A Comprehensive Guide to 14 Foundational DSA Patterns for Coding Interviews

Introduction

The prevailing philosophy for success in technical interviews posits that mastering a finite set of recurring algorithmic patterns is a more effective and scalable strategy than attempting to memorize solutions to thousands of individual problems.[1] This report serves as a strategic toolkit for deconstructing a wide array of complex problems into manageable, solvable components by leveraging these foundational patterns.

This document is structured into four primary sections, as requested: Arrays, Strings, Intervals, and Linked Lists. Each section will analyze the most critical patterns associated with that data structure. For every pattern, the analysis will follow a consistent format: a conceptual overview, heuristics for application, a representative problem with a complete C++ implementation, a detailed code walkthrough, and a formal complexity analysis. This structured approach is designed to build pattern recognition skills, enabling a systematic approach to problem-solving in a high-stakes interview environment.

Section 1: Foundational Array Patterns

Arrays are fundamental data structures, and a significant portion of interview questions revolve around their manipulation and analysis.[4] The following five patterns provide a robust framework for solving a majority of array-based problems.

Table 1: Summary of Foundational Array Patterns

Pattern Name	Core Function	Key Heuristics/Use Case	Time Complexity	Space Complexity
Sliding Window	Efficiently processes contiguous subarrays by maintaining a dynamic window.	Finding max/min in a fixed-size subarray; longest/shortest subarray satisfying a condition.	O(n)	O(1) or O(k)
Two Pointers	Uses two indices to scan an array, typically to find pairs or subarrays.	Finding pairs in a **sorted** array; problems involving palindromes or sorted data.	O(n)	O(1)
Cyclic Sort	In-place sorting for arrays containing numbers in a specific range (e.g., 1 to N).	Finding missing/duplicate numbers in an array with a specific range of values.	O(n)	O(1)
Prefix Sum	Pre-computes cumulative sums to answer range sum queries in constant time.	Problems requiring frequent calculation of sums of different subarrays.	O(n) pre-computation, O(1) per query	O(n)
Kadane's Algorithm	Finds the maximum sum of a contiguous subarray in an	The problem explicitly asks for the maximum sum of a	O(n)	O(1)

	array with negative numbers.	**contiguous** subarray.		

1.1 The Sliding Window Pattern

Conceptual Overview

The Sliding Window pattern is a technique for operating on a specific window size of a linear data structure, such as an array or string.[2] This conceptual window "slides" over the data by shifting its start and end points, allowing for the efficient processing of contiguous subarrays without redundant calculations.[7]

Application Heuristics

This pattern is applicable when the problem exhibits the following characteristics:

- The input is a linear data structure like an array, list, or string.[2]
- The problem requires calculations over a **contiguous** subarray or substring, which is a critical distinction from problems involving non-contiguous subsequences.[5]
- The objective is to find the longest/shortest subarray, maximum/minimum sum, or a specific value that satisfies a condition within that subarray.
- A brute-force solution involving nested loops (e.g., with a time complexity of O(n2)) is apparent, suggesting that an optimization to linear time is possible.[8]

Representative Problem: Maximum Sum Subarray of Size 'K'

This problem is a classic introduction to the fixed-size sliding window pattern and is cited by

multiple sources as a canonical example.[1] Given an array of integers and a number k, the task is to find the maximum sum of any contiguous subarray of size k.

C++ Implementation and Walkthrough

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Required for std::max

int maxSum(const std::vector<int>& arr, int k) {
    int n = arr.size();
    if (n < k) {
        std::cout << "Invalid Input: Array size is less than window size." << std::endl;
        return -1; // Indicate an error
    }

    // Step 1: Initialization - Compute sum of the first window
    int max_sum = 0;
    for (int i = 0; i < k; i++) {
        max_sum += arr[i];
    }

    int window_sum = max_sum;

    // Step 2: The Sliding Mechanism
    for (int i = k; i < n; i++) {
        // Update window sum in O(1) by adding the new element and subtracting the old one
        window_sum += arr[i] - arr[i - k];

        // Step 3: Tracking the Maximum
        max_sum = std::max(max_sum, window_sum);
    }

    return max_sum;
```

```
}

int main() {
    std::vector<int> arr = {1, 4, 2, 10, 23, 3, 1, 0, 20};
    int k = 4;
    std::cout << "Maximum sum of a subarray of size " << k << " is " << maxSum(arr, k) << std::endl; // Output: 39
    return 0;
}
```

Code Walkthrough:

1. **Initialization:** The process begins by calculating the sum of the initial window of size k. This sum is stored in both window_sum and max_sum, establishing a baseline for comparison.[8]
2. **The Sliding Mechanism:** The core logic resides in the for loop that starts from the k-th element. The expression window_sum += arr[i] - arr[i - k] is the source of the pattern's efficiency. Instead of re-calculating the sum of the entire new window, it performs an O(1) update by adding the element entering the window (arr[i]) and subtracting the element leaving it (arr[i - k]).[8]
3. **Tracking the Maximum:** In each iteration, the updated window_sum is compared with max_sum, and max_sum is updated if the current window's sum is greater.

Complexity Analysis

- **Time Complexity:** O(n). The algorithm makes a single pass through the array.[8]
- **Space Complexity:** O(1). The algorithm uses a constant number of variables to store the sums and pointers.[8]

1.2 The Two Pointers Pattern

Conceptual Overview

The Two Pointers pattern utilizes two indices that traverse a data structure, often from

opposite ends or in the same direction. By intelligently moving these pointers based on conditions, it can efficiently find elements that satisfy a certain constraint, significantly reducing the search space compared to brute-force methods.[2]

Application Heuristics

This pattern is a strong candidate when:

- The problem involves a **sorted** array or linked list. The sorted property is fundamental to the decision-making logic of moving the pointers.[1]
- The task is to find a pair, a triplet, or a subarray that fulfills a specific constraint, such as adding up to a target sum.[2]
- A brute-force O(n2) solution exists, but the sorted nature of the input suggests that a more optimal O(n) solution is achievable.[2]

Representative Problem: Pair with Target Sum

This is a canonical problem for this pattern: given a sorted array of integers and a target value, find if there exists a pair of elements whose sum is equal to the target.[12]

C++ Implementation and Walkthrough

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // For std::sort

// Function assumes the input array A is sorted
bool isPairSum(const std::vector<int>& A, int X) {
    int N = A.size();
```

```
    // Step 1: Initialization
    int left = 0;
    int right = N - 1;

    // Step 2: The Core Loop
    while (left < right) {
        int current_sum = A[left] + A[right];

        // Step 3: Decision Logic
        if (current_sum == X) {
            return true; // Pair found
        } else if (current_sum < X) {
            left++; // Sum is too small, need a larger value from the left
        } else { // current_sum > X
            right--; // Sum is too large, need a smaller value from the right
        }
    }

    return false; // No such pair found
}

int main() {
    std::vector<int> arr = {2, 3, 5, 8, 9, 10, 11};
    int val = 17;
    // The array is already sorted for this example
    if (isPairSum(arr, val)) {
        std::cout << "Pair with sum " << val << " exists." << std::endl;
    } else {
        std::cout << "Pair with sum " << val << " does not exist." << std::endl;
    }
    return 0;
}
```

