# 2-3. Sliding window

#_area/coding_test/leetcode/2_arrays_and_strings

> Like two pointers, sliding windows work the same with arrays and strings - the important thing is that they're iterables with ordered elements. For the sake of brevity, the first part of this article up until the examples will be focusing on **arrays**. However, all the logic is identical for strings.
> 투 포인터와 마찬가지로, 슬라이딩 윈도우 역시 배열과 문자열에 똑같이 적용됩니다. 중요한 것은, 두 자료형 모두 순서가 있는 요소들로 구성된 순회 가능한(iterable) 구조라는 점입니다. 간단함을 위해, 본문의 전반부에서는 **배열**에만 초점을 맞추겠지만, 사실 문자열에서도 동일한 로직이 적용됩니다.

Sliding window is another common approach to solving problems related to arrays. A sliding window is actually implemented using two pointers! Before we start, we need to talk about the concept of a **subarray**.
슬라이딩 윈도우(Sliding Window)는 배열과 관련된 문제를 해결할 때 자주 쓰이는 또 다른 기법입니다. 사실, 슬라이딩 윈도우는 내부적으로 투 포인터를 사용하여 구현됩니다! 본격적으로 살펴보기 전에, 먼저 **서브어레이(subarray)** 라는 개념을 짚고 넘어가겠습니다.

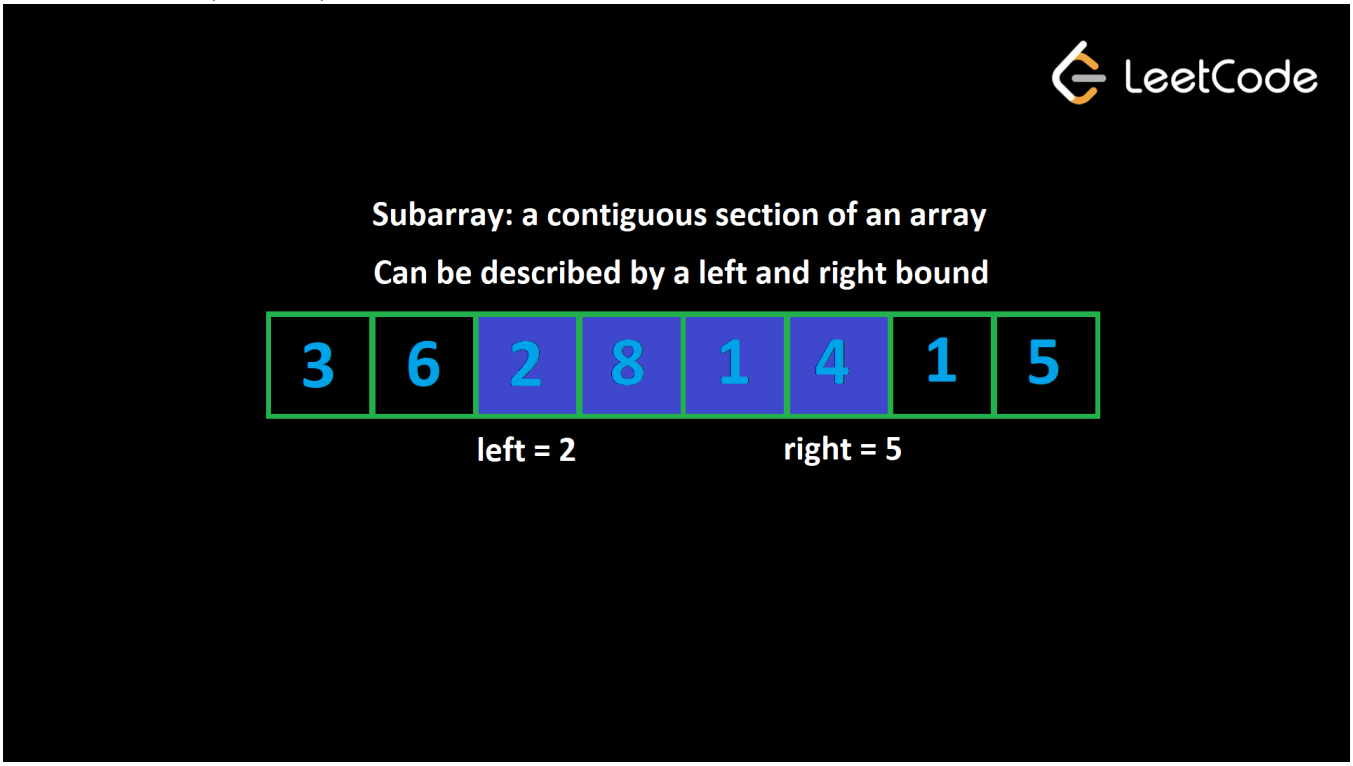
## Subarrays

Given an array, a **subarray** is a contiguous section of the array. All the elements must be adjacent to each other in the original array and in their original order. For example, with the array `[1, 2, 3, 4]`, the subarrays (grouped by length) are:
배열이 주어졌을 때, **서브어레이**란 배열에서 연속된 구간을 말합니다. 모든 요소는 원래 배열에서 서로 인접해 있어야 하며, 원래 순서를 유지해야 합니다. 예를 들어 `[1, 2, 3, 4]`라는 배열이 있을 때, (길이별로 묶어서 보면) 가능한 서브어레이들은 다음과 같습니다:

- `[1]`, `[2]`, `[3]`, `[4]`
- `[1, 2]`, `[2, 3]`, `[3, 4]`
- `[1, 2, 3]`, `[2, 3, 4]`
- `[1, 2, 3, 4]`

**A subarray can be defined by two indices, the start and end**. For example, with `[1, 2, 3, 4]`, the subarray `[2, 3]` has a starting index of `1` and an ending index of `2`. Let's call the starting index the **left bound** and the ending index the **right bound**. Another name for subarray in this context is "window".
**서브어레이는 시작 인덱스와 끝 인덱스, 이렇게 두 개의 인덱스로 정의할 수 있습니다.** 예를 들어 `[1, 2, 3, 4]`에서 `[2, 3]`이라는 서브어레이는 시작 인덱스 `1`과 끝 인덱스 `2`를 가집니다. 여기서 시작 인덱스를 **왼쪽 경계(left**

**bound)**, 끝 인덱스를 **오른쪽 경계(right bound)** 라고 부르겠습니다. 이 상황에서 서브어레이를 가리키는 또 다른 용어가 바로 "윈도우(window)"입니다.




---

## When should we use sliding window?

There is a very common group of problems involving subarrays that can be solved efficiently with sliding window. Let's talk about how to identify these problems.
서브어레이(subarray)에 대한 문제들 중 슬라이딩 윈도우로 효율적으로 해결할 수 있는 경우가 자주 있습니다. 이 문제들을 어떻게 식별하는지 알아봅시다.

