Strings

Bhai ye **String Basics** agar strong ho gaye na, to DSA me 30–40% problems automatically easy ho jaati hain 🔥

Chal ab sab kuch **Java context me** samajhte hain — concept + example + intuition.

🔷 1. What is String in Java?

✅ Java me String kya hota hai?

Java me `String` ek **class** hai (`java.lang.String`), primitive nahi.

```
String s = "hello";
```

Under the hood → ye ek **char array** hi hota hai, but wrapped inside class.

🔶 C++ vs Java Difference

C++	Java
`char arr[]`	`String` class
Mutable	Immutable
Manual handling	Built-in methods

🔷 2. String Length

```
String s = "hello";
System.out.println(s.length());   // 5
```

⚠️ Dhyaan:

- Array → `arr.length`
- String → `s.length()`

🔷 3. Indexing

Java me string **0-based indexing** follow karta hai.

```
String s = "hello";

System.out.println(s.charAt(0));  // h
System.out.println(s.charAt(4));  // o
```

🔷 4. Traversal

Method 1: Using for loop

```
String s = "hello";

for(int i = 0; i < s.length(); i++) {
    System.out.println(s.charAt(i));
}
```

🔷 5. ASCII Values

Har character ka ek number hota hai.

```
char ch = 'A';
System.out.println((int) ch);   // 65
```

```
char ch = 'a';
System.out.println((int) ch);   // 97
```

🔥 Important ASCII Difference:

```
'a' - 'A' = 32
```

🔷 6. Character vs String

Character	String
'a'	"a"
Single quotes	Double quotes

Character	String
1 byte	Multiple chars

Example:

```
char ch = 'a';
String s = "a";
```

🔷 7. Mutable vs Immutable

🔴 String is Immutable

```
String s = "hello";
s.concat(" world");

System.out.println(s);  // still "hello"
```

Change nahi hua.

🔥 Correct way:

```
s = s.concat(" world");
```

🔷 8. 'a' vs "a"

```
'a'  → char
"a"  → String
```

```
System.out.println('a' + 1);  // 98
```

Kyun?

Because:

```
'a' = 97
97 + 1 = 98
```

🔷 9. 'a' + 1 kya hota hai?

```
char ch = 'a';
System.out.println(ch + 1);  // 98 (int)
```

Agar character chahiye:

```
System.out.println((char)(ch + 1));  // b
```

🔷 10. Uppercase ↔ Lowercase Conversion

Method 1: Built-in

```
String s = "HELLO";
System.out.println(s.toLowerCase());
```

Method 2: Using ASCII (Manual way)

```
char ch = 'A';

char lower = (char)(ch + 32);
System.out.println(lower);  // a
```

Uppercase:

```
char ch = 'a';
char upper = (char)(ch - 32);
System.out.println(upper);  // A
```

🔥 Now Practice Problems (Java Solutions)

✅ 1. LeetCode 344 – Reverse String

```
class Solution {
    public void reverseString(char[] s) {
        int left = 0;
        int right = s.length - 1;

        while(left < right) {
            char temp = s[left];
            s[left] = s[right];
            s[right] = temp;

            left++;
            right--;
        }
    }
}
```

🧠 Concept: Two pointer approach

Time: O(n)

Space: O(1)

✅ 2. To Lower Case

```
class Solution {
    public String toLowerCase(String s) {
        StringBuilder result = new StringBuilder();

        for(int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);

            if(ch >= 'A' && ch <= 'Z') {
                ch = (char)(ch + 32);
            }

            result.append(ch);
        }

        return result.toString();
    }
}
```

✅ 3. Maximum Number of Words Found in Sentences

```
class Solution {
    public int mostWordsFound(String[] sentences) {

        int maxWords = 0;

        for(String sentence : sentences) {
            int wordCount = 1;

            for(int i = 0; i < sentence.length(); i++) {
                if(sentence.charAt(i) == ' ') {
                    wordCount++;
                }
            }

            maxWords = Math.max(maxWords, wordCount);
        }

        return maxWords;
    }
}
```

🧠 Concept: Count spaces → words = spaces + 1

🚀 Important Interview Tips

1. Strings are immutable → use `StringBuilder` for heavy modifications
2. Always remember:

    ```
    '0' = 48
    'A' = 65
    'a' = 97
    ```

3. Use `(char)` casting carefully
4. Practice:
    - Palindrome string

    - Anagram
    - Frequency array
    - Substring problems

Agar tu chahe to next me:

- 🔥 StringBuilder deep dive
- 🔥 Sliding window on strings
- 🔥 Hashing on strings
- 🔥 Pattern problems
- 🔥 Interview level string roadmap

Bata next kya todna hai 😎

Bhai ab aa gaye **real weapon** pe 🔥

Java me heavy string manipulation = **StringBuilder**

Ye interview me bahut important hai.

🔥 1️⃣ Why StringBuilder?

Problem with String

```
String s = "";

for(int i = 0; i < 5; i++) {
    s = s + i;
}
```

Har iteration me:

- naya object banega
- purana garbage ho jayega

⚠️ Time complexity becomes **O(n²)**

✅ Solution → StringBuilder

```
StringBuilder sb = new StringBuilder();

for(int i = 0; i < 5; i++) {
   sb.append(i);
}

String result = sb.toString();
```

✔️ No new object each time

✔️ Fast

✔️ Mutable

Time complexity → **O(n)**

🔷 2️⃣ String vs StringBuilder vs StringBuffer

Feature	String	StringBuilder	StringBuffer
Mutable	❌	✅	✅
Thread Safe	❌	❌	✅
Speed	Slow	Fastest	Slower

👉 Interview me mostly use `StringBuilder`

🔷 3️⃣ Internal Working (Important)

Internally:

- Uses **dynamic char array**
- Default capacity = 16
- If overflow → capacity = `(old * 2) + 2`

Example:

```
StringBuilder sb = new StringBuilder();
System.out.println(sb.capacity());  // 16
```

🔷 4️⃣ Important Methods

✅ append()

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");

System.out.println(sb);  // Hello World
```

✅ insert()

```
sb.insert(5, " Java");
```

✅ delete()

```
sb.delete(0, 5);  // delete index 0 to 4
```

✅ deleteCharAt()

```
sb.deleteCharAt(2);
```

✅ reverse()

```
sb.reverse();
```

✅ setCharAt()

```
sb.setCharAt(0, 'A');
```

✅ charAt()

```
sb.charAt(1);
```

✅ length()

```
sb.length();
```

🔷 5️⃣ Capacity vs Length

```
StringBuilder sb = new StringBuilder("hello");

System.out.println(sb.length());    // 5
System.out.println(sb.capacity());  // 21 (16 + 5)
```

🔥 6️⃣ Interview Important Use Cases

🧠 1. Reverse a String

```
public String reverse(String s) {
    StringBuilder sb = new StringBuilder(s);
    return sb.reverse().toString();
}
```

🧠 2. Remove Duplicates

```
public String removeDuplicates(String s) {
    StringBuilder sb = new StringBuilder();

    for(char ch : s.toCharArray()) {
        if(sb.indexOf(String.valueOf(ch)) == -1) {
            sb.append(ch);
        }
    }

    return sb.toString();
}
```

🧠 3. Palindrome Check

```
public boolean isPalindrome(String s) {
    StringBuilder sb = new StringBuilder(s);
    return s.equals(sb.reverse().toString());
}
```

🧠 4. Toggle Case

```
public String toggleCase(String s) {
    StringBuilder sb = new StringBuilder();

    for(char ch : s.toCharArray()) {
        if(ch >= 'A' && ch <= 'Z')
            sb.append((char)(ch + 32));
        else if(ch >= 'a' && ch <= 'z')
            sb.append((char)(ch - 32));
        else
            sb.append(ch);
    }

    return sb.toString();
}
```

🔥 7️⃣ Important Concept

`+` vs `append()`

```
String s = "a" + "b";
```

Compiler optimize karta hai (StringBuilder internally use hota hai)

But inside loop:

```
s = s + i;  ❌ slow
```

Use:

```
sb.append(i);  ✅ fast
```

🔥 8️⃣ Trick Question (Interview)

```
StringBuilder sb1 = new StringBuilder("hello");
StringBuilder sb2 = sb1;

sb2.append(" world");

System.out.println(sb1);
```

Output?

👉 `hello world`

Because:

Reference same object ko point kar raha hai.

🔥 9️⃣ Performance Comparison

If n = 10^5

Method	Time
String +	❌ TLE possible
StringBuilder	✅ Fast

🚀 When to Use StringBuilder?

✔ When modifying string

✔ Inside loops

✔ When building long strings

✔ Competitive programming

🔥 Important Practice (Must Do)

1.
    a. Valid Palindrome

2.
    a. Reverse Words in a String

3.
    a. String Compression
4.
    a. Longest Common Prefix

Agar tu serious DSA prep kar raha hai (Scaler + Striver sheet 👀),

to next level hoga:

- 🔥 Strings + Sliding Window
- 🔥 HashMap on Strings
- 🔥 Rabin Karp
- 🔥 KMP Algorithm

Bata next kis level pe jana hai 😎

Perfect 👍 ab sirf **Java** me clear karte hain — clean, interview-ready explanation.

🔥 1️⃣ String Input in Java

Java me input ke liye mostly use hota hai:

```
Scanner sc = new Scanner(System.in);
```

🔷 next() vs nextLine()

❌ `next()`

```
String s = sc.next();
```

👉 Space aate hi ruk jaata hai.

Input:

```
Hello World
```

Output:

```
Hello
```

✅ `nextLine()`

```
String s = sc.nextLine();
```

👉 Puri line read karega including spaces.

Input:

```
Hello World
```

Output:

```
Hello World
```

🔥 2️⃣ Most Common Trap (VERY IMPORTANT)

```
int n = sc.nextInt();
String s = sc.nextLine();  // ❌ Problem
```

Why problem?

Because:

- `nextInt()` number read karta hai
- But newline (`\n`) buffer me chhod deta hai
- `nextLine()` usi newline ko read kar leta hai

✅ Correct Way

```
int n = sc.nextInt();
sc.nextLine();  // consume leftover newline
String s = sc.nextLine();
```

🔥 3️⃣ Handling Spaces Properly

Always use:

```
String s = sc.nextLine();
```

If full sentence input ho.

🔥 4️⃣ Multiple Test Cases (Strings)

Example Input:

```
3
hello world
java is good
i love dsa
```

Correct Code:

```
Scanner sc = new Scanner(System.in);

int t = Integer.parseInt(sc.nextLine());

while(t-- > 0) {
    String s = sc.nextLine();
    System.out.println(s);
}
```

Why `Integer.parseInt(sc.nextLine())`?

👉 Because then newline issue hi nahi aayega.

🔥 5️⃣ Practice Problems

✅ 1816. Truncate Sentence

Problem:

Sentence diya hai, first k words return karo.

Solution:

```
class Solution {
    public String truncateSentence(String s, int k) {

        String[] words = s.split(" ");

