# 1840. Maximum Building Height

Hard Array Math Leetcode Link

## Problem Description

In this problem, we're tasked with determining the maximum possible height of the tallest building that can be built, given some specific constraints.

We have to build `n` new buildings in a line, labeled from `1` to `n`, with the following restrictions:

- The height of each building must be a non-negative integer.
- The height of the first building must be `0`.
- The height difference between any two adjacent buildings cannot exceed `1`. This means each building can either be the same height as the previous one, one unit taller, or one unit shorter.

Additionally, there's an array called `restrictions` that provides further limits on the height of certain buildings. This array is composed of elements `[id_i, maxHeight_i]`, which means the building with label `id_i` must not exceed the height `maxHeight_i`. It's assured that each building appears at most once in the array, and the first building (with label `1`) will not be included in the restrictions.

The goal is to find the maximum height achievable for the tallest building among those built under these constraints.

## Intuition

To approach this problem, we can follow these steps:

- First, we add a restriction for the first building with id `1` and height `0` since it's given that the first building's height must be `0`. Also, if the last building `n` doesn't have a restriction, we add a default restriction for it as `[n, n - 1]`, supposing we could reach the maximum height one less than its position since each building height can increase by at most `1`.

- Then, we sort the restrictions by building id so that we can iterate through the buildings in ascending order.

- The height restrictions for buildings in between the given restrictions can be effectively calculated by a forward pass, updating the `maxHeight` to the minimum of its own height or the height imposed by the previous building's `maxHeight` increased by the difference in their ids. This ensures that a building conforms not only to its explicit restrictions but also maintains the difference of `1` height unit constraint with previous buildings.

- We repeat a similar process in a backward pass. We iterate backward through the array and update each `maxHeight` to be the minimum of its own or the height possible due to the height constraint from the next building decreased by the difference in ids. This balances the height constraints both from forward and backward directions for each building.

- The tallest possible building will be at the location where we can maximize the given constraints. To find this maximum, we examine each gap between two adjacent restrictions after the above forward and backward passes. Due to the constraint that adjacent buildings can have at most a height difference of `1`, the maximum height in a gap will be at the halfway point between the two restrictions.

- By examining each gap and calculating this potential maximum height using the adjusted restrictions, we can determine the overall maximum height of any building. The final answer is the maximum of these potential maximum heights.

The intuition behind this approach is that each restriction "ripples" through its neighbors ahead and behind, effectively spreading the constraints throughout the range of buildings. We seek a balance point within each gap between restrictions where we can attain the tallest height before descending down to meet the next restriction.

## Solution Approach

The solution provided uses a few important programming concepts to achieve the desired result: sorting, iteration, and comparative logic.

Here's a step-by-step breakdown of the implementation:

1. **Inserting Boundary Restrictions:**
    - The code begins by appending the artificial restrictions `r.append([1, 0])` for the first building, which must always be height `0`, and `r.append([n, n - 1])` for the last building, if there isn't any existing restriction for it, which ensures that the possible height of the last building `n` is within the permissible range subject to the incremental restriction.
2. **Sorting Restrictions:**
    - The list of restrictions `r` is then sorted by the building `id` (`r.sort()`), so that we can consider the restrictions in the order of building positions.
3. **Forward Pass:**
    - A forward iteration (`for i in range(1, m):`) adjusts each building's height based on the constraints "rippling forward." Specifically, each restriction's maxHeight is updated (`r[i][1] = min(r[i][1], r[i - 1][1] + r[i][0] - r[i - 1][0])`) to the smaller of the existing restriction or the height allowed by the previous building's restriction plus the difference between their ids. This maintains the constraint that no two adjacent buildings differ in height by more than `1`.
4. **Backward Pass:**
    - Next, the code performs a backward iteration (`for i in range(m - 2, 0, -1):`) to adjust the heights based on the restrictions "rippling backward." This adjustment (`r[i][1] = min(r[i][1], r[i + 1][1] + r[i + 1][0] - r[i][0])`) ensures that each building's height is the smaller of the existing restriction or what is allowed by the next building's restriction minus the difference between their ids.
5. **Finding the Maximum Height:**
    - The code then enters a loop that examines the gaps between two adjacent restrictions (`for i in range(m - 1):`). The potential maximum height at the midpoint of the gap is calculated as `t = (r[i][1] + r[i + 1][1] + r[i + 1][0] - r[i][0]) // 2`. This accounts for the "climbing up" from one restriction and "climbing down" from the next, finding the highest point where both constraints meet.
6. **Computing the Answer:**
    - Finally, within this loop, the answer (`ans`) is updated to the maximum height found (`ans = max(ans, t)`) after each iteration.

