# 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/435>.

## Registered Users

You will be given several usernames and dates of registry, in the following format:

{username} -> {date}

The **username** will be a **string** and the **date** will be a valid date in format “**dd/MM/yyyy**”.

You must read input lines until you receive the command “**end**”.

You need to store every user, with his date of registry. Then you must print the **LAST 5** registered names, or in other words the **OLDEST** registered users, in **order of registry** – from the **latest registered,** to the **oldest registered**.

If there are **LESS** than **5** people, print **them** in the **order**, **specified above**.  
if any entries have the **SAME date**, print the **LAST** entered one, **FIRST**, and then the other (in **input**).

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| John27 -> 12/02/2004  7H3D347H -> 15/09/2015  Bojo96 -> 15/09/2015  Sanity -> 04/05/2015  Innos -> 04/05/2015  Nakov -> 01/01/2013  Danny -> 04/04/2016  Yori -> 29/02/2016  end | Bojo96  Innos  Sanity  Nakov  John27 |
| Danny -> 04/04/2016  Innos -> 04/05/2015  Sanity -> 04/05/2015  Nakov -> 01/01/2013  end | Danny  Sanity  Innos  Nakov |

### Hints

* Use the DateTime.ParseExact() method. The method **accepts** a **string** (the **input data**), a **date format**, and CultureInfo.
* For CultureInfo, use CultureInfo.InvariantCulture.

## Default Values

You will be given several key-value pairs in the following format: “{key} -> {value}”

Your task is to store all these pairs until you receive the command “end”. When you do receive the ending command, you must read **one last line**, containing the **default value**. Then you must change all values, which are equal to “null” with the **given default value**.

At the **end,** you must print **all key-value pairs**, which have **NOT** been **replaced** with the **default value**, ordered in **descending** order, by their **value’s length**.

**After** them, you must print **all key-value** pairs which **HAVE been replaced** with the **default value**, in **order of input**.

Each key-value pair, must be printed in the following format: “{key} <-> {value}”.

All variables in the program logic are strings.

Note: If you receive the same key twice or more times, you should update its value every time.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| What -> null  How -> null  To -> Do  This -> null  Correctly -> null  end  welp | To <-> Do  What <-> welp  How <-> welp  This <-> welp  Correctly <-> welp |
| This -> Input  Will -> Be  Sorted -> null  By -> Descending  Order -> null  end  sortorder | By <-> Descending  This <-> Input  Will <-> Be  Sorted <-> sortorder  Order <-> sortorder |

## Shopping Spree

Pesho has a problem with taking decisions and mostly with deciding on what to eat for dinner. He asked you to write a program which chooses dinner for him.

On the **first** line, you will receive Pesho’s **budget** as **decimal** number.

On the **next** lines, until you receive the command “**end**”, you will receive strings with the **name** of the **product** and the **price** of the **product**. Save **all** products and if you receive the **same** product and its **price** is **lower** than the **already saved** one – **rewrite** the old value.

At the **end**, **sum** **all values** of the **received** products. If the **sum** is **bigger** than Pesho’s **budget** – print:

**“**Need more money... Just buy banichka**”**

If Pesho **has** **enough** money – print **all** **purchased** products, **ordered** by **price** in **descending** order, **then** by the **count** of the **characters** in its **name** in **ascending order**. Use the **following** format for **each** product:

**“**{productName} costs {productPrice}**”**

Print all **prices** **rounded** to the **2nd** decimal **sign**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 20.50  Bread 2.41  Butter 3.12  Cheese 4.20  Bread 2.12  Milk 4.20  end | Milk costs 4.20  Cheese costs 4.20  Butter costs 3.12  Bread costs 2.12 |
| 10.50  Bread 2.41  Butter 3.12  Cheese 4.20  Bread 2.12  Milk 4.20  end | Need more money… Just buy banichka |

## SoftUni Beer Pong

In SoftUni it is time for the annual beer pong tournament. Last year the competition was a total disaster, because nobody was in “condition” to keep track of the scores. Now it is your job to write a program, which will keep track of the total results. The competition’s rules are simple:

* **All** contestants are **divided** into **teams**
* Every **team** should have **exactly 3** **participants**
* After a team has a total of **3** participants, any **further** attempt to **add** a **participant** should be **ignored**.
* If a team has **less** **than** **3** participants – it gets **disqualified** and should **not** be **printed**.

