# More Exercises: Lambda and LINQ

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

Check your solutions here: <https://judge.softuni.bg/Contests/436>.

## Array Data

You will receive a list of **integers**, each separated with a single space. On the **next** line, you will receive **one** of the following commands: “**Min**”, “**Max**” or “**All**”.

Your task is to **remove** all numbers, which are **smaller** than the **average** of the **initial** **list** and depending on the next command, to print an **additional** **line**:

* If the command is “**Min**” – print the **smallest** element from the **remaining** numbers.
* If the command is “**Max**” – print the **largest** element from the **remaining** numbers.
* If the command is “**All**” – print **all** of the **remaining** numbers **ordered** in **ascending** **order** and **separated** by a **single** **whitespace**.

### Constraints

* The elements of the array will be in the interval [-2147483648…2147483647]

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 2 5 1 8 2 3  Min | 5 |
| 9 10 8 1 2 6 7 3 4 5  All | 6 7 8 9 10 |

## String Decryption

Write a program, which can decipher an array of integers into a string message.

### Input

On the **first** line, you will receive an **array** of **integers** with exactly **two** elements – **m** and **n**:

* **m** will be the elements you have to **skip** from the **beginning**.
* **n** will be the element you have to **take**, **after** the **skipped** elements.

On the **second** line, you will receive an **array** of **integers**.

### Output

Your task is to take **only** the **numbers**, which are in the **capital latin letter** ASCII range (**65…90 inclusive**). From the **filtered** numbers, **skip** the next **m** elements (**starting** from **0**) and take the **next** **n** elements. After that, take **only** the **unique** **elements** and **convert** **each** **number** into **ASCII** **characters**.

**Print** the resulting string of **ASCII characters** on the console.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 10  67 80 87 78 65 25 75 79 86 82 85 56 76 90 27 90 | NAKOVRULZZ |
| 4 5  23 70 71 72 14 73 80 25 73 12 90 18 90 65 | PIZZA |

## Camping

As a young and ambitious manager of a camping ground you want to invent a new system, which will keep track of the registered campervans (RVs).

Until you receive the command “**end**”, you will receive commands in the format:

* {nameOfThePerson} {camperModel} {timeToStay}

If you receive **already** **existing** **name** of a person 🡺 add the camper (only if it is **new**) to the already **owned** by the person campers and **add** the **days** to the already **accumulated** days.

Your task is to print **all** people in the following **format**:

|  |
| --- |
| \*{nameOfThePerson}: {countOfCampers}  \*\*\*{1st camperModel}  \*\*\*{2nd camperModel}  …  \*\*\*{nth camperModel}  Total stay: {countOfNights} nights |

Print all the **people**, **ordered** by the **count** of their **campers**. If **two people** have an **equal** **count** of campers, **order** them by the **length** of their **names** in **ascending** **order**. **Print** the **campers** for each person in **order** of **receiving**.

### Constraints

* The nights will be in the interval [0…255]
* You will not receive the **same** camper **model** **twice** for a **single** person.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Pesho BigTruck2000 20  Stefan HeliumV100 10  Pesho ILoveRVs100 5  Maria MyPrecious 20  Gesh BigTruck1500 7  Bibi Mercedes 3  Gesh AwesomeCamper1000 12  end | Gesh: 2  \*\*\*BigTruck1500  \*\*\*AwesomeCamper1000  Total stay: 19 nights  Pesho: 2  \*\*\*BigTruck2000  \*\*\*ILoveRVs100  Total stay: 25 nights  Bibi: 1  \*\*\*Mercedes  Total stay: 3 nights  Maria: 1  \*\*\*MyPrecious  Total stay: 20 nights  Stefan: 1  \*\*\*HeliumV100  Total stay: 10 nights |
| Pesho BigTruck2000 20  Pesho HeliumV100 10  Pesho ILoveRVs100 5  Pesho MyPrecious 20  Stefan BigTruck1500 7  Pesho Mercedes 3  end | Pesho: 5  \*\*\*BigTruck2000  \*\*\*HeliumV100  \*\*\*ILoveRVs100  \*\*\*MyPrecious  \*\*\*Mercedes  Total stay: 58 nights  Stefan: 1  \*\*\*BigTruck1500  Total stay: 7 nights |

## Most Valued Customer

You are an assistant-manager at a local supermarket and you are given the task to award a prize to your most dedicated customer. The **dedication** of your customers is measured by the **amount of money** they have **spent** at your store.

