826. Most Profit Assigning Work

Greedy Array Two Pointers Binary Search Sorting

Problem Description

their best possible job.

Medium

job, and profit[i] represents the profit for completing the i-th job. There's also a worker array where worker[j] indicates the maximum job difficulty that the j-th worker can handle. A worker can only be assigned to a job if the job's difficulty is less than or equal to the worker's ability. Also, a worker can only do

it. These are listed in the arrays difficulty and profit respectively, where difficulty[i] represents the difficulty of the i-th

You are given a scenario where there are n different jobs and m workers. Each job has a difficulty level and a profit associated with

one job while a job can be completed by multiple workers. The objective is to maximize the total profit gained by assigning workers to jobs.

Intuition

To find the maximum profit, we should assign the most profitable jobs that workers can perform to them. However, since the jobs and workers are unsorted and a worker can only perform jobs up to their ability, we need an efficient way to match workers with

The intuition behind the solution is to first sort the jobs based on their difficulty, ensuring that we always encounter the less difficult jobs first. Simultaneously, we sort the workers based on their ability so we can sequentially assign them to jobs without backtracking. After this, we can iterate through the workers in ascending order of their ability.

For each worker, we iterate through the sorted jobs, updating the maximum profit (t) that this worker can generate (only if the job difficulty is less than or equal to the worker's ability). Since the jobs are sorted by difficulty, once a job's difficulty is greater than a worker's ability, we can stop the search for that worker and proceed to the next one, because all subsequent jobs will also be

too difficult. As we find the best job for each worker, we accumulate the total profit (res). Once all workers have been assigned jobs (or determined they cannot complete any jobs), the accumulated res will contain the maximum total profit that can be achieved.

The implementation of the solution follows these steps: Preparation of Job Data: Before we start assigning jobs to the workers, we need to prepare the job data. We do this by

pairing each job's difficulty with its profit, creating a list of tuples (difficulty[i], profit[i]), and sorting this list by

difficulty. By doing so, we ensure that when we go through the jobs for a worker, we start with the easiest job that provides

profit and work our way up.

jobs.

Solution Approach

Sorting Workers: We sort the worker array. This sorting is crucial because it allows us to linearly assign jobs to each worker

without the need to check all jobs for every worker. Since the workers are sorted by their ability, once a worker can't do a specific job, we know that all following workers won't be able to do that job either (since they will be more skilled). Iterating and Matching Jobs to Workers: We go through each worker in ascending order and try to find the most profitable job that they can perform. An index i keeps track of the current job, and for each worker, we check jobs starting from this

index. When we encounter a job that the worker can do, we update t to the maximum profit seen so far. The t value

represents the best profit a worker with current ability can earn. By doing this, we won't miss any less difficult, higher-paying

Accumulating Profit: As we find the right job for each worker, we increment the total profit res by t, which at this point would

- have the highest possible profit that a worker could make according to their ability. **Returning the results**: After we have gone through all workers and maximized each of their profits, the variable res holds the maximum total profit that can be achieved, which we return as the final result.
- The solution utilizes greedy algorithms and sorts as the core patterns. Greedy algorithms are employed to ensure we are getting the maximum profit per worker before moving on. The data structures used include arrays/lists and tuples. Arrays/lists are mainly for recording workers' abilities, job difficulty, and profit, while tuples are used for pairing difficulty and profit for more convenient sorting and iteration.

workers to jobs with a complexity of $0(n \log n + m \log m)$ where n is the number of jobs, and m is the number of workers. This is because the sorting operations dominate the overall complexity. The linear pass used to match workers to jobs does not increase

Excellent use of Python's built-in sorting functionality and a double-pointer pattern allows the algorithm to efficiently match

the complexity as it's bounded by O(n + m) which is less than the sorting complexity. **Example Walkthrough**

• difficulty = [2, 4, 6, 8] • profit = [20, 40, 70, 80] • worker = [2, 7, 5] Following the steps from the solution approach:

Step 3: Iterating and Matching Jobs to Workers Now we iterate over each worker and find the highest profit job they can do:

• For the third worker (ability = 7), they can do the jobs with difficulty 2, 4, and 6. The job with difficulty 6 offers a profit of 70, which is the

facilitated by sorting both the job pairs and the workers to make iteration straightforward and efficient. The final total profit is

Step 5: Returning the results The maximum total profit that can be achieved with the given workers is res = 130.

