

Mastering Sliding Window Techniques



Ankit Singh · Follow

6 min read · Aug 8, 2023



179



SLIDING WINDOW TECHNIQUE

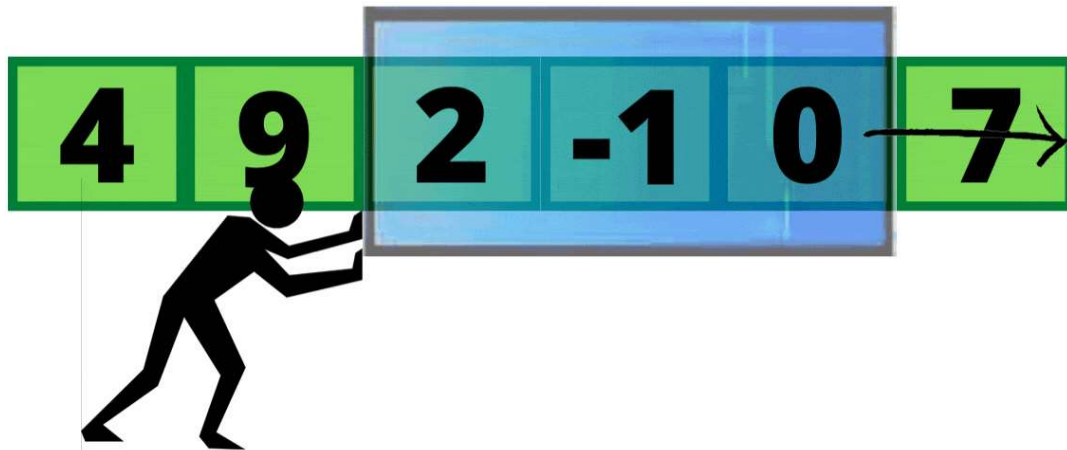


The sliding window technique is a common algorithmic approach used for solving various problems that involve processing or analyzing a sequential data structure, such as arrays, strings, or streams.

It involves creating a fixed-size window that moves through the data structure one step at a time, typically from left to right, to perform specific operations or computations on the elements within the window.

What is the Sliding Window Algorithm?

The Sliding Window algorithm is a method for finding a subset of elements that satisfy a certain condition in arrays.



The Sliding Window Algorithm is a specific technique used in computer science and programming to efficiently solve problems that involve arrays, strings, or other data structures by maintaining a “window” of elements within a certain range and moving that window through the data to perform operations or calculations.

Using the Sliding Window Technique:

The Sliding Window Technique is a powerful approach to efficiently solve problems involving arrays, strings, or sequences by maintaining a moving “**window**” of elements and performing operations as the window slides through the data. This technique helps reduce time complexity compared to brute-force methods.

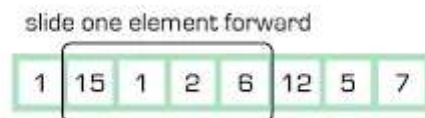
- **Determine Window Size:** Decide on a fixed window size that defines the number of elements to consider at each step.
- **Initialize and Process:** Start with the initial elements within the window. Perform any initial calculations or operations.
- **Slide the Window:** Iterate through the data, updating the window by adding the next element and removing the leftmost one.
- **Update and Evaluate:** Adjust calculations or data structures based on the new element. Evaluate if the current window meets the problem’s conditions.
- **Continue Sliding:** Repeat the sliding, updating, and evaluation steps until the end of the data is reached.
- **Return Result:** Return the final result or outcome based on the processed windows.

Problem: Given an array of integers, find the maximum sum of a subarray with a fixed window size.

Let’s consider the array: [2, 1, 5, 1, 3, 2] and a window size of 3.

1. **Initialization:** Start with the first 3 elements: [2, 1, 5] . Calculate their sum: $2 + 1 + 5 = 8$.

2. Slide the Window: Move the window one step to the right: $[1, 5, 1]$.
Calculate the sum: $1 + 5 + 1 = 7$.
3. Update and Evaluate: Compare the current sum (7) with the previous maximum sum (8). Since 8 is greater, keep the maximum sum as 8 .
4. Slide the Window: Move the window again: $[5, 1, 3]$. Calculate the sum:
 $5 + 1 + 3 = 9$.
5. Update and Evaluate: Compare the current sum (9) with the previous maximum sum (8). Update the maximum sum to 9 .
6. Slide the Window: Continue sliding the window: $[1, 3, 2]$. Calculate the sum: $1 + 3 + 2 = 6$.
7. Update and Evaluate: Compare the current sum (6) with the previous maximum sum (9). Since 9 is greater, the maximum sum remains 9 .
8. Final Result: After sliding through all windows, the maximum sum found is 9 .

