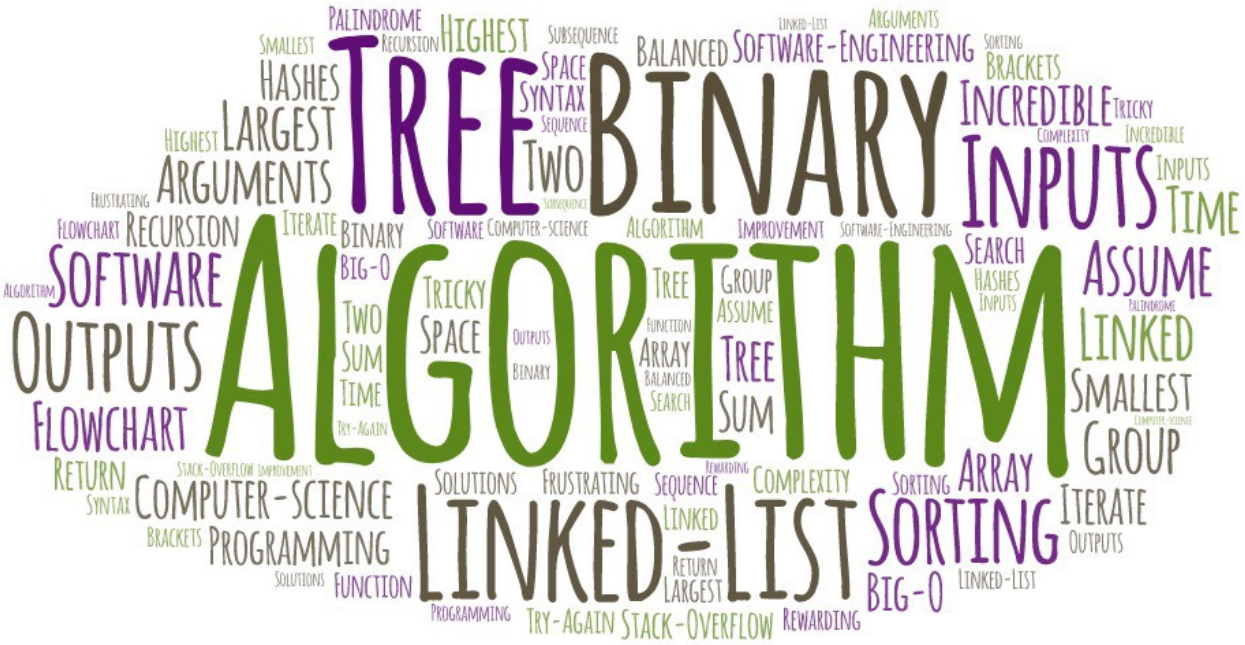
The Validate Subsequence Algorithm: A Deep Dive

Irene Scott Follow

Nov 29, 2020 · 8 min read



Like the way love and money make the world go round, algorithms are what makes the technical job interview go round, along with a sprinkle of data structure fundamentals of course. Understanding these two topics and knowing how to solve them quick is what lays out a firm foundation for any programmer to grasp the skill of how to write efficient code better.

Why is this true? There are many answers to why solving algorithms makes you a better developer, but the simplest way I'd like to address this is by stating a truth. The truth is algorithm is just a fancy word for procedure. In other words it's just a list (sometimes one that has to repeat before ending) of a series of steps a computer follows to perform a task. So if you can code a procedure to get a result, then it makes it easier to apply your patterns in your procedures to coding projects which is what makes it easier to write better code.

Still too complicated? Think of it this way: a cake mix box's instructions is considered an algorithm.

A trip down memory lane before I knew how to make a cake from scratch...

As software developers, and maybe cake makers on the side, our job is to code that list the way a computer would think in the smoothest way possible to complete the task at hand digitally.

Recently this week I have solved a number of algorithm challenges, but to do my part to help the programming community understand the concept of

how to solve algorithms, I am going to be picking out one then explaining how my thought process was when solving the algorithms. Afterwards, using a test example to explain it on a deeper level. The solution to this coding challenge will be coded in JavaScript.

The Validate Subsequence Algorithm

The prompt or problem statement: Given two non-empty arrays of integers, write a function that determines whether the second array is a subsequence of the first one.

A subsequence of an array is a set of numbers that aren't necessarily adjacent in the array but that are in the same order as they appear in the array. For example these numbers

```
[2, 3, 5]
```

are a subsequence of the array:

```
[1, 2, 3, 4, 5]
```

Note that a single number in an array and the array itself are both valid subsequences of the array.

A sample input (what goes into the argument): array = [5, 1, 22, 25, 6, -1, 8, 10]

sequence = [1, 2, -1, 10]

A sample output (an example of what is supposed to come out of the algorithm process): true

Step 1: Ask yourself some essential questions on how the solution will look like based on the output

My first thought in trying to solve this would be to ask myself: “what is the best way to get a true or false result out of a function?” My next thought to answer my own question was of course to use an if statement so at that point I knew I would need an if statement somewhere in my solution.

Then I thought, “How am I going to look through an array to find out if a condition is true?” It would be to use a loop to iterate (look through) over all the numbers they give me in the main array. There are many ways to iterate with JavaScript such as using the while loop, the for loop, and the

for of loop. I selected the for loop because indexes are easier to keep track of that way in my opinion.

