Coding Problem: Validate Subsequence in Go

Keep your computer science fundamentals sharp in this Pomodoro session.

Israel Miles Follow

Jun 16, 2021 · 4 min read ★

From Unsplash

The validate subsequence question is an easy problem but introduces the important concept of a sequence in which order matters. This can be compared to finding a subset of an array where the order is irrelevant. In this article, we will understand the subsequence question itself, write tests, and finally develop a solution along with it's big O time and space complexities.

Coding Pomodoro starts now!

Note: *This article assumes you know how to setup a Go workspace and install program dependencies.*

Understand the Problem

Validating a subsequence involves the input of two separate arrays. One of the arrays is a sequence, while the other array is a set of values that may or may not contain that sequence. It's always a good start with coding problems to write up some examples:

```
Example #1
array = [1,0,1]
sequence = [1]
result: true

Example #2
array = [1,5,9]
sequence = [9,5]
result: false

Example #3
array = [1,4,-1,6,4,9,10]
```

```
sequence = [1,-1,10]
result: true
```

What other base case examples can you think of?

Writing Table-Driven Tests in Go

Test Driven Development (TDD) is growing in popularity as it forces you to write minimal code that is highly tested. I've found that one of the best ways to improve your TDD skills is to start with coding problems such as these before working your way up to more standard software development situations. Thankfully Go has fantastic build-in testing support.

After we import the `assert` and `testing` packages, we create the function `TestValidateSubsequence()` which takes a pointer to the `testing.T` object. From there, we can initiate an array of structs as our test cases with four fields to describe each test. We define six total structs to cover the major cases of validating a subsequence, can you think of anymore?

```
1  package subsequence
2
```

```
import (
        "github.com/stretchr/testify/assert"
        "testing"
)

func TestValidateSubsequence(t *testing.T) {
        testCases := []struct {
                name           string
                inputArray     []int
                inputSequence  []int
                expectedResult  bool
        }{
                {
                        name: "empty array",
                        inputArray: []int{},
                        inputSequence: []int{1},
                        expectedResult: false,
                },
                {
                        name: "single element",
                        inputArray: []int{1},
                        inputSequence: []int{1},
                        expectedResult: true,
                },
                {
                        name: "array is shorter than sequence",
                        inputArray: []int{1},
                        inputSequence: []int{1,2,3},
                        expectedResult: false,
                },
                {
                        name: "missing subsequence values",
                        inputArray: []int{1,3,5,2},
                        inputSequence: []int{1,-1,2},
```

```
37                      expectedResult: false,
38              },
39              {
40                      name: "out of order subsequence values",
41                      inputArray: []int{1,5,2,10},
42                      inputSequence: []int{2,5,1},
43                      expectedResult: false,
44              },
45              {
46                      name: "valid subsequence",
47                      inputArray: []int{-1,6,2,8,-4,9},
48                      inputSequence: []int{-1,2,-4,9},
49                      expectedResult: true,
50              },
51      }
52
53      for i := range testCases {
54              tc := testCases[i]
55
56              t.Run(tc.name, func(t *testing.T) {
57                      t.Parallel()
58
59                      actualResult := ValidateSubsequence(tc.inputArray, tc.inputSequence)
60                      assert.Equal(t, actualResult, tc.expectedResult)
61              })
62      }
63  }
```

testvalidatesubsequence.go hosted with ❤ by GitHub view raw

After we define our array of structs to act as test cases, we declare a `for` loop to range over them. We obtain an individual test case, and then we call `t.Run()` to call Go's testing package. We can also call `t.Parallel()` from within for a performance boost (just building good habits). Next we obtain the result from our `ValidateSubsequence()` function (to be implemented next) by supplying it with the `inputArray` and `inputSequence` of the current test case. Finally we assert if the result from our function is the same as what we expect from the test case.

Go has the most elegant and simplistic testing approach of any language, change my mind.

Solution — O(N) Time & O(1) Space

I only present a single solution because there simply aren't many ways to solve this problem without overcomplicating it. You can't use a double for loop to run brute force checking of the subsequence — this would be a valid solution for finding a valid subset and would be $O(N^2)$ time.

Anyways, the solution to this problem is short, but it can be used for more complex problems whenever you need to find or validate a subsequence. Basically, we want to keep track of whether we have seen each element in the sequence before moving onto the next element. If we find a sequence element in the array, then we can increment to the next sequence element while keeping track of where we are at in the array.

So, first we initialize an index for the array as well as the sequence. Then we start a loop that terminates if either index goes out of bounds of its respective array. Our first check is to see if the current array element is equal to the current sequence element. If it is, then we found a sequence element match and can increment our sequence index. Whether we find a match or not, we have to increment the array index to keep moving forward in our search.

```
1  package subsequence
2
3  func ValidateSubsequence(array, sequence []int) bool {
4          arrayIdx := 0
5          sequenceIdx := 0
6          for arrayIdx < len(array) && sequenceIdx < len(sequence) {
7                  if array[arrayIdx] == sequence[sequenceIdx] {
8                          sequenceIdx += 1
9                  }
10                 arrayIdx += 1
11         }
```

```
12          return sequenceIdx == len(sequence)
13  }
```

validateSubsequence.go hosted with ❤ by GitHub view raw

Once the loop terminates, we return the boolean value of whether the sequence index is the same as the length of the original sequence. We do so because the only way this could be true is if we find an in-order match for every sequence element, so that we incremented the sequence index up to the length of the sequence itself.

The running time for this algorithm is bound by the length of the `array`. This is because the sequence is only incremented if there is a match, while we will always increment the `array` index. Even if the sequence is longer than the array, we will terminate once the array index is too large. Thus, our time complexity is O(N) where N is the length of `array`. The space complexity is a constant O(1) because we don't store any additional data structures.

I hope you enjoyed reading this article and were able to learn something new. If you found any parts particularly useful or would like tutorials on any

other coding problems, I encourage you to please leave a comment below! Thanks for reading.