Master-Level Strings Plan

Phase 1 – String Basics & Manipulation

1. String creation, immutability, and `StringBuilder` in Java
2. Reversing a string (iterative & recursive)
3. Reversing words in a sentence (with/without extra space)
4. Changing case (upper/lower, toggle case)
5. Removing vowels, consonants, digits, spaces
6. Removing duplicate characters from a string
7. Counting characters, digits, vowels, and consonants
8. Check if two strings are anagrams
9. Check if strings are rotations of each other
10. Palindrome check (case-sensitive & case-insensitive)
11. Longest palindromic prefix & suffix
12. Lexicographic comparison and sorting of strings

13. Converting between `char[]` and `String` in Java

```
import java.util.*;

public class StringBasicsManipulation {

    // --- Phase 1: String Basics & Manipulation ---

    // Problem 1: String creation, immutability, and StringBuilder in Java
    public static String createStringWithStringBuilder(String[] words) {
        StringBuilder sb = new StringBuilder();
        for (String word : words) {
            sb.append(word).append(" ");
        }
        return sb.toString().trim();
    }

    // Problem 2: Reverse a string iteratively
    public static String reverseStringIterative(String s) {
        char[] arr = s.toCharArray();
        int left = 0, right = arr.length - 1;
        while (left < right) {
            char temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;
            left++;
            right--;
        }
        return new String(arr);
    }
```

```
// Problem 3: Reverse a string recursively
public static String reverseStringRecursive(String s) {
    if (s.isEmpty() || s.length() == 1) return s;
    return reverseStringRecursive(s.substring(1)) + s.charAt(0);
}

// Problem 4: Reverse words in a sentence (with extra space)
public static String reverseWordsExtraSpace(String s) {
    String[] words = s.trim().split("\\s+");
    StringBuilder sb = new StringBuilder();
    for (int i = words.length - 1; i >= 0; i--) {
        sb.append(words[i]).append(" ");
    }
    return sb.toString().trim();
}

// Problem 5: Reverse words in a sentence (in-place, O(1) extra space)
public static String reverseWordsInPlace(String s) {
    char[] arr = s.toCharArray();
    // Reverse entire string
    reverseArray(arr, 0, arr.length - 1);
    // Reverse each word
    int start = 0;
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == ' ') {
            reverseArray(arr, start, i - 1);
            start = i + 1;
        }
    }
    reverseArray(arr, start, arr.length - 1);
    return new String(arr);
}

private static void reverseArray(char[] arr, int start, int end) {
    while (start < end) {
        char temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

// Problem 6: Convert string to uppercase
public static String toUpperCase(String s) {
    return s.toUpperCase();
}
```

```
// Problem 7: Convert string to lowercase
public static String toLowerCase(String s) {
    return s.toLowerCase();
}

// Problem 8: Toggle case of each character
public static String toggleCase(String s) {
    char[] arr = s.toCharArray();
    for (int i = 0; i < arr.length; i++) {
        if (Character.isUpperCase(arr[i])) {
            arr[i] = Character.toLowerCase(arr[i]);
        } else if (Character.isLowerCase(arr[i])) {
            arr[i] = Character.toUpperCase(arr[i]);
        }
    }
    return new String(arr);
}

// Problem 9: Remove vowels from a string
public static String removeVowels(String s) {
    return s.replaceAll("[aeiouAEIOU]", "");
}

// Problem 10: Remove consonants from a string
public static String removeConsonants(String s) {
    return s.replaceAll("[^aeiouAEIOU\\s\\d]", "");
}

// Problem 11: Remove digits from a string
public static String removeDigits(String s) {
    return s.replaceAll("[0-9]", "");
}

// Problem 12: Remove spaces from a string
public static String removeSpaces(String s) {
    return s.replaceAll("\\s", "");
}

// Problem 13: Remove duplicate characters from a string (keep first occurrence)
public static String removeDuplicates(String s) {
    Set<Character> seen = new LinkedHashSet<>();
    StringBuilder sb = new StringBuilder();
    for (char c : s.toCharArray()) {
        if (seen.add(c)) {
            sb.append(c);
        }
    }
    return sb.toString();
}
```

```
// Problem 14: Count characters in a string
public static int countCharacters(String s) {
    return s.length();
}

// Problem 15: Count digits in a string
public static int countDigits(String s) {
    int count = 0;
    for (char c : s.toCharArray()) {
        if (Character.isDigit(c)) count++;
    }
    return count;
}

// Problem 16: Count vowels in a string
public static int countVowels(String s) {
    return s.toLowerCase().replaceAll("[^aeiou]", "").length();
}

// Problem 17: Count consonants in a string
public static int countConsonants(String s) {
    return s.toLowerCase().replaceAll("[aeiou\\s\\d]", "").length();
}

// Problem 18: Check if two strings are anagrams
public static boolean areAnagrams(String s1, String s2) {
    if (s1.length() != s2.length()) return false;
    int[] count = new int[26];
    for (int i = 0; i < s1.length(); i++) {
        count[s1.charAt(i) - 'a']++;
        count[s2.charAt(i) - 'a']--;
    }
    for (int c : count) {
        if (c != 0) return false;
    }
    return true;
}

// Problem 19: Check if two strings are rotations of each other
public static boolean areRotations(String s1, String s2) {
    if (s1.length() != s2.length() || s1.length() == 0) return false;
    return (s1 + s1).contains(s2);
}
```

```
// Problem 20: Check if a string is palindrome (case-sensitive)
  public static boolean isPalindromeCaseSensitive(String s) {
     int left = 0, right = s.length() - 1;
     while (left < right) {
        if (s.charAt(left++) != s.charAt(right--)) return false;
     }
     return true;
  }

  // Problem 21: Check if a string is palindrome (case-insensitive)
  public static boolean isPalindromeCaseInsensitive(String s) {
     s = s.toLowerCase();
     int left = 0, right = s.length() - 1;
     while (left < right) {
        if (s.charAt(left++) != s.charAt(right--)) return false;
     }
     return true;
  }

  // Problem 22: Find longest palindromic prefix
  public static String longestPalindromicPrefix(String s) {
     for (int i = s.length(); i >= 0; i--) {
        String prefix = s.substring(0, i);
        if (isPalindromeCaseSensitive(prefix)) return prefix;
     }
     return "";
  }

  // Problem 23: Find longest palindromic suffix
  public static String longestPalindromicSuffix(String s) {
     for (int i = 0; i <= s.length(); i++) {
        String suffix = s.substring(i);
        if (isPalindromeCaseSensitive(suffix)) return suffix;
     }
     return "";
  }

  // Problem 24: Lexicographic comparison of two strings
  public static int lexicographicCompare(String s1, String s2) {
     return s1.compareTo(s2);
  }

  // Problem 25: Sort an array of strings lexicographically
  public static String[] sortStringsLexicographically(String[] arr) {
     Arrays.sort(arr);
     return arr;
  }
```

```
// Problem 26: Convert char[] to String
public static String charArrayToString(char[] arr) {
    return new String(arr);
}

// Problem 27: Convert String to char[]
public static char[] stringToCharArray(String s) {
    return s.toCharArray();
}

// Problem 28: Reverse vowels in a string
public static String reverseVowels(String s) {
    char[] arr = s.toCharArray();
    int left = 0, right = arr.length - 1;
    while (left < right) {
        while (left < right && !isVowel(arr[left])) left++;
        while (left < right && !isVowel(arr[right])) right--;
        char temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
        left++;
        right--;
    }
    return new String(arr);
}

private static boolean isVowel(char c) {
    return "aeiouAEIOU".indexOf(c) != -1;
}

// Problem 29: Remove specific character from string
public static String removeSpecificChar(String s, char c) {
    return s.replace(String.valueOf(c), "");
}

// Problem 30: Check if string contains only digits
public static boolean isDigitOnly(String s) {
    return s.matches("\\d+");
}

// Problem 31: Check if string contains only letters
public static boolean isLetterOnly(String s) {
    return s.matches("[a-zA-Z]+");
}
```

```
    // Problem 32: Find frequency of each character in a string
    public static Map<Character, Integer> characterFrequency(String s) {
        Map<Character, Integer> freq = new HashMap<>();
        for (char c : s.toCharArray()) {
            freq.put(c, freq.getOrDefault(c, 0) + 1);
        }
        return freq;
    }

    // Problem 33: Check if two strings are anagrams ignoring spaces and case
    public static boolean areAnagramsIgnoreSpaceCase(String s1, String s2) {
        s1 = s1.replaceAll("\\s", "").toLowerCase();
        s2 = s2.replaceAll("\\s", "").toLowerCase();
        if (s1.length() != s2.length()) return false;
        int[] count = new int[26];
        for (int i = 0; i < s1.length(); i++) {
            count[s1.charAt(i) - 'a']++;
            count[s2.charAt(i) - 'a']--;
        }
        for (int c : count) {
            if (c != 0) return false;
        }
        return true;
    }

    // Problem 34: Find first non-repeating character
    public static char firstNonRepeatingChar(String s) {
        Map<Character, Integer> freq = new LinkedHashMap<>();
        for (char c : s.toCharArray()) {
            freq.put(c, freq.getOrDefault(c, 0) + 1);
        }
        for (Map.Entry<Character, Integer> entry : freq.entrySet()) {
            if (entry.getValue() == 1) return entry.getKey();
        }
        return '\0';
    }

    // Problem 35: Replace all occurrences of a substring
    public static String replaceSubstring(String s, String oldSub, String newSub) {
        return s.replace(oldSub, newSub);
    }
}
```

