

# Exercises: Objects, Associative Arrays, Maps, Sets

Problems for exercises and homework for the [“JavaScript Fundamentals” course @ SoftUni](#). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/316/>.

## 1. Heroic Inventory

In the era of heroes, every hero has his own items which make him unique. Create a function which creates a **register for the heroes**, with their **names**, **level**, and **items**, if they have such. The register should accept data in a specified format, and return it presented in a specified format.

The **input** comes as array of strings. Each element holds data for a hero, in the following format:

“{heroName} / {heroLevel} / {item1}, {item2}, {item3}...”

You must store the data about every hero. The **name** is a **string**, the **level** is a **number** and the items are all **strings**.

The **output** is a **JSON representation** of the data for all the heroes you’ve stored. The data must be an **array of all the heroes**. Check the examples for more info.

### Examples

Input	Output
['Isacc / 25 / Apple, GravityGun', 'Derek / 12 / BarrelVest, DestructionSword', 'Hes / 1 / Desolator, Sentinel, Antara']	[{"name":"Isacc","level":25,"items":["Apple","GravityGun"]}, {"name":"Derek","level":12,"items":["BarrelVest","DestructionSword"]}, {"name":"Hes","level":1,"items":["Desolator","Sentinel","Antara"]}]
Input	Output
['Jake / 1000 / Gauss, HolidayGrenade']	[{"name":"Jake","level":1000,"items":["Gauss","HolidayGrenade"]}]

### Hints

- We need an array that will hold our hero data. That is the first thing we create.

```
function main(input) {  
    let heroData = [  
    ];  
}
```

- Next, we need to loop over the whole input, and process it. Let’s do that with a simple **for** loop.

```
function main(input) {  
    let heroData = [  
    ];  
  
    for(let i = 0; i < input.length; i++) {  
        let currentHeroArguments = input[i].split(" / ");  
    }  
}
```

- Every element from the input holds data about a hero, however the **elements from the data** we need are **separated by some delimiter**, so we just split each string with that **delimiter**.

- Next, we need to take the elements from the **string array**, which is a result of the **string split**, and parse them.

```
for(let i = 0; i < input.length; i++) {
    let currentHeroArguments = input[i].split(" / ");

    let currentHeroName = currentHeroArguments[0];
    let currentHeroLevel = Number(currentHeroArguments[1]);
    let currentHeroItems = currentHeroArguments[2].split(", ");
}
```

- However, if you do this, you could get quite the error in the current logic. If you go up and read the problem definition again, you will notice that there might be a **case** where the hero **has no items**; in that case, if we try to take the **3<sup>rd</sup> element** of the **currentHeroArguments** array, it will **result in an error**. That is why we need to perform a simple check.

```
let currentHeroItems = [];

if(currentHeroArguments.length > 2) {
    currentHeroItems = currentHeroArguments[2].split(", ");
}
```

- If **there are any items** in the **input**, the **variable** will be set to the **split version of them**. If not, it will just remain an **empty array**, as it is **supposed to**.
- We have now extracted the needed data – we have stored the **input name** in a **variable**, we have parsed the **given level** to a **number**, and we have also **split** the **items** that the **hero holds** by their **delimiter**, which would result in a **string array** of elements. By definition, the **items** are **strings**, so we don't need to process the array we've made anymore.
- Now what is left is to add that data into an **object** and **add** that object to the **array**.

```
let hero = {
    name: currentHeroName,
    level: currentHeroLevel,
    items: currentHeroItems
};

heroData.push(hero);
```

- Lastly, we need to turn the array of objects we have made, into a JSON string, which is done by the **JSON.stringify()** function

```
console.log(JSON.stringify(heroData));
```

## 2. JSON's Table

JSON's Table is a magical table which turns JSON data into an HTML table. You will be given **JSON strings** holding data about employees, including their **name**, **position** and **salary**. You need to **parse that data** into **objects**, and create an **HTML table** which holds the data for each **employee on a different row**, as **columns**.

