# Message Compression

To reduce bandwidth usage during message transmission, a company wants to design a system that compresses strings by summarizing consecutive repeated characters. The system works as follows:

* Each character is checked for consecutive repetitions.
* If a character appears consecutively more than once, it is followed by the count of occurrences in the compressed string.
* If a character appears only once, it is added directly to the compressed string.

For example:

* Input: "aabbccc"
* Compressed Output: "a2b2c3"

If the compressed version is longer than or equal to the original string, the original string is returned.

Input Format:

* A single string s consisting of lowercase English letters (1 ≤ |s| ≤ 10^5).

Output:

* A compressed version of the string based on the rules above, or the original string if compression does not reduce the length.

Example:

**Input:**  
"aaabbc"

**Output:**  
"a3b2c"

Java Solution:

public class MessageCompression {

public static String compressString(String s) {

StringBuilder compressed = new StringBuilder();

int count = 1;

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

if (s.charAt(i) == s.charAt(i - 1)) {

count++;

} else {

compressed.append(s.charAt(i - 1));

if (count > 1) {

compressed.append(count);

}

count = 1;

}

}

// Append the last character and its count

compressed.append(s.charAt(s.length() - 1));

if (count > 1) {

compressed.append(count);

}

// Return the compressed string if it's shorter, otherwise return the original string

return compressed.length() < s.length() ? compressed.toString() : s;

}

public static void main(String[] args) {

// Example Test Cases

String input1 = "aaabbc";

System.out.println(compressString(input1)); // Output: "a3b2c"

String input2 = "abc";

System.out.println(compressString(input2)); // Output: "abc"

}

}

# Backspace File Path Comparison

You are given two file paths, path1 and path2, consisting of directory names separated by / and backspace operators represented by ... A backspace operator removes the previous directory from the path. The paths may also contain redundant slashes (//).

Your task is to compare the final normalized paths to determine if they are the same.

Write a function comparePaths that takes two strings, path1 and path2, and returns:

* 1 if the normalized paths are the same.
* 0 otherwise.

Input Format

* The first line contains the string path1.
* The second line contains the string path2.

Output Format

Return an integer (1 or 0) indicating whether the paths are equivalent.

Constraints

* 1 ≤ length of path1, path2 ≤ 10^5
* Paths consist of lowercase letters, /, and .. only.

Example Input

path1: /home/user/../docs/

path2: /home/docs/

Example Output

1

Explanation

Both paths resolve to /home/docs, so the output is 1.

Solution (Java)

import java.util.Stack;

public class BackspaceFilePathComparison {

public static int comparePaths(String path1, String path2) {

String normalizedPath1 = normalizePath(path1);

String normalizedPath2 = normalizePath(path2);

return normalizedPath1.equals(normalizedPath2) ? 1 : 0;

}

private static String normalizePath(String path) {

Stack<String> stack = new Stack<>();

String[] parts = path.split("/");

for (String part : parts) {

if (part.equals("..")) {

if (!stack.isEmpty()) {

stack.pop();

}

} else if (!part.isEmpty() && !part.equals(".")) {

stack.push(part);

}

}

return "/" + String.join("/", stack);

}

public static void main(String[] args) {

String path1 = "/home/user/../docs/";

String path2 = "/home/docs/";

System.out.println(comparePaths(path1, path2)); // Output: 1

}

}

# Problem Statement

You are given a string s consisting of lowercase English letters. A string is considered *harmonious* if no two adjacent characters are the same, and no two adjacent characters are consecutive in the alphabet.

You can replace any character in the string s with any other lowercase English letter. Your task is to find the minimum number of replacements required to make the string harmonious.

Input:

* The first line contains a string s of length 2≤∣s∣≤1052 \leq |s| \leq 10^52≤∣s∣≤105.

Output:

* Print an integer representing the minimum number of operations required to transform s into a harmonious string.

Example:

**Input**:

bccb

**Output**:

2

**Explanation**:

* The original string contains violations: "cc" are the same, and "cb" are consecutive in the alphabet.
* By changing c to d and the last b to d, the string becomes "bdbd", which is harmonious.

