2838. Maximum Coins Heroes Can Collect

Two Pointers Binary Search Prefix Sum

Problem Description

Array

Medium

In this battle scenario, there are n heroes and m monsters. Each hero and monster have their own power level, represented by two arrays heroes and monsters, respectively. The goal for each hero is to defeat monsters and collect coins. The number of coins that can be collected from defeating each monster is given in the array coins. The key points to note:

Sorting

- After a hero defeats a monster, they earn the number of coins associated with that monster. Heroes remain unharmed after battles, meaning their power levels don't decrease.
- Although multiple heroes can defeat the same monster, each monster yields coins to a given hero only once.

• A hero can defeat a monster if the hero's power is equal to or greater than the monster's power.

- Given this setup, we need to determine the maximum number of coins each hero can earn in total from the battle.
- Intuition

The intuitive approach to solving this problem involves maximizing the coins each hero can collect. To accomplish this, we want to pair each hero with monsters they are capable of defeating and ensure we do so in a way that maximizes the total coins

earned.

1. Sort the monsters by their power level while keeping track of their original indices so that we can match them with the corresponding coin values. 2. Calculate the cumulative sum of coins in the order of sorted monsters. This allows us to easily determine the total coins collected up to any point in the sorted order of monsters.

3. For each hero, find the rightmost monster in the sorted list that the hero can defeat. This step uses binary search to quickly find the position, as

4. The cumulative sum up to the position found in step 3 gives us the maximum coins a hero can earn, since all prior monsters are weaker and thus

- Here's how we approach the solution:
- defeatable by the hero. By applying this strategy to all heroes, we create an array of the maximum coins collected by each hero, reflecting the optimal assignment of heroes to monsters based on their power levels.
- **Solution Approach** The solution uses a few key Python features and algorithms to achieve the goal:

Sorting: First, we are sorting the monsters based on their power level using the sorted() function but with an additional twist. We sort the indices of the monsters array, not the values themselves. This allows us to maintain a correlation with the coins array.

Cumulative Sum: We use the Python itertools.accumulate() function to compute the cumulative sum of the coins

associated with each monster in increasing order of monster power. The initial=0 parameter ensures that we start with a

• The binary search is made possible because we have sorted monsters by power, which allows us to effectively search for the largest set of

Binary Search: We use the bisect_right() function from Python's bisect module to perform a binary search. This function is

Example Walkthrough

the sorted list allows for such a search.

used to find the rightmost index at which the hero's power value (h) would get inserted in the sorted monsters list to keep it in

• idx is the list of monster indices sorted by their power.

earnable by defeating all monsters up to that point.

Now let's step through the solution process.

• s is the cumulative sum of coins in the order of the sorted monster powers.

1. Sort monsters by their power level and maintain a correlation with their coin values.

2. Calculate the cumulative sum of coins based on sorted monsters' power.

zero value, representing that no coins are earned before defeating any monsters.

ascending order. Thus, this gives us the number of monsters that the current hero can defeat.

monsters that a hero can defeat. Putting it all together, the solution iterates over each hero's power and calculates the maximum coins they can collect using the precomputed cumulative sums and binary search:

• For each h (hero power) in heroes, the binary search finds the number of monsters that the hero can defeat, and s[i] gives the total coins

• The ans list is populated with the maximum coins for each hero and returned at the end. By following these steps, the function efficiently matches heroes with the optimal set of monsters, ensuring that each hero earns the maximum possible coins.

Let's use a small example to illustrate the solution approach with n heroes and m monsters. Suppose we have the following:

- heroes = [5, 10, 3]monsters = [4, 9, 5, 8]coins = [3, 5, 2, 7]
- Sorted monsters (by power): [4, 5, 8, 9] Corresponding indices: [0, 2, 3, 1] The indices array helps us to match monsters with the original `coins` array.

For a hero with power 10:

For a hero with power 3:

monsters.

Python

For heroes' powers: [5, 10, 3]

from itertools import accumulate

from bisect import bisect_right

from typing import List

class Solution:

Using original coins: [3, 5, 2, 7]

Corresponding to sorted powers: [3, 2, 7, 5]

Cumulative sum: [3, 5 (3+2), 12 (3+2+7), 17 (3+2+7+5)]

3. For each hero, perform a binary search to find the rightmost monster they can defeat. For a hero with power 5:

This walkthrough should now provide a clear example of how the described solution approach is applied to solve the problem.

