# Exercise: Dictionaries & LINQ

Problems for exercises and homework for the [“Programming Fundamentals Extended” course @ SoftUni](https://softuni.bg/courses/programming-fundamentals).

# Anonymous Cache

The Anonymous are storing data on their dataservers about their activities. The CIA has higher the greatest hacker in the world – You. Your job is to extract their data and send it to the CIA. It won’t be an easy task, Get Ready!

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

* {dataSet}
* {dataKey} -> {dataSize} | {dataSet}

The dataSet and dataKey are both strings. The dataSize is an **integer**. The dataSets hold dataKeys and their dataSizes.

If you receive only a dataSet you should **add** it. If you receive a dataKey and a dataSize, you should add them to the **given** dataSet.

And here’s where the fun begins. If you receive a dataKey and a dataSize, but the given dataSet **does NOT exist**, you should **STORE** those **keys** and **values** in a cache. When the corresponding dataSet is **added**, you should **check** if the cache holds any **keys** and **values** referenced to it, and you should **add** them to the dataSet.

You should end your program when you receive the command “thetinggoesskrra”. At that point you should extract the dataSet from the data with the **HIGHEST** dataSize (**SUM** of all its dataSizes), and you should print it.

**NOTE**: Elements in the cache, **should be CONSIDERED NON-EXISTANT**. You should **NOT** count them in the **final output**.

In case there are **NO** dataSets in the data, you **should** **NOT** **do anything**.

### Input

* The input comes in the form of commands in one of the formats specified above.
* The input ends when you receive the command “thetinggoesskrra”.

### Output

* As output you must print the dataSet with the **HIGHEST** **SUM** of all dataSizes.
* The output format is:

Data Set: {dataSet}, Total Size: {sumOfAllDataSizes}

$.{dataKey1}

$.{dataKey2}

...

* In case there are **NO** dataSets in the data, print **nothing**.

### Constrains

* The dataSet and dataKey are **both strings** which may contain **ANY ASCII** character except ‘ ’, ‘-’, ‘>’, ‘|’.
* The dataSize is a **valid integer** in **range [0, 1.000.000.000]**.
* There will be **NO invalid input lines**.
* There will be **NO** dataSets with **EQUAL SUMMED** dataSize.
* There will be **NO DUPLICATE** keys.
* Allowed working **time/memory**: **100ms / 16MB**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Users  BankAccounts  ADDB444 -> 23111 | BankAccounts  Students -> 2000 | Users  Workers -> 24233 | Users  thetinggoesskrra | Data Set: Users, Total Size: 26233  $.Students  $.Workers |
| Cars  Car1 -> 233333 | Cars  Car23 -> 266666 | Cars  Warehouse2 -> 10000 | Buildings  Warehouse3 -> 480000 | Buildings  Warehouse5 -> 100000 | Buildings  Buildings  thetinggoesskrra | Data Set: Buildings, Total Size: 590000  $.Warehouse2  $.Warehouse3  $.Warehouse5 |

# Hornet Armada

The Hornet Overlord Nostalgia, who is famed for his absolute discipline and strict orders, owns the most sorted army in the Hornet history. Help Nostalgia “computerize” the process of sorting out his army.

You will be given **N** – an integer.  
On the next **N** lines you will be given input containing information about soldiers in the following format:

**{lastActivity} = {legionName} -> {soldierType}:{soldierCount}**

The **last activity** is an **integer**. The **legion name** and **soldier type**, will both be **strings**. The **soldier count** will be an **integer**. You must **store** **every** **legion** with its **activity**, and **every** **soldier type** with its **count**, in its **legion**.

If a **given legion already exists**, you must **add** the new **soldier type**, with its count. If the soldier type exists **ALSO**, you should just **add** the **soldier count**.

**IN** **BOTH** cases, stated above, you should **update** the **last** **activity**, with the newly entered one, **ONLY** if the **entered** **one** is **GREATER** than the **previous one**.

After you’ve read **all N** input lines, you will receive a line in one of the following formats:

* {activity}\{soldierType}
* {soldierType}

In the **first case,** you must print all **legions**, and the **count of soldiers** they have from the **given** **soldier** **type**, who’s **last activity** is **LOWER** than the **given activity**. The legions must be printed in **descending order** by **soldier count**.

In the **second case**, you must print all legions which **have** the **given soldier type**, with **last activity**, and **legion name**. The legions must be printed in **descending** **order** oftheir **activity**.

### Input

* On the first line you will receive **N** –the **integer**.
* On the next **N** lines you will receive data about **soldiers** and **legions**.
* On the last line you will receive **one** of the **two commands**, which will **determine** the **output**.

