Chapter 3

**Problem solving**

**Approaches & Patterns**

**3.1 Objectives**

To simplify the challenges in some code a problem you have two ways:

1. Problem solving approach: The basic approach to solving a problem you don't know how to solve. Steps you can take things you can do to make it easier.
2. Problem solving Patterns: This is about more code specific blueprints that help solve a lot of problems.

* 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*

**3.2 Algorithm**

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

* It could be a set of mathematical steps that we need to do to solve a problem.
* It can be a recommender algorithm, Facebook's algorithm that you know suggests certain ads to you or
* It can be search algorithm, Google's algorithm when you search for something.

But at its core it's just a *set of steps* to accomplish a task.

* Why do I need to know this?
* Almost everything that you do in programming involves some kind of algorithm!
* It's the *foundation* for being a *successful* problem *solving* and *developer*.

HOW DO YOU IMPROVE?

* Devise a plan for solving problems It's more about how do you approach the *problem strategies* for *breaking* it down.
* Master common problem solving patterns. In interviews a lot of *these* *problems* that come up can be *broken* into sort of *different categories*. And if you can identify some of those categories there are *some steps* you *can take* in your *code*.
* Some *recipes* you can *keep* that will help you solve the algorithm or the challenge.

**3.3 PROBLEM SOLVING**

1. *Understand* the *Problem*.
2. *Explore* Concrete *Examples*.
3. *Break* It *Down*.
4. *Solve/Simplify*.
5. *Look* Back and *Refactor*

**3.4 Step 1: Understand the Problem**

Learning a language is easy. You can learn the syntax, you can copy others code and follow step by step guide.

But solving a problem is much more difficult.

* It might take a little bit of time but you learn the syntax and you're done. You can copy other people's code or remake an application following step by step instructions.
* But the hardest thing has to do with solving new problems and new challenges.
* It's really important to have a *strong problem solving skill set* and a lot of it will come *naturally* with time but it's worth being deliberate.
* It's worth *having a game plan* to follow *step by step guide*.
* It's not going to solve your problems for you but it helps you sit and *think* about and *break problems down*.
* And it helps *solutions* come about *more naturally*.

|  |
| --- |
| * These strategies are adapted from a really good book on problem solving in general. The book is called How to solve it by George polya. |

* In an interview, you can just type a bunch of code pretty quickly and feel like you're making progress in a short amount of time.
* But *before* you start *typing* or *whiteboarding* or whatever you're doing I think it's really important to take a *step back* and just *make sure you understand the task* ahead of you.
* There are some very *deliberate questions* you can ask. A thorough *investigation* of the *problem* or a question and it can really help clarify things and sometimes will give you a *clear understanding*. So here are some of the questions that I recommend you *ask yourself* or if you're in an *interview setting* *ask the interviewer* but also if you're working on your *own challenge* you're trying to build something out and you're not sure how to approach it.

Imagine we are working with a problem: "Adding Two Numbers". Ask yourself these questions:

UNDERSTAND THE PROBLEM

1. Can I restate the problem in my own words?
   * Just make sure you can restate it back. Not word for word, but *change it a bit in your own words* to make sure that you actually understand what the question is.
2. What are the inputs that go into the problem?

We could just say something like write a function that adds two numbers or implement addition.

* Are we talking about *integer* numbers or are we talking about *floating* point. Depends on the language that we're working in of course.
* *How large* are these numbers going to be. In most languages there's an *upper bound* for the size for numbers and if you're trying to add *really large numbers* it *doesn't work well*.

In other languages you might be able to represent this better as a string for really large number.

* *Another thing is:* Are we only working with two inputs. Is it always two?

What happens if someone leaves and is put off.

Or what if someone wants to add four or 10.

1. What are the outputs that should come from the solution to the problem?
   * What should be returned from my function.

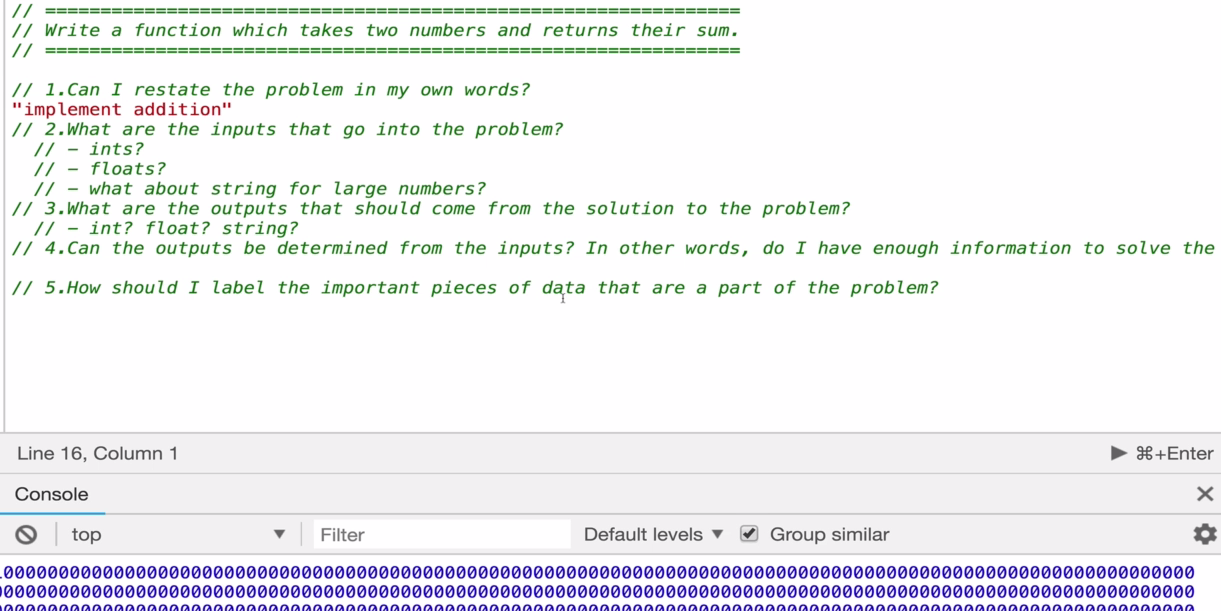
* What should the output look like.
* Should it be an *integer*.
* Should it be a *float*, depending on inputs.
* Are you even *allowed* to pass in a *float*.
* What about the *string* returning a string for a *massive addition*.

