The Ultimate Comprehensive Handbook on Strings and String Algorithms for Interview Mastery

This is the most complete, detailed handbook on strings and string algorithms designed to make you an absolute expert for technical interviews, competitive programming, and real-world development. Every concept, algorithm, and technique is explained in exhaustive detail with full implementations, complexity analysis, and practical insights.

Chapter 1: Foundations & Core String Concepts

1.1 String Fundamentals and Properties

Strings are fundamental data structures representing sequences of characters. Understanding their core properties is essential before diving into complex algorithms.[1] [2]

Definition and Characteristics:

- A string is an ordered sequence of characters from a finite alphabet
- In Python, strings are **immutable** objects, meaning any modification creates a new string
- Strings support zero-based indexing with negative indices for reverse access
- String slicing allows efficient substring extraction using `s[start:end:step]` notation

Memory Representation:

```
# String storage in memory
s = "hello"  # Creates immutable string object
print(id(s))  # Memory address

# Modifying creates new object
s += " world"  # New string object created
print(id(s))  # Different memory address
```

Character Encoding Systems:

- **ASCII**: 7-bit encoding supporting 128 characters (0-127)
- **Unicode**: Universal character set supporting multiple languages
- **UTF-8**: Variable-length encoding compatible with ASCII
- **UTF-16**: Fixed 16-bit encoding used by some systems

String Properties Analysis:

```
def analyze_string_properties(s):
    """Comprehensive string property analysis"""
    properties = {
        'length': len(s),
        'is_empty': len(s) == 0,
        'is_alphabetic': s.isalpha(),
        'is_numeric': s.isdigit(),
        'is_alphanumeric': s.isalnum(),
        'is_uppercase': s.isupper(),
        'is_lowercase': s.islower(),
        'char_frequency': {},
        'unique_chars': len(set(s)),
        'first_char': s if s else None,
        'last_char': s[-1] if s else None
    }

    # Character frequency distribution
    for char in s:
        properties['char_frequency'][char] = properties['char_frequency'].get(char, 0) +

    return properties

# Example usage
text = "Programming"
analysis = analyze_string_properties(text)
print(f"String Analysis: {analysis}")
```

1.2 Essential String Operations

Basic String Manipulation Operations:

```
class StringOperations:
    """Comprehensive collection of fundamental string operations"""

    @staticmethod
    def reverse_string(s):
        """
        Reverse string using multiple approaches
        Time: O(n), Space: O(n) for new string creation
        """
        # Method 1: Pythonic slicing (most efficient)
        return s[::-1]

    @staticmethod
    def reverse_string_inplace(s_list):
        """
        In-place reversal for character list
        Time: O(n), Space: O(1)
        """
        left, right = 0, len(s_list) - 1
        while left < right:
            s_list[left], s_list[right] = s_list[right], s_list[left]
```

```
            left += 1
            right -= 1
        return s_list

    @staticmethod
    def is_palindrome(s):
        """
        Check palindrome with multiple validation levels
        Time: O(n), Space: O(1)
        """
        # Clean string: remove non-alphanumeric, convert to lowercase
        cleaned = ''.join(char.lower() for char in s if char.isalnum())

        left, right = 0, len(cleaned) - 1
        while left < right:
            if cleaned[left] != cleaned[right]:
                return False
            left += 1
            right -= 1
        return True

    @staticmethod
    def first_non_repeating_char(s):
        """
        Find first non-repeating character with index
        Time: O(n), Space: O(1) - limited to alphabet size
        """
        char_count = {}

        # First pass: count frequencies
        for char in s:
            char_count[char] = char_count.get(char, 0) + 1

        # Second pass: find first non-repeating
        for i, char in enumerate(s):
            if char_count[char] == 1:
                return i, char

        return -1, None

    @staticmethod
    def remove_duplicates(s):
        """
        Remove duplicate characters while preserving order
        Time: O(n), Space: O(n)
        """
        seen = set()
        result = []

        for char in s:
            if char not in seen:
                seen.add(char)
                result.append(char)

        return ''.join(result)
```

```
    @staticmethod
    def character_frequency(s):
        """
        Comprehensive character frequency analysis
        Time: O(n), Space: O(k) where k is unique characters
        """
        frequency = {}
        for char in s:
            frequency[char] = frequency.get(char, 0) + 1

        # Sort by frequency (descending) then by character
        sorted_freq = sorted(frequency.items(), key=lambda x: (-x[^1], x))

        return {
            'frequency_map': frequency,
            'sorted_by_frequency': sorted_freq,
            'most_frequent': sorted_freq if sorted_freq else None,
            'least_frequent': min(frequency.items(), key=lambda x: x[^21]) if frequency e
        }

# Demonstration of string operations
demo_string = "Programming Interview"
ops = StringOperations()

print(f"Original: {demo_string}")
print(f"Reversed: {ops.reverse_string(demo_string)}")
print(f"Is Palindrome: {ops.is_palindrome(demo_string)}")
print(f"First Non-repeating: {ops.first_non_repeating_char(demo_string)}")
print(f"Remove Duplicates: {ops.remove_duplicates(demo_string)}")
print(f"Character Frequency: {ops.character_frequency(demo_string)}")
```

1.3 String Comparison and Similarity

Advanced String Comparison Techniques:

```
class StringComparison:
    """Advanced string comparison and similarity algorithms"""

    @staticmethod
    def are_anagrams(s1, s2):
        """
        Multiple approaches to anagram detection
        Time: O(n), Space: O(1) - limited alphabet
        """
        if len(s1) != len(s2):
            return False

        # Method 1: Character frequency comparison
        char_count = {}

        # Count characters in first string
        for char in s1.lower():
            char_count[char] = char_count.get(char, 0) + 1

        # Subtract characters from second string
```

```
        for char in s2.lower():
            if char not in char_count:
                return False
            char_count[char] -= 1
            if char_count[char] == 0:
                del char_count[char]

        return len(char_count) == 0

    @staticmethod
    def are_anagrams_sorting(s1, s2):
        """
        Anagram detection using sorting
        Time: O(n log n), Space: O(n)
        """
        return sorted(s1.lower()) == sorted(s2.lower())

    @staticmethod
    def hamming_distance(s1, s2):
        """
        Calculate Hamming distance (number of different positions)
        Time: O(n), Space: O(1)
        """
        if len(s1) != len(s2):
            raise ValueError("Strings must have equal length for Hamming distance")

        distance = 0
        for i in range(len(s1)):
            if s1[i] != s2[i]:
                distance += 1

        return distance

    @staticmethod
    def longest_common_prefix(strings):
        """
        Find longest common prefix among array of strings
        Time: O(S) where S is sum of all string lengths, Space: O(1)
        """
        if not strings:
            return ""

        # Find minimum length string
        min_len = min(len(s) for s in strings)

        for i in range(min_len):
            char = strings[i]
            # Check if character at position i is same in all strings
            for string in strings[1:]:
                if string[i] != char:
                    return strings[:i]

        return strings[:min_len]

    @staticmethod
    def is_rotation(s1, s2):
```

```
        """
        Check if s2 is a rotation of s1
        Time: O(n), Space: O(n)
        """
        if len(s1) != len(s2):
            return False

        # Trick: s2 will be substring of s1+s1 if it's a rotation
        return s2 in s1 + s1

    @staticmethod
    def is_subsequence(s, t):
        """
        Check if s is subsequence of t
        Time: O(n), Space: O(1)
        """
        i = 0  # Pointer for s

        for char in t:
            if i < len(s) and s[i] == char:
                i += 1

        return i == len(s)
```

Chapter 2: Pattern Matching Algorithms

2.1 Naive String Matching Algorithm

The foundation of pattern matching, essential for understanding more complex algorithms.[3] [4] [5]

Algorithm Explanation:
The naive approach compares the pattern with every possible position in the text. Despite its O(nm) worst-case complexity, it's important for understanding the problem structure.

```
class NaivePatternMatching:
    """Complete implementation of naive string matching with optimizations"""

    @staticmethod
    def naive_search(text, pattern):
        """
        Basic naive pattern matching
        Time: O(nm), Space: O(1)
        """
        matches = []
        n, m = len(text), len(pattern)

        if m > n:
            return matches

        for i in range(n - m + 1):
            # Check if pattern matches at position i
            j = 0
```

```
            while j < m and text[i + j] == pattern[j]:
                j += 1

            if j == m:  # Full match found
                matches.append(i)

        return matches

    @staticmethod
    def naive_search_with_statistics(text, pattern):
        """
        Naive search with detailed statistics
        Returns matches and performance metrics
        """
        matches = []
        n, m = len(text), len(pattern)
        comparisons = 0

        for i in range(n - m + 1):
            j = 0
            while j < m:
                comparisons += 1
                if text[i + j] != pattern[j]:
                    break
                j += 1

            if j == m:
                matches.append(i)

        return {
            'matches': matches,
            'total_comparisons': comparisons,
            'positions_checked': n - m + 1,
            'efficiency': comparisons / (n - m + 1) if n >= m else 0
        }

    @staticmethod
    def all_occurrences_naive(text, patterns):
        """
        Find all occurrences of multiple patterns
        Time: O(n * k * m), where k is number of patterns
        """
        results = {}

        for pattern in patterns:
            results[pattern] = NaivePatternMatching.naive_search(text, pattern)

        return results

# Example usage and performance analysis
text = "ABABDABACDABABCABCABCABCABC"
pattern = "ABABCAB"

naive_matcher = NaivePatternMatching()
result = naive_matcher.naive_search_with_statistics(text, pattern)
print(f"Naive Search Results: {result}")
```

2.2 Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm achieves linear time complexity by preprocessing the pattern to avoid redundant comparisons.[3] [4] [6]

