Sliding Window Pattern

Sliding Window is an algorithmic technique to process a subset (window) of data in a list (array/string) that slides from start to end. Instead of checking every possible subset (which is slow), we efficiently move a “window” forward to cover all possible cases in linear time.

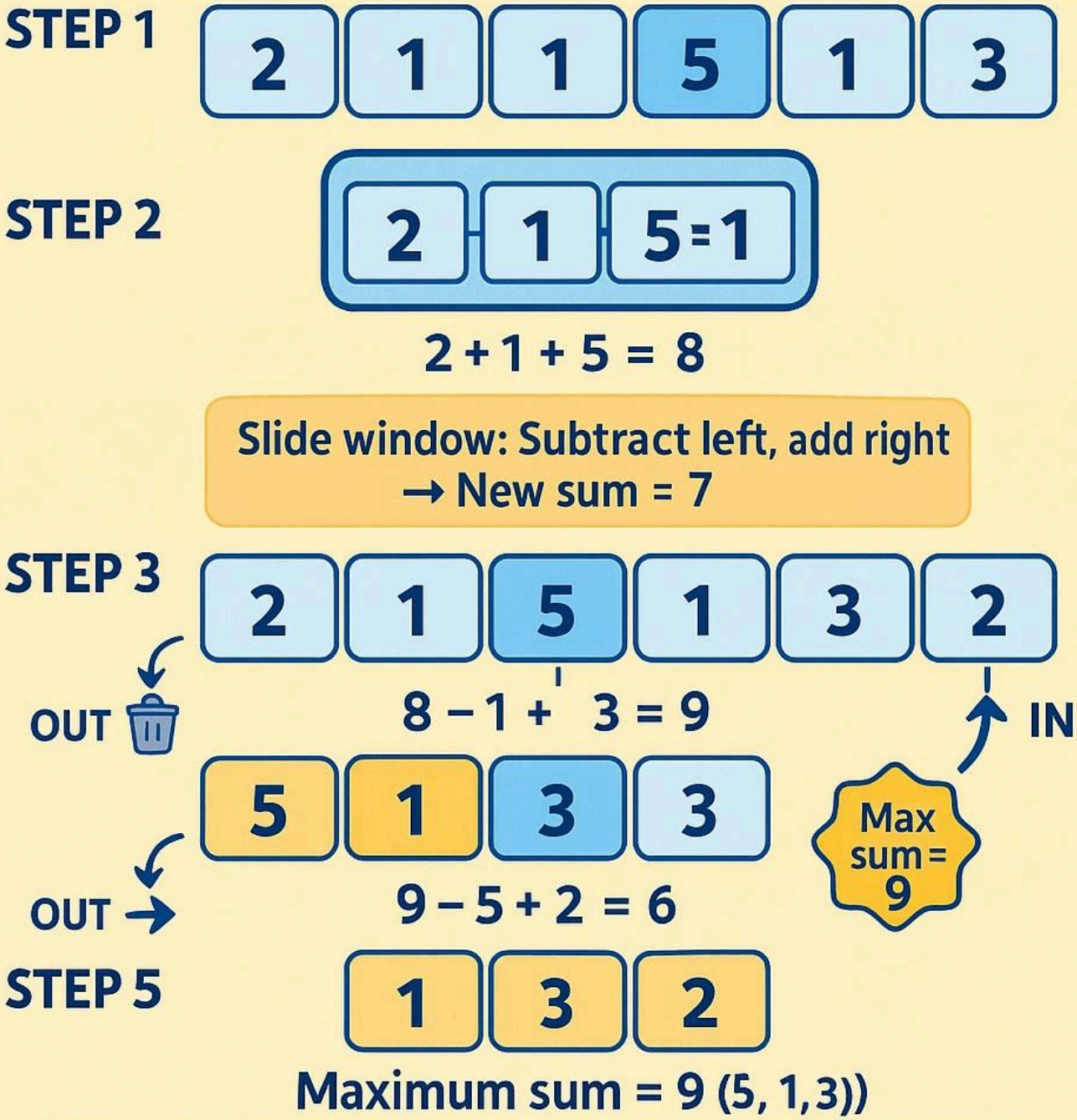
- **Use Cases:** When you need to calculate something (sum, max, count, etc.) over all continuous subarrays/substrings of a fixed size.