Code Walkthrough:

1. **Initialization:** A left pointer is set to the start of the array (index 0), and a right pointer is set to the end (index N-1).[13]
2. **The Core Loop:** The while (left < right) loop continues as long as the pointers have not crossed.
3. **Decision Logic:** This is the critical component. The sum of the values at the two pointers is compared to the target.
    - If the sum equals the target, the pair is found.
    - If the sum is less than the target, the sum must be increased. Since the array is

sorted, the only way to achieve this is by moving the left pointer to the right to select a larger number (left++).[13]
    - If the sum is greater than the target, the sum must be decreased. This is achieved by moving the right pointer to the left to select a smaller number (right--).[13]

Complexity Analysis

- **Time Complexity:** O(n). In each iteration, either the left pointer moves right or the right pointer moves left, ensuring that the pointers together make only one pass through the array's elements. If the array is not pre-sorted, the total complexity is dominated by the initial sort, making it O(nlogn).[11]
- **Space Complexity:** O(1). No extra space proportional to the input size is used.[11]

1.3 The Cyclic Sort Pattern

Conceptual Overview

Cyclic Sort is a specialized, in-place sorting algorithm designed for arrays that contain numbers within a specific, contiguous range, such as 1 to N.[15] The fundamental principle is to iterate through the array and place each number at its correct index. For an array with numbers from 1 to N, the number

x should be at index x-1. This is achieved by swapping elements until every number is in its rightful place.[17]

Application Heuristics

This pattern is highly effective when:

- The input is an array of numbers.
- The numbers fall within a known range, typically [1, N] or [0, N-1], where N is the length of

the array.[1]
- The problem involves finding missing numbers, duplicate numbers, or the smallest missing positive number in such an array.[1][2]
- The solution must be in-place, with a space complexity of O(1).

Representative Problem: Find the Missing Number

Given an array containing N distinct numbers taken from the range 1 to N+1, find the missing number.[1] A common variation uses the range 1 to N with one missing number.

C++ Implementation and Walkthrough

C++

```cpp
#include <iostream>
#include <vector>
#include <numeric> // For std::swap

// This function sorts an array containing numbers from 1 to N in-place.
void cyclicSort(std::vector<int>& arr) {
    int i = 0;
    int n = arr.size();
    // Step 1: The 'while' Loop
    while (i < n) {
        // Step 2: Finding the Correct Position
        // For a 1-based array, value 'x' should be at index 'x-1'.
        int correct_index = arr[i] - 1;

        // Step 3: The Swap Logic
        if (correct_index < n && arr[i]!= arr[correct_index]) {
            std::swap(arr[i], arr[correct_index]);
        } else {
            i++;
        }
```

```
    }
}

int findMissingNumber(std::vector<int>& arr) {
    cyclicSort(arr);
    int n = arr.size();
    // Step 4: Finding the Missing Number
    for (int i = 0; i < n; ++i) {
        if (arr[i]!= i + 1) {
            return i + 1;
        }
    }
    // If all numbers from 1 to N are present, the missing number is N+1.
    return n + 1;
}

int main() {
    std::vector<int> nums = {3, 1, 5, 4}; // Missing 2
    std::cout << "The missing number is: " << findMissingNumber(nums) << std::endl;
    return 0;
}
```

Code Walkthrough:

1. **The while Loop:** The algorithm uses a while loop instead of a standard for loop. The index i is only incremented when the number at arr[i] is already in its correct position. This ensures that we process the newly swapped-in number at index i before moving on.
2. **Finding the Correct Position:** The line int correct_index = arr[i] - 1; calculates the index where the current value arr[i] is supposed to be in a sorted 1-to-N array.[16]
3. **The Swap Logic:** The if condition checks if the current number is not at its correct index. If it isn't, it's swapped with the number at its correct index. If it is already in place, i is incremented to examine the next element.[16]
4. **Finding the Missing Number:** After the cyclicSort function completes, the array is nearly sorted. A final pass through the array identifies the first index i where the element arr[i] is not equal to i + 1. This indicates that i + 1 is the missing number.

Complexity Analysis

- **Time Complexity:** O(n). Although there is a nested loop structure (the while loop and the implicit loop of swaps), each number is swapped at most once to reach its correct position. Therefore, the total number of swaps is bounded by N, leading to a linear time

complexity.[16]
- **Space Complexity:** O(1). The sort is performed in-place.[16]

1.4 The Prefix Sum Pattern

Conceptual Overview

The Prefix Sum, or Cumulative Sum, technique is a form of pre-computation. It involves creating an auxiliary array, prefixSum, where each element prefixSum[i] stores the sum of all elements from the original array's start up to index i.[7] This pre-computation allows for rapid calculation of sums over any contiguous subarray.

Application Heuristics

This pattern is ideal when:

- The problem involves frequent queries for the sum of elements within various ranges (subarrays) of a static array.[7]
- The goal is to answer these range sum queries in O(1) time after an initial pre-computation step.[7]
- Problems like finding an "Equilibrium Index" (where the sum of left elements equals the sum of right elements) can be significantly optimized using this pattern.[21]

Representative Problem: Efficiently Answering Multiple Range Sum Queries

Given an array, prepare a data structure that can answer multiple queries for the sum of elements between indices L and R (inclusive) in constant time.

C++ Implementation and Walkthrough

C++

```
#include <iostream>
#include <vector>

class NumArray {
private:
    std::vector<int> prefixSum;

public:
    NumArray(const std::vector<int>& nums) {
        if (nums.empty()) {
            return;
        }
        // Step 1: Pre-computation
        prefixSum.resize(nums.size());
        prefixSum = nums;
        for (int i = 1; i < nums.size(); ++i) {
            prefixSum[i] = prefixSum[i - 1] + nums[i];
        }
    }

    // Step 2: Answering Queries
    int sumRange(int L, int R) {
        if (prefixSum.empty() |

| L < 0 |
| R >= prefixSum.size() |
| L > R) {
            return 0; // Or throw an error
        }
        if (L == 0) {
            return prefixSum;
        }
        return prefixSum - prefixSum[L - 1];
    }
```

```
};

int main() {
    std::vector<int> nums = {10, 20, 10, 5, 15};
    NumArray* obj = new NumArray(nums);
    std::cout << "Sum of range : " << obj->sumRange(2, 4) << std::endl; // Output: 30
    std::cout << "Sum of range : " << obj->sumRange(0, 2) << std::endl; // Output: 40
    delete obj;
    return 0;
}
```

Code Walkthrough:

1. **Pre-computation:** The constructor of the NumArray class builds the prefixSum array. It initializes prefixSum with nums. Then, it iterates from the second element, calculating each subsequent prefix sum using the formula prefixSum[i] = prefixSum[i - 1] + nums[i].[19]
2. **Answering Queries:** The sumRange(L, R) function leverages the pre-computed array. The sum of the subarray from index L to R is the total sum up to R minus the total sum up to L-1. This is calculated as prefixSum - prefixSum[L - 1]. An edge case is handled for when L is 0, in which case the sum is simply prefixSum.