1. 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.)
   * Meaning do you have *enough information to solve the problem* you have the *inputs* can you get the *outputs* that are *expected*.

* Can the outputs be *determined* from the *inputs*.
* Basically do we have *enough information* to do the *problem*.
* if someone only *passes* one number we don't have *enough information* to do addition at that *point*.
* So do we add *zero*?
* Do we return *undefined* or *no* or something.

1. How should I label the important pieces of data that are a part of the problem?
   * This would be you know *what are the things* that *really matter* in this problem what should you call them. What's the terminology that you should use.

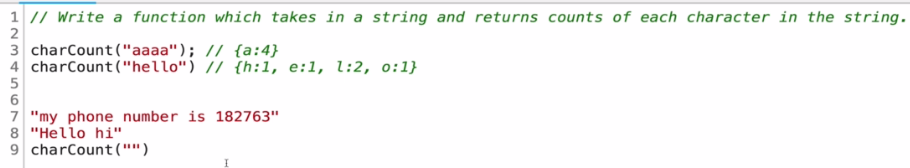
* And lastly how should I label the important pieces of data that are part of the problem.
* To name our function ***add*** and then we'll have number one and number two is the arguments and then some is our result that we return.



**3.5 Step 2: Work with concrete Examples**

EXPLORE EXAMPLES

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



* Consider the above function, counts characters in a string. Then we should consider the following:

1. Count letters in a word
2. Count letter in a sentence:

* Are 1,2 ,3 numerical values considered
* Are " " empty space ignored

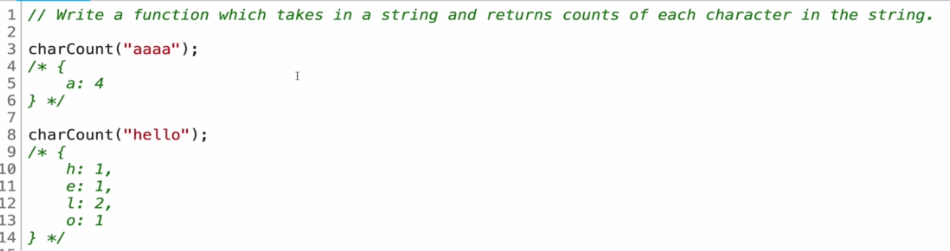
1. Try *invalid* *input*:

* Use a emty string " "
* Use a non-string value (eg: int or float)

Working with example will help us to understand the problem.

**3.6 Step 3: Break-Down the problem before coding**

* 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.
* Use comments as a guideline for the steps.
* Doesn’t have to be actual pseudo-code or actual syntax. Just use comments
* Example 3.1 : Write a function which takes in a string and returns counts of each character in the string.



|  |  |
| --- | --- |
| * At first just write the basic step   // do something  // return an object with keys that are lowercase alphanumeric characters in the string; values should be the counts for those characters  i.e only alphanumeric characters: a, b, c, .. z, 1, 2, .. 0.  We consider all as lower case. i.e. uppercase considered as lowercase. |  |

* Idea is: Loop over every characters and do something.

Notice How the steps evolves.

