2604. Minimum Time to Eat All Grains

Hard | Array | Binary Search | Sorting | Leetcode Link

Problem Description

In this problem, we are given two arrays representing the positions of `n` hens (`hens`) and `m` grains (`grains`) on a line respectively. The goal is to calculate the minimum time required for all hens to eat all the grains, assuming each hen can move one unit left or right in one second. Hens can move simultaneously and independently, and a hen can eat any grain if they share the same position on the line. Importantly, there's no limit to how many grains a single hen can eat, they just need to be at the same position. The question asks for the optimal strategy that minimizes the time for all the grains to be eaten.

Intuition

The intuition behind the solution involves recognizing that since hens can move independently and multiple hens can eat from the same pile of grains, we want to minimize the distance each hen needs to move. To find the minimum time, we can use a binary search strategy, where the search space is the time it takes for the hens to eat all the grains. We are essentially asking "Can all grains be eaten within `t` seconds?" for various values of `t`.

The solution approach uses binary search to minimize the time by trying to find the smallest `t` such that all conditions are satisfied - which involves checking whether the hens can eat all grains within that time frame.

To facilitate the binary search check, we sort both the hens and grains because only the relative positions matter, not the specific values. After sorting, we iterate through each hen and determine if it can eat the available grains within `t` seconds. If a hen at position `x` encounters a grain at position `y`, and `y` is to the left of `x` (`y <= x`), then the hen can eat the grain if it can reach it within `t` seconds (`x - y <= t`). If the grain is to the right of the hen (`y > x`), then we check if the hen can reach it within `t` seconds (`y - x <= t`). If during this check for all hens, we find that all grains have been accounted for within `t` seconds, we consider it successful.

It's critical that this checking function is efficient because it is called many times during binary search. By moving our "grain pointer" `j` only to the right, we ensure we don't do redundant checks, thus optimizing the process.

Solution Approach

In the implementation, our main goal is to determine the minimum time `t`, using binary search, during which all hens can eat all grains. To start, we need a `check(t)` function that takes `t` as an input and returns `True` if all grains can be eaten by hens within `t` seconds, else it returns `False`.

Here's a step-by-step breakdown of the `check(t)` function:

1. Sort both `hens` and `grains` in ascending order to efficiently pair hens with the nearest grains.

2. Use a variable `j` to keep track of the next grain to be checked.

3. Loop through each hen positioned at `x`, and for each, attempt to eat the grain located at `grains[j]`, which is `y`.

4. For each hen, check the position of the grain relative to the hen's position. There are two cases:

   - If the grain is to the left of or at the hen's position (`y <= x`):
     - Calculate the distance `d = x - y`. If `d > t`, `False` is returned, because the hen can't reach the grain within `t` seconds.
     - Iterate over `grains` starting from `j` to the right, moving `j` forward as long as the distance for the hen to return from the grain to its original position and then to the next grain is less than or equal to `t`.
   - If the grain is to the right of the hen (`y > x`):
     - Increment `j` to the right as long as the distance from the hen to the next grain is less than or equal to `t`.
5. If by the end of this process `j == m`, which means all grains have been visited, return `True`. Otherwise, return `False`.

Now, this check over all possible hens and grains determines if it's feasible for all hens to eat all grains in `t` seconds. By running this check function within a binary search over the range being considered for `t`, which is [0, r], where `r` is an initially large range to cover all possible time scenarios, we can efficiently home in on the smallest `t` that returns `True`.

In Python's `bisect` module, `bisect_left` is used, which returns the index of the first item in the sorted array that is greater than or equal to the specified value. Here, that specified value is `True`, meaning we're searching for the smallest `t` where `check` returns `True`.

The binary search continues to adjust this range until the lower bound meets the criteria, which is the minimum time `t` that guarantees all grains are eaten.

This solution is efficient and leverages the capabilities of sorting and binary search to reduce an otherwise complicated simulation problem down to a manageable and computationally viable process.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have `n = 2` hens and `m = 3` grains, with the hens positioned at `hens = [1, 4]` and the grains at `grains = [2, 3, 5]` on the line.

Firstly, we sort both arrays for efficient processing:

- Hens: `hens = [1, 4]` (already sorted)
- Grains: `grains = [2, 3, 5]` (already sorted)

Now we will define a range for our binary search. The maximum possible time `t` would be if the furthest grain is directly opposite the furthest hen, which in this case would be the hen at position `4` and the grain at position `5`, thus maximum time `t_max = 1`. We begin with `t` ranging from 0 to `t_max`.

We then start our binary search:

1. Middle of our range (`t = 0`) is checked first to see if it is possible for hens to eat all grains in `0` seconds, which is clearly not possible because the hens and grains are not at the same positions. Thus, `check(0)` returns `False`.

2. Since `t = 0` is not enough time, we need to increase `t`. The next value we check is `t = 1`, which is the midpoint of the updated range [1, 1]. We run `check(1)`:

   - The first hen at position `1` can reach and eat the grain at position `2` within `1` second.
   - The second hen at position `4` can reach and eat the grain at position `5` within `1` second. It can also backtrack to eat the grain at position `3` after eating the grain at position `5`.