**First**, the problem will either explicitly or implicitly define criteria that make a subarray "valid". There are 2 components regarding what makes a subarray valid:

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, or any other attribute.
2. A numeric restriction on the constraint metric. This is what the constraint metric should be for a subarray to be considered valid.

**첫째**, 문제에서 서브어레이가 "유효(valid)"하다고 간주되는 기준을 명시적으로 혹은 암묵적으로 제시합니다. 서브어레이가 유효해지는 조건에는 두 가지 요소가 있습니다:

1. 제약 지표(constraint metric). 이는 서브어레이의 어떤 속성을 의미합니다. 예를 들어 합계, 고유 원소의 개수, 특정 원소의 빈도 등이 될 수 있습니다.
2. 이 제약 지표에 대한 수치적 제한. 서브어레이가 유효하다고 간주되기 위해, 해당 제약 지표가 만족해야 하는 범위

혹은 조건입니다.

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`. A subarray is considered valid if its constraint metric conforms to the numeric restriction, i.e. the sum is less than or equal to `10`.
예를 들어, 어떤 문제에서 서브어레이의 합이 `10` 이하일 때 유효하다고 한다면, 여기서 제약 지표는 "서브어레이의 합"이고, 수치적 제한은 "`<= 10`"이 됩니다. 즉, 합이 `10` 이하인 서브어레이만 유효하다고 간주합니다.

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

1. The most common task you will see is finding the **best** valid subarray. The problem will define what makes a subarray **better** than another. For example, a problem might ask you to find the **longest** valid subarray.
2. Another common task is finding the number of valid subarrays. We will take a look at this later in the article.

**둘째**, 문제에서 유효한 서브어레이를 어떤 방식으로든 찾으라고 요구합니다.

1. 가장 흔한 작업은 **가장 좋은(best)** 유효 서브어레이를 찾는 것입니다. 문제에서는 서브어레이 간 우열을 평가하는 기준을 따로 정의해 둡니다. 예를 들어, 가장 **긴(longest)** 유효 서브어레이를 찾으라는 문제가 있을 수 있습니다.
2. 또 다른 작업으로, 유효한 서브어레이의 개수를 구하라는 요구가 있을 수 있습니다. 이것은 뒤에서 다시 살펴보겠습니다.

> Whenever a problem description talks about subarrays, you should figure out if sliding window is a good option by analyzing the problem description. If you can find the things mentioned above, then it's a good bet.
> 문제에서 서브어레이에 대해 언급하고 있다면, 위에서 설명한 사항들을 확인함으로써 슬라이딩 윈도우가 적절한지 판단해 보는 것이 좋습니다. 위 요소들을 확인할 수 있다면, 슬라이딩 윈도우가 유효한 방법일 가능성이 큽니다.

Here is a preview of some of the example problems that we will look at in this article, to help you better understand what sliding window problems look like:

- 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`

다음은 슬라이딩 윈도우 문제를 좀 더 잘 이해할 수 있도록 예시 문제들을 간단히 살펴본 것입니다.

- 합이 `k` 이하인 서브어레이 중 가장 긴 것을 찾아라
- `"0"`이 최대 한 개만 포함된 가장 긴 부분 문자열(substring)을 찾아라

