## 1642. Furthest Building You Can Reach

Greedy Array Heap (Priority Queue)

# **Problem Description**

Medium

certain number of bricks and ladders. As you move from one building to the next, you can either use bricks or ladders based on the following rules: • If the building you are currently on is of equal or greater height than the next one, you can move ahead without using any bricks or ladders.

In this problem, you are given an array heights that represents the heights of a series of buildings in a line. You are also given a

- If the building you are currently on is shorter than the next one, you need to bridge the height difference by either using a ladder or using as many bricks as the height difference.
- Your goal is to figure out the farthest building index (0-indexed) you can reach by using the ladders and bricks in the best way possible.

Intuition

The intuition behind the solution revolves around optimizing the use of ladders, which are more versatile and valuable than bricks.

#### Since ladders can cover any height difference and you have a limited number of them, you want to use ladders for the largest height differences. For smaller height differences, it's more economical to use bricks.

One effective approach to solve this problem is using a min-heap to keep track of the ladders largest height differences encountered so far as you iterate through the array. Here's the thought process: 1. As you traverse each building, calculate the height difference with the next building if it's greater than the current one.

2. Add these differences to a min-heap. Since the size of the heap is limited to the number of available ladders, whenever you add an item and the heap size exceeds the number of ladders, you pop the smallest item. This represents using bricks for the smallest climb that you initially thought

- of using a ladder for. 3. When popping from the heap (which means using bricks instead of a ladder), subtract the height difference (number of bricks needed) from the
- available bricks. 4. If at any point you run out of bricks, this means you can't make the climb using bricks anymore, and you've used all your ladders for higher climbs. The current index is the furthest you can reach.
- This strategy ensures that ladders are reserved for the biggest climbs, and bricks are used for the smaller ones, which is an optimal way to go as far as possible.

The solution approach uses a greedy algorithm to decide when to use bricks and when to save the ladders for taller buildings. To implement this strategy, we use a min-heap, which is a data structure that allows us to efficiently track and update the smallest height differences for which we've decided to use ladders.

## We initialize a min-heap h as an empty list, which will store the heights differences where a ladder is used. We then iterate through the heights array with the variable i indicating our current position, comparing each building a with

the next building b.

Here are the steps taken in the Python code implementation:

current index i because that's the furthest we can go.

overall time complexity of the algorithm controlled and efficient.

Let's walk through a small example to illustrate the solution approach:

Imagine you have an array heights = [4, 2, 7, 6, 9, 14, 12], bricks = 5, and ladders = 1.

**Solution Approach** 

For each pair of buildings where building b is taller (i.e., the height difference d is greater than 0), we push the difference to the heap using heappush(h, d).

We then check if the length of the heap exceeds the number of available ladders. If it does, this means we must use bricks

instead of a ladder for the smallest height difference encountered so far. To do this, we pop from the heap using heappop(h),

If we finish iterating through the array without running out of bricks, it means we were able to reach the last building. We

which removes and returns the smallest item from the heap. We then subtract this value from our available bricks. If at any point our count of bricks falls below 0, we can no longer proceed to the next building using bricks. We return the

return len(heights) - 1 as we have successfully reached the end. The use of a min-heap here is crucial for keeping our algorithm efficient. Without the heap, we would have to search through all

height differences we've encountered whenever we need to use bricks, which would make our solution much slower. With the

min-heap, we ensure that we are always removing the smallest height difference in logarithmic time complexity, which keeps the

**Example Walkthrough** 

Following the steps of the solution: Initialize a Min-Heap: Start with an empty min-heap h. Iterate Over Buildings: Begin traversing the buildings. The comparison starts at the first building (i = 0) and goes to the second last one.

• No check necessary since we have used only one ladder till now, and the heap size is within the limit of available ladders (heap size = 1,

○ Again, we have more items in the heap than ladders. Pop the smallest item (which is 5) and subtract from the bricks (bricks = 2 - 5 =

• We are currently at index 4 (heights[4] = 9) and can't move to index 5 because we have a negative brick count. The farthest index we

In the given example, the algorithm allows us to move as far as possible, using ladders for the most significant height differences

when necessary and bricks for the smallest one until the bricks run out. The index 4 is the farthest building we can reach given

### • From 7 to 6: No action needed, as the next building is shorter. **Heap Size Check and Bricks Use:**

ladders = 1).

