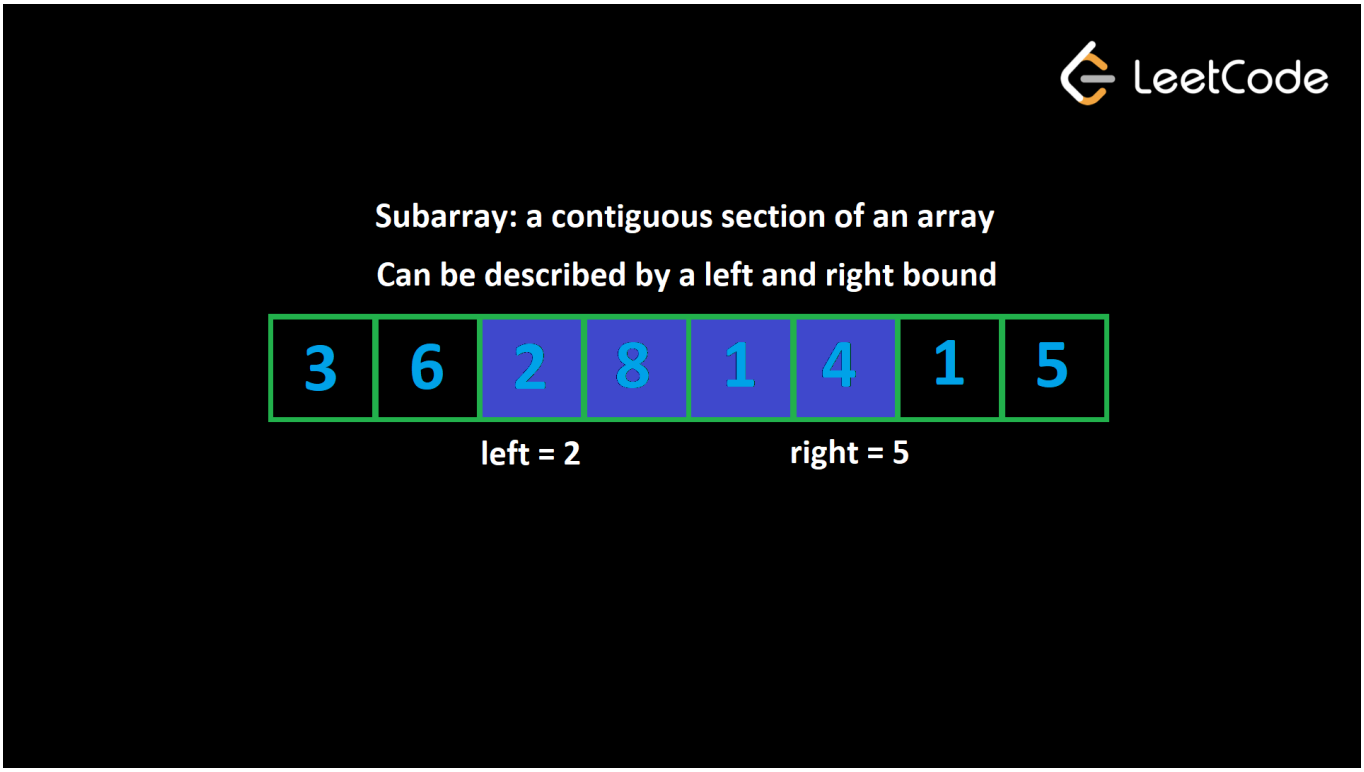
Sliding window

A subarray is aka a window. To be considered a subarray, the elements must be adjacent to each other in the original array and in their original order. A sliding window uses two pointers.



When should we use sliding window?

First, the problem with define criteria that make a subarray "valid". There are 2 components to this:

1. A constraint metric. This is some attribute of a subarray. It could be the sum, the number of unique elements, the frequency of a specific element, etc.
2. A numeric restriction on the constant metric

For example, let's say a problem declares a subarray is valid if it has a sum less than or equal to `10`. The constraint metric here is the sum of the subarray, and the numeric restriction is `<= 10`.

Second, the problem will ask you to find valid subarrays in some way.

1. The most common task you will see if finding the **best** valid subarray. For example, a problem might ask you to find the **longest** valid subarray.
2. Another common task is finding the number of valid subarrays.

