# More Exercises: Associative Arrays, Lambda and Stream API

You can check your solutions in [Judge.](https://judge.softuni.org/Contests/1313)

## Ranking

Here comes the final and the most interesting part - the Final ranking of the candidate interns. The final ranking is determined by the points of the interview tasks and from the exams in SoftUni. Here is your final task.

You will receive some lines of input in the format "**{contest}:{password for contest}**" until you receive "**end of contests**". Save that data because **you will need it later**.

After that, you will receive another type of input in the format "**{contest}=>{password}=>{username}=>{points}**" until you receive "**end of submissions**".

Here is what you need to do:

* Check if the **contest is valid (if you received it in the first type of input);**
* Check if the **password is correct for the given contest;**
* Save the user with the contest they take part in **(a user can take part in many contests)** and the points the user has in the given contest. If you receive the **same contest and the same user, update the points only if the new ones are more than the older ones.**

In the end, you have to print the info for the user with the **most points** in the format **"Best candidate is {user} with total {total points} points."**.

After that, print **all students ordered by their names**. **For each user, print each contest with the points in descending order**. See the examples.

### Input

* Strings in format "**{contest}:{password for contest}**"until the "**end of contests**"command. There will be no case with two equal contests.
* Strings in format "**{contest}=>{password}=>{username}=>{points}**"until the "**end of submissions**"command.
* **There will be no case with 2 or more users with the same total points!**

### Output

* On the first line, print the best user in a format:  
  "**Best candidate is {user} with total {total points} points.**"**.**
* Then print all students ordered as mentioned above in format:

"**{user1 name}**

**# {contest1} -> {points}**

**# {contest2} -> {points}**"

### Constraints

* The strings may contain any ASCII character except **(:, =, >).**
* The numbers will be in the range **[0 - 10000].**
* The second input is always valid.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Part One Interview:success  Js Fundamentals:Pesho  C# Fundamentals:fundPass  Algorithms:fun  end of contests  C# Fundamentals=>fundPass=>Tanya=>350  Algorithms=>fun=>Tanya=>380  Part One Interview=>success=>Nikola=>120  Java Basics Exam=>pesho=>Petkan=>400  Part One Interview=>success=>Tanya=>220  OOP Advanced=>password123=>BaiIvan=>231  C# Fundamentals=>fundPass=>Tanya=>250  C# Fundamentals=>fundPass=>Nikola=>200  Js Fundamentals=>Pesho=>Tanya=>400  end of submissions | Best candidate is Tanya with total 1350 points.  Ranking:  Nikola  # C# Fundamentals -> 200  # Part One Interview -> 120  Tanya  # Js Fundamentals -> 400  # Algorithms -> 380  # C# Fundamentals -> 350  # Part One Interview -> 220 |
| Java Advanced:funpass  Part Two Interview:success  Math Concept:asdasd  Java Web Basics:forrF  end of contests  Math Concept=>ispass=>Monika=>290  Java Advanced=>funpass=>Simona=>400  Part Two Interview=>success=>Drago=>120  Java Advanced=>funpass=>Petyr=>90  Java Web Basics=>forrF=>Simona=>280  Part Two Interview=>success=>Petyr=>0  Math Concept=>asdasd=>Drago=>250  Part Two Interview=>success=>Simona=>200  end of submissions | Best candidate is Simona with total 880 points.  Ranking:  Drago  # Math Concept -> 250  # Part Two Interview -> 120  Petyr  # Java Advanced -> 90  # Part Two Interview -> 0  Simona  # Java Advanced -> 400  # Java Web Basics -> 280  # Part Two Interview -> 200 |

## Judge

You know the Judge system, right?! Your job is to create a program similar to the Judge system.

You will receive **several input lines** in the following format:

"{username} -> {contest} -> {points}"

The constestName and username are strings. The given **points** will be an integer number. You need to keep track of **every contest** and **individual statistics of every user**. You should **check if such a contest already exists**, and if not, add it, otherwise, **check if the current user is participating in the contest**. If he is participating, **take the higher score**, otherwise, just **add it**.

Also, you need to keep **individual statistics** for each **user -** the **total points of all contests**.

You should end your program when you receive the command "no more time". At that point, you should print each contest in **order of input**. For each contest, print the participants **ordered by points in descending order and then by name in ascending order**. After that, you should print **individual statistics for every participant** ordered by **total points** in **descending** **order** and **then by alphabetical order**.

### Input / Constraints

* The input comes in the form of commands in the format specified above.
* Username and contest name **always will be one word.**
* Points will be an **integer** in the **range [0, 1000].**
* There will be **no invalid** input lines.
* If **all sorting criteria fail**, the order should be by **order** of **input.**
* The input ends when you receive the command "no more time".