Java Solution:

public class HarmoniousString {

public static int getMinimumOperationCount(String s) {

int operations = 0;

char[] chars = s.toCharArray();

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

// Check if adjacent characters are the same or consecutive

if (chars[i] == chars[i - 1] || Math.abs(chars[i] - chars[i - 1]) == 1) {

operations++; // Increment operation count

chars[i] = getReplacement(chars[i - 1]); // Replace with a non-conflicting character

}

}

return operations;

}

// Helper function to get a replacement character

private static char getReplacement(char previous) {

return (previous != 'a') ? 'a' : 'z'; // Replace with 'a' or 'z' to avoid conflicts

}

public static void main(String[] args) {

String s = "bccb";

System.out.println(getMinimumOperationCount(s)); // Output: 2

}

}

# Text Encoding

To optimize the storage of text messages, a technique called **Text Encoding** is introduced. This algorithm minimizes redundant consecutive character occurrences by replacing them with a concise representation.

Given a string, your task is to encode it using the following rules:

1. If a character appears consecutively more than once, replace the consecutive sequence with the character followed by its count.
2. If a character appears only once, keep it as is.

**Example**

For the string "aaabbcc", the encoded form is "a3b2c2".  
For the string "xyz", the encoded form is "xyz".

**Output**

* Returns the **encoded form** of the input string.

**Sample Input**

**Input 1:**

aaabbcc

**Output 1:**

a3b2c2

**Input 2:**

xyz

**Output 2:**

xyz

**Java Solution**

public class TextEncoder {

public static String encodeText(String message) {

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

return "";

}

StringBuilder encoded = new StringBuilder();

int count = 1;

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

if (message.charAt(i) == message.charAt(i - 1)) {

count++;

} else {

encoded.append(message.charAt(i - 1));

if (count > 1) {

encoded.append(count);

}

count = 1;

}

}

// Append the last character and its count

encoded.append(message.charAt(message.length() - 1));

if (count > 1) {

encoded.append(count);

}

return encoded.toString();

}

public static void main(String[] args) {

String input1 = "aaabbcc";

String input2 = "xyz";

System.out.println(encodeText(input1)); // Output: "a3b2c2"

System.out.println(encodeText(input2)); // Output: "xyz"

}

}

# Count balanced subarrays

Given an array componentValue of size n, a subarray is defined as "balanced" if:

1. It contains an odd number of elements with odd values.
2. It contains an even number of elements with even values.

A subarray is considered **balanced** if:

1. It contains an **odd number of odd-valued elements**.
2. It contains an **even number of even-valued elements**.

Write a function countBalancedSubarrays that returns the count of balanced subarrays.

**Solution in Java:**

import java.util.\*;

public class BalancedSubarray {

public static long countBalancedSubarrays(int[] componentValue) {

int n = componentValue.length;

long balancedCount = 0;

// Prefix counters for odd and even values

int oddCount = 0, evenCount = 0;

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

// To handle the initial case where a balanced subarray starts from index 0

prefixMap.put("0#0", 1);

for (int value : componentValue) {

// Increment odd/even counts

if (value % 2 == 0) {

evenCount++;

} else {

oddCount++;

}

// Create a unique key for the difference of counts

String key = (oddCount % 2) + "#" + (evenCount % 2);

// If the key already exists, add the count to balancedCount

balancedCount += prefixMap.getOrDefault(key, 0);

// Update the prefixMap

prefixMap.put(key, prefixMap.getOrDefault(key, 0) + 1);

}

return balancedCount;

}

public static void main(String[] args) {

int[] componentValue1 = {1, 2, 3}; // Sample Input 0

System.out.println(countBalancedSubarrays(componentValue1)); // Output: 2

int[] componentValue2 = {1, 4, 3, 2}; // Sample Input 1

System.out.println(countBalancedSubarrays(componentValue2)); // Output: 3

}

}

# Distinct triplets

You are given an array of integers arr with n elements and an integer d. Your task is to find the number of distinct triplets (i, j, k) such that:

1. i<j<ki < j < ki<j<k (indices must be distinct and in increasing order).
2. The sum of the triplet arr[i]+arr[j]+arr[k] is divisible by ddd.

**Example 1**

**Input:**

n = 4

arr = [2, 3, 1, 6]

d = 3

**Output:**

2

**Explanation:**

The triplets that satisfy the condition are:

1. Indices (1, 2, 3): 2+3+1=62 + 3 + 1 = 62+3+1=6, which is divisible by 333.
2. Indices (1, 3, 4): 2+1+6=92 + 1 + 6 = 92+1+6=9, which is divisible by 333.

