1. When a graph is represented with a adjacency matrix looping through all neighbors of a given vertex takes $\Theta(n)$ time. In a DFS we are going to do this once for every vertex. Thus, the runtime is $\Theta(n^2)$.

2. There was much confusion on this question. Full points will be given to answers that look like a traversal of a graph.

3. To compute the center(s) I will use the fact that removing all the leaves (degree one vertices) of a tree does not change the center, assuming it is not the case that only leaves remain. To see this notices that a longest path from a given node must end at a leaf. If this is not the case, it can be made longer my extending the path to a leaf. So removing all the leaves from a tree decreases every nodes eccentricity by one, which means it will not change the center(s) of the tree.

The algorithm then is to first remove all the leaves of the tree. The from the resulting graph remove all the leaves, and continue this process until only leaves remain or the graph is a single vertex. What sort of trees can have only degree one vertices? We know that

$$n - 1 = m = \frac{1}{2}\sum_{v} \deg(v) = n \quad \text{which implies} \quad n = 2$$

which means a tree of only leaves must have two vertices and one edge connecting them. So either we end up with an isolated vertex or K_2. This answer part b), i.e., a graph can have one or two centers.

The naive implementation of the algorithm described here would give a $O(n^2)$ running time, but we can get this down to $O(n)$ by using a queue.

```
def find_center(G):
  Q = find_leaves(G)  # O(n) time
  while Q:
    v = Q.pop(0)
    if neighbor of v has degree two:
      Q.append(neighbor of v)
    remove v from G
```

4. From each node do BFS four levels deep, recording every vertex you see. This will run in $O(n^2 + nm)$ time.

5. The runtime of Dijkstra's algorithm using a priority queue Q is $O(m \cdot d_Q + n \cdot r_Q)$ where d_Q and r_Q are respectively the time to decrease a key and remove the minimum key for the priority queue Q. For a heap based priority queue we have $d_Q = r_Q = O(\log n)$ giving us the runtime $O((m+n)\log n)$. We can instead use an unsorted list as our priority queue. In this case we can decrease a key in $d_Q = O(1)$ time, as we are just changing the value in an array. But removing the minimum key takes $r_Q = O(n)$ time, as we first need to find the minimum. This gives us a runtime of $O(m + n^2) = O(n^2)$. Not that if m is $\Omega(n^2)$ this is faster that Dijkstra with a heap.

A cool theoretical option would be to use a Fibonacci heap (`https://en.wikipedia.org/wiki/Fibonacci_heap`) which gives us $d_Q = O(1)$ and $r_Q = O(\log n)$ amortized. This gives use the best of both worlds and a runtime of $O(m + n \log n)$, which is optimal in the sparse and dense cases.