BITTIGER

CLASS_7

ADVANCED LINEAR ALGORITHM

Ben Gong

Content of Class_7

26. Remove Duplicates from Sorted Array
80. Remove Duplicates from Sorted Array II
3. Longest Substring Without Repeating Characters
159. Longest Substring with At Most Two Distinct Characters
340. Longest Substring with At Most K Distinct Characters
41. First Missing Positive
239. Sliding Window Maximum
42. Trapping Rain Water

26. Remove Duplicates from Sorted Array

Given a sorted array, remove the duplicates in place such that each element appear only *once* and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

For example,

Given input array *nums* = `[1,1,2]`,

Your function should return length = `2`, with the first two elements of *nums* being `1` and `2` respectively. It doesn't matter what you leave beyond the new length.

- Total Accep
- Total Submi
- Difficulty: Ea
- Contributor:

10 min

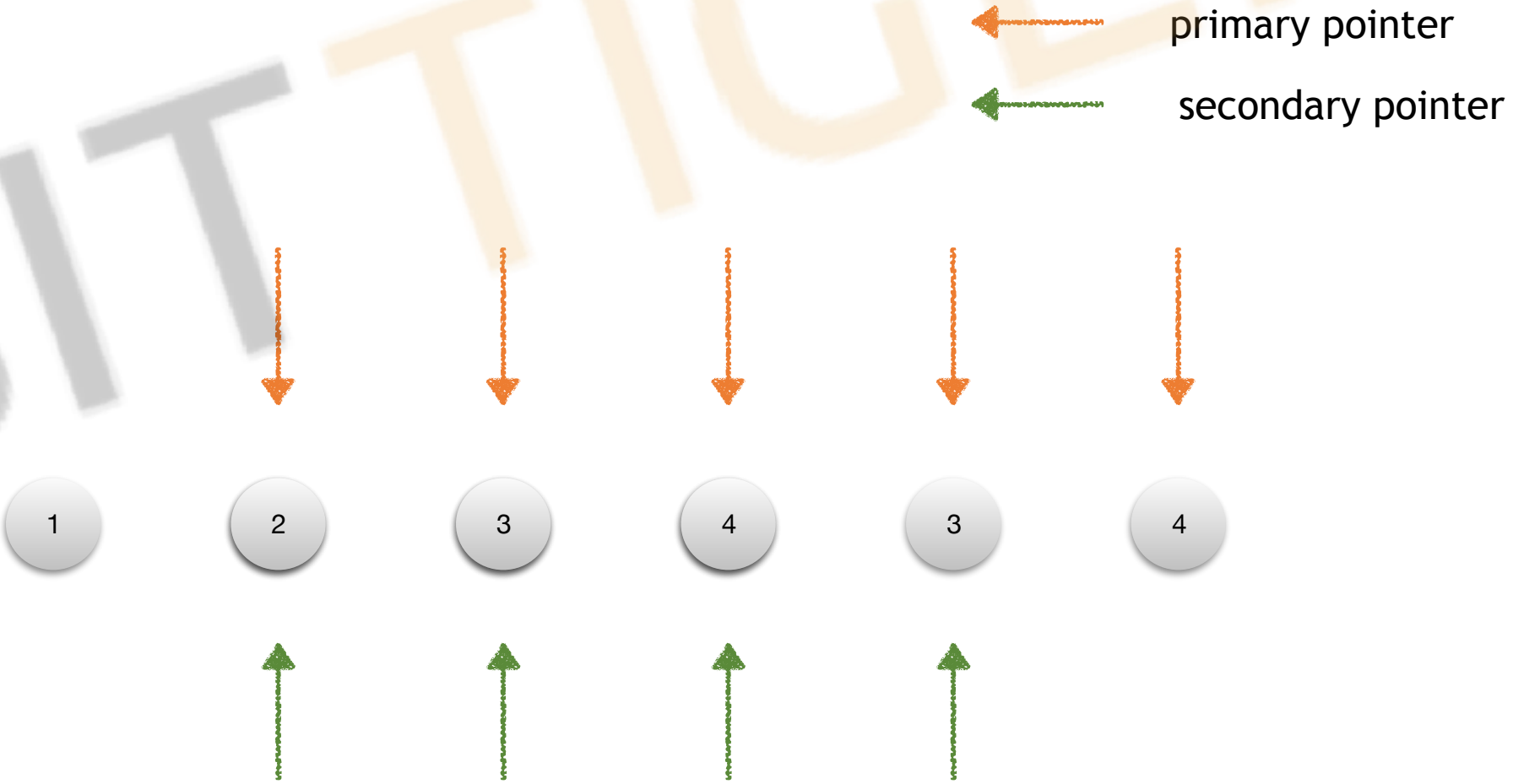
```
public int removeDuplicates(int[] nums) {

}
```