Phase 2 – Pattern Matching & Search

14. Naive pattern search algorithm
15. Knuth–Morris–Pratt (KMP) algorithm
16. Rabin–Karp algorithm (single & multiple patterns)
17. Z-algorithm for pattern matching
18. Finite automata-based pattern search
19. Boyer–Moore algorithm
20. Aho–Corasick algorithm (multiple pattern search)
21. Wildcard pattern matching (`?` and `*`)
22. Regular expression matching (DP-based)

```
import java.util.*;

public class PatternMatchingSearch {

    // --- Phase 2: Pattern Matching & Search ---

    // Problem 14: Naive pattern search algorithm
    public static List<Integer> naivePatternSearch(String text, String pattern) {
        List<Integer> result = new ArrayList<>();
        int n = text.length(), m = pattern.length();
        for (int i = 0; i <= n - m; i++) {
            int j;
            for (j = 0; j < m; j++) {
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    break;
                }
            }
            if (j == m) {
                result.add(i);
            }
        }
        return result;
    }
```

```
// Problem 15: Knuth–Morris–Pratt (KMP) algorithm
 public static List<Integer> kmpSearch(String text, String pattern) {
    List<Integer> result = new ArrayList<>();
    int[] lps = computeLPSArray(pattern);
    int n = text.length(), m = pattern.length();
    int i = 0, j = 0;
    while (i < n) {
       if (text.charAt(i) == pattern.charAt(j)) {
          i++;
          j++;
       }
       if (j == m) {
          result.add(i - j);
          j = lps[j - 1];
       } else if (i < n && text.charAt(i) != pattern.charAt(j)) {
          if (j != 0) {
             j = lps[j - 1];
          } else {
             i++;
          }
       }
    }
    return result;
 }

 private static int[] computeLPSArray(String pattern) {
    int m = pattern.length();
    int[] lps = new int[m];
    int len = 0, i = 1;
    while (i < m) {
       if (pattern.charAt(i) == pattern.charAt(len)) {
          len++;
          lps[i] = len;
          i++;
       } else {
          if (len != 0) {
             len = lps[len - 1];
          } else {
             lps[i] = 0;
             i++;
          }
       }
    }
    return lps;
 }
```

```
// Problem 16: Rabin–Karp algorithm (single pattern)
  public static List<Integer> rabinKarpSearch(String text, String pattern, int d, int q) {
     List<Integer> result = new ArrayList<>();
     int n = text.length(), m = pattern.length();
     if (m > n) return result;
     long pHash = 0, tHash = 0, h = 1;
     for (int i = 0; i < m - 1; i++) {
        h = (h * d) % q;
     }
     for (int i = 0; i < m; i++) {
        pHash = (d * pHash + pattern.charAt(i)) % q;
        tHash = (d * tHash + text.charAt(i)) % q;
     }
     for (int i = 0; i <= n - m; i++) {
        if (pHash == tHash) {
           boolean match = true;
           for (int j = 0; j < m; j++) {
              if (text.charAt(i + j) != pattern.charAt(j)) {
                 match = false;
                 break;
              }
           }
           if (match) result.add(i);
        }
        if (i < n - m) {
           tHash = (d * (tHash - text.charAt(i) * h) + text.charAt(i + m)) % q;
           if (tHash < 0) tHash += q;
        }
     }
     return result;
  }

  // Problem 17: Rabin–Karp for multiple patterns
  public static Map<String, List<Integer>> rabinKarpMultiplePatterns(String text, String[]
patterns, int d, int q) {
     Map<String, List<Integer>> result = new HashMap<>();
     for (String pattern : patterns) {
        result.put(pattern, rabinKarpSearch(text, pattern, d, q));
     }
     return result;
  }
```

```
// Problem 18: Z-algorithm for pattern matching
public static List<Integer> zAlgorithmSearch(String text, String pattern) {
    List<Integer> result = new ArrayList<>();
    String concat = pattern + "$" + text;
    int[] z = computeZArray(concat);
    int m = pattern.length();
    for (int i = 0; i < z.length; i++) {
        if (z[i] == m) {
            result.add(i - m - 1);
        }
    }
    return result;
}

private static int[] computeZArray(String s) {
    int n = s.length();
    int[] z = new int[n];
    int l = 0, r = 0;
    for (int i = 1; i < n; i++) {
        if (i <= r) {
            z[i] = Math.min(r - i + 1, z[i - l]);
        }
        while (i + z[i] < n && s.charAt(z[i]) == s.charAt(i + z[i])) {
            z[i]++;
        }
        if (i + z[i] - 1 > r) {
            l = i;
            r = i + z[i] - 1;
        }
    }
    return z;
}

// Problem 19: Finite automata-based pattern search
public static List<Integer> finiteAutomataSearch(String text, String pattern) {
    List<Integer> result = new ArrayList<>();
    int[][] tf = buildTransitionTable(pattern);
    int n = text.length(), m = pattern.length();
    int state = 0;
    for (int i = 0; i < n; i++) {
        state = tf[state][text.charAt(i)];
        if (state == m) {
            result.add(i - m + 1);
        }
    }
    return result;
}
```

```
private static int[][] buildTransitionTable(String pattern) {
    int m = pattern.length();
    int[][] tf = new int[m + 1][256]; // Assuming ASCII
    for (int state = 0; state <= m; state++) {
        for (int c = 0; c < 256; c++) {
            int k = Math.min(m, state + 1);
            while (k > 0 && !isSuffix(pattern, k, state, (char) c)) {
                k--;
            }
            tf[state][c] = k;
        }
    }
    return tf;
}

private static boolean isSuffix(String pattern, int k, int state, char c) {
    if (k == 0) return true;
    String temp = pattern.substring(0, state) + c;
    return temp.substring(temp.length() - k).equals(pattern.substring(0, k));
}

// Problem 20: Boyer–Moore algorithm
public static List<Integer> boyerMooreSearch(String text, String pattern) {
    List<Integer> result = new ArrayList<>();
    int n = text.length(), m = pattern.length();
    if (m == 0 || m > n) return result;
    int[] badChar = computeBadCharTable(pattern);
    int[] goodSuffix = computeGoodSuffixTable(pattern);
    int s = 0;
    while (s <= n - m) {
        int j = m - 1;
        while (j >= 0 && pattern.charAt(j) == text.charAt(s + j)) {
            j--;
        }
        if (j < 0) {
            result.add(s);
            s += goodSuffix[0];
        } else {
            s += Math.max(goodSuffix[j + 1], badChar[text.charAt(s + j)] - m + j + 1);
        }
    }
    return result;
}
```

```
private static int[] computeBadCharTable(String pattern) {
    int m = pattern.length();
    int[] badChar = new int[256];
    Arrays.fill(badChar, m);
    for (int i = 0; i < m; i++) {
        badChar[pattern.charAt(i)] = m - i - 1;
    }
    return badChar;
}

private static int[] computeGoodSuffixTable(String pattern) {
    int m = pattern.length();
    int[] suffix = computeSuffixArray(pattern);
    int[] goodSuffix = new int[m + 1];
    Arrays.fill(goodSuffix, m);
    for (int i = 0; i < m; i++) {
        if (suffix[i] == i + 1) {
            goodSuffix[m - i - 1] = m - i - 1;
        }
    }
    for (int i = 0; i < m - 1; i++) {
        goodSuffix[m - suffix[i] - 1] = Math.min(goodSuffix[m - suffix[i] - 1], i + 1);
    }
