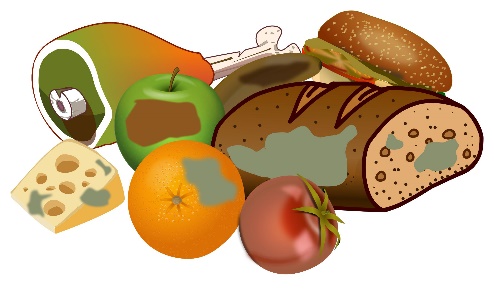
# Food Finder



You will be given **two sequences of characters, representing vowels and consonants**. Your task is to start checking if the following words could be created:

* **"pear"**
* **"flour"**
* **"pork"**
* **"olive"**

Start by taking the **first character** of the **vowels collection** and the **last character** from the **consonants collection.** Then check if these letters are present in one or more of the given words. If these letters are present, you should store the information. Then process to the next couple of letters until there are no more **consonant** letters left.

A **letter (vowels or consonants) could participate in more than one word, for example:**

The letter **'**o**'** is present in **"flour", "pork",** and **"olive".**

The letter **'**l**'** is present in **"flour",** and **"olive".**

**Keep in mind that:**

* A **vowel** letter is always returned to the collection, whether used or not.
* A **consonant** letter is always removed from the collection, whether used or not.

As a result, you should **check how many of the given words were** found and print:

**"Words found: {numberOfWordsFound}**

**{wordOne}**

**{wordTwo}**

**…"**

**Look at the provided examples for a better understanding of the problem.**

### Input

* On the **first line**, you will receive characters representing the **vowels**, **separated** by a single space (**" "**).
* On the **second line**, you will receive characters representing the **consonants**, **separated** by a single space (**" "**)**.**

### Output

* As a result, print on the first line how many words have been found and on the next N lines, every word:

**"Words found: {numberOfWordsFound}**

**{wordOne}**

**{wordTwo}**

**…"**

**Print words in the same order as in the problem's description.**

### Constraints

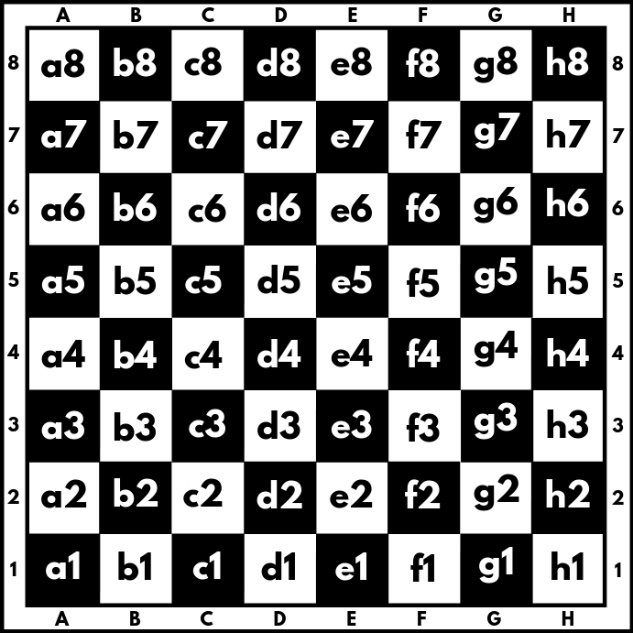
* All letters will be lowercase.
* All letters in the given words are unique.
* There will be no case where no word is matched.
* The letter **'**y**' will be always vowel.**

### Examples

|  |  |
| --- | --- |
| ****Input**** | ****Output**** |
| **e a u o**  **p r l x f** | **Words found: 2**  **pear**  **flour** |
| ****Comment**** | |
| We start by taking the first two letters **'e' and 'f'. We have concurrence, 'e' is found into "olive" and "pear", 'f' is found into "flour".**  **We add 'e' to the end of the vowels collection (a u o e) and remove 'f' out of the consonant collection (p r l x).**  Onto the next iteration we continue **'a' and 'x', where 'a' is found in "pear", and 'x' is not located into any word, 'a' is added at the end of the vowels collection (u o e a) and remove 'x' (p r l).**  **Next, we have 'u' and 'l', where both letters are found in "flour" and 'l' is found in "**olive**", we add 'u' back in the collection (o e a u) and remove 'l' (p r).**  **Next, we have 'o' and 'r', 'o' is found in "**flour**"**, **"**pork**"**, and **"**olive**"**, and **'r' is found in "**pear**"**, **"**flour**"**, **"**pork**"**, we add **'o' back in the collection (e a u o) and remove 'r' (p).**  **In this iteration, one word was found: "**flour**"**.  In the last iteration, we take **'e' and 'p'. We have already found all 'e' letters so far, so we only search for 'p', which is found in "**pear**"** and **"**pork**"**.  As a result, we found two words **"**pear**"** and **"**flour**",** so we print the corresponding output.  Final result: **"**pear**"**, **"**flour**"**, **"**pork, **"**olive**"**. | |
|  | |
| **a o y**  **b h p j r n k** | **Words found: 1**  **pork** |

