**Algorithm & Problem Solving Patterns**

**Some Common Problem Solving Patterns:-**

* Frequency Counter
* Multiple Pointers
* Sliding Window
* Divide and Conquer
* Dynamic Programming
* Greedy Algorithms
* Backtracking
* *Many More…*

All of these are different approaches, and different patterns for writing code.

Patterns means to say, a sort of programming mechanisms, different pieces that we can put in or say blue print.

**FREQUENCY COUNTER PATTERN**

This patterns 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-2:**

**Q. Write a function called same, which accepts two 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.**

**Test Case:**same([1,2,3], [4,1,9]) // true  
same([1,2,3], [1,9]) // false  
same([1,2,1], [4,4,1]) // false (must be same frequency

**Answers:-**

**#NAIVE SOLUTION**

function same(arr1, arr2){

if(arr1.length !== arr2.length){  
 return false;  
 }

for(let i = 0; i < arr1.length; i++){ //***O(n)***  
 let correctIndex = arr2.indexOf(arr1[i] \*\* 2) //***O(n)***

if(correctIndex === -1) {  
 return false;  
 }

arr2.splice(correctIndex,1);  
 }

return true  
}

same([1,2,3], [4,1,9]);

**Output:**true

**Time Complexity: O(n2)**

**Solution:**

**#REFACTORED**(Actual: Frequency Counter Pattern)

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  
}

same([1,2,3], [4,1,9]);  
**Output:**true

**Time Complexity: O(n)**

**Imp. Points:-**

* The order doesn’t matter, so it doesn’t have to be identical, just squared. It could be mixed up, but the frequency of values must be the same.
* Frequency means the squared of every element in the first array/object should be there in the second one.
* Naïve means not the best possible solution.

**Example-2 :**

**Frequency Counter Pattern: Anagram Challenge**

Given two strings, write a function to determine if the second string is an anagram of the first.  
An anagram is a word, phrase, or name formed by rearranging the letters of another, such as cinema, formed from iceman.

**Test Case:**validAnagram('', '') // true  
validAnagram('aaz', 'zza') // false  
validAnagram('anagram', 'nagaram') // true  
validAnagram("rat","car") // false) // false  
validAnagram('awesome', 'awesom') // false  
validAnagram('qwerty', 'qeywrt') // true  
validAnagram('texttwisttime', 'timetwisttext') // true

**Solution:**

function validAnagram(str1, str2){

let fstObjt = {}; ***//O(1)***  
 let secObjt = {}; ***//O(1)***

for(let arr1 of str1){ ***//O(n)***  
 arr1 = arr1.toLowerCase();  
 fstObjt[arr1] = (fstObjt[arr1] || 0) +1;   
 }

for(let arr2 of str2){ ***//O(n)***  
 arr2 = arr2.toLowerCase();  
 secObjt[arr2] = (secObjt[arr2] || 0) +1;  
 }

for(let key in fstObjt){ ***//O(n)***  
 if(!(key in secObjt)){  
 return false;  
 }

if(fstObjt[key] !== secObjt[key]){  
 return false;  
 }  
 }

return true; ***//O(1)***

}

validAnagram('chandan', 'Ncandah');  
**Output:**true

**Time Complexity: O(n)**

**MULTIPLE POINTERS PATTERN**

Creating pointers or values that correspond to an index or position and move towards the beginning, end or middle based on a certain condition.

Very efficient for solving problems with minimal space complexity as well.

**Example - 1:**

Write a function called sumZero 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.

**Test Case:**sumZero([-3,-2,-1,0,1,2,3]) // [-3,3]  
sumZero([-2,0,1,3]) // undefined  
sumZero([1,2,3]) // undefined

**Solution:**

**#Naive Solution**

function sumZero(arr){

for(let i=0; i<arr.length; i++){ **//O(n)**

for(let j=i+1; j<arr.length; j++){ **//O(n)**  
 let sum = arr[i] +arr[j];  
 if(sum === 0){  
 return [arr[i], arr[j]];  
 }

}

}

}  
sumZero ([-3,-2,-1,0,1,2,3]);

**Output:**[-3, 3];

**Time Complexity: O(n2)  
Space Complexity: O(1)**

**#REFACTORED**(Actual: Multiple Pointer Pattern)

**Solution:**

function sumZero(arr){

let left = 0; **//O(1)**  
 let right = arr.length-1; **//O(1)**

while(left < right){ **//O(n)**

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

if(sum === 0){  
 return [arr[left], arr[right]];  
 }else if(sum>0){  
 right--;  
 }else{  
 left++;  
 }  
 }

}

sumZero([-3,-2,-1,0]);

**Output:**[-3, 3];

**Time Complexity: O(n)  
Space Complexity: O(1)**

**Imp. Points:**

* We’ve a pointer, which is just a variable. We’re looking at specific location in an array or a string usually, and then we’ve a second one and we work towards each other or we work in the same direction. We work from the ends, towards the beginning , it doesn’t matter. But we used to pointers.
* It needs to be *sorted, not assorted*, but in order sorted from lowest to highest the function should find the first pair where the sum is zero.