### Output

* The output format for the contests is:

"{constestName}: {participants.Count} participants

{position}. {username} <::> {points}"

* After you print all contests, print the **individual statistics for every participant.**
* The output format is:

"Individual standings:

{position}. {username} -> {totalPoints}"

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Peter -> Algo -> 400  George -> Algo -> 300  Sam -> Algo -> 200  Peter -> DS -> 150  Maria -> DS -> 600  no more time | Algo: 3 participants  1. Peter <::> 400  2. George <::> 300  3. Sam <::> 200  DS: 2 participants  1. Maria <::> 600  2. Peter <::> 150  Individual standings:  1. Maria -> 600  2. Peter -> 550  3. George -> 300  4. Sam -> 200 |
| Peter -> OOP -> 350  George -> OOP -> 250  Sam -> Advanced -> 600  George -> OOP -> 300  Pitter -> OOP -> 300  Pitter -> Advanced -> 250  Anna -> JSCore -> 400  no more time | OOP: 3 participants  1. Peter <::> 350  2. George <::> 300  3. Pitter <::> 300  Advanced: 2 participants  1. Sam <::> 600  2. Pitter <::> 250  JSCore: 1 participants  1. Anna <::> 400  Individual standings:  1. Sam -> 600  2. Pitter -> 550  3. Anna -> 400  4. Peter -> 350  5. George -> 300 |

## MOBA Challenger

Peter is a pro MOBA player. He is struggling to become a master of the Challenger tier. So he watches the statistics in the tier carefully.

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

"{player} -> {position} -> {skill}"

"{player} vs {player}"

The player and position are strings. The given **skill** will be an integer number. You need to keep track of **every player**.

When you receive a **player and his position and skill**, add him to the player pool. If he isn't present, **else add** his position and skill **or update** his skill, **only if** the current position skill is lower than the new value.

If you receive "**{player} vs {player}**" and **both players exist** in the tier, **they duel** with the following rules:

Compare their positions, **if they got at least one in common**, the player with better **total skill points** wins, and the other is **demoted** from the tier -> remove him. If they have the same total skill points, **the duel is a tie,** and they both continue in the Season.

If they don't have positions in common, **the duel isn't happening,** and both continue in the Season.

You should end your program when you receive the command "**Season end**". At that point, you should print the players, **ordered by total skill in descending order, then ordered by player name in ascending order**. **For each** player print, their position and skill are **ordered descending by skill and then by position name in ascending order.**

### Input / Constraints

* The input comes in the form of commands in one of the formats specified above.
* Player and position **will always be one-word string, containing no whitespaces.**
* Skill will be an **integer** in the **range [0, 1000].**
* There will be **no invalid** input lines.
* The program ends when you receive the command "Season end".

### Output

* The output format for each player is:

"{player}: {totalSkill} skill

- {position} <::> {skill}"

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| Peter -> Adc -> 400  George -> Jungle -> 300  Sam -> Mid -> 200  Sam -> Support -> 250  Season end | Sam: 450 skill  - Support <::> 250  - Mid <::> 200  Peter: 400 skill  - Adc <::> 400  George: 300 skill  - Jungle <::> 300 | We order the players by total skill points descending, then by name. We print every position along its skill ordered descending by skill, then by position name. |
| Peter -> Adc -> 400  Bush -> Tank -> 150  Faker -> Mid -> 200  Faker -> Support -> 250  Faker -> Tank -> 250  Peter vs Faker  Faker vs Bush  Faker vs Hide  Season end | Faker: 700 skill  - Support <::> 250  - Tank <::> 250  - Mid <::> 200  Peter: 400 skill  - Adc <::> 400 | Faker and Peter don't have a common position, so the duel isn't valid.  Faker wins vs Bush /common position: "Tank". Bush is demoted.  Hide doesn't exist, so the duel isn't valid.  We print every player left in the tier. |

## Snowwhite

Snow White loves her dwarfs, but there are so many, and she doesn't know how to order them. Does she order them by name? Or by the color of their hat? Or by physics? She can't decide, so it's up to you to write a program that does it for her.

You will be receiving **several input lines** which contain **data** about **dwarfs** in the following format:

"{dwarfName} <:> {dwarfHatColor} <:> {dwarfPhysics}"

The dwarfName and the dwarfHatColor are **strings**. The dwarfPhysics is an **integer**.

You must **store** the **dwarfs** in your program. There are several rules, though:

* If **2 dwarfs** have the **same name** but **different hat colors**, they should be **considered different dwarfs**, and you should store **both** of them.
* If **2 dwarfs** have the **same name** and the **same hat color**, **store** the **one** with the **higher physics**.