SLIDING WINDOW PATTERN:
MAXIMUM SUM SUBARRARY OF SIZE *K*

Array: [2, 1, 5, 1, 3 2]

Window size (*K*): 3

Task: Find the maximum sum of any subarray of size 3.



Sliding Window lets you process fixed-size groups in

arrays in O(N) time—efficient and easy!

Analogy

Imagine a window of fixed size on a train. As the train moves, the scenery changes within the window. Similarly, in an array, the window covers a few elements, and as we “slide” the window, we move to the next group of elements.

Common Problems

- Maximum/Minimum sum of a subarray of size K
- Longest substring with no more than K distinct characters
- Smallest subarray with sum ≥ S

Let’s do the **most basic one**:

Example: Maximum Sum Subarray of Size K

Problem:
Given an array of integers and a number K, find the maximum sum of any contiguous subarray of size K.

Step-by-step Solution:

Naive Way:
Loop through every possible subarray of size K, calculate the sum. This is O(N*K).

Sliding Window Way:

- Calculate the sum of the first window (first K elements).
- For every next window, subtract the element going out of the window (on the left), and add the new element coming in (on the right).
- Update the maximum sum found.

C Code Example

```
#include <stdio.h>

int maxSumSubarray(int arr[], int n, int k) {
    if (n < k) {
        printf("Window size is larger than array.\n");
        return -1;
    }

    int max_sum = 0;
    // 1. Calculate sum of first window
    for (int i = 0; i < k; i++) {
        max_sum += arr[i];
    }

    int window_sum = max_sum;

    // 2. Slide the window from k to n-1
    for (int i = k; i < n; i++) {
        // Remove the leftmost element, add the new rightmost element
```

```
        window_sum += arr[i] - arr[i - k];
        if (window_sum > max_sum)
            max_sum = window_sum;
    }

    return max_sum;
}

int main() {
    int arr[] = {2, 1, 5, 1, 3, 2};
    int k = 3;
    int n = sizeof(arr) / sizeof(arr[0]);

    int result = maxSumSubarray(arr, n, k);
    printf("Maximum sum of a subarray of size %d = %d\n", k, result);

    return 0;
}
```

How This Works:

1. **Initial Window:**
   First, we sum the first 'k' elements (2 + 1 + 5 = 8).
   This is our starting window.
2. **Sliding the Window:**
   Next, instead of recalculating sum for [1,5,1] and [5,1,3]... we do this:
   - For window [1,5,1], remove 2 (leftmost), add 1 (rightmost):
     8 - 2 + 1 = 7
   - For [5,1,3], remove 1 (now at the left), add 3:
     7 - 1 + 3 = 9
   - For [1,3,2], remove 5, add 2:
     9 - 5 + 2 = 6
3. **Keep Track of Maximum:**
   Every time we get a new window sum, we check if it's the largest so far.
4. **Output:**
   In this example, the answer is **9** (from the window [5,1,3]).

Key Points

- **Why is this better?**
  Instead of recalculating the sum each time (slow), we do just **two operations** for each slide (subtract one, add one), so it's very fast—O(N) time.
- **Where is this useful?**
  Any time you need to look at every subarray or substring of a fixed size and do some operation (sum, max, min, etc.).

Visual Representation

For K=3 and arr = [2, 1, 5, 1, 3, 2]:

- Window 1: [2, 1, 5] → Sum = 8
- Window 2: [1, 5, 1] → Sum = 7
- Window 3: [5, 1, 3] → Sum = 9 (max)
- Window 4: [1, 3, 2] → Sum = 6

Max = 9

Practice Problem for You

Try to change the code to **find the minimum sum subarray of size K**.
Tip: Just initialize `min_sum` to a large value, and update it in the loop.