Until you receive the command “stop the game” you will receive **lines** of **input** in the following **format**:

**“**team|player|pointsMade**”**

The **total** score of the team is the **sum** of the **points** made from **every** **player**.

Print **every** team, which has **enough** **players** in the following **format**:

|  |
| --- |
| **{teamPosition}. {teamName}; Players:**  **###{firstPlayerName}: {points}**  **###{secondPlayerName}: {points}**  **###{thirdPlayerName}: {points}** |

Order the teams by **total points** in **descending** **order** and **order each team’s players** by the amount of **points made**, again in **descending order**.

### Constraints

* You will **not** receive a player’s name, which **already exists** in the team it’s trying to be inserted into.
* Points will be in the interval [0…2147483647].
* The **total** **points** of the teams and the **points** of the **players** in the same team will **not** be **equal**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Jubrata|Retro|57  Pesho|Destroyers|20  Gosho|Destroyers|35  Pesho|TechModule|32  Stefan|Destroyers|12  Kaloyan|TechModule|14  Meto|TechModule|24  Hrisi|Retro|59  Maria|TechModule|19  stop the game | 1. TechModule; Players:  ###Pesho: 32  ###Meto: 24  ###Kaloyan: 14  2. Destroyers; Players:  ###Gosho: 35  ###Pesho: 20  ###Stefan: 12 |
| Ivan|Loosers|5  Ivo|RoYaL|100  Vladi|RoYaL|120  Pesho|Trainers|22  Stefan|NoobS|12  RoYaL|RoYaL|90  Geotgi|NoobS|24  Anna|NoobS|70  Maria|Loosers|19  Pesho|Loosers|30  stop the game | 1. RoYaL; Players:  ###Vladi: 120  ###Ivo: 100  ###RoYaL: 90  2. NoobS; Players:  ###Anna: 70  ###Geotgi: 24  ###Stefan: 12  3. Loosers; Players:  ###Pesho: 30  ###Maria: 19  ###Ivan: 5 |

## Flatten Dictionary

You will be given several input lines containing info about a **key**, an **inner key** and **inner value**, **separated** by **SPACES**.

Your task is to store every **inner key** and **inner value**, in **every key’s value**. Check the Examples for more info.

If you receive the command “flatten {key}”, you must **flatten** all the **inner** **keys** and **inner** **values** at the **given key**, or in other words, **CONCATENATE** them.

When you receive the command “**end**”, the **input ends**. You must print all **keys** and their **inner-keys** and **inner-values**. **Flattened** **inner keys** and **inner values**, must be printed **AFTER** all else.

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

All **inner-keys** must be printed ordered by length in **ascending order**.

Flattened Values must be printed in **order of input**.

The format of printing regular keys is: Flattened values should be printed like this

|  |  |
| --- | --- |
| {key}  1. {innerKey} – {innerValue}  2. {innerKey} – {innerValue}  3. {innerKey} – {innerValue}  ... | {key}  1. {flattenedValue1}  2. {flattenedValue2}  3. {flattenedValue3}  ... |

If you receive an **inner** **key** that **already** **exists**, **replace** its value with the **new given one**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Cars Opel Astra  Cars Opel Vectra  Laptops Lenovo T500  Cars BMW X6  Laptops Acer Aspire  Cars Audi Q7  end | Laptops  1. Acer - Aspire  2. Lenovo - T500  Cars  1. BMW - X6  2. Opel - Vectra  3. Audi - Q7 |
| TV Samsung Home  TV Sony Office  TV Samsung Office  Default Get Set  Default Set Get  flatten TV  TV Lenovo General  TV Samsung Home  end | Default  1. Get - Set  2. Set - Get  TV  1. Lenovo - General  2. Samsung - Home  3. SamsungOffice  4. SonyOffice |

## CottageScraper \*

You’re a carpenter at the local woodworking shop. Your client wants you to build a wooden skyscraper/cottage. A **CottageScraper**, he calls it. But in order to accomplish this task, he needs the trees to be **taller** than a **certain height**. The problem is that he **doesn’t know** what type of tree he’ll use yet, and doesn’t know **how tall** he wants the **CottageScraper** to be. You’ve obviously got nothing better to do, so you go to work chopping down logs until he calls you with the details.