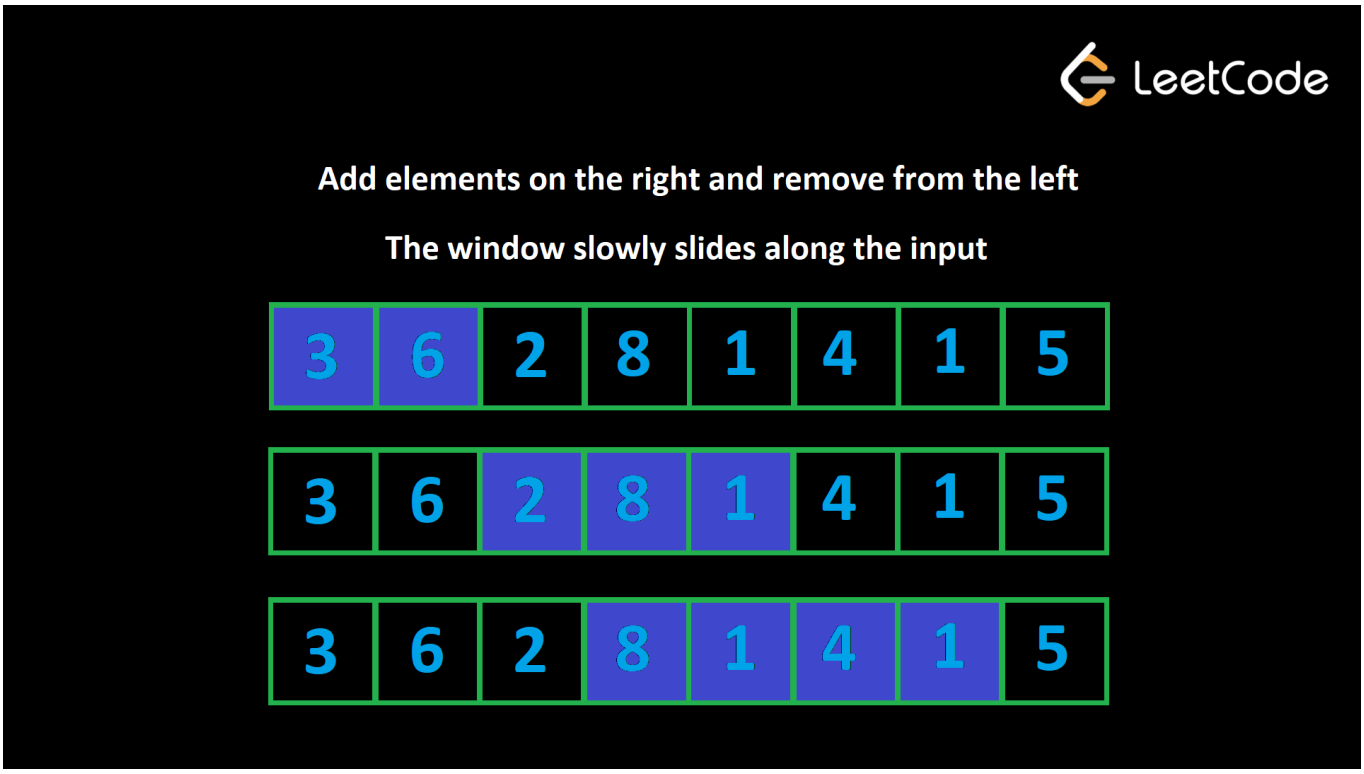
Whenever a problem talks about subarrays. you should figure out if sliding window is a good option by looking for the above characteristics of such problems.

Ex. Find the longest subarray with a sum less than or equal to `k`
Find the longest substring that has at most one "0"
Find the number of subarrays that have a product less than `k`

The idea behind a sliding window is to consider **only** valid subarrays. The `left` and `right` indices define the current subarray under consideration.

Initially, we have `left = right = 0`. To expand our "window", increment `right`. If our subarray becomes invalid, we can "remove" some elements by incrementing `left`.

As we add and remove elements, we are "sliding" our window along the input from left to right. The window's size is constantly changing - it grows as large as it can until it's invalid, and then it shrinks. However, it always slides to the right, until we reach the end.



Let's say we are given a positive integer array `nums` and an integer `k`. We need to find the length of the longest subarray that has a sum less than or equal to `k`. For this example, let `nums = [3, 2, 1, 3, 1, 1]` and `k = 5`.

We start with `left = right = 0`, so our window is the first element only: `[3]`. We can expand to the right until the constraint is broken. This will occur when `left = 0, right = 2`, and our window is `[3, 2, 1]` because the sum if `6` which is greater than `k`. We must now shrink the window from the lft until the constraint is no longer broken. After removing one element, the window becomes valid again: `[2, 1]`.

Why is it correct to remove the `3` and forget about it for the rest of the algorithm? Because the input has only positive integers, a longer subarray directly equals a larger sum. We know that `[3, 2, 1]` already results in a sum that is too large. There is no way for us to ever have a valid window again if we keep this `3` because if we were to add any more elements from the right, the sum would only get larger. That's why we can forget about the `3` for the rest of the algorithm.

