Suppose we have \( n \) skiers with heights given in an array \( P[1 .. n] \) and \( n \) skis with heights given in an array \( S[1 .. n] \). Our goal is to assign a ski to each skier, so that the average difference between the height of a skier and their assigned ski is as small as possible. The algorithm should compute an assignment array \( A[1 .. n] \), indicating that each skier \( i \) should be assigned ski \( A[i] \), such that the expression
\[
\frac{1}{n} \sum_{i=1}^{n} |P[i] - S[A[i]]|
\]
is as small as possible.

(a) Describe two input arrays \( P \) and \( S \) that show the following greedy strategy is not always optimal:

**Solution:** Let \( P = \langle 0, 6 \rangle \) and \( S = \langle 4, 10 \rangle \). The greedy strategy will set \( A \leftarrow \langle 2, 1 \rangle \) so that the average distance is \( \frac{1}{2}(2 + 10) = 6 \). However, the assignments \( A = \langle 1, 2 \rangle \) has a smaller average distance \( \frac{1}{2}(4 + 4) = 4 \).

**Rubric:** 3 points total.

(b) Describe and analyze an efficient greedy algorithm that does minimize the average difference between the height of each skier and their assigned ski.

**Solution:** We will use the following greedy strategy:

Let \( i \) be the lowest skier and \( j \) be the lowest ski. Assign ski \( j \) to skier \( i \) and recursively assign the remaining skis to the remaining skiers.

We can implement this strategy as an \( O(n \log n) \) time algorithm by simply sorting both \( P \) and \( S \) and setting \( A \) to be the numbers \( 1 \) through \( n \) in order (here, the assignments are using the final indices of the skis and skiers after sorting).

For simplicity in proving correctness, assume we have already sorted \( P \) and \( S \). We mainly need to argue that there exists an optimal assignment assigning ski 1 to skier 1. For any assignment \( A \), let \( \text{cost}(A) := \frac{1}{n} \sum_{i=1}^{n} |P[i] - S[A[i]]| \). Now, let \( A \) specifically be an optimal assignment of skis to skiers. If \( A[1] = 1 \), we are done. Otherwise, let \( i \) and \( j \) be the skier and ski, respectively, such that \( A[i] = 1 \) and \( j = A[1] \). We create a new assignment \( A' \) by modifying \( A \) with a single swap of assigned skis so that \( A'[1] = 1 \) and \( A'[i] = j \). We now argue that \( A' \) is also optimal, proving the claim.

We'll prove the claim assuming \( P[1] \leq S[1] \); the opposite case works symmetrically. Observe \( P[1] \leq S[1] \leq S[j] \). Suppose \( P[i] \leq S[1] \). In this case,
\[
\text{cost}(A') = \text{cost}(A) + \frac{1}{n} ((S[1] - P[1]) + (S[j] - P[i]) - (S[j] - P[1]) - (S[1] - P[i]))
\]
\[= \text{cost}(A)\]
Suppose $S[1] < P[i] \leq S[j]$. In this case,
\[
\text{cost}(A') = \text{cost}(A) + \frac{1}{n} ((S[1] - P[1]) + (S[j] - P[i]) - (S[j] - P[1]) - (P[i] - S[1]))
\]
\[
= \text{cost}(A) + \frac{1}{n} (2S[1] - 2P[i])
\]
\[
< \text{cost}(A)
\]

Finally, if $S[j] < P[i]$,
\[
\text{cost}(A') = \text{cost}(A) + \frac{1}{n} ((S[1] - P[1]) + (P[i] - S[j]) - (S[j] - P[1]) - (P[i] - S[1]))
\]
\[
= \text{cost}(A) + \frac{1}{n} (2S[1] - 2S[j])
\]
\[
\leq \text{cost}(A)
\]

In every case, \( \text{cost}(A') \leq \text{cost}(A) \). Assignment \( A' \) is also optimal, and it agrees with \( A'[1] = 1 \).

By induction, our algorithm will optimally assign the skis 2 through \( n \) to skiers 2 through \( n \). That observation along with the previous claim implies our algorithm is optimal. □

**Rubric:** 7 points total: 3 points for the algorithm. 3 points for the proof of correctness. 1 point for running time analysis.
You decide to study where all the Energy Stations are along your route so you can minimize the amount of money you spend on batteries.

(a) Describe and analyze a greedy algorithm that computes the minimum number of batteries you need to purchase to complete your test drive.

