1. **Solution:** (a) We need to prove that the local optimum tree is also the (unique, since weights are different) MST.
Let MST be tree T and let tree T' be any local optimum tree. As shown in class, for a graph with distinct edge weights, the set of all safe edges forms the unique MST T.
Case 1: T and T' are the same.
Nothing to prove here, T' is an MST as we set out to prove.
Case 2: T and T' differ at atleast one edge.
Let $e = (u, v)$ be an edge present in T but not in T'. Consider the path $p_{T'}(u, v)$. Since T' is a local optimum tree, we have the property that every edge e' in $p_{T'}(u, v)$ has a lower edge weight than e (i.e. $w(e') < w(e)$), since otherwise, we can replace e' by e to get a lower weight tree.
Now, we prove that e cannot be a safe edge. Consider any partition of the vertices into S and $V \backslash S$, such that e rosses the partition (i.e. $u \in S$ and $v \in V \backslash S$). Then, there exists some other edge $e'' \in p_{T'}(u, v)$ with one end in S and other in $V \backslash S$. (Note that $p_{T'}(u, v) \cup e$ forms a cycle and since e crosses the partition, some other edge should also cross the partition). But as seen before, $w(e'') < w(e)$. Thus, e'' crosses the partition and has a lower weight. This is implies that e is not the minimum weight edge across any partition.
Thus, e is not a safe edge. But then, e cannot be present in the MST T. Contradiction.

(b) We first check that the graph is connected by running DFS. If not, we return false. Otherwise, we can run the Boruvka's Algorithm for $\log \log n$ iterations, shrink each connected component in the forest formed, to a single vertex, and then run Prim's Algorithm on the resulting graph.
By shrinking the connected component, we mean creating a new graph, where for each connected component, we have one vertex. We add an edge from one connected component c_1 to connected component c_2, if there exists an edge between a vertex in c_1 to a vertex in c_2. The weight of the edge between c_1 and c_2 would be the minimum weight of all edges from some vertex in c_1 to some vertex in c_2.

Algorithm:
- Run DFS to ensure graph is connected. If not, return "No MST".
- Run Boruvka's Algorithm for $\log \log n$ iterations. Let the partial forest formed by T.
- Run DFS on T to find all connected components and create the connected component graph as described above. Let the new graph formed be $G' = (V', E')$
- Run Prim's Algorithm on G' with same weights for edges as in G. Let the tree formed be T'.
- Return $T \cup T'$.

Boruvka's Algorithm runs for $\log \log n$ iterations, each iteration taking $O(m)$ time. Thus, it runs in $O(m \log \log n)$ time.
Finding connected components and shrinking to a single vertex can be done using a single DFS, and takes O(m+n) time.
In every iteration, in the worst case, Boruvka's algorithm halves the number of connected components. Hence, after $\log \log n$ iterations, the number of connected components remaining would be $\frac{n}{2^{\log \log n}} = \frac{n}{\log n}$. This would be the number of vertices in the graph we run Prim on. The number of edges would still be O(m). Thus, Prim's

algorithm would now take $O\left(m + \frac{n}{\log n}\log\left(\frac{n}{\log n}\right)\right) = O(m + \frac{n}{\log n}\log(n)) = O(m+n)$ time.
Thus, the total time taken would be $O(m \log\log n)$ as required, since in a connected graph, we must have $m \geq n-1$

■

Rubric: (a) Part (a) 5 points

- -5 for proving every MST is a local optimum tree.
- Atmost -2 for minor mistakes in syntax etc.

(b) Part (b) 5 points

- 2 points for correct number of Boruvka's algorithm loops followed by Prim's Algorithm
- 1 point for using DFS to compute the connected components and shrinking the vertices appropriately.
- 2 points for runtime analysis.
- Do not deduct points for not checking for connected graph.

2. **Solution:** (a) Let $\mathsf{Stops}_n(i)$ be the minimum number of stops needed starting at station i and ending at station n not counting station n as a stop. We give an algorithm $\mathsf{GreedyStops}(D[i..n])$ to compute $\mathsf{Stops}_n(i)$. The minimum number of refueling stops needed for the entire trip is then $\mathsf{Stops}_n(1)$, which can be obtained by calling $\mathsf{GreedyStops}(D[1..n])$.

```
GreedyStops(D[i..n]):
  if i = n then
    return 0
  else
    k ← max{k | i < k ≤ n and D[k] − D[i] ≤ 100}
    if no such k exists then
      return ⊥ (infeasible instance)
    end if
    return 1 + GreedyStops(D[k..n])
  end if
```

Note that if any recursive call returns $\perp$, the first call returns $\perp$. We prove that if there exists a feasible solution, then $\mathsf{Stops}_n(i) = \mathsf{GreedyStops}(D[i..n])$. Then we show that if there does not exist a feasible solution, $\mathsf{GreedyStops}(D[1..n])$ will return $\perp$, which denotes infeasibility.