- 곱이 `k` 미만인 서브어레이의 개수를 구하라

---

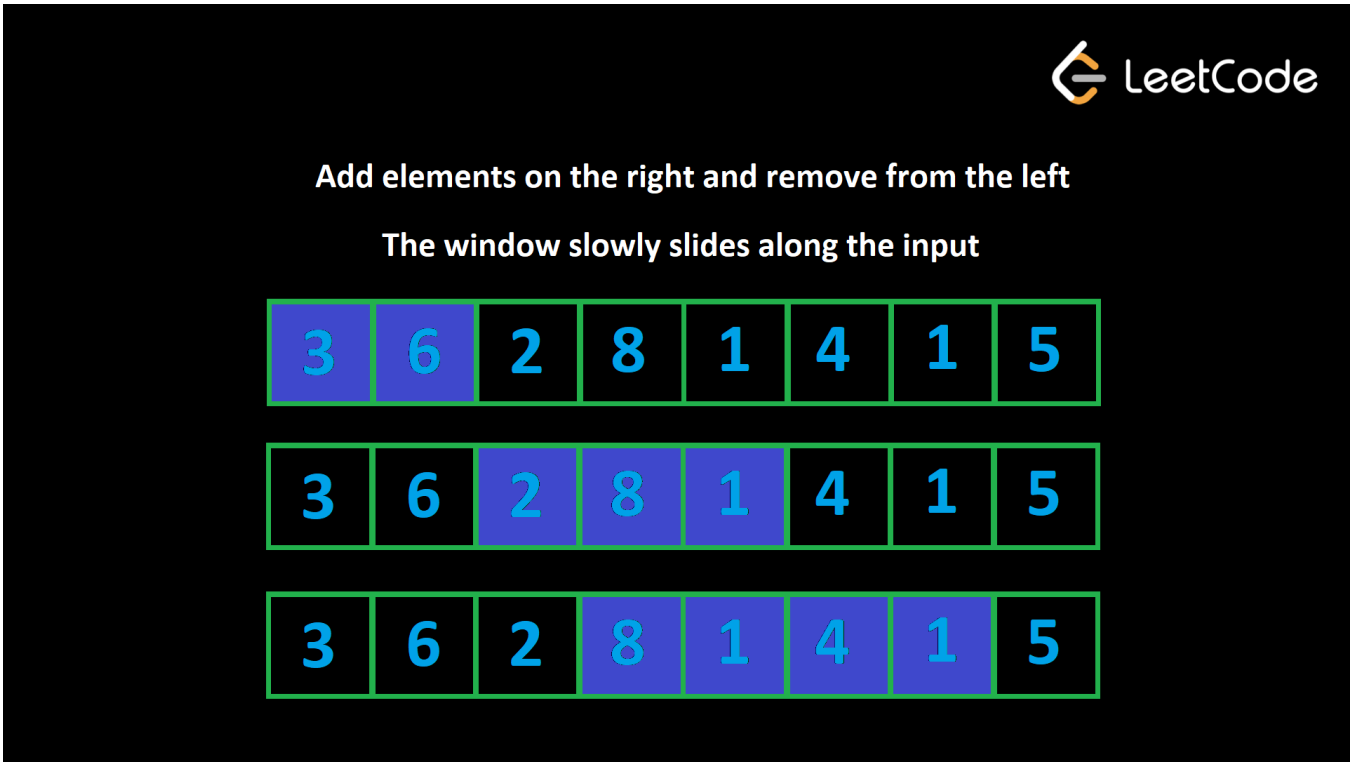
## The algorithm

The idea behind a sliding window is to consider **only** valid subarrays. Recall that a subarray can be defined by a left bound (the index of the first element) and a right bound (the index of the last element). In sliding window, we maintain two variables `left` and `right`, which at any given time represent the **current subarray** under consideration.
슬라이딩 윈도우의 핵심 아이디어는 **유효한 서브어레이만** 고려한다는 것입니다. 서브어레이는 왼쪽 경계(첫 요소의 인덱스)와 오른쪽 경계(마지막 요소의 인덱스)로 정의될 수 있습니다. 슬라이딩 윈도우에서는 `left`와 `right`라는 두 변수를 유지하며, 현재 시점에서 고려 중인 **서브어레이**(윈도우)를 나타냅니다.

Initially, we have `left = right = 0`, which means that the first subarray we look at is just the first element of the array on its own. We want to expand the size of our "window", and we do that by incrementing `right`. When we increment `right`, this is like "adding" a new element to our window.
처음에는 `left = right = 0`으로 시작하며, 이는 배열의 첫 번째 요소만 포함하는 서브어레이를 의미합니다. "윈도우"의 크기를 늘리고 싶다면, `right`를 증가시키면 됩니다. `right`가 증가한다는 것은 윈도우에 새 요소를 "추가"하는 것과 같습니다.

But what if after adding a new element, the subarray becomes invalid? We need to "remove" some elements from our window until it becomes valid again. To "remove" elements, we can increment `left`, which shrinks our window.
그런데 새 요소를 추가했을 때 서브어레이가 유효 범위를 벗어나면 어떻게 될까요? 이럴 때는 윈도우가 다시 유효해질 때까지 일부 요소를 "제거"해야 합니다. 즉, `left`를 증가시켜 윈도우의 왼쪽 부분을 줄이는 방식으로 요소를 제거할 수 있습니다.

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 along to the right, until we reach the end of the input.
이처럼 요소를 추가하고 제거하면서, 우리는 입력 배열을 왼쪽에서 오른쪽으로 "슬라이딩"하게 됩니다. 윈도우의 크기는 계속해서 바뀌는데, 가능한 한도까지 확장하다가 유효 범위를 벗어나면 다시 줄이는 과정을 반복합니다. 다만, 윈도우는 배열의 끝에 도달할 때까지 오른쪽으로만 이동한다는 점이 특징입니다.




To explain why this algorithm works, let's look at a specific example. Let's say that 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`.
이 알고리즘이 왜 유효한지 설명하기 위해, 구체적인 예시를 살펴봅시다. 양의 정수 배열 `nums`와 정수 `k`가 주어졌고, 합이 `k` 이하인 가장 긴 서브어레이의 길이를 구해야 한다고 해봅시다. 예를 들어 `nums = [3, 2, 1, 3, 1, 1]`, `k = 5`라고 하겠습니다.

Initially, we have `left = right = 0`, so our window is only the first element: `[3]`. Now, let's expand to the right until the constraint is broken. This will occur when `left = 0, right = 2`, and our window is: `[3, 2, 1]`. The sum here is `6`, which is greater than `k`. We must now shrink the window from the left until the constraint is no longer broken. After removing one element, the window becomes valid again: `[2, 1]`.
처음에 `left = right = 0`이므로, 윈도우는 `[3]`만 포함합니다. 여기서 제약이 깨질 때까지 오른쪽으로 확장해 봅시다. `left = 0`, `right = 2`일 때, 윈도우가 `[3, 2, 1]`이 되고, 이때 합은 `6`이 되어 `k`를 초과합니다. 따라서 윈도우가 다시 유효해질 때까지 왼쪽에서 요소를 제거해야 합니다. 한 요소를 제거하면 윈도우는 `[2, 1]`이 되어 합이 다시 `k` 이하가 됩니다.

Why is it correct to remove this `3` and forget about it for the rest of the algorithm? Because the input only has 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.
왜 이 `3`을 제거하고, 이후 알고리즘에서 완전히 배제해도 괜찮을까요? 왜냐하면 이 배열은 양의 정수로만 구성되어 있어, 서브어레이 길이가 길어지면 합도 무조건 커지기 때문입니다. 이미 `[3, 2, 1]`에서 합이 너무 커진 것을 확인했으니, 이 `3`을 계속 포함하고 있는 한 오른쪽 요소를 더 추가할 경우 합이 더 커질 뿐입니다. 그래서 나머지 알고리즘 과정에서는 이 `3`을 다시 고려할 이유가 없습니다.

