774. Minimize Max Distance to Gas Station

Hard | Array | Binary Search | Leetcode Link

Problem Description

In this problem, we are dealing with an optimization problem involving gas stations positioned linearly along an x-axis. The positions of these existing gas stations are provided as an integer array `stations`. We are tasked with adding `k` new gas stations to this line, and these new stations can be placed at any point along the x-axis, which doesn't need to be an integer value.

The goal is to minimize the "penalty," which is defined as the maximum distance between any two adjacent gas stations after all `k` new stations have been added. The final output should be the smallest possible value of this maximum distance, and an answer that is within `10^-6` of the actual answer is considered valid.

Intuition

To solve this problem, we can use a binary search approach to find the smallest possible maximum distance that satisfies the conditions. The main intuition here is that if we have a certain maximum distance `D`, we can determine whether it's possible to add `k` or fewer gas stations such that no two stations are more than `D` miles apart.

The binary search is performed over the range of possible distances which could be the penalty. We start by setting the left end (minimum possible penalty) to `0` and the right end (maximum possible penalty) to `1e8`, a sufficiently large number that is guaranteed to be larger than any possible maximum distance.

The `check` function is used within the binary search to determine if a given maximum distance `x` is feasible. If it is, that means we can place `k` new gas stations in such a way that the maximum distance between any two adjacent stations is not greater than `x`. To do this, for any two adjacent existing stations, we calculate how many new stations would need to be added between them to ensure that the distance between adjacent stations is less than or equal to `x`. If the sum of required stations for all gaps does not exceed `k`, the function returns `True`.

Using this check function, we continue to narrow down the range in the binary search until the difference between the left and right ends is less than `10^-6`, which means we have found the minimum possible penalty to the desired precision. We then return this value as the smallest possible value of `penalty()`.

Solution Approach

The solution employs a binary search algorithm to efficiently find the smallest possible "penalty", which is the maximum distance between adjacent gas stations after adding `k` new stations. The binary search operates over a range of possible values for the penalty, rather than searching through a list of items.

Here's an explanation of how the algorithm works, referring to the solution code:

Binary Search Range

- **Initial Range**: We define an initial range for the binary search with the left end set to `0` and the right end set to `1e8`. This wide range ensures that the actual minimum penalty is included.

`check` Function

- **Purpose**: It determines whether it's possible to add stations such that the maximum distance between any two stations doesn't exceed `x`.
- **Implementation**: It iterates through each pair of adjacent stations (using `pairwise` which groups the array elements in pairs, provided by the Python standard library), and computes how many additional stations need to be added in between to ensure that the distance between stations is at most `x`. The computation `(b - a) / x` gives the number of stations required between stations located at `a` and `b`. Summing these up for all pairs and comparing it to `k` tells us if `x` is a feasible penalty.

Iterative Binary Search

- **Loop**: The while loop runs until the left and right ends of the search range are within `10^-6` of each other, meaning we have reached the required precision.
- **Midpoint Calculation**: In each iteration, it calculates the midpoint `mid` of the current search range by averaging `left` and `right`.
- **Feasibility Check**: It calls the `check` function with this midpoint value.
  - If `check(mid)` returns `True`, it means that we can achieve the penalty `mid` by adding `k` or fewer stations, so we update the right end to `mid`.
  - If `check(mid)` returns `False`, we need a larger penalty to fit `k` stations, so we update the left end to `mid`.

Convergence and Result

- **Precision**: The search loop continues until the precision condition is met. This ensures our answer is within `10^-6` of the actual minimum penalty.
- **Return Value**: The loop exits when the left and right ends are sufficiently close, and `left` is returned as it represents the smallest tested penalty that can be achieved with `k` gas stations.

This approach efficiently zooms in on the correct answer using logarithmic time complexity, which is typical of binary search algorithms. This is much more optimal than a linear search or brute force approach, which could be time-prohibitive given the potentially large search space.

Example Walkthrough

Consider a scenario where there are three gas stations along a road at positions `stations = [1, 5, 10]` and we want to add `k = 2` new gas stations. We will illustrate how the solution approach using binary search helps us find the minimum possible penalty.

Initial Setup

The current maximum distance between adjacent stations is `5`, which is between the first and second stations (positions `1` and `5`). We aim to minimize this maximum distance by adding `2` new stations.

Binary Search Range Setup

We'll set up our binary search with an initial range for the penalty, from `left = 0` to `right = 1e8`.