Step 0 : Just write some thing

**function** **charCount**(str) {

    // *do something*

    // *return an object with keys that are lowercase alphanumeric characters in the string; values should be the counts for those characters*

}

Step 1: Generate basic idea

**function** **charCount**(str) {

    // *make object to return at end*

    // *loop over string, for each character...*

    // *return object at end*

}

Step 2: Think about the problem and add more comments

**function** **charCount**(str) {

    // *make object to return at end*

    // *loop over string, for each character...*

        //*if the char is a key in object, add one to count*

        //*if the char is not in object, add it to object and set value to 1*

    // *return object at end}*

Step 3: Think about the inputs and modify comments

**function** **charCount**(str) {

    // *make object to return at end*

    // *loop over string, for each character...*

        //*if the char is a number/letter AND is a key in object, add one to count*

        //*if the char is a number/letter AND not in object, add it to object and set value to 1*

    // *return object at end*

}

Step 4: Think about the inputs that need to avoid and modify comments

**function** **charCount**(str) {

    // *make object to return at end*

    // *loop over string, for each character...*

        //*if the char is a number/letter AND is a key in object, add one to count*

        //*if the char is a number/letter AND not in object, add it to object and set value to 1*

        //*if character is something else (space, period, etc.) don't do anything*

}

|  |
| --- |
| * In the interview, those *guidelines* are more *important* than the *actual code*. Interviewer will know about how you *tackle* the *problem* (even you didn’t write the actual code). |

**3.7 Step 4: Solve/Simplify the problem**

* Simplify: If the problem is too hard to solve and you are stuck at a step. Then you should simplify the problem.
* i.e. solve the problem with simpler cases.

SIMPLIFY

1. Find the core difficulty in what you're trying to do
2. Temporarily ignore that difficulty
3. Write a simplified solution
4. Then incorporate that difficulty back in

* Example: For example we take the previous problem, "count chars in a string".
* Loop: If you are stuck at how the loop works:
* Hard-code the 5 or 6 characters by yourself, then you can understand the pattern. After that you may implement the loop.
* Uppercase /Lowercase: If you don’t know the methods/function to lowercase-uppercase in JS then just skip this part and make the core. After that we can came back to this lowercase-uppercase in JS.

**function** **charCount**(str){

    // *make object to return at end*

**var** result={};

    // *loop over string, for each character...*

**for**(**var** i=0; i **<** str**.**length; i++){

**var** char=str[i]**.toLowerCase**()

        //*if the char is a number/letter AND is a key in object, add one to count*

**if**(result[char] **>** 0) {

            result[char]++;

        }

        //*if the char is a number/letter AND not in object, add it to object and set value*

**else** {

            result[char] = 1;

        };

    }

**return** result;

}

* How to filter space/non-alphanumeric: There are many other solutions to that. After creating the above core, we can just add in there.

1. Define an array containing all valid- alphanumeric characters. (Very simple but tedious solution).
2. Use a regular-expression (smart solution)
3. ASCII character code (use the defined range for the alphanumeric-character ASCII code, we did it already in C/C++)

**3.8 Step 5: Look-Back & Refactor the problem**

* Improve the code/code-structure
* Specify things by commenting (readability).
* Efficiency
* Legibility

* REFACTORING QUESTIONS:

1. Can you *check* the result?
2. Can you *derive* the result *differently*? (*Web/Online:* Google the problem to find better ways to solve).
3. Can you *understand* it at a *glance*? (use better commenting)
4. Can you use the *result* or *method* for some *other problem*? (Can unlock the solution to another problem).
5. Can you improve the *performance* of your solution? (Time-Space Complexity).
6. Can you think of *other ways* to *refactor*?
7. How have *other people* *solved* this problem? (Can learn more. Looking other language may help a lot. Find ideas & different appraches).

* Example 3.2 : Refactoring our Char Counter.
* Refactor 1: Using *regular expression* (RegExp)

function charCount(str){

    var obj = {};

    for (var i = 0; i < str.length; i++){

        var char = str[i].toLowerCase();

        if (/[a-z0-9]/.test(char)) {    //Using RegExp

            if (obj[char] > 0) {

                obj[char]++;

            } else {

                obj[char] = 1;

                };

        }

    }

    return obj;

}

* Refactor 2: Using ***for/of*** loop

function charCount(str){

    var obj = {};

    for (var char of str){

        var char = char.toLowerCase();

        if (/[a-z0-9]/.test(char)) {    //Using RegExp

            if (obj[char] > 0) {

                obj[char]++;

            } else {

                obj[char] = 1;

                };

        }

    }

    return obj;

}

* Refactor 3: Use *conditional statement* instead of if-else

function charCount(str){

    var obj = {};

    for (var char of str){

        var char = char.toLowerCase();

        if (/[a-z0-9]/.test(char)) {    //Using RegExp

            obj[char] = (++obj[char]) || 1

        }

    }

    return obj;

}

* Refactor 4: Using *Character code* is more faster than RegExp.

function charCount(str){

    var obj = {};

    for (var char of str){

        if (isAlphaNumeric(char)) {    //Using AlphaNuemeric character codes

            char = char.toLowerCase();

            obj[char] = (++obj[char]) || 1;

        }

    }

    return obj;

}

function isAlphaNumeric(char){

    var code = char.charCodeAt(0);

    if (!(code > 47 && code < 58) && // numeric (0-9)

        !(code > 64 && code < 91) && // upper alpha (A-Z)

        !(code > 96 && code < 123)) { // lower alpha (a-z) return false

        return false;

    }

    return true;

}

* The ***for/in*** loop and the ***for/of*** loop.
* For In Loop: The JavaScript for in statement loops through the properties of an Object:

for (key in object) {

  // code block to be executed

}

​

Example of for-in loop

const person = {fname:"John", lname:"Doe", age:25};

​

let text = "";

for (let x in person) {

  text += person[x];

}

* The for in loop *iterates* over a *person* object
* Each iteration *returns* a ***key (x)***
* The key is *used* to *access* the value of the key
* The value of the key is ***person[x]***
* For Of Loop: The JavaScript *for of* statement loops through the *values* of an *iterable* *object*.

