Exercises: Functions, Object Composition, Revealing Modules

Problems for exercises and homework for the "JavaScript Advanced" course @ SoftUni. Submit your solutions in the SoftUni judge system at https://judge.softuni.bg/Contests/299/.

1. Sort Array

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

Input

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**, then the array is sorted in **descending order** (largest first).

Output

The output should be the **return value** of your function and it is the **sorted array**.

Examples

Sample Input	Output
`solve([14, 7, 17, 6, 8], 'asc');`	`[6, 7, 8, 14, 17]`
`solve([14, 7, 17, 6, 8], 'desc');`	`[17, 14, 8, 7, 6]`

Hints

Arrays in JavaScript are by default sorted alphabetically, which means if we have an array of numbers [3, 1, 2, 10] and we call **sort()** on it, the result will be [1, 10, 2, 3]. You can however pass a **sorting criteria** as an argument in the form of a **function**. This function can be anonymously defined inline or a named function, or even a variable that holds a reference to a function:

JavaScript

`array.sort(function (a, b) {`
`    // TODO`
`});`

Let's start by building the body of our main function, which takes **two arguments** and returns a sorted array, using the default sorting strategy:

```
function sortArray(inputArray, sortMethod) {
    return inputArray.sort();
}
```

If you test this function, you'll see it uses the ASCII values to sort the objects inside the array. Next we need to pass an argument to the **sort()** function to get the desired result. It will consist of a **special function** which takes two arguments (**current element** and **next element** to be sorted), compares them and **returns a value**. If the value is **zero**, then they are equal. If it's **greater than zero**, the first element is larger. If it's **less than zero**, this means the second element is larger. In short, return a positive value to swap elements and zero or negative to keep the current order. Let's first implement ascending order:

```
function sortArray(inputArray, sortMethod) {
    return inputArray.sort((a, b)=> a - b);
}
```

What we did is define an anonymous function directly in the argument space. Note we could have explicitly written a conditional statement, which returns **1** if **a** is greater than **b**. However, we know that if this is true, the result of the expression above will be positive, so we can use this much shorter version. If we want descending order, all we have to do is swap **a** and **b** in the expression.

Let's use the functional nature of JavaScript and define the two comparator functions beforehand and assign them to variables:

```
var ascendingComparator = function (a, b) {
    return a - b;
};

var descendingComparator = function (a, b) {
    return b - a;
};
```

We can now pass either of them to the sorting function, depending on what we need. We'll save those in an object and use a string as a key, which would match the input shown in the problem description:

```
var sortingStrategies = {
    'asc': ascendingComparator,
    'desc': descendingComparator
};
```

Now whenever we need a new sorting method, we can just define it as a separate function and add it to this object with its corresponding key. No further change will be necessary. Finally, we modify our initial call to `sort()` to receive one of the stored functions, depending on the second argument of our main function:

```
function sortArray(inputArray, sortMethod) {

    var ascendingComparator = function (a, b) {
        return a - b;
    };

    var descendingComparator = function (a, b) {
        return b - a;
    };

    var sortingStrategies = {
        'asc': ascendingComparator,
        'desc': descendingComparator
    };

    return inputArray.sort(sortingStrategies[sortMethod]);
}
```

We are ready to submit our solution to the Judge.

2. 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.

Input

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

Output

Log to the **console** the **type** and **value** of each argument passed in the following format:

```
{argument type}: {argument value}
```

Place **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 format `{type} = {count}` If two types have the **same count**, use **order of appearance**. Don't print anything for types that do not appear in the list of arguments.

Examples

Sample Input
`result('cat', 42, function () { console.log('Hello world!'); });`
Output

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

Hints

JavaScript functions have a special property **`arguments`**, which contains all parameters passed to a function, regardless of whether you've specified them in the function declaration, or left the parenthesis empty.