The algorithm smartly leverages the given constraints in a constructive manner by first ensuring that all imposed restrictions are met and then finding the maximum height that can still be achieved within those limits. It doesn't use any complex data structures, just a list to store the restrictions and simple iterations to propagate the constraints through adjacent buildings.

### Example Walkthrough

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

Suppose we need to construct 5 buildings and the array of restrictions is `restrictions = [[3, 2], [5, 3]]`. This implies building 3 must not exceed height 2, and building 5 must not exceed height 3.

1. **Inserting Boundary Restrictions:**
    - We start by appending artificial restrictions `[1, 0]` for the first building and `[5, 4]` for the last building (since there is an existing restriction, we don't change it). Now the updated restrictions list is `r = [[1, 0], [3, 2], [5, 3]]`.
2. **Sorting Restrictions:**
    - `r` is already sorted by the building ids `[1, 3, 5]`, so we do not need to perform a sort in this case.
3. **Forward Pass:**
    - We then apply the forward pass. There is no update required for building 2, since the default increasing sequence satisfies the restrictions. The height of building 3 remains `2`, as it satisfies its restriction and the incremental rule.
    - Updated `r = [[1, 0], [3, 2], [5, 3]]` (no changes in this phase).
4. **Backward Pass:**
    - Now applying the backward pass: For building 3, the maxHeight becomes `min(2, 3 + (5 - 3)) = 2`, which is no change.
    - For building 1, there is still no change as the minHeight 0 is already as low as it can be.
    - Final updated `r = [[1, 0], [3, 2], [5, 3]]` (again, no changes in this phase as the restrictions match the increasing step rule).
5. **Finding the Maximum Height:**
    - Looking at the gap between buildings 1 and 3, using `(r[i][1] + r[i + 1][1] + r[i + 1][0] - r[i][0]) // 2`, we calculate `t = (0 + 2 + 3 - 1) // 2 = 2`.
    - For the gap between buildings 3 and 5, we calculate `t = (2 + 3 + 5 - 3) // 2 = 3`.
6. **Computing the Answer:**
    - The final answer `ans` is the maximum value from the calculated midpoint heights, which in this case is `max(2, 3) = 3`.

In this example, the tallest building that can be built will have a maximum height of 3. This approach illustrates how the algorithm efficiently adheres to the constraints by working within the boundaries defined by the restrictions and incrementally adjusting the heights of the buildings through forward and backward passes. The smart traversal of gaps between restrictions helps find the highest feasible point that satisfies both the '1 unit height difference' rule and the height limits given in any additional restrictions.

