**Problem 1: Write a program to implement a simple spell checker that finds**

**and suggests corrections for misspelled words using string distance**

**calculation**

**Hint =>**

**a. Take user input for a sentence and a dictionary of correct words (stored in an array)**

**b. Create a method to split the sentence into words without using split():**

**● i. Use charAt() to identify word boundaries (spaces, punctuation)**

**● ii. Extract each word using substring() method**

**● iii. Store words in an array**

**c. Create a method to calculate string distance between two words:**

**● i. Count character differences between words of same length**

**● ii. For different lengths, calculate insertion/deletion distance**

**● iii. Return the distance as an integer**

**d. Create a method to find the closest matching word from dictionary:**

**● i. Compare input word with each dictionary word**

**● ii. Find the word with minimum distance**

**● iii. Return the suggestion if distance is within acceptable range (≤ 2)**

**e. Create a method to display spell check results in tabular format:**

**● i. Show original word, suggested correction, distance score**

**● ii. Mark words as "Correct" or "Misspelled"**

**f. The main function processes the sentence and displays comprehensive spell check report**

import java.util.\*;

public class SpellChecker {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// Step a: Take inputs

System.out.println("Enter a sentence to check:");

String sentence = sc.nextLine();

// A simple dictionary

String[] dictionary = {"this", "is", "a", "simple", "spell", "checker",

"program", "written", "in", "java"};

// Step b: Split sentence into words without split()

String[] words = splitWords(sentence);

// Step f: Display results

System.out.println("\n=== SPELL CHECK REPORT ===");

System.out.printf("%-15s %-15s %-10s %-12s\n", "Word", "Suggestion", "Distance", "Status");

System.out.println("-------------------------------------------------------------");

for (String word : words) {

String suggestion = findClosestWord(word.toLowerCase(), dictionary);

int distance = (suggestion.equals(word.toLowerCase())) ? 0 : stringDistance(word.toLowerCase(), suggestion);

String status = (distance == 0) ? "Correct" : "Misspelled";

System.out.printf("%-15s %-15s %-10d %-12s\n", word, suggestion, distance, status);

}

sc.close();

}

// Step b: Split words manually

public static String[] splitWords(String text) {

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

int start = 0;

for (int i = 0; i < text.length(); i++) {

char c = text.charAt(i);

if (!Character.isLetter(c)) { // space or punctuation

if (start < i) {

words.add(text.substring(start, i));

}

start = i + 1;

}

}

if (start < text.length()) {

words.add(text.substring(start));

}

return words.toArray(new String[0]);

}

// Step c: Calculate string distance (basic edit distance)

public static int stringDistance(String w1, String w2) {

int len1 = w1.length();

int len2 = w2.length();

int[][] dp = new int[len1 + 1][len2 + 1];

for (int i = 0; i <= len1; i++) dp[i][0] = i;

for (int j = 0; j <= len2; j++) dp[0][j] = j;

for (int i = 1; i <= len1; i++) {

for (int j = 1; j <= len2; j++) {

if (w1.charAt(i - 1) == w2.charAt(j - 1))

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

else

dp[i][j] = 1 + Math.min(dp[i - 1][j - 1],

Math.min(dp[i - 1][j], dp[i][j - 1]));

}

}

return dp[len1][len2];

}

// Step d: Find closest matching word

public static String findClosestWord(String word, String[] dictionary) {

int minDist = Integer.MAX\_VALUE;

String closest = word; // default to original

for (String dictWord : dictionary) {

int dist = stringDistance(word, dictWord);

if (dist < minDist) {

minDist = dist;

closest = dictWord;