When you receive the command "Once upon a time", the input ends. You must **order** the **dwarfs** by **physics** in **descending order** and thenby the **total** **count** of **dwarfs** with the **same hat color** in **descending order**.   
Then you must print them all.

### Input

* The input will consist of **several input lines** containing **dwarf data** in the format specified above.
* The input **ends** when you receive the command "Once upon a time**".**

### Output

* As output, you must print the **dwarfs** **ordered** as specified above.
* The output format is: "({hatColor}) {name} <-> {physics}".

### Constraints

* The dwarfName will be a **string** that may contain **any ASCII** character except ' ' (space), '<', ':', '>'.
* The dwarfHatColor will be a **string** which may contain **any ASCII** character except ' ' (space), '<', ':', '>'.
* The dwarfPhysics will be an **integer** in the **range [0, 231 - 1].**
* There will be **no invalid** input lines.
* If **all sorting criteria fail**, the order should be by **order** of **input.**
* It allowed working **time**/**memory**: **100ms**/**16MB.**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Peter <:> Red <:> 2000  Todor <:> Blue <:> 1000  George <:> Green <:> 1000  Smith <:> Yellow <:> 4500  Premis <:> Sam <:> 1000  Once upon a time | (Yellow) Smith <-> 4500  (Red) Peter <-> 2000  (Blue) Todor <-> 1000  (Green) George <-> 1000  (Sam) Premis <-> 1000 |
| Peter <:> Red <:> 5000  Peter <:> Blue <:> 10000  Peter <:> Red <:> 10000  George <:> Blue <:> 10000  Once upon a time | (Blue) Peter <-> 10000  (Blue) George <-> 10000  (Red) Peter <-> 10000 |

## Dragon Army

Heroes III is the best game ever. Everyone loves it, and everyone plays it all the time. John is not an exclusion from this rule. His favorite units in the game are all **types** of dragons - black, red, gold, azure, etc. He likes them so much that he gives them **names** and keeps logs of their **stats**: **damage, health,** and **armor**. The process of aggregating all the data is quite tedious, so he would like to have a program doing it. Since he is no programmer, it's your task to help him.

You need to categorize dragons by their **type**. For each dragon, identified by **name,** keep information about his **stats.** Type is **preserved** as in the input order, but dragons are **sorted** alphabetically by name. For each type, you should also print the average **damage**, **health,** and **armor** of the dragons. For each dragon, print his stats.

There **may** be **missing** stats in the input, though. If a stat is missing, you should assign its default values. Default values are as follows: health **250**, damage **45,** and armor **10**. Missing stat will be marked by **null**.

The input is in the following format "**{type} {name} {damage} {health} {armor}**"**.** Any of the integers may be assigned a null value. See the examples below for a better understanding of your task.

If the same dragon is added a second time, the new stats should **overwrite** the previous ones. Two dragons are considered **equal** if they match by **both** name and type.

### Input

* On the first line, you are given the number **N** - the number of dragons to follow.
* On the next **N** lines, you are given input in the format described above. There will be a **single space** separating each element.

### Output

* Print the aggregated data on the console.
* For each type print average stats of its dragons in format: "**{Type}::({damage}/{health}/{armor})**"**.**
* Damage, health, and armor should be **rounded** to two digits after the decimal separator.
* For each dragon, print its stats in the format:  
  "**-{Name} -> damage: {damage}, health: {health}, armor: {armor}**"

### Constraints

* **N** is in the range **[1…100].**
* The dragon type and name are one word only, starting with a capital letter.
* Damage, health, and armor are integers in the range **[0 … 100000]** or **null.**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  Red Bazgargal 100 2500 25  Black Dargonax 200 3500 18  Red Obsidion 220 2200 35  Blue Kerizsa 60 2100 20  Blue Algordox 65 1800 50 | Red::(160.00/2350.00/30.00)  -Bazgargal -> damage: 100, health: 2500, armor: 25  -Obsidion -> damage: 220, health: 2200, armor: 35  Black::(200.00/3500.00/18.00)  -Dargonax -> damage: 200, health: 3500, armor: 18  Blue::(62.50/1950.00/35.00)  -Algordox -> damage: 65, health: 1800, armor: 50  -Kerizsa -> damage: 60, health: 2100, armor: 20 |
| 4  Gold Zzazx null 1000 10  Gold Traxx 500 null 0  Gold Xaarxx 250 1000 null  Gold Ardrax 100 1055 50 | Gold::(223.75/826.25/17.50)  -Ardrax -> damage: 100, health: 1055, armor: 50  -Traxx -> damage: 500, health: 250, armor: 0  -Xaarxx -> damage: 250, health: 1000, armor: 10  -Zzazx -> damage: 45, health: 1000, armor: 10 |