1.

1

6

2

2

1

1

2

4

4

1

2

Above is the undirected graph that I graphed according to the question. There are total of 4 minimum spanning trees that can be drawn from the undirected graph above.

(1)

1

2

1

1

2

1

(2)

1

2

1

1

2

1

(3)

1

2

1

1

1

2

(4)

1

2

1

1

1

2

In total, the minimum spanning tree has the weight of 8.

There are these four minimum spanning trees according to the definition of minimum spanning trees. Minimum spanning trees should include edges that have the minimum weight, which in this case is edges with weight 1. There are total of four edges that have weight of 1 in this undirected graph, which are edge that connects nodes a and b, edge that connects node b and c, edge that connects d and f, edge that connects e and g. If we must include these four edges, we can categorize the tree into three different subsets, the subset a and b and c, f and d, and e and g, as shown below.

1

1

1

1

Since spanning trees require the nodes to be connected, we need to connect those three subsets by adding the edges with lowest weights. The next lowest weights are edges that have weight of 2. Now, if we can connect these three subsets with the edge with weight 2, we can make the minimum spanning tree. Luckily, there are two edges with weight 2 that connect the subset a,b,c with subset d,f, which are the edge that connects b and d, and the edge that connects a and f. In addition, there are edges that connect the subset d,f and subset e,g, which are edge that connects d and e and edge that connects f and e. Therefore, if we can connect those three subsets with two edges, we have four possible choices (two ways to go from subset a,b,c to d,f and two ways to go from subset d,f to e,g – 2\*2=4), which are the four minimum spanning trees that I drew above.

2.

(a)

The algorithm 2 takes Omega(n^2) steps in special case, where we start the vertex in one corner with the lowest height and local maximum lies at the other corner. In this case, it takes n^2 runtime. Below is an example.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 8 | 7 | 6 | 5 |
| 9 | 10 | 11 | 12 |
| 16 | 15 | 14 | 13 |

Here, if we start from the corner that has the value of 1, and start comparing the height of it with the vertex at the right, we see that the height of 2 is greater than 1. Therefore, we move to the next vertex that has height of 2. At vertex that has height of 2, we see that the vertex on the right has height of 3, which is greater than 2, so we need to move to the vertex that has height of 3. With this pattern, we move to vertex that has height of 4. At vertex with height 4, there is a vertex that has height 5 below it, moving to the vertex with height of 5. Continuing to compare with the vertex at the left, we need to move all the way to the vertex with height of 8. At vertex with height 8, we move to 9 and at vertex with height 9, we move to vertex with height 12 and eventually we reach the vertex with height 16. At vertex with height 16, we still need to compare with all its neighbors, and since all the neighbors have height less than 16, 16 is the local maximum. Here, it took n^2 times because n^2 = 4^2 = 16 and we moved total of 16 times after comparing, to find the local maximum of the grid.

(b)

Total:

T(n2) = T(n2/2) + n

There are total of n2 elements that we are looking at, so we take T(n2) to begin from. According to the algorithm, this algorithm looks at the grid horizontally and then vertically. Since there are n horizontal lines, we need to add n since we are looking at the grid each line by line. Afterwards, the square is cut into sizes n/2 by n/2 from n by n to find the local maximum and we repeat this until the square has size 1. Therefore, the question is reduced into T(n2/2) from T(n2).

(c)

T (n2)<= T(n2/2) + O(n)

<= 2 \* T(n2/4) + O(n/2) + O(n)

<= 4 \* T(n2/8) + O(n/4) + O(n/2) + O(n)

<= 8 \* T(n2/16) + O(n/8) + O(n/4) + O(n/2) + O(n)

<= ….

<= 2k-1 \* T(n2/2k) + O(n) \* (1-0.5n)/(1-0.5) = 2k-1 \* T(n2/2k) + O(n) \* (1-0.5k)/0.5

We repeat this until it reaches to the square with size 1, which is the base case. T(1) for the base case is 0 (T(1) = 0), so T(1) = T(n2/2k) gives us n2 = 2k, giving k = log(n2)(log of base 2), which is equal to k = 2logn. When we substitute 2logn for k,

T(n) <= 22log(n)-1 \* T(n2/22log(n)) + (1 – 0.52logn) / 0.5 \* O(n)

<= O(n)

Since (1 – 0.52logn) / 0.5 is simply a constant, T(n) is cO(n), where c is the value of (1 – 0.52logn) / 0.5. When calculating the runtime, we can eliminate the constant in front of the big oh notation, giving us O(n) of runtime. Therefore, according to this proof, the runtime is O(n).