1. Write a function to get the most common character in the string (use function composition).

2. We have an array, filter the even numbers from the array. Then, calculate a sum of all the remaining numbers, and after that print sum of numbers in this format `The result is ${result}.` (use function composition).

3. Create a function that takes a list of functions and sorts them in ascending order based on how many calls are needed for them to return a non-function.

```
f1 = _ => "hello"
// f1() ➞ "hello"

f2 = _ => _ => "world"
// f2()() ➞ "world"

f3 = _ => _ => _ => "user"
// f3()()() ➞ "user"

funcSort([f2, f3, f1]) ➞ [f1, f2, f3]
// [f2, f3, f1] ➞ [2, 3, 1] ➞ [1, 2, 3] ➞ [f1, f2, f3]

funcSort([f1, f2, f3]) ➞ [f1, f2, f3]
// [f1, f2, f3] ➞ [1, 2, 3] ➞ [1, 2, 3] ➞ [f1, f2, f3]

funcSort([f2, "func"]) ➞ ["func", f2]
// [f2, "func"] ➞ [2, 0] ➞ [0, 2] ➞ ["func", f2]
```

4. Create a function that groups an array of numbers based on a size parameter. The size represents the maximum length of each sub-array.

```
 [1, 2, 3, 4, 5, 6], 3
[[1, 3, 5], [2, 4, 6]]
// Divide array into groups of size 3.

[1, 2, 3, 4, 5, 6], 2
[[1, 4], [2, 5], [3, 6]]
// Divide array into groups of size 2.

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], 4
[[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9]]
// "Leftover" arrays are okay.

group([1, 2, 3, 4], 2) ➞ [[1, 3], [2, 4]]

group([1, 2, 3, 4, 5, 6, 7], 4) ➞ [[1, 3, 5, 7], [2, 4, 6]]
```

```
group([1, 2, 3, 4, 5], 1) ➞ [[1], [2], [3], [4], [5]]

group([1, 2, 3, 4, 5, 6], 4) ➞ [[1, 3, 5], [2, 4, 6]]
```

The size parameter represents the maximum size for each sub-array (see ex.4).
You should try to fill each sub-array evenly. In other words, ex.4 should be `[[1, 3, 5], [2, 4, 6]]`, not `[[1, 3, 5, 6], [2, 4]]`.

Keep the relative order of the numbers in each sub-array the same as the order in the original array.

5. Write `Pagination` object that will get an `array` and `pageSize`, then will return the following.

    - nextPage
    - prevPage
    - firstPage
    - lastPage
    - goToPage
    - getPageItems

```
const alphabetArray = "abcdefghijklmnopqrstuvwxyz".split("");
Pagination.init(alphabetArray, 4);

Pagination.getPageItems(); // ["a", "b", "c", "d"]
Pagination.nextPage(); // add the current page by one

Pagination.getPageItems(); // ["e", "f", "g", "h"]

Pagination.nextPage().nextPage(); // the ability to call chainable

Pagination.goToPage(3); // current page must be set to 3
```

6. Write an Airplane object that initializes `name`.

    Give airplanes the ability to `.takeOff()` and `.land()`:

- If a plane lands, its `isFlying` property is set to false.
- If a plane takes off, its `isFlying` property is set to true.

7. Your task is to create a Circle constructor function that creates a circle with a radius provided by an argument. The circles constructed must have two methods getArea() (PI*r^2) and getPerimeter() (2*PI*r) which give both respective areas and perimeter (circumference).

```
let circy = new Circle(11);
circy.getArea();
// Should return 380.132711084365

let circy = new Circle(4.44);
circy.getPerimeter();
// Should return 27.897342763877365
```

8. Create a `gather` function that accepts a string argument and returns another function. The function calls should support continued chaining until the `order` is called.
   The `order` should accept a number as an argument and return another function. The function calls should support continued chaining until the `get` is called.
   `get` should return all of the arguments provided to the gather functions as a string in the order specified in the order functions.

```
gather("a")("b")("c").order(0)(1)(2).get() → "abc"

gather("a")("b")("c").order(2)(1)(0).get() → "cba"

gather("e")("l")("o")("l")("!")("h").order(5)(0)(1)(3)(2)(4).get()
→ "hello!"
```

   Expect to `gather` and `order` to chain any number of times, but always with the same number of invocations for each (i.e both functions will always be chained the same number of times).

9. A **complete bracelet** is an array with at least one subsequence (pattern) repeating *at least two times*, and *completely* - the subsequence cannot be cut-off at any point. The subsequence **must have a length two or greater**.

```
[1, 2, 3, 3, 1, 2, 3, 3]  // Pattern: [1, 2, 3, 3]

[1, 2, 1, 2, 1, 2, 1, 2]  // Pattern: [1, 2] or [1, 2, 1, 2]

[1, 1, 6, 1, 1, 7, 1, 1, 6, 1, 1, 7, 1, 1, 6, 1, 1, 7]  // Pattern:
[1, 1, 6, 1, 1, 7]

[4, 4, 3, 4, 4, 4, 4, 3, 4, 4]  // Pattern: [4, 4, 3, 4, 4]
```

**Incomplete bracelets:**

```
[1, 2, 2, 2, 1, 2, 2, 2, 1]  // Incomplete (chopped off)

[1, 1, 6, 1, 1, 7]  // Incomplete (subsequence repeats only once)
```

Create a function that returns `true` if a bracelet is complete, and `false` otherwise.

```
completeBracelet([1, 2, 2, 1, 2, 2]) → true

completeBracelet([5, 1, 2, 2]) → false

completeBracelet([5, 5, 5]) → false
// potential pattern [5, 5] cut-off (patterns >= 2)
```

Patterns must be at least two integers in length.

10. Create a function that takes an array of students and returns an object representing their notes distribution. Have in mind that all invalid notes should not be count in the distribution. Valid notes are: 1, 2, 3, 4, 5

```
getNotesDistribution([
        {
          "name": "Steve",
          "notes": [5, 5, 3, -1, 6]
        },
        {
          "name": "John",
          "notes": [3, 2, 5, 0, -3]
        }
```

```
] → {
  5: 3,
  3: 2,
  2: 1
})
```

11. Create a function that *recursively* counts the integer's number of digits.

```
count(318) → 3

count(-92563) → 5

count(4666) → 4

count(-314890) → 6

count(654321) → 6
```

12. Write recursive a function that, given the start `startNum` and end `endNum` values, return an array containing all the numbers **inclusive** to that range. See examples below.

```
inclusiveArray(1, 5) → [1, 2, 3, 4, 5]

inclusiveArray(2, 8) → [2, 3, 4, 5, 6, 7, 8]

inclusiveArray(10, 20) → [10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20]

inclusiveArray(17, 5) → [17]
```