Write a program which receives tree **types** and their **lengths** in the format “{type} -> {height}”. When you receive the command “chop chop”, it’s time to get paid.

On the next line, you will receive the **type** **of tree** that needs to be used to build the **CottageScraper**. On the final input line, you will receive the **minimum length per tree**, needed to accomplish the task. Filter the trees based on type and minimum length, making sure that you’ll only use the trees of the specified **type** and **minimum length**.

After which, calculate the **total price** of the **CottageScraper**, which includes the price of all the trees you collected up to that point. The price is calculated as being the **average** meters of **all** **logs** you collected, per **meter** of log, **rounded to the second decimal place**.

You’re going to charge the client **100%** of the **price per log** for logs you’ll use in the skyscraper, and **25%** of the price per log for logs you **won’t** use for the **CottageScraper**. Both fees are **rounded to the second decimal place**.

After you make the calculations, **print** them on the console in the following format:

* On the first line of the console, print “Price per meter: ${pricePerMeter:F2}”.
* On the second, print “Used logs price: ${usedLogsPrice:F2}”.
* On the third line, print “Unused logs price: ${unusedLogsPrice:F2}”.
* On the final line of the output, print “CottageScraper subtotal: ${subTotal:F2}”.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| Maple -> 20  Oak -> 12  Poplar -> 25  Maple -> 33  Poplar -> 11  Poplar -> 30  **chop chop**  Poplar  12 | Price per meter: $21.83  Used logs price: $1200.65  Unused logs price: $414.77  CottageScraper subtotal: $1615.42 | Needed type: Poplar  Needed height: at least 12m  Price per meter == (sum of all logs) / (count) == 21.83  Used logs (taller than 12m): Poplar -> 25, 30  Used logs price: (25 + 30) \* 21.83 = **$1200.65**  Unused logs: Maple -> 20, 33; Oak -> 12; Poplar -> 11  Unused logs price: (20 + 33 + 12 + 11) \* 21.83 \* 0.25 = **$414.77**  Used + unused logs price: 1200.65 + 414.77 = **$1615.42** |
| Cherry -> 918  Oak -> 112  Maple -> 1423  Maple -> 9118  Poplar -> 122  Oak -> 232  **chop chop**  Maple  250 | Price per meter: $1987.50  Used logs price: $20950237.50  Unused logs price: $687675.00  CottageScraper subtotal: $21637912.50 | Needed type: Maple  Needed height: at least 250m  Price per meter = 918 + 112 + 1423 + 9118 + 122 + 232 / 6 = **$1987.50**  Used logs (taller than 250m): Maple -> 1423, 9118  Used logs price: (1423 + 9118) \* 1987.50 = **$20950237.50**  Unused logs: Cherry -> 918, Oak -> 112, 232, Poplar -> 122  Unused logs price: (918+ 112+ 232+ 122) \* $1987.50 \* 0.25 = **$687675.00**  Used + unused logs price: **$21637912.50** |
| Apple -> 218  Pear -> 112  Apple -> 123  Apple -> 118  Pear -> 122  Cherry -> 232  **chop chop**  Apple  120 | Price per meter: $154.17  Used logs price: $52571.97  Unused logs price: $22508.82  CottageScraper subtotal: $75080.79 |  |

### Hints

* To flatten the dictionary’s values (so you can calculate the price per log), you can use the LINQ .SelectMany() method.
* The unused logs include not only the logs from different types than needed, but also the logs whose height was **lower** than the minimum height.