Complexity Analysis

- **Time Complexity:** O(n) for the one-time pre-computation. Each subsequent call to sumRange is O(1).[19]
- **Space Complexity:** O(n) to store the prefixSum array.[19]

  This pattern illustrates a classic space-time tradeoff: by using extra memory, we gain a significant improvement in the speed of subsequent query operations.4

1.5 Kadane's Algorithm

Conceptual Overview

Kadane's Algorithm is a dynamic programming approach tailored to solve the "Maximum Subarray Sum" problem in linear time.[7] It is particularly powerful because it handles arrays containing both positive and negative numbers. The algorithm iterates through the array, maintaining the maximum sum of a contiguous subarray that

ends at the current position. The overall maximum sum is then the maximum of these values found during the scan.[20]

Application Heuristics

This algorithm is the optimal choice when:

- The problem explicitly asks for the maximum sum of a **contiguous** subarray.[7]
- The input array may contain negative numbers. The presence of negative numbers introduces the core challenge: deciding whether to extend an existing subarray or to start a new one, as a negative number might reduce a positive running sum.[24]

Representative Problem: Maximum Subarray Sum

Given an integer array, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.[23]

C++ Implementation and Walkthrough

C++

```
#include <iostream>
#include <vector>
#include <algorithm>

int maxSubarraySum(const std::vector<int>& arr) {
```

```
    if (arr.empty()) {
        return 0; // Or handle as an error
    }

    // Step 1: Initialization
    int max_so_far = arr;
    int max_ending_here = arr;

    for (size_t i = 1; i < arr.size(); ++i) {
        // Step 2: The Core Logic
        max_ending_here = std::max(arr[i], max_ending_here + arr[i]);

        // Step 3: Updating Global Maximum
        max_so_far = std::max(max_so_far, max_ending_here);
    }

    return max_so_far;
}

int main() {
    std::vector<int> nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    std::cout << "The maximum subarray sum is: " << maxSubarraySum(nums) << std::endl; // Output:
6
    return 0;
}
```

Code Walkthrough:

1. **Initialization:** Two key variables are initialized with the value of the first element of the array: max_so_far stores the global maximum sum found, and max_ending_here stores the maximum sum of a subarray ending at the current index i.[23]
2. **The Core Logic:** The loop iterates from the second element. The line max_ending_here = std::max(arr[i], max_ending_here + arr[i]); represents the central decision of the algorithm. For each element, it chooses the better of two options: (1) start a new subarray beginning with the current element arr[i], or (2) extend the best subarray that ended at the previous position by adding arr[i] to it. If max_ending_here was negative, this logic effectively discards the previous sum and starts fresh.[23]
3. **Updating Global Maximum:** The line max_so_far = std::max(max_so_far, max_ending_here); ensures that max_so_far always holds the highest max_ending_here value seen throughout the traversal.

Complexity Analysis

- **Time Complexity:** O(n), as the algorithm involves a single pass through the array.[23]
- **Space Complexity:** O(1), as it only uses a few variables to track the sums.[23]

The patterns within this section demonstrate a spectrum of applicability. Two Pointers and Sliding Window are general-purpose *techniques* that can be adapted to a wide variety of problems involving sorted or contiguous data.[2] In contrast, Cyclic Sort and Kadane's Algorithm are highly specialized

algorithms, each designed to solve a very specific problem statement with optimal efficiency.[16] An effective problem-solver first identifies the general problem category (e.g., "this is a subarray problem") and then checks for special conditions that might allow for a specialized, more efficient algorithm (e.g., "it asks for max contiguous sum with negatives" -> Kadane's; "it's an array with numbers from 1 to N" -> Cyclic Sort). This layered approach to pattern recognition signifies a deeper understanding of the algorithmic landscape.

Section 2: Core String Manipulation Patterns

Strings, as sequences of characters, often share patterns with arrays. However, they also have unique challenges related to lexicographical properties and pattern matching. This section covers four patterns essential for string-based problems.

Table 2: Summary of Core String Manipulation Patterns

Pattern Name	Core Function	Key Heuristics/Use Case	Time Complexity	Space Complexity
Two Pointers	Validates string properties by comparing characters from both ends.	Palindrome checking, reversing a string.	O(n)	O(1)

Sliding Window	Finds substrings satisfying constraints using a dynamic window and frequency map.	Longest substring without repeats, smallest window containing a pattern.	O(n)	O(k) (k=charset size)
Rabin-Karp	Efficiently finds a pattern string in a text string using a rolling hash.	Substring search, plagiarism detection.	Average: O(n+m)	O(n+m)
Longest Common Substring	Finds the longest contiguous substring common to two strings using DP.	DNA sequence analysis, text comparison.	O(n·m)	O(n·m) or O(min(n,m))

2.1 Two Pointers for String Validation

Conceptual Overview

This is a direct application of the Two Pointers pattern to strings, which are fundamentally arrays of characters.[5] Pointers are typically placed at the start and end of the string and moved inwards to check for symmetric properties like palindromes.[27]

Application Heuristics

This pattern is used when:

- The problem involves checking if a string is a palindrome.[27]
- The task requires reversing a string or parts of a string in-place.

Representative Problem: Check if a String is a Palindrome

Determine if a given string reads the same forwards and backwards.[27]

C++ Implementation and Walkthrough

C++

```
#include <iostream>
#include <string>

bool isPalindrome(const std::string& s) {
    int left = 0;
    int right = s.length() - 1;

    while (left < right) {
        if (s[left]!= s[right]) {
            return false; // Mismatch found
        }
        left++;
        right--;
    }
    return true; // All characters matched
}

int main() {
```

```
    std::string str1 = "racecar";
    std::string str2 = "hello";
    std::cout << str1 << " is a palindrome: " << (isPalindrome(str1)? "Yes" : "No") << std::endl;
    std::cout << str2 << " is a palindrome: " << (isPalindrome(str2)? "Yes" : "No") << std::endl;
    return 0;
}
```

Code Walkthrough:
The implementation initializes a left pointer at index 0 and a right pointer at the last index. The while (left < right) loop continues until the pointers meet or cross. In each step, it compares the characters at s[left] and s[right]. If a mismatch is found, the string is not a palindrome. If the characters match, the pointers are moved closer to the center (left++, right--).27 If the loop completes without a mismatch, the string is a palindrome.

