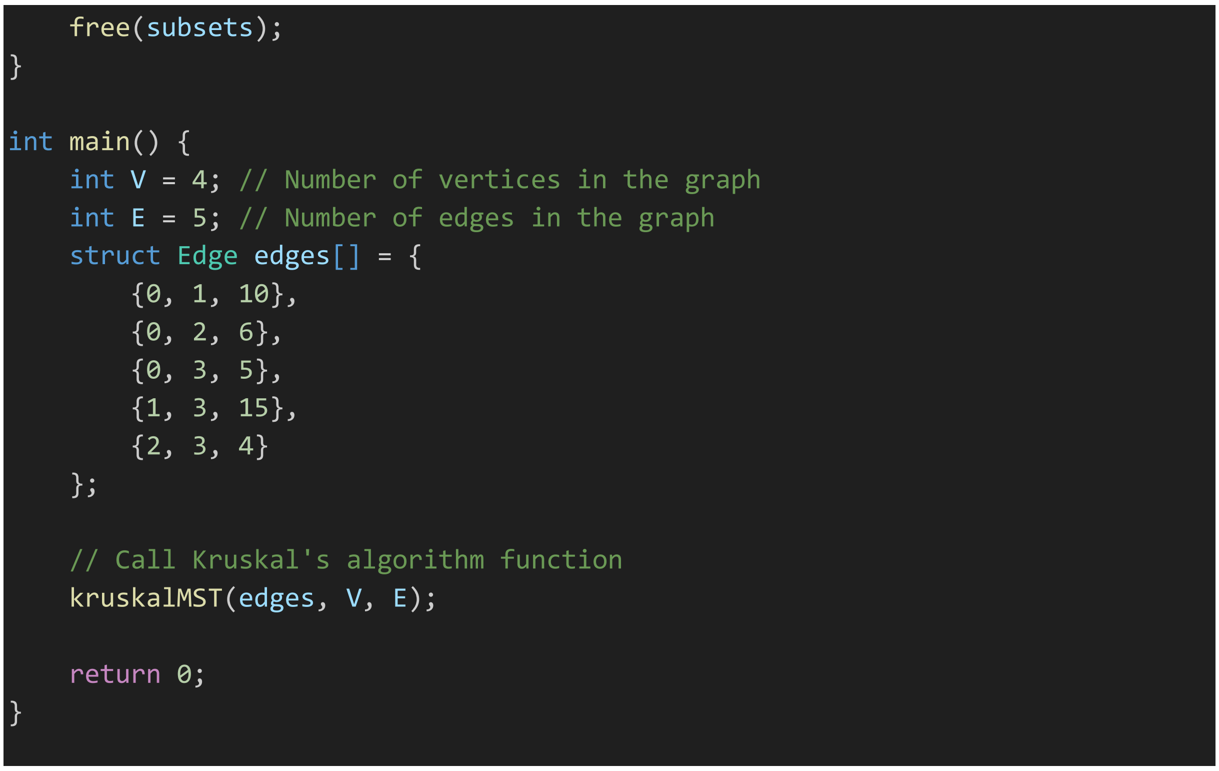
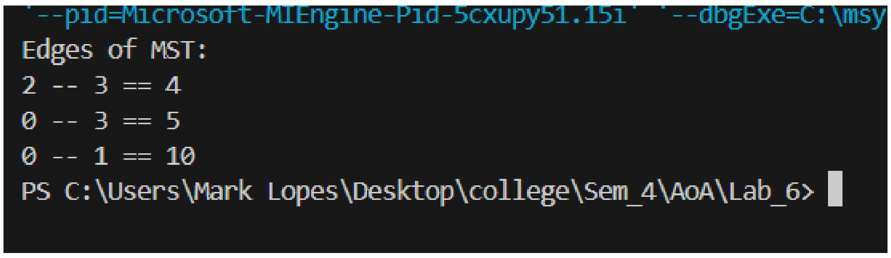
Kruskal’s algorithm:-

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

// Structure to represent an edge in the graph
struct Edge {
    int src, dest, weight;
};