    return goodSuffix;
}

private static int[] computeSuffixArray(String pattern) {
    int m = pattern.length();
    int[] suffix = new int[m];
    int[] f = new int[m];
    f[m - 1] = m;
    int g = m - 1;
    for (int i = m - 2; i >= 0; i--) {
        if (i > g && f[i + m - 1 - f[i]] < i - g) {
            f[i] = f[i + m - 1 - f[i]];
        } else {
            g = Math.min(g, i);
            f[i] = i + 1;
            while (g >= 0 && pattern.charAt(g) == pattern.charAt(g + m - 1 - f[i])) {
                g--;
            }
            f[i] = m - 1 - g;
        }
    }
    for (int i = 0; i < m; i++) {
        suffix[i] = f[i];
    }
    return suffix;
}
```

```
// Problem 21: Aho–Corasick algorithm for multiple pattern search
    static class AhoCorasick {
        static class TrieNode {
            TrieNode[] children = new TrieNode[26];
            TrieNode failure;
            List<Integer> output;
            TrieNode() {
                output = new ArrayList<>();
            }
        }

        TrieNode root;
        AhoCorasick(String[] patterns) {
            root = new TrieNode();
            buildTrie(patterns);
            buildFailureLinks();
        }

        private void buildTrie(String[] patterns) {
            for (int i = 0; i < patterns.length; i++) {
                TrieNode node = root;
                for (char c : patterns[i].toCharArray()) {
                    int idx = c - 'a';
                    if (node.children[idx] == null) {
                        node.children[idx] = new TrieNode();
                    }
                    node = node.children[idx];
                }
                node.output.add(i);
            }
        }

        private void buildFailureLinks() {
            Queue<TrieNode> queue = new LinkedList<>();
            root.failure = root;
            for (int i = 0; i < 26; i++) {
                if (root.children[i] != null) {
                    root.children[i].failure = root;
                    queue.offer(root.children[i]);
                } else {
                    root.children[i] = root;
                }
            }
            while (!queue.isEmpty()) {
                TrieNode curr = queue.poll();
                for (int i = 0; i < 26; i++) {
                    if (curr.children[i] != null) {
                        TrieNode fail = curr.failure;
                        while (fail.children[i] == null) {
                            fail = fail.failure;
                        }
```

```
                curr.children[i].failure = fail.children[i];
                curr.children[i].output.addAll(fail.children[i].output);
                queue.offer(curr.children[i]);
            } else {
                curr.children[i] = curr.failure.children[i];
            }
        }
    }
}

public Map<String, List<Integer>> search(String text, String[] patterns) {
    Map<String, List<Integer>> result = new HashMap<>();
    for (String p : patterns) result.put(p, new ArrayList<>());
    TrieNode curr = root;
    for (int i = 0; i < text.length(); i++) {
        int idx = text.charAt(i) - 'a';
        curr = curr.children[idx];
        for (int patternIdx : curr.output) {
            result.get(patterns[patternIdx]).add(i - patterns[patternIdx].length() + 1);
        }
    }
    return result;
}
}

public static Map<String, List<Integer>> ahoCorasickSearch(String text, String[]
patterns) {
    AhoCorasick ac = new AhoCorasick(patterns);
    return ac.search(text, patterns);
}

// Problem 22: Wildcard pattern matching (? and *)
public static boolean wildcardMatching(String s, String p) {
    int m = s.length(), n = p.length();
    boolean[][] dp = new boolean[m + 1][n + 1];
    dp[0][0] = true;
    for (int j = 1; j <= n; j++) {
        if (p.charAt(j - 1) == '*') dp[0][j] = dp[0][j - 1];
    }
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (p.charAt(j - 1) == '*') {
                dp[i][j] = dp[i][j - 1] || dp[i - 1][j];
            } else if (p.charAt(j - 1) == '?' || s.charAt(i - 1) == p.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1];
            }
        }
    }
    return dp[m][n];
}
```

```
// Problem 23: Regular expression matching (DP-based)
public static boolean regexMatching(String s, String p) {
    int m = s.length(), n = p.length();
    boolean[][] dp = new boolean[m + 1][n + 1];
    dp[0][0] = true;
    for (int j = 1; j <= n; j++) {
        if (p.charAt(j - 1) == '*') {
            dp[0][j] = dp[0][j - 2];
        }
    }
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (p.charAt(j - 1) == '*') {
                dp[i][j] = dp[i][j - 2] ||
                        (dp[i - 1][j] && (s.charAt(i - 1) == p.charAt(j - 2) || p.charAt(j - 2) == '.'));
            } else if (p.charAt(j - 1) == '.' || s.charAt(i - 1) == p.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1];
            }
        }
    }
    return dp[m][n];
}

// Problem 24: Longest common prefix using naive approach
public static String longestCommonPrefix(String[] strs) {
    if (strs.length == 0) return "";
    String prefix = strs[0];
    for (int i = 1; i < strs.length; i++) {
        while (!strs[i].startsWith(prefix)) {
            prefix = prefix.substring(0, prefix.length() - 1);
            if (prefix.isEmpty()) return "";
        }
    }
    return prefix;
}

// Problem 25: Find all occurrences of a pattern in text using Z-algorithm
public static int[] findAllOccurrencesZ(String text, String pattern) {
    String concat = pattern + "$" + text;
    int[] z = computeZArray(concat);
    List<Integer> list = new ArrayList<>();
    int m = pattern.length();
    for (int i = 0; i < z.length; i++) {
        if (z[i] == m) {
            list.add(i - m - 1);
        }
    }
    return list.stream().mapToInt(i -> i).toArray();
}
```

```
  // Problem 26: Longest repeated substring using suffix array
    public static String longestRepeatedSubstring(String s) {
        int n = s.length();
        int[] suffixArray = buildSuffixArray(s);
        int[] lcp = buildLCPArray(s, suffixArray);
        int maxLen = 0, idx = 0;
        for (int i = 0; i < lcp.length; i++) {
            if (lcp[i] > maxLen) {
                maxLen = lcp[i];
                idx = suffixArray[i];
            }
        }
        return maxLen == 0 ? "" : s.substring(idx, idx + maxLen);
    }

    private static int[] buildSuffixArray(String s) {
        int n = s.length();
        Integer[] suffixes = new Integer[n];
        for (int i = 0; i < n; i++) suffixes[i] = i;
        Arrays.sort(suffixes, (a, b) -> s.substring(a).compareTo(s.substring(b)));
        return Arrays.stream(suffixes).mapToInt(Integer::intValue).toArray();
    }

    private static int[] buildLCPArray(String s, int[] suffixArray) {
        int n = s.length();
        int[] lcp = new int[n - 1];
        for (int i = 0; i < n - 1; i++) {
            int l = 0;
            while (suffixArray[i] + l < n && suffixArray[i + 1] + l < n &&
                   s.charAt(suffixArray[i] + l) == s.charAt(suffixArray[i + 1] + l)) {
                l++;
            }
            lcp[i] = l;
        }
        return lcp;
    }
}
```

