3. Longest Substring Without Repeating Characters

Medium 24110 1069 Add to List Share

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

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

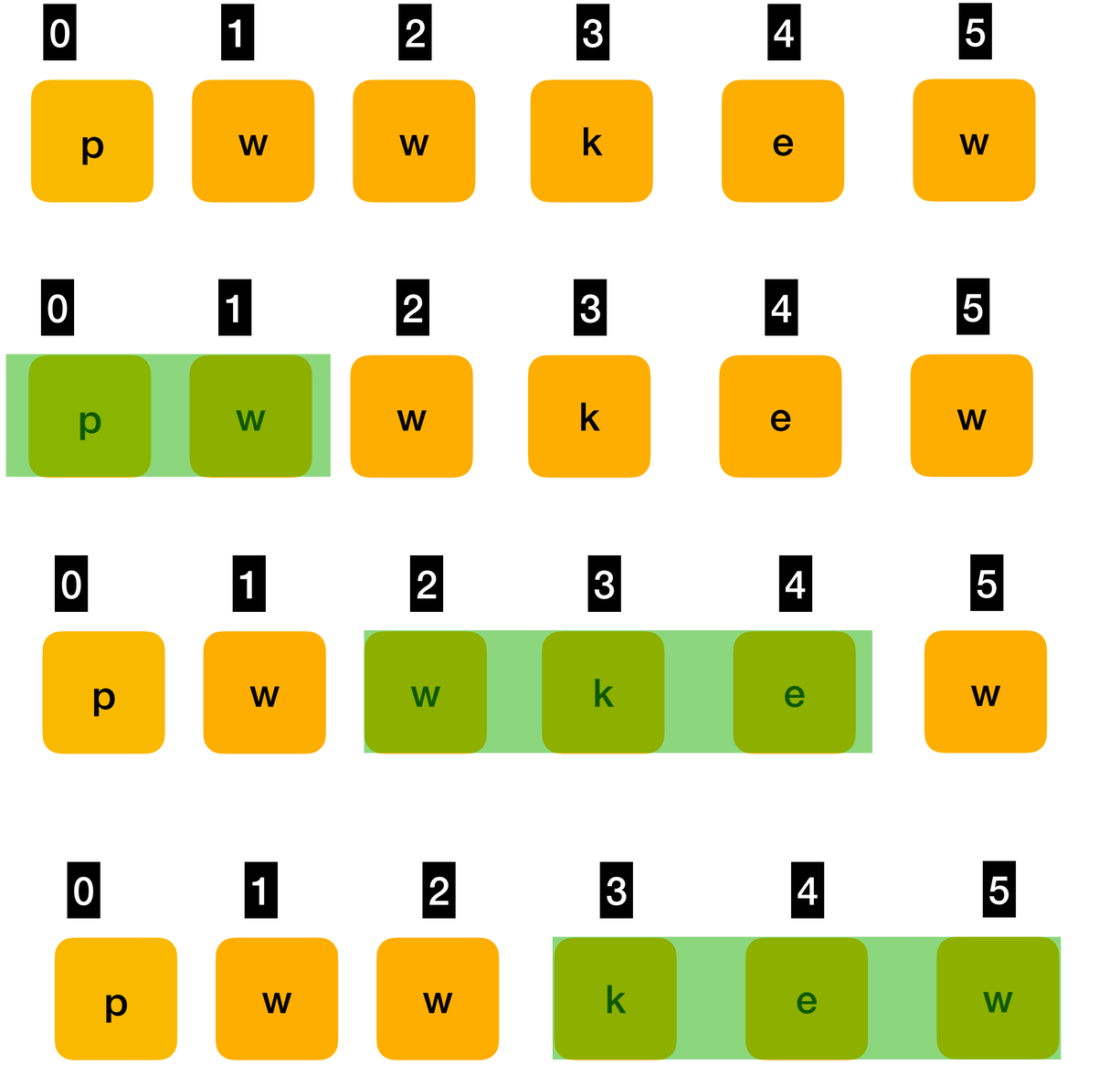
```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

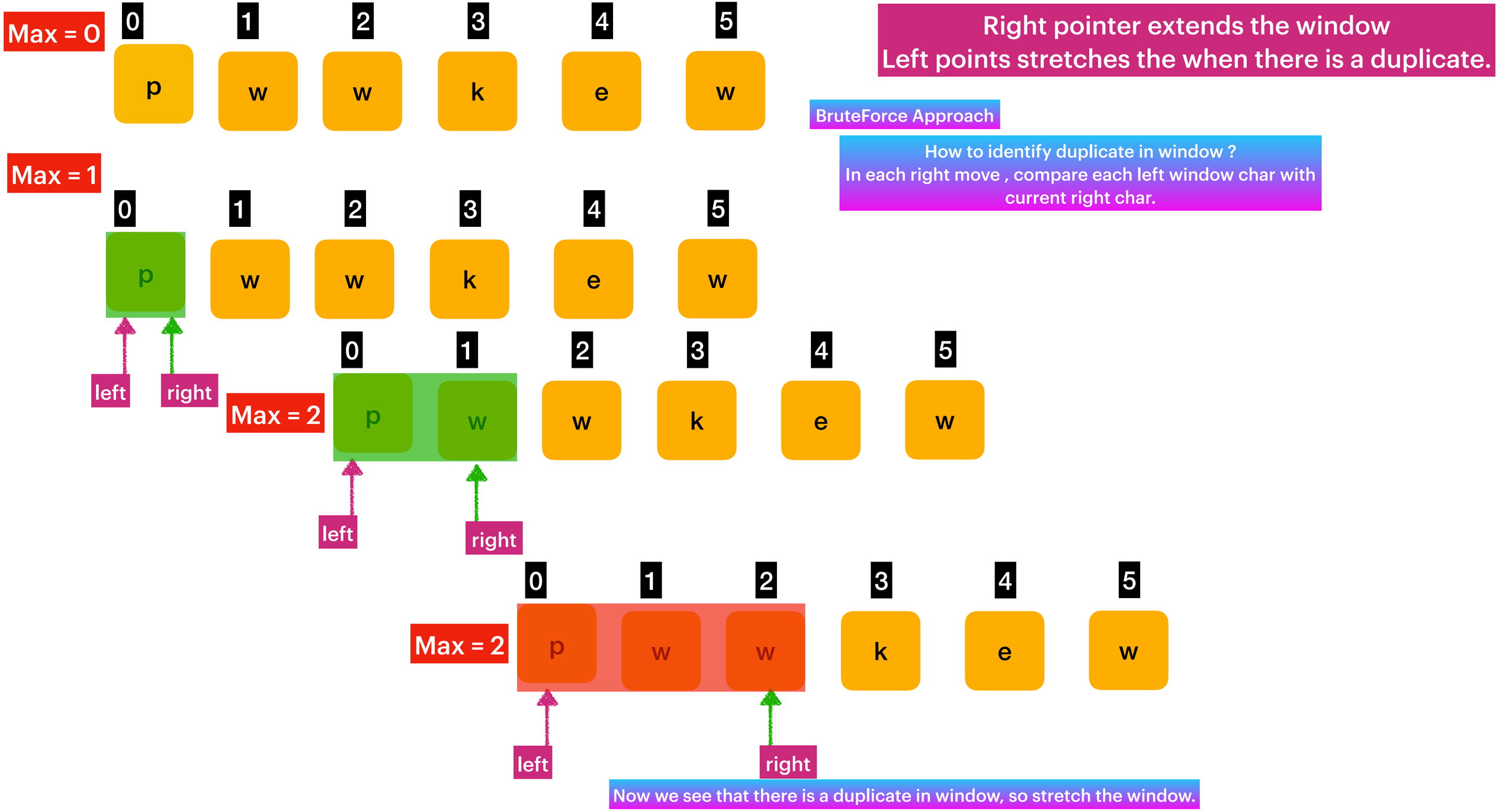
```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a
subsequence and not a substring.
