# Question

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of the line i is at (i, ai) and (i, 0). Find two lines, which, together with the x-axis forms a container, such that the container contains the most water.

**Notice** that you may not slant the container.

**Example 1:**



**Input:** height = [1,8,6,2,5,4,8,3,7]

**Output:** 49

**Explanation:** The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

**Example 2:**

**Input:** height = [1,1]

**Output:** 1

**Example 3:**

**Input:** height = [4,3,2,1,4]

**Output:** 16

**Example 4:**

**Input:** height = [1,2,1]

**Output:** 2

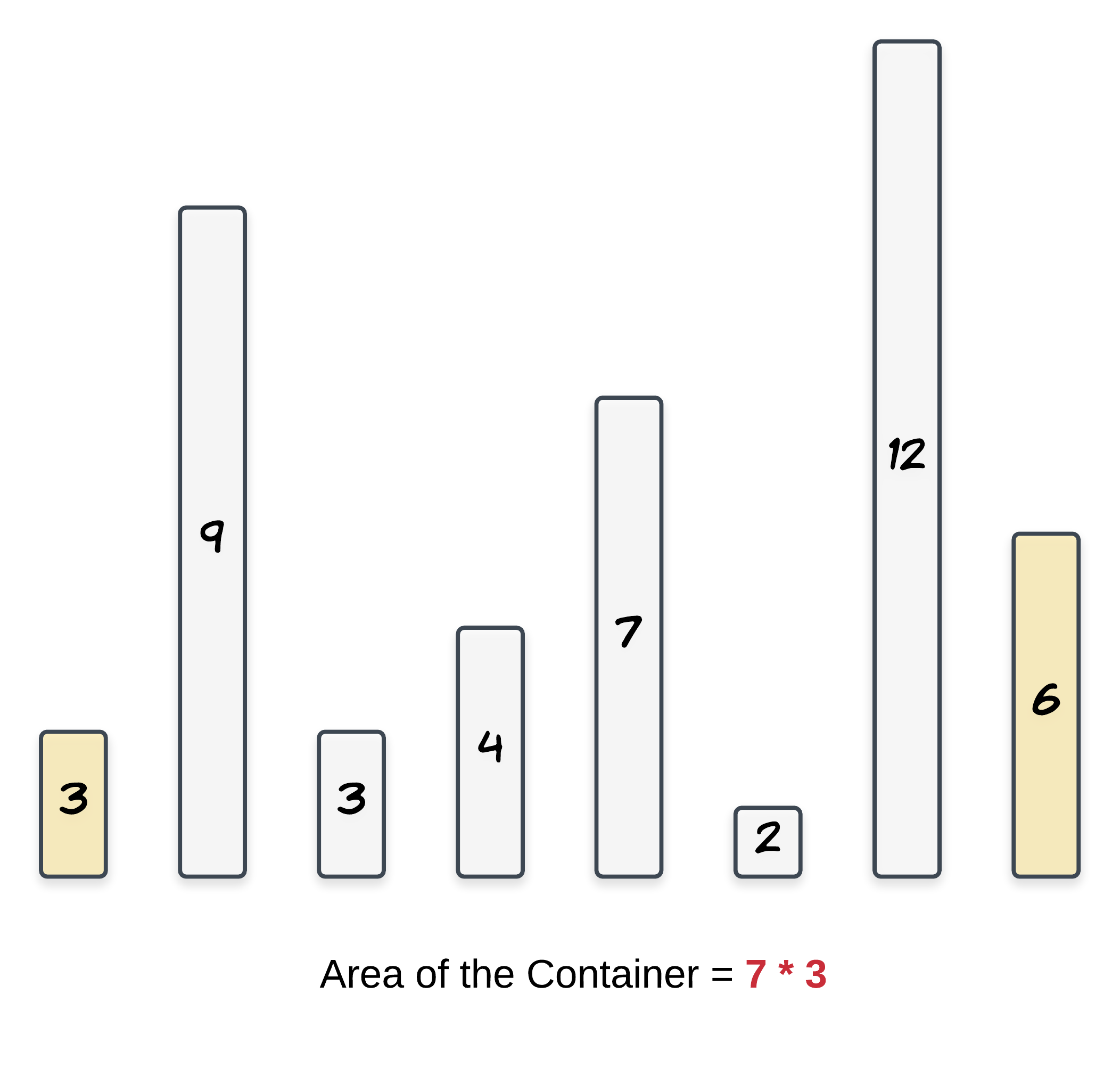
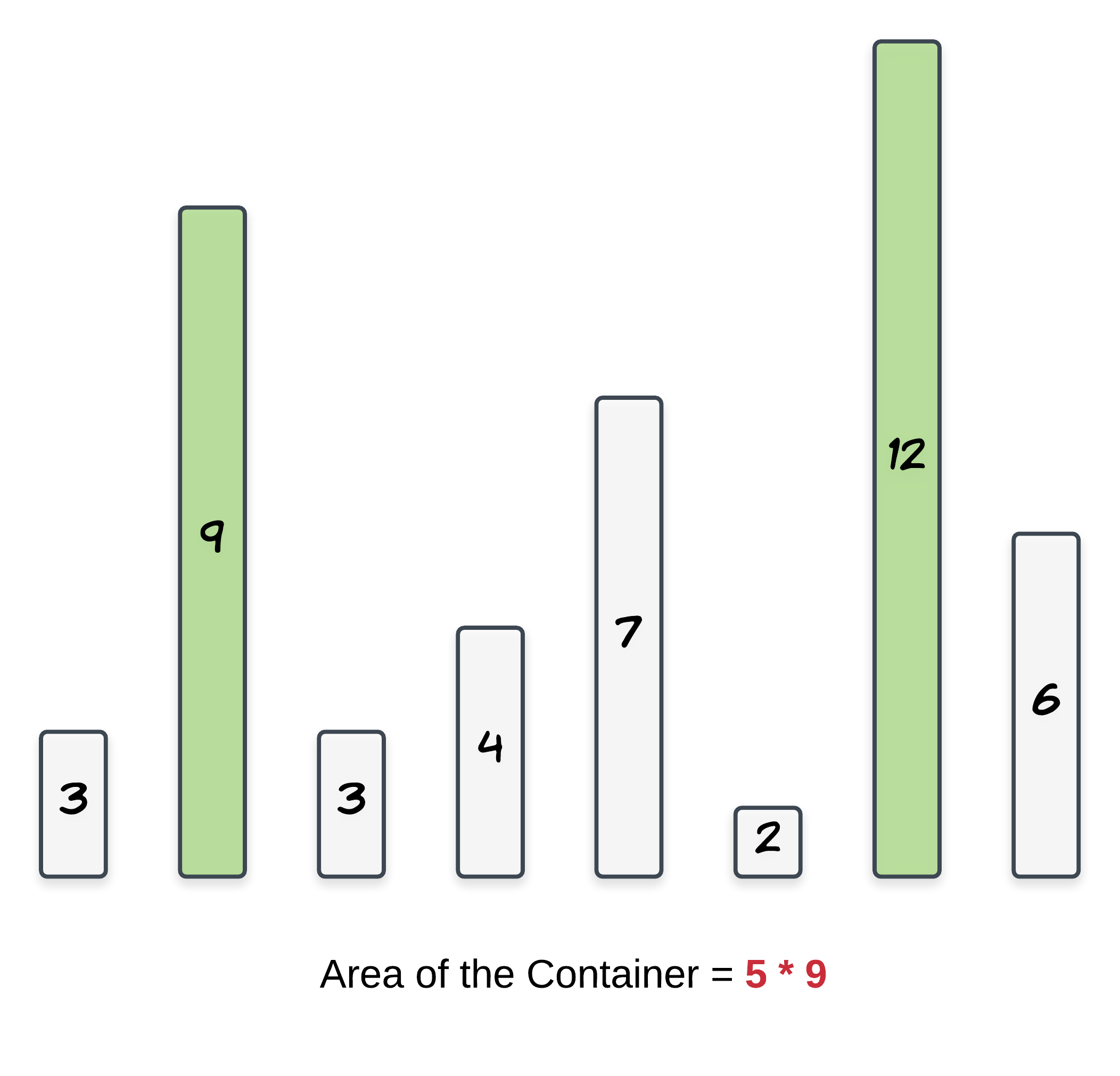
**Constraints:**