### Input

Until you receive the command “Shop is open”, you will receive **products** in the format:

{productName} {productPrice}

**After** you receive “Shop is open”, you will start receiving **customers** and **products**, which they bought in the following **format**:

{nameOfCustomer}: {firstBoughtProduct}, {secondBoughtProduct}, … {nthBoughtProduct}

Every customer’s total **spending** is equal to the **sum** of the **prices** of **all** **products** which they bought. If you receive a **product**, which you **do** **not** **sell** – **ignore** **it**. A customer can buy **one** product **more** **than** **once**.

You might also receive the command “Discount”. When you receive it – **decrease** the prices of the top **3** **most** **expensive** products by **10%**. Calculate the **total** sum, using the **prices** **after** **all** discounts are made.

When you **receive** the command “Print”, **stop** taking input.

### Output

After you receive “Print”, **print** the customer, which **spent the** **most** **money** and **all** of the products they **bought** (for the **price**, use the **discounted** **one**) in the following **format**:

|  |
| --- |
| Biggest spender: {nameOfTheCustomer}  ^Products bought:  ^^^{1st Product}: {priceOfTheProduct}  ^^^{2nd Product}: {priceOfTheProduct}  ...  ^^^{nth Product}: {priceOfTheProduct}  Total: {priceOfAllProducts} |

Print the **products** ordered by **price** in **descending** order. Print **each** product only **once**. Format **all** prices to the **2nd** **decimal place**.

### Constraints

* When you are receiving the **prices** of the **products**, you will **not** receive **duplicate** product **names**.
* There will never be **two** **people** with the **same** **total** **sum**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Bread 1.20  Butter 3.60  Milk 4  Beer 1.40  Sweets 4.20  Shop is open  Gosho: Bread, Beer, Bread  Pesho: Sweets, Butter  Print | Biggest spender: Pesho  ^Products bought:  ^^^Sweets: 4.20  ^^^Butter: 3.60  Total: 7.80 |
| Bread 1.20  Butter 3.60  Beer 1.40  Sweets 4.20  Shop is open  Gosho: Bread, Beer, Bread  Discount  Discount  Discount  Discount  Discount  Discount  Discount  Discount  Discount  Discount  Discount  Discount  Discount  Discount  Discount  Pesho: Sweets, Butter  Print | Biggest spender: Gosho  ^Products bought:  ^^^Beer: 0.60  ^^^Bread: 0.57  Total: 1.75 |

## Lambada Expressions

IT Researchers at the MIT (Massachusetts Institute of Technology) have discovered that, at certain (room) temperatures, the lambda expressions in their code are acting strange. In particular, a dancing behavior has been observed in them. The specialists choose to call this phenomenon – The Lambada expressions.

### Input

You will be given several input lines in the following format:

{selector} => {selectorObject}.{property}

All elements of the input are **strings**. You need to store every **lambada expression**, as it is **given**.

In some rare cases, you will receive the input “**dance**”, which indicates that the lambada expressions are starting to dance. When that happens, you must **COPY** the **selectorObject ONCE** with a **dot** (“**.**”) **between** it., so that it becomes “{selectorObject}.{selectorObject}”

in **every** lambada expression’s **condition**.   
In other words, if you have “x => x.Value”, and you say “**dance**”, you should get “x => x.x.Value”.

Check the examples for more info.

If you meet a **selectorObject** which **already exists**, you must **replace** its **property** with the **given new one**.

When you receive the command “lambada”, you must **stop** reading input.

### Output

Print all **lambada expressions** in **order of insertion**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| x => x.Key  object => object.Property  entry => entry.Name  lambada | x => x.Key  object => object.Property  entry => entry.Name |
| x => x.Key  x => x.Value  x => x.Name  dance  dance  dance  lambada | x => x.x.x.x.Name |

## Ordered Banking System

You have been given the task to write software for banking – in particular, an optimized program, which stores only the bank account’s name, the bank account’s balance and the bank it is stored at.

### Input

You will be given several input lines in the following format:

{bank} -> {account} -> {balance}

The **bank** and the **account** are **strings**, and the **balance** is a **VARIABLE TYPE** which is **SUITABLE** for storing **MONEY**.

