# Python Advanced: Exam Preparation

# Ramen Shop

**Link to Judge:** <https://judge.softuni.org/Contests/Practice/Index/3430#0>

You will be given **two sequences** of **integers** representing **bowls of ramen** and **customers**. Your task is to find out **if you can serve all the customers**.

Start by taking **the last bowl** of ramen and **the first customer**. Try to serve every customer with ramen **until we have no more ramen or customers left**:

* **Each time** the value of the ramen **is equal** to the value of the customer, **remove them both** and **continue** with the next **bowl of ramen** and the **next customer**.
* **Each time** the value of the ramen **is bigger** than the value of the customer, **decrease** the **value of that ramen** **with** the **value of that customer** and **remove the customer**. Then try to **match** the **same bowl of ramen** (which has been decreased) with the **next customer**.
* **Each time** the customer's value **is bigger** than the value of the ramen bowl, **decrease** the **value of the customer** **with** the **value of the ramen bowl** and **remove the bowl**. Then try to **match** the **same customer** (which has been decreased) with the **next bowl of ramen**.

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

### Input

* On the **first line**, you will receive **integers** representing the bowls of ramen, **separated** by a single space and a comma ", ".
* On the **second line**, you will receive **integers** representing the customers**,** **separated** by a single space and a comma ", ".

### Output

* If all customers are served, print: "**Great job! You served all the customers.**"
  + Print all of the left **ramen bowls** (**if any**) separated by **comma and space** in the format: **"Bowls of ramen left: {bowls of ramen left}"**
* Otherwise, print: **"Out of ramen! You didn't manage to serve all customers."**
  + Print all **customers** left separated by **comma and space** in the format **"Customers left: {customers left}"**

### Examples

|  |  |  |
| --- | --- | --- |
| ****Input**** | | ****Output**** |
| **14, 25, 37, 43, 19**  **58, 23, 37** | | **Great job! You served all the customers**  **Bowls of ramen left: 14, 6** |
| ****Comment**** | | |
| Start by taking the last bowl 19 and the first customer 58. The customer value is higher, so we remove the bowl and decrease the value of the customer by 19. Now the two lists should look like this:  Bowls = [14, 25, 37, 43]  Customers = [39, 23, 37]  Next, we take the following bowl (43) and continue with the same customer who is 39 now. The value of the bowl with ramen is higher than the customer's value, so we remove the customer and decrease the value of the ramen bowl. Now the two lists should look like this:  Bowls = [14, 25, 37, 4]  Customers = [23, 37]  We take the last bowl of ramen, which is 4 now, and compare it with the next customer (23). The value of the customer is higher, so we decrease his value by 4 and remove the last bowl. Now the two lists should look like this:  Bowls = [14, 25, 37]  Customers = [19, 37]  Then we continue with the ball 37 and customer 19. The bowl is higher. We remove the customer and decrease the bowl value with the value of the customer 19. Now the two lists should look like this:  Bowls = [14, 25, 18]  Customers = [37]  Then we continue with bowl 18 and customer 37. The customer value is higher. We remove the bowl and decrease the customer value with the value of bowl 18. Now the two lists should look like this:  Bowls = [14, 25]  Customers = [19]  Then we continue with the ball 25 and customer 19. The bowl is higher. We remove the customer and decrease the bowl value with the value of the customer 19. Now the two lists should look like this:  Bowls = [14, 6]  Customers = []  We see that we served all of the customers and print the appropriate string for that case. After that, we print the leftover bowls of ramen. | | |
| ****Input**** | ****Output**** | |
| **30, 13, 45**  **70, 25, 55, 15** | **Out of ramen! You didn't manage to serve all customers.**  **Customers left: 7, 55, 15** | |
| ****Input**** | ****Output**** | |
| **30, 25**  **20, 35** | **Great job! You served all the customers.** | |

# Navy Battle

**Link to Judge:** <https://judge.softuni.org/Contests/Practice/Index/3744#1>

*1914, September 22 – German submarine* [*U-9*](https://en.wikipedia.org/wiki/SM_U-9)[*sinks three unescorted British armored cruisers*](https://en.wikipedia.org/wiki/Action_of_22_September_1914)[*HMS*Aboukir](https://en.wikipedia.org/wiki/HMS_Aboukir_(1900))*,* [*HMS*Hogue](https://en.wikipedia.org/wiki/HMS_Hogue_(1900)), *and* [*HMS*Cressy](https://en.wikipedia.org/wiki/HMS_Cressy_(1899)) *in approximately one hour. Imagine that they had the technology to make themselves a navigational program for the submarine and you are chosen to implement the logic. Navigate U-9 through the battlefield, find and sink the British cruisers in the dark night, avoiding the floating mines all over the North Sea.*

You will be given an integer **n** for the size of the **battlefield** (square shape). On the next **n** lines, you will receive the rows of the **battlefield**. The submarine will start at a **random** position, marked with the letter '**S**'. The submarine surveys the surrounding area through its periscope, so it has to climb up to periscope depth, where it might run across naval **mines**.

