2141. Maximum Running Time of N Computers

Hard Greedy Array Binary Search Sorting Leetcode Link

Problem Description

You're tasked with powering `n` computers using a given set of batteries, each with a specific charge duration in minutes. The goal is to make all `n` computers run simultaneously for the maximum amount of time possible. One battery can be used per computer, and you can switch batteries between the computers at any whole minute without any time penalty. However, the batteries can't be recharged - once a battery's charge is used up, it can't be used again.

Your function's job is to determine the maximum number of minutes that all `n` computers can be kept running. You're given `n` and an array `batteries`, where each element represents the lifetime of a battery.

Intuition

To solve this problem, we have to find a way to balance the battery use across all computers to maximize the time they can run. The intuitive leap is to think of a "load balancing" approach, where we distribute the power supply evenly amongst the computers. Since we're looking for a "fair share" of power that each computer would get, and we aim to maximize this share under the constraints, it seems that binary search could help us zero in on the solution.

We perform a binary search on the range of possible minutes (from `0` to the sum of all battery capacities) to find the maximum time each computer could run. The key intuition is that if it's possible to run all computers for `mid` minutes, then it's also possible to run them for any amount of time less than `mid`. Conversely, if it's not possible to run the computers for `mid` minutes, it won't be possible for any time greater than `mid`.

In each step of our binary search, we check if it's possible to run all computers for `mid` minutes. We do this by calculating the sum of `min(x, mid)` for each battery `x` in the array `batteries`. If this sum is at least `n * mid`, it means that it's possible to distribute the batteries in a way that all computers can run for `mid` minutes.

If it's possible, we move our search upward (`l = mid`); otherwise, we move downward (`r = mid - 1`). When our binary search converges, we've found the maximum time that we can run all computers simultaneously, which will be the value of `l`.

The solution code implements this binary search strategy to find the optimal run time.

Solution Approach

The implementation of the solution uses binary search as the central algorithm, applying this technique to find the maximum possible running time where all `n` computers can operate simultaneously using the batteries provided.

We'll break down the key components of the binary search loop from the reference solution:

1. **Initialization of Search Bounds**: We set our search bounds for binary search — `l` (left) is set to `0`, and `r` (right) is set to the sum of all elements in `batteries`. These represent the minimum and maximum possible running times, respectively.
2. **The Binary Search Loop**: Binary search proceeds by repeatedly dividing the search interval in half. If the condition is met, we work with the right half; otherwise, the left half. We continue until `l` and `r` converge.
3. **Middle Point Calculation**: `mid = (l + r + 1) >> 1`. Calculating the midpoint of our search range, where `>> 1` effectively divides the sum by 2 (bitwise right shift operation which is equivalent to integer division by 2).
4. **Testing if the Midpoint is Feasible**: `sum(min(x, mid) for x in batteries) >= n * mid`. For each battery `x`, we take the minimum of `x` and `mid` since if a battery has more charge than needed (`x > mid`), we only need up to `mid` for our current guess. We sum these minimums and compare it to `n * mid` to verify if we can run all computers for `mid` minutes: if `sum >= n * mid`, it's feasible.
5. **Updating Search Bounds**: Based on feasibility:
   - If it's feasible to run all computers for `mid` minutes, then we set `l` to `mid`. The `+1` in the midpoint calculation ensures we don't get stuck in an infinite loop.
   - If it's not feasible, we set `r` to `mid - 1`.
6. **Convergence and Result**: When `l` equals `r`, the binary search has converged to the maximum amount of time that all computers can run simultaneously, and we return `l`.

The solution uses a common data structure, a list (or array), to store the durations of batteries. There's no need for advanced data structures since the problem doesn't require any dynamic ordering or storage—just a straightforward computation of the sum under different conditions.

The pattern here is iterative refinement: with each iteration, we get closer to the maximum run time by ruling out half of the remaining possibilities. Binary search is highly efficient for this problem because it cuts down the search space exponentially with each iteration, leading us quickly to the optimal solution.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose we have `n = 2` computers and `batteries = [3, 7]`, which means we have 2 batteries that can last for 3 and 7 minutes, respectively. We want to find out the maximum amount of time that both computers can run simultaneously.