}

}

return (minDist <= 2) ? closest : word; // only suggest if close enough

}

}

**Problem 2: Write a program to create a password strength analyzer and**

**generator using ASCII values and StringBuilder**

**Hint =>**

**a. Take user input for multiple passwords to analyze**

**b. Create a method to analyze password strength using ASCII values:**

**● i. Count uppercase letters (ASCII 65-90)**

**● ii. Count lowercase letters (ASCII 97-122)**

**● iii. Count digits (ASCII 48-57)**

**● iv. Count special characters (other printable ASCII)**

**● v. Check for common patterns and sequences**

**c. Create a method to calculate password strength score:**

**● i. Length points: +2 per character above 8**

**● ii. Character variety: +10 for each type present**

**● iii. Deduct points for common patterns (123, abc, qwerty)**

**● iv. Return strength level: Weak (0-20), Medium (21-50), Strong (51+)**

**d. Create a method using StringBuilder to generate strong passwords:**

**● i. Take desired length as parameter**

**● ii. Ensure at least one character from each category**

**● iii. Fill remaining positions with random characters**

**● iv. Shuffle the password for better randomness**

**e. Create a method to display analysis results in tabular format:**

**● i. Password, Length, Uppercase count, Lowercase count, Digits, Special chars, Score,**

**Strength**

**f. The main function analyzes existing passwords and generates new strong passwords based**

**on user requirements**

import java.util.\*;

public class PasswordStrengthAnalyzer {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// Step a: Take user input for multiple passwords

System.out.println("Enter number of passwords to analyze: ");

int n = sc.nextInt();

sc.nextLine(); // consume newline

String[] passwords = new String[n];

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

System.out.print("Enter password " + (i + 1) + ": ");

passwords[i] = sc.nextLine();

}

// Display analysis results

System.out.println("\n=== PASSWORD ANALYSIS REPORT ===");

System.out.printf("%-15s %-7s %-10s %-10s %-10s %-12s %-6s %-10s\n",

"Password", "Length", "Uppercase", "Lowercase", "Digits", "SpecialChars", "Score", "Strength");

System.out.println("-----------------------------------------------------------------------------------------");

for (String pwd : passwords) {

int[] counts = analyzePassword(pwd);

int score = calculateScore(pwd, counts);

String strength = getStrengthLevel(score);

System.out.printf("%-15s %-7d %-10d %-10d %-10d %-12d %-6d %-10s\n",

pwd, pwd.length(), counts[0], counts[1], counts[2], counts[3], score, strength);

}

// Step d: Generate strong passwords

System.out.print("\nEnter desired length for new strong password: ");

int length = sc.nextInt();

String strongPwd = generateStrongPassword(length);

System.out.println("Generated Strong Password: " + strongPwd);

sc.close();

}

// Step b: Analyze password

public static int[] analyzePassword(String pwd) {

int upper = 0, lower = 0, digit = 0, special = 0;

for (int i = 0; i < pwd.length(); i++) {

char c = pwd.charAt(i);

int ascii = (int) c;

if (ascii >= 65 && ascii <= 90) upper++;

else if (ascii >= 97 && ascii <= 122) lower++;

else if (ascii >= 48 && ascii <= 57) digit++;

else if (ascii >= 33 && ascii <= 126) special++;

}

return new int[]{upper, lower, digit, special};

}

// Step c: Calculate score

public static int calculateScore(String pwd, int[] counts) {

int score = 0;

// Length points

if (pwd.length() > 8) {

score += (pwd.length() - 8) \* 2;

}

// Variety

if (counts[0] > 0) score += 10; // uppercase

if (counts[1] > 0) score += 10; // lowercase

if (counts[2] > 0) score += 10; // digit

if (counts[3] > 0) score += 10; // special

// Deduct points for common patterns

String lowerPwd = pwd.toLowerCase();

if (lowerPwd.contains("123") || lowerPwd.contains("abc") || lowerPwd.contains("qwerty")) {

score -= 10;

}

return Math.max(score, 0); // ensure not negative

}

// Step c: Strength level

public static String getStrengthLevel(int score) {

if (score <= 20) return "Weak";

else if (score <= 50) return "Medium";

else return "Strong";

}

// Step d: Generate strong password using StringBuilder

public static String generateStrongPassword(int length) {

Random rand = new Random();

String upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

String lower = "abcdefghijklmnopqrstuvwxyz";

String digits = "0123456789";

String special = "!@#$%^&\*()-\_=+<>?";

String all = upper + lower + digits + special;

StringBuilder sb = new StringBuilder();

// Ensure at least one of each

sb.append(upper.charAt(rand.nextInt(upper.length())));

sb.append(lower.charAt(rand.nextInt(lower.length())));

sb.append(digits.charAt(rand.nextInt(digits.length())));

sb.append(special.charAt(rand.nextInt(special.length())));