三种 two pointers 题目 （追及， 相遇， 相离）

追及问题

primary pointer —-> 主动pointer 遍历

secondary pointer ——>从动pointer 记录


primary pointer
secondary pointer
1
2
3
4
3
4

```java
public int removeDuplicates(int[] nums) {
    if(nums == null || nums.length == 0){
        return 0;
    }
    if(nums.length == 1){
        return 1;
    }
    int beg = 0;

    for(int i = 0; i < nums.length; i++){
        if(beg < 1 || nums[i] > nums[beg - 1]){
            nums[beg] = nums[i];
            beg++;
        }
    }

    return beg;
}
```

compare with secondary pointer

80. Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates":

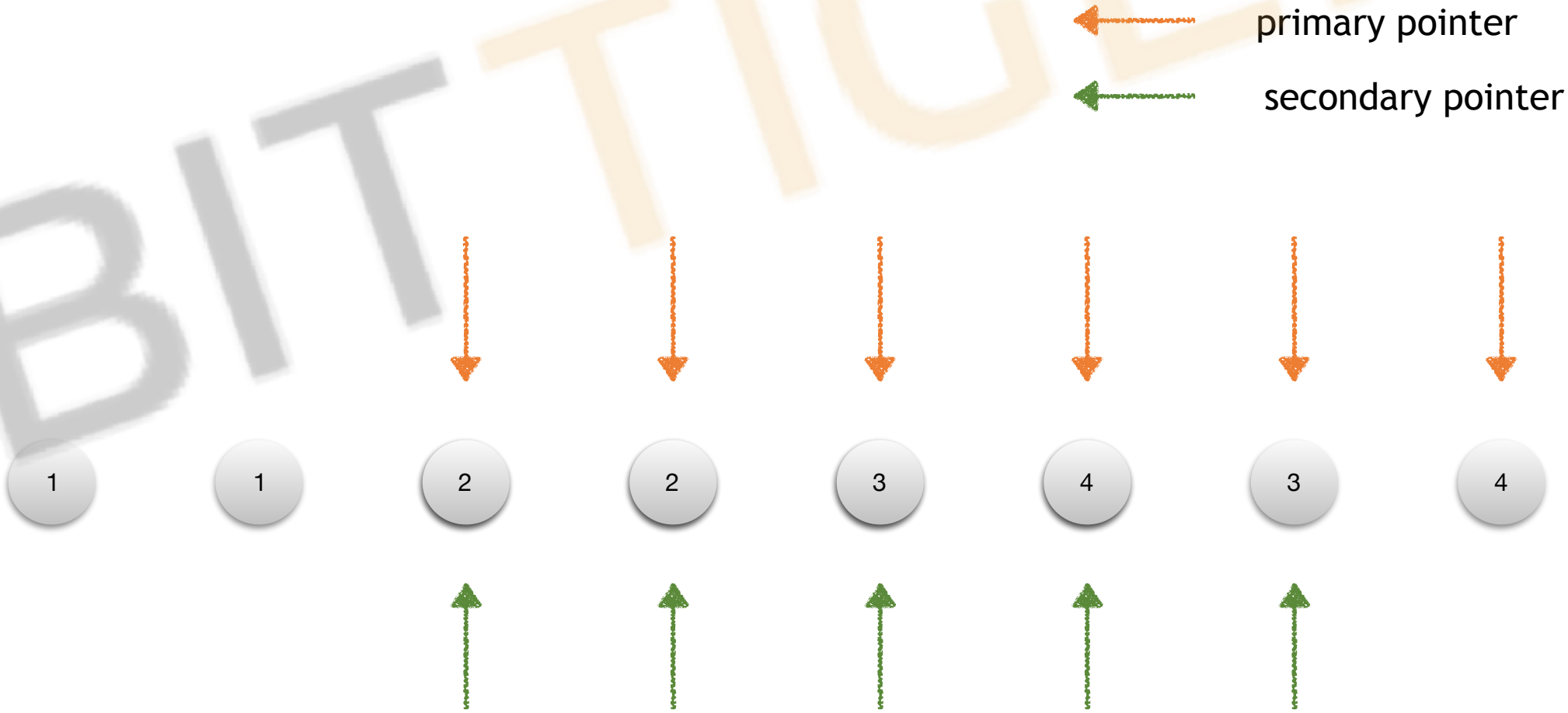
What if duplicates are allowed at most *twice*?