```

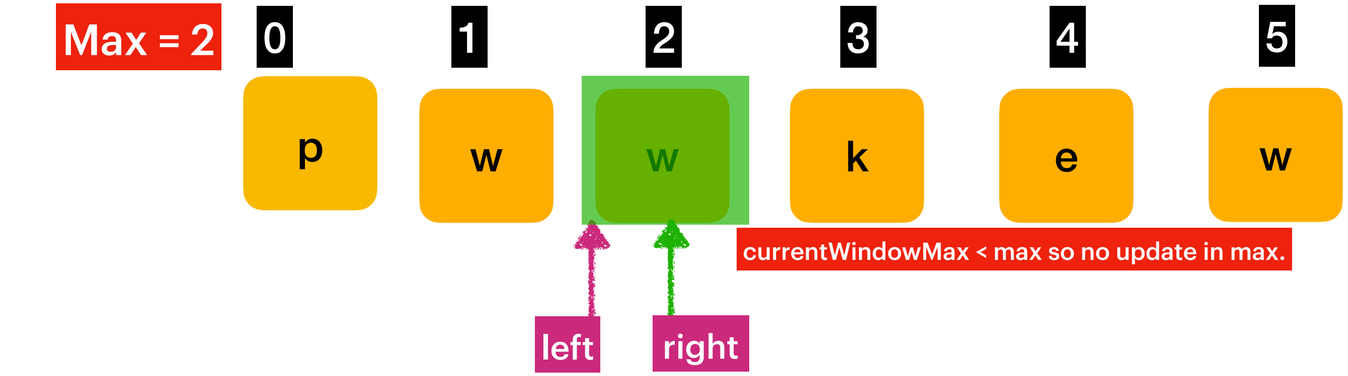
Constraints:

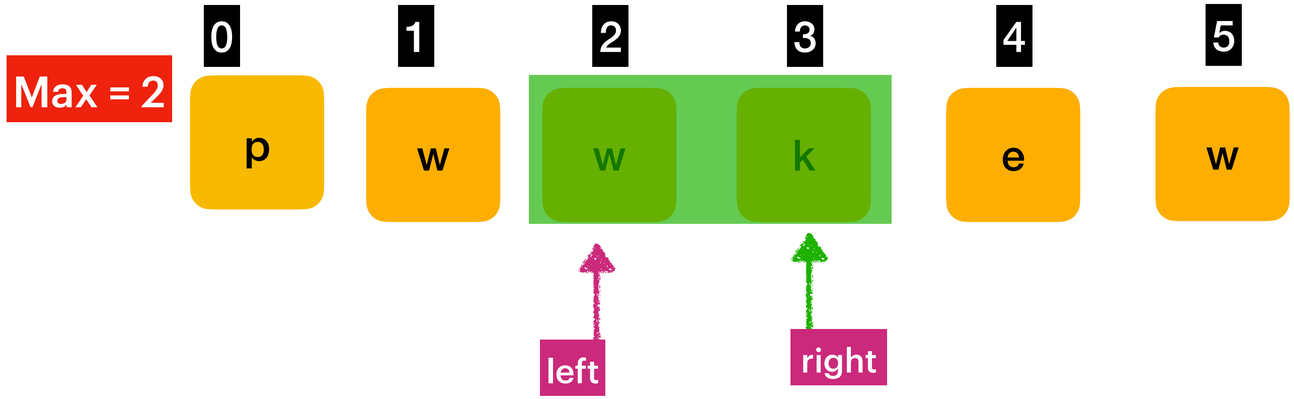
- `0 <= s.length <= 5 * 10^4`
- `s` consists of English letters, digits, symbols and spaces.