The **name** and **position** of the employee are **strings**, the **salary** is a **number**.

The **input** comes as array of strings. Each element is a JSON string which represents the data for a certain employee.

The **output** is the HTML code of a table which holds the data exactly as explained above. Check the examples for more info.

## Examples

Input	Output
<pre>[{"name":"Pesho","position":"Promenliva","salary":100000}, {"name":"Teo","position":"Lecturer","salary":1000}, {"name":"Georgi","position":"Lecturer","salary":1000}]</pre>	<pre>&lt;table&gt;   &lt;tr&gt;     &lt;td&gt;Pesho&lt;/td&gt;     &lt;td&gt;Promenliva&lt;/td&gt;     &lt;td&gt;100000&lt;/td&gt;   &lt;/tr&gt;   &lt;tr&gt;     &lt;td&gt;Teo&lt;/td&gt;     &lt;td&gt;Lecturer&lt;/td&gt;     &lt;td&gt;1000&lt;/td&gt;   &lt;/tr&gt;   &lt;tr&gt;     &lt;td&gt;Georgi&lt;/td&gt;     &lt;td&gt;Lecturer&lt;/td&gt;     &lt;td&gt;1000&lt;/td&gt;   &lt;/tr&gt; &lt;/table&gt;</pre>

## Hints

- You might want to **escape the HTML**. Otherwise you might find yourself victim to vicious JavaScript **code in the input**, which aims only to hack you.

## 3. Cappy Juice

You will be given different juices, as **strings**. You will also **receive quantity** as a **number**. If you receive a juice, you already have, **you must sum** the **current quantity** of that juice, with the **given one**. When a juice reaches **1000 quantity**, it produces a bottle. You must **store all produced bottles** and you must **print them** at the end.

**Note:** **1000 quantity** of juice is **one bottle**. If you happen to have **more than 1000**, you must make **as much bottles as you can**, and store **what is left** from the juice.

**Example:** You have **2643 quantity** of Orange Juice – this is **2 bottles** of Orange Juice and **643 quantity left**.

The **input** comes as array of strings. Each element holds data about a juice and quantity in the following format:

**"{juiceName} => {juiceQuantity}"**

The **output** is the produced bottles. The bottles are to be printed in **order of obtaining the bottles**. Check the second example bellow - even though we receive the Kiwi juice first, we don't form a bottle of Kiwi juice until the 4<sup>th</sup> line, at which point we have already create Pear and Watermelon juice bottles, thus the Kiwi bottles appear last in the output.

## Examples

Input	Output
['Orange => 2000', 'Peach => 1432', 'Banana => 450', 'Peach => 600', 'Strawberry => 549']	Orange => 2 Peach => 2

Input	Output
['Kiwi => 234', 'Pear => 2345', 'Watermelon => 3456', 'Kiwi => 4567', 'Pear => 5678', 'Watermelon => 6789']	Pear => 8 Watermelon => 10 Kiwi => 4

## 4. Store Catalogue

You have to create a sorted catalogue of store products. You will be given the products' names and prices. You need to order them by **alphabetical order**.

The **input** comes as array of strings. Each element holds info about a product in the following format:

**"{productName} : {productPrice}"**

The **product's name** will be a **string**, which will **always start with a capital letter**, and the **price** will be a **number**. You can safely assume there will be **NO duplicate product input**. The comparison for alphabetical order is **case-insensitive**.

As **output** you must print all the products in a specified format. They must be ordered **exactly as specified above**. The products must be **divided into groups**, by the **initial of their name**. The **group's initial should be printed**, and after that the products should be printed with **2 spaces before their names**. For more info check the examples.

