Palindromes assignment

Izaak van Dongen

December 8, 2017

Contents

Introduction and definitions

In general, we say we have some 1-indexed string of letters s of length k, and let I denote the set of valid indices ($I = \{x : x \in \mathbb{N} \wedge 1 \leq x \leq k\}$). We say that the following predicate determines if a string is a palindrome: $\forall i \in I\ s_i = s_{k-i+1}$.

To make the following programs better fit the specification, we also say that strings containing characters other than letters are palindromes if the sequence of letters that do occur in the string forms a palindrome. We also say that we do not consider the casing of characters when testing for equality. These additions mean that, for example, 'race car' and 'Dennis and Edna sinned!' are palindromes.

However, this approach involves a lot of redundancy. We can use the property that equality is symmetric ($a = b \iff b = a$) to formulate a new set of indices to be checked: ($I' = \{x : x \in \mathbb{N} \wedge 1 \leq x \leq \left\lfloor \frac{k}{2} \right\rfloor\}$). (as we know that $s_i = s_{k-i+1} \iff s_{k-i+1} = s_i$).

Note that the flooring operation here excludes the middle index of an odd string. This is desired, as the middle index of an odd string has index $i = \frac{k+1}{2}$. Substituting into our predicate, this yields the condition $s_{\frac{k+1}{2}} = s_{k - \frac{k+1}{2} + 1}$. By some simplification, we can see that $k - \frac{k+1}{2} + 1 = \frac{2k-(k+1)+2}{2} = \frac{k+1}{2}$ This is the same as the index, so by the reflexivity of equality we know the condition will always hold, and we don't have to check it.

Checking from either end

The first approach to this is to simply check if each character 'lines up' with its complementary character at the other end of the string. Once the middle of the string is reached, the algorithm terminates.

Imperative Pascal

Below is a simplistic function implementing the idea - track an upper and lower bound, and draw them together, checking pairs iteratively:

```
function nr_ispal(s: string): boolean;
var
    lower, upper: integer;
begin
    lower := 1;
    upper := length(s);
    while upper > lower do
        if s[lower] <> s[upper] then
            exit(False)
        else begin
            lower := lower + 1;
            upper := upper - 1;
        end;
    nr_ispal := True
```

```
end;
```

Listing 1: Nonrecursive palindrome function in Pascal

This uses the more C-like idiom of breaking from a for loop, as opposed to explicitly defining some while loop to stop. To match this iterative function, I wrote another iterative function to 'clean' an input string. This uses a pretty simple for ... in loop to iterate over each character, excluding non-alpha characters. Checking if characters are letters is done with a Pascal set expression, and the resultant string is built up by the concatenation operator, which apparently does not trigger intermediary object creation so is not a concern in Pascal.

```
function nrclean(s: string): string;
var
    i: char;
begin
    nrclean := '';
    for i in s do
        if i in ['A'..'Z','a'..'z'] then
            nrclean := nrclean + LowerCase(i);
end;
```

Listing 2: Nonrecursive 'clean' function in Pascal

These two functions then culminate in the following full program:

```
program Palindromes;

uses
    SysUtils;

...

var
    i: integer;
begin
    for i := 1 to ParamCount do
        writeln(ParamStr(i), ' ', nr_ispal(nrclean(ParamStr(i))));
end.
```

Listing 3: Remainder of nonrecursive Pascal program

My chosen method of input is by command-line arguments, as this is easily repeatable, and even automatable - this way of taking parameters for console scripts is far more standard than by a menu interface that reads from stdin presumed to be a tty. It means that, for example, I can more easily call this entire program from another script, which might be useful for testing purposes.

By the way, speaking of C ...