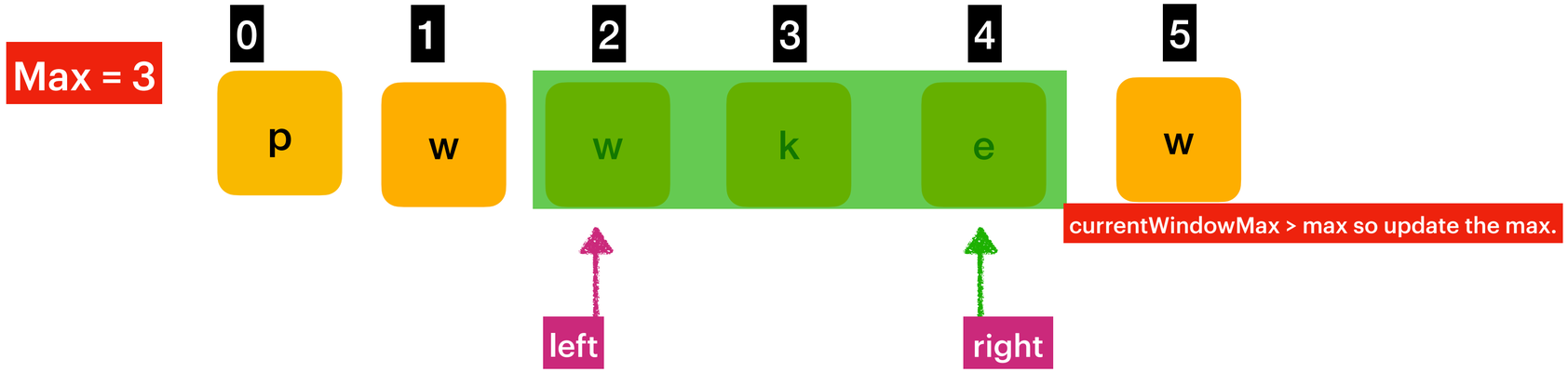


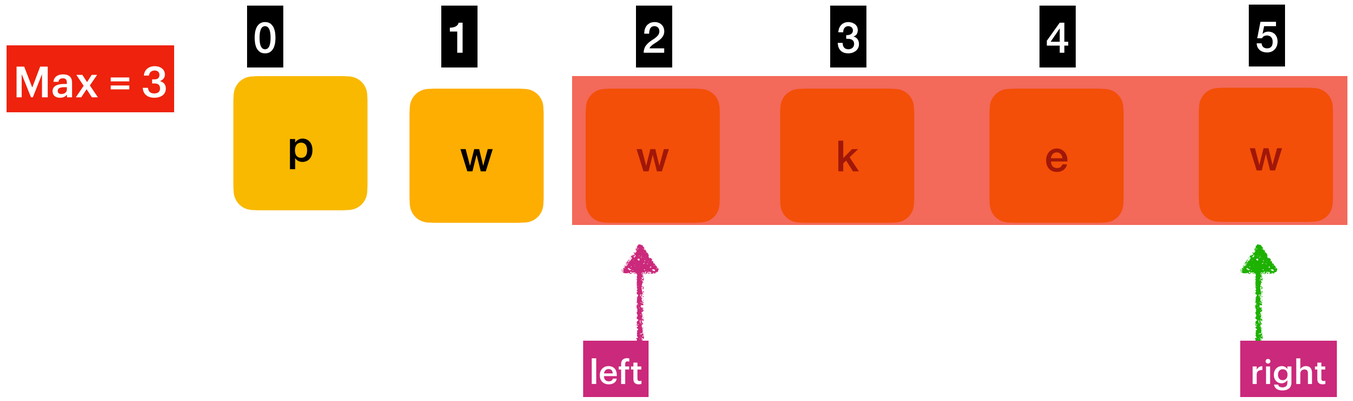
Expected Longest SubString without repeating characters :
either “wke” or “kew”
Return length = 3

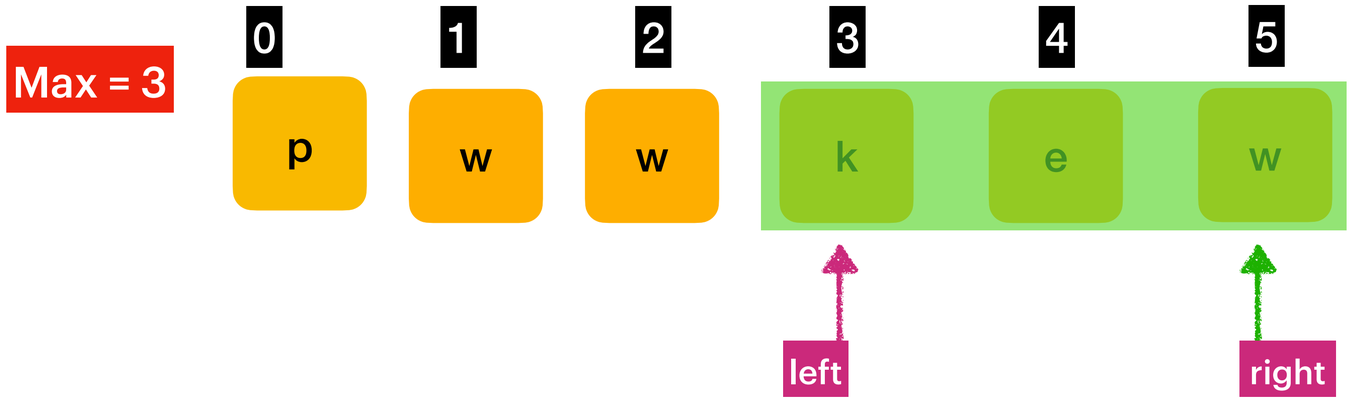

Max = 0
0
1
2
3
4
5
p
w
w
k
e
w
Right pointer extends the window
Left points stretches the when there is a duplicate.
BruteForce Approach
How to identify duplicate in window ?
In each right move , compare each left window char with current right char.
Max = 1
left
right
Max = 2
left
right
Max = 2
left
right
Now we see that there is a duplicate in window, so stretch the window.


Max = 2
0
1
2
3
4
5
p
w
w
k
e
w
currentWindowMax < max so no update in max.
left
right


Max = 2
0
1
2
3
4
5
p
w
w
k
e
w
left
right


Max = 3
0
1
2
3
4
5
p
w
w
k
e
w
currentWindowMax > max so update the max.
