Goals
Goals
Developing a problem solving process
The process
1. Understand the Problem
2. Explore Concrete Examples
3. Break It Down
4. Solve A Simpler Problem
5. Use Tools Strategically
6. Look back and refactor
Now that you have the plan....
Common problem solving patterns
Common problem solving patterns
Frequency counters
An example
A naive solution
Using a frequency counter - first a helper function
Using a frequency counter - solution
Your turn!
Multiple pointers
Multiple pointers
An example
A naive solution
Using multiple pointers
Your turn!
Recap

Problem Solving Process and Patterns

Download Demo Code

Goals

- Develop problem solving process & learn fundamental patterns
- Use frequency counters to solve problems more efficiently
- Use multiple pointers to solve problems more efficiently
- Compare different runtimes

Developing a problem solving process

The process

1. Understand the Problem
2. Explore Concrete Examples
3. Break It Down
4. Solve a Simpler Problem
5. Use Tools Strategically
6. Look Back and Refactor

1. Understand the Problem

- Can I restate the problem in my own words?
- What are the inputs that go into the problem?
- What are the outputs that should come from the solution to the problem?
- Do I have enough information?
- How should I label the important pieces of data that are a part of the problem?

2. Explore Concrete Examples

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

3. Break It Down

- Explicitly write out the steps you need to take.
- You can type this as pseudocode or write it on a whiteboard (or desk)
- This forces you to think about the code you'll write before you write it
- This helps you catch any lingering conceptual issues or misunderstandings
- Don't write code!

4. Solve A Simpler Problem

> *If there is a problem you can't solve, then there is an easier problem you can solve: find it.*
>
> *--George Pólya*

- 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

Note: Easier said than done.

This fourth strategy (solve a simpler problem) is easier said than done. If you simplify too much, you may make the problem too simple, in which case solving the simpler problem provides little insight into the original. But if you don't simplify enough, you still might be stuck on a problem that is too challenging. Finding the right sub-problem to isolate takes a decent amount of practice.

5. Use Tools Strategically

- Use your debugging tools.
- Don't guess and check!
- Scientific approach: formulate hypotheses, test, draw conclusions. Repeat.

6. Look back and refactor

- Does the result match your expected output?
- Can you improve the performance of your solution?
- What other ideas could you have pursued?

Now that you have the plan....

- The only way to get better is to practice using this plan!

Common problem solving patterns

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

Frequency counters

- This pattern uses objects, maps, 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 **squares**, 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.

```
squares([1,2,3], [4,1,9]); // true
squares([1,1,3], [1,9]); // false
squares([1,2,1], [4,4,1]); // false (must be same frequency)
```

A naive solution

```
function squares(nums1, nums2) {
  if (nums1.length !== nums2.length) {
    return false;
  }

  for (let i = 0; i < nums1.length; i++) {
    let correctIndex = nums2.indexOf(nums1[i] ** 2);

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

    nums2.splice(correctIndex, 1);
  }

  return true;
}
```

Time Complexity - O(n^2)

Using a frequency counter - first a helper function

```
// a function to create a simple
// frequency counter using an object
function createFrequencyCounter(array) {
  let frequencies = {};

  for (let val of array) {
    let valCount = frequencies[val] || 0;
    frequencies[val] = valCount + 1;
  }

  return frequencies;
}
```

```
// a function to create a simple
// frequency counter using a map
function createFrequencyCounter(array) {
  let frequencies = new Map();

  for (let val of array) {
    let valCount = frequencies.get(val) || 0;
    frequencies.set(val, valCount + 1);
  }

  return frequencies;
}
```

Note: Maps vs. objects

Maps and objects are similar in JavaScript, as both can be used to store collections of key-value pairs. While objects have been around since the beginning of JavaScript, Maps came to the language as part of ES2015. You can read more about the difference between these to data structures at MDN.

Using a frequency counter - solution

```
function squaresWithFreqCounter(nums1, nums2) {
  if (nums1.length !== nums2.length) return false;

  let nums1Freqs = createFrequencyCounter(nums1);
  let nums2Freqs = createFrequencyCounter(nums2);

  for (let key of nums1Freqs.keys()) {
    if (nums2Freqs.has(key ** 2) === false) {
      return false;
    }

    if (nums2Freqs.get(key ** 2) !== nums1Freqs.get(key)) {
      return false;
    }
  }

  return true;
}
```

Time Complexity - O(n

Your turn!

Given two strings, write a function called **validAnagram**, which determines 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
validAnagram("awesome", "awesom"); // false
validAnagram("qwerty", "qeywrt"); // true
validAnagram("texttwisttime", "timetwisttext"); // true
```

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

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
```

A naive solution

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

- Time Complexity - O(n^2)

Using multiple pointers

```
function sumZeroMultiplePointers(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)

Your turn!

Implement a function, ***countUniqueValues***, which accepts a sorted array, and counts unique values in 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
```

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
- Frequency counters and multiple pointers are just the start
- Do not overfit!