Q: Strings in Java

1. Problem Understanding

- In Java, String is a class, not a primitive type.
- Strings are immutable â€” once created, their content cannot change.
- Any modification (like concatenation) creates a new String object.
- String literals are stored in the String Pool for memory efficiency.
    - Example:
        - String s1 = "hello";
        - String s2 = s1;
        - s1 = s1 + " world"; // creates new String
- **String Creation**
    - String s1 = "hello"; â†’ stored in String Pool.
    - String s2 = new String("hello"); â†’ stored in Heap memory.
    - Prefer literals for DSA â€” they are faster and reused automatically.
- **Key Properties**
    - Strings are immutable.
    - String literals are reused from the String Pool.
    - Implement Comparable for lexicographic comparison.
    - 0-indexed like arrays.
    - Supports index-based operations via charAt().
- **Common and Useful String Methods**
    - Length and Character Access
        - length() â†’ returns number of characters.
        - charAt(i) â†’ returns character at index i.
    - Substring and Searching
        - substring(i, j) â†’ substring from index i to j-1.
        - indexOf(ch) â†’ first occurrence of a character.
        - lastIndexOf(ch) â†’ last occurrence of a character.
        - contains(str) â†’ checks if a substring exists.
    - Comparison
        - equals(str) â†’ compares values.
        - equalsIgnoreCase(str) â†’ ignores case.
        - compareTo(str) â†’ lexicographically compares (returns +, -, or 0).
    - Prefix and Suffix
        - startsWith(prefix) â†’ checks prefix.
        - endsWith(suffix) â†’ checks suffix.

- Case and Spaces
  - toLowerCase() / toUpperCase() â†’ converts case.
  - trim() â†’ removes leading/trailing spaces.
- Replacement and Split
  - replace(old, new) â†’ replaces characters or substrings.
  - split(delimiter) â†’ splits into array based on delimiter.
- Conversion
  - toCharArray() â†’ converts to char array.
  - String.valueOf(data) â†’ converts other data types to string.

- **StringBuilder (Mutable Alternative)**
  - Strings are immutable; concatenation repeatedly is costly (O(nÂ²)).
  - Use StringBuilder for modification-heavy tasks (like reversing or appending).
  - Example:
    - StringBuilder sb = new StringBuilder("abc");
    - sb.append("xyz"); // "abcxyz"
    - sb.insert(3, "123"); // "abc123xyz"
    - sb.delete(2, 4); // deletes characters 2â€“3
    - sb.reverse(); // reverses string
    - String result = sb.toString();
  - StringBuilder is faster but not thread-safe.
  - StringBuffer is thread-safe (rarely needed in DSA).

- **DSA Operations with Strings**
  - Basic String Operations
    - Get Character at Index:
      - s.charAt(i)
    - Get Length of String:
      - s.length()
    - Check Equality:
      - Case-sensitive â†’ s.equals(t)
      - Case-insensitive â†’ s.equalsIgnoreCase(t)
    - Find Substring:
      - s.contains(sub) or s.indexOf(sub) â†’ returns -1 if not found.
    - Get Substring:
      - s.substring(start, end) â†’ from start (inclusive) to end (exclusive).
    - Replace Characters or Substrings:
      - Replace single character â†’ s.replace('a', 'b')
      - Replace using regex â†’ s.replaceAll("regex", "replacement")
    - Split String by Delimiter:
      - s.split(" ") or s.split(",")
    - Trim Whitespace:
      - s.trim()
    - Convert Between String and Integer:
      - String â†’ Integer â†’ Integer.parseInt(str)
      - Integer â†’ String â†’ String.valueOf(num)

- Common DSA Operations with Strings
  - Reverse a String:
    - new StringBuilder(s).reverse().toString();
  - Check Palindrome:
    - Compare s with its reversed version.
  - Count Character Frequency:
    - Use an int[26] array for lowercase letters.
  - Convert Char to Int:
    - ch - '0'
  - Convert Int to Char:
    - (char)(num + '0')
  - Convert String to Char Array:
    - s.toCharArray()
  - Join Array of Strings:
    - String.join("", array)
  - Sort Characters in String:
    - char[] c = s.toCharArray();
    - Arrays.sort(c);
    - String sorted = new String(c);
  - Compare Substrings:
    - s.substring(i, j).equals(t.substring(x, y))
- Character Case Operations
  - Check uppercase â†’ Character.isUpperCase(ch)
  - Check lowercase â†’ Character.isLowerCase(ch)
  - Convert to upper â†’ Character.toUpperCase(ch)
  - Convert to lower â†’ Character.toLowerCase(ch)
  - String â†’ uppercase â†’ str.toUpperCase()
  - String â†’ lowercase â†’ str.toLowerCase()

- **String vs char[] (Quick Comparison)**
  - String â†’ immutable, more built-in methods, easy to use.
  - char[] â†’ mutable, good for in-place edits like reversals.
  - Use char[] in problems requiring frequent modifications.

- **Time Complexity Notes**
  - charAt() â†’ O(1)
  - equals() â†’ O(n)
  - Concatenation (+) â†’ O(n)
  - Append in StringBuilder â†’ O(1) amortized
  - substring() â†’ O(n)

- **Key Takeaways**
  - Use StringBuilder for concatenation-heavy logic.
  - Use equals() instead of == for value comparison.
  - Remember Strings are immutable.

- Master substring(), charAt(), and toCharArray().
- Understand compareTo() for lexicographic sorting and comparison.

QBitmasking(Used in q: 55)

---

1. Problem Understanding

- Bitmasking is a technique that uses bits inside an integer to represent multiple true/false states compactly.
- Each bit in an integer acts like a flag â€" 1 means "set/present", 0 means "unset/absent".
- It allows efficient manipulation and checking of states using bitwise operations.
- **Why Use Bitmasking**
  - Saves memory â€" stores 32 (or 64) flags in a single integer.
  - Bitwise operations are extremely fast â€" each runs in O(1).
  - Ideal for problems involving:
    - Presence or absence of items (like letters in a string).
    - Subset generation and combinatorial search.
    - State tracking in dynamic programming (DP).
    - Representing visited/unvisited states in graphs.
    - Encoding multiple binary states efficiently.
- **Bitwise Operators**
  - & (AND) â†' Result bit is 1 only if both bits are 1.
  - | (OR) â†' Result bit is 1 if any of the bits is 1.
  - ^ (XOR) â†' Result bit is 1 if bits are different.
  - ~ (NOT) â†' Inverts every bit (1 becomes 0, 0 becomes 1).
  - << (Left Shift) â†' Shifts bits left â†' multiplies number by 2 for each shift.
  - > (Right Shift) â†' Shifts bits right â†' divides number by 2 for each shift.
- **Common Bitmask Operations**
  - Set a bit (turn ON) â†' mask |= (1 << i) â†' Sets the i-th bit to 1.
  - Unset a bit (turn OFF) â†' mask &= ~(1 << i) â†' Sets the i-th bit to 0.
  - Toggle a bit â†' mask ^= (1 << i) â†' Flips the i-th bit (1 â†' 0 or 0 â†' 1).
  - Check if bit is set â†' (mask & (1 << i)) != 0 â†' Returns true if the i-th bit is 1.
  - Count bits that are ON â†' Integer.bitCount(mask) â†' Gives how many bits are 1 in total.
- **Bitmask Expressions Explained**
  - mask |= (1 << (ch - 'a'))
    - Purpose: Marks a letter as seen in the bitmask.
    - Step-by-step:

        - ch - 'a' â†’ Converts the letter 'a'â€“'z' to index 0â€“25.
        - Example: 'a' - 'a' = 0, 'b' - 'a' = 1, â€¦ 'z' - 'a' = 25.
        - 1 << (ch - 'a') â†’ Creates a number where only that bit is 1.
      - Example: 'c' â†’ 1 << 2 â†’ 000...0100 (3rd bit ON).
      - mask |= â€¦ â†’ Performs bitwise OR with the current mask.
      - Turns ON that bit without affecting other bits.
      - Effect: Marks the letter as â€œpresentâ€ in the mask.
    - mask == (1 << 26) - 1
      - Purpose: Checks if all 26 letters are present.
      - Step-by-step:
        - 1 << 26 â†’ Creates a number with 1 followed by 26 zeros in binary: 100â€¦0 (27 bits).
        - (1 << 26) - 1 â†’ Subtract 1 â†’ all 26 bits become 1: 111â€¦1 (binary with 26 ones).
        - mask == (1 << 26) - 1 â†’ Compares current mask with â€œall letters presentâ€ mask.
          - If equal â†’ all letters 'a'â€“'z' seen.
          - If not equal â†’ some letters missing.