   After simulating the process, we find that all the grains can be eaten within `1` second. Thus, `check(1)` returns `True`.

Since `check(1)` returns `True`, we have found our solution: all hens can eat all grains in a minimum time of `1` second. There's no need to continue the binary search because we can't find a smaller non-negative value of `t` that satisfies the conditions.

To summarize with our example, we used binary search to determine that `1` second is the minimum time required for the hens at positions `1` and `4` to eat all grains at positions `2`, `3`, and `5`. This demonstrates the effectiveness of sorting and binary search in finding the optimal solution for the problem.

Python Solution

```python
from bisect import bisect_left
from typing import List

class Solution:
    def minimumTime(self, hens: List[int], grains: List[int]) -> int:
        # Helper function to check if all grains can be picked within time 't'
        # It simulates the picking process for a given time 't' and returns True if possible, False otherwise
        def can_pick_all(t: int) -> bool:
            grain_index = 0  # Start with the first grain
            m = len(grains)
            for hen in hens:
                if grain_index == m:
                    return True  # All grains have been picked
                next_grain = grains[grain_index]
                if next_grain <= hen:
                    diff = hen - next_grain
                    if diff > t:
                        return False  # hen is too far from the grain to pick it within time 't'
                    while grain_index < m and grains[grain_index] <= hen:
                        grain_index += 1
                    while grain_index < m and min(diff, grains[grain_index] - hen) + grains[grain_index] - next_grain <= t:
                        grain_index += 1
                else:
                    # The hen will pick the grains until it is closer to them than time 't'
                    while grain_index < m and grains[grain_index] - hen <= t:
                        grain_index += 1
            return grain_index == m  # Check if all grains are picked

        # Sort both hens and grains lists to make it easier to process them in order
        hens.sort()
        grains.sort()

        # Compute the initial range for 't', which can be up to the furthest distance a hen might need to travel
        range_max = abs(hens[0] - grains[0]) + grains[-1] - grains[0] + 1

        # Find the minimum time within which all grains can be picked using binary search
        minimum_time = bisect_left(range(range_max), True, key=can_pick_all)

        return minimum_time

```

Java Solution

```java
import java.util.Arrays;

class Solution {
    private int[] hensPositions;
    private int[] grainPositions;
    private int totalGrains;

    public int minimumTime(int[] hens, int[] grains) {
        totalGrains = grains.length;
        hensPositions = hens;
        grainPositions = grains;
        Arrays.sort(hensPositions);
        Arrays.sort(grainPositions);

        // Set the initial range of time to search for the minimum time needed.
        // It's between 0 and the maximum possible distance a hen needs to travel minus the first gain position.
        int left = 0;
        int right = Math.abs(hensPositions[0] - grainPositions[0]) + grainPositions[totalGrains - 1] - grainPositions[0];

        // Use binary search to find the minimum time needed.
        while (left < right) {
            int mid = (left + right) / 2;
            if (isTimeSufficient(mid)) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        // 'left' will hold the minimum time needed when the while-loop finishes.
        return left;
    }

    private boolean isTimeSufficient(int allowedTime) {
        int grainIndex = 0;

        // Check if the time allowed is sufficient for each hen.
        for (int henPosition : hensPositions) {
            // If all grains have been checked, return true.
            if (grainIndex == totalGrains) {
                return true;
            }

            int grainPosition = grainPositions[grainIndex];
            if (grainPosition <= henPosition) {
                int distance = henPosition - grainPosition;
                if (distance > allowedTime) {
                    // If the current grain is too far for the time allowed, return false.
                    return false;
                }

                // Find the next grain that is farther than the hen but within the allowed time.
                while (grainIndex < totalGrains && grainPositions[grainIndex] <= henPosition) {
                    grainIndex++;
                }

                // Attempt to get as close as possible to the current hen position without exceeding the allowed time.
                while (grainIndex < totalGrains && Math.min(distance, grainPositions[grainIndex] - henPosition) + grainPositions[gr
                    grainIndex++;
                }
            } else {
                // Find the grain that is within the allowed time for the current hen.
                while (grainIndex < totalGrains && grainPositions[grainIndex] - henPosition <= allowedTime) {
                    grainIndex++;
                }
            }
        }

        // Return true if all grains have been assigned, false otherwise.
        return grainIndex == totalGrains;
    }
}

```

C++ Solution

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

class Solution {
public:
    // This method computes the minimum time needed for all hens to eat grains
    // given their positions and the positions of the grains.
    int minimumTime(vector<int>& hens, vector<int>& grains) {
        // Sort the hens and grains in non-decreasing order
        sort(hens.begin(), hens.end());
        sort(grains.begin(), grains.end());

        int numberOfGrains = grains.size();

        // Setup the binary search boundaries
        // The starting left boundary 'l' is set to 0 indicating the lowest possible time
        // The starting right boundary 'r' is based on the furthest possible distance a hen
        // might need to travel, which is from the first hen to the first grain location plus
        // from the first grain location to the last grain location
        int leftBoundary = 0;
        int rightBoundary = abs(hens[0] - grains[0]) + grains[numberOfGrains - 1] - grains[0];