---

## Implementation

Now that you have an idea of how sliding window works, let's talk about how to implement it. For this section, we will use the previous example (find the longest subarray with a sum less than or equal to `k`).
슬라이딩 윈도우가 어떻게 작동하는지 알았다면, 이제 이를 구현하는 방식을 살펴봅시다. 여기서는 이전에 예시로 들었던, 합이 `k` 이하인 가장 긴 서브어레이를 찾는 문제를 예로 들어 설명합니다.

As described above, we need to identify a **constraint metric**. In our example, the constraint metric is the sum of the window. How do we keep track of the sum of the window as elements are added and removed? One way that we could do it is by keeping the window in a separate array. When we add elements from the right, we add them to our array. When we remove elements from the left, we remove the corresponding elements from the array. This way, we can always find the sum of our current window just by summing the elements in the separate array.
앞서 살펴본 것처럼, 먼저 **제약 지표**를 정해야 합니다. 여기서는 윈도우(서브어레이)의 합을 제약 지표로 삼습니다. 그렇다면, 요소가 추가되거나 제거될 때마다 이 윈도우의 합을 어떻게 추적할 수 있을까요? 한 가지 방법은 윈도우에 속한 요소를 별도의 배열로 저장하는 것입니다. 오른쪽에서 요소를 추가할 때마다 별도 배열에도 추가하고, 왼쪽에서 요소를 제거할 때마다 별도 배열에서도 제거하는 것이죠. 이렇게 하면, 별도 배열에 담긴 요소들의 합을 매번 구함으로써 현재 윈도우의 합을 알 수 있습니다.

This is very inefficient as removing elements and finding the sum of the window will be (O(n)) operations. How can we do better?
하지만 이 방법은 매우 비효율적입니다. 요소를 제거하거나 별도 배열의 합을 매번 구하는 과정이 (O(n)) 연산이 되기 때문입니다. 더 효율적인 방법은 없을까요?

We don't actually need to store the window in a separate array. All we need is some variable, let's call it `curr`, that keeps track of the current sum. When we add a new element from the right, we just do `curr += nums[right]`. When we remove an element from the left, we just do `curr -= nums[left]`. This way, all operations are done in (O(1)).

사실 윈도우를 별도의 배열로 보관할 필요는 없습니다. 단지 현재 윈도우의 합을 추적할 변수 하나만 있으면 됩니다. 이를 `curr`라고 합시다. 오른쪽에서 새 요소를 추가할 때는 `curr += nums[right]`, 왼쪽에서 요소를 제거할 때는 `curr -= nums[left]`만 수행하면 됩니다. 이렇게 하면 모든 연산이 (O(1))에 이뤄지게 됩니다.

Next, how do we move the pointers `left` and `right`? Remember, we want to keep expanding our window, and the window always slides to the right - it just might shrink a few times in between. Because `right` is always moving forward, we can use a for loop to iterate `right` over the input. In each iteration of the for loop, we will be adding the element `nums[right]` to our window.
그렇다면 `left`와 `right`는 어떻게 이동할까요? 기본적으로 우리는 윈도우를 계속 확장하고 싶고, 윈도우는 오른쪽으로만 이동합니다. `right`는 항상 앞으로 이동하므로, `for` 루프를 사용해 `right`를 배열 끝까지 순회하게 만들 수 있습니다. `for` 루프의 각 반복마다 `nums[right]`라는 요소를 윈도우에 추가하게 되는 셈입니다.

What about `left`? When we move `left`, we are shrinking our window. We only shrink our window when it becomes invalid. By maintaining `curr`, we can easily tell if the current window is valid by checking the condition `curr <= k`. When we add a new element and the window becomes invalid, we may need to remove multiple elements from the left. For example, let's say `nums = [1, 1, 1, 3]` and `k = 3`. When we arrive at the `3` and add it to the window, the window becomes invalid. We need to remove three elements from the left before the window becomes valid again.
그렇다면 `left`는 어떻게 움직일까요? `left`를 이동시키면 윈도우는 줄어들게 됩니다. 윈도우가 유효 범위를 벗어났을 때만 윈도우를 줄이면 되죠. `curr` 값을 통해 `curr <= k`인지 확인함으로써 윈도우가 유효한지 쉽게 판단할 수 있습니다. 만약 새 요소를 추가해서 윈도우가 유효 범위를 벗어났다면, 왼쪽에서 여러 요소를 제거해야 할 수도 있습니다. 예를 들어 `nums = [1, 1, 1, 3]`, `k = 3`일 때, `3`을 윈도우에 추가하면 윈도우가 유효 범위를 초과합니다. 윈도우가 다시 유효해지려면 왼쪽에서 세 개의 요소를 제거해야 할 것입니다.

This suggests that we should use a while loop to perform the removals. The condition will be `while (curr > k)` (while the window is invalid). To perform the removals, we do `curr -= nums[left]` and then increment `left` in each iteration of the while loop.
이로부터, 요소 제거 시에는 `while` 루프가 필요하다는 사실을 알 수 있습니다. 즉, `while (curr > k)`인 동안(윈도우가 유효하지 않은 동안)은 계속해서 왼쪽 요소를 제거해야 합니다. 이때는 `curr -= nums[left]`와 같이 현재 합에서 왼쪽 요소를 빼 주고, 이어서 `left`를 증가시키면 됩니다.

Finally, how do we update the answer? In each for loop iteration, after the while loop, the current window is valid. We can write code here to update the answer. The formula for the length of a window is `right - left + 1`.
마지막으로, 정답(가장 긴 유효 윈도우의 길이)을 어떻게 갱신할까요? `for` 루프의 각 반복에서 `while` 루프를 끝낸 뒤에는, 윈도우가 유효 상태로 돌아옵니다. 이 시점에서 정답을 갱신하면 됩니다. 윈도우의 길이는 `right - left + 1`로 구할 수 있습니다.