- **Example â€” Pangram Problem**
  - Each alphabet letter corresponds to one bit (26 total).
  - When a letter appears, turn ON its bit: mask |= (1 << (ch - 'a')).
  - After processing all characters, check if all 26 bits are ON: mask == (1 << 26) - 1.
  - If true → pangram; else → not pangram.

- **Where Bitmasking is Used**
  - Pangram check (presence of 26 letters).
  - Subset generation (for (int mask = 0; mask < (1 << n); mask++)).
  - Traveling Salesman Problem (TSP) and DP on subsets.
  - Counting pairs or combinations efficiently.
  - Representing visited states in graph search.
  - Problems involving toggling or tracking multiple binary choices.

Q: Regex

---

1. Problem Understanding

- Regex (Regular Expression) is a pattern that describes a set of strings.
- Used for searching, matching, replacing, or splitting strings efficiently.
- Extremely useful in string validation, filtering, and transformation problems.
- **Common Patterns and Usage**
  - Match only letters

        - Expression: [a-zA-Z]
        - Purpose: Matches any single uppercase or lowercase letter.
        - DSA Example: Check if a string contains only letters before applying transformations.
            - String s = "HelloWorld";
            - boolean valid = s.matches("[a-zA-Z]+"); // true
        - Explanation:
            - matches returns true if the entire string consists of letters a-z or A-Z. The + ensures at least one letter.
    - Match only digits
        - Expression: [0-9] or \d
        - Purpose: Matches any single digit.
        - DSA Example: Count digits in a string or validate numeric input.
            - String s = "123abc";
            - int countDigits = s.replaceAll("\D", "").length(); // removes non-digits, count = 3
        - Explanation:
            - \D matches non-digits. replaceAll removes them. length() counts remaining digits.
    - Match word characters
        - Expression: \w â†’ matches [a-zA-Z0-9_]
        - Purpose: Matches letters, digits, or underscore.
        - DSA Example: Validate variable names (alphanumeric + underscore).
            - String s = "var_1";
            - boolean valid = s.matches("\w+"); // true
        - Explanation:
            - \w+ ensures all characters are letters, digits, or underscores.
    - Match non-word characters
        - Expression: \W â†’ matches anything except [a-zA-Z0-9_]
        - DSA Example: Remove special characters from a string.
            - String s = "hello@world!";
            - String clean = s.replaceAll("\W", ""); // "helloworld"
        - Explanation:
            - \W matches special characters. replaceAll removes them, leaving only letters/digits/underscore.
    - Match whitespace
        - Expression: \s â†’ space, tab, newline
        - DSA Example: Split a sentence into words.
            - String sentence = "We are learning";
            - String[] words = sentence.split("\s"); // ["We","are","learning"]
        - Explanation:
            - split("\s") breaks the string wherever whitespace occurs.
    - Match non-whitespace
        - Expression: \S â†’ any character except whitespace
        - DSA Example: Count all non-space characters in a string.
            - String s = "Hi there";
            - int count = s.replaceAll("\S", "").length(); // count spaces
        - Explanation:

        - \S matches non-space characters. replaceAll removes them. Remaining length = spaces count.

- **Quantifiers**
    - " * " â†’ 0 or more occurrences
    - " + " â†’ 1 or more occurrences
    - ? â†’ 0 or 1 occurrence
    - {n} â†’ exactly n occurrences
    - {n,m} â†’ between n and m occurrences
    - DSA Example: Validate repeated patterns in strings.
        - String s = "aaab";
        - boolean match = s.matches("a{3}b"); // true
    - Explanation:
        - a{3}b matches exactly three 'a's followed by 'b'. Returns true if string matches.

- **Character Classes**
    - [abc] â†’ match a, b, or c
    - [^abc] â†’ match any character except a, b, or c
    - DSA Example: Remove vowels from a string.
        - String s = "leetcode";
        - String noVowels = s.replaceAll("[aeiou]", ""); // "ltcd"
    - Explanation:
        - [aeiou] matches vowels. replaceAll removes them.

- **Anchors**
    - ^ â†’ start of string
    - $ â†’ end of string
    - DSA Example: Check if string starts with a letter.
        - String s = "Hello123";
        - boolean valid = s.matches("^[a-zA-Z].*"); // true
    - Explanation:
        - ^ ensures the first character is a letter, .* matches the rest of the string.

- **Groups & Capturing**
    - (pattern) â†’ captures group
    - DSA Example: Convert snake_case to CamelCase.
        - String s = "hello_world";
        - String camel = s.replaceAll("_(.)", m -> m.group(1).toUpperCase()); // "helloWorld"
    - Explanation:
        - _(.) captures the character after _. group(1).toUpperCase() converts it to uppercase.

- **Negation / Exclusion**
    - [^...] â†’ exclude characters inside brackets
    - DSA Example: Keep only letters from string.
        - String s = "Hello123!";

        - String lettersOnly = s.replaceAll("[^a-zA-Z]", ""); // "Hello"
    - Explanation:
        - [^a-zA-Z] matches anything except letters. replaceAll removes them.

- **Predefined Classes**
    - \d â†’ digit [0-9]
    - \D â†’ non-digit [^0-9]
    - \w â†’ word [a-zA-Z0-9_]
    - \W â†’ non-word
    - \s â†’ whitespace
    - \S â†’ non-whitespace
    - DSA Tip: Combine these for filtering strings efficiently instead of manual iteration.

- **Useful Methods in Java**
    - matches(regex) â†’ check if full string matches pattern
    - replaceAll(regex, replacement) â†’ replace all matches
    - replaceFirst(regex, replacement) â†’ replace first match
    - split(regex) â†’ split string by pattern
    - Pattern.compile(regex) & Matcher â†’ advanced matching with groups
    - Explanation:
    - These methods allow you to search, modify, and split strings in one line instead of loops.

- **DSA Usage Examples**
    - Remove non-alphabetic characters: [a-zA-Z]
        - String s = "a1b2c3!";
        - String letters = s.replaceAll("[^a-zA-Z]", ""); // "abc"
    - Check Pangram / unique letters: [a-z] with bitmask or regex filtering
    - CamelCase conversion: _(.) with replaceAll
    - Validate input strings: ^[a-zA-Z0-9]+$
    - Extract numbers from string: \d+
    - Explanation:
        - Regex simplifies data cleaning and pattern checks commonly used in string DSA problems.

- **Tips & Observations**
    - Regex is fast for matching/cleaning strings in O(n) time.
    - Use precompiled Pattern for repeated regex to reduce overhead.
    - Remember escaping characters (\ → \ in Java).
    - For DSA prep, focus on:
    - Cleaning strings [^\p{L}] / [^\w]
    - Splitting strings by whitespace \s+
    - Group capture for transformations (.)

