Go Interview Test Explained In Color

REVISION 1

Hawthorne-Press.com

Go Interview Test Explained in Color

Published by

Hawthorne-Press.com

10310 Moorberry Lane
Houston, Texas 77043, USA

Table of Contents

GO INTERVIEW TEST EXPLAINED IN COLOR

INTRODUCTION

The following code example is the result of an interview code test. The criteria stated were as follows:

"*Given an argument, determine if it can be made into a palindrome and output the palindrome or 'No'*"

I was given a short time of explain how I would code against this requirement. While I only had to write a function to determine the outcome, I have decided to explore several programs to accomplish this operation.

As I told the interviewer, the first thing I would do is "brute force" the solution. That is, I would not do anything to optimize the solution before writing some code. In this document, I will present a brute force solution and later a more optimized solution. We discussed the ways to optimize the solution, and the interviewer had a couple of insights that I did not see going in.

The purpose here is to give the readers a window into how to think about a problem and then refine the solution. No developer divines the perfect solution the first time through a problem set, at least not in my experience. In any case, I hope this discussion will help the reader work through development problems.

WHAT IS A PALINDROME?

A ***palindrome*** is a word that can be spelled the same in the both forward and backward directions. For the sake of this problem, we will assume that the characters are a collection of UTF-8 Runes that corresponds to the ASCII Alphabet. That is, the input must contain the letters **a-z** or **A-Z**. We will assume that internally that all characters are converted to lower case for processing and output if it is a valid palindrome. The reader can assume that the input argument will be one space-delimited word.

Before reading any further, please think about the problem and what conditions have to be considered!

- Can the input word have an odd number of characters?
- Does the input contain character values outside the allowed alphabet?
- Do all words with even number of characters qualify as a possible palindrome?
- What will examining the input argument entail?
- How will the output be constructed?

Write down what the solution will entail before going to the next page.

THE SOLUTION OUTLINE

Does the input word contain invalid characters? This simply comes down to filtering the input word. We will use the regex package to filter out words that contain non-alphabetic characters.

The following is the ***cand.go*** program that simply accepts the first argument on the command line as a candidate word for a palindrome. If it contains characters other than alphabetic characters, it prints **"No"** as it output. Otherwise, it converts the candidate word to lower case and prints the result as its output.

```go
// Candidate Validation Test Program
package main

import (
    "fmt"
    "os"
    "regexp"
    "strings"
)

func textNotAlpha(w string) bool {
    r, _ := regexp.Compile(`[^[:alpha:]+]`)
    return r.MatchString(w)
}

func main() {
    argOne := os.Args[1:2]
    if textNotAlpha(argOne[0]) {               // Insure candidate is all alphabetic characters
        fmt.Println("No")
        os.Exit(-1)
    }
    candidate := strings.ToLower(argOne[0])    // Convert candidate to all lower case characters
    fmt.Println("Candidate = ", candidate)
}
```

The program above solves the problem of validating the candidate word. It insures that it contains only alphabetic characters and then converts them to all lower case.

We now have two main questions to answer:

1. Can we make a palindrome of the candidate word?
2. If yes -- How do we construct the palindrome?

Again, take a minute and think about the requirements of a palindrome!

Can The Candidate Become A Palindrome?

Can the input word be odd? The answer is yes with a caveat. You may only have one unique character in the word if its length is odd. Because for a word to read the same way both in directions you need pairs of identical characters such as **abba**. The one exception is if there is one and only one unique character in the input, say the letter **t**, it can be placed in the center of the palindrome. If the input word is **tbaba** then it can be changed into the valid palindrome **batab** or **abtba**.

Leaving out the special case for odd word lengths, the other requirement is that the candidate word contains pairs of identical alphabetic characters. How are we going to accomplish that goal?

Well the "brute force" idea is to setup an array of 26 cells, one for each letter and increment the cell count for each letter found in the candidate word! If array counts are either zero or an even number then we are basically OK. We also allow one and only one cell to contain a one(1), the unique odd character exception. After scanning the candidate word and determining that it meets are requirements, it only remains to build the palindrome.