        StringBuilder sb = new StringBuilder();

        for(int i = 0; i < k; i++) {
            sb.append(words[i]);

            if(i != k - 1)
                sb.append(" ");
        }

        return sb.toString();
    }
}
```

Time: O(n)

Space: O(n)

✅ Length of Last Word

Example:

```
"Hello World"
Output: 5
```

🔷 Approach 1 (Best Way – Traverse from end)

```
class Solution {
    public int lengthOfLastWord(String s) {

        int length = 0;
        int i = s.length() - 1;

        // skip trailing spaces
        while(i >= 0 && s.charAt(i) == ' ')
```

```
            i--;

        // count last word
        while(i >= 0 && s.charAt(i) != ' ') {
            length++;
            i--;
        }

        return length;
    }
}
```

🔥 Ye best approach hai (O(1) space)

❌ Easy but extra space approach

```
String[] words = s.trim().split(" ");
return words[words.length - 1].length();
```

🚀 Interview Tips

1. Always prefer `nextLine()` for string
2. Handle newline issue carefully
3. Avoid `split()` if space optimization needed
4. Traverse from end in last word type problems

🧠 Next Level (Important for DSA)

Ab next kya karein?

- 🔥 String + Sliding Window
- 🔥 Frequency Array on Strings
- 🔥 HashMap based string problems
- 🔥 Substring problems (Very important)

Bata konsa level unlock karein 😎

Bhai ye topic **String ka backbone** hai 🔥

Anagram, Palindrome, Sliding Window, Permutation — sab me frequency lagegi.

Chal step-by-step solid karte hain.

🔥 1️⃣ Frequency Array (Size 26)

Use when:

- String me **only lowercase letters** ho (`a–z`)
- Fastest approach chahiye

🧠 Idea

ASCII trick:

```
'a' = 97
index = ch - 'a'
```

So:

```
int[] freq = new int[26];

for(char ch : s.toCharArray()) {
    freq[ch - 'a']++;
}
```

Example:

```
s = "aba"
freq:
a → 2
b → 1
```

⚡ Time & Space

Time → O(n)

Space → O(1) (constant 26)

🔥 Competitive programming me best.

🔥 2️⃣ HashMap Based Frequency

Use when:

- Uppercase + lowercase
- Digits
- Special characters
- Unicode

```
HashMap<Character, Integer> map = new HashMap<>();

for(char ch : s.toCharArray()) {
    map.put(ch, map.getOrDefault(ch, 0) + 1);
}
```

⚡ Time & Space

Time → O(n)

Space → O(n)

🔥 3️⃣ Case Sensitive vs Case Insensitive

🔷 Case Sensitive

```
'A' != 'a'
```

Just count normally.

🔷 Case Insensitive

Convert first:

```
s = s.toLowerCase();
```

or

```
char ch = Character.toLowerCase(ch);
```

🔥 4️⃣ Used In (VERY IMPORTANT)

Problem	Concept
Anagram	Same frequency
Palindrome	At most 1 odd freq
Permutation	Compare freq arrays
Sliding window	Maintain freq

🟢 Practice Problems

✅ 242. Valid Anagram

🔥 Best Approach (Frequency Array)

```
class Solution {
    public boolean isAnagram(String s, String t) {

        if(s.length() != t.length())
            return false;

        int[] freq = new int[26];

        for(int i = 0; i < s.length(); i++) {
            freq[s.charAt(i) - 'a']++;
            freq[t.charAt(i) - 'a']--;
        }

        for(int count : freq) {
            if(count != 0)
                return false;
        }

        return true;
    }
}
```

🔥 Smart trick:

- ek loop me increment + decrement

Time: O(n)

Space: O(1)

✅ First Unique Character in a String

Problem:

Return index of first character whose freq = 1

```
class Solution {
    public int firstUniqChar(String s) {

        int[] freq = new int[26];

        for(char ch : s.toCharArray()) {
            freq[ch - 'a']++;
        }

        for(int i = 0; i < s.length(); i++) {
            if(freq[s.charAt(i) - 'a'] == 1)
                return i;
        }

        return -1;
    }
}
```

✅ Sort Characters By Frequency

Example:

```
"tree"
```

```
Output: "eert"
```

🔥 Approach: HashMap + MaxHeap

```
import java.util.*;

class Solution {
    public String frequencySort(String s) {

        HashMap<Character, Integer> map = new HashMap<>();

        for(char ch : s.toCharArray()) {
            map.put(ch, map.getOrDefault(ch, 0) + 1);
        }

        PriorityQueue<Character> pq =
            new PriorityQueue<>((a, b) → map.get(b) - map.get(a));

        pq.addAll(map.keySet());

        StringBuilder sb = new StringBuilder();

        while(!pq.isEmpty()) {
            char ch = pq.poll();
            int count = map.get(ch);

            while(count-- > 0) {
                sb.append(ch);
            }
        }

        return sb.toString();
    }
}
```

Time: O(n log n)

🔥 Palindrome Trick (Important Concept)

For palindrome:

- Even length → all freq even
- Odd length → at most 1 odd freq allowed

Check:

```
int oddCount = 0;

for(int count : freq) {
    if(count % 2 != 0)
        oddCount++;
}

if(oddCount > 1)
    return false;
```

🔥 Interview Pattern Recognition

If question me aaye:

- "rearrange"
- "check if possible"
- "permutation"
- "anagram"
- "unique"
- "count"

👉 80% chance → frequency array lagega.

🚀 Ab Next Level (Important)

Tu Striver sheet follow kar raha hai, to next unlock kare:

- 🔥 Sliding Window on Strings
- 🔥 Longest Substring Without Repeating
- 🔥 Minimum Window Substring
- 🔥 Find All Anagrams in String

Ye sab frequency + hashmap combo hai.

Bata next konsa todna hai 😎

Bhai ye topic **interview gold mine** hai 🔥

FAANG me directly ya indirectly palindrome aata hi hai.

Main har variation ko **intuition + clean Java code + detailed line-by-line comments** ke saath samjha raha hoon.

🔥 1️⃣ Two Pointer Palindrome (Base Technique)

🧠 Core Idea

- Ek pointer start se
- Ek pointer end se
- Compare karte jao
- Mismatch → palindrome nahi

Time: **O(n)**

Space: **O(1)**

✅ 125. Valid Palindrome

(Ignore special chars + case insensitive)

Example:

```
"A man, a plan, a canal: Panama"
→ true
```

🔥 Java Solution (Detailed Comments)

```
class Solution {
    public boolean isPalindrome(String s) {

        // left pointer starts from beginning
        int left = 0;
```

```
// right pointer starts from end
int right = s.length() - 1;

// loop until pointers cross
while(left < right) {

    // get left character
    char l = s.charAt(left);

    // get right character
    char r = s.charAt(right);

    // if left character is not letter or digit
    if(!Character.isLetterOrDigit(l)) {
        left++;      // skip it
        continue;    // move to next iteration
    }

    // if right character is not letter or digit
    if(!Character.isLetterOrDigit(r)) {
        right--;     // skip it
        continue;
    }

    // convert both to lowercase for case-insensitive comparison
    l = Character.toLowerCase(l);
    r = Character.toLowerCase(r);

    // if characters do not match
    if(l != r) {
        return false;  // not palindrome
    }

    // move both pointers inward
    left++;
    right--;
}

// if loop completes, it is palindrome
```

```
        return true;
    }
}
```

🔥 Interview tip:

`Character.isLetterOrDigit()` is cleanest way.

🔥 2️⃣ Valid Palindrome II

(Allow 1 character removal)

Example:

```
"abca" → true (remove 'c')
```

🧠 Idea

- Normal two pointer
- First mismatch pe:
    - skip left OR
    - skip right
- Check both possibilities

✅ Java Code (Detailed)

```
class Solution {

    public boolean validPalindrome(String s) {

        int left = 0;
        int right = s.length() - 1;

        while(left < right) {

            if(s.charAt(left) != s.charAt(right)) {

                // try skipping left character
                boolean skipLeft = checkPalindrome(s, left + 1, right);

                // try skipping right character
                boolean skipRight = checkPalindrome(s, left, right - 1);

                return skipLeft || skipRight;
            }

            left++;
            right--;
        }

        return true;
    }

    // helper function to check if substring is palindrome
    private boolean checkPalindrome(String s, int left, int right) {

        while(left < right) {

            if(s.charAt(left) != s.charAt(right)) {
                return false;
            }

            left++;
            right--;
        }

        return true;
    }
}
```

Time: O(n)

Space: O(1)

🔥 Very common Google/Amazon question.

🔥 3️⃣ Longest Palindromic Substring (FAANG Level)

Problem:

Return longest palindromic substring.

Example:

```
"babad"
→ "bab" or "aba"
```

🧠 Best Interview Approach → Expand Around Center

Because:

- Every palindrome expands from center
- Total centers = 2n - 1
    - odd length center
    - even length center

Time: $O(n^2)$

Space: $O(1)$

✅ Java Code (Detailed Comments)

```
class Solution {

    public String longestPalindrome(String s) {

        // if string is empty or length 1
        if(s == null || s.length() < 2) {
            return s;  // already palindrome
        }

        // start index of longest palindrome found
        int start = 0;

        // end index of longest palindrome found
        int end = 0;
        // iterate over each character as potential center
        for(int i = 0; i < s.length(); i++) {

            // check odd length palindrome (center at i)
            int len1 = expandFromCenter(s, i, i);

            // check even length palindrome (center between i and i+1)
            int len2 = expandFromCenter(s, i, i + 1);

            // take maximum length
            int maxLen = Math.max(len1, len2);

            // if we found longer palindrome than previous
            if(maxLen > end - start) {

                // calculate new start index
                start = i - (maxLen - 1) / 2;

                // calculate new end index
                end = i + maxLen / 2;
            }
        }

        // return substring from start to end inclusive
        return s.substring(start, end + 1);
    }

    // helper method to expand around center
    private int expandFromCenter(String s, int left, int right) {

        // expand while valid and characters match
        while(left >= 0 && right < s.length()
              && s.charAt(left) == s.charAt(right)) {

            left--;   // move left outward
            right++;  // move right outward
        }

        // return length of palindrome
        return right - left - 1;
    }
}
```

🔥 Why Expand Around Center is Best?

Brute force → O(n³)

DP → O(n²) space

Center expand → O(n²) time, O(1) space ✅ best practical

🔥 Summary Table

Problem	Technique
Valid Palindrome	Two pointer
Ignore special chars	Character methods
Almost Palindrome	Skip once
Longest Palindrome	Expand center

🚀 Next Level (Real FAANG Prep)

If tu serious hai:

- 🔥 Palindromic Subsequence (DP)
- 🔥 Count Palindromic Substrings
- 🔥 Manacher's Algorithm (O(n))
- 🔥 Sliding Window + Palindrome

Bata next kaunsa unlock karein 😎

Bhai 🔥 **Two Pointer on Strings = interview ka default weapon**

Sliding window alag cheez hai,

par simple **left–right pointer** pattern strings me bahut use hota hai.

Aaj hum 3 heavy patterns dekhenge:

1. ✅ Left–Right compare
2. ✅ Shrinking window
3. ✅ Build result smartly (reverse words type)

Sab Java me + clean explanation.

🔥 1️⃣ Pattern: Left – Right (Basic Form)

Structure:

```
int left = 0;
int right = s.length() - 1;

while(left < right) {
   // compare or swap
   left++;
   right--;
}
```

Used in:

- Reverse string
- Palindrome
- Compare two strings from ends

🔥 2️⃣ Pattern: Shrinking Window

Use when:

- Extra spaces remove karne ho
- Trim karna ho
- Specific character remove karna ho

Structure:

```
while(left <= right && condition) left++;
while(left <= right && condition) right--;
```

🔥 3️⃣ Pattern: Compare / Merge Strings

Use when:

- 2 strings simultaneously traverse karne ho

- Alternating merge
- Backspace compare

🟢 Practice Problems

✅ 151. Reverse Words in a String (Important)

Example:

```
Input: "  the sky   is blue  "
Output: "blue is sky the"
```

🔥 Key Points:

- Extra spaces remove
- Words reverse order me
- Words ke andar characters same

🧠 Approach (Two Pointer + StringBuilder)

1. Trim spaces
2. End se start tak traverse
3. Word extract karo
4. Append to result

✅ Java Code (Detailed but clean)

```
class Solution {

    public String reverseWords(String s) {

        StringBuilder result = new StringBuilder();

        int i = s.length() - 1;

        while(i >= 0) {
            // skip trailing spaces
            while(i >= 0 && s.charAt(i) == ' ')
                i--;

            if(i < 0) break;

            int j = i;

            // find start of word
            while(i >= 0 && s.charAt(i) != ' ')
                i--;

            // append word
            result.append(s.substring(i + 1, j + 1));

            result.append(" ");
        }

        // remove last extra space
        return result.toString().trim();
    }
}
```

Time: O(n)

Space: O(n)

🔥 Interview me ye approach impress karta hai.

✅ Reverse String (LeetCode 344)

Input: char[]

🔥 Two Pointer Swap

```
class Solution {
    public void reverseString(char[] s) {

        int left = 0;
        int right = s.length - 1;
        while(left < right) {

            char temp = s[left];
            s[left] = s[right];
            s[right] = temp;

            left++;
            right--;
        }
    }
}
```

Time: O(n)

Space: O(1)

✅ Merge Strings Alternately

Example:

```
word1 = "abc"
word2 = "pqr"

Output: "apbqcr"
```

🧠 Pattern

- Traverse both strings
- Pick char from each
- Append remainder

✅ Java Code

```
class Solution {

    public String mergeAlternately(String word1, String word2) {

        StringBuilder sb = new StringBuilder();
        int i = 0, j = 0;

        while(i < word1.length() && j < word2.length()) {

            sb.append(word1.charAt(i));
            sb.append(word2.charAt(j));

            i++;
            j++;
        }

        // append remaining part
        while(i < word1.length())
            sb.append(word1.charAt(i++));

        while(j < word2.length())
            sb.append(word2.charAt(j++));

        return sb.toString();
    }
}
```

Time: O(n + m)

🔥 Remove Character Pattern (Very Important)

Example:

Remove all 'a'

```
public String removeChar(String s, char target) {

    StringBuilder sb = new StringBuilder();

    for(int i = 0; i < s.length(); i++) {
        if(s.charAt(i) != target) {
            sb.append(s.charAt(i));
        }
    }

    return sb.toString();
}
```

🔥 Used in filtering problems.

🔥 Compare Strings Pattern

Example:

Backspace string compare type questions

Use:

- Two pointers from end
- Skip characters dynamically

🚀 Pattern Recognition Cheat Sheet

If question says:

Keywords	Pattern
reverse	left-right
compare from both ends	left-right
remove spaces	shrinking window
merge	dual pointer
build new string	StringBuilder

🔥 Important for Striver Sheet

Ab next level me ye aayega:

- 🔥 Longest Substring Without Repeating
- 🔥 Minimum Window Substring
- 🔥 Find All Anagrams in String
- 🔥 Sliding Window on Strings

Ye sab Two Pointer + HashMap combo hai 🔥

Bata next kaunsa todna hai?

Sliding window pe chalte hain? 😎

Bhai 🔥 ab aa gaye **FAANG ka favourite topic** pe —

Sliding Window on Strings

Yeh topic agar strong ho gaya na, to Google/Amazon ke 60–70% string questions crack ho jaate hain.

Main ise 3 levels me todunga:

1. ✅ Fixed Window
2. ✅ Variable Window
3. ✅ Hard Level (Minimum Window Substring)

Sab Java me + clear logic.

🔥 1️⃣ Sliding Window Basics

Structure (Variable Window)

```
int left = 0;

for(int right = 0; right < s.length(); right++) {

    // include s[right] in window

    while(condition is invalid) {
        // shrink from left
        left++;
    }

    // update answer
}
```

🔥 2️⃣ Fixed Window Pattern

Window size constant hota hai (k size).

Used in:

- Count occurrences
- Anagram check
- Substring of size k

🔥 3️⃣ Variable Window Pattern

Window size dynamic hota hai.

Used in:

- Longest substring
- Minimum window
- At most k distinct characters

🟢 Practice Problems

✅ 3. Longest Substring Without Repeating Characters

Example:

```
"abcabcbb"
Output: 3 ("abc")
```

🧠 Idea

- HashSet maintain karo
- Duplicate mile → shrink window
- Max length update karo

✅ Java Code

```
class Solution {

    public int lengthOfLongestSubstring(String s) {
        int left = 0;
        int maxLen = 0;

        HashSet<Character> set = new HashSet<>();

        for(int right = 0; right < s.length(); right++) {

            // if duplicate found, shrink window
            while(set.contains(s.charAt(right))) {
                set.remove(s.charAt(left));
                left++;
            }

            // add current character
            set.add(s.charAt(right));

            // update max length
            maxLen = Math.max(maxLen, right - left + 1);
        }

        return maxLen;
    }
}
```

Time: O(n)

Space: O(1) (max 128 chars)

🔥 FAANG classic question.

✅ Permutation in String

Check if s2 contains permutation of s1.

Example:

```
s1 = "ab"
s2 = "eidbaooo"
Output: true
```

🧠 Idea

- Fixed window size = s1.length()
- Frequency array use karo
- Window slide karte jao

✅ Java Code

```
class Solution {

    public boolean checkInclusion(String s1, String s2) {

        if(s1.length() > s2.length())
            return false;

        int[] freq = new int[26];

        // count s1 chars
        for(char ch : s1.toCharArray())
            freq[ch - 'a']++;

        int left = 0;

        for(int right = 0; right < s2.length(); right++) {

            freq[s2.charAt(right) - 'a']--;

            // window size exceed
            if(right - left + 1 > s1.length()) {
                freq[s2.charAt(left) - 'a']++;
                left++;
            }

            // check if all zero
            if(allZero(freq))
                return true;
        }

        return false;
    }

    private boolean allZero(int[] freq) {
        for(int num : freq)
            if(num != 0)
                return false;
        return true;
    }
}
```

Time: O(26*n) ≈ O(n)

✅ Find All Anagrams in a String

Same concept, but return all start indices.

✅ Java Code

```
class Solution {

    public List<Integer> findAnagrams(String s, String p) {

        List<Integer> result = new ArrayList<>();

        if(p.length() > s.length())
            return result;

        int[] freq = new int[26];

        for(char ch : p.toCharArray())
            freq[ch - 'a']++;

        int left = 0;

        for(int right = 0; right < s.length(); right++) {

            freq[s.charAt(right) - 'a']--;

            if(right - left + 1 > p.length()) {
                freq[s.charAt(left) - 'a']++;
                left++;
            }

            if(allZero(freq))
                result.add(left);
        }

        return result;
    }

    private boolean allZero(int[] freq) {
        for(int num : freq)
            if(num != 0)
                return false;
        return true;
    }
}
```

🔥 Hard Level

✅ Minimum Window Substring (Google Favorite)

Example:

```
s = "ADOBECODEBANC"
t = "ABC"

Output: "BANC"
```

🧠 Idea

- Frequency map for t
- Expand right
- Decrease count

- When valid → shrink from left
- Track minimum

✅ Java Code

```
class Solution {

    public String minWindow(String s, String t) {

        if(s.length() < t.length())
            return "";

        int[] freq = new int[128];

        for(char ch : t.toCharArray())
            freq[ch]++;

        int left = 0;
        int count = t.length();
        int minLen = Integer.MAX_VALUE;
        int start = 0;

        for(int right = 0; right < s.length(); right++) {

            if(freq[s.charAt(right)] > 0)
                count--;

            freq[s.charAt(right)]--;

            while(count == 0) {

                if(right - left + 1 < minLen) {
                    minLen = right - left + 1;
                    start = left;
                }

                freq[s.charAt(left)]++;

                if(freq[s.charAt(left)] > 0)
```

```
                    count++;

                left++;
            }
        }

        return minLen == Integer.MAX_VALUE
                ? ""
                : s.substring(start, start + minLen);
    }
}
```

Time: O(n)

Space: O(1)

🔥 This is true FAANG level sliding window.

🔥 Sliding Window Pattern Summary

Problem Type	Window
Fixed length	Permutation / Anagram
No duplicates	Longest substring
Min satisfy condition	Minimum window

🚀 Final Advice

If tu Google-level prep kar raha hai:

Next topics:

- 🔥 Longest Substring with K Distinct
- 🔥 At Most K Replacement
- 🔥 Subarray Sum Equals K (numbers version)
- 🔥 Rabin Karp (pattern matching)

Tu DSA me serious jaa raha hai 👀

Next kaunsa todna hai?

Bhai 🔥 ye topic interview me bahut baar indirectly pucha jata hai.

Log confuse ho jaate hain kyunki naam similar hai — **substring vs subsequence**.

Aaj isko crystal clear karte hain.

🔥 Substring vs Subsequence (CORE DIFFERENCE)

✅ Substring

👉 **Continuous / contiguous** part of string.

Example:

```
s = "abcde"

Valid substrings:
"abc"
"bcd"
"cde"
"de"

Invalid:
"ace" ❌ (gap hai)
```

Definition:

- Continuous hona zaroori hai.
- Order same hi hoga (obviously).

✅ Subsequence

👉 Order same hona chahiye,

👉 BUT gaps allowed.

Example:

```
s = "abcde"

Valid subsequences:
"ace"  ✅
```

```
"bd"  ✅
"abc" ✅
```

Remove characters kar sakte ho,

reorder nahi kar sakte.

🔥 Key Comparison Table

Feature	Substring	Subsequence
Continuous	✅	❌
Order same	✅	✅
Gaps allowed	❌	✅
Typical technique	Sliding Window	DP / Two pointer

🔥 Interview Pattern Recognition

If question me aaye:

- “contiguous”
- “subarray”
- “substring”

👉 Sliding Window / Two Pointer

If aaye:

- “remove characters”
- “can we form”
- “common subsequence”
- “LCS”

👉 DP / Two pointer

🟢 Practice Problems

✅ 392. Is Subsequence

Example:

```
s = "abc"
t = "ahbgdc"
Output: true
```

🧠 Idea

- Two pointer
- s ko match karte jao t me

✅ Java Code

```
class Solution {

    public boolean isSubsequence(String s, String t) {

        int i = 0; // pointer for s
        int j = 0; // pointer for t

        // traverse t
        while(i < s.length() && j < t.length()) {

            // if characters match
            if(s.charAt(i) == t.charAt(j)) {
                i++; // move in s
            }

            j++; // always move in t
        }

        // if we matched all characters of s
        return i == s.length();
    }
}
```

Time: O(n)

Space: O(1)

🔥 Very common Amazon question.

🔥 Longest Common Subsequence (LCS)

Example:

```
text1 = "abcde"
text2 = "ace"

Output: 3
```

Common subsequence = "ace"

🧠 Why DP?

Because:

- Choices at every step
- Include / exclude pattern

🔥 DP Idea

Let:

```
dp[i][j] = LCS length of
text1[0..i-1] and text2[0..j-1]
```

Transition:

If chars match:

```
dp[i][j] = 1 + dp[i-1][j-1]
```

Else:

```
dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

✅ Java Code

```
class Solution {

    public int longestCommonSubsequence(String text1, String text2) {

        int n = text1.length();
        int m = text2.length();

        int[][] dp = new int[n + 1][m + 1];

        for(int i = 1; i <= n; i++) {
            for(int j = 1; j <= m; j++) {

                if(text1.charAt(i - 1) == text2.charAt(j - 1)) {

                    dp[i][j] = 1 + dp[i - 1][j - 1];

                } else {

                    dp[i][j] = Math.max(
                        dp[i - 1][j],
                        dp[i][j - 1]
                    );
                }
            }
        }

        return dp[n][m];
    }
}
```

Time: O(n × m)

Space: O(n × m)

🔥 FAANG standard DP problem.

🔥 Longest Palindromic Subsequence

Example:

```
"bbbab"
Output: 4
```

Subsequence = "bbbb"

🧠 Important Trick

Longest Palindromic Subsequence =

LCS of:

```
s and reverse(s)
```

✅ Java Code

```
class Solution {

    public int longestPalindromeSubseq(String s) {

        String rev = new StringBuilder(s).reverse().toString();

        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) == rev.charAt(j - 1)) {

                    dp[i][j] = 1 + dp[i - 1][j - 1];

                } else {

                    dp[i][j] = Math.max(
                        dp[i - 1][j],
                        dp[i][j - 1]
                    );
                }
            }
        }

        return dp[n][n];
    }
}
```

Time: $O(n^2)$

🔥 Google/Amazon me frequently pucha gaya.

🚀 Concept Clarity Booster

Substring → Continuous → Sliding Window

Subsequence → Gaps allowed → DP

🔥 Next Level Topics (Very Important)

- 🔥 Count Distinct Subsequences
- 🔥 Edit Distance
- 🔥 Shortest Common Supersequence
- 🔥 KMP (Substring search)
- 🔥 Rabin Karp

Tu ab DSA me serious FAANG track pe aa gaya hai 😎

Next DP level kholna hai ya pattern matching?

Bhai 🔥 ab aa gaye **String Matching Algorithms** pe —

ye interview me "bonus points" dete hain kyunki 90% log sirf naive approach jaante hain.

Aaj hum 3 cheeze cover karenge:

1. ✅ Naive Matching
2. ✅ KMP (LPS array samajh ke)
3. ✅ Rabin Karp (rolling hash basics)

Phir practice:

-
    1. Find the Index of the First Occurrence in a String
- Shortest Palindrome (KMP trick)

🔥 1️⃣ Naive String Matching

🧠 Idea

For every index in `haystack`:

- Check if substring matches `needle`

Time Complexity:

Worst case → **O(n × m)**

✅ Java (Naive)

```
class Solution {
    public int strStr(String haystack, String needle) {

        int n = haystack.length();
        int m = needle.length();

        for(int i = 0; i <= n - m; i++) {

            int j = 0;

            while(j < m && haystack.charAt(i + j) == needle.charAt(j)) {
                j++;
            }

            if(j == m)
                return i;
        }

        return -1;
    }
}
```

✔️ Simple

❌ Slow in worst case

🔥 2️⃣ KMP Algorithm (Important)

KMP avoids re-checking characters.

🧠 Core Idea

When mismatch happens:

👉 pattern ko smartly shift karo

👉 using LPS (Longest Prefix which is also Suffix)

🔥 LPS Array

For pattern:

```
"ababaca"
```

LPS:

```
[0 0 1 2 3 0 1]
```

Meaning:

At index i,

LPS[i] = longest proper prefix == suffix length

🔥 Step 1: Build LPS

✅ Java Code (Detailed)

```
private int[] buildLPS(String pattern) {

    int n = pattern.length();

    int[] lps = new int[n];
    int len = 0;   // length of previous longest prefix suffix
    int i = 1;     // start from second character

    while(i < n) {

        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;
}
```

Time: O(m)

🔥 Step 2: KMP Search

✅ Java Code

```
class Solution {

    public int strStr(String text, String pattern) {

        if(pattern.length() == 0)
            return 0;

        int[] lps = buildLPS(pattern);
        int i = 0; // pointer for text
        int j = 0; // pointer for pattern

        while(i < text.length()) {

            if(text.charAt(i) == pattern.charAt(j)) {
                i++;
                j++;
            }

            if(j == pattern.length()) {
                return i - j;  // match found
            }

            else if(i < text.length()
                    && text.charAt(i) != pattern.charAt(j)) {

                if(j != 0)
                    j = lps[j - 1];
                else
                    i++;
            }
        }

        return -1;
    }

    private int[] buildLPS(String pattern) {
        int n = pattern.length();
        int[] lps = new int[n];
        int len = 0;
        int i = 1;

        while(i < n) {
            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;
    }
}
```

Time: **O(n + m)**

🔥 Interview bonus guaranteed.

🔥 3️⃣ Rabin Karp (Rolling Hash Basics)

Instead of comparing characters:

👉 Compare hash values.

If hash match → verify string.

🧠 Idea

Hash of substring:

```
hash = (previous_hash * base + new_char) % mod
```

When window moves:

- remove left char
- add right char

Time: Average O(n)

⚠️ Collision possible

But rare if mod large.

🔥 28. Find the Index of the First Occurrence

Best answer in interview:

- Naive → acceptable
- KMP → impressive
- Rabin Karp → bonus

🔥 Shortest Palindrome (KMP Trick)

Problem:

Add characters in front to make shortest palindrome.

Example:

```
"aacecaaa"
Output: "aaacecaaa"
```

🧠 Trick

1. Reverse string
2. Make new string:

```
s + "#" + reverse(s)
```

1. Build LPS on this
2. Last value of LPS = longest palindromic prefix

✅ Java Code

```
class Solution {

    public String shortestPalindrome(String s) {

        String rev = new StringBuilder(s).reverse().toString();

        String combined = s + "#" + rev;
        int[] lps = buildLPS(combined);

        int longestPrefix = lps[lps.length - 1];

        String addPart = rev.substring(0, s.length() - longestPrefix);

        return addPart + s;
    }

    private int[] buildLPS(String str) {

        int n = str.length();
        int[] lps = new int[n];

        int len = 0;
        int i = 1;

        while(i < n) {

            if(str.charAt(i) == str.charAt(len)) {
                len++;
                lps[i] = len;
                i++;
            } else {
                if(len != 0)
                    len = lps[len - 1];
                else {
                    lps[i] = 0;
                    i++;
                }
            }
        }

        return lps;
    }
}
```

Time: O(n)

🔥 This is high-level interview solution.

🚀 When to Use What?

Situation	Use
Small constraints	Naive
Competitive programming	KMP
Multiple pattern search	Rabin Karp
Palindrome prefix	KMP trick

🔥 Final Advice

Ab tum string me cover kar chuke ho:

- Two pointer
- Sliding window
- Frequency
- DP (LCS)
- KMP

Next FAANG-level topics:

- 🔥 Z Algorithm
- 🔥 Manacher's Algorithm
- 🔥 Suffix Array
- 🔥 Rolling Hash advanced
- 🔥 Aho-Corasick

Tu kaafi deep aa chuka hai 😎

Next algorithm todna hai ya full string roadmap bana du?

Bhai 🔥 ye topic simple lagta hai but interviews me tricky hota hai —

especially jab **custom comparator** lagana padta hai.

Chalo step-by-step strong karte hain.

🔥 1️⃣ Lexicographical Order Kya Hota Hai?

👉 Dictionary order.

Compare character by character using ASCII / Unicode value.

Example:

```
"apple" < "banana"
```

Why?

- Compare first char:
  - 'a' < 'b' → done

Example:

```
"app" < "apple"
```

Why?

- First 3 characters same
- Shorter string smaller

🔥 ASCII Insight (Important)

```
'A' = 65
'Z' = 90
'a' = 97
'z' = 122
```

So:

```
"Zebra" < "apple"
```

Because:

```
'Z'(90) < 'a'(97)
```

⚠️ Case sensitive by default.

🔥 Java String Comparison

```
s1.compareTo(s2);
```

Returns:

- Negative → s1 < s2
- Positive → s1 > s2
- 0 → equal

Example:

```
"apple".compareTo("banana");  // negative
```

🔥 Case Insensitive Compare

```
s1.compareToIgnoreCase(s2);
```

🔥 2️⃣ Custom Sorting (Important)

Use:

```
Arrays.sort(array, comparator);
```

OR

```
Collections.sort(list, comparator);
```

Example:

```
Arrays.sort(arr, (a, b) → a.length() - b.length());
```

🔥 Practice Problems

✅ 937. Reorder Data in Log Files

Problem Summary

Logs:

```
let1 art can
dig1 8 1 5 1
let2 own kit dig
dig2 3 6
```

Rules:

1. Letter-logs come before digit-logs
2. Letter-logs sorted lexicographically by content
3. If tie → sort by identifier
4. Digit-logs maintain original order

🧠 Idea

- Check if log is digit or letter
- Custom comparator
- Digit logs → push to bottom
- Maintain relative order

✅ Java Code

```
import java.util.*;

class Solution {

    public String[] reorderLogFiles(String[] logs) {

        Arrays.sort(logs, (a, b) → {

            String[] splitA = a.split(" ", 2);
            String[] splitB = b.split(" ", 2);

            boolean isDigitA = Character.isDigit(splitA[1].charAt(0));
            boolean isDigitB = Character.isDigit(splitB[1].charAt(0));

            // Case 1: both letter logs
            if(!isDigitA && !isDigitB) {

                int cmp = splitA[1].compareTo(splitB[1]);

                if(cmp != 0)
                    return cmp;

                return splitA[0].compareTo(splitB[0]);
            }

            // Case 2: one digit, one letter
            if(!isDigitA && isDigitB)
                return -1;

            if(isDigitA && !isDigitB)
                return 1;

            // Case 3: both digit → keep order
            return 0;
        });

        return logs;
    }
}
```

Time: O(n log n)

🔥 Interview me comparator clarity bahut important hoti hai.

