# 01. Blacksmith

You can test your solutions in [Judge](https://judge.softuni.org/Contests/Practice/Index/3285#0)

*You are the most well-known blacksmith in Middle Earth. What makes your swords so good is the perfect ratio between* *steel and carbon*, *which makes them extremely sharp and durable.*

First, you will be given **a sequence representing steel**. Afterward, you will be given another **sequence representing carbon**.

You should start from the **first steel** and try to mix it with the **last carbon.** If the **sum** of their values is **equal** to **any of the swords in the table below**, you should forge the **sword, corresponding** to the **value** and **remove** **both** the **steel** and the **carbon**. Otherwise, **remove only the steel**, **increase** the **value** of the **carbon by 5** and **insert** it **back** into the **collection**. You need to **stop** forging when you have **no more steel** or **carbon left**.

|  |  |
| --- | --- |
| **Sword** | **Resources needed** |
| Gladius | 70 |
| Shamshir | 80 |
| Katana | 90 |
| Sabre | 110 |
| Broadsword | 150 |

Forge as **many swords as possible.**

### Input

* On the **first line**, you will receive the steel, **separated** by a **single space (**' '**)**.
* On the **second line**, you will receive the carbon, **separated** by a **single space ('** '**)**.

### Output

* On the **first** line of output depending on the result:
  + If at least one sword was forged: "**You have forged {totalNumberOfSwords} swords.**"
  + If no sword was forged: "**You did not have enough resources to forge a sword.**"
* On the **second** line – print all steel you have left:
  + If there are no steel: "**Steel left: none**"
  + If there are steel: "**Steel left: {steel1}, {steel2}, {steel3},** **(…)**"
* On the **third** line - print all carbon you have left:
  + If there are no carbon: "**Carbon** **left: none**"
  + If there are carbon: "**Carbon** **left: {carbon1}, {carbon2}, {carbon3},** **(…)"**
* Then**,** you need to print **only the swords that you manage to forge** and how many **of them**, **ordered** **alphabetically**:
  + **"Broadsword: {amount}"**
  + **"Sabre: {amount}"**
  + **"Katana: {amount}"**
  + **"Shamshir: {amount}"**
  + **"Gladius: {amount}"**

### Constraints

* All of the given numbers will be valid resources in the range **[0, 130]**.

### Examples

|  |  |  |
| --- | --- | --- |
| ****Input**** | ****Output**** | ****Comment**** |
| **40 50 70 120 10 20**  **30 20 30 20 30 50** | **You have forged 4 swords.**  **Steel left: none**  **Carbon left: 30, 30**  **Broadsword: 1**  **Katana: 2**  **Shamshir: 1** | We start by taking **40 (steel) + 50 (carbon) = 90**. The first sword is forged "**Katana**" and we remove both materials.  Next **50 (steel) + 30 (carbon) = 80**, "**Shamshir** " is forged and we remove both materials.  Next **70 (steel) + 20 (carbon) = 90**, "**Katana**" is forged and we remove both materials.  Next **120 (steel) + 30 (carbon) = 150**, "**Broadsword**" is forged and we remove both materials.  Next **10 (steel) + 20 (carbon) = 30**, no sword could be made, we remove **steel**, increase **carbon** by 5 and insert back **(25)** into the collection.  In the last iteration **20 (steel) + 25 (carbon) = 45** no sword could be made, we remove **steel**, increase **carbon** by 5 and insert back **(30)** into the collection.  We’ve menage to forge **4 swords** and left with **2 pieces (30, 30)** of carbon. |
| **10 5 30**  **30 20 10** | **You did not have enough resources to forge a sword.**  **Steel left: none**  **Carbon left: 25, 20, 30** |  |

# 02. Truffle Hunter

You can test your solutions in [Judge](https://judge.softuni.org/Contests/Practice/Index/3388#1)

*In the summer Peter loves to hunt truffles. He has a well-trained dog, but his problem is how to deal with the wild boars, which eat his trifles.*

Create a program that helps Peter to **pick up truffles**. There are **three** kinds of **truffles** in the forest:

* **Black truffle** - **'B'**
* **Summer truffle** - **'S'**
* **White truffle** - **'W'**