JavaScript

function myFunc() {
    var firstArgument = arguments[0];
}

We can iterate this variable like an array to get access to every parameter in the order in which they were passed and inspect them:

```
for (var i = 0; i < arguments.length; i++) {
    var obj = arguments[i];
    var type = typeof obj;
    console.log(type + ': ' + obj);
```

We can use an object as an associative array to store the number of each type occurrence. Each type will be a property and its value will be the number of times it occurs in the arguments. We can access them just like we would the keys of an array:

```
if (!summary[type]) {
    summary[type] = 1;
} else {
    summary[type]++;
}
```

Since object properties cannot be sorted, and even if they could, different JavaScript implementations iterate the order differently, we need to transfer the information to an array of key-value pairs. We could use a Map instead of an object, but this cannot be sorted either, so we'll end up with an array in the end anyway.

```
var sortedTypes = [];
for (var currentType in summary) {
    sortedTypes.push([currentType, summary[currentType]]);
}
```

Note we are pushing an array with two values to the array which needs to be sorted. Later when we implement a sorting function, we'll use the second value of the key-value pair – the number of occurrences. All we need to do after the array is sorted is to output the information in the correct format.

3. Functional Sum

Write a function that **adds** a number passed to it to an **internal sum** and returns **itself** with its internal sum set to the **new value**, so it can be **chained** in a functional manner.

Example

Sample Input	Sample Output
`console.log(add(1));`	1
`console.log(add(1)(6)(-3));`	4

Input

Your function needs to take one **numeric argument**.

Output

Your function needs to **return** itself with updated context.

Hints

Making a function return itself is easy enough, but to keep a sum that's shared across all instances requires some effort. You'll need to place it inside a closure and expose just the function. Finally, to get the stored value, you'll have to override the built-in **`toString()`** method that all JavaScript objects have so that it returns the internal sum – this will allow any other function to access it either for printing or to use it in an expression, without being able to modify it. You can attach it directly to your function from inside the closure:

```
add.toString = function () {
    return sum;
};
```

Note that NodeJS will not implicitly call **`toString()`** when you try to log a value to the console. Keep this in mind when testing your solution locally.

4. Personal BMI

A wellness clinic has contacted you with an offer – they want you to write for them 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.

Sample Input	Sample 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' }`

5. Vector Math

Write several functions for preforming **calculations** with **vectors** in 2D space $\vec{a} = (x, y)$ and collect them all in a **single object** (namespace), so they don't pollute the global scope. Implement the following functions:

- **`add(vec1, vec2)`** – Addition of two vectors – $f(\vec{a}, \vec{b}) = \overrightarrow{\begin{pmatrix} x_a + x_b \\ y_a + y_b \end{pmatrix}}$
- **`multiply(vec1, scalar)`** – Scalar multiplication – $f(\vec{a}, s) = \overrightarrow{\begin{pmatrix} x_a \times s \\ y_a \times s \end{pmatrix}}$
- **`length(vec1)`** – Vector length – $f(\vec{a}) = \sqrt{x_a^2 + y_a^2}$
- **`dot(vec1, vec2)`** – Dot product of two vectors – $f(\vec{a}, \vec{b}) = x_a x_b + y_a y_b$
- **`cross(vec1, vec2)`** – Cross product of two vectors – $f(\vec{a}, \vec{b}) = x_a y_b - y_a x_b$

The math-savvy may notice that the given cross product formula results in a scalar, instead of a vector – we're actually only measuring the length of the resulting vector, since cross product is not possible in 2D, it will exist purely in the z-dimension. If you don't know what this all means, ignore this paragraph, it's irrelevant to the solution.

Input

Each separate function in your namespace will be tested with individual values. It must expect **one or two arguments**, as described above, and **return** a value. Vectors will be 2D **arrays** with format `[x, y]`.

Output

Your program needs to **return** an object, containing **all functions** described above. Each individual function must **return** a value, as required. Don't round any values.

Sample Input	Output	Explanation
`solution.add([1, 1], [1, 0]);`	`[2, 1]`	`[1 + 1, 1 + 0] = [2, 1]`
`solution.multiply([3.5, -2], 2);`	`[7, -4]`	`[3.5 * 2, (-2) * 2] = [7, -4]`
`solution.length([3, -4]);`	`5`	`sqrt(3 * 3 + (-4) * (-4)) = 5`
`solution.dot([1, 0], [0, -1]);`	`0`	`1 * 0 + 0 * (-1) = 0`
`solution.cross([3, 7], [1, 0]);`	`-7`	`3 * 0 – 7 * 1 = -7`

6. Breakfast Robot

It's finally the future! Robots take care of everything and man has been freed from the mundane tasks of living. There is still work to be done though, since those robots need to be programmed first – we may have robot chefs, but we don't yet have robot software developers.

