1580. Put Boxes Into the Warehouse II

Medium Greedy Array Sorting Leetcode Link

Problem Description

You've got several boxes and a warehouse with a series of rooms in it. Each box has a height, and each room in the warehouse has a height as well. Your job is to figure out the maximum number of these boxes that can fit into the warehouse based on the following set of rules:

- You can't stack the boxes on top of each other; each must sit on the warehouse floor.
- You are allowed to reorder the boxes before you start placing them in the warehouse.
- Boxes can be pushed into the warehouse from either end (left or right side).
- If a box is too tall for a particular room in the warehouse, it, and any boxes behind it, can't proceed further. They are effectively stopped at that point.

The challenge is to maximize the number of boxes that can fit into the warehouse given these constraints.

Intuition

To solve this problem, take a look at both the boxes and the warehouse room heights. Since the boxes can be reorganized, sort them from smallest to tallest, which allows you to begin with filling from the smallest box and move up. For the warehouse rooms, consider that a box can come from either side. You should find the effective height for each room in the warehouse, which is the maximum height that a box can be to pass through that room from either side.

To do this, create two arrays, `left` and `right`:

- `left[i]` gives the minimum height of the rooms on the left of room `i`, including `i` itself.
- `right[i]` gives the minimum height of the rooms on the right of room `i`, including `i` itself.

After knowing the minimum of the left and right for each room, calculate the effective height for each room by taking the maximum of the minimums. Then sort the warehouse room heights to likewise place smaller effective heights first.

Finally, place boxes in the warehouse by checking:

- Start placing the smallest box.
- Increment the number of boxes placed (`ans`) each time a box fits the room height.
- If you encounter a warehouse room that is shorter than the current box, skip to the next room until you find a room tall enough or run out of rooms.
- Keep going until all boxes are checked or there is no more room with sufficient height for the remaining boxes.

This approach will give you the maximum number of boxes that can fit into the warehouse.

Solution Approach

The implementation is quite straightforward if you follow the intuition behind the solution. Let's break down the steps of the algorithm used, referring to the Python code snippet provided:

1. **Initialization**: Two arrays `left` and `right` are initialized to record the minimum height so far from the left and the right respectively. These arrays help us determine the effective height of a warehouse room. Additionally, the variable `n` stores the length of the warehouse.
2. **Populate the `left` and `right` arrays:**
   - For `left`, start from the second room (`index 1`) and move to the last, for each room storing the minimum height of all rooms encountered before, including the current one.
   - For `right`, start from the second to last room (`index n - 2`) and move towards the first room (`index 0`), with the same principle as the `left`.
   - In this context, `inf` is used to initialize `left[0]` and `right[-1]` so these values don't restrict any room height.
3. **Calculate the effective heights**: Iterate through the `warehouse` array and calculate the effective height for each room. This is the minimum height that a box can be to pass through that room from either direction.
4. **Sort both boxes and warehouse**: Boxes are sorted in non-decreasing order, so you always attempt to place the smallest boxes first, which increases the likelihood of accommodating more boxes. The warehouse is sorted because we've converted the warehouse into a set of potential box heights, from smallest to largest effective height.
5. **Place boxes into the warehouse**: The algorithm then iterates through the `boxes` array and tries to place each box into the sorted `warehouse`:
   - Use a pointer `i` which starts at `0`, representing the effective height to insert into the warehouse.
   - If the current box is taller than the `warehouse[i]`, increment `i` to find an appropriate height.
   - If a suitable room is found, increase the number of placed boxes `ans` by 1 and move to the next box.
6. **Return the result**: Once all possible boxes are placed, or there's no more room in the warehouse, the loop ends, and `ans` gives the number of boxes placed. Return `ans`.

By undertaking these steps, the algorithm ensures that we are optimizing the number of boxes placed in the warehouse by strategically choosing dimensions and placements.

Example Walkthrough

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

Suppose we have the following boxes and warehouse room heights:

Boxes: [3, 4, 1, 2]
Warehouse rooms: [5, 3, 4, 1]

First, we sort the array of boxes in non-decreasing order: `[1, 2, 3, 4]`.

For the warehouse rooms, we need to find the effective height for each room. First, we initialize two arrays, `left` and `right`, and then populate them:

- **Initialize** `left` and `right`: `left = [inf, -1, -1, -1]`, `right = [-1, -1, -1, inf]`
- **Populating `left`**: We compare each room's height to the minimum of all previous rooms, starting from the left. So `left` becomes `[5, 3, 3, 1]`.
- **Populating `right`**: Similarly, we compare each room's height to the minimum of all previous rooms, starting from the right. So `right` becomes `[1, 1, 3, 5]`.

Now, we calculate the effective height for each room by determining the maximum height that a box can pass through from either end. We take the maximum of the minimums from `left` and `right` for each room:

Effective heights: `[5, 3, 3, 5]`
(Note that we take the maximum between `left[i]` and `right[i]` for each room.)

We then sort the effective heights for the rooms to facilitate the insertion of boxes: `[3, 3, 5, 5]`.