Here's some pseudocode for a general template:

```
function fn(arr):
    left = 0
    for (int right = 0; right < arr.length; right++):
        Do some logic to "add" element at arr[right] to window

        while WINDOW_IS_INVALID:
            Do some logic to "remove" element at arr[left] from window
            left++

        Do some logic to update the answer
```

Why is the sliding window efficient?

For an array, how many subarrays are there? If the array has length of `n`, there are `n` subarrays of length `1`. Then there are `n - 1` subarrays of length `2` (every index except the last one can be a starting index), `n - 2` subarrays of length `3` and so on until there is only `1` subarray of length `n`. This means there are

$$n + n - 1 + n - 2 + \ldots + 1 = n(n+1)/2$$

subarrays in total. In terms of time complexity, any algorithm that looks at every subarray will be at least $O(n^2)$, which is usually too slow. A sliding window guarantees a maximum of $2n$ window iterations - the right pointer can move n times in total and the left pointer can move n times in total. This means that if the logic done for each window is $O(1)$, sliding window algorithms run in $O(n)$, which is much faster.

Solution:

```
def find_length(nums, k):
    # curr is the current sum of the window
    left = curr = ans = 0
    for right in range(len(nums)):
        curr += nums[right]
```

```
        while curr > k:
            curr -= nums[left]
            left += 1
        ans = max(ans, right - left + 1)

    return ans
```

Example 2: You are given a binary string `s` (a string containing only `"0"` and `"1"`). You may choose up to one `"0"` and flip it to a `"1"`. What is the length of the longest substring achievable that contains only `"1"`?

For example, given `s = "1101100111"`, the answer is `5`. If you perform the flip at index `2`, the string becomes 1111100111.

Another way to look at this problem is "what is the longest substring that contains at most one `"0"`"? This makes it easy for us to solve with a sliding window where our condition is `window.count("0") <= 1`. We can use an integer `curr` that keeps track of how many `"0"` we currently have in our window.

```
def find_length(s):
    # curr is the current number of zeros in the window
    left = curr = ans = 0
    for right in range(len(s)):
        if s[right] == "0":
            curr += 1
        while curr > 1:
            if s[left] == "0":
                curr -= 1
            left += 1
        ans = max(ans, right - left + 1)

    return ans
```

This solution is $O(n)$ in time like the previous example and $O(1)$ in space.

If a problem asks for **the number of subarrays** that fit some constraint and assuming that all subarrays ending at `right` of a valid subarray ending at `right` are valid, we can still use

sliding window, but we need to use a neat math trick to calculate the number of subarrays.

Let's say that we are using the sliding window algorithm we have learned and currently have a valid window `(left, right)`. How many valid windows **end** at index `right`?

There's the current window `(left, right)`, then `(left + 1, right)`, ..., `(right, right)`.

You can fix the right bound and then choose any value between `left` and `right` inclusive for the left bound. Therefore, the number of valid windows **ending** at index `right` is also equal to the size of the window, `right - left + 1`.

Example 3: 713 - Subarray Product Less Than K

Given an array of positive integers `nums` and an integer `k`, return the number of subarrays where the product of all the elements in the subarray is strictly less than `k`.

For example, given the input `nums = [10, 5, 2, 6], k = 100`, the answer is `8`. The subarrays with products less than `k` are:

```
[10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [5, 2, 6]
```

```
class Solution:
    def numSubarrayProductLessThanK(self, nums: List[int], k: int) -> int:
        if k <= 1:
            return 0

        ans = left = 0
        curr = 1

        for right in range(len(nums)):
            curr *= nums[right]
            while curr >= k:
                curr //= nums[left]
                left += 1
            ans += right - left + 1

        return ans
```

Again the runtime is $O(n)$ and the space complexity is $O(1)$).

In the examples we looked at above, our window size was dynamic. Sometimes, a problem will specify a **fixed** length `k`. These problems are easy because the difference between any two adjacent windows is only two elements (we add one element on the right and remove one element on the left to maintain the length).

Example 4: Given an integer array `nums` and an integer `k`, find the sum of the subarray with the largest sum whose length is `k`.

We can build a window of length `k` and then slide it along the array. Add and remove one element at a time to make the window stays size `k`. If were are adding the value at `i`, then we need to remove the value at `i - k`.

```
def find_best_subarray(nums, k):
    curr = 0
    for i in range(k):
        curr += nums[i]

    ans = curr
    for i in range(k, len(nums)):
        curr += nums[i] - nums[i - k]
        ans = max(ans, curr)

    return ans
```

This runs in $O(n)$ since each update of the window is just an addition and a subtraction and we go through the array linearly. The space used was $O(1)$.

Frequently, sliding window problems also use a hashmap.

SEE PROBLEMS 643, 1004