1564. Put Boxes Into the Warehouse I

Medium Greedy Array Sorting

Leetcode Link

Problem Description

In this problem, we are given two arrays: 'boxes', which holds the heights of boxes, and 'warehouse', which holds the heights of the rooms in a warehouse. The boxes can only be pushed into the warehouse from left to right, and boxes cannot be stacked on top of one another. The goal is to find the maximum number of boxes that can fit into the warehouse according to the given rules.

One important rule to note is that if a box is too tall to fit into a room, it blocks all the boxes behind it from entering the warehouse as well. However, we can rearrange the boxes before we start placing them into the warehouse. Given these constraints, we need to find the best order of boxes and the strategy to place as many boxes as possible into the warehouse rooms.

Intuition

To solve this problem, we can start with a greedy approach. A greedy approach means that we make the locally optimal choice at each step with the hope that these choices lead us to a globally optimal solution.

Since we can rearrange the boxes, it will be beneficial to try to fit smaller boxes first as they have a higher chance of fitting into the rooms of the warehouse, leaving the larger boxes for the bigger rooms towards the end. We start by sorting the 'boxes' array to organize the boxes from smallest to largest.

Furthermore, as we can only push boxes from left to right, we need to pre-process the 'warehouse' array to accommodate the fact that if we encounter a smaller room, it will affect all subsequent rooms. To do that, we can create a new array, 'left', which represents the maximum height of a box that can be placed in the subsequent rooms after considering previous smaller rooms. We initialize the 'left' array with the first room's height and then, for each subsequent room, we take the minimum height between the current room and the previous room. This way, we account for the potential blocking effect caused by smaller rooms.

Finally, we iterate over both the sorted 'boxes' array and the pre-processed 'left' array from the end towards the front. We put a box into the warehouse if the box's height is lower than or equal to the current room's height. If the box is too tall, we move to the next room. We continue this process until we have placed all the possible boxes or we have checked all the rooms.

The variable 'i' will keep track of how many boxes have been placed, and when we either run out of boxes or rooms, the value of 'i' will give us the number of boxes that can be put into the warehouse.

Solution Approach

The solution approach for this problem uses a greedy method, sorted arrays, and pre-processing of the warehouse data. The implementation in Python can be broken down into several clear steps:

1. **Pre-Process the Warehouse Heights**: The warehouse array is traversed to create a 'left' array. This array stores the maximum height of a box that can be inserted into the warehouse up to that point, where the value for each room is the minimum between the current room and the previous one:

```
1 left = [warehouse[0]] * n
2 for i in range(1, n):
3     left[i] = min(left[i - 1], warehouse[i])
```

This step ensures that any height restrictions imposed by a smaller room are carried over to all subsequent rooms.

2. **Sort the Boxes**: The boxes array is sorted in increasing order of their heights. This allows us to try fitting the smallest boxes first:

```
1 boxes.sort()
```

3. **Place Boxes into the Warehouse**: Two pointers, `i` and `j` are used to iterate through the 'boxes' and 'left' arrays, respectively. The `i` pointer starts at `0` to point to the smallest box, whereas `j` starts at `n - 1` to point to the last room of the warehouse. The algorithm proceeds to check if each box can fit into the current room of the warehouse. If a box fits (box height ≤ room height), both pointers are moved one step (to the next box and the next room), and the process is repeated. If a box does not fit, the pointer `j` is decremented to move to the next room that could potentially accommodate the current box:

```
1 i, j = 0, n - 1
2 while i < len(boxes):
3     while j >= 0 and left[j] < boxes[i]:
4         j -= 1
5     if j < 0:
6         break
7     i, j = i + 1, j - 1
```

The loop breaks when either we run out of boxes (`i` is equal to the length of boxes array) or when there are no more rooms to check (`j` is less than `0`).

4. **Return the Result**: At the end of the iteration process, `i` represents the number of boxes that have been placed in the warehouse. Since `i` starts at `0`, it also conveniently matches the count of boxes placed:

```
1 return i
```

In essence, the algorithm sorts the boxes so that we attempt to place the smallest ones first, and then iteratively checks each box against the limitations of the warehouse. Using two pointers facilitates an efficient traversal over both arrays without the need to check all possible combinations, focusing only on feasible fits. The approach ensures that we maximize the number of boxes placed in the warehouse.

Example Walkthrough

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

Imagine we have a `boxes` array with heights `[3, 8, 1]` and a `warehouse` array with heights `[5, 4, 3, 2, 1]`.

Step 1: Pre-Process the Warehouse Heights

First, we'll create the `left` array from the `warehouse` heights to make sure we account for the height constraints.