The following is the brute force function for determining that we have a candidate word that can be made into a palindrome. The important thing to notice about this routine is the running count variable. The character map only keeps track of whether we have seen one or two of the letter in question. If the count is equal to one(1) we return the cell to zero and decrement the running count. Conversely, if the cell is zero, we set the cell to one(1) and increment the running count. If all the characters in the word are paired, then the running count is zero. If there is one and only one unique character the running count is one(1). Thus, we return true if the running count is less than or equal to one(1).

```go
func textMapScan(candidate string) bool {
    var charMap [26]int
    runCount := 0
    for _, char := range candidate {
        c := char - 97          // Zero base the character value
        if c > 26 {
            fmt.Println("Character out of range value = ", char)
        }
        if charMap[c] == 0 {    // Manage Character Counts
            charMap[c] = 1
            runCount += 1
        } else {
            charMap[c] = 0
            runCount -= 1
        }
    }
    if runCount <= 1 {
        return true             // Return True if candidate can be a Palindrome!
    }
    return false
}
```

Take a minute and ask yourself if this can be improved.

A Better Solution

The first thing to consider is instead of an array to store character counts, it is more space efficient to use a map, such as **map[rune]int.** The following is the range loop modified to use a map.

```go
type cMap map[rune]int

func textMapScan(candidate string) bool {
    runCount := 0
    charMap := make(map cMap)                  //Empty Map
    for _, char := range candidate {
        if _, ok := charMap[char]; !ok {        // Add new character to map (first time)
            charMap[char] = 1
            runCount++
        } else {
            if charMap[char] == 1 {
                charMap[char] = 0
                runCount--
            } else {
                charMap[char] = 0
                runCount++
            }
        }
    }
    if runCount <= 1 {
        return true
    }
    return false
}
```

Both the previous solutions do the job of determining whether a word is capable of being turned into a palindrome. However, they do not provide any help in creating a palindrome.

To create a palindrome, we need to know how many of each character we have in a word. Now one solution is to keep the validating function and create a new function that contains duplicated code. That would work but we would be redoing a lot of work. What we need is to keep a count of each character and a valid running count.

We will modify the final function to return both the **charMap** and a Boolean **err** flag. The problem of the odd/even processing is solved by using the integer remainder operator. Additionally, we move the character increment code to avoid duplicating code. The new processing section is as follows:

```go
    for _, char := range candidate {
        if _, ok := charMap[char]; !ok {   // Add new character to the charMap (first time)
            charMap[char] = 1
            runCount++
        } else {
            if charMap[char]%2 == 1 {      // If odd - Then decrement running count
                runCount--
            } else {
                runCount++                 // If even - Then increment the running count
            }
            charMap[char]++                // Increment Character count for each character found
        }
    }
```

Building A Palindrome

Now have the character counts and a validated the candidate word, how do we create the palindrome?

The basic procedure is to build from the center outward:

- If **runCount** is one create a slice with the unique character, otherwise build an empty slice.
- If **runCount** is one set the count for the unique character to zero(0).
- For each character in the **charMap**, divide the count by two(2) and append that number of characters to both the front and back of the slice.

The current main function is as follows:

```go
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage:  %s <candidate word>\n", os.Args[0])
        os.Exit(-1)
    }
    argOne := os.Args[1:2]
    if textNotAlpha(argOne[0]) {             // Insure candidate is all alpha chars
        fmt.Println("No")
        os.Exit(-1)
    }
    candidate := strings.ToLower(argOne[0]) // Convert candidate to all lower case
    fmt.Println("Candidate = ", candidate)

    if charMap, err := textMapScan(candidate); err {
        pWord := palindrome(charMap)
        fmt.Println("The Palindrome is ", pWord)
    } else {
        fmt.Println("No")
    }
}
```

We need a function to create the palindrome with the signature: **palindrome(charMap cMap) string.**

This function will step through the **charMap** and insert characters into the result word. This requires a generalized character inserter that can automatically extend the slice, insert characters at both ends of the slice and in the middle of the slice. The function has the signature **charInsert(i int, s []rune, x rune) []rune.**

Where **'i'** is the index location to insert character **'x'** in slice **'s'.** Index position zero(0) inserts before the first character. Index position **len(s)+1** performs a standard append. All other index positions are inserted *before* the index specified.

```go
func charInsert(i int, s []rune, x rune) []rune {
    s = append(s, x)          // Extend slice by one
    if len(s) == i {          // Return if standard append
        return s
    }
    copy(s[i+1:], s[i:])      // Make space and insert specified character
    s[i] = x
    return s
}
```

PALINDROME FUNCTION

Now that we have the support function, **charInsert(),** creating a palindrome is straight forward.