*iterable* - An object that has *iterable* *properties*.

It lets you loop over ***iterable data structures*** such as ***Arrays***, ***Strings***, ***Maps***, ***NodeLists***, and more:

Syntax

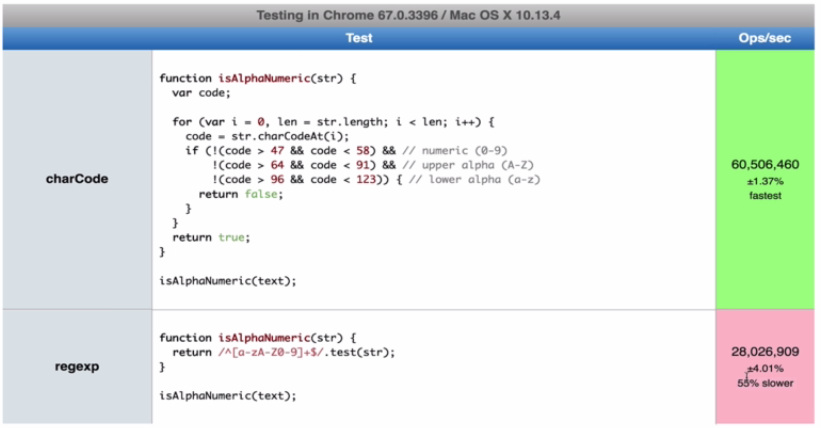
**for** (variable of iterable) {

// code block to be executed

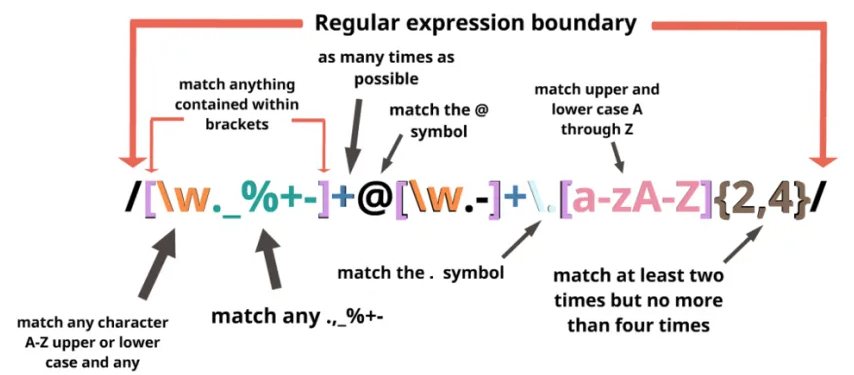
}

***variable*** - For every iteration the value of the next *property* is *assigned* to the *variable*. Variable can be declared with ***const***, ***let***, or ***var***.

***Character code* is more faster than RegExp**



* Regular Expression: A regular expression is a *sequence of characters* that specifies a *search pattern in text*. Usually such patterns are used by *string-searching algorithms* for "find" or "find and replace" operations on strings, or for input validation.



**Problem Solving Patterns**

**3.9 Problem Solving Patterns**

Here we'll discuss *different* *Problem-solving patterns*. Some of those may help you *1* out of *10* *programming problems*. These patterns are also think as programming mechanism or blue-print.

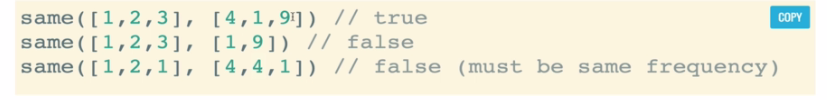
* SOME PATTERNS: Some of those are not official names but the methodologies are well known.

1. Frequency Counter
2. Multiple Pointers
3. Sliding Window
4. Divide and Conquer
5. Dynamic Programming
6. Greedy Algorithms
7. Backtracking
8. Many more!

* Here we'll:
* Explain some of the above patterns
* Give some examples:
* Poorly solved (Naïve version)
* Solved using a pattern.

**3.10 FREQUENCY COUNTERS**

* This pattern uses objects or sets to collect ***values/frequencies of values*** (i.e. how many times an element occurs in objects).
* This can often ***avoid*** the need for ***nested loops*** oroperations with arrays / strings.
* Officially it's not really called frequency counters. In algorithms and challenges when you have multiple pieces of data/ multiple inputs you need to compare them to see if they consist of similar values.
* If they are Anagrams of one another (same letters but different word ), if a value is contained inside of another value, anytime you're *comparing pieces of data* to *inputs* or more than two and frequencies of certain things occur.
* Example 3.3 : Write a function called ***same()*** which accepts *two arrays* and the function should return *true* if every value in the array has its *corresponding* value *squared* in the *second array*.



* Not The Best: Following is not the best possible solution in this case.