Step 2: Code your main structure to hold everything together with the appropriate arguments

With those two thoughts in my head, I had the main structure of my algorithm solution. I started with the keyword function and gave the function an appropriate name and passed in the two arguments they talked about in the question:

```
function isValidSubsequence(array, sequence) {

}
```

Step 3: Code what's inside the main structure based on step 1. Note: Repeat step 1 when necessary.

Now it was time to write my for loop. My for loop needs to check every number in the main array against the subsequence array so I knew that for the first part of my for loop I'd have to start at 0 for the counter variable to check the first number of the main array.

```
function isValidSubsequence(array, sequence) {
   for(i=0;
}
```

Then I knew that I would need to work with the main array all the way until the index of the number being checked reached the end of the list, so I set my condition in my for loop to say that the counter variable value always had to be less than the array's length.

```
function isValidSubsequence(array, sequence) {
   for(i=0; i < array.length;
}
```

Now I needed a way to say that for each time a number was compared in the main array, to move on to the next number in the list to be compared to which is why I used an increment operator.

```
function isValidSubsequence(array, sequence) {
   for(i=0; i < array.length; i++){
```

Now I needed an if statement to see if the current number in the array is equal to the current sequence number in the sequence array. If it was equal, I needed a way to move on to the next number in my sequence array to check that next number. This is why I set up a new variable called j to be set equal to 0 just like the main array's counter variable as a starting point. The condition ended up looking like the following along with my helpful comments.

```
function isValidSubsequence(array, sequence) {
   let j = 0 //set a variable equal to j, this represents the index
of the sequence array


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


//if the number we are on in the first array matches the number we
are on in the second array then add 1 to the variable j
   if (array[i] === sequence[j]) {
         j++


//Add 1 to j
      }
   }


}
```

Step 4: At some point, find some way to specifically end the algorithm solution by returning the output requested.

At this point the for loop will run over and over again checking if the numbers match for each number from each array. Now I thought to myself "there has to be some kind of way to return true or false with an if statement after this loop ends." So I decided to write an if statement that says that if the index of the sequence array is equal to the sequence's length of its array, then return true; otherwise, return false.

```
function isValidSubsequence(array, sequence) {
   let j = 0 //set a variable equal to j, this represents the index
of the sequence array

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

   //if the number we are on in the first array matches the number
we are on in the second array then add 1 to the variable j
   if (array[i] === sequence[j]) {
         j++

//Add 1 to j
      }
   }
  if (j === sequence.length) {

//if the final index of the sequence array is equal to the sequence's
length then return true
```

```
    return true
   } else {

  //in all other cases return false
    return false
   }

  }
```

I coded it that way because as soon as the full sequence is found to be matched to the main array after the sequence's ending index matches the number of items (length) in the sequence, then it is true that the sequence exists in the main array. On the flip side, if not all the numbers in the sequence match some part of the main array then j will not increment as seen in our for loop. Remember j can only increment if the main array's number matches the sequence's number. This is why in all other cases the function will return false.

That's the end of my algorithm solution.

Putting it into Practice — A Test Case Example

At this point I would like to implement a test case using my solution because I know this whole thing can seem confusing without a good example.

Test case input: array = [1, 1, 6, 1] and sequence = [1, 1, 1, 6]

Test case output: false

Okay so here we go:

```
1 function isValidSubsequence(array, sequence) {
 2 let j=0
 3 for(i=0; i < array.length; i++){

 4 if (array[i] === sequence[j]) {
 5 j++
 6 }
 7 }
 8 if (j === sequence.length) {
 9   return true
 10 } else {
    11 return false
 12 }
13 }
```

The first time this goes through we are starting with 1 in the 0th place (represented by i) in the main array and 1 in the 0th place (represented by j) in the sequence array.

In line 4 if the number we are checking, in this case 1, is equal to the number we are checking in the other array, also in this case 1, then increment j. J now equals 1 and i always incrementing by 1 as long as the for loop goes through another round so now i = 1. (i=1, j=1)

The for loop keeps repeating until it has gone through all the numbers in the main array. It repeats by running lines 3 through 7.

The second time this goes through we are starting with 1 again in the 1st place (again represented by i) in the main array and 1 in the 1st place (again represented by j) in the sequence array. Again the numbers we are checking are equal to each other, so now j and i are incremented again. (i=2, j=2)

Now circumstances will change in the 3rd round with 6 in the main array being compared to 1 in the sequence array. This time j will not be incremented because our condition is false, but the variable i will still be incremented. (i=3, j=2)

The for loop runs once again and will again not increment j, so now we technically would have i=4 against j=2. The for loop ends because all elements in the array have been checked and now the code runs at line 8.

```
8 if (j === sequence.length) {
 9   return true
 10 } else {
    11 return false
 12 }
13 }
```

The final check states that if j is equal to the sequence's length in the array, then return true. We know that 2, which represents j, does not equal 3 which is the length of the sequence's array counting from 0 as computers do. So this test case example returns false on line 11.

This algorithm was not the hardest one I've ever done for sure, but was definitely one that put me on a good start to mastering some of the more difficult challenges. There will most likely be some more blog posts just like this later on down this rocky road of algorithms probably because of my fascination with them and my understanding of how important they are to the software development world.

As mentioned earlier, it's important to remember that algorithms are just procedural lists and if you can figure out what to code for each part of the list to end up with the results to make the test cases work, then you've reached a solution.

I would highly recommend if you yourself want to become better at algorithms to try Algoexpert.io. I know you've probably seen the cheesy YouTube ads, but this site really works to make learning algorithms easier with beautiful UI. Happy algorithm solving and cake making :)

Algorithms Data Structures Software Development Web Development JavaScript