Complexity Analysis

- **Time Complexity:** O(n), where n is the length of the string, as it involves a single pass through about half the string.[27]
- **Space Complexity:** O(1), as no extra space is used.[27]

2.2 Sliding Window for Substring Analysis

Conceptual Overview

The Sliding Window pattern is adapted for strings to find substrings that satisfy certain conditions. This often involves a variable-size window and a frequency map (e.g., a hash map or an array) to keep track of the characters and their counts within the current window.[9]

Application Heuristics

This pattern is highly suitable for:

- Finding the longest or shortest **substring** with a specific property.[2]
- Problems such as finding the longest substring with no repeating characters, the smallest window containing all characters of another string, or finding all anagrams of a pattern within a larger string.[2]

Representative Problem: Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring that does not contain repeating characters.[9]

C++ Implementation and Walkthrough

C++

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

int lengthOfLongestSubstring(const std::string& s) {
    std::vector<int> charIndex(128, -1); // ASCII character set
    int longest = 0;
    int left = 0;

    for (int right = 0; right < s.length(); ++right) {
        // If the character is already in the window, move the left pointer
        if (charIndex[s[right]] >= left) {
            left = charIndex[s[right]] + 1;
        }
        // Store the next index of the current character
        charIndex[s[right]] = right;
        // Update the longest substring length
        longest = std::max(longest, right - left + 1);
    }
```

```
    return longest;
}

int main() {
    std::string str = "abcabcbb";
    std::cout << "Length of longest substring: " << lengthOfLongestSubstring(str) << std::endl; // Output: 3
    return 0;
}
```

Code Walkthrough:
This implementation uses a vector charIndex as a hash map to store the most recent index of each character encountered.

1. **Window Expansion:** The right pointer iterates through the string from left to right, expanding the window.
2. **Window Contraction:** For each character s[right], it checks if that character has been seen before *within the current window* (charIndex[s[right]] >= left). If it has, a duplicate is found. The left pointer is then jumped forward to the position right after the last occurrence of that character (charIndex[s[right]] + 1), effectively shrinking the window to exclude the duplicate.
3. **State Update:** The index of the current character s[right] is updated in charIndex. The length of the current valid window (right - left + 1) is compared with longest to keep track of the maximum length found so far.

Complexity Analysis

- **Time Complexity:** O(n). Each character is visited by the right pointer once, and the left pointer only moves forward, ensuring a single pass over the string.[32]
- **Space Complexity:** O(k), where k is the size of the character set (e.g., 128 for ASCII). This is considered constant space, O(1), as it does not depend on the input string's length.[32]

2.3 Rabin-Karp Algorithm

Conceptual Overview

The Rabin-Karp algorithm is a sophisticated pattern-searching algorithm that employs hashing to find occurrences of a pattern string within a larger text string.[33] Its defining feature is the "rolling hash," a technique that allows the hash of a new substring to be calculated in

O(1) time from the hash of the previous substring, avoiding the need to re-hash from scratch at every position.

Application Heuristics

This algorithm is appropriate when:

- The problem is to find all occurrences of a pattern p in a text t.
- A simple brute-force check with complexity O(n·m) is too slow, and a more efficient approach is needed.[33]
- The algorithm is particularly advantageous when searching for multiple different patterns in the same text, as the hash for the text can be pre-computed once.

Representative Problem: Implement Rabin-Karp for Pattern Searching

Given a text string txt and a pattern string pat, find all occurrences of pat in txt.

C++ Implementation and Walkthrough

C++

```
#include <iostream>
#include <string>
#include <vector>

void search(const std::string& pat, const std::string& txt, int q) {
```

```
    int M = pat.length();
    int N = txt.length();
    int i, j;
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;
    int d = 256; // number of characters in the input alphabet

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;

    // Calculate the hash value of pattern and first window of text
    for (i = 0; i < M; i++) {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= N - M; i++) {
        // Check the hash values of current window of text and pattern.
        // If the hash values match then only check for characters one by one
        if (p == t) {
            // Check for characters one by one
            for (j = 0; j < M; j++) {
                if (txt[i + j]!= pat[j])
                    break;
            }
            if (j == M)
                std::cout << "Pattern found at index " << i << std::endl;
        }

        // Calculate hash value for next window of text: Remove leading digit, add trailing digit
        if (i < N - M) {
            t = (d * (t - txt[i] * h) + txt[i + M]) % q;
            // We might get negative value of t, converting it to positive
            if (t < 0)
                t = (t + q);
        }
    }
}

int main() {
```

```
    std::string txt = "GEEKS FOR GEEKS";
    std::string pat = "GEEK";
    int q = 101; // A prime number
    search(pat, txt, q); // Output: Pattern found at index 0, Pattern found at index 10
    return 0;
}
```

Code Walkthrough:

1. **Hashing:** The algorithm uses a polynomial rolling hash. The hash for the initial window of the text and the pattern is calculated.
2. **Rolling Hash:** In the main loop, as the window slides, the hash for the new window is calculated efficiently in O(1) time. This is done by mathematically removing the contribution of the character leaving the window and adding the contribution of the character entering it.[35]
3. **Collision Handling:** A critical aspect of Rabin-Karp is handling hash collisions (when two different strings produce the same hash). If the hash of the current text window matches the pattern's hash, a character-by-character comparison is performed to confirm a true match and rule out a false positive.[35]

Complexity Analysis

- **Average/Best Case Time Complexity:** O(n+m), where n is the length of the text and m is the length of the pattern.
- **Worst Case Time Complexity:** O(n·m). This occurs in the pathological case of frequent hash collisions, forcing the algorithm to degrade to brute-force character checking.
- **Space Complexity:** O(1) for the basic implementation. Implementations that pre-compute hashes may use more space.[35]

The contrast between a probabilistic algorithm like Rabin-Karp and a deterministic one like KMP (Knuth-Morris-Pratt) highlights a fundamental choice in algorithm design. Rabin-Karp is probabilistic; it relies on the low probability of hash collisions to achieve excellent average-case performance, but it must include a fallback character check for correctness.[35] KMP is deterministic and guarantees

O(n+m) performance by building a finite automaton. This introduces the broader concept that for some problems, accepting a small probability of extra work can lead to a simpler and often faster real-world algorithm than a more complex deterministic one.

2.4 Longest Common Substring

Conceptual Overview

This pattern aims to find the longest string that is a **contiguous** substring of two given strings. It is a classic dynamic programming problem and should be distinguished from the "Longest Common Subsequence" (LCS) problem, where the characters of the common sequence do not need to be contiguous in the original strings.[36]

Application Heuristics

This pattern is used when:

- The problem explicitly asks for the "longest common substring" between two strings.
- The solution requires comparing all possible substrings, which suggests that a DP approach is necessary to store results of subproblems and avoid redundant computations.

Representative Problem: Find the length of the Longest Common Substring

Given two strings, X and Y, find the length of the longest common substring.

C++ Implementation and Walkthrough

C++

```
#include <iostream>
```

```cpp
#include <string>
#include <vector>
#include <algorithm>

int LCSubStr(const std::string& X, const std::string& Y) {
    int m = X.length();
    int n = Y.length();

    // DP Table: LCSuff[i][j] contains length of longest common suffix of X[0..i-1] and Y[0..j-1]
    std::vector<std::vector<int>> LCSuff(m + 1, std::vector<int>(n + 1, 0));
    int result = 0; // To store length of the longest common substring