Total number of valid triplets: **2**.

**Example 2**

**Input:**

n = 5

arr = [5, 3, 5, 5, 8]

d = 11

**Output:**

0

**Explanation:**

There are no triplets where the sum of the elements is divisible by 111111. Therefore, the output is **0**.

**Java Solution**

import java.util.Scanner;

public class CountingTriplets {

// Function to count the number of valid triplets

public static int getTripletCount(int[] arr, int d) {

int n = arr.length;

int count = 0;

// Iterate through all possible triplets (i, j, k)

for (int i = 0; i < n - 2; i++) {

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

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

// Check if the sum of the triplet is divisible by d

int sum = arr[i] + arr[j] + arr[k];

if (sum % d == 0) {

count++;

}

}

}

}

return count;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Input the size of the array

int n = scanner.nextInt();

int[] arr = new int[n];

// Input array elements

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

arr[i] = scanner.nextInt();

}

// Input the divisor

int d = scanner.nextInt();

// Calculate the number of triplets

int result = getTripletCount(arr, d);

// Output the result

System.out.println(result);

scanner.close();

}

}

**Data Reorganization:**

It focuses on minimizing the smallest value in a dataset after performing a series of specific operations

Given an array data[] of size n, you can perform up to maxOperations operations. In each operation, you:

1. Select two indices (i, j) such that 0≤i<j<len(data)0 \leq i < j < len(data)0≤i<j<len(data).
2. Compute the absolute difference between the elements at these indices: ∣data[i]−data[j]∣|data[i] - data[j]|∣data[i]−data[j]∣.
3. Append the computed value to the end of the array, increasing its length by 1.

The goal is to determine the smallest possible value of the minimum element in the array after exactly maxOperations operations.

import java.util.PriorityQueue;

import java.util.Collections;

public class DataReorganization {

public static int getMinimumValue(int[] data, int maxOperations) {

// Create a max-heap to store elements in descending order

PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

for (int num : data) {

maxHeap.offer(num);

}

// Perform operations

while (maxOperations > 0) {

// Remove the largest element from the heap

int largest = maxHeap.poll();

// If the largest element is already 1, we can't reduce it further

if (largest == 1) {

maxHeap.offer(largest);

break;

}

// Split the largest element into two parts

int part1 = largest / 2;

int part2 = largest - part1;

// Add the two parts back to the heap

maxHeap.offer(part1);

maxHeap.offer(part2);

// Decrement the operation count

maxOperations--;

}

// The smallest value in the heap is the result

int minValue = Integer.MAX\_VALUE;

while (!maxHeap.isEmpty()) {

minValue = Math.min(minValue, maxHeap.poll());

}

return minValue;

}

public static void main(String[] args) {

// Example Input

int[] data = {5, 18, 3, 12, 11};

int maxOperations = 2;

// Call the function and print the result

int result = getMinimumValue(data, maxOperations);

System.out.println("The smallest possible value of the minimum element is: " + result);

}

}

# Passing baton

Friends pass a baton in a specific sequence, and based on a given time, determine which friend is passing and which is receiving the baton.

**Problem Description:**

* You have n friends numbered from 1 to n standing in a line.
* The baton starts with Friend 1 at time t = 0.
* Every second, the baton is passed to the next friend in line.
* Upon reaching the last friend (n), the direction of passing reverses, and the baton is passed back towards Friend 1.
* This back-and-forth passing continues indefinitely.

**Objective:**

Given the total number of friends n and a specific time t, determine:

1. The friend who is currently passing the baton.
2. The friend who is currently receiving the baton.

**Approach:**

1. **Calculate the Cycle Duration:**
   * The baton moves from Friend 1 to Friend n in (n - 1) seconds.
   * It then returns from Friend n back to Friend 1 in another (n - 1) seconds.
   * Thus, a complete cycle (forward and backward) takes 2 \* (n - 1) seconds.
2. **Determine the Effective Time:**
   * Since the passing pattern repeats every 2 \* (n - 1) seconds, compute the effective time within the first cycle: effectiveTime = t % (2 \* (n - 1)).
3. **Identify the Passing Direction and Friends Involved:**
   * **Forward Direction (0 to n-1 seconds):**
     + If effectiveTime is between 0 and n - 1 (inclusive), the baton is moving forward.
     + The friend passing the baton is effectiveTime + 1.
     + The friend receiving the baton is effectiveTime + 2.
   * Backward Direction (n to 2(n-1) - 1 seconds):\*
     + If effectiveTime is between n and 2 \* (n - 1) - 1 (inclusive), the baton is moving backward.
     + The friend passing the baton is 2 \* n - 1 - effectiveTime.
     + The friend receiving the baton is 2 \* n - 2 - effectiveTime.