**Building Heights Comparison:** 

**Building Heights Comparison Continued:** 

**Heap Size Exceeds Available Ladders:** 

**Building Heights Comparison Continued:** 

**Heap Size Exceeds Again:** 

**Conclusion:** 

could reach is 4.

Solution Implementation

from heapq import heappush, heappop

height\_diffs\_heap = []

for i in range(len(heights) - 1):

if height diff > 0:

return len(heights) - 1

**if** (diff > 0) {

current height = heights[i]

next height = heights[i + 1]

height\_diff = next\_height - current\_height

10.

**Python** 

Java

C++

**}**;

function popHeap() {

return buildingCount - 1;

height\_diffs\_heap = []

for i in range(len(heights) - 1):

if height diff > 0:

current height = heights[i]

next height = heights[i + 1]

if bricks < 0:</pre>

return i

height diff = next height - current height

heappush(height diffs heap, height diff)

buildings). For each building, it may add a height difference to a min-heap:

if len(height diffs heap) > ladders:

// Example usage:

// Go through each building except the last one.

for (let i = 0; i < buildingCount - 1; ++i) {</pre>

class Solution {

class Solution:

 Because we have more items in the heap than available ladders, we must pop the smallest item. We pop 3 and subtract it from the bricks (bricks = 5 - 3 = 2).

From 6 to 9: The height difference is 3, add this to the heap (h = [3, 5]).

From 9 to 14: The height difference is 5. Add this to the heap (h = [5, 5]).

• From 4 to 2: No bricks or ladders needed because the next building is shorter.

From 2 to 7: The height difference is 5, so we push 5 into the heap (h = [5]).

- **−3). Bricks Are Depleted:** Our brick count goes below 0, which means we cannot proceed further using bricks.
- the constraints of our bricks and ladders.

:return: The index of the furthest building that can be reached

public int furthestBuilding(int[] heights, int bricks, int ladders) {

// Get the number of buildings from the heights array.

// Iterate through the array of building heights.

for (int i = 0; i < numberOfBuildings - 1; <math>i++) {

int numberOfBuildings = heights.length;

int currentHeight = heights[i];

int nextHeight = heights[i + 1];

int diff = nextHeight - currentHeight;

heightDifferences.offer(diff);

**if** (bricks < 0) {

return i;

return numberOfBuildings - 1;

PriorityQueue<Integer> heightDifferences = new PriorityQueue<>();

// Current building height and the next building height.

// If the next building is taller, a climb is needed.

if (heightDifferences.size() > ladders) {

bricks -= heightDifferences.poll();

// If we can climb all buildings, return the last building index.

// Add the height difference to the priority queue.

# A priority queue (min heap) to store the heights that we have used ladders for.

# We use a ladder and add the height difference to the heap.

// Create a priority queue to store the heights that we can climb using ladders.

// Calculate the height difference between the current and next building.

// If we have used more ladders than available, we use bricks.

// Remove the smallest height difference and use bricks to climb up.

// If we do not have enough bricks to climb, return the current index.

# Calculate the height difference between the current building and the next one.

# Only if the next building is higher than the current one do we need ladders or bricks

def furthest\_building(self, heights, bricks, ladders):

This method determines how far you can reach by climbing buildings of various heights using a given number of bricks and ladd :param heights: A list of integers representing the heights of buildings :param bricks: The total number of bricks available to climb up the buildings :param ladders: The total number of ladders available to climb up the buildings

