STRING MANIPULATION - Interview Point Of view

Contents

1. Reverse a String

Efficient Solution: Use StringBuilder (O(n) time, O(n) space)
Why? StringBuilder's reverse() method is optimized and avoids extra loops.

```
public static String reverseString(String str) {
    return new StringBuilder(str).reverse().toString();
}
```

2. Check if a String is a Palindrome

Efficient Solution: Two-pointer approach (O(n) time, O(1) space)
Why? Avoids extra space used by recursion or string builders.

```
public static boolean isPalindrome(String str) {
    int left = 0, right = str.length() - 1;
    while (left < right) {
        if (str.charAt(left) != str.charAt(right)) return false;
        left++;
        right--;
    }
    return true;
}
```

3. Remove Duplicates from a String

Efficient Solution: Use a HashSet (O(n) time, O(n) space)
Why? Uses a HashSet for O(1) lookups.

```
public static String removeDuplicates(String str) {
    StringBuilder sb = new StringBuilder();
    HashSet<Character> seen = new HashSet<>();
    for (char ch : str.toCharArray()) {
        if (seen.add(ch)) sb.append(ch);
    }
    return sb.toString();
}
```

4. Find the First Non-Repeating Character

Efficient Solution: Use a frequency map (O(n) time, O(1) space)
Why? Only two passes over the string, avoiding nested loops

```
public static char firstNonRepeatingChar(String str) {
    int[] freq = new int[256]; // ASCII characters
    for (char ch : str.toCharArray()) freq[ch]++;

    for (char ch : str.toCharArray()) {
        if (freq[ch] == 1) return ch;
    }
    return '\0'; // No unique character found
}
```

5. Count the Occurrences of Each Character

Efficient Solution: Use an array (O(n) time, O(1) space)
Why? Uses a single pass and avoids sorting.

```
public static Map<Character, Integer> countCharacters(String str) {
    Map<Character, Integer> countMap = new HashMap<>();
    for (char ch : str.toCharArray()) {
        countMap.put(ch, countMap.getOrDefault(ch, 0) + 1);
    }
    return countMap;
}
```

6. Reverse Words in a Sentence

Efficient Solution: Use split() and StringBuilder (O(n) time, O(n) space)
Why? Splitting and reversing words directly is faster than handling each character.

```
public static String reverseWords(String sentence) {
    String[] words = sentence.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();
}
```

7. Check if Two Strings are Anagrams

Efficient Solution: Use frequency counting (O(n) time, O(1) space)
Why? Avoids sorting (O(n log n)), making it **O(n)** time complexity.

```java
public static boolean areAnagrams(String s1, String s2) {
    if (s1.length() != s2.length()) return false;
    int[] count = new int[26]; // Assuming lowercase letters

    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;
}
```

8. Find the Longest Substring Without Repeating Characters

Efficient Solution: Sliding window technique (O(n) time, O(min(n, alphabet)) space)
Why? Uses a single pass and a HashMap to track character positions efficiently.

```java
public static int lengthOfLongestSubstring(String s) {
    int maxLen = 0, left = 0;
    Map<Character, Integer> seen = new HashMap<>();

    for (int right = 0; right < s.length(); right++) {
        char ch = s.charAt(right);
        if (seen.containsKey(ch)) {
            left = Math.max(left, seen.get(ch) + 1);
        }
        seen.put(ch, right);
        maxLen = Math.max(maxLen, right - left + 1);
    }
    return maxLen;
}
```

9. Convert a String to an Integer (atoi Implementation)

Efficient Solution: Manual parsing (O(n) time, O(1) space)
Why? Efficient handling of overflow and signs.

```
public static int myAtoi(String str) {
    str = str.trim();
    if (str.isEmpty()) return 0;

    int i = 0, sign = 1, num = 0;
    if (str.charAt(0) == '-' || str.charAt(0) == '+') {
        sign = str.charAt(0) == '-' ? -1 : 1;
        i++;
    }

    while (i < str.length() && Character.isDigit(str.charAt(i))) {
        int digit = str.charAt(i) - '0';
        if (num > (Integer.MAX_VALUE - digit) / 10) {
            return sign == 1 ? Integer.MAX_VALUE : Integer.MIN_VALUE; // Handle overflow
        }
        num = num * 10 + digit;
        i++;
    }
    return num * sign;
}
```

10. Compress a String (Run-Length Encoding)

Efficient Solution: Two-pointer approach (O(n) time, O(n) space)
Why? Uses **O(n) time** with a single pass.

```
public static String compressString(String str) {
    StringBuilder sb = new StringBuilder();
    int i = 0, n = str.length();

    while (i < n) {
        char ch = str.charAt(i);
        int count = 0;
        while (i < n && str.charAt(i) == ch) {
            count++;
            i++;
        }
        sb.append(ch).append(count);
    }
    return sb.length() < str.length() ? sb.toString() : str;
}
```

11. Find the Most Frequent Character

Efficient Solution: Use a frequency array (O(n) time, O(1) space)
Why? Avoids sorting and maps, making it **O(n)**.

```java
public static char mostFrequentChar(String str) {
    int[] freq = new int[256]; // Supports ASCII characters
    char maxChar = '\0';
    int maxFreq = 0;

    for (char ch : str.toCharArray()) {
        freq[ch]++;
        if (freq[ch] > maxFreq) {
            maxFreq = freq[ch];
            maxChar = ch;
        }
    }
    return maxChar;
}
```