Solution: We’ll assume no two consecutive stations are more than 100 miles apart as the trip would be impossible otherwise. We use the following greedy strategy:

Purchase a battery at $D[1]$. Then, drive to the farthest station $i$ at most 100 miles away and recursively plan a trip starting from station $i$.

The strategy can be implemented using the following iterative algorithm:

```plaintext
MINBATTERIES(D[1 .. n]):
    count ← 1
    last ← 1
    for i ← 2 to n
        if $D[i] > D[last] + 100$  \{(Farthest we could reach is i − 1)\}
            count ← count + 1
            last ← i − 1
    return count
```

The algorithm has just the one for loop and runs in $O(n)$ time.

To prove the strategy is correct, we need to argue after the first purchase, an optimal solution does not purchase again until the farthest station that can be reached. Let $X$ be any optimal solution. Let $i$ be the farthest station that can be reached after station 1. If $X$’s next purchase after station 1 is at $i$, we are done. Otherwise, we observe solution $X$ must buy a battery at some station $i' < i$ so that it can reach station $i + 1$. We remove the purchase at station $i'$ and instead buy at $i$. We can still reach the next purchase location after $i'$, if any, because $i > i'$, so our modification results in a still-optimal solution agreeing with our first choice. Inductively, we do the optimal choices after reaching that station as well.

(b) Describe and analyze an efficient algorithm to compute the minimum total cost of the batteries you need to complete your test drive.

Solution: We begin by designing a backtracking approach for the problem. Suppose we reach some station $i$ where station $i + 1$ is at most 100 miles from the last place we purchased a battery. We have the choice of buying or not buying a battery at station $i$, leaving us to
work with the remaining stations, but the next time we must buy a battery is still dependent on the last time we bought one. We'll need to pass in two station indices for recursive subproblems; one index for where to start making decisions and one to say the last time we purchased a battery.

Let \( \text{MinCost}(i, j) \) denote the minimum total cost in battery replacements driving from station \( i \) to station \( n \) assuming the last replacement was at station \( j < i \). For simplicity, we'll let \( A[0] \leftarrow D[1] - 100 \). Pretending our last purchase was at station \( 0 \) when we start at station \( 1 \) is the same as just pretending we have no battery at all at station \( 1 \), so we want to compute \( \text{MinCost}(1, 0) \).

If \( i = n \), we've already completed our trip, implying \( \text{MinCost}(i, j) = 0 \). Suppose \( i < n \) and \( D[i + 1] > D[j] + 100 \). We are forced to buy a battery at station \( i \), and we'll want to minimize our costs afterward, so \( \text{MinCost}(i, j) = C[i] + \text{MinCost}(i + 1, i) \). Finally, suppose \( i < n \) and \( D[i + 1] \leq D[j] + 100 \). We could purchase a battery right away as in the last case. We also have the option of skipping the purchase at station \( i \) making the total cost \( \text{MinCost}(i + 1, j) \). Naturally, we want the better of the two choices.

\[
\text{MinCost}(i, j) = \begin{cases} 
0 & \text{if } i = n \\
C[i] + \text{MinCost}(i + 1, i) & \text{if } i < n \text{ and } D[i + 1] > D[j] + 100 \\
\min \{C[i] + \text{MinCost}(i + 1, i), \text{MinCost}(i + 1, j)\} & \text{otherwise}
\end{cases}
\]

We have \( 1 \leq i \leq n \) and \( 0 \leq j \leq n - 1 \) for our subproblems, so we'll store solutions in an array \( \text{MinCost}[1..n,0..n-1] \). Subproblems always depend on a larger first parameter, so we'll consider \( i \) in decreasing order from \( n \) to \( 1 \) and \( j \) in, say, increasing order from \( 0 \) to \( i - 1 \). There are \( O(n^2) \) subproblems to solve in constant time each, so the algorithm will take \( O(n^2) \) time in total.

```
LEASTCOST(D[1..n], C[1..n]):
for i ← n down to 1
    for j ← 0 to i - 1
        if i = n
            MinCost[i, j] ← 0
        else if D[i + 1] > D[j] + 100
            MinCost[i, j] ← C[i] + MinCost[i + 1, i]
        else
            MinCost[i, j] ← min \{C[i] + MinCost[i + 1, i], MinCost[i + 1, j]\}
return MinCost[1, 0]
```

**Rubric:** 5 points total: 2 points for the recurrence. 2 points for the rest of the algorithm. 1 point for running time analysis.