10 Shortest pair of subarrays with target sum

Given an array of positive integers and a target sum, find two non-overlapping subarrays such that the sum of the elements of each array is equal to the given target sum, and their total length is minimal. Return their total length.

Example: For `[1, 2, 1, 1, 1]` and target sum 3, the two subarrays are `[1, 2]` and `[1, 1, 1]`, with total length 5. There are also two shorter subarrays `[1, 2]` and `[2, 1]` having the target sum 3, but they cannot be considered since they overlap.

Clarification questions

Q: What result should be returned for an empty array, or if there is no solution?
A: The result should be 0.

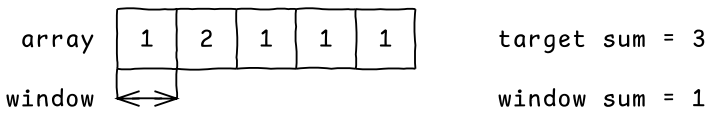
Q: What is the maximum length of the input?
A: 1,000,000 elements.

Solution 1: dynamic programming + sliding window, $O(n)$ time, $O(n)$ space

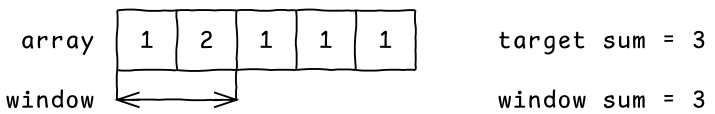
Let's try to solve an easier problem first: enumerate efficiently all the subarrays having the target sum, regardless of whether they overlap or not.

For arrays of positive numbers, there is an easy solution that involves advancing a sliding window:

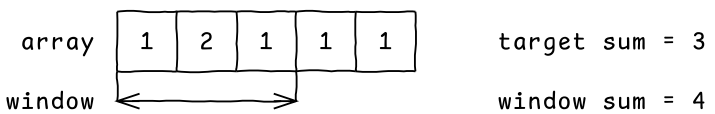
- We start with a window containing just the first element of the array:



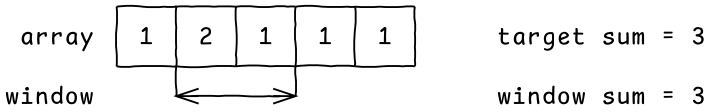
- The window sum is smaller than the target sum, so we expand the window to the right:



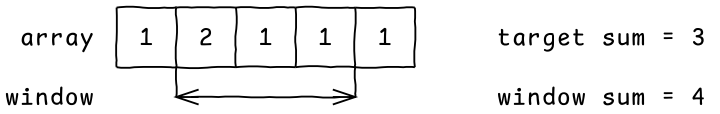
- We found the first subarray. We expand the window to the right again:



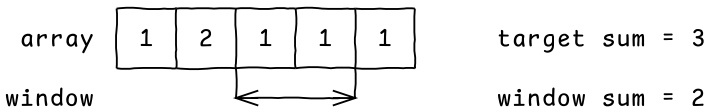
- The window sum is larger than the target sum. We need to shrink the window to reduce its sum. There is no point in shrinking from the right, since we would obtain the window from the previous step. So we shrink from the left, removing the leftmost element:



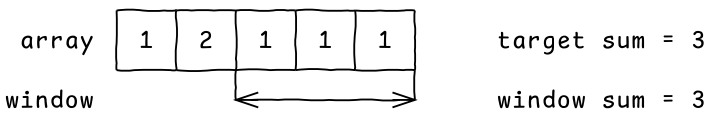
- We found the second subarray. We expand the window to the right again:



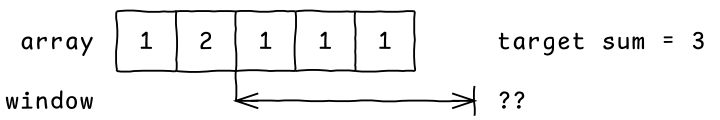
- The window sum is larger than the target sum, so we shrink from the left, removing the leftmost element:



- The window sum is smaller than the target sum, so we expand the window to the right:



