**Exercises: Associative Arrays**

Problems for exercises and homework for the ["Technology Fundamentals" course @ SoftUni](https://softuni.bg/courses/technology-fundamentals).

You can check your solutions in [Judge](https://judge.softuni.bg/Contests/1220/).

* **Gold Miner**

You are given a sequence of strings, each on a new line. Every odd line on the console is representing the type of the gold (e.g. Yellow, Rose, White, and so on), and its karats. Your task is to collect the resources while you receive **"stop"** аnd print each on a new line.

**Print the resources and their quantities in format:**

**"{type} –> {karats}K"**

The karats inputs will be in the range [1 … 24 000]

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| White  18  Yellow  24  Green  10  stop | White -> 18K  Yellow -> 24K  Green -> 10K |
| Rose  14  White  24  Yellow  10  Rose  10  stop | Rose -> 24K  White -> 24K  Yellow -> 10K |

* **Student Academy**

Write a program, which keeps information about **students** and **their grades**.

You will receive **n pair of rows**. First you will receive the **student's name**, after that you will receive **his grade**. **Check if student already exists, and if not, add him**. Keep track of all grades for each student.

When you finish reading data, keep students with **average grade higher or equal to 4.50**. Order filtered students by **average grade in descending**.

**Print the students and their average grade in format:**

**"{name} –> {averageGrade}"**

**Format** the average grade to the **2nd decimal place**.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  John  5.5  John  4.5  Alice  6  Alice  3  George  5 | John -> 5.00  George -> 5.00  Alice -> 4.50 |
| 5  Amanda  3.5  Amanda  4  Rob  5.5  Christian  5  Robert  6 | Robert -> 6.00  Rob -> 5.50  Christian -> 5.00 |

* **Courses**

Write a program, which keeps information about **courses**. Each course has a name and registered students.

You will receive **course** **name** and **student** **name,** until you receive the command "**end**". **Check if such course already exists, and if not, add the course.** Register the user into the course. When you do receive the command "**end**", print the courses with their **names** and **total registered users**, ordered by the **count of registered users** in **descending** order. For each contest print registered users **ordered by name in ascending order**.

**Input**

* Until you receive "**end**", the input come in the format: "**{courseName} : {studentName}**".

**Output**

* Print information about **each** **course**, following the format:   
  **"{courseName}: {registeredStudents}"**
* Print information about each student, following the format:  
  **"-- {studentName}"**

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| Programming Fundamentals : John Smith  Programming Fundamentals : Linda Johnson  JS Core : Will Wilson  Java Advanced : Harrison White  end | Programming Fundamentals: 2  -- John Smith  -- Linda Johnson  JS Core: 1  -- Will Wilson  Java Advanced: 1  -- Harrison White |
| Algorithms : Jay Moore  Programming Basics : Martin Taylor  Python Fundamentals : John Anderson  Python Fundamentals : Andrew Robinson  Algorithms : Bob Jackson  Python Fundamentals : Clark Lewis  end | Python Fundamentals: 3  -- Andrew Robinson  -- Clark Lewis  -- John Anderson  Algorithms: 2  -- Bob Jackson  -- Jay Moore  Programming Basics: 1  -- Martin Taylor |

* **Playing Cards**

You are given a sequence of people and for every person what cards he draws from the deck. The input will be separate lines in the format:

* **{personName}: {PT, PT, PT,… PT}**

Where **P (2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A**) is the power of the card and **T (S, H, D, C)** is the type. The input ends when a **"JOKER"** is drawn. The name can contain any ASCII symbol except **':'**. The input will always be valid and in the format described, there is no need to check it.

A single person **cannot have more than one** card with the **same power and type**, if they draw such a card they **discard** it. The people are playing with **multiple decks**. Each card has a value that is calculated by the **power multiplied by the type**. Powers **2 to 10** have the same value and **J to A** are **11 to 14**. Types are mapped to multipliers the following way (**S -> 4, H-> 3, D -> 2, C -> 1**).

