# 1326. Minimum Number of Taps to Open to Water a Garden



## Problem Description

In this problem, you have a garden represented as a one-dimensional line that starts at point 0 and ends at point `n`. Along this line, there are `n + 1` taps at positions `i` through `n`. Each tap has a range associated with it, given by the array `ranges`. The range `ranges[i]` —which is associated with the tap at position `i`—indicates that this tap can water the garden from position `i - ranges[i]` to `i + ranges[i]`.

Your objective is to figure out the minimum number of taps that need to be opened to water the entire length of the garden (from point `0` to point `n`). If it's not possible to water the whole garden with the given taps, the result should be `-1`.

## Intuition

The problem is essentially about finding the minimum number of intervals (tap ranges) needed to cover a continuous segment (the garden). This is similar to the classic interval covering problem.

To solve this problem, we can use a greedy approach:

1. We start by transforming the problem into an interval covering problem. We'll create an array `last` where each index represents the starting point of an interval in the garden, and the value at each index represents the farthest point that can be watered from this interval. This is done by iterating over the `ranges` array and determining for each tap the leftmost and rightmost points it can water (`i - ranges[i]` and `i + ranges[i]` respectively).
2. We then apply a greedy method to find the minimum number of intervals needed to cover the entire garden.
    - Initialize counters for the current max right end of intervals `mx`, the previous farthest right end `pre`, and the number of taps `ans`.
    - Iterate over the garden from `0` to `n`. At each point, we update `mx` to be the maximum of `mx` and the furthest we can reach from the current position (found in `last[i]`). If at any point `mx` is less than or equal to `i`, it means there is a gap in coverage, and we cannot water the entire garden—return `-1`.
    - Whenever we reach the point `pre` (which is the end of the coverage from the selected taps), it means we need to select a new tap. Increment `ans` and set `pre` to `mx`.
3. By the end of the iteration, `ans` will hold the minimum number of taps needed to water the entire garden.

This approach ensures that at each step, we are extending the watering range as much as possible with the current number of taps, and we only add a new tap when necessary to cover new ground.

## Solution Approach

The solution provided uses a greedy algorithm to minimize the number of taps opened to cover the garden. Here's a detailed walkthrough of the implementation:

1. **Create the `last` array**: A list called `last` is initialized with `0` values and has a length of `n + 1`. This array will hold the maximum distance that can be watered starting from each position. The goal is to use this array to find the best tap to cover the longest distance from each point.
2. **Populate the `last` array**: We loop through the `ranges` array with the index `i` and range `x`. The variables `l` and `r` represent the leftmost (`i - x`) and rightmost (`i + x`) points that can be watered by tap `i`. We need to make sure that `l` does not go below `0`, hence we use `max(0, i - x)`. For each position `l`, we save the farthest point `r` that can be reached, by setting `last[l]` to `max(last[l], r)`. This ensures that for each starting position `l`, we have the tap that waters the farthest.
3. **Iterating through the garden**: We initialize three variables `ans`, `mx`, and `pre` to track the number of taps used (`ans`), the current maximal right boundary (`mx`), and the previous maximal right boundary (`pre`). Now, we iterate through the garden from `0` to `n`.
4. **Select taps and track coverage**: During each iteration, we update `mx` to be the furthest point we can reach from the current point `i`, based on `last[i]`. If at any point `mx` is less than or equal to `i`, it means there is a gap, and the entire garden can't be watered, so we return `-1`. If we reach the `pre` point (where the current taps' coverage ends), it means we need to open another tap to extend the coverage, therefore we increment `ans` and update `pre` to `mx`.
5. **Greedy selection**: Each time we select a tap, we make the greedy choice by picking the tap that extends our coverage to the rightmost point possible. This is done by always updating `mx` with the maximum distance we can cover from `i`. Only when we have to move beyond `pre` do we lock in our choice of a tap and increment `ans` since we know up to that point we have covered with the minimum number of taps.
6. **Returning the answer**: Once the loop is complete, we have either returned `-1` if the garden can't be watered completely, or we have the minimum number of taps required in `ans`, which we return as the final result.

In summary, by transforming the problem into finding the optimal coverage for each interval and then iteratively expanding these intervals using a greedy approach, we efficiently find the minimum number of taps needed to cover the whole garden represented by the x-axis from `0` to `n`.

### Example Walkthrough

Let's illustrate the greedy solution approach with a small example. Suppose our garden is represented by a line of length `n = 5` and we have 6 taps positioned at `0` through `5`. The `ranges` array provided is `[1, 0, 2, 1, 2, 0]`, representing the range each tap can water on both sides.