Phase 3 – Substrings & Subsequences

23. Generate all substrings of a string
24. Generate all subsequences of a string
25. Longest common substring
26. Longest common subsequence (LCS)
27. Shortest common supersequence
28. Longest palindromic substring
29. Longest palindromic subsequence
30. Count distinct substrings using trie/suffix automaton
31. Count distinct subsequences
32. Number of distinct palindromic substrings

33. Smallest & largest lexicographic substring of length k

```
import java.util.*;

public class SubstringsSubsequences {

    // --- Phase 3: Substrings & Subsequences ---

    // Problem 1: Generate all substrings of a string
    public static List<String> generateAllSubstrings(String s) {
        List<String> result = new ArrayList<>();
        int n = s.length();
        for (int i = 0; i < n; i++) {
            for (int len = 1; i + len <= n; len++) {
                result.add(s.substring(i, i + len));
            }
        }
        return result;
    }

    // Problem 2: Generate all subsequences of a string
    public static List<String> generateAllSubsequences(String s) {
        List<String> result = new ArrayList<>();
        generateSubsequencesHelper(s, 0, new StringBuilder(), result);
        return result;
    }

    private static void generateSubsequencesHelper(String s, int idx, StringBuilder curr,
List<String> result) {
        if (idx == s.length()) {
            if (curr.length() > 0) result.add(curr.toString());
            return;
        }
        // Include current character
        curr.append(s.charAt(idx));
        generateSubsequencesHelper(s, idx + 1, curr, result);
        curr.deleteCharAt(curr.length() - 1);
```

```
    // Exclude current character
    generateSubsequencesHelper(s, idx + 1, curr, result);
}

// Problem 3: Longest common substring
public static String longestCommonSubstring(String s1, String s2) {
    int n = s1.length(), m = s2.length();
    int[][] dp = new int[n + 1][m + 1];
    int maxLen = 0, endIdx = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
                if (dp[i][j] > maxLen) {
                    maxLen = dp[i][j];
                    endIdx = i;
                }
            }
        }
    }
    return s1.substring(endIdx - maxLen, endIdx);
}

// Problem 4: Longest common subsequence (LCS)
public static int longestCommonSubsequence(String s1, String s2) {
    int n = s1.length(), m = s2.length();
    int[][] dp = new int[n + 1][m + 1];
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[n][m];
}

// Problem 5: Shortest common supersequence
public static String shortestCommonSupersequence(String s1, String s2) {
    int n = s1.length(), m = s2.length();
    int[][] dp = new int[n + 1][m + 1];
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
```

```
    }
    StringBuilder result = new StringBuilder();
    int i = n, j = m;
    while (i > 0 && j > 0) {
        if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
            result.append(s1.charAt(i - 1));
            i--;
            j--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            result.append(s1.charAt(i - 1));
            i--;
        } else {
            result.append(s2.charAt(j - 1));
            j--;
        }
    }
    while (i > 0) result.append(s1.charAt(--i));
    while (j > 0) result.append(s2.charAt(--j));
    return result.reverse().toString();
}

// Problem 6: Longest palindromic substring
public static String longestPalindromicSubstring(String s) {
    int n = s.length();
    boolean[][] dp = new boolean[n][n];
    int start = 0, maxLen = 1;
    // Single characters
    for (int i = 0; i < n; i++) dp[i][i] = true;
    // Two characters
    for (int i = 0; i < n - 1; i++) {
        if (s.charAt(i) == s.charAt(i + 1)) {
            dp[i][i + 1] = true;
            start = i;
            maxLen = 2;
        }
    }
    // Length 3 and above
    for (int len = 3; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1]) {
                dp[i][j] = true;
                if (len > maxLen) {
                    start = i;
                    maxLen = len;
                }
            }
        }
    }
    return s.substring(start, start + maxLen);
}
```

```
// Problem 7: Longest palindromic subsequence
public static int longestPalindromicSubsequence(String s) {
    int n = s.length();
    int[][] dp = new int[n][n];
    for (int i = 0; i < n; i++) dp[i][i] = 1;
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s.charAt(i) == s.charAt(j) && len == 2) {
                dp[i][j] = 2;
            } else if (s.charAt(i) == s.charAt(j)) {
                dp[i][j] = dp[i + 1][j - 1] + 2;
            } else {
                dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[0][n - 1];
}

// Problem 8: Count distinct substrings using trie
static class TrieNode {
    TrieNode[] children = new TrieNode[26];
    boolean isEnd;
}

public static int countDistinctSubstrings(String s) {
    TrieNode root = new TrieNode();
    int count = 0;
    for (int i = 0; i < s.length(); i++) {
        TrieNode node = root;
        for (int j = i; j < s.length(); j++) {
            int idx = s.charAt(j) - 'a';
            if (node.children[idx] == null) {
                node.children[idx] = new TrieNode();
                count++;
            }
            node = node.children[idx];
        }
    }
    return count;
}
```

```
    // Problem 9: Count distinct subsequences
    public static int countDistinctSubsequences(String s) {
        int n = s.length();
        int[] last = new int[26];
        Arrays.fill(last, -1);
        int[] dp = new int[n + 1];
        dp[0] = 1;
        for (int i = 1; i <= n; i++) {
            int idx = s.charAt(i - 1) - 'a';
            dp[i] = 2 * dp[i - 1];
            if (last[idx] != -1) {
                dp[i] -= dp[last[idx]];
            }
            last[idx] = i - 1;
        }
        return dp[n];
    }

    // Problem 10: Number of distinct palindromic substrings
    public static int countDistinctPalindromicSubstrings(String s) {
        Set<String> palindromes = new HashSet<>();
        int n = s.length();
        // Odd length palindromes
        for (int i = 0; i < n; i++) {
            expandAroundCenter(s, i, i, palindromes);
        }
        // Even length palindromes
        for (int i = 0; i < n - 1; i++) {
            expandAroundCenter(s, i, i + 1, palindromes);
        }
        return palindromes.size();
    }

    private static void expandAroundCenter(String s, int left, int right, Set<String>
palindromes) {
        while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
            palindromes.add(s.substring(left, right + 1));
            left--;
            right++;
        }
    }
```

```
// Problem 11: Smallest lexicographic substring of length k
  public static String smallestLexicographicSubstring(String s, int k) {
     if (k > s.length()) return "";
     String smallest = s.substring(0, k);
     for (int i = 1; i <= s.length() - k; i++) {
        String curr = s.substring(i, i + k);
        if (curr.compareTo(smallest) < 0) {
           smallest = curr;
        }
     }
     return smallest;
  }

  // Problem 12: Largest lexicographic substring of length k
  public static String largestLexicographicSubstring(String s, int k) {
     if (k > s.length()) return "";
     String largest = s.substring(0, k);
     for (int i = 1; i <= s.length() - k; i++) {
        String curr = s.substring(i, i + k);
        if (curr.compareTo(largest) > 0) {
           largest = curr;
        }
     }
     return largest;
  }

  // Problem 13: Edit distance between two strings
  public static int editDistance(String s1, String s2) {
     int n = s1.length(), m = s2.length();
     int[][] dp = new int[n + 1][m + 1];
     for (int i = 0; i <= n; i++) dp[i][0] = i;
     for (int j = 0; j <= m; j++) dp[0][j] = j;
     for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
           if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
              dp[i][j] = dp[i - 1][j - 1];
           } else {
              dp[i][j] = Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1])) + 1;
           }
        }
     }
     return dp[n][m];
  }
```

```
// Problem 14: Longest repeating subsequence
public static int longestRepeatingSubsequence(String s) {
    int n = s.length();
    int[][] dp = new int[n + 1][n + 1];
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (s.charAt(i - 1) == s.charAt(j - 1) && i != j) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[n][n];
}

// Problem 15: Minimum insertions to make string palindromic
public static int minInsertionsToPalindrome(String s) {
    return s.length() - longestPalindromicSubsequence(s);
}

// Problem 16: Count subsequences of a given pattern in string
public static int countSubsequencesPattern(String s, String pattern) {
    int n = s.length(), m = pattern.length();
    int[][] dp = new int[n + 1][m + 1];
    for (int i = 0; i <= n; i++) dp[i][0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s.charAt(i - 1) == pattern.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[n][m];
}
```

```
  // Problem 17: Longest substring with at most k distinct characters
   public static int longestSubstringAtMostKDistinct(String s, int k) {
      Map<Character, Integer> freq = new HashMap<>();
      int left = 0, maxLen = 0;
      for (int right = 0; right < s.length(); right++) {
         freq.put(s.charAt(right), freq.getOrDefault(s.charAt(right), 0) + 1);
         while (freq.size() > k) {
            freq.put(s.charAt(left), freq.get(s.charAt(left)) - 1);
            if (freq.get(s.charAt(left)) == 0) freq.remove(s.charAt(left));
            left++;
         }
         maxLen = Math.max(maxLen, right - left + 1);
      }
      return maxLen;
   }
}
```

Phase 4 – String Transformations

34. Minimum insertions to make string palindrome
35. Minimum deletions to make string palindrome
36. Minimum swaps to make string palindrome
37. Edit distance (Levenshtein distance)
38. Convert one string to another with minimum operations (insert/delete/replace)
39. Check if one string is a subsequence of another
40. Transform string under given mapping rules
41. Minimum adjacent swaps to convert string A to string B
42. Sort characters in a string by frequency