// Structure to represent a subset for union-find
struct Subset {
    int parent;
    int rank;
};

// Function prototypes
int find(struct Subset subsets[], int i);
void unionSet(struct Subset subsets[], int x, int y);
int comparator(const void* a, const void* b);
void kruskalMST(struct Edge* edges, int V, int E);

// Find set of an element i (uses path compression technique)
int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// Perform union of two sets
void unionSet(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        // If ranks are same, then make one as root and increment its rank by
one
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Comparator function for sorting edges by weight
```

```c
int comparator(const void* a, const void* b) {
    struct Edge* edge1 = (struct Edge*)a;
    struct Edge* edge2 = (struct Edge*)b;
    return edge1->weight - edge2->weight;
}

// Kruskal's algorithm to find Minimum Spanning Tree of a given graph
void kruskalMST(struct Edge* edges, int V, int E) {
    struct Edge result[V]; // To store the resultant MST
    int e = 0; // Index variable for result[]
    int i = 0; // Index variable for sorted edges

    // Sort all edges in non-decreasing order of their weight
    qsort(edges, E, sizeof(edges[0]), comparator);

    // Allocate memory for creating V subsets
    struct Subset* subsets = (struct Subset*)malloc(V * sizeof(struct
Subset));

    // Initialize subsets
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < E) {
        // Pick the smallest edge
        struct Edge next_edge = edges[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge doesn't cause a cycle, include it in result
and increment the index
        if (x != y) {
            result[e++] = next_edge;
            unionSet(subsets, x, y);
        }
    }

    // Print the edges of MST
    printf("Edges of MST:\n");
    for (i = 0; i < e; i++)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest,
result[i].weight);

    // Free dynamically allocated memory
```



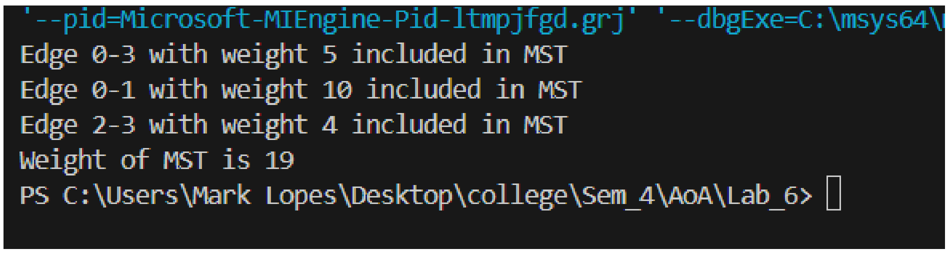
```c
    free(subsets);
}

int main() {
    int V = 4; // Number of vertices in the graph
    int E = 5; // Number of edges in the graph
    struct Edge edges[] = {
        {0, 1, 10},
        {0, 2, 6},
        {0, 3, 5},
        {1, 3, 15},
        {2, 3, 4}
    };

    // Call Kruskal's algorithm function
    kruskalMST(edges, V, E);