Your task is to write the management software for a breakfast chef robot – it needs to **take orders**, keep track of available **ingredients** and output an error if something's wrong. Someone else has already installed the cooking instructions, so your module needs to **plug into** the system and only take care of orders and ingredients. And since this is the future and food is printed with nano-particle beams, all ingredients are microelements – **protein**, **carbohydrates**, **fat** and **flavours**. The library of recipes includes the following meals:

- **`Apple`** – made with **1 carb** and **2 flavour**
- **`Coke`** – made with **10 carb** and **20 flavour**
- **`Burger`** – made with **5 carb**, **7 fat** and **3 flavour**
- **`Omelet`** – made with **5 protein**, **1 fat** and **1 flavour**
- **`Cheverme`** – made with **10 protein**, **10 carb**, **10 fat** and **10 flavour**

The robot receives instructions either to restock the supply, cook a meal or report statistics. The input consists of one of the following commands:

- **`restock <microelement> <quantity>`** – increases the stored quantity of the given microelement
- **`prepare <recipe> <quantity>`** – use the available ingredients to prepare the given meal
- **`report`** – return information about the stored microelements, in the order described below, including zero elements

The robot is equipped with a quantum field storage, so it can hold an **unlimited quantity** of ingredients, but there is no guarantee there will be enough available to prepare a recipe, in which case an **error message** should be returned. Their availability is checked in the **order** in which they **appear** in the recipe, so the error should reflect the first requirement which wasn't met.

Submit a **closure** that returns the management function. The management function must take one parameter.

Input

Instructions are passed as a **string argument** to your management function. It will be called **several times** per session, so internal state must be **preserved** throughout the entire session.

Output

The **return** value of each operation is one of the following strings:

- **`Success`** – when restocking or completing cooking without errors
- **`Error: not enough <ingredient> in stock`** – when the robot couldn't muster enough microelements
- **`protein={qty} carbohydrate={qty} fat={qty} flavour={qty}`** – when a report is requested, in a single string

Constraints

- Recipes and ingredients in commands will always have valid names.

Examples

Sample Execution

`let manager = solution();`
`manager("restock flavour 50"); // Success`
`manager("prepare coke 4");     // Error: not enough carbohydrate in stock`

Sample Input	Sample Output
restock carbohydrate 10	Success
restock flavour 10	Success
prepare apple 1	Success
restock fat 10	Success
prepare burger 1	Success
report	protein=0 carbohydrate=4 fat=3 flavour=5

Sample Input	Sample Output
prepare cheverme 1	Error: not enough protein in stock
restock protein 10	Success
prepare cheverme 1	Error: not enough carbohydrate in stock
restock carbohydrate 10	Success
prepare cheverme 1	Error: not enough fat in stock
restock fat 10	Success
prepare cheverme 1	Error: not enough flavour in stock
restock flavour 10	Success
prepare cheverme 1	Success
report	protein=0 carbohydrate=0 fat=0 flavour=0

7. Monkey Patcher

Your employer placed you in charge of an old forum management project. The client requests new functionality, but the legacy code has high coupling, so you don't want to change anything, for fear of breaking everything else. You know which values need to be accessed and modified, so it's time to monkey patch!

Write a program to extend a forum post record with voting functionality. It needs to have the options to **upvote**, **downvote** and tally the **total score** (positive minus negative votes). Furthermore, to prevent abuse, if a post has more than 50 **total votes**, the numbers must be obfuscated – the stored values remains the same, but the **reported** amounts of upvotes and downvotes have a number **added** to them. This number is 25% of the **greater number** of votes (positive or negative), rounded up. The actual numbers should **not be modified**, just the reported amounts.

