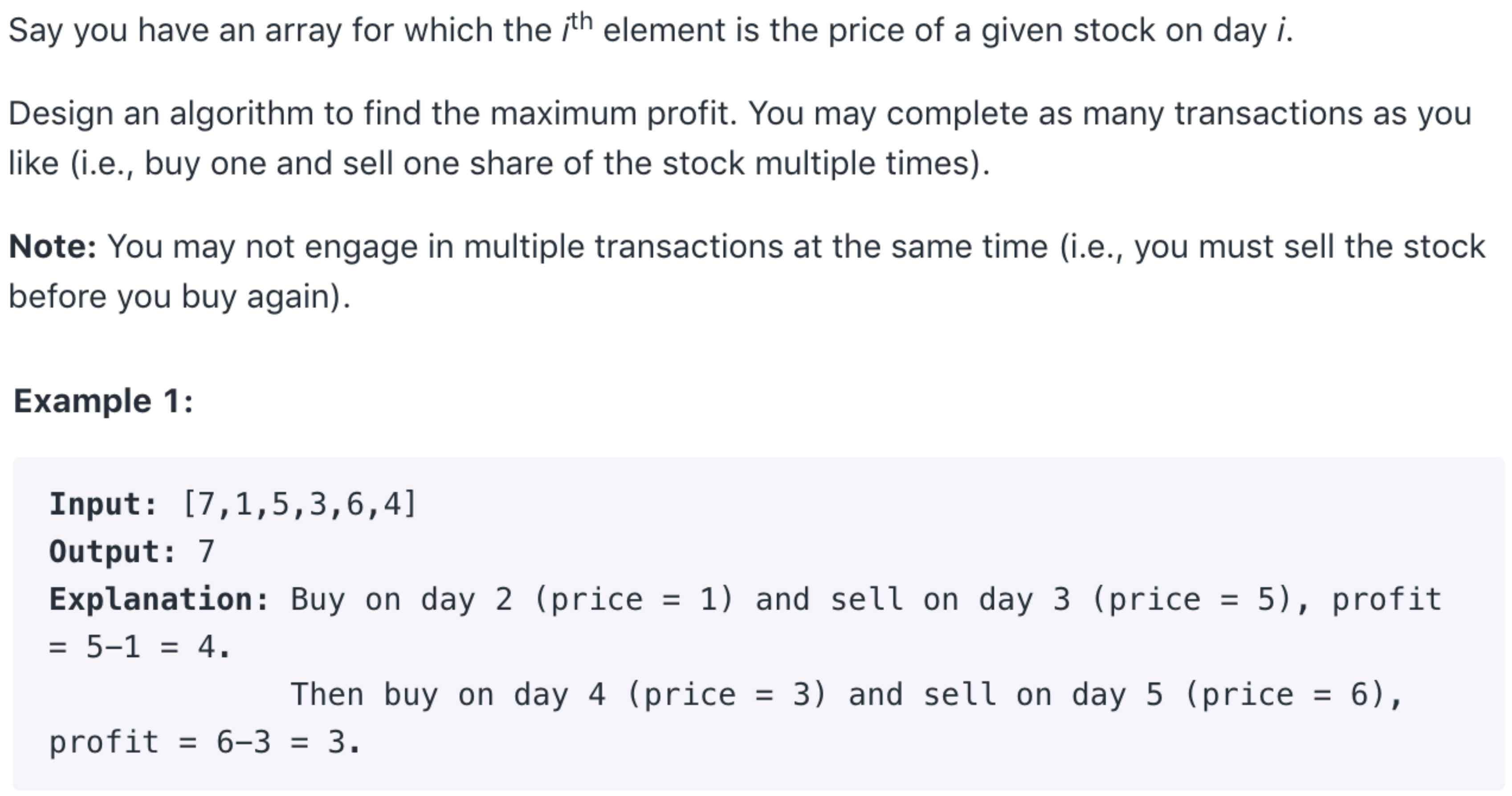
CS 180 Discussion

Outline

- Greedy related questions
  - Best Time to Buy and Sell Stock II (lc 122)
  - Best Time to Buy and Sell Stock (lc 121)
  - Majority Element on GeeksforGeeks
  - Split Array into Consecutive Subsequences (lc 659)

Best Time to Buy and Sell Stock II



Say you have an array for which the *i*th element is the price of a given stock on day *i*.

