Notebook

August 2, 2024

Standard Template to solve sliding window problems

```
def longestOnes(self, nums: List[int], k: int) -> int:
        left, right = 0, 0
        ans, cntZ = 0, 0
        n = len(nums)

        while right < n:
            if nums[right] == 0: cntZ += 1
            while cntZ > k:
                if nums[left] == 0: cntZ -= 1
                left += 1
            right += 1
            ans = max(ans, right-left)
        return ans
```

[ ]:

```
# https://leetcode.com/problems/sliding-window-maximum/description/
# Sliding Window Maximum

# Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
# Output: [3,3,5,5,6,7]

from collections import deque

class Solution:
    def maxSlidingWindow(self, nums, k):
        ans = []
        dq = deque()

        for i in range(k):
            while dq and nums[dq[-1]] < nums[i]:
                dq.pop()
            dq.append(i)

        for i in range(k, len(nums)):
            ans.append(nums[dq[0]])
            while dq and dq[0] <= i - k:
                dq.popleft()
            while dq and nums[dq[-1]] < nums[i]:
                dq.pop()
            dq.append(i)

        if dq:
            ans.append(nums[dq[0]])

        return ans
```

[ ]:
```
# https://leetcode.com/problems/longest-substring-without-repeating-characters/
↪description/

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

class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        left, right = 0, 0
        ans, n = 0, len(s)
        mp = defaultdict(int)
        while right < n:
            mp[s[right]] += 1
            while left <= right and mp[s[right]] > 1:
                mp[s[left]] -= 1
                left += 1
            right += 1
            ans = max(ans, right-left)
        return ans
```

[ ]:
```
# https://leetcode.com/problems/
↪longest-subarray-of-1s-after-deleting-one-element/
# Longest Subarray of 1's After Deleting One Element

# Input: nums = [0,1,1,1,0,1,1,0,1]     # Output: 5
# Explanation: After deleting the number in position 4, [0,1,1,1,1,1,0,1]␣
↪longest subarray with value of 1's is [1,1,1,1,1].

class Solution:
    def longestSubarray(self, nums: List[int]) -> int:
        left, right = 0, 0
        ans, n = 0, len(nums)
        cntZeroes = 0
        while right < n:
            if nums[right] == 0: cntZeroes += 1
            while left <= right and cntZeroes > 1:
                if nums[left] == 0: cntZeroes -= 1
                left += 1
            right += 1
            ans = max(ans, right-left-1)
        return ans
```

[ ]:
```
# https://leetcode.com/problems/max-consecutive-ones-iii/description/

# Input: nums = [1,1,1,0,0,0,1,1,1,1,0], k = 2
# Output: 6 Explanation: [1,1,1,0,0,1,1,1,1,1,1]

from typing import List
from collections import deque
class Solution:
    def longestOnes(self, nums: List[int], k: int) -> int:
        dq = deque()
        left = 0
        ans = 0
        for right in range(len(nums)):
            if nums[right] == 0:
                dq.append(right)
            if len(dq) > k:  # Case nums[left, right] contains more than k␣
 ↪zeros, move `left` util the subarray has no more than k zeros
                left = dq.popleft() + 1
            ans = max(ans, right - left + 1)
        return ans

    def longestOnes(self, nums: List[int], k: int) -> int:
        left, right = 0, 0
        ans, cntZ = 0, 0
        n = len(nums)

        while right < n:
            if nums[right] == 0: cntZ += 1
            while cntZ > k:
                if nums[left] == 0: cntZ -= 1
                left += 1
            right += 1
            ans = max(ans, right-left)
        return ans

# https://leetcode.com/problems/maximize-the-confusion-of-an-exam/description/
# Either replace all 'T's with 'F's or all 'F's with 'T's. similar to above 0␣
 ↪and 1
# Input: answerKey = "TTFF", k = 2
# Output: 4

class Solution:
    def maxConsecutiveAnswers(self, answerKey: str, k: int) -> int:
        def longestOnes(typ) -> int:
            left, right = 0, 0
            ans, cnt = 0, 0
            n = len(answerKey)

            while right < n:
                if answerKey[right] == typ: cnt += 1
                while cnt > k:
                    if answerKey[left] == typ: cnt -= 1
                    left += 1
                right += 1
                ans = max(ans, right-left)
            return ans

        return max(longestOnes('T'), longestOnes('F'))
```