Every post also has a **rating**, depending on its score. If **positive** votes are the overwhelming majority (>66%), the rating is **hot**. If there is no majority, but the balance is non-negative and **either** votes are more than 100, its rating is

`controversial`. If the balance is negative, the rating becomes **`unpopular`**. If the post has less than 10 **total** votes, or no other rating is met, it's rating is **new** regardless of balance. These calculations are performed on the actual numbers.

Your function will be invoked with **`call(object, arguments)`**, so treat it as though it is internal for the object. A forum post, to which the function will be attached, has the following structure:

JavaScript

```
{
  id: <id>,
  author: <author name>,
  content: <text>,
  upvotes: <number>,
  downvotes: <number>
}
```

The arguments will be one of the following strings:

- **`upvote`** – increase the positive votes by one
- **`downvote`** – increase the negative votes by one
- **`score`** – report positive and negative votes, balance and rating, in an array; obfuscation rules apply

Input

Input will be passed as arguments to your function through a **`call()`** invocation.

Output

Output from the report command should be **returned** as a result of the function in the form of an **array** of three **numbers** and a **string**, as described above.

Examples

Sample execution

```
let post = {
    id: '3',
    author: 'emil',
    content: 'wazaaaaa',
    upvotes: 100,
    downvotes: 100
};
solution.call(post, 'upvote');
solution.call(post, 'downvote');
let score = solution.call(post, 'score'); // [127, 127, 0, 'controversial']
solution.call(post, 'downvote'); …         // (executed 50 times)
score = solution.call(post, 'score');     // [139, 189, -50, 'unpopular']
```

Explanation

The post begins at 100/100, we add one upvote and one downvote, bringing it to 101/101. The reported score is inflated by 25% of the greater value, rounded up (26). The balance is 0, and at least one of the numbers is greater than 100, so we return an array with rating 'controversial'.

We downvote 50 times, bringing the score to 101/151, the reported values are inflated by 151*0.25=38 (rounded up) and since the balance is negative with return an array with rating 'unpopular'.

8. * Euclid's Algorithm

Write a program that receives **two numbers** as arguments and finds the **greatest common divisor** between them.

Input

Input will be passed as two **numeric arguments** to your function.

Output

Return the greatest common divisor as a result of the function.

Examples

Sample Input	Sample Output
`252, 105`	`21`

9. *** Kepler's Problem

Write a function that, given the mean anomaly and orbital eccentricity of a celestial body, calculates its eccentric anomaly. The eccentric anomaly ***E*** is related to the mean anomaly ***M*** by Kepler's equation:

$$M = E - e \sin E$$

Where ***e*** is the eccentricity. Note this equation is transcendental, which means it cannot be solved for ***E*** algebraically. Use numerical analysis to approximate a root with accuracy $1x10^{-9}$. You can find information about Newton's Method here: https://en.wikipedia.org/wiki/Newton's_method. Try to implement it recursively.

The **input** comes as two number parameters. The first parameter is the current mean anomaly in radians and the second is the orbital eccentricity of the body.

The **output** is an approximation of the eccentric anomaly and should be printed on the console. Display only the significant digits.

Examples

Input	Output
`1, 0`	`1`

Input	Output
`3.1415926535, 0.75`	`3.141592654`

Input	Output
`0.25, 0.99`	`1.156077258`

Input	Output
`4.8, 0.2`	`4.601234265`

Hints

Newton’s method works with functions that equal zero. We shift the variables around to arrive at the following form:

$$f(E) = E - e\sin E - M = 0$$

Not coincidentally, this is also our progress check – as we look for a closer approximation for ***E***, the solution of this equation will be closer to zero. Once it’s within the aforementioned ***epsilon*** (required accuracy), we stop iterating and print the result. When implementing recursively, this condition will be the bottom of our recursion. The last bit we need is the first derivative of the function:

$$f'(E) = (1 - e\cos E)$$

And to plug it all into Newton’s equation:

$$E_1 = E_0 - \frac{f(E_0)}{f'(E_0)}$$

Where $\boldsymbol{E_0}$ is the result of the previous iteration and $\boldsymbol{E_1}$ will be the result of the current iteration. When beginning the iteration, pick an initial value for $\boldsymbol{E_0}$ that might be close enough to our desired result (chose a value that is either zero or equal to the mean anomaly).