    return 0;
}
```



```
 '--pid=Microsoft-MIEngine-Pid-5cxupy51.15i' '--dbgExe=C:\msy
Edges of MST:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
PS C:\Users\Mark Lopes\Desktop\college\Sem_4\AoA\Lab_6>
```

Boruvka’s algorithm:-

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

// Structure to represent an edge in the graph
struct Edge {
    int src, dest, weight;
};

// Structure to represent a graph
struct Graph {
    int V, E; // Number of vertices and edges in the graph
    struct Edge* edge; // Array of edges
};

// Function to create a graph with V vertices and E edges
struct Graph* createGraph(int V, int E) {
    // Allocate memory for the graph structure
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    // Set the number of vertices and edges
    graph->V = V;
    graph->E = E;
    // Allocate memory for the array of edges
    graph->edge = (struct Edge*)malloc(E * sizeof(struct Edge));
    return graph;
}

// A utility function to find the set of an element i
int find(int parent[], int i) {
    if (parent[i] == i)
        return i;
    return parent[i] = find(parent, parent[i]); // Path compression
}

// A function that does union of two sets of x and y
void unionSet(int parent[], int rank[], int x, int y) {
    int xroot = find(parent, x);
    int yroot = find(parent, y);

    // Attach smaller rank tree under root of high rank tree (Union by Rank)
    if (rank[xroot] < rank[yroot])
        parent[xroot] = yroot;
    else if (rank[xroot] > rank[yroot])
        parent[yroot] = xroot;
    else {
        parent[yroot] = xroot;
        rank[xroot]++;
    }
}

// Boruvka's algorithm to find Minimum Spanning Tree of a given graph
void boruvkaMST(struct Graph* graph) {
    int V = graph->V; // Number of vertices
    int E = graph->E; // Number of edges
    struct Edge* edge = graph->edge; // Array of edges

    // Allocate memory for parent, cheapest, and rank arrays
    int* parent = (int*)malloc(V * sizeof(int));
    int* cheapest = (int*)malloc(V * sizeof(int));
    int* rank = (int*)malloc(V * sizeof(int));

    // Initialize sets
    for (int i = 0; i < V; i++) {
        parent[i] = i;
        rank[i] = 0;
        cheapest[i] = -1;
    }

    int numTrees = V; // Number of trees initially equals the number of
vertices
    int MSTweight = 0; // Weight of the minimum spanning tree

    while (numTrees > 1) {
        // Traverse through all edges and update the cheapest of every
component
        for (int i = 0; i < E; i++) {
            int set1 = find(parent, edge[i].src);
            int set2 = find(parent, edge[i].dest);

            if (set1 != set2) {
                if (cheapest[set1] == -1 || edge[cheapest[set1]].weight >
edge[i].weight)
                    cheapest[set1] = i;

                if (cheapest[set2] == -1 || edge[cheapest[set2]].weight >
edge[i].weight)
                    cheapest[set2] = i;
            }
        }

        // Consider the above picked cheapest edges and add them to MST
        for (int i = 0; i < V; i++) {
            if (cheapest[i] != -1) {
                int set1 = find(parent, edge[cheapest[i]].src);
                int set2 = find(parent, edge[cheapest[i]].dest);

                if (set1 != set2) {
                    MSTweight += edge[cheapest[i]].weight;
                    unionSet(parent, rank, set1, set2);
                    printf("Edge %d-%d with weight %d included in MST\n",
edge[cheapest[i]].src, edge[cheapest[i]].dest, edge[cheapest[i]].weight);
                    numTrees--;
                }
            }
        }

        // Reset cheapest array
        for (int i = 0; i < V; i++)
            cheapest[i] = -1;
    }

    printf("Weight of MST is %d\n", MSTweight); // Print the weight of the
minimum spanning tree
}

int main() {
    // Example graph represented as a matrix
    int graph[4][4] = {
        {0, 10, 6, 5},
        {10, 0, INT_MAX, 15},
        {6, INT_MAX, 0, 4},
        {5, 15, 4, 0}
    };
    int V = 4; // Number of vertices
    int E = 5; // Total edges

    // Create a graph
    struct Graph* g = createGraph(V, E);

    // Fill the graph with edges from the matrix
    int edgeCount = 0;
    for (int i = 0; i < V; i++) {
        for (int j = i + 1; j < V; j++) {
            if (graph[i][j] != 0 && graph[i][j] != INT_MAX) {
                g->edge[edgeCount].src = i;
                g->edge[edgeCount].dest = j;
                g->edge[edgeCount].weight = graph[i][j];
                edgeCount++;
            }
        }
    }
    // Run Boruvka's algorithm
    boruvkaMST(g);

    return 0;
}
```



```
 '--pid=Microsoft-MIEngine-Pid-ltmpjfgd.grj' '--dbgExe=C:\msys64\
Edge 0-3 with weight 5 included in MST
Edge 0-1 with weight 10 included in MST
Edge 2-3 with weight 4 included in MST
Weight of MST is 19
PS C:\Users\Mark Lopes\Desktop\college\Sem_4\AoA\Lab_6> 
```