Thus, by following each of these steps, we can determine the maximum number of coins that each hero can earn from defeating

Binary search will give index 1 (since the hero can defeat monsters with powers 4 and 5).

So the maximum coins this hero can earn are the cumulative sum at index 3, which is 17.

def maximumCoins(self, heroes: List[int], monsters: List[int], coins: List[int]) -> List[int]:

's' will contain the cumulative coins we get after defeating monsters in sorted order

bisect_right returns the index where to insert hero_strength to keep the list sorted

Append the cumulative coins up to that monster position for current hero's strength

monster_position = bisect_right(sorted_indices, hero_strength, key=lambda i: monsters[i])

The 'initial=0' argument ensures that there is a 0 at the beginning of the list

cumulative_coins = list(accumulate((coins[i] for i in sorted_indices), initial=0))

Initialize a list to store the maximum coins that can be collected by each hero

Binary search will give index 3 (since the hero can defeat all monsters).

4. Compile the results into an array representing the maximum coins each hero can collect.

Create sorted indices of the monsters based on their strength

sorted_indices = sorted(range(num_monsters), key=lambda i: monsters[i])

Calculate the cumulative sum of coins based on the sorted indices

collected_coins.append(cumulative_coins[monster_position])

public long[] maximumCoins(int[] heroes, int[] monsters, int[] coins) {

Integer[] sortedIndices = new Integer[monsterCount];

// Initialize sortedIndices with array indices

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

Return the list containing the maximum coins that each hero can collect

The hero cannot defeat any monster as the lowest monster power is 4.

Hence, the maximum coins this hero can earn are the cumulative sum at index 1, which is 5.

Thus, the cumulative sum for this hero is 0.

The maximum coins they can collect are: [5, 17, 0]

- Solution Implementation
- # Define the number of monsters num_monsters = len(monsters)

The weakest monsters come first in the list

Iterate over each hero for hero_strength in heroes: # Find the furthest right position in the sorted monster list that the hero can defeat

int monsterCount = monsters.length;

sortedIndices[i] = i;

collected_coins = []

return collected_coins

```
Java
```

class Solution {

```
// Sort the indices based on the monsters' strength
    Arrays.sort(sortedIndices, Comparator.comparingInt(j -> monsters[j]));
    // Create a prefix sum array for coins based on sorted indices of monsters
    long[] prefixSums = new long[monsterCount + 1];
    for (int i = 0; i < monsterCount; ++i) {</pre>
        prefixSums[i + 1] = prefixSums[i] + coins[sortedIndices[i]];
    int heroCount = heroes.length;
    long[] answer = new long[heroCount];
   // For each hero, find their maximum possible collectable coins
    for (int k = 0; k < heroCount; ++k) {</pre>
        // Find the number of monsters a hero can defeat
        int monsterDefeated = search(monsters, sortedIndices, heroes[k]);
        // Assign the sum of coins from the monsters a hero can defeat
        answer[k] = prefixSums[monsterDefeated];
    return answer;
// Binary search to find the number of monsters a hero can defeat
private int search(int[] nums, Integer[] indices, int heroStrength) {
    int left = 0, right = indices.length;
    while (left < right) {</pre>
        int mid = (left + right) >> 1;
       // Check if the mid-point monster is stronger than the hero
        if (nums[indices[mid]] > heroStrength) {
            right = mid; // Look in the left subarray
        } else {
            left = mid + 1; // Look in the right subarray
    // left now points to the number of monsters the hero can defeat
    return left;
```

vector<long long> maximumCoins(vector<int>& heroes, vector<int>& monsters, vector<int>& coins) {

// The number of monsters, used for setting up various bounds and loops

// Create a vector of indices corresponding to monster array positions

sort(monsterIndices.begin(), monsterIndices.end(), [&](int i, int j) {

prefixSum[i] = prefixSum[i - 1] + coins[monsterIndices[i - 1]];

// so we can later find out how many monsters a hero can defeat

// Sort the indices based on the monster strengths (from the monsters array),

// Prefix sum array to quickly calculate total coins up to a certain monster

// A lambda to search for the right-most position where a hero can defeat monsters

// Use the search function defined above to calculate the total coins each hero can collect

// Note: Although this code compiles and adheres to standard C++ syntax, without additional context it is unclear what this algor

