S5.Problem Solving Patterns

Pattern #1: Frequency Counter

1. Comparing frequency between arrays, or counting how many times it frequents
2. Multiple pieces of data, and you need to compare them
3. See if they consist of the same individual pieces
4. Example
    a. Write a function that test wether array1 elements are squared in array2
    b.
```
function same(arr1, arr2){
    if(arr1.length !== arr2.length){
        return false;
    } // short circuit
    for(let i = 0; i < arr1.length; i++){ // loop 1
        let correctIndex = arr2.indexOf(arr1[i] ** 2) // loop 2 (indexOf)
        if(correctIndex === -1) {
            return false;
        }
        arr2.splice(correctIndex, 1) // once it has been paired, array2 element i
s removed so as it is not counted twice
    }
    return true;
}

same([1,2,3,2], [9,1,4,4])
```
    c. Time complexity = O(n^2)
    d. We should avoid double loops, for performance
    e. Refactored
```
function same(arr1, arr2){
    // short circuit length comparison test
    if(arr1.length !== arr2.length){
        return false;
    }
```

```
    // initiate 2 object where we will store the count of array1 and
array2 elements
    let frequencyCounter1 = {}
    let frequencyCounter2 = {}

    // for each val of arr1
    for(let val of arr1){
        // let frequencyCounter1[key] be value +1, or initialized value at 1
        frequencyCounter1[val] = (frequencyCounter1[val] || 0) + 1
    }
    // for each val of arr2
    for(let val of arr2){
        // let frequencyCounter2[key] be value +1, or initialized value at 1
        frequencyCounter2[val] = (frequencyCounter2[val] || 0) + 1
    }
    console.log(frequencyCounter1); // {1:1, 2:2, 3:1, 5:1}
    console.log(frequencyCounter2); // {1:1, 4:2, 9:1, 11:1}


    // for each key of frequencyCounter1
    for(let key in frequencyCounter1){
        // if !(it is not that) key squared is in frequencyCounter2 => false
        if(!(key ** 2 in frequencyCounter2)){return false}
        }
        // if frequencyCounter2[frequencyCounter1[key] squared] is not equal to f
requencyCounter1 key => false
        if(frequencyCounter2[key ** 2] !== frequencyCounter1[key]){return false}
    }
    return true
}

same([1,2,3,2,5], [9,1,4,4,11])
```

f. Time complexity O(n)

g. 2 loops is better than nested loops

h. Here we are drawing objects and key/value pairs and then looping thru both objects

i. Another example

    i.

```
//_____________________________________________________________________
// Compare two string and see if they are anagrams, meaning they have the same ch
aracters, and same amount of the corresponding character, and the word is not nec
cessarily in order
```

```
// Input is 2 strings, single words, no spaces, only lowercase alphabets
// output is true | false

// "hello" and "hi" = false
// "cinema" and "iceman" = true
// "caat" and "cat" = false

function anagrams(string1, string2) {
  // short circuit lenght comparision test
  if (string1.length !== string2.length) return false;

  // see if they are anagrams
  // same characters (no order)
  // same amount of corresponding characters
  // initiate 2 objects that frequency count both strings
  const freqCounter1 = {};
  const freqCounter2 = {};
  // loop thru string1, if key exists in freqCounter1 => count +1, if it doesn't
 exits, intiate key in freqCounter1 at 1
  for (const char of string1) {
    freqCounter1[char] = (freqCounter1[char] || 0) + 1;
  }
  // loop thru string2, if key exists in freqCounter2 => count +1, if it doesn't
 exits, intiate key in freqCounter2 at 1
  for (const char of string2) {
    freqCounter2[char] = (freqCounter2[char] || 0) + 1;
  }

  // loop thru freqCounter1
  for (let key in freqCounter1) {
    // test if frequency1 has same characters as frequency2 => 1{i:1} 2{i:2} (i v
s i)
    if (!(key in freqCounter2)) {
      return false;
    }
    // test if frequnecy1 has the same number of chracters as frequency2 => 1{i:
1} 2{i:2} (1 vs 2)
    if (freqCounter1[key] !== freqCounter2[key]) return false;
  }

  // // if passed all test, return true
  return true;
}

// console.log(anagrams("thankyou", "youthank"));

//____________________________________________________________________________
// Colt Steele solution

function validAnagram(first, second) {
  // if not same lenght, return false
  if (first.length !== second.length) return false;

  // creat an object where we store first word frequency
```

```
  const lookup = {};

  // loop thru first
  for (let i = 0; i < first.length; i++) {
    let letter = first[i];
    // if lookup object had letter, letter +1
    // if object doesn't have letter, letter=1
    lookup[letter] ? (lookup[letter] += 1) : (lookup[letter] = 1);
  }
  console.log(lookup);
  // {a: 0, n: 0, g: 0, r: 0, m: 0}

  // loop thru second
  for (let i = 0; i < second.length; i++) {
    let letter = second[i];
    // ! = if its not the case
    // if it is not the case that lookup has letter key, or value is 0, return fa
lse
    // sidenote: if key=0 -> false. E.g. obj={a:0}....obj[a]->false
    if (!lookup[letter]) {
      return false;
      // else (if letter is in lookup, letter count -1)
    } else {
      lookup[letter] -= 1;
    }
  }
  // once passed all false test, return true
  return true;
}

validAnagram("anagrams", "nagaram");
```

