**Frequency Counter**

*Purpose*: this pattern uses objects or sets to collect values/frequencies of values. This can often avoid the need for nested loops or O(n^2) operations with arrays/strings.

*Example*: Write a function , which accepts 2 arrays. The function should return true if every value in the array has it’s corresponding value squared in the second array. The frequency of values must be the same.

*Solution*:

function same(arr1, arr2){

if(arr1.length !== arr2.length){

return false;

}

let frequencyCounter1 = {}

let frequencyCounter2 = {}

for(let val of arr1){

frequencyCounter1[val] = (frequencyCounter1[val] || 0) + 1

}

for(let val of arr2){

frequencyCounter2[val] = (frequencyCounter2[val] || 0) + 1

}

for(let key in frequencyCounter1){

if(!(key \*\* 2 in frequencyCounter2)){

return false

}

if(frequencyCounter2[key \*\* 2] !== frequencyCounter1[key]){

return false

}

}

return true

}

**Multiple Pointers**

*Purpose:* creating pointers or values that correspond to an index or position and move towards the beginning, end or middle based on a certain solution. Very efficient for solving problems with minial space complexity.

*Example:* Write a function which accepts a sorted array of integers. The function should find the first pair where the sum is 0. Return an array that includes both values that sum to zero or undefined if a pair does not exist.

*Solution:*

function sumZero(arr) {

let left = 0;

let right = arr.length – 1;

while (left < right) {

let sum = arr[left] + arr[right];

if (sum == 0) {

return arr[left], arr[right];

} else if (sum > 0) {

right--;

} else {

left++;

}

}

}

**Sliding Window**

*Purpose:* this pattern involves creating a window which can either be an array or number from one position to another.

Depending on a certain condition, the window either increases or closes (and a new window is created)

Very useful for keeping track of a subset of data in an array/string etc

*Example:* Write a function which accepts an array of integers and a number. The function should calculate the maximum sum of n consecutive elements in the array.

*Solution:*

function maxSubarraySum(arr, num){

let maxSum = 0;

let tempSum = 0;

if (arr.length < num) return null;

for (let i = 0; i < num; i++) {

maxSum += arr[i];

}

tempSum = maxSum;

for (let i = num; i < arr.length; i++) {

tempSum = tempSum - arr[i - num] + arr[i];

maxSum = Math.max(maxSum, tempSum);

}

return maxSum;

}

**Divide and Conquer**

*Purpose:* This pattern involves dividing a data set into smaller chunks and then repeating a process with a subset of data. This pattern can tremendously decrease time complexity.

*Example:* Given a sorted array of integers, write a function called search, that accepts a value and returns the index where the value passed to the function is located. If the value not found, return – 1.