**Example:**

Let's consider an example to illustrate this:

* **Number of Friends (n):** 5
* **Time (t):** 7 seconds

**Result:** At time t = 7 seconds, Friend 2 is passing the baton to Friend 1.

**Solution**

public class BatonPassing {

public static void findBatonPasserReceiver(int n, int t) {

int cycleDuration = 2 \* (n - 1);

int effectiveTime = t % cycleDuration;

int passer, receiver;

if (effectiveTime <= n - 1) {

// Forward direction

passer = effectiveTime + 1;

receiver = passer + 1;

} else {

// Backward direction

passer = 2 \* n - 1 - effectiveTime;

receiver = passer - 1;

}

System.out.println("At time " + t + " seconds:");

System.out.println("Friend " + passer + " is passing the baton to Friend " + receiver + ".");

}

public static void main(String[] args) {

int n = 5; // Number of friends

int t = 7; // Time in seconds

findBatonPasserReceiver(n, t);

}

}

**Output:**

At time 7 seconds:

Friend 2 is passing the baton to Friend 1.

### Word Count Tool

The task is to determine the number of valid words in a given string s. A word is considered valid if it meets the following criteria:

1. It contains at least 3 characters.
2. It includes at least one vowel (a, e, i, o, u).
3. It includes at least one consonant.
4. It only consists of alphanumeric characters (0-9, A-Z, a-z).

Invalid words include those that:

* Are shorter than 3 characters.
* Do not contain a vowel or a consonant.
* Contain invalid characters (e.g., $, #).

### Example

#### Input:

This is Form16 submi$$ion date

#### Output:

3

#### Explanation:

* Valid words: This, Form16, and date.
* Invalid words:
  + is (less than 3 characters),
  + submi$$ion (contains invalid characters $).

#### Another Input:

Bob wins the game

#### Output:

4

#### Explanation:

All words (Bob, wins, the, game) are valid.

### Java Solution

Here is the Java solution to implement the functionality:

import java.util.\*;

public class WordCountTool {

public static int countValidWords(String s) {

// Split the input string into words using whitespace as the delimiter

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

int validWordCount = 0;

// Loop through each word to check if it is valid

for (String word : words) {

if (isValidWord(word)) {

validWordCount++;

}

}

return validWordCount;

}

private static boolean isValidWord(String word) {

// A valid word must contain at least 3 characters

if (word.length() < 3) return false;

boolean hasVowel = false;

boolean hasConsonant = false;

// Check each character of the word

for (char c : word.toCharArray()) {

// If the character is not alphanumeric, the word is invalid

if (!Character.isLetterOrDigit(c)) return false;

// Check for vowels

if ("aeiouAEIOU".indexOf(c) != -1) {

hasVowel = true;

}

// Check for consonants

else if (Character.isLetter(c)) {

hasConsonant = true;

}

}

// A valid word must have at least one vowel and one consonant

return hasVowel && hasConsonant;

}

public static void main(String[] args) {

// Example input

String input = "This is Form16 submi$$ion date";

int result = countValidWords(input);

System.out.println(result); // Output: 3

}

}

# Movie Library

You are tasked to create a **Movie Library** application that:

1. Adds movies with their title, director, and release year.
2. Searches for movies by title or director.
3. Removes a movie by title.
4. Returns a list of all movies.
5. Returns the total count of movies in the library.

The input includes:

1. The number of movies to add.
2. Each movie's details (title, director, year).
3. A search query (title or director).
4. A title of a movie to delete.

The program should process these inputs, perform the operations, and output the results as described in the sample cases.

### Solution Design

#### Classes and Interfaces

1. **IFilm**: Interface for the Film class.
2. **Film**: Represents a movie with properties like title, director, and year.
3. **IFilmLibrary**: Interface for the FilmLibrary class.
4. **FilmLibrary**: Manages a collection of Film objects and provides methods for adding, searching, and removing films.

