<https://leetcode.com/problems/fruit-into-baskets/>

You are visiting a farm that has a single row of fruit trees arranged from left to right. The trees are represented by an integer array fruits where fruits[i] is the type of fruit the ith tree produces.

You want to collect as much fruit as possible. However, the owner has some strict rules that you must follow:

You only have two baskets, and each basket can only hold a single type of fruit. There is no limit on the amount of fruit each basket can hold.

Starting from any tree of your choice, you must pick exactly one fruit from every tree (including the start tree) while moving to the right. The picked fruits must fit in one of your baskets.

Once you reach a tree with fruit that cannot fit in your baskets, you must stop.

Given the integer array fruits, return the maximum number of fruits you can pick.

**Example 1:**

Input: fruits = [1,2,1]

Output: 3

Explanation: We can pick from all 3 trees.

**Example 2:**

Input: fruits = [0,1,2,2]

Output: 3

Explanation: We can pick from trees [1,2,2].

If we had started at the first tree, we would only pick from trees [0,1].

**Example 3:**

Input: fruits = [1,2,3,2,2]

Output: 4

Explanation: We can pick from trees [2,3,2,2].

If we had started at the first tree, we would only pick from trees [1,2].

**Constraints:**

1 <= fruits.length <= 10^5

0 <= fruits[i] < fruits.length

**Attempt 1: 2022-09-10**

**Equal to find longest subarray contains at most two distinct number**

**Solution 1: Not fixed length Sliding Window (5 min, Map store the count (frequency) of each type of fruit)**

class Solution {

    // Equal to find longest subarray contains at most two distinct number

    public int totalFruit(int[] fruits) {

        Map<Integer, Integer> freq = new HashMap<Integer, Integer>();

        int i = 0;

        int len = fruits.length;

        int maxLen = 0;

        for(int j = 0; j < len; j++) {

            freq.put(fruits[j], freq.getOrDefault(fruits[j], 0) + 1);

            int size = freq.size();

            while(size > 2) {

                freq.put(fruits[i], freq.get(fruits[i]) - 1);

                if(freq.get(fruits[i]) == 0) {

                    freq.remove(fruits[i]);

                    size--;

                }

                i++;

            }

            maxLen = Math.max(maxLen, j - i + 1);

        }

        return maxLen;

    }

}

Space Complexity: O(n)

Time Complexity: O(n)

**Solution 2: Not fixed length Sliding Window (30 min, Map store the most recent index of each type of fruit)**

class Solution {

    // Equal to find longest subarray contains at most two distinct number

    public int totalFruit(int[] fruits) {

        Map<Integer, Integer> index\_map = new HashMap<Integer, Integer>();

        int i = 0;

        int len = fruits.length;

        int maxLen = 0;

        for(int j = 0; j < len; j++) {

            index\_map.put(fruits[j], j);

            int size = index\_map.size();

            if(size > 2) {

                // Shrink left end to skip all same value of fruits[i]

                // and reach to the very last one of fruits[i]

                // 'i' means current left boundary index, if it not equal

                // to most recent index of current type of fruit which

                // we want to remove, we have to keep forwarding 'i'

                // until match most recent index, physically it points

                // to the last current type fruit

// Test example input:

// {1, 1, 3, 3, 3, 2, 2, 2, 2, 4, 4, 4, 4}

                while(i != index\_map.get(fruits[i])) {

                    i++;

                }

                // Since our purpose is removing fruits[i] totally from

                // index map, we can remove it first then update i to

                // next position which point to a new value differ than

                // fruits[i], and no need 'size--', since we remove key

                // fruits[i] already, in next for loop iteration, size

                // will auto re-calculate

                index\_map.remove(fruits[i]);

                i++;

            }

            maxLen = Math.max(maxLen, j - i + 1);

        }

        return maxLen;

    }

}

Space Complexity: O(n)

Time Complexity: O(n)

**Refer to chatGPT**

**Key Differences:**

HashMap as an Index Map:

In this solution, the index\_map stores the most recent index of each fruit type instead of counting occurrences.

When more than two types of fruits are present, it uses these indices to identify and remove the least recent fruit type.

Shrink Logic:

The logic for shrinking the window is more nuanced. Instead of decrementing counts like the earlier solution, this version skips all contiguous duplicates of the leftmost fruit type (fruits[i]) in one step and removes it from the map.

**Advantages of This Solution:**

Efficient Shrinking:

By skipping all contiguous duplicates of the leftmost fruit in one step, it minimizes unnecessary iterations when many duplicates exist.

This approach avoids repeatedly decrementing the count of the leftmost fruit, which can reduce overhead in specific cases (e.g., when the input has long runs of the same fruit).

Simpler Map Update:

The use of indices (index\_map) allows direct removal of the leftmost fruit type without recalculating or tracking counts. This makes the logic simpler and avoids off-by-one errors.

**When to Prefer This Solution:**

When the input contains many consecutive duplicates of the same fruit type (e.g., [1, 1, 3, 3, 3, 2, 2, 2, 2, 4, 4, 4, 4]), this solution can be slightly faster due to its efficient shrinking logic.

If you prefer a more concise shrinking operation without explicitly tracking counts.

**Which Solution to Choose:**

Previous Solution: Generally easier to understand and sufficient for most use cases. Prefer it for clarity, as the count-based logic is more intuitive.

This Solution: Better when working with inputs where duplicates are common, or if you favor using indices for direct removal and prefer skipping contiguous duplicates efficiently.

**Refer to**

[L340.P2.3.Longest Substring with At Most K Distinct Characters (Ref.L424)](note://4E7F61484E324A0D83B155FFCC1A1D0B)