For example,

Given sorted array *nums* = `[1,1,1,2,2,3]`.

Your function should return length = `5`, with the first five elements of *nums* being `1`, `1`, `2`, `2` and `3`. It doesn't matter what you leave beyond the new length.

- Total Acce
- Total Subn
- Difficulty: I
- Contributo


primary pointer
secondary pointer
1
1
2
2
3
4
3
4

```
3   public int removeDuplicates(int[] nums) {
4       if(nums == null){
5           return 0;
6       }
7       if(nums.length <= 2){
8           return nums.length;
9       }
10
11      int pre = 0;
12
13      for(int i = 0; i < nums.length; i++){
14          if(pre < 2 || nums[i] > nums[pre - 2]){
15              nums[pre] = nums[i];
16              pre++;
17          }
18      }
19      return pre;
20  }
```

compare with the element before secondary pointer

3. Longest Substring Without Repeating Characters

Given a string, find the length of the **longest substring** without repeating characters.

Examples:

Given `"abcabcbb"`, the answer is `"abc"`, which the length is 3.

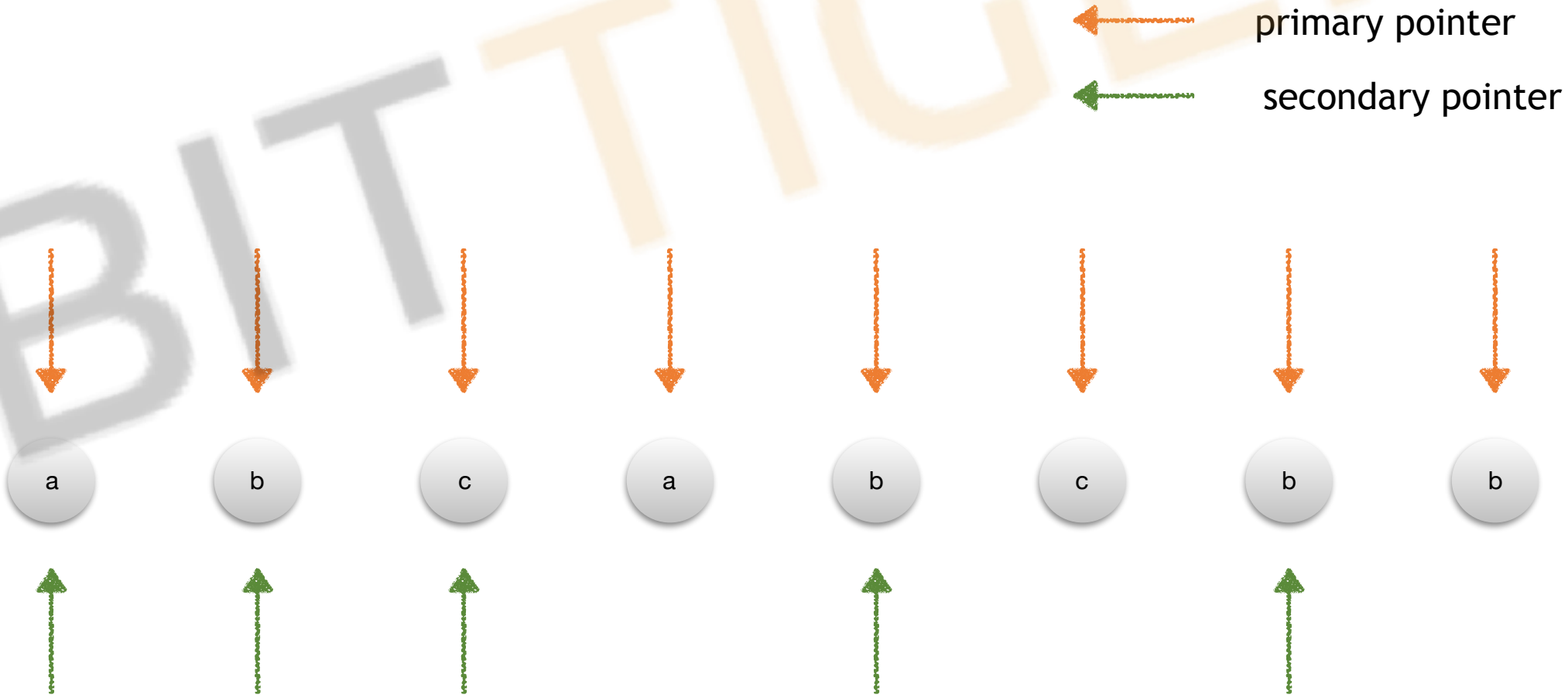
Given `"bbbbb"`, the answer is `"b"`, with the length of 1.