// Fill remaining

for (int i = 4; i < length; i++) {

sb.append(all.charAt(rand.nextInt(all.length())));

}

// Shuffle characters

List<Character> chars = new ArrayList<>();

for (int i = 0; i < sb.length(); i++) {

chars.add(sb.charAt(i));

}

Collections.shuffle(chars);

StringBuilder finalPwd = new StringBuilder();

for (char c : chars) {

finalPwd.append(c);

}

return finalPwd.toString();

}

}

**Problem 3: Write a program to implement a text-based data compression**

**algorithm using character frequency and StringBuilder**

**Hint =>**

**a. Take user input for text to compress**

**b. Create a method to count character frequency without using HashMap:**

**● i. Create arrays to store characters and their frequencies**

**● ii. Use charAt() to iterate through text**

**● iii. Count occurrences of each unique character**

**● iv. Return parallel arrays of characters and frequencies**

**c. Create a method to create compression codes using StringBuilder:**

**● i. Assign shorter codes to more frequent characters**

**● ii. Use numbers/symbols for common characters**

**● iii. Create a mapping table of original character to code**

**● iv. Return the mapping as a 2D array**

**d. Create a method to compress text using the generated codes:**

**● i. Replace each character with its corresponding code**

**● ii. Use StringBuilder for efficient string building**

**● iii. Calculate compression ratio (original size vs compressed size)**

**e. Create a method to decompress the text:**

**● i. Reverse the compression process using the mapping table**

**● ii. Validate that decompression returns original text**

**f. Create a method to display compression analysis:**

**● i. Show character frequency table**

**● ii. Display compression mapping**

**● iii. Show original text, compressed text, decompressed text**

**● iv. Calculate and display compression efficiency percentage**

**g. The main function performs compression, decompression, and displays complete analysis**

import java.util.\*;

public class TextCompression {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// Step a: Take user input

System.out.print("Enter text to compress: ");

String text = sc.nextLine();

// Step b: Count character frequency

Object[] freqResult = countFrequency(text);

char[] chars = (char[]) freqResult[0];

int[] freqs = (int[]) freqResult[1];

int uniqueCount = (int) freqResult[2];

// Step c: Generate compression codes

String[][] mapping = generateCodes(chars, freqs, uniqueCount);

// Step d: Compress text

String compressed = compressText(text, mapping);

double ratio = (double) compressed.length() / text.length();

// Step e: Decompress text

String decompressed = decompressText(compressed, mapping);

// Step f: Display results

displayAnalysis(chars, freqs, uniqueCount, mapping, text, compressed, decompressed, ratio);

sc.close();

}

// Step b: Count character frequency (no HashMap)

public static Object[] countFrequency(String text) {

char[] chars = new char[text.length()];

int[] freqs = new int[text.length()];

int uniqueCount = 0;

for (int i = 0; i < text.length(); i++) {

char c = text.charAt(i);

int index = -1;

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

if (chars[j] == c) {

index = j;

break;

}

}

if (index == -1) { // new character

chars[uniqueCount] = c;

freqs[uniqueCount] = 1;

uniqueCount++;

} else {

freqs[index]++;

}

}

return new Object[]{chars, freqs, uniqueCount};

}

// Step c: Generate compression codes

public static String[][] generateCodes(char[] chars, int[] freqs, int uniqueCount) {

String[][] mapping = new String[uniqueCount][2];

// Sort by frequency (descending)

for (int i = 0; i < uniqueCount - 1; i++) {

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

if (freqs[i] < freqs[j]) {

int tempF = freqs[i];

freqs[i] = freqs[j];

freqs[j] = tempF;

char tempC = chars[i];

chars[i] = chars[j];

chars[j] = tempC;

}

}

}

// Assign shorter codes to frequent chars

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

mapping[i][0] = String.valueOf(chars[i]); // original char

if (i < 10) mapping[i][1] = String.valueOf(i); // single digit for top 10

else mapping[i][1] = "@" + i; // longer code for less frequent

}

return mapping;

}

// Step d: Compress text

public static String compressText(String text, String[][] mapping) {

StringBuilder sb = new StringBuilder();

for (int i = 0; i < text.length(); i++) {

String ch = String.valueOf(text.charAt(i));

for (String[] map : mapping) {

if (map[0].equals(ch)) {

sb.append(map[1]);

break;