**Step-by-Step Solution:**

1. **Create the `last` array**: We create an array `last` of length `n + 1`, which gives us `last = [0, 0, 0, 0, 0, 0]`.
2. **Populate the `last` array**: We process the provided `ranges`:
    - Tap 0 has range 1: it can water the garden from `max(0, 0-1)` to `0+1`, hence we update `last[0] = 1`.
    - Tap 1 has range 0: it can only water its own position, so no update is necessary (`last[1]` remains `0`).
    - Tap 2 has range 2: it can water from `max(0, 2-2)` to `2+2`, so we update `last[0] = 4` (since this tap offers a better range than tap 0).
    - Tap 3 has range 1: waters from `max(0, 3-1)` to `3+1`, we update `last[2] = 4`.
    - Tap 4 has range 2: waters from `max(0, 4-2)` to `4+2`, we update `last[2] = 6`.
    - Tap 5 has range 0: waters only its position, no update. After updating, `last = [4, 0, 6, 4, 6, 0]`.
3. **Iterating through the garden**: We initialize the counters `ans = 0`, `mx = 0`, and `pre = 0`.
4. **Select taps and track coverage**: We iterate from `i = 0` to `i = 5` (position `n` in the garden):
    - At `i = 0`: `mx = max(0, 4) = 4`. No gap, continue.
    - At `i = 1`: `mx = 4`. Still no gap, continue.
    - At `i = 2`: `mx = max(4, 6) = 6`. As `i = pre`, we select this tap and update `ans = 1`, `pre = mx = 6`.
    - At `i = 3` and `5`: we don't need to update `mx` or `ans` since we've already covered the garden with `pre = 6`.
5. **Returning the answer**: Since we've iterated through the whole garden without encountering a gap, the final answer is `ans = 1`. Only one tap (at position 2) is needed to water the entire garden from 0 to 5.

## Python Solution

```python
class Solution:
    def min_taps(self, garden_length: int, ranges: List[int]) -> int:
        # Initialize an array to track the furthest right position that can be covered by opening a tap from each point
        max_right_from_left = [0] * (garden_length + 1)

        # Iterate over the taps and calculate the range each tap can cover. Then update the `max_right_from_left` array.
        for tap_index, tap_range in enumerate(ranges):
            left_bound = max(0, tap_index - tap_range)
            right_bound = tap_index + tap_range
            max_right_from_left[left_bound] = max(max_right_from_left[left_bound], right_bound)

        # Initialize variables for tracking the answer, the maximum distance covered so far, and the previous maximum before the last taps_required = 0
        max_covered_so_far = 0
        previous_max = 0

        # Iterate through the garden and update the max coverage.
        for pos in range(garden_length):
            max_covered_so_far = max(max_covered_so_far, max_right_from_left[pos])

            # If at any position, the max covered distance is less than or equal to the current position, the garden cannot be fully
            if max_covered_so_far <= pos:
                return -1

            # If the current position reaches the previous maximum coverage, it's time to open a new tap
            if previous_max == pos:
                taps_required += 1
                previous_max = max_covered_so_far

        # Return the minimum number of taps required to water the entire garden.
        return taps_required
```

## Java Solution

```java
class Solution {
    public int minTaps(int n, int[] ranges) {
        // Create an array to hold the farthest extent of water from each position.
        int[] farthestReach = new int[n + 1];

        // Populate the farthest extent each tap can reach for each position.
        for (int i = 0; i <= n; i++) {
            int left = Math.max(0, i - ranges[i]); // Ensure the left index is within bounds
            int right = i + ranges[i]; // Calculate the rightmost position current tap can cover
            // Update the farthestReach from the left position to what the current tap can reach
            farthestReach[left] = Math.max(farthestReach[left], right);
        }

        // Initialize variables to track the tapping.
        int tapsRequired = 0;    // To count the minimum number of taps
        int currentFarthest = 0;  // To keep track of the farthest we can reach at this point
        int lastTapPosition = 0;  // To remember the last position where we placed the tap

        // Iterate through the area to be watered excluding the last position.
        for (int i = 0; i < n; i++) {
            // Find the maximum distance that can be covered from the current position or before.
            currentFarthest = Math.max(currentFarthest, farthestReach[i]);

            // If the maximum distance we can reach at this point is less than or equals to current position,
            // it means we can't move forward from here as there is a gap. Hence, return -1.
            if (currentFarthest <= i) {
                return -1;
            }

            // If the last tap position is the same as the current position,
            // it means we need to place a new tap here and update the last tap position.
            if (lastTapPosition == i) {
                tapsRequired++;  // Increase the number of taps.
                lastTapPosition = currentFarthest; // Update the last tap position to the farthest reachable from here.
            }
        }

        // Return the minimum number of taps required.
        return tapsRequired;
    }
}
```