Imperative palindrome checking in C

```
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h>
#include <string.h>

// also return true is char is a null byte so we can stop at the end of the
// string
bool alpha_or_null(char c) {
    return c == '\0' || isalpha(c);
}

// seek the next two letters that need to be compared in a palindrome testing
// scheme. lo can be safely incremented until it hits the end of the string,
// but hi must check that it doesn't go past lo (so that it doesn't go past hte
// start of the string. Returns true if the indices haven't passed each other.
bool seek_next(char s[], int *lo, int *hi) {
    while (!alpha_or_null(s[++*lo]));
    while (!alpha_or_null(s[--*hi])) {
        if (*hi <= *lo)
            return false;
    }
    return *lo < *hi;
}

// determine if string is palindrome
bool is_pal(char s[]) {
```

```c
    int lo = -1; // initialise lo to be out of bounds
    int hi = strlen(s);
    if (!hi)
        return true;
    // continue to contract the indices until either the letters aren't the
    // same or they pass each other
    while (seek_next(s, &lo, &hi)) {
        if (tolower(s[hi]) != tolower(s[lo])) {
            return false;
        }
    }
    return true;
}

int main(int argc, char **argv) {
    for (int i = 1; i < argc; i++) {
        printf("\"%s\": palindromity %s\n",
                        argv[i],
                        is_pal(argv[i]) ? "true" : "false");
    }
    return 0;
}
```

Listing 4: Palindrome function in C

This uses a number of lower-level constructs - this is also exceptional in that it does not first 'clean' the string - rather, the palindrome function itself is written to ignore non-alphabetic characters.

Recursive palindrome checking in Pascal

What I've been using so far, this whole conception of 'loops' seems a little antiquated, and restricts the class of problems we can solve (without implementing a stack). The same algorithm can also be implemented entirely recursively, without the need for loops or even var statements:

```pascal
function _is_palindrome(s: string; lower, upper: integer): boolean;
begin
    if lower >= upper then
        _is_palindrome := True
    else if s[lower] = s[upper] then
        _is_palindrome := _is_palindrome(s, lower + 1, upper - 1)
    else
        _is_palindrome := False;
end;

function is_palindrome(s: string): boolean;
begin
    is_palindrome := _is_palindrome(s, 1, length(s));
end;
```

Listing 5: Recursive palindrome function in Pascal

This uses a wrapper function to 'initialise' the variables. As this function is tail-recursive, at least in theory it should be optimisable to exactly the same machine code as the earlier, boring implementation. The benefits of writing functions like this is that once you get used to the idea, it can actually be very easy to write faultless, performant code. All you need to do is consider how the problem might be approached as suproblems, and consider the possible base cases. This kind of formalisation also encourages a good understanding of what the problem formulation actually means.

As it happens, because this problem lends itself so well to a recursive, functional approach, as shown just now, it can be really quite concisely encoded in Haskell.

```haskell
import System.Environment
import Control.Monad
import Data.Char

is_pal :: (Eq a) => [a] -> Bool
is_pal [] = True
is_pal [_] = True
is_pal (x:xs) = (x == last xs) && (is_pal $ init xs)

letters :: [Char] -> [Char]
letters = (map toLower) . (filter isAlpha)

main :: IO ()
main = do
    tests <- getArgs
```

```
    forM_ tests (\w -> do
                    (putStr . show) w
                    putStr " "
                    (print . is_pal . letters) w
                )
```

Listing 6: Palindrome function in Haskell

These 20 lines are actually the full program, including stylistic whitespace/formatting and type declarations. This is so much more concise because Haskell is designed around the functional paradigm, and because certain 'string' operations become a lot easier in Haskell. Haskell strings can be treated as immutable linked lists of characters, which means that you can just pass subsections of the string to the next function, rather than indices, without having to worry about performance, as it doesn't involve intermediary object creation.

Obligatory slow approach

As the assignment spec requires, I shall also provide another 'approach'. This is to first reverse the string, and then perform a straightforward comparison.

```
program revpals;

uses
    SysUtils;

function _clean(s: string; i: integer): string;
begin
    if i > length(s) then
        _clean := ''
    else if s[i] in ['A'..'Z', 'a'..'z'] then
        _clean := LowerCase(s[i]) + _clean(s, i + 1)
    else
        _clean := _clean(s, i + 1);
end;

function clean(s: string): string;
begin
    clean := _clean(s, 1);
end;

function _reverse(s: string; i: integer): string;
begin
    if i > length(s) then
        _reverse := ''
    else if s[i] in ['A'..'Z','a'..'z'] then
        _reverse := _reverse(s, i + 1) + LowerCase(s[i])
    else
        _reverse := _reverse(s, i + 1);
end;

function reverse(s: string): string;
begin
    reverse := _reverse(s, 1);
end;

function is_pal(s: string): boolean;
begin
    is_pal := reverse(s) = clean(s);
end;

var
    i: integer;
begin
    for i := 1 to ParamCount do
        writeln(ParamStr(i), ' ', is_pal(ParamStr(i)));
end.
```

Listing 7: Method by string reversal

This program is pretty boring, albeit recursive. It uses a similar approach to the previous 'clean' function to reverse the string and clean it at the same time - in fact the only modification needed is to append rather than prepend filtered characters.

