2594. Minimum Time to Repair Cars

Medium | Array | Binary Search | Leetcode Link

Problem Description

In this problem, we are tasked with determining the minimum amount of time needed to repair a certain number of cars (`cars`) given the ranks of available mechanics (`ranks`). Each mechanic has a unique rank, which determines their repair efficiency. The time taken for a mechanic of rank `r` to repair `n` cars is calculated by the formula `r * n^2`, meaning that as a mechanic repairs more cars, the time taken increases exponentially with respect to the number of cars. An important note is that mechanics can work on repairing cars at the same time, which significantly affects how we approach finding the minimum time.

We are asked to return the smallest possible total time required to repair all the cars in the garage. The goal is to efficiently assign cars to the available mechanics in such a way that the total time is minimized, taking into account that higher-ranked mechanics repair cars faster.

Intuition

The problem hints at an optimization scenario where we are looking to minimize a certain quantity (repair time) subject to a set of conditions (number of cars and mechanic ranks). Such problems are often candidates for a binary search approach if we can define a monotonic function that relates the quantity to be optimized with the decision variable. In this case, the decision variable is time, and the monotonic function is the total number of cars that can be repaired within that time.

Intuitively, if we allow more time for repairs, we can repair more cars. If we allow less time, fewer cars can be repaired. This property allows us to perform a binary search for the minimum time required as follows:

1. We start with a range of possible times — the lower bound (left) being zero (assuming instantaneous repair is impossible) and the upper bound (right) being the slowest mechanic's rank multiplied by the square of the total number of cars (assuming the case where only the slowest mechanic repairs all the cars alone).
2. We aim to find the smallest time `t` where the combined total of cars that can be repaired within that time by all mechanics is at least the number of cars needing repair.
3. To do this, we define a check function that takes time `t` as an argument and calculates how many cars would be fixed by all mechanics in that time. If the total is greater than or equal to `cars`, it means that `t` is a feasible solution.
4. The binary search iterates by choosing a midpoint in the current left-right range, using the check function to see if it's a feasible solution, and adjusting the search range accordingly (if `mid` is feasible, look left for a potentially smaller feasible time; if not, look right).
5. When the search is complete, the left edge of our search space represents the smallest feasible time.

Understanding that the number of cars a mechanic of rank `r` can repair in time `t` is the integer square root of `t / r` springs from rearranging the equation `time = rank * num_cars^2`. This leads us to the realization that the process can be expedited using efficient computation of square roots and integer flooring to count cars repaired within time `t`.

The provided solution utilizes this understanding, implementing a binary search using Python's `bisect_left` function and a custom check function, to efficiently converge on the minimum repair time.

Solution Approach

The solution provided employs a common algorithmic pattern called binary search to efficiently find the minimum time required to repair all `cars`. Binary search is a divide-and-conquer strategy that can significantly reduce the search space and time when looking for a specific value in a sorted array or, as in our case, when seeking a threshold in a monotonically increasing or decreasing function. In this context, the time taken to repair cars can be seen as such a function.

Here's how the solution incorporates binary search, step by step:

1. Set up the search space for the possible minimum time. We define two variables, `left` as 0, and `right` as `ranks[0] * cars * cars`. The right boundary is an overestimate, assuming that only the slowest mechanic, with the smallest rank, repairs all the cars.
2. The core algorithm relies on the `bisect_left` function from Python's `bisect` module, which performs binary search. This function is used to find the leftmost insertion point in a sorted list (or range in our case) where a given condition is true. In this implementation, the range goes from `left` to `right`, and we are searching for the first instance where the condition evaluates to `True`.
3. The `check` function acts as the condition for `bisect_left`. For a given time `t`, it determines whether the sum of cars repaired by all mechanics within that time frame is at least equal to the total number of cars that need to be repaired. To find the number of cars that each mechanic can repair within time `t`, we calculate the integer square root of `t // r`, where `r` is the rank of the mechanic.
4. Inside the `check` function, the integer square root is computed for each mechanic with `int(sqrt(t // r))`. This operation gives us the maximum number of cars that a mechanic could repair in time `t` without exceeding it. By using integer division and the floor of the square root, we ensure we get a whole number of cars as it isn't practical to repair a fraction of a car.
5. Once `check` returns `True` for some value of `t`, it means that at least `cars` number of cars can be repaired in `t` minutes by the available mechanics. This `t` is either the minimum time required, or there might exist an even smaller minimum time; hence `bisect_left` would continue to narrow down the range until it cannot be reduced further.
6. The value returned by `bisect_left` is assigned to and returned by the `repairCars` function, thus providing us with the minimum time we sought.