Core Innovation - The Failure Function:
The KMP algorithm's power lies in the Longest Proper Prefix which is also Suffix (LPS) array, which tells us how much we can safely skip when a mismatch occurs.

```
class KMPAlgorithm:
    """Complete KMP implementation with detailed explanation and variants"""

    @staticmethod
    def compute_lps_array(pattern):
        """
        Compute Longest Proper Prefix which is also Suffix array
        This is the heart of KMP algorithm
        Time: O(m), Space: O(m)
        """
        m = len(pattern)
        lps =  * m
        length = 0  # Length of the previous longest prefix suffix
        i = 1

        # Build lps array
        while i < m:
            if pattern[i] == pattern[length]:
                length += 1
                lps[i] = length
                i += 1
            else:
                if length != 0:
                    # This is tricky. Consider the example "AAACAAAA" and i = 7.
                    # The idea is similar to search step.
                    length = lps[length - 1]
                    # Also note that we do not increment i here
                else:
                    lps[i] = 0
                    i += 1

        return lps

    @staticmethod
    def kmp_search(text, pattern):
        """
        KMP pattern matching algorithm
        Time: O(n + m), Space: O(m)
        """
        n, m = len(text), len(pattern)

        if m == 0:
            return []
        if m > n:
            return []
```

```
        # Preprocess pattern to get LPS array
        lps = KMPAlgorithm.compute_lps_array(pattern)

        matches = []
        i = 0  # Index for text
        j = 0  # Index for pattern

        while i < n:
            if text[i] == pattern[j]:
                i += 1
                j += 1

            if j == m:
                matches.append(i - j)
                j = lps[j - 1]  # Get next position to match
            elif i < n and text[i] != pattern[j]:
                if j != 0:
                    j = lps[j - 1]
                else:
                    i += 1

        return matches

    @staticmethod
    def kmp_search_detailed(text, pattern):
        """
        KMP search with step-by-step execution details
        """
        n, m = len(text), len(pattern)
        lps = KMPAlgorithm.compute_lps_array(pattern)

        matches = []
        steps = []
        i = j = 0
        comparisons = 0

        while i < n:
            comparisons += 1

            if text[i] == pattern[j]:
                steps.append(f"Match at text[{i}]='{text[i]}' with pattern[{j}]='{patterr
                i += 1
                j += 1

            if j == m:
                matches.append(i - j)
                steps.append(f"Complete match found at position {i - j}")
                j = lps[j - 1]
            elif i < n and text[i] != pattern[j]:
                steps.append(f"Mismatch at text[{i}]='{text[i]}' with pattern[{j}]='{patt
                if j != 0:
                    j = lps[j - 1]
                    steps.append(f"Using LPS: jumping to position {j} in pattern")
                else:
                    i += 1
```

```
        return {
            'matches': matches,
            'lps_array': lps,
            'total_comparisons': comparisons,
            'execution_steps': steps
        }

    @staticmethod
    def kmp_all_overlapping_matches(text, pattern):
        """
        Find all overlapping occurrences using KMP
        """
        n, m = len(text), len(pattern)
        lps = KMPAlgorithm.compute_lps_array(pattern)

        matches = []
        i = j = 0

        while i < n:
            if text[i] == pattern[j]:
                i += 1
                j += 1

            if j == m:
                matches.append(i - j)
                # For overlapping matches, don't skip as much
                j = lps[j - 1] if lps[j - 1] > 0 else 0
            elif i < n and text[i] != pattern[j]:
                if j != 0:
                    j = lps[j - 1]
                else:
                    i += 1

        return matches

# Detailed KMP demonstration
text = "ABABCABABA"
pattern = "ABAB"

kmp = KMPAlgorithm()
detailed_result = kmp.kmp_search_detailed(text, pattern)

print(f"Text: {text}")
print(f"Pattern: {pattern}")
print(f"LPS Array: {detailed_result['lps_array']}")
print(f"Matches found at: {detailed_result['matches']}")
print(f"Total comparisons: {detailed_result['total_comparisons']}")
```

LPS Array Construction Detailed Analysis:

```
def explain_lps_construction(pattern):
    """
    Step-by-step explanation of LPS array construction
    """
    print(f"Constructing LPS array for pattern: {pattern}")
```

```
    m = len(pattern)
    lps =  * m
    length = 0
    i = 1

    print(f"Initial: lps = {lps}, length = {length}, i = {i}")

    step = 1
    while i < m:
        print(f"\nStep {step}:")
        print(f"Comparing pattern[{i}]='{pattern[i]}' with pattern[{length}]='{pattern[le

        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            print(f"Match! length = {length}, lps[{i}] = {length}")
            i += 1
        else:
            if length != 0:
                print(f"Mismatch! Using lps[{length-1}] = {lps[length-1]}")
                length = lps[length - 1]
            else:
                lps[i] = 0
                print(f"Mismatch and length = 0, so lps[{i}] = 0")
                i += 1

        print(f"Current state: lps = {lps}, length = {length}, i = {i}")
        step += 1

    return lps

# Example LPS construction
pattern = "ABABCABAB"
lps_result = explain_lps_construction(pattern)
```

2.3 Rabin-Karp Algorithm with Rolling Hash

The Rabin-Karp algorithm uses hashing to achieve efficient pattern matching, particularly useful for multiple pattern searches.[7] [3] [8] [9]

Rolling Hash Concept:
The key innovation is the rolling hash function that allows us to compute the hash of the next substring in constant time by removing the contribution of the first character and adding the contribution of the new character.

```
class RabinKarpAlgorithm:
    """Complete Rabin-Karp implementation with collision handling"""

    def __init__(self, base=31, mod=10**9 + 7):
        """
        Initialize with base and modulo for hash function
        base: Usually a prime number
        mod: Large prime to prevent overflow
        """
```

```
        self.base = base
        self.mod = mod

    def polynomial_hash(self, s):
        """
        Compute polynomial rolling hash for string
        Hash = (s * base^(n-1) + s[^21] * base^(n-2) + ... + s[n-1]) mod mod
        """
        hash_value = 0
        power = 1

        for char in reversed(s):
            hash_value = (hash_value + ord(char) * power) % self.mod
            power = (power * self.base) % self.mod

        return hash_value

    def search(self, text, pattern):
        """
        Rabin-Karp pattern matching with collision handling
        Time: O(n + m) average, O(nm) worst case
        Space: O(1)
        """
        n, m = len(text), len(pattern)

        if m > n:
            return []

        matches = []

        # Calculate hash of pattern and first window of text
        pattern_hash = self.polynomial_hash(pattern)
        window_hash = self.polynomial_hash(text[:m])

        # Calculate base^(m-1) % mod for rolling hash
        h = 1
        for _ in range(m - 1):
            h = (h * self.base) % self.mod

        # Check first window
        if pattern_hash == window_hash and text[:m] == pattern:
            matches.append(0)

        # Slide the pattern over text
        for i in range(1, n - m + 1):
            # Remove leading character and add trailing character
            window_hash = (self.base * (window_hash - ord(text[i-1]) * h) + ord(text[i+m-

            # Ensure positive hash
            if window_hash < 0:
                window_hash += self.mod

            # Check if hash matches and verify string equality
            if pattern_hash == window_hash and text[i:i+m] == pattern:
                matches.append(i)
```

```
        return matches

    def search_with_statistics(self, text, pattern):
        """
        Rabin-Karp search with detailed statistics
        """
        n, m = len(text), len(pattern)

        if m > n:
            return {'matches': [], 'hash_collisions': 0, 'string_comparisons': 0}

        matches = []
        hash_collisions = 0
        string_comparisons = 0

        pattern_hash = self.polynomial_hash(pattern)
        window_hash = self.polynomial_hash(text[:m])

        h = 1
        for _ in range(m - 1):
            h = (h * self.base) % self.mod

        # Check first window
        if pattern_hash == window_hash:
            string_comparisons += 1
            if text[:m] == pattern:
                matches.append(0)
            else:
                hash_collisions += 1

        # Slide the pattern over text
        for i in range(1, n - m + 1):
            window_hash = (self.base * (window_hash - ord(text[i-1]) * h) + ord(text[i+m-
            if window_hash < 0:
                window_hash += self.mod

            if pattern_hash == window_hash:
                string_comparisons += 1
                if text[i:i+m] == pattern:
                    matches.append(i)
                else:
                    hash_collisions += 1

        return {
            'matches': matches,
            'hash_collisions': hash_collisions,
            'string_comparisons': string_comparisons,
            'total_positions': n - m + 1,
            'efficiency': string_comparisons / (n - m + 1) if n >= m else 0
        }

    def multiple_pattern_search(self, text, patterns):
        """
        Search for multiple patterns simultaneously
        This is where Rabin-Karp really shines
        """
```

```
        if not patterns:
            return {}

        n = len(text)
        pattern_hashes = {}
        results = {}

        # Group patterns by length for efficiency
        patterns_by_length = {}
        for pattern in patterns:
            length = len(pattern)
            if length not in patterns_by_length:
                patterns_by_length[length] = []
            patterns_by_length[length].append(pattern)

        # Search for each length group
        for m, pattern_group in patterns_by_length.items():
            if m > n:
                continue

            # Calculate hashes for all patterns of this length
            for pattern in pattern_group:
                pattern_hashes[pattern] = self.polynomial_hash(pattern)
                results[pattern] = []

            # Calculate rolling hash coefficient
            h = 1
            for _ in range(m - 1):
                h = (h * self.base) % self.mod

            # Calculate initial window hash
            window_hash = self.polynomial_hash(text[:m])

            # Check first window against all patterns
            for pattern in pattern_group:
                if pattern_hashes[pattern] == window_hash and text[:m] == pattern:
                    results[pattern].append(0)

            # Slide window and check remaining positions
            for i in range(1, n - m + 1):
                window_hash = (self.base * (window_hash - ord(text[i-1]) * h) + ord(text[
                if window_hash < 0:
                    window_hash += self.mod

                for pattern in pattern_group:
                    if pattern_hashes[pattern] == window_hash and text[i:i+m] == pattern:
                        results[pattern].append(i)

        return results

# Comprehensive Rabin-Karp demonstration
text = "GEEKS FOR GEEKS"
pattern = "GEEK"

rk = RabinKarpAlgorithm()
basic_result = rk.search(text, pattern)
```

```
detailed_result = rk.search_with_statistics(text, pattern)

print(f"Text: {text}")
print(f"Pattern: {pattern}")
print(f"Matches: {basic_result}")
print(f"Detailed stats: {detailed_result}")

# Multiple pattern search example
patterns = ["GEEK", "FOR", "EKS", "SEEK"]
multi_result = rk.multiple_pattern_search(text, patterns)
print(f"\nMultiple pattern search results:")
for pattern, matches in multi_result.items():
    print(f"'{pattern}': {matches}")
```