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            // State Transition
            if (X[i - 1] == Y[j - 1]) {
                LCSuff[i][j] = LCSuff[i - 1][j - 1] + 1;
                result = std::max(result, LCSuff[i][j]);
            } else {
                LCSuff[i][j] = 0;
            }
        }
    }
    return result;
}

int main() {
    std::string X = "OldSite:GeeksforGeeks.org";
    std::string Y = "NewSite:GeeksQuiz.com";
    std::cout << "Length of Longest Common Substring is " << LCSubStr(X, Y) << std::endl; // Output: 5
(Geeks)
    return 0;
}
```

Code Walkthrough:

1. **DP Table:** A 2D DP table, LCSuff, is created. The entry LCSuff[i][j] stores the length of the longest common suffix of the prefixes X[0...i-1] and Y[0...j-1].[37]
2. **State Transition:** The table is filled iteratively. For each pair of characters X[i-1] and Y[j-1]:
    - If the characters match, the common suffix is extended. Thus, LCSuff[i][j] is set to 1 + LCSuff[i-1][j-1].
    - If the characters do not match, the contiguity is broken. The common suffix length resets to zero, so LCSuff[i][j] = 0. This is the key difference from the LCS algorithm.

3. **Finding the Result:** Since a common substring can end anywhere, the final answer is the maximum value found anywhere in the LCSuff table, which is tracked by the result variable during the iteration.[37]

Complexity Analysis

- **Time Complexity:** O(m·n), where m and n are the lengths of the two strings, due to the nested loops for filling the DP table.[37]
- **Space Complexity:** O(m·n) for the DP table. This can be optimized to O(min(m,n)) because each row's calculation only depends on the previous row.[37]

Section 3: Essential Interval Handling Patterns

Interval-based questions are common in interviews, often appearing in the context of scheduling, resource allocation, and geometric problems.[12] Mastering the two primary patterns for handling intervals is crucial.

Table 3: Summary of Essential Interval Handling Patterns

Pattern Name	Core Function	Key Heuristics/Use Case	Time Complexity	Space Complexity
Merge Intervals	Consolidates overlapping intervals into a minimal set of non-overlapping intervals.	Scheduling problems, calendar management, simplifying numeric ranges.	O(nlogn)	O(n)
Intervals Intersection	Finds the common, overlapping portions	Finding common availability slots,	O(n+m)	O(k) (k=intersections)

	between two lists of intervals.	geometric intersection problems.		

A unifying principle across many interval problems is the strategic use of sorting. By sorting intervals based on their start times, a potentially complex geometric problem is transformed into a more manageable linear scan. This pre-processing step creates an order that enables greedy, local decisions to yield a globally correct solution. For merging, it means we only need to compare an interval with the most recently processed one. For intersection, it underpins the logic of the two-pointer scan.

3.1 Merge Intervals

Conceptual Overview

The Merge Intervals pattern is used to consolidate a collection of intervals by merging any that overlap into a single, comprehensive interval.[15] The objective is to produce a final list of intervals that are mutually exclusive.[39]

Application Heuristics

This pattern is indicated when:

- The input is a collection of intervals, such as time ranges or numeric ranges.[38]
- The problem requires simplifying this set by merging any overlapping or adjacent intervals.[39]
- A common and critical prerequisite is to sort the intervals by their start times, which simplifies the merging logic.[7]

Representative Problem: Merge Intervals

Given a collection of intervals, merge all overlapping intervals.[38]

C++ Implementation and Walkthrough

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<std::vector<int>> merge(std::vector<std::vector<int>>& intervals) {
    if (intervals.empty()) {
        return {};
    }

    // Step 1: Sorting
    std::sort(intervals.begin(), intervals.end());

    std::vector<std::vector<int>> mergedIntervals;
    mergedIntervals.push_back(intervals);

    // Step 2: Iteration and Merging
    for (size_t i = 1; i < intervals.size(); ++i) {
        std::vector<int>& last = mergedIntervals.back();
        std::vector<int>& current = intervals[i];

        // Check for overlap
        if (current <= last) {
            // Merge by updating the end of the last interval
            last = std::max(last, current);
        } else {
            // No overlap, add the current interval as a new one
            mergedIntervals.push_back(current);
        }
    }
```

```
    return mergedIntervals;
}

int main() {
    std::vector<std::vector<int>> intervals = {{1, 3}, {2, 6}, {8, 10}, {15, 18}};
    std::vector<std::vector<int>> result = merge(intervals);
    for (const auto& interval : result) {
        std::cout << "[" << interval << ", " << interval << "] ";
    }
    std::cout << std::endl; // Output:
    return 0;
}
```

Code Walkthrough:

1. **Sorting:** The crucial first step is to sort the intervals based on their start times. This ensures that we can process them in a linear fashion, and any potential overlaps will be with the most recently added interval in our result set.[38]
2. **Iteration and Merging:** The algorithm initializes a result list with the first interval. It then iterates through the remaining sorted intervals. For each current interval, it compares its start time with the end time of the last interval in the result list.
    - If there is an overlap (current <= last), the intervals are merged. This is done by updating the end time of the last interval to be the maximum of its current end and the current interval's end.
    - If there is no overlap, the current interval is added as a new, distinct interval to the result list.[38]

Complexity Analysis

- **Time Complexity:** O(nlogn), which is dominated by the initial sorting step. The subsequent merging pass is O(n).[38]
- **Space Complexity:** O(n) in the worst case (if no intervals merge) to store the result. In the best case (all intervals merge into one), it is O(1).[38]

3.2 Intervals Intersection

Conceptual Overview

This pattern is used to find the common, overlapping portions between two separate lists of intervals.[41] It produces a new list of intervals, where each interval represents a segment that is present in both of the input lists.

Application Heuristics

This pattern is the correct choice when:

- The input consists of two lists of intervals. These lists are typically pairwise disjoint and sorted by their start times.[41]
- The objective is to find the set of intervals representing the intersection, such as common available time slots between two calendars.