[4]:
```
# https://leetcode.com/problems/longest-repeating-character-replacement/
 ↪description/
# Take the same approach as above, but instead of counting 0s, we count the␣
 ↪most frequent character in the window

from collections import defaultdict

class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        cnt = defaultdict(int)
        n, ans = len(s), 0
        left, right = 0, 0
        while right < n:
            c = s[right]
            cnt[c] += 1
            maxV = max(cnt.values())
            while right-left+1 > maxV + k:
                cc = s[left]
                cnt[cc] -= 1
                maxV = max(cnt.values())
                left += 1
            right += 1
            ans = max(ans, right-left)

        return ans
```

[3]:
```
### https://leetcode.com/problems/grumpy-bookstore-owner/description

from typing import List
```

```
class Solution:
    def maxSatisfied(self, customers: List[int], grumpy: List[int], X: int) ->␣
↪int:

        # Part 1 requires counting how many customers
        # are already satisfied, and removing them
        # from the customer list.
        already_satisfied = 0
        for i in range(len(grumpy)):
            if grumpy[i] == 0: #He's happy
                already_satisfied += customers[i]
                customers[i] = 0

        # Part 2 requires finding the optinal number
        # of unhappy customers we can make happy.
        best_we_can_make_satisfied = 0
        current_satisfied = 0
        for i, customers_at_time in enumerate(customers):
            current_satisfied += customers_at_time # Add current to rolling␣
↪total
            if i >= X: # We need to remove some from the rolling total
                current_satisfied -= customers[i - X]
            best_we_can_make_satisfied = max(best_we_can_make_satisfied,␣
↪current_satisfied)

        # The answer is the sum of the solutions for the 2 parts.
        return already_satisfied + best_we_can_make_satisfied
```

[ ]:
```
### https://leetcode.com/problems/count-number-of-nice-subarrays/description/
### Exactly K odd numbers = At most K odd numbers - At most (K-1) odd numbers

"""
Input: nums = [2,2,2,1,2,2,1,2,2,2], k = 2
Output: 16
"""

import collections

class Solution:
    def numberOfSubarrays(self, nums: List[int], k: int) -> int:
        def atMostK(nums, k):
            start, end = 0, 0
            ans, n = 0, len(nums)

            while end < n:
                if nums[end] % 2 : k -= 1
                while (k < 0):
```

```
                if nums[start] % 2 : k += 1
                start += 1
            end += 1
            ans += end - start
        return ans

    return atMostK(nums, k) - atMostK(nums, k-1)


### https://leetcode.com/problems/subarrays-with-k-different-integers/
↪description/
### Exactly K different integers = At most K different integers - At most (K-1)␣
↪different integers

"""
Input: nums = [1,2,1,2,3], k = 2
Output: 7
Explanation: Subarrays formed with exactly 2 different integers: [1,2], [2,1],␣
↪[1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2]
"""

class Solution:
    def subarraysWithKDistinct(self, nums: List[int], k: int) -> int:
        def atMostK(nums, k):
            start, end = 0, 0
            ans, n = 0, len(nums)
            mp = collections.defaultdict(int)
            while end < n:
                mp[nums[end]] += 1
                if (mp[nums[end]] == 1): k -= 1
                while (start <= end and k < 0):
                    mp[nums[start]] -= 1
                    if (mp[nums[start]] == 0):
                        k += 1
                    start += 1
                end += 1
                ans += end - start
            return ans

        return atMostK(nums, k) - atMostK(nums, k-1)


### https://leetcode.com/problems/binary-subarrays-with-sum/description/
### Exactly S = At most S - At most (S-1)

"""
Input: nums = [1,0,1,0,1], goal = 2
```

```
Output: 4
Explanation: The 4 subarrays are bolded and underlined below:
[1,0,1,0,1] [1,0,1,0,1] [1,0,1,0,1] [1,0,1,0,1]
"""

class Solution:
    def numSubarraysWithSum(self, nums: List[int], goal: int) -> int:
        def atMostGoal(nums, goal):
            start, end = 0, 0
            ans, n = 0, len(nums)
            curr_sum = 0

            while end < n:
                curr_sum += nums[end]
                while (start <= end and curr_sum > goal):
                    curr_sum -= nums[start]
                    start += 1
                end += 1
                ans += end - start
            return ans

        return atMostGoal(nums, goal) - atMostGoal(nums, goal-1)
```

