Roadmap to Codeforces Expert and LeetCode Guardian

Achieving **Expert** on Codeforces and **Guardian** on LeetCode is an ambitious but achievable goal with consistent effort. Here is a structured plan tailored for C++ coders with 2–3 hours daily for preparation.

1. Understanding the Goal

Platform	Target Rank	Approx. Rating/Standing
Codeforces	Expert	1600+ rating
LeetCode	Guardian	Top 2-5% contest rank (varies, ~2200+)

2. General Guidelines

- **Focus**: Practice both platforms, but prioritize one if immediate results matter.
- **Language**: Continue with C++, but keep up with STL and templates.
- **Topics**: Master all core DSA topics, with special attention to algorithms required in contests.
- **Analysis**: Always upsolve and analyze mistakes after every contest.

3. Monthly Preparation Breakdown

Month 1–2: Strengthen Foundations

- **Codeforces**: Solve A/B and some C problems from recent Div.2 contests. Aim for accuracy and speed.
- **LeetCode**: Focus on Classic DSA patterns; finish Easy and Medium questions in Arrays, Strings, Hashing, Binary Search, Recursion, Linked Lists, Trees, and Two Pointers.
- **Contests**: Participate weekly on both platforms.
- Spend 30 min daily reviewing editorial solutions.

Month 3–4: Intermediate Challenges

- **Codeforces**: Progress to consistently solving C and attempt D problems.
- **LeetCode**: Move to Hard-level questions for practiced topics, and start Timed Contests.
- **Practice**:
  - 5–8 problems/day mixing both sites.
  - Upsolve from past contests immediately after participating.

- **Study Advanced C++**: Lambdas, custom comparators, fast I/O, debugging tricks.

Month 5–7: Advanced Practice & Strategy

- **Specialize**: Identify weak points (e.g., DP, Graphs, Math, Bitmasks) and drill those on both Codeforces and LeetCode.
- **Codeforces**: Aim to solve ABCD in Div.2 consistently, upsolve E if possible.
- **LeetCode**:
  - Grind through Locked and Explore problems.
  - Target “Contest” problems and participate in Biweekly/Weekly rounds.
- **Systematic Review**:
  - Revisit failed problems monthly.
  - Read grandmaster/Guardian post-contest writeups if available.

Month 8+: Strategy for Climbing Rank

- **Codeforces**:
  - Focus on optimization and speed, not just correctness.
  - Consider virtual participation in old Div.2 for simulation.
- **LeetCode**:
  - Focus on reducing penalty, increasing speed in contests (fast submissions are crucial for rank).
  - Learn to identify “trap” questions and avoid overthinking.
- **Mock Contests**: Simulate real contests under timed conditions at least weekly.
- **Peer Review**: Discuss tough problems in community forums.

4. Sample Weekly Plan

Day	Task
Mon–Tue	LeetCode DSA topic drills (2h), editorials review (1h)
Wed	Codeforces practice (2h), upsolving past contests (1h)
Thu–Fri	Timed LeetCode contest (1h), revisit missed problems (2h)
Sat	Codeforces weekly contest/live (2h), post-contest analysis (1h)
Sun	Community writeups, theory review, rest or bonus contest

5. Resources

- **Codeforces Problem Rating Filter**: Practice by problem rating to gradually step up.
- **LeetCode Explore**: Use guided study plans for each topic.
- **Standard Handbooks**:
    - "Competitive Programmer's Handbook" by Antti Laaksonen
    - Codeforces Gym for non-standard problems.

6. Tips for Long-Term Success

- **Track Progress**: Maintain a spreadsheet of solved problems and topics.
- **Master Templates**: Write code snippets for common DSA tasks.
- **Community**: Join Codeforces/LeetCode Discord and forums for tips/motivation.
- **Rest**: Avoid burnout—take breaks, especially after contests.

With focused daily practice and persistent analysis of mistakes, reaching **Expert** on Codeforces and **Guardian** on LeetCode is within your grasp. Good luck!

First Month In-Depth Topics and Patterns for Competitive Programming

Based on your goal to become **Expert** on Codeforces and **Guardian** on LeetCode with daily 2-3 hour study sessions, here's a comprehensive breakdown of the essential topics and patterns to master in your first month.

Core Foundation Topics (Month 1)

1. Programming Language Mastery (C++)

STL (Standard Template Library) Components[1] [2]

- **Containers**: `vector`, `string`, `array`, `set`, `map`, `unordered_set`, `unordered_map`
- **Algorithms**: `sort()`, `reverse()`, `find()`, `lower_bound()`, `upper_bound()`
- **Iterators**: Forward, bidirectional, and random access iterators
- **Utility Functions**: `min()`, `max()`, `swap()`, `__gcd()`, `abs()`

Essential C++ Features[3]

- Fast input/output using `ios_base::sync_with_stdio(false)`
- Lambda functions for custom comparators
- Auto keyword for type deduction
- Range-based for loops

2. Mathematical Foundations

Number Theory Basics[3]

- **Modular Arithmetic**: Understanding `(a + b) % m`, `(a * b) % m`, modular inverse
- **Prime Numbers**: Sieve of Eratosthenes for finding primes up to N
- **GCD and LCM**: Euclidean algorithm implementation
- **Divisors and Multiples**: Finding all divisors efficiently
- **Binary Exponentiation**: Computing `a^b % m` efficiently

Bitwise Operations[3]

- Basic operators: AND (`&`), OR (`|`), XOR (`^`), NOT (`~`)
- Left shift (`<<`) and right shift (`>>`)
- Checking if a number is power of 2
- Setting, clearing, and toggling bits
- Counting set bits

3. Implementation and Ad-hoc Problems

Problem Categories[4] [5]

- **Simulation**: Step-by-step problem execution
- **String manipulation**: Basic operations, character counting
- **Mathematical formulas**: Direct formula applications
- **Pattern recognition**: Finding mathematical or logical patterns
- **Constructive algorithms**: Building solutions step by step

4. Basic Data Structures

Arrays and Vectors[6]

- Declaration, initialization, and basic operations
- 1D and 2D array manipulation
- Dynamic arrays using `vector`
- Common operations: insertion, deletion, searching

Strings[7]

- String creation and manipulation
- Character array vs string class
- String comparison and sorting[8] [9]
- Basic string algorithms: palindrome checking, character frequency

Basic Sorting[10]

- Understanding built-in `sort()` function
- Custom comparators for different sorting criteria
- Stable vs unstable sorting
- Sorting arrays of structures/pairs

Essential LeetCode Patterns (Month 1)

1. Two Pointers Pattern[11]

When to Use: Problems with sorted arrays, finding pairs, or opposite-end processing

Key Problems:

- Two Sum (when array is sorted)
- Valid Palindrome
- Container with Most Water

Template:

```
int left = 0, right = n - 1;
while (left < right) {
    // Process elements at left and right
    if (condition) left++;
    else right--;
}
```

2. Sliding Window Pattern[11]

When to Use: Contiguous subarray/substring problems with specific conditions

Key Problems:

- Maximum Sum Subarray of Size K
- Longest Substring Without Repeating Characters

Template:

```
int left = 0, right = 0;
while (right < n) {
    // Expand window
    // If condition violated, shrink from left
    right++;
}
```

3. Prefix Sum Pattern[11]

When to Use: Range sum queries, subarray sum problems

Key Problems:

