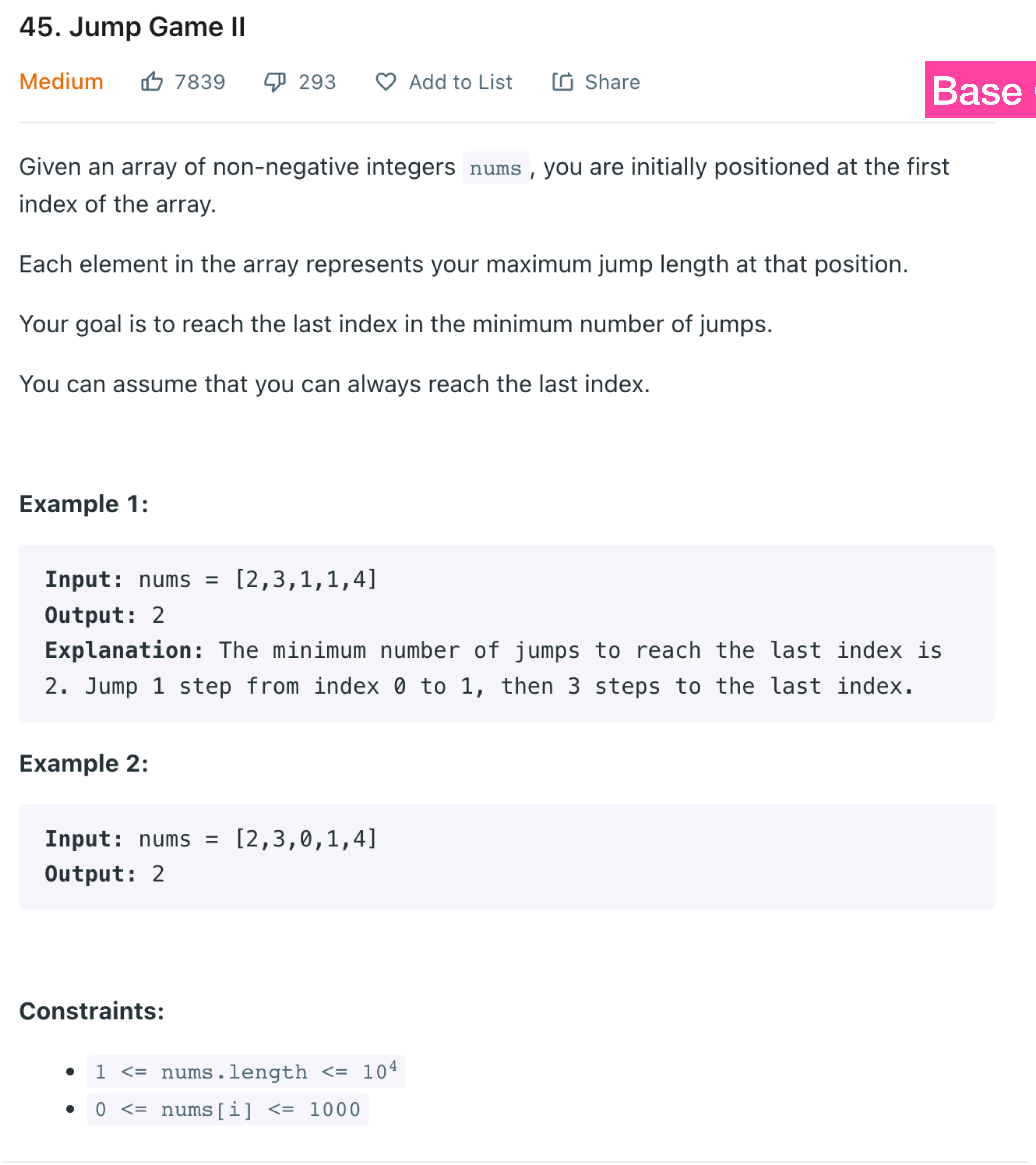


45. Jump Game II

Given an array of non-negative integers `nums`, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

You can assume that you can always reach the last index.

Example 1:

```
Input: nums = [2,3,1,1,4]
Output: 2
Explanation: The minimum number of jumps to reach the last index is
2. Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

Example 2:

```
Input: nums = [2,3,0,1,4]
Output: 2
```

Constraints:

- 1 <= nums.length <= 10^4