[Open in app](#)[Sign up](#)[Sign in](#)[Write](#)

Implementations of the sliding window technique:

- In CPP:



a1

```
#include <bits/stdc++.h>
using namespace std;

int maxSubarraySum(vector<int>& nums, int k) {
    int maxSum = INT_MIN;
    int windowSum = 0;

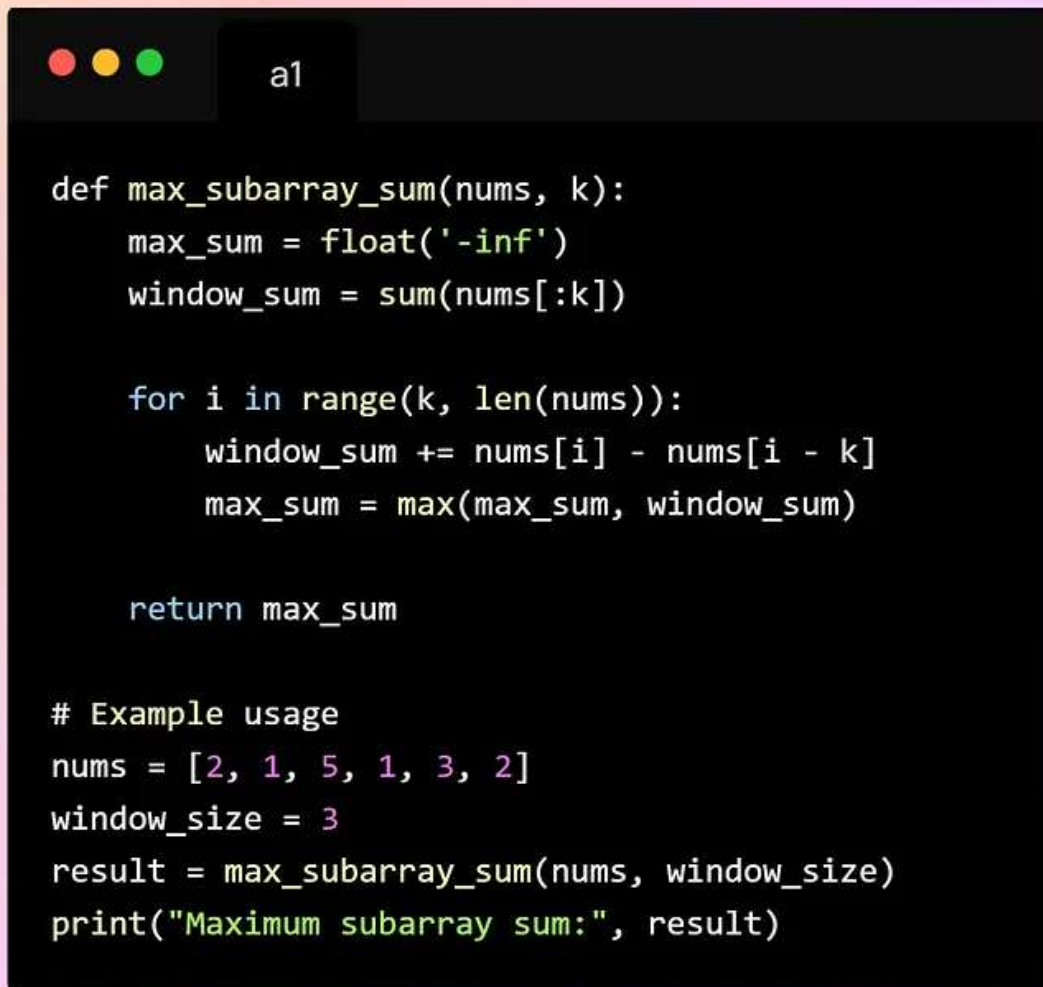
    for (int i = 0; i < k; i++) {
        windowSum += nums[i];
    }

    for (int i = k; i < nums.size(); i++) {
        windowSum += nums[i] - nums[i - k];
        maxSum = max(maxSum, windowSum);
    }

    return maxSum;
}

int main() {
    vector<int> nums = {2, 1, 5, 1, 3, 2};
    int windowSize = 3;
    int result = maxSubarraySum(nums, windowSize);
    cout << "Maximum subarray sum: " << result << endl;
    return 0;
}
```