First Iteration

- We calculate the midpoint `mid` of `left` and `right`, which is `mid = (0 + 1e8) / 2 = 5e7`. Obviously, this is an overestimate.
- We run the `check` function with `mid`. For each pair of stations, we calculate how many new stations would need to be placed to ensure no adjacent stations are more than `5e7` miles apart. Clearly, zero stations are needed since the largest gap is `5` which is much less than `5e7`.
- Since `check(mid)` would return true (no stations needed is less than or equal to `k = 2`), we update `right` to `mid`.

Subsequent Iterations

- Next `mid` is `mid = (0 + 5e7) / 2 = 2.5e7`, and the checks and updates continue similarly. The midpoint is still much larger than the maximum gap, and the `right` will continue to decrease dramatically.
- As we keep iterating, the `mid` value will come down significantly, closer to the actual gap size.

Narrowing Down

Eventually, after several iterations, `mid` will be close to the actual maximum distance we need. Suppose `mid` comes down to `2.5`. At this distance, we check how many new stations are needed:

- Between stations at `1` and `5` (`gap = 4`), we need at most `1` new station to ensure the gap is no larger than `2.5`.
- Between stations at `5` and `10` (`gap = 5`), we need `2` new stations.

Since we only need `2` stations and we are allowed to add `2`, this `mid` value is feasible. If `mid` were smaller, we might need more than `2` stations, and the `check` would return false, prompting us to adjust the `left` end of the range.

Final Iteration and Result

As we continue the binary search, updating `left` and `right`, the interval of `left` and `right` narrows down to within `10^-6` of each other. Suppose through the binary search we reach `left = 2.499999` and `right = 2.500001`, we then stop as this is within our acceptable precision.

At this point, if `check(2.499999)` returns true, that means we can place `2` new stations such that no segment is longer than `2.499999` miles, and thus, this is our final answer. Since our target is a precision within `10^-6`, the minimum penalty, or the smallest maximum distance between adjacent gas stations after adding the new ones, is approximately `2.5`.

This illustrates the effectiveness of the binary search technique for our optimization problem, where instead of exhaustively trying every possible location for new stations, the binary search algorithm quickly zeroes in on the smallest possible penalty, the maximum distance between adjacent stations after adding the new ones.

Python Solution

```python
from typing import List
from itertools import pairwise

class Solution:
    def minmaxGasDist(self, stations: List[int], k: int) -> float:
        # Helper function to calculate if it's possible to have maximum
        # distance no more than 'x' between two adjacent gas stations after
        # adding 'k' new gas stations
        def is_possible(x):
            # Calculate number of stations needed to ensure no segment
            # between the existing stations is longer than 'x'
            additional_stations_needed = sum(int((b - a) / x) for a, b in pairwise(stations))
            # Return True if the calculated number of stations needed is
            # less than or equal to 'k', False otherwise
            return additional_stations_needed <= k

        # Initialize the binary search bounds
        left, right = 0, 1e8  # Start with a wide range since distances could be large

        # Perform binary search with precision up to 1e-6
        while right - left > 1e-6:
            mid = (left + right) / 2
            # If it's possible to achieve mid as value can satisfy the condition
            if is_possible(mid):
                right = mid  # If it is possible, the answer is less than or equal to 'mid'
            else:
                left = mid  # If not possible, the answer is greater than 'mid'

        # Once the loop ends, 'left' will be our answer to the smallest possible maximum distance
        # that fulfills the requirements, with the required precision
        return left
```

Java Solution

```java
class Solution {
    // Finds the minimum possible distance between gas stations after adding k extra stations.
    public double minmaxGasDist(int[] stations, int k) {
        // Define the range for the possible solution (left is minimum distance, right is maximum distance)
        double left = 0, right = 1e8;

        // Continue searching while the precision is greater than 1e-6
        while (right - left > 1e-6) {
            // Take the middle of the current range as a guess
            double mid = (left + right) / 2.0;

            // If it's possible to place gas stations such that the maximum distance is less than or equal to 'mid',
            // then we update right to the current 'mid' to see if we can find an even smaller maximum distance
            if (isPossible(mid, stations, k)) {
                right = mid;
            } else {
                // If it's not possible, we update left to be 'mid' to look for solutions greater than 'mid'
                left = mid;
            }
        }

        // Since the left and right converge to the point where right - left <= 1e-6, left is the most accurate answer
        return left;
    }

    // Helper method to check if it's possible to have a maximum gas station distance less than x after adding K gas stations.
    private boolean isPossible(double x, int[] stations, int k) {
        int count = 0; // The number of gas stations we need to add to satisfy the condition

        // Go through all pairs of adjacent stations
        for (int i = 0; i < stations.length - 1; ++i) {
            // Find the number of additional stations needed for this segment so that the distance between stations is <= x
            count += (int) (stations[i + 1] - stations[i]) / x;
        }

        // Check if the count of additional stations required is less than or equal to K (the count we can add)
        return count <= k;
    }
}
```