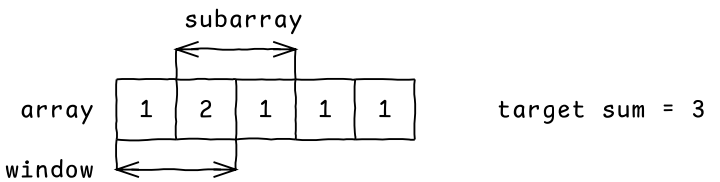
- We found the third subarray. We expand the window to the right again:



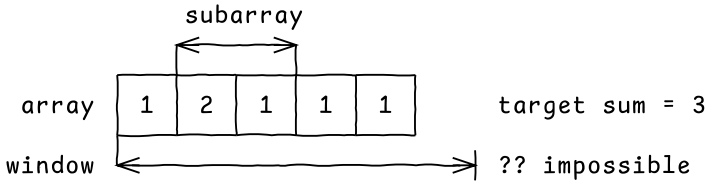
- We moved past the edge of the array, so we stop the search.

This method enumerates all possible windows that match the target sum. This can be proven in the following way:

- The search ends when the sliding window reaches the end of the array. This means that every element of the array is considered at least once as part of some sliding window.
- Consider a subarray matching the target sum. We want to show that our sliding window algorithm will consider it. From the previous point, it follows that the leftmost element of the subarray is considered at least once as part of some sliding window:

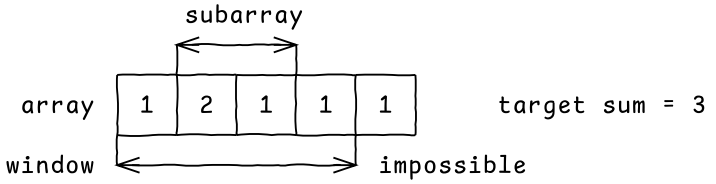


- There is at least one sliding window that starts with the leftmost element. This can be shown by contradiction: If there is no sliding window that starts with the leftmost element, it means that the last sliding window is starting before the leftmost element, and ends at the end of the array, then the search ends:

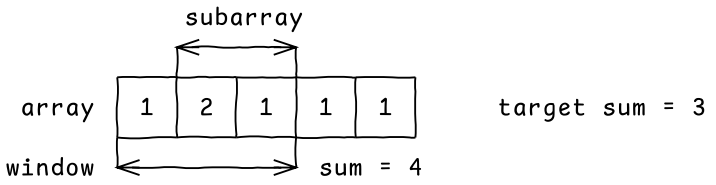


But this is impossible, since the sum of this window exceeds the target sum, so we can apply the shrinking rule to advance it. But then it cannot be the last window in the search, so we have a contradiction.

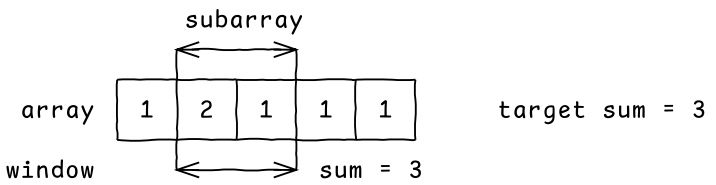
- The right edge of the sliding window will not advance past the rightmost element in the subarray before its left edge reaches the leftmost element. For example, the following is impossible:



This is because the sum of the window exceeds the target sum when its right edge reaches the rightmost element of the subarray:



Thus the next step must be a shrinking, not an expansion. The window will be shrunk until its left edge reaches the leftmost element of the subarray:



This proves that our solution will not miss any subarrays having the target sum, so it is correct. Note that the method only works if array elements are positive numbers.

> **Sliding window**
>
> There are many problems that can be solved using sliding windows. Not just algorithmic puzzles, but also fundamental problems in computer science, such as reliable data transmission. For example, TCP uses a sliding window to keep track of data that has been sent but not yet acknowledged by the receiver, thus may require retransmission. Understand the concept well: there are many problems for which it can be used to obtain a natural and intuitive solution.

This method can be implemented as:

```
def enumerate_subarrays_with_sum(array, target_sum):
    window_sum = 0
    left = 0
    for right, value in enumerate(array):
        # Expand to the right.
        window_sum += array[right]
        # Contract from the left.
        while window_sum > target_sum:
            window_sum -= array[left]
            left += 1
        if window_sum == target_sum:
            # Handle subarray.
            print(left, right)
```

Notice how we compute the window sum incrementally in $O(1)$ time at each step, instead of using `sum(array[left:right+1])`, which takes $O(n)$ time. This allows us to enumerate all windows in linear time.