- In Python:



```
def max_subarray_sum(nums, k):  
    max_sum = float('-inf')  
    window_sum = sum(nums[:k])  
  
    for i in range(k, len(nums)):  
        window_sum += nums[i] - nums[i - k]  
        max_sum = max(max_sum, window_sum)  
  
    return max_sum  
  
# Example usage  
nums = [2, 1, 5, 1, 3, 2]  
window_size = 3  
result = max_subarray_sum(nums, window_size)  
print("Maximum subarray sum:", result)
```

- In Java:

```
public class SlidingWindow {  
    public static int maxSubarraySum(int[] nums, int k) {  
        int maxSum = Integer.MIN_VALUE;  
        int windowSum = 0;  
  
        for (int i = 0; i < k; i++) {  
            windowSum += nums[i];  
        }  
  
        for (int i = k; i < nums.length; i++) {  
            windowSum += nums[i] - nums[i - k];  
            maxSum = Math.max(maxSum, windowSum);  
        }  
  
        return maxSum;  
    }  
  
    public static void main(String[] args) {  
        int[] nums = {2, 1, 5, 1, 3, 2};  
        int windowSize = 3;  
        int result = maxSubarraySum(nums, windowSize);  
        System.out.println("Maximum subarray sum: " + result);  
    }  
}
```

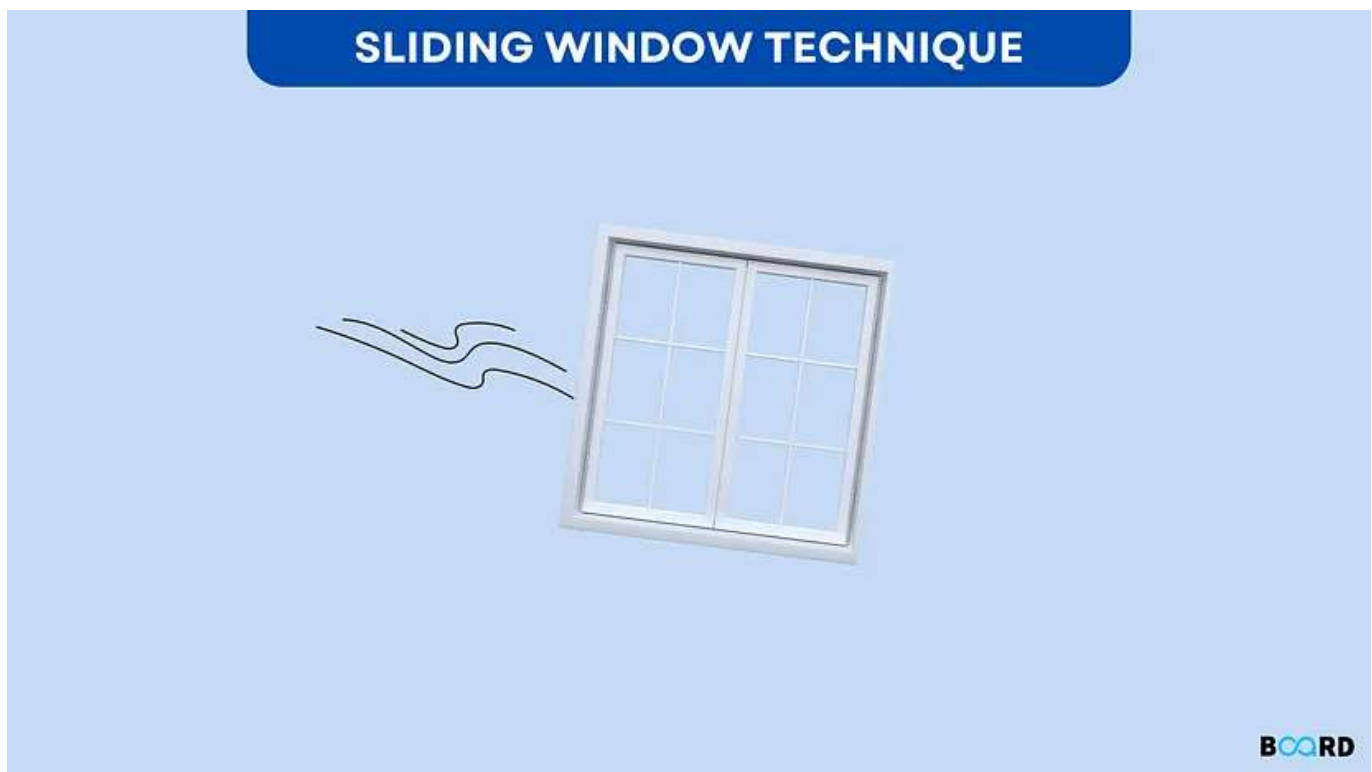
Time and Space complexity of the sliding window technique:

- **Time Complexity:**
- The time complexity of the sliding window technique is usually linear or close to linear, $O(n)$, where n is the size of the input data structure (e.g.,

array or string). This is because you process each element once as the window slides through the data.

- **Space Complexity:**
- The space complexity of the sliding window technique is generally constant, $O(1)$, because you're maintaining a fixed-size window and a few additional variables to perform calculations or store intermediate results. The amount of extra memory used doesn't grow with the input size; it remains constant regardless of the input size.

Common Problems based on the “sliding window technique”:



1. **Maximum/Minimum Subarray Sum:**
2. **Longest Substring with K Distinct Characters:**
3. **Longest Subarray with Ones after Replacement:**

4. Find All Anagrams in a String:
5. Smallest Subarray with Sum at Least K:
6. Maximum Consecutive Ones after Flipping Zeros:
7. Minimum Window Substring:
8. Longest Repeating Character Replacement:
9. Fruit Into Baskets:
10. Subarrays with Product Less than K:

Introducing the concept of a variable-size window:



Let's delve into the concept of a variable-size sliding window.

While the basic sliding window technique involves a fixed-size window that moves through the data structure, the variable-size sliding window

introduces flexibility by allowing the window size to change dynamically based on certain conditions.

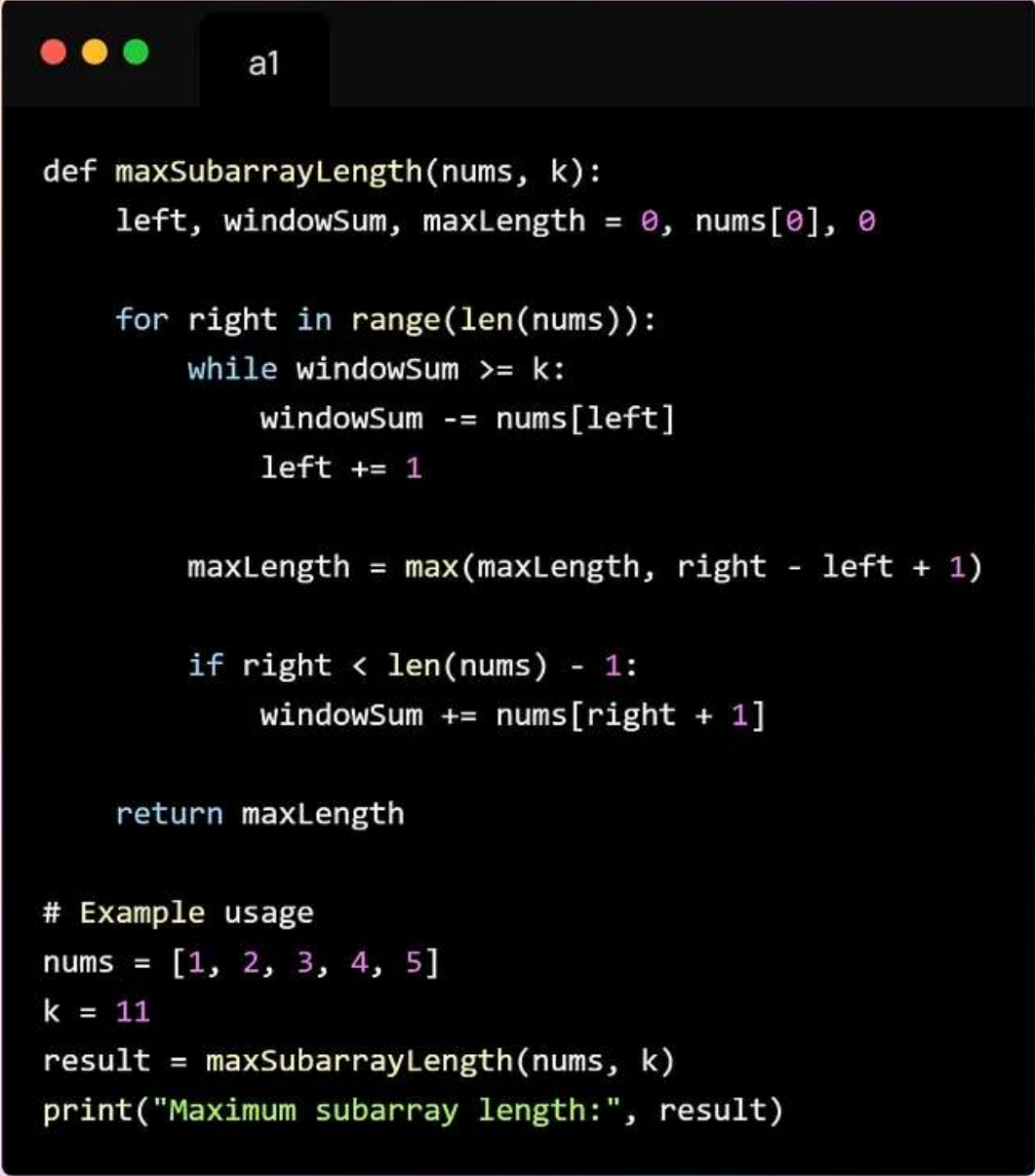
This is particularly useful when the problem involves finding a subarray or substring that satisfies certain criteria.