1. **Initialization of Search Bounds**: We set `l = 0` and `r = 10` (sum of all battery lifetimes).
2. **The Binary Search Loop** begins:

   a. **First Iteration**: - Calculate `mid = (0 + 10 + 1) >> 1 = 5`. - Check if `mid` is feasible: - Computer 1 can use a battery for 3 minutes (as 3 < 5). - Computer 2 can use a battery for 5 minutes (as 7 > 5, but we only need 5). - The total is 3 + 5 = 8, which is less than `n * mid` (2 * 5 = 10). Not feasible. - Update search bounds: `r = mid - 1 = 4`.

   b. **Second Iteration**: - Calculate `mid = (0 + 4 + 1) >> 1 = 2`. - Check if `mid` is feasible: - Computer 1 can use a battery for 2 minutes (as 3 > 2). - Computer 2 can use a battery for 2 minutes (as 7 > 2). - The total is 2 + 2 = 4, which equals `n * mid` (2 * 2 = 4). Feasible. - Update search bounds: `l = mid = 2`.

   Since `l` and `r` have not converged, the search continues.

   c. **Third Iteration**: - Calculate `mid = (2 + 4 + 1) >> 1 = 3`. - Check if `mid` is feasible: - Computer 1 can use a battery for 3 minutes (as 3 >= 3). - Computer 2 can use a battery for 3 minutes (as 7 > 3). - The total is 3 + 3 = 6, which is greater than `n * mid` (2 * 3 = 6). Feasible. - Update search bounds: `l = mid = 3`.

   d. **Fourth Iteration**: - Since `l` has been updated to 3, 'r' still being 4, so we need to search the upper half `[4, 4]`: - Calculate `mid = (3 + 4 + 1) >> 1 = 4`. - Check if `mid` is feasible: - Computer 1 can use a battery for 3 minutes (as 3 < 4). - Computer 2 can use a battery for 4 minutes (as 7 > 4). - The total is 3 + 4 = 7, which is less than `n * mid` (2 * 4 = 8). Not feasible. - Since it's not feasible, we set `r = mid - 1 = 3`.

3. **Convergence and Result**: `l` and `r` have converged to `3`, meaning the maximum time that both computers can run simultaneously using the given batteries is 3 minutes.

In this example, we've seen how binary search efficiently zeroes in on the maximum time by testing the feasibility at the midpoint of the current search interval and narrowing down based on the results of those tests.

Python Solution

```python
from typing import List

class Solution:
    def maxRunTime(self, n: int, batteries: List[int]) -> int:
        # Defines the search space with the minimum possible runtime as `left`
        # and the maximum as the sum of the capacities of all batteries as `right`
        left, right = 0, sum(batteries)

        # Performs binary search to find the maximum runtime
        while left < right:
            # Sets `middle` to the average of `left` and `right`
            # The `+1` ensures that the middle is biased towards the higher end
            # to prevent infinite loop in case `left` and `right` are consecutive numbers
            middle = (left + right + 1) // 2

            # Computes the total run time possible with each battery contributing
            # either its full capacity or the current `middle` value, which is the
            # assumed runtime for each UPS if batteries are distributed optimally
            total_runtime = sum(min(battery_capacity, middle) for battery_capacity in batteries)

            # If the total achievable runtime is at least `n * middle`, this means
            # we can guarantee each of the `n` UPS a runtime of `middle`
            # So we move the `left` boundary to `middle`
            if total_runtime >= n * middle:
                left = middle
            # Otherwise, we have overestimated the maximal achievable runtime,
            # and so we decrease the `right` boundary
            else:
                right = middle - 1

        # The search ends when `left` and `right` meet, meaning `left` is the maximal
        # achievable runtime for each UPS
        return left