- Range Sum Query
- Subarray Sum Equals K

Implementation:

```
vector<int> prefixSum(n + 1, 0);
for (int i = 0; i < n; i++) {
    prefixSum[i + 1] = prefixSum[i] + arr[i];
}
```

4. Frequency Counter Pattern[12]

When to Use: Character/element counting, anagram problems

Key Problems:

- Valid Anagram
- First Unique Character
- Group Anagrams

Template:

```
unordered_map<char, int> freq;
for (char c : s) {
    freq[c]++;
}
```

Practice Strategy for Month 1

Codeforces Practice[1] [4] [5]

Rating Range: 800-1200 problems
Topics to Focus On[4]:

1. **Mathematics**: 40% of beginner problems
2. **Greedy**: 35% of problems
3. **Implementation**: 25% of problems
4. **Constructive Algorithms**: 20% of problems
5. **Brute Force**: 15% of problems

Daily Target: 3-5 problems mixing these topics
Approach: Solve problems topic-wise initially, then mix different topics

LeetCode Practice[2]

Difficulty Distribution:

- **Easy**: 60% (focus on pattern recognition)
- **Medium**: 35% (gradually increase)
- **Hard**: 5% (only if time permits)

Recommended Lists[12]:

- LeetCode 75 (essential patterns)
- Top 100 Liked Problems
- Explore Cards for Arrays & Strings[13]

Weekly Schedule

Day	Focus Area	Time Distribution
Mon-Tue	Codeforces Math + Implementation	2h Codeforces, 1h LeetCode patterns
Wed-Thu	LeetCode patterns practice	1.5h LeetCode, 1.5h theory review
Fri	Mixed practice	1.5h each platform
Sat	Contest participation	Live contest + upsolving
Sun	Review + weak areas	Error analysis + concept reinforcement

Month 1 Milestones

Week 1-2: Foundation Building

- Master STL basics and fast I/O
- Solve 20+ implementation problems on Codeforces (800-1000 rating)
- Complete basic array/string problems on LeetCode

Week 3-4: Pattern Recognition

- Implement all 4 basic patterns with 5-7 problems each
- Solve first contest problems (A, B) consistently
- Begin recognizing problem types quickly

Study Resources

Theory[14]:

- Competitive Programmer's Handbook (CSES) - Chapters 1-3
- CP-Algorithms website for implementation details[15]

Practice Platforms:

- Codeforces Problem Set (filtered by rating and tags)
- LeetCode Explore section[13]
- CSES Problem Set for additional practice

Key Tips for Success[16] [17]:

1. **Time Management**: Give 30-60 minutes per problem before checking solutions[18]
2. **Implementation Focus**: Priority on clean, bug-free code over complex algorithms[1]
3. **Pattern Recognition**: Keep a notebook of patterns and their templates[19]
4. **Contest Participation**: Join Codeforces Div 3/4 and LeetCode weekly contests
5. **Upsolving**: Always solve problems you couldn't during contests

Remember, the first month is about building a solid foundation rather than learning advanced topics like Dynamic Programming or Graph algorithms[3]. Focus on mastering these fundamentals, and you'll be well-prepared to tackle more complex topics in subsequent months.

⁂

First Month In-Depth Study Content - Week by Week Plan

Week 1: C++ Fundamentals & Basic Implementation

Day 1-2: C++ STL Foundation

Theory to Master:

- **Fast Input/Output Setup**[20] [21]

  ```
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  cout.tie(NULL);
  ```

- **Essential STL Containers**[20] [21]:
  - `vector<int> v` - Dynamic arrays with push_back(), pop_back(), size()
  - `string s` - Character manipulation with substr(), find(), length()
  - `pair<int,int> p` - Storing two values together
  - `set<int> st` - Unique elements with insert(), find(), erase()
  - `map<int,int> mp` - Key-value pairs with mp[key] = value

Practice Problems (Codeforces 800-1000 rating)[22] [23]:

- Implementation problems focusing on arrays and basic loops

- String manipulation: character counting, basic transformations
- Simple mathematical calculations with modular arithmetic

Daily Target: 2-3 implementation problems, 1 hour STL practice

Day 3-4: Mathematical Foundations

Core Number Theory Concepts[24] [25] [26]:

- **GCD and LCM Algorithms**:
  ```
  int gcd(int a, int b) {
      return b == 0 ? a : gcd(b, a % b);
  }
  int lcm(int a, int b) {
      return (a / gcd(a, b)) * b;
  }
  ```
- **Prime Number Basics**:
  - Checking primality in O(√n)
  - Basic sieve concept for multiple queries
- **Modular Arithmetic**[25]:
  - `(a + b) % MOD`, `(a * b) % MOD`
  - Understanding why modular arithmetic prevents overflow

Practice Focus[27]:

- Divisor counting problems
- Basic prime checking
- Simple modular arithmetic applications

Daily Target: 2-3 math problems, implement GCD/LCM from scratch

Day 5-6: Basic Data Structures

Array and Vector Mastery[20]:

- 1D and 2D array manipulation
- Vector operations: resize(), clear(), reverse()
- Sorting with custom comparators:
  ```
  sort(v.begin(), v.end(), greater<int>());
  ```

String Operations[28]:

- Character frequency counting
- Basic palindrome detection

- String comparison and lexicographic ordering

Practice Problems[22]:

- Array rearrangement problems
- Character frequency-based problems
- Simple string matching without advanced algorithms

Daily Target: 3-4 problems mixing arrays and strings

Day 7: Week 1 Assessment & Contest Simulation

Virtual Contest Practice:

- Attempt 3-4 problems from a recent Codeforces Div 3 or Educational round
- Time limit: 2 hours
- Focus on A, B problems with occasional C attempts

Week 1 Milestone Check:

- Can solve Codeforces Div 3 A problems consistently
- Comfortable with basic STL containers
- Implemented GCD/LCM algorithms independently

Week 2: Pattern Recognition & Basic Algorithms

Day 8-9: Two Pointers Technique

Pattern Understanding[29] [30] [31]:

- **Opposite Direction Pointers**:

  ```
  int left = 0, right = n - 1;
  while (left < right) {
      int sum = arr[left] + arr[right];
      if (sum == target) return true;
      else if (sum < target) left++;
      else right--;
  }
  ```

Common Applications[30]:

- Two Sum in sorted arrays
- Palindrome checking with pointers
- Container with most water type problems

LeetCode Practice[32]:

- Valid Palindrome
- Two Sum II (sorted array)

- Container With Most Water

Daily Target: Master the template, solve 4-5 two-pointer problems

Day 10-11: Sliding Window Fundamentals

Fixed Window Technique[33] [34]:

```
int maxSum = 0, currentSum = 0;
// Calculate first window
for (int i = 0; i < k; i++) {
    currentSum += arr[i];
}
maxSum = currentSum;

// Slide the window
for (int i = k; i < n; i++) {
    currentSum += arr[i] - arr[i - k];
    maxSum = max(maxSum, currentSum);
}
```

Variable Window Concept[33]:

- Expand window when condition not met
- Shrink window when condition satisfied

Practice Problems:

- Maximum sum of subarray of size K
- Longest substring without repeating characters (basic version)
- Minimum window containing pattern