Phase 5 – Advanced String Algorithms

43. Trie (prefix tree) for string storage & prefix queries
44. Suffix array construction (O(n log n))
45. Suffix tree (compressed trie of suffixes)
46. Suffix automaton
47. Longest repeated substring
48. Longest prefix which is also a suffix (LPS array from KMP)
49. String hashing (polynomial rolling hash)
50. Double hashing to reduce collisions

```
import java.util.*;

public class StringTransformationsAndAdvanced {

    // --- Phase 4: String Transformations ---

    // Problem 1: Minimum insertions to make string palindrome
    public static int minInsertionsToPalindrome(String s) {
        return s.length() - longestPalindromicSubsequence(s);
    }

    private static int longestPalindromicSubsequence(String s) {
        int n = s.length();
        int[][] dp = new int[n][n];
        for (int i = 0; i < n; i++) dp[i][i] = 1;
        for (int len = 2; len <= n; len++) {
            for (int i = 0; i <= n - len; i++) {
                int j = i + len - 1;
                if (s.charAt(i) == s.charAt(j) && len == 2) {
                    dp[i][j] = 2;
                } else if (s.charAt(i) == s.charAt(j)) {
                    dp[i][j] = dp[i + 1][j - 1] + 2;
                } else {
                    dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[0][n - 1];
    }

    // Problem 2: Minimum deletions to make string palindrome
    public static int minDeletionsToPalindrome(String s) {
        return s.length() - longestPalindromicSubsequence(s);
    }

    // Problem 3: Minimum swaps to make string palindrome
    public static int minSwapsToPalindrome(String s) {
        char[] arr = s.toCharArray();
        int n = arr.length, swaps = 0;
        int left = 0, right = n - 1;
        while (left < right) {
            if (arr[left] != arr[right]) {
                int k = right;
                while (k > left && arr[k] != arr[left]) k--;
                if (k == left) {
                    // Swap with adjacent to make progress
                    char temp = arr[left];
                    arr[left] = arr[left + 1];
                    arr[left + 1] = temp;
                    swaps++;
```

```
            } else {
                // Move matching character to right position
                while (k < right) {
                    char temp = arr[k];
                    arr[k] = arr[k + 1];
                    arr[k + 1] = temp;
                    k++;
                    swaps++;
                }
                left++;
                right--;
            }
        } else {
            left++;
            right--;
        }
    }
    // Check if result is palindrome
    for (int i = 0; i < n / 2; i++) {
        if (arr[i] != arr[n - 1 - i]) return -1;
    }
    return swaps;
}

// Problem 4: Edit distance (Levenshtein distance)
public static int editDistance(String s1, String s2) {
    int n = s1.length(), m = s2.length();
    int[][] dp = new int[n + 1][m + 1];
    for (int i = 0; i <= n; i++) dp[i][0] = i;
    for (int j = 0; j <= m; j++) dp[0][j] = j;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1])) + 1;
            }
        }
    }
    return dp[n][m];
}

// Problem 5: Convert one string to another with minimum operations
public static int minOperationsToConvert(String s1, String s2) {
    return editDistance(s1, s2);
}
```

```
// Problem 6: Check if one string is a subsequence of another
  public static boolean isSubsequence(String s1, String s2) {
     int m = s1.length(), n = s2.length();
     int i = 0, j = 0;
     while (i < m && j < n) {
        if (s1.charAt(i) == s2.charAt(j)) i++;
        j++;
     }
     return i == m;
  }

  // Problem 7: Transform string under given mapping rules
  public static String transformString(String s, Map<Character, Character> mapping) {
     StringBuilder sb = new StringBuilder();
     for (char c : s.toCharArray()) {
        sb.append(mapping.getOrDefault(c, c));
     }
     return sb.toString();
  }

  // Problem 8: Minimum adjacent swaps to convert string A to string B
  public static int minAdjacentSwaps(String A, String B) {
     if (A.length() != B.length() || !areAnagrams(A, B)) return -1;
     int n = A.length(), swaps = 0;
     char[] arr = A.toCharArray();
     for (int i = 0; i < n; i++) {
        if (arr[i] != B.charAt(i)) {
           int j = i + 1;
           while (j < n && arr[j] != B.charAt(i)) j++;
           while (j > i) {
              char temp = arr[j];
              arr[j] = arr[j - 1];
              arr[j - 1] = temp;
              swaps++;
              j--;
           }
        }
     }
     return swaps;
  }
```

```
private static boolean areAnagrams(String s1, String s2) {
    if (s1.length() != s2.length()) return false;
    int[] count = new int[26];
    for (int i = 0; i < s1.length(); i++) {
        count[s1.charAt(i) - 'a']++;
        count[s2.charAt(i) - 'a']--;
    }
    for (int c : count) {
        if (c != 0) return false;
    }
    return true;
}

// Problem 9: Sort characters in a string by frequency
public static String sortByFrequency(String s) {
    Map<Character, Integer> freq = new HashMap<>();
    for (char c : s.toCharArray()) {
        freq.put(c, freq.getOrDefault(c, 0) + 1);
    }
    List<Character> chars = new ArrayList<>(freq.keySet());
    chars.sort((a, b) -> {
        int cmp = freq.get(b) - freq.get(a);
        return cmp == 0 ? a - b : cmp;
    });
    StringBuilder sb = new StringBuilder();
    for (char c : chars) {
        for (int i = 0; i < freq.get(c); i++) {
            sb.append(c);
        }
    }
    return sb.toString();
}

// Problem 10: Minimum insertions to make string palindrome (alternative approach)
public static int minInsertionsToPalindromeIterative(String s) {
    int n = s.length();
    int[][] dp = new int[n][n];
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s.charAt(i) == s.charAt(j)) {
                dp[i][j] = dp[i + 1][j - 1];
            } else {
                dp[i][j] = Math.min(dp[i + 1][j], dp[i][j - 1]) + 1;
            }
        }
    }
    return dp[0][n - 1];
}
```

```
// --- Phase 5: Advanced String Algorithms ---

  // Problem 11: Trie (prefix tree) for string storage & prefix queries
  static class TrieNode {
     TrieNode[] children = new TrieNode[26];
     boolean isEnd;
  }

  static class Trie {
     TrieNode root;

     Trie() {
        root = new TrieNode();
     }

     void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
           int idx = c - 'a';
           if (node.children[idx] == null) {
              node.children[idx] = new TrieNode();
           }