2.4 Boyer-Moore Algorithm

The Boyer-Moore algorithm is highly efficient for large alphabets and long patterns, using two heuristics: bad character and good suffix.[3] [4] [10]

```
class BoyerMooreAlgorithm:
    """Complete Boyer-Moore implementation with both heuristics"""

    @staticmethod
    def build_bad_character_table(pattern):
        """
        Build bad character heuristic table
        Time: O(m + σ) where σ is alphabet size
        """
        table = {}
        m = len(pattern)

        # Initialize all characters to -1 (not found)
        for i in range(256):  # Extended ASCII
            table[chr(i)] = -1

        # Fill actual characters in pattern
        for i in range(m):
            table[pattern[i]] = i

        return table

    @staticmethod
    def compute_good_suffix_array(pattern):
        """
        Compute good suffix heuristic array
        More complex but provides better skip distances
        """
        m = len(pattern)
        good_suffix =  * (m + 1)
        border_pos =  * (m + 1)

        i = m
        j = m + 1
        border_pos[i] = j
```

```
        while i > 0:
            while j <= m and pattern[i - 1] != pattern[j - 1]:
                if good_suffix[j] == 0:
                    good_suffix[j] = j - i
                j = border_pos[j]

            i -= 1
            j -= 1
            border_pos[i] = j

        j = border_pos
        for i in range(m + 1):
            if good_suffix[i] == 0:
                good_suffix[i] = j

            if i == j:
                j = border_pos[j]

        return good_suffix

    @staticmethod
    def boyer_moore_search(text, pattern):
        """
        Boyer-Moore pattern matching with both heuristics
        Time: O(n/m) best case, O(nm) worst case
        """
        n, m = len(text), len(pattern)

        if m > n:
            return []

        # Preprocessing
        bad_char_table = BoyerMooreAlgorithm.build_bad_character_table(pattern)
        good_suffix = BoyerMooreAlgorithm.compute_good_suffix_array(pattern)

        matches = []
        shift = 0  # Position of pattern in text

        while shift <= n - m:
            j = m - 1  # Start from rightmost character of pattern

            # Match characters from right to left
            while j >= 0 and pattern[j] == text[shift + j]:
                j -= 1

            if j < 0:
                # Pattern found
                matches.append(shift)
                # Shift by good suffix heuristic
                shift += good_suffix
            else:
                # Calculate shift using both heuristics
                bad_char_shift = max(1, j - bad_char_table.get(text[shift + j], -1))
                good_suffix_shift = good_suffix[j + 1]
```

```
                # Take maximum shift from both heuristics
                shift += max(bad_char_shift, good_suffix_shift)

        return matches

    @staticmethod
    def boyer_moore_simple(text, pattern):
        """
        Simplified Boyer-Moore using only bad character heuristic
        Easier to understand and implement
        """
        n, m = len(text), len(pattern)

        if m > n:
            return []

        # Build bad character table
        bad_char = BoyerMooreAlgorithm.build_bad_character_table(pattern)

        matches = []
        i = 0  # Position in text

        while i <= n - m:
            j = m - 1  # Start from end of pattern

            # Match from right to left
            while j >= 0 and pattern[j] == text[i + j]:
                j -= 1

            if j < 0:
                # Pattern found
                matches.append(i)
                # Move to next potential match
                i += 1
            else:
                # Skip based on bad character heuristic
                skip = max(1, j - bad_char.get(text[i + j], -1))
                i += skip

        return matches

    @staticmethod
    def boyer_moore_with_stats(text, pattern):
        """
        Boyer-Moore with performance statistics
        """
        n, m = len(text), len(pattern)
        comparisons = 0
        shifts = 0

        bad_char = BoyerMooreAlgorithm.build_bad_character_table(pattern)
        matches = []
        i = 0

        while i <= n - m:
            j = m - 1
```

```
            shifts += 1

            while j >= 0:
                comparisons += 1
                if pattern[j] != text[i + j]:
                    break
                j -= 1

            if j < 0:
                matches.append(i)
                i += 1
            else:
                skip = max(1, j - bad_char.get(text[i + j], -1))
                i += skip

        return {
            'matches': matches,
            'comparisons': comparisons,
            'shifts': shifts,
            'text_length': n,
            'pattern_length': m,
            'efficiency': n / comparisons if comparisons > 0 else 0
        }

# Boyer-Moore demonstration and comparison
text = "ABAAABCDABABCABCABCABC"
pattern = "ABCAB"

bm = BoyerMooreAlgorithm()
simple_result = bm.boyer_moore_simple(text, pattern)
full_result = bm.boyer_moore_search(text, pattern)
stats_result = bm.boyer_moore_with_stats(text, pattern)

print(f"Text: {text}")
print(f"Pattern: {pattern}")
print(f"Simple Boyer-Moore: {simple_result}")
print(f"Full Boyer-Moore: {full_result}")
print(f"Performance stats: {stats_result}")
```

2.5 Z Algorithm

The Z algorithm computes, for each position in a string, the length of the longest substring starting from that position which is also a prefix of the string.[11] [12] [13]

```
class ZAlgorithm:
    """Complete Z Algorithm implementation with applications"""

    @staticmethod
    def compute_z_array(s):
        """
        Compute Z array for string s
        Z[i] = length of longest substring starting from i that is also prefix of s
        Time: O(n), Space: O(n)
        """
```

```
        n = len(s)
        z =  * n
        z = n  # Entire string is prefix of itself

        left = right = 0

        for i in range(1, n):
            if i <= right:
                # We are inside a Z-box, so we can use previously computed values
                z[i] = min(right - i + 1, z[i - left])

            # Try to extend the match
            while i + z[i] < n and s[z[i]] == s[i + z[i]]:
                z[i] += 1

            # If we extended past the current Z-box, update it
            if i + z[i] - 1 > right:
                left = i
                right = i + z[i] - 1

        return z

    @staticmethod
    def z_algorithm_search(text, pattern):
        """
        Pattern matching using Z algorithm
        Time: O(n + m), Space: O(n + m)
        """
        if not pattern or not text:
            return []

        # Create combined string: pattern + delimiter + text
        combined = pattern + '$' + text
        z_array = ZAlgorithm.compute_z_array(combined)

        matches = []
        pattern_length = len(pattern)

        # Look for Z values equal to pattern length
        for i in range(pattern_length + 1, len(combined)):
            if z_array[i] == pattern_length:
                # Match found at position i - (pattern_length + 1) in original text
                matches.append(i - pattern_length - 1)

        return matches

    @staticmethod
    def z_algorithm_detailed(s):
        """
        Z algorithm with step-by-step explanation
        """
        n = len(s)
        z =  * n
        z = n

        left = right = 0
```

```
        steps = []

        steps.append(f"Initial: Z = {n} (entire string)")

        for i in range(1, n):
            steps.append(f"\nProcessing position {i}:")

            if i <= right:
                # Inside Z-box
                alpha = i - left
                beta = right - i + 1
                z_alpha = z[alpha]

                steps.append(f"  Inside Z-box [L={left}, R={right}]")
                steps.append(f"  α = i - L = {i} - {left} = {alpha}")
                steps.append(f"  β = R - i + 1 = {right} - {i} + 1 = {beta}")
                steps.append(f"  Z[α] = Z[{alpha}] = {z_alpha}")

                if z_alpha < beta:
                    z[i] = z_alpha
                    steps.append(f"  Z[α] < β, so Z[{i}] = Z[α] = {z_alpha}")
                else:
                    z[i] = beta
                    steps.append(f"  Z[α] ≥ β, so Z[{i}] = β = {beta}, then extend")
            else:
                steps.append(f"  Outside Z-box, start from 0")
                z[i] = 0

            # Try to extend
            original_z = z[i]
            while i + z[i] < n and s[z[i]] == s[i + z[i]]:
                z[i] += 1

            if z[i] > original_z:
                steps.append(f"  Extended from {original_z} to {z[i]}")

            # Update Z-box if necessary
            if i + z[i] - 1 > right:
                left = i
                right = i + z[i] - 1
                steps.append(f"  Updated Z-box: L = {left}, R = {right}")

            steps.append(f"  Final: Z[{i}] = {z[i]}")

        return {
            'z_array': z,
            'steps': steps
        }

    @staticmethod
    def find_period(s):
        """
        Find the smallest period of string using Z algorithm
        """
        z = ZAlgorithm.compute_z_array(s)
        n = len(s)
```

```
        for i in range(1, n):
            if i + z[i] == n:  # Pattern extends to end
                return i

        return n  # No period found, entire string is period

    @staticmethod
    def is_rotation_z_algo(s1, s2):
        """
        Check if s2 is rotation of s1 using Z algorithm
        """
        if len(s1) != len(s2):
            return False

        # Check if s2 appears in s1 + s1
        combined = s2 + '$' + s1 + s1
        z = ZAlgorithm.compute_z_array(combined)
        pattern_length = len(s2)

        for i in range(pattern_length + 1, len(combined)):
            if z[i] >= pattern_length:
                return True

        return False

# Z Algorithm comprehensive demonstration
string = "aabaaaba"
pattern = "aab"
text = "aabaaababaabaab"

z_algo = ZAlgorithm()

# Basic Z array computation
z_result = z_algo.compute_z_array(string)
print(f"String: {string}")
print(f"Z array: {z_result}")

# Pattern matching
matches = z_algo.z_algorithm_search(text, pattern)
print(f"\nPattern '{pattern}' found in '{text}' at positions: {matches}")

# Detailed explanation
detailed = z_algo.z_algorithm_detailed("aabaab")
print(f"\nDetailed Z algorithm steps:")
for step in detailed['steps']:
    print(step)

# Period finding
period = z_algo.find_period("abcabcabc")
print(f"\nSmallest period of 'abcabcabc': {period}")
```