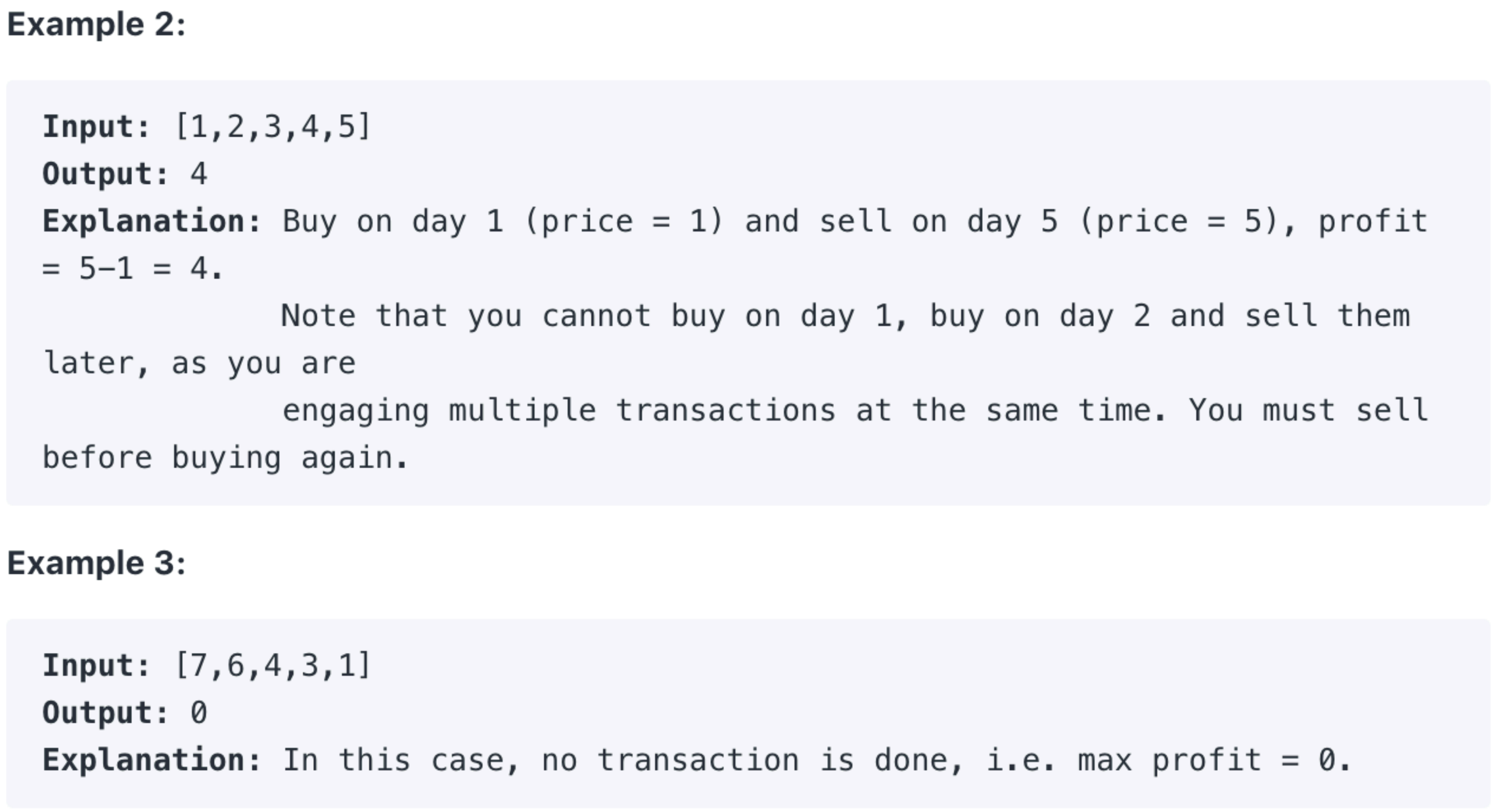
Design an algorithm to find the maximum profit. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit
= 5-1 = 4.