**On the first line**, the size of the forest is given, which will be a matrix with a square shape. Then, for **each** row, you will receive the **truffles**. **All of the empty positions** will be marked with a **'-' (dash).**

Then you will start receiving **commands**. Here are the possible ones you can receive:

* **"Collect {row} {col}"** – Peter goes to the given place in the forest and collects the truffle, **if it exists**. When he collects a truffle, the cell's value should change to a dash (**'-'**)**.**
* **"Wild\_Boar {row} {col} {direction}"** – a wild boar appears in the given coordinates (they will be always **valid**) in the forest and it goes all the way in that direction. Which means the wild boar goes from the given **cell** to the **last cell** in the given direction. It **eats** the **given cell**, **skips the next**, and **eats** **the next one**, if there is a truffle there and so on, **until it reaches the last cell in the given direction**. **Mark** the **eaten** cells with **a dash** (**'-'**). There are four possible directions:
  + **"up", "down", "left" and "right".**
* **"Stop the hunt"** – the program stops and the result is printed.

Here is an example (**up** and **right**) **of the wild boar eating pattern**:

A screenshot of a game

Description automatically generated with medium confidence

In the end, **print** the **truffles** that Peter has successfully **found** and the **count** of truffles that the **wild boar** ate in the following format:

**"Peter manages to harvest {count black truffles} black, {count summer truffles} summer, and {count white truffles} white truffles."**

**"The wild boar has eaten {count of truffles} truffles."**

Then **print the last state of the forest.**

### Input

* On the first line, you will receive **the size of the forest (matrix) in square shape**.
* On the next lines, for **each** **row**, you will receive the truffles in the described format.
* Next, until you receive **"Stop the hunt"**, you will be receiving commands in the described format.

### Output

As an output **print three lines**:

* All types of truffles that Peter has **collected** in the format described above.
* Truffles **eaten by the wild boar** in the format described above.
* The **final state of the forest** - each cell separated by a single space.

### Constraints

* The size of the square matrix will be between [3…10].
* The coordinate of the wild boar will be always valid.
* The input will always be **valid** and you don't need to check it explicitly.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | |
| **5**  **B W S - -**  **S S B W W**  **S S - W B**  **B B S - W**  **S S - - -**  **Collect 0 2**  **Collect 3 0**  **Collect 4 2**  **Collect 3 4**  **Collect 2 3**  **Wild\_Boar 2 2 up**  **Wild\_Boar 1 1 right**  **Stop the hunt** | **Peter manages to harvest 1 black, 1 summer, and 2 white truffles.**  **The wild boar has eaten 2 truffles.**  **B W - - -**  **S - B - W**  **S S - - B**  **- B S - -**  **S S - - -** | |
| **Comment** | | |
| The first command is **"Collect",** so we go to the given coordinates and collect **'S' (summer truffle).**  **The next command is** "**Collect**"**, so we go to the given coordinates and collect 'B' (black truffle).**  **The next command is** "**Collect**"**, so we go to the given coordinates but there is no truffle there, so we continue to the next command.**  **The next command is** "**Collect**"**, so we go to the given coordinates and collect 'W' (white truffle).**  **The next command is** "**Collect**"**, so we go to the given coordinates and collect 'W' (white truffle).**  The next command is **"Wild\_Boar"**, so to the wild boar start from the given index in a direction up. There were not any truffles in the coordinates, so the wild boar leave without eating anything**.**  The next command is **"Wild\_Boar"**, so to the wild boar start from the given index in a direction right. He menage to eat two truffles, which are **'S' (summer truffle) and 'W' (white truffle).**  **The last command is** "**Stop the hunt**"**, the program stops, and the corresponding message is printed, with the last state of the forest.** | | |
| **Input** | | **Output** |
| **4**  **B - S W**  **S - B W**  **S S - B**  **B W S -**  **Collect 0 0**  **Collect 1 2**  **Collect 3 2**  **Collect 3 1**  **Stop the hunt** | | **Peter manages to harvest 2 black, 1 summer, and 1 white truffles.**  **The wild boar has eaten 0 truffles.**  **- - S W**  **S - - W**  **S S - B**  **B - - -** |

# 03. Stock Market

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

*As a young investor, you decide to create a software, which will help you to keep track of the 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 **MoneyToInvest** 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** have 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.