2323. Find Minimum Time to Finish All Jobs II

Medium Greedy Array Sorting

Problem Description

You are provided with two integer arrays, `jobs` and `workers`, each indexed starting from 0. The arrays have the same lengths, meaning they contain an equal number of elements. `jobs[i]` represents the time required to complete the i-th job, while `workers[j]` represents the time a j-th worker can dedicate to working each day. The task is to assign each job to one worker so that every worker gets exactly one job. Your goal is to calculate the minimum number of days necessary to finish all the jobs after they have been allocated to the workers.

Intuition

The intuition behind the solution involves two observations:

1. Sorting Pattern: If we sort both the `jobs` and `workers` arrays in ascending order, we ensure the biggest jobs are matched with the workers who can work the most per day. This helps us in minimizing the overall time since we are using the maximum capacity of the most efficient workers.
2. Minimum Days Calculation: After sorting, we pair each job with a worker. To calculate the number of days a worker needs to complete a job, we can take the time needed for the job (`a`) and divide it by the time the worker can put in each day (`b`). Since we are dealing with whole days, we need to round up, which is achieved by `(a + b - 1) // b` in Python (integer division with ceiling).

The `max` function applied to the result of these calculations across all job-worker pairs ensures that we find the scenario that takes the longest. This is the minimum number of days needed to complete all jobs, since all jobs are being worked on simultaneously, and once the job taking the longest is done, all other jobs would have also been completed.

Solution Approach

The solution uses a greedy algorithm approach which is implemented through sorting and pairing, leading to an efficient way to calculate the minimum time necessary to complete all jobs. Here's a walkthrough of the implementation:

- Firstly, we sort both the `jobs` and `workers` arrays in ascending order. By doing this, we can pair the largest job with the worker who can work the longest hours, and so on down the line. Sorting is a common algorithmic pattern for problems related to optimization, as it often helps to align data in a way that simplifies pairing or comparison.
- We then iterate over the pairs of sorted `jobs` and `workers`. These pairs are generated by zipping the two arrays together using Python's built-in `zip()` function, which pairs elements with the same index from each of the input sequences.
- For each pair `(a, b)` taken from `zip(jobs, workers)`, we calculate the number of days it would take for worker `b` to complete job `a`. This is where the formula `(a + b - 1) // b` comes into play. It ensures that we perform the integer division with a ceiling effect since we are dealing with full days. If a job doesn't require a full extra day, we still count the partial day as a full one, since a worker cannot be partially assigned to a job each day.
- The `max()` function is then used on the sequence of these calculations. Because each worker is working on different jobs in parallel, the overall completion time for all jobs is dictated by the job-worker pair that takes the longest. This longest time will be the maximum value out of the individual times calculated.

In summary, by sorting the jobs and workers, we facilitate an optimal pairing strategy. Then we calculate the time each optimal job-worker pair takes, ensuring to account for partial days as full. Lastly, we identify the longest duration out of these pairs as our solution.

Data Structures used:

- Arrays: to store the input `jobs` and `workers` data and manipulate it through sorting.
- Integers: to represent individual job times and worker capacities, and to calculate and represent the minimum number of days required.

Patterns used:

- Sorting: to optimally align the jobs to workers.
- Greedy: to pair the highest capacity workers with the largest jobs to minimize overall time.
- Iteration: to go through the pairs of jobs and workers and calculate the time taken for each.
- Calculation: to determine the number of days for each pair and identify the maximum completion time.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the following arrays for jobs and workers:

```
jobs = [3, 2, 7]
workers = [1, 5, 3]
```

Here's how we implement our solution:

1. First, we sort both arrays to align the largest jobs with the workers who can dedicate the most time per day. After sorting:

   ```
   jobs = [2, 3, 7]
   workers = [1, 3, 5]
   ```

2. Next, we pair each job with a worker by zipping the two sorted arrays.

   ```
   paired = zip([2, 3, 7], [1, 3, 5])
   paired = [(2, 1), (3, 3), (7, 5)]
   ```

3. Now, for each paired tuple, calculate the number of days it takes for the worker to finish the job using `(a + b - 1) // b`. Here, `a` is the time needed for the job, and `b` is the time the worker can work per day.
   - For the first pair `(2, 1)`, the calculation is `(2 + 1 - 1) // 1 = 2 days`.
   - For the second pair `(3, 3)`, the calculation is `(3 + 3 - 1) // 3 = 1 day`.
   - For the third pair `(7, 5)`, the calculation is `(7 + 5 - 1) // 5 = 2 days`.
4. Last, we find the maximum number of days from all the pairs; this will be the minimum number of days required to complete all the jobs.

   ```
   max_days = max([2, 1, 2]) = 2
   ```

So, with the given job and worker assignments, the minimum number of days required to finish all the jobs is 2 days.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def minimumTime(self, jobs: List[int], workers: List[int]) -> int:
        # Sort the lists to ensure that the shortest jobs are matched with
        # the workers who take the least amount of time per unit.
        jobs.sort()
        workers.sort()

        # Calculate the time taken for each worker to complete their job.
        # We're using (job duration + worker speed - 1) // worker speed to
        # determine the minimum number of full time units needed for completion.
        # The -1 is used to ensure that we do not add an extra unnecessary
        # time unit if job duration is perfectly divisible by worker speed.
        # For each pair of job and worker, the time required for that worker
        # to complete their job is calculated.
        times = [(job_duration + worker_speed - 1) // worker_speed for job_duration, worker_speed in zip(jobs, workers)]

        # The maximum time among all the workers is what will determine
        # the total time required to finish all jobs, since we have to wait for
        # the slowest job-worker pair to finish.
        return max(times)
```

Java

