Strings Cheatsheet

Topic Overview

Strings in Java are sequences of characters, crucial for text processing, pattern matching, and algorithmic challenges. This cheatsheet provides detailed approaches for string-related problems.

Prerequisites

Arrays

List of Subtopics

- String Manipulation
- Pattern Matching
- Rabin-Karp Algorithm
- KMP (Knuth-Morris-Pratt) Algorithm
- Palindrome Check
- Longest Palindromic Substring (LPS)
- Z-Algorithm
- Manacher's Algorithm

Key Concepts Explained

- **String Manipulation**: Involves altering string content, such as extracting parts or reversing order.
- **Pattern Matching**: Locates a substring within a larger string using efficient methods.
- **Palindrome Check**: Verifies if a string is identical when reversed, ignoring non-essential characters.

Approaches to Solve Problems with Step-by-Step Algorithms

- **String Manipulation**:
  - **Algorithm**:
    1. Convert the string to a character array for index-based access.
    2. For substring extraction, identify start and end indices, then copy the range to a new string.
    3. For reversal, use two pointers from ends, swapping characters until they meet.
    4. For case conversion, iterate through characters, applying toUpperCase() or toLowerCase() transformations.
    5. Handle edge cases like empty or null strings.
  - **Context**: Essential for text preprocessing, with O(n) time for most operations.
- **Pattern Matching**:
  - **Algorithm**:
    1. Initialize a window of the pattern's length at the text's start.
    2. Compare each character of the window with the pattern.
    3. If a mismatch occurs, slide the window one position forward.
    4. Continue until the window reaches the text's end or a match is found.
    5. Return the starting index of the first match.
  - **Context**: Basic method with O(n*m) complexity, improved by advanced algorithms.
- **Rabin-Karp Algorithm**:
  - **Algorithm**:
    1. Choose a prime number and base (e.g., 101 and 256 for ASCII).
    2. Compute the hash of the pattern and the first window of text.
    3. Slide the window, updating the hash by removing the leftmost character and adding the next.
    4. Compare hashes; if equal, verify character-by-character to avoid collisions.
    5. Continue until a match is confirmed or the end is reached.
  - **Context**: Efficient for multiple pattern searches, with O(n+m) average time.
- **KMP Algorithm**:
  - **Algorithm**:
    1. Precompute the Longest Proper Prefix which is also Suffix (LPS) array for the pattern.
    2. Initialize text and pattern pointers at the start.

    3. Match characters; if a mismatch occurs, use LPS to shift the pattern pointer.
    4. Advance the text pointer and repeat matching.
    5. Return the text index where the pattern starts if a full match is found.
  - **Context**: Optimizes pattern matching to O(n+m) by avoiding backtracking.

- **Palindrome Check**:
  - **Algorithm**:
    1. Convert the string to lowercase and filter out non-alphanumeric characters.
    2. Set two pointers at the start and end of the cleaned string.
    3. Move pointers inward, comparing characters, skipping non-matches.
    4. Continue until pointers meet or cross, returning true if all match.
  - **Context**: Handles text validation, with O(n) time after cleaning.

- **Longest Palindromic Substring**:
  - **Algorithm**:
    1. Treat each character as a potential palindrome center.
    2. Expand outward from the center, checking symmetry for odd and even lengths.
    3. Track the longest palindrome's start and length during expansion.
    4. Repeat for all characters, updating the maximum length found.
  - **Context**: Solves string symmetry problems in O(nš) time.

- **Z-Algorithm**:
  - **Algorithm**:
    1. Concatenate pattern with text, adding a separator.
    2. Initialize a Z-array with zeros, and set the first value to the pattern length.
    3. Use two pointers (left, right) to maintain the current Z-box.
    4. For each position, compute the Z-value by comparing with the prefix, updating pointers.
    5. Continue until all positions are processed.
  - **Context**: Efficient for pattern matching, with O(n) time.

- **Manacher's Algorithm**:
  - **Algorithm**:
    1. Transform the string by inserting special characters between each character.
    2. Initialize a palindrome length array and center pointers.
    3. For each position, expand around the center using symmetry from the rightmost palindrome.
    4. Update the center and right boundary if a longer palindrome is found.
    5. Return the longest palindrome length after processing all positions.
  - **Context**: Computes LPS in O(n) time, handling odd/even cases.

Common LeetCode Problems with Approaches

- **Longest Palindromic Substring (5)**: Expand around each character as a center, checking palindrome symmetry, and track the longest substring.
- **Valid Palindrome (125)**: Clean the string and use two pointers to verify palindrome properties from both ends.
- **Implement strStr() (28)**: Use KMP by precomputing the LPS array, then match the pattern efficiently in the text.
- **Minimum Window Substring (76)**: Apply a sliding window with a frequency map, expanding and shrinking to find the smallest covering substring.
- **Longest Substring Without Repeating Characters (3)**: Maintain a sliding window, adjusting boundaries when characters repeat, to maximize length.
- **Repeated String Match (686)**: Repeatedly append the string, checking if the pattern is a substring, to determine the minimum repetitions.

Time & Space Complexities

- Access: O(1)
- Search: O(n) (naive), O(n+m) (KMP), O(n) avg (Rabin-Karp)
- Manacher's: O(n)
- Space: O(n) for auxiliary arrays, O(m) for pattern preprocessing

Important Tips & Tricks

- Use StringBuilder for mutable string operations in Java.
- Precompute LPS for KMP to enhance pattern matching efficiency.
- Handle edge cases like empty or single-character strings.
- Select a large prime modulus for Rabin-Karp to minimize collisions.
- Apply Manacher's for linear-time longest palindrome detection.