AXSOS ACADEMY

Problem-Solving Patterns
Sliding Window

Outline

- Introduce the sliding window technique including Idea, Problem statement, and solution.
- Practice a challenge and solve a LeetCode problem
- Review the solution and discuss it.

Introduction

What is a coding pattern?

a code blueprint that can be customized to optimally solve many related problems.

>No need to memorize the solution.

To master these patterns:
- What pattern
- How it works

Why patterns?

- Your **problem-solving skills** will be taken to the next level
- This will ensure that you reach the **optimal solution** by developing a clear understanding of the problem, as well as an understanding of which pattern suits the problem best.
- to **ace the interview** for your dream job

16 Pattern

- Sliding Window
- Two Pointers
- Fast Slow Pointers
- Merge Intervals
- Cycle Sort
- In-place reversal of a LinkedList
- Tree Breadth-First Search
- Tree Depth First Search
- Two Heaps
- Subsets
- Modified Binary Search
- Bitwise XOR
- Top K elements
- K-way merge
- Knapsack
- Topological Sort

Introduction

What is Sliding window pattern?

It is a technique used to solve problems that involve arrays, strings, or other sequential data structures. By breaking down a larger problem into smaller subproblems, it can be a more efficient and effective approach than brute-force methods.

Why?
To reduce the use of nested loops

Introduction

How it works?

- It involves creating a "window" of fixed size or dynamic size and sliding it over the data structure, performing some operation on the data within the window, and then moving the window to the next position.

Problem Statement

Given an array of integers *nums* and a number *k*, find the maximum sum of a contiguous **subarray of size k.**

nums

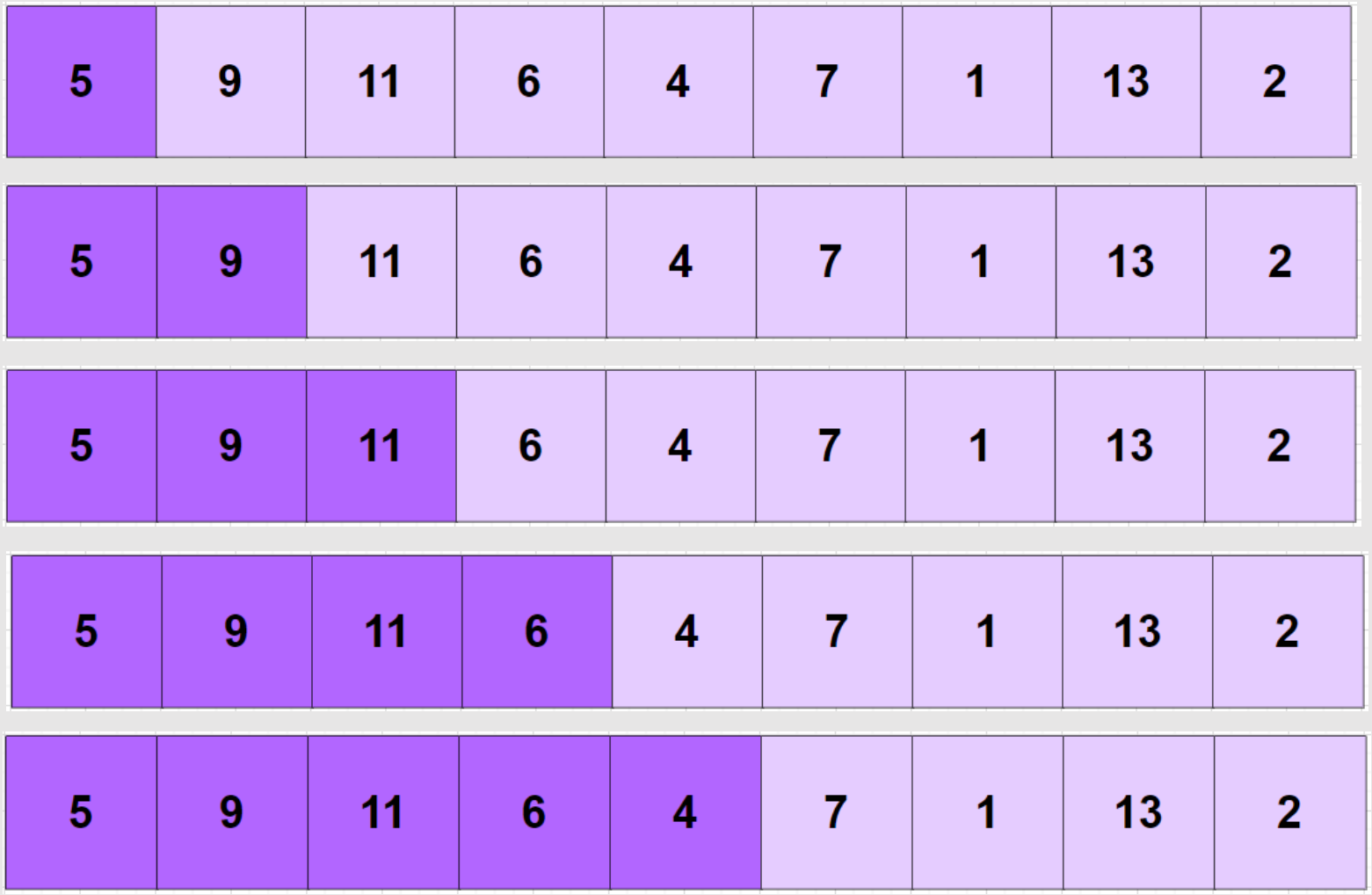
K = 5
maxSum = ?