The solution effectively uses the mechanic's repair formula `r * n^2` to assess feasibility within different time frames iteratively, leveraging both the power of binary search for fast narrowing down of possibilities and the mathematical properties of square roots for accurate computation of cars repaired over time.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have three mechanics with ranks `[2, 3, 5]` (where a smaller number represents a higher rank and greater efficiency) and we need to repair `7` cars.

- Our first step is to determine an initial search space for the minimum time required. In this case, let's assume the slowest mechanic (rank `5`) would take the longest time if they repaired all `7` cars by themselves. The formula `r * n^2` would give us `5 * 7 * 7 = 245` as the maximum time. Therefore, our search space for time `t` starts at `left = 0` and ends at `right = 245`.
- Using the binary search approach, we find a midpoint in the current search space. Let's try the midpoint of our initial range, which is `(0+245)//2 = 122`. Now, we use the check function to see how many cars can be repaired by the mechanics in `122` minutes. We compute the integer square root of `122 // r` for each mechanic to find out how many cars each mechanic can repair.
  - For the mechanic with a rank of `2`, we have `int(sqrt(122 // 2)) = int(sqrt(61)) ≈ 7` cars.
  - For the mechanic with a rank of `3`, we have `int(sqrt(122 // 3)) = int(sqrt(40)) ≈ 6` cars.
  - For the mechanic with a rank of `5`, we have `int(sqrt(122 // 5)) = int(sqrt(24)) ≈ 4` cars.

  We can see that the first mechanic alone could repair upward of `7` cars in this time, so `122` minutes is certainly enough time to repair all `7` cars.
- Because `122` is a feasible solution where more cars can be repaired than necessary, we then adjust our search space to look for potentially smaller times. We set the `right` boundary to `122` and recalculate the midpoint, continuing our binary search.
- We repeat the above steps iteratively, updating our `left` and `right` range boundaries based on the outcome of the check function until the gap between `left` and `right` cannot be reduced further.
- After the `check` function returns `True` for the smallest value of `t`, `bisect_left` will conclude that we have found the minimum time where at least `7` cars can be repaired by the mechanics.

By dividing the problem space and using the mechanics' repair capacity as a guide, we're able to narrow down the minimum time needed efficiently. In practice, the actual implementation may involve a few iterations of this binary search process to pinpoint the exact minimum time.

Python Solution

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

class Solution:
    def repairCars(self, ranks: List[int], cars: int) -> int:
        # Define a function to check if a given time 't' is sufficient
        # to repair 'cars' number of cars with the given 'ranks'.
        def is_time_sufficient(t: int) -> bool:
            # For each rank, calculate how many cars can be repaired
            # by the given time 't', and sum them all up. If the sum
            # equals or exceeds the total number of cars needed to be repaired,
            # then the time 't' is sufficient.
            return sum(int(sqrt(t // rank)) for rank in ranks) >= cars

        # Since 'check' function returns a boolean, we need a range
        # to apply bisect_left on. The range starts from 0 to an upper limit.
        # The upper limit is the maximum rank times the square of the number
        # of cars as a worst-case scenario for the required time.
        max_time = ranks[0] * cars * cars

        # Perform a binary search for the leftmost time 't' for which
        # is_time_sufficient(t) is True. This will be the minimum time
        # required to repair 'cars' cars given the 'ranks'.
        min_time_required = bisect_left(range(max_time), True, key=is_time_sufficient)

        # Return the minimum time found
        return min_time_required

```

Java Solution

```java
class Solution {
    // Repair cars by using ranks of mechanics to calculate the minimum time.
    public long repairCars(int[] ranks, int totalCars) {
        long low = 0; // Set the lower bound of the search space to 0.
        // Set the upper bound of the search space:
        // the product of the maximum rank, total number of cars and total cars squared.
        long high = 1L * ranks[0] * totalCars * totalCars;

        // Implement binary search to find the minimum amount of time needed.
        while (low < high) {
            long mid = (low + high) >> 1; // Find the midpoint between low and high.
            long count = 0; // Initialize count of cars that can be repaired in 'mid' time.