Chapter 3: Advanced String Searching Algorithms

3.1 Aho-Corasick Algorithm for Multiple Pattern Matching

The Aho-Corasick algorithm efficiently searches for multiple patterns simultaneously by building a finite automaton.[14] [15] [16]

```
class AhoCorasickAlgorithm:
    """Complete Aho-Corasick implementation for multiple pattern matching"""

    class TrieNode:
        def __init__(self):
            self.children = {}
            self.failure_link = None
            self.output_link = None
            self.patterns = []  # Patterns ending at this node
            self.pattern_indices = []

    def __init__(self, patterns):
        """
        Initialize Aho-Corasick automaton with list of patterns
        Time: O(sum of pattern lengths), Space: O(trie size)
        """
        self.root = self.TrieNode()
        self.patterns = patterns
        self._build_trie()
        self._build_failure_links()
        self._build_output_links()

    def _build_trie(self):
        """Build trie from patterns"""
        for idx, pattern in enumerate(self.patterns):
            current = self.root

            for char in pattern:
                if char not in current.children:
                    current.children[char] = self.TrieNode()
                current = current.children[char]

            current.patterns.append(pattern)
            current.pattern_indices.append(idx)

    def _build_failure_links(self):
        """Build failure links using BFS"""
        from collections import deque

        queue = deque()

        # Initialize failure links for nodes at depth 1
        for char, child in self.root.children.items():
            child.failure_link = self.root
            queue.append(child)

        # Build failure links for deeper nodes
        while queue:
```

```
            current_node = queue.popleft()

            for char, child in current_node.children.items():
                queue.append(child)

                # Find the failure link for this child
                failure_node = current_node.failure_link

                while failure_node is not None and char not in failure_node.children:
                    failure_node = failure_node.failure_link

                if failure_node is None:
                    child.failure_link = self.root
                else:
                    child.failure_link = failure_node.children[char]

    def _build_output_links(self):
        """Build output links for efficient pattern reporting"""
        from collections import deque

        queue = deque()

        # Initialize for root's children
        for child in self.root.children.values():
            queue.append(child)

        while queue:
            current_node = queue.popleft()

            # Set output link
            if current_node.failure_link.patterns:
                current_node.output_link = current_node.failure_link
            else:
                current_node.output_link = current_node.failure_link.output_link

            # Add children to queue
            for child in current_node.children.values():
                queue.append(child)

    def search(self, text):
        """
        Search for all patterns in text
        Time: O(n + z) where z is number of pattern occurrences
        """
        results = []
        current_node = self.root

        for i, char in enumerate(text):
            # Follow failure links until we find a match or reach root
            while current_node is not None and char not in current_node.children:
                current_node = current_node.failure_link

            if current_node is None:
                current_node = self.root
                continue
```

```
            current_node = current_node.children[char]

            # Report all patterns ending at current position
            temp_node = current_node
            while temp_node is not None:
                for j, pattern in enumerate(temp_node.patterns):
                    pattern_idx = temp_node.pattern_indices[j]
                    start_pos = i - len(pattern) + 1
                    results.append({
                        'pattern': pattern,
                        'pattern_index': pattern_idx,
                        'start_position': start_pos,
                        'end_position': i
                    })
                temp_node = temp_node.output_link

        return results

    def search_with_counts(self, text):
        """Search and return pattern counts"""
        occurrences = self.search(text)
        counts = {}

        for pattern in self.patterns:
            counts[pattern] = 0

        for occurrence in occurrences:
            counts[occurrence['pattern']] += 1

        return counts, occurrences

    def get_automaton_info(self):
        """Get information about the constructed automaton"""
        from collections import deque

        queue = deque([self.root])
        node_count = 0
        max_depth = 0

        while queue:
            level_size = len(queue)
            max_depth += 1

            for _ in range(level_size):
                node = queue.popleft()
                node_count += 1

                for child in node.children.values():
                    queue.append(child)

        return {
            'total_nodes': node_count,
            'max_depth': max_depth - 1,
            'total_patterns': len(self.patterns),
            'alphabet_size': len(set(''.join(self.patterns)))
        }
```

```
# Comprehensive Aho-Corasick demonstration
patterns = ["he", "she", "his", "hers", "her"]
text = "she sells seashells by the seashore"

ac = AhoCorasickAlgorithm(patterns)
results = ac.search(text)
counts, occurrences = ac.search_with_counts(text)
info = ac.get_automaton_info()

print(f"Text: {text}")
print(f"Patterns: {patterns}")
print(f"Automaton info: {info}")
print(f"\nPattern counts: {counts}")
print(f"\nAll occurrences:")
for occ in occurrences:
    print(f"  '{occ['pattern']}' at position {occ['start_position']}-{occ['end_position']
```

3.2 Suffix Array Construction and Applications

Suffix arrays provide a space-efficient alternative to suffix trees while supporting powerful string operations.[17] [18] [19] [20] [21]

```
class SuffixArray:
    """Complete suffix array implementation with construction algorithms"""

    def __init__(self, text):
        """
        Initialize suffix array for given text
        """
        self.text = text + '$'  # Add sentinel character
        self.n = len(self.text)
        self.suffix_array = self._construct_suffix_array_efficient()
        self.lcp_array = self._construct_lcp_array()

    def _construct_suffix_array_naive(self):
        """
        Naive suffix array construction
        Time: O(n²log n), Space: O(n)
        """
        suffixes = []

        for i in range(self.n):
            suffixes.append((self.text[i:], i))

        # Sort suffixes
        suffixes.sort()

        return [suffix[^21] for suffix in suffixes]

    def _construct_suffix_array_efficient(self):
        """
        Efficient suffix array construction using radix sort
        Time: O(n log n), Space: O(n)
```

```
        """
        # Initial ranking based on first character
        rank = [ord(char) for char in self.text]
        temp_rank =  * self.n
        suffix_array = list(range(self.n))

        k = 1
        while k < self.n:
            # Sort by second half of 2k-character substrings
            suffix_array.sort(key=lambda x: (rank[x], rank[x + k] if x + k < self.n else

            # Update ranks
            temp_rank[suffix_array] = 0
            for i in range(1, self.n):
                prev_suffix = suffix_array[i - 1]
                curr_suffix = suffix_array[i]

                prev_key = (rank[prev_suffix], rank[prev_suffix + k] if prev_suffix + k <
                curr_key = (rank[curr_suffix], rank[curr_suffix + k] if curr_suffix + k <

                if prev_key == curr_key:
                    temp_rank[curr_suffix] = temp_rank[prev_suffix]
                else:
                    temp_rank[curr_suffix] = temp_rank[prev_suffix] + 1

            rank, temp_rank = temp_rank, rank
            k *= 2

            # Early termination if all ranks are unique
            if rank[suffix_array[-1]] == self.n - 1:
                break

        return suffix_array

    def _construct_lcp_array(self):
        """
        Construct Longest Common Prefix array using Kasai's algorithm
        Time: O(n), Space: O(n)
        """
        lcp =  * self.n
        rank =  * self.n

        # Create rank array (inverse of suffix array)
        for i in range(self.n):
            rank[self.suffix_array[i]] = i

        h = 0  # Height of LCP

        for i in range(self.n):
            if rank[i] == 0:
                continue

            j = self.suffix_array[rank[i] - 1]

            # Calculate LCP between current suffix and previous in sorted order
            while i + h < self.n and j + h < self.n and self.text[i + h] == self.text[j +
```

```
                h += 1

            lcp[rank[i]] = h

            if h > 0:
                h -= 1

        return lcp

    def search_pattern(self, pattern):
        """
        Binary search for pattern in suffix array
        Time: O(m log n), Space: O(1)
        """
        left, right = 0, self.n - 1
        m = len(pattern)

        # Find leftmost occurrence
        while left < right:
            mid = (left + right) // 2
            suffix = self.text[self.suffix_array[mid]:]

            if suffix[:m] < pattern:
                left = mid + 1
            else:
                right = mid

        if left < self.n and self.text[self.suffix_array[left]:self.suffix_array[left] +
            # Find range of all occurrences
            start = left
            left, right = 0, self.n - 1

            # Find rightmost occurrence
            while left < right:
                mid = (left + right + 1) // 2
                suffix = self.text[self.suffix_array[mid]:]

                if suffix[:m] <= pattern:
                    left = mid
                else:
                    right = mid - 1

            end = left

            return [self.suffix_array[i] for i in range(start, end + 1)]

        return []

    def longest_repeated_substring(self):
        """
        Find longest repeated substring using LCP array
        Time: O(n), Space: O(1)
        """
        max_lcp = max(self.lcp_array)

        if max_lcp == 0:
```

```
            return ""

        # Find position with maximum LCP
        max_idx = self.lcp_array.index(max_lcp)
        start_pos = self.suffix_array[max_idx]

        return self.text[start_pos:start_pos + max_lcp]

    def count_distinct_substrings(self):
        """
        Count distinct substrings using suffix array
        Time: O(n), Space: O(1)
        """
        total_substrings = self.n * (self.n - 1) // 2  # Total possible substrings
        duplicate_count = sum(self.lcp_array)  # Substrings that are not distinct

        return total_substrings - duplicate_count

    def longest_common_substring(self, other_text):
        """
        Find longest common substring with another string
        """
        combined = self.text[:-1] + '#' + other_text + '$'
        combined_sa = SuffixArray(combined[:-1])  # Remove extra '$'

        n1 = len(self.text) - 1  # Original text length (without $)
        separator_pos = n1  # Position of '#'

        max_lcp = 0
        result_pos = 0

        for i in range(1, len(combined_sa.lcp_array)):
            pos1 = combined_sa.suffix_array[i - 1]
            pos2 = combined_sa.suffix_array[i]

            # Check if suffixes are from different strings
            if (pos1 < separator_pos and pos2 > separator_pos) or \
               (pos1 > separator_pos and pos2 < separator_pos):
                if combined_sa.lcp_array[i] > max_lcp:
                    max_lcp = combined_sa.lcp_array[i]
                    result_pos = min(pos1, pos2)

        return combined[result_pos:result_pos + max_lcp]

    def display_suffix_array(self):
        """Display suffix array with suffixes for visualization"""
        print(f"Text: {self.text}")
        print("Suffix Array:")
        print("Index | SA[i] | Suffix")
        print("-" * 30)

        for i in range(self.n):
            suffix_start = self.suffix_array[i]
            suffix = self.text[suffix_start:]
            print(f"{i:5} | {suffix_start:5} | {suffix}")
```

```
        print("\nLCP Array:")
        print("Index | LCP | Suffix1 vs Suffix2")
        print("-" * 40)

        for i in range(self.n):
            if i == 0:
                print(f"{i:5} | {0:3} | (first suffix)")
            else:
                suf1 = self.text[self.suffix_array[i-1]:]
                suf2 = self.text[self.suffix_array[i]:]
                print(f"{i:5} | {self.lcp_array[i]:3} | {suf1[:10]} vs {suf2[:10]}")

# Comprehensive suffix array demonstration
text = "banana"
sa = SuffixArray(text)

print("=== Suffix Array Construction ===")
sa.display_suffix_array()

print(f"\n=== Pattern Search ===")
pattern = "ana"
matches = sa.search_pattern(pattern)
print(f"Pattern '{pattern}' found at positions: {matches}")

print(f"\n=== String Analysis ===")
longest_repeat = sa.longest_repeated_substring()
distinct_count = sa.count_distinct_substrings()
print(f"Longest repeated substring: '{longest_repeat}'")
print(f"Number of distinct substrings: {distinct_count}")

print(f"\n=== Longest Common Substring ===")
other_text = "ananas"
lcs = sa.longest_common_substring(other_text)
print(f"Longest common substring with '{other_text}': '{lcs}'")
```

