1691. Maximum Height by Stacking Cuboids

Hard | Array | Dynamic Programming | Sorting | Leetcode Link

Problem Description

The problem presents a scenario where we have `n` cuboids, each with 3 dimensions: width, length, and height. The goal is to stack a subset of these cuboids on top of each other to achieve the maximum possible height. However, there are constraints when placing one cuboid on top of another. Specifically, we can only place cuboid `i` on top of cuboid `j` if every dimension of `i` is less than or equal to the corresponding dimension of `j`. Furthermore, cuboids can be rotated such that any of their edges can serve as the 'height' for the purposes of stacking, which means before stacking we can reorder the dimensions of each cuboid to maximize the height.

Intuition

A key insight to solving this problem is to notice that the task resembles the classic problem of Longest Increasing Subsequence (LIS), but with an additional dimension. Instead of a simple numeric sequence, we are dealing with a sequence of 3-dimensional objects where the "increase" condition is the ability to place one cuboid upon another.

To harness the principle of LIS, the following steps outline our approach:

1. Sort the dimensions of each individual cuboid so that for cuboid `i`, `width_i <= length_i <= height_i`. This step ensures that we consider every possible orientation of each cuboid to maximize the height when it is placed on top of the stack.

2. Sort the cuboids themselves, first by width, then by length, and then by height. Sorting the cuboids ensures that, when considering a cuboid to add to the stack, all cuboids that come before it in the sorted order are potential candidates to be directly underneath it in the stack.

3. Apply dynamic programming to find the maximum height of the stack:

   a. Create an array `f` with the same length as the number of cuboids, which will store the maximum stack height ending with the corresponding cuboid.

   b. For each cuboid `i`, look at all previous cuboids `j` and check if you can place `i` on top of `j`. This is possible if the width, length, and height of `j` are all less than or equal to those of `i`.

   c. For every `j` that `i` can be placed upon, update the maximum height of the stack ending at `i` by considering the height of the stack ending at `j` plus the height of `i`.

   d. The entry `f[i]` corresponds to the height of the tallest stack with cuboid `i` on top.

4. The answer to the problem is then the maximum value in the array `f`, as it represents the height of the tallest stack that can be formed with the given set of cuboids, considering all possible orderings and orientations.

This approach ensures that we examine every possible ordering of the cuboids to find the tallest stack, satisfying the constraints given by the problem.

Solution Approach

The solution implemented in Python follows a clear and structured approach that utilizes sorting and dynamic programming to solve the problem outlined earlier. Here's a step-by-step walkthrough of the algorithm, referencing the provided solution code:

1. **Preparing the Cuboids:**
   - Each cuboid's dimensions are sorted to ensure the smallest dimension is considered as the width, and the largest as the height. This is done using `c.sort()` for each cuboid.
   - The `cuboids` array itself is sorted to prepare for the dynamic programming step. This sorts all cuboids by their widths, lengths, and then heights in ascending order due to Python's built-in lexicographic sorting behavior when sorting lists of lists.
2. **Dynamic Programming Initialization:**
   - An array `f` is initialized with the same length as the number of `cuboids` , where `f[i]` will hold the maximum height of a stack with the `i`th cuboid on top.
3. **Building the DP Table:**
   - We go through each cuboid `i` and compare it with all previously considered cuboids `j` (0 through `i-1`). This step implements the core LIS-like comparison.
   - If cuboid `i` can be placed on top of cuboid `j`—which we check by comparing their widths, lengths, and heights—we consider the height of the `j`th stack plus the height of `i` to potentially update the height of the stack with `i` at the top. This step is performed by checking `if cuboids[j][1] <= cuboids[i][1] and cuboids[j][2] <= cuboids[i][2]:` and selecting the maximum height `f[i] = max(f[i], f[j])`.
4. **Capturing the Height of the Stack:**
   - After considering all possible cuboids `j` that can sit below `i`, we add the height of cuboid `i` (`cuboids[i][2]` since it's sorted to be last) to `f[i]` to reflect the total height of the stack with `i` at the top.
5. **Final Answer:**
   - The maximum value in the array `f` represents the tallest stack achievable, so `return max(f)` gives us the desired output.

Note that in dynamic programming, each subproblem (finding the maximum height of a stack with a particular cuboid on top) is dependent on the solutions to smaller subproblems (maximum height of stacks with other cuboids on top that could potentially be below the current one). This dependency chain allows the final solution to effectively build upon previously computed results, hence reducing what would be a complex combinatorial problem into manageable steps with greatly decreased computational redundancy.

By considering each cuboid as a node in a graph, where a directed edge from `j` to `i` implies that `i` can be placed on top of `j`, the problem becomes finding the longest path in this graph, a variation of which, in our case, is efficiently solvable using dynamic programming.

Remember, the keys to this approach are first to enable all possible orientations through sorting individual cuboids, then to sort the cuboids themselves to apply LIS principles extended to three dimensions, and finally effectively perform dynamic programming to determine the height of the tallest stack possible.