             Then buy on day 4 (price = 3) and sell on day 5 (price = 6),
profit = 6-3 = 3.
```

Best Time to Buy and Sell Stock II



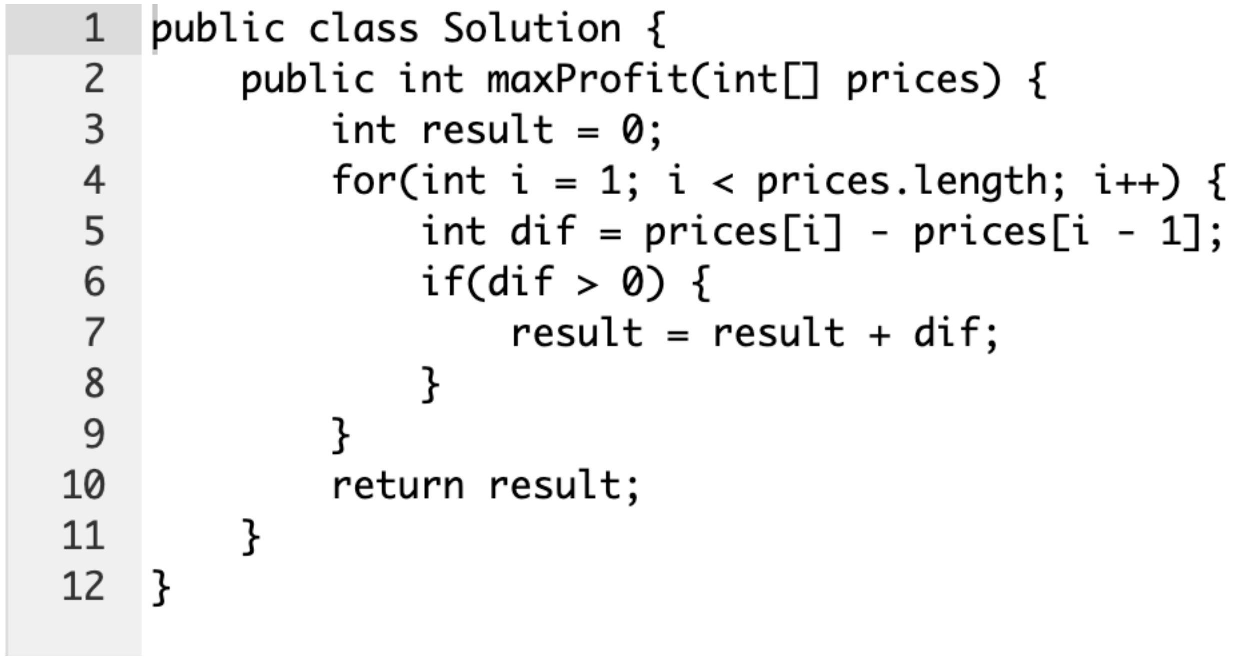
Example 2:

```
Input: [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit
= 5-1 = 4.
             Note that you cannot buy on day 1, buy on day 2 and sell them
later, as you are
             engaging multiple transactions at the same time. You must sell
before buying again.
```

Example 3:

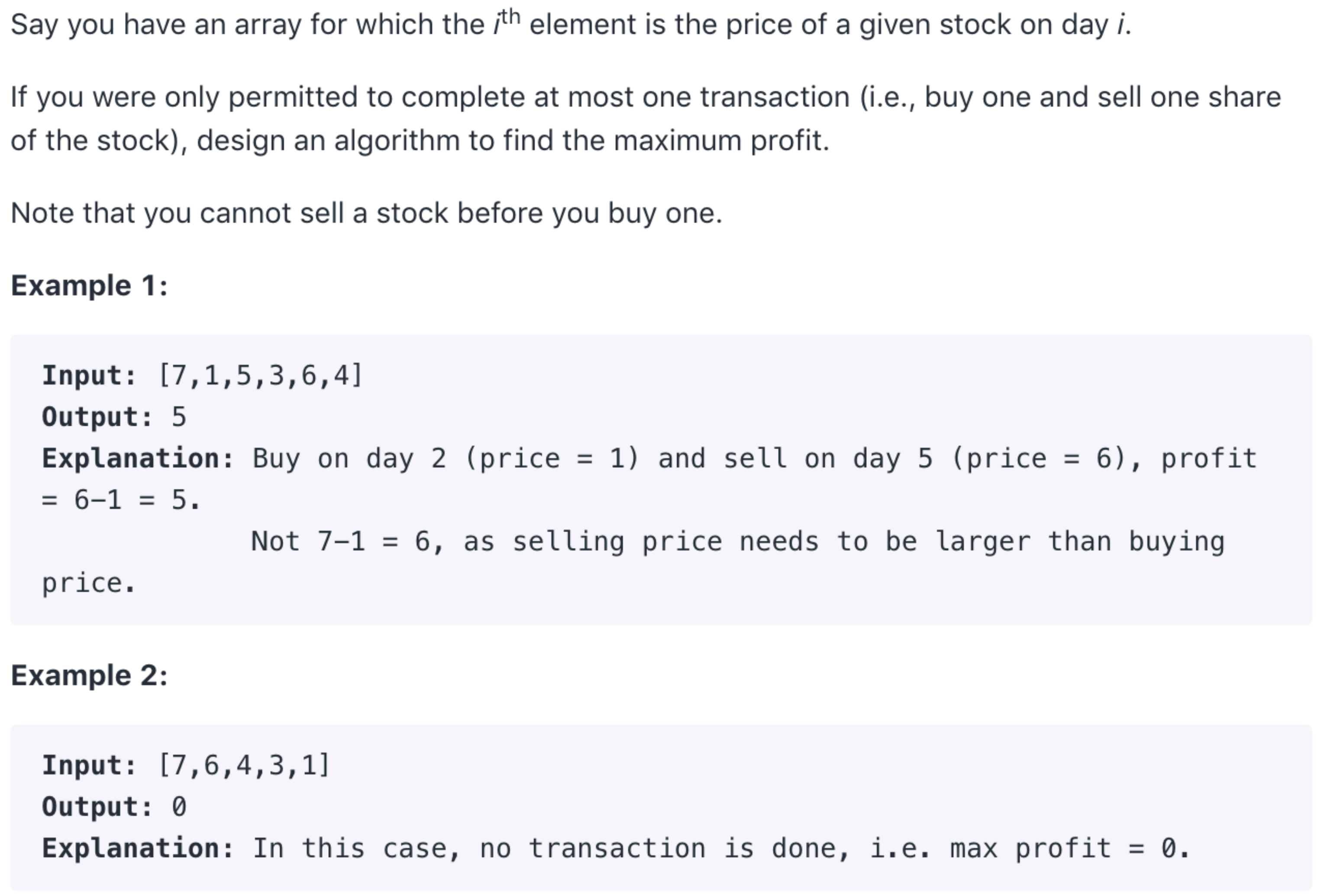
```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

