1465. Maximum Area of a Piece of Cake After Horizontal and Vertical Cuts

Problem Description

Medium Greedy Array Sorting

The given problem presents a scenario where we have a rectangular cake with a specific height (h) and width (w). We are

provided with two lists of integers: horizontalCuts and verticalCuts. The integers in horizontalCuts represent positions of horizontal slices measured from the top edge of the cake, while the integers in verticalCuts represent positions of vertical slices measured from the left edge of the cake. Our task is to determine the maximum area of a single piece of the cake that results from making these cuts. It's important to note that when making cuts, we are essentially dividing the cake into smaller rectangular pieces. The challenge here is to identify

which of these pieces will have the maximum area after performing all the given cuts. The maximum area of a piece of cake can be found by looking at the largest spacing between horizontal cuts and the largest spacing between vertical cuts. When multiplied together, these spaces will give us the area of the largest piece.

Since the resulting area can be quite large, we are instructed to return the answer modulo 10^9 + 7, which is a common technique used to avoid overflow in programming problems that involve large numbers.

Intuition

The intuition behind the solution is to first add the edges of the cake to our list of cuts since we can consider them as cuts at

positions 0 and h for horizontal cuts, and 0 and w for vertical cuts. Next, we sort both horizontalCuts and verticalCuts

The maximum area of a piece of cake can then be derived from the largest horizontal gap (the maximal difference between any

arrays. This ordered list of cuts allows us to simply iterate through each array and calculate the differences between successive cuts.

two successive horizontal cuts) and the largest vertical gap (the maximal difference between any two successive vertical cuts). By multiplying these two largest gaps together, we get the area of the largest piece of cake possible after performing all the cuts. To calculate the maximum differences, we can use the pairwise function provided by Python, which gives us each pair of adjacent elements from our sorted list. Then, we simply find the maximum gap (difference) in both horizontal and vertical directions. In conclusion, our solution strategy starts with sorting the cuts, finding the largest gaps, and then calculating the resulting

Solution Approach The solution approach follows an algorithmic pattern that can be broken down into the following steps:

We extend the lists horizontalCuts and verticalCuts to include the boundary cuts at the starting and ending of the cake, which

are 0 and h for the horizontal cuts and 0 and w for the vertical cuts. This ensures that we consider the entire cake, from the first

Sorting is efficiently done using the built-in sort function in Python, which typically has a time complexity of O(n log n), where n is

maximal piece area, while also keeping in mind to return the result under modular arithmetic to handle very large numbers.

Find the Maximum Gaps

the number of elements in the list.

Calculate and Return the Maximum Area

Sort the Cut Lists

cut to the very last one, including the edges of the cake.

Extend the Cut Lists

Sorting the lists is a critical step because it orders the cuts, which is necessary for calculating the maximum gaps between cuts.

To find the maximum gaps, we iterate through the sorted lists of cuts using the pairwise function, which gives us each pair of adjacent elements. For each adjacent pair (a, b), we calculate the difference b - a to determine the gap between them. We are

piece. The solution multiplies these maximum gaps: x * y. **Apply Modulo Operation**

Since the numbers we're dealing with can be very large, we apply a modulo operation to the result, % (10**9 + 7). This is to

By integrating these steps, the solution effectively navigates through the data to find the size of the largest piece post-cuts. The

careful extension, sorting, gap calculation, and result formatting make up the core components of this approach. With such

organization, the problem that initially can appear complex is broken down into simpler, sequential actions that lead to the

ensure the final output stays within integer limits and is consistent with the constraints specified in the problem.

The maximum horizontal gap (x) and maximum vertical gap (y) are multiplied to find the area of the resulting maximum cake

interested in the maximum gap from each list as this gap will determine the dimensions of the largest possible piece of the cake.

Example Walkthrough

After this step, our cut lists become:

horizontalCuts = [0, 1, 2, 5] verticalCuts = [0, 1, 4]

Now, we'll use the pairwise approach to get the differences:

Step 4: Calculate and Return the Maximum Area

In the horizontal direction: (1-0), (2-1), (5-2). The largest gap is 5-2=3.

In the vertical direction: (1-0), (4-1). The largest gap is 4-1=3.

