Let's call any (contiguous) subarray B (of A) a *mountain* if the following properties hold:

* B.length >= 3
* There exists some 0 < i < B.length - 1 such that B[0] < B[1] < ... B[i-1] < B[i] > B[i+1] > ... > B[B.length - 1]

Given an array A of integers, return the length of the longest *mountain*.

Return 0 if there is no mountain.

**Example 1:**

**Input:** [2,1,4,7,3,2,5]

**Output:** 5

**Explanation:** The largest mountain is [1,4,7,3,2] which has length 5.

**Example 2:**

**Input:** [2,2,2]

**Output:** 0

**Explanation:** There is no mountain.

**Note:**

1. 0 <= A.length <= 10000
2. 0 <= A[i] <= 10000
3. public int longestMountain(int[] A) {
4. int res = 0, up = 0, down = 0;
5. for (int i = 1; i < A.length; ++i) {
6. if (down > 0 && A[i - 1] < A[i] || A[i - 1] == A[i]) up = down = 0;
7. if (A[i - 1] < A[i]) up++;
8. if (A[i - 1] > A[i]) down++;
9. if (up > 0 && down > 0 && up + down + 1 > res) res = up + down + 1;
10. }
11. return res;
12. }

int longestMountain(vector<int> A)

{

int res = 0, up = 0, down = 0;

for (int i = 1; i < A.size(); ++i)

{

//If we are going down and current element is greater than prev MEANS last mountain ended

//And the current mountain began (Up will be updated below)

if(down>0 && A[i - 1] < A[i] || A[i - 1] == A[i])

up = down = 0;

//If current element is greater then previous then we are going up

//Else we are going down the mountain

A[i] > A[i -1] ? up++ : down++;

if (up>0 && down>0)

res = max(res, up + down + 1);

}

return res;

}



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

**Output:** 49

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.
* 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.

Input: nums = [1, 1, 2, 45, 46, 46], target = 47

Output: 2

Explanation:

1 + 46 = 47

2 + 45 = 47

**Example 2:**

Input: nums = [1, 1], target = 2

Output: 1

Explanation:

1 + 1 = 2

**Example 3:**

Input: nums = [1, 5, 1, 5], target = 6

Output: 1

Explanation:

[1, 5] and [5, 1] are considered the same.

public static int uniquePairs(int[] nums, int target){

Set<Integer> set = new HashSet<Integer>();

Set<Integer> seen = new HashSet<Integer>();

int count = 0;

for(int num : nums){

if(set.contains(target-num) && !seen.contains(num)){

count++;

seen.add(target-num);

seen.add(num);

}

else if(!set.contains(num)){

set.add(num);

}

}

return count;

}