Example Walkthrough

Let's consider a small example where we have 3 cuboids with the following dimensions (width, length, height):

1. Cuboid A: (4, 6, 7)
2. Cuboid B: (1, 2, 3)
3. Cuboid C: (5, 8, 9)

Preparing the Cuboids:
We start by sorting the dimensions within each cuboid:

- A: (4, 6, 7) becomes (4, 6, 7) – no change since it's already sorted.
- B: (1, 2, 3) remains (1, 2, 3) – already sorted.
- C: (5, 8, 9) remains (5, 8, 9) – already sorted.

Next, we sort all the cuboids based on their dimensions:

- Sorted cuboids: B (1, 2, 3), A (4, 6, 7), C (5, 8, 9) – they are sorted by their widths, lengths, and heights.

Dynamic Programming Initialization:
We initialize an array `f` with the lengths of all three cuboids, as that's the maximum height they can contribute individually if placed on the bottom:
`f = [height of B, height of A, height of C]` which becomes `f = [3, 7, 9]`.

Building the DP Table:
Now, for each cuboid `i`, we check if it can sit on top of another cuboid `j`:

- Start with `i = B`, there is no `j` before `B`, so we move to the next cuboid.
- For `i = A`, we compare with previous cuboid `j = B`. Since `(1, 2, 3)` is smaller than `(4, 6, 7)` in all dimensions, `A` can sit on top of `B`. So, `f[A] = max(f[A], f[B] + height of A)` which transforms `f` into `[3, 10, 9]`.
- For `i = C`, we compare with both `A` and `B`:
  - It can sit on top of `B`, so `f[C] = max(f[C], f[B] + height of C)` does not change `f` because `f[C]` is already greater.
  - It can also sit on top of `A`, so `f[C] = max(f[C], f[A] + height of C)` updates `f[C]` from `9` to `10 + 9 = 19`.

The updated `f` array is now `[3, 10, 19]`.

Capturing the Height of the Stack:
The maximum heights of stacks ending with each cuboid are already stored in `f`.

Final Answer:
`return max(f)` will give us the height of the tallest possible stack, which is `max([3, 10, 19])` or `19`.

In this example, the tallest stack is obtained by placing `C` on top of `A`, with `B` not being used. Thus, by performing these steps, we have calculated the maximum height of a stack that can be formed from these cuboids considering all constraints and possible orientations.

Python Solution

```python
from typing import List

class Solution:
    def maxHeight(self, cuboids: List[List[int]]) -> int:
        # Sort each individual cuboid's dimensions to have them in non-decreasing order
        for cuboid in cuboids:
            cuboid.sort()

        # Sort all the cuboids based on their dimensions
        cuboids.sort()

        # Initialize the number of cuboids
        n = len(cuboids)

        # Initialize an array to store the maximum height of a stack ending with the i-th cuboid
        max_heights = [0] * n

        # Iterate over each cuboid to calculate the maximum height of a stack
        for i in range(n):
            # Check all previous cuboids to see if we can stack them
            for j in range(i):
                # If the dimensions of the j-th cuboid are less than or equal to
                # the dimensions of i-th cuboid, we may stack it on top
                if cuboids[j][1] <= cuboids[i][1] and cuboids[j][2] <= cuboids[i][2]:
                    # We update the max height for the i-th cuboid
                    max_heights[i] = max(max_heights[i], max_heights[j])
            # Add the height of the current cuboid to the max height value
            max_heights[i] += cuboids[i][2]

        # Return the maximum height from the max_heights array
        return max(max_heights)

# An example of how this could be used:
solution = Solution()
print(solution.maxHeight([[1, 2, 3], [3, 2, 1], [2, 3, 1]]))  # Example input