So, the maximum area of a single piece of cake after making the cuts is 9.

Sort the cuts to calculate the maximum gaps between them

Find the maximum horizontal gap after performing all cuts

Find the maximum vertical gap after performing all cuts

max_horizontal_gap = max(b - a for a, b in pairwise(horizontal_cuts))

max_vertical_gap = max(b - a for a, b in pairwise(vertical_cuts))

and verticalCuts = [1].

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

desired outcome.

Step 1: Extend the Cut Lists We add the edges of the cake to our list of cuts. This means adding 0 and 5 to horizontalCuts, and 0 and 4 to verticalCuts.

Suppose we have a cake with dimensions height = 5 (h) and width = 4 (w). We also have the lists horizontalCuts = [1, 2]

Step 2: Sort the Cut Lists Our lists are already sorted as we extended the lists with the edges in the correct order. **Step 3: Find the Maximum Gaps**

As per the problem description, we apply the modulo operation to the result, 9 % (10***9 + 7). Since 9 is not larger than 10***9

We calculate the maximum area by multiplying the maximum gaps found in step 3. Therefore, the area is 3 (horizontal gap) * 3(vertical gap) = 9.

Step 5: Apply Modulo Operation

+ 7, the result remains 9.

Solution Implementation

from itertools import pairwise

horizontal cuts.extend([0, height])

vertical_cuts.extend([0, width])

horizontal cuts.sort()

vertical_cuts.sort()

from typing import List

Python

class Solution: def maxArea(self, height: int, width: int, horizontal cuts: List[int], vertical_cuts: List[int]) -> int: # Add the edges of the rectangle to the list of cuts

Compute the maximum area of a piece and modulo it with $(10^9 + 7)$ for the result

Arrays.sort(horizontalCuts); Arrays.sort(verticalCuts);

// Find the maximum distance between two horizontal cuts

// Find the maximum distance between two vertical cuts

// Calculate the largest possible area of a cake piece and take the modulo

long maxArea = (maxHorizontalDistance * maxVerticalDistance) % MODULO;

int horizontalCutsCount = horizontalCuts.length;

for (int i = 1; i < horizontalCutsCount; ++i) {</pre>

for (int i = 1; i < verticalCutsCount; ++i) {</pre>

// Return the maximum area as integer

return (int) maxArea;

int verticalCutsCount = verticalCuts.length;

 $max_area = (max_horizontal_gap * max_vertical_gap) % (10**9 + 7)$ return max_area Java class Solution { public int maxArea(int height, int width, int[] horizontalCuts, int[] verticalCuts) { // Define the modulo constant for the case when the result is very large final int MODULO = (int) 1e9 + 7;// Sort the arrays of cuts to facilitate the calculation of maximum sections // Store the length of the arrays to avoid recalculating

// Calculate the maximum distance between the first horizontal cut or edge and the last one or edge

// Calculate the maximum distance between the first vertical cut or edge and the last one or edge

long maxVerticalDistance = Math.max(verticalCuts[0], width - verticalCuts[verticalCutsCount - 1]);

maxVerticalDistance = Math.max(maxVerticalDistance, verticalCuts[i] - verticalCuts[i - 1]);

long maxHorizontalDistance = Math.max(horizontalCuts[0], height - horizontalCuts[horizontalCutsCount - 1]);

maxHorizontalDistance = Math.max(maxHorizontalDistance, horizontalCuts[i] - horizontalCuts[i - 1]);

C++ #include <vector> #include <algorithm>

public:

class Solution {

```
// Function to find the maximum area of a piece of cake after horizontal and vertical cuts
    int maxArea(int height, int width, std::vector<int>& horizontalCuts, std::vector<int>& verticalCuts) {
        // Add border cuts for horizontal and vertical cuts
        horizontalCuts.push back(0);
        horizontalCuts.push back(height);
        verticalCuts.push back(0);
        verticalCuts.push_back(width);
        // Sort the vectors for horizontal and vertical cuts
        std::sort(horizontalCuts.begin(), horizontalCuts.end());
        std::sort(verticalCuts.begin(), verticalCuts.end());
        // Initialize maximum height and width to 0
        int maxHeight = 0, maxWidth = 0;
        // Calculate the maximum height segment after the cuts
        for (int i = 1; i < horizontalCuts.size(); ++i) {</pre>
            maxHeight = std::max(maxHeight, horizontalCuts[i] - horizontalCuts[i - 1]);
        // Calculate the maximum width segment after the cuts
        for (int i = 1; i < verticalCuts.size(); ++i) {</pre>
            maxWidth = std::max(maxWidth, verticalCuts[i] - verticalCuts[i - 1]);
        // Modulo to prevent integer overflow; 10^9 + 7 is a large prime number
        const int mod = 1e9 + 7;
        // Cast to long long to prevent integer overflow during multiplication
        // Then calculate the maximum area of the piece of cake and apply modulo
        return static_cast<long long>(maxHeight) * maxWidth % mod;
};
TypeScript
function maxArea(height: number, width: number, horizontalCuts: number[], verticalCuts: number[]): number {
    // Define the modulo value to handle large numbers.
    const MODULO = 1e9 + 7;
    // Add the borders of the chocolate to the horizontal and vertical cuts.
    horizontalCuts.push(0, height);
    verticalCuts.push(0, width);
    // Sort the arravs to facilitate calculation of maximum gaps.
    horizontalCuts.sort((a, b) => a - b);
```

Find the maximum vertical gap after performing all cuts max_vertical_gap = max(b - a for a, b in pairwise(vertical_cuts)) # Compute the maximum area of a piece and modulo it with $(10^9 + 7)$ for the result

Time and Space Complexity

return max_area

horizontal cuts.sort()

vertical_cuts.sort()

verticalCuts.sort((a, b) => a - b);

let maxWidth = 0;

// Example usage:

class Solution:

from itertools import pairwise

from typing import List

let maxHeight = 0;

// Initialize variables to store the maximum width and height.

// Find the maximum height gap between two horizontal cuts.

// Find the maximum width gap between two vertical cuts.

maxHeight = Math.max(maxHeight, horizontalCuts[i] - horizontalCuts[i - 1]);

maxWidth = Math.max(maxWidth, verticalCuts[i] - verticalCuts[i - 1]);

return Number((BigInt(maxHeight) * BigInt(maxWidth)) % BigInt(MODULO));

// console.log(maxArea(5, 4, [1, 2, 4], [1, 3])); // Expected output: 4

Add the edges of the rectangle to the list of cuts

Sort the cuts to calculate the maximum gaps between them

Find the maximum horizontal gap after performing all cuts

max_horizontal_gap = max(b - a for a, b in pairwise(horizontal_cuts))

 $max_area = (max_horizontal_gap * max_vertical_gap) % (10**9 + 7)$

horizontal cuts.extend([0, height])

vertical_cuts.extend([0, width])

// Calculate the maximum area, convert the result to BigInt and apply the modulo.

for (let i = 1; i < horizontalCuts.length; i++) {</pre>

for (let i = 1; i < verticalCuts.length; i++) {</pre>

The time complexity of the provided code is determined by the sorting of the horizontalCuts and verticalCuts lists and the pairwise iterations through the sorted lists.

 Sorting the verticalCuts list takes O(n log n) time, where n is the number of vertical cuts. • The pairwise operation and the calculation of maximum differences for horizontal and vertical cuts are 0(m) for horizontal cuts and 0(n) for vertical cuts since each list is traversed once.

Extending the lists with [0, h] and [0, w] takes 0(1) time since it's adding a constant number of elements to the lists.

def maxArea(self, height: int, width: int, horizontal cuts: List[int], vertical_cuts: List[int]) -> int:

The overall time complexity is the sum of these, hence $0(m \log m + n \log n)$.

by the sorting algorithm (typically a version of quicksort or mergesort used in Python's sort function).

• The list slices and pairs generated by pairwise are iterators and only require constant space, 0(1).

• Sorting the horizontalCuts list takes 0(m log m) time, where m is the number of horizontal cuts.

pairwise function. • The space required for the sort function can typically be O(log m) for horizontalCuts and O(log n) for verticalCuts due to the space used

The space complexity is determined by the additional space required for sorting the cuts and the space needed for the output of

Considering the additional constant space needed to store the input list extensions and the pairs generated by pairwise, the overall space complexity remains $0(\log m + \log n)$.