Q55: Pangrams

---

1. Problem Understanding

- You are given a string s, and you need to determine whether it is a pangram â€” a sentence that contains every letter of the English alphabet at least once.
- Ignore case sensitivity and spaces.
- Return "pangram" if all 26 letters (aâ€“z) are present, otherwise return "not pangram".

---

2. Constraints

- 0<âˆ£sâˆ£â‰¤1000
- Each character s[i] âˆˆ {aâ€“z, Aâ€“Z, space}

---

3. Edge Cases

- Empty string â†’ "not pangram"
- String with only spaces â†’ "not pangram"
- Mixed case letters should count as same letter
- Long strings with missing one or two letters
- Punctuation is not included here, only spaces and letters

---

4. Examples

```
Example 1:

Input: "We promptly judged antique ivory buckles for the next prize"

Output: pangram

âœ… Explanation: Every letter from â€˜aâ€™ to â€˜zâ€™ is present.

Example 2:

Input: "We promptly judged antique ivory buckles for the prize"

Output: not pangram

â□Œ Explanation: The string is missing the letter 'x'.
```

---

5. Approaches

Approach 1: Using a Set

Idea:

- Convert the string to lowercase.

- Add every character to a set only if itâ€™s a letter.
- If the setâ€™s size becomes 26, itâ€™s a pangram.

Steps:

- Initialize an empty set.
- Traverse through each character of the string.
- If the character is an alphabet, add it to the set.
- Finally, check if the size of the set == 26.

Java Code:

```
class Solution {
    public static String isPangram(String s) {
        s = s.toLowerCase();
        Set<Character> letters = new HashSet<>();

        for (char c : s.toCharArray()) {
            if (c >= 'a' && c <= 'z')
                letters.add(c);
        }

        return letters.size() == 26 ? "pangram" : "not pangram";
    }
}
```

Complexity (Time & Space):

- Time: O(n) â€” single traversal of the string
- Space: O(1) â€” at most 26 characters stored

Approach 2: Using Boolean Array

Idea:

- Maintain a boolean array of size 26 for each letter.
- Mark each letter encountered as true.
- In the end, check if all are true.

Steps:

- Initialize a boolean array of size 26 with false.
- For each letter, mark the corresponding index (c - 'a') as true.
- After processing, check if all entries are true.

Java Code:

```
class Solution {
    public static String isPangram(String s) {
        boolean[] seen = new boolean[26];
```

```
        s = s.toLowerCase();

        for (char c : s.toCharArray()) {
            if (c >= 'a' && c <= 'z')
                seen[c - 'a'] = true;
        }

        for (boolean b : seen) {
            if (!b) return "not pangram";
        }
        return "pangram";
    }
}
```

Complexity (Time & Space):

- Time: O(n + 26) â‰ˆ O(n)
- Space: O(1)

Approach 3: Using Bitmask (Optimized Space)

Idea:

- Represent each letterâ€™s presence with a bit in an integer.
- Toggle the bit for each letter found.
- If the final mask has all 26 bits set, itâ€™s a pangram.

Steps:

- Initialize mask = 0.
- For each letter, set the bit (1 << (c - 'a')).
- After iteration, compare mask with (1 << 26) - 1.

Java Code:

```
class Solution {
    public static String isPangram(String s) {
        int mask = 0;
        s = s.toLowerCase();

        for (char c : s.toCharArray()) {
            if (c >= 'a' && c <= 'z')
                mask |= (1 << (c - 'a'));
        }

        return mask == (1 << 26) - 1 ? "pangram" : "not pangram";
    }
}
```

Complexity (Time & Space):

- Time: O(n)
- Space: O(1)

---

6. Justification / Proof of Optimality

- Using a set is simplest and intuitive.
- The boolean array is slightly faster.
- The bitmask approach is optimal in space and elegant for interview answers.

---

7. Variants / Follow-Ups

- âœ… Check pangrammatic lipogram â†’ a sentence missing exactly one letter.
- âœ… Count how many letters are missing to form a pangram.
- âœ… Apply to Unicode alphabets or other languages.

---

8. Tips & Observations

- âœ… You can safely ignore spaces and case â€” always convert to lowercase first.
- âœ… 26 is constant, so any alphabet-based structure (set/boolean array) uses O(1) space.
- âœ… Bitmasking is a smart trick often reused in similar problems like â€œCheck if two strings are anagramsâ€□ or â€œFind unique charactersâ€□.
- âœ… If youâ€™re asked to find missing letters, you can easily list them using a simple check on unmarked positions.
- âœ… Common interview follow-up: â€œCan you do this for non-English alphabets (like Unicode)?â€□ â€” leads into set-based or dictionary-based solutions.

Q56: Camel Case Conversion

---

1. Problem Understanding

- Convert a string with words separated by underscores (_) to camel case.
- Rules:
  - Remove underscores _.
  - First letter of the entire string â†’ lowercase.
  - First letter of each subsequent word â†’ uppercase.

---

2. Constraints

- 1 <= T <= 10 (number of test cases)
- 1 <= |S| <= 100000 (length of each string)
- String contains lowercase letters and underscores only.

---

3. Edge Cases

- String has consecutive underscores: "a__b" â†’ "aB"
- String starts or ends with underscores: "*abc*" â†’ "Abc"
- String with only one word â†’ stays lowercase
- String with maximum length â†’ performance matters (regex vs iterative)

---

4. Examples

```
Example 1
Input: abb_b_cc_d
Output: abbBCcD

Example 2
Input:how_are_you
Output:howAreYou

Example 3 (Edge Case)
Input: __start_middle__end__
Output: StartMiddleEnd
```

---

5. Approaches

Approach 1: Iterative (Character Traversal)

Idea:

- Traverse string character by character. Track previous character to detect underscores. Convert next character to uppercase if previous was _.

Steps:

- Initialize prev = ' ' (space).
- For each character c in string:
  - If c is not _ and prev == '_' â†’ uppercase c.
  - Else if c is not _ â†’ keep as is.
  - Skip underscores.
  - Update prev = c.

Java Code:

```java
static void camelCaseIterative(String s) {
    char prev = ' ';
    for(int i = 0; i < s.length(); i++){
        char c = s.charAt(i);
        if(c != '_' && prev == '_') System.out.print(Character.toUpperCase(c));
        else if(c != '_') System.out.print(c);
```

```
            prev = c;
        }
    }
```

Complexity (Time & Space):

- Time â†’ O(n)
- Space â†’ O(1) (in-place printing)

Approach 2: Using StringBuilder

Idea:

- Similar to iterative, but store result in StringBuilder to return string instead of printing.

Steps:

- Initialize StringBuilder sb.
- Traverse string, skip underscores.
- Capitalize first letter after _.
- Append to sb

Java Code:

```
static String camelCaseSB(String s) {
    StringBuilder sb = new StringBuilder();
    boolean upperNext = false;
    for(char c : s.toCharArray()){
        if(c == '_') upperNext = true;
        else {
            sb.append(upperNext ? Character.toUpperCase(c) : c);
            upperNext = false;
        }
    }
    return sb.toString();
}
```

Complexity (Time & Space):

- Time â†’ O(n)
- Space â†’ O(n) (for StringBuilder)

Approach 3: Regex Approach

Idea:

- Use regular expression to find underscores followed by a character and replace them with the uppercase character.

Steps:

- Use regex "_([a-z])" to capture the character after _.
- Replace match with its uppercase using replaceAll() and lambda.
- Optional: convert first character to lowercase if needed.