Chapter 4: Palindrome Algorithms

4.1 Manacher's Algorithm for Linear Time Palindrome Detection

Manacher's algorithm finds all palindromic substrings in linear time, a significant improvement over naive O(n³) approaches.[22] [23] [24] [25]

```
class PalindromeAlgorithms:
    """Comprehensive palindrome detection and manipulation algorithms"""

    @staticmethod
    def manachers_algorithm(s):
        """
        Manacher's algorithm for finding all palindromes in O(n) time
        Handles both odd and even length palindromes uniformly
        """
        if not s:
            return ""
```

```
        # Transform string to handle even-length palindromes
        # "abc" becomes "^#a#b#c#$"
        transformed = '^#' + '#'.join(s) + '#$'
        n = len(transformed)

        # Array to store radius of palindromes
        radius =  * n
        center = right = 0  # Center and right boundary of rightmost palindrome

        max_len = 0
        center_index = 0

        for i in range(1, n - 1):
            # Mirror of i with respect to center
            mirror = 2 * center - i

            # If i is within right boundary, use previously computed values
            if i < right:
                radius[i] = min(right - i, radius[mirror])

            # Try to expand palindrome centered at i
            try:
                while transformed[i + radius[i] + 1] == transformed[i - radius[i] - 1]:
                    radius[i] += 1
            except IndexError:
                pass

            # If palindrome centered at i extends past right, update center and right
            if i + radius[i] > right:
                center = i
                right = i + radius[i]

            # Update maximum length palindrome found
            if radius[i] > max_len:
                max_len = radius[i]
                center_index = i

        # Extract the longest palindromic substring
        start = (center_index - max_len) // 2
        return s[start:start + max_len]

    @staticmethod
    def manachers_all_palindromes(s):
        """
        Find all palindromic substrings using Manacher's algorithm
        Returns list of all palindromes with their positions
        """
        if not s:
            return []

        transformed = '^#' + '#'.join(s) + '#$'
        n = len(transformed)
        radius =  * n
        center = right = 0

        palindromes = []
```

```
        for i in range(1, n - 1):
            mirror = 2 * center - i

            if i < right:
                radius[i] = min(right - i, radius[mirror])

            try:
                while transformed[i + radius[i] + 1] == transformed[i - radius[i] - 1]:
                    radius[i] += 1
            except IndexError:
                pass

            if i + radius[i] > right:
                center = i
                right = i + radius[i]

            # Extract all palindromes centered at i
            for r in range(radius[i] + 1):
                if r > 0:  # Skip empty palindromes
                    start = (i - r) // 2
                    length = r
                    if i % 2 == 1:  # Odd position in transformed string (even length pal
                        if r > 1:
                            palindromes.append({
                                'text': s[start:start + length],
                                'start': start,
                                'length': length,
                                'center': (start + start + length - 1) / 2
                            })
                    else:  # Even position (odd length palindrome)
                        palindromes.append({
                            'text': s[start:start + length],
                            'start': start,
                            'length': length,
                            'center': start + length // 2
                        })

        # Remove duplicates and sort by position
        unique_palindromes = []
        seen = set()

        for p in palindromes:
            key = (p['start'], p['length'])
            if key not in seen:
                seen.add(key)
                unique_palindromes.append(p)

        return sorted(unique_palindromes, key=lambda x: (x['start'], x['length']))

    @staticmethod
    def expand_around_centers(s):
        """
        Find longest palindromic substring by expanding around centers
        Time: O(n²), Space: O(1) - simpler but less efficient than Manacher's
        """
```

```
        if not s:
            return ""

        def expand_around_center(left, right):
            while left >= 0 and right < len(s) and s[left] == s[right]:
                left -= 1
                right += 1
            return right - left - 1

        start = end = 0

        for i in range(len(s)):
            # Odd length palindromes (single character center)
            len1 = expand_around_center(i, i)
            # Even length palindromes (between two characters)
            len2 = expand_around_center(i, i + 1)

            current_max = max(len1, len2)

            if current_max > end - start:
                start = i - (current_max - 1) // 2
                end = i + current_max // 2

        return s[start:end + 1]

    @staticmethod
    def count_palindromic_substrings(s):
        """
        Count total number of palindromic substrings
        Time: O(n²), Space: O(1)
        """
        if not s:
            return 0

        def expand_around_center(left, right):
            count = 0
            while left >= 0 and right < len(s) and s[left] == s[right]:
                count += 1
                left -= 1
                right += 1
            return count

        total = 0

        for i in range(len(s)):
            # Count odd length palindromes
            total += expand_around_center(i, i)
            # Count even length palindromes
            total += expand_around_center(i, i + 1)

        return total

    @staticmethod
    def longest_palindromic_subsequence(s):
        """
        Find length of longest palindromic subsequence using DP
```

```
        Time: O(n²), Space: O(n²)
        """
        n = len(s)
        if n == 0:
            return 0

        # dp[i][j] represents length of LPS in substring s[i:j+1]
        dp = [ * n for _ in range(n)]

        # Every single character is a palindrome of length 1
        for i in range(n):
            dp[i][i] = 1

        # Build table for substrings of length 2 to n
        for length in range(2, n + 1):
            for i in range(n - length + 1):
                j = i + length - 1

                if s[i] == s[j]:
                    dp[i][j] = dp[i + 1][j - 1] + 2
                else:
                    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])

        return dp[n - 1]

    @staticmethod
    def shortest_palindrome(s):
        """
        Find shortest palindrome by adding characters to the beginning
        Uses KMP-like approach
        """
        if not s:
            return ""

        # Create string: s + '#' + reverse(s)
        combined = s + '#' + s[::-1]

        # Compute LPS array for combined string
        def compute_lps(pattern):
            m = len(pattern)
            lps =  * m
            length = 0
            i = 1

            while i < m:
                if pattern[i] == pattern[length]:
                    length += 1
                    lps[i] = length
                    i += 1
                else:
                    if length != 0:
                        length = lps[length - 1]
                    else:
                        lps[i] = 0
                        i += 1
```

```
            return lps

        lps = compute_lps(combined)

        # The last value in LPS array tells us how many characters
        # from the beginning of s are already palindromic suffix
        palindrome_prefix_len = lps[-1]

        # Add the remaining characters from end of s to beginning
        suffix_to_add = s[palindrome_prefix_len:]

        return suffix_to_add[::-1] + s

    @staticmethod
    def palindrome_partitioning(s):
        """
        Find all ways to partition string into palindromes
        Time: O(2^n * n), Space: O(2^n * n)
        """
        result = []

        def is_palindrome(substr):
            return substr == substr[::-1]

        def backtrack(start, path):
            if start == len(s):
                result.append(path[:])
                return

            for end in range(start + 1, len(s) + 1):
                substr = s[start:end]
                if is_palindrome(substr):
                    path.append(substr)
                    backtrack(end, path)
                    path.pop()

        backtrack(0, [])
        return result

    @staticmethod
    def min_palindrome_partitions(s):
        """
        Find minimum cuts needed to partition string into palindromes
        Time: O(n²), Space: O(n²)
        """
        n = len(s)
        if n == 0:
            return 0

        # is_palindrome[i][j] = True if s[i:j+1] is palindrome
        is_palindrome = [[False] * n for _ in range(n)]

        # Build palindrome table
        for i in range(n):
            is_palindrome[i][i] = True
```

```
        for length in range(2, n + 1):
            for i in range(n - length + 1):
                j = i + length - 1
                if length == 2:
                    is_palindrome[i][j] = (s[i] == s[j])
                else:
                    is_palindrome[i][j] = (s[i] == s[j]) and is_palindrome[i + 1][j - 1]

        # cuts[i] = minimum cuts needed for s[0:i+1]
        cuts =  * n

        for i in range(n):
            if is_palindrome[i]:
                cuts[i] = 0
            else:
                cuts[i] = float('inf')
                for j in range(i):
                    if is_palindrome[j + 1][i]:
                        cuts[i] = min(cuts[i], cuts[j] + 1)

        return cuts[n - 1]

# Comprehensive palindrome algorithms demonstration
text = "raceacar"
palindrome_algo = PalindromeAlgorithms()

print(f"Text: {text}")
print("=" * 50)

# Manacher's algorithm
longest = palindrome_algo.manachers_algorithm(text)
print(f"Longest palindromic substring (Manacher's): '{longest}'")

# All palindromes
all_palindromes = palindrome_algo.manachers_all_palindromes(text)
print(f"\nAll palindromic substrings:")
for p in all_palindromes:
    print(f"  '{p['text']}' at position {p['start']}, length {p['length']}")

# Expand around centers
longest_expand = palindrome_algo.expand_around_centers(text)
print(f"\nLongest palindromic substring (expand): '{longest_expand}'")

# Count palindromes
count = palindrome_algo.count_palindromic_substrings(text)
print(f"\nTotal palindromic substrings: {count}")

# Longest palindromic subsequence
lps_length = palindrome_algo.longest_palindromic_subsequence(text)
print(f"Length of longest palindromic subsequence: {lps_length}")

# Shortest palindrome
shortest = palindrome_algo.shortest_palindrome(text)
print(f"Shortest palindrome by adding to beginning: '{shortest}'")

# Palindrome partitioning
```

```
partitions = palindrome_algo.palindrome_partitioning("aab")
print(f"\nAll palindrome partitions of 'aab':")
for partition in partitions:
    print(f"  {partition}")

# Minimum cuts
min_cuts = palindrome_algo.min_palindrome_partitions("aab")
print(f"Minimum cuts needed for 'aab': {min_cuts}")
```