Finally print out the total value each player has in his hand in the format:

* **{personName}: {value}**

**Examples**

**9+40+11+26+20 = 106  
18+28+13+10 = 69**

|  |  |
| --- | --- |
| **Input** | **Output** |
| Pesho: 2C, 4H, 9H, AS, QS  Slav: 3H, 10S, JC, KD, 5S, 10S  Peshoslav: QH, QC, QS, QD  Slav: 6H, 7S, KC, KD, 5S, 10C  Peshoslav: QH, QC, JS, JD, JC  Pesho: JD, JD, JD, JD, JD, JD  JOKER | Pesho: 167  Slav: 175  Peshoslav: 197 |

* **SoftUni Parking**

SoftUni just got a new **parking lot**. It’s so fancy, it even has online **parking validation**. Except, the online service doesn’t work. It can only receive users’ data, but doesn’t know what to do with it. Good thing you’re on the dev team and know how to fix it, right?

Write a program, which validates parking for an online service. Users can **register** to park and **unregister** to leave.

The program **receives 2 commands**:

* "**register {username} {licensePlateNumber}**":
* The system only supports **one car per user** at the moment, so if a user tries to register **another license plate**, using the **same username**, the system should print:  
  "**ERROR: already registered with plate number {licensePlateNumber}**"
* If the aforementioned checks pass successfully, the plate can be registered, so the system should print:  
   **"{username} registered {licensePlateNumber} successfully"**
* "**unregister {username}**":
* If the user is **not present** in the database, the system should print:  
  "**ERROR: user {username} not found**"
* If the aforementioned check passes successfully, the system should print:  
  "**{username} unregistered successfully**"

After you execute all of the commands, **print** all the currently **registered users** and their **license plates** in the format:

* "**{username} => {licensePlateNumber}**"

**Input**

* First line: **n** – **number of commands** – **integer**
* Next **n** lines: **commands** in one of **two** possible formats:
* Register: "**register {username} {licensePlateNumber}**"
* Unregister: "**unregister {username}**"

The input will **always** be **valid** and you **do not need** to check it explicitly.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  register John CS1234JS  register George JAVA123S  register Andy AB4142CD  register Jesica VR1223EE  unregister Andy | John registered CS1234JS successfully  George registered JAVA123S successfully  Andy registered AB4142CD successfully  Jesica registered VR1223EE successfully  Andy unregistered successfully  John => CS1234JS  George => JAVA123S  Jesica => VR1223EE |
| 4  register Jony AA4132BB  register Jony AA4132BB  register Linda AA9999BB  unregister Jony | Jony registered AA4132BB successfully  ERROR: already registered with plate number AA4132BB  Linda registered AA9999BB successfully  Jony unregistered successfully  Linda => AA9999BB |
| 6  register Jacob MM1111XX  register Anthony AB1111XX  unregister Jacob  register Joshua DD1111XX  unregister Lily  register Samantha AA9999BB | Jacob registered MM1111XX successfully  Anthony registered AB1111XX successfully  Jacob unregistered successfully  Joshua registered DD1111XX successfully  ERROR: user Lily not found  Samantha registered AA9999BB successfully  Anthony => AB1111XX  Joshua => DD1111XX  Samantha => AA9999BB |

* **Company Users**

Write a program which keeps information about companies and their employees.

You will receive **company** **name** and **employee's id,** until you receive the command "**End**". Add each employee to the given company. Keep in mind that a company cannot have two employees with the same id.

When you finish reading data, **order the companies by the name in ascending order**.

Print the company name and each employee's id in the following format:

**{companyName}**

**-- {id1}**

**-- {id2}**

**-- {idN}**

**Input / Constraints**

* Until you receive "**End**", the input come in the format: "**{companyName} -> {employeeId}**".
* The input always will be valid.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| SoftUni -> AA12345  SoftUni -> BB12345  Microsoft -> CC12345  HP -> BB12345  End | HP  -- BB12345  Microsoft  -- CC12345  SoftUni  -- AA12345  -- BB12345 |
| SoftUni -> AA12345  SoftUni -> CC12344  Lenovo -> XX23456  SoftUni -> AA12345  Movement -> DD11111  End | Lenovo  -- XX23456  Movement  -- DD11111  SoftUni  -- AA12345  -- CC12344 |