Best Time to Buy and Sell Stock II



```java
public class Solution {
    public int maxProfit(int[] prices) {
        int result = 0;
        for(int i = 1; i < prices.length; i++) {
            int dif = prices[i] - prices[i - 1];
            if(dif > 0) {
                result = result + dif;
            }
        }
        return result;
    }
}
```

Best Time to Buy and Sell Stock



Say you have an array for which the i^{th} element is the price of a given stock on day *i*.

If you were only permitted to complete at most one transaction (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Note that you cannot sell a stock before you buy one.

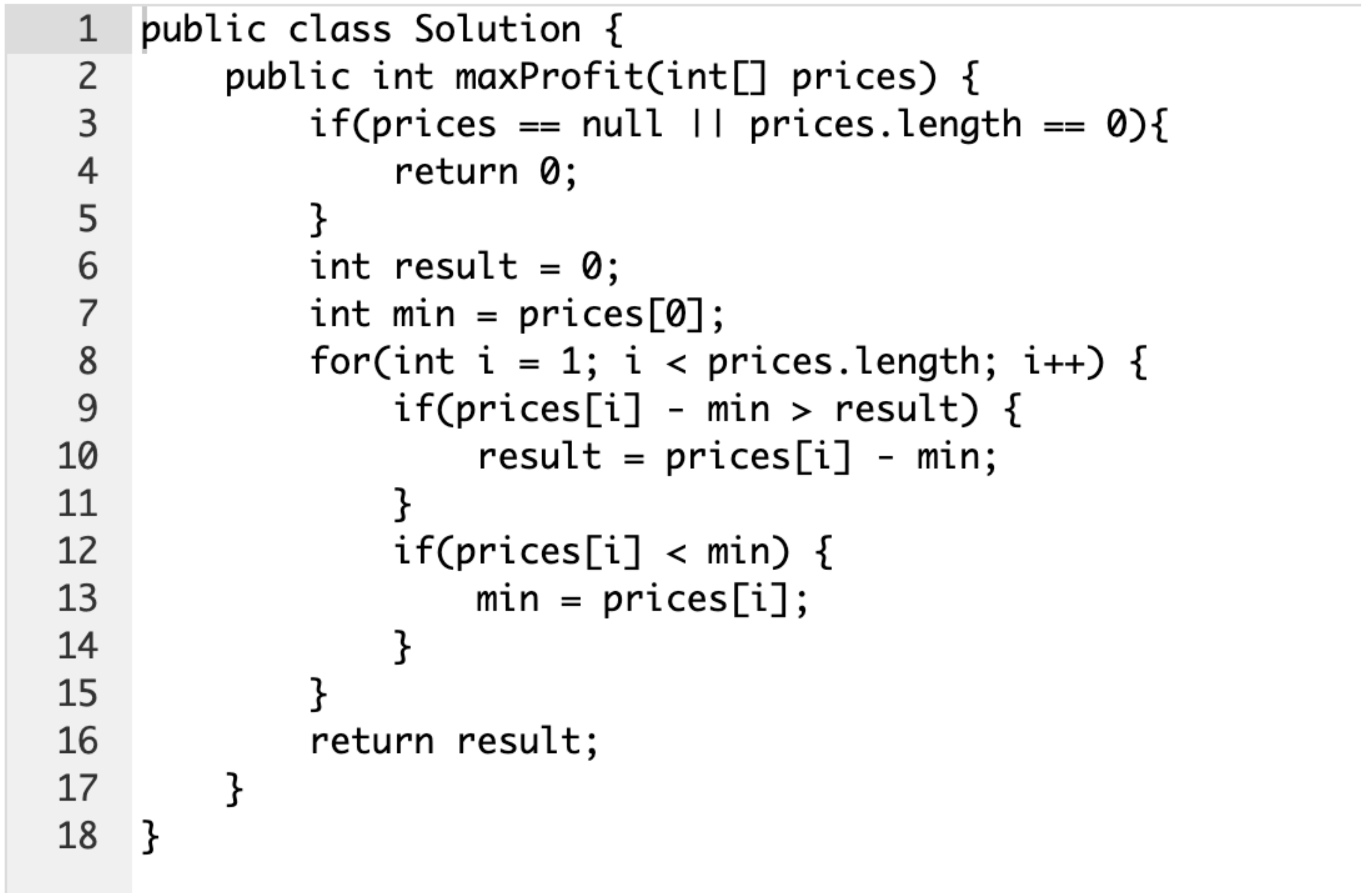
Example 1:

```
Input: [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit
= 6-1 = 5.
             Not 7-1 = 6, as selling price needs to be larger than buying
price.
```

Example 2:

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

Best Time to Buy and Sell Stock



```java
public class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length == 0){
            return 0;
        }
        int result = 0;
        int min = prices[0];
        for(int i = 1; i < prices.length; i++) {
            if(prices[i] - min > result) {
                result = prices[i] - min;
            }
            if(prices[i] < min) {
                min = prices[i];
            }
        }
        return result;
    }
}
```

Majority Element

Write a function which takes an array and prints the majority element (if it exists), otherwise prints “No Majority Element”. A ***majority element*** in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

Examples :

```
Input : {3, 3, 4, 2, 4, 4, 2, 4, 4}
Output : 4

Input : {3, 3, 4, 2, 4, 4, 2, 4}
Output : No Majority Element
```