## Examples

Input	Output
['Appricot : 20.4', 'Fridge : 1500', 'TV : 1499', 'Deodorant : 10', 'Boiler : 300', 'Apple : 1.25', 'Anti-Bug Spray : 15', 'T-Shirt : 10']	A Anti-Bug Spray: 15 Apple: 1.25 Appricot: 20.4 B Boiler: 300 D Deodorant: 10 F Fridge: 1500 T T-Shirt: 10 TV: 1499

Input	Output
['Banana : 2', 'Rubic's Cube : 5', 'Raspberry P : 4999', 'Rolex : 100000', 'Rollon : 10', 'Rali Car : 2000000', 'Pesho : 0.000001', 'Barrel : 10']	B Banana: 2 Barrel: 10 P Pesho: 0.000001 R Rali Car: 2000000 Raspberry P: 4999 Rolex: 100000 Rollon: 10 Rubic's Cube: 5

## 5. Auto-Engineering Company

You are tasked to create a register for a company that produces cars. You need to store **how many cars** have been produced from a **specified model** of a **specified brand**.

The **input** comes as array of strings. Each element holds information in the following format:

**"{carBrand} | {carModel} | {producedCars}"**

The car **brands** and **models** are **strings**, the **produced cars** are **numbers**. If the **car brand** you've received **already exists**, just add the **new car model** to it with the **produced cars as its value**. If even the car model exists, just **add** the **given value** to the **current one**.

As **output** you need to print – **for every car brand**, the **car models**, and **number of cars produced** from that model. The output format is:

```
"{carBrand}
  ###{carModel} -> {producedCars}
  ###{carModel2} -> {producedCars}
  ..."
```

The order of printing is the **order in which the brands and models first appear in the input**. The first brand in the input should be the first printed and so on. For each brand, the first model received from that brand, should be the first printed and so on.

## Examples

Input	Output
['Audi   Q7   1000', 'Audi   Q6   100', 'BMW   X5   1000', 'BMW   X6   100', 'Citroen   C4   123', 'Volga   GAZ-24   1000000', 'Lada   Niva   1000000', 'Lada   Jigula   1000000', 'Citroen   C4   22', 'Citroen   C5   10']	Audi ###Q7 -> 1000 ###Q6 -> 100 BMW ###X5 -> 1000 ###X6 -> 100 Citroen ###C4 -> 145 ###C5 -> 10 Volga ###GAZ-24 -> 1000000 Lada ###Niva -> 1000000 ###Jigula -> 1000000

## Hints

- The **Map structure** should be perfect for this problem.

## 6. System Components

You will be given a register of systems with components and subcomponents. You need to build an ordered database of all the elements that have been given to you.

The elements are registered in a very simple way. When you have processed all of the input data, you must print them in a specific order. For every System you must print its components in a specified order, and for every Component, you must print its Subcomponents in a specified order.

The **Systems** you've stored must be ordered by **amount of components**, in **descending order**, as **first criteria**, and by **alphabetical order** as **second criteria**. The **Components** must be ordered by **amount of Subcomponents**, in **descending order**.

The **input** comes as array of strings. Each element holds **data** about a **system**, a **component** in that **system**, and a **subcomponent** in that **component**. If the given **system already exists**, you should just **add the new component** to it.

If even the **component exists**, you should just **add** the **new subcomponent** to it. The **subcomponents** will **always be unique**. The input format is:

```
"{systemName} | {componentName} | {subcomponentName}"
```

All of the elements are strings, and can contain **any ASCII character**. The **string comparison** for the alphabetical order is **case-insensitive**.

As **output** you need to print all of the elements, ordered exactly in the way specified above. The format is:

```
"{systemName}
  |||{componentName}
  |||{component2Name}
  |||||{subcomponentName}
  |||||{subcomponent2Name}
  {system2Name}
  ..."
```