Pattern #2: Multiple Pointers

1. Targeting an index or element and moving position towards beginning, end, middle
2. Example: Given a sorted array of numbers, find the first pair that sums to 0

a.

```
function sumZero(arr) {
  // beginning
  let left = 0;
  // end
  let right = arr.length - 1;
  // while left index is smaller than right index (can't crossover)
  while (left < right) {
    // sum of beginning and end
    let sum = arr[left] + arr[right];
    // if sum is 0, return numbers
    if (sum === 0) {
```

```
      return [arr[left], arr[right]];
      // else if sum is greater than 0...
    } else if (sum > 0) {
      // reduce right by 1, so as it closes range towards the beginning
      right--;
      // if sum is 0 or below 0, no need to iterate further, so increase left by
 one, closing range towards end
    } else {
      left++;
    }
  }
}

sumZero([-4,-3,-2,-1,0,1,2,3,10]); // [-3,3]
```

3. Example 2
    a. given a sorted array, return the count (single number) of unique values in the array (there can be negative numbers in the array, but always sorted)
    b. My first solution (not following Colt's pattern)

```
function countUniqueValues(array) {
  // if array length is 0; return 0
  if (array.length == 0) return 0
  // if array length is 1; return 1
  if (array.length == 1) return 1

  // uniqueArray = []
  const uniqueArray = [];
  // push array[0]
  uniqueArray.push(array[0]);

  // iterate thru array
  for (const num of array) {
    // if array[i] is != to uniqueArray[uniqueArray.length - 1], push array[i]
    if (num != uniqueArray[uniqueArray.length - 1]) {
      uniqueArray.push(num)
    }
  }

  return uniqueArray.length
}

console.log(countUniqueValues([1,2,3,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,7,
7,12,12,13]));
```

Pattern #3: Sliding Window

1. When we look for a subset that is continues in some way
    a. E.g. longest sequence of unique characters
    b. Or largest sum of X consecutive numbers
2. We create a sliding window that check for conditions
3. Example
    a. Given an array and n, write a function that calculates the max sum of n consecutive elements
    b. Think Big O, argument could be hundreds of number, with a n of 25
    c. O(n)

```
 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;
}

maxSubarraySum([2,6,9,2,1,8,5,6,3],3)
```

    d. Refactored - Still O(n), but we only loop for the array one time

```
function maxSubarraySum(arr, num) {
  let maxSum = 0;
  let tempSum = 0;
  // edge case
  if (arr.length < num) return null;
  // loop thru array
  for (let i = 0; i < num; i++) {
    // store sum of n elements
    maxSum += arr[i];
  }
  tempSum = maxSum;
  // loop beginning at after n elements
```

```
  for (let i = num; i < arr.length; i++) {
    // sum = subtract last number, add new number
    tempSum = tempSum - arr[i - num] + arr[i];
    // if new sum is higher, update maxSum
    maxSum = Math.max(maxSum, tempSum);
    // i=i+1, loop again
  }
  return maxSum;
}

maxSubarraySum([2, 6, 9, 2, 1, 8, 5, 6, 3], 3);
```

e. Argument

[1 2 3 5 2 4 5 7 3 4 6 6 4 2 5 6 8]

Naive code: grab n numbers, sum them, shift one, repeat

Refactor: grab n numbers, sum them, shift one, subtract shifted number, add new number, repeat *SLIDING WINDOW*

Pattern #4: Divide and Conquer

1. We grab a large list of data
2. We divide it into smaller chunks and run operations there
3. Example
    a. Given a ***sorted*** array of integers, return the index of the argument, if no index, return -1
    b. Naive version: Loop thru every element and if found, return index
    c. Example, array [1,2,3,5,6,8,12,15,16,29,30] and n=29
    d. We grab the median 12, and see if n is lower or higher, if higher, we can ignore half of the array = [~~1,2,3,5,6,8,12~~,15,16,29,30]
    e. Repeat same concept, is 16 lower or higher = lower = [~~1,2,3,5,6,8,12,15,16~~,29,30]
    f. Time complexity Log(N)