left
right

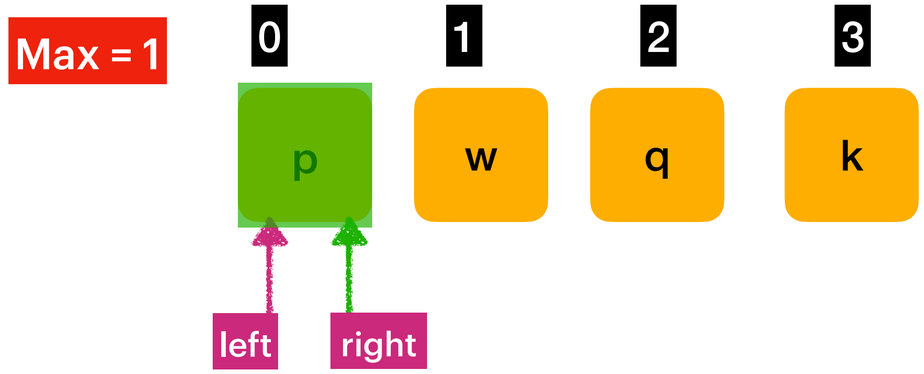




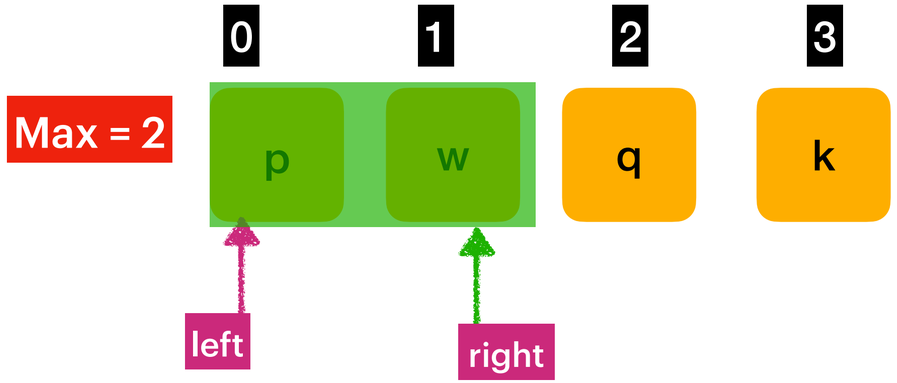
Return max : 3

Time Complexity : O(n^2)
[If all the characters are unique left will not move then
Leads to n^2 comparisons]

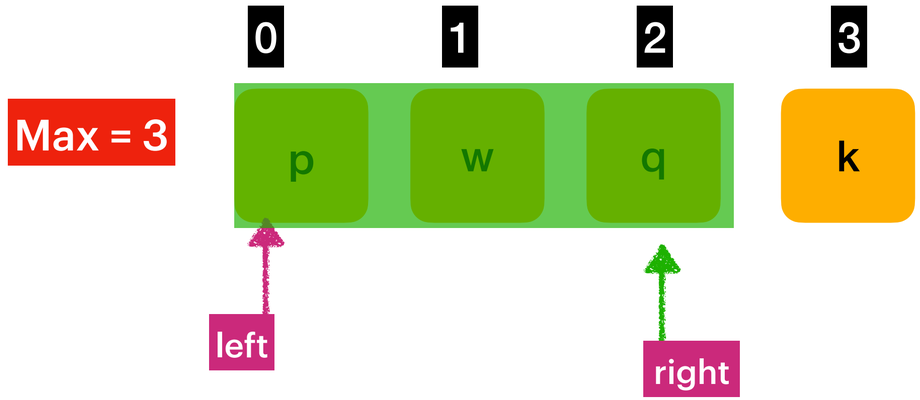
Space Complexity : O(1)

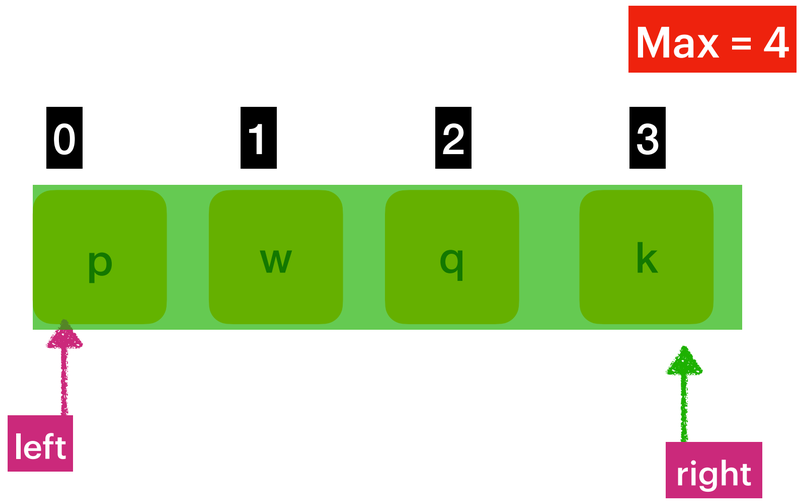


p -> visited —> 4 times
w -> visited —> 3 times
q -> visited —> 2 times
k —> 1 tme
-> n+(n-1)+(n-2)+..+1
= n(n-1)/2 = O(n^2)



If all the characters are unique
then BruteForce Approach leads to O(n^2)
Time Complexity.