You should store every bank, each of its accounts, and their balances.

If you are given a **bank** which **already exists**, you must **ADD** the **new account** to it.  
If **EVEN** the **account** which **already exists**, you must **INCREASE** the **old balance**, with the new **given one**.

When you receive the command “**end**”, you must end the input sequence. Then you must print the data in a specified ordered format.

### Output

You must print each bank’s accounts and balances, ordering the **BANKS** by the **sum** of **all account balances**, in **descending order**. If **2 banks** have the **same sum**, you should print the **one** with the **highest maximum balance** in its **accounts**, **first**.

The **bank’s accounts** must be ordered in **descending order**, by **their balance**.

The balances, must be printed, as they are given. “**500**” = “**500**” … “**500.00**” = “**500.00**”.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| RaifaizenBank -> Toshko -> 200.4456  DSK -> Toshko -> 500.00  PIB -> Ivan -> 400.521  Piraeus -> Ivan -> 212.345  end | Toshko -> 500.00 (DSK)  Ivan -> 400.521 (PIB)  Ivan -> 212.345 (Piraeus)  Toshko -> 200.4456 (RaifaizenBank) |
| DSK -> Tesla -> 2015125.45623  DSK -> Tesla -> 2000000  DSK -> Microsoft -> 400000  DSK -> Apple -> 200000  DSK -> Microsoft -> 350000  DSK -> Apple -> 550000  end | Tesla -> 4015125.45623 (DSK)  Microsoft -> 750000 (DSK)  Apple -> 750000 (DSK) |

## LINQuistics

LINQ is the greatest .NET component of all time. You can do almost anything with it. That’s why you have been tasked to do almost everything you can with it.

### Input

You will be given several input lines containing information about collections, and LINQ methods that have been called on them, in the following format:

{collection}.{method1}().{method2}()....{methodN}()

The **count** of **methods** may **vary**. Your task is to **store** every collection and the **methods** that have been executed on it. If the collection **already exists**, you must **ADD** the new methods to it. **Duplicate** methods should be **REMOVED**.

### Output

If you are given **only** a **collection name**, you must **print** the **methods** that have been **executed** on the collection, **ordered** by their **length** in **descending order**. If **2 methods** have the **same length**, **order** them by the **count** of **unique symbols** they have in **their names** in **descending order**.

Each method must be printed on a **new line**, with a prefixed **asterisk and space** (“\* ”).

If the collection name does **NOT** exist, you should **IGNORE** that line of input.

If you are given **only** a **digit**, you must **take** **the collection** which **has the most methods**, and print the **first N methods**, with the **lowest length** (**N** being the **digit** **given** in the **input**). If there are **less** than **N** methods you must print **all** of them in the **same order**.

**NOTE**: When printing, you must print **only** the **method name**, **without** its brackets (e.g. “First”, **not** “First()”).

The input sequence ends when you receive the command “**exit**”. After the ending command, you will receive one last line in the following format:  
{method} {selection}

You must **take all** collections, which **CONTAIN** the **given method**, and print them. The selection will either be “collection” or “all”.   
If you have “collection”, you must **print** only the **collections’ names** in the final output.  
If you have “all”, you must **print** the **collections** **and their methods** in the following format:

|  |
| --- |
| {collection}  \* {method1}  \* {method2} ... |

The collections must be printed ordered by the **count of their methods** in **descending order**.

If **2 collections** have the **same** **amount of methods**, print the **one** whose **shortest method name** is **longer** than the other one’s **shortest method name**.

The **methods** must be printed, ordered by their **length** in **descending order**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| participants.Max().Reverse().ThenBy()  participants.OrderBy.Select()  participants  participants.ToDictionary()  collection.Max()  collection.Break()  exit  Max all | \* OrderBy  \* Reverse  \* ThenBy  \* Select  \* Max  participants  \* ToDictionary  \* Reverse  \* OrderBy  \* ThenBy  \* Select  \* Max  collection  \* Break  \* Max |
| elements.Sort()  elements.OrderBy()  bound  elements.Reverse().Select().ThenBy()  keys.Reverse().OrderByDescending()  keys.Reverse().ThenByDescending()  3  keys.Reverse().OrderBy().ThenBy()  values.ToString().ToString().ThenBy()  exit  Reverse collection | \* Sort  \* Select  \* ThenBy  keys  elements |