Given `"pwwkew"`, the answer is `"wke"`, with the length of 3. Note that the answer must be a **substring**, `"pwke"` is a *subsequence* and not a substring.

- Total
- Total
- Diffic
- Cont

10 min

```
public int lengthOfLongestSubstring(String s) {

}
```



HashMap 记录char对应 最新 坐标

```java
public int lengthOfLongestSubstring(String s) {
    if(s == null || s.length() == 0){
        return 0;
    }
    char[] str = s.toCharArray();
    int[] map = new int[256];

    int max = 0;
    int beg = 0;

    for(int i = 0; i < s.length(); i++){
        if(map[str[i]] == 0){
            max = Math.max(max, i - beg + 1);
        }else{
            beg = Math.max(beg, map[str[i]]);
            max = Math.max(max, i - beg + 1);
        }
        map[str[i]] = i + 1;
    }

    return max;
}
```

map 记录每个char出现的Most Right 位置

不需要挪动secondary pointer

挪动secondary pointer 到合法位置

159. Longest Substring with At Most Two Distinct Characters

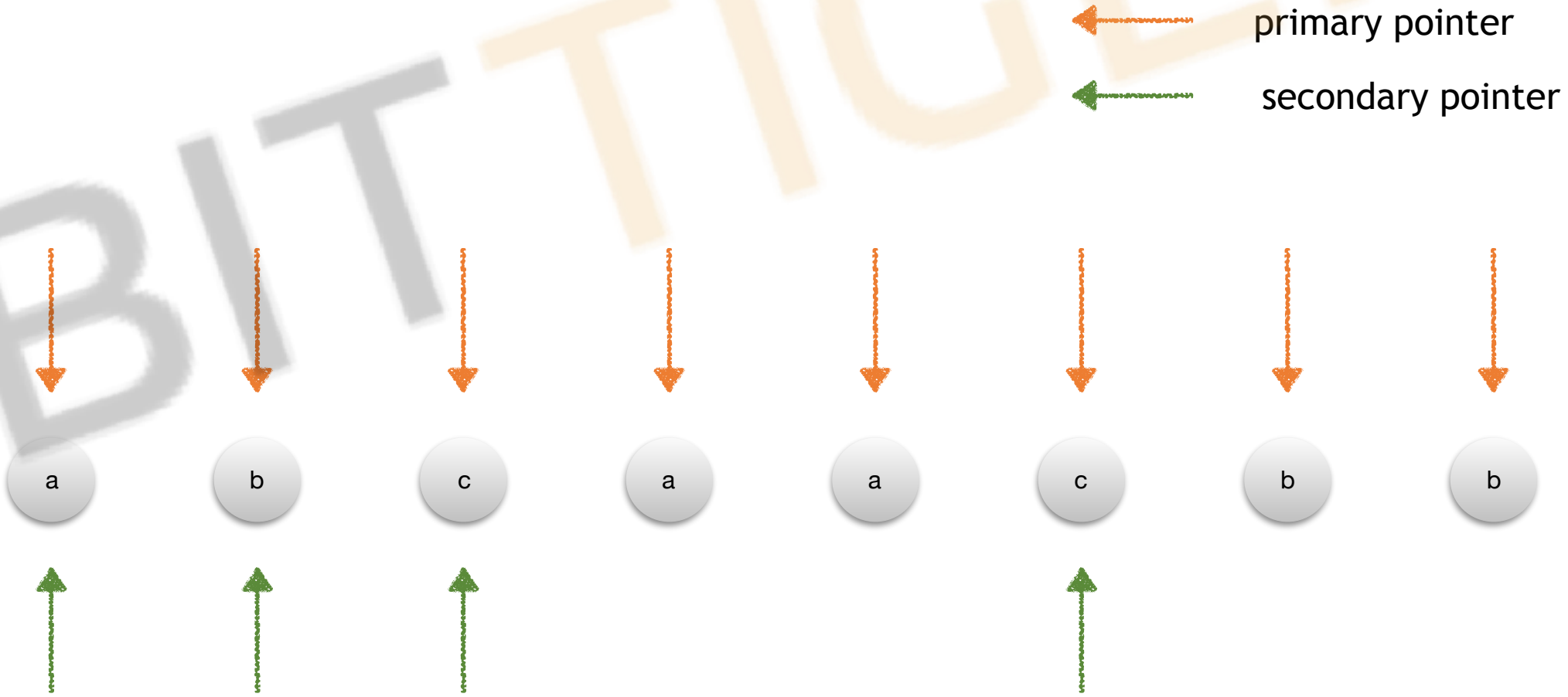
Given a string, find the length of the longest substring T that contains at most 2 distinct characters.