heappush(height diffs heap, height diff) # If we have used more ladders than available, we must replace one ladder with bricks. if len(height diffs heap) > ladders: bricks -= heappop(height diffs heap) # Replace the ladder for the smallest height diff. # If at any point we do not have enough bricks, we cannot move to the next building. if bricks < 0:</pre> return i # If we can climb all the buildings with the given bricks and ladders, return the last building index.

```
#include <vector>
#include <queue>
using namespace std;
class Solution {
public:
    int furthestBuilding(vector<int>& heights, int bricks, int ladders) {
        // Min-heap to keep track of the minimum heights we have had to jump using ladders
        priority queue<int, vector<int>, greater<int>> min heap;
        int buildingCount = heights.size(); // Total number of buildings
        // Go through each building except the last one
        for (int i = 0; i < buildingCount - 1; ++i) {
            int current height = heights[i]:  // Height of the current building
            int next height = heights[i + 1];  // Height of the next building
            int height_difference = next_height - current_height; // Calculate the height difference
            if (height difference > 0) { // If the next building is taller
                min heap.push(height difference); // Use a ladder for now to climb up
                if (min heap.size() > ladders) { // If we have used more ladders than we have
                    bricks -= min heap.top(); // Replace one ladder use with bricks
                    min heap.pop(); // Remove the smallest height difference we overcame with a ladder
                    if (bricks < 0) { // If we don't have enough bricks to go to the next building</pre>
                        return i; // Return the index of the current building
        // If we manage to consider all the buildings, return the index of the last building
        return buildingCount - 1;
TypeScript
function furthestBuilding(heights: number[], bricks: number, ladders: number): number {
    // Since TypeScript does not have a built-in priority queue, we'll use an array to simulate it.
    let minHeap: number[] = []; // This will function as our min-heap.
    const buildingCount = heights.length; // Total number of buildings.
    // Helper function to simulate the push operation of the priority queue.
    function pushHeap(value: number) {
        minHeap.push(value):
        minHeap.sort((a, b) => a - b); // Sort to keep the smallest elements at the start.
    // Helper function to simulate the pop operation of the priority queue.
```

```
// const result = furthestBuilding([4, 2, 7, 6, 9, 14, 12], 5, 1);
// console.log(result); // Outputs the index of the furthest building that can be reached.
from heapq import heappush, heappop
class Solution:
   def furthest building(self, heights, bricks, ladders):
       This method determines how far you can reach by climbing buildings of various heights using a given number of bricks and ladd
```

minHeap.shift(); // Remove the smallest element, akin to popping from a min-heap.

const heightDifference = nextHeight - currentHeight; // Calculate the height difference.

pushHeap(heightDifference); // Use a ladder, represented by storing heightDifference.

if (minHeap.length > ladders) { // If we've simulated using more ladders than we have.

popHeap(); // Remove the smallest height difference we've overcome with a ladder.

if (bricks < 0) { // If we don't have enough bricks to reach the next building.</pre>

const currentHeight = heights[i]; // Height of the current building.

bricks -= minHeap[0]; // Replace one ladder use with bricks.

return i; // Return the index of the current building.

// If able to consider all the buildings, return the index of the last building.

:param heights: A list of integers representing the heights of buildings

:return: The index of the furthest building that can be reached

:param bricks: The total number of bricks available to climb up the buildings

# We use a ladder and add the height difference to the heap.

:param ladders: The total number of ladders available to climb up the buildings

# A priority queue (min heap) to store the heights that we have used ladders for.

# Calculate the height difference between the current building and the next one.

# Only if the next building is higher than the current one do we need ladders or bricks

# If we have used more ladders than available, we must replace one ladder with bricks.

bricks -= heappop(height diffs heap) # Replace the ladder for the smallest height diff.

# If at any point we do not have enough bricks, we cannot move to the next building.

const nextHeight = heights[i + 1]; // Height of the next building.

if (heightDifference > 0) { // If the next building is taller.

# If we can climb all the buildings with the given bricks and ladders, return the last building index. return len(heights) - 1 Time and Space Complexity **Time Complexity:** 

The function furthestBuilding iterates through the heights array once, which has n elements (n being the number of

number of elements in the heap. In the worst case, the heap size could be equal to the number of ladders (1). Thus, the worst-case time for insertion over n - 1 iterations is O((n - 1) \* log 1).

The for-loop iterates n - 1 times as it skips the last building. We know that inserting into a heap is  $0(\log k)$ , where k is the

The number of removals from the heap is at most equal to the number of ladders 1, so the total time for all removals is 0(1 \*

log 1) assuming each removal is followed by a heapify operation, which is 0(log 1). Combining these, since 1 is less than or equal to n - 1, we get the total time complexity to be  $0(n \log 1)$ .

# **Space Complexity:**

•

The space complexity is mostly determined by the min-heap that stores the height differences. In the worst case, the heap could store as many elements as there are ladders, so the space complexity is 0(1).