Brute Force Approach

```
1   function maxSumSubarr(nums, k){
2   if (nums.length < k) { return "invalid" }
3   let newArr = [];
4   for (let i = 0; i<= nums.length-k; i++) {
5       let sum = 0;
6       for (let j = i; j < k+i; j++) {
7           sum += nums[j];
8       }
9       newArr.push(sum)
10  }
11  let max = newArr[0];
12  for (let index = 1; index < newArr.length; index++) {
13      if(newArr[index] > max){
14      max = newArr[index];
15      }
16  }
17  return max;
18  }
```

Brute Force Approach

i = 0



Brute Force Approach

i = 1

j = 0

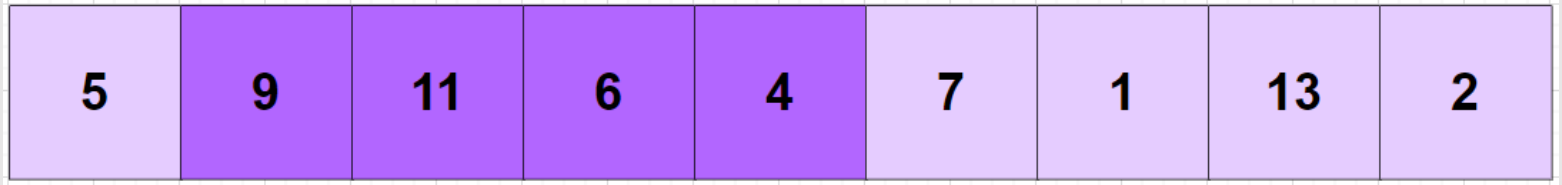
sum = 0+9 =9

j = 1

sum = 9 + 11= 20

j = 2

sum = 20 + 6 = 26



j = 3

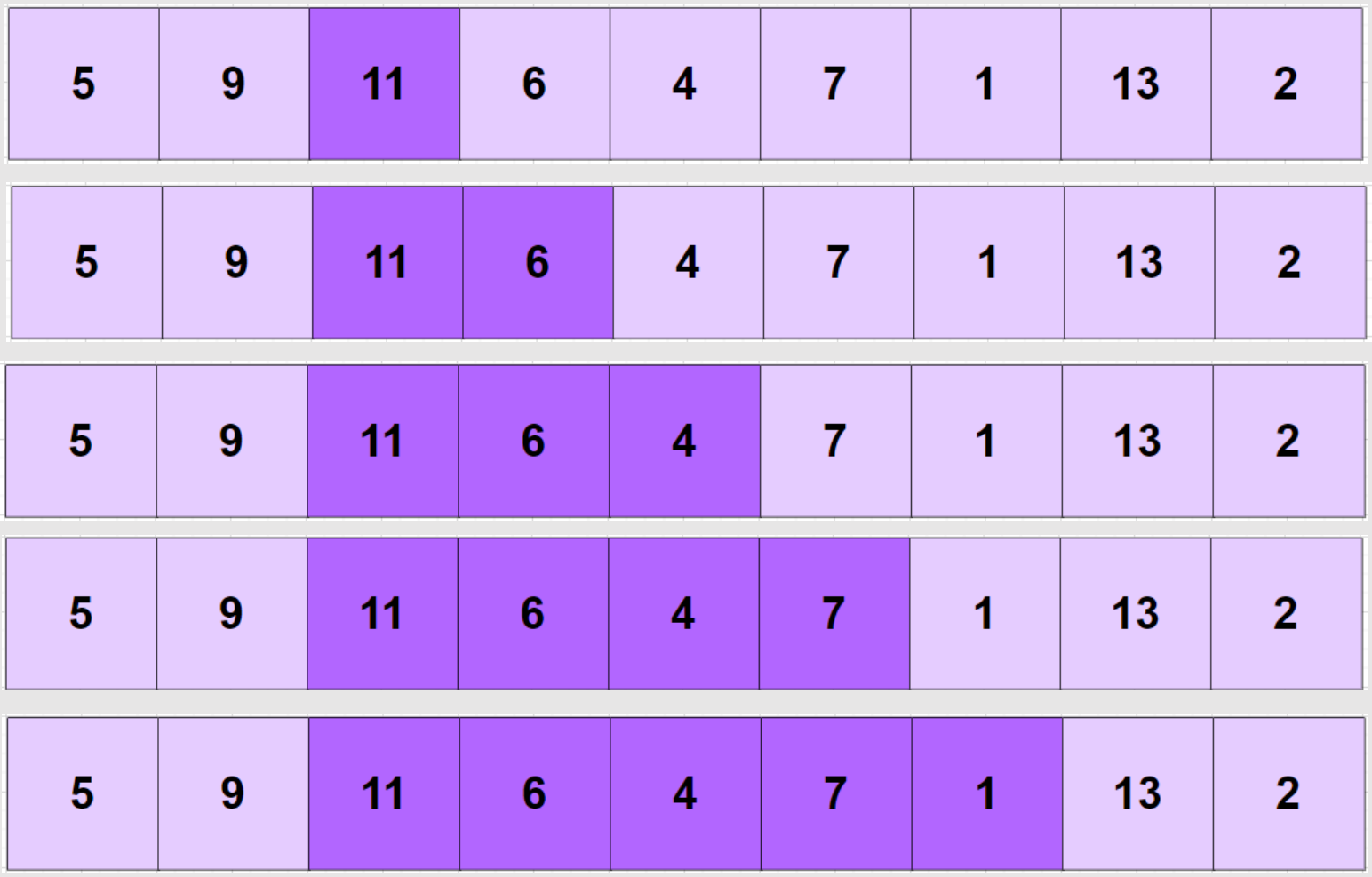
sum = 26 + 4 = 30

j = 4

sum = 30 + 7 = 37

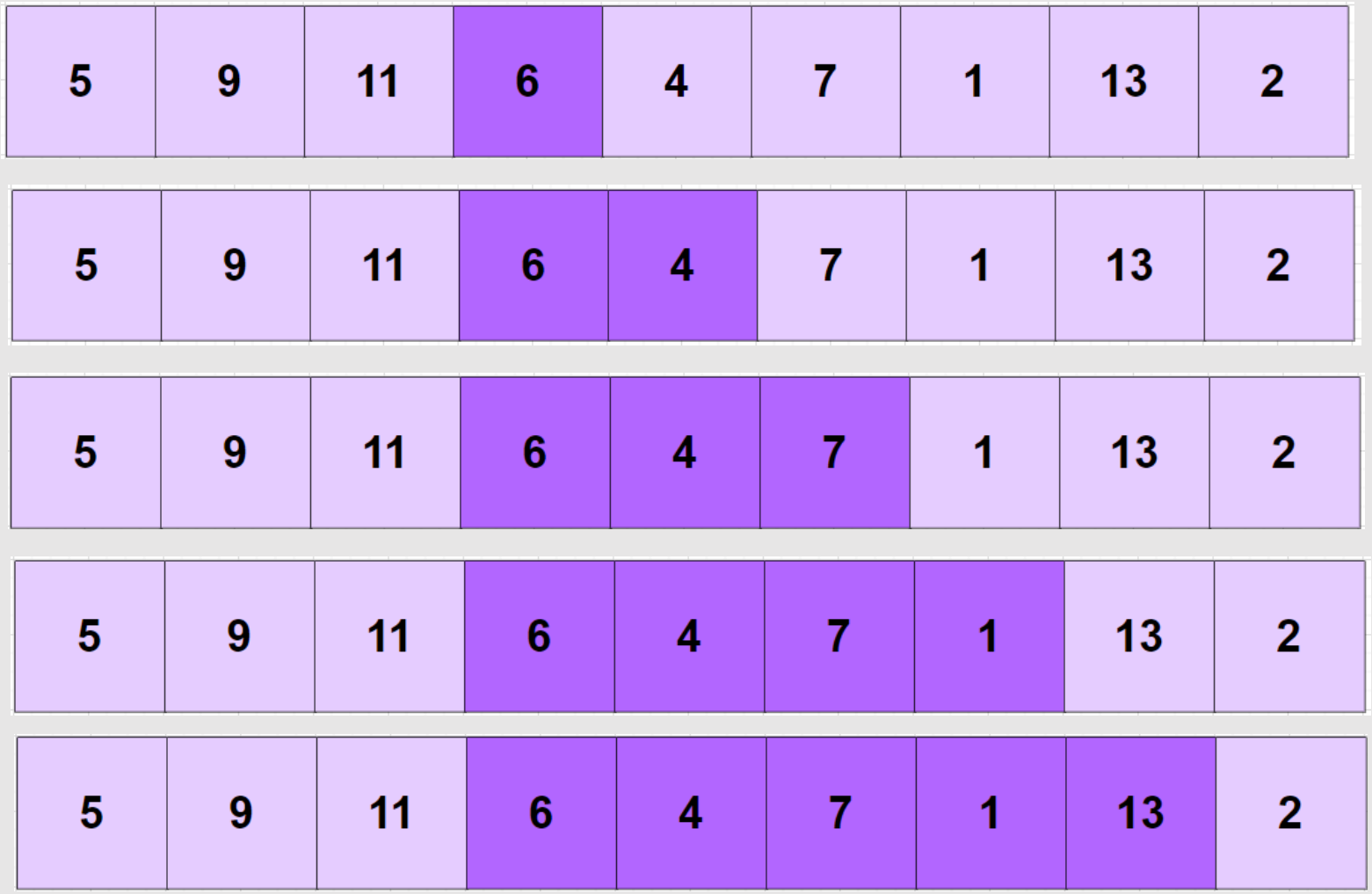