            // Calculate the number of cars each mechanic can fix in 'mid' time.
            for (int rank : ranks) {
                count += Math.sqrt(mid / rank);
            }

            // If count is at least equal to the total number of cars,
            // we could potentially reduce the high to mid.
            if (count >= totalCars) {
                high = mid;
            } else {
                // Otherwise, we have to increase the low to mid + 1 to find the minimum time.
                low = mid + 1;
            }
        }

        // When low meets high, we've found the minimum time needed for the repairs.
        return low;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cmath>

class Solution {
public:
    // Function to calculate minimum time required to repair `cars` number of cars
    // by mechanics with various ranks.
    // `ranks`: A vector of integers where each element represents the rank of a mechanic
    // `cars`: The total number of cars that need to be repaired
    long long repairCars(vector<int>& ranks, int cars) {
        // Set lower bound of search space to 0
        long long lowerBound = 0;
        // Set upper bound of search space to maximum possible value
        // assuming the worst mechanic has to repair all cars, one at a time.
        long long upperBound = 1LL * ranks[0] * cars * cars;

        // Perform binary search to find the minimum time
        while (lowerBound < upperBound) {
            // Calculate the middle value of the search space
            long long mid = (lowerBound + upperBound) >> 1;
            long long count = 0; // Count of cars that can be repaired in 'mid' time

            // Loop through each mechanic's rank
            for (int rank : ranks) {
                // Each mechanic can repair floor(sqrt(mid / rank)) cars in 'mid' time
                count += std::sqrt(mid / rank);
            }

            // If the count of cars repaired is equal to or greater than the number needed,
            // we can potentially reduce the upper bound of the search space to 'mid'.
            if (count >= cars) {
                upperBound = mid;
            }
            // Otherwise, we need more time, so we'll increase the lower bound
            // of the search space.
            else {
                lowerBound = mid + 1;
            }
        }
        // At the end of the while loop, 'lowerBound' is the minimal time
        // required to repair the cars, hence return 'lowerBound'.
        return lowerBound;
    }
};

```

Typescript Solution

```typescript
function repairCars(ranks: number[], carsToRepair: number): number {
    // Initialize the search interval for the minimum time needed.
    let minTime = 0;
    // The right end of the search interval is set assuming the mechanic with the lowest rank
    // takes the square of the total number of cars time units for each car.
    let maxTime = ranks[0] * carsToRepair * carsToRepair;

    // Use binary search to find the minimum time required to repair the cars.
    while (minTime < maxTime) {
        // Calculate the middle time to check how many cars can be repaired by this time.
        const midTime = minTime + Math.floor((maxTime - minTime) / 2);
        let carsRepaired = 0;

        // Sum up the total number of cars that can be repaired by midTime by each mechanic.
        for (const rank of ranks) {
            carsRepaired += Math.floor(Math.sqrt(midTime / rank));
        }

        // Check if the number of repaired cars meets or exceeds the target amount of cars.
        if (carsRepaired >= carsToRepair) {
            // If more than required cars can be repaired, search the left (lower) half.
            maxTime = midTime;
        } else {
            // If not enough cars can be repaired, search the right (higher) half.
            minTime = midTime + 1;
        }
    }

    // minTime is our answer since it's the minimum time required to repair the given amount of cars.
    return minTime;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily dictated by two factors: the use of binary search and the `check` function that is called within it.

The binary search is performed on a range of `ranks[0] * cars * cars`. Since binary search has a time complexity of `O(log N)` where `N` is the size of the element space you are searching over, the `log` component refers to the powers of two that you divide the search space by. Therefore, this portion contributes a `log(ranks[0] * cars * cars)` complexity.

However, for each step of the binary search, the `check` function is called. This function iterates over every rank and performs an operation that takes constant time, `int(sqrt(t // r))`, and sums the results. Since this iteration is over all mechanics which is `n` in number, the operation within the `check` function has a complexity of `O(n)`.

Consequently, the overall time complexity of the algorithm is the product of the two, which corresponds to `O(n * log(ranks[0] * cars * cars))`. It is simplified to `O(n * log n)` in the reference answer under the assumption that `ranks[0] * cars * cars` grows proportionally to `n^2`, making the `log(ranks[0] * cars * cars)` a constant multiplier of `log n`.

Space Complexity

The space complexity is `O(1)` because the algorithm uses a fixed amount of additional space. The variables used within the `check` function and the storage for the result of `bisect_left` do not grow with the size of the input list `ranks`.