct Edge *temp;
     temp=edge[i];
     edge[i]=edge[min_index];
     edge[min_index]=temp;
```

```
        }
    }
    int find(int parent[], int i)
    {
     if(parent[i]==-1)
        return i;
     return find(parent,parent[i]);
    }
    void Union(int parent[],int x,int y)
    {
        parent[x]=y;
    }
    void minimum_spanning_tree()
    {
     int n_sorted_edges =0;
     int n_solution_edges=0;
     int parent[15];
     for (int i = 0; i < 15; i++)
     {
         parent[i]=-1;
     }
     while(n_solution_edges<V-1 && n_sorted_edges<E)
     {
         struct Edge* new_edge = edge[n_sorted_edges++];
         int x = find(parent,new_edge->source);
         int y = find(parent,new_edge->destination);
         if(x!=y)
         {
             solution[n_solution_edges++]=new_edge;
             count++;
             Union(parent,x,y);
         }
     }
     printf("\nFollowing are the edges in the constructed MinimumSpanning Tree:\n");
     printf("No.\tSource\tDestination\tWeight\n");
     for (int i = 0; i < count; i++)
     {
        printf(" %d\t %d\t %d\t\t%d \n",i+1,solution[i]->source,solution[i]-
>destination,solution[i]->weight);
     }
    }
    int main()
    {
     input();
     sort_edges();
     printf("Sorted Edges are:\n ");
     printf("No.\tSource\tDestination\tWeight\n");
     for(int i=0;i<E;i++)
     {
        printf(" %d\t %d\t %d\t\t%d \n",i+1,edge[i]->source,edge[i]->destination,edge[i]-
>weight);
     }
     minimum_spanning_tree();
     return 0;
    }
```

OUTPUT:



```
"D:\Submission\DAA\Kruskals Algorithm\Kruskals.exe"
Enter the number of vertices: 9

Enter the number of edges: 14
Enter the data in the form source, weight, destination
0 4 1
1 8 2
2 7 3
3 9 4
4 10 5
5 2 6
6 1 7
7 7 8
7 8 0
7 11 1
2 2 8
2 4 5
3 14 5
8 6 6
Sorted Edges are:
 No.     Source  Destination     Weight
 1        6       7              1
 2        2       8              2
 3        5       6              2
 4        2       5              4
 5        0       1              4
 6        8       6              6
 7        2       3              7
 8        7       8              7
 9        1       2              8
 10       7       0              8
 11       3       4              9
 12       4       5              10
 13       7       1              11
 14       3       5              14

Following are the edges in the constructed MinimumSpanning Tree:
No.      Source  Destination     Weight
 1        6       7              1
 2        2       8              2
 3        5       6              2
 4        2       5              4
 5        0       1              4
 6        2       3              7
 7        1       2              8
 8        3       4              9

Process returned 0 (0x0)   execution time : 101.163 s
Press any key to continue.
```

CONCLUSION:

1] I understood how to implement the concepts of union set theory.

2] I also understood how to implement structures efficiently.