### Output

* If you are given the **last activity** and **soldier type** on the last command, you must print the legions in this format:
* {legionName} -> {soldierCount}
* If you are given **only** the **soldier type** on the last command, you must print the legions in this format:
* {lastActivity} : {legionName}

### Constrains

* The first integer – **N**, will be in **range [0; 10,000]**.
* The **legion names** and **soldier types** may consist of **any ASCII** character, except “**=**”, “**-**”, “**>**”, “**:**”, “ ”(**space**).
* The **soldier count** and **last activity** will be integers in **range [0; 1,000,000,000]**.
* All input data will be exactly as stated above. There will be **NO invalid** input lines.
* Data which has **NO specified order** must be sorted in **order of** **input**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 6  1 = BlackBeatles -> Soldier:2000  2 = BlackBeatles -> Worker:1000  1 = Red\_Ones -> Soldier:10000  5 = Rm -> Soldier:30000  2 = Red\_Ones -> Soldier:20000  10 = RND -> Soldier:100000  10\Soldier | Red\_Ones -> 30000  Rm -> 30000  BlackBeatles -> 2000 |
| 7  1000 = F1rstL3gion -> Aisers:15000  500 = F1rstL3gion -> Aisers:1000  200 = F1rstL3gion -> Guards:2000  2000 = Second!egion -> Guards:2000  1500 = Second!egion -> Aisers:15000  2500 = Second!egion -> Spies:2000  1000 = Forked\_Ones -> Guards:10000000  Guards | 2500 : Second!egion  1000 : F1rstL3gion  1000 : Forked\_Ones |

# CODE: Phoenix Oscar Romeo November

The fire creatures are assembling in squads to fight The Evil Phoenix God. You have been tasked to determine which squad is the strongest, so it will be sent as The Vanguard.

You will begin receiving input lines containing information about fire creatures in the following format:

{creature} -> {squadMate}

The creature and the squadMate are **strings**. You should store every **creature**, and his **squad mates**. If the **creature** already **exists**, you should **add** the **new squad mate** to it.

* If there is **already** a **squad mate** with the **given name** in the **given creature’s squad**, **IGNORE** that **line** of **input**.
* If the **given squad mate name** is the **same** as the **given** **creature**, **IGNORE** that **line** of **input**.

The **input sequence ends** when you receive the command “Blaze it!”.

When that happens you must **print** the **creatures ordered** in **descending** order by **count** of **squad mates**. Sounds simple right? But there is one little **DETAIL**.

If a particular creature has a squadMate, and that squadMate has that creature in his squadMates, you **should NOT consider** them as **part** of the **count** of **squad mates**.

**Example**:

Creature 1: **Mozilla** -> {Tony, Dony, Mony}

Creature 2: **Tony** -> {Mozilla, Franzilla, Godzilla}

**Mozilla** has **2 squad mates** in total, because **Tony** also has **Mozilla** in his **squad mates**.

**Tony** has **2 squad mates** in total, because **Mozilla** also has **Tony** in his **squad mates**.

### Input

* As input you will receive several input lines containing information about the fire creatures.
* The input sequence ends when you receive the command “Blaze it!”.

### Output

* As output you must print each of the creatures the following information:
  + {creature} : {countOfSquadMates}
* As it was stated above, mind the **count** of **squad mates**. If **2 creatures** have themselves in their **squad mates**, they should **NOT** be **counted**.

### Constrains

* The creature and the squadMate will be **strings** which may contain **any ASCII character**.
* There will be **NO invalid** input lines.
* Allowed time / memory: **100ms / 16MB**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Mozilla -> Tony  Tony -> Godzilla  Mozilla -> Dony  Tony -> Franzilla  Mozilla -> Mony  Tony -> Mozilla  Blaze it! | Mozilla : 2  Tony : 2 |
| FireBird -> FireMane  Phoenix -> FireVoid  FireVoid -> FireMane  FireSnow -> FireMane  Phoenix -> FireBird  FireMane -> FireBird  FireMane -> FireVoid  Phoenix -> FireSnow  FireMane -> FireSnow  FireMane -> FireMane  Phoenix -> FireMane  Phoenix -> FireVoid  Blaze it! | Phoenix : 4  FireBird : 0  FireVoid : 0  FireSnow : 0  FireMane : 0 |

# Trainlands

So a train goes from A to B… Okay that’s too mainstream. Trainlands! Trainlands is a game of trains, in which trains develop themselves into gigantic trains for the competition. It got too trainy so let’s get to the main logic.

The **train** has a **name** (**string**), and **wagons**. The **wagons** have a **name** (**string**) and **power** (**integer**).   
The input comes in one of the following formats:

{trainName} -> {wagonName} : {wagonPower}

**Creates** a **train** with the **given name** and **adds** a **wagon** to it with the **given name** and **power**. If the train **already exists**, it just **adds** the **given wagon** to it.

{trainName} -> {otherTrainName}

**Adds all** of the **wagons** from the **other train** to the **first train**, and **REMOVES** the **other train**. If the first train **does NOT exist**, create it, and then **add** the **wagons**. The other train will **ALWAYS** be **EXISTENT**.

{trainName} = {otherTrainName}

**Copies** the **other train’s wagons**, without affecting the other train. **Copying** means, that the **first train’s wagons** **become** the **same** as the **other train’s wagons**. If the first train **does NOT exist**, create it. The other train will **ALWAYS** be **EXISTENT**.