For example, Given s = `“eceba”`,

T is "ece" which its length is 3.

10 min

```
public int lengthOfLongestSubstringTwoDistinct(String str) {

}
```



HashMap 记录char对应 最新 坐标

```java
public int lengthOfLongestSubstringTwoDistinct(String str) {
    if(str == null || str.length() == 0){
        return 0;
    }

    Map<Character, Integer> map = new HashMap<>();

    int beg = 0;
    char[] strArray = str.toCharArray();
    int res = 0;

    for(int i = 0; i < str.length(); i++){
        if(map.size() < 2 || (map.size() == 2 && map.containsKey(strArray[i]))){
            // meet the requirement
            map.put(strArray[i], i);
        }else{
            // do not meet. replace most left one
            int left = i;
            for(int val: map.values()){
                left = Math.min(left, val);
            }
            beg = left + 1;
            map.remove(strArray[left]);
            map.put(strArray[i], i);
        }

        res = Math.max(res, i - beg + 1);
    }

    return res;
}
```

小于等于2个char 不需挪动 secondary pointer

大于2个char 挪动 secondary pointer

340. Longest Substring with At Most K Distinct Characters

Given a string, find the length of the longest substring T that contains at most *k* distinct characters.

For example, Given s = `“eceba”` and k = 2,

T is "ece" which its length is 3.

```java
public int lengthOfLongestSubstringKDistinct(String s, int k) {
    if(s == null || s.length() == 0 || k == 0){
        return 0;
    }

    Map<Character, Integer> map = new HashMap<>();

    int beg = 0;
    int res = 0;
    char[] str = s.toCharArray();

    for(int i = 0; i < s.length(); i++){
        if(map.size() < k || (map.size() == k && map.containsKey(str[i]))){
            map.put(str[i], i);
        }else{
            int left = i;
            for(int val: map.values()){
                left = Math.min(left, val);
            }
            beg = left + 1;
            map.remove(str[left]);
            map.put(str[i], i);
        }
         res = Math.max(res, i - beg + 1);
    }

    return res;
}
```

change to k

41. First Missing Positive

Given an unsorted integer array, find the first missing positive integer.

