```
int arr1[100000];
int arr2[100000];
```

Массивы arr1 – родители arr2 – ранги вершин

```
// disjoint set union
void init_dsu() {
        for (int i = 0; i < 100000; i++) {
                arr1[i] = i;
                arr2[i] = 1;
        }
}
```

Инициализация массивов. По умолчанию каждая вершина связана только с собой же. Ранг = 1

```
// Return tree root
int get_root(int v) {
        if (arr1[v] == v) {
                return v;
        }
        else {
                return arr1[v] = get_root(arr1[v]);
        }
}
```

Функция поиска к какой компоненте связности принадлежит вершина. Ищется вершина-представитель компоненты.

```
// Merge unions
bool merge(int a, int b) {
        int ra = get_root(a), rb = get_root(b);

        if (ra == rb) {
                return false;
        }
        else {
                if (arr1[ra] < arr2[rb]) {
                        arr1[ra] = rb;
                }
                else if (arr1[rb] < arr2[ra]) {
                        arr1[rb] = ra;
                }
                else {
                        arr1[ra] = rb;
                        arr2[rb]++;
                }

                return true;
        }
}
```

Слияние 2-х компонент связности. Функция принимает на вход номера 2-х вершин. Если 2 вершины уже пренадлежат 1 компоненте возвращается false, в противном случае происходит слияние.

```
bool dfs(graph* root, int val)
{


        if (root == NULL) { return false; }

        else if (root->data == val) { return true; }


        if ( root->left !=NULL) { return dfs(root->left, val); }

        else if (root->right != NULL) { return dfs(root->right, val); }

        else { return dfs(root->parent, val); }
}
```

Поиск в глубину вершины со значение val. Если такая вершина есть, вернет true.

```
struct graph
{
        int data;
        graph* left = NULL;
        graph* right = NULL;
        graph* parent = NULL;
};
```

Структура графа

```
// Vertexes

class Vertex
{
public:
        string name = " ";
        int number = 0;
};

// Edges
class Edge
{
public:
        Vertex vertex1;
        Vertex vertex2;
        int weight = 0;

};
```

Классы вершины и ребра. Вершина — имя(a,b,c …) - номер вершины
Ребро — 2 вершины и вес ребра

```
setlocale(LC_ALL, "ru");
int sum = 0;
Edge* edge_arr = new Edge[1000];
int j = 0, i = 0;
string vertex1, vertex2;
int weight;
int size, vertex_counter = 0 , edge_counter;
int number = 1;

// Input number of edges
cout << "input number of edges" << endl;    //
cin >> size;                                  //
edge_counter = size;                           //
Vertex* vrtx_arr_with_duplic = new Vertex[1000];          //
                                              //
//////////////////////////////////////////////////
```

Создается массив ребер и массив вершин(с повторениями)
Считывается кол-во ребер;
vertex1 vertex2 – имена вершин

```
cout << "input edges info" << endl;                          //
while (edge_counter != 0)                                       //
{                                                               //
        cin >> vertex1 >> vertex2 >> weight;                        //
        cout << endl;                                               //
        edge_arr[j].vertex1.name = vertex1;                         //
        vrtx_arr_with_duplic[i].name = edge_arr[j].vertex1.name;    //
        i++;                                                        //
        edge_arr[j].vertex2.name = vertex2;                         //
        vrtx_arr_with_duplic[i].name = edge_arr[j].vertex2.name;    //
        i++;                                                        //
        edge_arr[j].weight = weight;                                //
        j++;                                                        //
        edge_counter--;                                             //
}                                                               //
```

Считываются ребра и вершины. Данные заносятся в соответствуюие массивы

```
//deleting 0 from vertex array                          //
                                                        //
Vertex* vrtx_arr = new Vertex[1000];                    //
string c;                                               //
int k = 0;                                              //
for (i = 0; i < size * 2; i++)                          //
{                                                       //
        c = vrtx_arr_with_duplic[i].name;                   //
        if (c != " ")                                       //
        {                                                   //Φ
                vrtx_arr[k].name = c;                           //
                vrtx_arr[k].number = k + 1;                     //
                k++;                                            //
                vertex_counter++;                               //
        }                                                   //
        for (j = i; j < size * 2; j++)                      //
        {                                                   //
                if (c == vrtx_arr_with_duplic[j].name)          //
                {                                               //
                        vrtx_arr_with_duplic[j].name = " ";     //
                }                                               //
        }                                                   //
                                                            //
}                                                       //
                                                        //
```

vrtx_arr – массив вершин без повторений. Формируется соответствующий массив вершин без повторов.

```
delete[] vrtx_arr_with_duplic;
```