Claim 1. *Let $n \geq 1$. If there exists a feasible solution, then $\mathsf{GreedyStops}(D[i..n]) = \mathsf{Stops}_n(i)$ for any $1 \leq i \leq n$.*

Proof: We proceed by induction on n.

Base case. Assume $n = 1$, which implies $i = 1$. As the starting station is the same as the destination, the minimum number of stops is 0. As $n = 1$, GreedyStops($D[i..n]$) returns 0. Thus, GreedyStops($D[i..n]$) = $\mathsf{Stops}_n(1)$.

Inductive step. Assume $n \geq 1$. Assume that for $k \leq n$ and for $1 \leq j \leq k$, GreedyStops($D[j..k]$) = $\mathsf{Stops}_k(j)$. Now consider an instance with $n+1$ stations and suppose $1 \leq i \leq n+1$. If $i = n+1$, then we can simply apply the base case. So we only consider $i \leq n$. Let

$$k = \max\{k \mid i < k \leq n+1 \text{ and } D[k] - D[i] \leq 100\}.$$

Such a k must exist as $i \leq n$ and there exists a feasible solution by assumption. Let S be the set of fueling stations chosen by GreedyStops($D[i..n+1]$) and let S' be an optimal set of fueling stations not containing k. (Note $k \in S$.)

Let $S' = \{s_1, \ldots, s_\ell\}$ such that $D[s_1] \leq \cdots \leq D[s_\ell]$. By the choice of k, we have $D[s_1] \leq D[k]$. Since S' is feasible, there must be enough fuel to get from s_1 to s_2, implying $D[s_2] - D[s_1] \leq 100$. As $D[s_1] \leq D[k]$, it follows that $D[s_2] - D[k] \leq 100$. Thus, $S' - s_1 + k$ is also a feasible solution.

By the inductive hypothesis, GreedyStops($D[k..n+1]$) is optimal on stations k to $n+1$. Since k can be included in an optimal solution and GreedyStops($D[k..n+1]$) is optimal, $\mathsf{Stops}_{n+1}(i) = 1 + \text{GreedyStops}(D[k..n+1])$. Finally, as there exists a feasible solution, GreedyStops($D[i..n+1]$) = $1 + \text{GreedyStops}(D[k..n+1])$. This concludes the proof. □

Claim 2. *If there does not exist a feasible solution, then* GreedyStops($D[1..n]$) *returns* $\perp$*, implying that there does not exist a feasible solution.*

Proof: First, we claim there exists i such that $D[i+1] - D[i] > 100$. If this were not the case, for every i, $D[i+1] - D[i] \leq 100$. This would imply that stopping at each station is a feasible solution, which would contradict the assumption.

Suppose that $D[i+1] - D[i] > 100$. If the greedy algorithm reports that there is no feasible solution prior to reaching station i, then the algorithm correctly reports that there is no feasible solution. Suppose the greedy policy takes station i. At this point, the algorithm would find that there is no station within 100 miles of station i and correctly report that there is no feasible solution. □

(b) Let $D = [0, 50, 75, 125, 150]$ and $C = [0, 1, 100, 1, 0]$. The greedy policy described above first refuels at the station furthest from the start. In this case, it refuels at station 3. Then there is enough fuel to finish the trip. As $C[3] = 100$, the cost of the greedy policy for this instance is 100. However, a feasible solution is to stop at stations 2 and 4. As $C[2] = C[4] = 1$, the cost of this solution is 2. Thus, the greedy policy is not optimal in terms of cost.

Rubric: Out of 10 points:
- 8 points for part (a)
  - 4 points for a correct algorithm
  - 4 points for a correct analysis
- 2 points for part (b)
  - 2 points for a correct counterexample

3. **Solution:** (a) We first prove that if L is a recursive language, then L can be generated by a well-behaved generator.

Claim 3. *Let L be a recursive language. Then there is a well-behaved generator G such that $L = L(G)$.*

Proof: As L is recursive, there is some Turing machine M which decides L. Consider the following generator G.

```
G:
  for w ∈ Σ* in lexicographic order:
      if M(w) accepts:
          write w# to the output tape
```

It is clear from the definition of G that the output of G is lexicographically ordered, so G is well-behaved. Recall that M decides L; in particular, M is guaranteed to halt in finite time on any input. Let $w \in \Sigma^\star$. If $w \notin L$, then $M(w)$ rejects, so G will not generate w. If $w \in L$, then $M(w)$ accepts, so G will generate w *assuming G reaches w in its main loop*.

