

Fruits Into Baskets

Longest subarray with at most $K=2$ distinct values ■■

● MEDIUM

■ Hook — The Story

You're walking through a fruit farm with exactly **2 baskets**. Each basket can hold only one type of fruit, but unlimited quantity. Starting from any tree, you want to collect the maximum number of fruits. The catch: once you pick up a third type, you have to stop and count your haul. This is *exactly* the sliding window problem underneath: **longest subarray with at most 2 distinct values**.

■ Generalization

This problem is a special case of 'Longest Subarray with at most K Distinct Elements' with $K=2$. Master this and you immediately understand the K-distinct family of problems.

■ Core Intuition

Use a HashMap to count frequency of each fruit type in the current window. When the map has more than 2 keys, shrink the window from the left — decrement frequency and remove the key when it hits 0. At every step, track the window size.

■ Visual — Fruits: [1,2,1,2,3]

```
Fruits: [1, 2, 1, 2, 3]
         0  1  2  3  4

right=0: add 1 → {1:1}           window=[1]       len=1
right=1: add 2 → {1:1,2:1}       window=[1,2]      len=2
right=2: add 1 → {1:2,2:1}       window=[1,2,1]   len=3
right=3: add 2 → {1:2,2:2}       window=[1,2,1,2] len=4 ★
right=4: add 3 → {1:2,2:2,3:1} → 3 keys! SHRINK:
    remove fruits[left=0]=1 → {1:1,2:2,3:1} left=1
    still 3 keys! remove fruits[left=1]=2 → {1:1,2:1,3:1} left=2
    still 3 keys! remove fruits[left=2]=1 → {2:1,3:1}      left=3
    now 2 keys ✓  window=[2,3]   len=2
```

★ ANSWER = 4

■ Code Implementation (Java)

```
import java.util.HashMap;

public class FruitsIntoBaskets {

    public int totalFruit(int[] fruits) {
        HashMap<Integer, Integer> basket = new HashMap<>();
        int left = 0, maxFruits = 0;

        for (int right = 0; right < fruits.length; right++) {
            // Add right fruit to basket
            basket.merge(fruits[right], 1, Integer::sum);

            // More than 2 types? shrink from left
            while (basket.size() > 2) {
                basket.merge(fruits[left], -1, Integer::sum);
                if (basket.get(fruits[left]) == 0)
                    basket.remove(fruits[left]);
                left++;
            }

            maxFruits = Math.max(maxFruits, right - left + 1);
        }

        return maxFruits;
    }
}
```

■■ Time & Space Complexity

■ TIME COMPLEXITY

$O(n)$

■ SPACE COMPLEXITY

$O(K) = O(1)$

Each element enters and leaves the window at most once. Map holds at most $K+1$ keys during shrinking. For $K=2$, that's $O(3) = O(1)$ space.

■■ Common Mistakes

■■ Using if instead of while for shrinking

When you add a 3rd type, you might need to remove multiple elements from the left to get back to 2 types. An if only removes one — use `while(basket.size() > 2)` to keep shrinking until valid.

■ Practice Problems

#	Problem	Difficulty	Pattern
1	Fruit Into Baskets (LeetCode 904)	Medium	SW + HashMap
2	Longest Substring with At Most K Distinct Chars	Medium	SW + HashMap
3	Longest Substring with At Most 2 Distinct Chars	Medium	SW + HashMap
4	Max Consecutive Ones III (K flips)	Medium	SW + Counter
5	Subarrays with K Different Integers	Hard	SW + AtMost trick

■ Quick Revision

- ✓ HashMap maps fruit_type → count_in_window
- ✓ Expand right freely, shrink left when map.size() > K
- ✓ Remove key from map when its count hits 0
- ✓ Answer = max(right - left + 1)
- ✓ Generalizes to K distinct types (change 2 → K)
- ✓ Time: O(n) | Space: O(K)

■ MENTAL MODEL — Remember This Forever

Picture a sliding food tray with exactly K slots for fruit types. Keep adding fruits from the right. The moment a new type needs a slot but all K are taken, push the left side forward until one type falls off completely — freeing a slot. The tray's length at its widest point is your answer.