When the submarine receives direction, it goes deep and moves **one position toward the given direction**. On each turn, you will be guiding the submarine and giving it the **direction**, in which it should move. The commands will be "**up**", "**down**", "**left**" and "**right**".

When a **new position is reached**, the submarine climbs up to periscope depth to search for a cruiser:

* If a position with '**-**' (dash) is reached, it means that the field is empty and the submarine awaits its next direction.
* **If** it **runs across** a naval **mine (**'**\***'**)**, the submarine takes serious **damage**. When a mine is blown, the position of the mine will be marked with '**-**' (dash). U-9 can **withstand two hits** from naval mines. The **third time** the submarine **is hit** by a mine, it **disappears** and the **mission** **is** **failed. The battle is over** and the following message should be printed on the Console: "**Mission failed, U-9 disappeared! Last known coordinates [{row}, {col}]!**"
* If a battle cruiser is reached **(**'**C**'**)**, the submarine destroys it and the position of the destroyed cruiser will be marked with '**-**' (dash).
* If this is the **last** (third) battle **cruiser** on the **battlefield**, **the battle is over** and the following message should be printed on the Console: "**Mission accomplished, U-9 has destroyed all battle cruisers of the enemy!**"

**The program will end when the battle is over (All battle cruisers are destroyed or the submarine hits mines three times).**

### Input

* On the first line, you are given the integer **n** – the size of the matrix (wall).
* The **next n lines** hold the values for every **row** (**NOT** separated by anything).
* On each of the next lines you will get a direction command.

### Output

* If all battle **cruisers** are destroyed, print: "**Mission accomplished, U-9 has destroyed all battle cruisers of the enemy!**"
* If U-9 is hit by a mine three times, print: "**Mission failed, U-9 disappeared! Last known coordinates [{row}, {col}]!**".
* At the end, print the **final state** of the matrix (**battlefield**) with the **last known U-9’s position** on it.

### Constraints

* The size of the **square** matrix (**battlefield**) will be between **[4…10].**
* U-9’s starting position will always be marked with '**S**'.
* There will be always three battle cruisers - fields marked with '**C**'.
* There will be always enough mines on the battlefield to destroy the submarine.
* The commands given will direct the submarine only in the limits of the battlefield.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  \*--\*-  -S-\*C  -\*---  -----  -C-\*C  right  down  left  up  right  right  right  down  down  down  up  left  left  left  down | Mission accomplished, U-9 has destroyed all battle cruisers of the enemy!  \*--\*-  -----  -----  -----  -S-\*- |
| 5  \*--\*-  -S-\*C  -\*---  -----  \*C-\*C  right  right  up  left  left  left | Mission failed, U-9 disappeared! Last known coordinates [0, 0]!  S----  ----C  -\*---  -----  \*C-\*C |

# Hourly Forecast

**Link to Judge:** <https://judge.softuni.org/Contests/Practice/Index/3596#2>

*Patricia wants to go on vacation for the weekend and wants to know where the weather will be the best, so she can see the most sights. Patricia is busy at work and doesn't have time to think about the perfect place for her vacation, so she wants your help.*

Write a function called **forecast** that **stores information** about the **weather**, and **returns sorted information for all locations**. The function will receive a **different number of arguments**. The arguments will be passed as **tuples with two elements** - the **first** one is the **location**, and **the second one** is the **weather**:

* **Location name**: string
  + any string
* **Weather**:string
  + "Sunny"
  + "Rainy"
  + "Cloudy"

First, **sort all** **locations by weather**. **First** are positioned the locations with **sunny weather**, next are the locations with **cloudy weather**, and **last** are the locations with **rainy weather**. For each sequence of locations (e.g. all sunny locations), sort them by their name in **ascending order** (alphabetically).

**In the end, return** the output as described below.

***Note: Submit only the function in the judge system***

### Input

* There will be **no input from the console**, just parameters passed to your function

### Output

* The **output** should look like this**:**

**"{first\_sorted\_location} - {weather}"**

**"{second\_sorted\_location} - {weather}"**

**…**

**"{last\_sorted\_location} - {weather}"**

### Constraints

* Each **tuple** given will always contain the **location** with its **weather**.
* You will **never** receive the **same location** twice or more times.

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| print(forecast(  ("Sofia", "Sunny"),  ("London", "Cloudy"),  ("New York", "Sunny"))) | New York - Sunny  Sofia - Sunny  London - Cloudy |
| print(forecast(  ("Beijing", "Sunny"),  ("Hong Kong", "Rainy"),  ("Tokyo", "Sunny"),  ("Sofia", "Cloudy"),  ("Peru", "Sunny"),  ("Florence", "Cloudy"),  ("Bourgas", "Sunny"))) | Beijing - Sunny  Bourgas - Sunny  Peru - Sunny  Tokyo - Sunny  Florence - Cloudy  Sofia - Cloudy  Hong Kong - Rainy |
| print(forecast(  ("Tokyo", "Rainy"),  ("Sofia", "Rainy"))) | Sofia - Rainy  Tokyo - Rainy |