// Subsets

Write a function that takes an array and returns all of its subsets. How many sets will it return?

arrays have strings of length 1

```
subsets(['a', 'b', 'c'])
```

```
[['a'], ['c', 'a'], ['b', 'a'], ['c', 'b', 'a']]
```

```
const subsets = (arr) => {
	let responseArray = []
	for(let i = 0; i < arr.length; i++) {
		let sub = []
		for(let j = i; j < arr.length; j++) {
			let slc = arr.slice(i, j + 1)
			sub.push(slc)
		}
		responseArray.push(sub)
	}
	return responseArray
}
```

Given arr1 and arr2, find the intersection of both sets. It should be trivial to write an O(n**2) solution. Use sorting to solve in O(nlog(n)). Next, improve this to O(n) time (maybe use a non-array datastructure).

["banana", "potato", "papayas"] ["lazers", "elevator", "potato"]

O(n**2) Solution

```
fastIntersection(arr1, arr2) {
	Arr1.filter( (ele) => {arr2.includes(ele)} )
}
```

O(n) Solution

```
fastIntersection(arr1, arr2) {
	Const setArr = new Set(arr1);
	Let selected = [];

	For (let i = 0; i < arr2.length; i++) {
		Let ele = arr2[i];

		If (setArr.includes(ele)) return selected.push(ele);
	}
}
```

Non-Comparison Sorts

Part 1: Say that I gave you an array of length `n`, containing the numbers `1..n` in jumbled order. "Sort" this array in `O(n)` time. You should be able to do this without looking at the input.

Part 2: Say that I give you an array of length `n` with numbers in the range `1..N` (`N >= n`). Sort this array in `O(n + N)` time. You may use `O(N)` memory.

Part 3: Say I give you an array of `n` strings, each of length `k`. I claim that, using merge sort, you can sort this in `O(knlog(n))`, since comparing a pair of strings takes `O(k)` time.

I want you to beat that. Sort the strings in `O(kn)`. **Hint**: do not compare any two strings. You may assume all strings contain only lowercase letters `a..z` without whitespace or punctuation.

```
const sort1 = (arr) => {
	return arr.map((num, i) => num = i + 1);
}

const sort2 = (arr) => {
	let arrSet = new Set(arr)
	let sorted = []
	for(let i = 1; i < arr.length; i++) {
		if (arrSet.includes(i)) {
			sorted.push(i)
		}
	}
	return sorted
}

const sort3 = (arr) => {
	let arrSet = new Set(arr)
	let sorted = []
	let i = 1
	while(sorted !== arr) {
		sorted.concat(arrSet.filter((str) => str === i))
		i++
	}
	return sorted
}
```

Implement the 'look and say' function. 'Look and say' takes an input array and outputs an array that describes the count of the elements in the input array as they appear in order.

```
# there is one '1' in the input array
look_and_say([1]) == [[1, 1]]

# there are two '1's in the input array
look_and_say([1, 1]) == [[2, 1]]

# there is one '2', followed by one '1' in the input array
look_and_say([2, 1]) == [[1, 2], [1, 1]]

# is one '1', followed by one '2', followed by 2 '1's in the input
# array
look_and_say([1, 2, 1, 1]) == [[1, 1], [1, 2], [2, 1]]


lookAndSee(arr) {
	Let match = [];
	Let count = 1;

	For (let j = 0; j < arr.length; j++) {

		If (arr[j] === arr[j + 1] ) {
			Count += 1;
		} else {
			match.push([count, arr[j]])
			Count = 1;
		}
	}

	Return match;
}
```

Sums Upon Sums

I give you a scrambled list of n unique integers between 0 and n. Tell me what number is missing.
If I let you use O(nlog(n)) time, what is a naive way of doing this?
Next, what if I require that you solve the problem in O(n) time? What datastructure might you use?
Finally, how could you solve the problem in O(n), and also O(1) space?

// sumsOnSums([1, 2, 4, 5]) => 3

// look up if you can run forEach on a set and if converting an array into a set is considered o(n) or o(1)

```
const sumsOnSums = (arr) => {
	length = arr.length
	arr = new Set(arr)
	for(let i = 0; i < length; i++) {
		if (!arr.has(i)) return i
	}
	return null
}
```

StackQueue
Implement a queue using stacks. That is, write enqueue and dequeue using only push and pop operations.

In terms of performance, enqueue should be O(1), but dequeue may be worst-case O(n). In terms of ammortized time, dequeue should be O(1). Prove that your solution accomplishes this.

```
Class StackQueue {
	constructor() {
		This.in = [];
		this.out = []
	}


	enqueue(ele) {
		this.queue.push(ele);
	}

	dequeue() {
		if (this.out.length === 0) {
			this.out.push(this.in.pop())
		}
		return this.out.pop()
	}
}
```