        // Define the check loop as a lambda function to determine if a given time 't' is sufficient
        auto check = [&](int time) -> bool {
            int grainIndex = 0;
            for (int henPos : hens) {
                if (grainIndex == numberOfGrains) {
                    return true;
                }
                int grainPos = grains[grainIndex];
                if (grainPos <= henPos) {
                    int distance = henPos - grainPos;
                    if (distance > time) {
                        return false;
                    }
                    // Move the grain index to the next grain past the current hen position
                    while (grainIndex < numberOfGrains && grains[grainIndex] <= henPos) {
                        ++grainIndex;
                    }
                    // Keep moving the grain index as long as the hen can reach the grain within the 'time'
                    while (grainIndex < numberOfGrains && min(distance, grains[grainIndex] - henPos) +
                           grains[grainIndex] - grainPos <= time) {
                        ++grainIndex;
                    }
                } else {
                    // Move the grain index as long as the hen can reach the grain within the 'time'
                    while (grainIndex < numberOfGrains && grains[grainIndex] - henPos <= time) {
                        ++grainIndex;
                    }
                }
            }
            // If we've assigned a grain to each hen within the time, return true
            return grainIndex == numberOfGrains;
        };

        // Perform a binary search to find the minimum time required
        while (leftBoundary < rightBoundary) {
            int midTime = (leftBoundary + rightBoundary) / 2;
            if (check(midTime)) {
                // If midTime is enough for all hens to eat, try to find a smaller time
                rightBoundary = midTime;
            } else {
                // If midTime is not enough, ignore the left half
                leftBoundary = midTime + 1;
            }
        }
        // Return the smallest time in which all hens can eat
        return leftBoundary;
    }
};

```

Typescript Solution

```typescript
function minimumTime(hens: number[], grains: number[]): number {
    // Sort both hens and grains arrays in ascending order
    hens.sort((a, b) => a - b);
    grains.sort((a, b) => a - b);

    const grainCount = grains.length;
    // The range of possible minimum times starts at 0 and extends up to a logical upper bound
    let lowerBound = 0;
    let upperBound = Math.abs(hens[0] - grains[0]) + grains[grainCount - 1] - grains[0] + 1;

    // A helper function that checks if all hens can eat grains within the time 'timeLimit'
    const canFeedWithinTimeLimit = (timeLimit: number): boolean => {
        let grainIndex = 0;
        for (const henPosition of hens) {
            if (grainIndex === grainCount) {
                // If all grains have been used, return true
                return true;
            }
            const grainPosition = grains[grainIndex];
            if (grainPosition <= henPosition) {
                // Compute distance from hen to grain
                const distance = henPosition - grainPosition;
                if (distance > timeLimit) {
                    // Hen can't reach grain within the time limit
                    return false;
                }
                // Move up to the closest grain that the hen can reach within the time limit
                while (grainIndex < grainCount && grains[grainIndex] <= henPosition) {
                    grainIndex++;
                }
                // Move up to the grain that can be eaten in the optimal time
                while (grainIndex < grainCount && Math.min(distance, grains[grainIndex] - henPosition) + grains[grainIndex] - grain
                    grainIndex++;
                }
            } else {
                // Move up to the grain that the hen can eat within the time limit
                while (grainIndex < grainCount && grains[grainIndex] - henPosition <= timeLimit) {
                    grainIndex++;
                }
            }
        }
        // Check if all grains are distributed
        return grainIndex === grainCount;
    };

    // Perform a binary search to find the minimum time needed
    while (lowerBound < upperBound) {
        const mid = (lowerBound + upperBound) >> 1;
        if (canFeedWithinTimeLimit(mid)) {
            // If it's possible to feed within this time frame, search the lower half
            upperBound = mid;
        } else {
            // Otherwise, search the upper half
            lowerBound = mid + 1;
        }
    }

    // Return the smallest time within which all hens can eat
    return lowerBound;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed as follows:

1. Sorting both the `hens` and `grains` lists: `hens.sort()` and `grains.sort()` both have a time complexity of $O(n \log n)$ and $O(m \log m)$ respectively, where `n` is the number of hens and `m` is the number of grains.

2. Binary search: The `bisect_left` function performs binary search on a range of size `r`. The size of this range can be considered as `U` because it is determined by the maximum gap between positions of grains. Since it performs the check function at each step of the binary search, the time it takes is $O(\log U)$ for the binary search times the complexity of the `check` function itself.

3. `check` function: Inside the binary search, the `check` function is called, which runs in $O(m + n)$ because it goes through all grains and potentially all hens in the worst case.

Combining these factors, we get a total time complexity of $O(n \log n + m \log m + (m + n) \log U)$.

Space Complexity

The space complexity of the given code can be analyzed as follows:

1. Sorting requires $O(1)$ additional space if the sort is in-place (which Python's `sort` method is, for example).

2. The binary search uses $O(1)$ space.

3. The `check` function uses $O(1)$ space since it only uses a few variables and all operations are done in place.

Adding these up, we get a space complexity of $O(\log m + \log n)$ for the recursion stack of the sorting algorithms if the sort implementation used is not in-place.