Brute Force Approach

i = 2



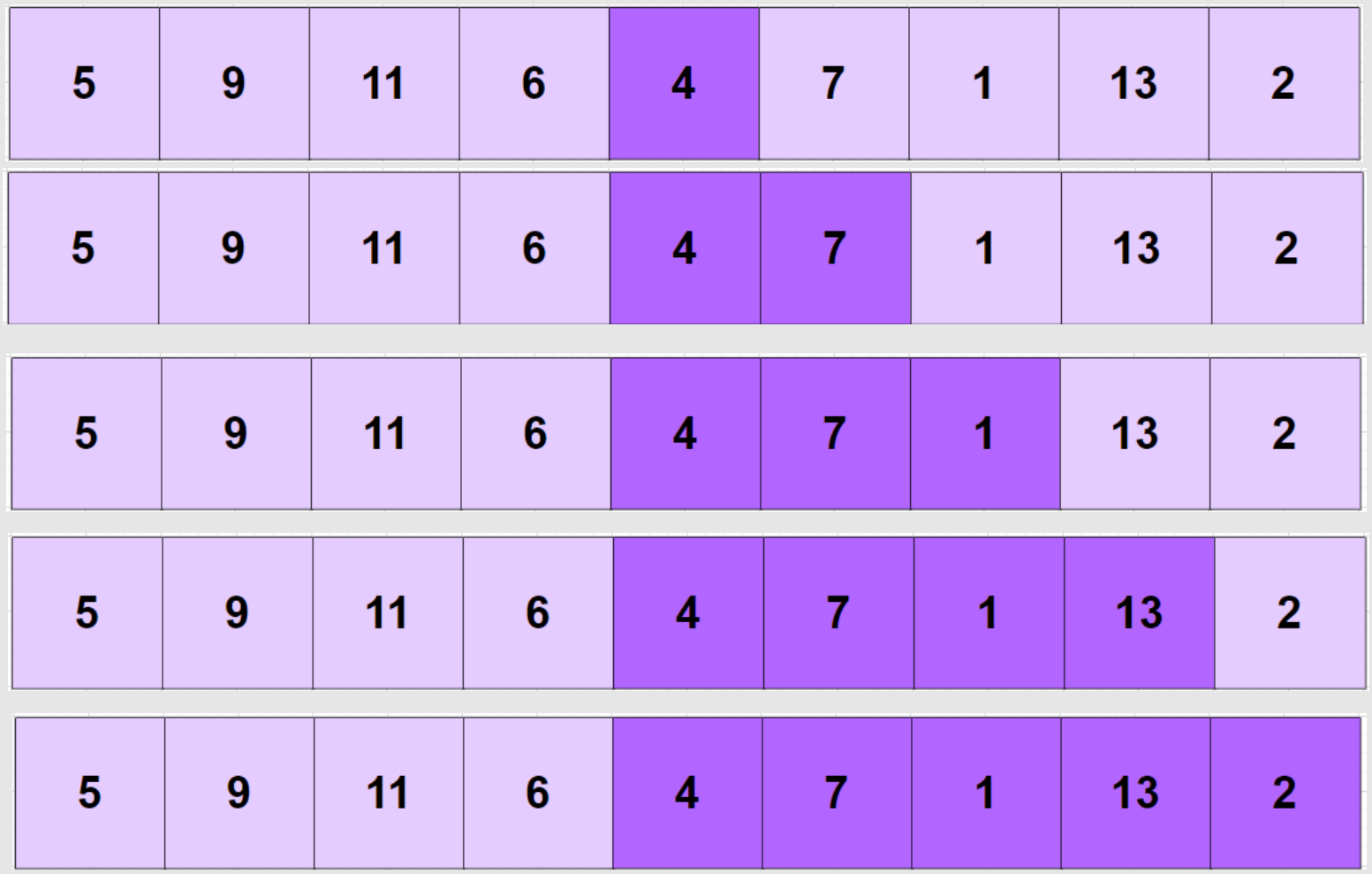
Brute Force Approach

i = 3



Brute Force Approach

i = 4



Efficiency

Time Complexity: **O(nk)**
Space Complexity: **O(n)**

Steps of Sliding Window

The Sliding Window boils down to 3 key steps.

