2070. Most Beautiful Item for Each Query

Medium | Array | Binary Search | Sorting | Leetcode Link

Problem Description

In this problem, we have a list of `items`, each represented by a pair `[price, beauty]`. Our goal is to answer a series of queries, each asking for the maximum beauty value among all items whose price is less than or equal to the query value.

If no items fit the criteria for a given query (all items are more expensive than the query value), the answer for that query is `0`.

We are asked to return a list of the maximum beauty values corresponding to each query.

Intuition

To solve the problem efficiently, we first notice that we can handle the queries independently. So, we want a quick way to find the maximum beauty for any given price limit.

We approach this by sorting the `items` by price. Sorting the items allows us to employ a binary search technique to efficiently find the item with the highest beauty below a certain price threshold.

After sorting `items`, we create two lists: `prices`, which holds the sorted prices, and `mx`, which holds the running maximum beauty observed as we iterate through the sorted items. This ensures that for each price `prices[i]`, `mx[i]` is the maximum beauty of all items with a price less than or equal to `prices[i]`.

The binary search is carried out by using the `bisect_right` function from Python's `bisect` module. For each query, `bisect_right` finds the index `j` in the sorted `prices` list such that all prices to the left of `j` are less than or equal to the query value.

If such an index `j` is found and is greater than `0`, it means there exists at least one item with a price lower than or equal to the query value. We use `j - 1` as the index to get the maximum beauty value from the `mx` list.

Otherwise, if no index `j` is returned because all items are too expensive, we default the answer for that query to `0`.

This algorithm allows us to answer each query in logarithmic time with respect to the number of items, which is desirable when dealing with a large number of items or queries.

Solution Approach

The solution uses a mix of sorting, dynamic programming, and binary search to efficiently answer the maximum beauty queries for given price limits.

Here's the step-by-step implementation strategy:

1. **Sorting Items:** Start by sorting the `items` based on their price. This is vital because it allows us to leverage binary search later on. Sorting is done using Python's default sorting algorithm, Timsort, which has a time complexity of O(n log n).
2. **Extracting Prices and Initializing Maximum Beauty List (`mx`):**
   - Extract the sorted prices into a list called `prices`.
   - Initialize a list `mx`, which keeps track of the maximum beauty encountered so far as we iterate through the items. The first element of `mx` is simply the beauty of the first item in the sorted list.
3. **Building a Running Maximum Beauty:**
   - Iterate through each item, starting from the second one (since the first element's max beauty is already recorded).
   - For each item, update the `mx` list with the greater value between the current item's beauty and the last recorded max beauty in `mx`. This is a form of dynamic programming, where the result of each step is based on the previous step's result.
4. **Answering Queries with Binary Search:**
   - Initialize an answer list `ans` of the same size as `queries`, defaulting all elements to `0`.
   - For each query, use the `bisect_right` function from the `bisect` module to perform a binary search on `prices` to find the point where the query value would be inserted while maintaining the list's order.
     - `bisect_right` returns an index `j` that is one position past where the query value would be inserted, so a price less than or equal to the query must be at an index before `j`.
   - If `j` is not `0`, it means an item with a suitable price exists, and the answer for this query is `mx[j - 1]` - the corresponding max beauty by that price. If `j` is `0`, it means no items are cheaper than the query, and the answer remains `0`.
5. **Return the Answer List:**
   - After all queries have been processed, return the answer list `ans` filled with the maximum beauties for each respective query.

This approach effectively decouples the item price-beauty relationship from the queries, by pre-computing a list of maximum beauties (`mx`) that can later be quickly referenced using binary search. This transforms what could be an O(n*m) problem (naively checking n items for each of m queries) into an O(n log n + m log n) problem, where n is the number of items and m the number of queries.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have the following `items` and `queries`:

- items = [[3, 2], [5, 4], [3, 1], [10, 7]]
- queries = [2, 4, 6]

Following the steps:

1. **Sorting Items:**
   - We sort `items` by price: sorted_items = [[3, 2], [3, 1], [5, 4], [10, 7]]
2. **Extracting Prices and Initializing Maximum Beauty List (`mx`):**
   - Extract prices: prices = [3, 3, 5, 10]
   - Initialize `mx` with the maximum beauty of the first item: mx = [2]
3. **Building a Running Maximum Beauty:**
   - Process the second item: it has the same price but lower beauty, so `mx` remains the same: mx = [2]
   - Process the third item: new price with higher beauty, update `mx`: mx = [2, 4]
   - Process the fourth item: new price with higher beauty, update `mx`: mx = [2, 4, 7]
4. **Answering Queries with Binary Search:**
   - Initialize an answer list `ans` with all zeros: ans = [0, 0, 0]
   - Query 1: 2 is less than all prices, therefore `ans[0]` remains `0`.
   - Query 2: 4 is equal to the second price, `bisect_right` would place it after index 1, so we use `mx[0]`: ans = [0, 2, 0]
   - Query 3: 6 would fit between indexes 2 and 3, `bisect_right` returns 3 so we use `mx[2]`: ans = [0, 2, 4]
5. **Return the Answer List:**
   - The final answer list reflecting maximum beauties for each query is: ans = [0, 2, 4]

Thus, for the `queries` [2, 4, 6], the maximum beauty values for items within these price limits are [0, 2, 4], respectively.

Python Solution

```python
from bisect import bisect_right

class Solution:
    def maximumBeauty(self, items: List[List[int]], queries: List[int]) -> List[int]:
        # Sort the items by price first (since the first item of each sub-list is price)
        items.sort()
        # Extract a list of prices for binary search
        prices = [price for price, _ in items]

        # Create a list to store the maximum beauty encountered so far
        max_beauty = [items[0][1]]  # initialize with the first item's beauty
        for index in range(1, len(items)):
            # Update the max_beauty list with the maximum beauty seen up to current index
            max_beauty.append(max(max_beauty[-1], items[index][1]))

        # Initialize the answer list for the queries with zeroes
        answers = [0] * len(queries)

        # Process each query to find the maximum beauty for that price
        for i, query in enumerate(queries):
            # Find the rightmost item that is not greater than the query price
            index = bisect_right(prices, query)
            # If we found an item, store the corresponding max beauty (if not, zero stays by default)
            if index:
                answers[i] = max_beauty[index - 1]

        # Return the list of answers to the queries
        return answers
```

Java Solution

```java
class Solution {
    public int[] maximumBeauty(int[][] items, int[] queries) {
        // Sort the items array based on the price in increasing order
        Arrays.sort(items, (item1, item2) -> item1[0] - item2[0]);

        // Update the beauty value in the sorted items array to ensure that each
        // item has the maximum beauty value at or below its price.
        for (int i = 1; i < items.length; ++i) {
            // The current maximum beauty is either the beauty of the current item
            // or the maximum beauty of all previous items.
            items[i][1] = Math.max(items[i - 1][1], items[i][1]);
        }

        // The number of queries to process
        int queryCount = queries.length;
        // Array to store the answer for each query
        int[] answers = new int[queryCount];

        // Process each query to find the maximum beauty for the specified price
        for (int i = 0; i < queryCount; ++i) {
            // Use binary search to find the rightmost item with a price not
            // exceeding the query (price we can spend).
            int left = 0, right = items.length;
            while (left < right) {
                int mid = (left + right) >> 1; // equivalent to (left + right) / 2
                if (items[mid][0] > queries[i]) {
                    // If the mid item's price exceeds the query price, move the right pointer
                    right = mid;
                } else {
                    // Otherwise, move the left pointer to continue searching to the right
                    left = mid + 1;
                }
            }

            // If there's at least one item that costs less than or equal to the query price
            if (left > 0) {
                // The answer is the maximum beauty found among all the affordable items
                answers[i] = items[left - 1][1];
            }
            // If no such item is found, the default answer of 0 (for beauty) will remain
        }

        // Return the array of answers for all the queries
        return answers;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function that returns the maximum beauty item that does not exceed the query price
    vector<int> maximumBeauty(vector<vector<int>>& items, vector<int>& queries) {
        // Sort items based on their price in ascending order
        sort(items.begin(), items.end());

        // Preprocess items to keep track of the maximum beauty so far at each price point
        for (int i = 1; i < items.size(); ++i) {
            items[i][1] = max(items[i - 1][1], items[i][1]);
        }

        int numOfQueries = queries.size();
        vector<int> answers(numOfQueries);

        // Iterate over each query to find the maximum beauty that can be obtained
        for (int i = 0; i < numOfQueries; ++i) {
            int left = 0, right = items.size();

            // Perform a binary search to find the rightmost item with price less than or equal to the query price
            while (left < right) {
                int mid = (left + right) / 2;
                if (items[mid][0] > queries[i])
                    right = mid;       // Item is too expensive, reduce the search range
                else
                    left = mid + 1;    // Item is affordable, potentially look for more expensive items
            }

            // If search ended with left pointing to an item, take the beauty value of the item to the left of it
            // because the binary search gives us the first item with a price higher than the query
            if (left > 0) answers[i] = items[left - 1][1];
            // If left is 0, then all items are too expensive, thus the answer for this query is 0 by default
        }

        return answers;
    }
};
```