When you get the command “It’s Training Men!” you should end the input sequence and print all of the trains and their wagons. The **trains** must be **ordered** by **total wagon power**, in **descending order**, and by **wagon count** in **ascending order** as **secondary criteria**. For each train you should print its wagons, **ordered** by **wagon power** in **descending order**.

### Input

* The input comes in the form of commands in one of the formats specified above.
* The input ends when the command “It’s Training Men!” is entered.

### Output

* As output you must print all of the trains and their wagons ordered as specified above.
* The format of printing is:

Train: {trainName}

###{wagon1Name} – {wagon1Power}

###{wagon2Name} – {wagon2Power}

. . .

### Constrains

* The **names** of the **trains** and the **wagons** will be **strings**.
* The **names** may contain **any ASCII character**, except “ ”, “-”, “**:**”, “>”, “=”.
* The **wagon power** will be a valid **integer** in **range [0, 1.000.000]**.
* There will be **NO invalid input**.
* The **wagons** will always have **unique** **names** in the **scope** of their **train**.
* Allowed time / memory: 100ms / 16 MB.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Kivil -> KAKA : 1387  Zone -> Gh : 4081  Kivil -> RAMZES666 : 4677  Desolator -> MiraclE~ : 8432  Zone -> Kivil  It's Training Men! | Train: Zone  ###RAMZES666 - 4677  ###Gh - 4081  ###KAKA - 1387  Train: Desolator  ###MiraclE~ - 8432 |
| Kepler -> MinD\_ContRoL : 3782  Daun -> Fn : 6816  Miner -> Gh : 1198  Miner -> Sccc : 9030  Miner -> KAKA : 7409  Anna -> Miner  Daun = Anna  It's Training Men! | Train: Daun  ###Sccc - 9030  ###KAKA - 7409  ###Gh - 1198  Train: Anna  ###Sccc - 9030  ###KAKA - 7409  ###Gh - 1198  Train: Kepler  ###MinD\_ContRoL - 3782 |

# Pokemon Evolution

You have been tasked to keep track of pokemons and their evolutions. A pokemon can evolve in several phases and types. When it evolves, the pokemon has an evolution index, which indicates how much it has evolved.

You will receive input lines in the following format:  
{pokemonName} -> {evolutionType} -> {evolutionIndex}

The pokemonName and evolutionType will be **strings**. The evolutionIndex will be an **integer**. Your task is to store every **pokemon** and his **evolutions**.   
If you receive an existent pokemonName, you should **add** the **new** **evolution** to it.

A single **pokemon** may have **many evolutions** with the **same type** and the **same index**.

In some rare cases you may receive the following input:  
{pokemonName}

When you receive only a pokemonName, you must **check if there is** such a **pokemon**, and if there is, you must print all of its **evolutions** by **order of input**.

The **input sequence ends** when you **receive** the command “wubbalubbadubdub”.   
Then you must print all pokemons and their evolutions. The pokemons must be printed by **order of input**. Each **pokemon’s evolutions** must be **ordered** by **evolution index** in **descending order**.

### Input

* The input will come in the form of lines in the format specified above.
* In some rare cases you may have only one element of the input – the pokemonName.
* The input sequence ends when you receive the command “wubbalubbadubdub”.

### Output

* **Pokemons** and their **evolutions** must be printed in the following format:

“# {pokemoName}  
 {evolution1Type} <-> {evolution1Index}  
 {evolution2Type} <-> {evolution2Index}

…”

* If you have received a pokemonName and you are **printing its evolutions**, the order is – by **order of input**.
* If you have received the **ending command**, and you are printing the **pokemons’ evolutions**, the order is – by evolutionIndex in **descending order**.

### Constrains

* The pokemonName and evolutionType are strings which may contain any ASCII character   
  (except ‘-’, ‘ ’, ‘>’).
* The evolutionIndex will be an **integer** in **range [0, 1.000.000.000]**.
* There will be **NO invalid** input data.
* Allowed time / memory: **100ms / 16 MB**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Ekans -> Hybrid -> 100  Nidoran -> Physical -> 150  Ekans -> Psychological -> 50  Jigglypuff -> Hybrid -> 1000  Jigglypuff -> Physical -> 2000  wubbalubbadubdub | # Ekans  Hybrid <-> 100  Psychological <-> 50  # Nidoran  Physical <-> 150  # Jigglypuff  Physical <-> 2000  Hybrid <-> 1000 |
| Pikachu -> Hybrid -> 100  Meowth -> Physical -> 100  Pikachu -> Psychological -> 50  Meowth -> Physical -> 50  Pikachu -> Hybrid -> 150  Meowth  Pikachu  wubbalubbadubdub | # Meowth  Physical <-> 100  Physical <-> 50  # Pikachu  Hybrid <-> 100  Psychological <-> 50  Hybrid <-> 150  # Pikachu  Hybrid <-> 150  Hybrid <-> 100  Psychological <-> 50  # Meowth  Physical <-> 100  Physical <-> 50 |

## b56f3c0f767242e9a52b947a2b80436877d733b0_hq