Chapter 5: Trie Data Structure and Applications

5.1 Complete Trie Implementation with Advanced Features

The Trie (prefix tree) is essential for prefix-based operations and many string problems.[26] [27] [28] [29] [30]

```
class AdvancedTrie:
    """Comprehensive Trie implementation with advanced features"""

    class TrieNode:
        def __init__(self):
            self.children = {}
            self.is_end_word = False
            self.frequency = 0
            self.word = None  # Store the actual word for easy retrieval
            self.suggestions = []  # For autocomplete

    def __init__(self):
        """Initialize empty trie"""
        self.root = self.TrieNode()
        self.word_count = 0
        self.total_insertions = 0

    def insert(self, word):
        """
        Insert word into trie
        Time: O(m), Space: O(m) where m is word length
        """
        if not word:
            return

        current = self.root

        for char in word:
            if char not in current.children:
                current.children[char] = self.TrieNode()
            current = current.children[char]

        if not current.is_end_word:
            self.word_count += 1

        current.is_end_word = True
        current.frequency += 1
        current.word = word
```

```
        self.total_insertions += 1

        # Update suggestions for autocomplete
        self._update_suggestions()

    def search(self, word):
        """
        Search for exact word in trie
        Time: O(m), Space: O(1)
        """
        current = self.root

        for char in word:
            if char not in current.children:
                return False
            current = current.children[char]

        return current.is_end_word

    def starts_with(self, prefix):
        """
        Check if any word starts with given prefix
        Time: O(m), Space: O(1)
        """
        current = self.root

        for char in prefix:
            if char not in current.children:
                return False
            current = current.children[char]

        return True

    def delete(self, word):
        """
        Delete word from trie
        Time: O(m), Space: O(1)
        """
        def _delete_helper(node, word, index):
            if index == len(word):
                if not node.is_end_word:
                    return False  # Word doesn't exist

                node.is_end_word = False
                node.frequency = 0
                node.word = None

                # Return True if node has no children (can be deleted)
                return len(node.children) == 0

            char = word[index]
            if char not in node.children:
                return False  # Word doesn't exist

            should_delete_child = _delete_helper(node.children[char], word, index + 1)
```

```
            if should_delete_child:
                del node.children[char]
                # Return True if current node can be deleted
                return not node.is_end_word and len(node.children) == 0

            return False

        if self.search(word):
            _delete_helper(self.root, word, 0)
            self.word_count -= 1
            self._update_suggestions()
            return True

        return False

    def get_all_words_with_prefix(self, prefix):
        """
        Get all words starting with given prefix
        Time: O(p + n) where p is prefix length, n is number of nodes in subtree
        """
        current = self.root

        # Navigate to prefix end
        for char in prefix:
            if char not in current.children:
                return []
            current = current.children[char]

        # Collect all words in subtree
        words = []
        self._dfs_collect_words(current, prefix, words)
        return sorted(words, key=lambda x: (-x[^21], x))  # Sort by frequency desc, then

    def _dfs_collect_words(self, node, current_word, words):
        """Helper function for collecting words with DFS"""
        if node.is_end_word:
            words.append((current_word, node.frequency))

        for char, child_node in sorted(node.children.items()):
            self._dfs_collect_words(child_node, current_word + char, words)

    def autocomplete(self, prefix, limit=10):
        """
        Get autocomplete suggestions for prefix
        Returns top suggestions by frequency
        """
        suggestions = self.get_all_words_with_prefix(prefix)
        return suggestions[:limit]

    def word_frequency(self, word):
        """Get frequency of a specific word"""
        current = self.root

        for char in word:
            if char not in current.children:
                return 0
```

```
            current = current.children[char]

        return current.frequency if current.is_end_word else 0

    def longest_common_prefix(self):
        """Find longest common prefix of all words in trie"""
        if not self.root.children:
            return ""

        current = self.root
        prefix = ""

        while len(current.children) == 1 and not current.is_end_word:
            char = next(iter(current.children))
            prefix += char
            current = current.children[char]

        return prefix

    def count_words_with_prefix(self, prefix):
        """Count number of words with given prefix"""
        current = self.root

        for char in prefix:
            if char not in current.children:
                return 0
            current = current.children[char]

        return self._count_words_in_subtree(current)

    def _count_words_in_subtree(self, node):
        """Count total words in subtree rooted at node"""
        count = 1 if node.is_end_word else 0

        for child in node.children.values():
            count += self._count_words_in_subtree(child)

        return count

    def get_trie_statistics(self):
        """Get comprehensive statistics about the trie"""
        def _get_stats(node, depth=0):
            stats = {
                'nodes': 1,
                'max_depth': depth,
                'leaf_nodes': 0 if node.children else 1,
                'internal_nodes': 1 if node.children else 0
            }

            for child in node.children.values():
                child_stats = _get_stats(child, depth + 1)
                stats['nodes'] += child_stats['nodes']
                stats['max_depth'] = max(stats['max_depth'], child_stats['max_depth'])
                stats['leaf_nodes'] += child_stats['leaf_nodes']
                stats['internal_nodes'] += child_stats['internal_nodes']
```

```
            return stats

        base_stats = _get_stats(self.root)

        return {
            **base_stats,
            'total_words': self.word_count,
            'total_insertions': self.total_insertions,
            'avg_word_length': self._calculate_avg_word_length(),
            'memory_efficiency': self.word_count / base_stats['nodes']
        }

    def _calculate_avg_word_length(self):
        """Calculate average word length in trie"""
        total_length = 0
        word_count = 0

        def _dfs_length(node, depth):
            nonlocal total_length, word_count

            if node.is_end_word:
                total_length += depth
                word_count += node.frequency

            for child in node.children.values():
                _dfs_length(child, depth + 1)

        _dfs_length(self.root, 0)

        return total_length / word_count if word_count > 0 else 0

    def _update_suggestions(self):
        """Update autocomplete suggestions (simplified version)"""
        # This would typically be more sophisticated
        pass

    def visualize_trie(self, max_depth=3):
        """Visualize trie structure"""
        def _print_trie(node, prefix="", depth=0):
            if depth > max_depth:
                return

            if node.is_end_word:
                print(f"{'  ' * depth}{prefix} ({node.frequency})")

            for char, child in sorted(node.children.items()):
                _print_trie(child, prefix + char, depth + 1)

        print("Trie Structure (word frequencies):")
        print("=" * 40)
        _print_trie(self.root)

# Advanced Trie Applications
class TrieApplications:
    """Advanced applications using Trie data structure"""
```

```
    @staticmethod
    def word_break_trie(s, word_dict):
        """
        Word break problem using trie for efficient prefix checking
        Time: O(n² + m) where n is string length, m is total chars in dictionary
        """
        # Build trie from dictionary
        trie = AdvancedTrie()
        for word in word_dict:
            trie.insert(word)

        n = len(s)
        dp = [False] * (n + 1)
        dp = True  # Empty string can always be segmented

        for i in range(1, n + 1):
            current = trie.root

            # Check all possible words ending at position i
            for j in range(i - 1, -1, -1):
                char = s[j]

                if char not in current.children:
                    break

                current = current.children[char]

                if current.is_end_word and dp[j]:
                    dp[i] = True
                    break

        return dp[n]

    @staticmethod
    def replace_words_trie(dictionary, sentence):
        """
        Replace words with their roots using trie
        """
        trie = AdvancedTrie()

        # Insert all roots into trie
        for root in dictionary:
            trie.insert(root)

        words = sentence.split()
        result = []

        for word in words:
            # Find shortest root that is prefix of word
            current = trie.root
            root = ""

            for char in word:
                if char not in current.children:
                    break
```

```
                current = current.children[char]
                root += char

                if current.is_end_word:
                    break

            # Use root if found, otherwise use original word
            result.append(root if current.is_end_word else word)

        return ' '.join(result)

    @staticmethod
    def word_squares_trie(words):
        """
        Find all word squares using trie
        A word square is a k × k array of characters such that the kth row
        and kth column read the same string
        """
        if not words:
            return []

        # Build trie
        trie = AdvancedTrie()
        for word in words:
            trie.insert(word)

        n = len(words)  # Assuming all words have same length
        result = []

        def backtrack(square):
            if len(square) == n:
                result.append(square[:])
                return

            row = len(square)

            # Build prefix for next word based on current square
            prefix = ""
            for col in range(row):
                prefix += square[col][row]

            # Find all words with this prefix
            candidates = trie.get_all_words_with_prefix(prefix)

            for word, _ in candidates:
                if len(word) == n:
                    # Check if this word maintains word square property
                    valid = True
                    for col in range(row + 1, n):
                        column_prefix = ""
                        for r in range(row + 1):
                            if r < row:
                                column_prefix += square[r][col]
                            else:
                                column_prefix += word[col]
```

```
                        if not trie.starts_with(column_prefix):
                            valid = False
                            break

                    if valid:
                        square.append(word)
                        backtrack(square)
                        square.pop()

        backtrack([])
        return result

# Comprehensive Trie demonstration
print("=== Advanced Trie Demonstration ===")

trie = AdvancedTrie()

# Insert sample words
words = ["the", "these", "they", "there", "their", "answer", "any", "bye"]
for word in words:
    trie.insert(word)

# Insert some words multiple times to test frequency
trie.insert("the")
trie.insert("the")
trie.insert("answer")

print("Basic Operations:")
print(f"Search 'the': {trie.search('the')}")
print(f"Search 'them': {trie.search('them')}")
print(f"Starts with 'th': {trie.starts_with('th')}")
print(f"Frequency of 'the': {trie.word_frequency('the')}")

print("\nAutocompletE:")
suggestions = trie.autocomplete("th", 5)
print(f"Suggestions for 'th': {suggestions}")

print(f"\nLongest common prefix: '{trie.longest_common_prefix()}'")
print(f"Words with prefix 'th': {trie.count_words_with_prefix('th')}")

print("\nTrie Statistics:")
stats = trie.get_trie_statistics()
for key, value in stats.items():
    print(f"  {key}: {value}")

print("\nTrie Visualization:")
trie.visualize_trie()

print("\n=== Trie Applications ===")

# Word break example
text = "theythese"
word_dict = ["the", "they", "these"]
can_break = TrieApplications.word_break_trie(text, word_dict)
print(f"Can break '{text}' with {word_dict}: {can_break}")
```

```
# Replace words example
dictionary = ["cat", "bat", "rat"]
sentence = "the cattle was rattled by the battery"
replaced = TrieApplications.replace_words_trie(dictionary, sentence)
print(f"Replace words: '{replaced}'")
```