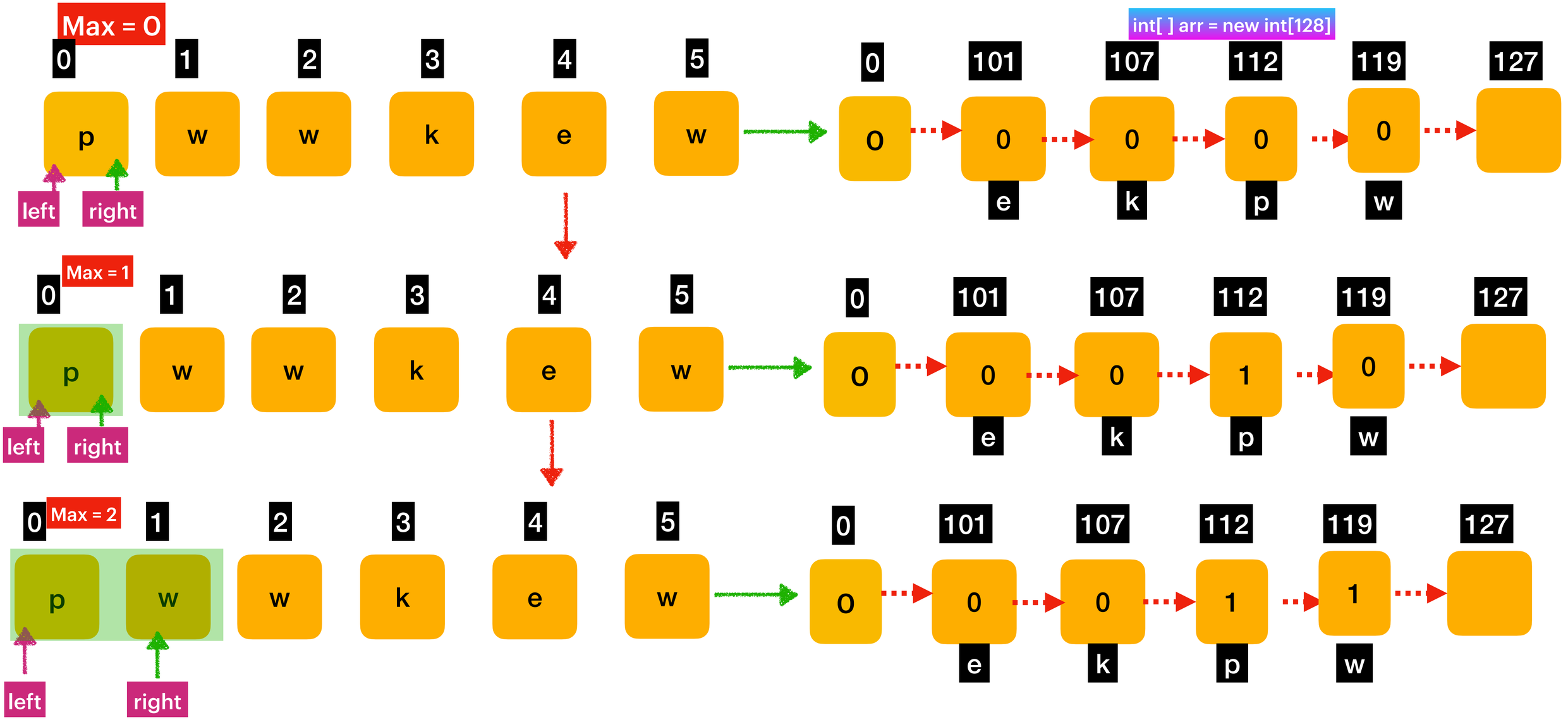


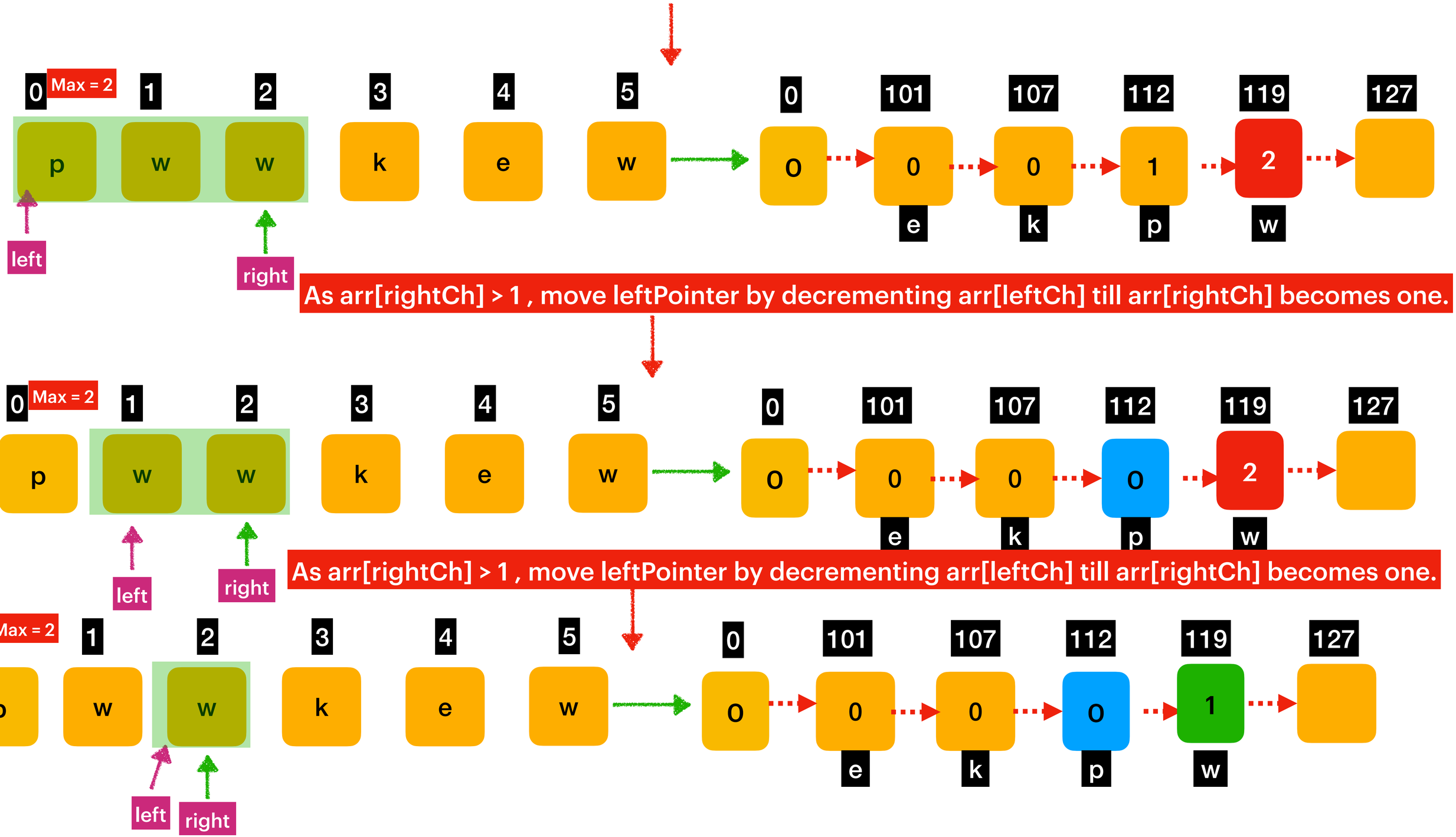


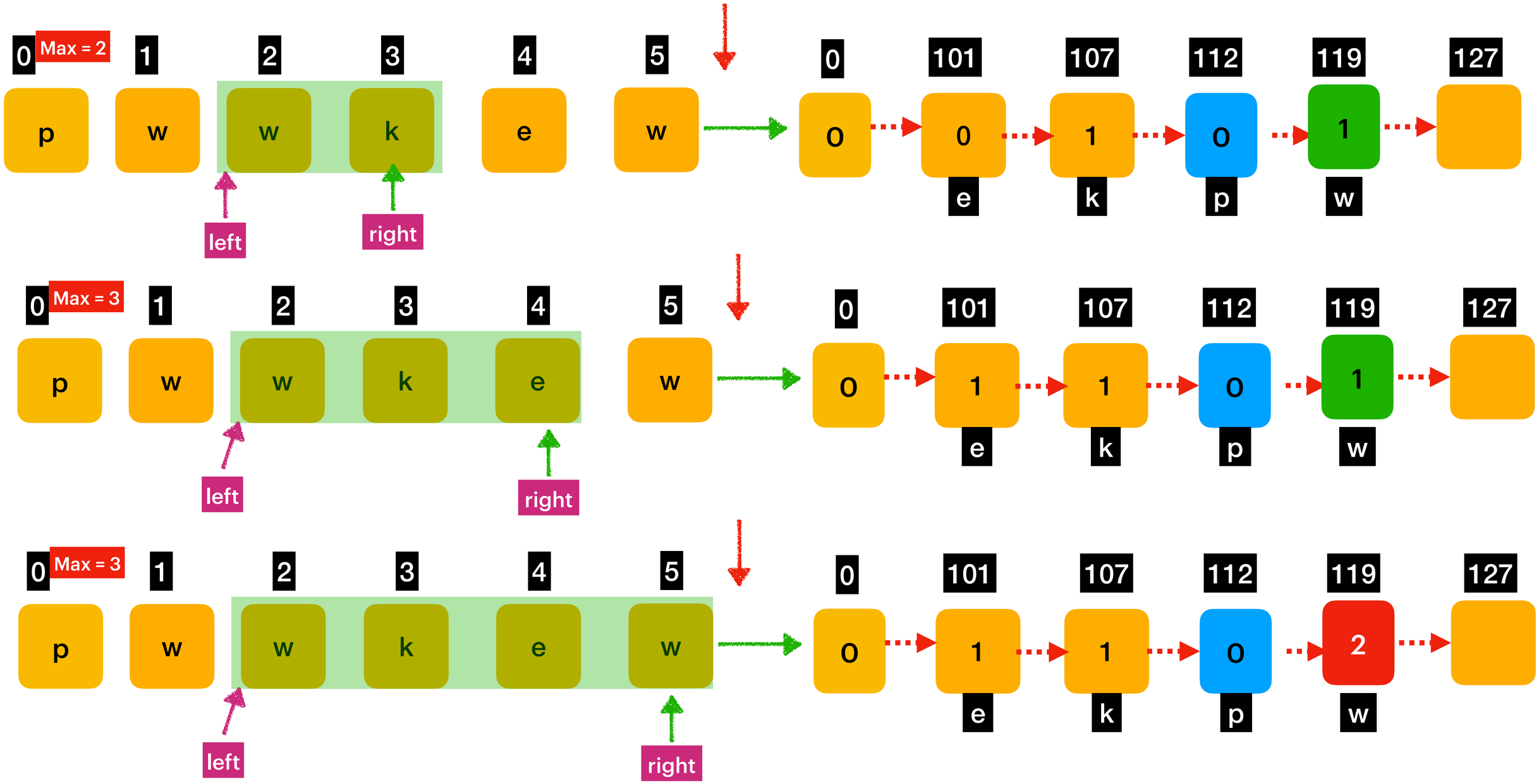
Hashing Approach with Array Index

As per the constraints the input consists of English letters, digits, symbols and spaces.
We can go with array of size 128 to represent each character.

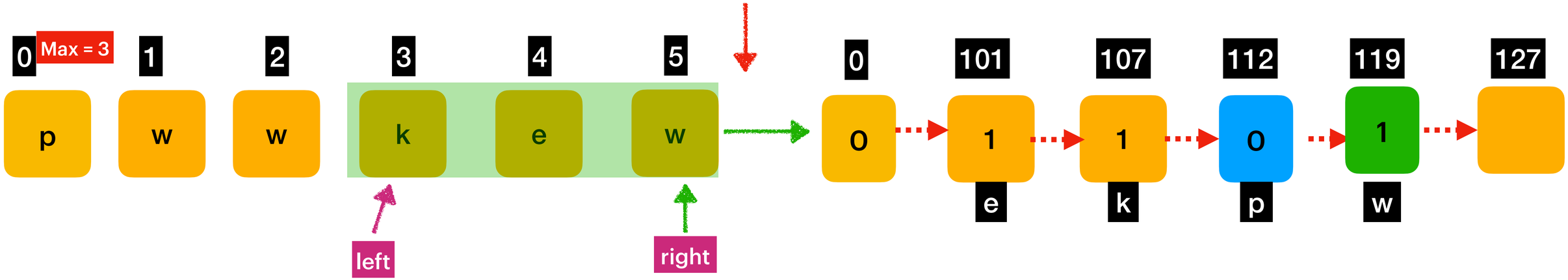
-> On each right move increment the right char ascii index in an array.
-> when the arr[rightCh] > 1 then its a duplicate so move the left pointer by decrementing the arr[leftCh] till arr[rightCh] == 1.




0
Max = 2
1
2
3
4
5
p
w
w
k
e
w
left
right
0
101
107
112
119
127
0
0
0
1
2
e
k
p
w
As arr[rightCh] > 1 , move leftPointer by decrementing arr[leftCh] till arr[rightCh] becomes one.
0
Max = 2
1
2
3
4
5
p
w
w
k
e
w
left
right
0
101
107
112
119
127
0
0
0
0
2
e
k
p
w
As arr[rightCh] > 1 , move leftPointer by decrementing arr[leftCh] till arr[rightCh] becomes one.
0
Max = 2
1
2
3
4
5
p
w
w
k
e
w
left
right
0
101
107
112
119
127
0
0
0
0
1
e
k
p
w



As arr[w] > 1 movie leftPointer by decrementing arr[leftCh] till arr[w] becomes one.