For example,

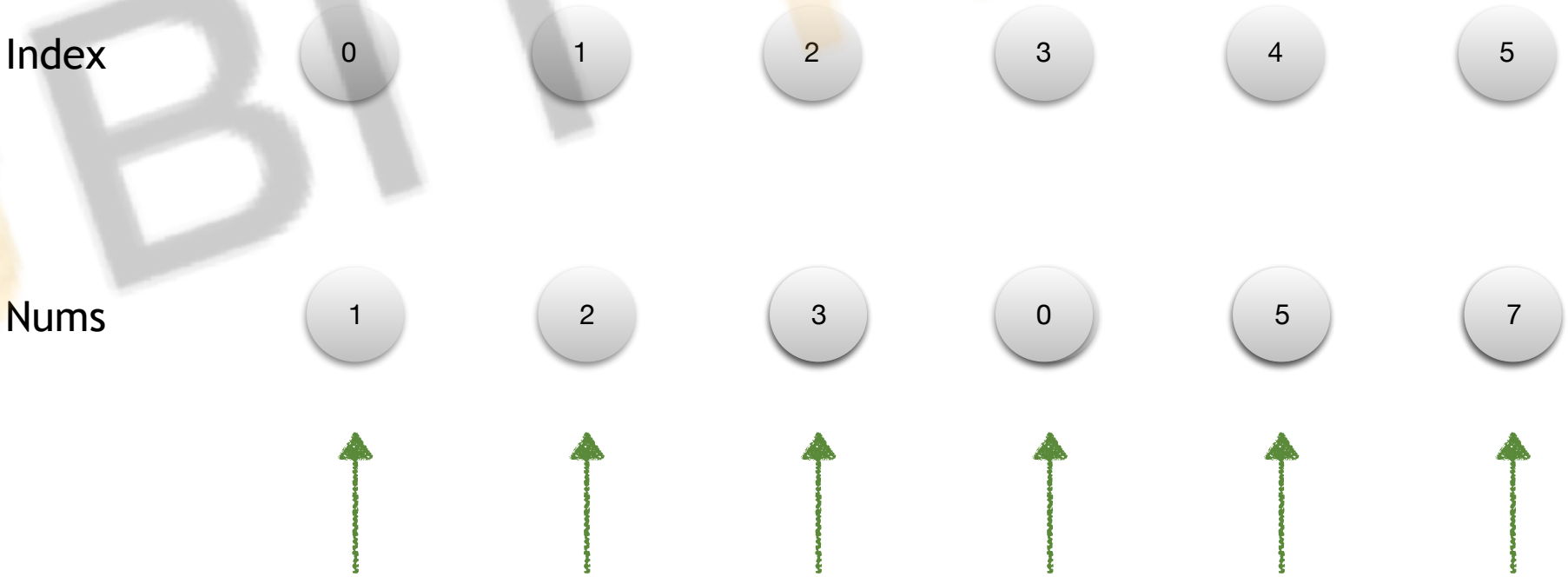
Given `[1,2,0]` return `3`,

and `[3,4,-1,1]` return `2`.

Your algorithm should run in $O(n)$ time and uses constant space.

10 min

```
public int firstMissingPositive(int[] nums) {

}
```


Index
0
1
2
3
4
5
Nums
1
2
3
0
5
7

```java
public int firstMissingPositive(int[] nums) {
    if(nums == null || nums.length == 0){
        return 1;
    }
    for(int i = 0; i < nums.length; i++){
        if(nums[i] - 1 >= 0 && nums[i] - 1 < nums.length && nums[nums[i] - 1] != nums[i]){
            swap(nums, i, nums[i] - 1);
            i--;
        }
    }

    for(int i = 0; i < nums.length; i++){
        if(nums[i] != i + 1){
            return i + 1;
        }
    }

    return nums.length + 1;
}

private void swap(int[] nums, int i, int j){
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
```

Index based

239. Sliding Window Maximum

Given an array *nums*, there is a sliding window of size *k* which is moving from the very left of the array to the very right. You can only see the *k* numbers in the window. Each time the sliding window moves right by one position.

