Algorithms and Problem Solving Patterns

OBJECTIVES

- Define what an algorithm is
- Devise a plan to solve algorithms
- Compare and contrast problem solving patterns including frequency counters, two pointer problems and divide and conquer

WHAT IS AN ALGORITHM?

A **process** or **set of steps** to accomplish a certain task.

Why do I need to know this?

Almost **everything** that you do in programming involves some kind of **algorithm!**

It's the <u>foundation</u> for being a successful problem solving and developer

Also...

INTERVIEWS

INTERVIEWS

HOW DO YOU IMPROVE?

1. **Devise** a plan for solving problems
2. **Master** common problem solving patterns

First Up…

PROBLEM SOLVING

Strategies

A SIMPLE, FOOLPROOF, MAGICAL, MIRACULOUS, FAIL-SAFE APPROACH

* NOT REALLY

* BUT IT'S STILL HELPFUL

PROBLEM SOLVING

- Understand the Problem
- Explore Concrete Examples
- Break It Down
- Solve/Simplify
- Look Back and Refactor

Note: many of these strategies are adapted from George Polya, whose book *How To Solve It* is a great resource for anyone who wants to become a better problem solver

UNDERSTAND THE PROBLEM

1. Can I restate the problem in my own words?
2. What are the inputs that go into the problem?
3. What are the outputs that should come from the solution to the problem?
4. Can the outputs be determined from the inputs? In other words, do I have enough information to solve the problem? (You may not be able to answer this question until you set about solving the problem. That's okay; it's still worth considering the question at this early stage.)
5. How should I label the important pieces of data that are a part of the problem?

Write a function which takes two numbers and returns their sum.

ok...

EXPLORE EXAMPLES

Coming up with examples can help you understand the problem better

Examples also provide sanity checks that your eventual solution works how it should

User Stories!

Unit Tests!

EXPLORE EXAMPLES

- Start with Simple Examples
- Progress to More Complex Examples
- Explore Examples with Empty Inputs
- Explore Examples with Invalid Inputs

Write a function which takes in a string and returns counts of each character in the string.

BREAK IT DOWN

Explicitly write out the steps you need to take.

This forces you to think about the code you'll write before you write it, and helps you catch any lingering conceptual issues or misunderstandings before you dive in and have to worry about details (e.g. language syntax) as well.

Write a function which takes in a string and returns counts of each character in the string.

SOLVE THE PROBLEM

If you can't...

SOLVE A SIMPLER PROBLEM!

SIMPLIFY

- Find the core difficulty in what you're trying to do
- Temporarily ignore that difficulty
- Write a simplified solution
- Then incorporate that difficulty back in

Write a function which takes in a string and returns counts of each character in the string.

LOOK BACK & REFACTOR

Congrats on solving it, but you're not done!

REFACTORING QUESTIONS

- Can you check the result?
- Can you derive the result differently?
- Can you understand it at a glance?
- Can you use the result or method for some other problem?
- Can you improve the performance of your solution?
- Can you think of other ways to refactor?
- How have other people solved this problem?

Write a function which takes in a string and returns counts of each character in the string.

RECAP!

- Understand the Problem
- Explore Concrete Examples
- Break It Down
- Solve/Simplify
- Look Back and Refactor

HOW DO YOU IMPROVE?

1. **Devise** a plan for solving problems ✓
2. **Master** common problem solving patterns

Next Up!!!

Next Up...

PROBLEM SOLVING

Patterns

SOME PATTERNS...

- Frequency Counter
- Multiple Pointers
- Sliding Window
- Divide and Conquer
- Dynamic Programming
- Greedy Algorithms
- Backtracking
- **Many more!**

FREQUENCY COUNTERS

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

AN EXAMPLE

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.

```
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)
```

A NAIVE SOLUTION

```
function same(arr1, arr2){
    if(arr1.length !== arr2.length){
        return false;
    }
    for(let i = 0; i < arr1.length; i++){
        let correctIndex = arr2.indexOf(arr1[i] ** 2)
        if(correctIndex === -1) {
            return false;
        }
        arr2.splice(correctIndex,1)
    }
    return true
}
```

Time Complexity - N^2

REFACTORED

```
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
}
```

Time Complexity - O(n)

ANAGRAMS

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*.

```
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
```

YOUR
TURN

MULTIPLE POINTERS

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

AN EXAMPLE

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

```
sumZero([-3,-2,-1,0,1,2,3]) // [-3,3]
sumZero([-2,0,1,3]) // undefined
sumZero([1,2,3]) // undefined
```

NAIVE SOLUTION

```
function sumZero(arr){
    for(let i = 0; i < arr.length; i++){
        for(let j = i+1; j < arr.length; j++){
            if(arr[i] + arr[j] === 0){
                return [arr[i], arr[j]];
            }
        }
    }
}
```

Time Complexity - O(N^2)

Space Complexity - O(1)

REFACTOR

```
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++;
        }
    }
}
```

Time Complexity - O(N)

Space Complexity - O(1)

PATTERN #1

MULTIPLE POINTERS

```
function     Zero(arr){
  for(le   i =     i < arr.   gth;   ++){
    for(   t j = i+1; j < arr.len   ; j++){
          f(arr[i] + arr[j] === 0)
            return [arr[i], arr[j  ;

    }
  }
}
```

Time Complexity - O(N^2)

Space Complexity - O(1)

```
function     Zero(arr){
  let le    = 0
  let r    t =    .length    ;
  while    eft < right){
    let    um = arr[left] + arr[rig    ;
    if(    n ==
          turn              r[r    t]];
    } el    if(s
          ri    --;
    } else
```

Time Complexity - O(N)

Space Complexity - O(1)

countUniqueValues

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.

```
countUniqueValues([1,1,1,1,1,2]) // 2
countUniqueValues([1,2,3,4,4,4,7,7,12,12,13]) // 7
countUniqueValues([]) // 0
countUniqueValues([-2,-1,-1,0,1]) // 4
```

YOUR TURN

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.

An 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.

```
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
```

A naive solution

```
function maxSubarraySum(arr, num) {
  if ( num > arr.length){
    return null;
  }
  var max = -Infinity;
  for (let i = 0; i < arr.length - num + 1; i ++){
    temp = 0;
    for (let j = 0; j < num; j++){
      temp += arr[i + j];
    }
    if (temp > max) {
      max = temp;
    }
  }
  return max;
}
```

Time Complexity - O(N^2)

Refactor

```
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;
}
```

Time Complexity - O(N)

YOUR

TURN

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**

An 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

```
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
```

A naive solution

```
function search(arr, val){
    for(let i = 0; i < arr.length; i++){
        if(arr[i] === val){
            return i;
        }
    }
    return -1;
}
```

Linear Search

Time Complexity O(N)

Refactor

```
function search(array, val) {

    let min = 0;
    let max = array.length - 1;

    while (min <= max) {
        let middle = Math.floor((min + max) / 2);
        let currentElement = array[middle];

        if (array[middle] < val) {
            min = middle + 1;
        }
        else if (array[middle] > val) {
            max = middle - 1;
        }
        else {
            return middle;
        }
    }

    return -1;
}
```

Time Complexity - Log(N) - Binary Search!

YOUR

TURN

Recap

- Developing a problem solving approach is incredibly important
- Thinking about code before writing code will always make you solve problems faster
- Be mindful about problem solving patterns
- Using frequency counters, multiple pointers, sliding window and divide and conquer will help you reduce time and space complexity and help solve more challenging problems