```

Java Solution

```java
class Solution {

    // This method is designed to find the maximum running time for 'n' computers using given batteries
    public long maxRunTime(int n, int[] batteries) {
        long left = 0; // Initialize the lower bound of binary search
        long right = 0; // Initialize the upper bound of binary search

        // Calculate the sum of all the batteries which will be the upper bound
        for (int battery : batteries) {
            right += battery;
        }

        // Use binary search to find the maximum running time
        while (left < right) {
            // Calculate the middle point, leaning towards the higher half
            long mid = (left + right + 1) >> 1;
            long sum = 0; // Sum of the minimum of mid and the battery capacities

            // Calculate the sum of the minimum of mid or the battery capacities
            for (int battery : batteries) {
                sum += Math.min(mid, battery);
            }

            // If the sum is sufficient to run 'n' computers for 'mid' amount of time
            if (sum >= n * mid) {
                // We have enough capacity, so try a longer time in the next iteration
                left = mid;
            } else {
                // Otherwise, try a shorter time in the next iteration
                right = mid - 1;
            }
        }
        // Return the maximum running time found through binary search
        return left;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    long long maxRunTime(int n, vector<int>& batteries) {
        // Initialize the search space boundary variables.
        long long left = 0, right = 0;

        // Calculate the sum of all batteries as the upper bound for binary search.
        for (int battery : batteries) {
            right += battery;
        }

        // Perform binary search to find the optimal runtime.
        while (left < right) {
            // Mid represents a candidate for the maximum runtime.
            long long mid = (left + right + 1) >> 1;

            // Sum variable to store the total uptime using the candidate runtime.
            long long sum = 0;

            // Calculate total runtime without exceeding individual battery's capacity.
            for (int battery : batteries) {
                sum += min(static_cast<long long>(battery), mid); // Use long long to avoid overflow.
            }

            // If total uptime is enough to support n UPSs running for 'mid' duration, search in the higher half.
            if (sum >= n * mid) {
                left = mid;
            } else { // Otherwise, search in the lower half.
                right = mid - 1;
            }
        }

        // At this point, 'left' is the max runtime possible for n UPSs.
        return left;
    }
};

```

Typescript Solution

```typescript
function maxRunTime(numBatteries: number, batteries: number[]): number {
    // Define the range for possible maximum run times using BigInt for large numbers handling
    let low = 0n;
    let high = 0n;

    // Calculate the sum of all batteries' capacities
    for (const capacity of batteries) {
        high += BigInt(capacity);
    }

    // Use binary search to find the maximum running time
    while (low < high) {
        // Calculate the middle value of the current range
        const mid = (low + high + 1n) >> 1n;
        let sum = 0n;

        // Sum the minimum between each battery capacity and the mid value
        for (const capacity of batteries) {
            sum += BigInt(Math.min(capacity, Number(mid)));
        }

        // Check if the current mid value can be achieved with the available batteries
        if (sum >= mid * BigInt(numBatteries)) {
            // If yes, then mid is a possible maximum run time, so we discard the left half of the search range
            low = mid;
        } else {
            // Otherwise, discard the right half of the search range
            high = mid - 1n;
        }
    }

    // At the end of the while loop, low will have the maximum runtime value, convert it to number before returning
    return Number(low);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by the binary search and the summation of the batteries within the while loop:

- **Binary Search**: The binary search runs on the interval from `l` to `r` and the interval is halved in each iteration of the loop. Since `r` is initialized as the sum of all elements in `batteries`, the maximum number of iterations of the binary search would be `O(log(S))` where `S` is the sum of the `batteries` array.
- **Summation within Loop**: Within each iteration of the binary search, there is a summation of the minima of the individual battery and `mid`, which is `O(N)`, where `N` is the number of elements in `batteries`.

Combining these two steps, the overall time complexity is `O(N * log(S))`.

Space Complexity

The space complexity of the code is `O(1)`:

- There are a few variables initialized (`l`, `r`, `mid`) that use constant space.
- The summation operation uses `min()` within a generator expression which computes the sum in-place and doesn't require extra space proportional to `N`.
- The binary search does not rely on any additional data structures.

Therefore, the additional space used by the program is constant, irrespective of the input size.