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;

        }

        console.log(arr2); */\*To view how the 2nd array shrinks\*/*

        arr2.splice(correctIndex,1)

    }

    return true;

}

​

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

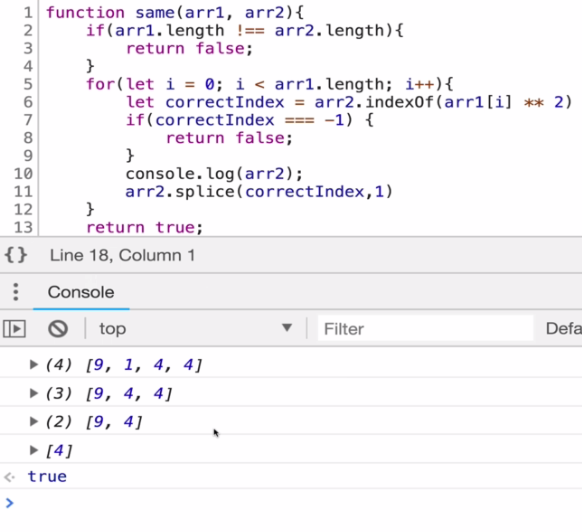
​

​

* Here's a naive solution where we use nested loops. It is .
* Technically we're only writing one ***for-loop*** but then we're using ***indexOf()*** method, which is *itself looping*.
* We have two arrays. Let's say **[1, 2, 3, 2]** and then next array will look like **[9, 1, 4, 4]**. *Order* doesn't matter but *frequency* does. So this should return true.
* The approach here is:

1. First we check the length of two arrays. If they are not same, there will be no frequency.
2. Inside the for loop, we

* loop over each element of the first array
* We search the squared-value in the second array using ***indexOf()*** method (which is a loop itself),
* Then if the value found we delete from the 2nd array using ***splice()***. If not found, the frequency not matched, returns false.
* So the second array decreases.
* Note: You don't have to do this solution with ***indexOf()***. You could manually iterate over the entire array yourself by another for-loop.



* So due to nested-loop the complexity is because for 1000 element array we need 1000,000 checking and 999000 worst cases.
* Refactored solution using Fecuency counter:

*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*

    }

*console***.**log(*frequencyCounter1*);

*console***.**log(*frequencyCounter2*);

*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*,*2*,*5*], [*9*,*1*,*4*,*4*,*11*])

* Here we loop over each array one time individually.
* Remember that two separate loops is better than two nested loops. Two separate loops complexity will be .
* This means 2000 iterations but for nested-loop, we have ***1000*** on the ***outer loop*** and then ***1000*** times on the ***inner loop*** we're looking at **1000x1000** which is a  **1-millioin** compared to what we say **2000**.
* Frequency Counter: The idea behind the frequency counter is: use an object to break down the contents of an array or a string (usually some sort of linear structure like an array or a string) and then you're able to quickly compare that breakdown to how another object looks.
* We had two arrays.
* We break them down into objects that sort of classify what's in those arrays by counting the frequency of elements and store that info as key-value pair.
* Then we can compare those objects.

This allows us to improve our code significantly. Say for 1000 length array, we had 3000 operations using 3 separated for loops where in nested-loop we had 1000000 operations.

* Walk-through: In above code, we had 3 loop

1. First 2 loops first two loops counts the frequency for the 2 arrays and creates a key-value pair.

    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

    }

1. Then in the 3rd loop we access each key of the first key-value object (represents the first array) and we do two things:
2. Check if the square of the 1st objects key is a key of 2nd object. i.e **(key\_1st\_obj)^2 = key\_2nd\_obj**

*key in* *frequencyCounter1*

then

*key* \*\* *2* *in* *frequencyCounter2*

1. Then we check that the keys **key\_1st\_obj** and **key\_2nd\_obj** have same frequency

*frequencyCounter2*[*key* \*\* *2*] !== *frequencyCounter1*[*key*]

We do these using:

    for(let key in frequencyCounter1){

        if(!(key \*\* 2 in frequencyCounter2)){

            return false

        }

        if(frequencyCounter2[key \*\* 2] !== frequencyCounter1[key]){

            return false

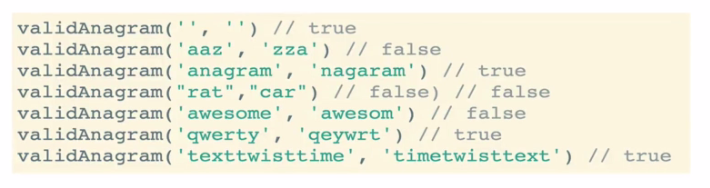
        }

    }

If all condition satisfied, finally it returns ***true***.

|  |  |
| --- | --- |
|  | |
|  |  |

* Example 3.4: Frequency Counter Anagram. 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***.