## Python Solution

```python
class Solution:
    def maxBuilding(self, n: int, restrictions: List[List[int]]) -> int:
        # Add the first building restriction (building 1 can't exceed height 0)
        restrictions.append([1, 0])
        # Sort restrictions based on the building index
        restrictions.sort()

        # If the last restriction is not for the last building, add a restriction for it
        # The height of the last building can't exceed n - 1 (1-indexed)
        if restrictions[-1][0] != n:
            restrictions.append([n, n - 1])

        # Update the maximum height for each building in the forward direction
        # Ensuring that each next building isn't taller than the previous plus the distance between them
        for i in range(1, len(restrictions)):
            restrictions[i][1] = min(restrictions[i][1], restrictions[i - 1][1] + restrictions[i][0] - restrictions[i - 1][0])

        # Update the maximum height for each building in the backward direction
        # Making sure it follows restrictions coming from the buildings in front
        for i in range(len(restrictions) - 2, -1, -1):
            restrictions[i][1] = min(restrictions[i][1], restrictions[i + 1][1] + restrictions[i + 1][0] - restrictions[i][0])

        # Calculate the maximum height we can achieve between each adjacent pair of buildings
        max_height = 0
        for i in range(len(restrictions) - 1):
            # Maximum possible height between two buildings, considering their restrictions and distance
            peak_height = (restrictions[i][1] + restrictions[i + 1][1] + restrictions[i + 1][0] - restrictions[i][0]) // 2
            max_height = max(max_height, peak_height)

        # Return the maximum height found
        return max_height

```

## Java Solution

```java
import java.util.*;

class Solution {
    public int maxBuilding(int totalBuildings, int[][] restrictions) {
        // Create a list to hold the restrictions and add the initial and end restrictions
        List<int[]> restrictionList = new ArrayList<>();
        restrictionList.add(new int[]{1, 0}); // Add a restriction for the first building, height 0
        restrictionList.addAll(Arrays.asList(restrictions)); // Add all other restrictions

        // Sort the list of restrictions by the building index in ascending order
        Collections.sort(restrictionList, (a, b) -> Integer.compare(a[0], b[0]));

        // Ensure an end restriction is in place if not already specified
        if (restrictionList.get(restrictionList.size() - 1)[0] != totalBuildings) {
            restrictionList.add(new int[]{totalBuildings, totalBuildings - 1});
        }

        int restrictionCount = restrictionList.size();

        // Forward pass: ensure that each restriction respects the previous one
        for (int i = 1; i < restrictionCount; ++i) {
            int[] previousRestriction = restrictionList.get(i - 1);
            int[] currentRestriction = restrictionList.get(i);
            currentRestriction[1] = Math.min(currentRestriction[1], previousRestriction[1] + currentRestriction[0] - previousRestri
        }

        // Backward pass: adjust restrictions based on later restrictions
        for (int i = restrictionCount - 2; i >= 0; --i) {
            int[] currentRestriction = restrictionList.get(i);
            int[] nextRestriction = restrictionList.get(i + 1);
            currentRestriction[1] = Math.min(currentRestriction[1], nextRestriction[1] + nextRestriction[0] - currentRestriction[0]
        }

        // Find the maximum possible height between each pair of adjacent restrictions
        int maximumHeight = 0;
        for (int i = 0; i < restrictionCount - 1; ++i) {
            int[] currentRestriction = restrictionList.get(i);
            int[] nextRestriction = restrictionList.get(i + 1);
            // Calculate potential max height between restrictions, considering the allowed increase/decrease in height
            int maxHeightBetween = (currentRestriction[1] + nextRestriction[1] + nextRestriction[0] - currentRestriction[0]) / 2;
            maximumHeight = Math.max(maximumHeight, maxHeightBetween);
        }

        return maximumHeight; // Return the maximum height found
    }
}

