Algorithm & Data-structure

Problem Solving skills

A good approach to problem solving are listed below, most of the solution are adopted from ***George Plya*** book titled ***How to solve it***

1. Understanding the problem
2. Explore concrete examples
3. Break it down
4. Solve/Simplify
5. Look back and refactor

1. Understanding the problem

- Can I restate the problem in my own words?
- What are the input that goes into the problem?
- What are the outputs that should come out of the problem?
- Can the outputs be determined from the inputs? In other words, do i have enough information to solve the problem?
- How should i label the important pieces of data that are a part of the problem.

2. Explore concrete examples

- start with simple example
- Progress to more complex examples
- Explore example with empty input
- Explore examples with invalid input

3. Break it down

- Determine inputs
- Computaions
- Edge cases
- Output

4. Solve and simplify

- Find the core dificulty in what you are trying todo
- Temporary ignore that difficulty
- Write a simplify solution
- Then incorporate that difficulty back in

5. Look back and refactor

- 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 another problem?
- Can you improve the performance of your solution?
- Can you think of other way to refactor?
- How have other people solved this problem?

How to tackle problem

- **Device** a plan for solving the problem
- **Master** common problem solving pattern

Some patterns

1. Frequency Counter
2. Multiple pointers
3. Sliding window
4. Divide and Conquer
5. Dynamic programming
6. Greedy Algoriyhm
7. Backtracking etc.

1. Frequency Counter

a). Anagram: *An anagram is two word with the same letters of alphabet eg. rare and rear, dear and read*

```
// this is anagram. O(N)
let isAnagram = function (word1, word2) {
  let first = {};
  let second = {};

  for (const char of word1) {
    first[char] = ++first[char] || 1;
  }
  for (const char of word2) {
    second[char] = ++second[char] || 1;
  }
  for (const key in first) {
    if (first[key] !== second[key]) {
      return false;
    }
  }
  return true;
};
```

b). countUniqueValue: *return the count of unique values in an array*

```
let countUniqueValues = (nums) => {
  if (nums.length < 1) {
    return undefined;
  }
  let numV = {};
  let count = 0;
  for (let i = 0; i < nums.length; i++) {
    numV[nums[i]] = ++numV[nums[i]] || 1;
  }
  for (let key in numV) {
    count++;
  }
  // return Object.keys(numV).length;
  return count;
};
```

2. Multiple pointers

Note: multple pointer can only be used in a sorted list or array

a). SumZero: *Sumzero is used to determine if there are two numbers in a sorted array who their sum is equal to zero(0)*

```
// using multiple pointer

let sumZero = () => {
  let l = 0;
  let r = arr.length - 1;
  while (l < r) {
    let sum = arr[l] + arr[r];
    if (sum < 0) {
      l++;
    } else if (sum > 0) {
      r--;
    } else {
      return [arr[l], arr[r]];
    }
  }
};
```

b). countUniqueValue: *return the count of unique values in an array*

```
// using two pointer
let methodTwo = (arr) => {
  if (arr.length < 1) {
    return undefined;
  }
```

```
  let left = 0;
  for (let j = 1; j < arr.length; j++) {
    if (arr[left] !== arr[j]) {
      left++;
      arr[left] = arr[j];
    }
    j++;
  }
  return left + 1;
};
```

3. Sliding window

4. Divide and Conquer

5. Dynamic programming

6. Greedy Algoriyhm

7. Backtracking etc.

Recurssion

A recurssion is a function that called itself. It make use of **STACK**

Stack: A stack is a data-structure that follows LIFO (*Last In, First Out*).

Examples of Recurssion

i. count down

```
let countDownTwo = (num) => {
  if (num <= 0) {
    console.log("All done");
    return;
  }
  console.log(num);
  num--;
  countDownTwo(num);
};
```

ii. Sum of natural number

```
let soNNThree = (num) => {
  if (num === 0) return 0;
  return num + soNNThree(--num);
};
```

iii. Factorial

```
let factorialTwo = (num) => {
  if (num < 0) return 0;
  if (num < 1) return 1;
  return num * factorial(--num);
};
```

iv. Power

```
let powerTwo = (b, p) => {
  if (p === 0) return 1;
  return b * powerTwo(b, p - 1);
};
```

V. Product of Array

```
let poa = (arr) => {
  if (arr.length === 0) return 1;
  return arr[0] * poa(arr.slice(1));
};
```

Search Algorithm

Sorting Algorithm

There are many types of sorting such as bubble sort, selection sort insertion sort merge sort, quick sort, and radix sort.

Bubble Sort

- **Step 1:** Start with an unsorted array of elements.
- **Step 2:** Set up a flag called `swapped` and initialize it to `true`. This flag helps track if any swaps were made during a pass through the array.
- **Step 3:** Begin a loop that continues as long as `swapped` is `true`. This loop ensures you keep iterating through the array until no more swaps are needed.

- **Step 4:** Inside the loop, set `swapped` to `false` to reset it for this pass.
- **Step 5:** Iterate through the array from the beginning to the second-to-last element.
- **Step 6:** For each pair of adjacent elements, compare them. If the element on the left is greater than the element on the right, they are out of order.
- **Step 7:** If the elements are out of order, swap them. Temporarily store the left element, copy the right element to the left, and put the temporarily stored element in the right position.
- **Step 8:** After swapping, set `swapped` to `true` to indicate a swap was made in this pass.
- **Step 9:** Continue this process, comparing and swapping adjacent elements until you reach the end of the array.
- **Step 10:** Once you complete a full pass without making any swaps (i.e., `swapped` remains false), the array is fully sorted, and you can exit the loop.
- **Step 11:** The array is now sorted in ascending order, and you've successfully used the Bubble Sort algorithm.