**Practiced Version**

function same(str1, str2){

    if(str1.length !== str2.length){

        return false;

    }

    let frequencyCounter1 = {}

    let frequencyCounter2 = {}

    for(let val of str1){

        frequencyCounter1[val] = (frequencyCounter1[val] || 0) + 1

    }

    for(let val of str2){

        frequencyCounter2[val] = (frequencyCounter2[val] || 0) + 1

    }

    console.log(frequencyCounter1);

    console.log(frequencyCounter2);

    for(let key in frequencyCounter1){

        if(!(key in frequencyCounter2)){

            return false

        }

        if(frequencyCounter2[key] !== frequencyCounter1[key]){

            return false

        }

    }

    return true

}

​

same("hoof", "foho")

​

**Instructor Version**

function validAnagram(first, second) {

    //    Cheking the length first

    if (first.length !== second.length) {

        return false;

    }

    /\*Creating dictionary type object with key-value pair\*/

    const lookup = {};

    for (let i = 0; i < first.length; i++) {

        let letter = first[i];

        //  if letter exists, increment,, otherwise set to 1

        lookup[letter] ? lookup[letter] += 1 : lookup[letter] = 1;

    }

​

    console.log(lookup);

    /\* Check the 2nd String and compare it to first-string with frequency \*/

    for (let i = 0; i < second.length; i++) {

        let letter = second[i];

        // can't find letter or letter is zero then it's not an anagram

        if (!lookup[letter]) {

            return false;

        } else {

            /\*if letter is found in the "lookup" object, then decrease the value.\*/

            lookup[letter] -= 1;

        }

    }

    /\*If all condition satisfied return true\*/

    return true;

}

​

// {a: 0, n: 0, g: 0, r: 0, m: 0,s:1}

validAnagram('anagrams', 'nagaramm')

* This version is better than practiced version.
* It has only 2 for-loops, where 1st version has 3 loops.
* It uses different approach to count frequency. It does two job simultaneously using if (!lookup[letter]):

1. Look for a existed letter, if it doesn't exist then lookup[letter] is ***false*** (i.e. 0), and !lookup[letter] is ***true***.
2. Other-hand if letter exist but *frequency* is *different*, then for at-least one letter ***lookup[letter]*** will be **0** (i.e ***false***). Then !lookup[letter] is ***true***.
3. Else the letter is found in the "lookup" object, then *decrease* the *value* of the found key-letter

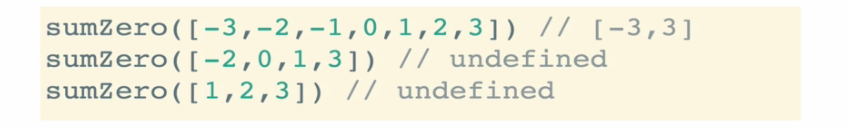
**lookup[letter] -= 1;**

**3.11 Multiple Pointers Pattern**

* 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

|  |  |
| --- | --- |
| * We are dealing with some sort of Linear structure like:  1. an **Array** or 2. a **String** or 3. a **Doubly Linked List** or 4. a **Singly Linked List**. |  |

* The *idea is* that we are going to *searching* for a *pair of values* or searching for something that meets the condition (usually we're looking for a pair). We use two references.
* So we put one reference maybe at the ***end*** and another one at the ***start*** and we walk towards the ***middle***, towards each other. (Actually there is no specific direction for the pointers, we move them according to certain condition.)
* Example 3.5: Write a function called ***sumZero*** which accepts a *sorted array* of *integers* (sorted from lowest to highest). The function should find the first pair where the sum is zero.



* The key point is: We are working with Sorted Array. If it was ***unsorted*** then we're kind of *out of luck* as far as *coming up* with an *efficient solution*.
* The naive solution: Here we have two for-loops, first loop starts from the lowest value and 2nd loop compare the rest of the values. So there are lots of iterations.
* For example: **sumZero([-3,-2,-1,0,1,2,3])**
* First loop starts a ***pointer*** looking at ***- 3*** and then we have a second loop that iterates over the *entire rest of the array*.
* If the value is not found, first loop set the pointer to -2 and second loop iterates *rest of the array*.

**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]];

            }

        }

    }

}

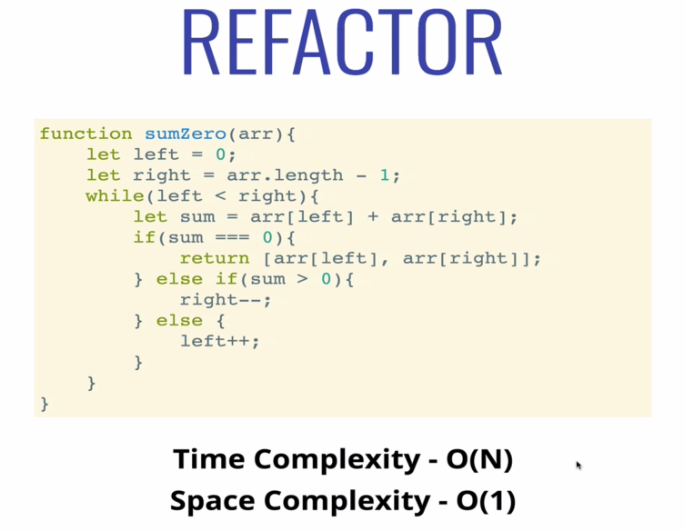
​

​

sumZero([-4,-3,-2,-1,0,1,2,5])

|  |  |
| --- | --- |
| Time complexity **O(n^2)**  Space complexity **O(1)** |  |

* If we're talking about a 10000 item array that's a lot of iteration when we have these nested loops.