The palindrome function does the following:

- Create an empty slice.
- Iterate through the **charMap**.
    - If count associated with a character greater than one, process even counts.
        - For the number of pairs, insert a character in front and back of slice
    - If count is equal to one, process unique character
        - Find center index for current slice
        - Insert Unique Character

The actual palindrome function is presented below:

```go
func palindrome(charMap cMap) string {
    s := make([]rune, 0)
    for key, count := range charMap {
        if count > 1 {
            cnt := count / 2                       // Find Number of characters to add
            for i := 0; i < cnt; i++ {
                s = charInsert(0, s, key)          // Add value to front of Slice
                s = charInsert(len(s)+1, s, key)   // Append value at end of Slice
            }
        } else if count == 1 {                     // Process Unique Character
            j := len(s) / 2
            s = charInsert(j, s, key)              // Insert Unique Character at center position
        }
    }
    return string(s)
}
```

Notice that the **charInsert()** support function is what makes the palindrome function easy to implement.

The full listing follows on the next page.

Complete Program Listing

```go
package main

import (
    "fmt"
    "os"
    "regexp"
    "strings"
)

type cMap map[rune]int

func textNotAlpha(w string) bool {
    r, _ := regexp.Compile(`[^[:alpha:]+]`)
    return r.MatchString(w)
}

func textMapScan(candidate string) (charMap cMap, err bool) {
    runCount := 0
    charMap = make(cMap)
    for _, char := range candidate {
        if _, ok := charMap[char]; !ok { // Insert Characters not in charMap
            charMap[char] = 1
            runCount++
        } else {
            if charMap[char]%2 == 1 {     // Decrement running Count if odd
                runCount--
            } else {
                runCount++                // Increment running Count if even
            }
            charMap[char]++               // Increment specified character Count
        }
    }
    if runCount <= 1 {
        return charMap, true              // Valid Palindrome Candidate
    }
    return charMap, false                 // Invalid Palindrome Candidate
}

func charInsert(i int, s []rune, x rune) []rune {
    s = append(s, x)                      // Extend slice by one
    if len(s) == i {                      // Return if standard append
        return s
    }
    copy(s[i+1:], s[i:])                  // Make space and insert specified character
    s[i] = x
    return s
}

func palindrome(charMap cMap) string {
    s := make([]rune, 0)
    for key, count := range charMap {
        if count > 1 {
            cnt := count / 2 // Find Number of characters to add
            for i := 0; i < cnt; i++ {
                s = charInsert(0, s, key)        // Add value to front of Slice
                s = charInsert(len(s)+1, s, key) // Append to end of Slice
            }
        } else if count == 1 {           // Process Unique Character
            j := len(s) / 2
            s = charInsert(j, s, key)    // Insert Unique Character at center position
        }
    }
    return string(s)
}

func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage:  %s <candidate word>\n", os.Args[0])
        os.Exit(-1)
    }
    argOne := os.Args[1:2]
    if textNotAlpha(argOne[0]) {              // Insure candidate is all alpha chars
        fmt.Println("No")
        os.Exit(-1)
    }
    candidate := strings.ToLower(argOne[0])  // Convert candidate to all lower case

    if charMap, ok := textMapScan(candidate); ok {
        pWord := palindrome(charMap)
        fmt.Println("The Palindrome is ", pWord)
    } else {
        fmt.Println("No")
    }
}
```

A Final Word

This White Paper shows what can be done in a relatively short time.

I tried to show some of the refactoring of functions I performed and my reasoning for the decisions I made.

As a developer, never get “married” to a design decision. If it becomes obvious that a piece of code is awkward or problematic, do not let pride keep you from changing it. *The only people who do not make mistakes are the ones who are not working.*

Strive for simplicity in your code. The *Occam’s Razor principle* applies as strongly for software as it does for many other walks of life. In most situations, giving up a little in efficiency to simply your code is often the right decision. Code Simplicity reduces the chances of subtle errors.

As always, comments are welcome at hawthornepresscom@gmail.com.

As to why I have an email address on **gmail** instead of on my website, has to do with AT&T and their inability to handle the concept that not everyone is running a Microsoft OS and my stubborn insistence on running Linux.

Published by

Hawthorne-Press.com

10310 Moorberry Lane
Houston, Texas 77043, USA

Revision History

Date	By	Section	Changes
05/15/2016	C.E. Thornton	All	Initial Document