Algorithm Master: C# Problem-Solving Templates

Complete Guide to Mastering Algorithmic Problem Solving in C#

This comprehensive guide provides production-ready templates, detailed explanations, and practical examples for solving the most common algorithmic problems. Each template includes Big O analysis, usage patterns, and real-world applications.

Target Framework: .NET 8.0+
Language Version: C# 12
Last Updated: December 11, 2025

Table of Contents

1. Arrays & Strings

Arrays and strings are fundamental data structures that form the building blocks of most algorithms. Mastering these patterns is essential for technical interviews and real-world problem solving.

Key Patterns:
• Two Pointers (Opposite & Same Direction)
• Sliding Window (Fixed & Variable Size)
• Prefix Sum & Difference Array
• In-place Operations
• String Manipulation Techniques

Common Problems:
• Two Sum, Three Sum
• Container With Most Water
• Longest Substring Without Repeating Characters
• Maximum Subarray (Kadane's Algorithm)

2. Hash Maps & Dictionaries

Hash maps provide O(1) average-case lookup time, making them invaluable for problems requiring fast access to data. They're particularly useful for frequency counting, two-sum problems, and caching.

Key Patterns:
• Frequency Counting
• Two Sum Variations
• Subarray Sum Patterns
• LRU Cache Implementation
• Union-Find Data Structure

Common Problems:
• Two Sum, Three Sum
• Group Anagrams
• Longest Substring with K Distinct Characters
• Subarray Sum Equals K

3. Two Pointers Technique

The two pointers technique is an elegant way to solve problems involving sorted arrays or linked lists. It reduces time complexity from $O(n^2)$ to $O(n)$ in many cases.

Key Patterns:
• Opposite Pointers (Two ends)
• Same Direction Pointers (Fast/Slow)
• Three Pointers (Dutch National Flag)
• Merge Operations

Common Problems:
• Two Sum in Sorted Array
• Remove Duplicates
• Find Cycle in Linked List

• Merge Sorted Arrays

4. Sliding Window

Sliding window is a powerful technique for problems involving contiguous subarrays or substrings. It maintains a window of elements and slides it through the data.

Key Patterns:
• Fixed Size Window
• Variable Size Window
• Two Pointers Window
• Deque-based Window

Common Problems:
• Maximum Sum of K Consecutive Elements
• Longest Substring Without Repeating Characters
• Minimum Window Substring
• Sliding Window Maximum

5. Binary Search

Binary search is a fundamental algorithm that reduces search space by half in each iteration. It's not limited to sorted arrays - many problems can be solved using binary search on answer space.

Key Patterns:
• Classic Binary Search
• Search in Rotated Sorted Array
• Find Peak Element
• Binary Search on Answer

Common Problems:
• Find Minimum in Rotated Sorted Array
• Search 2D Matrix
• Find Kth Smallest Element
• Capacity to Ship Packages

Array Template Usage

Code Example:

```
// Example: Using ArrayTemplate for Two Sum problem int[] nums = {2, 7, 11, 15};
int target = 9; int[] result = ArrayTemplate.TwoPointersBothEnds(nums, target);
// Returns indices [0, 1] because nums[0] + nums[1] = 2 + 7 = 9 // Example:
Prefix Sum for range queries var prefixSum = new ArrayTemplate.PrefixSum(new
int[] {1, 2, 3, 4, 5}); int sum = prefixSum.SumRange(1, 3); // Returns sum of
elements at indices 1, 2, 3
```

Explanation:

The array template provides optimized solutions for common array problems. Two pointers pattern reduces $O(n^2)$ to $O(n)$, while prefix sum enables $O(1)$ range queries after $O(n)$ preprocessing.

HashMap Template Usage

Code Example:

```
// Example: Two Sum using HashMap int[] nums = {2, 7, 11, 15}; int target = 9; int[] result = HashMapTemplate.TwoSum(nums, target); // Returns [0, 1] in O(n) time with O(n) space // Example: LRU Cache var cache = new HashMapTemplate.LRUCache(3); cache.Put(1, 1); cache.Put(2, 2); cache.Put(3, 3); int val = cache.Get(1); // Returns 1 cache.Put(4, 4); // Evicts key 2 (least recently used)
```

Explanation:

Hash maps provide O(1) average-case lookup, making them ideal for problems requiring fast data access. The LRU cache template demonstrates practical application of hash maps with doubly linked list for O(1) operations.

Binary Search Template Usage

Code Example:

```
// Example: Classic binary search int[] sortedArray = {1, 3, 5, 7, 9, 11, 13, 15}; int index = BinarySearchTemplate.BinarySearch(sortedArray, 7); // Returns 3 (index of 7 in the array) // Example: Search in rotated sorted array int[] rotated = {4, 5, 6, 7, 0, 1, 2}; int found = BinarySearchTemplate.SearchRotated(rotated, 0); // Returns 4 (index of 0 in the rotated array)
```

Explanation:

Binary search is versatile beyond sorted arrays. The rotated array search handles cases where the array is sorted but rotated. The key insight is determining which half of the array is sorted and searching accordingly.