}

}

}

return sb.toString();

}

// Step e: Decompress text

public static String decompressText(String compressed, String[][] mapping) {

StringBuilder sb = new StringBuilder();

int i = 0;

while (i < compressed.length()) {

boolean matched = false;

// First try single-char codes

String token = String.valueOf(compressed.charAt(i));

for (String[] map : mapping) {

if (map[1].equals(token)) {

sb.append(map[0]);

i++;

matched = true;

break;

}

}

if (!matched) {

// Try multi-char code starting with "@"

if (compressed.charAt(i) == '@') {

int j = i + 1;

while (j < compressed.length() && Character.isDigit(compressed.charAt(j))) {

j++;

}

String token2 = compressed.substring(i, j);

for (String[] map : mapping) {

if (map[1].equals(token2)) {

sb.append(map[0]);

i = j;

matched = true;

break;

}

}

}

}

if (!matched) {

i++; // skip unknown codes

}

}

return sb.toString();

}

// Step f: Display analysis

public static void displayAnalysis(char[] chars, int[] freqs, int uniqueCount,

String[][] mapping, String original,

String compressed, String decompressed,

double ratio) {

System.out.println("\n=== CHARACTER FREQUENCY TABLE ===");

System.out.printf("%-10s %-10s\n", "Character", "Frequency");

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

System.out.printf("%-10s %-10d\n", chars[i], freqs[i]);

}

System.out.println("\n=== COMPRESSION MAPPING ===");

System.out.printf("%-10s %-10s\n", "Character", "Code");

for (String[] map : mapping) {

System.out.printf("%-10s %-10s\n", map[0], map[1]);

}

System.out.println("\n=== COMPRESSION RESULTS ===");

System.out.println("Original Text : " + original);

System.out.println("Compressed Text : " + compressed);

System.out.println("Decompressed Text : " + decompressed);

System.out.printf("Compression Efficiency: %.2f%%\n", (1 - ratio) \* 100);

}

}

**Problem 4: Write a program to create a text-based calculator that can parse**

**and evaluate mathematical expressions from strings**

**Hint =>**

**a. Take user input for mathematical expressions as strings (e.g., "15 + 23 \* 4 - 10")**

**b. Create a method to validate expression format:**

**● i. Check for valid characters (digits, operators, spaces, parentheses)**

**● ii. Validate operator placement and parentheses matching**

**● iii. Use ASCII values to identify different character types**

**● iv. Return boolean indicating if expression is valid**

**c. Create a method to parse numbers from string:**

**● i. Use charAt() to identify digit sequences**

**● ii. Extract multi-digit numbers using substring()**

**● iii. Convert string numbers to integers**

**● iv. Store numbers and operators in separate arrays**

**d. Create a method to evaluate expression using order of operations:**

**● i. Handle multiplication and division first**

**● ii. Then handle addition and subtraction**

**● iii. Process from left to right for same precedence**

**● iv. Use StringBuilder to show step-by-step calculation**

**e. Create a method to handle parentheses:**

**● i. Find innermost parentheses using indexOf() and lastIndexOf()**

**● ii. Evaluate expressions inside parentheses first**

**● iii. Replace parenthetical results in main expression**

**f. Create a method to display calculation steps:**

**● i. Show original expression**

**● ii. Display each step of evaluation**

**● iii. Show final result with validation**

**g. The main function processes multiple expressions and shows detailed calculation process**

import java.util.\*;

public class TextCalculator {

// ------------------- MAIN -------------------

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("=== TEXT-BASED CALCULATOR ===");

while (true) {

System.out.print("\nEnter expression (or 'exit' to quit): ");

String expr = sc.nextLine().trim();

if (expr.equalsIgnoreCase("exit")) break;

if (!validateExpression(expr)) {

System.out.println("Invalid Expression! Try again.");

continue;

}

System.out.println("\n--- CALCULATION STEPS ---");

StringBuilder steps = new StringBuilder();

int result = evaluateExpression(expr, steps);

// Step f: Display results

System.out.println("Original: " + expr);

System.out.println(steps.toString());

