# Meal Plan

You can test your solutions in [Judge](https://judge.softuni.org/Contests/Practice/Index/3388#0)

*The summer is around the corner and John wants to lose some weight. To do that he should lower his calories intake, by keeping a count of the total amount of calories intake for a day.*

Create a program that helps John in calculating his calories intake, by predefined meals.

You will be given **two lines of input.** The first line will be an **array of strings**, separated by a single space, representing **meals** that John can eat. The second line will be an **array of integers**, separated by a single space, representing the maximum **calories intake per day**.

Here is a table with the exact **meals** and **their calories**:

|  |  |
| --- | --- |
| **Meal** | **Calories** |
| salad | 350 |
| soup | 490 |
| pasta | 680 |
| steak | 790 |

Start calculating calories as follows: take the first **meal's calories** and the **last element** in the sequence of **calorie intake for a day**. Each time John eats a meal, the **calories intake** for the current day should be **decreased** by the calories of the **given meal**, and the **current meal** should be **removed** from the collection. When the calories intake for the given day **goes to zero**, **remove** them from the collection, and continue to the **next day**. If the calories for the given day go below zero, take as **many calories as possible** from the meal, so the current daily calories intake reaches **zero**. The **rest** of the meal's calories should be subtracted from the next day in the **calorie intake** sequence.

For more clarification, check the provided examples.

The program stops when there are no **more meals to be eaten**, or there are **no more daily calories** to be calculated.

### Input

* On the **first line**, a sequence of **strings will be given** representing **meals**, **separated** by a single space (**' '**).
* On the **second line**, a sequence of **integers will be given,** representing the **daily calories intake**, **separated** by a single space (' **'**)**.**

### Output

* As a result, you should print two lines for output:
  + If John managed to **eat all the given meals**, print:

**"John had {number of meals} meals."**

**"For the next few days, he can eat {dailyCalories1, dailyCalories2, etc.} calories."**

Separate the **daily calories intake** by comma and space **(", ")**.

* + If John could **not eat every meal**:

**"John ate enough, he had {number of meals} meals."**

**"Meals left: {meal1, meal2, etc.}."**

Separate the **meals** by comma and space **(", ")**.

### Constraints

* The **daily calories** will be in the range [1200…4000].
* No recursion is needed for solving this problem.
* The input will be always valid.
* There will be no case where **daily calories** and **meals** go to zero simultaneously.

### Examples

|  |  |
| --- | --- |
| ****Input**** | ****Output**** |
| salad soup salad steak  2500 1800 1500 | **John had 4 meals.**  **For the next few days, he can eat 1320, 2500 calories.** |
| ****Comment**** | |
| We start by taking a **salad** (350 cal.) and a day with 1500 cal. We subtract the daily calories intake by the **salad's** calories:  1500 - 350 = 1150 cal.  The **salad** is removed from the sequence, and John can consume 1150 cal. more for the **current day**.  Next, we have **soup** (490 cal.), we subtract the daily calories intake by the **soup's** calories:  1150 - 490 = 660 cal.  The **soup** is removed from the sequence, and John can consume 660 cal. more for the **current day**.  Next, we have a **salad** (350 cal.), the same operation is performed:  660 - 350 = 310 cal.  The **salad** is removed from the sequence, and John can consume 310 cal. more for the **current day**.  Next, we have **steak** (790 cal.), and 310 cal. that can be consumed for the **current day**:  310 - 790 = -480 cal.  We take 310 cal. form the **steak's** total of 790 cal. and remove the current daily calory intake. Next, we check do we have another day in the collection, in this case, we do, which is 1800 cal.:  1800 - 480 = 1320 cal.  The **steak** is removed from the sequence, and John can consume 1320 cal. more for the **current day**.  There are **no more meals** in the collection, so we stop the program and print the corresponding messages. | |

|  |  |
| --- | --- |
| ****Input**** | ****Output**** |
| **pasta soup salad steak pasta steak pasta**  **1800 1500** | **John ate enough, he had 6 meals.**  **Meals left: pasta.** |

# 02. Beaver at Work

You can test your solutions in [Judge](https://judge.softuni.org/Contests/Practice/Index/3349#1)

*The Beaver got married and he needs to build a new home for his family. Help him collect as many wood branches as possible for his new home.*

You will receive an integer **n** for the **size** of the **pond** with a **square** shape. On the next **n lines,** you will receive the **matrix**, which represents the **pond.**

The Beaver will be placed in a **random** **cell**, marked with the letter '**B**'. Each cell stands for a place where the beaver can move. If the cell is marked with а **lowercase character**, that means there is a **wood** **branch**. If thecell is **marked** with **'F'**, the beaver catch and eats a **fish**. **All of the empty positions** will be marked with a **'-' (dash).**

The Beaver can move "**up**", "**down**", "**left"** and "**right**". These will be the **commands** that you**'**ll receive. Anytime the beaver moves, change the value of the cell of his new position to '**B**' and the cell it left to **'-' (dash)**.