Representative Problem: Interval List Intersections

Given two lists of closed intervals, firstList and secondList, where each list is pairwise disjoint and sorted, return the intersection of these two interval lists.[41]

C++ Implementation and Walkthrough

C++

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
std::vector<std::vector<int>> intervalIntersection(
    const std::vector<std::vector<int>>& list1,
    const std::vector<std::vector<int>>& list2) {

    std::vector<std::vector<int>> result;
    int i = 0, j = 0;

    while (i < list1.size() && j < list2.size()) {
        // Step 2: Calculating Overlap
        int start = std::max(list1[i], list2[j]);
        int end = std::min(list1[i], list2[j]);

        // Step 3: Validating and Storing
        if (start <= end) {
            result.push_back({start, end});
        }

        // Step 4: Advancing Pointers
        if (list1[i] < list2[j]) {
            i++;
        } else {
            j++;
        }
    }
    return result;
}

int main() {
    std::vector<std::vector<int>> list1 = {{0, 2}, {5, 10}, {13, 23}, {24, 25}};
    std::vector<std::vector<int>> list2 = {{1, 5}, {8, 12}, {15, 24}, {25, 26}};
    std::vector<std::vector<int>> result = intervalIntersection(list1, list2);
    for (const auto& interval : result) {
        std::cout << "[" << interval << ", " << interval << "] ";
    }
    std::cout << std::endl; // Output:
    return 0;
}
```

Code Walkthrough:

1. **Two Pointers:** The algorithm uses two pointers, i and j, to iterate through list1 and list2, respectively.[42]
2. **Calculating Overlap:** For the current pair of intervals (list1[i] and list2[j]), the potential

start of an intersection is the maximum of their start times, and the potential end is the minimum of their end times.[42]

3. **Validating and Storing:** An intersection is valid only if start <= end. If this condition holds, the new interval [start, end] is added to the result.
4. **Advancing Pointers:** This step is key to the algorithm's efficiency. The pointer associated with the interval that finishes *earlier* is advanced. For example, if list1[i] ends before list2[j], then list1[i] cannot possibly intersect with any subsequent intervals in list2, so we can safely discard it and move to the next interval in list1 by incrementing i.[42]

Complexity Analysis

- **Time Complexity:** O(n+m), where n and m are the lengths of the two lists. Each interval from both lists is processed exactly once.[42]
- **Space Complexity:** O(k), where k is the number of intersecting intervals found. In the worst case, this could be O(n+m).[42]

Section 4: Fundamental Linked List Patterns

Linked list problems test a candidate's understanding of pointers and memory references. Unlike arrays, they do not offer constant-time access, so solutions often involve clever pointer manipulation.[44]

Table 4: Summary of Fundamental Linked List Patterns

Pattern Name	Core Function	Key Heuristics/Use Case	Time Complexity	Space Complexity
Fast & Slow Pointers	Detects cycles or finds specific nodes using pointers moving at different speeds.	List has a cycle, find cycle start, find cycle length.	O(n)	O(1)

In-place Reversal	Reverses the links between nodes without using extra data structures.	Reverse a list or sub-list with an O(1) space constraint.	O(n)	O(1)
Finding the Middle	A specific application of Fast & Slow pointers to find the list's midpoint.	Find the middle node of a list in a single pass.	O(n)	O(1)

4.1 Fast & Slow Pointers (Hare & Tortoise)

Conceptual Overview

This algorithm, also known as Floyd's Cycle-Finding Algorithm, uses two pointers that traverse a list at different speeds.[2] A

slow pointer typically moves one step at a time, while a fast pointer moves two steps. This relative speed difference is the key to solving several types of linked list problems.[46]

Application Heuristics

This pattern is the go-to solution when:

- The problem involves detecting a cycle or loop in a linked list.[2]
- The task is to find the starting node of a cycle or determine its length.[47]

Representative Problem: Linked List Cycle Detection

Given the head of a linked list, determine if the list has a cycle in it.[1]

C++ Implementation and Walkthrough

C++

```
#include <iostream>

struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

bool hasCycle(ListNode *head) {
    // Step 1: Initialization
    if (head == NULL |

| head->next == NULL) {
        return false;
    }
    ListNode *slow = head;
    ListNode *fast = head;

    // Step 2: Traversal
    while (fast!= NULL && fast->next!= NULL) {
        // Step 3: Movement
        slow = slow->next;
        fast = fast->next->next;

        // Step 4: Collision
        if (slow == fast) {
            return true; // Cycle detected
```

```
        }
    }

    return false; // No cycle
}

int main() {
    ListNode* head = new ListNode(3);
    head->next = new ListNode(2);
    head->next->next = new ListNode(0);
    head->next->next->next = new ListNode(-4);
    head->next->next->next->next = head->next; // Creates a cycle

    if (hasCycle(head)) {
        std::cout << "The linked list has a cycle." << std::endl;
    } else {
        std::cout << "The linked list does not have a cycle." << std::endl;
    }
    return 0;
}
```

Code Walkthrough:

1. **Initialization:** Two pointers, slow and fast, are initialized to the head of the list.
2. **Traversal and Termination:** The while loop continues as long as fast and fast->next are not NULL. This condition is critical to prevent a segmentation fault when fast reaches the end of a non-cyclic list.
3. **Movement:** Inside the loop, slow advances by one node and fast advances by two nodes.
4. **Collision:** If at any point slow and fast point to the same node, it means the fast pointer has lapped the slow pointer within a cycle. A cycle is therefore detected, and the function returns true. If the loop terminates, it means the fast pointer reached the end of the list, so no cycle exists.[46]

Complexity Analysis

- **Time Complexity:** O(n). In a list with a cycle, the pointers will meet within a number of steps proportional to the list length. In a list without a cycle, the fast pointer traverses the list once.[49]
- **Space Complexity:** O(1). The solution uses only two pointers.[49]

4.2 In-place Reversal of a Linked List

Conceptual Overview

This fundamental pattern reverses the order of nodes in a linked list by re-wiring their next pointers. It is an "in-place" algorithm because it modifies the list structure directly without using auxiliary data structures like a stack or an array, thus achieving constant space complexity.[1]

Application Heuristics

This pattern should be used when:

- The problem explicitly asks to reverse a linked list or a sub-list.[15]
- A constraint of O(1) space complexity is given, which rules out solutions that copy node values or pointers to an auxiliary structure.[12]

Representative Problem: Reverse a Linked List

Given the head of a singly linked list, reverse the list and return the new head.[50]

C++ Implementation and Walkthrough

C++

```cpp
#include <iostream>

struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

ListNode* reverseList(ListNode* head) {
    // Step 1: Pointer Setup
    ListNode* prev = nullptr;
    ListNode* current = head;
    ListNode* nextTemp = nullptr;

    // Step 2: The Reversal Loop
    while (current!= nullptr) {
        nextTemp = current->next; // Store the next node
        current->next = prev;     // Reverse the current node's pointer

        // Move pointers one position ahead
        prev = current;
        current = nextTemp;
    }

    // Step 3: Returning the New Head
    return prev;
}

void printList(ListNode* node) {
    while(node!= NULL) {
        std::cout << node->val << " ";
        node = node->next;
    }
    std::cout << std::endl;
}

int main() {
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    printList(head); // Output: 1 2 3
    head = reverseList(head);
```