In this case, the list is already sorted by difficulty.

After sorting: [2, 5, 7]

Workers before sorting: [2, 7, 5]

Let's walk through a small example to illustrate the solution approach as described above.

Suppose we have the following job difficulties and profits, and worker capabilities:

• For the first worker (ability = 2), the best job they can do is the one with difficulty 2 and profit 20. We set t = 20. • For the second worker (ability = 5), they can also do the job with difficulty 4, which has a profit of 40. But since the job with difficulty 2 and a

profit of 20 (from the previous computation) is in their range, we compare profits and still set t = 40 as it's higher.

Total profit res = 20 (first worker) + 40 (second worker) + 70 (third worker) = 130.

Step 1: Preparation of Job Data Pair up the job difficulties with the corresponding profits and sort them:

highest they can earn, so we update t = 70. Step 4: Accumulating Profit We add up the profits calculated by t for each worker:

Job data before sorting: [(2, 20), (4, 40), (6, 70), (8, 80)]

After sorting by difficulty: [(2, 20), (4, 40), (6, 70), (8, 80)]

Step 2: Sorting Workers Sort the worker array by their ability:

In this example, each worker was matched to the most profitable job they could do following a greedy approach, which was

job_count = len(difficulty)

Iterating through each worker

job_index += 1

for (int i = 0; i < jobCount; ++i) {</pre>

Arrays.sort(worker);

// Sort the jobs list by their difficulty

// Iterate through each worker's ability

jobIndex++; // Move to the next job

// Importing the required functionalities from standard libraries

// Creating an array of jobs by pairing difficulties with profits

jobs.push({ difficulty: difficulties[i], profit: profits[i] });

let maxProfit = 0; // Variable to keep track of the maximum profit

// Iterate over each worker to find the best job they can perform

return maxProfit; // Return the total maximum profit that can be earned

let jobIndex = 0; // Index to iterate through the sorted jobs

// Function to calculate the maximum profit that can be earned by assigning jobs to workers

let bestProfit = 0; // Keeps the best profit at the current or a lower difficulty level

// Update the best profit while the worker's ability is higher or equal to the job's difficulty

// Structure to hold a job with its difficulty and profit

import { max, sortBy } from 'lodash';

const numJobs = difficulties.length;

for (let i = 0; i < numJobs; i++) {</pre>

// Sorting the jobs based on difficulty

// Sorting the workers based on their ability level

sortBy(jobs, job => job.difficulty);

for (const workerAbility of workers) {

maxProfit += bestProfit;

interface Job {

difficulty: number;

function maxProfitAssignment(

difficulties: number[],

const jobs: Job[] = [];

profits: number[],

workers: number[]

sortBy(workers);

): number {

profit: number;

// Sum up the maximum profit the worker can make

for (int ability : worker) {

totalProfit += maxProfit;

jobs.sort(Comparator.comparing(a -> a[0]));

for capability in worker:

Solution Implementation

Python

class Solution:

maximized as required by the problem statement.

from typing import List

def max_profit_assignment(self, difficulty: List[int], profit: List[int], worker: List[int]) -> int:

Pairing each job's difficulty with its profit and storing them as tuples

while job_index < job_count and jobs[job_index][0] <= capability:</pre>

public int maxProfitAssignment(int[] difficulty, int[] profit, int[] worker) {

// Add each job's difficulty and profit as int array to the jobs list

int jobCount = difficulty.length; // The total number of jobs

jobs.add(new int[] {difficulty[i], profit[i]});

// Sort the worker array to prepare for the job assignment

int totalProfit = 0; // Variable to keep track of the total profit

while (jobIndex < jobCount && jobs.get(jobIndex)[0] <= ability) {</pre>

maxProfit = Math.max(maxProfit, jobs.get(jobIndex)[1]);

// Update the maximum profit if the current job offers more

int jobIndex = 0; // Index to iterate through the sorted jobs

int maxProfit = 0; // Variable to keep track of the maximum profit found so far

// While the job index is within bounds and the worker can handle the job difficulty

total_profit = 0 # Summing up the total profit for all workers

max_profit = max(max_profit, jobs[job_index][1])

job_index = 0 # Index to keep track of the current job

jobs = [(difficulty[i], profit[i]) for i in range(job_count)] # Sorting jobs based on their difficulty jobs.sort(key=lambda x: x[0]) # Sorting workers based on their capabilities worker.sort() max_profit = 0 # Tracks the maximum profit that can be achieved so far