Daily Target: 3-4 sliding window problems, focus on template recognition

Day 12-13: Prefix Sum & Frequency Arrays

Prefix Sum Implementation[33] [35]:

```
vector<int> prefixSum(n + 1, 0);
for (int i = 0; i < n; i++) {
    prefixSum[i + 1] = prefixSum[i] + arr[i];
}
// Range sum from l to r: prefixSum[r + 1] - prefixSum[l]
```

Character Frequency Patterns[36] [37]:

- Using unordered_map for character counting
- Palindrome formation conditions
- Anagram detection techniques

Practice Applications:

- Range sum queries
- Subarray sum equals K (basic version)
- Character rearrangement problems

Daily Target: 3-4 prefix sum problems, 2-3 frequency-based problems

Day 14: Week 2 Review & Mixed Practice

Pattern Integration:

- Problems requiring combination of two pointers and prefix sum
- String problems mixing frequency counting and sliding window
- Array problems combining multiple learned techniques

Contest Simulation:

- Attempt Codeforces Div 3 problems A, B, C
- Focus on recognizing which pattern applies to each problem

Week 2 Milestone Check:

- Can identify two pointer opportunities in problem statements
- Comfortable implementing sliding window for fixed and variable sizes
- Successfully applied prefix sum to range query problems

Week 3: Greedy Algorithms & Problem Solving

Day 15-16: Greedy Algorithm Fundamentals

Core Greedy Principles[38] [39]:

- **Local Optimal Choices**: Always choose the best available option at each step
- **No Backtracking**: Once a choice is made, it's final
- **Proof Techniques**: Exchange argument and staying ahead

Classic Greedy Problems[38]:

- **Activity Selection**: Choose activities with earliest finish time
- **Coin Change** (when greedy works): Use largest denomination first
- **String Formation**: Take maximum valid characters greedily

Template Recognition[38]:

```
// Greedy choice: always pick maximum/minimum available
sort(arr.begin(), arr.end());
int result = 0;
for (int i = 0; i < n; i++) {
    if (canTake(arr[i])) {
        result += arr[i];
    }
}
```

Daily Target: 3-4 greedy problems, understand when greedy fails

Day 17-18: Implementation & Constructive Problems

Implementation Problem Strategies[40]:

- **Careful Reading**: Understand all constraints and edge cases
- **Step-by-step Simulation**: Follow the problem statement exactly
- **Edge Case Handling**: Empty inputs, single elements, maximum constraints

Constructive Algorithm Approach[41]:

- Build solution incrementally
- Ensure each step maintains problem constraints
- Often requires mathematical insight or pattern recognition

Problem Categories to Practice[40]:

- Matrix manipulation and traversal
- Simulation of processes or games
- Construction of sequences with given properties

Daily Target: 4-5 implementation problems, focus on bug-free coding

Day 19-20: String Manipulation & Pattern Problems

Advanced String Techniques[28]:

- **Character Frequency Analysis**: Using arrays vs maps
- **Palindrome Properties**[36] [37]:
    - At most one character can have odd frequency
    - Symmetric structure requirements
- **Anagram Detection**: Sorted strings or frequency matching

String Problem Categories:

- Rearrangement problems (palindrome formation)
- Substring matching without advanced algorithms
- Character replacement and transformation

Implementation Focus:

```
// Efficient character frequency for lowercase letters
int freq[^3_26] = {0};
for (char c : s) {
    freq[c - 'a']++;
}
```

Daily Target: 3-4 string problems, emphasize clean implementation

Day 21: Week 3 Assessment & Contest Practice

Comprehensive Problem Solving:

- Mix of greedy, implementation, and string problems
- Time-pressured solving (90 minutes for 3-4 problems)
- Focus on solution verification and testing

Week 3 Milestone Check:

- Can recognize greedy opportunities and implement solutions
- Comfortable with string manipulation and frequency analysis
- Successfully handles implementation problems with multiple steps

Week 4: Integration & Contest Preparation

Day 22-23: Binary Search Fundamentals

Binary Search Template[42] [43]:

```
int binarySearch(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

Binary Search on Answer Pattern:

- When you can check if answer X is possible
- Search space has monotonic property
- Common in optimization problems

Practice Applications:

- Search in sorted arrays
- Finding square roots
- Basic binary search on answer problems

Daily Target: 3-4 binary search problems, master the template

Day 24-25: Basic Graph Concepts

Graph Representation[42]:

```
vector<vector<int>> adj(n); // Adjacency list
// Add edge from u to v
adj[u].push_back(v);
adj[v].push_back(u); // For undirected graphs
```

Fundamental Traversals:

- **DFS (Depth-First Search)**:

```
void dfs(int node, vector<bool>& visited) {
    visited[node] = true;
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, visited);
        }
    }
}
```

Basic Graph Problems:

- Connected components counting
- Path existence between nodes
- Simple tree traversal

Daily Target: 2-3 basic graph problems, understand representation

Day 26-27: Dynamic Programming Introduction

DP Fundamentals[42] [43]:

- **Optimal Substructure**: Solution can be built from optimal solutions to subproblems
- **Overlapping Subproblems**: Same subproblems are solved multiple times

Classical DP Problems:

- **Fibonacci with Memoization**:

```
int fib(int n, vector<int>& memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    return memo[n] = fib(n-1, memo) + fib(n-2, memo);
}
```

- **Climbing Stairs**: Ways to reach the top
- **Minimum Path Sum**: Simple 2D grid problems

Daily Target: 2-3 basic DP problems, understand memoization

Day 28: Final Week Assessment

Comprehensive Contest Simulation:

- Complete Codeforces Div 2 or Div 3 contest
- Time limit: 2.5 hours
- Attempt problems A, B, C, and explore D if time permits

Problem Mix Assessment:

- Implementation + greedy combination
- String problems with multiple constraints
- Graph traversal with additional conditions

Month 1 Final Milestone Check:

- Consistently solves Codeforces Div 2 A problems
- Successfully attempts Div 2 B problems
- Has basic understanding of all fundamental patterns
- Ready to move to more advanced topics in Month 2

Study Resources Summary

Primary Practice Platforms[44] [45] [46]:

- **Codeforces**: problemset filtered by rating 800-1400
- **LeetCode**: Focus on Easy and Medium problems from fundamental patterns
- **CSES Problem Set**: Introductory problems section

Theory Resources[42] [47] [48]:

- Competitive Programmer's Handbook (first 3 chapters)
- CP-Algorithms website for implementation details
- GeeksforGeeks for concept explanations

Daily Schedule Recommendation:

- **Morning (1 hour)**: Theory study and template implementation
- **Evening (2 hours)**: Problem solving and practice
- **Weekends**: Virtual contests and comprehensive review

This structured month provides comprehensive coverage of all fundamental competitive programming concepts necessary before advancing to more complex topics like advanced DP, complex graph algorithms, and specialized data structures[42] [43] [40].

⁂

In-Depth Theory for Weeks 1 & 2: Competitive Programming Foundations

Week 1: C++ Fundamentals & Basic Implementation

C++ STL Essentials

Fast Input/Output

- **Why:** Standard input/output can be slow for large IO-heavy problems. Disabling synchronization between C and C++ IO streams and untethering cin/cout from stdio significantly speeds up processing.
- **How:**