Java Code:

```
static String camelCaseRegex(String s) {
    // Replace '_' followed by a letter with uppercase letter
    String result = s.replaceAll("_(.)", m -> m.group(1).toUpperCase());
    return result;
}

Method Calls Explained:
replaceAll(regex, lambda) â†’ finds all matches of regex and replaces them.
_(.) â†’ matches underscore + single character; () captures the character.
m.group(1) â†’ retrieves captured character.
toUpperCase() â†’ converts it to uppercase.
```

Complexity (Time & Space):

- Time â†’ O(n) (Java regex engine scans the string)
- Space â†’ O(n) (new string created)

---

6. Justification / Proof of Optimality

- Iterative â†’ fastest, minimal memory.
- StringBuilder â†’ easier to return string, avoids repeated print.
- Regex â†’ very concise, ideal for pattern-based transformations, slightly slower for huge strings.

---

7. Variants / Follow-Ups

- PascalCase (first letter uppercase)
- SnakeCase â†” CamelCase conversions
- Remove digits, punctuation before camel case (replaceAll("[^a-zA-Z_]", ""))
- Kebab-case (-) to CamelCase

---

8. Tips & Observations

- Use Character.toUpperCase() / Character.toLowerCase() instead of ASCII math.
- Regex "_(.)" is versatile and works for any character after underscore.
- For strings of length > 10^5, iterative approach is slightly faster than regex.
- Combining replaceAll("[^a-zA-Z_]", "") first can clean string before camel case.

Q57: All Substrings of a String

---

1. Problem Understanding

- Given a string str, print all possible non-empty substrings of it, in order of occurrence.
- A substring is a contiguous sequence of characters within a string.

---

2. Constraints

- 0â‰¤âˆ£strâˆ£â‰¤280
- String contains only lowercase English letters (typically).
- The output may have up to
- nÃ—(n+1)/2 substrings â†’ be cautious for large inputs.

---

3. Edge Cases

- Empty string ("") â†’ no output.
- Single character ("a") â†’ only "a".
- Repeated characters (e.g. "aaa") â†’ substrings will repeat but are considered valid

---

4. Examples

```
Example 1:
Input:abc
Output:
a
ab
abc
b
bc
c
Explanation: All possible non-empty substrings of "abc" printed in order.

Example 2:
Input: abcd
Output:
a
ab
abc
abcd
b
bc
bcd
c
cd
d
```

---

5. Approaches

Approach 1: Brute Force using Nested Loops (Simple & Clear)

Idea:

- Generate all substrings using two loops â€" start index i and end index j.
- For every (i, j) pair, print the substring str.substring(i, j).

Steps:

- Loop i from 0 to n-1 (start index).
- For each i, loop j from i+1 to n (end index exclusive).
- Print the substring str.substring(i, j).

Java Code:

```
public static void printSubstrings(String str) {
    int n = str.length();
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j <= n; j++) {
            System.out.println(str.substring(i, j));
        }
    }
}
```

Complexity (Time & Space):

- Time: O(n^2) â†' total substrings = n(n+1)/2
- Space: O(1) â†' no extra space used

Approach 2: Using StringBuilder (More Efficient for Printing)

Idea:

- Use a StringBuilder to build substrings incrementally, avoiding repeated string creation.

Steps:

- Initialize a StringBuilder for each i.
- Append characters one by one to build and print substrings.
- This avoids creating multiple intermediate string objects.

Java Code:

```
public static void printSubstrings(String str) {
    int n = str.length();
    for (int i = 0; i < n; i++) {
        StringBuilder sb = new StringBuilder();
        for (int j = i; j < n; j++) {
            sb.append(str.charAt(j));
            System.out.println(sb.toString());
```

```
        }
    }
}
```

Complexity (Time & Space):

- Time: O(n^2) (same as before)
- Space: O(n) for StringBuilder buffer
- Why Better:
- Avoids repeatedly creating new string objects in memory â€” more efficient for large strings.

Approach 3: Recursive (Less Common, Educational)

Idea:

- Use recursion to print all substrings by reducing the problem â€” fix a start index and recursively print the rest.

Steps:

- Start from index i=0, recursively print substrings starting at i.
- Increment start index and repeat until end.

Java Code:

```
public static void printSubstrings(String str, int start, int end) {
    if (start == str.length()) return;
    if (end == str.length() + 1) {
        printSubstrings(str, start + 1, start + 1);
        return;
    }
    System.out.println(str.substring(start, end));
    printSubstrings(str, start, end + 1);
}
Call:

printSubstrings("abc", 0, 1);
```

Complexity (Time & Space):

- Time: O(n^2)
- Space: O(n) recursion stack

---

6. Variants / Follow-Ups

- Count Distinct Substrings
    - Use Trie or Suffix Array to count unique substrings efficiently.
- Longest Palindromic Substring

    - Similar substring enumeration, but with palindrome checking (O(nÂ²) or Manacherâ€™s Algorithm O(n)).
- Substring Search / Matching
    - Problems like â€œFind substring in another stringâ€□ use similar substring concepts + pattern matching.
- Count Substrings with Given Property
    - Example: count substrings containing equal 0s and 1s, or at most k distinct characters.
- Generate Substrings for Pattern Problems
    - Used in DP for string segmentation or word break problems.

---

7. Tips & Observations

- Total non-empty substrings = n(n+1)/2
- Using StringBuilder avoids memory waste
- Substrings differ from subsequences (substrings are contiguous)
- Avoid using recursion for large strings (stack overflow risk)
- For competitive programming, prefer Approach 2

Q58: Good Strings

---

1. Problem Understanding

- You are given a set of distinct characters S and an array of strings A.
- A string in A is called â€œgoodâ€□ if all its characters are present in S.
- We must count how many strings in A are good.

---

2. Constraints

- 1 â‰¤ T â‰¤ 10
- 1 â‰¤ |S| â‰¤ 26
- 1 â‰¤ N â‰¤ 1000
- 1 â‰¤ |A[i]| â‰¤ 1000
- Characters are lowercase English letters.

---

3. Edge Cases

- All strings are good (when all characters in A exist in S).
- No string is good (when none of the characters match).
- Empty S â†’ all counts = 0.
- Single-character strings (should still be valid).
- Large test cases (N = 1000, |A[i]| = 1000) â†’ efficiency matters.

---

4. Examples

```
Example 1:
Input:

1
abc
4
ab
abd
cacb
cabef


Output:

2


Explanation:
â€… â€œabâ€□, â€œcacbâ€□ are good â€” all letters are within {a, b, c}.
â□Œ â€œabdâ€□, â€œcabefâ€□ contain disallowed characters (d, e, f).

Example 2:
Input
1
pq
3
pqqqpp
abc
rst
Output:
1
Explanation:
â€… â€œpqqqppâ€□ is good.
â□Œ â€œabcâ€□ and â€œrstâ€□ have letters not in {p, q}.
```

---

5. Approaches

Approach 1: Brute Force (Using String.contains())

Idea:

- For each string in A, check every character using s.contains(char) to verify if it exists in the set string S.

Steps:

- Loop through all strings in A.
- For each string, loop through its characters.
- If any character is not in S, mark it as invalid.

Java Code:

```
static int goodStrings(String s, String[] A, int n) {
    int c = 0;
    for (int i = 0; i < A.length; i++) {
        String z = A[i];
        boolean f = true;
        for (int j = 0; j < z.length(); j++) {
            if (!s.contains(String.valueOf(z.charAt(j)))) {
                f = false;
                break;
            }
        }
        if (f) c++;
    }
    return c;
}
```

Complexity (Time & Space):

- Time: O(N Ã— |A[i]| Ã— |S|)
- Space: O(1)
- âš ï¸ Drawback:
- Repeated .contains() makes it inefficient for large strings.

