1. Strategy: we shall use kruskal's algorithm to find the minimum spanning tree. Also, notice that at the d-th day, if 2 edges have initial costs C_1, C_2, where $C_1 > C_2$, then their purchasing prices at the d-th day are $C_1 \cdot 1.05^{d-1}$ and $C_2 \cdot 1.05^{d-1}$, and we have $C_1 \cdot 1.05^{d-1} > C_2 \cdot 1.05^{d-1}$. Therefore, the less the initial cost is, the less it costs at day d.

Pseudocode:

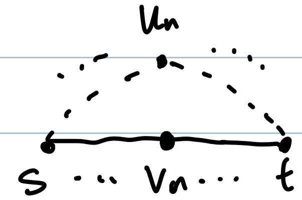
```
Let V = [] # keep tracking of all connected nodes. Make V to be a hashable array   ← O(1)
                                                                  (or a hash table)
let E = [] # keep tracking of all roads we purchased.   ← O(1)
Let H be an empty priority heap   ← O(1)
For each edge e=(u,v) where u,v are 2 hubs:   ← O(m)
    H.insert((c(e), e))   ← O(log n)
# by kruskal:
while ((!H.empty()) && (len(V) != number of all hubs)):   ← O(n)
    (u, v) = H.extract_min()[1]  # the index [1] extracts the edge e, which is (u,v).   ← O(log n)
    if (!V.find(v)||!V.find(u)): # to avoid cycles.   ← O(1), since we let V to be hashable.
        E.append((u,v))   ← O(1)
        if (u ∉ V):   ← O(1)
            V.append(u)   ← O(1)
        if (v ∉ V):   ← O(1)
            V.append(v)   ← O(1)
return E.reverse()   ← O(1)
```

Now E contains the edges we need to purchase, where the edge at index i should be purchased at day i.

And in total, the Algorithm above is $O(m \log n)$.

2. (a) We can use kruskal's algorithm. Each time we add a new edge with maximum bandwith available to our maximum spanning tree. And after the graph containing the maximum spanning tree is constructed, we let s be the root, and perform BFS to find a path to t. Since a tree is acyclic, then this path from s to t is unique.

Using kruskal is valid here because every time we are adding the available path with the maximum bandwith, which would not create a cycle. So, if s and t are connected by a path $P=(s,(s,v_1),v_1,\cdots,v_i,\cdots,v_n,(v_n,t),t)$ in the maximum spanning tree, and there is some other path $q=(s,(s,u_1),u_1,\cdots,u_i,\cdots,u_n,(u_n,t),t)$ not in the maximum spanning tree, then $\inf\{\text{bandwith}(e) \mid e\in P\} \geq \inf\{\text{bandwidth}(e) \mid e\in q\}$. ①



assume ① is false, then every edge in q has larger bandwidth than the edge in p with the least bandwith. Also q is acyclic, so before the minimum edge in P is added to the maximum spanning tree, every edge in q should already be in the tree. Then, a cycle would be created, so ① must not be false.
Now we've proved that the kruskal's algorithm is valid here.

Pseudocode:

```
Let V = [] # heap tracking of all connected nodes. Make V to be a hashable array (or a hash table)   ← O(1)
let G = [] # the graph containing maximum tree   ← O(1)
Let H be an empty priority heap   ← O(1)
For each edge e=(u,v) where u,v are 2 hubs:   ← O(m)
    H.insert((b(e), e))   ← O(log n)

# by kruskal:
While ((!H.empty()) && (len(V) != number of all hubs)):   ← O(n)
    (u, v) = H.extract_max()[1]  # the index [1] extracts the edge e, which is (u,v).   ← O(log n)
    if (!V.find(v) || !V.find(u)): # to avoid cycles.   ← O(1), since we let V to be hashable.
        G.add((u,v))   ← O(1)
        if (u ∉ V):   ← O(1)
            V.append(u)   ← O(1)
        if (v ∉ V):   ← O(1)
            V.append(v)   ← O(1)

Perform BFS on G from s, to find a path from s to t, let the path be p.   ← G has n nodes and n-1 edges, so this BFS is O(n)
Return P.   ← O(1)
```

And in total, the Algorithm above is $O(m \log n)$.

(b). Similar to the reason we listed in (a), a kruskal's algorithm can be used here.

Pseudocode:

```
Let V = [] # heap tracking of all connected nodes. Make V to be a hashable array   ← O(1)
                                                            (or a hash table)
let G = [] # the graph containing maximum tree   ← O(1)
Let H be an empty priority heap   ← O(1)
For each edge e=(u,v) where u,v are 2 hubs:   ← O(m)
    H.insert((b(e), e))   ← O(log n)

# by kruskal:
while ((!H.empty()) && (len(V) != number of all hubs)):   ← O(n)
    (u, v) = H.extract_max()[1]  # the index [1] extracts the edge e, which is (u,v).   ← O(log n)
    if (!V.find(v) || !V.find(u)): # to avoid cycles.   ← O(1), since we let V to be hashable.
        G.add((u,v))   ← O(1)
        if (u ∉ V):   ← O(1)
            V.append(u)   ← O(1)
        if (v ∉ V):   ← O(1)
            V.append(v)   ← O(1)
Return G.   ← O(1).
```

And in total, the Algorithm above is $O(m \log n)$.