Big O Notation Reference

Complexity	Name	Example Operations	Suitable For
O(1)	Constant	Array access, Hash lookup	Real-time systems
O(log n)	Logarithmic	Binary search, Tree operations	Large datasets
O(n)	Linear	Array traversal, Linear search	Most operations
O(n log n)	Linearithmic	Merge sort, Quick sort	Sorting algorithms
O(n^2)	Quadratic	Nested loops, Bubble sort	Small datasets
O(2■)	Exponential	Recursive fibonacci	Small inputs only
O(n!)	Factorial	Permutations, TSP	Very small inputs

Performance Optimization Tips

- **Choose the right data structure:** Hash maps for O(1) lookup, heaps for priority operations
- **Avoid premature optimization:** Profile before optimizing, focus on algorithmic improvements
- **Use appropriate collection types:** List for indexed access, Dictionary for key-value pairs
- **Consider memory allocation:** Reuse objects, avoid unnecessary copying
- **Parallel processing:** Use Parallel.For, PLINQ for CPU-intensive operations
- **Caching results:** Memoization for recursive algorithms, dynamic programming
- **Early termination:** Break loops when condition is met, use flags to avoid unnecessary work

Testing & Debugging Guide

Unit Testing Best Practices:
• Test edge cases (empty arrays, single elements, duplicates)
• Test boundary conditions (minimum/maximum values)
• Test performance with large datasets
• Use descriptive test names that explain what is being tested
• Follow Arrange-Act-Assert pattern

Debugging Techniques:
• Use breakpoints and step through code
• Add console logging for intermediate values
• Visualize data structures (arrays, trees, graphs)
• Test with smaller, hand-crafted inputs
• Use assertions to validate assumptions

Performance Profiling:
• Use BenchmarkDotNet for accurate measurements
• Compare different algorithmic approaches
• Profile memory usage alongside time complexity
• Test with various input sizes and patterns

Project Structure & Usage

Project Structure:
AlgorithmMaster/
■■■ AlgorithmMaster.csproj # .NET 8 project file
■■■ Program.cs # Main entry point with menu system
■■■ Templates/ # Algorithm templates
■ ■■■ ArrayTemplate.cs # Array operations and patterns
■ ■■■ StringTemplate.cs # String manipulation
■ ■■■ HashMapTemplate.cs # Hash map patterns
■ ■■■ TwoPointersTemplate.cs # Two pointers technique
■ ■■■ SlidingWindowTemplate.cs # Sliding window patterns
■ ■■■ BinarySearchTemplate.cs # Binary search variations
■ ■■■ BFSTemplate.cs # Breadth-first search
■ ■■■ DFSTemplate.cs # Depth-first search
■ ■■■ BacktrackingTemplate.cs # Backtracking patterns
■ ■■■ PriorityQueueTemplate.cs # Heap and priority queue
■■■ Examples/ # Usage examples
■ ■■■ ArrayExamples.cs
■ ■■■ HashMapExamples.cs
■ ■■■ [Other example files]
■■■ Utils/ # Utility classes
■■■ CommonStructures.cs # TreeNode, ListNode, etc.
■■■ TestRunner.cs # Test execution utilities
■■■ PerformanceProfiler.cs # Performance testing

How to Use the Templates:

1. Reference the Template:
```csharp
using AlgorithmMaster.Templates;
using AlgorithmMaster.Utils;
```

2. Choose Appropriate Template:
• Array problems → ArrayTemplate
• String manipulation → StringTemplate
• Key-value operations → HashMapTemplate
• Sorted data search → BinarySearchTemplate
• Graph traversal → BFS/DFSTemplate

3. Adapt to Your Problem:
Each template provides methods that can be directly used or modified. The examples show practical usage patterns and common modifications.

4. Test Your Solution:
Use the TestRunner utility for automated testing:
```csharp
TestRunner.RunTest("Two Sum Test", () => TwoSum(nums, target), expectedResult);
```

5. Profile Performance:
Use PerformanceProfiler for benchmarking different approaches.

Best Practices:
• Understand the algorithm before using the template

• Modify templates to fit specific problem constraints
• Always test with edge cases
• Consider time and space complexity trade-offs
• Document any modifications for future reference

Conclusion

This guide provides comprehensive templates for solving the most common algorithmic problems in C#. Each template is production-ready, well-tested, and follows best practices for .NET development.

Key Takeaways:
• Master the fundamental patterns before moving to complex algorithms
• Understand time and space complexity trade-offs
• Practice implementing templates from scratch to build intuition
• Use these templates as a starting point, not a crutch
• Always test thoroughly with various inputs

Next Steps:
• Implement each template without looking at the code
• Solve 5-10 problems using each pattern
• Learn to recognize which pattern applies to new problems
• Explore advanced topics like dynamic programming and greedy algorithms
• Contribute improvements back to the template library

Remember: The goal is not to memorize templates, but to understand the underlying patterns and principles that make them work. With practice, you'll develop the intuition to recognize patterns and adapt solutions to new problems.

Happy Coding!