Problem 1:

a)

We can use Dijkstra's algorithm to find the shortest route from each of the towns to the distribution center.

Vertex	Distance	Parent	Shortest path to G
A	12	C	G - E - D - C - A
B	6	H	G - H - B
C	8	D	G - E - D - C
D	5	E	G - E - D
E	2	G	G - E
F	8	G	G - F
G	0	Starting	G
H	3	G	G - H

b)

```
function optimal_location(city[][], t): //city is graph of city, t number of towns

    optimal = -1
    sum = infinity //initilize all distances to infinity

    //find the optimal locations
    for i = 0 to t-1:
        currSum = 0
        distances[] = Dijkstra(city[][], t, i) //our source vertex will be i
        edges = length(distances)

        //find the sum of all edges
        for j=0 to edges - 1:
            currSum += distances[j]

        if sum > currSum:
            sum = currSum
            optimal = i
```

```
    return optimal
```

Time complexity:
Since we need to run Dijkstra's algorithm t times, the final runtime will be **O(towns*edges + towns^2(log(towns))),** or **O(ve+v^2(log v))**

c)
Town **D** is the optimal location for a distribution center.

d)

```
function two_locations(city[][], t): //city is graph of city, t number of towns

    optimal = -1
        furthest_town // This will store the vertex of the furthest town
    sum = infinity //initilize all distances to infinity
        //find the optimal locations
    for i = 0 to t-1:
        max_distance = 0; //This will be updated to the max town from optimal vertex
                currSum = 0

        distances[] = Dijkstra(city[][], t, i) //our source vertex will be i
        edges = length(distances)

        //find the sum of all edges
        for j=0 to edges - 1:
            currSum += distances[j]

        if sum > currSum:
            sum = currSum
            optimal = i
        else if currSum > max_distance:
            furthest_town = i

return optimal and furthest_town
```

This has same runtime as algorithm in b, O(ve+v^2(log v))

e)
G and A

Problem 2:
For this problem, I will be implementing Prim's algorithm using an adjacency matrix (2D array) for the data structure. I'm using an adjacency matrix because since our graph will be a complete one, using a heap will not be as effective. Adjacency matrices are also much easier to implement with Prim's algorithm.

The algorithm we use is as follows:
We create a set that will keep track of all of the vertices in our MST. We then initialize key values to each vertice in the graph, starting at infinity. We set our root vertex to 0 as it's the first point. While the MST set doesn't contain every vertice, we pick a vertex that is not in the MST set and has a minimum key value, we put that vertex in the set, and then update the key values of all adjacent vertices of that vertex. We continue this until all vertices are in the MST set.

The running time of this algorithm as I have implemented it is **O(C * |v|^2),** where **C** is the number of test cases, and **|v|** is our number of vertices.

```
function prims():
    parent[V]
    key[V]
    MSTset[V]

    //init all keys as infinite
    for each key:
        key = infinity
        MSTset[i] = false

    //include first vertex in set, init it to 0
    key[0] = 0
    partent[0] = -1

    //Loop through each vertex until all vertices have been added to the set
    for each count up to vertex - 1:
        next_key = minKey(key, MSTset)

        //add picked value to set
        MSTset[next_key] = true

        //update key values and parent index of all adjacent vertices
        for each vertex:
            if(graph[next_key][v] AND MSTset[v] == false AND graph[next_key][v] < key[v]):
                parent[v] = next_key //update parent
                key[v] = graph[next_key][v] //update key value
```

```
//This function finds the vertex with the minimum key value
function minKey(key[], MSTset[]):
    min = MAX //Initialize the min value
    for each vertex in our graph:
        if(MSTset[v] is false and key[v] < min):
            min = key[v]
            min_index = v;
    return min_index
```