```java
import java.util.Arrays; // Necessary import for Arrays.sort()

class Solution {

    // Method to calculate the minimum time needed to assign jobs to workers such that each worker gets exactly one job.
    public int minimumTime(int[] jobs, int[] workers) {
        Arrays.sort(jobs); // Sort the array of jobs in non-decreasing order.
        Arrays.sort(workers); // Sort the array of workers in non-decreasing order.

        int maximumTime = 0; // Initialize maximum time to 0.

        for (int i = 0; i < jobs.length; ++i) {
            // Calculate the time it takes for worker i to complete job i.
            // Divide the job[i] work units by the workers[i] efficiency, rounding up to the nearest whole number.
            int currentTime = (jobs[i] + workers[i] - 1) / workers[i];

            // Update maximumTime to be the maximum of itself and the time calculated for the current worker.
            maximumTime = Math.max(maximumTime, currentTime);
        }

        // Return the overall maximum time it takes for all jobs to be completed.
        return maximumTime;
    }
}
```

C++

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

class Solution {
public:
    // This function calculates the minimum time required to assign jobs to workers
    // such that each job is assigned to one worker, and the time taken is minimized.
    // The time taken for each job-worker pair is the ceil of division of job by worker.
    int minimumTime(vector<int>& jobs, vector<int>& workers) {
        // Sort the jobs and workers vectors to match the smallest job with the fastest worker.
        sort(jobs.begin(), jobs.end());
        sort(workers.begin(), workers.end());

        int minimumTimeRequired = 0; // Initialize the minimum time required as 0.

        // Iterate over the matched pairs of jobs and workers.
        for (int i = 0; i < jobs.size(); ++i) {
            // Calculate the time taken for each job-worker pair, which is jobs[i] divided by workers[i],
            // rounded up to the nearest integer, and update minimumTimeRequired if this time is larger.
            minimumTimeRequired = max(minimumTimeRequired, (jobs[i] + workers[i] - 1) / workers[i]);
        }

        // Return the maximum time out of all job-worker pairs as that would be the bottleneck.
        return minimumTimeRequired;
    }
};
```

TypeScript

```typescript
// Import the necessary functions from lodash for sorting and max operations
import { sortBy, max } from 'lodash';

// This function calculates the minimum time required to assign jobs to workers
// such that each job is assigned to one worker, and the time taken is minimized.
// The time taken for each job-worker pair is the ceil of the division of the job by the worker.
function minimumTime(jobs: number[], workers: number[]): number {
    // Sort the jobs and workers arrays to match the smallest job with the fastest worker.
    jobs = sortBy(jobs);
    workers = sortBy(workers);

    let minimumTimeRequired = 0; // Initialize the minimum time required as 0.

    // Iterate over the matched pairs of jobs and workers.
    for (let i = 0; i < jobs.length; ++i) {
        // Calculate the time taken for each job-worker pair, which is jobs[i] divided by workers[i],
        // rounded up to the nearest integer, by adding workers[i] - 1 before the division.
        // Update minimumTimeRequired if this time is larger.
        minimumTimeRequired = max([minimumTimeRequired, Math.ceil(jobs[i] / workers[i])]);
    }

    // Return the maximum time out of all job-worker pairs as that would be the bottleneck.
    return minimumTimeRequired;
}

// Usage example (if you want to test the function):
// const jobsExample = [3, 2, 10];
// const workersExample = [1, 2, 3];
// const minTime = minimumTime(jobsExample, workersExample);
// console.log(minTime); // Output will be the minimum time required based on the job and worker assignments.
```


```python
from typing import List

class Solution:
    def minimumTime(self, jobs: List[int], workers: List[int]) -> int:
        # Sort the lists to ensure that the shortest jobs are matched with
        # the workers who take the least amount of time per unit.
        jobs.sort()
        workers.sort()

        # Calculate the time taken for each worker to complete their job.
        # We're using (job duration + worker speed - 1) // worker speed to
        # determine the minimum number of full time units needed for completion.
        # The -1 is used to ensure that we do not add an extra unnecessary
        # time unit if job duration is perfectly divisible by worker speed.
        # For each pair of job and worker, the time required for that worker
        # to complete their job is calculated.
        times = [(job_duration + worker_speed - 1) // worker_speed for job_duration, worker_speed in zip(jobs, workers)]

        # The maximum time among all the workers is what will determine
        # the total time required to finish all jobs, since we have to wait for
        # the slowest job-worker pair to finish.
        return max(times)
```


Time and Space Complexity

Time Complexity

The time complexity of the given code can be broken down into the following parts:

1. Sorting the `jobs` list: Sorting an array of `n` items has a time complexity of `O(n log n)`.
2. Sorting the `workers` list: This is another sorting operation also with `O(n log n)`, assuming `workers` list has the same length as `jobs`.

Since these are two consecutive operations, the combined time complexity for both sorts will still be `O(n log n)` because the constants are dropped in Big O notation.

3. The `zip` function and the comprehension: Creating pairs of jobs and workers with `zip` is `O(n)` and the comprehension iterates over each pair once, making it `O(n)` as well.

Combining all the above, we see that the most time-consuming operations are the sorts. Therefore, the overall time complexity of the code is `O(n log n)`.

Space Complexity

For space complexity:

1. Sorting the lists in-place: Since Python's sort method on lists sorts the list in place, it doesn't use extra space proportional to the input size. Therefore, the space complexity for the sort operation is `O(1)`.
2. `zip` and the comprehension: `zip` takes `O(1)` additional space as it returns an iterator, and the comprehension also takes `O(1)` space because it only computes the maximum time without storing intermediate results in an array or list.

Hence, the total space complexity of the code is `O(1)`.