           node = node.children[idx];
        }
        node.isEnd = true;
     }

     boolean searchPrefix(String prefix) {
        TrieNode node = root;
        for (char c : prefix.toCharArray()) {
           int idx = c - 'a';
           if (node.children[idx] == null) return false;
           node = node.children[idx];
        }
        return true;
     }

     boolean searchWord(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
           int idx = c - 'a';
           if (node.children[idx] == null) return false;
           node = node.children[idx];
        }
        return node.isEnd;
     }
  }
```

```
public static Trie buildTrie(String[] words) {
        Trie trie = new Trie();
        for (String word : words) {
            trie.insert(word);
        }
        return trie;
    }

    // Problem 12: Suffix array construction (O(n log n))
    public static int[] buildSuffixArray(String s) {
        int n = s.length();
        Integer[] suffixes = new Integer[n];
        for (int i = 0; i < n; i++) suffixes[i] = i;
        Arrays.sort(suffixes, (a, b) -> s.substring(a).compareTo(s.substring(b)));
        return Arrays.stream(suffixes).mapToInt(Integer::intValue).toArray();
    }

    // Problem 13: Suffix tree (simplified using trie for demonstration)
    static class SuffixTreeNode {
        Map<Character, SuffixTreeNode> children = new HashMap<>();
        List<Integer> indices = new ArrayList<>();
    }

    public static SuffixTreeNode buildSuffixTree(String s) {
        SuffixTreeNode root = new SuffixTreeNode();
        for (int i = 0; i < s.length(); i++) {
            SuffixTreeNode node = root;
            for (int j = i; j < s.length(); j++) {
                char c = s.charAt(j);
                node.children.computeIfAbsent(c, k -> new SuffixTreeNode());
                node = node.children.get(c);
                node.indices.add(i);
            }
        }
        return root;
    }




 // Problem 14: Suffix automaton (simplified for basic substring queries)
    static class SuffixAutomaton {
        static class State {
            Map<Character, State> next = new HashMap<>();
            State link;
            int len;
            boolean isClone;
        }
```

```
    State root, last;

    SuffixAutomaton(String s) {
        root = new State();
        root.link = null;
        root.len = 0;
        last = root;
        for (char c : s.toCharArray()) {
            extend(c);
        }
    }

    private void extend(char c) {
        State curr = new State();
        curr.len = last.len + 1;
        State p = last;
        last = curr;
        while (p != null && !p.next.containsKey(c)) {
            p.next.put(c, curr);
            p = p.link;
        }
        if (p != null) {
            State q = p.next.get(c);
            if (p.len + 1 == q.len) {
                curr.link = q;
            } else {
                State clone = new State();
                clone.len = p.len + 1;
                clone.next = new HashMap<>(q.next);
                clone.link = q.link;
                clone.isClone = true;
                curr.link = clone;
                q.link = clone;
                while (p != null && p.next.get(c) == q) {
                    p.next.put(c, clone);
                    p = p.link;
                }
            }
        } else {
            curr.link = root;
        }
    }
}

public static SuffixAutomaton buildSuffixAutomaton(String s) {
    return new SuffixAutomaton(s);
}
```

```
// Problem 15: Longest repeated substring
 public static String longestRepeatedSubstring(String s) {
    int[] suffixArray = buildSuffixArray(s);
    int[] lcp = buildLCPArray(s, suffixArray);
    int maxLen = 0, idx = 0;
    for (int i = 0; i < lcp.length; i++) {
       if (lcp[i] > maxLen) {
          maxLen = lcp[i];
          idx = suffixArray[i];
       }
    }
    return maxLen == 0 ? "" : s.substring(idx, idx + maxLen);
 }

 private static int[] buildLCPArray(String s, int[] suffixArray) {
    int n = s.length();
    int[] lcp = new int[n - 1];
    for (int i = 0; i < n - 1; i++) {
       int l = 0;
       while (suffixArray[i] + l < n && suffixArray[i + 1] + l < n &&
             s.charAt(suffixArray[i] + l) == s.charAt(suffixArray[i + 1] + l)) {
          l++;
       }
       lcp[i] = l;
    }
    return lcp;
 }

 // Problem 16: Longest prefix which is also a suffix (LPS array from KMP)
 public static int[] computeLPSArray(String s) {
    int n = s.length();
    int[] lps = new int[n];
    int len = 0, i = 1;
    while (i < n) {
       if (s.charAt(i) == s.charAt(len)) {
          len++;
          lps[i] = len;
          i++;
       } else {
          if (len != 0) {
             len = lps[len - 1];
          } else {
             lps[i] = 0;
             i++;
          }
       }
    }
    return lps;
 }
```

```
    // Problem 17: String hashing (polynomial rolling hash)
    public static long computeStringHash(String s, long p, long m) {
        long hash = 0;
        for (char c : s.toCharArray()) {
            hash = (hash * p + c) % m;
        }
        return hash;
    }

    // Problem 18: Double hashing to reduce collisions
    public static long[] computeDoubleHash(String s, long p1, long m1, long p2, long m2) {
        long hash1 = 0, hash2 = 0;
        for (char c : s.toCharArray()) {
            hash1 = (hash1 * p1 + c) % m1;
            hash2 = (hash2 * p2 + c) % m2;
        }
        return new long[]{hash1, hash2};
    }

    // Problem 19: Minimum deletions to make string sorted
    public static int minDeletionsToSorted(String s) {
        int n = s.length();
        int[][] dp = new int[n + 1][26];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < 26; j++) {
                dp[i + 1][j] = dp[i][j] + (s.charAt(i) - 'a' <= j ? 0 : 1);
                if (j > 0) dp[i + 1][j] = Math.min(dp[i + 1][j], dp[i + 1][j - 1]);
            }
        }
        return dp[n][25];
    }

    // Problem 20: Transform string to palindrome with given character set
    public static int minOperationsToPalindromeWithCharset(String s, Set<Character>
charset) {
        int n = s.length();
        int[][] dp = new int[n][n];
        for (int len = 2; len <= n; len++) {
            for (int i = 0; i <= n - len; i++) {
                int j = i + len - 1;
                if (s.charAt(i) == s.charAt(j) && charset.contains(s.charAt(i))) {
                    dp[i][j] = dp[i + 1][j - 1];
                } else {
                    dp[i][j] = Math.min(dp[i + 1][j], dp[i][j - 1]) + 1;
                }
            }
        }
        return dp[0][n - 1];
    }
}
```