```

## C++ Solution

```cpp
#include <vector>
#include <algorithm>

using namespace std;

class Solution {
public:
    int maxBuilding(int n, vector<vector<int>>& restrictions) {

        // Append the first restriction explicitly - starting at building 1, height is 0.
        restrictions.push_back({1, 0});
        sort(restrictions.begin(), restrictions.end());

        // If there is no restriction for the last building, add it with maximum possible height.
        if (restrictions.back()[0] != n) restrictions.push_back({n, n - 1});

        // Update restrictions from left to right.
        for (int i = 1; i < restrictions.size(); ++i) {
            // The restriction height should be the minimum of its current value and the maximum height allowed by the previous res
            restrictions[i][1] = min(restrictions[i][1], restrictions[i - 1][1] + restrictions[i][0] - restrictions[i - 1][0]);
        }

        // Update restrictions from right to left.
        for (int i = restrictions.size() - 2; i > 0; --i) {
            // The restriction height should be the minimum of its current value and the maximum height allowed by the next restric
            restrictions[i][1] = min(restrictions[i][1], restrictions[i + 1][1] + restrictions[i + 1][0] - restrictions[i][0]);
        }

        // Calculate the maximum height we can build for the entire range.
        int maxPossibleHeight = 0;  // Initialize the answer to be the maximum possible height.
        for (int i = 0; i < restrictions.size() - 1; ++i) {
            // This is the theoretical maximum height that can be achieved at the mid-point between two restrictions, taking both i
            int peakHeight = (restrictions[i][1] + restrictions[i + 1][1] + restrictions[i + 1][0] - restrictions[i][0]) / 2;
            // Update the maximum possible height if the current peak is higher.
            maxPossibleHeight = max(maxPossibleHeight, peakHeight);
        }

        // Return the maximum possible height that can be achieved.
        return maxPossibleHeight;
    }
};

```

## Typescript Solution

```typescript
function maxBuilding(n: number, restrictions: number[][]): number {
  // Append the first restriction explicitly - starting at building 1, height is 0.
  restrictions.push([1, 0]);
  restrictions.sort((a, b) => a[0] - b[0]);

  // If there is no restriction for the last building, add it with maximum possible height.
  if (restrictions[restrictions.length - 1][0] !== n) {
    restrictions.push([n, n - 1]);
  }

  // Update the restrictions from left to right.
  for (let i = 1; i < restrictions.length; ++i) {
    restrictions[i][1] = Math.min(
      restrictions[i][1],
      restrictions[i - 1][1] + restrictions[i][0] - restrictions[i - 1][0]
    );
  }

  // Update the restrictions from right to left.
  for (let i = restrictions.length - 2; i >= 0; --i) {
    restrictions[i][1] = Math.min(
      restrictions[i][1],
      restrictions[i + 1][1] + restrictions[i + 1][0] - restrictions[i][0]
    );
  }

  // Calculate the maximum height we can build for the entire range.
  let maxPossibleHeight = 0; // Initialize the variable to keep track of the maximum possible building height.
  for (let i = 0; i < restrictions.length - 1; ++i) {
    // Calculate the maximum height at the midpoint between two restrictions.
    let peakHeight =
      (restrictions[i][1] + restrictions[i + 1][1] +
        restrictions[i + 1][0] - restrictions[i][0]) /
      2;
    // Update the maximum possible height if the current peak height is higher.
    maxPossibleHeight = Math.max(maxPossibleHeight, peakHeight);
  }

  // Return the maximum possible height that can be achieved given the restrictions.
  return maxPossibleHeight;
}

```

## Time and Space Complexity

### Time Complexity

The time complexity consists of the following parts:

1. Appending initial and end restrictions: `O(1)` - Constant time operations to append two additional restrictions.
2. Sorting the restrictions: `O(m log m)` - Sorting `m` restrictions where `m` is the number of restrictions including the two appended ones.
3. Forward pass to adjust the heights: `O(m)` - A single pass through the sorted restrictions to enforce that the height difference between consecutive buildings doesn't exceed the difference in their indices.
4. Backward pass to adjust the heights: `O(m)` - Another single pass through the sorted restrictions in the reverse direction for the same purpose as in step 3.
5. Finding the maximum height: `O(m)` - A final pass to find the maximum height between consecutive buildings.

Since sorting dominates the complexity, the total time complexity is `O(m log m)` where `m` is the number of restrictions including the added ones.

### Space Complexity

The space complexity is comprised of:

1. Storing the restrictions list with the two additional restrictions: `O(m)` - Where `m` is the number of restrictions including the two appended ones.
2. No additional significant space is used.

Thus, the total space complexity is `O(m)` where `m` is the number of restrictions including the added ones.