## 1.Fraud Detection Concurrency

The code below represents a multithreaded system that processes financial transactions.

def analyze_transactions(transactions): chunks = split_into_chunks(transactions) suspicious_transactions =[] for chunk in chunks: Thread(target=lambda: suspicious_transactions.extend(check_for_fraud(chunk))).start() for thread in threading.enumerate(): if thread.is_alive(): thread.join() return suspicious_transactions

There is a potential flaw in the system.Which option indicates the flaw? Pick ONE option
check_for_fraud(chunk): The processing time could cause delays.

split_into_chunks(transactions): This might lead to an imbalanced workload.

Shared resource suspicious_transactions:Without synchronization,this could lead to race conditions.
Thread creation: Starting a new thread for each chunk could lead to resource exhaustion.

```
C:Shared resource suspicious_transactions:Without synchronization,this could lead to race conditions.
```

## 3. SQL: Freelance Platform Skill Matching

A freelance platform in development requires a query for an algorithm to match projects with freelancers. The result should have the following columns: title, primary_skill, email, skills_list. ●title - project title .primary_skill - project primary skill email - freelancer email skills_list - freelancer skill list .job_success_score - freelancer job success score

The result should be sorted in ascending order by title, then in descending order of job_success_score, then in ascending order of email. Note: .A freelancer is considered matched if the main skill of a specific project is present in the freelancer's list of skills. . Even if a project did not find a match, it still appeared. Schema

```sql
SELECT
    p.title,
    p.primary_skill,
    f.email,
    f.skills_list,
    COALESCE(f.job_success_score, 0) AS job_success_score
FROM projects p
LEFT JOIN freelancers f
ON LOCATE(p.primary_skill, f.skills_list) > 0
WHERE f.job_success_score >= 0.85 OR f.job_success_score IS NULL
ORDER BY
    p.title ASC,
    job_success_score DESC,
    f.email ASC;
```

## 4. Optimal Storage

The engineers at ByteDance are developing an optimal storage system for storing all the data of the Toutiao platform. For storing a string word of lowercase english alphabets, the following operation is applied to the word at most max_operations number of times - A character is chosen from 'a' to 'z'. All the occurrences of the chosen character in the word are replaced with the previous character in alphabetical order in a circular manner. For example, replace all 'd' with c or replace all 'a' with 'z', The stored_word is the lexicographically smallest string that can be obtained by applying at most max_operations number of operations on the string word. Given a string word and an integer max_operations,find the string stored_word. Note: A string a is lexicographically smaller than the string bif j is the first index at which the two strings differ, then a[j] < b[j] Example Given word = "cba" and max_operations =2. One of the optimal ways to transform the string is: Operation no. Operation word(After) 1 Replace all 'c' with 'b'. "bba" 2 Replace all 'b' with 'a'. "aaa" Hence, the stored_word is "aaa". Function Description Complete the function getStoredWord in the editor below. getStoredWord has the following parameters: string word: the word to be stores int max_operations: the maximum number of operations allowed Returns string: the string stored_word Constraints 1 ≤length of word≤2*105·1 ≤max_operations ≤ 109

```
public class LexicographicallySmallerString {

    /**
     * Converts the given string to a lexicographically smaller string by
applying
     * operations to decrease the character values with a given limit of
max_operations.
     *
     * @param s              The input string.
     * @param max_operations The maximum number of operations that can be
applied.
     * @return A lexicographically smaller string.
     */
    public static String helper(String s, int max_operations) {
        int n = s.length();
        int[] arr = new int[n];

        // Convert each character of the string to its corresponding
alphabetical position (0-25).
        for (int i = 0; i < n; i++) {
            arr[i] = s.charAt(i) - 'a';
        }

        // Find the index of the character whose position is greater than
or equal to max_operations.
        int k = 0;
        for (int i = 0; i < n; i++) {
            if (arr[i] >= max_operations) {
                k = i;
                break;
            }
        }

        // Find the maximum value before index k.
        int maxx = 0;
```

```
        for (int i = 0; i < k; i++) {
            if (arr[i] > maxx) {
                maxx = arr[i];
            }
        }

        // Calculate the difference between max_operations and the maximum
value.
        int extra = max_operations - maxx;

        // Set characters with a value less than or equal to maxx to 'a'.
        for (int i = 0; i < n; i++) {
            if (arr[i] <= maxx) {
                arr[i] = 0;
            }
        }

        // Determine the range to update characters after index k.
        int left = arr[k] - extra;
        int right = arr[k];
        for (int i = 0; i < n; i++) {
            if (arr[i] != 0 && arr[i] <= right) {
                arr[i] = left;
            }
        }

        // Convert the array back to a string.
        StringBuilder strBuilder = new StringBuilder();
        for (int v : arr) {
            strBuilder.append((char) (v + 'a'));
        }

        return strBuilder.toString();
    }

    public static void main(String[] args) {
        String str = helper("yzwyz", 3); //test case given by question
        System.out.println(str);  // Outputs a lexicographically smaller
string based on the logic.
    }
}
```

## 5. Vulnerable Password

The developers at ByteDance have developed a new algorithm for finding the ALL vulnerability of passwords. The more the vulnerability of a password, the higher the risk of getting hacked. A password is represented by a string of lengthn,consisting of English lowercase letters. The vulnerability of a string password is calculated in the following way: √The following operation is performed on the string password at most max_ops times: 。Replace any character of the string password with any English lowercase letter. The vulnerability of the password is the maximum possible length of a substring √having the same

character that can be obtained by applying the above operations optimally. 3 Given a string password and an integer max_ops, find the vulnerability of the 4 password. Note: A substring is a contiguous sequence of characters within a string. 5 Example Given, password ="ababc", and max_ops =1. Some of the ways the operation can be applied are explained below:

Constraints 1 ≤ length of password≤2*105.0≤ max_ops≤2*105 ALL password contains only lowercase English letters.

Input Format For Custom Testing Sample Case 0 Sample Input For Custom Testing √STDIN FUNCTION -- _-- --—---- abcdabcd → password ="abcdabcd" 3 2 → max_ops =2 Sample Output3 5 Explanation One of the optimal ways is: ·Replace password[2]= 'b'to 'a' in the first operation. Replace password[3]= '°C'to 'a'in the second operation.

```
public class VulnerablePassword {
    public static void main(String[] args) {
        String password = "abcdabcd";
        int max_ops = 3;
        System.out.println(vulnerabilityOfPassword(password, max_ops));
    }

    public static int vulnerabilityOfPassword(String password, int
max_ops) {
        int maxLength = 0;
        int n = password.length();

        for (char ch = 'a'; ch <= 'z'; ch++) {  // For each character
            int left = 0, right = 0, opsUsed = 0;

            while (right < n) {
                // If the current character is not our target character,
we increment opsUsed
                if (password.charAt(right) != ch) {
                    opsUsed++;
                }
                // If opsUsed exceeds max_ops, then move the left pointer
to the right
                while (opsUsed > max_ops) {
                    if (password.charAt(left) != ch) {
                        opsUsed--;
                    }
                    left++;
                }
                // Update the maximum length
                maxLength = Math.max(maxLength, right - left + 1);
                right++;
            }
        }
        return maxLength;
    }
}
```