* Step by Step walk through: Lets consider the array:

[-4,-3,-2,-1,0,1,2,5])

1. We set **first** **pointer** at lowest value **-4** (start) and **other pointer** at **highest** value **5** (end). We add these values, which results **1 > 0**.
2. In this case we move ***second*** pointer one step ***toward*** the ***first*** pointer (by setting it to value ***2***, ***one index decrease*** ).

Then ***-4 + 2 = -2 < 0***

1. In this case we move ***first*** pointer one step ***toward*** the ***second*** pointer (by setting it to value ***-3***, ***one index increase*** ). Now it results **-3 + 2 = -1**.
2. Now we move ***first*** pointer one step ***toward*** the ***second*** pointer (by setting it to value ***-2***, ***one index increase*** ). And finally we get **-2 + 2 = 0**.

* If no match is found, the pointers stopped where they collide with each other.

So we can see, it is improved version. And we get the result after a few iterations.

​

function sumZero(arr){

    let left = 0;                  /\*set at start of the array\*/

    let right = arr.length - 1      /\*set at end of the array\*/

    while(left < right){

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

        if(sum === 0){

            return [arr[left], arr[right]];

        } else if(sum > 0){

            right--;

        } else{

            left++;

        }

    }

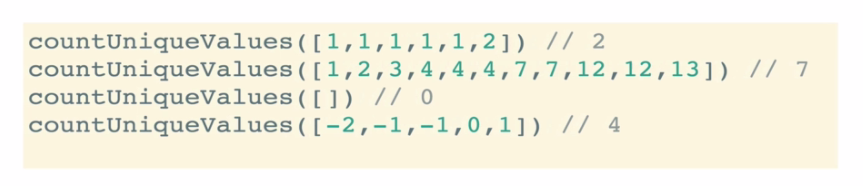
}

​

sumZero([-4,-3,-2,-1,0,1,2,5])

1. ***left*** and ***right*** are two pointers.
2. Notice we've used **while(left < right)** instead of **while(left <= right)** to avoid false-positive **0-0 = 0**
3. We *compare* the values and *increment/decrement* the *index* in the *if-else* ladder.
4. Here we set two pointers at *opposite direction*, we also can use two pointers at *same direction*.

* Example 3.6: **countUniqueValues**. Another example to use Multiple Pointers Pattern.
* 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*.



Hint:

* Use two pointers at the same direction.
* Set *first-pointer* at **0-index** and *second-pointer* at **1-index**. Compare them:
* Same: *Move* the *second* pointer to next index, *first* is *unmoved*
* Different: *Interchange* the values (between the indices), set the *interchanged value* at ***k+1*** th index, where *first-pointer* at ***[k]***, move *first-pointer* at ***[k+1]*** and move *second-pointer* to next index.

**Practiced Version (O(n) time complexity)**

function countUniqueValues(arr){

    var i = 0;

    if(arr.length === 0) return 0;

    for(var j=1; j< arr.length; j++){

        if(arr[i] !== arr[j]){

            i++;            /\* increment the first pointers index \*/

            arr[i]=arr[j]; /\* Interchange the values \*/

        }

    }

    return i+1; /\*Counted unique values\*/

}

​

/\*Works with sorted array\*/

countUniqueValues([0, 0, 1, 2, 3, 3, 3, 4])

​

​

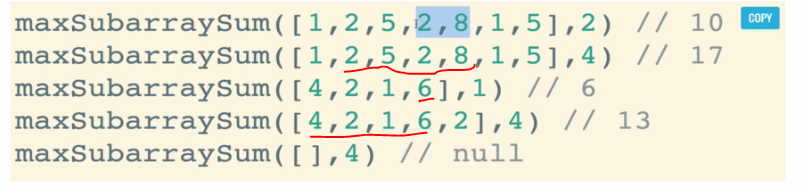
**3.12 Sliding Window Pattern**

*Sliding Window Pattern* is really is useful when we have a set of data like an *array* or a *string* and we're looking for a subset of that data that is continuous in some way.

* For example to find the longest sequence of unique characters from a string:



* 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 3.7: Another example is: finding a **max subarray sum**. 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.



**Naïve Approach**

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;

}

​

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

​

* In this naïve approach, we used two for-loop, and obviously it is O(n^2).
* var max = -Infinity; in case of –ve integers.
* *arr.length - num + 1* because we have to stop for the *last n values*.

**Sliding Window Approach**

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)

* In this approach, we don’t calculate the sum entirely for each step, i.e. we get rid of the inner loop. The trick is:
* We calculate the sum of first n-values.
* Then we just add **sum + next\_(n+1)th\_value – first\_index\_value**.
* Assume at step i, then

**tempSum = tempSum - arr[i - num] + arr[i];**

* We are just sliding the window, we do not count the sum after the first step
* We minus the first element from the window and adding the new next element. That's the trick to calculating the sum by not using a loop.
* For example: say we have **[a, b, c, d, e, f, g]** we have **n=3**
* In naïve approach: sum1 = a+b+c then

sum2 = b+c+d and we compare these sum.