# 02. Pawn Wars

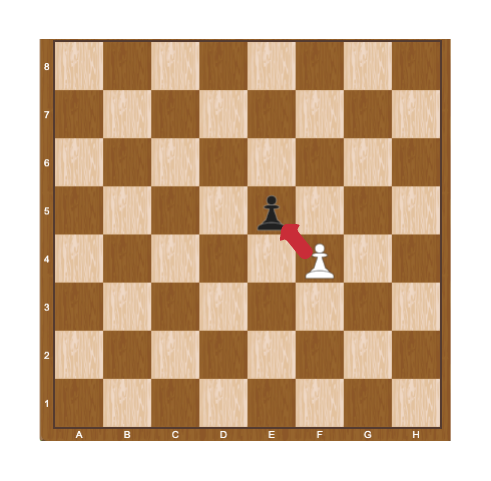
Before start solving this problem get familiar with a chessboard:



A chessboard has 8 rows and 8 columns. Rows also called ranks, are marked from number 1 to 8, and columns are marked from **a** to **h**. We have a total of 64 squares, each square is represented by a combination of letters and a number (a1, b1, c1, etc.). In this problem colors of the board will be ignored.

We will play the game with two pawns **white (w)** and **black (b)**, where they can:

* Only move forward:
  + - White (**w**) moves from the 1st rank to the 8th rank direction.
    - Black (**b**) moves from 8th rank to the 1st rank direction.
* Can move only 1 square at a time.
* Can capture another pawn only diagonally:



When a pawn reaches the **last rank**, for **white this is the 8th** rank, and **for black, this is the 1st** rank, can be **promoted** to a queen.

Two pawns (**w** and **b**) will be placed on two random squares of the bord. The **first** move is always made by the **white pawn** (**w**), then black moves (b), then white (w) again, and so on. When **a pawn marches forward**, the **previous position** is marked by "-" (dash).

Some rules will be applied when moving paws:

* If the two pawns interact diagonally, the player, in turn, must capture the opponent’s pawn. When a pawn capture another pawn the game is over and "**Game over! {White/Black} capture on {coordinates}.**" is printed to the console.

**Example:**

White pawn is on the move and captures black in "**e5**". We print "**Game over! White capture on e5.**"

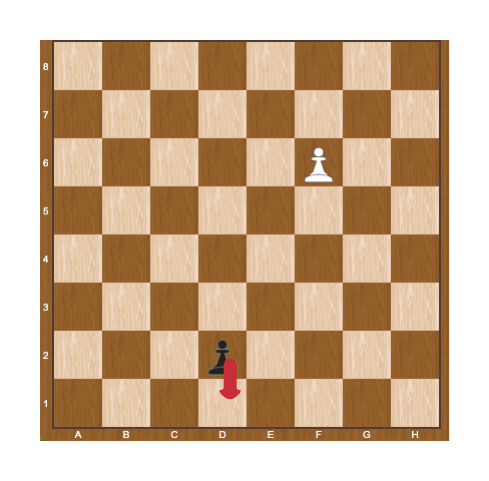
A picture containing text, checker

Description automatically generated

* If no capture is possible, the pawns keep on moving until one of them reaches the last rank. When one of the pawns reaches the last rank we print: "**Game over! {White/Black} pawn is promoted to a queen at {coordinates}.**"

**Example:**

It is black**'**s turn and the pawn reaches the d1 square, we print "**Game over! Black pawn is promoted to a queen at d1.**"



## Constraints

* The input will be always valid.
* The matrix will always be 8x8.
* There will be no case where two pawns are placed on the same square.
* There will be no case where two pawns are placed on the same column.
* There will be no case where black/white will be placed on the last rank.

## Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| ------b-  --------  --------  --------  --------  -w------  --------  -------- | Game over! White pawn is promoted to a queen at b8. | We start by pushing the white pawn to b4, next, we push the black pawn to g7:  --------  ------b-  --------  --------  -w------  --------  --------  --------  Then white play b5, black play g6:  --------  --------  ------b-  -w------  --------  --------  --------  --------  …  Capturing is not possible here, so after a few more moves, the white pawn is promoted to a queen on b8. |
| --------  --------  --------  --------  --------  b-------  -w------  -------- | Game over! White capture on a3. | Here white captures black on a3 in the first move:  --------  --------  --------  --------  --------  w-------  --------  -------- |

# Parrots

# Flying parrot PNG images, free download

## Preparation

Download the skeleton provided in Judge. **Do not** change the **packages**.

**Pay attention to name the package (parrots), all the classes, their fields and methods the same way they are presented in the following document. It is also important to keep the project structure as described.**

## Problem description

Your task is to create a repository which stores departments by creating the classes described below.

# Parrot

First, write a Java class **Parrot** with the following fields:

* **name: String**
* **species: String**
* **available: boolean - true by default**

The class **constructor** should receive **(name, species)**.

The class should also have the following methods:

* **getName()**
* **getSpecies()**
* **isAvailable()**
* **setAvailable()**
* Override the **toString()** method in the format:

**"Parrot ({species}): {name}"**

# Cage

**Next**, write a Java class **Cage** that has **data** (a collection which stores the entity **Parrot**). All entities inside the repository have the **same fields**. Also, the **Cage** class should have those **fields**:

* **name: String**
* **capacity: int**
* **data**: **List<Parrot>** that holds added parrots

The class **constructor** should receive **(name**, **capacity)**, also it should initialize the **data** with a new instance of the collection.

Implement the following features:

* **getName()**
* **getCapacity()**
* add(Parrot parrot) method - **adds** an **entity** to the data **if** **there** **is** **room** for it
* remove(String name) method - removes a parrot by **given name,** if such **exists**, and **returns boolean**
* sellParrot(String name) method - **sell** (**set** its available **property** to **false** without removing it from the collection) the **first parrot** with the **given name**, also **return** the **parrot**
* sellParrotBySpecies(String species) method - sells and returns **all parrots** from that **species as a List**
* count() - **returns** the **number** of parrots
* **report()** - **returns** a **String** in the following **format, including only not sold parrots**:
  + **"**Parrots **available at {cageName}:  
    {**Parrot **1}  
    {**Parrot **2}  
    (…)**"

## Constraints

* The **names** of the parrots will be **always unique**.
* You will always have a parrot added before receiving methods manipulating the Cage’s parrots.

## Examples

This is an example how the **Cage** class is **intended to be used**.

|  |
| --- |
| Sample code usage |
| //Initialize the repository (Cage)  Cage cage = new Cage("Wildness", 20);  //Initialize entity  Parrot parrot = new Parrot("Fluffy", "Loriinae");  //Print Parrot  System.out.println(parrot); // Parrot (Loriinae): Fluffy  //Add Parrot  cage.add(parrot);  System.out.println(cage.count()); //1  //Remove Parrot  cage.remove("Parrot Name"); //false  Parrot secondParrot = new Parrot("Bunny", "Cacatuidae");  Parrot thirdParrot = new Parrot("Jumpy", "Strigopoidea");  Parrot fourthParrot = new Parrot("Puffy", "Strigopoidea");  Parrot fifthParrot = new Parrot("Marlin", "Arinae");    //Add Parrots  cage.add(secondParrot);  cage.add(thirdParrot);  cage.add(fourthParrot);  cage.add(fifthParrot);    //Sell Parrot by name  System.out.println(cage.sellParrot("Bunny")); //Parrot (Cacatuidae): Bunny  //Sell Parrot by species  List<Parrot> soldSpecies = cage.sellParrotBySpecies("Strigopoidea");  soldSpecies.forEach(f-> {  System.out.println(f.getName()); });  //Jumpy  //Puffy  System.out.println(cage.report());  //Parrots available at Wildness:  //Parrot (Loriinae): Fluffy  //Parrot (Arinae): Marlin |

## Submission

Submit **single .zip file**, containing **parrots package, with the classes inside (Parrot, Cage and the Main class)**, there is no specific content required inside the Main class e. g. you can do any kind of local testing of youр program there. However, there should be **main(String[] args)** method inside.