- 0 <= nums[i] <= 1000

Base Cases:

When size is : 1

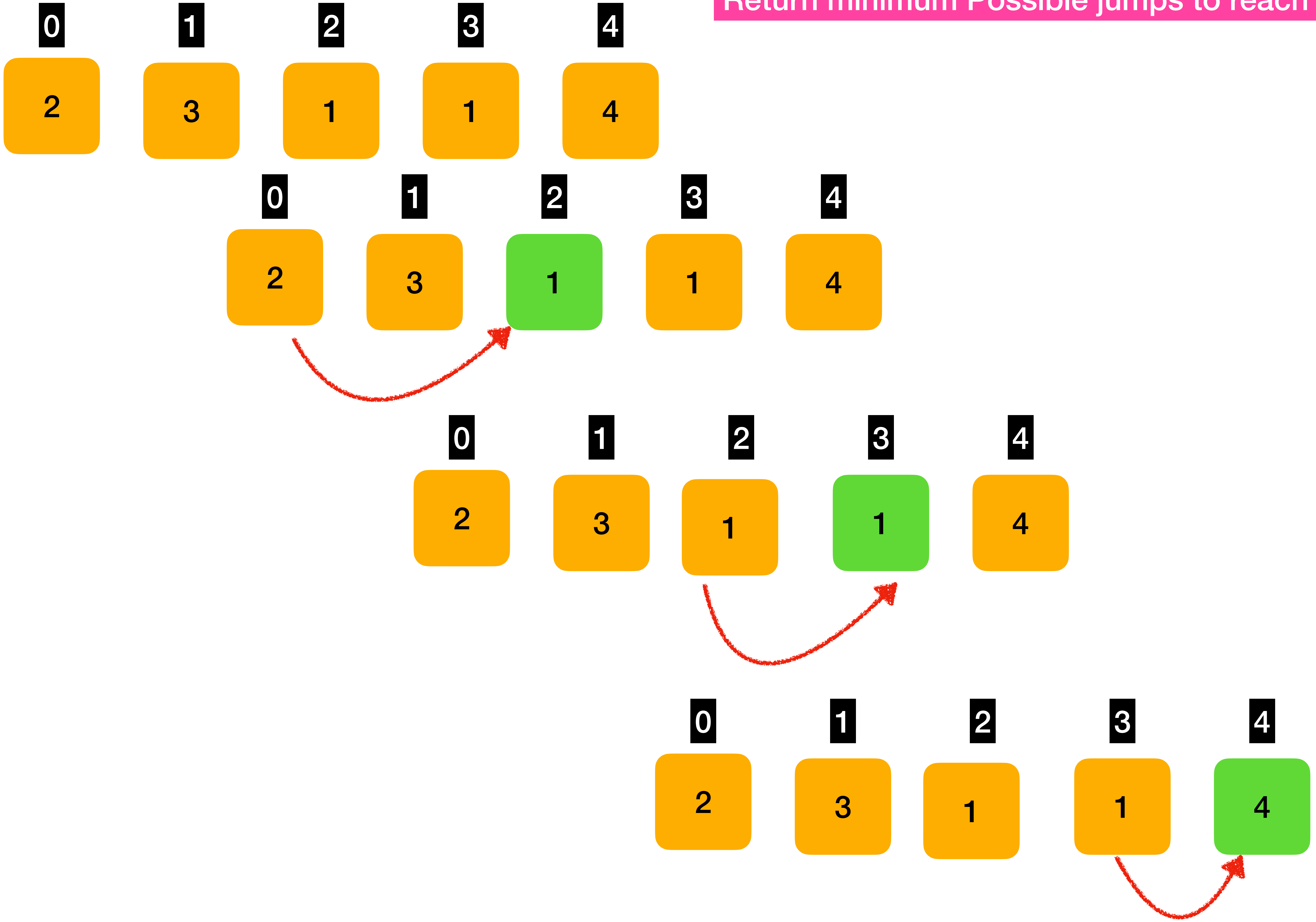
As we arleady in n-1 index

minimumNoOfJumps to Reach Target = 0

0

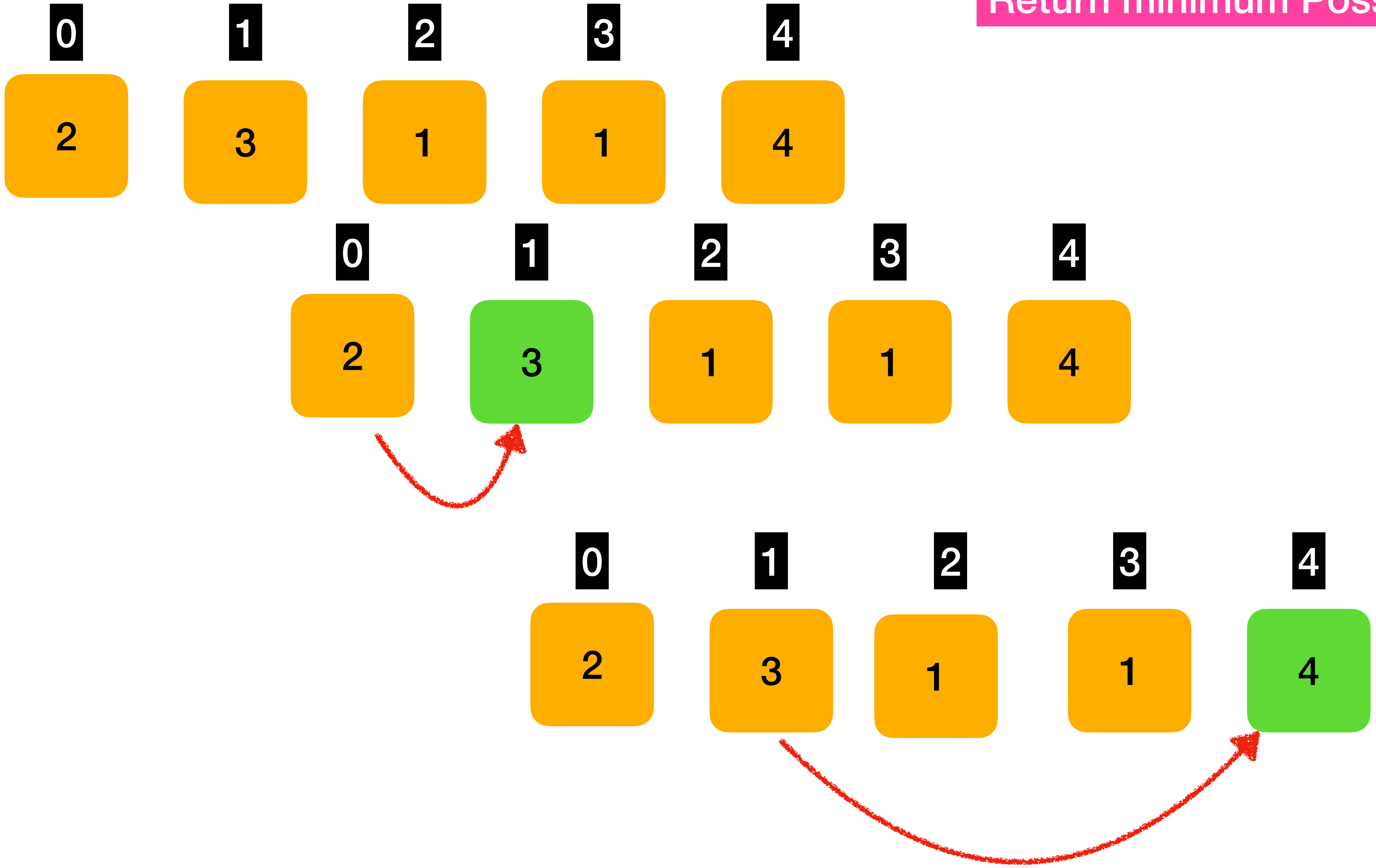
2

Return minimum Possible jumps to reach to the target



Total Jumps : 3

Return minimum Possible to reach to the target



0	1	2	3	4
2	3	1	1	4

0	1	2	3	4
2	3	1	1	4

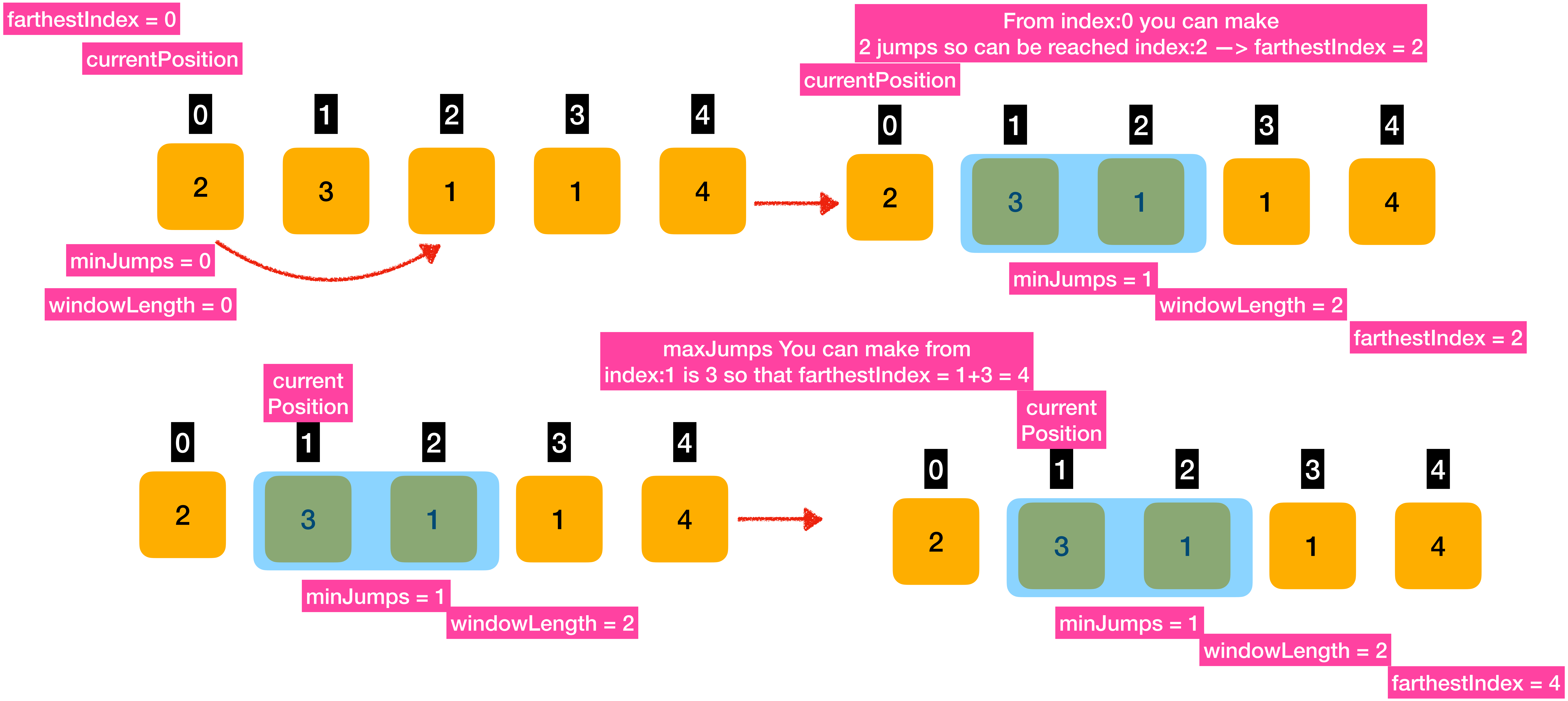
0	1	2	3	4
2	3	1	1	4

Total Jumps : 2

Output is : 2

Greedy

Local Optimisation.