For example,

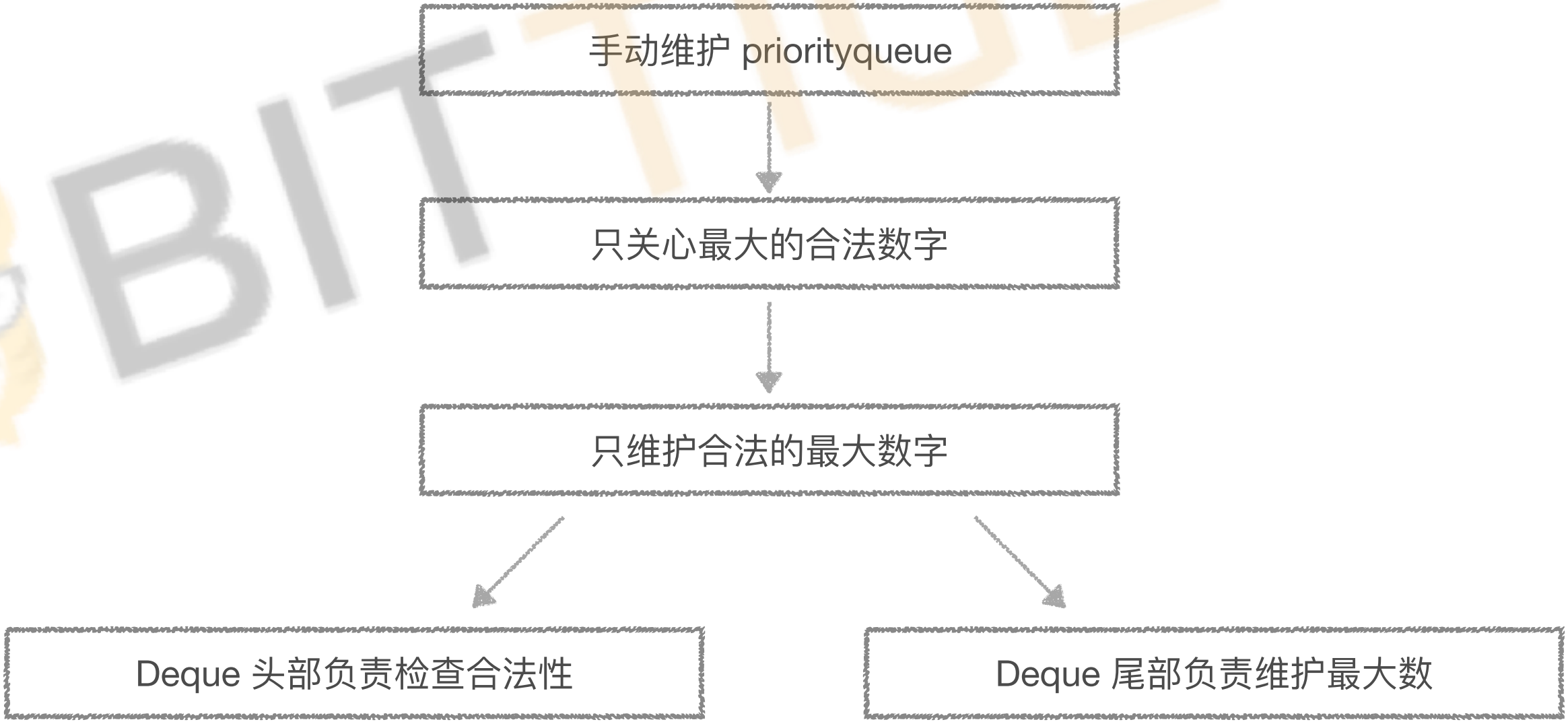
Given *nums* = `[1,3,-1,-3,5,3,6,7]`, and *k* = 3.

```
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

Therefore, return the max sliding window as `[3,3,5,5,6,7]`.

10 min

```
public int[] maxSlidingWindow(int[] nums, int k) {
}
```


手动维护 priorityqueue
只关心最大的合法数字
只维护合法的最大数字
Deque 头部负责检查合法性
Deque 尾部负责维护最大数

```java
public int[] maxSlidingWindow(int[] nums, int k) {
    if(nums == null || nums.length == 0){
        return new int[0];
    }
    int len = nums.length;
    int[] res = new int[len - k + 1];
    Deque<Integer> heap = new ArrayDeque<>();

    for(int i = 0; i < len; i++){
        // manually manipulate your heap
        if(!heap.isEmpty() && heap.peekFirst() == i - k){
            heap.removeFirst();
        }

        while(!heap.isEmpty() && nums[heap.peekLast()] < nums[i]){
            heap.removeLast();
        }

        heap.addLast(i);

        if(i - k + 1 >= 0){
            res[i - k + 1] = nums[heap.peekFirst()];
        }
    }