## C++ Solution

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    int minTaps(int gardenLength, vector<int>& ranges) {
        // Create a vector to store the furthest extent each tap can water, initializing to the garden's length.
        vector<int> maxWateredPosition(gardenLength + 1);

        // Determine the range each tap can water and update the maxWateredPosition array.
        for (int i = 0; i <= gardenLength; ++i) {
            int leftMost = max(0, i - ranges[i]); // Ensure we don't go below index 0.
            int rightMost = i + ranges[i]; // Calculate the rightmost watering position of current tap.
            // Update the furthest position that can be watered from leftMost.
            maxWateredPosition[leftMost] = max(maxWateredPosition[leftMost], rightMost);
        }

        // Initialize counters and flags for finding the minimum number of taps.
        int minTapsrequired = 0; // Counts the minimum number of taps required.
        int maxReachable = 0; // Keeps the maximum index that can be reached so far.
        int lastMaxReach = 0; // Remembers the last maximum reach when a new tap is turned on.

        // Iterate over the garden, keeping track of the reachable range.
        for (int i = 0; i <= gardenLength; ++i) {
            // Update maxReachable with the furthest position watered by taps up to position i.
            maxReachable = max(maxReachable, maxWateredPosition[i]);

            // If the maxReachable position is less or equal to the current position, we can't water the whole garden.
            if (maxReachable <= i) {
                return -1; // The whole garden cannot be watered, so return -1.
            }

            // When we reach the lastMaxReach, it means we have to turn on a new tap.
            if (lastMaxReach == i) {
                ++minTapsRequired; // Increase the number of taps used.
                lastMaxReach = maxReachable; // Update the last maximum reach.
            }
        }

        // Return the calculated minimum number of taps required to water the entire garden.
        return minTapsRequired;
    }
};
```

## Typescript Solution

```typescript
function minTaps(n: number, ranges: number[]): number {
    // Initialize an array to hold the furthest right each tap can reach starting from every point
    const furthestRight = new Array(n + 1).fill(0);

    // Iterate through the taps and calculate the furthest right reach for each point
    for (let i = 0; i <= n + 1; ++i) {
        const leftBoundary = Math.max(0, i - ranges[i]); // The leftmost point the tap can cover
        const rightBoundary = i + ranges[i]; // The rightmost point the tap can cover
        // Update the furthest right point that can be reached from the left boundary
        furthestRight[leftBoundary] = Math.max(furthestRight[leftBoundary], rightBoundary);
    }

    // Initialize counters for the minimum number of taps needed
    let currentMax = 0; // The current maximum right boundary that can be reached
    let previousMax = 0; // The previous maximum right boundary upon the last tap opening

    // Iterate over the array to find the minimum number of taps needed to cover the range [0, n]
    for (let i = 0; i < n; ++i) {
        // Update current maximum reach
        currentMax = Math.max(currentMax, furthestRight[i]);
        // If at any point the maximum reach is less than or equal to the current position,
        // it means the garden cannot be fully covered, so return -1
        if (currentMax <= i) {
            return -1;
        }
        // If the current position reaches the previous max, a new tap must be opened
        if (previousMax == i) {
            tapsNeeded++;
            previousMax = currentMax; // Update the previous max to the current max
        }
    }

    // Return the minimum number of taps needed to water the entire garden
    return tapsNeeded;
}
```

## Time and Space Complexity

### Time Complexity

The given Python code involves a single pass to transform the `ranges` list into the `last` list, followed by another single pass to find the minimum number of taps needed. Therefore, this algorithm runs in two linear passes over an array of length `n + 1`.

- The first pass is constructing the `last` array, which involves iterating over the `ranges` list once, resulting in `O(n)` complexity, where `n` is the length of the `ranges` list.
- The second pass is the loop that determines the minimum number of taps by iterating once through the `last` array, which is also `O(n)` complexity.

Combining both passes, the overall time complexity remains `O(n)` because they are sequential, not nested.

### Space Complexity

The space complexity is determined by the additional space used by the algorithm besides the input. Here, the `last` array of size `n + 1` is an auxiliary space used to store the farthest point that can be reached from each index. Therefore, the space complexity for the `last` array is `O(n)`.

No other data structures that are dependent on the size of the input are used, so the total space complexity is `O(n)`.