System.out.println("Final Result = " + result);

}

sc.close();

}

// ------------------- STEP B: Validation -------------------

public static boolean validateExpression(String expr) {

int balance = 0;

char prev = ' ';

for (int i = 0; i < expr.length(); i++) {

char c = expr.charAt(i);

// Allowed: digits, operators, spaces, parentheses

if (!(Character.isDigit(c) || "+-\*/() ".indexOf(c) >= 0)) {

return false;

}

// Check parentheses balance

if (c == '(') balance++;

if (c == ')') balance--;

if (balance < 0) return false;

// No two operators in a row (except unary minus)

if ("+-\*/".indexOf(c) >= 0 && "+\*/".indexOf(prev) >= 0) {

return false;

}

if (c != ' ') prev = c;

}

return balance == 0;

}

// ------------------- STEP C: Parsing -------------------

private static void parseExpression(String expr, List<Integer> numbers, List<Character> ops) {

expr = expr.replaceAll(" ", "");

int i = 0;

while (i < expr.length()) {

char c = expr.charAt(i);

if (Character.isDigit(c) || (c == '-' && (i == 0 || "+-\*/(".indexOf(expr.charAt(i - 1)) >= 0))) {

// Parse multi-digit or negative number

int j = i + 1;

while (j < expr.length() && Character.isDigit(expr.charAt(j))) j++;

numbers.add(Integer.parseInt(expr.substring(i, j)));

i = j;

} else {

ops.add(c);

i++;

}

}

}

// ------------------- STEP D: Evaluation (no parentheses) -------------------

private static int evalSimple(String expr, StringBuilder steps) {

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

List<Character> ops = new ArrayList<>();

parseExpression(expr, numbers, ops);

// Handle \* and /

for (int i = 0; i < ops.size(); ) {

char op = ops.get(i);

if (op == '\*' || op == '/') {

int a = numbers.get(i);

int b = numbers.get(i + 1);

int res = (op == '\*') ? a \* b : a / b;

steps.append(a).append(" ").append(op).append(" ").append(b).append(" = ").append(res).append("\n");

numbers.set(i, res);

numbers.remove(i + 1);

ops.remove(i);

} else {

i++;

}

}

// Handle + and -

while (!ops.isEmpty()) {

int a = numbers.remove(0);

int b = numbers.remove(0);

char op = ops.remove(0);

int res = (op == '+') ? a + b : a - b;

steps.append(a).append(" ").append(op).append(" ").append(b).append(" = ").append(res).append("\n");

numbers.add(0, res);

}

return numbers.get(0);

}

// ------------------- STEP E: Parentheses -------------------

private static int evaluateExpression(String expr, StringBuilder steps) {

expr = expr.replaceAll(" ", "");

while (expr.contains("(")) {

int close = expr.indexOf(")");

int open = expr.lastIndexOf("(", close);

String inside = expr.substring(open + 1, close);

int val = evalSimple(inside, steps);

expr = expr.substring(0, open) + val + expr.substring(close + 1);

}

return evalSimple(expr, steps);

}

}

**Problem 5: Write a program to analyze and format structured data from**

**CSV-like text input using string manipulation methods**

**Hint =>**

**a. Take user input for CSV-like data (comma-separated values in multiple lines)**

**b. Create a method to parse CSV data without using split():**

**● i. Use charAt() to identify commas and newlines**

**● ii. Extract each field using substring() method**

**● iii. Handle quoted fields that may contain commas**

**● iv. Store data in a 2D array structure**

**c. Create a method to validate and clean data:**

**● i. Remove leading/trailing spaces from each field**

**● ii. Validate numeric fields using ASCII values**

**● iii. Check for missing or invalid data**

**● iv. Apply data type conversions where needed**

**d. Create a method to perform data analysis:**

**● i. Calculate column statistics (min, max, average for numeric columns)**

**● ii. Count unique values in categorical columns**

**● iii. Identify data quality issues (missing, invalid entries)**

**e. Create a method using StringBuilder to format output:**

**● i. Create aligned tabular display with fixed column widths**

**● ii. Add borders and headers for better readability**

**● iii. Format numeric values with proper decimal places**

**● iv. Highlight data quality issues**

**f. Create a method to generate data summary report:**

**● i. Show total records processed**

**● ii. Display column-wise statistics**

**● iii. List data quality findings**

**● iv. Calculate data completeness percentage**

**g. The main function processes CSV input and generates formatted output with analysis report**

import java.util.\*;

public class CSVDataAnalyzer {

// ------------------- MAIN -------------------

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("=== CSV-LIKE DATA ANALYZER ===");