C++ Solution

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

class Solution {
public:
    double minmaxGasDist(vector<int>& stations, int k) {
        double minDist = 0, maxDist = 1e8; // Initializing the range for possible answers

        // Lambda function to check if a given maximum distance 'maxDistBetweenStations' can be achieved
        // by adding at most 'k' gas stations
        auto isPossible = [&](double maxDistBetweenStations) {
            int requiredStations = 0;

            // Count the number of additional stations required for each interval between stations
            for (int i = 0; i < stations.size() - 1; ++i) {
                requiredStations += (int)((stations[i + 1] - stations[i]) / maxDistBetweenStations);
            }

            // Return true if the number of additional stations to maintain the maximum distance
            // is less than or equal to 'k'
            return requiredStations <= k;
        };

        // Binary search to find the smallest possible maximum distance
        while (maxDist - minDist > 1e-6) { // 1e-6 is used as the precision for the answer
            double midDist = (minDist + maxDist) / 2.0;

            // If it is possible to achieve this maximum distance, we try to find a smaller one
            if (isPossible(midDist)) {
                maxDist = midDist;
            } else {
                // If it is not possible, we need to increase the maximum distance
                minDist = midDist;
            }
        }

        // After the loop, 'minDist' will be our answer to the problem, rounded to the nearest 1e-6
        return minDist;
    }
};
```

Typescript Solution

```typescript
function minmaxGasDist(stations: number[], k: number): number {
    let minDist: number = 0, maxDist: number = 1e8; // Initialize the range for possible answers

    // Function to check if a given maximum distance 'maxDistBetweenStations' can be achieved
    // by adding at most 'k' gas stations
    const isPossible = (maxDistBetweenStations: number): boolean => {
        let requiredStations: number = 0;

        // Count the number of additional stations required for each interval between stations
        for (let i = 0; i < stations.length - 1; ++i) {
            requiredStations += Math.int((stations[i + 1] - stations[i]) / maxDistBetweenStations) - 1;
        }

        // Return true if the number of additional stations to maintain the max distance is less than or equal to 'k'
        return requiredStations <= k;
    };

    // Binary search to find the smallest possible maximum distance
    while (maxDist - minDist > 1e-6) { // 1e-6 is the precision for the answer
        let midDist: number = (minDist + maxDist) / 2.0;

        // If it is possible to achieve this max distance, we try to find a smaller one
        if (isPossible(midDist)) {
            maxDist = midDist;
        } else {
            // If it is not possible, we increase the maximum distance
            minDist = midDist;
        }
    }

    // After the loop, 'minDist' will be our answer to the problem, rounded to the nearest 1e-6
    return minDist;
};
```

Time and Space Complexity

The time complexity of the provided code is primarily determined by the while loop that performs a binary search over a range of possible answers from `left` to `right` to find the minimum possible maximum gas station distance with a precision of `1e-6`. During each iteration of the binary search, a `check` function is called which takes linear time relative to the number of stations `O(N)`, as it iteratively calculates the number of additional gas stations needed from each pair of consecutive stations. The binary search will run for `O(log(right-left)/precision)` iterations, `right-left` being the initial search range and `precision` being `1e-6`. Therefore, the total time complexity of the algorithm is `O(N * log(right-left)/precision)`. In this case, the initial search range `right-left` is fixed at `1e8`, so we can consider this a constant and the log term simplifies to `O(log1/precision)`, resulting in a final time complexity of `O(N * log1/precision)`.

The space complexity of the code is `O(1)` which means constant space complexity, as only a fixed number of variables are used and there are no data structures that grow with the input size. The space used by the `check` function and the binary search variables `left`, `right`, and `mid` does not scale with the number of `stations`.