Now we are ready to place the boxes into the effective heights of the warehouse:

- Start with the smallest box (`1`). It fits in the first room which has an effective height of `3`.
- Move to the second box (`2`). It also fits in the next room with an effective height of `3`.
- Then the third box (`3`) fits in the next room with an effective height of `5`.
- Lastly, the fourth box (`4`) fits in the final room with an effective height of `5`.

Since all boxes can be placed into the warehouse, our answer (`ans`) is `4`.

Thus, using this approach, we have placed all four boxes into the warehouse effectively.

Python Solution

```python
class Solution:
    def max_boxes_in_warehouse(self, boxes: List[int], warehouse: List[int]) -> int:
        # Find the length of the warehouse
        warehouse_length = len(warehouse)

        # Initialize the left and right limits, which are used to track
        # the minimum height of the left and right side at each position
        left_min = [0] * warehouse_length
        right_min = [0] * warehouse_length

        # Set the first element of left limits and the last element of right limits
        # to infinity as the starting point
        left_min[0] = right_min[-1] = float('inf')

        # Calculate the left_min for each position in the warehouse
        for i in range(1, warehouse_length):
            left_min[i] = min(left_min[i - 1], warehouse[i - 1])

        # Calculate the right_min for each position in the warehouse
        for i in range(warehouse_length - 2, -1, -1):
            right_min[i] = min(right_min[i + 1], warehouse[i + 1])

        # Update each position in the warehouse to be the minimum height restriction considering
        # the limitations from both sides
        for i in range(warehouse_length):
            warehouse[i] = min(warehouse[i], max(left_min[i], right_min[i]))

        # Sort the boxes and warehouse to prepare for the greedy approach
        boxes.sort()
        warehouse.sort()

        # Initialize the answer and pointer i for the warehouse array
        box_count = i = 0

        # Iterate over each box to see if it fits in the warehouse
        for box in boxes:
            # Find a space in the warehouse where the current box can fit
            while i < warehouse_length and warehouse[i] < box:
                i += 1
            if i == warehouse_length:
                # If we reached the end of the warehouse, no more boxes can fit
                break
            # If a box is placed in the warehouse, increase the count
            box_count += 1

            # Move to the next position for the following box
            i += 1

        # Return the total number of boxes that can be placed in the warehouse
        return box_count

```

Java Solution

```java
import java.util.Arrays;

class Solution {
    // Maximum number of boxes that can be placed in the warehouse
    public int maxBoxesInWarehouse(int[] boxes, int[] warehouse) {
        int warehouseSize = warehouse.length;
        int[] minLeftHeight = new int[warehouseSize]; // Minimum height to the left
        int[] minRightHeight = new int[warehouseSize]; // Minimum height to the right
        final int infinity = Integer.MAX_VALUE;
        minLeftHeight[0] = infinity;
        minRightHeight[warehouseSize - 1] = infinity;

        // Populate the minimum height from left to right
        for (int i = 1; i < warehouseSize; ++i) {
            minLeftHeight[i] = Math.min(minLeftHeight[i - 1], warehouse[i - 1]);
        }

        // Populate the minimum height from right to left
        for (int i = warehouseSize - 2; i >= 0; --i) {
            minRightHeight[i] = Math.min(minRightHeight[i + 1], warehouse[i + 1]);
        }

        // Update warehouse heights to the minimum height at each position
        for (int i = 0; i < warehouseSize; ++i) {
            warehouse[i] = Math.min(warehouse[i], Math.max(minLeftHeight[i], minRightHeight[i]));
        }

        // Sort both the box sizes and the warehouse aisle heights
        Arrays.sort(boxes);
        Arrays.sort(warehouse);

        int maxBoxes = 0; // Maximum boxes that can be placed
        int warehouseIndex = 0; // Pointer to navigate through warehouse heights

        // Loop through each box to find its place in the warehouse
        for (int box : boxes) {
            // Find a position in the warehouse where the box can fit
            while (warehouseIndex < warehouseSize && warehouse[warehouseIndex] < box) {
                warehouseIndex++;
            }
            // If there are no more positions left, break out of the loop
            if (warehouseIndex == warehouseSize) {
                break;
            }
            // Place the box in the warehouse and move to the next position
            maxBoxes++;
            warehouseIndex++;
        }
        return maxBoxes; // Return the maximum number of boxes that can be placed
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maxBoxesInWarehouse(vector<int>& boxes, vector<int>& warehouse) {
        // Get the size of the warehouse
        int warehouseSize = warehouse.size();
        // Define an infinity value used for comparisons
        const int INF = 1 << 30;

        // These vectors will hold the minimum height limitation when looking from the left and right
        vector<int> minLeft(warehouseSize, INF);
        vector<int> minRight(warehouseSize, INF);

        // Fill the minLeft vector with the minimum height limitation so far from the left
        for (int i = 1; i < warehouseSize; ++i) {
            minLeft[i] = min(minLeft[i - 1], warehouse[i - 1]);
        }

        // Fill the minRight vector with the minimum height limitation so far from the right
        for (int i = warehouseSize - 2; i >= 0; --i) {
            minRight[i] = min(minRight[i + 1], warehouse[i + 1]);
        }

        // Update the warehouse's limitations with the stricter of the minLeft and minRight at each position
        for (int i = 0; i < warehouseSize; ++i) {
            warehouse[i] = min(warehouse[i], max(minLeft[i], minRight[i]));
        }

        // Sort the array of boxes and the modified warehouse array
        sort(boxes.begin(), boxes.end());
        sort(warehouse.begin(), warehouse.end());

        // Initialize the count of boxes that can be put into the warehouse
        int boxCount = 0;
        // Index for the warehouse's position
        int warehouseIndex = 0;

        // Iterate through each box
        for (int box : boxes) {
            // Increment the warehouseIndex until a position is found where the box fits
            while (warehouseIndex < warehouseSize && warehouse[warehouseIndex] < box) {
                warehouseIndex++;
            }
            // If we've reached the end of the warehouse, we can't fit any more boxes
            if (warehouseIndex == warehouseSize) {
                break;
            }
            // Successfully placed the box, increment count and move to next warehouse position
            boxCount++;
            warehouseIndex++;
        }
        // Return the total count of boxes that can be put into the warehouse
        return boxCount;
    }
};

