475. Heaters

Medium | Array | Two Pointers | Binary Search | Sorting | Leetcode Link

Problem Description

In this problem, we're given two arrays: `houses` and `heaters`. Each element in the `houses` array represents the position of a house on a horizontal line, and each element in the `heaters` array represents the position of a heater on the same line. Our objective is to find the minimum radius required for all the heaters so that every house can be covered by at least one heater's warm radius. Keep in mind that all heaters are set to the same radius, and we must choose it so every house is within the radius of at least one heater.

The goal is to minimize this radius while still ensuring that every house is within the warm range of at least one heater.

Intuition

Intuitively, we can think of this as a form of the binary search problem. We want to find the smallest radius such that every house is warm, meaning it falls within the range of [heater position - radius, heater position + radius] for at least one heater. To do this, we start by sorting both the `houses` and `heaters` arrays to simplify the process of checking if a house is within the warm radius of a heater.

The key insight behind the solution is to use binary search to find the minimum required radius. The binary search space is from 0 to the maximum possible distance between a house and a heater, which we initially assume to be a large number (like 10^9). We define a function `check` that, given a radius value `r`, determines whether all houses can be warmed with the heaters set to the specified radius.

Our binary search works as follows:

1. We set `left` to 0 and `right` to a large number, defining our initial search space for the radius.
2. We enter a loop where we continually narrow down our search space:
   - We calculate the middle value `mid` between `left` and `right`.
   - We call the `check` function with this `mid` value to see if it's possible to cover all houses with this radius.
   - If `check` returns `True`, it means the radius `mid` is sufficient, and we might be able to find a smaller radius that also works. So we set `right` to `mid` to look for a potentially smaller valid radius.
   - If `check` returns `False`, it means the radius `mid` is not enough to cover all houses, so we need a larger radius. We set `left` to `mid + 1` to search for a larger valid radius.
3. When `left` is equal to `right`, we have found the smallest radius that can cover all the houses.

The `check` function works by iterating through each house and determining if it's within the warm radius of at least one heater.

Once we exit the binary search loop, `left` will hold the minimum radius needed so that all houses are within the range of at least one heater.

Solution Approach

The solution leverages the binary search algorithm to zero in on the smallest possible radius. Here are the details of the implementation steps, referencing the provided solution code:

1. **Sorting Input Arrays**: The problem starts with sorting both `houses` and `heaters` arrays. Sorting is crucial because it allows the `check` function to iterate over houses and heaters in order, efficiently determining if each house is covered by the current radius.

2. **Binary Search Setup**: To find the correct radius, the code sets up a binary search with `left` as `0` and `right` as `int(1e9)` – a large enough number to ensure it's greater than any potential radius needed.

3. **The `check` Function**: This helper function checks whether a given radius `r` is sufficient to warm all houses. It initializes two pointers: `i` for houses and `j` for heaters. For each house, it checks if it falls in the range of the heater at index `j`. If not, it moves to the next heater and repeats the check. If all houses can be covered with the given radius, it returns `True`; otherwise, `False`.

4. **Using Binary Search**:

   - A loop continues until `left` is not less than `right`, meaning the binary search space has been narrowed down to a single value.
   - We calculate a `mid` value as the average of `left` and `right`. This `mid` represents the potential minimum radius.
   - We call `check(mid)` to verify if all houses are warmed by heaters with radius `mid`.
     - If `True`, the `mid` radius is large enough, and we attempt to find a smaller minimum by setting `right` to `mid`.
     - If `False`, the `mid` radius is too small, so we look for a larger radius by setting `left` to `mid + 1`.
5. **Convergence and Result**: By continuously halving the search space, the binary search converges to the minimum radius required, which is then returned by the function when `left` equals `right`. This is because when `check(mid)` can no longer find a radius that covers all houses, `left` will now point to the smallest valid radius found.

The main data structure used is the sorted arrays, which enable efficient traversal. The binary search algorithm is the core pattern used to minimize the heater radius, and the `check` function is an implementation of a two-pointer technique to match heaters to houses within a given warm radius. Combining sorting, binary search, and two-pointer techniques makes the solution efficient and enables it to handle a wide range of input sizes.

Example Walkthrough

Let's say we have the following array of `houses`: `[1, 2, 3, 4]`, and `heaters`: `[1, 4]`. We want to use the solution approach to find the minimum heater radius.

