2086. Minimum Number of Food Buckets to Feed the Hamsters

Medium Greedy String Dynamic Programming Leetcode Link

Problem Description

In this problem, we're given a string `hamsters`, which represents a row of positions that can either contain a hamster, denoted by `'H'`, or be empty, denoted by `'.'`. We need to place food buckets at some of these empty positions. There are two conditions to ensure that a hamster can be fed:

- There must be at least one food bucket immediately to the left (`i - 1`) or immediately to the right (`i + 1`) of the hamster's position.
- We want to place the minimum number of food buckets necessary to feed all the hamsters.

Our task is to find this minimum number of food buckets or determine that it's impossible to feed all hamsters, in which case we return `-1`.

Intuition

The solution approach is to iterate through the string and decide the placement of the food buckets optimally to minimize their number while ensuring all hamsters can access at least one bucket. This optimality comes from the following rule: whenever we have a choice to place a food bucket either to the left or the right of a hamster, we prefer to place it to the right. Doing so might enable us to feed the next hamster with the same bucket, hence reducing the total number.

Here's the approach in detail:

- We iterate through each position in the string. If we find a hamster (`'H'`), we check its neighbouring positions.
- If we have an empty position to the right of the hamster, we always place a bucket there, as it could potentially feed another hamster to the right. We increment the answer by `1` and skip checking the next position by advancing two steps in the iteration.
- If the right side is not available, we check the left side. If it's empty and we haven't already placed a bucket there while placing it for a previous hamster, we place a bucket there and increment the answer by `1`.
- If neither side is available for placing a bucket, we return `-1` as it's impossible to feed this hamster.

The critical insight is that by considering the right side first, we utilize each bucket to its maximum potential, thereby ensuring the least number of buckets is used.

Solution Approach

The solution uses a greedy algorithm, which is a common approach for optimization problems where we aim to make the locally optimal choice at each stage with the hope of finding a global optimum.

Here's a step-by-step walkthrough of the implementation based on the reference solution:

1. Initialize a counter `ans` to `0`, which will hold the minimum number of food buckets required.

2. Start iterating through the string `street` using variable `i` that begins from index `0`.

3. Check if the current character at index `i` is a hamster (`'H'`).

    - If there is a hamster at index `i` and the next index `i + 1` is within bounds and empty (contains `'.'`), place a food bucket at `i + 1`. This is because placing the bucket to the right could potentially also feed a hamster at index `i + 2`.

        - After placing the bucket, increase `ans` by `1` to count the newly placed bucket.
        - Move `i` two positions forward to `i + 2`, as we have already dealt with hamster at `i` and potentially at `i + 1` if there is one.

    - If you can't place the bucket to the right because `i + 1` is out of bounds or not empty, check if the left of the hamster at `i` is available (make sure you're not out of bounds and the index `i - 1` is empty).

        - If a bucket can be placed at `i - 1`, increase `ans` by `1`. There's no need to move `i` forward by two positions in this case since we are only dealing with the current hamster.

    - If neither the left nor the right side of the current hamster can have a bucket placed, it is impossible to feed this hamster, so we return `-1`.

4. If the current character is not a hamster, we simply move to the next index by incrementing `i`.

5. Continue this process until we have iterated over the entire string. The `ans` variable by the end of iteration corresponds to the minimum number of buckets required. We return `ans`.

No additional data structures are used apart from variables for indexing and counting. This makes the algorithm efficient both in terms of time and space complexity, achieving (O(n)) time complexity, where (n) is the length of the `street` string, and (O(1)) space complexity since we only use a fixed number of extra variables.

Example Walkthrough

Let's go through an example to illustrate the solution approach.

Assume the input string representing the row of positions is: "H..HH..H.."

Following the solution approach, we'll determine where to place the food buckets:

1. Starting at index `0`, we see `'H'`, the position contains a hamster.

    - `i + 1` is within bounds and empty, so we place a bucket at `i + 1` (position 1).
    - We increment `ans` by `1` (now `ans` is `1`).
    - We skip the next position by moving two positions forward to index `2`.

2. At index `2`, there's no hamster, just a food bucket, so we move to the next index.

3. At index `3`, again, there's no hamster; thus, we move to index `4`.

4. At index `4`, there's a hamster `'H'`.

    - `i + 1` is within bounds and empty, so we put a bucket at `i + 1` (position 5).
    - We increment `ans` by `1` (now `ans` is `2`).
    - We move two positions forward to index `6`.

5. At index `6`, there's no hamster; we move to index `7`.

6. At index `7`, there's another hamster `'H'`.

    - `i + 1` is within bounds but contains another hamster, so we can't place a bucket there.
    - We look to the left (`i - 1`), but there's already a food bucket there placed for the previous hamster, so no new bucket is needed here.

7. We move to the next index `8` and find it empty. Move to index `9`.

8. At index `9`, there's a hamster `'H'`.

    - `i + 1` is out of bounds, so we look at `i - 1`, which is within bounds and empty.
    - We place a bucket at `i - 1` (position 8).
    - We increment `ans` by `1` (now `ans` is `3`).