System.out.println("Enter CSV-like data (end with an empty line):");

StringBuilder input = new StringBuilder();

while (true) {

String line = sc.nextLine();

if (line.trim().isEmpty()) break;

input.append(line).append("\n");

}

// Step b: Parse CSV

String[][] data = parseCSV(input.toString());

// Step c: Clean and validate

cleanAndValidate(data);

// Step d: Analyze data

String analysisReport = analyzeData(data);

// Step e: Format tabular output

String table = formatTable(data);

// Step f: Summary Report

String summary = generateSummary(data, analysisReport);

// Final Output

System.out.println("\n--- FORMATTED TABLE ---");

System.out.println(table);

System.out.println("\n--- DATA ANALYSIS REPORT ---");

System.out.println(analysisReport);

System.out.println("\n--- SUMMARY REPORT ---");

System.out.println(summary);

sc.close();

}

// ------------------- STEP B: Parse CSV -------------------

public static String[][] parseCSV(String text) {

List<String[]> rows = new ArrayList<>();

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

int i = 0;

boolean inQuotes = false;

int start = 0;

while (i < text.length()) {

char c = text.charAt(i);

if (c == '"') {

inQuotes = !inQuotes;

} else if (c == ',' && !inQuotes) {

fields.add(text.substring(start, i));

start = i + 1;

} else if (c == '\n' && !inQuotes) {

fields.add(text.substring(start, i));

rows.add(fields.toArray(new String[0]));

fields.clear();

start = i + 1;

}

i++;

}

if (start < text.length()) {

fields.add(text.substring(start));

rows.add(fields.toArray(new String[0]));

}

return rows.toArray(new String[0][]);

}

// ------------------- STEP C: Clean & Validate -------------------

public static void cleanAndValidate(String[][] data) {

for (int i = 0; i < data.length; i++) {

for (int j = 0; j < data[i].length; j++) {

if (data[i][j] == null) continue;

// Trim spaces

data[i][j] = data[i][j].trim();

// Remove quotes

if (data[i][j].startsWith("\"") && data[i][j].endsWith("\"")) {

data[i][j] = data[i][j].substring(1, data[i][j].length() - 1);

}

}

}

}

// ------------------- STEP D: Data Analysis -------------------

public static String analyzeData(String[][] data) {

StringBuilder report = new StringBuilder();

int rows = data.length - 1; // excluding header

int cols = data[0].length;

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

String header = data[0][j];

boolean isNumeric = true;

List<Double> numericValues = new ArrayList<>();

Set<String> uniqueValues = new HashSet<>();

int missing = 0;

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

String value = data[i][j];

if (value == null || value.isEmpty()) {

missing++;

continue;

}

if (isNumber(value)) {

numericValues.add(Double.parseDouble(value));

} else {

isNumeric = false;

uniqueValues.add(value);

}

}

report.append("Column: ").append(header).append("\n");

if (isNumeric && !numericValues.isEmpty()) {

double min = Collections.min(numericValues);

double max = Collections.max(numericValues);

double avg = numericValues.stream().mapToDouble(Double::doubleValue).average().orElse(0);

report.append(" Numeric Stats -> Min: ").append(min)

.append(", Max: ").append(max)

.append(", Avg: ").append(String.format("%.2f", avg)).append("\n");

} else {

report.append(" Unique Values: ").append(uniqueValues.size()).append("\n");

}

report.append(" Missing/Invalid: ").append(missing).append("\n\n");

}

return report.toString();

}

// ------------------- STEP E: Tabular Formatting -------------------

public static String formatTable(String[][] data) {

StringBuilder table = new StringBuilder();

int cols = data[0].length;

int[] colWidths = new int[cols];

// Find max column width

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

int max = 0;

for (int i = 0; i < data.length; i++) {

if (data[i][j] != null) {

max = Math.max(max, data[i][j].length());