Удаление массива вершин с повторами. Больше он не нужен.

```
for (int k = 0; k != vertex_counter; k++)
{
        cout << vrtx_arr[k].name << " " << vrtx_arr[k].number << endl;
}
*/
```

Вывод массива вершин без повторов. (Имя и номер вершины)

```
for (int i = 0; i < size; i++)
{                                       //
        for (int j = 0; j < vertex_counter; j++)
        {                               //
                if (edge_arr[i].vertex1.name == vrtx_arr[j].name)               //
                {                                                               //
                        edge_arr[i].vertex1.number = vrtx_arr[j].number;            //
                }                                                               //
                if (edge_arr[i].vertex2.name == vrtx_arr[j].name)               //
                {                                                               //
                        edge_arr[i].vertex2.number = vrtx_arr[j].number;            //
                }                                                               //
        }                                                                   //
}                                                                       //
```

Пробегаем по массивам ребер и вершин и присваиваем вершинам связанным по ребрам их номера из массива vrtx_arr.

```
    // Creating adjacency matrix


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

    for (int i = 0; i < vertex_counter; i++)
    {
            for (int j = 0; j < vertex_counter; j++)
            {
                    adjacency_matrix[i][j] = 0;
            }
    }
```

Инициализация матрицы смежности.

```
for (int i = 0; i < vertex_counter; i++)
{
	for (int j = 0; j < vertex_counter; j++)
	{
		for (int k = 0; k < size; k++)
		{
			if (i + 1 == edge_arr[k].vertex1.number && j + 1 == edge_arr[k].vertex2.number)
			{
				adjacency_matrix[i][j] = edge_arr[k].weight;
			}
			if (i + 1 == edge_arr[k].vertex2.number && j + 1 == edge_arr[k].vertex1.number)
				adjacency_matrix[i][j] = edge_arr[k].weight;
			if (j + 1 == edge_arr[k].vertex1.number && i + 1 == edge_arr[k].vertex2.number)
			{
				adjacency_matrix[i][j] = edge_arr[k].weight;
			}
			if (j + 1 == edge_arr[k].vertex2.number && i + 1 == edge_arr[k].vertex1.number)
				adjacency_matrix[i][j] = edge_arr[k].weight;
		}
	}
}
```

Заполнение матрицы смежности. Проверяем связаны ли вершины с номерами i и j. Если связаны, записываем в ячейку adjacency_matrix[i][j] вес связывающего их ребра.

```
for (int i = 0; i < vertex_counter; i++)
{
	delete adjacency_matrix[i];
 }

delete adjacency_matrix;
```

Удаляем матрицу смежности, тк в дальнейшем алгоритме не участвует.(Просто требование, можно вывести при желании как двумерный массив)

```
Edge temp;

for (int i = 1; i < size; i++)
{
	for (int j = i; j > 0 && edge_arr[j - 1].weight > edge_arr[j].weight; j--)
	{
		temp = edge_arr[j - 1];
		edge_arr[j - 1] = edge_arr[j];
		edge_arr[j] = temp;
	}
}
```

Сортируем массив ребер по их весу в порядке возрастания. Сортировка вставкой

```
init_dsu();
```

Создаем массив arr1 (родителей вершины, с номером I) и arr2(ранги вершин)

```
for (int i = 0; i < size; i++)
{
        if (merge(edge_arr[i].vertex1.number, edge_arr[i].vertex2.number))
        {
                cout << edge_arr[i].vertex1.name << " " << edge_arr[i].vertex2.name << endl;
                sum += edge_arr[i].weight;
        }
}
```

Реализация самого алгоритма. Бежим по массиву ребер. Если вершины связанные ребром, пока пренадлежат к разным компонентам связности, мы их склеиваем. Sum – суммарный вес ребер, используемых для связности графа.