## Examples

Input	Output
['SULS   Main Site   Home Page', 'SULS   Main Site   Login Page', 'SULS   Main Site   Register Page', 'SULS   Judge Site   Login Page', 'SULS   Judge Site   Submission Page', 'Lambda   CoreA   A23', 'SULS   Digital Site   Login Page', 'Lambda   CoreB   B24', 'Lambda   CoreA   A24', 'Lambda   CoreA   A25', 'Lambda   CoreC   C4', 'Indice   Session   Default Storage', 'Indice   Session   Default Security']	Lambda    CoreA      A23      A24      A25    CoreB      B24    CoreC      C4 SULS    Main Site      Home Page      Login Page      Register Page    Judge Site      Login Page      Submission Page    Digital Site      Login Page Indice    Session      Default Storage      Default Security

## Hints

- Creating a sorting function with two criteria might seem a bit daunting at first, but it can be simplified to the following:
  - If elements **a** and **b** are different based on the **first criteria**, then that result is the result of the sorting function, checking the second criteria is not required.

- If elements **a** and **b** are **equal** based on the **first criteria**, then the result of comparing **a** and **b** on the **second criteria** is the result of the sorting.

## 7. Usernames

You are tasked to create a catalogue of usernames. The usernames will be strings that **may contain any ASCII** character. You **need to order** them **by their length**, in **ascending order**, as **first criteria**, and by **alphabetical order** as **second criteria**.

The **input** comes as array of strings. Each element represents a **username**. Sometimes the input may contain **duplicate usernames**. Make it so that there are **NO duplicates** in the output.

The **output** is all of the usernames, **ordered** exactly as **specified above** – each printed on a new line.

### Examples

Input	Output	Input	Output
['Ashton', 'Kutcher', 'Ariel', 'Lilly', 'Keyden', 'Aizen', 'Billy', 'Braston']	Aizen Ariel Billy Lilly Ashton Keyden Braston Kutcher	['Denise', 'Ignatius', 'Iris', 'Isacc', 'Indie', 'Dean', 'Donatello', 'Enfuego', 'Benjamin', 'Biser', 'Bounty', 'Renard', 'Rot']	Rot Dean Iris Biser Indie Isacc Bounty Denise Renard Enfuego Benjamin Ignatius Donatello

### Hints

- Try to find a **structure** which **does NOT allow duplicates**, it will be best for the current problem.

## 8. Unique Sequences

You are tasked with storing sequences of numbers. You will receive an unknown amount of **arrays containing numbers** from which you must store only the **unique** arrays (duplicate arrays should be discarded). An array is considered the **same (NOT unique)** if it contains the **same numbers** as another array, **regardless of their order**.

After storing all arrays, your program should print them back in **ascending** order based on their **length**, if two arrays have the same length they should be printed in **order of being received from the input**. Each individual array should be printed in **descending order** in the format "**[a<sub>1</sub>, a<sub>2</sub>, a<sub>3</sub>,... a<sub>n</sub>]**". Check the examples bellow.

The **input** comes as an array of strings where each entry is a JSON representing an array of numbers.

The **output** should be printed on the console - each array printed on a new line in the format "**[a<sub>1</sub>, a<sub>2</sub>, a<sub>3</sub>,... a<sub>n</sub>]**", following the above mentioned ordering.

## Examples

Input	Output
<code>"[-3, -2, -1, 0, 1, 2, 3, 4]", "[10, 1, -17, 0, 2, 13]", "[4, -3, 3, -2, 2, -1, 1, 0]"</code>	<code>[13, 10, 2, 1, 0, -17] [4, 3, 2, 1, 0, -1, -2, -3]</code>

Input	Output
<code>"[7.14, 7.180, 7.339, 80.099]", "[7.339, 80.0990, 7.140000, 7.18]", "[7.339, 7.180, 7.14, 80.099]"</code>	<code>[80.099, 7.339, 7.18, 7.14]</code>

## Hints

- Think of an easy way to compare arrays.
- Sometimes the most obvious collection choice is not the best one.

## 9. \*Arena Tier

Pesho is a pro gladiator, he is struggling to become master of the Arena.

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

`"{gladiator} -> {technique} -> {skill}"`

`"{gladiator} vs {gladiator}"`

The **gladiator** and **technique** are strings, the given **skill** will be an integer number. You need to keep track of **every gladiator**.