The overall time complexity is $O(n)$, with `n = len(array)`, since the window expands up to n times and contracts up to n times.

Still, this does not solve our problem, just a part of it. We need to add a few more pieces: applying this to a pair of arrays; ensuring there are no overlaps; and minimizing the total length.

Let's start from minimizing the total length, since it is the easiest next step. We transform the code to solve the following problem: finding the *minimum length* subarray having a target sum.

To do this, we must keep track of the shortest window matching the target sum, when enumerating all possible windows:

```
def find_shortest_subarray_with_sum(array, target_sum):
    window_sum = 0
    left = 0
    min_length = float('inf')
    for right, value in enumerate(array):
```

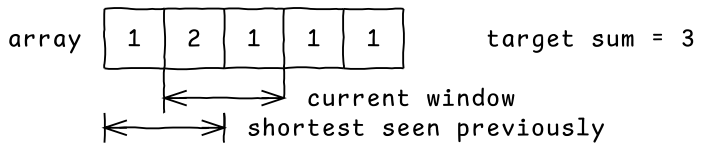
```
        # Expand to the right.
        window_sum += array[right]
        # Contract from the left.
        while window_sum > target_sum:
            window_sum -= array[left]
            left += 1
        if window_sum == target_sum:
            # Handle subarray.
            length = right - left + 1
            # Update min_length.
            min_length = min(min_length, length)
    if min_length == float('inf'):
        return 0
    return min_length
```

Let's now extend this solution to find pairs of subarrays having the smallest total length.

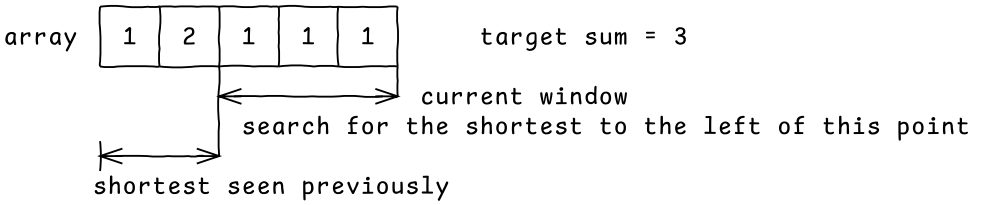
We can modify the search such that each time we find a matching window, we sum up its length with the minimum length among matching windows seen previously, and take the minimum overall:

```
    ...
    min_length = float('inf')
    result = float('inf')
    ...
        if window_sum == target_sum:
            # Handle subarray.
            length = right - left + 1
            # Update result.
            result = min(result, length + min_length)
            # Update min_length.
            min_length = min(min_length, length)
    if result == float('inf'):
        return 0
    return result
```

The last point to address is handling subarrays that overlap. The current solution may return overlapping pairs:



To find the pair of *non-overlapping* subarrays having the smallest total length, we have to sum up the current length with the minimum length seen so far *on the left of the current window*, and take the minimum overall:



To do this, for each possible left point 0, 1, 2, …, we have to store the length of the shortest array seen with the right edge lower or equal than that left point. We can keep track of it in an array.

This algorithm can be implemented as:

```
def find_shortest_subarray_with_sum(array, target_sum):
    window_sum = 0
    left = 0
    result = float('inf')
    # min_length_up_to[i] = min. length of subarrays with target sum
    #                       having right index <= i.
    min_length_up_to = [float('inf')] * len(arr)
    for right, value in enumerate(array):
        # Propagate min_len_up_to. We may update it later.
        min_length_up_to[right] = min_length_up_to[right - 1]
        # Expand to the right.
        window_sum += array[right]
        # Contract from the left.
        while window_sum > target_sum:
            window_sum -= array[left]
            left += 1
        if window_sum == target_sum:
            # Handle subarray.
            length = right - left + 1
            # Update result.
            result = min(result,
                         min_length_up_to[left - 1] + length)
            # Update min_length_up_to[right].
            min_length_up_to[right] = min(min_length_up_to[right],
                                          length)
    if result == float('inf'):
        return 0
    return result
```

The time complexity is $O(n)$ and the space complexity is $O(n)$ as well.