✅ Verifying an Alien Dictionary

Problem:

Given custom order of characters.

Check if words sorted according to that order.

🧠 Idea

1. Make map: char → rank
2. Compare adjacent words

3. Character by character compare using rank

✅ Java Code

```
class Solution {

    public boolean isAlienSorted(String[] words, String order) {

        int[] rank = new int[26];

        // assign rank to each character
        for(int i = 0; i < order.length(); i++) {
            rank[order.charAt(i) - 'a'] = i;
        }

        for(int i = 0; i < words.length - 1; i++) {

            String w1 = words[i];
            String w2 = words[i + 1];

            int minLen = Math.min(w1.length(), w2.length());
            boolean foundDifference = false;

            for(int j = 0; j < minLen; j++) {

                char c1 = w1.charAt(j);
                char c2 = w2.charAt(j);

                if(c1 != c2) {

                    if(rank[c1 - 'a'] > rank[c2 - 'a'])
                        return false;

                    foundDifference = true;
                    break;
                }
            }

            // edge case: prefix issue
            if(!foundDifference && w1.length() > w2.length())
```

```
                return false;
        }

        return true;
    }
}
```

Time: O(total characters)

🔥 Prefix case important:

```
"apple"
"app"
```

Invalid.

🔥 Interview Patterns

If question me aaye:

Keyword	Use
Custom order	Rank array
Dictionary order	compareTo
Reorder logs	Comparator
Sort by multiple keys	Nested comparison

🔥 Important Concepts Recap

- Lexicographical = dictionary order
- Compare char by char
- Shorter prefix smaller
- Custom sorting → comparator
- Stable ordering important in some cases

🚀 Next Level Topics

Ab tumne cover kar liya:

- Two pointer
- Sliding window
- Frequency
- DP
- KMP
- Sorting

Next heavy topics ho sakte hain:

- 🔥 Trie (very powerful in strings)
- 🔥 Z Algorithm
- 🔥 Suffix Array
- 🔥 Advanced Rolling Hash
- 🔥 String DP deep dive

Tu ab strong DSA string track pe hai 😎

Next kya unlock karein?

Bhai ye topic interview me kaafi baar aata hai 🔥

Yaha interviewer check karta hai:

- In-place modification samajhta hai ya nahi
- Two pointer ka control
- StringBuilder vs char[] use karna aata hai ya nahi

Chal step by step samajhte hain 👇

🔷 1️⃣ String Modification Basics (Java Context)

⚠️ Java String **immutable** hoti hai

Matlab:

```
String s = "hello";
s = s + "x";   // new object banega
```

Agar modify karna hai efficiently:

- `StringBuilder`
- `char[]`

🔷 Insert Character

```
StringBuilder sb = new StringBuilder("helo");
sb.insert(3, 'l');  // hel lo → hello
```

🔷 Delete Character

```
sb.deleteCharAt(3);
```

🔷 Replace Character

```
sb.setCharAt(0, 'H');
```

🔥 2️⃣ LeetCode 443 — String Compression (IMPORTANT)

📌 Problem

Given `char[] chars`, compress in-place.

Example:

```
Input:  ["a","a","b","b","c","c","c"]
Output: ["a","2","b","2","c","3"]
Return length = 6
```

🔥 Approach (Two Pointer)

- One pointer → read
- One pointer → write

✅ Java Code (Line by Line Detailed Comments)

```
class Solution {
    public int compress(char[] chars) {

        int write = 0;  // position where we write compressed output
        int read = 0;   // pointer to read original characters

        while (read < chars.length) {

            char currentChar = chars[read];  // store current character
            int count = 0;          // count occurrences

            // count frequency of current character
            while (read < chars.length && chars[read] == currentChar) {
                read++;
                count++;
            }

            // write the character once
            chars[write] = currentChar;
            write++;

            // if count > 1, write its digits
            if (count > 1) {

                // convert count to string
                String countStr = String.valueOf(count);

                // write each digit separately
                for (char c : countStr.toCharArray()) {
                    chars[write] = c;
                    write++;
                }
            }
        }

        return write;  // new length
    }
}
```

🧠 Why This Works?

Example:

```
a a a a a a a a a a   (10 times)
```

We write:

```
a 1 0
```

Important:

Count > 9 bhi handle hota hai because:

`String.valueOf(count)` digit by digit likhta hai.

🔥 3️⃣ Check If Two String Arrays are Equivalent

📌 Problem

```
word1 = ["ab", "c"]
word2 = ["a", "bc"]

Output: true
```

Because both form: `"abc"`

✅ Simple Approach (Join and Compare)

```
class Solution {
    public boolean arrayStringsAreEqual(String[] word1, String[] word2) {

        StringBuilder sb1 = new StringBuilder();
        StringBuilder sb2 = new StringBuilder();

        for (String s : word1) {
            sb1.append(s);
        }
        for (String s : word2) {
            sb2.append(s);
        }

        return sb1.toString().equals(sb2.toString());
    }
}
```

Time: O(n)

Space: O(n)

🔥 Interview Optimized (No Extra String)

Two pointer simulation 🔥

```
class Solution {
    public boolean arrayStringsAreEqual(String[] word1, String[] word2) {

        int i = 0, j = 0;  // pointer for array index
        int p1 = 0, p2 = 0; // pointer inside string

        while (i < word1.length && j < word2.length) {

            // compare characters
            if (word1[i].charAt(p1) != word2[j].charAt(p2)) {
                return false;
            }

            p1++;
            p2++;

            // move to next string if end reached
            if (p1 == word1[i].length()) {
                i++;
                p1 = 0;
            }

            if (p2 == word2[j].length()) {
                j++;
                p2 = 0;
            }
        }

        return i == word1.length && j == word2.length;
    }
}
```

Time: O(n)

Space: O(1) 🔥

🧠 Interview Insights

Problem Type	Technique
Compression	Two pointer (read/write)
Compare arrays	Two pointer simulation
Insert/Delete	StringBuilder
Heavy modification	char[]

⚡ Pattern Recognition

If problem says:

- "in-place modify" → use char[]
- "return new string" → StringBuilder
- "memory optimized" → two pointer simulation

Agar tu chahe toh next level karte hain:

- 🔥 Remove duplicates in string
- 🔥 Backspace string compare
- 🔥 Encode & Decode strings (System design type)

Bol kaunsa karna hai next 😎

Ye section interview me logic test karta hai 🔥

Yaha brute force nahi — **pattern observation** ka game hota hai.

Chal ek-ek karke deeply samajhte hain (Java me).

🔥 1️⃣ Count and Say (LeetCode 38)

📌 Pattern samajh

Sequence:

```
1
11      → one 1
21      → two 1
1211    → one 2 one 1
111221  → one 1 one 2 two 1
312211  → three 1 two 2 one 1
```

Rule:

Har term previous term ko “read” karke banti hai.

🧠 Core Idea

- Previous string ko traverse karo
- Consecutive same digits count karo
- Append: count + digit

✅ Java Code (Detailed Comments)

```
class Solution {
    public String countAndSay(int n) {

        // Base case
        if (n == 1) {
            return "1";
        }

        // Start from first term
        String result = "1";

        // Generate terms from 2 to n
        for (int i = 2; i <= n; i++) {

            StringBuilder current = new StringBuilder();
            int count = 1; // count of consecutive digits

            // Traverse previous result
            for (int j = 1; j < result.length(); j++) {

                // If same as previous digit
                if (result.charAt(j) == result.charAt(j - 1)) {
                    count++;
                }
                else {
                    // Append count and previous digit
                    current.append(count);
                    current.append(result.charAt(j - 1));

                    // Reset count
                    count = 1;
                }
            }

            // Append last group
            current.append(count);
            current.append(result.charAt(result.length() - 1));

            // Update result
            result = current.toString();
        }

        return result;
    }
}
```

⏱️ Complexity

Time ≈ O(n × length_of_string)

Space ≈ O(length)

🔥 2️⃣ Run Length Encoding (RLE)

Same logic as Count and Say but for one string.

Example:

```
Input:  aaabbc
Output: a3b2c1
```

✅ Java Code

```
public String runLengthEncode(String s) {

    StringBuilder sb = new StringBuilder();
    int count = 1;

    for (int i = 1; i < s.length(); i++) {

        if (s.charAt(i) == s.charAt(i - 1)) {
            count++;
        } else {
            sb.append(s.charAt(i - 1));
            sb.append(count);
            count = 1;
        }
    }

    // last group
    sb.append(s.charAt(s.length() - 1));
    sb.append(count);

    return sb.toString();
}
```

🔥 3️⃣ Encode and Decode Strings (Very Important – Google/Amazon)

📌 Problem:

Given list of strings, encode into single string.

Then decode back.

Example:

```
["neet","code","love","you"]
```

❌ Wrong Way

Using delimiter like `#`

Problem:

If string itself contains `#`, break ho jayega.

🔥 Correct Way: Length Prefix Encoding

Store as:

```
4#neet4#code4#love3#you
```

Meaning:

- 4 chars → neet
- 4 chars → code
- etc

✅ Encode Code

```
class Codec {

    // Encodes list of strings to single string
    public String encode(List<String> strs) {

        StringBuilder sb = new StringBuilder();

        for (String s : strs) {
            // append length + delimiter + string
            sb.append(s.length());
            sb.append('#');
            sb.append(s);
        }

        return sb.toString();
    }
```

✅ Decode Code

```
    // Decodes string to list of strings
    public List<String> decode(String s) {

        List<String> result = new ArrayList<>();

        int i = 0;

        while (i < s.length()) {

            int j = i;

            // find delimiter '#'
            while (s.charAt(j) != '#') {
                j++;
            }

            // length of next word
            int length = Integer.parseInt(s.substring(i, j));

            // move pointer after '#'
            j++;

            // extract string using length
            String word = s.substring(j, j + length);

            result.add(word);

            // move pointer to next section
            i = j + length;
        }

        return result;
    }
}
```

🧠 Why This Works?

Even if string contains:

```
"ab#c#d"
```

No problem.

Because:

We trust length, not delimiter.

🔥 Interview Insight

Problem	Pattern
Count and Say	Consecutive group counting
RLE	Two pointer group counting
Encode/Decode	Length-based parsing

🧠 Pattern Recognition Trick

If problem says:

- “consecutive characters”
- “compress”
- “generate sequence”

→ Think **grouping logic**

If problem says:

- “combine multiple strings safely”

→ Think **length prefix encoding**

Agar tu chahe next level kare:

- 🔥 Decode String (394) – stack heavy
- 🔥 Remove K Digits
- 🔥 Basic Calculator
- 🔥 Expression parsing

Bol kaunsa todna hai next 😎

Bhai ye wala section interview ka **real test** hota hai 🔥🔥

Yaha brute force nahi — pattern recognition + string manipulation clarity check hoti hai.

Chal ek-ek karke todte hain (Java me, detailed explanation ke saath).

