# Food Finder

You can test your solutions in [Judge](https://judge.softuni.org/Contests/Practice/Index/3228#0)

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**"**. | |

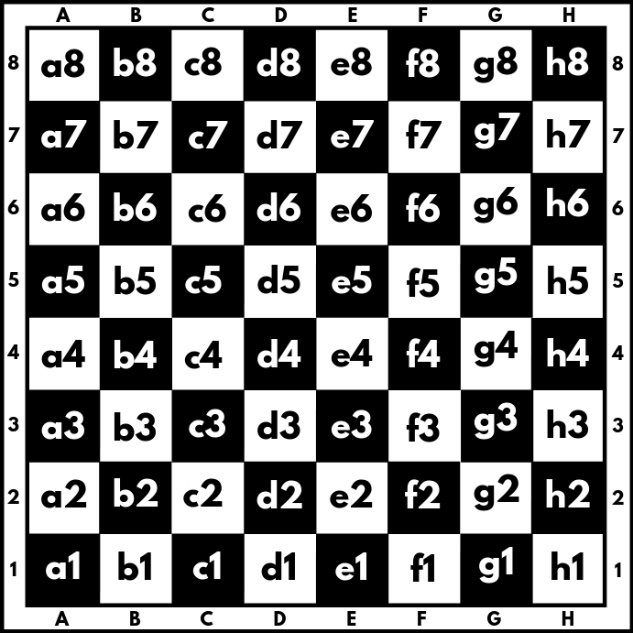
|  |  |
| --- | --- |
| **Input** | **Output** |
| **a o y**  **b h p j r n k** | **Words found: 1**  **pork** |

# 02. Pawn Wars

You can test your solutions in [Judge](https://judge.softuni.org/Contests/Practice/Index/3228#1)

## Rules

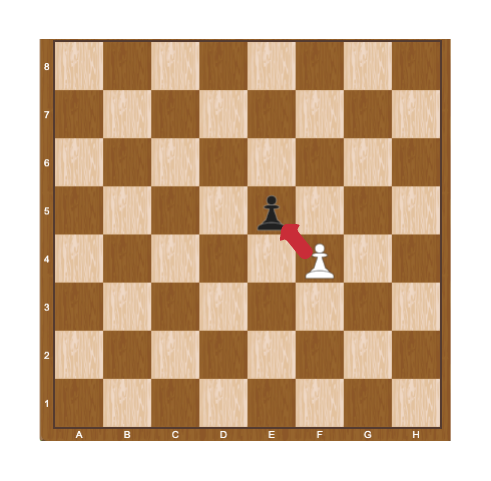
Before start solving this problem get familiar with a chessboard:



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.

## Problem Description

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.**"

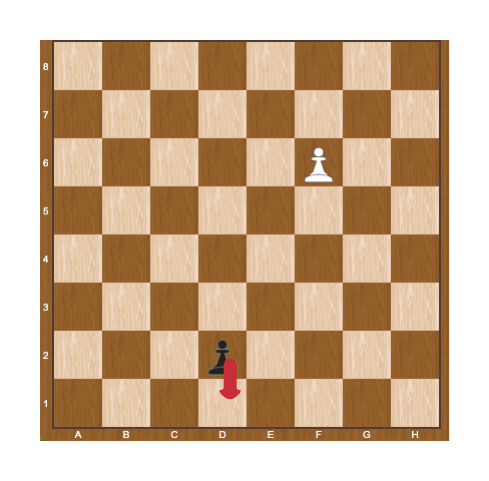
A picture containing text, checker

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-------  --------  -------- |

# 03. Stock Market

You can test your solutions in [Judge](https://judge.softuni.org/Contests/Practice/Index/3228#2)

*As a yang investor, you decide to create software, which will help you to keep track of stocks you own.*

## Preparation

Download the skeleton provided in Judge. **Do not** change the **StartUp** class or its **namespace**.

## Problem description

Your task is to create an investor with a portfolio of different stocks.

### Stock

You’ve been given a C# **class**, called **Stock** with properties:

* **CompanyName: string**
* **Director: string**
* **PricePerShare: decimal**
* **TotalNumberOfShares: int**
* **MarketCapitalization: decimal**

The **constructor**of **the Stock** class should receive **the CompanyName, Director, PricePerShare, and the TotalNumberOfShares.  MarketCapitalization** is a calculated property between **PricePerShare** and **a TotalNumberOfShares**.