    return res;
}
```

移除 不在sliding window范围中的元素

移除 smaller 元素

42. Trapping Rain Water

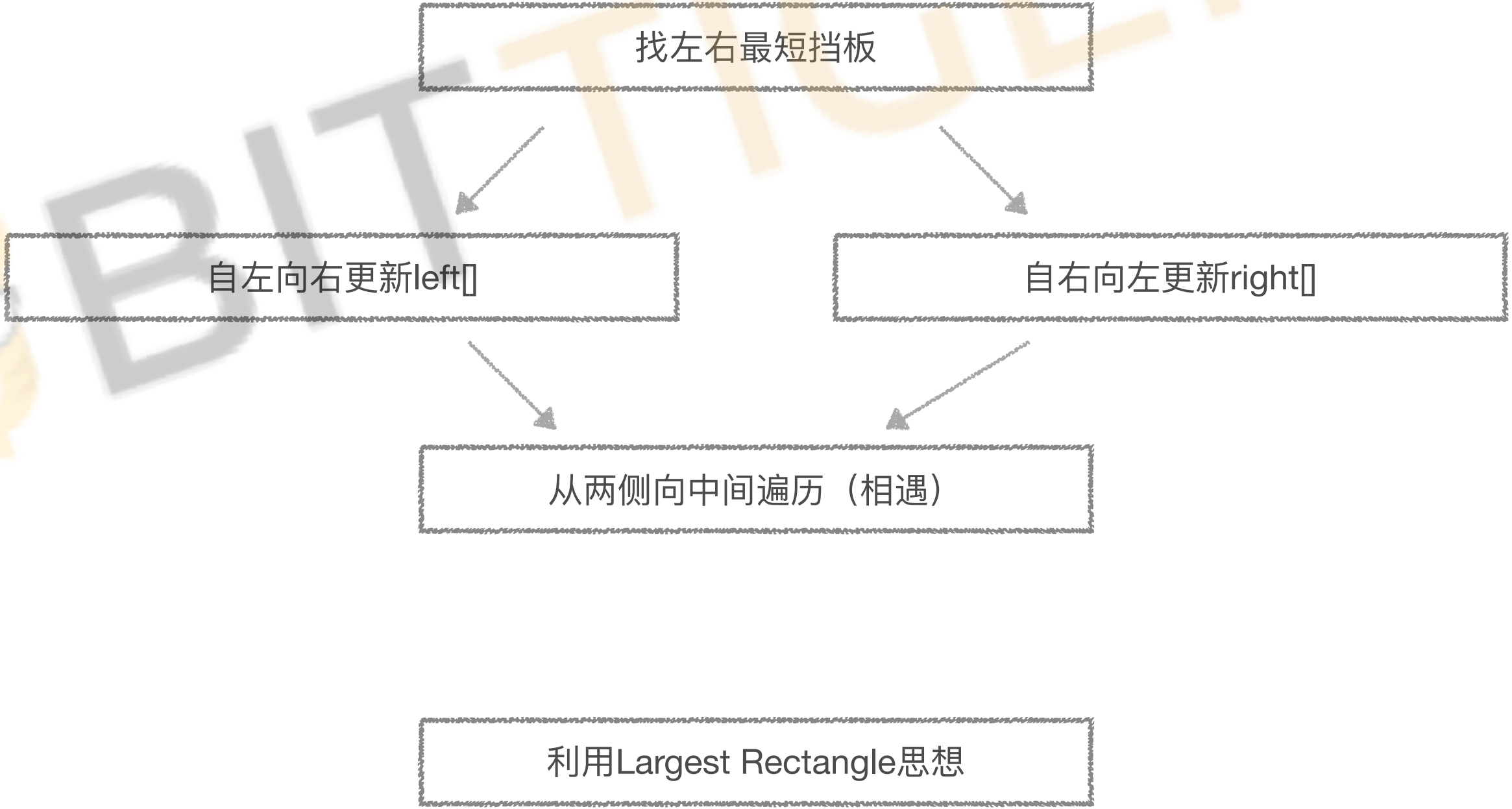
Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example,

Given `[0,1,0,2,1,0,1,3,2,1,2,1]`, return `6`.

10 min

```
public int trap(int[] height) {
}
```


找左右最短挡板
自左向右更新left[]
自右向左更新right[]
从两侧向中间遍历（相遇）
利用Largest Rectangle思想

```java
public int trap(int[] height) {
    if(height == null || height.length <= 1){
        return 0;
    }

    int left = 0;
    int right = height.length - 1;
    int maxLeft = 0;
    int maxRight = 0;
    int res = 0;

    while(left <= right){
        if(height[left] <= height[right]){
            if(height[left] >= maxLeft){
                maxLeft = height[left];
            }else{
                res += maxLeft - height[left];
            }

            left++;
        }else{
            if(height[right] >= maxRight){
                maxRight = height[right];
            }else{
                res += maxRight - height[right];
            }

            right--;
        }

    }

    return res;
}
```

判断左右短板

确认是否可以存水

```java
public int trapStack(int[] height) {
    if(height == null || height.length <= 1){
        return 0;
    }

    Deque<Integer> stack = new ArrayDeque<>();

    int res = 0;
    for(int i = 0; i < height.length; ){
        if(stack.isEmpty() || height[i] <= height[stack.peekFirst()]){
            stack.addFirst(i);
            i++;
        }else{
            // height[i] > height[stack.peekFirst()] start trapping previous water
            int curIndex = stack.removeFirst();
            if(stack.isEmpty()){
                // no left boundary
                continue;
            }
            int preIndex = stack.peekFirst();
            res += (Math.min(height[preIndex], height[i]) - height[curIndex]) * (i - preIndex - 1);
        }
    }

    return res;

}
```

维护降序stack

Homework

280. Wiggle Sort
324. Wiggle Sort II
456. 132 Pattern
25. Reverse Nodes in k-Group
442. Find All Duplicates in an Array
287. Find the Duplicate Number

Q & A

Thank you