# Bootstrap and Front End Basics Lab

## Sort Array

Write a function that **sorts an array** with **numeric** values in **ascending** or **descending** order, depending on an **argument** that is passed to it.

You will receive a **numeric array** and a **string** as arguments to the first function in your code.

* If the second argument is asc, the array should be sorted in **ascending order** (smallest values first).
* If it is desc, the array should be sorted in **descending order** (largest first).

### Input

You will receive a **numeric array** and a **string** as input parameters.

### Output

The output should be the **sorted array**.

### Examples

|  |  |
| --- | --- |
| Input | Output |
| [14, 7, 17, 6, 8], 'asc' | [6, 7, 8, 14, 17] |
| [14, 7, 17, 6, 8], 'desc' | [17, 14, 8, 7, 6] |

## Argument Info

Write a function that displays **information** about the **arguments** which are passed to it (**type** and **value**) and a **summary** about the number of each type in the following format:

"{argument type}: {argument value}"

Print **each** argument description on a **new line**. At the end print a **tally** with counts for each type in **descending order**, each on a **new line** in the following format:

"{type} = {count}"

If two types have the **same count**, use **order of appearance**.

Do **NOT** print anything for types that do not appear in the list of arguments.

### Input

You will receive a series of arguments **passed** to your function.

### Output

**Print on the console** the **type** and **value** of each argument passed into your function.

### Example

|  |
| --- |
| Input |
| 'cat', 42, function () { console.log('Hello world!'); } |
| Output |
| string: cat  number: 42  function: function () { console.log('Hello world!'); }  string = 1  number = 1  function = 1 |

## Personal BMI

A wellness clinic has contacted you with an offer - they want you to write a program that composes **patient charts** and performs some preliminary evaluation of their condition. The data comes in the form of **several arguments**, describing a person - their **name**, **age**, **weight** in kilograms and **height** in centimeters. Your program must compose this information into an **object** and **return** it for further processing.

The patient chart object must contain the following properties:

* name
* personalInfo, which is an object holding their age, weight and height as properties
* BMI - body mass index. You can find information about how to calculate it here: <https://en.wikipedia.org/wiki/Body_mass_index>
* status

The status is one of the following:

* underweight, for BMI less than 18.5;
* normal, for BMI less than 25;
* overweight, for BMI less than 30;
* obese, for BMI 30 or more;

Once the BMI and status are calculated, you can make a recommendation. If the patient is obese, add an additional property called recommendation and set it to “admission required”.

### Input

Your function needs to take four arguments - name, age, weight and height

### Output

Your function needs to **return** an **object with properties** as described earlier. All numeric values should be **rounded** to the nearest whole number. All fields should be named **exactly as described** (their order is not important).  
Look at the sample output for more information.

|  |  |
| --- | --- |
| Input | Output |
| “Peter”, 29, 75, 182 | { name: 'Peter',  personalInfo: {  age: 29,  weight: 75,  height: 182  }  BMI: 23  status: 'normal' } |
| “Honey Boo Boo”, 9, 57, 137 | { name: 'Honey Boo Boo', personalInfo: { age: 9, weight: 57, height: 137 }, BMI: 30, status: 'obese', recommendation: 'admission required' } |

## 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.

### Input

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.**

### Output

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"]}] |
| ['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.



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



* 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.



* 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 **3rd element** of the currentHeroArguments array, it will **result in an error**. That is why we need to perform a simple check.



* 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**.



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



## 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**.

### Input

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

### Output

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** |
| ['{"name":"Pesho","position":"Promenliva","salary":100000}',  '{"name":"Teo","position":"Lecturer","salary":1000}',  '{"name":"Georgi","position":"Lecturer","salary":1000}'] | <table>  <tr>  <td>Pesho</td>  <td>Promenliva</td>  <td>100000</td>  </tr>  <tr>  <td>Teo</td>  <td>Lecturer</td>  <td>1000</td>  </tr>  <tr>  <td>Georgi</td>  <td>Lecturer</td>  <td>1000</td>  </tr>  </table> |

### Hints

* You might want to **escape the HTML**. Otherwise you might find yourself victim to vicious JavaScript **code in the input**.

## 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**.

### Input

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

“{juiceName} => {juiceQuantity}”

### Output

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