```
ios_base::sync_with_stdio(false);
cin.tie(NULL);
cout.tie(NULL);
```

- **Effect:** Up to 10x faster IO, especially crucial in tight time-limit contests.

STL Containers

- **Array vs Vector:**
  - *Array*: Fixed size, faster, stack-allocated.
  - *Vector*: Dynamic size, heap-allocated, supports push_back/pop_back.
- **String:** Mutable container for text, with built-in operations (concat, substr, find).
- **Pair:** Store two related values as a single object.
- **Set:** Stores unique elements in sorted order, allows fast insertions/searches.
- **Map:** Key-value storage with fast insert/find; unordered_map is faster on average due to hashing, but keys may not be sorted.

Useful STL Algorithms

- `sort()`, `reverse()`, `lower_bound()`, `upper_bound()` are crucial for many sorting and search tasks.
  - `sort(v.begin(), v.end())` sorts vector v in ascending order.
  - `lower_bound` finds the first element not less than a value in a sorted sequence.

Essential C++ Language Features

- **Auto Keyword:** Lets the compiler deduce types, making code concise:

  ```
  for (auto x : v) { /* ... */ }
  ```

- **Range-based for loops:** Cleaner iteration over containers.

Mathematical Foundations

GCD & LCM

- **GCD (Greatest Common Divisor):** Largest integer that divides two numbers.
  - Euclid's algorithm: $ gcd(a, b) = gcd(b, a%b) $
- **LCM (Least Common Multiple):** Smallest number divisible by two numbers.
  - $ lcm(a, b) = \frac{a \times b}{gcd(a, b)} $

Prime Numbers & Sieve

- **Prime Number:** Number greater than 1 with exactly two divisors: 1 and itself.
- **Primality Test (Basic):** Check divisibility up to $ \sqrt{n} $.
- **Sieve of Eratosthenes:** Efficiently lists all primes up to N in $ O(N \log \log N) $.

Modular Arithmetic

- Used to prevent overflow and for problems involving "mod" constraints.
  - $ (a + b) % m $, $ (a \times b) % m $, etc.
  - Fermat's little theorem is sometimes used in modular inverses (generally for advanced topics).

Bitwise Operations

- **Operators:** AND (&), OR (|), XOR (^), NOT (~), left shift (<<), right shift (>>).
- **Application:** Setting/toggling/checking bits, counting bits (bitmasking is a big deal in CP).

Implementation & Ad-hoc

- **Simulation:** Copy the described process step by step in code.
- **String Manipulation:** Substring search, character replacement, counting characters.
- **Pattern Recognition:** Identify recurring logic or numerical/formulaic patterns.

Data Structures

- **Arrays:** C-style, static size; easiest and fastest for fixed-length data.
- **Vectors:** Dynamic size, more flexible; supports all array operations but can resize.
- **Usage:** Know how to declare, resize (vectors), and use both for data storage in problems.

Week 2: Pattern Recognition & Basic Algorithms

Two Pointers Technique

Theory

- **When to use:** Involves processing an array or list from both ends, typically for problems like finding pairs that sum to a value, palindrome checks, merging sorted arrays, etc.
- **Pattern:** Maintain two indices (left and right), move them towards/away from each other based on conditions.
- **Efficiency:** Reduces time complexity from $ O(N^2) $ (brute force) to $ O(N) $.

Key Ideas

- Look for "sorted array" and "pair-wise comparison" in problem description.
- Useful in substring search and window problems.

Sliding Window Pattern

Theory

- **When to use:** Efficiently finding subarrays/substrings with desired properties (sum, unique elements, etc.), instead of checking all possible subarrays.
- **Fixed Window:** Window of size k, slide right by adding new elements and removing the starting element.
- **Variable Window:** Window changes size based on problem constraints (e.g., unique characters).

Key Ideas

- Reduces nested loops to linear/near linear scans.
- Always maintain window invariant (e.g., the window sum/count/condition stays valid).

Prefix Sum

Theory

- **Use Case:** Range sum queries or problems where sum/frequency up to index N needs to be computed efficiently.
- **Prefix Array:** prefixSum[i] = sum of elements up to i.
- **Query Range:** sum from l to r is prefixSum[r+1] - prefixSum[l].
- **Build Time:** O(N), each query O(1).

Frequency Arrays & Maps

Theory

- **Pattern:** Tracking how often elements or characters appear (for anagrams, unique characters, etc.).
- **Frequency Array:** For a small set of values (like lowercase letters), use an array for count.
- **Hash Map:** For larger or arbitrary keys.

Applications

- Anagram/Histo-gram matching.
- Finding majority/minority elements.
- Checking palindrome possibility.

Greedy Algorithms (Toward Week 3)

- **Idea:** Make the best local choice at each step, expecting that this leads to the global optimum.
- **Prove Correctness:** Greedy is not always correct; correctness typically needs to be established (by exchange argument or proof).
- **Classic Examples:** Activity selection, coin change problem (when denominations allow greedy), minimum/maximum string/array transformations under constraints.

Pattern Summary Table

Pattern/Concept	Key Use Cases	Complexity Benefit
Two Pointers	Ordered arrays, pairwise matching	O(N)
Sliding Window	Substrings/subarrays with constraints	O(N)
Prefix Sum	Fast range queries	O(1) per query
Frequency Counting	Duplicates, anagrams, frequency queries	O(N)
Greedy Algorithms	Simple optimal property problems	Typically O(NlogN)

Studying the theory deeply and working through small problems for each concept in these two weeks will cement your foundation for intermediate and harder topics in competitive programming.

Complete In-Depth Study Content for First 2 Weeks

Week 1: C++ Fundamentals & Basic Implementation

Day 1-2: C++ STL Foundation

Fast Input/Output - Complete Theory

Why is I/O Speed Important in Competitive Programming?

In competitive programming, programs often need to process millions of inputs quickly. Standard C++ I/O operations (`cin`/`cout`) by default synchronize with C-style I/O (`scanf`/`printf`) to maintain compatibility, but this synchronization creates overhead[49]. Additionally, `cin` is tied to `cout`, meaning every input operation flushes the output buffer, which slows down execution.

The Solution - Fast I/O Setup:

```
#include <iostream>
using namespace std;

int main() {
    ios_base::sync_with_stdio(false);  // Disable sync with C I/O
    cin.tie(NULL);                     // Untie cin from cout
    cout.tie(NULL);                    // Untie cout (optional)

    // Your program code here
    return 0;
}
```

What Each Line Does:

- `ios_base::sync_with_stdio(false)`: Disables synchronization between C and C++ I/O streams, making `cin`/`cout` up to 10x faster[49] [50]
- `cin.tie(NULL)`: Prevents automatic flushing of `cout` before each `cin` operation[49]
- `cout.tie(NULL)`: Optional optimization to prevent automatic flushing

Important Warnings:

- Never mix `cin`/`cout` with `scanf`/`printf` after using these optimizations[51]
- Don't use in interactive problems where you need immediate output feedback[51]
- Use `'\n'` instead of `endl` as `endl` forces buffer flushing, negating speed benefits[49]

Performance Example:

```
// Slow version - may get TLE (Time Limit Exceeded)
#include <iostream>
using namespace std;

int main() {
    int n, k, count = 0;
    cin >> n >> k;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        if (x % k == 0) count++;
    }
    cout << count << endl;  // endl is slow!
    return 0;
}

// Fast version - passes time limits
#include <iostream>
using namespace std;

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n, k, count = 0;
    cin >> n >> k;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        if (x % k == 0) count++;
    }
    cout << count << '\n';  // '\n' is faster than endl
    return 0;
}
```

Essential STL Containers - Deep Dive

1. Vector - Dynamic Arrays

Vectors are the most commonly used container in competitive programming. They provide dynamic resizing and random access[52].

