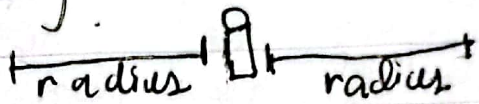
CS 401 Homework 3

(1a.) We can use a greedy algorithm to find the minimum number of base stations.

```
#Parameters:
# houses: A sorted list of integers which is the position of house
#radius : An integer representing how far 5 miles is.
```




```
def solve(houses, radius):
    index = 0
    stations = []  # our answer

    while index < len(houses):
        station = houses[index] + range  # place a station
        stations.append(station)

        #Find the next index of the first uncovered
        # house:
        while(index < len(houses) and
            houses[index] <= (station + range)):
                index += 1
    return stations
```

Explanation: The algorithm / pseudocode above is greedy because we are putting a station at (index + range) away as soon as we find an uncovered house until we have traversed all of our houses.

(16.) The time complexity of this algorithm is $O(n)$.
Essentially, we are only going through the houses array once. Despite having a nested while loop, that loop increments the index.

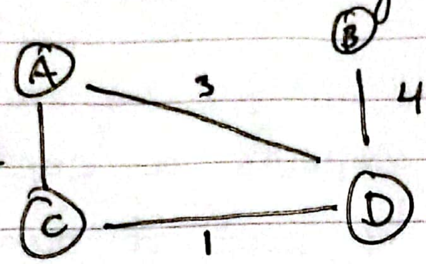
, The algorithm also expects a sorted array so we don't have to sort it ourselves which would make it $O(n \log n)$.

* that said, the function has a linear time complexity $O(N)$ where N is the number of houses.

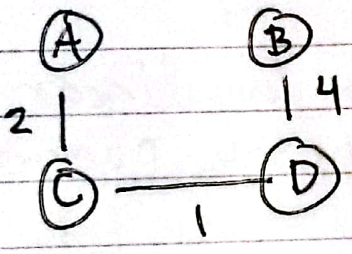
(2a) Is every minimum-bottleneck tree of G a MST of G?

No.

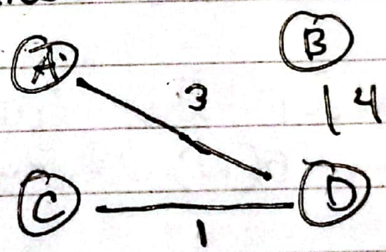
Counterexample; Consider this graph:



All the minimum-bottleneck trees of G are:



↑ This is an MST



↑ This is not an MST

*Both graphs have a bottleneck edge of 4. And, both are a min-bottleneck graph. However, the 2nd graph is not an MST. Hence, the answer is No.

(2b.) Yes, every MST is a minimum bottleneck tree of G.

+ This can be proved by examining how the MST is found. We will look at Kruskal's algorithm.

Essentially, the MST algorithm chooses the edges with the smallest weights that connect all vertices without forming cycles. It minimizes the maximum edge weight in doing so.

- It would not make sense to have a tree with a smaller maximum edge weight because it would have been chosen by the MST algorithm.

Kruskal's Algorithm is a greedy algorithm because it always chooses the minimum weight edge to add. In doing so, it makes sure that the bottleneck edge is minimized.

Thus, every MST is a min-bottleneck tree.

3. We can use a greedy algorithm to solve this question.

1. Loop through each job and assign it with a value.

2. This value is: $weight_i / time_i$

3. Sort it in descending order. based on the value assigned for each job.

* The intuition behind this algorithm is that we are trying to get the most "expensive" job done as quick as possible. to minimize the weighted sum.

4. The current ordering is the most optimal. To get the sum, simply loop through the sorted list and calculate the running sum

4) We can accept as many jobs as possible with an algorithm that runs polynomial in n.

This problem is essentially the typical weighted interval scheduling problem but with a twist where the interval is circular in nature.

We can adopt an algorithm that uses greedy because we are able to isolate the problem to its counterpart by removing the intervals that start before and end after midnight. As a result, we are left with a "non-circular" interval and the solution is trivial

In other words, reduce the problem: For each job i that goes past midnight, solve the reduced list of jobs using greedy where you sort it based on the end times and choose the earliest end time and resolve all conflicts. Repeat until you reach the start of job i. Continue this process for every job and record the maximum number of non overlapping jobs

* The time complexity is $O(N^2)$ because we are doing a linear search for every job that crosses midnight.

(5) We can use Kruskal's Algorithm to find the MST in O(n) time because it's a near-tree.

Step 1: We need to initialize our data structures: to implement Union-Find which Kruskal uses.

* Create parent and rank dictionaries for disjoint sets.

Step 2: Define the Union find functions:
* find() => Find the root node
* union() => Merge two nodes using union by rank.

Step 3: Sort the edges in ascending order (by weight):
* This is O(N) because there are at most n+8 edges which is N.

Step 4: Process our edges:
1. This is the heart of Kruskal's algorithm.
2. For each edge, if find(u) != find(v), then union them because they won't form a cycle.
3. Add the edge to a return list.

Step 5: Return MST

6a. We can achieve this in $O(M+N)$ time where M is the number of edges and N is the number of nodes.

* If we analyze how the MST is created, we built the MST in an ascending order in weight. We can deduce the algorithm needed to check whether an edge is in the MST.

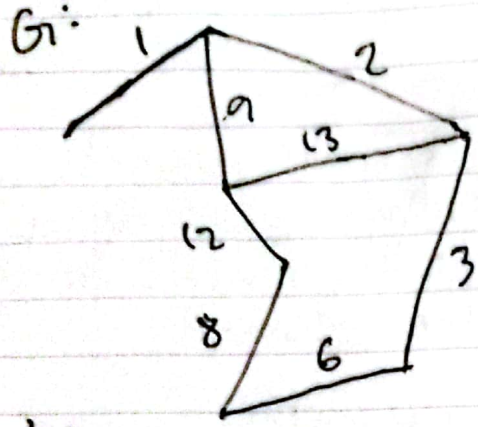
Algorithm: # this is in pseudocode

```
def solve(G, e):
    vicited = set()
    def dfs(node, target):
        if node == target: return True
        visited.add(node)
        for neighbor in G.neighbors(node):
            if neighbor not in visited and
                current edge < e.edge:
                return dfs(neighbor, target)
        return False
    reached = dfs(e.u, e.v)
    return not reached.
```

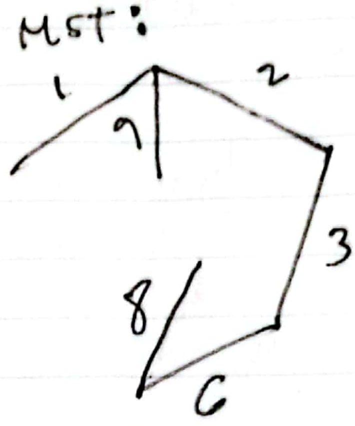
* Essentially, an edge is in MST if we can't reach node u from v.

(6b.) Proof of correctness:

G:



MST:



Case 1:

If we were able to reach node v from u, then that means there was a cycle. This path also means that the edge e is the max weight in our traversal.
By the MST property, e cannot be the edge.

Case 2:

If we weren't able to reach node v from u, then that means that edge e is the lightest edge that connects u→v. Thus, it's a part of the MST as it satisfies the property.

In all cases, we were able to determine the correctness of this algorithm.