* If the **beaver** **moves to a wood** **branch**, he puts it away for his new home.
* If the beaver moves outside of the pond (field), he drops **his last collected wood branch** (if there are any), **without changing its current position**.
* If the beaver moves to **'F'**, he eats a **fish** and gains the **strength to swim underwater for a very long time.**
  + **If the fish is NOT located in the last index, the beaver swims to the last index in the direction it received.**

**Еxample**: If the beaver is located on **[0;0] and** moves **right**, eats a **fish in [0;1]**, he swims all the way **right** to the **column's last position**, **[0;lastIndex]** **(row remains the same)**.

* + **If the braver eats a fish on the last index – he swims in the opposite direction**. Opposite of "**up**" is "**down**", opposite of "**left"** is"**right"** and vice versa. Set the value of the fish**'**s cell to **'-' (dash)**. If there is a **wood branch at the cell the beaver swims to,** it **collects** the wood branch. While swimming underwater, the beaver **does not** collect any wood branches.

**Еxample**: If the beaver moves **up**, and eats a **fish in [0;0]**, he **can't swim all the way up**, so **he swims all the way down** to the **row's last position**, **[lastIndex;0]** **(column remains the same)**.

When the command"**end**" is received or the beaver **manages** to **collect** **all the wood** **branches**, **stop** the program, and **print** the result in the format below.

### Input

* On the first line – integer **n** – the size of the pond (field).
* The **next n lines** hold the values for every **row.**
* On each of the next lines, you will receive a command.

### Output

* If the Beaver manages to collect **all wood branches** in the pond, print:

**"The Beaver successfully collect {numberOfbranches} wood branches: {branch1}, {branch2}, {branch3}(…)."**

* If the Beaver could not collect every branch in the pond, print:

**"The Beaver failed to collect every wood branch. There are {numberOfbranches} branches left."**

* Then print the last state of the pond.

### Constraints

* The size of the square matrix will be between [2…15].
* The Beaver's position will be marked with **'B'**.
* The fish will be marked with **'F'**.
* **Wood branches** will be lower case letters from the English **alphabet (a - z)**.
* There will be no case where the **Beaver** will **move** to **a fish two consecutive times**.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 4  - F e -  - B F y  - - - q  - - z x  up  right  right  right  up  end | The Beaver failed to collect every wood branch. There are 2 branches left.  - - e -  - - F y  - - - B  - - - - | The first command is **up,** so **B (beaver)** moves up.  **On turn 1**: the beaver moves **to fish**. He should **swim up**, but the fish is placed at the **end of the pond**, so the braver swims in the **opposite direction, which is down**. **[3;1]**  **On turn 2:** thecommand is **right**, the beaver moves right and collects a **wood branch**. Now he has one wood branch ('z').  **On turn 3:** the command is **right**, the beaver moves right and collects another **wood branch**. Now he has **two wood** branches ('z', 'x').  **On turn 4**: command is **right**, the beaver moves right, which is an **invalid index**, so he **loses his last wood branch** ('x'). Now he has only one wood branch ('z').  **On turn 5**: the command is **up**, the beaver moves up and collects another branch. Now he has two branches ('z', 'q').  We receive command **end** so we stop the program. There are **two branches left in the pond**. We print the output and the last state of the pond.  **Turn 1 Turn 2 Turn 3 Turn 4 Turn 5**  - - e - - - e - - - e - - - e - - - e -  - - F y - - F y - - F y - - F y - - F y  - - - q - - - q - - - q - - - q - - - B  - B z x - - B x - - - B - - - B - - - - |
| 3  - - -  B F -  d b m  down  left  right  right  right | The Beaver successfully collect 2 wood branches: b, m.  - - -  - F -  - - B |  |

# 03. Drones

You can test your solutions in [Judge](https://judge.softuni.org/Contests/Practice/Index/3285#2)

## Preparation

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

## Problem description

Your task is to create an airfield, where drones can take off and land.

# Drone

You are given a class **Drone,** create the following fields:

* **Name: string**
* **Brand: string**
* **Range: int**
* **Available: boolean - true by default**

The class **constructor** should receive **(name, brand, range)**.

