# Tiles Master

You can test your solutions in [Judge](https://judge.softuni.org/Contests/Practice/Index/3525#0)

*Summer is coming which means it's that time of the year when everybody is renovating their home…*

Victoria wants to renovate her apartment, too. She's starting with the kitchen and she has hired a handyman to put **new tiles** on the **walls** and the **floor**. She wants a modern kitchen and she has bought fancy tiles, which must be put in an **exact order** and **locations** to achieve the glamorous look of the kitchen that she expects.

Unfortunately, the **tiles** come in **different sizes and colours**, and your task is to help Victoria and the handyman Vanko **place them in the right order** and **location** in the kitchen.

First, you will be given **a sequence of numbers, representing the areas of the white tiles**. Afterward, you will be given another **sequence, representing the areas of the grey tiles.**

You **start** from the **first** **grey** tile and compare its **area** to the **area** of the **last** **white** tile.

If their **areas** are **equal**, **together they form a new larger tile**, that the handyman will be able to use. After that, you should check whether the **area of the new-formed tile matches** one of the numbers in the **table** **below** (the numbers correspond to a particular **location** in the kitchen). The **area** of the **new** **tile** is formed by **summing** the **areas** of the **white** and the **grey** tile.

If the **area of the new-formed tile matches** the necessary area tile for any of the locations in the kitchen, you should remove both the grey and white tiles from the sequences. If the **area doesn't match** any of the specified **locations**, the **new tile** will be used for a location, named **Floor**.

If their **areas don't match at all**, you take the **white** tile, **decrease its area in half** and insert it **back** **to the sequence**. After that, you **change the grey's tile position by putting it at the back of the sequence**.

|  |  |
| --- | --- |
| **Location** | **Tile area needed** |
| Sink | 40 |
| Oven | 50 |
| Countertop | 60 |
| Wall | 70 |

**Compare** all of the **areas** while **keeping track of the tiles** that will be **used** for the particular **locations** in the kitchen. You stop **comparing** them in case the handyman is both **out of white and grey tiles**, or in case **any one of them is out**. Finally, **print** the **number** of tiles that will be used for the **different** locations.

### Input

* On the **first line**, you will receive the areas of the white tiles, **separated** by a **single space (**' '**)**.
* On the **second line**, you will receive the areas of the grey tiles, **separated** by a **single space (' ')**.

### Output

* On the **first** line – print all white tiles you have left:
  + If there are no white tiles left: "**White tiles left: none**"
  + If there are white tiles left: "**White tiles left: {whiteTile1}, {whiteTile2}, {whiteTile3},** **(…)**"
* On the **second** line - print all grey tiles you have left:
  + If there are no grey tiles left: "**Grey tiles** **left: none**"
  + If there are grey tiles left: "**Grey tiles** **left: {greyTile1}, {greyTile2}, {greyTile3},** **(…)"**
* Then**,** you need to print **only the locations in the kitchen that will be decorated with the new-formed tiles (including the floor),** and the count of new tiles that will be used for them. The locations must be ordered descending by number (count of new tiles per location) and then sorted ascending alphabetically.
  + **"Countertop: {amount}"**
  + **"Floor: {amount}"**
  + **"Oven: {amount}"**
  + **"Sink: {amount}"**
  + **"Wall: {amount}"**

### Constraints

* All of the given numbers will be valid integers in the range **[0…200]**.
* There will be **no** case where the white tiles' area reaches 0.
* The areas of the white tiles that must be divided in half will **always** be **even** numbers.