Approach 2: Using HashSet (Optimized O(1) Lookup)

Idea:

- Store characters of S in a HashSet for O(1) lookup per character.

Steps:

- Convert all characters of S into a HashSet.
- For each string in A, check every characterâ€™s existence using the HashSet.
- Count if all characters are found.

Java Code:

```
static int goodStrings(String s, String[] A, int n) {
    Set<Character> set = new HashSet<>();
    for (char ch : s.toCharArray()) set.add(ch);

    int count = 0;
    for (String word : A) {
        boolean good = true;
        for (char ch : word.toCharArray()) {
            if (!set.contains(ch)) {
                good = false;
                break;
```

```
                }
            }
            if (good) count++;
        }
        return count;
    }
```

Complexity (Time & Space):

- Time: O(N Ã— |A[i]|)
- Space: O(|S|)
- âœ… Advantages:
    - O(1) membership check per character.
    - Simple and very readable.
    - Great practical balance between simplicity and efficiency.

Approach 3: Bitmasking (Most Optimal)

Idea:

- Represent the set S and each word as a bitmask of 26 bits (for each lowercase letter).
- If all bits of a word exist in S, the word is â€œgoodâ€□.

Steps:

- Create a bitmask for S â†’ set bits for each letter.
- For each word:
    - Create its bitmask.
    - Check (maskWord & ~maskS) == 0.
- If true, increment count.

Java Code:

```
static int goodStrings(String s, String[] A, int n) {
    int maskS = 0;
    for (char ch : s.toCharArray()) {
        maskS |= (1 << (ch - 'a'));
    }

    int count = 0;
    for (String word : A) {
        int maskW = 0;
        for (char ch : word.toCharArray()) {
            maskW |= (1 << (ch - 'a'));
        }

        if ((maskW & ~maskS) == 0) // all chars in S
            count++;
    }
```

```
        return count;
    }
```

Complexity (Time & Space):

- Time: O(N Ã— |A[i]|)
- Space: O(1)
- âœ… Advantages:
- Extremely fast (bitwise operations = O(1)).
- Ideal when only lowercase letters are used.
- No extra data structures.

---

6. Justification / Proof of Optimality

- Approach 1: Easiest to understand but slow due to O(nÂ²) checks.
- Approach 2: Best balance between clarity and performance â€” widely used in real-world DSA problems.
- Approach 3: Most efficient (constant space and O(1) bit-level checks), perfect for large test cases or competitive programming.

---

7. Variants / Follow-Ups

- Count bad strings instead of good ones.
- Return the list of good strings instead of count.
- Handle mixed-case alphabets or Unicode (extend bitmask to HashSet).
- Similar problems:
    - â€œCheck if two strings are anagramsâ€□
    - â€œFind common characters across all stringsâ€□
    - â€œPangram Checkâ€□

---

8. Tips & Observations

- Prefer HashSet for flexible character sets.
- Use Bitmasking for only lowercase alphabets â†’ super fast O(1).
- Precompute bitmasks when handling multiple test cases.
- Avoid using String.contains() in nested loops â€” itâ€™s O(nÂ²).
- This problem teaches character presence checking â€” a fundamental subskill for substring, anagram, and frequency-based problems.

Q59: Camel Case Word Separator

---

1. Problem Understanding

- You are given a string S written in Camel Case format (e.g., IAmAJavaProgrammer).
- Each new word starts with an uppercase letter.
- You must separate and print each word on a new line.
- The goal is to enhance readability by splitting the Camel Case string into individual words.

---

2. Constraints

- 1â‰¤â^£Sâ^£â‰¤10^4
- S contains only English letters (a-z, A-Z).

---

3. Edge Cases

- String has only one word â†’ "Java" â†’ Output: Java
- String has all uppercase letters â†’ "ABC" â†’ Output:
- A
- B
- C
- String starts with lowercase (non-standard CamelCase) â†’ handle gracefully.

---

4. Examples

```
Example 1

Input:  IAmAJavaProgrammer
Output:
I
Am
A
Java
Programmer
```

---

5. Approaches

Approach 1: Index-Based Substring (Mathematical Traversal)

Idea:

- Use two pointers (start, i) to mark word boundaries.
- Every time you encounter an uppercase character, treat it as the beginning of a new word.
- Use substring extraction to print words.

Steps:

- Initialize start = 0
- Traverse string from index 1 to n-1

- If str.charAt(i) is uppercase:
  - Print substring from start to i
  - Update start = i
- After loop, print substring from start to end

Java Code:

```
public static void solution(String str) {
    int n = str.length();
    int start = 0;

    for (int i = 1; i < n; i++) {
        if (Character.isUpperCase(str.charAt(i))) {
            System.out.println(str.substring(start, i));
            start = i;
        }
    }
    System.out.println(str.substring(start));
}
```

Complexity (Time & Space):

- Time: O(n)
- Space: O(1)

Approach 2: Manual Traversal (Using StringBuilder)

Idea:

- Build words character by character.
- When an uppercase letter is found (and a word already exists), print the accumulated word.
- Continue building until the next uppercase boundary.

Steps:

- Initialize an empty StringBuilder
- For each character:
  - If uppercase and builder not empty â†' print word and reset builder
  - Append current character
- Print the final word after loop

Java Code:

```
public static void solution(String str) {
    StringBuilder word = new StringBuilder();

    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        if (Character.isUpperCase(ch) && word.length() > 0) {
```

```
            System.out.println(word.toString());
            word.setLength(0);
        }
        word.append(ch);
    }

    if (word.length() > 0) {
        System.out.println(word.toString());
    }
}
```

Complexity (Time & Space):

- Time: O(n)
- Space: O(1)

Approach 3: Regex Split (String Utility)

Idea:

- Use a regex pattern to split before every uppercase letter.
- Pattern (?=[A-Z]) means â€œsplit before uppercase characters.â€□

Steps:

- Use split("(?=[A-Z])") to divide the string.
- Print each resulting substring.

Java Code:

```
public static void solution(String str) {
    String[] words = str.split("(?=[A-Z])");
    for (String w : words) {
        System.out.println(w);
    }
}
```

Complexity (Time & Space):

- Time: O(n)
- Space: O(k) (where k = number of words)

---

6. Justification / Proof of Optimality

- The index-based approach is the most memory-efficient and logical (O(1) space).
- The StringBuilder approach offers clarity and flexibility for modifications.
- The regex approach is concise but involves slightly higher runtime overhead due to regex processing.

---

7. Variants / Follow-Ups

- Convert CamelCase â†’ snake_case (IAmAJavaProgrammer â†’ i_am_a_java_programmer)
- Count the number of words instead of printing them
- Handle PascalCase or mixedCase strings

---

8. Tips & Observations

- Character.isUpperCase(ch) helps detect word starts safely.
- Avoid concatenating strings directly inside loops â€“ use StringBuilder or substring slicing.
- Regex (?=[A-Z]) is a powerful pattern to handle CamelCase splits elegantly.
- The index-based traversal is the most intuitive and optimal for large inputs.

Q60: Distinct Palindromic Substrings

---

1. Problem Understanding

- Given a string s, find all substrings that are palindromic.
- Print these substrings in lexicographical (alphabetical) order.
- Palindrome: A string that reads the same forward and backward.
- Substring: A contiguous sequence of characters within a string.

---

2. Constraints

- 1â‰¤â£sâ£â‰¤1000
- String contains only English letters (a-z, A-Z).

---

3. Edge Cases

- Single-character strings â†’ the single character is a palindrome.
- All characters identical â†’ every substring is a palindrome.
- No repeated palindromes should be printed more than once.