Typescript Solution

```typescript
// Import necessary functions from 'lodash' for sorting and binary search
import _ from 'lodash';

// Function that returns the maximum beauty item that does not exceed the query price
function maximumBeauty(items: number[][], queries: number[]): number[] {
    // Sort items based on their price in ascending order
    items.sort((a, b) => a[0] - b[0]);

    // Preprocess items to keep track of the maximum beauty so far at each price point
    for (let i = 1; i < items.length; ++i) {
        items[i][1] = Math.max(items[i - 1][1], items[i][1]);
    }

    let numOfQueries = queries.length;
    let answers = new Array(numOfQueries).fill(0);

    // Iterate over each query to find the maximum beauty that can be obtained
    for (let i = 0; i < numOfQueries; ++i) {
        let left = 0, right = items.length;

        // Perform a binary search to find the rightmost item with price less than or equal to the query price
        while (left < right) {
            let mid = Math.floor((left + right) / 2);
            if (items[mid][0] > queries[i])
                right = mid;       // Item is too expensive, reduce the search range
            else
                left = mid + 1;    // Item is affordable, potentially look for more expensive items
        }

        // If search ended with left pointing to an item, take the beauty value of the item to the left of it
        // because the binary search gives us the first item with a price higher than the query
        if (left > 0) {
            answers[i] = items[left - 1][1];
        }
        // If left is 0, then all items are too expensive, thus the answer for this query is 0 by default
    }

    return answers;
}
```

Time and Space Complexity

Time Complexity

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

1. **Sorting the `items` list**: The `items.sort()` method is called on the list of items, which typically has a time complexity of `O(n * log(n))`, where `n` is the number of items.

2. **Creating the `prices` list**: This involves iterating over the sorted `items` list to build a new list of prices, which will take `O(n)`.

3. **Creating the `mx` list**: A single for-loop is used to construct the `mx` list. This also runs in `O(n)` time as it iterates over `n` items once.

4. **Answering the queries by binary search**: Each query performs a binary search to find the right index in the `prices` list, which takes `O(log(n))`. Since this is done for `q` queries, the total time complexity for this step is `O(q * log(n))`.

Combining these steps, the overall time complexity is `O(n * log(n) + n + n + q * log(n))`, which simplifies to `O(n * log(n) + q * log(n))` because the linear terms are overshadowed by the `n * log(n)` term when `n` is large.

Space Complexity

The space complexity of the code can be analyzed by considering the additional data structures used:

1. **The `prices` list**: This consumes `O(n)` space.

2. **The `mx` list**: This also consumes `O(n)` space.

3. **The `ans` list**: Space needed is `O(q)` for storing the answers for `q` queries.

Therefore, the overall space complexity is `O(n + n + q)` which simplifies to `O(n + q)` as we add the space required for the two lists related to items and the space for the answers to the queries.