Variable Size Sliding Window Approach:

In this approach, instead of maintaining a fixed-size window throughout the entire process, you adjust the window size as needed. The window can grow or shrink depending on the problem's requirements.

Example Problem: Longest Subarray with Sum Less Than K

- **Problem:** Given an array of positive integers and an integer K , find the length of the longest subarray whose sum is less than K .
1. Initialize variables: `left` to track the start of the subarray and `right` to iterate through the array.
 2. Initialize `windowSum` as the first element of the array.
 3. Initialize `maxLength` to keep track of the maximum subarray length.



```
def maxSubarrayLength(nums, k):  
    left, windowSum, maxLength = 0, nums[0], 0  
  
    for right in range(len(nums)):  
        while windowSum >= k:  
            windowSum -= nums[left]  
            left += 1  
  
        maxLength = max(maxLength, right - left + 1)  
  
        if right < len(nums) - 1:  
            windowSum += nums[right + 1]  
  
    return maxLength  
  
# Example usage  
nums = [1, 2, 3, 4, 5]  
k = 11  
result = maxSubarrayLength(nums, k)  
print("Maximum subarray length:", result)
```

Conclusion:



In conclusion, the sliding window technique is a powerful and versatile algorithmic approach that provides an efficient solution for various problems involving sequential data structures like arrays and strings.

It offers a structured way to process contiguous segments of data within these structures while optimizing time complexity and sometimes space complexity.

Advantages of using the sliding window technique:

1. **Optimization:** By maintaining a window of elements, the technique avoids redundant calculations and comparisons, leading to optimized computations.
2. **Constant Space Complexity:** The sliding window technique often requires only a constant amount of additional memory, making it memory-efficient.

3. Efficiency: The sliding window technique often reduces time complexity from a naive brute-force approach to linear or nearly linear, making it well-suited for processing large datasets.



“Elevate Your Solutions with the Sliding Window Technique: Unveiling Efficiency, One Window Slide at a Time!”

Sliding Window Algorithm

Datastrucutre



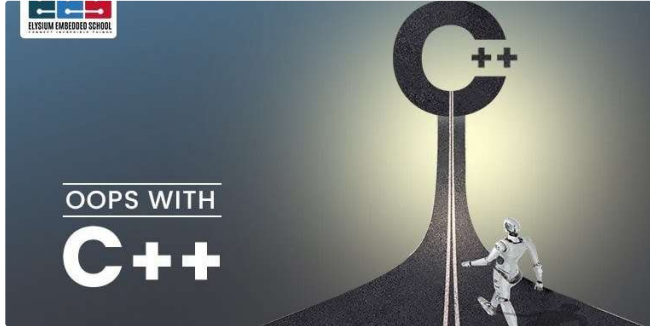
Written by Ankit Singh

23 Followers

Follow



More from Ankit Singh



Ankit Singh

“Mastering Object-Oriented Programming (OOP) in C++: A...