---

4. Examples

```
Example 1
Input:  abc
Output:
a
b
c

Example 2
```

```
Input:  abccbc
Output:
a
b
bccb
c
cbc
cc
```

---

5. Approaches

Approach 1: Brute Force (Generate All Substrings)

Idea:

- Generate all possible substrings of s.
- Check each substring for palindrome property.
- Use a Set to store unique palindromic substrings.
- Sort the set lexicographically.

Steps:

- Initialize a Set for unique palindromes.
- For all pairs (i, j) where 0 â‰¤ i â‰¤ j < n:
  - Extract substring s[i...j].
  - Check if it is a palindrome.
  - If yes, add to the set.
- Convert the set to a list and sort lexicographically.
- Print each substring.

Java Code:

```java
public static void solution(String s) {
    Set<String> palSet = new HashSet<>();

    int n = s.length();
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            String sub = s.substring(i, j + 1);
            if (isPalindrome(sub)) {
                palSet.add(sub);
            }
        }
    }

    List<String> palList = new ArrayList<>(palSet);
    Collections.sort(palList);
    for (String p : palList) {
        System.out.println(p);
```

```
        }
    }

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

Complexity (Time & Space):

- Time: O(nÂ³) â†' O(nÂ²) substrings Ã— O(n) palindrome check
- Space: O(k) â†' number of distinct palindromes

Approach 2: Expand Around Centers

Idea:

- Each palindrome is symmetric around a center.
- A palindrome can have one center (odd length) or two centers (even length).
- Expand around every possible center and collect palindromes.

Steps:

- Initialize a Set for unique palindromes.
- For each index i in the string:
  - Expand around center i (odd-length palindrome)
  - Expand around center (i, i+1) (even-length palindrome)
  - Add each found substring to the set
- Sort and print the set lexicographically.

Java Code:

```
public static void solution(String s) {
    Set<String> palSet = new HashSet<>();
    int n = s.length();

    for (int i = 0; i < n; i++) {
        expand(s, i, i, palSet);     // odd length
        expand(s, i, i + 1, palSet); // even length
    }

    List<String> palList = new ArrayList<>(palSet);
    Collections.sort(palList);
    for (String p : palList) {
        System.out.println(p);
    }
}
```

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

Complexity (Time & Space):

- Time: O(nÂ²) â†' expanding from each center takes O(n) in total
- Space: O(k) â†' distinct palindromic substrings

Approach 3: Dynamic Programming (DP Table)

Idea:

- Use a DP table dp[i][j] to store whether substring s[i..j] is a palindrome.
- Build table for increasing lengths.
- Store palindromic substrings in a set.

Steps:

- Initialize boolean dp[n][n]
- For all i: dp[i][i] = true â†' single letters
- For all pairs (i, j) in increasing length:
  - dp[i][j] = true if s[i] == s[j] and (j-i < 2 || dp[i+1][j-1])
- Add s[i..j] to set if dp[i][j] == true
- Sort and print

Java Code:

```
public static void solution(String s) {
    int n = s.length();
    boolean[][] dp = new boolean[n][n];
    Set<String> palSet = new HashSet<>();

    for (int len = 1; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s.charAt(i) == s.charAt(j) && (len <= 2 || dp[i+1][j-1])) {
                dp[i][j] = true;
                palSet.add(s.substring(i, j + 1));
            }
        }
    }

    List<String> palList = new ArrayList<>(palSet);
    Collections.sort(palList);
    for (String p : palList) {
        System.out.println(p);
```

```
        }
    }
```

Complexity (Time & Space):

- Time: O(nÂ²)
- Space: O(nÂ² + k) â†’ DP table + set of palindromes

---

6. Justification / Proof of Optimality

- Brute force: simple but inefficient (O(nÂ³))
- Expand around center: optimal O(nÂ²) solution, intuitive
- DP: also O(nÂ²), useful if you need extra information about palindrome positions

---

7. Variants / Follow-Ups

- Count number of distinct palindromes instead of printing
- Find the longest palindromic substring
- Modify to return palindromes of length â‰¥ k
- Lexicographically smallest/largest palindrome

---

8. Tips & Observations

- Use a Set to remove duplicates automatically.
- Expanding around centers is often simpler than DP for substring printing.
- Sorting lexicographically can be done via Collections.sort() in Java.
- For large strings, avoid brute force (O(nÂ³)) due to time constraints.

Q61: Shuffle String

---

1. Problem Understanding

- Given a string s and an integer array indices of the same length, shuffle the string such that the character at the ith position moves to indices[i] in the shuffled string.
- Print the shuffled string.

---

2. Constraints

- 1 <= n <= 500 (length of string)
- indices contains all integers from 0 to n-1 without repetition

---

3. Edge Cases

- No shuffling needed (indices = [0,1,2,...]) â†' string stays same
- Single-character string â†' no change
- Maximum length string â†' solution must be O(n)

---

4. Examples

```
Example 1
s = "acciojob"
indices = [4,5,6,7,0,2,1,3]
Output: oojbacci

Example 2
s = "abc"
indices = [0,1,2]
abc
```

---

5. Approaches

Approach 1: Direct Mapping (Extra Array)

Idea:

- Create a new char array of the same length as s.
- Place each character at its target index given by indices.

Steps:

- Initialize char[] result = new char[n].
- Loop i = 0 to n-1:
  - result[indices[i]] = s.charAt(i)
- Convert result to a string and print

Java Code:

```
static void shuffleString(int[] indices, String s) {
    char[] result = new char[s.length()];
    for (int i = 0; i < s.length(); i++) {
        result[indices[i]] = s.charAt(i);
    }
    System.out.print(new String(result));
}
```

Complexity (Time & Space):

- Time: O(n) â†' single pass through string
- Space: O(n) â†' extra array for result

Approach 2: In-Place Swapping (Space Optimized)

Idea:

- Swap characters in the original array to their correct positions using indices.
- Avoid extra space for result array.

Steps:

- Convert s to char[] sArr.
- Loop i = 0 to n-1:
- While indices[i] != i:
  - Swap sArr[i] with sArr[indices[i]]
  - Swap indices[i] with indices[indices[i]]
- Print sArr

Java Code:

```
static void shuffleStringInPlace(int[] indices, String s) {
    char[] sArr = s.toCharArray();
    for (int i = 0; i < sArr.length; i++) {
        while (indices[i] != i) {
            int target = indices[i];
            // swap characters
            char tempChar = sArr[i];
            sArr[i] = sArr[target];
            sArr[target] = tempChar;
            // swap indices
            int tempIndex = indices[i];
            indices[i] = indices[target];
            indices[target] = tempIndex;
        }
    }
    System.out.print(new String(sArr));
}
```

Complexity (Time & Space):

- Time: O(n) â†' each element visited at most once
- Space: O(1) â†' in-place, no extra array

---

6. Justification / Proof of Optimality

- Approach 1: Simple, intuitive, safe for small n (â‰¤500)
- Approach 2: More space-efficient, optimal for large arrays

---

7. Variants / Follow-Ups

- Shuffle only a substring
- Shuffle multiple strings using the same indices
- Handle repeated indices â†’ characters may overwrite each other

---

8. Tips & Observations

- Always distinguish pick vs place: this problem is about placing characters at indices, not picking from indices.
- In-place swapping avoids extra memory but modifies input indices.
- For small n, using extra array is simplest and less error-prone.

Q62: Autori: Shorten Author Names

---

1. Problem Understanding

- Input: A "long variation" of authors, e.g., Knuth-Morris-Pratt.
- Task: Convert to "short variation" using only the first letters of each last name, e.g., KMP.
- **Rules:**
  - Names are separated by hyphens -.
  - Each name starts with an uppercase letter.
  - Hyphens are always followed by uppercase letters.

---

2. Constraints

- 1 â‰¤ |s| â‰¤ 100
- Only letters (a-z, A-Z) and hyphens -

---

3. Edge Cases

- Single author â†’ return the first letter
- Maximum length string â†’ |s| = 100

---

4. Examples