Take max possible jumps from currentPosition at each window.



farthestIndex = 0

currentPosition

minJumps = 0

windowLength = 0

From index:0 you can make
2 jumps so can be reached index:2 —> farthestIndex = 2

currentPosition

minJumps = 1

windowLength = 2

farthestIndex = 2

current
Position

minJumps = 1

windowLength = 2

maxJumps You can make from
index:1 is 3 so that farthestIndex = 1+3 = 4

current
Position

minJumps = 1

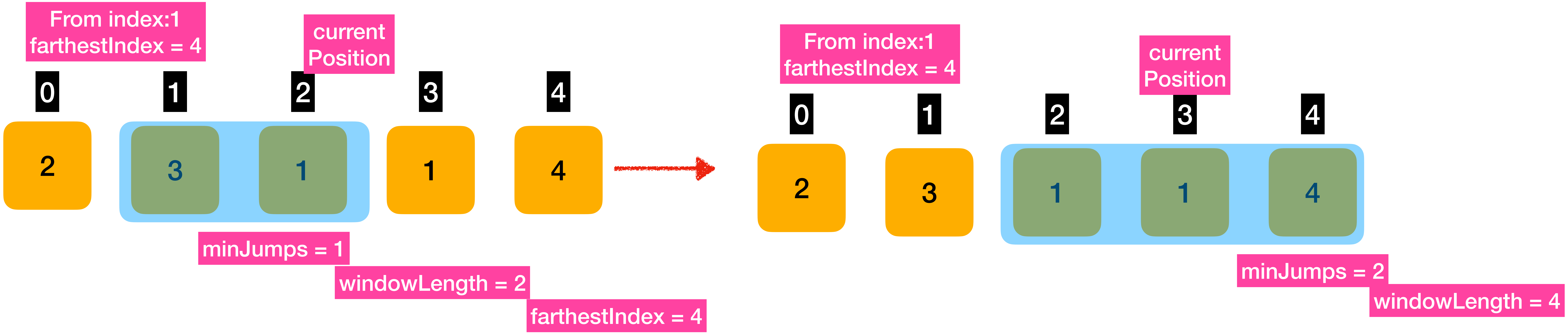
windowLength = 2

farthestIndex = 4

From index:2 you can make max jump as 1 so we can only reach = index:3.

As part of greedy we should consider Highest jump from each window so no update In farthestIndex.