}

}

colWidths[j] = max + 2;

}

// Build formatted table

for (int i = 0; i < data.length; i++) {

table.append("|");

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

String field = (data[i][j] == null) ? "" : data[i][j];

table.append(String.format(" %-" + colWidths[j] + "s", field)).append("|");

}

table.append("\n");

}

return table.toString();

}

// ------------------- STEP F: Summary Report -------------------

public static String generateSummary(String[][] data, String analysis) {

int totalRecords = data.length - 1; // exclude header

int totalColumns = data[0].length;

int missing = 0;

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

for (String field : data[i]) {

if (field == null || field.isEmpty()) missing++;

}

}

int totalFields = totalRecords \* totalColumns;

double completeness = 100.0 \* (totalFields - missing) / totalFields;

return "Total Records: " + totalRecords +

"\nTotal Columns: " + totalColumns +

"\nMissing Fields: " + missing +

"\nData Completeness: " + String.format("%.2f", completeness) + "%";

}

// ------------------- Helper: Check if Number -------------------

public static boolean isNumber(String s) {

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

if (!(s.charAt(i) >= '0' && s.charAt(i) <= '9') && s.charAt(i) != '.') {

return false;

}

}

return true;

}

}

**Problem 6: Write a program to create a simple text-based file organizer that**

**categorizes and renames files based on their extensions and content**

**analysis**

**Hint =>**

**a. Take user input for multiple file names with extensions**

**b. Create a method to extract file components without using split():**

**● i. Use lastIndexOf() to find the last dot for extension**

**● ii. Extract filename and extension using substring()**

**● iii. Validate file name format and characters**

**● iv. Store file information in structured format**

**c. Create a method to categorize files by extension:**

**● i. Define categories (Documents: .txt, .doc; Images: .jpg, .png; etc.)**

**● ii. Use string comparison methods to match extensions**

**● iii. Count files in each category**

**● iv. Identify unknown file types**

**d. Create a method using StringBuilder to generate new file names:**

**● i. Create naming convention based on category and date**

**● ii. Handle duplicate names by adding numbers**

**● iii. Ensure generated names follow proper file naming rules**

**● iv. Validate that new names don't contain invalid characters**

**e. Create a method to simulate content-based analysis:**

**● i. For text files, analyze content for keywords**

**● ii. Suggest subcategories based on content (Resume, Report, Code, etc.)**

**● iii. Calculate file priority based on name patterns and content**

**● iv. Use ASCII values to validate content characters**

**f. Create a method to display file organization report:**

**● i. Show original filename, category, new suggested name**

**● ii. Display category-wise file counts in tabular format**

**● iii. List files that need attention (invalid names, unknown types)**

**● iv. Show organization statistics and recommendations**

**g. Create a method to generate batch rename commands:**

**● i. Create command strings for renaming operations**

**● ii. Show before/after comparison**

**● iii. Calculate storage organization improvement**

**h. The main function processes file list and generates comprehensive organization plan with**

**statistics**

import java.util.\*;

import java.text.SimpleDateFormat;

public class FileOrganizer {

// ------------------- Data Structure -------------------

static class FileInfo {

String originalName;

String baseName;

String extension;

String category;

String subCategory;

String suggestedName;

boolean valid;

boolean unknown;

}

// ------------------- MAIN -------------------

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("=== TEXT-BASED FILE ORGANIZER ===");

System.out.println("Enter file names (end with empty line):");

List<FileInfo> files = new ArrayList<>();

while (true) {

String name = sc.nextLine().trim();

if (name.isEmpty()) break;

files.add(extractFileComponents(name));

}

// Categorize

categorizeFiles(files);

// Generate new names

generateNewNames(files);

// Simulate content analysis (only text files)

simulateContentAnalysis(files);

// Show Report

displayReport(files);

// Batch rename commands

generateRenameCommands(files);

sc.close();

}

// ------------------- STEP B: Extract file components -------------------

public static FileInfo extractFileComponents(String name) {

FileInfo file = new FileInfo();

file.originalName = name;

int dotIndex = name.lastIndexOf('.');

if (dotIndex == -1 || dotIndex == name.length() - 1) {

file.baseName = name;

file.extension = "";

file.valid = false;

file.unknown = true;

return file;