```
Example 1
Input: Knuth-Morris-Pratt
Output: KMP

Example 2
```

```
Input: Mirko-Slavko
Output: MS
```

---

5. Approaches

Approach 1: Split and concatenate (your current approach)

Idea:

- Split string by -
- Take first character of each substring
- Concatenate

Java Code:

```
static String autori(String str) {
    String[] parts = str.split("-");
    String result = "";
    for (String part : parts) {
        result += part.charAt(0);
    }
    return result;
}
```

Complexity (Time & Space):

- Time: O(n) â†' split + iterate through characters
- Space: O(n) â†' array of substrings + result string

Approach 2: Single pass without split (more optimal)

Idea:

- Iterate over the string once
- Append a character to result if it is uppercase (first character of each name)

Java Code:

```
static String autori(String str) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        if (Character.isUpperCase(ch)) {
            sb.append(ch);
        }
    }
    return sb.toString();
}
```

Complexity (Time & Space):

- Time: O(n) â†’ single pass through string
- Space: O(n) â†’ StringBuilder
- âœ… No need to create extra array for splitting

---

6. Justification / Proof of Optimality

- Approach 2 is slightly more memory-efficient and avoids creating substrings
- Both approaches are linear in time, which is optimal for |s| â‰¤ 100

---

7. Variants / Follow-Ups

- Handle names with middle initials
- Ignore non-alphabet characters
- Convert all output letters to uppercase (if input format is inconsistent)

---

8. Tips & Observations

- Using StringBuilder avoids repeated string concatenation overhead
- Iterating and checking Character.isUpperCase() is a neat trick when first letters are guaranteed to be uppercase

---

Q63: Maximum Frequency Character

---

1. Problem Understanding

- Input: A string s containing only lowercase letters.
- Task: Return the character that occurs most frequently in s.
- Tie-breaker: If multiple characters have the same maximum frequency, return the lexicographically smallest one.

---

2. Constraints

- 1 < |s| < 100
- Only lowercase English letters (a to z)

---

3. Edge Cases

- All characters occur once â†’ return the smallest (a if present)
- Single character string â†’ return that character

- Multiple characters with same max frequency â†’ choose lexicographically smallest

---

4. Examples

```
Example 1
Input: abbbc
Output: b

Example 2
Input: aabbbccc
Output: b

Example 3
Input: xyzxyz
Output: x
Explanation: x, y, z all appear 2 times â†’ return smallest lexicographically â†’
x
```

---

5. Approaches

Approach 1: Frequency Array (Optimal for lowercase letters)

Idea:

- Count occurrences of each character using an array of size 26.
- Traverse array to find max frequency.
- Traverse in order 'a' â†’ 'z' to get lexicographically smallest in case of tie.

Steps:

- Initialize freq[26] = 0.
- Iterate through string s: freq[s.charAt(i) - 'a']++.
- Initialize maxFreq = 0 and maxChar = 'a'.
- Traverse freq array from 0 to 25:
  - If freq[i] > maxFreq: update maxFreq and maxChar = 'a'+i.
- Return maxChar.

Java Code:

```java
static char MaximumFrequencyChar(String s) {
    int[] freq = new int[26];
    for (char ch : s.toCharArray()) {
        freq[ch - 'a']++;
    }

    int maxFreq = 0;
    char maxChar = 'a';
```

```
    for (int i = 0; i < 26; i++) {
        if (freq[i] > maxFreq) {
            maxFreq = freq[i];
            maxChar = (char) (i + 'a');
        }
    }

    return maxChar;
}
```

Complexity (Time & Space):

- Time: O(n + 26) â†' O(n), n = length of string
- Space: O(26) â†' constant space

Approach 2: HashMap (General approach for any characters)

Idea:

- Use a HashMap to store frequency of each character.
- Traverse keys to find max frequency and lexicographically smallest key in case of tie.

Java Code:

```
static char MaximumFrequencyChar(String s) {
    Map<Character, Integer> freqMap = new HashMap<>();
    for (char ch : s.toCharArray()) {
        freqMap.put(ch, freqMap.getOrDefault(ch, 0) + 1);
    }

    char maxChar = s.charAt(0);
    int maxFreq = freqMap.get(maxChar);

    for (char ch : freqMap.keySet()) {
        int f = freqMap.get(ch);
        if (f > maxFreq || (f == maxFreq && ch < maxChar)) {
            maxFreq = f;
            maxChar = ch;
        }
    }

    return maxChar;
}
Works for any characters (not just lowercase)
```

Complexity (Time & Space):

- Time: O(n + k), k = number of distinct characters
- Space: O(k)

---

6. Justification / Proof of Optimality

- Frequency array is optimal for lowercase letters â†’ constant space + linear time.
- HashMap approach generalizes for any input but slightly slower due to hashing.

---

7. Variants / Follow-Ups

- Return all characters with max frequency in lexicographical order.
- Handle uppercase letters or mixed characters.
- Find second most frequent character.

---

8. Tips & Observations

- Always traverse array or keys in lexicographical order to handle ties.
- Frequency array is more efficient than sorting the string.
- Avoid counting consecutive characters only â€" total frequency matters.

Q64: Ptice

---

1. Problem Understanding

- You are given the correct answers to an exam consisting of n questions (each being A, B, or C).
- Three people â€" Adrian, Bruno, and Goran â€" guess answers according to their fixed repeating patterns:
  - Adrian: A, B, C, A, B, C, ...
  - Bruno: B, A, B, C, B, A, B, C, ...
  - Goran: C, C, A, A, B, B, C, C, A, A, B, B, ...
- You must find:
  - Who got the maximum number of correct answers.
  - Print the highest score.
  - Then, print the names of all who achieved that score (in alphabetical order).

---

2. Constraints

- 1 â‰¤ n â‰¤ 100
- Each character in input string âˆˆ {A, B, C}

---

3. Edge Cases

- All get the same number of correct answers.
- Only one person gets the highest score.
- n smaller than length of any pattern (needs modular repetition).
- All answers are same (e.g. "AAAAAA").

4. Examples

```
Example 1
Input:
5
BAACC
Output:
3
Bruno
Explanation:
Brunoâ€™s sequence best matches the correct answers â€” 3 matches.

Example 2
Input:
9
AAAABBBBB
Output:
4
Adrian
Bruno
Goran
Explanation:
All three get 4 correct answers.
```

5. Approaches

Approach 1: Brute Force Simulation

Idea:

- Generate the repeating pattern for each person up to length n.
- Compare each answer in s to the corresponding answer in each pattern.
- Count matches for each person.
- Find the maximum score and print all who achieved it.

Steps:

- Read n and the string answers.
- Define patterns:
  - String adrian = "ABC";
  - String bruno = "BABC";
  - String goran = "CCAABB";
- For each i in 0 to n-1:
  - Compare answers[i] with pattern[i % pattern.length()].
  - Increment count for matching ones.
- Compute maxScore = max(countA, countB, countG).
- Print maxScore and names (alphabetically) of all with that score.

Java Code:

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        String answers = sc.next();

        String adrian = "ABC";
        String bruno = "BABC";
        String goran = "CCAABB";

        int a = 0, b = 0, g = 0;

        for (int i = 0; i < n; i++) {
            char ans = answers.charAt(i);
            if (ans == adrian.charAt(i % adrian.length())) a++;
            if (ans == bruno.charAt(i % bruno.length())) b++;
            if (ans == goran.charAt(i % goran.length())) g++;
        }

        int max = Math.max(a, Math.max(b, g));
        System.out.println(max);

        if (a == max) System.out.println("Adrian");
        if (b == max) System.out.println("Bruno");
        if (g == max) System.out.println("Goran");
    }
}
```

Complexity (Time & Space):

- Time Complexity: O(n) â€” single pass comparison for each of the three sequences.
- Space Complexity: O(1) â€” only uses counters.