🔥 1️⃣ Longest Common Prefix (LeetCode 14)

📌 Problem

Given array of strings, return longest common prefix.

Example:

```
["flower","flow","flight"]
Output: "fl"
```

🧠 Observation

Common prefix always start se hi hota hai.

Approach:

- First string ko reference lo
- Har character ko baaki strings se compare karo

✅ Approach 1 — Vertical Scanning (Best)

```
class Solution {
    public String longestCommonPrefix(String[] strs) {

        // Edge case
        if (strs == null || strs.length == 0) {
            return "";
        }

        // Take first string as reference
        String first = strs[0];
        // Traverse each character of first string
        for (int i = 0; i < first.length(); i++) {

            char c = first.charAt(i);

            // Compare with remaining strings
            for (int j = 1; j < strs.length; j++) {

                // If:
                // 1. index out of bound
                // 2. character mismatch
                if (i >= strs[j].length() || strs[j].charAt(i) != c) {

                    // Return substring till mismatch
                    return first.substring(0, i);
                }
            }
        }

        // If full first string matched
        return first;
    }
}
```

⏱️ Time Complexity

O(N × M)

N = number of strings

M = length of shortest string

🔥 2️⃣ Rotate String (LeetCode 796)

📌 Problem

Check if `goal` can be obtained by rotating `s`.

Example:

```
s = "abcde"
goal = "cdeab"
```

```
Output: true
```

🧠 Golden Trick

If `goal` is rotation of `s`,

Then `goal` must be substring of `s + s`.

Example:

```
abcde + abcde = abcdeabcde
```

Now:

```
cdeab  → present? yes
```

✅ Java Code

```
class Solution {
    public boolean rotateString(String s, String goal) {

        // Length must match
        if (s.length() != goal.length()) {
            return false;
        }

        // Concatenate string with itself
        String doubled = s + s;

        // Check if goal exists inside doubled string
        return doubled.contains(goal);
    }
}
```

⏱️ Time: O(n)

🔥 3️⃣ Compare Version Numbers (LeetCode 165)

📌 Problem

```
version1 = "1.01"
version2 = "1.001"
Output: 0  (equal)
```

Rules:

- Compare segment by segment
- Ignore leading zeros

🧠 Strategy

1. Split using `"\\."`
2. Compare each segment as integer
3. Missing segment = 0

✅ Java Code (Detailed)

```
class Solution {
    public int compareVersion(String version1, String version2) {

        // Split using dot (escape required)
        String[] v1 = version1.split("\\.");
        String[] v2 = version2.split("\\.");

        int n = Math.max(v1.length, v2.length);

        for (int i = 0; i < n; i++) {

            // If index out of range → treat as 0
            int num1 = i < v1.length ? Integer.parseInt(v1[i]) : 0;
            int num2 = i < v2.length ? Integer.parseInt(v2[i]) : 0;

            if (num1 > num2) return 1;
            if (num1 < num2) return -1;
        }

        return 0;  // equal
    }
}
```

🔥 Longest Repeating Substring (Concept Level)

Interview variant:

Find longest substring that appears at least twice.

Brute:

- Check all substrings → O(n^3)

Optimized:

- Binary search + HashSet
- Or Rabin-Karp rolling hash
- Or Suffix Array (very advanced)

Agar tu FAANG prep mode me hai to ye topic next level hai 😈

🔥 Interview Insight Table

Problem	Core Trick
Longest Common Prefix	Vertical scanning
Rotate String	(s + s).contains(goal)
Compare Version	Split + integer compare
Longest Repeating	Binary search + hashing

🧠 Pattern Recognition

If problem says:

- "prefix" → compare index by index
- "rotation" → double string trick
- "versions / IP / parsing" → split + manual compare
- "repeating substring" → hashing / binary search

Agar tu chahe next level kare:

🔥 Suffix Array basics

🔥 Z Algorithm

🔥 Rolling Hash deep dive

🔥 Trie on strings

Bol bhai, ab DSA level ko FAANG mode me daal dein? 😎

Ab aa gaye real string mastery zone me 😈🔥

Ye topics **interview bonus + competitive programming power tools** hote hain.

Main simple se start karunga → phir optimized idea tak le jaunga.

🔥 1️⃣ Suffix Array (Basics)

📌 Concept

Suffix array = string ke **saare suffixes ko sort karke unka index store karna**

Example:

```
s = "banana"
```

Suffixes:

```
0 → banana
1 → anana
2 → nana
3 → ana
4 → na
5 → a
```

Sorted:

```
a       (5)
ana     (3)
anana   (1)
banana  (0)
na      (4)
nana    (2)
```

Suffix Array:

```
[5, 3, 1, 0, 4, 2]
```

🔥 Use Cases

- Longest Repeating Substring
- Pattern matching
- Lexicographical problems
- Bioinformatics

✅ Basic Construction (Naive $O(n^2 \log n)$)

```
import java.util.*;

class SuffixArrayBasic {

    static class Suffix {
        int index;
        String suffix;

        Suffix(int index, String suffix) {
            this.index = index;
            this.suffix = suffix;
        }
    }

    public static int[] buildSuffixArray(String s) {

        int n = s.length();
        Suffix[] arr = new Suffix[n];

        // Create all suffixes
        for (int i = 0; i < n; i++) {
            arr[i] = new Suffix(i, s.substring(i));
        }

        // Sort lexicographically
        Arrays.sort(arr, (a, b) → a.suffix.compareTo(b.suffix));

        // Store indices
        int[] suffixArray = new int[n];
        for (int i = 0; i < n; i++) {
            suffixArray[i] = arr[i].index;
        }

        return suffixArray;
    }
}
```

⚠ Real interview optimized version = O(n log n) using ranking & doubling technique.

Par basic understanding important hai.

🔥 2️⃣ Z Algorithm (VERY POWERFUL)

📌 What It Does?

For every index i,

Find longest substring starting at i that matches prefix of string.

Z[i] = length of match with prefix.

Example

```
s = "aabxaabxcaabxaabxay"
```

Z-array helps find pattern occurrences in O(n)

🔥 Trick for Pattern Matching

Instead of:

search pattern in text

We do:

```
pattern + "$" + text
```

Then compute Z-array.

Where Z[i] == pattern.length → match found.

✅ Z Algorithm Code (O(n))

```
class ZAlgorithm {

    public static int[] computeZ(String s) {

        int n = s.length();
        int[] Z = new int[n];

        int left = 0, right = 0;

        for (int i = 1; i < n; i++) {

            if (i <= right) {
                Z[i] = Math.min(right - i + 1, Z[i - left]);
            }

            while (i + Z[i] < n &&
                   s.charAt(Z[i]) == s.charAt(i + Z[i])) {
                Z[i]++;
            }

            if (i + Z[i] - 1 > right) {
                left = i;
                right = i + Z[i] - 1;
            }
        }

        return Z;
    }
}
```

🧠 Why Z is Powerful?

- Pattern matching in O(n)
- Find repeating prefix

- String periodicity
- DNA matching

🔥 3️⃣ Rolling Hash (Rabin-Karp Deep Dive)

📌 Idea

Convert string into numeric hash.

```
abc
= a * $p^2$ + b * $p^1$ + c * $p^0$
```

p = prime (31 or 37)

mod = large prime (1e9+7)

🔥 Why?

So we can:

- Compare substrings in O(1)
- Detect duplicates
- Binary search + hash for longest repeating substring

✅ Rolling Hash Template

```
class RollingHash {

    static final int P = 31;
    static final int MOD = 1_000_000_007;

    public static long[] computeHash(String s) {

        int n = s.length();
        long[] hash = new long[n];
        long power = 1;

        hash[0] = s.charAt(0) - 'a' + 1;
        for (int i = 1; i < n; i++) {
            power = (power * P) % MOD;
            hash[i] = (hash[i - 1] +
                        (s.charAt(i) - 'a' + 1) * power) % MOD;
        }

        return hash;
    }
}
```

🔥 Substring Hash in O(1)

```
hash(l, r) =
prefix[r] - prefix[l-1]
```

(Adjust with modular inverse or power normalization)

🔥 Applications

Problem	Technique
Longest Repeating Substring	Binary search + Hash
Rabin-Karp	Rolling hash
Check duplicate substring	HashSet
DNA sequences	Hash sliding window

🧠 Comparison Table

Algorithm	Time	Use
Suffix Array	O(n log n)	Full lexicographic power
Z Algorithm	O(n)	Pattern match
Rolling Hash	O(n)	Fast substring compare

🚀 Interview Strategy

If interviewer says:

- “pattern match efficiently” → Z or KMP
- “longest repeating substring” → Binary search + rolling hash
- “lexicographical heavy problem” → Suffix array

Agar tu next level jaana chahe to:

🔥 Suffix Array optimized build

🔥 LCP Array (Kasai Algorithm)

🔥 Double Hashing

🔥 Manacher’s Algorithm (Longest palindrome in O(n))

Bol bhai, kaunsa FAANG weapon unlock karein next? 😈

Ab aa gaye real competitive programming zone me 😈🔥

Optimized Suffix Array (O(n log n)) = Interview + CP dono me powerful weapon.

Main tujhe:

- Concept clear karunga
- Rank + Doubling technique samjhaunga
- Full Java code dunga
- Har important line explain karunga

🔥 Optimized Suffix Array (Doubling Technique)

📌 Core Idea

Hum suffix ko directly compare nahi karte (wo $O(n^2)$ ho jayega).

Instead:

We sort based on:

```
First 1 character
Then 2 characters
Then 4
Then 8
Then 16
...
```

Har step pe:

- Ranking assign karte hain
- Previous rank use karke next sort karte hain

Isliye isko **doubling algorithm** kehte hain.