Majority Element

1. Finding a Candidate :

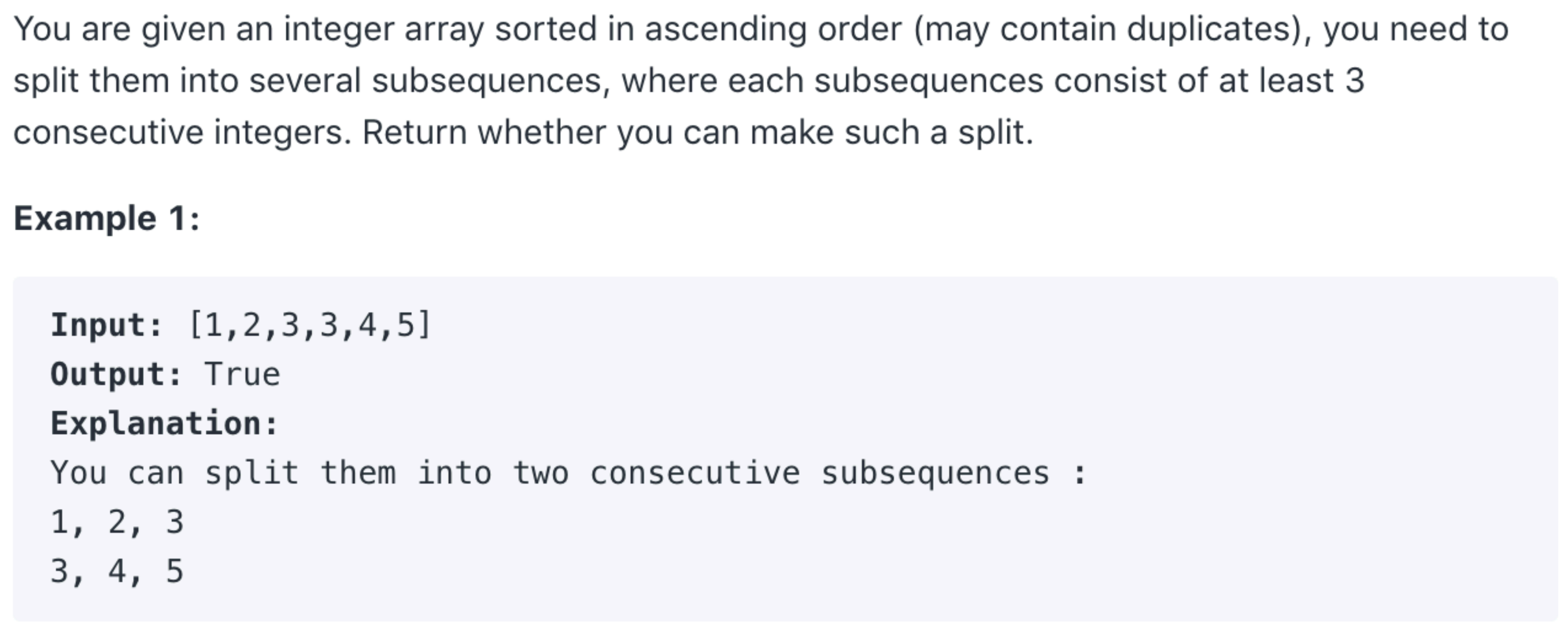
The algorithm for first phase that works in O(n) is known as Moore's Voting Algorithm. Basic idea of the algorithm is that if we cancel out each occurrence of an element *e* with all the other elements that are different from *e* then *e* will exist till end if it is a majority element.

```
findCandidate(a[], size)
1.  Initialize index and count of majority element
     maj_index = 0, count = 1
2.  Loop for i = 1 to size – 1
    (a) If a[maj_index] == a[i]
          count++
    (b) Else
        count--;
    (c) If count == 0
          maj_index = i;
          count = 1
3.  Return a[maj_index]
```

2. Check if the element obtained in step 1 is majority element or not :

```
printMajority (a[], size)
1.  Find the candidate for majority
2.  If candidate is majority. i.e., appears more than n/2 times.
       Print the candidate
3.  Else
       Print "No Majority Element"
```