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

If you receive `"{gladiator} vs {gladiator}"` and both gladiators exist in the tier, they duel with the following rules:

Compare their techniques, if they got at least one in common, the gladiator with better total skill points wins and the other is demoted from the tier -> remove him.

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

You should end your program when you receive the command **"Ave Cesar"**. At that point you should print the gladiators, **ordered by total skill in descending order, then ordered by name in ascending order**. For each gladiator print their technique and skill, **ordered descending, then ordered by technique name in ascending order**

## Input / Constraints

You will receive an **array of strings** as a parameter to your solution.

- The input comes in the form of commands in one of the formats specified above.
- Gladiator and technique **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 **"Ave Cesar"**.



## Output

- The output format for each gladiator is:  
`"{gladiator}: {totalSkill} skill"`  
`"- {technique} <!!> {skill}"`

Scroll down to see examples.

## Examples

Input	Output	Comments
Pesho -> BattleCry -> 400 Gosho -> PowerPunch -> 300 Stamat -> Duck -> 200 Stamat -> Tiger -> 250 Ave Cesar	Stamat: 450 skill - Tiger <!!> 250 - Duck <!!> 200 Pesho: 400 skill - BattleCry <!!> 400 Gosho: 300 skill - PowerPunch <!!> 300	We order the gladiators by total skill points descending, then by name. We print every technique along its skill ordered descending by skill, then by technique name.
Input	Output	
Pesho -> Duck -> 400 Julius -> Shield -> 150 Gladius -> Heal -> 200 Gladius -> Support -> 250 Gladius -> Shield -> 250 Pesho vs Gladius Gladius vs Julius Gladius vs Gosho Ave Cesar	Gladius: 700 skill - Support <!!> 250 - Shield <!!> 250 - Heal <!!> 200 Pesho: 400 skill - Duck <!!> 400	Gladius and Pesho don't have common technique, so the duel isn't valid. Gladius wins vs Julius /common technique: "Shield". Julius is demoted. Gosho doesn't exist so the duel isn't valid. We print every gladiator left in the tier.

## 10. \*Game of Epicness

Write a JavaScript program that **determines** the **winner** from **all battles**. You will receive **two** arguments:

The **first** argument is an **array of kingdoms** with **generals** and their **army** in the form of an **object** with format:

```
{ kingdom: String, general: String, army: Number }
```

Every **general** has his own **army** that fights for a certain **kingdom**. Note that, every **kingdom's name** is **unique**, and every general's **name** is **unique** in **this kingdom**. If the **general** already **exists** in this **kingdom** **add** the **army** to his current one. After you go through all the kingdoms with their generals with armies and store the information about them, it's time to start the battles.

The **second** argument is **matrix of strings** showing which **kingdom's generals** are **fighting** in this format:

```
[
  ["{AttackingKingdom} ", "{AttackingGeneral}", "{DefendingKingdom} ", "{DefendingGeneral}" ],
  ...
]
```

The **first two elements** are the **names** of the **attacking general** from certain **kingdom** and the **second two** are the **names** of the **defending general** from certain **kingdom**. **Compare** the two general's **armies** to **determine** who **wins** and who **losses** based on who have the **larger army** **wins**. The **winner's army** **increases** with **10%** and the **loser's army** **decreases** with **10%**. Keep in mind to **round** them **down** if there is any excess **army** **after the battle**. If there is a **draw**, **do not do anything**. **Keep track** of the **wins** and **losses** for every general's battle.

Note that, **generals** from the **same kingdom** **cannot attack each other**.

After you finish with all battles you need to **find** which **kingdom** **wins** the game. To decide that, **first order** them by all their **general's wins** (**descending**) then by their **losses** (**ascending**), and finally by the **kingdom's name** in **ascending alphabetical** order.

## Input

You will receive **two arguments** – an **array of objects** with properties and a **matrix of strings** as shown above.