```
    printList(head); // Output: 3 2 1
    return 0;
}
```

Code Walkthrough:
This process is a careful exercise in state management. Three pointers are necessary to reverse the links without losing track of the list's structure.52

1. **Pointer Setup:** prev is initialized to nullptr (as it will become the next of the original head), current starts at head, and nextTemp is used to temporarily hold the reference to the rest of the list.[50]
2. **The Reversal Loop:** The loop iterates through the list. In each step:
    - nextTemp = current->next; saves a pointer to the next node. This is essential because the next line will overwrite current->next.
    - current->next = prev; performs the actual reversal, making the current node point backward.
    - prev = current; and current = nextTemp; advance the pointers for the next iteration.
3. **Returning the New Head:** When the loop finishes, current is nullptr, and prev is pointing to the last node of the original list, which is the new head of the reversed list.[50]

Complexity Analysis

- **Time Complexity:** O(n), as it requires one pass through the linked list.[50]
- **Space Complexity:** O(1), as the reversal is done in-place.[50]

4.3 Finding the Middle of a Linked List

Conceptual Overview

This is a specific and highly common application of the Fast & Slow Pointers pattern, often treated as a standalone pattern due to its frequency in interviews.[12] It provides an elegant single-pass solution to find the midpoint of a list without needing to know its length beforehand.

Application Heuristics

This pattern is the optimal solution when:

- The problem explicitly asks to find the middle node of a linked list.[54]
- The solution must be completed in a single pass with O(1) space, which rules out the two-pass approach of first counting the nodes and then traversing to the middle.[56]

Representative Problem: Middle of the Linked List

Given the head of a singly linked list, return the middle node. If there are two middle nodes, return the second middle node.[54]

C++ Implementation and Walkthrough

C++

```
#include <iostream>

struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

ListNode* middleNode(ListNode* head) {
    ListNode* slow = head;
    ListNode* fast = head;

    while (fast!= NULL && fast->next!= NULL) {
        slow = slow->next;
```

```
        fast = fast->next->next;
    }
    return slow;
}

int main() {
    ListNode* head1 = new ListNode(1);
    head1->next = new ListNode(2);
    head1->next->next = new ListNode(3);
    head1->next->next->next = new ListNode(4);
    head1->next->next->next->next = new ListNode(5);
    std::cout << "Middle of 1->2->3->4->5 is: " << middleNode(head1)->val << std::endl; // Output: 3