12. Find All Substrings of a Given String

Efficient Solution: Generate substrings using two loops (O(n²) time, O(1) space)
Why? Uses substring efficiently, avoiding redundant operations.

```java
public static List<String> findAllSubstrings(String str) {
    List<String> substrings = new ArrayList<>();
    int n = str.length();

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j <= n; j++) {
            substrings.add(str.substring(i, j));
        }
    }
    return substrings;
}
```

13. Check if a String is a Rotation of Another String

Efficient Solution: Use concatenation (O(n) time, O(n) space)
Why? Eliminates the need for nested loops.

```
public static boolean isRotation(String s1, String s2) {
    if (s1.length() != s2.length()) return false;
    return (s1 + s1).contains(s2);
}
```

14. Remove All White Spaces from a String

Efficient Solution: Use replaceAll() (O(n) time, O(1) space)
Why? Uses regex to remove all spaces in a single pass.

```
public static String removeWhitespaces(String str) {
    return str.replaceAll("\\s", "");
}
```

15. Check if a String is a Valid Shuffle of Two Strings

Efficient Solution: Sorting + Two-Pointer (O(n log n) time, O(n) space)
Why? Sorting makes comparison simpler.

```
public static boolean isValidShuffle(String s1, String s2, String result) {
    if (s1.length() + s2.length() != result.length()) return false;

    char[] temp = result.toCharArray();
    Arrays.sort(temp);
    char[] temp1 = (s1 + s2).toCharArray();
    Arrays.sort(temp1);

    return Arrays.equals(temp, temp1);
}
```

16. Convert a String to Title Case

Efficient Solution: Use split() and StringBuilder (O(n) time, O(n) space)
Why? Processes words in a single pass.

```
public static String toTitleCase(String str) {
    String[] words = str.toLowerCase().split("\\s+");
    StringBuilder sb = new StringBuilder();

    for (String word : words) {
        if (!word.isEmpty()) {
            sb.append(Character.toUpperCase(word.charAt(0)))
              .append(word.substring(1)).append(" ");
        }
    }
    return sb.toString().trim();
}
```

17. Find the Longest Common Prefix

Efficient Solution: Sort and compare first & last strings (O(n log n) time, O(1) space)
Why? Sorting helps quickly find the common prefix.

```
public static String longestCommonPrefix(String[] strs) {
    if (strs == null || strs.length == 0) return "";
    Arrays.sort(strs);
    String first = strs[0], last = strs[strs.length - 1];

    int i = 0;
    while (i < first.length() && first.charAt(i) == last.charAt(i)) i++;

    return first.substring(0, i);
}
```

18. Convert a String to a Character Array

Efficient Solution: Use toCharArray() (O(n) time, O(n) space)
Why? Most efficient way in Java.

```
public static char[] toCharArray(String str) {
    return str.toCharArray();
}
```

19. Replace Spaces with %20 (URL Encoding)

Efficient Solution: Use replace() (O(n) time, O(1) space)
Why? Avoids manual iteration.

```
public static String urlEncode(String str) {
    return str.replace(" ", "%20");
}
```

20. Convert a Sentence into an Acronym

Efficient Solution: Use split() and StringBuilder (O(n) time, O(n) space)
Why? Processes words efficiently.

```
public static String toAcronym(String sentence) {
    StringBuilder acronym = new StringBuilder();
    for (String word : sentence.split("\\s+")) {
        if (!word.isEmpty()) {
            acronym.append(Character.toUpperCase(word.charAt(0)));
        }
    }
    return acronym.toString();
}
```

21. Check if a String Contains Only Digits

Efficient Solution: Use regex (O(n) time, O(1) space)
Why? Single regex check avoids looping.

```
public static boolean isNumeric(String str) {
    return str.matches("\\d+");
}
```

22. Find the Number of Words in a String

Efficient Solution: Use split() (O(n) time, O(n) space)
Why? Uses regex to efficiently count words.

```
public static int countWords(String str) {
    return str.trim().isEmpty() ? 0 : str.trim().split("\\s+").length;
}
```

23. Remove a Given Character from a String

Efficient Solution: Use replace() (O(n) time, O(1) space)

Why? More optimized than manually iterating.

```
public static String removeCharacter(String str, char ch) {
    return str.replace(Character.toString(ch), "");
}
```

24. Find the Shortest Word in a String

Efficient Solution: Use split() and min() (O(n) time, O(n) space)
Why? Uses Java Streams for concise logic.

```
public static String shortestWord(String sentence) {
    return Arrays.stream(sentence.split("\\s+"))
                 .min(Comparator.comparingInt(String::length))
                 .orElse("");
}
```

25. Find the Longest Palindromic Substring

Efficient Solution: Expand Around Center (O(n²) time, O(1) space)
Why? Faster than brute force.

```
public static String longestPalindromicSubstring(String s) {
    if (s == null || s.length() < 1) return "";

    int start = 0, end = 0;
    for (int i = 0; i < s.length(); i++) {
        int len1 = expandAroundCenter(s, i, i);
        int len2 = expandAroundCenter(s, i, i + 1);
        int len = Math.max(len1, len2);

        if (len > end - start) {
            start = i - (len - 1) / 2;
            end = i + len / 2;
        }
    }
    return s.substring(start, end + 1);
}

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