```

Java Solution

```java
class Solution {

    public int maxHeight(int[][] cuboids) {
        // Sort each individual cuboid array to have the dimensions in non-decreasing order
        for (int[] cuboid : cuboids) {
            Arrays.sort(cuboid);
        }

        // Sort cuboids based on the first dimension, then second, then third if preceding are equal
        Arrays.sort(cuboids, (a, b) -> {
            if (a[0] != b[0]) return a[0] - b[0];
            if (a[1] != b[1]) return a[1] - b[1];
            return a[2] - b[2];
        });

        int numCuboids = cuboids.length; // The total number of cuboids
        int[] dp = new int[numCuboids];  // Dynamic programming array for storing maximum heights

        // Populate the dp array with the maximum heights
        for (int i = 0; i < numCuboids; ++i) {
            // Initialize dp[i] with cuboid's own height, as it can stand on its own without any cuboids beneath it
            dp[i] = cuboids[i][2];
            // Check all previous cuboids to see if we can stack current cuboid on top of them
            for (int j = 0; j < i; ++j) {
                // If the lower and upper base dimensions of cuboid j are less than or equal to those of cuboid i,
                // it means cuboid j can be placed under cuboid i while maintaining the stacking rules
                if (cuboids[j][1] <= cuboids[i][1] && cuboids[j][2] <= cuboids[i][2]) {
                    // Find the maximum height if we stack cuboid i on top of cuboid j
                    dp[i] = Math.max(dp[i], dp[j] + cuboids[i][2]);
                }
            }
        }

        // Return the maximum value from the dp array, which is the tallest possible stack height
        return Arrays.stream(dp).max().getAsInt();
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maxHeight(vector<vector<int>>& cuboids) {
        // Sort dimensions of each cuboid individually to make sure they are in non-decreasing order.
        for (auto& cuboid : cuboids) {
            sort(cuboid.begin(), cuboid.end());
        }

        // Sort the list of cuboids based on their dimensions to ensure a non-decreasing order.
        // This will help to easily find out if a cuboid can be placed on top of another.
        sort(cuboids.begin(), cuboids.end());

        int n = cuboids.size(); // The total number of cuboids.
        vector<int> dp(n);      // Create dp array to store the maximum height up to each cuboid.

        // Iterate through each cuboid to calculate the maximum stack height when it is at the top.
        for (int i = 0; i < n; ++i) {
            // Check all previous cuboids to see if we can stack any of them on the current one.
            for (int j = 0; j < i; ++j) {
                // A cuboid can be placed on top if its length and width are less than or equal
                // to those of the cuboid under it.
                if (cuboids[j][1] <= cuboids[i][1] && cuboids[j][2] <= cuboids[i][2]) {
                    // Update dp[i] with the maximum height we can get by stacking cuboids up to j.
                    dp[i] = max(dp[i], dp[j]);
                }
            }
            // Add the height of the current cuboid to the maximum height of the stack below it.
            dp[i] += cuboids[i][2];
        }

        // The final answer is the maximum value in dp, which represents the tallest stack possible.
        return *max_element(dp.begin(), dp.end());
    }
};

```

Typescript Solution

```typescript
/**
 * This function calculates the maximum height of the stacked cuboids.
 * Each cuboid's dimensions are first sorted to facilitate stacking.
 * The function then computes the maximum stack height for every cuboid that
 * could be placed at the bottom and then returns the maximum height obtained.
 *
 * @param {number[][]} cuboids - The array of cuboids with unsorted dimensions.
 * @return {number} The maximum height stack that can be obtained by stacking the cuboids.
 */
function maxHeight(cuboids: number[][]): number {
    // Sort each individual cuboid's dimensions from smallest to largest
    cuboids.forEach((cuboid: number[]) => {
        cuboid.sort((a, b) => a - b);
    });

    // Sort cuboids in ascending order of dimensions to stack them properly
    cuboids.sort((a, b) => {
        if (a[0] !== b[0]) return a[0] - b[0];
        if (a[1] !== b[1]) return a[1] - b[1];
        return a[2] - b[2];
    });

    // n is the total number of cuboids
    const n: number = cuboids.length;
    // f represents the maximum stack height ending with cuboid i
    const f: number[] = new Array(n).fill(0);

    // Calculate the maximum height for each cuboid as the base
    for (let i = 0; i < n; ++i) {
        for (let j = 0; j < i; ++j) {
            // Check if cuboid j can be stacked on cuboid i
            const canStack: boolean =
                cuboids[j][1] <= cuboids[i][1] && cuboids[j][2] <= cuboids[i][2];
            // Update the maximum height if stacking is possible
            if (canStack) f[i] = Math.max(f[i], f[j]);
        }
        // Include the height of the current cuboid
        f[i] += cuboids[i][2];
    }
    // Return the maximum value from the array of maximum heights
    return Math.max(...f);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function primarily involves two sorts and a double nested loop.

1. Sorting each cuboid in `cuboids` - Each individual sort takes `O(1)` time since the size of each cuboid is constant (3). Therefore, the time for this part is `O(n)` where `n` is the number of cuboids.

2. Sorting the array of cuboids - This is done using the default sort function, which for Python's Timsort algorithm has a time complexity of `O(n log n)`.

3. The double nested loop for computing the maximum height:

   - The outer loop runs `n` times.
   - The inner loop runs up to `i` times which in the worst case is `n-1`.
   - Each comparison operation inside the inner loop is `O(1)`.

Combining the number of iterations for both loops, the double nested loop contributes `O(n^2)` to the time complexity.

Adding all of these together, the dominating term is the `O(n^2)` from the double nested loop. Thus, the overall time complexity of the code is `O(n^2)`.

Space Complexity

The space complexity is determined by the additional memory that the algorithm uses:

1. A new list `f` of size `n` is created to store the maximum heights of each cuboid.

Other than that and the input list manipulation, no other significant space is used.

Thus, the space complexity of the algorithm is `O(n)`.