Here's some pseudocode that puts it all together:
위 과정을 종합한 의사 코드는 다음과 같습니다:

```
function fn(nums, k):
    left = 0
    curr = 0
    answer = 0
    for (int right = 0; right < nums.length; right++):
        curr += nums[right]
        while (curr > k):
            curr -= nums[left]
            left++

        answer = max(answer, right - left + 1)

    return answer
```

Here's some pseudocode for a general template:
이것은 좀 더 일반화한 템플릿 의사 코드입니다:

```
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 sliding window efficient?

For any array, how many subarrays are there? If the array has a 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 `∑ (k=1 .. n) k = n·(n+1) / 2` subarrays (it's the partial sum of this series). 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 and the left pointer can move (n) times. This means if the logic done for each window is (O(1)), sliding window algorithms run in (O(n)), which is **much** faster.
어떤 길이 `n`인 배열에서 가능한 서브어레이의 수를 생각해 봅시다. 길이가 1인 서브어레이는 총 `n`개, 길이가 2인 서브어레이는 `n - 1`개, ... 이런 식으로 합하면 `∑ (k=1 .. n) k = n·(n+1) / 2`개가 됩니다. 따라서 모든 서브어레이를 전부 확인하는 알고리즘은 적어도 (O(n^2))가 소요되어, 보통은 너무 느립니다. 반면, 슬라이딩 윈도우 기법은 최대 (2n)번의 윈도우 이동만을 보장합니다. 즉, 오른쪽 포인터는 최대 `n`번, 왼쪽 포인터도 최대 `n`번 이동할 수 있습니다. 따라서 각 윈도우에 대한 연산이 (O(1))에 이뤄진다면, 전체 알고리즘은 (O(n)) 시간 안에 동작하므로 훨씬 빠릅니다.

> You may be thinking: there is a while loop inside of the for loop, isn't the time complexity (O(n^2))? The reason it is still (O(n)) is that the while loop can only iterate (n) times in total for the entire algorithm (`left` starts at `0`, only increases, and never exceeds `n`). If the while loop were to run `n` times on one iteration of the for loop, that would mean it wouldn't run at all for all the other iterations of the for loop. This is what we refer to as **amortized analysis** - even though the worst case for an iteration inside the for loop is (O(n)), it averages out to (O(1)) when you consider the entire runtime of the algorithm.
> "for" 루프 안에 "while" 루프가 있으니 시간 복잡도가 (O(n^2))가 되지 않을까 생각할 수도 있습니다. 하지만 실제로는 `while` 루프가 알고리즘 전 과정에서 최대 `n`번만 실행될 수 있기 때문에 (O(n))을 유지합니다. `left`는 0에서 시작해 증가만 하며, `n`을 넘어서지 않기 때문입니다. 어떤 반복에서 `while` 루프가 `n`번 도는 경우가 있었다면, 다른 반복에서는 거의 돌지 않게 됩니다. 이러한 방식을 **암묵적(분할) 분석(amortized analysis)**이라고 부르며, 단일 반복의 최악 시간 복잡도가 (O(n))이라고 해도 전체적으로 보면 평균적으로 (O(1))로 귀결됩니다.

Now let's look at some sliding window examples.
이제 슬라이딩 윈도우를 적용한 예시들을 살펴봅시다.

---

> **Example 1:** Given an array of positive integers `nums` and an integer `k`, find the length of the longest subarray whose sum is less than or equal to `k`. This is the problem we have been talking about above. We will now formally solve it.
> **예제 1:** 양의 정수 배열 `nums`와 정수 `k`가 주어졌을 때, 합이 `k` 이하인 가장 긴 서브어레이의 길이를 구하시오. 우리가 지금까지 계속 살펴봤던 문제이며, 이제 이것을 공식적으로 풀어보겠습니다.

Let's use an integer `curr` that tracks the sum of the current window. Since the problem wants subarrays whose sum is less than or equal to `k`, we want to maintain `curr <= k`. Let's look at an example where `nums = [3, 1, 2, 7, 4, 2, 1, 1, 5]` and `k = 8`.
현재 윈도우의 합을 추적할 정수 변수 `curr`를 둡니다. 이 문제에서는 합이 `k` 이하인 서브어레이를 구해야 하므로, `curr <= k`를 유지하도록 하면 됩니다. 예를 들어 `nums = [3, 1, 2, 7, 4, 2, 1, 1, 5]`, `k = 8`인 경우를 살펴봅시다.

The window starts empty, but we can grow it to `[3, 1, 2]` while maintaining the constraint. However, after adding the `7`, the window's sum becomes too large. We need to tighten the window until the sum is below `8` again, which doesn't happen until our window looks like `[7]`. When we try to add the next element, our window again becomes too large, and we need to remove the `7` which means we have `[4]`. We can now grow the window until it looks like `[4, 2, 1, 1]`, but adding the next element makes the sum too large. We remove elements from the left until it fits the constraint again, which happens at `[1, 1, 5]`. The longest subarray we found was `[4, 2, 1, 1]` which means the answer is `4`.
초기에 윈도우는 비어 있지만, 조건을 만족하며 `[3, 1, 2]`까지 확장할 수 있습니다. 그러나 `7`을 추가하면 윈도우의 합이 너무 커집니다. 합이 다시 `8` 이하가 되도록 윈도우를 줄여야 하며, 결국 `[7]`만 남겨야 조건을 만족하게 됩니다. 다음 요소를 추가해보면 또다시 합이 너무 커지므로, 이번에는 `[7]`을 제거해 `[4]`로 만듭니다. 이제 `[4, 2, 1, 1]`까지 윈도우를 다시 확장할 수 있지만, 다음 요소를 추가하면 또 합이 커집니다. 왼쪽에서 요소들을 제거해 조건을 다시 만족하게 만든 결과는 `[1, 1, 5]`입니다. 이 과정에서 찾은 가장 긴 서브어레이는 `[4, 2, 1, 1]`이며, 길이는 `4`입니다.

When we add an element to the window by moving the right bound, we just do `curr += value`. When we remove an element from the window by moving the left bound, we just do `curr -= value`. We should remove elements so long as `curr > k`.
오른쪽 경계를 이동해 요소를 추가할 때는 `curr += value`, 왼쪽 경계를 이동해 요소를 제거할 때는 `curr -= value`만 수행하면 됩니다. 만약 `curr`가 `k`를 초과한다면, 초과하지 않을 때까지 요소 제거를 진행합니다.

**Detailed explanation**

To summarize what each variable does in the code:

- `left`: the leftmost index of our current window
- `right`: the rightmost index of our current window
- `curr`: the sum of our current window
- `ans`: the length of the longest valid window we have seen so far

코드에서 각 변수가 하는 역할은 다음과 같습니다:

- `left`: 현재 윈도우의 가장 왼쪽 인덱스

- `right`: 현재 윈도우의 가장 오른쪽 인덱스
- `curr`: 현재 윈도우의 합
- `ans`: 지금까지 찾은 가장 긴 유효 윈도우의 길이

Iterate `right` over the input to add elements to the window. Update `curr` by adding `nums[right]` to it. When the window becomes invalid `(curr > k)`, remove elements from the window by subtracting `nums[left]` from `curr`. Then increment `left`. We need to do this until the window becomes valid again, so we use a while loop.
`right`를 0부터 입력 배열의 끝까지 순회하면서, `nums[right]`를 `curr`에 더해 윈도우에 추가합니다. 만약 `(curr > k)`가 되어 윈도우가 유효하지 않아지면, `curr`에서 `nums[left]`를 빼는 식으로 왼쪽 요소를 제거하고 `left`를 증가시킵니다. 유효 상태로 돌아올 때까지 반복해야 하므로, `while` 루프를 사용합니다.

The size of a window is `right - left + 1`. Update our answer only when the window becomes valid.
윈도우의 크기는 `right - left + 1`로 계산하며, 윈도우가 유효해진 시점에 `ans`를 갱신합니다.

If you're still having trouble understanding the algorithm, review the sections of this article before the example along with the video walkthrough.
알고리즘 이해가 어려우시다면, 예제 이전의 설명과 동영상 가이드를 다시 살펴보시기 바랍니다.

**Video**