### Examples

|  |  |  |
| --- | --- | --- |
| ****Input**** | ****Output**** | ****Comment**** |
| **35 16 30 3 25 9 20**  **20 9 25 3 30 16 35** | **White tiles left: none**  **Grey tiles left: none**  **Floor: 3**  **Countertop: 1**  **Oven: 1**  **Sink: 1**  **Wall: 1** | We start by taking the area of the last white tile and compare it to the area of the first grey tile – 20. They are equal, so we form the new tile by summing the areas of both tiles (20 + 20 = 40). After that, we check if there is a location that requires a tile with area of 40. There is such a location, so we add the new tile to the collection with new-formed tiles and remove both the white and the grey tile.  Next, we have 9 (area of the next white tile) and 9 (area of the next grey tile). They are equal, so we check if there is a location that requres a tile with area of 9 + 9 = 18. There is no such location, so the tiles will be used for the **floor** and we add it to the collection with new-formed tiles. Again, we remove both the white and the grey tile, because they were just used.  Next, **25 (white tile area) = 25 (grey tile area).** 25 + 25 = 50, the new tile will be used for the **Oven** location. Grey and white tiles are removed.  Next, **3 (white tile area) = 3 (grey tile area).** 3 + 3 = 6, the new tile will be used for the **Floor** location. Grey and white tiles are removed.  Next, **30 (white tile area) = 30 (grey tile area).** 30 + 30 = 60, the new tile will be used for the **Countertop** location. Grey and white tiles are **removed**.  Next, **16 (white tile area) = 16 (grey tile area).** 16 + 16 = 32, the new tile will be used for the **Floor** location. Grey and white tiles are **removed**.  Next, **35 (white tile area) = 35 (grey tile area).** 35 + 35 = 70, the new tile will be used for the **Wall** location. Grey and white tiles are **removed**.  Finally, we have no white and grey tiles left and we've managed to use all of the tiles for the different locations. |
| **20 30 6 10 10**  **10 20 5 6 30** | **White tiles left: none**  **Grey tiles left: none**  **Floor: 3**  **Countertop: 1**  **Sink: 1** | **10 = 10, 10 + 10 = 20** → ****Floor****  10 ≠ 20, 10 / 2 = 5 goes back to the sequence, 20 goes **to the back of** the sequence 5 = 5, 5 + 5 = 10 → **Floor 6 = 6, 6 + 6 = 12** → **Floor 30 = 30, 30 + 30 = 60** → **Countertop** 20 = 20, 20 + 20 = 40→ **Sink** |
| **30 6 10 10**  **10 20 5 6 30 35** | **White tiles left: none**  **Grey tiles left: 35, 20**  **Floor: 3**  **Countertop: 1** | **10 = 10, 10 + 10 = 20** → ****Floor** (both removed)**  10 ≠ 20, 10 / 2 = 5 goes back to the sequence, 20 goes to the back of the sequence 5 = 5, 5 + 5 = 10 → ****Floor** (both removed) 6 = 6, 6 + 6 = 12** → ****Floor** (both removed) 30 = 30, 30 + 30 = 60**→ Countertop **(both removed)** **No white tiles left. 2 grey tiles left with areas of 35 and 20.** |

# Wall Destroyer

You can test your solutions in [Judge](https://judge.softuni.org/Contests/Practice/Index/3525#1)

*Now that Victoria and Vanko have managed to put the tiles in the correct order, it's time to install shelves and hang paintings on one of the walls. In order to do that, Vanko will have to make a few holes in the wall. But he has to be extra careful, because there are cables and steel rods in the wall, that must not be destroyed. You have to guide him, so that he doesn't hit a cable or a rod.*

You will be given an integer **n** for the size of the wall (square shape). On the next **n** lines, you will receive the rows of the wall. Vanko will start at a **random** position, marked with the letter '**V**'. After he makes his **first** **move**, his **initial** position will be **considered a successfully created hole** and you must **mark his starting position with a '\*'**. The **steel rods** and the **cables** will also be on **random** positions. The **rods** will be marked with the letter '**R**' and the **cables** will be marked with '**C**'. All of the **other** positions will be marked with '**-**' (dash).

Until you receive the "**End**" command, on each turn you will be guiding Vanko and telling him the **direction**, which he should move to and make a hole at. The commands will be "**up**", "**down**", "**left**" and "**right**".

If Vanko **manages to create a hole** at the desired location, mark the position with a '**\***'.

If he **hits a rod**, Vanko **returns to his previous** position and **continues** with the **next** directions. Print "**Vanko hit a rod!**" and consider that he did **not** make a **hole**.

If he **hits a cable**, he gets **electrocuted**, the position is marked with an '**E**' and the program **ends**. The **position** that holds the '**E**' letter is **considered a successfully created hole**.

If Vanko lands on a position that **already has a hole on it**, print "**The wall is already destroyed at position [row, col]!**". In case the directions lead Vanko outside of the wall, Vanko **doesn't move at all** and you must **do nothing.**

Keep track of the holes that Vanko manages to create and of the times that he has hit a steel rod.

**The program will end when Vanko gets electrocuted оr the "End" command is given.**

### 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**.
* On each of the next lines, until you receive the "**End**" command, you will get a move command.

