# Question

In a row of trees, the i-th tree produces fruit with type tree[i].

You **start at any tree of your choice**, then repeatedly perform the following steps:

1. Add one piece of fruit from this tree to your baskets.  If you cannot, stop.
2. Move to the next tree to the right of the current tree.  If there is no tree to the right, stop.

Note that you do not have any choice after the initial choice of starting tree: you must perform step 1, then step 2, then back to step 1, then step 2, and so on until you stop.

You have two baskets, and each basket can carry any quantity of fruit, but you want each basket to only carry one type of fruit each.

What is the total amount of fruit you can collect with this procedure?

**Example 1:**

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

**Output:** 3

**Explanation:** We can collect [1,2,1].

**Example 2:**

**Input:** [0,1,2,2]

**Output:** 3

**Explanation:** We can collect [1,2,2].

If we started at the first tree, we would only collect [0, 1].

**Example 3:**

**Input:** [1,2,3,2,2]

**Output:** 4

**Explanation:** We can collect [2,3,2,2].

If we started at the first tree, we would only collect [1, 2].

**Example 4:**

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

**Output:** 5

**Explanation:** We can collect [1,2,1,1,2].

If we started at the first tree or the eighth tree, we would only collect 4 fruits.

**Note:**

1. 1 <= tree.length <= 40000
2. 0 <= tree[i] < tree.length

# Solution

#### **Approach 1: Scan Through Blocks**

**Intuition**

Equivalently, we want the longest subarray with at most two "types" (values of tree[i]).

Instead of considering each element individually, we can consider blocks of adjacent elements of the same type.

For example, instead of tree = [1, 1, 1, 1, 2, 2, 3, 3, 3], we can say this is blocks = [(1, weight = 4), (2, weight = 2), (3, weight = 3)].

Now say we brute forced, scanning from left to right. We'll have something like blocks = [1, \_2\_, 1, 2, 1, 2, \_1\_, 3, ...] (with various weights).

The key insight is that when we encounter a 3, we do not need to start from the second element 2 (marked \_2\_ for convenience); we can start from the first element (\_1\_) before the 3. This is because if we started two or more elements before, the sequence must have types 1 and 2, and that sequence is going to end at the 3, and thus be shorter than anything we've already considered.

Since every starting point (that is the left-most index of a block) was considered, this solution is correct.

**Algorithm**

As the notation and strategy around implementing this differs between Python and Java, please see the inline comments for more details.

Java

|  |
| --- |
| class Solution {  public int totalFruit(int[] tree) {  // We'll make a list of indexes for which a block starts.  List<Integer> blockLefts = new ArrayList();  // Add the left boundary of each block  for (int i = 0; i < tree.length; ++i)  if (i == 0 || tree[i-1] != tree[i])  blockLefts.add(i);  // Add tree.length as a sentinel for convenience  blockLefts.add(tree.length);  int ans = 0, i = 0;  search: while (true) {  // We'll start our scan at block[i].  // types : the different values of tree[i] seen  // weight : the total number of trees represented  // by blocks under consideration  Set<Integer> types = new HashSet();  int weight = 0;  // For each block from the i-th and going forward,  for (int j = i; j < blockLefts.size() - 1; ++j) {  // Add each block to consideration  types.add(tree[blockLefts.get(j)]);  weight += blockLefts.get(j+1) - blockLefts.get(j);  // If we have 3+ types, this is an illegal subarray  if (types.size() >= 3) {  i = j - 1;  continue search;  }  // If it is a legal subarray, record the answer  ans = Math.max(ans, weight);  }  break;  }  return ans;  }  } |

Python

|  |
| --- |
| class Solution(object):  def totalFruit(self, tree):  blocks = [(k, len(list(v)))  for k, v in itertools.groupby(tree)]  ans = i = 0  while i < len(blocks):  # We'll start our scan at block[i].  # types : the different values of tree[i] seen  # weight : the total number of trees represented  # by blocks under consideration  types, weight = set(), 0  # For each block from i and going forward,  for j in xrange(i, len(blocks)):  # Add each block to consideration  types.add(blocks[j][0])  weight += blocks[j][1]  # If we have 3 types, this is not a legal subarray  if len(types) >= 3:  i = j-1  break  ans = max(ans, weight)  # If we go to the last block, then stop  else:  break  return ans |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of tree.
* Space Complexity: O(N)*O*(*N*).

#### **Approach 2: Sliding Window**

**Intuition**

As in Approach 1, we want the longest subarray with at most two different "types" (values of tree[i]). Call these subarrays valid.

Say we consider all valid subarrays that end at index j. There must be one with the smallest possible starting index i: lets say opt(j) = i.

Now the key idea is that opt(j) is a monotone increasing function. This is because any subarray of a valid subarray is valid.

**Algorithm**

Let's perform a sliding window, keeping the loop invariant that i will be the smallest index for which [i, j] is a valid subarray.

We'll maintain count, the count of all the elements in the subarray. This allows us to quickly query whether there are 3 types in the subarray or

Java

|  |
| --- |
| class Solution {  public int totalFruit(int[] tree) {  int ans = 0, i = 0;  Counter count = new Counter();  for (int j = 0; j < tree.length; ++j) {  count.add(tree[j], 1);  while (count.size() >= 3) {  count.add(tree[i], -1);  if (count.get(tree[i]) == 0)  count.remove(tree[i]);  i++;  }  ans = Math.max(ans, j - i + 1);  }  return ans;  }  }  class Counter extends HashMap<Integer, Integer> {  public int get(int k) {  return containsKey(k) ? super.get(k) : 0;  }  public void add(int k, int v) {  put(k, get(k) + v);  }  } |

Python

|  |
| --- |
| class Solution(object):  def totalFruit(self, tree):  ans = i = 0  count = collections.Counter()  for j, x in enumerate(tree):  count[x] += 1  while len(count) >= 3:  count[tree[i]] -= 1  if count[tree[i]] == 0:  del count[tree[i]]  i += 1  ans = max(ans, j - i + 1)  return ans |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of tree.
* Space Complexity: O(N)*O*(*N*).