Crucially, for any fixed $w \in \Sigma^\star$, there are finitely many strings occurring before w in lexicographic order. The machine M runs for finite time on each of these strings, so G will only run for finite time on the strings preceding w. In particular, G will reach w in its main loop. Thus G generates a string w if and only if $w \in L$, so G is a well-behaved generator such that $L = L(G)$, which is what we wanted to prove. □

We now prove the other direction: if there is a well-behaved generator G for a language L, then L is recursive.

Claim 4. *Let G be a well-behaved Turing machine generator and let $L = L(G)$. Then L is recursive.*

Proof: Consider the following Turing machine M on input $x \in \Sigma^\star$.

```
M(x):
  while G has not halted:
      continue running G until a new string y is generated
      if x = y:
          ACCEPT
      else if x < y in lexicographic order:
          REJECT
  REJECT
```

Suppose $x \in L$. Then G will generate x in finite time. Moreover, before generating x, G will only generate strings which precede x in lexicographic order. This implies that M will not reject x before G generates x, at which point M will accept, so M correctly accepts x.

Suppose instead that $x \notin L$. If, for all $y \in L$, we have that $y < x$ in lexicographic order, then L must be finite, as only finitely many strings are less than any fixed string x in lexicographic order. Since L is finite in this case, G will eventually generate all strings in L and halt, so $M(x)$ will reject. On the other hand, if there is some $y \in L$ such that $x < y$ in lexicographic order, then G will generate y in finite time, so $M(x)$ will reject. Note that in both cases, M is guaranteed to *halt* on x.

Hence $x \in L$ if and only if M accepts x. Moreover, $M(x)$ always halts, so M decides L, thus L is recursive. □

(b) As before, we prove both directions.

Claim 5. *Let G be a Turing machine generator and let $L = L(G)$. Then L is recursively enumerable.*

Proof: Consider the following Turing machine M on input $x \in \Sigma^\star$.

```
M(x):
  while G has not halted:
      continue running G until a new string y is generated
      if x = y:
          ACCEPT
  REJECT
```

Suppose $x \in L$. Then G will generate x in finite time, so $M(x)$ will accept.

Suppose instead that $x \notin L$. Then G will never generate x, so $M(x)$ will not accept.

Thus M accepts $x \in \Sigma^\star$ if and only if $x \in L$, so M recognizes L. In particular, L is recursively enumerable. □

Claim 6. *Let L be a recursively enumerable language. Then there is a Turing machine generator G such that $L = L(G)$.*

Proof: As L is recursively enumerable, let M be a machine which recognizes L. Intuitively, we would like to mimic the generator construction from the previous problem. However, M is no longer guaranteed to halt, so we cannot simply run M on all strings one-by-one. We get around this by only running M for finitely many steps on any string and slowly increasing the amount of time for which we run M.

Formally, let $w_1, w_2, w_3, \ldots$ be the enumeration of the strings in $\Sigma^\star$ in lexicographic order.[1] Consider the following generator.

```
G:
  for k ← 1 to ∞:
      for i ← 1 to k:
          run M for k steps on w_i
          if M accepts w_i:
              write w_i#
```

Note that in the k^{th} iteration of the outer loop, the generator G will only run M for at most k steps on k strings. This implies that G completes the k^{th} iteration of its outer loop in finite time.

Consider some $x \in L$. Let i be the index of x in the lexicographic order, i.e., choose i such that $w_i = x$. Since $x \in L$ and M recognizes L, we know that M will accept x. Let t be a large enough integer so that M accepts x within t steps, and let $k = \max(i, t)$. Then on the k^{th} iteration of the outer loop, G will run M on $x = w_i$ for t steps and see that M accepts x, so G will print $x\#$ to the output tape. In particular, G generates x. Remark that G will print $x\#$ to the output tape infinitely often. This is not a concern, as it does not change the fact that G generates x.

Conversely, consider some $x \notin L$. Then M never accepts x, so G will never write $x\#$ to the output tape.

Thus, G generates a string x if and only if $x \in L$, so G generates L as desired. □

[1] There is nothing particularly special about lexicographic ordering here; we simply need an order in which every string appears at some finite index in the ordering.

Rubric: Out of 10 points:

- 5 points for part (a)
  - 2.5 points for each direction
- 5 points for part (b)
  - 2.5 points for each direction