43. Manacher’s algorithm for longest palindromic substring
44. Burrows–Wheeler transform
45. Lyndon factorization

Phase 6 – String Processing in Competitive Programming

54. String compression (run-length encoding)
55. Decompression from run-length format
56. Big integer addition & multiplication using strings
57. Multiply two large numbers represented as strings
58. Division of large numbers represented as strings
59. Check if string is a valid number (parsing)
60. Word break problem (DP + trie)
61. Minimum cuts for palindrome partitioning
62. Smallest window in a string containing all characters of another string
63. Longest substring without repeating characters (sliding window)
64. Minimum window substring
65. Count anagram substrings of a pattern in text
66. Find all permutations of a string
67. Group anagrams from a list of strings

```
import java.util.*;

public class StringAlgorithms {

    // 43. Manacher’s Algorithm - Longest Palindromic Substring
    public String longestPalindromicSubstringManacher(String s) {
        if (s == null || s.length() == 0) return "";
        char[] t = new char[s.length() * 2 + 3];
        t[0] = '^';
        for (int i = 0; i < s.length(); i++) {
            t[2 * i + 1] = '#';
            t[2 * i + 2] = s.charAt(i);
        }
        t[s.length() * 2 + 1] = '#';
        t[s.length() * 2 + 2] = '$';
        int[] p = new int[t.length];
        int center = 0, right = 0;
        for (int i = 1; i < t.length - 1; i++) {
            int mirror = 2 * center - i;
            if (i < right) p[i] = Math.min(right - i, p[mirror]);
            while (t[i + (1 + p[i])] == t[i - (1 + p[i])]) p[i]++;
            if (i + p[i] > right) {
                center = i;
                right = i + p[i];
            }
        }
```

```
    int maxLen = 0, centerIndex = 0;
    for (int i = 1; i < p.length - 1; i++) {
        if (p[i] > maxLen) {
            maxLen = p[i];
            centerIndex = i;
        }
    }
    int start = (centerIndex - maxLen) / 2;
    return s.substring(start, start + maxLen);
}

// 44. Burrows–Wheeler Transform
public String burrowsWheelerTransform(String s) {
    s += "$";
    int n = s.length();
    String[] rotations = new String[n];
    for (int i = 0; i < n; i++) {
        rotations[i] = s.substring(i) + s.substring(0, i);
    }
    Arrays.sort(rotations);
    StringBuilder lastCol = new StringBuilder();
    for (String row : rotations) lastCol.append(row.charAt(n - 1));
    return lastCol.toString();
}

// 45. Lyndon Factorization (Duval's algorithm)
public List<String> lyndonFactorization(String s) {
    List<String> factors = new ArrayList<>();
    int n = s.length();
    int i = 0;
    while (i < n) {
        int j = i, k = i + 1;
        while (k < n && s.charAt(j) <= s.charAt(k)) {
            if (s.charAt(j) < s.charAt(k)) j = i;
            else j++;
            k++;
        }
        while (i <= j) {
            factors.add(s.substring(i, i + k - j));
            i += k - j;
        }
    }
    return factors;
}

// 54. String compression (Run-length encoding)
public String runLengthEncoding(String s) {
    if (s == null || s.isEmpty()) return "";
    StringBuilder sb = new StringBuilder();
    int count = 1;
    for (int i = 1; i <= s.length(); i++) {
```

```
        if (i < s.length() && s.charAt(i) == s.charAt(i - 1)) count++;
        else {
            sb.append(s.charAt(i - 1)).append(count);
            count = 1;
        }
    }
    return sb.toString();
}

// 55. Decompression from run-length format
public String runLengthDecoding(String s) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < s.length(); i++) {
        char ch = s.charAt(i);
        int j = i + 1;
        StringBuilder num = new StringBuilder();
        while (j < s.length() && Character.isDigit(s.charAt(j))) {
            num.append(s.charAt(j++));
        }
        int count = Integer.parseInt(num.toString());
        for (int k = 0; k < count; k++) sb.append(ch);
        i = j - 1;
    }
    return sb.toString();
}

// 56. Big integer addition using strings
public String addStrings(String num1, String num2) {
    StringBuilder sb = new StringBuilder();
    int i = num1.length() - 1, j = num2.length() - 1, carry = 0;
    while (i >= 0 || j >= 0 || carry > 0) {
        int sum = carry;
        if (i >= 0) sum += num1.charAt(i--) - '0';
        if (j >= 0) sum += num2.charAt(j--) - '0';
        sb.append(sum % 10);
        carry = sum / 10;
    }
    return sb.reverse().toString();
}

// 57. Multiply two large numbers represented as strings
public String multiplyStrings(String num1, String num2) {
    if (num1.equals("0") || num2.equals("0")) return "0";
    int[] result = new int[num1.length() + num2.length()];
    for (int i = num1.length() - 1; i >= 0; i--) {
        for (int j = num2.length() - 1; j >= 0; j--) {
            int mul = (num1.charAt(i) - '0') * (num2.charAt(j) - '0');
            int sum = mul + result[i + j + 1];
            result[i + j + 1] = sum % 10;
            result[i + j] += sum / 10;
        }
```

```
    }
    StringBuilder sb = new StringBuilder();
    for (int num : result) if (!(sb.length() == 0 && num == 0)) sb.append(num);
    return sb.toString();
}

// 58. Division of large numbers represented as strings
public String divideStrings(String dividend, String divisor) {
    if (divisor.equals("0")) throw new ArithmeticException("Divide by zero");
    if (dividend.equals("0")) return "0";
    StringBuilder result = new StringBuilder();
    String current = "";
    long div = Long.parseLong(divisor);
    for (int i = 0; i < dividend.length(); i++) {
        current += dividend.charAt(i);
        long curVal = Long.parseLong(current);
        if (curVal < div) {
            if (result.length() > 0) result.append("0");
        } else {
            long count = curVal / div;
            result.append(count);
            current = String.valueOf(curVal % div);
        }
    }
    return result.toString();
}

// 59. Check if string is a valid number
public boolean isValidNumber(String s) {
    s = s.trim();
    if (s.isEmpty()) return false;
    boolean numSeen = false, dotSeen = false, eSeen = false;
    int numAfterE = 0;
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (Character.isDigit(c)) {
            numSeen = true;
            if (eSeen) numAfterE++;
        } else if (c == '.') {
            if (dotSeen || eSeen) return false;
            dotSeen = true;
        } else if (c == 'e' || c == 'E') {
            if (eSeen || !numSeen) return false;
            eSeen = true;
            numAfterE = 0;
        } else if (c == '+' || c == '-') {
            if (i != 0 && s.charAt(i - 1) != 'e' && s.charAt(i - 1) != 'E') return false;
        } else {
            return false;
        }
    }
```

```
        return numSeen && (!eSeen || numAfterE > 0);
    }


}

    // 60. Word Break Problem (DP + Trie not required here but can be added for
optimization)
    public boolean wordBreak(String s, List<String> wordDict) {
        Set<String> dict = new HashSet<>(wordDict);
        boolean[] dp = new boolean[s.length() + 1];
        dp[0] = true;
        for (int i = 1; i <= s.length(); i++) {
            for (int j = 0; j < i; j++) {
                if (dp[j] && dict.contains(s.substring(j, i))) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[s.length()];
    }

    // 61. Minimum cuts for palindrome partitioning
    public int minCutPalindromePartition(String s) {
        int n = s.length();
        boolean[][] isPal = new boolean[n][n];
        int[] cuts = new int[n];
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = 0; j <= i; j++) {
                if (s.charAt(i) == s.charAt(j) && (i - j <= 1 || isPal[j + 1][i - 1])) {
                    isPal[j][i] = true;
                    min = j == 0 ? 0 : Math.min(min, cuts[j - 1] + 1);
                }
            }
            cuts[i] = min;
        }
        return cuts[n - 1];
    }

    // 62. Smallest window containing all characters of another string
    public String smallestWindowContainingAllChars(String s, String t) {
        if (s.length() < t.length()) return "";
        Map<Character, Integer> map = new HashMap<>();
        for (char c : t.toCharArray()) map.put(c, map.getOrDefault(c, 0) + 1);
        int required = map.size(), formed = 0;
        Map<Character, Integer> windowCounts = new HashMap<>();
        int l = 0, r = 0, minLen = Integer.MAX_VALUE, start = 0;
        while (r < s.length()) {
            char c = s.charAt(r);
```

```
            windowCounts.put(c, windowCounts.getOrDefault(c, 0) + 1);
            if (map.containsKey(c) && windowCounts.get(c).intValue() ==
map.get(c).intValue()) formed++;
            while (l <= r && formed == required) {
                if (r - l + 1 < minLen) {
                    minLen = r - l + 1;
                    start = l;
                }
                char lc = s.charAt(l);
                windowCounts.put(lc, windowCounts.get(lc) - 1);
                if (map.containsKey(lc) && windowCounts.get(lc).intValue() <
map.get(lc).intValue()) formed--;
                l++;
            }
            r++;
        }
        return minLen == Integer.MAX_VALUE ? "" : s.substring(start, start + minLen);
    }

    // 63. Longest substring without repeating characters
    public int lengthOfLongestSubstringWithoutRepeating(String s) {
        Set<Character> set = new HashSet<>();
        int l = 0, maxLen = 0;
        for (int r = 0; r < s.length(); r++) {
            while (set.contains(s.charAt(r))) {
                set.remove(s.charAt(l++));
            }
            set.add(s.charAt(r));
            maxLen = Math.max(maxLen, r - l + 1);
        }
        return maxLen;
    }

    // 64. Minimum window substring
    public String minWindowSubstring(String s, String t) {
        if (s.length() < t.length()) return "";
        Map<Character, Integer> map = new HashMap<>();
        for (char c : t.toCharArray()) map.put(c, map.getOrDefault(c, 0) + 1);
        int required = map.size(), formed = 0;
        Map<Character, Integer> windowCounts = new HashMap<>();
        int l = 0, r = 0, minLen = Integer.MAX_VALUE, start = 0;
        while (r < s.length()) {
            char c = s.charAt(r);
            windowCounts.put(c, windowCounts.getOrDefault(c, 0) + 1);
            if (map.containsKey(c) && windowCounts.get(c).intValue() ==
map.get(c).intValue()) formed++;
            while (l <= r && formed == required) {
                if (r - l + 1 < minLen) {
                    minLen = r - l + 1;
                    start = l;
                }
```

```
            char lc = s.charAt(l);
            windowCounts.put(lc, windowCounts.get(lc) - 1);
            if (map.containsKey(lc) && windowCounts.get(lc).intValue() <
map.get(lc).intValue()) formed--;
            l++;
        }
        r++;
    }
    return minLen == Integer.MAX_VALUE ? "" : s.substring(start, start + minLen);
  }

  // 65. Count anagram substrings of a pattern in text
  public int countAnagramSubstrings(String text, String pattern) {
    if (pattern.length() > text.length()) return 0;
    int[] pCount = new int[26], sCount = new int[26];
    for (char c : pattern.toCharArray()) pCount[c - 'a']++;
    int count = 0;
    for (int i = 0; i < text.length(); i++) {
      sCount[text.charAt(i) - 'a']++;
      if (i >= pattern.length()) sCount[text.charAt(i - pattern.length()) - 'a']--;
      if (Arrays.equals(pCount, sCount)) count++;
    }
    return count;
  }

  // 66. Find all permutations of a string
  public List<String> permutations(String s) {
    List<String> result = new ArrayList<>();
    permuteHelper("", s, result);
    return result;
  }
  private void permuteHelper(String prefix, String remaining, List<String> result) {
    if (remaining.isEmpty()) {
      result.add(prefix);
    } else {
      for (int i = 0; i < remaining.length(); i++) {
        permuteHelper(prefix + remaining.charAt(i),
            remaining.substring(0, i) + remaining.substring(i + 1),
            result);
      }
    }
  }

  // 67. Group anagrams from a list of strings
  public List<List<String>> groupAnagrams(String[] strs) {
    Map<String, List<String>> map = new HashMap<>();
    for (String str : strs) {
      char[] chars = str.toCharArray();
      Arrays.sort(chars);
      String key = new String(chars);
      map.computeIfAbsent(key, k -> new ArrayList<>()).add(str);
```

```
        }
        return new ArrayList<>(map.values());
    }
}
```