* **Legendary Farming**

You’ve beaten all the content and the last thing left to accomplish is own a legendary item. However, it’s a tedious process and requires quite a bit of farming. Anyway, you are not too pretentious – any legendary will do.  
 The possible items are:

* **Shadowmourne** – requires **250 Shards**;
* **Valanyr** – requires **250 Fragments**;
* **Dragonwrath** – requires **250 Motes**;

**Shards, Fragments** and **Motes** are the **key materials**, all else is **junk.** You will be given lines of input, such as   
**2 motes 3 ores 15 stones.** Keep track of the **key materials -** the **first** that reaches the **250 mark** **wins** the **race**. At that point, print the corresponding legendary obtained. Then, print the **remaining** shards, fragments, motes, ordered by **quantity** in **descending** order, then by **name** in **ascending** order, each on a new line. Finally, print the collected **junk** items, in **alphabetical** order.

**Input**

* Each line of input is in format **{quantity} {material} {quantity} {material} … {quantity} {material}**

**Output**

* On the first line, print the obtained item in format **{Legendary item} obtained!**
* On the next three lines, print the remaining key materials in descending order by quantity
* All materials are printed in format **{material}: {quantity}**
* If two key materials have the same quantity, print them in alphabetical order
* On the final several lines, print the junk items in alphabetical order
* All materials are printed in format **{material}: {quantity}**
* All output should be **lowercase**, except the first letter of the legendary

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 Motes 5 stones 5 Shards  6 leathers 255 fragments 7 Shards | Valanyr obtained!  fragments: 5  shards: 5  motes: 3  leathers: 6  stones: 5 |
| 123 silver 6 shards 8 shards 5 motes  9 fangs 75 motes 103 MOTES 8 Shards  86 Motes 7 stones 19 silver | Dragonwrath obtained!  shards: 22  motes: 19  fragments: 0  fangs: 9  silver: 123 |

* **Orders**

Write a program, which keeps information about **products** and their **prices**. Each product has a **name**, a **price** and its **quantity**. If the product **doesn’t exist** yet, **add** it with its **starting quantity**.

If you receive a product, which **already exists** **increase** its quantity by the input quantity and if its **price** is different, **replace** the price as well.

You will receive products **names**, **prices** and **quantities** on **new lines**.Until you receive the command "**buy**", keep adding items. When you do receive the command "**buy**", print the items with their **names** and **total price** of all the products with that name.

**Input**

* Until you receive "**buy**", the products come in the format: "**{name} {price} {quantity}**".
* The product data is **always** delimited by a **single space**.

**Output**

* Print information about **each** **product**, following the format:   
  **"{productName} -> {totalPrice}"**
* **Format** the average grade to the **2nd decimal place**.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| Beer 2.20 100  IceTea 1.50 50  NukaCola 3.30 80  Water 1.00 500  buy | Beer -> 220.00  IceTea -> 75.00  NukaCola -> 264.00  Water -> 500.00 |
| Beer 2.40 350  Water 1.25 200  IceTea 5.20 100  Beer 1.20 200  IceTea 0.50 120  buy | Beer -> 660.00  Water -> 250.00  IceTea -> 110.00 |
| CesarSalad 10.20 25  SuperEnergy 0.80 400  Beer 1.35 350  IceCream 1.50 25  buy | CesarSalad -> 255.00  SuperEnergy -> 320.00  Beer -> 472.50  IceCream -> 37.50 |

* **\*ForceBook**

The force users are struggling to remember which side are the different forceUsers from, because they switch them too often. So you are tasked to create a web application to manage their profiles. You should store information for every **unique forceUser**, registered in the application.

You will receive **several input lines** in one of the following formats:

**{forceSide} | {forceUser}**

**{forceUser} -> {forceSide}**

The **forceUser and forceSide** are strings, containing any character.

If you receive **forceSide | forceUser** you should **check if such forceUser already exists**, and **if not**, **add** him/her to the corresponding side.