```
#include <vector>
using namespace std;

// Declaration and Initialization
vector<int> v1;                    // Empty vector
vector<int> v2(5);                 // Vector with 5 zeros
vector<int> v3(5, 10);            // Vector with 5 elements, all = 10
vector<int> v4 = {1, 2, 3, 4, 5}; // Initialize with values
```

```
// Essential Operations
v1.push_back(42);      // Add element at end - O(1) amortized
v1.pop_back();         // Remove last element - O(1)
v1.size();             // Get number of elements - O(1)
v1.empty();            // Check if empty - O(1)
v1.clear();            // Remove all elements - O(n)

// Accessing elements
v4[^5_0] = 100;           // Direct access (no bounds checking) - O(1)
v4.at(0) = 100;        // Safe access (with bounds checking) - O(1)
v4.front();            // First element - O(1)
v4.back();             // Last element - O(1)

// Useful for competitive programming
v4.resize(10);         // Change size to 10
v4.resize(15, 99);     // Resize to 15, fill new elements with 99
```

2. String - Text Processing

Strings in C++ are mutable sequences of characters with powerful built-in operations[53].

```
#include <string>
using namespace std;

// Declaration and Initialization
string s1;                      // Empty string
string s2("Hello");            // Initialize with C-style string
string s3 = "World";           // Initialize with string literal
string s4(5, 'A');            // "AAAAA" - 5 copies of 'A'

// Essential Operations
s1.length();         // Get length - O(1)
s1.size();          // Same as length() - O(1)
s1.empty();         // Check if empty - O(1)
s1.push_back('X');  // Add character at end - O(1)
s1.pop_back();      // Remove last character - O(1)

// String manipulation
s1 = s2 + s3;       // Concatenation - O(n)
s1.substr(2, 3);    // Substring from index 2, length 3 - O(k)
s1.find("ll");      // Find substring, returns position or string::npos - O(nm)

// Character access and modification
s2[^5_0] = 'h';        // Direct character access - O(1)
s2.at(1) = 'E';     // Safe character access - O(1)

// Useful for problems
reverse(s1.begin(), s1.end());  // Reverse string - O(n)
sort(s1.begin(), s1.end());     // Sort characters - O(n log n)
```

3. Pair - Storing Two Related Values

Pairs are fundamental for storing related data and are heavily used in competitive programming[53] [54].

```cpp
#include <utility>  // for pair
using namespace std;

// Declaration and Initialization
pair<int, int> p1;                    // Default: (0, 0)
pair<int, string> p2(42, "answer");   // Constructor
pair<int, string> p3 = {100, "test"}; // Brace initialization
pair<int, int> p4 = make_pair(3, 7);  // Using make_pair

// Accessing elements
p2.first = 50;      // Access first element
p2.second = "new";  // Access second element

// Comparison (lexicographic order)
pair<int, int> a = {1, 5};
pair<int, int> b = {1, 3};
// a > b is true (first elements equal, compare second)

// Useful in algorithms
vector<pair<int, string>> students = {
    {85, "Alice"},
    {92, "Bob"},
    {78, "Charlie"}
};
sort(students.begin(), students.end());  // Sorts by marks (first element)
```

4. Set - Unique Sorted Elements

Sets maintain unique elements in sorted order and provide fast insertion, deletion, and search[53].

```cpp
#include <set>
using namespace std;

set<int> s;

// Insertion - O(log n)
s.insert(10);
s.insert(5);
s.insert(15);
s.insert(10);  // Won't be inserted (duplicate)

// Search - O(log n)
if (s.find(10) != s.end()) {
    cout << "10 exists in set";
}
bool exists = s.count(5);  // Returns 1 if exists, 0 otherwise

// Deletion - O(log n)
s.erase(5);         // Remove specific element
s.erase(s.find(15)); // Remove using iterator

// Iteration (elements in sorted order)
for (int x : s) {
    cout << x << " ";  // Will print in ascending order
}

// Size and bounds
cout << s.size();           // Number of elements
auto it = s.lower_bound(8); // First element >= 8
auto it2 = s.upper_bound(12); // First element > 12
```

5. Map - Key-Value Storage

Maps store key-value pairs with fast access based on keys[53].

```
#include <map>
using namespace std;

map<string, int> age;

// Insertion and access - O(log n)
age["Alice"] = 25;
age["Bob"] = 30;
age.insert({"Charlie", 28});

// Access - O(log n)
cout << age["Alice"];        // Prints 25
// Note: age["David"] would insert David with value 0

// Safe access
if (age.find("David") != age.end()) {
    cout << age["David"];
}

// For competitive programming, often use unordered_map for O(1) average access
#include <unordered_map>
unordered_map<string, int> fast_age;  // O(1) average, O(n) worst case
fast_age["Alice"] = 25;
```

Essential STL Algorithms

1. Sorting

```
#include <algorithm>
vector<int> v = {3, 1, 4, 1, 5};

// Basic sorting - O(n log n)
sort(v.begin(), v.end());              // Ascending: {1, 1, 3, 4, 5}
sort(v.begin(), v.end(), greater<int>()); // Descending: {5, 4, 3, 1, 1}

// Custom comparator for pairs
vector<pair<int, string>> students = {{85, "Alice"}, {92, "Bob"}};
sort(students.begin(), students.end(), [](const pair<int, string>& a, const pair<int, str
    return a.second < b.second;  // Sort by name
});
```

2. Binary Search Functions

Binary search functions work only on sorted containers and provide O(log n) operations[55] [56] [57].

```
vector<int> v = {1, 3, 5, 7, 9, 11};

// binary_search - O(log n)
bool found = binary_search(v.begin(), v.end(), 5);  // true

// lower_bound - first position where element can be inserted
auto it1 = lower_bound(v.begin(), v.end(), 6);
// it1 points to 7 (first element >= 6)
int pos1 = it1 - v.begin();  // Position: 3

// upper_bound - first position after all equal elements
auto it2 = upper_bound(v.begin(), v.end(), 5);
// it2 points to 7 (first element > 5)
int pos2 = it2 - v.begin();  // Position: 3

// Practical usage: find range of equal elements
vector<int> nums = {1, 2, 2, 2, 3, 4};
auto first = lower_bound(nums.begin(), nums.end(), 2);
auto last = upper_bound(nums.begin(), nums.end(), 2);
int count = last - first;  // Count of 2's = 3
```

Mathematical Foundations

GCD and LCM - Complete Implementation

Theory Behind GCD:

The Greatest Common Divisor (GCD) of two numbers is the largest number that divides both without remainder. The Euclidean algorithm is based on the principle: GCD(a, b) = GCD(b, a mod b)[58] [59] [60].

Why does this work?
If d divides both a and b, then d also divides (a - b). Since a mod b = a - (a/b)*b, any common divisor of a and b is also a common divisor of b and (a mod b)[60] [61].

Complete Implementation:

```
#include <iostream>
using namespace std;

// Recursive GCD - most elegant
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

// Iterative GCD - avoids recursion overhead
```

```cpp
int gcd_iterative(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// Handle negative numbers
int gcd_safe(int a, int b) {
    a = abs(a);
    b = abs(b);
    if (b == 0) return a;
    return gcd_safe(b, a % b);
}