Testing

Having written all my programs, I first needed to compile them. To this end I wrote a brief, *very* simplistic Makefile:

```
.PHONY: all
all:
  make bin/cpals
  make bin/hpals
  make bin/recpals
  make bin/revpals
  make bin/Palindromes

bin/cpals: cpals.c
  gcc -Wall -std=gnu99 -pedantic -O3 -o bin/cpals cpals.c

bin/hpals: hpals.hs
  ghc -Wall -O2 -outputdir bin -o bin/hpals hpals.hs

bin/recpals: recpals.pas
  fpc -vw -TLINUX -O3 -obin/recpals recpals.pas

bin/revpals: revpals.pas
  fpc -vw -TLINUX -O3 -obin/revpals revpals.pas

bin/Palindromes: Palindromes.pas
  fpc -vw -TLINUX -O3 -obin/Palindromes Palindromes.pas
```

Listing 8: Makefile for programs

It might also be of note that I used a short makefile to compile this document, too:

```
writeup.pdf: writeup.tex
  pdflatex writeup.tex
  pdflatex writeup.tex
.PHONY: open
open:
  make writeup.pdf
  gnome-open writeup.pdf
```

Listing 9: LaTeX makefile

Anyway, having compiled all the code (using `make all`), I could then go about testing each script:

```
$ bin/Palindromes ""
 TRUE
$ bin/Palindromes ","
, TRUE
$ bin/Palindromes aA
aA TRUE
$ bin/Palindromes "Dennis and Edna sinned"
Dennis and Edna sinned TRUE
$ bin/Palindromes "R a, ,c EC a r                    ^M"
 TRUE,c EC a r
$ bin/Palindromes aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab FALSE
$ bin/Palindromes ab
ab FALSE
```