The class should also have the following methods:

* Override **ToString()** method in the format:

**"****Company: {CompanyName}**

**Director: {Director}**

**Price per share: ${PricePerShare}**

**Market capitalization: ${MarketCapitalization}"**

### Investor

The **Investor**class has a **collection**(**portfolio**) of type **Stock** with corresponding **unique** **Company Name**of a **Stock**. The name of the collection should be **Portfolio**. All the entities of the **Portfolio** collection have the **same** properties. The **Investor** has also some additional properties:

* **FullName: string**
* **EmailAddress: string**
* **MoneyToInvest: decimal**
* **BrokerName: string**

The **constructor** of the **Investor** class should receive the **FullName, EmailAddress, MoneyToInvest,** and **BrokerName**.

Implement the coming features:

* Getter **Count** - returns the count of the currently owned stocks.
* Method **void** **BuyStock(Stock stock)** – add a single stock of the given company **if** the stock’s market capitalization is **bigger than $10000** and the **investor has enough money**. Then reduce the **MoneyToInvest**by the price of the stock. If a stock cannot be bought the method should not do anything.
* Method **string** **SellStock(string companyName, decimal sellPrice)** - sell a Stock from the portfolio with the given **company name for the given price**:
  + - * If the company does not exist return: **"{companyName} does not exist."**
      * If the selling price is smaller than the price per **share** return: **"Cannot sell {companyName}."**
      * Upon successful sell, you should remove the company from the portfolio, increase Money to Invest with the selling price, and return: **"{companyName} was sold."**
* Method **Stock** **FindStock(string companyName)** - returns a **Stock** with the given company name. If it doesn't exist, return **null**.
* Method **Stock** **FindBiggestCompany() –** returns the **Stock** with the biggest market capitalization. If there are no stocks in the portfolio, the method should return null.
* Method **string** **InvestorInformation()** - returns information about the Investor and his portfolio in the following format:

**"The investor {fullName} with a broker {brokerName} has stocks:**

**{Stock1}**

**{Stock2}**

**… "**

## Constraints

* Only a single stock of a company could be bought.
* The company name of each Stock in the portfolio will always be unique.
* The PricePerShare of a Stock and the MoneyToInvest of the Investor will always be positive numbers.
* There will not be a case where two Stock has the same CompanyName.
* You will always be given Stock added before receiving the method for its manipulation.

### Examples

|  |
| --- |
| Sample code usage |
| // Sample Code Usage:  // Initialize Investor  Investor investor = new Investor("Peter Lynch", "p.lynch@gmail.com", 2000m, "Fidelity");  // Initialize Stock  Stock ibmStock = new Stock("IBM", "Arvind Krishna", 138.50m, 5000);  // Print a stock  Console.WriteLine(ibmStock.ToString());  // Company: IBM  // Director: Arvind Krishna  // Price per share: $138.50  // Market capitalization: $692500.00  // Buy a stock  investor.BuyStock(ibmStock);  // Sell a stock  Console.WriteLine(investor.SellStock("IBM", 150.00m)); // "IBM was sold."  // Add stocks  Stock teslaStock = new Stock("Tesla", "Elon Musk", 743.17m, 6520);  Stock amazonStock = new Stock("Amazon", "Jeff Bezos", 3457.17m, 8500);  Stock twitterStock = new Stock("Twitter", "Jack Dorsey", 59.66m, 11200);  Stock blizzardStock = new Stock("Activision Blizzard", "Bobby Kotick", 78.53m, 5520);  // Buy more stocks  investor.BuyStock(teslaStock);  investor.BuyStock(amazonStock);  investor.BuyStock(twitterStock);  investor.BuyStock(blizzardStock);  // FindBiggestCompany  Console.WriteLine(investor.FindBiggestCompany());  // Company: Tesla  // Director: Elon Musk  // Price per share: $743.17  // Market capitalization: $4845468.40  // Print investor information  Console.WriteLine(investor.InvestorInformation());  // The investor Peter Lynch with a broker Fidelity has stocks:  // Company: Tesla  // Director: Elon Musk  // Price per share: $743.17  // Market capitalization: $4845468.40  // Company: Twitter  // Director: Jack Dorsey  // Price per share: $59.66  // Market capitalization: $668192.00  // Company: Activision Blizzard  // Director: Bobby Kotick  // Price per share: $78.53  // Market capitalization: $433485.60 |

## Submission

Zip all the files in the project folder except **bin** and **obj** folders.