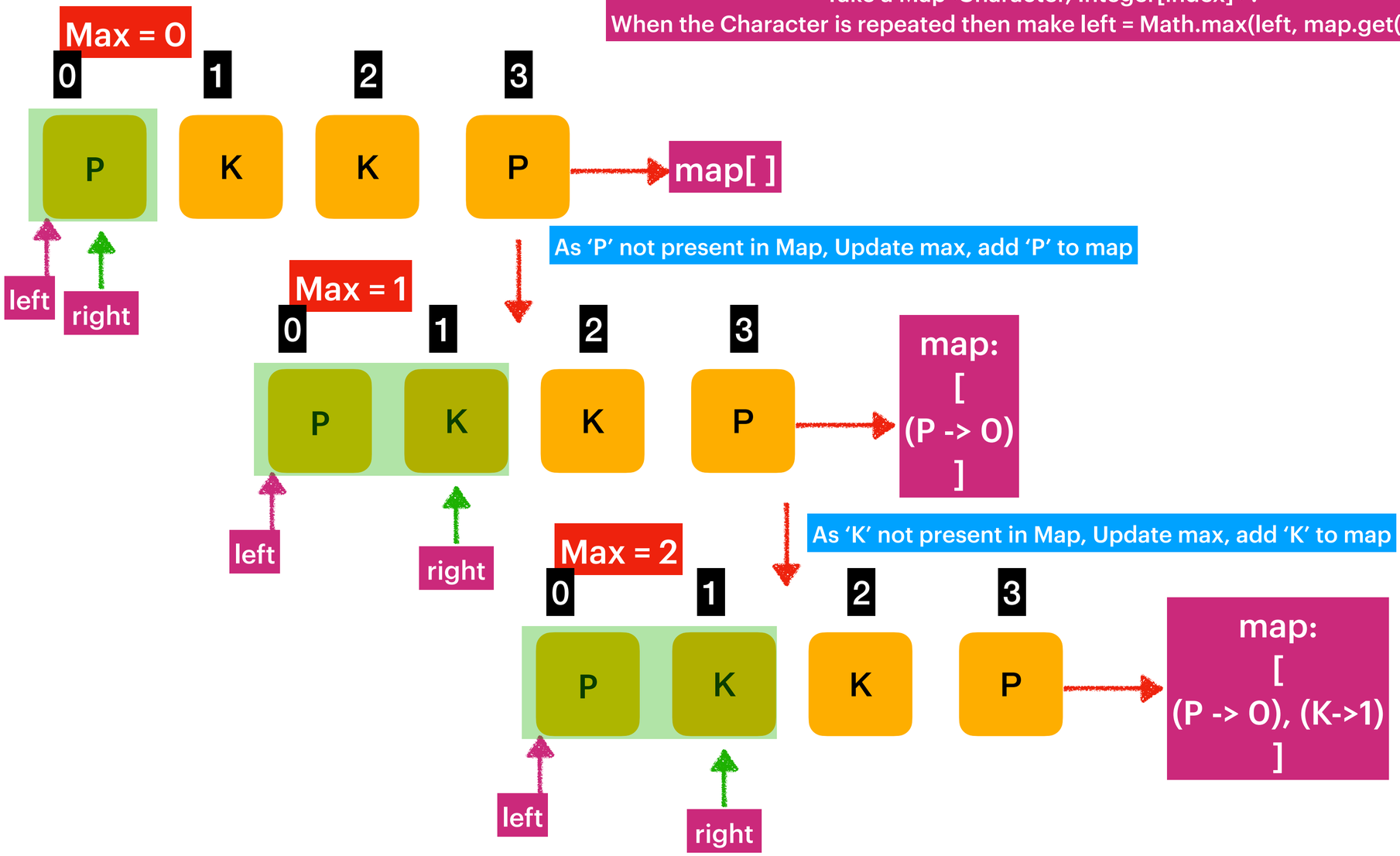


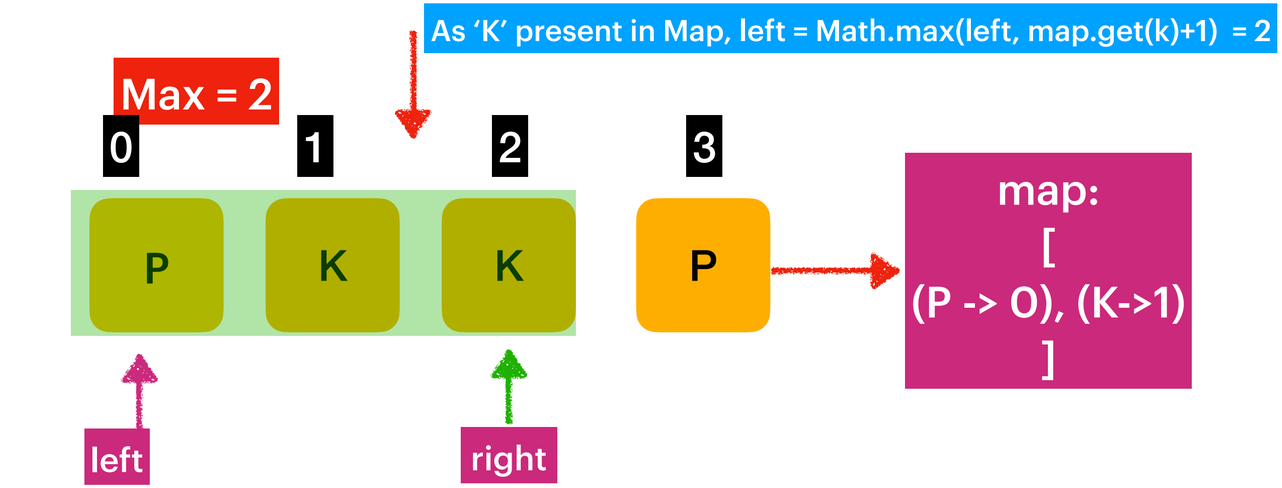
Return max = 3

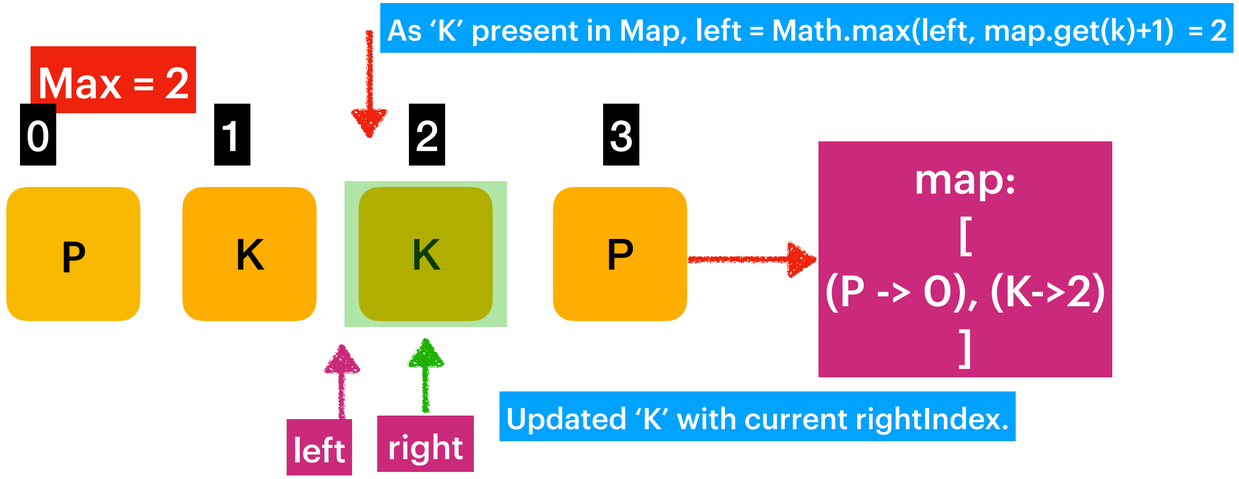
In Worst case each character would be visited twice:
Time Complexity : O(2n) = O(n)
Space Complexity : O(1)
—> Array Size 128 is fixed irrespective of the input.