// LCM using GCD (avoid overflow)
long long lcm(int a, int b) {
    return ((long long)a / gcd(a, b)) * b;  // Divide first to prevent overflow
}

// Extended Euclidean Algorithm - finds x, y such that ax + by = gcd(a,b)
int extended_gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1; y = 0;
        return a;
    }
    int x1, y1;
    int g = extended_gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return g;
}

// Example usage and testing
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    // Test cases
    cout << gcd(48, 18) << "\n";    // Output: 6
    cout << lcm(12, 18) << "\n";    // Output: 36
    cout << gcd(17, 19) << "\n";    // Output: 1 (coprime numbers)

    // Extended GCD example
    int x, y;
    int g = extended_gcd(30, 20, x, y);
    cout << g << " = " << 30 << "*" << x << " + " << 20 << "*" << y << "\n";
    // Output: 10 = 30*1 + 20*(-1)

    return 0;
}
```

Modular Arithmetic - Deep Understanding

Why Modular Arithmetic in Competitive Programming?

Problems often ask for answers modulo 10^9 + 7 (1000000007) for several reasons[62] [63] [64]:

1. **Prevent Overflow**: Results can be extremely large (beyond long long capacity)
2. **Standardization**: Ensures same answers across different languages/implementations
3. **Mathematical Properties**: 10^9 + 7 is prime, enabling modular inverse operations

Core Properties:

```
const int MOD = 1000000007;

// Addition: (a + b) % m = ((a % m) + (b % m)) % m
int add_mod(int a, int b) {
    return ((a % MOD) + (b % MOD)) % MOD;
}

// Subtraction: (a - b) % m = ((a % m) - (b % m) + m) % m
int sub_mod(int a, int b) {
    return ((a % MOD) - (b % MOD) + MOD) % MOD;  // +MOD handles negative results
}

// Multiplication: (a * b) % m = ((a % m) * (b % m)) % m
long long mul_mod(int a, int b) {
    return ((long long)(a % MOD) * (b % MOD)) % MOD;
}

// Power: a^b % m using binary exponentiation
long long power_mod(long long base, long long exp) {
    long long result = 1;
    base %= MOD;
    while (exp > 0) {
        if (exp & 1) {  // If exp is odd
            result = (result * base) % MOD;
        }
        base = (base * base) % MOD;
        exp >>= 1;  // exp = exp / 2
    }
    return result;
}

// Modular inverse: a^(-1) % m = a^(m-2) % m (when m is prime)
long long mod_inverse(int a) {
    return power_mod(a, MOD - 2);
}

// Division: (a / b) % m = (a * b^(-1)) % m
long long div_mod(int a, int b) {
    return mul_mod(a, mod_inverse(b));
}
```

Practical Examples:

```
#include <iostream>
using namespace std;

const int MOD = 1000000007;

// Calculate nCr % MOD using precomputed factorials
vector<long long> fact, inv_fact;

void precompute_factorials(int n) {
    fact.resize(n + 1);
    inv_fact.resize(n + 1);

    fact[^5_0] = 1;
    for (int i = 1; i <= n; i++) {
        fact[i] = (fact[i-1] * i) % MOD;
    }

    inv_fact[n] = power_mod(fact[n], MOD - 2);
    for (int i = n - 1; i >= 0; i--) {
        inv_fact[i] = (inv_fact[i + 1] * (i + 1)) % MOD;
    }
}

long long nCr(int n, int r) {
    if (r > n || r < 0) return 0;
    return (fact[n] * inv_fact[r] % MOD) * inv_fact[n - r] % MOD;
}
```

Prime Numbers and Sieve of Eratosthenes

Basic Primality Testing:

```
#include <cmath>

bool is_prime(int n) {
    if (n < 2) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;

    for (int i = 3; i <= sqrt(n); i += 2) {
        if (n % i == 0) return false;
    }
    return true;
}
```

Sieve of Eratosthenes - Complete Implementation:

The Sieve efficiently finds all primes up to n in O(n log log n) time[65] [66] [67].

Theory: Mark all multiples of each prime as composite. What remains unmarked are primes[68].

```
#include <vector>
#include <iostream>
using namespace std;

vector<bool> sieve(int n) {
    vector<bool> is_prime(n + 1, true);
    is_prime[^5_0] = is_prime[^5_1] = false;  // 0 and 1 are not prime

    for (int i = 2; i * i <= n; i++) {  // Only check up to sqrt(n)
        if (is_prime[i]) {
            // Mark all multiples of i starting from i^2
            for (int j = i * i; j <= n; j += i) {
                is_prime[j] = false;
            }
        }
    }
    return is_prime;
}

// Get list of all primes up to n
vector<int> get_primes(int n) {
    vector<bool> is_prime = sieve(n);
    vector<int> primes;

    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
        }
    }
    return primes;
}

// Optimized sieve for memory efficiency
vector<int> segmented_sieve(int n) {
    int limit = sqrt(n) + 1;
    vector<int> small_primes = get_primes(limit);
    vector<int> primes = small_primes;

    for (int low = limit; low <= n; low += limit) {
        vector<bool> mark(limit, true);
        int high = min(low + limit - 1, n);

        for (int prime : small_primes) {
            int start = max(prime * prime, (low + prime - 1) / prime * prime);
            for (int j = start; j <= high; j += prime) {
                mark[j - low] = false;
            }
        }

        for (int i = low; i <= high; i++) {
            if (mark[i - low] && i > 1) {
                primes.push_back(i);
            }
        }
    }
    return primes;
}

// Example usage
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n = 30;
    vector<bool> is_prime = sieve(n);

    cout << "Primes up to " << n << ": ";
    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) {
            cout << i << " ";
        }
    }
    cout << "\n";

    return 0;
}
```

Week 2: Pattern Recognition & Basic Algorithms

Two Pointers Technique - Comprehensive Guide

Theory and When to Use:

The two pointers technique uses two indices to traverse data structures efficiently, often reducing time complexity from O(n²) to O(n)[69] [70] [71] [72].

Types of Two Pointers:

1. **Converging Pointers**: Start from opposite ends, move toward each other
2. **Parallel Pointers**: Both start from same end, move in same direction
3. **Fixed Distance**: Maintain constant distance between pointers

Implementation Patterns:

```
#include <vector>
#include <algorithm>
using namespace std;

// Pattern 1: Converging Pointers (for sorted arrays)
vector<int> two_sum_sorted(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;

    while (left < right) {
        int sum = arr[left] + arr[right];
        if (sum == target) {
            return {left, right};
        } else if (sum < target) {
            left++;
        } else {
```

```cpp
            right--;
        }
    }
    return {-1, -1};  // Not found
}

// Pattern 2: Parallel Pointers (fast/slow)
vector<int> remove_duplicates(vector<int>& arr) {
    if (arr.empty()) return arr;

    int slow = 0;
    for (int fast = 1; fast < arr.size(); fast++) {
        if (arr[fast] != arr[slow]) {
            slow++;
            arr[slow] = arr[fast];
        }
    }
    arr.resize(slow + 1);
    return arr;
}

// Pattern 3: Palindrome Check
bool is_palindrome(string s) {
    int left = 0, right = s.length() - 1;

    while (left < right) {
        if (s[left] != s[right]) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}