    ListNode* head2 = new ListNode(1);
    head2->next = new ListNode(2);
    head2->next->next = new ListNode(3);
    head2->next->next->next = new ListNode(4);
    head2->next->next->next->next = new ListNode(5);
    head2->next->next->next->next->next = new ListNode(6);
    std::cout << "Middle of 1->2->3->4->5->6 is: " << middleNode(head2)->val << std::endl; // Output:
4
    return 0;
}
```

Code Walkthrough:
The implementation is identical to the pointer movement in the cycle detection pattern. The key is understanding why it works for finding the middle.

- **Proportional Movement:** By the time the fast pointer reaches the end of the list (having traversed n or n-1 nodes), the slow pointer, moving at exactly half the speed, will have traversed n/2 nodes. This naturally places it at the middle of the list.[53]
- **Even/Odd Case:** The loop termination condition while (fast!= NULL && fast->next!= NULL) gracefully handles both odd and even length lists. For an odd-length list, fast will land on the last node, and fast->next will be NULL. For an even-length list, fast will become NULL after landing on the node before last. In both scenarios, slow is positioned correctly at the middle (or the second middle, as is the common convention).[54]

The Fast & Slow Pointers pattern demonstrates a powerful duality. It is used to solve two logically distinct problems: cycle detection and finding the middle. The underlying mechanical process—creating a relative speed difference between pointers—is the same. In cycle detection, this relative speed guarantees an intersection within a finite loop. In finding the middle, it creates a proportional relationship between the total distance traversed by the fast pointer and the final position of the slow pointer. Recognizing that one core mechanism can

solve two different types of problems is a significant step in mastering algorithmic patterns.

Complexity Analysis

- **Time Complexity:** O(n), as it requires a single pass through the list.[55]
- **Space Complexity:** O(1).[55]

Conclusion

The 14 patterns detailed in this report represent a foundational toolkit for modern software engineering interviews. The analysis demonstrates that a pattern-based strategy is a more robust and adaptable method for problem-solving than rote memorization. By learning to identify the underlying structure of a problem—whether it involves a contiguous subarray, a sorted sequence, overlapping intervals, or pointer manipulation in a linked list—one can select the appropriate pattern and systematically construct an efficient solution.

The true goal should extend beyond simply memorizing these patterns. A deeper understanding involves recognizing the core computer science principles they embody: the space-time tradeoffs evident in Prefix Sum, the greedy choices enabled by sorting intervals, the probabilistic efficiency of Rabin-Karp, and the meticulous state management required for in-place linked list reversal.

For continued development, it is recommended to practice problems on platforms like LeetCode and GeeksforGeeks, with a deliberate focus on first identifying the relevant pattern before proceeding to implementation.[30] This active practice of pattern recognition is the key to internalizing the heuristics presented in this report and developing the confidence to solve unfamiliar problems under pressure.

Works cited

1. 14 Essential Patterns to Master Coding Interviews: A Cheat Sheet | by Vaibhav Vudayagiri, accessed on September 12, 2025, https://medium.com/@vaibhav.vudayagiri/14-essential-patterns-to-master-coding-interviews-a-cheat-sheet-e045c639bd1e
2. 14 Patterns to Ace Any Coding Interview Question | HackerNoon, accessed on September 12, 2025, https://hackernoon.com/14-patterns-to-ace-any-coding-interview-question-c5bb3357f6ed

3. Important DSA Patterns | 100% Must-Know for Cracking Coding Interviews - LeetCode, accessed on September 12, 2025, https://leetcode.com/discuss/study-guide/5908573/Important-DSA-Patterns-100-to-Crack-Coding-Interviews/
4. Data structures and algorithms study cheatsheets for coding interviews, accessed on September 12, 2025, https://www.techinterviewhandbook.org/algorithms/study-cheatsheet/
5. Array cheatsheet for coding interviews - Tech Interview Handbook, accessed on September 12, 2025, https://www.techinterviewhandbook.org/algorithms/array/
6. Sliding Window Technique - GeeksforGeeks | PDF | Pointer (Computer Programming) | Time Complexity - Scribd, accessed on September 12, 2025, https://www.scribd.com/document/881912887/Sliding-Window-Technique-GeeksforGeeks
7. 14 DSA Pattern to solve problems efficiently | by Gagan Bansal - Medium, accessed on September 12, 2025, https://medium.com/@gaganbansal475/14-dsa-pattern-to-solve-problems-efficiently-fefa463ae5e4
8. Sliding Window Technique - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/window-sliding-technique/
9. How to Solve Sliding Window Problems | by Sergey Piterman | Outco - Medium, accessed on September 12, 2025, https://medium.com/outco/how-to-solve-sliding-window-problems-28d67601a66
10. GeeksforGeeks - Maximum Sum Subarray of Size K - Code Recipe, accessed on September 12, 2025, https://www.code-recipe.com/post/maximum-sum-subarray-of-size-k
11. Two Pointers Technique - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/two-pointers-technique/
12. Mastering the 20 Coding Patterns for Interviews - Design Gurus, accessed on September 12, 2025, https://www.designgurus.io/blog/grokking-the-coding-interview-patterns
13. C++ Program for Two Pointers Technique - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/cpp-program-for-two-pointers-technique/
14. DATA STRUCTURES AND ALGORITHMS. Two Pointers Technique | by John Kamau, accessed on September 12, 2025, https://medium.com/@johnnyJK/data-structures-and-algorithms-907a63d691c1
15. Ultimate Coding Patterns Cheat Sheet for Tech Interviews - Design Gurus, accessed on September 12, 2025, https://www.designgurus.io/blog/coding-patterns-for-tech-interviews
16. Cycle Sort - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/cycle-sort/
17. Cyclic Sort - A Hidden Gem.. A sorting technique which is not much... | by Amaan Tarique | Medium, accessed on September 12, 2025, https://medium.com/@amaantarique/cyclic-sort-a-hidden-gem-efb4ce7caab1

18. Cyclic Sort | Important pattern - Discuss - LeetCode, accessed on September 12, 2025, https://leetcode.com/discuss/study-guide/2958275/cyclic-sort-important-pattern
19. Prefix Sum Array - Implementation - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/prefix-sum-array-implementation-applications-competitive-programming/
20. This 5 Patterns Help You Solve Some of the Most Asked DSA Questions in Arrays - Medium, accessed on September 12, 2025, https://medium.com/@rakeshkumarr/this-5-patterns-help-you-solve-some-of-the-most-asked-dsa-questions-in-arrays-cab00129857c
21. Prefix Sum Technique - Tutorial - takeUforward, accessed on September 12, 2025, https://takeuforward.org/data-structure/prefix-sum-technique/
22. Top Problems on Prefix Sum Technique for Interviews - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/top-problems-on-prefix-sum-technique-for-interviews/
23. Maximum Subarray Sum - Kadane's Algorithm - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/largest-sum-contiguous-subarray/
24. Kadane's Algorithm : Maximum Subarray Sum in an Array - Tutorial - takeUforward, accessed on September 12, 2025, https://takeuforward.org/data-structure/kadanes-algorithm-maximum-subarray-sum-in-an-array/
25. Maximum Subarray Sum in C++ - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/maximum-subarray-sum-in-cpp/
26. Maximum Subarray - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/maximum-subarray/
27. Palindrome String - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/palindrome-string/
28. C++ Program to Check if a Given String is Palindrome or Not - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/c-program-to-check-if-a-given-string-is-palindrome-or-not/
29. Check if a given String is palindrome using two pointer in C++ - Tutorialspoint, accessed on September 12, 2025, https://www.tutorialspoint.com/check-if-a-given-string-is-palindrome-using-two-pointer-in-cplusplus
30. Grokking the coding interview equivalent leetcode problems - GitHub Gist, accessed on September 12, 2025, https://gist.github.com/tykurtz/3548a31f673588c05c89f9ca42067bc4
31. Learn Two Pointers and Sliding Window with Abhinav Awasthi| GeeksforGeeks - YouTube, accessed on September 12, 2025, https://www.youtube.com/watch?v=8zBzUAQH5y8
32. Sliding Window Problems | Identify, Solve and Interview Questions ..., accessed on

September 12, 2025, https://www.geeksforgeeks.org/dsa/sliding-window-problems-identify-solve-and-interview-questions/
33. Naive Algorithms In C++ (Pattern Matching - HeyCoach | Blogs, accessed on September 12, 2025, https://heycoach.in/blog/naive-algorithms-in-c-pattern-matching/
34. String Problems implementations ,which all data structures and Algorithms are used ?? | by Krishna Gupta | Medium, accessed on September 12, 2025, https://medium.com/@krishnacse20/string-problems-implementations-which-all-data-structures-and-algorithms-are-used-b3d17b45c1e1
35. Rabin-Karp Algorithm for Pattern Searching - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/
36. Longest Common Subsequence - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/longest-common-subsequence/
37. Longest Common Substring - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/longest-common-substring-dp-29/
38. Merge Overlapping Intervals - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/merging-intervals/
39. Merge Intervals - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/merge-intervals/
40. Merge Overlapping Sub-intervals - Tutorial - takeUforward, accessed on September 12, 2025, https://takeuforward.org/data-structure/merge-overlapping-sub-intervals/
41. Interval List Intersections - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/interval-list-intersections/
42. Intersection of intervals given by two lists - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/find-intersection-of-intervals-given-by-two-lists/
43. The easiest way to find intersection of two intervals - Computational Science Stack Exchange, accessed on September 12, 2025, https://scicomp.stackexchange.com/questions/26258/the-easiest-way-to-find-intersection-of-two-intervals
44. Linked List Data Structure - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/linked-list-data-structure/
45. Floyd's Cycle Finding Algorithm - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/floyds-cycle-finding-algorithm/
46. Detect Cycle in Linked List - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/detect-loop-in-a-linked-list/
47. Floyd's Cycle Detection Algorithm | Tortoise and Hare Problem | The Startup - Medium, accessed on September 12, 2025, https://medium.com/swlh/floyds-cycle-detection-algorithm-32881d8eaee1
48. How does Floyd's cycle-finding algorithm work? How does moving the tortoise to the beginning of the linked list, while keeping the hare at the meeting place,

followed by moving both one step at a time, make them meet at starting point of the cycle? - Quora, accessed on September 12, 2025, https://www.quora.com/How-does-Floyds-cycle-finding-algorithm-work-How-does-moving-the-tortoise-to-the-beginning-of-the-linked-list-while-keeping-the-hare-at-the-meeting-place-followed-by-moving-both-one-step-at-a-time-make-them-meet-at-starting-point-of-the-cycle
49. Detect Loop or Cycle in Linked List - GeeksforGeeks | Videos, accessed on September 12, 2025, https://www.geeksforgeeks.org/videos/detect-loop-or-cycle-in-linked-list/
50. Reverse a Linked List - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/reverse-a-linked-list/
51. Reverse Linked List - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/reverse-linked-list/
52. Reversing a Linked List: Easy as 1, 2, 3 | by Sergey Piterman | Outco - Medium, accessed on September 12, 2025, https://medium.com/outco/reversing-a-linked-list-easy-as-1-2-3-560fbffe2088
53. Two-Pointer Technique in a Linked List - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/two-pointer-technique-in-a-linked-list/
54. Middle of the Linked List - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/middle-of-the-linked-list/
55. Middle of a Linked List — GeeksforGeeks Problem | by Tushar Jain - Medium, accessed on September 12, 2025, https://medium.com/@tusharjain5/middle-of-a-linked-list-geeksforgeeks-problem-solution-6b9dfc77cb04
56. Middle Node in a Linked List - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/write-a-c-function-to-print-the-middle-of-the-linked-list/
57. These are my two approaches to find middle of a linked list. : r/leetcode - Reddit, accessed on September 12, 2025, https://www.reddit.com/r/leetcode/comments/1b8t3pf/these_are_my_two_approaches_to_find_middle_of_a/
58. LeetCode was HARD until I learned these 14 patterns! | Coding Templates and Animations, accessed on September 12, 2025, https://www.youtube.com/watch?v=PVjKqhi4qpw