Before we start learning about OOP (Object-Oriented Programming), let's figure out why...

10 min read · Dec 24, 2023



57



Ankit Singh

“Node.js Database Magic: Exploring the Powers of Sequelize...

For many developers, linking an Object-relational Mapper (ORM) to a Node.js...

7 min read · Aug 29, 2023



51

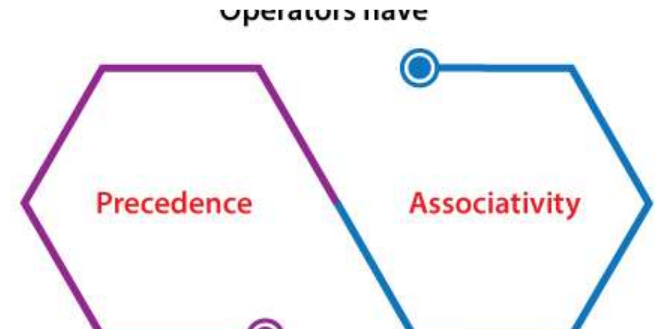


Ankit Singh

“Mastering CPU Scheduling and Dealing with Deadlocks in...

“CPU scheduling in modern operating systems is really important. It helps manage...

12 min read · Oct 29, 2023



Ankit Singh

This article will teach you about operator precedence and...

In Python, the order in which operators are evaluated in an expression is determined by...

6 min read · Aug 3, 2023

[See all from Ankit Singh](#)

Recommended from Medium



Ankit Singh

“Getting Started with Bit Manipulation Made Simple”

You might be asking yourself why learning about bit manipulation is useful.

7 min read · Nov 12, 2023



Vishesh Rawal

Question Mastering the Art of Sliding Window: A Comprehensive...

In the realm of algorithmic problem-solving, the Sliding Window technique is a powerful...

7 min read · Sep 22, 2023

Lists

**Staff Picks**

563 stories · 662 saves

**Stories to Help You Level-Up at Work**

19 stories · 430 saves

**Self-Improvement 101**

20 stories · 1242 saves

**Productivity 101**

20 stories · 1140 saves



Pratiyush Prakash in Dev Genius

Understanding Dynamic Programming with Levenshtein...

Dynamic programming, a powerful problem-solving technique in computer science. In thi...

6 min read · Jan 15



51



Priya Salvi in Stackademic

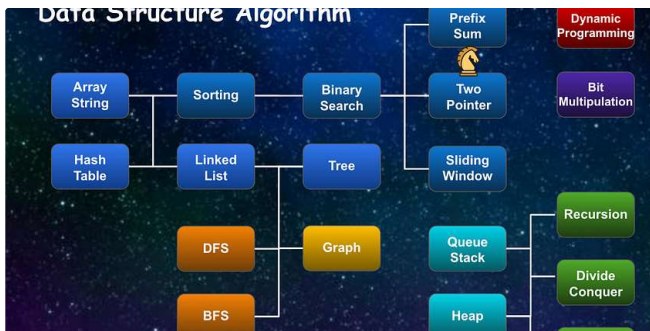
How to calculate Big O notation time complexity

As programmers, at some point in time, we have always wondered what exactly is time...

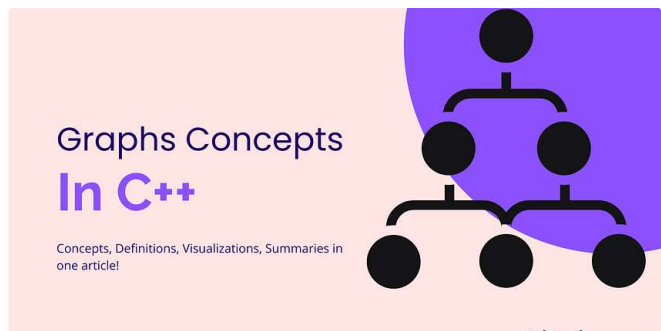
4 min read · Aug 21, 2023



27



Ting

Two Pointer in DSA

Vinay Vashisht

Graphs Data Structure in C++

The Two Pointer technique is a versatile algorithm in the realm of Data Structures an...

3 min read · Aug 27, 2023



One of the most visualizing concepts in C++ is Graphs. Starting from BFS, and DFS and...

23 min read · Aug 7, 2023



See more recommendations