Listing 10: Initial testing

However this was a little dull... Doing just a couple of inventive tests for a single one of the scripts was turning out to be pretty time-consuming. Conveniently, having implemented a rigid specification in the same way for each script, I could take the *far* quicker approach of writing a script to automate this.

```
"""
Test given palindrome determiner executables
"""

import sys
import os
import shutil
import subprocess
import argparse
import string
import random
import time

std_true = [
    "Dennis and Edna sinned",
    "   d e,NNis aN D EDNASINNED!",
```

```python
    "",
    ",",
    ",a,",
    "!a,abaa",
    "a",
    " a ",
    "race car",
    "a man, a plan, a canal. PANAMA!",
    "aaaaaaaaaaaaabaaaaaaaaaaaaa",
    ]

std_false = [
    "ab",
    "wherefore art thou romeo",
    "python",
    "Guido Van Rossum",
    "aab",
    "a, b",
    "bbbbbbbbbbbbbbbbbbbbbbba",
    "aaaaaaaaaaaaaabaaaaaaaaaaaaa",
    "b,,,,,,,,,,,,,,,,a"]

def generate_pal(n):
    s = "".join(random.choice(string.printable) for _ in range(n))
    return f"{s}{s[::-1]}"

def gen_nonpal(n):
    p = f"{''.join(random.choice(string.printable) for _ in range(n))}{random.choice(string.ascii_letters
    )}"
    app = string.ascii_uppercase[(ord(next(filter(str.isalpha, p)).upper()) - 65 + 13) % 26]
    return f"{p}{app}"

def get_args():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("execs", nargs="*", type=str,
                        help="executable scripts to target")
    parser.add_argument("--no-test", action="store_true",
                        help="don't test each palindrome generated")
    parser.add_argument("-n", type=int, default=500,
                        help="number of extra test cases to generate for both true and false")
    parser.add_argument("-d", action="store_true",
                        help="dump testcases")
    return parser.parse_args()

def is_pal(s):
    s = list(filter(str.isalpha, s.lower()))
    return s == s[::-1]

def is_true(out):
    return "true" in out.decode().lower()

def script_result(script, *args):
    proc = subprocess.Popen([script, *args], stdout=subprocess.PIPE)
    return proc.communicate()

def test_script(script):
    start = time.time()
    exe = shutil.which(script)
    if not exe:
        sys.exit(f"** can't find executable {script!r}")
    for tst in std_true:
        out, err = script_result(exe, tst)
        if err:
            sys.exit(f"{exe!r} wrote {err!r} on {tst!r}")
        if not is_true(out):
            sys.exit(f"{exe!r} failed {out!r} on {tst!r}")
    for tst in std_false:
        out, err = script_result(exe, tst)
        if err:
            sys.exit(f"{exe!r} wrote {err!r} on {tst!r}")
        if is_true(out):
            sys.exit(f"{exe!r} failed {out!r} on {tst!r}")
    print(f"{exe!r} ok in {time.time() - start:.3f}s")

if __name__ == "__main__":
    args = get_args()
    start = time.time()
```

```
    for _ in range(args.n):
        std_true.append(generate_pal(random.randrange(20)))
        std_false.append(gen_nonpal(random.randrange(20)))
    print(f"generated {args.n} testcases in {time.time() - start:.3f}s")
    if not args.no_test:
        print("verifying testcases")
        start = time.time()
        for t in std_true:
            if not is_pal(t):
                sys.exit(f"bad palindrome {t!r}")
        for t in std_false:
            if is_pal(t):
                sys.exit(f"bad non-palindrome {t!r}")
        print(f"finished in {time.time() - start:.3f}s")
    else:
        print("skipping verification")
    if args.d:
        print(f"{chr(10).join(map('true: {!r}'.format, std_true))}\n{chr(10).join(map('false: {!r}'.format, std_false))}")
    if not args.execs:
        print("nb no scripts were supplied")
    for script in args.execs:
        test_script(script)
```

Listing 11: Testing script

This may seem like a very large script that took a lot of time to write, and that may be true, but none of it is particularly interesting - most of the code is book-keeping and tying bits together.

An observant reader may also notice that I've written a two-line, probably somewhat slow Python function to verify the palindromes generated. The other kind of interesting thing here is the automatic palindrome-non palindrome generation. A palindrome is very easy to generate - simply take a string of random characters, and append the same string but reversed This clearly satisfies our condition $\forall i \in I \; s_i = s_{k-i+1}$. as a character at i naturally maps to $k-i+1$. We can also generate a non-palindrome from a string by breaking the condition. We require that the string has at least one letter. We then take the first letter, apply some function that cannot have the same output as an input (the function produces a derangement), and append it to the string. In this case, our derangement is a 'rot13' shift. As this is a derangement, we have $s_1 \neq s_k$ so we know this isn't a palindrome.

The way it determines if an executable's response is affirmitave is by the heuristic 'is 'true' a substring of the output?'

Having written this script, I can test an executable that takes command line arguments by calling it. For ease of use, I wrote a zsh script calling it with all the executables, and passing on all arguments with `$*`.

```
#!/usr/bin/zsh
python3 aux/test.py bin/cpals bin/recpals bin/hpals bin/Palindromes bin/revpals $*
```

Listing 12: Test wrapper script

So at last, I can test my executables:

```
$ aux/all.zsh
generated 500 testcases in 0.011s
verifying testcases
finished in 0.002s
'bin/cpals' ok in 0.654s
'bin/recpals' ok in 0.574s
'bin/hpals' ok in 1.151s
'bin/Palindromes' ok in 0.567s
'bin/revpals' ok in 0.575s
```

Listing 13: Actual testing

The timings here probably aren't particularly indicative of anything - this is because I've written the scripts to only perform one check for their entire runtime, so most of the time is probably IO or system call bound, or is spent "warming up", depending on the mode of compilation. A good confirmation of this is the fact that the bytecode-interpreted Python that has direct access to the array takes around 11ms, close to two orders of magnitude below all the others.

Source

All involved files, including this LaTeX document, can be found at `https://github.com/elterminad0r/palindromes`.