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

## **Problem Description**

Greedy Array Dynamic Programming

line, there are n + 1 taps at positions 0 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.

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

The problem is essentially about finding the minimum number of intervals (tap ranges) needed to cover a continuous segment

Intuition

#### (the garden). This is similar to the classic interval covering problem.

Hard

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 furthest right end pre, and the number of taps ans.

• 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.

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

- 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
- walkthrough of the implementation:

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

**Populate the last array**: We loop through the ranges array with the index i and range x. The variables 1 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 1 does not go

current maximal right boundary (mx), and the previous maximal right boundary (pre). Now, we iterate through the garden

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

# below 0, hence we use $\max(0, i - x)$ . For each position 1, we save the farthest point r that can be reached, by setting

from 0 to n.

**Example Walkthrough** 

tap can water on both sides.

garden—return -1.

Increment ans and set pre to mx.

the longest distance from each point.

covered with the minimum number of taps.

Populate the last array: We process the provided ranges:

def min taps(self, garden length: int, ranges: List[int]) -> int:

# Iterate through the garden and update the max coverage.

max\_right\_from\_left = [0] \* (garden\_length + 1)

for tap index, tap range in enumerate(ranges):

 $\circ$  At i = 0: mx = max(0, 4) = 4. No gap, continue.

• At i = 1: mx = 4. Still no gap, continue.

last[l] to max(last[l], r). This ensures that for each starting position l, we have the tap that waters the farthest. Iterating through the garden: We initialize three variables ans, mx, and pre to track the number of taps used (ans), the

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.

**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

Returning the answer: Once the loop is complete, we have either returned -1 if the garden can't be watered completely, or

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.

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

we have the minimum number of taps required in lans, which we return as the final result.

○ 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).

**Select taps and track coverage:** We iterate from i = 0 to i = 5 (position n in the garden):

 $\circ$  At i = 2: mx = max(4, 6) = 6. As i = pre, we select this tap and update ans = 1, pre = mx = 6.

 $\circ$  At i = 3, 4 and 5: we don't need to update mx or ans since we've already covered the garden with pre = 6.

**Step-by-Step Solution:** Create the last array: We create an array last of length n + 1, which gives us last = [0, 0, 0, 0, 0, 0].

 $\circ$  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]. **Iterating through the garden:** We initialize the counters ans = 0, mx = 0, and pre = 0.

 $\circ$  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).

#### 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.

class Solution:

taps required = 0

previous\_max = 0

max covered so far = 0

for pos in range(garden length):

if previous max == pos:

taps required += 1

public int minTaps(int n, int[] ranges) {

for (int i = 0;  $i \le n$ ; i++) {

int tapsRequired = 0;

int[] farthestReach = new int[n + 1];

// Initialize variables to track the tapping.

if (currentFarthest <= i) {</pre>

if (lastTapPosition == i) {

// Return the minimum number of taps required.

tapsRequired++;

if (maxReachable <= i) {</pre>

if (lastMaxReach == i) {

function minTaps(n: number, ranges: number[]): number {

const furthestRight = new Array(n + 1).fill(0);

return minTapsRequired;

for (let i = 0; i < n + 1; ++i) {

if (previousMax == i) {

tapsNeeded++;

return tapsNeeded;

class Solution:

**TypeScript** 

return -1;

return tapsRequired;

#include <vector>

#include <algorithm>

previous\_max = max\_covered\_so\_far

**Python** 

Solution Implementation

# Initialize an array to track the furthest right position that can be covered by opening a tap from each point

# Iterate over the taps and calculate the range each tap can cover. Then update the `max\_right\_from\_left` array.

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)

max\_covered\_so\_far = max(max\_covered\_so\_far, max\_right\_from\_left[pos])

// Create an array to hold the farthest extent of water from each position.

// Populate the farthest extent each tap can reach for each position.

farthestReach[left] = Math.max(farthestReach[left], right);

currentFarthest = Math.max(currentFarthest, farthestReach[i]);

// If the last tap position is the same as the current position,

# 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:</pre> return -1

# Initialize variables for tracking the answer, the maximum distance covered so far, and the previous maximum before the last

# Return the minimum number of taps required to water the entire garden. return taps\_required Java class Solution {

# If the current position reaches the previous maximum coverage, it's time to open a new tap

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++) {

int left = Math.max(0, i - ranges[i]); // Ensure the left index is within bounds

// To count the minimum number of taps

// Find the maximum distance that can be covered from the current position or before.

// it means we can't move forward from here as there is a gap. Hence, return -1.

// it means we need to place a new tap here and update the last tap position.

// Increase the number of taps.

// If the maximum distance we can reach at this point is less than or equals to current position,

lastTapPosition = currentFarthest; // Update the last tap position to the farthest reachable from here.

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

```
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) {</pre>
            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 farthest 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; // Tracks the maximum position reachable 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.

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.

// Return the calculated minimum number of taps required to water the entire garden.

// Initialize an array to hold the furthest right each tap can reach starting from every point

// Iterate through the taps and calculate the furthest right reach for each point

previousMax = currentMax; // Update the previous max to the current max

// Return the minimum number of taps needed to water the entire garden

def min taps(self, garden length: int, ranges: List[int]) -> int:

# Iterate through the garden and update the max coverage.

max\_right\_from\_left = [0] \* (garden\_length + 1)

for tap index, tap range in enumerate(ranges):

right bound = tap index + tap range

left bound =  $max(0, tap index - tap_range)$ 

lastMaxReach = maxReachable; // Update the last maximum reach.

++minTapsRequired: // Increase the number of taps used.

```
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);
let tapsNeeded = 0: // Counter 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) {</pre>
        return -1;
   // If the current position reaches the previous max, a new tap must be opened
```

# Initialize an array to track the furthest right position that can be covered by opening a tap from each point

# Iterate over the taps and calculate the range each tap can cover. Then update the `max\_right\_from\_left` array.

# Initialize variables for tracking the answer, the maximum distance covered so far, and the previous maximum before the last

max\_right\_from\_left[left\_bound] = max(max\_right\_from\_left[left\_bound], right\_bound)

max\_covered\_so\_far = max(max\_covered\_so\_far, max\_right\_from\_left[pos])

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

```
# 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:</pre>
    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 = 0

previous\_max = 0

max covered so far = 0

for pos in range(garden length):

taps required += 1

previous\_max = max\_covered\_so\_far

### **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

return taps required

Time and Space Complexity

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 0(n) complexity.

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

**Space Complexity** 

Combining both passes, the overall time complexity remains 0(n) because they are sequential, not nested.

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).

n + 1 is an auxiliary space used to store the farthest point that can be reached from each index. Therefore, the space

The space complexity is determined by the additional space used by the algorithm besides the input. Here, the last array of size