* But in Sliding Window: sum1 = a+b+c,

sum2 = sum1 – fisrt\_element + new\_element = sum1 –a + d

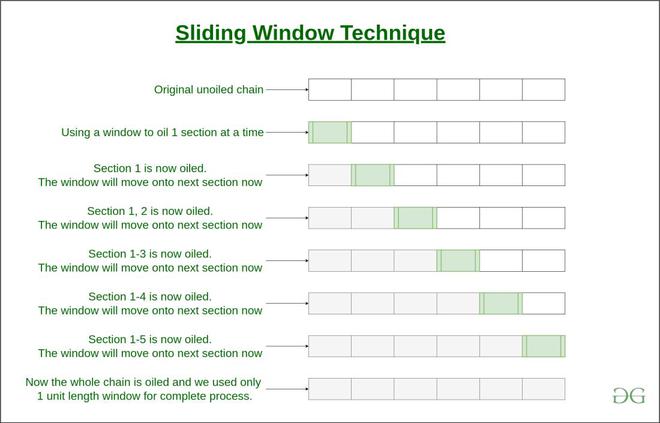
Here we do not need to use nested for-loop.

* What is Sliding Window?
* Consider a long chain connected together. Suppose you want to apply oil in the complete chain with your hands, without pouring the oil from above.

One way to do so is to:

* pick some oil,
* apply onto a section of chain,
* then again pick some oil
* then apply it to the next section where oil is not applied yet
* and so on till the complete chain is oiled.
* Another way to do so, is to use a cloth, dip it in oil, and then hold onto one end of the chain with this cloth. Then instead of re-dipping it again and again, just slide the cloth with hand onto the next section, and next, and so on till the other end.

The second way is known as the Sliding window technique and the *portion* which is *slided* from one end to end, is known as *Sliding* *Window*.



* Sliding Window Technique:

1. We compute the sum of first ***k elements*** out of ***N terms*** using a linear loop and store the ***sum*** in variable ***window\_sum***.
2. Then we will graze linearly over the array till it *reaches* the *end* and simultaneously *keep* *track* of *maximum sum*.
3. To get the current *sum* of *block of* ***k*** *elements* just *subtract the first element* from the *previous block* and add the *last element* of the *current block* .

The below representation will make it clear how the window slides over the array.

* For example: consider an array **arr = [5, 2, -1, 0, 3]** and value of window **k = 3** and array length **n = 5**
* This is the ***initial phase*** where we have calculated the ***initial window*** ***sum*** starting from ***index 0*** . At this stage the window sum is ***6***. Now, we set the ***maximum\_sum*** as ***current\_window*** i.e ***6***.



* Now, we *slide our window* by a *unit* *index*. Therefore, now it *discards* ***5*** from the window and *adds* ***0*** to the window. Hence, we will get our *new window sum* by subtracting ***5*** and then adding ***0*** to it. So, our window sum now becomes 1. Now, we will compare this ***window\_sum*** with the ***maximum\_sum***. As it is smaller we wont the change the ***maximum\_sum***.



* Similarly, now once again we *slide our window* by a *unit index* and obtain the *new window* *sum* to be ***2***. Again we check if this current window sum is greater than the ***maximum\_sum*** till now. Once, again it is smaller so we don’t change the ***maximum\_sum***.
* Therefore, for the above array our ***maximum\_sum*** is 6.



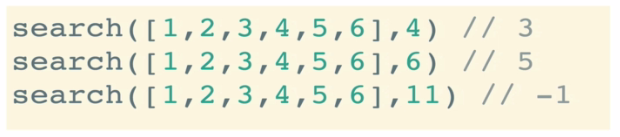
Now, it is quite obvious that the *Time Complexity* is *linear* as we can see that *only one loop runs* in our code (other loop just initially calculate the initial sum). Hence, our Time Complexity is **O(n)**.

We can use this technique to find ***max/min k-subarray***, ***XOR***, ***product***, ***sum***, etc. Refer sliding window problems for such problems.

**3.13 Divide And Conquer Pattern**

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*.

* In following chapters, we do a lot of *divide and conquer* algorithms. Most of them are a little bit *complex*.
* We'll talk about different sorting algorithms, quicksort and merge sort are examples of divide and conquer algorithms.
* When we get to binary search trees, we'll talk about divide and conquer.
* The general idea: We take a larger set of data usually an array of string (i could be a linked list or a tree).
* Let's say we're searching for value.
* We start by dividing it into smaller pieces and then doing something to each smaller piece to determine where to go next.
* For 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***.



Let

arr = [1, 2 , 3, 4, 5, 7, 9, 11, 13, 17, 19, 31, 37, 44, 144]

We want to find 37.

* In *divide and conquer* approach,
* We set a mid point (index), say ***13*** is the value of that mid point, then we campare it with the given value 37. That determines to look in first half or second half.
* Again we set mid-value for second half say 31, then we select the mid point again, this time 44, and finally we find 37.
* So we divide up a *larger chunk* of *data* into *smaller chunks*. And we select hose smaller chunks according to conditions and devide those again.
* It Decreases Time Complexity of . Where in liniear search (naïve / brut force approach) the time complexity is **O(n)**.