As the currentPosition reaching window length then take Jump to the farthestIndex, it means update the windowLength to farthestIndex then Increment the minJumps.



From index:1 farthestIndex = 4

current Position

minJumps = 1

windowLength = 2

farthestIndex = 4

From index:1 farthestIndex = 4

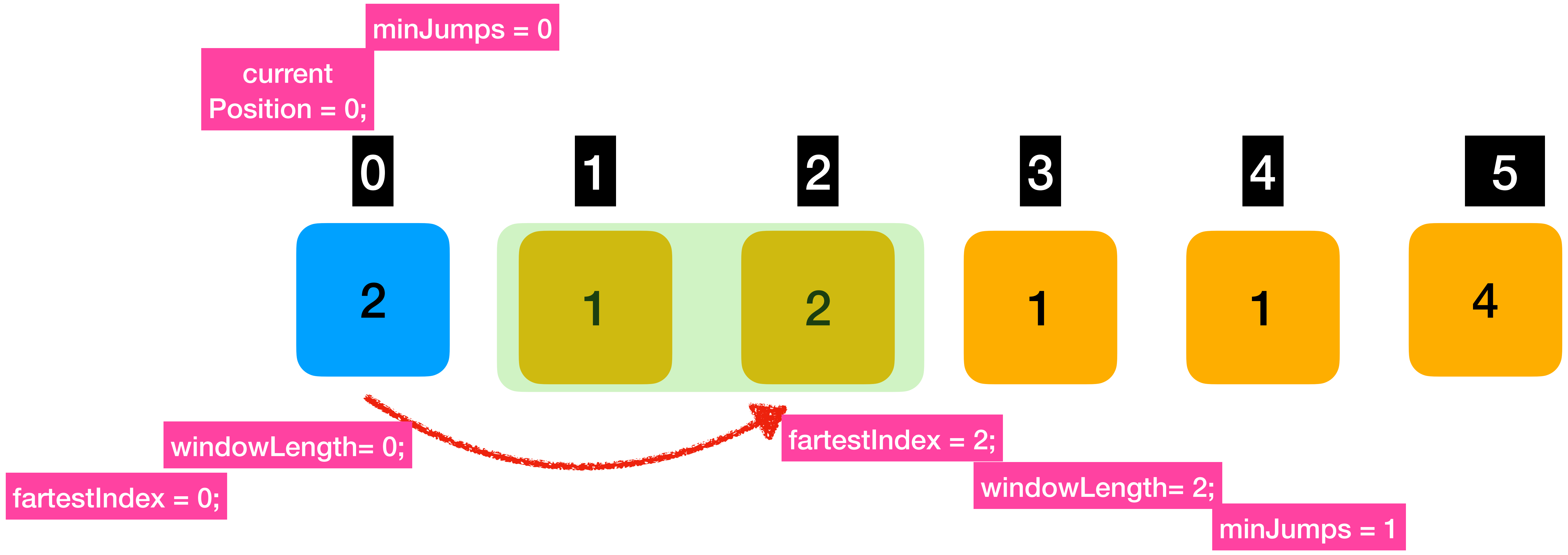
current Position

minJumps = 2

windowLength = 4

As the windowLength reached n-1 return minJumps: 2

Time Complexity : O(n)
Space Complexity : O(1)



minJumps = 0

current Position = 0;

windowLength= 0;

fartestIndex = 0;

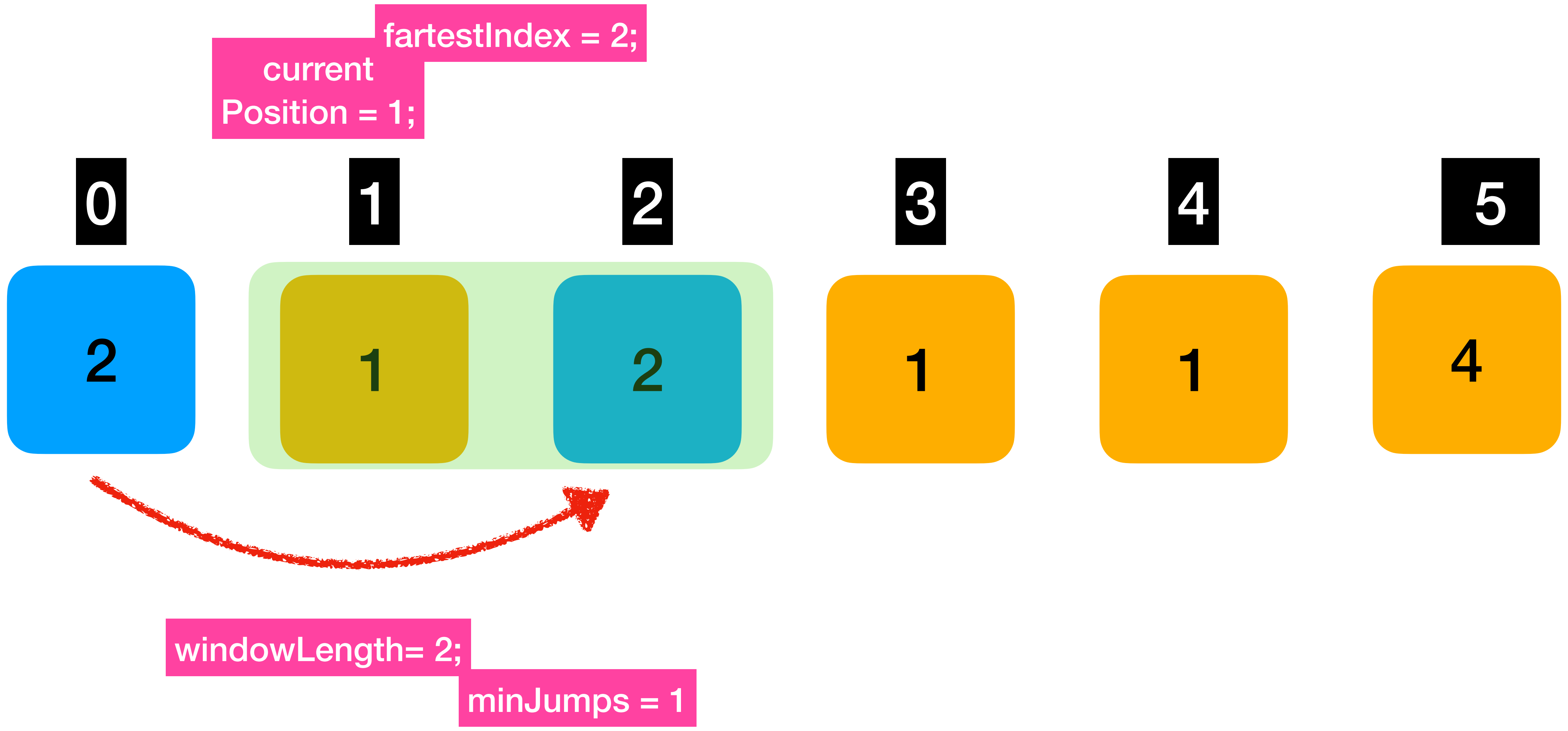
fartestIndex = 2;

windowLength= 2;

minJumps = 1

currentPosition = 0;
windowLength = 0;
farthestIndex = 0;
minJumps = 0

Time Complexity : O(n)
Space Complexity : O(1)

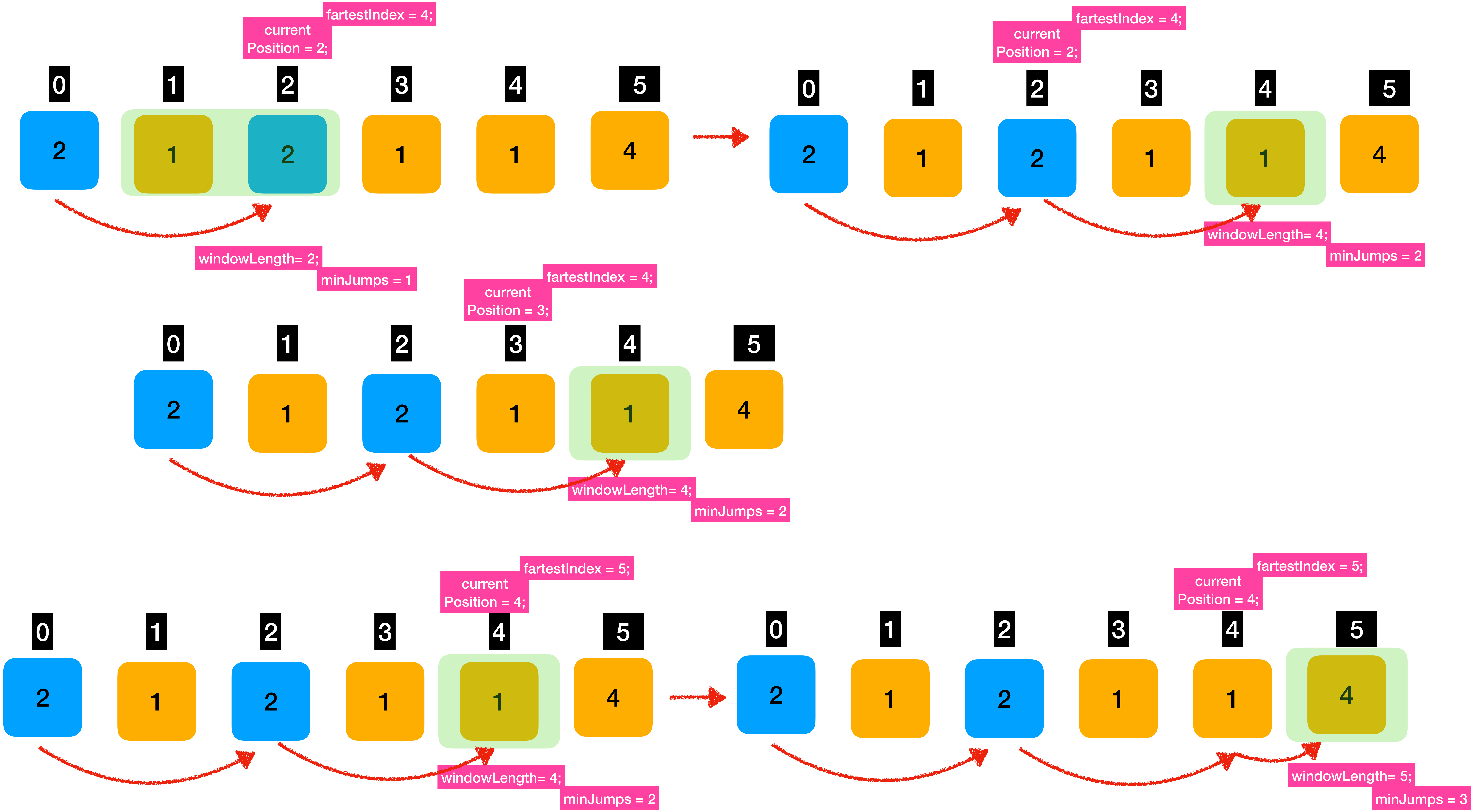