Split Array into Consecutive subsequences

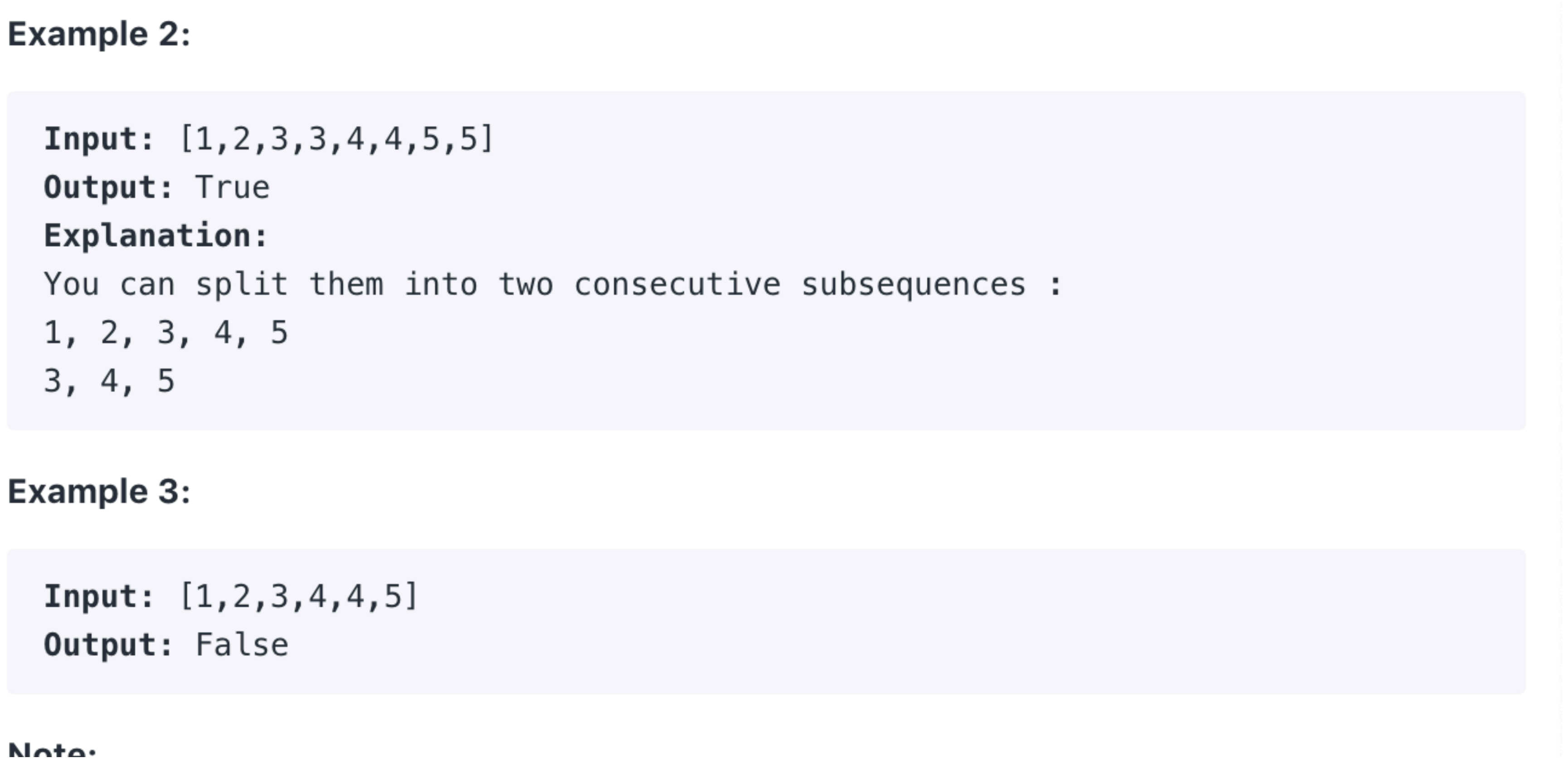


You are given an integer array sorted in ascending order (may contain duplicates), you need to split them into several subsequences, where each subsequences consist of at least 3 consecutive integers. Return whether you can make such a split.

Example 1:

```
Input: [1,2,3,3,4,5]
Output: True
Explanation:
You can split them into two consecutive subsequences :
1, 2, 3
3, 4, 5
```

Split Array into Consecutive subsequences



Example 2:

```
Input: [1,2,3,3,4,4,5,5]
Output: True
Explanation:
You can split them into two consecutive subsequences :
1, 2, 3, 4, 5
3, 4, 5
```

Example 3:

```
Input: [1,2,3,4,4,5]
Output: False
```

Note:

Split Array into Consecutive subsequences

```
class Solution {
public:
    bool isPossible(vector<int>& nums) {
        unordered_map<int, int> freq, need;
        for (int num : nums) ++freq[num];
        for (int num : nums) {
            if (freq[num] == 0) continue;
            else if (need[num] > 0) {
                --need[num];
                ++need[num + 1];
            } else if (freq[num + 1] > 0 && freq[num + 2] > 0) {
                --freq[num + 1];
                --freq[num + 2];
                ++need[num + 3];
            } else return false;
            --freq[num];
        }
        return true;
    }
};
```