The class should also have a method:

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

**"Drone: {name}**

**Manufactured by: {brand}**

**Range: {range} kilometers"**

# Airfield

Next, a class named **Airfield**is given, that has a **collection**(**drones**) of type **Drone**. The name of the collection should be **Drones**. All the entities of the **Drones** collection have the **same** properties. The **Airfield** has also some additional properties:

* **Name: string**
* **Capacity: int**
* **LandingStrip: double**

The **constructor** of the **Airfield** class should receive the **name, capacity and landing strip**.

Implement the following features:

* Getter **Count** - returns the count of the drones in the airfield.
* string AddDrone(Drone drone) - **adds** a drone to the drones collection, **if** **there** **is** **room** for it. Before adding a drone, check:
  + - * If the **name** or **brand** are **null or empty**.
      * If the **range** is **NOT** between **5-15 kilometers**.

If the **name, brand** or **range** properties are not valid, return: **"Invalid drone.".** If the airfield is full (there is no room for more drones), return **"Airfield is full.".** Otherwise, return: **"Successfully added {droneName} to the airfield."**

* bool RemoveDrone(string name) - removes a drone by **given name,** if such **exists return true**, otherwise **false.**
* int RemoveDroneByBrand(string brand) - removes **all drones** by the given **brand,** if such **exists, return how many drones were removed,** otherwise **0.**
* Drone FlyDrone(string name) method – **fly** (**set** its available **property** to **false** without removing it from the collection) the **drone** with the **given name, if exists**. As a result, **return** the **drone or return null, if does not exist.**
* List<Drone> FlyDronesByRange(int range) method - **fly** and return **all drones,** which have a range **equal or bigger** to the given. **Return a list of all drones which have been flown.** The range will always be valid.
* **Report()** -returns information about the airfield and drones which are **not in flight** in the following format:
  + **"**Drones **available at {airfieldName}:  
    {**Drone**1}  
    {**Drone**2}  
    (…)**"

**Note: Do not use** "\n\r" **for a new line.**

## Constraints

* The **names** of the drones will be **always unique**.
* You will always have a drone added before receiving methods manipulating the Airfield’s drones.

## Examples

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

|  |
| --- |
| Sample code usage |
| // Initialize the repository (Airfield)  Airfield airfield = new Airfield("Heathrow", 10, 10.5);  // Initialize entity  Drone drone = new Drone("D20", "DEERC", 6);  // Print Drone  Console.WriteLine(drone);  // Drone: D20  // Manufactured by: DEERC  // Range: 6 kilometers  // Add Drone  Console.WriteLine(airfield.AddDrone(drone));  // Successfully added D20 to the airfield.  Console.WriteLine(airfield.Count); // 1  // Remove Drone  Console.WriteLine(airfield.RemoveDrone("DE51"));  // False  Drone secondDrone = new Drone("CW4", "Cheerwing", 8);  Drone thirdDrone = new Drone("X5SW-V3", "Cheerwing", 7);  Drone fourthDrone = new Drone("X20", "Cheerwing", 4);  Drone fifthDrone = new Drone("EVO2", "Autel", 10);  Drone sixtDrone = new Drone("XL5-6S-FPV", "iFlight", 10);  // Add Drones  Console.WriteLine(airfield.AddDrone(secondDrone));  // Successfully added CW4 to the airfield.  Console.WriteLine(airfield.AddDrone(thirdDrone));  // Successfully added X5SW-V3 to the airfield.  Console.WriteLine(airfield.AddDrone(fourthDrone));  // Invalid drone.  Console.WriteLine(airfield.AddDrone(fifthDrone));  // Successfully added EVO2 to the airfield.  Console.WriteLine(airfield.AddDrone(sixtDrone));  // Successfully added XL5-6S-FPV to the airfield.  // Fly drone by name  Console.WriteLine(airfield.FlyDrone("CW4"));  // Drone: CW4  // Manufactured by: Cheerwing  // Range: 8 kilometers  Console.WriteLine("-----------------FlyDronesByRange-----------------");  List<Drone> flyDrones = airfield.FlyDronesByRange(10);  foreach (var droneToFly in flyDrones)  {  Console.WriteLine(droneToFly);  }  /\*  Drone: EVO2  Manufactured by: Autel  Range: 10 kilometers  Drone: XL5-6S-FPV  Manufactured by: iFlight  Range: 10 kilometers  \*/  // Remove drone by brand  Console.WriteLine(airfield.RemoveDroneByBrand("Cheerwing"));  // 2  Console.WriteLine("----------------------Report----------------------");  Console.WriteLine(airfield.Report());  /\*  Drones available at Heathrow:  Drone: D20  Manufactured by: DEERC  Range: 6 kilometers  \*/ |

## Submission

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