}

file.baseName = name.substring(0, dotIndex);

file.extension = name.substring(dotIndex + 1).toLowerCase();

file.valid = isValidName(file.baseName);

return file;

}

public static boolean isValidName(String base) {

for (int i = 0; i < base.length(); i++) {

char c = base.charAt(i);

if (!(Character.isLetterOrDigit(c) || c == '\_' || c == '-')) {

return false;

}

}

return true;

}

// ------------------- STEP C: Categorize -------------------

public static void categorizeFiles(List<FileInfo> files) {

for (FileInfo f : files) {

switch (f.extension) {

case "txt": case "doc": case "pdf":

f.category = "Documents"; break;

case "jpg": case "png": case "gif":

f.category = "Images"; break;

case "mp4": case "avi": case "mkv":

f.category = "Videos"; break;

case "mp3": case "wav":

f.category = "Audio"; break;

case "java": case "py": case "cpp":

f.category = "Code"; break;

default:

f.category = "Unknown";

f.unknown = true;

}

}

}

// ------------------- STEP D: Generate New Names -------------------

public static void generateNewNames(List<FileInfo> files) {

Map<String, Integer> nameCount = new HashMap<>();

String date = new SimpleDateFormat("yyyyMMdd").format(new Date());

for (FileInfo f : files) {

if (!f.valid || f.unknown) {

f.suggestedName = "INVALID\_" + f.originalName;

continue;

}

String newBase = f.category + "\_" + date;

int count = nameCount.getOrDefault(newBase, 0) + 1;

nameCount.put(newBase, count);

f.suggestedName = newBase + "\_" + count + "." + f.extension;

}

}

// ------------------- STEP E: Content Analysis (simulation) -------------------

public static void simulateContentAnalysis(List<FileInfo> files) {

for (FileInfo f : files) {

if (f.extension.equals("txt")) {

// Fake rules: detect keywords from filename itself

if (f.baseName.toLowerCase().contains("resume")) {

f.subCategory = "Resume";

} else if (f.baseName.toLowerCase().contains("report")) {

f.subCategory = "Report";

} else if (f.baseName.toLowerCase().contains("code")) {

f.subCategory = "Code Snippet";

} else {

f.subCategory = "General Text";

}

} else {

f.subCategory = "-";

}

}

}

// ------------------- STEP F: Report -------------------

public static void displayReport(List<FileInfo> files) {

StringBuilder sb = new StringBuilder();

sb.append("\n--- FILE ORGANIZATION REPORT ---\n");

sb.append(String.format("%-25s %-12s %-15s %-25s %-15s\n",

"Original Name", "Category", "SubCategory", "Suggested Name", "Status"));

sb.append("---------------------------------------------------------------------------------------------\n");

Map<String, Integer> categoryCount = new HashMap<>();

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

for (FileInfo f : files) {

sb.append(String.format("%-25s %-12s %-15s %-25s %-15s\n",

f.originalName,

f.category,

f.subCategory,

f.suggestedName,

(f.valid ? "OK" : "Invalid")));

categoryCount.put(f.category, categoryCount.getOrDefault(f.category, 0) + 1);

if (!f.valid || f.unknown) {

attention.add(f.originalName);

}

}

sb.append("\n--- CATEGORY-WISE COUNTS ---\n");

for (Map.Entry<String, Integer> e : categoryCount.entrySet()) {

sb.append(e.getKey()).append(": ").append(e.getValue()).append("\n");

}

sb.append("\n--- FILES NEEDING ATTENTION ---\n");

if (attention.isEmpty()) {

sb.append("None\n");

} else {

for (String a : attention) sb.append(a).append("\n");

}

System.out.println(sb.toString());

}

// ------------------- STEP G: Batch Rename Commands -------------------

public static void generateRenameCommands(List<FileInfo> files) {

StringBuilder sb = new StringBuilder();

sb.append("\n--- BATCH RENAME COMMANDS ---\n");

for (FileInfo f : files) {

if (f.valid && !f.unknown) {

sb.append("rename \"").append(f.originalName).append("\" \"")

.append(f.suggestedName).append("\"\n");

}

}

System.out.println(sb.toString());

}

}