1. Expand our window
2. Meet the condition and process the window
3. Contract our window

Sliding Window Pseudocode:

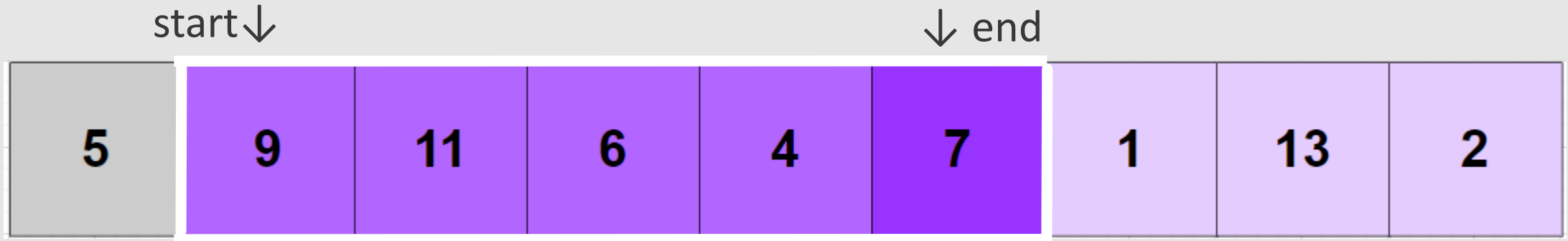
```
# Iterate over elements in our input
    # Expand the window
    # Meet the condition to stop expansion
        # Process the current window
        # Contract the window
```

Sliding Window Approach

```
1   function maxSumSubarr(arr,k){
2       if (arr.length < k){
3           return "Invalid";
4       }
5       let maxSum = -Infinity,
6           windowSum = 0,
7           windowStart = 0;
8       for (let windowEnd=0; windowEnd< arr.length; windowEnd++){
9           windowSum += arr[windowEnd]
10          if (windowEnd >= k-1){
11              maxSum = Math.max(maxSum, windowSum);
12              windowSum -= arr[windowStart];
13              windowStart += 1
14          }
15      }
16      return maxSum;
17  }
```

Sliding Window Approach

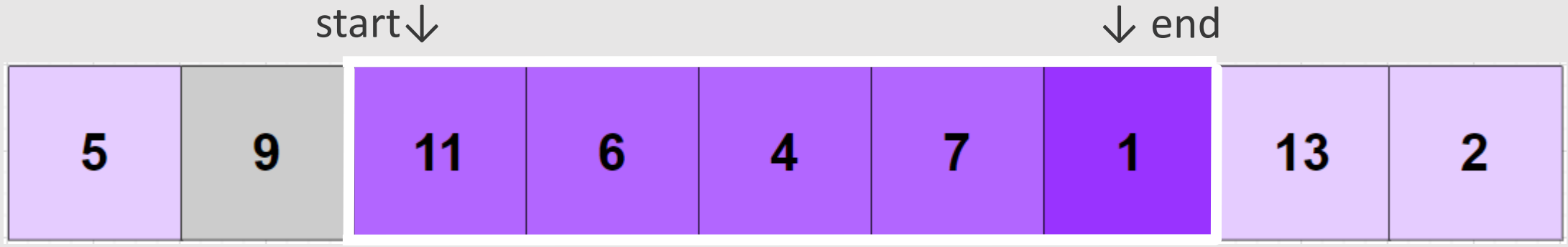
Sliding Window Approach



windowSum = 35 – 5 + 7 = 37.

maxSum = 37.