`left = [5]` (initialize with the first room's height)

Then we process the other rooms and get:

`left = [5, 4, 3, 2, 1]` (each entry is the minimum of the current room height and the previous entry in the `left` array)

Step 2: Sort the Boxes

We sort the `boxes` array to prioritize fitting smaller boxes first:

`boxes = [1, 3, 8]` (sorted in increasing order)

Step 3: Place Boxes into the Warehouse

Iterate with two pointers, `i` points to boxes, starting with the smallest, and `j` points to the `left` array, starting with the last position.

`i = 0, j = 4`

The smallest box (1) can fit in the last room (1), so we move to the next box and the next room.

`i = 1, j = 3`

The next box (3) can fit into the fourth room (2), but only just, so we skip this room and move to the one before it.

`i = 1, j = 2`

Now, the same box (3) fits into the third room (3), so we move to the next box and next room.

`i = 2, j = 1`

Our final box (8) cannot fit in the second room (4), nor in the first room (5), so we cannot place this box.

`i = 2, j = -1`

Step 4: Return the Result

We were able to place 2 boxes into the warehouse. Since `i` now equals `2`, we return this value as the result.

The output for the example is `2`, indicating the maximum number of boxes we can place in the warehouse with the given constraints.

Python Solution

```
1 class Solution:
2     def maxBoxesInWarehouse(self, boxes, warehouse):
3         # Calculate the number of slots in the warehouse
4         num_slots = len(warehouse)
5
6         # Initialize the left_min array with the first element of the warehouse
7         # left_min[i] will store the minimum height encountered so far from the left
8         left_min = [warehouse[0]] + [0] * (num_slots - 1)
9         for i in range(1, num_slots):
10             # Update the left_min array with the minimum value seen so far
11             left_min[i] = min(left_min[i - 1], warehouse[i])
12
13         # Sort the boxes by height in ascending order
14         boxes.sort()
15
16         # Initialize pointers, i for boxes and j for slots in the warehouse
17         box_index, slot_index = 0, num_slots - 1
18
19         # Iterate through the sorted boxes
20         while box_index < len(boxes):
21             # Decrease the slot index until we find a slot that can accommodate the current box
22             while slot_index >= 0 and left_min[slot_index] < boxes[box_index]:
23                 slot_index -= 1
24             # If no slots are left, break the loop
25             if slot_index < 0:
26                 break
27             # Once a box is placed, move to the next box and the preceding slot
28             box_index, slot_index = box_index + 1, slot_index - 1
29
30         # Return the number of boxes that have been successfully placed
31         return box_index
32
33 # Example usage:
34 # solution = Solution()
35 # print(solution.maxBoxesInWarehouse([1, 2, 2, 3], [3, 4, 1, 2]))
36
```

Java Solution

```
1 class Solution {
2     public int maxBoxesInWarehouse(int[] boxes, int[] warehouse) {
3         // Number of rooms in the warehouse
4         int warehouseSize = warehouse.length;
5
6         // Create an array to store the max height limit from the left to each room
7         int[] maxHeightFromLeft = new int[warehouseSize];
8         maxHeightFromLeft[0] = warehouse[0];
9
10        // Calculate the max height limit for each room from the left towards right
11        for (int i = 1; i < warehouseSize; ++i) {
12            maxHeightFromLeft[i] = Math.min(maxHeightFromLeft[i - 1], warehouse[i]);
13        }
14
15        // Sort the boxes in non-decreasing order of their sizes
16        Arrays.sort(boxes);
17
18        // Start pointers from the beginning of boxes and from the end of maxHeightFromLeft
19        int boxesIndex = 0, leftIndex = warehouseSize - 1;
20
21        // Iterate over all the boxes to try to place them in the warehouse
22        while (boxesIndex < boxes.length) {
23            // Move leftIndex leftward until we find a room tall enough for the current box
24            while (leftIndex >= 0 && maxHeightFromLeft[leftIndex] < boxes[boxesIndex]) {
25                --leftIndex;
26            }
27
28            // If we have exceeded the leftmost room, break as we can't place more boxes
29            if (leftIndex < 0) {
30                break;
31            }
32
33            // Move to the next box and the next room to the left
34            ++boxesIndex;
35            --leftIndex;
36        }
37
38        // The number of boxes placed is equal to the number of boxes indexed
39        return boxesIndex;
40    }
41 }
42
```