```
public int findLength(int[] nums, int k) {
    int left = 0;
    int curr = 0; // curr is the current sum of the window
    int ans = 0;

    for (int right = 0; right < nums.length; right++) {
        curr += nums[right];
        while (curr > k) {
            curr -= nums[left];
            left++;
        }

        ans = Math.max(ans, right - left + 1);
    }

    return ans;
```

```
}
```

Given a subarray starting at `left` and ending at `right`, the length is `right - left + 1`. As mentioned before, this algorithm has a time complexity of (O(n)) since all work done inside the for loop is **amortized** (O(1)), where (n) is the length of `nums`. The space complexity is constant because we are only using 3 integer variables.
이처럼 `left`부터 `right`까지의 서브어레이 길이는 `right - left + 1`입니다. 이미 설명했듯이, 이 알고리즘은 `for` 루프 안에서의 모든 작업이 **평균적으로** (O(1))에 수행되므로 전체 시간 복잡도는 (O(n))입니다(`n`은 `nums`의 길이). 또한 정수 변수 3개만을 사용하므로, 공간 복잡도는 상수 시간에 머무릅니다.

---

> **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 `<u>11111</u>00111`.
> **예제 2:** 이진 문자열 `s`가 주어집니다(문자열은 `"0"`과 `"1"`만 포함합니다). 여기서 `"0"`을 최대 한 번 `"1"`로 뒤집을 수 있습니다. 그 결과, 오직 `"1"`만으로 이루어진 가장 긴 부분 문자열(substring)의 길이는 얼마일까요? 예를 들어, `s = "1101100111"`이라고 할 때, 정답은 `5`입니다. 인덱스 `2`에서 `"0"`을 `"1"`로 뒤집으면, 문자열은 `<u>11111</u>00111`이 됩니다.

Because the string can only contain `"1"` and `"0"`, 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.
이 문자열은 `"1"`과 `"0"`만 포함하므로, 문제를 "최대 한 개의 `"0"`을 포함할 수 있는 가장 긴 부분 문자열"로 해석할 수 있습니다. 즉, `window.count("0") <= 1`이라는 조건을 사용하는 슬라이딩 윈도우로 쉽게 해결이 가능합니다. 현재 윈도우에 포함된 `"0"`의 개수를 추적하는 `curr` 변수를 두면 됩니다.

**Detailed explanation**

The input can only contain `"1"` or `"0"`. We want to find the max consecutive `"1"`. Because any element that isn't a `"1"` is a `"0"`, this problem is equivalent to "what is the longest substring with **at most one** `"0"`".
이 문자열은 `"1"` 또는 `"0"`만 포함합니다. 우리는 연속된 `"1"`의 최댓값을 구하고 싶고, `"1"`이 아닌 문자는 `"0"`뿐

이므로 "**최대 한 개**의 `"0"`을 포함하는 가장 긴 부분 문자열" 문제와 동일합니다.

Notice that the problem is asking for the length of a substring, and also has defined what makes a substring valid. The constraint metric is "how many 0s are in the substring". The numeric restriction is `<= 1`. Therefore, if we use an integer `curr` to track the constraint metric, the condition to determine if a window is valid is `curr <= 1`.
문제에서 요구하는 것은 부분 문자열의 길이이며, 동시에 무엇이 유효한 부분 문자열인지 정의하고 있습니다. 제약 지표는 부분 문자열 내 `"0"`의 개수이며, 수치적 제한은 `<= 1`입니다. 따라서 `curr` 변수를 이용해 `"0"`의 개수를 추적하면, 윈도우가 유효한지 여부는 `curr <= 1`로 판단할 수 있습니다.

We can use the exact same process as in the previous example now. We iterate over the elements with a pointer `right`. At each element, if `s[right]` is equal to `"1"`, we don't need to do anything. If it's equal to `"0"`, we increment `curr`.
이제 이전 예제에서와 완전히 같은 프로세스를 적용할 수 있습니다. `right` 포인터로 문자열을 순회하면서, `s[right]`가 `"1"`이면 아무 작업도 하지 않고, `"0"`이면 `curr`를 증가시킵니다.

Whenever the window becomes invalid (`curr > 1`), we remove elements from the left. If `s[left] == "0"`, then we can decrement `curr`. We increment `left` to remove elements.
윈도우가 유효 범위를 벗어나면(`curr > 1`), 왼쪽에서 요소를 제거하여 `curr`를 감소시키면 됩니다. 이 때 `s[left] == "0"`이면 `curr`를 1 감소시키고, `left`를 증가시켜 윈도우에서 제외합니다.

Again, the size of a window is `right - left + 1`. We update our answer with this value after the while loop because the window is guaranteed to be valid.
여기서도 윈도우의 크기는 `right - left + 1`입니다. `while` 루프 이후, 윈도우는 유효 상태이므로 이 값을 이용해 답을 갱신하면 됩니다.

### Video