* n == height.length
* 2 <= n <= 3 \* 104
* 0 <= height[i] <= 3 \* 104

   Hide Hint #1

The aim is to maximize the area formed between the vertical lines. The area of any container is calculated using the shorter line as length and the distance between the lines as the width of the rectangle.

Area = length of shorter vertical line \* distance between lines

We can definitely get the maximum width container as the outermost lines have the maximum distance between them. However, this container **might not be the maximum in size** as one of the vertical lines of this container could be really short.  
  


  Hide Hint #2

Start with the maximum width container and go to a shorter width container if there is a vertical line longer than the current containers shorter line. This way we are compromising on the width but we are looking forward to a longer length container.

# Solution

## **Summary**

We have to maximize the Area that can be formed between the vertical lines using the shorter line as length and the distance between the lines as the width of the rectangle forming the area.

## **Solution**

#### **Approach 1: Brute Force**

**Algorithm**

In this case, we will simply consider the area for every possible pair of the lines and find out the maximum area out of those.

|  |
| --- |
| public class Solution {  public int maxArea(int[] height) {  int maxarea = 0;  for (int i = 0; i < height.length; i++)  for (int j = i + 1; j < height.length; j++)  maxarea = Math.max(maxarea, Math.min(height[i], height[j]) \* (j - i));  return maxarea;  }  } |

**Complexity Analysis**

* Time complexity : O(n^2)*O*(*n*2). Calculating area for all \dfrac{n(n-1)}{2}2*n*(*n*−1)​ height pairs.
* Space complexity : O(1)*O*(1). Constant extra space is used.

#### **Approach 2: Two Pointer Approach**

**Algorithm**

The intuition behind this approach is that the area formed between the lines will always be limited by the height of the shorter line. Further, the farther the lines, the more will be the area obtained.

We take two pointers, one at the beginning and one at the end of the array constituting the length of the lines. Futher, we maintain a variable \text{maxarea}maxarea to store the maximum area obtained till now. At every step, we find out the area formed between them, update \text{maxarea}maxarea and move the pointer pointing to the shorter line towards the other end by one step.

The algorithm can be better understood by looking at the example below:

1 8 6 2 5 4 8 3 7

How this approach works?

Initially we consider the area constituting the exterior most lines. Now, to maximize the area, we need to consider the area between the lines of larger lengths. If we try to move the pointer at the longer line inwards, we won't gain any increase in area, since it is limited by the shorter line. But moving the shorter line's pointer could turn out to be beneficial, as per the same argument, despite the reduction in the width. This is done since a relatively longer line obtained by moving the shorter line's pointer might overcome the reduction in area caused by the width reduction.

For further clarification click [here](https://leetcode.com/problems/container-with-most-water/discuss/6099/yet-another-way-to-see-what-happens-in-the-on-algorithm) and for the proof click [here](https://leetcode.com/problems/container-with-most-water/discuss/6089/Anyone-who-has-a-O(N)-algorithm/7268).

|  |
| --- |
| public class Solution {  public int maxArea(int[] height) {  int maxarea = 0, l = 0, r = height.length - 1;  while (l < r) {  maxarea = Math.max(maxarea, Math.min(height[l], height[r]) \* (r - l));  if (height[l] < height[r])  l++;  else  r--;  }  return maxarea;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). Single pass.
* Space complexity : O(1)*O*(1). Constant space is used.