### Output

* On the first line:
  + If Vanko manages to make all of the holes, print "**Vanko managed to make {countOfHoles} hole(s) and he hit only {countOfRods} rod(s).**" .
  + If Vanko gets electrocuted, print "**Vanko got electrocuted, but he managed to make** **{countOfHoles} hole(s).**"
* If Vanko lands on a position that already has a hole on it, print "**The wall is already destroyed at position [row, col]!**"
* If Vanko hits a rod, print "**Vanko hit a rod!**".
* At the end, print the **final state** of the matrix (wall) **with Vanko's position on it**.

### Constraints

* The size of the **square** matrix (wall) will be between **[2…10].**
* Vanko's starting position will always be marked with '**V**'.
* There may be cases where the given directions will be outside of the wall.
* There will be always two output scenarios:
  + Vanko manages to make all of the holes, until the "**End**" command;
  + Vanko gets electrocuted.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 5  ---C-  -C-R-  -----  -V---  R----  down  left  right  up  left  up  up  right  down | Vanko hit a rod! The wall is already destroyed at position [3, 1]! Vanko got electrocuted, but he managed to make 6 hole(s).  ---C-  -E-R-  -\*---  -\*\*--  R\*\*-- | The first command is "**down**". Vanko moves to the position, its value is '**-**', so he makes a hole there and increases the counter.  ---C-  -C-R-  -----  -\*---  RV---  **Next command** is "**left**", but there is a rod there, so we only print the corresponding message.  Next command is "**right**" and this position holds a '**-**', so Vanko makes a hole there and we increase the counter.  ---C-  -C-R-  -----  -\*---  R\*V--  The next command is "**up**". Vanko moves to the position and we increase the counter.  Next command us "**left**", but **there's already a hole there**, so we print the corresponding message, without increasing the counter.  The next command is "**up**". Vanko moves to the position and we increase the counter.  Next command is "**up**" again and the position holds a cable, so Vanko gets electrocuted. We increase the counter of the created holes and we print the corresponding message and the final state of the wall, before ending the program. |
| 5  --V--  ---R-  CC---  -----  -----  up  down  right  down  right  right  End | Vanko hit a rod!  Vanko managed to make 5 hole(s) and he hit only 1 rod(s).  --\*--  --\*R-  CC\*\*V  -----  ----- | The first command is "**up**", but that means that Vanko will go out of the wall, so we do nothing.  Next command is "**down**" and we increase the counter.  Next command is "**right**", but there is a rod, so Vanko doesn't change his position and we only print the corresponding message and increase the rod hits counter.  Next command is "**down**" and we increase the counter.  Next command is "**right**" and we increase the counter.  Next command is "**right**" and we increase the counter again.  The last command is "**End**", so we end the program and print the corresponing message and the final state of the wall. |
| 4  ----  C--R  -V--  ----  up  right  End | Vanko managed to make 3 hole(s) and he hit only 0 rod(s).  ----  C\*VR  -\*--  ---- |  |

# Renovators

You can test your solutions in [Judge](https://judge.softuni.org/Contests/Practice/Index/3525#2)

*Despite your efforts and directions, Vanko totally destroyed Victoria's wall. Now she wants to choose another handyman, but this time she'll be more careful. That's why she asked you to help her and create a catalog with different renovators and their info.*

## Preparation

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

**Note**: The target framework of your project must be .NET Core 3.1.

## Problem description

Your task is to create a catalog, containing information for various renovators.

# Renovator

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

* **Name: string**
* **Type: string**
* **Rate: double**
* **Days: int**
* **Hired: boolean - false by default**

The class **constructor** should receive **(name, type, rate, days)**.

The class should also have a method:

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

**"-Renovator: {name}**

**--Specialty: {type}**

**--Rate per day: {rate} BGN"**

# Catalog

Next, a class named **Catalog**is given that has a **collection**(**renovators**) of type **Renovator**. All the entities of the **renovators** collection have the **same** properties. The **Catalog** has also some additional properties:

* **Name: string**
* **NeededRenovators: int**
* **Project: string**

The **constructor** of the **Catalog** class should receive the **name, neededRenovators** and **project.**

Implement the following features:

* Getter **Count** - returns the count of the renovators in the catalog.
* string AddRenovator(Renovator renovator) - **adds** a renovator to the catalog's collection, **if** **renovators are still needed**. Before adding a renovator, check:
  + - * + If the **name** or **specialty** are **null or empty**, return **"Invalid renovator's information.".**
  + If renovators are no more needed, return "**Renovators are no more needed.". Renovators are needed when the count of the added renovators is less than the NeededRenovators property of the Catalog.**
  + If the **rate** is **above 350 BGN**,return **"Invalid renovator's rate.".**
  + Otherwise, return: **"Successfully added {renovatorName} to the catalog.".**
* bool RemoveRenovator(string name) - removes a renovator by **given name.**
  + - If such **exists returns true**;
    - Otherwise, returns **false.**
* int RemoveRenovatorBySpecialty(string type) - removes **all renovators** by the given **specialty.**
  + - If such **exist(s), returns the count of the removed renovators;**
    - Otherwise, returns **0.**
* Renovator HireRenovator(string name) method – **hire** (**set** their available **property** to **true** without removing them from the collection) the **renovator** with the **given name, if they exist**. As a result, **return** the **renovator, or null, if they don't exist.**
* List<Renovator> PayRenovators (int days) method – **return** a **list** with **all renovators** that have been working for **days** days or more.
* **Report()** –returns a string with information about the catalog and renovators who are **not hired** in the following format:

**"**Renovators **available for Project {project}:  
{**Renovator**1}  
{**Renovator**2}  
{…}**"

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

# Constraints

* The **names** of the renovators will be **always unique**.
* You will always have a renovator added before receiving methods manipulating the catalogs' renovators.

# Examples

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

|  |
| --- |
| Sample code usage |
| // Initialize the repository (Catalog)  Catalog catalog = new Catalog("Quality renovators", 5, "Kitchen");  // Initialize entity  Renovator renovator = new Renovator("Gosho", "Painter", 270, 7);  //Print Renovator  Console.WriteLine(renovator);  /\*  -Renovator: Gosho  --Specialty: Painter  --Rate per day: 270 BGN  \*/  // Add Renovator  Console.WriteLine(catalog.AddRenovator(renovator)); // Successfully added Gosho to the catalog.  Console.WriteLine(catalog.Count); // 1  // Remove Renovator  Console.WriteLine(catalog.RemoveRenovator("Pesho")); // False  Renovator secondRenovator = new Renovator("Pesho", "Tiles", 200, 9);  Renovator thirdRenovator = new Renovator("Ivan", "Tiles", 450, 7);  Renovator fourthRenovator = new Renovator("Velichko", "Painter", 120, 3);  Renovator fifthRenovator = new Renovator("Stamat", "Furniture", 300, 4);  Renovator sixthRenovator = new Renovator("Genadi", "Furniture", 80, 15);  Renovator seventhRenovator = new Renovator("Unufri", "Furniture", 80, 15);  // Add Renovators  Console.WriteLine(catalog.AddRenovator(secondRenovator)); // Successfully added Pesho to the catalog.  Console.WriteLine(catalog.AddRenovator(thirdRenovator)); // Invalid renovator's rate.  Console.WriteLine(catalog.AddRenovator(fourthRenovator)); // Successfully added Velichko to the catalog.  Console.WriteLine(catalog.AddRenovator(fifthRenovator)); // Successfully added Stamat to the catalog.  Console.WriteLine(catalog.AddRenovator(sixthRenovator)); // Successfully added Genadi to the catalog.  Console.WriteLine(catalog.AddRenovator(seventhRenovator)); // Renovators are no more needed.  // Hire renovator by name  Console.WriteLine(catalog.HireRenovator("Genadi"));  /\*  -Renovator: Genadi  --Specialty: Furniture  --Rate per day: 80 BGN  \*/  // Pay renovators  List<Renovator> renovators = catalog.PayRenovators(8);  foreach (var renovatorToBePaid in renovators)  {  Console.WriteLine(renovatorToBePaid);  }  /\*  -Renovator: Pesho  --Specialty: Tiles  --Rate per day: 200 BGN  -Renovator: Genadi  --Specialty: Furniture  --Rate per day: 80 BGN  \*/  // Remove renovators by specialty  Console.WriteLine(catalog.RemoveRenovatorBySpecialty("Painter")); // 2  Console.WriteLine("----------------------Report----------------------");  Console.WriteLine(catalog.Report());  /\*  Renovators available for Project Kitchen:  -Renovator: Pesho  --Specialty: Tiles  --Rate per day: 200 BGN  -Renovator: Stamat  --Specialty: Furniture  --Rate per day: 300 BGN  \*/ |

# Submission

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