[ ]:
```
### https://github.com/doocs/leetcode/blob/main/solution/0100-0199/0159.
↪Longest%20Substring%20with%20At%20Most%20Two%20Distinct%20Characters/
↪README_EN.md
###  Longest Substring with At Most Two Distinct Characters

from collections import Counter


class Solution:
    def lengthOfLongestSubstringTwoDistinct(self, s: str) -> int:
        cnt = Counter()
        ans = j = 0
        for i, c in enumerate(s):
            cnt[c] += 1
            while len(cnt) > 2:
                cnt[s[j]] -= 1
                if cnt[s[j]] == 0:
                    cnt.pop(s[j])
                j += 1
            ans = max(ans, i - j + 1)
        return ans

### https://github.com/doocs/leetcode/blob/main/solution/0300-0399/0340.
↪Longest%20Substring%20with%20At%20Most%20K%20Distinct%20Characters/README_EN.
↪md
### Longest Substring with At Most K Distinct Characters

"""
Input: s = "eceba", k = 2
Output: 3
Explanation: The substring is "ece" with length 3.
"""

class Solution:
    def lengthOfLongestSubstringKDistinct(self, s: str, k: int) -> int:
        cnt = Counter()
        n = len(s)
        ans = j = 0
        for i, c in enumerate(s):
            cnt[c] += 1
            while len(cnt) > k:
                cnt[s[j]] -= 1
                if cnt[s[j]] == 0:
                    cnt.pop(s[j])
                j += 1
            ans = max(ans, i - j + 1)
        return ans

# https://leetcode.com/problems/
↪length-of-longest-subarray-with-at-most-k-frequency/description/
#  Length of Longest Subarray With at Most K Frequency

class Solution:
    def maxSubarrayLength(self, nums: List[int], k: int) -> int:
        left, right = 0, 0
        ans, n = 0, len(nums)
        mp = defaultdict(int)
        while right < n :
            mp[nums[right]] += 1
            while left <= right and mp[nums[right]] > k:
                mp[nums[left]] -= 1
                left += 1
            right += 1
            ans = max(ans, right-left)
        return ans
```

[ ]:
```
#### https://leetcode.com/problems/ways-to-split-array-into-good-subarrays/
 ↪description/

"""
Input: [0,1,0,0,1,0,0,1]
Output: 9

Logic:
1. Find the first position wherer it is 1, then initialize ans = 1
2. For next 1, ans = ans * (end - start) % mod
3. one next 1 is found, start = end to get the range of next 1s
"""

class Solution:
    def numberOfGoodSubarraySplits(self, nums: List[int]) -> int:
        start, end = 0, 0
        ans, n = 0, len(nums)
        mod = 10 ** 9 + 7
        while end < n:
            if nums[end] == 1:
                if ans == 0:
                    ans = 1
                else:
                    ans = (ans * (end-start)) % mod
                start = end
            end += 1

        return ans
```

[ ]:
```
### https://leetcode.com/problems/
 ↪longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/

"""
class Solution {
public:
    int longestSubarray(vector<int>& nums, int limit) {
        int ans = 1, n = nums.size();
        map<int, int> mp;
        int start = 0, end = 0;
        while (end < n) {
            mp[nums[end]]++;
            while (start < end and (rbegin(mp)->first - begin(mp)->first) >␣
 ↪limit) {
                if (!--mp[nums[start]]) {
                    mp.erase(nums[start]);
                }
                start++;
            }
            end++;
            ans = max(ans, end-start);
        }

        return ans;
    }
};
"""
```