Updating the max_profit by comparing with each job's profit if the worker can handle the job

List<int[]> jobs = new ArrayList<>(); // A list to hold jobs with their difficulty and profit

Accumulating the profit earned by this worker based on max_profit so far total_profit += max_profit # Returning the total profit that can be earned by all workers return total_profit

class Solution {

Java

```
return totalProfit; // Return the total profit from all workers
C++
#include <vector> // Required for using the vector
#include <algorithm> // Required for using the sort and max functions
class Solution {
public:
    int maxProfitAssignment(std::vector<int>& difficulties, std::vector<int>& profits, std::vector<int>& workers) {
        int numJobs = difficulties.size();
        std::vector<std::pair<int, int>> jobs; // Pairing each difficulty with its profit
       // Create a job list by pairing difficulties with profits
        for (int i = 0; i < numJobs; ++i) {
            jobs.push_back({difficulties[i], profits[i]});
       // Sort the jobs based on difficulty (the first element of the pair)
       std::sort(jobs.begin(), jobs.end());
       // Sort the workers based on their ability level
        std::sort(workers.begin(), workers.end());
        int maxProfit = 0; // To keep track of the maximum profit that can be earned
        int jobIndex = 0; // Index to iterate through the sorted jobs
        int bestProfit = 0; // Keeps the best profit at current worker's difficulty level or below
       // for every worker, find the best job that the worker can perform
        for (auto workerAbility : workers) {
            // Keep updating the best profit while the worker's ability is higher or equal to the difficulty
            while (jobIndex < numJobs && jobs[jobIndex].first <= workerAbility) {</pre>
                bestProfit = std::max(bestProfit, jobs[jobIndex].second);
                jobIndex++;
           // After finding the best profit for the current worker, add it to the total maxProfit
           maxProfit += bestProfit;
       return maxProfit; // Return the total max profit that can be earned
};
TypeScript
```

while (jobIndex < numJobs && jobs[jobIndex].difficulty <= workerAbility) {</pre> bestProfit = max([bestProfit, jobs[jobIndex].profit])!; jobIndex++; // After finding the best profit for the current worker, add it to maxProfit

from typing import List

```
class Solution:
   def max_profit_assignment(self, difficulty: List[int], profit: List[int], worker: List[int]) -> int:
        job_count = len(difficulty)
       # Pairing each job's difficulty with its profit and storing them as tuples
        jobs = [(difficulty[i], profit[i]) for i in range(job_count)]
       # Sorting jobs based on their difficulty
        jobs.sort(key=lambda x: x[0])
       # Sorting workers based on their capabilities
       worker.sort()
       max_profit = 0 # Tracks the maximum profit that can be achieved so far
        total_profit = 0 # Summing up the total profit for all workers
        job_index = 0 # Index to keep track of the current job
       # Iterating through each worker
       for capability in worker:
           # Updating the max_profit by comparing with each job's profit if the worker can handle the job
           while job_index < job_count and jobs[job_index][0] <= capability:</pre>
               max_profit = max(max_profit, jobs[job_index][1])
               job_index += 1
           # Accumulating the profit earned by this worker based on max_profit so far
           total_profit += max_profit
       # Returning the total profit that can be earned by all workers
       return total_profit
Time and Space Complexity
Time Complexity
```

Pairing the difficulty and profit and creating a job list: This runs in O(n) time, where n is the length of the difficulty list (assuming profit is of the same length).

Sorting the worker list: This also runs in $O(m \log m)$ time, where m is the number of workers. Iterating over the sorted worker list and updating total profit: In the worst case, each worker runs through the entire job list,

Sorting the job list: This is the most time-consuming step and it runs in O(n log n) time.

The given code consists of the following steps contributing to the time complexity:

- resulting in a potential 0(m * n) time complexity, but due to the sorting and one traversal mechanism, this is reduced to 0(m +
- n) since each job is looked at most once. The combined time complexity from these steps would be linearithmic in the larger of the two input sizes, hence the total time
- complexity is $O(\max(m, n) \log \max(m, n))$.

Space Complexity

- The space complexity is analyzed by looking at the extra space being used besides the input: Temporary variables i, t, and res use constant space 0(1).
 - The job list which pairs difficulty with profit: Since it creates a new list of tuples, it takes 0(n) additional space.
- No additional space is used that grows with the size of the input other than the job list. Hence, the space complexity is O(n).