Sliding Window_1

@kbbhatt04

@September 2, 2023

Sliding Window Maximum

- Approach
  - Brute-force
    - Iterate over all windows of size k and find maximum in each window
    - Time Complexity: $O(n * k)$
    - Space Complexity: $O(k)$
  - Optimal
    - Use deque
    - Every time before entering a new element, we first need to check whether the element present at the front is out of bounds of our present window size. If so, we need to pop that out.
    - Also, we need to check from the rear that the element present is smaller than the incoming element. If yes, there's no point storing them and hence we pop them out.
    - Finally, the element present at the front would be our largest element.
    - Time Complexity: $O(n)$
    - Space Complexity: $O(k)$

```
# Python3
# Brute-force Solution
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List
        res = []
        for i in range(len(nums)-k+1):
```

```
            temp = nums[i:i+k]
            res += max(temp),
        return res
```

```
# Python3
# Optimal Solution
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List
        dq = deque()
        res = []

        for i in range(len(nums)):
            if dq and dq[0] == i - k:
                dq.popleft()
            while dq and nums[i] >= nums[dq[-1]]:
                dq.pop()

            dq.append(i)
            if i >= k-1:
                res.append(nums[dq[0]])

        return res
```

```
// C++
// Optimal Solution
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> dq;
        vector<int> ans;

        for (int i = 0; i < nums.size(); i++) {
            if (!dq.empty() && dq.front() == i - k) {
                dq.pop_front();
```

```
            }
            while(!dq.empty() && nums[i] >= nums[dq.back()]) {
                dq.pop_back();
            }

            dq.push_back(i);
            if (i >= k - 1) ans.push_back(nums[dq.front()]);
        }
        return ans;
    }
};
```

Template

- Approach
  - Brute-force
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Optimal
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```