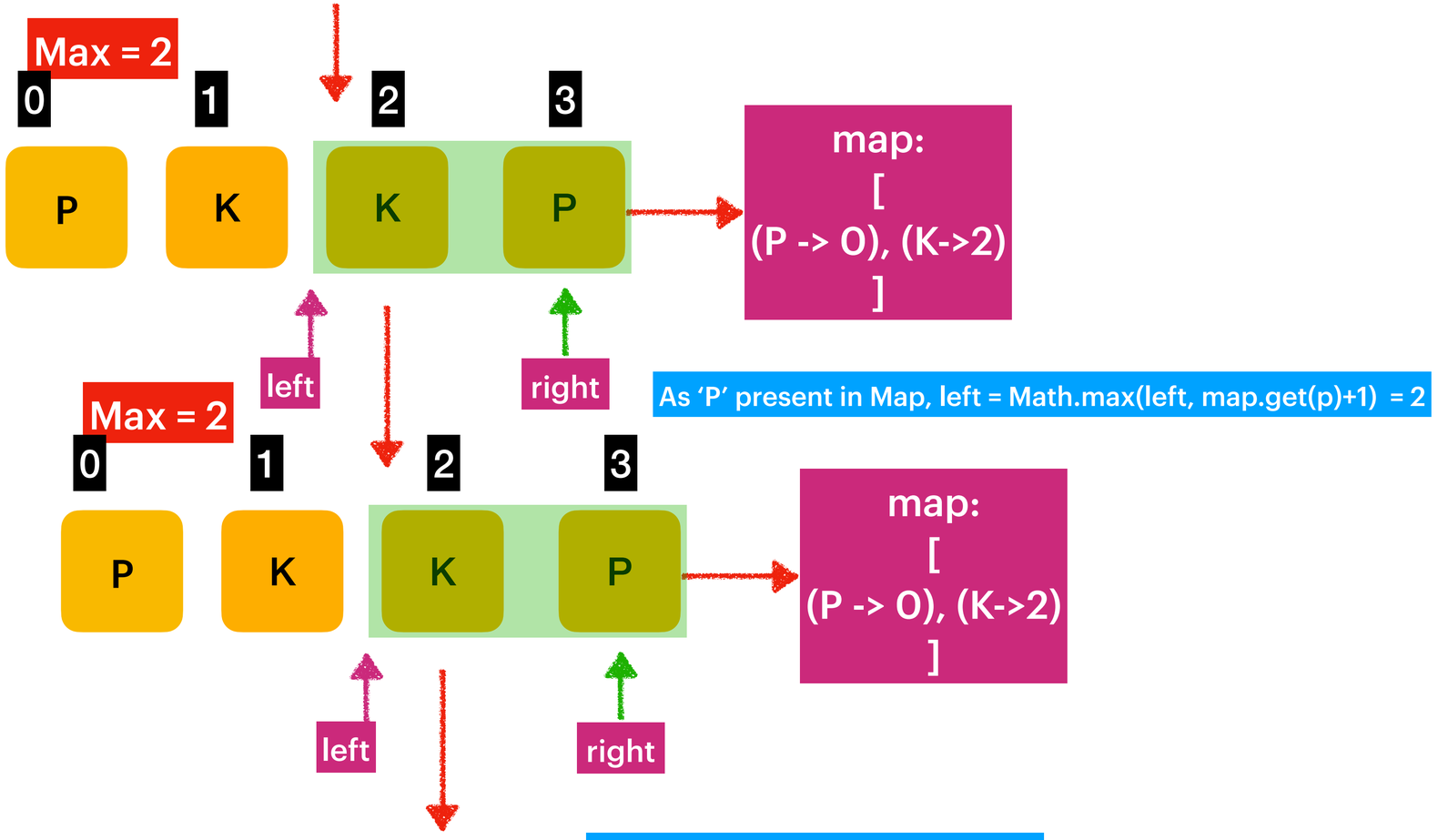
Hashing Approach with Map

Take a Map<Character, Integer[index] >.
When the Character is repeated then make left = Math.max(left, map.get(ch)+1);




As ‘K’ present in Map, left = Math.max(left, map.get(k)+1) = 2
Max = 2
0
1
2
3
P
K
K
P
map:
[
(P -> 0), (K->1)
]
left
right


As ‘K’ present in Map, left = Math.max(left, map.get(k)+1) = 2
Max = 2
0
1
2
3
P
K
K
P
map:
[
(P -> 0), (K->2)
]
left
right
Updated ‘K’ with current rightIndex.



return 2

Time Complexity : O(n)
Space Complexity :Math.min(map.size(), inputLength)