C++ Solution

```
1 #include <vector>
2 #include <algorithm>
3
4 class Solution {
5 public:
6     int maxBoxesInWarehouse(vector<int>& boxes, vector<int>& warehouse) {
7         // Get the number of rooms in the warehouse.
8         int warehouseSize = warehouse.size();
9
10        // Create a vector to store the minimum height from the start to each room.
11        vector<int> minHeightsFromWarehouseSize(warehouseSize);
12
13        // Initialize the first room's minimum height.
14        minHeights[0] = warehouse[0];
15
16        // Compute the minimum height for each room progressing from the entrance.
17        for (int i = 1; i < warehouseSize; ++i) {
18            minHeights[i] = min(minHeights[i - 1], warehouse[i]);
19        }
20
21        // Sort the boxes in non-decreasing order of their sizes.
22        sort(boxes.begin(), boxes.end());
23
24        // Initialize pointers for boxes and spots in the warehouse.
25        int boxIndex = 0, warehouseIndex = warehouseSize - 1;
26
27        // Try to place each box in the warehouse.
28        while (boxIndex < boxes.size()) {
29            // Find the first spot from the end where the current box can fit.
30            while (warehouseIndex >= 0 && minHeights[warehouseIndex] < boxes[boxIndex]) {
31                --warehouseIndex;
32            }
33
34            // If we have scanned all spots and none can accommodate the current box, we stop.
35            if (warehouseIndex < 0) {
36                break;
37            }
38
39            // Move to the next box and the previous spot, as the current spot is taken by the current box.
40            ++boxIndex;
41            --warehouseIndex;
42        }
43
44        // The index of boxes gives the total number of boxes that can be placed in the warehouse.
45        return boxIndex;
46    }
47 };
48
```

Typescript Solution

```
1 function maxBoxesInWarehouse(boxes: number[], warehouse: number[]): number {
2     // Get the number of rooms in the warehouse.
3     const roomCount = warehouse.length;
4
5     // Create an array to store the maximum height available to the left (inclusive) starting from each room.
6     const maxHeightsToLeft: number[] = new Array(roomCount);
7
8     // The maximum height for the first room is its own height.
9     maxHeightsToLeft[0] = warehouse[0];
10
11    // Fill the maxHeightsToLeft array with the maximum height from the start up to the current room.
12    for (let i = 1; i < roomCount; ++i) {
13        maxHeightsToLeft[i] = Math.min(maxHeightsToLeft[i - 1], warehouse[i]);
14    }
15
16    // Sort the array of boxes by their sizes in ascending order.
17    boxes.sort((a, b) => a - b);
18
19    // Initialize pointers for the boxes and the warehouse rooms.
20    let boxIndex = 0;
21    let roomIndex = roomCount - 1;
22
23    // Loop through the boxes to see how many can fit in the warehouse.
24    while (boxIndex < boxes.length) {
25        // Find a room that can accommodate the current box by moving from the end to the start.
26        while (roomIndex >= 0 && maxHeightsToLeft[roomIndex] < boxes[boxIndex]) {
27            --roomIndex;
28        }
29
30        // If we've run out of rooms, stop the process.
31        if (roomIndex < 0) {
32            break;
33        }
34
35        // Increment boxIndex as the current box fits, and decrement roomIndex to fill the next box.
36        ++boxIndex;
37        --roomIndex;
38    }
39
40    // The number of boxes placed is represented by the final value of boxIndex.
41    return boxIndex;
42 }
43
```

Time and Space Complexity

Time Complexity

The time complexity of the code provided can be analyzed in the following steps:

1. Constructing the `left` array by walking through the `warehouse` array and picking the minimum of the previous `left` entry and the current `warehouse` height.
   - This process takes O(n) time where n is the length of `warehouse`.
2. Sorting the `boxes` array.
   - Assuming the sorting algorithm is Timsort (the default in Python), this will take O(b log b) time where b is the number of `boxes`.
3. The two-pointer approach to count how many boxes can fit into the warehouse.
   - This loop runs at most min(b, n) times in the worst case, where b is the length of sorted `boxes` and n is the length of the `warehouse` array.
   - Therefore, the worst-case time complexity for this loop is O(min(b, n)).

Combining all the steps, the overall time complexity of the code is O(n) + O(b log b) + O(min(b, n)). Since the sorting step is likely to dominate the time complexity, we can simplify this to O(b log b) assuming b > n.

Space Complexity

The space complexity of the code provided is as follows:

1. The `left` array of size n is created to store the minimum height of the warehouse at every point.
   - This consumes O(n) space.
2. The sorting of the `boxes` is done in-place, which does not consume additional space (ignoring the space used by the sorting algorithm itself).
   - Python's Timsort requires O(log b) space.

Therefore, the total space complexity of the code is the larger of O(n) and O(log b), which simplifies to O(n) assuming n >= log b.