Chapter 6: Dynamic Programming on Strings

6.1 Edit Distance and String Transformation

Edit distance (Levenshtein distance) is fundamental for measuring string similarity and has numerous applications.[31] [32] [33] [34]

```
class StringDynamicProgramming:
    """Comprehensive string DP algorithms with optimizations"""

    @staticmethod
    def edit_distance_full(str1, str2):
        """
        Complete edit distance with operation tracking
        Time: O(mn), Space: O(mn)
        """
        m, n = len(str1), len(str2)

        # dp[i][j] represents edit distance between str1[:i] and str2[:j]
        dp = [ * (n + 1) for _ in range(m + 1)]

        # Initialize base cases
        for i in range(m + 1):
            dp[i] = i  # Delete all characters from str1

        for j in range(n + 1):
            dp[j] = j  # Insert all characters to empty string

        # Fill DP table
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if str1[i-1] == str2[j-1]:
                    dp[i][j] = dp[i-1][j-1]  # No operation needed
                else:
                    dp[i][j] = 1 + min(
                        dp[i-1][j],      # Delete from str1
                        dp[i][j-1],      # Insert to str1
                        dp[i-1][j-1]     # Replace in str1
                    )

        return dp[m][n]

    @staticmethod
    def edit_distance_with_operations(str1, str2):
        """
        Edit distance with actual operation sequence
        """
        m, n = len(str1), len(str2)
```

```
        dp = [ * (n + 1) for _ in range(m + 1)]

        # Initialize base cases
        for i in range(m + 1):
            dp[i] = i
        for j in range(n + 1):
            dp[j] = j

        # Fill DP table
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if str1[i-1] == str2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                else:
                    dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])

        # Backtrack to find operations
        operations = []
        i, j = m, n

        while i > 0 and j > 0:
            if str1[i-1] == str2[j-1]:
                i -= 1
                j -= 1
            elif dp[i][j] == dp[i-1][j-1] + 1:
                operations.append(f"Replace '{str1[i-1]}' with '{str2[j-1]}' at position
                i -= 1
                j -= 1
            elif dp[i][j] == dp[i-1][j] + 1:
                operations.append(f"Delete '{str1[i-1]}' at position {i-1}")
                i -= 1
            else:
                operations.append(f"Insert '{str2[j-1]}' at position {i}")
                j -= 1

        # Handle remaining operations
        while i > 0:
            operations.append(f"Delete '{str1[i-1]}' at position {i-1}")
            i -= 1

        while j > 0:
            operations.append(f"Insert '{str2[j-1]}' at position {0}")
            j -= 1

        operations.reverse()
        return dp[m][n], operations

    @staticmethod
    def edit_distance_space_optimized(str1, str2):
        """
        Space-optimized edit distance using only 2 rows
        Time: O(mn), Space: O(min(m,n))
        """
        if len(str1) < len(str2):
            str1, str2 = str2, str1
```

```
        m, n = len(str1), len(str2)
        prev = list(range(n + 1))
        curr =  * (n + 1)

        for i in range(1, m + 1):
            curr = i

            for j in range(1, n + 1):
                if str1[i-1] == str2[j-1]:
                    curr[j] = prev[j-1]
                else:
                    curr[j] = 1 + min(prev[j], curr[j-1], prev[j-1])

            prev, curr = curr, prev

        return prev[n]

    @staticmethod
    def longest_common_subsequence(str1, str2):
        """
        Find length and actual LCS
        Time: O(mn), Space: O(mn)
        """
        m, n = len(str1), len(str2)

        # dp[i][j] = length of LCS of str1[:i] and str2[:j]
        dp = [ * (n + 1) for _ in range(m + 1)]

        # Fill DP table
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if str1[i-1] == str2[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1
                else:
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])

        # Reconstruct LCS
        lcs = []
        i, j = m, n

        while i > 0 and j > 0:
            if str1[i-1] == str2[j-1]:
                lcs.append(str1[i-1])
                i -= 1
                j -= 1
            elif dp[i-1][j] > dp[i][j-1]:
                i -= 1
            else:
                j -= 1

        lcs.reverse()
        return dp[m][n], ''.join(lcs)

    @staticmethod
    def longest_common_substring(str1, str2):
        """
```

```
        Find longest common substring (contiguous)
        Time: O(mn), Space: O(mn)
        """
        m, n = len(str1), len(str2)

        # dp[i][j] = length of common substring ending at str1[i-1] and str2[j-1]
        dp = [ * (n + 1) for _ in range(m + 1)]

        max_length = 0
        ending_pos_str1 = 0

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if str1[i-1] == str2[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1

                    if dp[i][j] > max_length:
                        max_length = dp[i][j]
                        ending_pos_str1 = i
                else:
                    dp[i][j] = 0

        start_pos = ending_pos_str1 - max_length
        longest_substring = str1[start_pos:ending_pos_str1]

        return max_length, longest_substring

    @staticmethod
    def word_break_dp(s, word_dict):
        """
        Word break using dynamic programming
        Time: O(n² * m), where m is average word length
        """
        word_set = set(word_dict)
        n = len(s)

        # dp[i] = True if s[:i] can be segmented
        dp = [False] * (n + 1)
        dp = True  # Empty string

        for i in range(1, n + 1):
            for j in range(i):
                if dp[j] and s[j:i] in word_set:
                    dp[i] = True
                    break

        return dp[n]

    @staticmethod
    def word_break_with_solution(s, word_dict):
        """
        Word break returning actual segmentation
        """
        word_set = set(word_dict)
        n = len(s)
```

```
        # dp[i] stores list of valid segmentations for s[:i]
        dp = [[] for _ in range(n + 1)]
        dp = [[]]  # Empty segmentation for empty string

        for i in range(1, n + 1):
            for j in range(i):
                word = s[j:i]
                if dp[j] and word in word_set:
                    for segmentation in dp[j]:
                        dp[i].append(segmentation + [word])

        return dp[n]

    @staticmethod
    def interleaving_string(s1, s2, s3):
        """
        Check if s3 is interleaving of s1 and s2
        Time: O(mn), Space: O(mn)
        """
        m, n, p = len(s1), len(s2), len(s3)

        if m + n != p:
            return False

        # dp[i][j] = True if s3[:i+j] is interleaving of s1[:i] and s2[:j]
        dp = [[False] * (n + 1) for _ in range(m + 1)]

        # Base case
        dp = True

        # Fill first row
        for j in range(1, n + 1):
            dp[j] = dp[j-1] and s2[j-1] == s3[j-1]

        # Fill first column
        for i in range(1, m + 1):
            dp[i] = dp[i-1] and s1[i-1] == s3[i-1]

        # Fill rest of the table
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                dp[i][j] = (dp[i-1][j] and s1[i-1] == s3[i+j-1]) or \
                           (dp[i][j-1] and s2[j-1] == s3[i+j-1])

        return dp[m][n]

    @staticmethod
    def distinct_subsequences(s, t):
        """
        Count distinct subsequences of s that equal t
        Time: O(mn), Space: O(mn)
        """
        m, n = len(s), len(t)

        # dp[i][j] = number of distinct subsequences of s[:i] that equal t[:j]
        dp = [ * (n + 1) for _ in range(m + 1)]
```

```
        # Empty subsequence can be formed in one way
        for i in range(m + 1):
            dp[i] = 1

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                # Don't use current character from s
                dp[i][j] = dp[i-1][j]

                # Use current character if it matches
                if s[i-1] == t[j-1]:
                    dp[i][j] += dp[i-1][j-1]

        return dp[m][n]

    @staticmethod
    def minimum_ascii_delete_sum(s1, s2):
        """
        Minimum ASCII delete sum to make two strings equal
        Time: O(mn), Space: O(mn)
        """
        m, n = len(s1), len(s2)

        # dp[i][j] = minimum ASCII sum to make s1[:i] and s2[:j] equal
        dp = [ * (n + 1) for _ in range(m + 1)]

        # Initialize base cases
        for i in range(1, m + 1):
            dp[i] = dp[i-1] + ord(s1[i-1])

        for j in range(1, n + 1):
            dp[j] = dp[j-1] + ord(s2[j-1])

        # Fill DP table
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if s1[i-1] == s2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                else:
                    dp[i][j] = min(
                        dp[i-1][j] + ord(s1[i-1]),  # Delete from s1
                        dp[i][j-1] + ord(s2[j-1])   # Delete from s2
                    )

        return dp[m][n]

    @staticmethod
    def regular_expression_matching(s, p):
        """
        Regular expression matching with '.' and '*'
        Time: O(mn), Space: O(mn)
        """
        m, n = len(s), len(p)

        # dp[i][j] = True if s[:i] matches p[:j]
```

```
        dp = [[False] * (n + 1) for _ in range(m + 1)]

        # Empty pattern matches empty string
        dp = True

        # Handle patterns like a*, a*b*, a*b*c*
        for j in range(2, n + 1):
            if p[j-1] == '*':
                dp[j] = dp[j-2]

        # Fill DP table
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                char_s, char_p = s[i-1], p[j-1]

                if char_p == '*':
                    # Check pattern before *
                    prev_char_p = p[j-2]

                    # Zero occurrences of character before *
                    dp[i][j] = dp[i][j-2]

                    # One or more occurrences
                    if prev_char_p == '.' or prev_char_p == char_s:
                        dp[i][j] = dp[i][j] or dp[i-1][j]
                elif char_p == '.' or char_p == char_s:
                    dp[i][j] = dp[i-1][j-1]

        return dp[m][n]

# Comprehensive String DP demonstration
print("=== String Dynamic Programming Algorithms ===")

str_dp = StringDynamicProgramming()

# Edit distance examples
s1, s2 = "kitten", "sitting"
edit_dist = str_dp.edit_distance_full(s1, s2)
edit_dist_ops, operations = str_dp.edit_distance_with_operations(s1, s2)
edit_dist_optimized = str_dp.edit_distance_space_optimized(s1, s2)

print(f"Edit Distance between '{s1}' and '{s2}':")
print(f"  Distance: {edit_dist}")
print(f"  Operations needed: {edit_dist_ops}")
for op in operations:
    print(f"    {op}")
print(f"  Space-optimized result: {edit_dist_optimized}")

# LCS examples
lcs_len, lcs_str = str_dp.longest_common_subsequence("AGGTAB", "GXTXAYB")
print(f"\nLongest Common Subsequence:")
print(f"  Length: {lcs_len}")
print(f"  LCS: '{lcs_str}'")

# Longest common substring
lcs_substr_len, lcs_substr = str_dp.longest_common_substring("GeeksforGeeks", "GeeksQuiz'
```