🧠 Example

```
s = "banana"
```

Step 1 → Sort by 1 character

Step 2 → Sort by first 2 characters

Step 3 → Sort by first 4 characters

... until 2^k ≥ n

🔥 Data Structure

Har suffix ke liye store karte hain:

```
index
rank[0] → current rank
rank[1] → next rank (k distance pe)
```

✅ Java Implementation (O(n log n))

```
import java.util.*;

class SuffixArrayOptimized {

    static class Suffix {
        int index;      // original index
        int rank;       // rank of first half
        int nextRank;   // rank of second half

        Suffix(int index, int rank, int nextRank) {
            this.index = index;
            this.rank = rank;
            this.nextRank = nextRank;
        }
    }

    public static int[] buildSuffixArray(String s) {

        int n = s.length();
        Suffix[] suffixes = new Suffix[n];

        // Initial ranking based on first character
        for (int i = 0; i < n; i++) {
            suffixes[i] = new Suffix(
                i,
                s.charAt(i),           // current char rank
                (i + 1 < n) ? s.charAt(i+1) : -1  // next char rank
            );
        }

        // Sort based on first 2 characters
        Arrays.sort(suffixes, (a, b) → {
            if (a.rank != b.rank)
                return Integer.compare(a.rank, b.rank);
            return Integer.compare(a.nextRank, b.nextRank);
        });

        int[] indexMap = new int[n]; // to map index → position in suffix array

        // k = length of substring to compare
        for (int k = 4; k < 2 * n; k *= 2) {

            int rank = 0;
            int prevRank = suffixes[0].rank;
            suffixes[0].rank = 0;
            indexMap[suffixes[0].index] = 0;

            // Assign new ranks
            for (int i = 1; i < n; i++) {

                if (suffixes[i].rank == prevRank &&
                    suffixes[i].nextRank == suffixes[i - 1].nextRank) {

                    prevRank = suffixes[i].rank;
                    suffixes[i].rank = rank;
                } else {
                    prevRank = suffixes[i].rank;
                    suffixes[i].rank = ++rank;
                }

                indexMap[suffixes[i].index] = i;
            }

            // Update nextRank for next iteration
            for (int i = 0; i < n; i++) {
                int nextIndex = suffixes[i].index + k / 2;
                suffixes[i].nextRank =
                    (nextIndex < n) ?
                    suffixes[indexMap[nextIndex]].rank : -1;
            }

            // Sort again based on new ranks
            Arrays.sort(suffixes, (a, b) -> {
                if (a.rank != b.rank)
                    return Integer.compare(a.rank, b.rank);
                return Integer.compare(a.nextRank, b.nextRank);
            });
        }

        // Extract suffix array
        int[] suffixArray = new int[n];
        for (int i = 0; i < n; i++) {
            suffixArray[i] = suffixes[i].index;
        }

        return suffixArray;
    }

    // Test
    public static void main(String[] args) {
        String s = "banana";
        int[] sa = buildSuffixArray(s);
        System.out.println(Arrays.toString(sa));
    }
}
```

🧠 Why This Works

Each iteration:

- Compare 2^k length substrings
- Use previous ranking
- Sorting takes O(n log n)
- Total iterations ≈ log n

So total:

```
O(n log n)
```

🔥 After Suffix Array → LCP Array (Kasai)

Once you have suffix array:

You can compute:

Longest Common Prefix between adjacent suffixes in O(n)

Used in:

- Longest repeating substring
- Number of distinct substrings

Agar tu bole to next Kasai algorithm todte hain 😈

🚀 Real Interview Insight

Level	What Interviewer Expects
Mid	Concept of suffix array
Senior	Doubling technique
CP	Suffix + LCP + RMQ

Level	What Interviewer Expects
FAANG Hard	Suffix tree / suffix automaton

Ab bata:

🔥 LCP (Kasai Algorithm)

🔥 Manacher's Algorithm

🔥 Suffix Automaton

🔥 Double Rolling Hash

Ab tu advanced zone me ghus chuka hai 😎

Bhai 🔥 ye section literally **strings ke edge-case killer** hai.

FAANG level ke interviews me ye hi wo points hain jo log bhool jaate hain aur fail ho jaate hain.

Chal ek-ek karke detail me dekhte hain (Java context me).

1️⃣ Empty String `""`

⚠ Issues:

- Length = 0 → loops crash ho sakte hain
- `charAt(0)` ya `substring(0,1)` → **Exception**
- Sliding window / two pointer → loop miss

✅ Safe handling:

```
if (s == null || s.length() == 0) return "";
```

Example:

```
String s = "";
System.out.println(s.length()); // 0 ✅
```

2️⃣ Single Character `"a"`

⚠ Issues:

- Sliding window / two pointer start = end
- Palindrome / compression → count = 1

- Loop `for(int i=0; i<n-1; i++)` → may skip single character

✅ Always handle explicitly:

```
if (s.length() == 1) return s;
```

Example:

```
String s = "a";
System.out.println(s.charAt(0)); // works ✅
```

3️⃣ All Same Characters `"aaaa"`

⚠ Issues:

- Compression / run-length → last group overcount
- Sliding window for unique chars → max length = 1
- Sorting / prefix → duplicates

✅ Test explicitly:

```
String s = "aaaa";
int count = 1;
for (int i = 1; i < s.length(); i++) {
    if (s.charAt(i) == s.charAt(i-1)) count++;
}
// last group counted ✅
```

4️⃣ Case Sensitivity `"aA"`

⚠ Issues:

- `'A' != 'a'`
- ASCII values differ: `'A' = 65`, `'a' = 97`
- Frequency array `int[26]` → lowercase only works

✅ Handle properly:

```
char c = 'A';
```

```
int index = Character.toLowerCase(c) - 'a'; // normalize
```

5️⃣ Unicode / Extended ASCII

⚠ Issues:

- Java `char` = UTF-16
- `int[26]` cannot store emojis / extended symbols
- Direct comparison may fail

✅ Use `HashMap` instead:

```
String s = "a😊b";
Map<Character, Integer> freq = new HashMap<>();
for (char c : s.toCharArray()) {
    freq.put(c, freq.getOrDefault(c, 0) + 1);
}
System.out.println(freq);
```

🧠 Summary Table

Edge Case	Common Pitfall	Fix / Tip
Empty String	Loop crash / charAt exception	Always check length == 0
Single Character	Sliding window / palindrome loops	Explicitly handle length 1
All Same Characters	Compression / frequency miscount	Count last group carefully
Case Sensitivity	'A' != 'a', array indexing	Use `toLowerCase()` or normalize
Unicode / ASCII	Extended chars crash freq array	Use `HashMap<Character, Integer>`

💡 Final Tip (FAANG Mastery)

Strings = **Sliding Window + Two Pointer + Hashing**

Edge cases = **always dry-run**

Yahi 5 points miss na karo → 80% tricky questions solve ho jaayenge.

Agar tu chaahe, main **“Ultimate String Edge Cases + Template Table”** bana ke de doon:

- Har pattern + edge case + Java template

- Ek hi page me FAANG ready 😎

Banau bhai?

Perfect bhai 😎🔥

Chal banate hain **Ultimate String Edge Cases + Templates Cheat Sheet (Java)** — ek hi page me FAANG ready.

💎 Ultimate String Edge Cases & Templates (Java)

Edge Case	Pitfall / Common Mistake	Template / Fix	Notes / Tips
Empty String `""`	Loop crash, `charAt(0)`, `substring(0,1)` → Exception	```java if(s == null	
Single Character `"a"`	Sliding window / two pointer loops skip, palindrome count	`java if(s.length() == 1) return s;`	Consider length 1 as valid palindrome / substring
All Same Characters `"aaaa"`	Compression / run-length → last group overcount	`java int count = 1; for(int i=1;i<n;i++){ if(s.charAt(i)==s.charAt(i-1)) count++; }`	Count the last group carefully
Case Sensitivity `"aA"`	`'A' != 'a'` → frequency array / comparison wrong	`java int idx = Character.toLowerCase(c)-'a';`	Normalize before counting / comparing
Unicode / Extended ASCII	Char outside `[a-z]` → int[26] fails	`java Map<Character,Integer> freq = new HashMap<>(); for(char c:s.toCharArray()){ freq.put(c,freq.getOrDefault(c,0)+1); }`	Always use HashMap for generic chars
Prefix / Suffix Edge	Substring index out of bounds	`java s.substring(start, Math.min(end,s.length()));`	Avoid crash when end > length
Sliding Window Empty / Single Char	Window length = 0 or 1 → max/min	`java int left=0,right=0; while(right<n){ // handle }`	Always check window length before update

Edge Case	Pitfall / Common Mistake	Template / Fix	Notes / Tips
	calculation wrong		
Two Pointer Equal / Overlap	left == right, skip comparison	`java while(left<right){ if(s.charAt(left)!=s.charAt(right)) break; left++; right--; }`	Works for palindrome & reverse logic
Hashing / Map Updates	Forget remove for sliding window → wrong freq	`java map.put(c,map.get(c)-1); if(map.get(c)==0) map.remove(c);`	Maintain accurate counts
Concatenation / Encoding	Delimiter exists in string → split fails	`java sb.append(s.length()).append('#').append(s);`	Always length-prefix for safe decoding

🔷 FAANG Level Patterns Cheat (Strings)

Pattern	Typical Problem Types	Edge Case Watch
Sliding Window	Longest substring, min window, count occurrence	Empty string, single char, all same char
Two Pointer	Reverse, palindrome, remove/replace	Single char, left/right overlap
Hashing / Frequency Map	Anagrams, duplicates, pattern match	Case sensitivity, Unicode chars
Prefix / Suffix Array / Z / Rolling Hash	LCP, repeating substring, rotations	Empty string, small string
StringBuilder / Mutable	Compression, insert/delete, encode/decode	Null string, single char

🔷 Quick Java Templates

1️⃣ Sliding Window Max Length

```
int left=0, right=0, maxLen=0;
while(right<s.length()){
    // expand window
    right++;
    // shrink window if needed
    while(condition){
        left++;
    }
    maxLen = Math.max(maxLen,right-left);
}
```

2️⃣ Two Pointer Palindrome Check

```
int left=0, right=s.length()-1;
while(left<right){
    if(s.charAt(left)!=s.charAt(right)) return false;
    left++;
    right--;
}
return true;
```

3️⃣ Frequency Map

```
Map<Character,Integer> freq = new HashMap<>();
for(char c:s.toCharArray()){
    freq.put(c,freq.getOrDefault(c,0)+1);
}
```

4️⃣ Safe Substring

```
String sub = s.substring(start, Math.min(end, s.length()));
```

5️⃣ String Compression / RLE Template

```
StringBuilder sb = new StringBuilder();
int count = 1;
for(int i=1;i<s.length();i++){
    if(s.charAt(i)==s.charAt(i-1)) count++;
    else {
        sb.append(s.charAt(i-1));
        if(count>1) sb.append(count);
        count=1;
    }
}
sb.append(s.charAt(s.length()-1));
if(count>1) sb.append(count);
return sb.toString();
```

🧠 Final Advice (FAANG Level)

1. Strings = **Sliding Window + Two Pointer + Hashing**
2. Always dry-run **edge cases**: empty, single char, all same, case, Unicode
3. Prefix / suffix / encoding → **length-based safe operations**
4. Mutable (`StringBuilder`) vs immutable (`String`) → know when to use
5. Observe patterns → **almost all problems are combination of these pillars**

Bhai agar tu chaahe, main **is cheat sheet ko ek visual mind-map + coding template PDF style** me bana ke de sakta hu, jisse interview me **1 second me recall ho jaye** 😎

Banau kya?

Bhai, ab hum aa gaye **Naive Pattern Matching** ke section me 🔥

Ye basic hai, lekin **interview me bonus points** deta hai aur Rabin-Karp/KMP ke liye foundation bhi banata hai.

Chal detail me todte hain (Java context me).

🔥 Naive Pattern Matching

📌 Problem

Given:

- Text: `"ABABDABACDABABCABAB"`
- Pattern: `"ABABCABAB"`

Find all positions where pattern occurs in text.