fartestIndex = 2;

current Position = 1;

windowLength= 2;

minJumps = 1



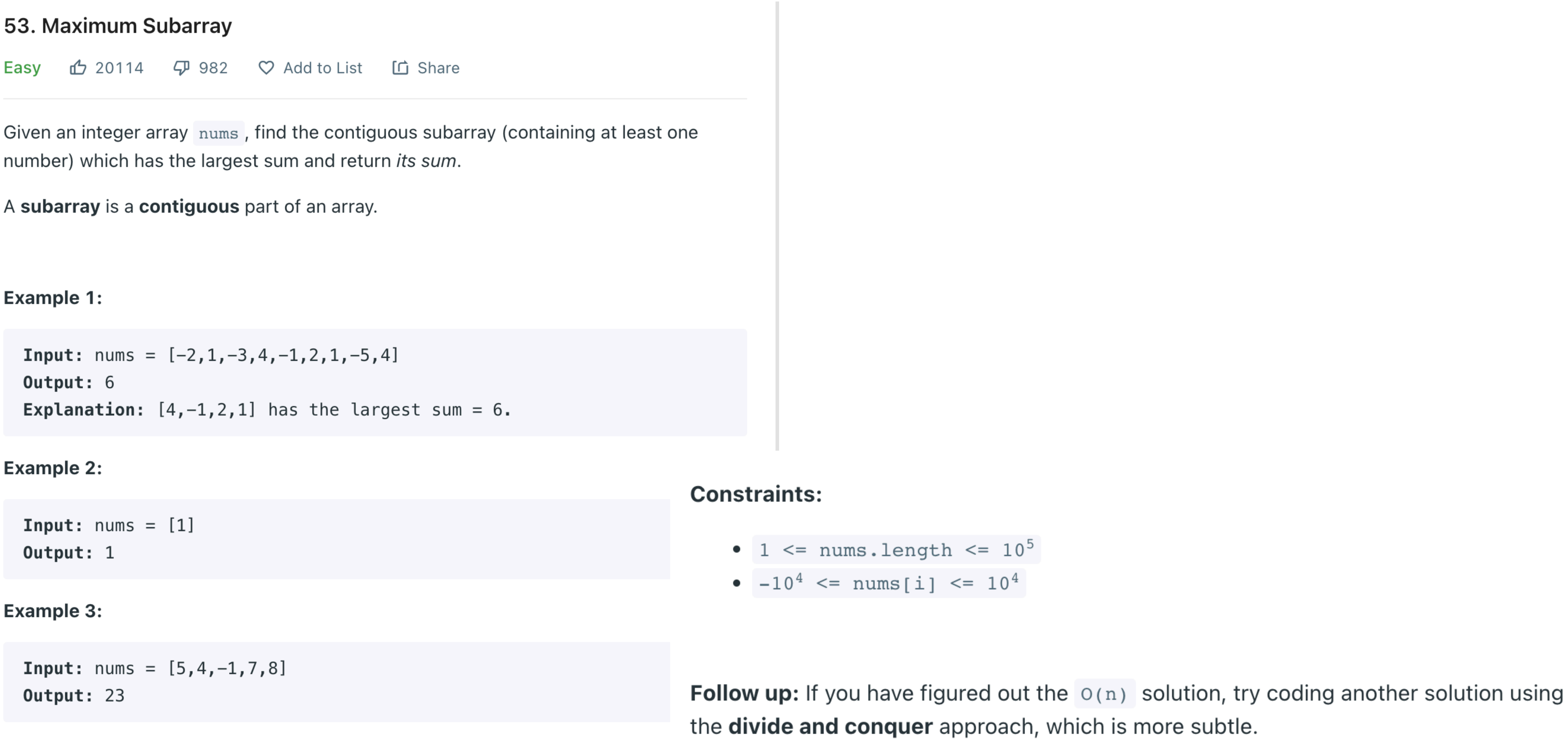
fartestIndex = 4;
current Position = 2;
0 1 2 3 4 5
2 1 2 1 1 4
windowLength= 2;
minJumps = 1

fartestIndex = 4;
current Position = 2;
0 1 2 3 4 5
2 1 2 1 1 4
windowLength= 4;
minJumps = 2

fartestIndex = 4;
current Position = 3;
0 1 2 3 4 5
2 1 2 1 1 4
windowLength= 4;
minJumps = 2

fartestIndex = 5;
current Position = 4;
0 1 2 3 4 5
2 1 2 1 1 4
windowLength= 4;
minJumps = 2

fartestIndex = 5;
current Position = 4;
0 1 2 3 4 5
2 1 2 1 1 4
windowLength= 5;
minJumps = 3



53. Maximum Subarray

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

A **subarray** is a **contiguous** part of an array.