1. **Sorting Input Arrays**:

   - The `houses` array is already sorted: `[1, 2, 3, 4]`.
   - The `heaters` array is already sorted: `[1, 4]`.
2. **Binary Search Setup**:

   - We set `left` to `0` and `right` to `10^9` (a very large number).
3. **Binary Search Start**

   - The initial value of `left` is `0`, and `right` is `10^9`.
4. **Binary Search Iteration**

   - **First Iteration**:
     - `mid = (left + right) / 2` which calculates to `(0 + 10^9) / 2 = 5 * 10^8`.
     - We use `check(5 * 10^8)` which will definitely return `True` because the radius is very large and will cover all houses. We then set `right` to `mid` which now becomes `5 * 10^8`.
   - **Second Iteration**:
     - Now we have `left = 0`, `right = 5 * 10^8`.
     - `mid = (left + right) / 2 = 2.5 * 10^8`.
     - Again, `check(2.5 * 10^8)` will return `True`. We update `right` to `2.5 * 10^8`.
   - We keep iterating, and the value of `right` will continue to decrease until `check` returns `False`, which indicates the `mid` value is insufficient for a heater radius.
5. **Finding the Minimum Radius**

   - We continue the binary search to narrow down the minimum radius that can cover all houses. Assuming our houses and heaters, a `mid` of `2` would be enough to cover all the houses (house at position `2` is at a distance of `2` from heater `1`, and house at `3` is also at a distance of `1` from heater `4`).
   - So, when the check function is called with `mid` values lower than `2`, it will start to return `False`, prompting the binary search to stop decreasing the `right` boundary.
6. **Arriving at the Solution**

   - The binary search algorithm will continue to narrow down until `left` and `right` converge, giving us the minimum radius needed.
   - For our example, after enough iterations, `left` will equal `right` at the value of `2`. This means the minimum radius required for the heaters is `2`.

In this way, we use the binary search algorithm to efficiently find the minimum radius for the heaters to cover all houses.

Python Solution

```python
from typing import List

class Solution:
    def findRadius(self, houses: List[int], heaters: List[int]) -> int:
        # First, sort the houses and heaters to enable efficient scanning
        houses.sort()
        heaters.sort()

        # The function 'can_cover' checks if a given radius 'radius'
        # is enough to cover all houses by heaters. Returns True if it is, False otherwise.
        def can_cover(radius):
            # Get the number of houses and heaters
            num_houses, num_heaters = len(houses), len(heaters)
            house_idx, heater_idx = 0, 0   # Start scanning from the first house and heater

            # Iterate over all houses to check coverage
            while house_idx < num_houses:
                # If no more heaters are available to check, return False
                if heater_idx >= num_heaters:
                    return False

                # Calculate the minimum and maximum coverage range of the current heater
                min_range = heaters[heater_idx] - radius
                max_range = heaters[heater_idx] + radius

                # If the current house is not covered, attempt to use the next heater
                if houses[house_idx] < min_range:
                    return False
                # If the current house is outside the current heater's max range,
                # move to the next heater
                if houses[house_idx] > max_range:
                    heater_idx += 1
                else: # Otherwise the house is covered, move to the next house
                    house_idx += 1

            return True  # All houses are covered

        # Start with a radius range of 0 to a large number (1e9 is given as a maximum)
        left, right = 0, int(1e9)

        # Perform a binary search to find the minimum radius
        while left < right:
            mid = (left + right) // 2  # Choose the middle value as the potential radius
            # If all houses can be covered with the 'mid' radius, search the lower half
            if can_cover(mid):
                right = mid
            else: # If not, search the upper half
                left = mid + 1

        # The minimum radius required to cover all houses is 'left' after the loop
        return left