// Pattern 4: Container With Most Water
int max_area(vector<int>& height) {
    int left = 0, right = height.size() - 1;
    int max_water = 0;

    while (left < right) {
        int width = right - left;
        int h = min(height[left], height[right]);
        max_water = max(max_water, width * h);

        // Move pointer at shorter line
        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }
    return max_water;
}

// Pattern 5: Three Sum Problem
```

```
vector<vector<int>> three_sum(vector<int>& nums) {
    vector<vector<int>> result;
    sort(nums.begin(), nums.end());

    for (int i = 0; i < nums.size() - 2; i++) {
        if (i > 0 && nums[i] == nums[i-1]) continue;  // Skip duplicates

        int left = i + 1, right = nums.size() - 1;
        while (left < right) {
            int sum = nums[i] + nums[left] + nums[right];
            if (sum == 0) {
                result.push_back({nums[i], nums[left], nums[right]});

                // Skip duplicates
                while (left < right && nums[left] == nums[left+1]) left++;
                while (left < right && nums[right] == nums[right-1]) right--;

                left++;
                right--;
            } else if (sum < 0) {
                left++;
            } else {
                right--;
            }
        }
    }
    return result;
}
```

Recognition Patterns:

- Array/string problems involving pairs or subarrays
- "Find two elements that..." problems
- Problems with sorted input
- Palindrome-related problems
- Problems asking for optimization in O(n) time

Sliding Window Technique - Complete Mastery

Theory:

Sliding window maintains a "window" of elements and slides it across the data structure to solve problems involving contiguous subarrays/substrings[73] [74] [75] [76].

Types:

1. **Fixed Size Window**: Window size remains constant
2. **Variable Size Window**: Window size changes based on conditions

Fixed Size Window Implementation:

```
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

// Maximum sum of subarray of size k
int max_sum_subarray(vector<int>& arr, int k) {
    if (arr.size() < k) return -1;

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

    int max_sum = window_sum;

    // Slide the window
    for (int i = k; i < arr.size(); i++) {
        window_sum += arr[i] - arr[i - k];  // Add new, remove old
        max_sum = max(max_sum, window_sum);
    }

    return max_sum;
}

// Average of all subarrays of size k
vector<double> averages_of_subarrays(vector<int>& arr, int k) {
    vector<double> result;
    double window_sum = 0;

    // Calculate first window
    for (int i = 0; i < k; i++) {
        window_sum += arr[i];
    }
    result.push_back(window_sum / k);

    // Slide window
    for (int i = k; i < arr.size(); i++) {
        window_sum += arr[i] - arr[i - k];
        result.push_back(window_sum / k);
    }

    return result;
}
```

Variable Size Window Implementation:

```
#include <unordered_map>
#include <string>

// Longest substring without repeating characters
int longest_unique_substring(string s) {
    unordered_map<char, int> char_index;
    int max_length = 0;
    int window_start = 0;

    for (int window_end = 0; window_end < s.length(); window_end++) {
        char right_char = s[window_end];

        // If character already seen, move window start
        if (char_index.find(right_char) != char_index.end()) {
            window_start = max(window_start, char_index[right_char] + 1);
        }

        char_index[right_char] = window_end;
        max_length = max(max_length, window_end - window_start + 1);
    }

    return max_length;
}

// Minimum window substring containing all characters of pattern
string min_window_substring(string s, string t) {
    if (s.length() < t.length()) return "";

    unordered_map<char, int> pattern_freq;
    for (char c : t) {
        pattern_freq[c]++;
    }

    int matched = 0, window_start = 0, min_len = INT_MAX, min_start = 0;
    unordered_map<char, int> window_freq;

    for (int window_end = 0; window_end < s.length(); window_end++) {
        char right_char = s[window_end];
        window_freq[right_char]++;

        if (pattern_freq.count(right_char) && window_freq[right_char] == pattern_freq[rig
            matched++;
        }

        // Try to shrink window
        while (matched == pattern_freq.size()) {
            if (window_end - window_start + 1 < min_len) {
                min_len = window_end - window_start + 1;
                min_start = window_start;
            }

            char left_char = s[window_start++];
            window_freq[left_char]--;
            if (pattern_freq.count(left_char) && window_freq[left_char] < pattern_freq[le
                matched--;
            }
        }
    }

    return min_len == INT_MAX ? "" : s.substr(min_start, min_len);
}
```

```
// Longest subarray with sum <= k
int longest_subarray_with_sum_k(vector<int>& arr, int k) {
    int max_length = 0, window_start = 0;
    long long window_sum = 0;

    for (int window_end = 0; window_end < arr.size(); window_end++) {
        window_sum += arr[window_end];

        // Shrink window while sum > k
        while (window_sum > k && window_start <= window_end) {
            window_sum -= arr[window_start++];
        }

        max_length = max(max_length, window_end - window_start + 1);
    }

    return max_length;
}
```

Prefix Sum - Master Implementation

Theory:

Prefix sum allows O(1) range sum queries after O(n) preprocessing. prefix[i] = sum of elements from index 0 to i[77] [78] [79].

Complete Implementation:

```
#include <vector>
using namespace std;

class PrefixSum {
private:
    vector<long long> prefix;

public:
    // Build prefix sum array - O(n)
    PrefixSum(vector<int>& arr) {
        int n = arr.size();
        prefix.resize(n + 1, 0);  // 1-indexed for easier calculation

        for (int i = 0; i < n; i++) {
            prefix[i + 1] = prefix[i] + arr[i];
        }
    }

    // Range sum query [left, right] inclusive - O(1)
    long long range_sum(int left, int right) {
        return prefix[right + 1] - prefix[left];
    }

    // Sum from start to index - O(1)
    long long prefix_sum(int index) {
        return prefix[index + 1];
    }
};

// 2D Prefix Sum for matrix range queries
class PrefixSum2D {
private:
    vector<vector<long long>> prefix;
    int rows, cols;

public:
    PrefixSum2D(vector<vector<int>>& matrix) {
        rows = matrix.size();
        cols = matrix[^5_0].size();
        prefix.assign(rows + 1, vector<long long>(cols + 1, 0));

        // Build 2D prefix sum
        for (int i = 1; i <= rows; i++) {
            for (int j = 1; j <= cols; j++) {
                prefix[i][j] = matrix[i-1][j-1] + prefix[i-1][j] +
                              prefix[i][j-1] - prefix[i-1][j-1];
            }
        }
    }

    // Sum of rectangle from (r1,c1) to (r2,c2) inclusive - O(1)
    long long range_sum(int r1, int c1, int r2, int c2) {
        return prefix[r2+1][c2+1] - prefix[r1][c2+1] -
               prefix[r2+1][c1] + prefix[r1][c1];
    }
};

// Applications and Examples
class PrefixSumApplications {
public:
    // Count subarrays with sum = k
    int subarrays_with_sum_k(vector<int>& arr, int k) {
        unordered_map<long long, int> prefix_count;
        prefix_count[^5_0] = 1;  // Empty prefix

        long long running_sum = 0;
        int count = 0;

        for (int num : arr) {
            running_sum += num;