Example 1:

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

Example 2:

```
Input: nums = [1]
Output: 1
```

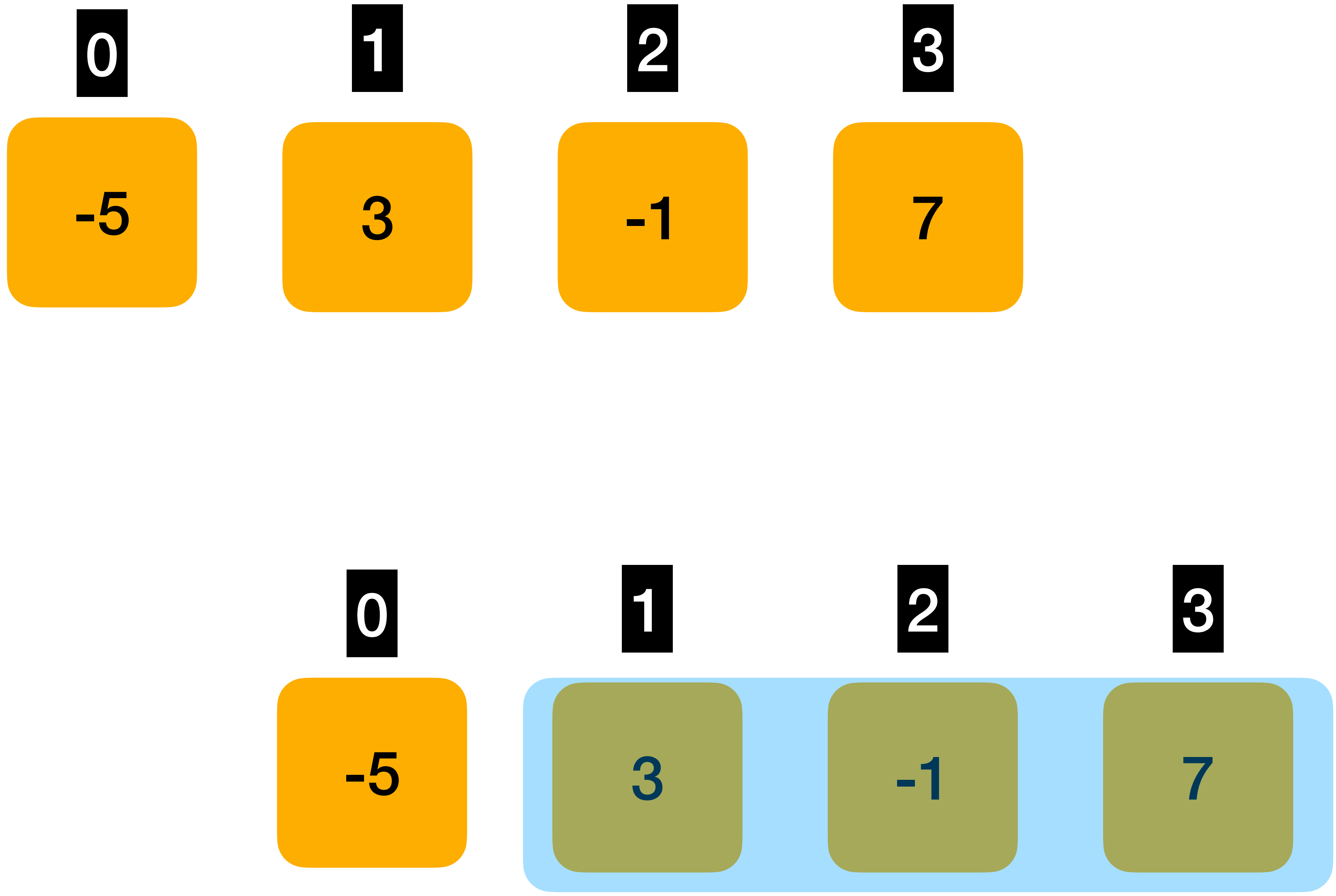
Example 3:

```
Input: nums = [5,4,-1,7,8]
Output: 23
```

Constraints:

- `1 <= nums.length <=` 10^5
- -10^4 `<= nums[i] <=` 10^4

Follow up: If you have figured out the `O(n)` solution, try coding another solution using the **divide and conquer** approach, which is more subtle.



0	1	2	3
-5	3	-1	7

0	1	2	3
-5	3	-1	7

The Max sum we can make is 9 : start from index:1 to index:3

Greedy

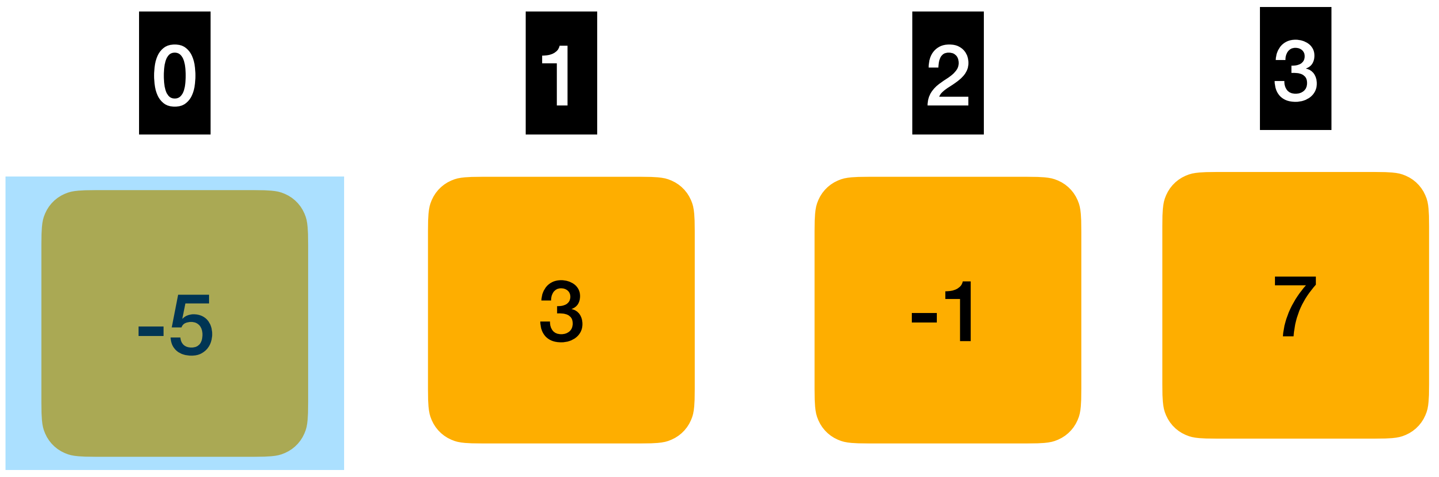
Local Optimisation.
For each move obtain the localMax,
compare with globalMax.