## Output

Print on the **console** the winning kingdom and **sort** the generals by their **armies in descending** order, **formatted** as seen in the examples.

## Constraints

- The **number of elements** in the **first input argument** will be in range **[1..100]** **inclusive**
- The **number of elements** in the **second input argument** will be in range **[0..100]** **inclusive**
- General's **army** will be always an **integer** in range **[0..1,000,000]** **inclusive**
- There **will be no invalid input**
- There **will be no matching number** of **armies** in the **output**

## Examples

Input
<pre>[ { kingdom: "Maiden Way", general: "Merek", army: 5000 },   { kingdom: "Stonegate", general: "Ulric", army: 4900 },   { kingdom: "Stonegate", general: "Doran", army: 70000 },   { kingdom: "YorckenShire", general: "Quinn", army: 0 },   { kingdom: "YorckenShire", general: "Quinn", army: 2000 },   { kingdom: "Maiden Way", general: "Berinon", army: 100000 } ], [ ["YorckenShire", "Quinn", "Stonegate", "Ulric"],   ["Stonegate", "Ulric", "Stonegate", "Doran"],   ["Stonegate", "Doran", "Maiden Way", "Merek"],   ["Stonegate", "Ulric", "Maiden Way", "Merek"],   ["Maiden Way", "Berinon", "Stonegate", "Ulric"] ]</pre>

Output
<p>Winner: Stonegate</p> <p>/\general: Doran</p> <p>---army: 77000</p> <p>---wins: 1</p> <p>---losses: 0</p> <p>/\general: Ulric</p> <p>---army: 5336</p> <p>---wins: 2</p> <p>---losses: 1</p>
Explanation
<p>After you successfully store the kingdoms information, the first battle's result is victory for the defender Ulric and a loss for the attacker Quinn. Second battle is ignored because the generals are from the same kingdom. Third battle is a victory for Doran and a loss for Merek. Fourth battle is a win for Ulric and a loss for Merek. Fifth battle is a victory for Berinon and a defeat for Ulric. All winners increase their armies with 10% for each win and all losers decrease their armies with 10% for each loss.</p> <p>The result from the battles are – Stonegate: 3 wins and 1 loss; Maiden Way: 1 win and 2 losses; YorkenShire: 0 wins and 1 loss. Making Stonegate the winner of the games because they have the most wins from kingdoms.</p>

Input
<pre>[ { kingdom: "Stonegate", general: "Ulric", army: 5000 },   { kingdom: "YorkenShire", general: "Quinn", army: 5000 },   { kingdom: "Maiden Way", general: "Berinon", army: 1000 } ], [ ["YorkenShire", "Quinn", "Stonegate", "Ulric"],   ["Maiden Way", "Berinon", "YorkenShire", "Quinn"] ]</pre>
Output
<p>Winner: YorkenShire</p> <p>/\general: Quinn</p> <p>---army: 5500</p> <p>---wins: 1</p> <p>---losses: 0</p>
Explanation
<p>The first battle between Quinn and Ulric is a draw because they have even armies because of that it is not recorded and their armies size does not change. The second battle is a win for Quinn and a loss for Berinon making YorkenShire the winner of the game with 1 win and 0 losses.</p>

Input
-------

```
[ { kingdom: "Maiden Way", general: "Merek", army: 5000 },
  { kingdom: "Stonegate", general: "Ulric", army: 4900 },
  { kingdom: "Stonegate", general: "Doran", army: 70000 },
  { kingdom: "YorkenShire", general: "Quinn", army: 0 },
  { kingdom: "YorkenShire", general: "Quinn", army: 2000 } ],
[ ["YorkenShire", "Quinn", "Stonegate", "Doran"],
  ["Stonegate", "Ulric", "Maiden Way", "Merek"] ]
```

#### Output

Winner: Maiden Way

/\general: Merek

---army: 5500

---wins: 1

---losses: 0