If you receive a **forceUser -> forceSide** you should check if there is such **forceUser** already and if so, **change his/her side**. If there is no such **forceUser**, add him/her to the corresponding forceSide, treating the command **as new registered forceUser.**  
**Then you should print on the console: "{forceUser} joins the {forceSide} side!"**

You should end your program when you receive the command **"Lumpawaroo"**. At that point you should print each force side, **ordered descending by forceUsers count, than ordered by name**. For each side print the **forceUsers**, **ordered by name**.

In case there are **no forceUsers in a side**, you **shouldn`t print** the side information.

**Input / Constraints**

* The input comes in the form of commands in one of the formats specified above.
* The input ends when you receive the command **"Lumpawaroo"**.

**Output**

* As output for each forceSide, **ordered descending by forceUsers count**, **then by name**, you must print all the forceUsers, **ordered by name alphabetically**.
* The output format is:

**Side: {forceSide}, Members: {forceUsers.Count}**

**! {forceUser}**

**! {forceUser}**

**! {forceUser}**

* In case there are **NO** **forceUsers**, don`t print this side.

**Examples**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| Light | Gosho  Dark | Pesho Chronos | Raziel  Lumpawaroo | Side: Light, Members: 2  ! Gosho ! Raziel  Side: Dark, Members: 1  ! Pesho | We register Gosho in the Light side and Pesho in the Dark side. After receiving "Lumpawaroo" we print both sides, ordered by membersCount and then by name. |
| Lighter | Royal  Darker | DCay  Ivan Ivanov -> Lighter  DCay -> Lighter  Lumpawaroo | Ivan Ivanov joins the Lighter side!  DCay joins the Lighter side!  Side: Lighter, Members: 3  ! DCay  ! Ivan Ivanov  ! Royal | Although Ivan Ivanov doesn`t have profile, we **register** him and add him to the Lighter side.  We **remove DCay** from Darker side and add him to Lighter side.  We print only Lighter side because Darker side **has no members.** |

* **SoftUni Exam Results**

Judge statistics on the last Programing Fundamentals exam was not working correctly, so you have the task to take all the submissions and analyze them properly. You should collect all the submission and print the final results and statistics about each language that the participants submitted their solutions in.

You will be receiving lines in the following format: **"{username}-{language}-{points}" until you receive "exam finished"**.You should store each username and his submissions and points.   
You can receive a **command to ban** a user for cheating in the following format: **"{username}-banned".** In that case, you should **remove** the user from the contest, but **preserve his submissions in the total count of submissions for each language**.

After receiving **"exam finished"** print each of the participants, ordered descending by their max points, then by username, in the following format:

**Results:**

**{username} | {points}**

**…**

After that print each language, used in the exam, ordered descending by total submission count and then by language name, in the following format:

**Submissions:**

**{language} – {submissionsCount}**

**…**

**Input / Constraints**

Until you receive "exam finished" you will be receiving participant submissions in the following format: **"{username}-{language}-{points}"**.

You can receive a ban command -> **"{username}-banned"**

The points of the participant will always be a **valid integer in the range [0-100];**

**Output**

* Print the exam results for each participant, ordered descending by max points and then by username, in the following format:

**Results:**

**{username} | {points}**

**…**

* After that print each language, ordered descending by total submissions and then by language name, in the following format:

**Submissions:**

**{language} – {submissionsCount}**

**…**

* Allowed working **time** / **memory**: **100ms** / **16MB**.

**Examples**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| Pesho-Java-84  Gosho-C#-70  Gosho-C#-84  Kiro-C#-94  exam finished | Results:  Kiro | 94  Gosho | 84  Pesho | 84  Submissions:  C# - 3  Java - 1 | When receiving Gosho for second time, we keep the better result.  We order the participant descending by max points and then by name, printing only the username and the max points.  After that we print each language along with the count of submissions, ordered descending by submissions count, and then by language name. |
| Pesho-Java-91  Gosho-C#-84  Kiro-JavaScript-90  Kiro-C#-50  Kiro-banned  exam finished | Results:  Pesho | 91  Gosho | 84  Submissions:  C# - 2  Java - 1  JavaScript - 1 | Kiro is banned so he is removed from the contest, but he`s submissions are still preserved in the languages submissions count.  So althou there are only 2 participants in the results, there are 4 submissions in total. |