```

Typescript Solution

```typescript
function maxBoxesInWarehouse(boxes: number[], warehouse: number[]): number {
    // Get the size of the warehouse
    let warehouseSize: number = warehouse.length;

    // Define an infinity value used for comparisons
    const INF: number = 1 << 30;

    // These arrays will hold the minimum height limitation when looking from the left and right
    let minLeft: number[] = new Array(warehouseSize).fill(INF);
    let minRight: number[] = new Array(warehouseSize).fill(INF);

    // Fill the minLeft with the minimum height limitation so far from the left
    for (let i = 1; i < warehouseSize; ++i) {
        minLeft[i] = Math.min(minLeft[i - 1], warehouse[i - 1]);
    }

    // Fill the minRight with the minimum height limitation so far from the right
    for (let i = warehouseSize - 2; i >= 0; --i) {
        minRight[i] = Math.min(minRight[i + 1], warehouse[i + 1]);
    }

    // Update the warehouse's limitations with the stricter of the minLeft and minRight at each position
    for (let i = 0; i < warehouseSize; ++i) {
        warehouse[i] = Math.min(warehouse[i], Math.max(minLeft[i], minRight[i]));
    }

    // Sort the array of boxes and the modified warehouse array
    boxes.sort((a, b) => a - b);
    warehouse.sort((a, b) => a - b);

    // Initialize the count of boxes that can be put into the warehouse
    let boxCount: number = 0;
    // Index for the warehouse's position
    let warehouseIndex: number = 0;

    // Iterate through each box
    for (const box of boxes) {
        // Increment the warehouseIndex until a position is found where the box fits
        while (warehouseIndex < warehouseSize && warehouse[warehouseIndex] < box) {
            warehouseIndex++;
        }
        // If we've reached the end of the warehouse, we can't fit any more boxes
        if (warehouseIndex === warehouseSize) {
            break;
        }
        // Successfully placed the box, increment count, and move to the next warehouse position
        boxCount++;
        warehouseIndex++;
    }

    // Return the total count of boxes that can be put into the warehouse
    return boxCount;
}

```

Time and Space Complexity

Time Complexity

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

1. Initializing the `left` and `right` arrays with `0` and `inf` respectively, and the subsequent loop for populating `left` takes $O(n)$ time, where n is the number of positions in the warehouse.
2. The second loop for populating `right` also takes $O(n)$ time.
3. The third loop where the `warehouse` heights are updated by taking the minimum of warehouse height, left, and right at each position requires $O(n)$ time.
4. Sorting the `boxes` array is $O(m \log m)$, where m is the number of boxes.
5. Sorting the modified `warehouse` array takes $O(n \log n)$ time.
6. The final loop, which matches boxes to positions in the warehouse, will in the worst case iterate through all the boxes and all positions, which takes $O(m + n)$ time.

Adding up all these parts, the time complexity is $O(n) + O(n) + O(n) + O(m \log m) + O(n \log n) + O(m + n)$.

Since sorting operations have the highest complexity, the overall time complexity is dominated by them: $O(m \log m) + O(n \log n)$. In the case where n and m are similar in magnitude, this can be simplified to $O(n \log n)$.

Space Complexity

The space complexity is determined by the extra space used in the algorithm apart from the input, which are:

1. The space used by the `left` array, which is $O(n)$.
2. The space used by the `right` array, also $O(n)$.
3. There is constant space used for variables like `ans` and `i`, which we represent as $O(1)$.

So, the total extra space used by the algorithm is $O(n) + O(n)$ which simplifies to $O(n)$.

In conclusion, the time complexity of the code is $O(m \log m) + O(n \log n)$ and the space complexity is $O(n)$.