### Sample Input and Output

**Input:**

4

Film-1 Director-2 2004

Film-2 Director-1 2018

Film-3 Director-1 2001

Film-4 Director-3 2017

Director-1

Film-1

**Output:**

Total Film Count: 4

Search Results for Director-1:

Film-2 (Director-1, 2018)

Film-3 (Director-1, 2001)

Removed Film: Film-1

All Films:

Film-2 (Director-1, 2018)

Film-3 (Director-1, 2001)

Film-4 (Director-3, 2017)

### Java solution

Here is the full Java solution:

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

// Interface for Film

interface IFilm {

String getTitle();

String getDirector();

int getYear();

}

// Implementation of Film class

class Film implements IFilm {

private String title;

private String director;

private int year;

// Constructor

public Film(String title, String director, int year) {

this.title = title;

this.director = director;

this.year = year;

}

@Override

public String getTitle() {

return title;

}

@Override

public String getDirector() {

return director;

}

@Override

public int getYear() {

return year;

}

@Override

public String toString() {

return title + " (" + director + ", " + year + ")";

}

}

// Interface for FilmLibrary

interface IFilmLibrary {

void addFilm(IFilm film);

void removeFilm(String title);

List<IFilm> getFilms();

List<IFilm> searchFilms(String query);

int getTotalFilmCount();

}

// Implementation of FilmLibrary class

class FilmLibrary implements IFilmLibrary {

private List<IFilm> films;

// Constructor

public FilmLibrary() {

this.films = new ArrayList<>();

}

@Override

public void addFilm(IFilm film) {

films.add(film);

}

@Override

public void removeFilm(String title) {

films.removeIf(film -> film.getTitle().equalsIgnoreCase(title));

}

@Override

public List<IFilm> getFilms() {

return new ArrayList<>(films);

}

@Override

public List<IFilm> searchFilms(String query) {

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

for (IFilm film : films) {

if (film.getTitle().equalsIgnoreCase(query) || film.getDirector().equalsIgnoreCase(query)) {

result.add(film);

}

}

return result;

}

@Override

public int getTotalFilmCount() {

return films.size();

}

}

// Main class to execute the program

public class MovieLibraryApp {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

FilmLibrary filmLibrary = new FilmLibrary();

// Input the number of films

int n = Integer.parseInt(scanner.nextLine());

// Input film details

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

String[] details = scanner.nextLine().split(" ");

String title = details[0];

String director = details[1];

int year = Integer.parseInt(details[2]);

filmLibrary.addFilm(new Film(title, director, year));

}

// Input the search query

String searchQuery = scanner.nextLine();

// Input the title to delete

String deleteTitle = scanner.nextLine();

// Output the total film count

System.out.println("Total Film Count: " + filmLibrary.getTotalFilmCount());

// Search for films and display results

List<IFilm> searchResults = filmLibrary.searchFilms(searchQuery);

System.out.println("Search Results for " + searchQuery + ":");

for (IFilm film : searchResults) {

System.out.println(film);

}

// Remove a film and display confirmation

filmLibrary.removeFilm(deleteTitle);

System.out.println("Removed Film: " + deleteTitle);

// Display all films

System.out.println("All Films:");

for (IFilm film : filmLibrary.getFilms()) {

System.out.println(film);

}

scanner.close();

}

}

# Problem Edge class:

You are tasked with implementing a class Edge to represent an edge in a directed graph. The program should be able to:

1. **Create an edge** between two nodes using a factory method.
2. **Compare two edges** to determine if they are equal (based on fromNode and toNode values).
3. **Generate a hash code** for an edge to support storing edges in hash tables.
4. **Display an edge** in the format: From node: <fromNode> To node: <toNode>.
5. **Process multiple queries** and remove duplicate edges in the graph.

### Sample Input:

4

1 3

1 2

2 1

1 3

### Sample Output:

3

From node: 1 To node: 2

From node: 1 To node: 3

From node: 2 To node: 1

### Implementation Steps:

1. Define the Edge class with the required properties and methods.
2. Override methods:
   * equals() for equality comparison.
   * hashCode() to ensure correct hashing.
   * toString() for edge representation.
3. Use a HashSet to store edges without duplication.
4. Parse the input, add edges to the set, and display the unique edges.