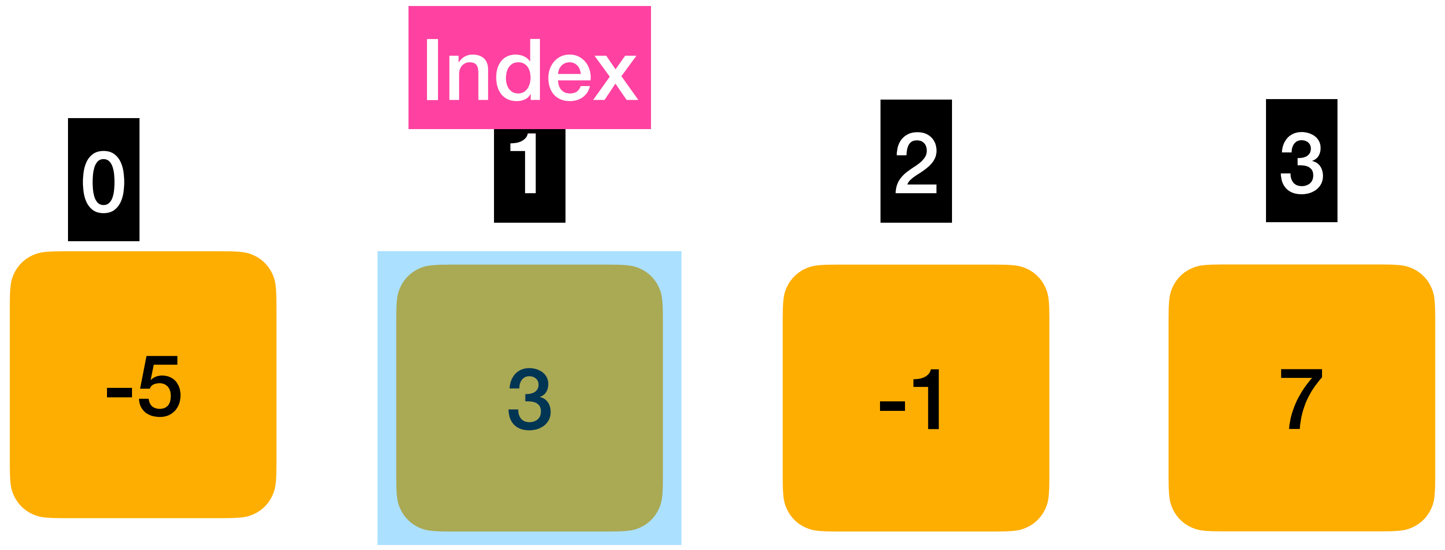
localMax = Math.max(localMax+num[i] , nums[i])

globalMax = Math.max(localMax, globalMax)

globalMax = -5

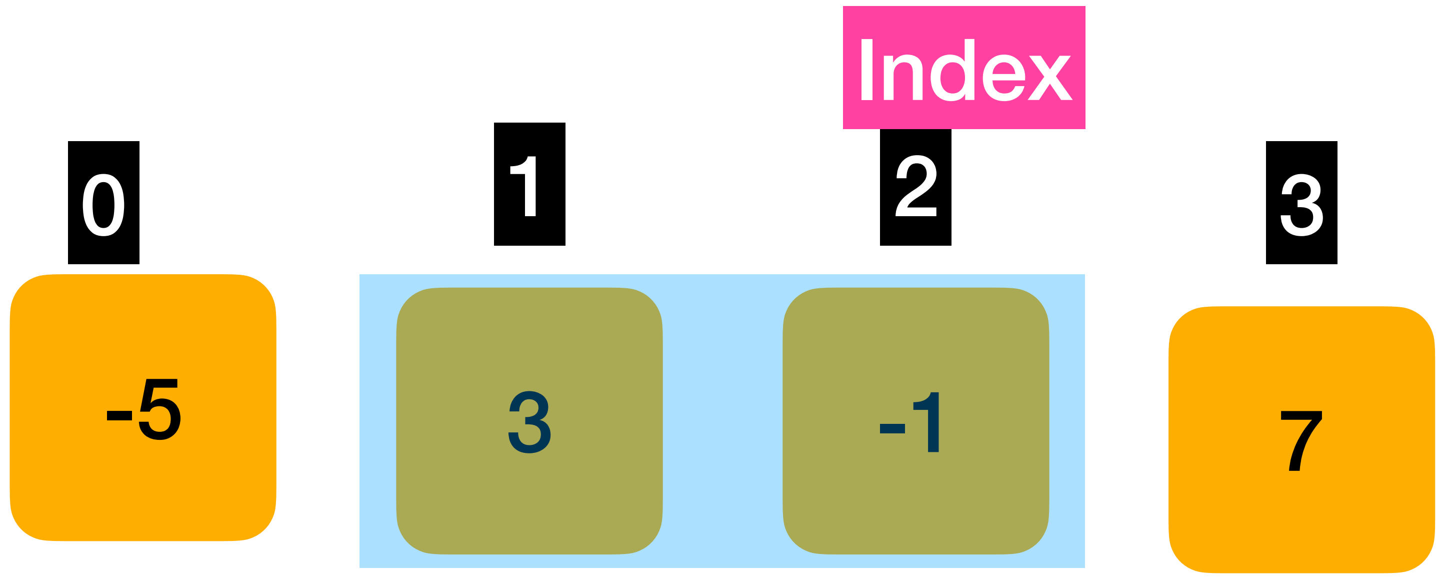
localMax = -5





localMax = Math.max(localMax+num[i] , nums[i]) = Max(-5+3, 3) = 3
—> considering only index:1 gets higher value.

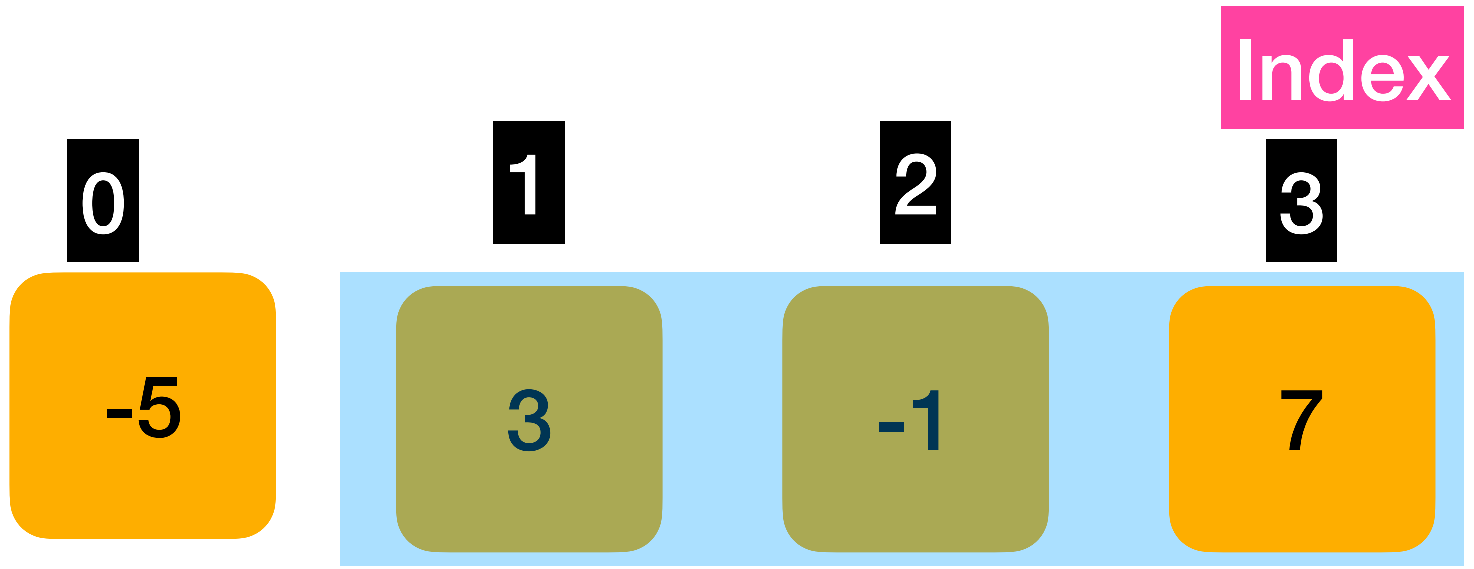
globalMax = Math.max(globalMax, localMax) = max (-5,3) = 3



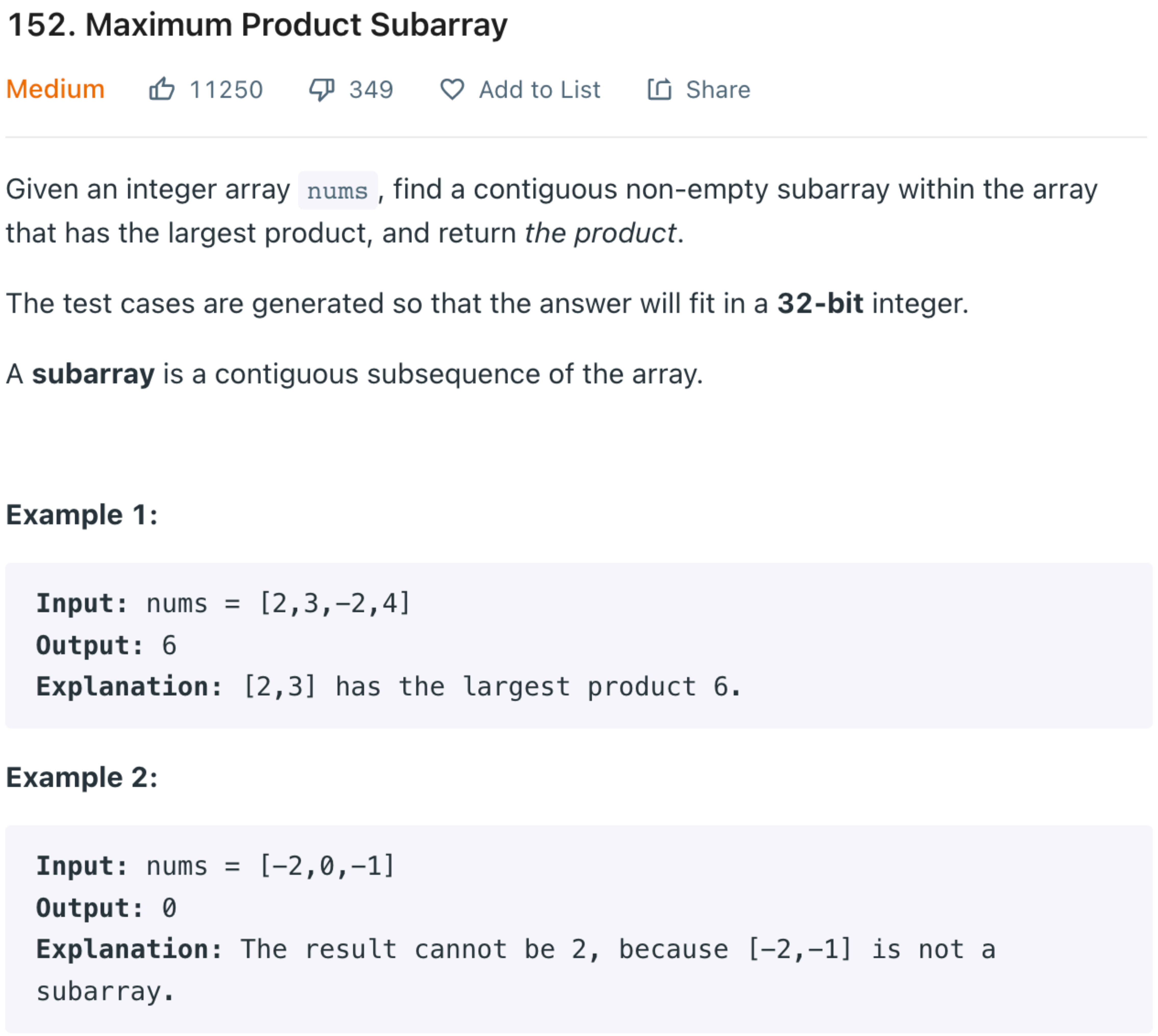
localMax = Math.max(localMax+nums[i] , nums[i])
= Max(3-1, -1) = 2

globalMax = Math.max(globalMax, localMax)
= max (3,2) = 3

globalMax = Math.max(globalMax, localMax) = max (3,9) = 9



localMax = Math.max(localMax+num[i] , nums[I]) = Max(2+7, 7) = 9



152. Maximum Product Subarray