Phase 7 – Advanced Applications & Contest Tricks

68. Cyclic rotations & minimal lexicographic rotation (Booth's algorithm)
69. Binary string problems (flip bits, max consecutive 1s, min flips to make alternate)
70. Count binary strings without consecutive 1s
71. Decode ways (string to number mappings)
72. Gray code generation from binary strings
73. Password generation/validation problems
74. Substring queries with hash + binary search (fast compare)
75. Find kth substring in lexicographic order
76. Detect plagiarism using string hashing / suffix arrays
77. String periodicity problems (smallest period)
78. Infinite string repetition queries
79. Palindromic tree (eertree) for counting palindromes

```
import java.util.*;

public class AdvancedStringAlgorithms {

    // 68. Minimal Lexicographic Rotation (Booth's Algorithm)
    public String minimalLexicographicRotation(String s) {
        s = s + s;
        int n = s.length();
        int f[] = new int[n];
        Arrays.fill(f, -1);
        int k = 0;
        for (int j = 1; j < n; j++) {
            char sj = s.charAt(j), sk = s.charAt(k + j - f[k] - 1);
            if (sj < sk) {
                k = j - f[k] - 1;
            } else if (sj > sk) {
                f[j] = j - k - 1;
            } else {
                f[j] = f[k + j - f[k] - 1];
            }
        }
        return s.substring(k, k + n / 2);
    }

    // Variation: Check if two strings are rotations of each other
    public boolean areRotations(String s1, String s2) {
        return s1.length() == s2.length() && (s1 + s1).contains(s2);
    }
```

```
// 69. Binary string problems
public int maxConsecutiveOnes(String s) {
    int max = 0, count = 0;
    for (char c : s.toCharArray()) {
        if (c == '1') max = Math.max(max, ++count);
        else count = 0;
    }
    return max;
}

public int minFlipsToMakeAlternate(String s) {
    int flips1 = 0, flips2 = 0;
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) != (i % 2 == 0 ? '0' : '1')) flips1++;
        if (s.charAt(i) != (i % 2 == 0 ? '1' : '0')) flips2++;
    }
    return Math.min(flips1, flips2);
}

public String flipBits(String s) {
    char[] arr = s.toCharArray();
    for (int i = 0; i < arr.length; i++) arr[i] = arr[i] == '0' ? '1' : '0';
    return new String(arr);
}

// 70. Count binary strings without consecutive 1s (DP)
public int countBinaryStringsNoConsecutive1s(int n) {
    if (n == 0) return 0;
    if (n == 1) return 2;
    int a = 1, b = 1; // ending with 0, ending with 1
    for (int i = 2; i <= n; i++) {
        int newA = a + b;
        int newB = a;
        a = newA;
        b = newB;
    }
    return a + b;
}

// 71. Decode ways (string to number mappings)
public int numDecodings(String s) {
    if (s == null || s.length() == 0 || s.charAt(0) == '0') return 0;
    int n = s.length();
    int[] dp = new int[n + 1];
    dp[0] = 1; dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        int one = Integer.parseInt(s.substring(i - 1, i));
        int two = Integer.parseInt(s.substring(i - 2, i));
        if (one >= 1) dp[i] += dp[i - 1];
        if (two >= 10 && two <= 26) dp[i] += dp[i - 2];
```

```
        }
        return dp[n];
    }

    // 72. Gray code generation
    public List<String> generateGrayCode(int n) {
        List<String> res = new ArrayList<>();
        if (n <= 0) return res;
        res.add("0");
        res.add("1");
        for (int i = 2; i <= n; i++) {
            int size = res.size();
            for (int j = size - 1; j >= 0; j--) res.add(res.get(j));
            for (int j = 0; j < size; j++) res.set(j, "0" + res.get(j));
            for (int j = size; j < res.size(); j++) res.set(j, "1" + res.get(j));
        }
        return res;
    }

    // 73. Password validation (length, char types, etc.)
    public boolean isValidPassword(String password) {
        if (password.length() < 8) return false;
        boolean hasUpper = false, hasLower = false, hasDigit = false, hasSpecial = false;
        for (char c : password.toCharArray()) {
            if (Character.isUpperCase(c)) hasUpper = true;
            else if (Character.isLowerCase(c)) hasLower = true;
            else if (Character.isDigit(c)) hasDigit = true;
            else hasSpecial = true;
        }
        return hasUpper && hasLower && hasDigit && hasSpecial;
    }

    // 74. Substring queries using hash + binary search (Rabin-Karp style)
    public boolean substringExists(String text, String pattern) {
        int n = text.length(), m = pattern.length();
        if (m > n) return false;
        long base = 257, mod = 1_000_000_007;
        long pHash = 0, tHash = 0, pow = 1;
        for (int i = 0; i < m; i++) {
            pHash = (pHash * base + pattern.charAt(i)) % mod;
            tHash = (tHash * base + text.charAt(i)) % mod;
            pow = (pow * base) % mod;
        }
        if (pHash == tHash && text.substring(0, m).equals(pattern)) return true;
        for (int i = m; i < n; i++) {
            tHash = (tHash * base + text.charAt(i) - pow * text.charAt(i - m) % mod + mod) %
mod;
            if (pHash == tHash && text.substring(i - m + 1, i + 1).equals(pattern)) return true;
        }
        return false;
    }
```

```
// 75. Find k-th substring in lexicographic order (suffix array + set)
public String kthSubstring(String s, int k) {
    TreeSet<String> set = new TreeSet<>();
    for (int i = 0; i < s.length(); i++) {
        for (int j = i + 1; j <= s.length(); j++) {
            set.add(s.substring(i, j));
        }
    }
    int count = 0;
    for (String sub : set) {
        count++;
        if (count == k) return sub;
    }
    return "";
}

// 76. Detect plagiarism using string hashing
public double plagiarismScore(String text1, String text2, int k) {
    Set<Long> hashes1 = new HashSet<>();
    Set<Long> hashes2 = new HashSet<>();
    long base = 257, mod = 1_000_000_007;
    for (int i = 0; i + k <= text1.length(); i++) {
        String sub = text1.substring(i, i + k);
        long h = 0;
        for (char c : sub.toCharArray()) h = (h * base + c) % mod;
        hashes1.add(h);
    }
    for (int i = 0; i + k <= text2.length(); i++) {
        String sub = text2.substring(i, i + k);
        long h = 0;
        for (char c : sub.toCharArray()) h = (h * base + c) % mod;
        hashes2.add(h);
    }
    Set<Long> common = new HashSet<>(hashes1);
    common.retainAll(hashes2);
    return (double) common.size() / Math.max(hashes1.size(), hashes2.size());
}

// 77. String periodicity (smallest period length)
public int smallestPeriod(String s) {
    int n = s.length();
    int[] lps = new int[n];
    for (int i = 1, len = 0; i < n; ) {
        if (s.charAt(i) == s.charAt(len)) lps[i++] = ++len;
        else if (len > 0) len = lps[len - 1];
        else lps[i++] = 0;
    }
    int len = lps[n - 1];
    if (len > 0 && n % (n - len) == 0) return n - len;
    return n;
```

```
}

// 78. Infinite string repetition queries
public char charAtInInfiniteString(String s, long index) {
    return s.charAt((int) (index % s.length()));
}

// 79. Palindromic tree (Eertree) for counting distinct palindromes
// Simplified implementation
class Node {
    int len, occ;
    Map<Character, Integer> next = new HashMap<>();
    int link;
    Node(int len) { this.len = len; }
}
private List<Node> tree;
private String buildStr;
private int suff, idx;

public void initEertree(String s) {
    buildStr = s;
    tree = new ArrayList<>();
    tree.add(new Node(-1));
    tree.add(new Node(0));
    tree.get(0).link = 0;
    tree.get(1).link = 0;
    suff = 1;
    for (int i = 0; i < s.length(); i++) addLetter(i);
}

private void addLetter(int pos) {
    int cur = suff;
    char c = buildStr.charAt(pos);
    while (true) {
        int curlen = tree.get(cur).len;
        if (pos - curlen - 1 >= 0 && buildStr.charAt(pos - curlen - 1) == c) break;
        cur = tree.get(cur).link;
    }
    if (tree.get(cur).next.containsKey(c)) {
        suff = tree.get(cur).next.get(c);
        tree.get(suff).occ++;
        return;
    }
    Node newNode = new Node(tree.get(cur).len + 2);
    tree.add(newNode);
    int newNodeIdx = tree.size() - 1;
    tree.get(cur).next.put(c, newNodeIdx);
    if (newNode.len == 1) {
        newNode.link = 1;
        newNode.occ = 1;
        suff = newNodeIdx;
```

```
            return;
        }
        while (true) {
            cur = tree.get(cur).link;
            int curlen = tree.get(cur).len;
            if (pos - curlen - 1 >= 0 && buildStr.charAt(pos - curlen - 1) == c) {
                newNode.link = tree.get(cur).next.get(c);
                break;
            }
        }
        newNode.occ = 1;
        suff = newNodeIdx;
    }

    public int countDistinctPalindromes() {
        return tree.size() - 2; // excluding two roots
    }
}
```