---

6. Justification / Proof of Optimality

- This approach is optimal since it directly compares each answer with the pre-determined pattern in O(n).
- No extra data structures are required, and repeating patterns are handled neatly via modulo indexing.

---

7. Variants / Follow-Ups

- Similar problems can involve:
  - Customizable patterns.
  - Weighted answers (e.g., +2 for correct, -1 for wrong).

- More players with unique repeating patterns.

---

8. Tips & Observations

- Use modulo to repeat patterns efficiently instead of generating full-length strings.
- Sorting names alphabetically can be skipped here since the order â€œAdrian, Bruno, Goranâ€□ is already lexicographical.
- A good problem to practice pattern repetition and string indexing with modulo logic.

Q65: Time Conversion

---

1. Problem Understanding

- You are given a time in 12-hour format (hh:mm:ssAM or hh:mm:ssPM).
- Your task is to convert it to 24-hour (military) format.
- Key Rules:
  - "12:00:00AM" â†’ "00:00:00" (midnight)
  - "12:00:00PM" â†’ "12:00:00" (noon)
  - For AM, hours 01â€“11 stay the same, except 12AM â†’ 00.
  - For PM, hours 01â€–11 become 13â€–23.

---

2. Constraints

- s.length() == 10
- Time format is always valid: "hh:mm:ssAM" or "hh:mm:ssPM"
- â†’ i.e. first 2 chars = hour, next 2 = minutes, next 2 = seconds, last 2 = AM/PM.

---

3. Edge Cases

- "12:00:00AM" â†’ "00:00:00" âœ…
- "12:00:00PM" â†’ "12:00:00" âœ…
- "01:00:00AM" â†’ "01:00:00" âœ…
- "01:00:00PM" â†’ "13:00:00" âœ…
- Works for all values between 01â€–12 hours.

---

4. Examples

```
Example 1
Input:
07:05:45PM
Output:
19:05:45
```

```
Explanation:7 PM means 7 + 12 = 19, while minutes and seconds remain same.

Example 2
Input:
12:01:00PM
Output:
12:01:00
Explanation:12 PM â†’ 12 (no change), minutes and seconds remain same.
```

---

5. Approaches

Approach 1: String Parsing

Idea:

- Extract hour (hh), minutes (mm), seconds (ss), and meridian (AM/PM).
- Apply conversion rule:
  - If AM and hour == 12 â†’ hour = 00
  - If PM and hour != 12 â†’ hour += 12
- Combine back in "HH:MM:SS" format.

Steps:

- Read input string s.
- Split into:
  - hour = Integer.parseInt(s.substring(0, 2))
  - minute = s.substring(3, 5)
  - second = s.substring(6, 8)
  - ampm = s.substring(8, 10)
- Apply conversion rules.
- Print formatted 24-hour time.

Java Code:

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s = sc.next();

        int hour = Integer.parseInt(s.substring(0, 2));
        String minutes = s.substring(3, 5);
        String seconds = s.substring(6, 8);
        String meridian = s.substring(8, 10);

        if (meridian.equals("AM")) {
            if (hour == 12) hour = 0; // midnight case
        } else {
```

```
            if (hour != 12) hour += 12; // add 12 for PM hours
        }

        System.out.printf("%02d:%s:%s", hour, minutes, seconds);
    }
}
```

Complexity (Time & Space):

- Time Complexity: O(1) â€" only fixed string operations.
- Space Complexity: O(1) â€" only a few variables used.

---

6. Justification / Proof of Optimality

- This is the most direct and optimal solution â€" parsing the input and performing constant-time adjustments.
- No loops or data structures needed.

---

7. Variants / Follow-Ups

- Convert 24-hour â†' 12-hour format with AM/PM.
- Handle user input with missing leading zeros (e.g., "7:05:45PM").
- Implement using DateTime API (e.g., LocalTime.parse() in Java 8+).

---

8. Tips & Observations

- %02d ensures leading zeros for single-digit hours (like 07:00:00).
- Always separate logic for AM and PM to avoid mixing edge cases.
- Simple string slicing problems like this test your attention to detail and formatting precision.

Q66: Keyboard Row Words

---

1. Problem Understanding

- You are given an array of lowercase strings.
- You must print all words that can be typed using only one row of the QWERTY keyboard.
- Keyboard layout:
    - Row 1: qwertyuiop
    - Row 2: asdfghjkl
    - Row 3: zxcvbnm
- A word qualifies if all its characters belong to only one of the above rows.

---

2. Constraints

- 1 â‰¤ n â‰¤ 1000
- 1 â‰¤ |s| â‰¤ 100
- Each word contains only lowercase English letters.

---

3. Edge Cases

- Word with characters from multiple rows â†’ â□Œ
- Single-character word (always valid âœ…).
- Duplicate words â†’ print as they appear.
- No valid words â†’ print nothing.

---

4. Examples

```
Example 1
Input:
2
glad
monkey
Output:
glad
Explanation:
glad uses only row 2 characters (asdfghjkl).

Example 2
Input:
3
horse
peter
jass
Output:
peter
jass
Explanation:
Both peter and jass can be typed using characters from a single keyboard row.
```

---

5. Approaches

Approach 1: Set Comparison

Idea:

- Represent each keyboard row as a set of characters.
- For each word:
    - Convert it into a set of its characters.
    - If that set is a subset of any keyboard row set, print the word.

Steps:

- Read integer n.
- Initialize:
  - Set row1 = Set.of('q','w','e','r','t','y','u','i','o','p');
  - Set row2 = Set.of('a','s','d','f','g','h','j','k','l');
  - Set row3 = Set.of('z','x','c','v','b','n','m');
- For each word:
  - Create a character set from it.
  - Check if the wordâ€™s characters are all in row1 or row2 or row3.
- Print valid words.

Java Code:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        sc.nextLine(); // consume newline

        Set<Character> row1 = Set.of('q','w','e','r','t','y','u','i','o','p');
        Set<Character> row2 = Set.of('a','s','d','f','g','h','j','k','l');
        Set<Character> row3 = Set.of('z','x','c','v','b','n','m');

        for (int i = 0; i < n; i++) {
            String word = sc.nextLine().toLowerCase();
            Set<Character> chars = new HashSet<>();
            for (char c : word.toCharArray()) chars.add(c);

            if (row1.containsAll(chars) || row2.containsAll(chars) ||
row3.containsAll(chars)) {
                System.out.println(word);
            }
        }
    }
}
```

Complexity (Time & Space):

- Time Complexity: O(n * |s|)
- â†’ Each wordâ€™s characters are checked once.
- Space Complexity: O(1)
- â†’ Only small fixed-size sets are used.

---

6. Justification / Proof of Optimality

- Using sets makes subset checking (containsAll) efficient and concise.
- This avoids manually checking each character row by row.

---

7. Variants / Follow-Ups

- Handle uppercase letters (just convert input to lowercase).
- Find which row number each valid word belongs to.
- Modify for custom keyboard layouts (input-driven).

---

8. Tips & Observations

- Using Set.containsAll() simplifies row validation greatly.
- Converting all words to lowercase avoids case-mismatch issues.
- Could also be done using string membership checks with allMatch() in Java streams.