Having gone through the string, we needed a minimum of `3` food buckets to feed all hamsters in the given row of positions. Hence, we return `3`.

Python Solution

```python
class Solution:
    def minimumBuckets(self, street: str) -> int:
        # Initialize the number of buckets needed
        bucket_count = 0

        # Initialize the index and get the length of the street
        index, length = 0, len(street)

        # Loop through each character in the street string
        while index < length:
            # If we encounter a house 'H'
            if street[index] == 'H':
                # Check if there is space on the right to place a bucket and avoid out of range error
                if index + 1 < length and street[index + 1] == '.':
                    # Skip the next house as the current bucket will water this house
                    index += 2
                    # Increment the bucket count
                    bucket_count += 1
                # If no space on the right, check for space on the left
                elif index > 0 and street[index - 1] == '.':
                    # We don't skip the index here because the bucket waters the house on the left
                    # Increment the bucket count
                    bucket_count += 1
                else:
                    # If no space available to place a bucket, it's not possible to water the house
                    return -1
            # Move to the next position in street
            index += 1

        # Return the total number of buckets needed after placing them optimally
        return bucket_count

```

Java Solution

```java
class Solution {
    public int minimumBuckets(String street) {
        // Get the length of the street represented as a string
        int streetLength = street.length();
        // Initialize the count of buckets required to water flowers
        int bucketCount = 0;

        // Iterate over each position in the street
        for (int i = 0; i < streetLength; ++i) {
            // Check if there is a house in the current position
            if (street.charAt(i) == 'H') {
                // If there is an empty space next to the house, place the bucket there
                if (i + 1 < streetLength && street.charAt(i + 1) == '.') {
                    ++bucketCount;
                    // Skip the next position as we've already placed the bucket
                    i += 2;
                // If there is an empty space before the house, place the bucket there
                } else if (i > 0 && street.charAt(i - 1) == '.') {
                    ++bucketCount;
                // If there are no empty spaces around the house, it's impossible to water
                } else {
                    return -1;
                }
            }
        }

        // Return the minimum number of buckets needed
        return bucketCount;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minimumBuckets(string street) {
        // Variable 'n' holds the size of the street string
        int streetSize = street.size();
        // Initialize 'bucketCount' to count the minimum number of buckets required
        int bucketCount = 0;

        // Iterate through the street string
        for (int i = 0; i < streetSize; ++i) {
            // Check if the current position is a house 'H'
            if (street[i] == 'H') {
                // Check if there is a spot for a bucket to the right of the house
                if (i + 1 < streetSize && street[i + 1] == '.') {
                    // Increase the bucket count and skip the next spot since it's covered by the bucket
                    ++bucketCount;
                    i += 2;
                // Check if there is a spot for a bucket to the left of the house
                } else if (i > 0 && street[i - 1] == '.') {
                    // Increase the bucket count
                    ++bucketCount;
                } else {
                    // If there are no spots to place a bucket around the house, return -1 indicating it's not possible
                    return -1;
                }
            }
        }

        // Return the total minimum number of buckets required
        return bucketCount;
    }
};

```

Typescript Solution

```typescript
// Function to compute the minimum number of buckets required
function minimumBuckets(street: string): number {
    // Variable 'streetSize' holds the size of the 'street' string
    let streetSize: number = street.length;
    // Initialize 'bucketCount' to count the minimum number of buckets required
    let bucketCount: number = 0;

    // Iterate through the 'street' string
    for (let i = 0; i < streetSize; i++) {
        if (street[i] === 'H') {
            // Check if there is a spot for a bucket to the right of the house
            if (i + 1 < streetSize && street[i + 1] === '.') {
                // Place a bucket to the right of the house, increase the bucket count,
                // and skip the next position because it is covered by the bucket
                bucketCount++;
                i += 2;
            // Check if there is a spot for a bucket to the left of the house
            } else if (i > 0 && street[i - 1] === '.') {
                // Place a bucket to the left of the house, increase the bucket count
                // This does not require skipping a position because the previous
                // position has already been considered
                bucketCount++;
            } else {
                // If there are no spots to place a bucket around the house,
                // it's not possible to water all houses, so return -1
                return -1;
            }
        }
    }

    // Return the total minimum number of buckets required
    return bucketCount;
}

```

Time and Space Complexity

Time Complexity

The given code involves a single loop through the length of the string `street`. Within this loop, the operations are constant time checks and assignments. The loop iterates at most `n` times, where `n` is the length of `street`. In the worst case, every character is visited once, and update operations are constant time `O(1)`. Therefore, the time complexity is `O(n)`.

Space Complexity

The space complexity of the code is `O(1)`. This is because the code only uses a fixed number of extra variables (`ans`, `i`, and `n`) that do not depend on the size of the input `street`. There are no data structures used that grow with the input size. Hence, the space required does not scale with the size of the input.