```

Java Solution

```java
class Solution {
    public int findRadius(int[] houses, int[] heaters) {
        // Sort the array of heaters to perform efficient searches later on
        Arrays.sort(heaters);

        // Initialize the minimum radius required for heaters to cover all houses
        int minRadius = 0;

        // Iterate through each house to find the minimum radius needed
        for (int house : houses) {
            // Perform a binary search to find the insertion point or the actual position of the house
            int index = Arrays.binarySearch(heaters, house);

            // If the house is not a heater, calculate the potential insert position
            if (index < 0) {
                index = ~index;  // index = -(index + 1)
            }

            // Calculate distance to the previous heater, if any, else set to max value
            int distanceToPreviousHeater = index > 0 ? house - heaters[index - 1] : Integer.MAX_VALUE;

            // Calculate distance to the next heater, if any, else set to max value
            int distanceToNextHeater = index < heaters.length ? heaters[index] - house : Integer.MAX_VALUE;

            // Calculate the minimum distance to the closest heater for this house
            int minDistanceToHeater = Math.min(distanceToPreviousHeater, distanceToNextHeater);

            // Update the minimum radius to be the maximum of previous radii or the minimum distance for this house
            minRadius = Math.max(minRadius, minDistanceToHeater);
        }

        // Return the minimum radius required
        return minRadius;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to find the minimum radius of heaters required to cover all houses.
    int findRadius(std::vector<int>& houses, std::vector<int>& heaters) {
        // Sort the house and heater positions to perform binary search later.
        std::sort(houses.begin(), houses.end());
        std::sort(heaters.begin(), heaters.end());

        // Setting initial binary search range.
        int left = 0, right = 1e9;

        // Binary search to find the minimum radius.
        while (left < right) {
            int mid = (left + right) / 2;
            // Check if with this radius all houses can be covered.
            if (canCoverAllHouses(houses, heaters, mid))
                right = mid;  // Radius can be smaller or is optimal, reduce the search to left half.
            else
                left = mid + 1;  // Radius too small, need to increase, search in right half.
        }
        // Once left meets right, we've found the smallest radius needed.
        return left;
    }

    // Function to check if all houses can be covered with a given radius 'radius'.
    bool canCoverAllHouses(const std::vector<int>& houses, const std::vector<int>& heaters, int radius) {
        int numHouses = houses.size(), numHeaters = heaters.size();
        int i = 0, j = 0;  // Initialize pointers for houses and heaters

        // Iterate over houses to check coverage.
        while (i < numHouses) {
            if (j >= numHeaters) return false;  // Run out of heaters, can't cover all houses.
            int minHeaterRange = heaters[j] - radius;
            int maxHeaterRange = heaters[j] + radius;

            // Current house is not covered by heater j's range.
            if (houses[i] < minHeaterRange)
                return false;  // House i can't be covered by any heater, so return false.

            // Current house is outside the range of heater j, move to next heater.
            if (houses[i] > maxHeaterRange)
                ++j;
            else
                ++i;  // Current house is covered, move to the next house.
        }
        // All houses are covered.
        return true;
    }
};

```

Typescript Solution

```typescript
function findRadius(houses: number[], heaters: number[]): number {
    // Sort arrays to enable binary search later.
    houses.sort((a, b) => a - b);
    heaters.sort((a, b) => a - b);

    // Initialize variables for the lengths of the houses and heaters arrays
    const totalHouses = houses.length;
    const totalHeaters = heaters.length;

    // Initialize answer variable to store the final result, the minimum radius needed
    let minRadius = 0;

    // Initialize houseIndex and heaterIndex for traversing through the houses and heaters.
    for (let houseIndex = 0, heaterIndex = 0; houseIndex < totalHouses; houseIndex++) {
        // Calculate current radius for this house to the current heater
        let currentRadius = Math.abs(houses[houseIndex] - heaters[heaterIndex]);

        // Continue moving to the next heater to find the closest heater for this house
        while (heaterIndex + 1 < totalHeaters &&
            Math.abs(houses[houseIndex] - heaters[heaterIndex]) >= Math.abs(houses[houseIndex] - heaters[heaterIndex + 1])) {

            // Keep track of the minimum radius required for the current house
            currentRadius = Math.min(Math.abs(houses[houseIndex] - heaters[++heaterIndex]), currentRadius);
        }

        // Update the minimum radius required for all houses
        minRadius = Math.max(currentRadius, minRadius);
    }

    // Return the minimum radius required for all houses
    return minRadius;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the solution is determined by the following factors:

- Sorting the `houses` and `heaters` arrays, which takes `O(mlogm + nlogn)` time, where `m` is the number of houses and `n` is the number of heaters.
- The binary search to find the minimum radius, which takes `O(log(max(houses) - min(houses)))` iterations. Each iteration performs a check which has a linear pass over the `houses` and `heaters` arrays, taking `O(m + n)` time in the worst case.
- Combining these, the overall time complexity is `O(mlogm + nlogn + (m + n)log(max(houses) - min(houses)))`.

Space Complexity

The space complexity of the solution is determined by the following factors:

- The space used to sort the `houses` and `heaters` arrays, which is `O(m + n)` if we consider the space used by the sorting algorithm.
- The space for the variables and pointers used within the `findRadius` method and the `check` function, which is `O(1)` since it's a constant amount of space that doesn't depend on the input size.

Combining the sorting space and the constant space, the overall space complexity is `O(m + n)`.