```java
public int findLength(String s) {
    // curr is the current number of zeros in the window
    int left = 0;
    int curr = 0;
    int ans = 0;

    for (int right = 0; right < s.length(); right++) {
        if (s.charAt(right)== '0') {
            curr++;
```

```
        }

        while (curr > 1) {
            if (s.charAt(left) == '0') {
                curr--;
            }

            left++;
        }

        ans = Math.max(ans, right - left + 1);
    }

    return ans;
}
```

Like the previous example, this problem runs in (O(n)) time, where (n) is the length of `s`, as the work done in each loop iteration is amortized constant. Only a few integer variables are used as well, which means this algorithm uses (O(1)) space.
이전 예제와 마찬가지로, 이 문제도 각 반복에서의 작업이 평균적으로 상수 시간이 걸리므로 전체 시간 복잡도는 (O(n))입니다(`n`은 문자열 `s`의 길이). 또한 몇 개의 정수 변수만 사용하므로, 공간 복잡도 역시 (O(1))입니다.

---

## Number of subarrays

If a problem asks for **the number of subarrays** that fit some constraint, we can still use sliding window, but we need to use a neat math trick to calculate the number of subarrays.
만약 어떤 제약을 만족하는 **서브어레이의 개수**를 구하라는 문제가 있다면, 여전히 슬라이딩 윈도우를 적용할 수 있지만, 서브어레이의 개수를 계산하기 위해 간단한 수학적 트릭을 사용해야 합니다.

Let's say that we are using the sliding window algorithm we have learned and currently have a window `(left, right)`. How many valid windows **end** at index `right`?
현재 `(left, right)`라는 윈도우가 있다고 합시다. 그렇다면 인덱스 `right`에서 **끝나는** 유효한 윈도우는 총 몇 개일까요?

There's the current window `(left, right)`, then `(left + 1, right)`, `(left + 2, right)`, and so on until `(right, right)` (only the element at `right`).

`(left, right)`라는 현재 윈도우가 있고, 그 다음은 `(left + 1, right)`, `(left + 2, right)`, ..., 그리고 `(right, right)`(단일 요소)까지 생각해볼 수 있습니다.

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 equal to the size of the window, which we know is `right - left + 1`.
오른쪽 경계를 `right`로 고정했을 때, 왼쪽 경계는 `left`부터 `right`까지 어떤 인덱스를 골라도 됩니다. 따라서 `right`에서 끝나는 유효한 윈도우의 개수는 윈도우의 크기( `right - left + 1` )와 동일해집니다.

> **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]`
> **예제 3:** 713. Subarray Product Less Than K
> 양의 정수 배열 `nums`와 정수 `k`가 주어졌을 때, 서브어레이의 모든 요소 곱이 `k` 미만인 서브어레이의 개수를 구하시오.
> 예를 들어, `nums = [10, 5, 2, 6], k = 100`이라는 입력이 주어지면, 정답은 `8`입니다. 곱이 `k` 미만인 서브어레이는 다음과 같습니다.
> `[10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [5, 2, 6]`

To demonstrate the property we have just learned, let's look at the example in the description. When we reach index `2`, the product becomes too large, so we need to remove the leftmost element `10`. Now, the window is valid, and it has a length of `2`. That means that there are `2` valid subarrays that end here (`[2]` and `[5, 2]`).
앞서 배운 내용을 예시에 적용해 봅시다. 문제 설명의 예시에서 인덱스 `2`에 도달했을 때, 곱이 너무 커져서 가장 왼쪽 요소 `10`을 제거해야 했습니다. 이제 윈도우가 유효해졌고, 길이가 `2`가 되었습니다. 이는 인덱스 `2`에서 끝나는 유효한 서브어레이가 `2`개라는 뜻입니다. 즉 `[2]`, `[5, 2]`입니다.

Recall that in the previous examples, we updated the answer (longest length) after the while loop, when the window must be valid. Here, we can add the current size of the window to our answer instead. The constraint that determines if a window is valid is that the product is less than `k`.
이전 예제들에서는, while 루프가 끝나 윈도우가 유효해진 시점에 가장 긴 길이를 갱신했습니다. 여기서는 그 대신, 유효한 윈도우의 크기를 바로 정답에 더해주면 됩니다. 윈도우가 유효한지 여부는 곱이 `k` 미만인가로 판단합니다.

Additionally, note that if `k <= 1` we can never have any valid windows, so we can just return `0`

immediately.
추가로, `k <= 1`인 경우에는 어떤 서브어레이도 곱이 `k` 미만이 될 수 없으므로, 바로 `0`을 반환하면 됩니다.

**Detailed explanation**

The constraint metric is: product of the window. The numeric restriction is `< k`. If we use an integer `curr` to represent the current product of the window, the condition that makes a window invalid is `curr >= k`.
여기서 제약 지표는 윈도우의 곱이고, 수치적 제한은 `< k`입니다. `curr` 변수를 이용해 현재 윈도우의 곱을 나타낸다고 할 때, 윈도우가 유효하지 않게 되는 조건은 `curr >= k`입니다.

Add elements to the window with `curr *= nums[right]`. Remove them with `curr /= nums[left]`.
오른쪽 요소를 추가할 때는 `curr *= nums[right]`, 왼쪽 요소를 제거할 때는 `curr /= nums[left]`를 수행합니다.

After the while loop, we know the window is valid. Add the window size `right - left + 1` to our answer.
`while` 루프가 종료되면 윈도우가 유효해졌으므로, `right - left + 1`을 정답에 추가하면 됩니다.

**Video**