            // Check if (running_sum - k) exists
            if (prefix_count.count(running_sum - k)) {
                count += prefix_count[running_sum - k];
            }

            prefix_count[running_sum]++;
        }

        return count;
    }
    // Maximum subarray sum (Kadane's algorithm with prefix)
    int max_subarray_sum(vector<int>& arr) {
        int max_sum = arr[^5_0], current_sum = arr[^5_0];

        for (int i = 1; i < arr.size(); i++) {
            current_sum = max(arr[i], current_sum + arr[i]);
            max_sum = max(max_sum, current_sum);
        }

        return max_sum;
    }

    // Equilibrium point (prefix sum from left equals suffix sum)
    int equilibrium_point(vector<int>& arr) {
        long long total_sum = 0;
        for (int x : arr) total_sum += x;

        long long left_sum = 0;
        for (int i = 0; i < arr.size(); i++) {
            if (left_sum == total_sum - left_sum - arr[i]) {
                return i;
            }
            left_sum += arr[i];
        }

        return -1;  // No equilibrium point
    }
};
```

This comprehensive coverage of the first two weeks provides complete theory, implementation details, and practical examples for all fundamental concepts. Students can learn entirely from this content without needing additional resources, as it includes detailed explanations, code implementations, and real-world applications of each technique.

⁂

1. https://www.youtube.com/watch?v=W2RS0wJQMtU
2. https://www.youtube.com/watch?v=G5_Q2_yRFsY
3. https://www.youtube.com/watch?v=U2E9PxRd680
4. https://www.youtube.com/watch?v=GrGWAFEidXk
5. https://www.geeksforgeeks.org/blogs/10-best-tips-to-get-started-with-codeforces/
6. https://www.datacamp.com/blog/dsa-roadmap
7. https://www.geeksforgeeks.org/competitive-programming/strings-for-competitive-programming/
8. https://www.geeksforgeeks.org/dsa/sort-array-strings-according-string-lengths/
9. https://www.youtube.com/watch?v=LZNxRSoEG2M
10. https://algocoding.wordpress.com/2013/03/22/sorting-arrays-in-java-for-competitive-programming/
11. https://blog.algomaster.io/p/15-leetcode-patterns
12. https://dev.to/idsulik/empowering-newbies-building-confidence-through-600-leetcode-solutions-a-guide-for-beginners-3960

13. https://leetcode.com/explore/interview/card/leetcodes-interview-crash-course-data-structures-and-algorithms/703/arraystrings/
14. https://cses.fi/book.pdf
15. https://cp-algorithms.com/index.html
16. https://hackernoon.com/7-essential-tips-for-competitive-programming-and-dsa
17. https://www.reddit.com/r/csMajors/comments/z4qjzx/a_guide_to_competitive_programming/
18. https://www.youtube.com/watch?v=bSdp2WeyuJY
19. https://www.reddit.com/r/leetcode/comments/1d31ksp/leetcode_patternstechniques_cheat_sheet/
20. https://noi.ph/training/weekly/week1.pdf?x71985
21. https://github.com/mansikagrawal/STL-NOTES
22. https://www.youtube.com/watch?v=kHzISrOamX4
23. https://www.youtube.com/watch?v=oX7xOUUGJbs
24. https://www.geeksforgeeks.org/engineering-mathematics/number-theory/
25. https://www.geeksforgeeks.org/competitive-programming/number-theory-competitive-programming/
26. https://www.youtube.com/watch?v=rW9yyqxZzms
27. https://www.youtube.com/watch?v=rk2X8I7nJtg
28. https://blog.heycoach.in/competitive-programming-with-strings/
29. https://www.youtube.com/watch?v=8XHE3nzjIOA
30. https://www.geeksforgeeks.org/dsa/two-pointers-technique/
31. https://usaco.guide/silver/two-pointers
32. https://www.piratekingdom.com/leetcode/study-guide
33. https://algo.monster/problems/subarray_sum
34. https://www.codechef.com/practice/two-pointers
35. https://www.geeksforgeeks.org/dsa/prefix-sum-array-implementation-applications-competitive-programming/
36. https://www.geeksforgeeks.org/competitive-programming/cses-solutions-palindrome-reorder/
37. https://algo.monster/liteproblems/266
38. https://www.youtube.com/watch?v=g9-sRYP9Lvg
39. https://www.youtube.com/watch?v=R9rdPtv7XoY
40. https://www.geeksforgeeks.org/how-to-prepare-for-competitive-programming/
41. https://www.reddit.com/r/codeforces/comments/1ckslg9/why_codeforces_last_rounds_are_all_about_greedy/
42. https://www.youtube.com/watch?v=U2E9PxRd680
43. https://www.youtube.com/watch?v=W2RS0wJQMtU
44. https://www.youtube.com/watch?v=bSdp2WeyuJY
45. https://www.youtube.com/watch?v=HgIdbQu3yQI
46. https://www.linkedin.com/posts/vedant-rawale-9992a924b_leetcode-achievement-leetcodebadge-activity-7212658537477378048-6xJP
47. https://github.com/ShahjalalShohag/Competitive-Programming-A-Complete-Guideline

48. https://duoblogger.github.io/assets/pdf/memonvyftw/guide-t-cp.pdf
49. https://www.geeksforgeeks.org/competitive-programming/fast-io-for-competitive-programming/
50. https://usaco.guide/general/fast-io
51. https://www.youtube.com/watch?v=aNF4DEluWnI
52. https://www.geeksforgeeks.org/cpp/how-to-create-vector-of-pairs-in-cpp/
53. https://www.geeksforgeeks.org/cpp/pair-in-cpp-stl/
54. https://www.tutorialspoint.com/cplusplus-program-to-implement-pairs-in-stl
55. https://www.w3schools.com/cpp/ref_algorithm_lower_bound.asp
56. https://www.geeksforgeeks.org/cpp/lower_bound-in-cpp/
57. https://www.geeksforgeeks.org/cpp/upper_bound-and-lower_bound-for-vector-in-cpp-stl/
58. https://www.shiksha.com/online-courses/articles/euclidean-algorithm/
59. https://en.wikipedia.org/wiki/Euclidean_algorithm
60. https://cp-algorithms.com/algebra/euclid-algorithm.html
61. https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclid ean-algorithm
62. https://www.geeksforgeeks.org/competitive-programming/modular-arithmetic-for-competitive-progra mming/
63. https://www.youtube.com/watch?v=-OPohCQqi_E
64. https://www.geeksforgeeks.org/engineering-mathematics/modular-arithmetic/
65. https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html
66. https://www.topcoder.com/thrive/articles/sieve-of-eratosthenes-algorithm
67. https://www.scaler.com/topics/sieve-of-eratosthenes-cpp/
68. https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
69. https://www.youtube.com/watch?v=QzZ7nmouLTI
70. https://www.designgurus.io/course-play/grokking-the-coding-interview/doc/introduction-to-two-pointe rs-pattern
71. https://emre.me/coding-patterns/two-pointers/
72. https://www.geeksforgeeks.org/dsa/two-pointers-technique/
73. https://www.sharpener.tech/blog/sliding-window-technique-in-algorithms/
74. https://www.youtube.com/watch?v=dOonV4byDEg
75. https://www.freecodecamp.org/news/sliding-window-technique/
76. https://builtin.com/data-science/sliding-window-algorithm
77. https://www.geeksforgeeks.org/dsa/prefix-sum-array-implementation-applications-competitive-progr amming/
78. https://www.scaler.in/prefix-sum/
79. https://algo.monster/problems/subarray_sum