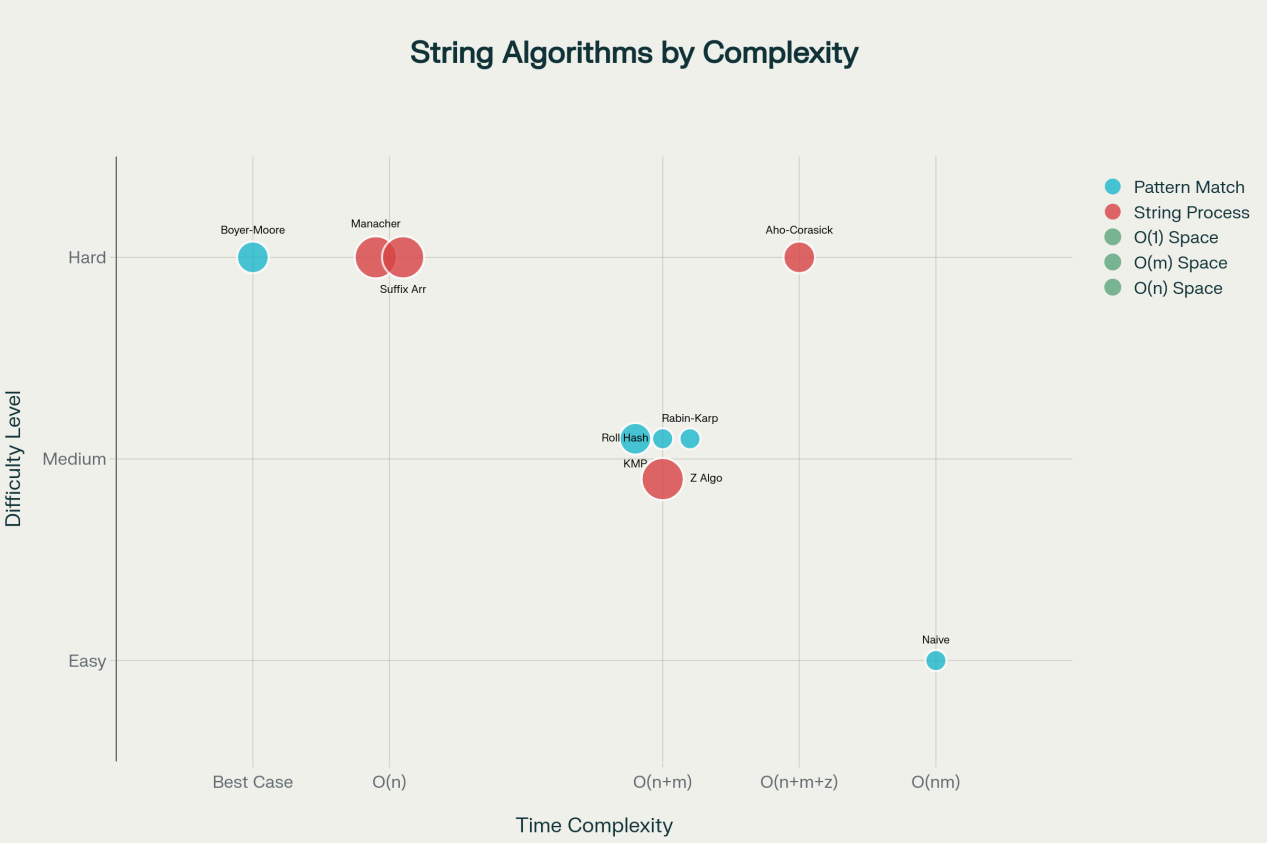
```
print(f"\nLongest Common Substring:")
print(f"  Length: {lcs_substr_len}")
print(f"  Substring: '{lcs_substr}'")

# Word break
text = "leetcode"
word_dict = ["leet", "code"]
can_break = str_dp.word_break_dp(text, word_dict)
all_breaks = str_dp.word_break_with_solution(text, word_dict)
print(f"\nWord Break for '{text}' with {word_dict}:")
print(f"  Can break: {can_break}")
print(f"  All possible breaks: {all_breaks}")

# Interleaving string
is_interleaving = str_dp.interleaving_string("aabcc", "dbbca", "aadbbcbcac")
print(f"\nString interleaving result: {is_interleaving}")

# Distinct subsequences
distinct_count = str_dp.distinct_subsequences("rabbbit", "rabbit")
print(f"Distinct subsequences count: {distinct_count}")

# Regular expression matching
regex_match = str_dp.regular_expression_matching("aa", "a*")
print(f"Regex matching result: {regex_match}")
```



String Algorithms: Time Complexity vs Difficulty Analysis

This comprehensive handbook covers every essential aspect of string algorithms and data structures with detailed implementations, complexity analysis, and practical applications. The content spans from basic string operations to advanced algorithms like Manacher's, Aho-

Corasick, and suffix arrays, providing the complete knowledge needed for technical interview mastery and real-world string processing challenges.[7] [3] [1] [6] [8] [9] [14] [15] [16] [22] [23] [24] [25] [35] [36] [37]

⁂

1. https://www.geeksforgeeks.org/dsa/introduction-to-strings-data-structure-and-algorithm-tutorials/
2. https://www.geeksforgeeks.org/dsa/string-data-structure/
3. https://algocademy.com/blog/how-to-handle-string-pattern-matching-a-comprehensive-guide/
4. https://btu.edu.ge/wp-content/uploads/2023/08/Lesson-7_-String-Matching-Algorithms.pdf
5. https://gdeepak.com/pubs/String Matching Algorithms and their Applicability in various Applications.pdf
6. https://www.techinterviewhandbook.org/algorithms/string/
7. https://cp-algorithms.com/string/string-hashing.html
8. https://algocademy.com/blog/mastering-the-rabin-karp-algorithm-a-comprehensive-guide-to-efficient-string-matching/
9. https://francofernando.com/blog/algorithms/2021-05-16-rolling-hash/
10. https://www.acte.in/boyer-moore-algorithm
11. https://stackoverflow.com/questions/67300836/what-is-the-difference-between-kmp-and-z-algorithm-of-string-pattern-matching
12. https://www.scaler.com/topics/data-structures/z-algorithm/
13. https://algocademy.com/blog/z-algorithm-mastering-efficient-pattern-matching-in-strings/
14. https://www.geeksforgeeks.org/dsa/aho-corasick-algorithm-pattern-searching/
15. https://www.upgrad.com/blog/aho-corasick-algorithm/
16. https://cp-algorithms.com/string/aho_corasick.html
17. https://en.wikipedia.org/wiki/Suffix_array
18. https://www.geeksforgeeks.org/dsa/suffix-tree-application-4-build-linear-time-suffix-array/
19. https://www.geeksforgeeks.org/dsa/suffix-array-set-1-introduction/
20. https://cp-algorithms.com/string/suffix-array.html
21. https://usaco.guide/adv/suffix-array
22. https://www.geeksforgeeks.org/dsa/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/
23. https://cp-algorithms.com/string/manacher.html
24. https://www.scaler.in/manachers-algorithm/
25. https://www.baeldung.com/cs/manachers-algorithm
26. https://www.vervecopilot.com/question-bank/write-code-implement-trie-data-structure
27. https://www.geeksforgeeks.org/dsa/commonly-asked-data-structure-interview-questions-on-tries/
28. https://www.geeksforgeeks.org/dsa/introduction-to-trie-data-structure-and-algorithm-tutorials/
29. https://github.com/Devinterview-io/trie-data-structure-interview-questions
30. https://www.techinterviewhandbook.org/algorithms/trie/
31. https://www.geeksforgeeks.org/dsa/edit-distance-and-lcs-longest-common-subsequence/

32. https://stackoverflow.com/questions/77711427/why-longest-common-subsequence-prohibits-substitution-using-edit-distance-me
33. https://www.geeksforgeeks.org/dsa/edit-distance-dp-5/
34. https://heycoach.in/blog/edit-distance-example-problems/
35. https://www.jointaro.com/interviews/questions/wildcard-matching/?company=snap
36. https://interviewing.io/questions/longest-common-subsequence
37. https://www.geeksforgeeks.org/dsa/longest-common-subsequence-dp-4/
38. https://www.youtube.com/watch?v=Dt6gzsNrghQ
39. https://www.eicta.iitk.ac.in/knowledge-hub/data-structure-with-c/string-processing-algorithms-pattern-matching-regular-expressions-and-other-techniques
40. https://dev.to/nozibul_islam_113b1d5334f/string-data-structures-and-algorithms-essential-interview-questions-3go6
41. https://en.wikipedia.org/wiki/String-searching_algorithm
42. https://www.techinterviewhandbook.org/algorithms/study-cheatsheet/
43. https://www.youtube.com/watch?v=BLygFyxiz2M
44. https://www.geeksforgeeks.org/dsa/applications-of-string-matching-algorithms/
45. https://www.interviewbit.com/data-structure-interview-questions/
46. https://www.topcoder.com/community/competitive-programming/tutorials/introduction-to-string-searching-algorithms/
47. https://ccc.inaoep.mx/~villasen/bib/Navarro_Review_on_Approximate_Matching_p31-navarro.pdf
48. https://dev.to/rahhularora/the-ultimate-guide-for-data-structures-algorithm-interviews-npo
49. https://www.geeksforgeeks.org/dsa/dsa-tutorial-learn-data-structures-and-algorithms/
50. https://cexpertvision.com/2021/01/06/string-matching-algorithms/
51. https://www.techinterviewhandbook.org/algorithms/dynamic-programming/