### Java Solution:

import java.util.\*;

class Edge implements Comparable<Edge> {

private int fromNode;

private int toNode;

// Factory method to create an edge

public static Edge createEdge(int fromNode, int toNode) {

return new Edge(fromNode, toNode);

}

// Constructor

public Edge(int fromNode, int toNode) {

this.fromNode = fromNode;

this.toNode = toNode;

}

// Override equals() for edge comparison

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Edge edge = (Edge) obj;

return fromNode == edge.fromNode && toNode == edge.toNode;

}

// Override hashCode() for hashing

@Override

public int hashCode() {

return Objects.hash(fromNode, toNode);

}

// Override toString() for custom edge representation

@Override

public String toString() {

return "From node: " + fromNode + " To node: " + toNode;

}

// Implement compareTo for sorting edges

@Override

public int compareTo(Edge other) {

if (this.fromNode != other.fromNode) {

return Integer.compare(this.fromNode, other.fromNode);

}

return Integer.compare(this.toNode, other.toNode);

}

}

public class GraphEdges {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int numberOfQueries = scanner.nextInt();

Set<Edge> edgeSet = new TreeSet<>();

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

int fromNode = scanner.nextInt();

int toNode = scanner.nextInt();

Edge edge = Edge.createEdge(fromNode, toNode);

edgeSet.add(edge); // Add edge to the set (duplicates automatically handled)

}

// Output results

System.out.println(edgeSet.size()); // Total unique edges

for (Edge edge : edgeSet) {

System.out.println(edge); // Display each edge

}

scanner.close();

}

}

# Grocery Receipt Calculation

You are developing a billing software for an e-commerce platform. The application takes input for items with their respective prices, discounts (if applicable), and the quantity purchased. Your task is to calculate the total price for each item, considering the applied discount, and generate a final invoice. The invoice should be sorted by the product name in ascending order.

**Requirements:**

1. Each item has a name, price per unit, and a percentage discount.
2. Discounts apply only to specific items, and their percentage is provided separately.
3. The shopping list includes items with quantities purchased.
4. Calculate the total price for each item as (price per unit \* quantity) - discount.
5. Display the final invoice with the item name, price per unit, and the total price after applying the discount.

**Input:**

* First line: Number of items, n.
* Next n lines: Each line contains the item name, price per unit.
* Next line: Number of items with discounts, m.
* Next m lines: Each line contains the item name, percentage discount.
* Next line: Number of items purchased, k.
* Next k lines: Each line contains the item name, quantity purchased.

**Output:**

The output should display the invoice with item name, price per unit, and the total price after discount.

### Example Input:

mathematica

CopyEdit

3

Apple 34

Banana 14

Orange 4

1

Orange 10

2

Apple 2

Apple 5

### Example Output:

CopyEdit

Apple 34.0 238.0

### Java Solution

import java.util.\*;

class GroceryReceipt {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// Read number of items and their prices

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

Map<String, Double> priceMap = new HashMap<>();

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

String[] itemData = sc.nextLine().split(" ");

priceMap.put(itemData[0], Double.parseDouble(itemData[1]));

}

// Read discounts

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

Map<String, Double> discountMap = new HashMap<>();

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

String[] discountData = sc.nextLine().split(" ");

discountMap.put(discountData[0], Double.parseDouble(discountData[1]));

}

// Read purchased items and quantities

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

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

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

String[] purchaseData = sc.nextLine().split(" ");

purchaseMap.put(purchaseData[0], purchaseMap.getOrDefault(purchaseData[0], 0) + Integer.parseInt(purchaseData[1]));

}

// Calculate total price and generate invoice

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

for (Map.Entry<String, Integer> entry : purchaseMap.entrySet()) {

String item = entry.getKey();

int quantity = entry.getValue();

double pricePerUnit = priceMap.getOrDefault(item, 0.0);

double discount = discountMap.getOrDefault(item, 0.0);

double totalPrice = pricePerUnit \* quantity;

totalPrice -= (totalPrice \* discount / 100);

result.add(String.format("%s %.1f %.1f", item, pricePerUnit, totalPrice));

}

// Sort the result by item name

result.sort(Comparator.naturalOrder());

for (String line : result) {

System.out.println(line);

}

sc.close();

}

}