[ ]:
```
### https://leetcode.com/problems/minimum-number-of-k-consecutive-bit-flips/

"""
class Solution {
public:
    int minKBitFlips(vector<int>& nums, int k) {
        int n = nums.size();
        int flipped = 0;
        int ans = 0;
        vector<int> isFlipped(n, 0);

        for (int i=0; i<n; i++) {
            if (i >= k) {
                flipped ^= isFlipped[i-k];
            }

            if (flipped == nums[i]) {
                if (i+k > n) return -1;
                isFlipped[i] = 1;
                flipped ^= 1;
                ans++;
            }
        }

        return ans;
    }
};
"""
```

[ ]:
```
# https://leetcode.com/problems/find-the-longest-equal-subarray/
# Find the Longest Equal Subarray

# Input: nums = [1,3,2,3,1,3], k = 3   Output: 3

class Solution:
    def longestEqualSubarray(self, nums: List[int], k: int) -> int:
        left, right = 0, 0
        ans, n = 0, len(nums)
        mp = defaultdict(int)
        mxf = 0
        while right < n :
            mp[nums[right]] += 1
            mxf = max(mxf, mp[nums[right]])
            while left <= right and (right-left-mxf+1) > k:
                mp[nums[left]] -= 1
                left += 1
            right += 1

        return mxf
```

[ ]:
```
# https://leetcode.com/problems/
 ↪maximum-beauty-of-an-array-after-applying-operation/description/
# Maximum Beauty of an Array After Applying Operation

# sort the array and then use sliding window to find the maximum beauty

class Solution:
    def maximumBeauty(self, nums: List[int], k: int) -> int:
        left, right = 0, 0
        ans, n = 0, len(nums)
        nums.sort()
        while right < n :
            while left <= right and nums[right] - nums[left] > k*2 :
                left += 1
            right += 1
            ans = max(ans, right-left)
        return ans
```

Count number of subarrays with given conditiion variation

[ ]:
```
# https://leetcode.com/problems/
 ↪count-subarrays-where-max-element-appears-at-least-k-times/description/
# Count Subarrays Where Max Element Appears at Least K Times

# Input: nums = [1,3,2,3,3], k = 2     # Output: 6
# Explanation: The subarrays that contain the element 3 at least 2 times are:␣
 ↪[1,3,2,3], [1,3,2,3,3], [3,2,3], [3,2,3,3], [2,3,3] and [3,3].

# move left pointer until the max element appears at least k times
# add left to the answer since all subarrays ending at right will be valid

class Solution:
    def countSubarrays(self, nums: List[int], k: int) -> int:
        left, right = 0, 0
        ans, n = 0, len(nums)
        mx = max(nums)
        mp = defaultdict(int)
        while right < n :
            mp[nums[right]] += 1
            while left <= right and mp[mx] >= k:
                mp[nums[left]] -= 1
                left += 1
            right += 1
            ans += left
        return ans

# https://leetcode.com/problems/count-complete-subarrays-in-an-array/
 ↪description/
# Count Complete Subarrays in an Array

# Input: nums = [1,3,1,2,2]     # Output: 4
# Explanation: The complete subarrays are the following: [1,3,1,2],␣
 ↪[1,3,1,2,2], [3,1,2] and [3,1,2,2].

# Similar idea as above, add left to ans once all distinct elements are covered

class Solution:
    def countCompleteSubarrays(self, nums: List[int]) -> int:
        left, right = 0, 0
        ans, n = 0, len(nums)
        mp = defaultdict(int)
        distinct = len(set(nums))
        while right < n :
            mp[nums[right]] += 1
            if mp[nums[right]] == 1: distinct -= 1
            while left <= right and distinct == 0:
                mp[nums[left]] -= 1
                if mp[nums[left]] == 0: distinct += 1
                left += 1
            right += 1
            ans += left
        return ans
```