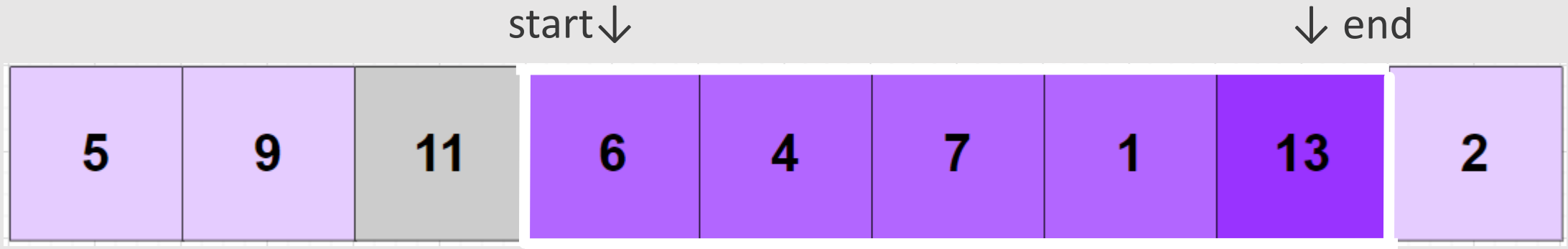
Sliding Window Approach



windowSum = 37 – 9 + 1= 29.

maxSum = 37.

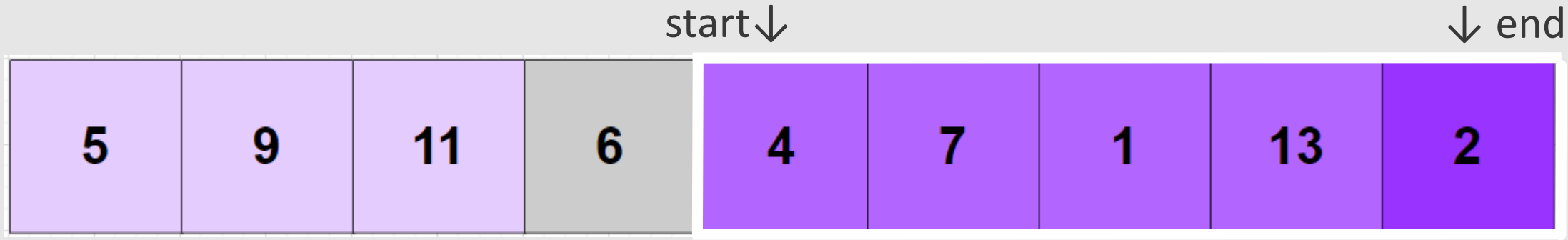
Sliding Window Approach



windowSum = 29 – 11 + 13= 31.

maxSum = 37.

Sliding Window Approach



windowSum = 31 – 6 + 2 = 27.

maxSum = 37.

Efficiency

Time Complexity: ~~O(nk)~~ → **O(n)**

Space Complexity: ~~O(n)~~ → **O(1)**

Limitations

-Data structure type: The sliding window pattern is best suited for problems involving sequential data structures such as **arrays, strings, or linked lists**. It may not be appropriate for more complex data structures such as trees or graphs.
-Keywords: longest, shortest, min, max, contains

Practice

Now try to solve this challenge on coding dojo: leetcode

Zettachring 6 · 70567 Stuttgart
Fon +49 711.901196 0
Fax +49 711.901196 111

Daimlerstraße 17/1 · 72581 Dettingen an der Erms
Fon +49 711.901196 0
Fax +49 711.901196 111

Neuenhofer Str. 11 · 42657 Solingen
Fon +49 212.2245505 0
Fax +49 212.2245505 110

Ougarit Bld. 1st floor · Al Irsal Street · Al Masayef · Ramallah
Fon +970 2 2988350
Fax +970 2 2988364

19 Hartom Street · Har Hotzvim · Jerusalem
Fon +970 2 2988350
Fax +970 2 2988364

Karatasou 7 · 54627 Thessaloniki
Fon +49 711.901196 0
Fax +49 711.901196 111