Given an integer array `nums`, find a contiguous non-empty subarray within the array that has the largest product, and return *the product*.

The test cases are generated so that the answer will fit in a **32-bit** integer.

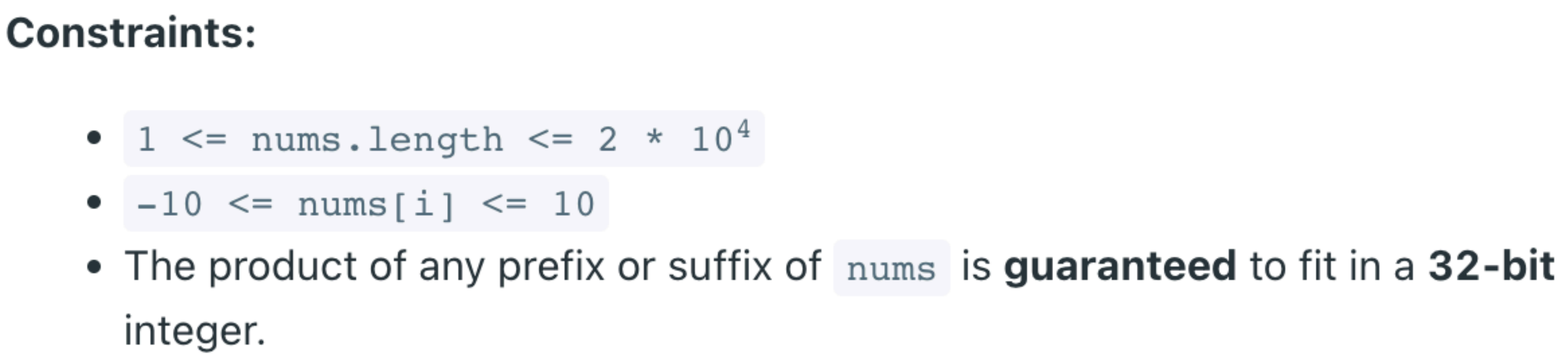
A **subarray** is a contiguous subsequence of the array.

Example 1:

```
Input: nums = [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product 6.
```

Example 2:

```
Input: nums = [-2,0,-1]
Output: 0
Explanation: The result cannot be 2, because [-2,-1] is not a
subarray.
```



Constraints:

- `1 <= nums.length <= 2 * 10`^{4}
- `-10 <= nums[i] <= 10`
- The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.

{2,3,-2,4} —-> maxProduct = 6

{-5,4,-3} —-> maxProduct = 60