🧠 Idea (Brute Force)

- Start at each index `i` in text (0 to n-m)

- Compare substring of length `m` with pattern
- If match → record index

Time Complexity: O((n-m+1) * m) → O(n*m)

Space Complexity: O(1) (except output list)

✅ Java Code (Detailed Comments)

```
import java.util.*;

class NaivePatternMatching {

    public static List<Integer> findPattern(String text, String pattern) {
        List<Integer> result = new ArrayList<>();

        int n = text.length();
        int m = pattern.length();

        // Edge case: empty pattern
        if (m == 0 || n < m) return result;

        // Traverse text from 0 to n-m
        for (int i = 0; i <= n - m; i++) {

            int j;

            // Compare pattern with current substring
            for (j = 0; j < m; j++) {
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    break; // mismatch → break inner loop
                }
            }

            // If full pattern matched
            if (j == m) {
                result.add(i); // store starting index
            }
        }

        return result;
    }

    // Test
    public static void main(String[] args) {
        String text = "ABABDABACDABABCABAB";
        String pattern = "ABABCABAB";
        List<Integer> positions = findPattern(text, pattern);
        System.out.println("Pattern found at indices: " + positions);
    }
}
```

🔷 Step-by-Step Example

```
Text:    A B A B D A B A C D A B A B C A B A B
Pattern: A B A B C A B A B
```

1. Start i = 0 → compare first 9 chars → mismatch at 4 → move i++
2. i = 1 → mismatch at 1 → i++
3. i = 10 → full match → record index 10

⚠ Edge Cases

1. **Empty Text or Pattern** → return empty list
2. **Pattern longer than text** → return empty list
3. **Overlapping patterns** → works fine (check i++)
4. **Single character pattern** → works automatically

🧠 Notes

- Simple, easy to implement
- O(n*m) → slow for large text/pattern → use **Rabin-Karp / KMP** for optimization
- Foundation for **rolling hash / Z-array / prefix function**

Bhai, agar tu chaahe, main **Naive → Rabin-Karp → KMP → Z Algorithm** pattern matching **all in one FAANG-ready pipeline** bana ke template de doon, jisse interview me ek hi code frame yaad rahe 😎

Chahte ho bana doon?

Bhai, ab hum aa gaye **Rabin-Karp Algorithm** pe, jo **Naive pattern matching ka optimized version** hai using **hashing**.

Ye FAANG & CP me direct pattern matching + substring detection me **hero algorithm** hai 🔥.

Chal step by step todte hain:

🔥 Rabin-Karp Algorithm

📌 Problem

Given:

- Text: `"ABABDABACDABABCABAB"`
- Pattern: `"ABABCABAB"`

Find all positions where pattern occurs.

🧠 Idea

- Instead of comparing **all characters every time** (O(n*m))
- Compute **hash** of pattern
- Compute **rolling hash** of substrings of text of length m
- If hash matches → compare actual substring (avoid collisions)

Time Complexity:

- Best case: O(n + m)
- Worst case (hash collisions): O(n*m)

🔷 Steps

1. Choose **prime base** `p` (commonly 31 or 101)
2. Choose **large prime modulus** `mod` (like 1e9+7)
3. Compute hash of pattern → `hashP`
4. Compute initial hash of first substring of text → `hashT`
5. Slide window of length `m` over text:

- Update hash in O(1) using **rolling hash formula**
- If `hashT == hashP` → compare actual substring

6. Record matching indices

🔷 Rolling Hash Formula

For substring `s[i..i+m-1]`:

```
hash(s[i+1..i+m]) = (hash(s[i..i+m-1]) - s[i]*p^(m-1)) * p + s[i+m]
```

All operations mod `mod`.

✅ Java Implementation (Detailed Comments)

```
import java.util.*;

class RabinKarp {

    static final int p = 31;              // prime base
    static final int mod = 1_000_000_007;  // large prime

    // Function to compute hash of string
    static long computeHash(String s) {
        long hash = 0;
        long power = 1;

        for (int i = 0; i < s.length(); i++) {
            hash = (hash + (s.charAt(i) - 'a' + 1) * power) % mod;
            power = (power * p) % mod;
        }

        return hash;
    }

    public static List<Integer> rabinKarp(String text, String pattern) {
        List<Integer> result = new ArrayList<>();
        int n = text.length();
        int m = pattern.length();

        if (m == 0 || n < m) return result;

        long hashPattern = computeHash(pattern);

        // Precompute p^(m-1) for rolling hash
        long pPowMMinus1 = 1;
        for (int i = 1; i < m; i++) {
            pPowMMinus1 = (pPowMMinus1 * p) % mod;
        }

        // Compute hash of first substring of length m
        long hashText = computeHash(text.substring(0, m));

        // Slide the window
        for (int i = 0; i <= n - m; i++) {
            // If hashes match, compare actual substring to avoid collision
            if (hashText == hashPattern) {
                if (text.substring(i, i + m).equals(pattern)) {
                    result.add(i);
                }
            }

            // Update hash: remove first char and add next char
            if (i < n - m) {
                hashText = (hashText - (text.charAt(i) - 'a' + 1) + mod) % mod; // remo
ve s[i]
                hashText = (hashText * p) % mod;                                // shift
                hashText = (hashText + (text.charAt(i + m) - 'a' + 1)) % mod;     // add
next char
            }
        }

        return result;
    }

    // Test
    public static void main(String[] args) {
        String text = "ABABDABACDABABCABAB".toLowerCase();
        String pattern = "ABABCABAB".toLowerCase();

        List<Integer> positions = rabinKarp(text, pattern);
        System.out.println("Pattern found at indices: " + positions);
    }
}
```

🧠 Step-by-Step Example

```
Text:    a b a b d a b a c d a b a b c a b a b
Pattern: a b a b c a b a b
```

1. Compute `hashP` = hash("ababcabab")
2. Compute `hashT` = hash("ababdabab") → mismatch → slide
3. Slide window → update hash using formula → compare hash
4. When hash matches → compare substring → match found
5. Record index → continue

⚠ Edge Cases

- Empty pattern / empty text → return empty list
- Pattern longer than text → return empty list
- Case sensitivity → normalize using `.toLowerCase()`
- Collisions → substring comparison required

🔷 Advantages

- Efficient for **multiple pattern searches**
- Best case: O(n + m)
- Rolling hash = O(1) per step
- Foundation for **Rabin-Karp + Longest Repeating Substring + Duplicate detection**

🔥 Quick Notes for Interview

Step	Key Trick
Hash Pattern	Precompute once
Rolling Hash	O(1) update formula
Collision	Always verify substring
Precompute Power	Avoid recomputation each step
Modulus	Avoid overflow

Bhai, agar tu chaahe, main **Naive → Rabin-Karp → KMP → Z Algorithm → Rolling Hash pipeline** ek **FAANG ready string pattern matching template** bana ke de doon, jisme **code + edge case handling + template** ek saath ho.

Chahte ho bana doon?

Bhai ab hum aa gaye **KMP Algorithm (Knuth-Morris-Pratt)** pe 🔥

Ye **pattern matching ka FAANG-level hero** hai — O(n + m) time without rolling hash.

Chal step-by-step samjhate hain:

🔥 KMP Algorithm

📌 Problem

Given:

- Text: `"ABABDABACDABABCABAB"`
- Pattern: `"ABABCABAB"`

Find all positions where pattern occurs **efficiently**.

🧠 Idea

- Naive method → O(n*m) (compare each substring)
- KMP → avoid re-comparing already matched characters
- Use **Longest Prefix Suffix (LPS) array**:

```
lps[i] = length of longest proper prefix which is also suffix for pattern[0..i]
```

Example: Pattern "ABABCABAB"

```
Index: 0 1 2 3 4 5 6 7 8
Char:  A B A B C A B A B
LPS:   0 0 1 2 0 1 2 3 4
```

- LPS tells **how much we can skip** if mismatch happens.

🔷 Steps

1. Build **LPS array** for pattern → O(m)
2. Traverse text with two pointers: `i` (text), `j` (pattern)
   - If `text[i] == pattern[j]` → i++, j++
   - If j == m → pattern found, record index, reset j = lps[j-1]
   - If mismatch:
     - If j != 0 → j = lps[j-1]
     - Else i++

Time Complexity: O(n + m)

Space Complexity: O(m)

✅ Java Implementation (Detailed Comments)

```
import java.util.*;

class KMP {

    // Build LPS array
    private static int[] buildLPS(String pattern) {
        int m = pattern.length();
        int[] lps = new int[m];
        int len = 0; // length of previous longest prefix suffix
        int i = 1;   // start from 1, lps[0] = 0

        while (i < m) {
            if (pattern.charAt(i) == pattern.charAt(len)) {
                len++;
                lps[i] = len;
                i++;
            } else {
                if (len != 0) {
                    // try previous prefix
                    len = lps[len - 1];
                } else {
                    lps[i] = 0;
                    i++;
                }
            }
        }

        return lps;
    }

    // KMP search
    public static List<Integer> kmpSearch(String text, String pattern) {
        List<Integer> result = new ArrayList<>();
        int n = text.length();
        int m = pattern.length();

        if (m == 0 || n < m) return result;

        int[] lps = buildLPS(pattern);

        int i = 0; // text index
        int j = 0; // pattern index

        while (i < n) {
            if (text.charAt(i) == pattern.charAt(j)) {
                i++;
                j++;
            }

            if (j == m) {
                // Pattern found at index i - j
                result.add(i - j);
                j = lps[j - 1]; // continue searching
            } else if (i < n && text.charAt(i) != pattern.charAt(j)) {
                if (j != 0) {
                    j = lps[j - 1]; // jump using lps
                } else {
                    i++;
                }
            }
        }

        return result;
    }

    // Test
    public static void main(String[] args) {
        String text = "ABABDABACDABABCABAB";
        String pattern = "ABABCABAB";
        List<Integer> positions = kmpSearch(text, pattern);
        System.out.println("Pattern found at indices: " + positions);
    }
}
```

🔷 Step-by-Step Example

Pattern: `"ABABCABAB"` → LPS = `[0,0,1,2,0,1,2,3,4]`

Text: `"ABABDABACDABABCABAB"`

- Start comparing from i=0, j=0
- Mismatch at i=4, j=4 → use LPS → j = lps[3] = 2
- Continue → matches → i=10, j=9 → match → record i-j = 10
- Continue → next match at i=15

⚠ Edge Cases

1. Empty text or pattern → return empty list
2. Pattern length > text length → return empty list
3. Single character pattern → works automatically

4. Case sensitivity → normalize using `.toLowerCase()`

🧠 Why KMP is FAANG favorite

- No backtracking → O(n + m) guaranteed
- Handles multiple pattern occurrences efficiently
- Foundation for **suffix array / Z-array / advanced pattern matching**

🔷 Comparison with Rabin-Karp

Feature	KMP	Rabin-Karp
Time	O(n + m) worst case	O(n + m) avg, O(n*m) worst
Method	Prefix table (LPS)	Rolling hash
Collisions	No	Yes, must check substring
Use Case	Single pattern / repeated	Multiple patterns / substring detection

Bhai, agar tu chaahe, main **"Naive → Rabin-Karp → KMP → Z Algorithm → Rolling Hash" FAANG string matching pipeline** bana ke de doon, jisme **templates + edge cases + dry-run tips** sab ek saath ho, jisse interview ready ho jaaye.

Chahte ho bana doon?