```
class Solution {
    public int numSubarrayProductLessThanK(int[] nums, int k) {
        if (k <= 1) {
            return 0;
        }

        int ans = 0;
        int left = 0;
        int curr = 1;

        for (int right = 0; right < nums.length; right++) {
            curr *= nums[right];
            while (curr >= k) {
                curr /= nums[left];
                left++;
            }
```

```
            ans += right - left + 1;
        }

        return ans;
    }
}
```

Again, the work done in each loop iteration is amortized constant, so this algorithm has a runtime of (O(n)), where (n) is the length of `nums`, and (O(1)) space.
여기서도 각 반복에서의 작업이 평균적으로 상수 시간에 이뤄지므로 전체 시간 복잡도는 (O(n))이며, `nums`의 길이를 `n`이라고 했을 때, 공간 복잡도도 (O(1))입니다.

---

## Fixed window size

In the examples we looked at above, our window size was dynamic. We tried to expand it to the right as much as we could while keeping the window within some constraint and removed elements from the left when the constraint was violated. Sometimes, a problem will specify a **fixed** length `k`.
지금까지 살펴본 예제에서는 윈도우의 크기가 동적으로 변했습니다. 조건을 만족하는 한 최대한 오른쪽으로 확장하고, 조건이 깨지면 왼쪽에서 요소를 제거하는 식이었죠. 하지만 어떤 문제들은 윈도우 크기가 **고정**된 `k`로 주어질 때가 있습니다.

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).
이 경우는 비교적 간단한데, 인접한 두 윈도우 간 차이가 단지 2개의 요소뿐이기 때문입니다(오른쪽에서 한 요소를 추가하고, 왼쪽에서 한 요소를 제거해 고정 길이를 유지).

Start by building the first window (from index `0` to `k - 1`). Once we have a window of size `k`, if we add an element at index `i`, we need to remove the element at index `i - k`. For example, `k = 2` and you currently have elements at indices `[0, 1]`. Now, we add `2`: `[0, 1, 2]`. To keep the window size at `k = 2`, we need to remove `2 - k = 0`: `[1, 2]`.
먼저 인덱스 `0`부터 `k - 1`까지를 포함하는 윈도우를 만듭니다. 이렇게 크기가 `k`인 윈도우가 준비된 상태에서, 인덱스 `i`에 있는 요소를 새로 추가하려면 인덱스 `i - k`에 있던 요소를 제거해야 합니다. 예를 들어 `k = 2`라면, 현재 윈도우가 `[0, 1]`이라 가정해 봅시다. 여기서 인덱스 `2`를 추가하면 `[0, 1, 2]`가 되지만, 윈도우 크기를 `k = 2`로

유지하려면 `i - k = 0`에 있는 요소를 제거해 `[1, 2]`로 만들어야 합니다.

Here's some pseudocode:
여기서의 의사 코드는 다음과 같습니다:

```
function fn(arr, k):
    curr = some data to track the window

    // build the first window
    for (int i = 0; i < k; i++)
        Do something with curr or other variables to build first window

    ans = answer variable, probably equal to curr here depending on the
problem
    for (int i = k; i < arr.length; i++)
        Add arr[i] to window
        Remove arr[i - k] from window
        Update ans

    return ans
```

---

> **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`.
> **예제 4:** 정수 배열 `nums`와 정수 `k`가 주어졌을 때, 길이가 `k`인 서브어레이 중 합이 가장 큰 것의 합을 구하시오.

As we mentioned before, 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 sure the window stays size `k`. If we are adding the value at `i`, then we need to remove the value at `i - k`.
이미 언급했듯이, 우리는 먼저 길이가 `k`인 윈도우를 구성한 뒤, 이것을 배열 전체에 걸쳐 슬라이딩하면 됩니다. 한 번에 하나의 요소를 추가하고 하나의 요소를 제거하여 윈도우 크기를 `k`로 유지합니다. 인덱스 `i`에 있는 값을 새로 추가했다면, 인덱스 `i - k`에 있는 값을 제거해야 합니다.

After we build the first window we initialize our answer to `curr` to consider the first window's sum.
첫 번째 윈도우를 만든 뒤, 그 합을 `curr`에 저장하고 이것으로 초기 답변을 설정합니다.

**Video**

```java
public int findBestSubarray(int[] nums, int k) {
    int curr = 0;
    for (int i = 0; i < k; i++) {
        curr += nums[i];
    }

    int ans = curr;
    for (int i = k; i < nums.length; i++) {
        curr += nums[i] - nums[i - k];
        ans = Math.max(ans, curr);
    }

    return ans;
}
```

The total for loop iterations is equal to `n`, where `n` is the length of `nums`, and the work done in each iteration is constant, giving this algorithm a time complexity of (O(n)), using (O(1)) space.
이 알고리즘에서 전체 `for` 루프 반복 횟수는 `n`(`nums`의 길이)이고, 각 반복마다 상수 시간 작업만 수행하므로 전체 시간 복잡도는 (O(n))이며, 공간 복잡도 또한 (O(1))입니다.

---

## Closing notes

Sliding window is extremely common and versatile as a pattern. We only scratched the surface here because many sliding window problems will also need to use a hashmap, which we will talk about in the hashing chapter. After learning about hashmaps, we'll look at some more sliding window problems. In the meantime, test your knowledge by solving the upcoming practice problems.
슬라이딩 윈도우는 매우 흔하고도 다양한 방식으로 활용되는 패턴입니다. 사실 여기서 다룬 내용은 극히 일부에 불과합니다. 왜냐하면 많은 슬라이딩 윈도우 문제들이 해시맵(HashMap)을 함께 사용하는 경우가 많기 때문입니다. 해시맵에 대해 배운 뒤, 더 심화된 슬라이딩 윈도우 문제들을 살펴볼 예정이니, 그 전까지는 이번 챕터에서 배운 내용으로 연습 문제를 풀어보며 이해를 다져 보세요.

---

## 연습 문제

- 643. Maximum Average Subarray I
- 1004. Max Consecutive Ones III
- 2090. K Radius Subarray Averages