// It seems to match heroes against a sorted list of monsters by their strength and calculate the maximum coins each hero can col

auto search = [&](int strength) { int left = 0, right = monsterCount; while (left < right) {</pre> int mid = (left + right) >> 1; if (monsters[monsterIndices[mid]] > strength) { } else {

};

};

TypeScript

C++

#include <vector>

#include <numeric>

#include <algorithm>

using namespace std;

int monsterCount = monsters.size();

vector<int> monsterIndices(monsterCount);

return monsters[i] < monsters[j];</pre>

long long prefixSum[monsterCount + 1];

right = mid;

for (int heroStrength : heroes) {

left = mid + 1;

for (int i = 1; i <= monsterCount; ++i) {</pre>

iota(monsterIndices.begin(), monsterIndices.end(), 0);

// The answer vector to store maximum coins for each hero

answer.push_back(prefixSum[search(heroStrength)]);

class Solution {

});

prefixSum[0] = 0;

vector<long long> answer;

return left;

return answer;

let low = 0;

while (low < high) {</pre>

} else {

return low;

from itertools import accumulate

from bisect import bisect_right

from typing import List

};

class Solution:

let high = numberOfMonsters;

high = mid;

Define the number of monsters

The weakest monsters come first in the list

num_monsters = len(monsters)

low = mid + 1;

// Look for the rightmost monster that hero can defeat

if (monsters[indices[mid]] > strength) {

public:

function maximumCoins(heroes: number[], monsters: number[], coins: number[]): number[] { // Length of the monsters array const numberOfMonsters = monsters.length; // Create an index array from 0 to numberOfMonsters-1 const indices: number[] = Array.from({ length: numberOfMonsters }, (_, i) => i); // Sort the indices array based on the corresponding value in the monsters array indices.sort((a, b) => monsters[a] - monsters[b]); // 'prefixSum' represents cumulative coins amount from monsters sorted on strength const prefixSum: number[] = Array(numberOfMonsters + 1).fill(0); for (let i = 0; i < numberOfMonsters; ++i) {</pre> prefixSum[i + 1] = prefixSum[i] + coins[indices[i]]; // Binary search helper method that finds how many monsters a hero can defeat const searchMonsters = (strength: number): number => {

const mid = (low + high) >> 1; // Equivalent to Math.floor((low + high) / 2)

// Map heroes to their maximum coins earnings based on which monsters they can defeat

def maximumCoins(self, heroes: List[int], monsters: List[int], coins: List[int]) -> List[int]:

's' will contain the cumulative coins we get after defeating monsters in sorted order

Find the furthest right position in the sorted monster list that the hero can defeat

The 'initial=0' argument ensures that there is a 0 at the beginning of the list

cumulative_coins = list(accumulate((coins[i] for i in sorted_indices), initial=0))

Initialize a list to store the maximum coins that can be collected by each hero

return heroes.map(heroStrength => prefixSum[searchMonsters(heroStrength)]);

Create sorted indices of the monsters based on their strength

sorted_indices = sorted(range(num_monsters), key=lambda i: monsters[i])

Calculate the cumulative sum of coins based on the sorted indices

bisect_right returns the index where to insert hero_strength to keep the list sorted monster_position = bisect_right(sorted_indices, hero_strength, key=lambda i: monsters[i]) # Append the cumulative coins up to that monster position for current hero's strength collected_coins.append(cumulative_coins[monster_position]) # Return the list containing the maximum coins that each hero can collect return collected_coins

Time and Space Complexity

collected_coins = []

Iterate over each hero

for hero_strength in heroes:

• Creating the s list with accumulated coins: The accumulate function runs in O(m) since it processes each element once. • The for loop to fill ans list: For each hero in heroes, a binary search is performed using bisect_right, which takes 0(log m). Let n be the length of the heroes list, so the loop runs in O(n log m) time.

• The s list also takes O(m) space.

Time Complexity

dominated by the m log m term. **Space Complexity**

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

• Sorting the index list idx: This takes 0(m log m) time, where m is the length of the monsters list.

The space complexity can be analyzed as follows: • The idx list takes O(m) space.

• Combining these, the total time complexity is 0(m log m + m + n log m) which simplifies to 0(m log m + n log m) because the m term is

 Temporary variables used inside the for loop take constant space. • Thus, the total space complexity is 0(m + n).

• The ans list takes O(n) space, where n is the length of the heroes list.