**Multiple Pointer Pattern: countUniqueValues Challenge**

**Example-2 :**

Implement a function called countUniqueValues, which accepts a sorted array, and counts the unique values in the array. There can be negative numbers in the array, but it will always be sorted.

**Test Cases:**countUniqueValues([1,1,1,1,1,2]) **// 2** {total 1 & 2 are unique values}  
countUniqueValues([1,2,3,4,4,4,7,7,12,12,13]) **// 7** { 1,2,3,4,7,12,13 are unique values}  
countUniqueValues([]) **// 0** {Zero Numbers of Unique Values}  
countUniqueValues([-2,-1,-1,0,1]) **// 4** {-2-1, 0 &1 are unique values. Total: 4}

**Solution:**

function countUniqueValues(arr){

let count = 0  ***//O(1)***

for(let i=0;i<arr.length;i++){ ***//O(n)***  
 let down = arr[i+1];  
 let up = arr[i];

if(up!==down){  
 count+=1;   
 }else{  
 down++;  
 }

}

return count;

}

countUniqueValues([1,2,3,4,4,4,7,7,12,12,13]); ***//array should be sorted***

**Output:**7

**Time Complexity: O(n)**

**SLIDING WINDOW**

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 called maxSubarraySum which accepts an array of integers and a number called n. The function should calculate the maximum sum of n consecutive elements in the array.

**Note:**‘**n**’ will be the number of digits are trying to sum together  
A **SubArray** is a contiguous part of an array.

**Test Cases:**maxSubarraySum([1,2,5,2,8,1,5],2) // 10  
maxSubarraySum(([1,2,5,2,8,1,5]),4) // 17  
maxSubarraySum([4,2,1,6],1) // 6  
maxSubarraySum([4,2,1,6,2],4) // 13  
maxSubarraySum([],4) // null

**#NAIVE SOLUTION**

**Solution:**

function maxSubarraySum(arr, num) {

if ( num > arr.length){  
 return null;  
 }

var max = -Infinity;

for (let i = 0; i < arr.length - num + 1; i ++){ ***//arr.length-num+1 = 4 & T.C: O(n)***   
 temp = 0;

for (let j = 0; j < num; j++){ ***// T.C: O(n)***  
 temp += arr[i + j]; ***//Ex: arr[i+j] 🡪 arr[0+1] 🡪 arr[1]***  
 }

if (temp > max) {  
 max = temp;  
 }

}

return max;

}

maxSubarraySum([1,2,5,2,8,1,5],4);

**Output:**17

**Time Complexity: O(n2)**

**#REFACTORED**(Actual: Sliding Window Pattern)

**Solution:**

function maxSubarraySum(arr, num){

let maxSum = 0;  
 let tempSum = 0;

if (arr.length < num) return null;

for (let i = 0; i < num; i++) { ***//O(n)***  
 maxSum += arr[i];  
 }

tempSum = maxSum;

for (let i = num; i < arr.length; i++) { ***//O(n)***

tempSum = tempSum - arr[i - num] + arr[i];  
 maxSum = Math.max(maxSum, tempSum);

}

return maxSum;

}

maxSubarraySum([1,2,5,2,8,1,5], 4); ***//O(n)***

**Output:**17

**Time Complexity: O(n)**

**Divide and Conquer**

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

Quick Sort and Merge sort are examples of sorting algorithm & divide and Conquer algorithm as well.

Binary search is a Divide and Conquer algorithm. When we get to Binay Serach trees, we talk about dividing and conquering algoritm.

**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 is not found, return -1

**Test Cases:**search([1,2,3,4,5,6],4) // 3  
search([1,2,3,4,5,6],6) // 5  
search([1,2,3,4,5,6],11) // -1

**#Naive Solution -1**

**Solution:**

function search(arr, num){  
 return arr.indexOf(num);  
}

search([1,2,3,4,5,6],4);

**Output:**3

**#Naive Solution - 2**(Linear Search)

**Solution:**

function search(arr, num){

if(arr.length<num)return -1;

for(let i=0; i<arr.length;i++){  ***//O(n)***  
 if(arr[i]===num){  
 return i;  
 }

}

}

search([1,2,3,4,5,6],4);

**Output:**3

**Time Complexity: O(n)**

**#REFACTOR**(Binary Search)

Here, what we do is, we take a larger set of data, usually an array, or a string. It could be a linked list or a tree as well. For now we’re going to take an array.

In Binary Search, rather than starting from the left and moving all the way to the right,   
Let’s say we’re searching for a value. We start by dividing it into smaller pieces and then doing something to each smaller piece to determine where to go next. So we divide up a larger chunk of data into smaller chunks.

**Solution:**

function search(arr, num) {

if(arr.length<num) return -1;

let left = 0;  
 let right = arr.length - 1;

while(left<=right){

let middle = Math.floor((left + right)/2);

if(arr[middle]<num){  
 left = middle + 1;

}else if(arr[middle]>num){

right = middle -1;

}else{

return middle; ***//when both left & right will be equal at same index.***

}

}

}

search([1,2,3,4,5,6],4);

**Output:**3

**Time Complexity: log(n)**

**PROBLEMS RELATED TO THE PREVIOUS PATTERNS ========================================**

**Section-6 (Remaining Excercises) (Including Tick & Untick)**