Palindromes assignment

Izaak van Dongen

December 12, 2017

Contents

Introduction and definitions

The notion of an anagram is actually quite simple to set-theoretically represent. We say two strings S and T, of lengths k and l respectively, are palindromes iff $k = l \wedge \{x : x \in S\} = \{x : x \in T\}$. This capitalises on the fact that sets are unordered. As this seems a little terse let's also define $A = B \iff A \subseteq B \wedge B \subseteq A$ where $C \subseteq D \iff \forall x \in C : x \in D$.

Similarly to palindromes, some algorithms work better on just letters, and this is closer to the notion of what an anagram really is. To avoid having "cleanup" code everywhere, this is resolved by saying that the code need only behave correctly when supplied with just letters, and when an input contains a non-letter, behaviour may be considered undefined and will not be tested. An implementation may choose to discard extra letters, or keep them.

Suggested approach by sorting

The suggested algorithm was to apply a selection sort to each string. I first implemented it like this:

```
{$MODE OBJFPC}

program Sort;

uses SysUtils;

procedure swap_vars(var a, b: char);
var
    t: char;
begin
    t := b;
    b := a;
    a := t;
end;

procedure swap_vars(var a, b: integer);
var
    t: integer;
begin
    t := b;
    b := a;
    a := t;
end;

procedure find_minmax(plain: string; lower, upper: integer; out min, max: integer);
var
    i: integer;
begin
```

```
    min := lower;
    max := upper;
    if min > max then
        swap_vars(min, max);
    for i := lower to upper do
        if plain[i] < plain[min] then
            min := i
        else if plain[i] > plain[max] then
            max := i;
end;

procedure selection_sort(var plain: string);
var
    lower, upper, min, max: integer;
begin
    lower := 1;
    upper := length(plain);
    while upper > lower do begin
        find_minmax(plain, lower, upper, min, max);
        swap_vars(plain[max], plain[upper]);
        if min = upper then
            swap_vars(plain[max], plain[lower])
        else
            swap_vars(plain[min], plain[lower]);
        lower := lower + 1;
        upper := upper - 1;
    end;
end;

function is_anagram(a, b: string): boolean;
begin
    selection_sort(a);
    selection_sort(b);
    is_anagram := a = b;
end;

begin
    writeln(is_anagram(ParamStr(1), ParamStr(2)));
end.
```

Listing 1: Initial selection sort

This also implements a small optimisation - rather than just searching for the minimum each pass, it finds a selects both the maximum and minimum. This won't change the complexity of the algorithm, but probably improves the constant factor a bit.

This implementation seemed a little vanilla, so, inspired somewhat by the following Haskell:

```
import System.Environment

set_item :: [a] -> Int -> a -> [a]
set_item (x:xs) 0 y = y:xs
set_item (x:xs) n y = x:set_item xs (n - 1) y

min_of_two :: (Ord a) => a -> a -> a
min_of_two x y | x < y = x | otherwise = y

min_item :: (Ord a) => [a] -> (a, Int)
min_item [a] = (a, 0)
min_item (x:xs) = let (alt, ind) = min_item xs in
                        min_of_two (x, 0) (alt, ind + 1)

sel_sort :: (Ord a) => [a] -> [a]
sel_sort [] = []
sel_sort [a] = [a]
sel_sort xs = let (min_it, pos) = min_item xs in
                        min_it:(sel_sort . tail . (set_item xs pos) . head) xs

is_anagram :: (Ord a) => [a] -> [a] -> Bool
is_anagram x y = (sel_sort x) == (sel_sort y)

main = do
    [a, b] <- getArgs
    print $ is_anagram a b
```

Listing 2: Selection sort in Haskell

I wrote a recursive implementation of the more conventional variation of selection sort in Pascal:

```
program RecSort;

function min(plain: string; a, b: integer): integer;
begin
    if plain[a] < plain[b] then
        min := a
    else
        min := b;
end;

function min_of_string(plain: string; i: integer): integer;
begin
    if i = length(plain) then
        min_of_string := i
    else
        min_of_string := min(plain, i, min_of_string(plain, i + 1));
end;

function swap_chars(plain: string; a, b: integer): string;
var
    t: char;
begin
    t := plain[a];
    plain[a] := plain[b];
    plain[b] := t;
    swap_chars := plain;
end;

function _selection_sort(plain: string; i: integer): string;
begin
    if length(plain) = i then
        _selection_sort := plain
    else
        _selection_sort := _selection_sort(
                              swap_chars(plain, i,
                                         min_of_string(plain, i)),
                              i + 1);
end;

function selection_sort(plain: string): string;
begin
    selection_sort := _selection_sort(plain, 1);
end;

function is_anagram(a, b: string): boolean;
begin
    is_anagram := selection_sort(a) = selection_sort(b);
end;

begin
    writeln(is_anagram(ParamStr(1), ParamStr(2)));
end.
```

Listing 3: Recursive selection sort in Pascal

However none of this was particularly to much avail, as selection sort has a complexity of $O(n^2)$, owed to its linear number of passes it must make. A better idea really would be to implement something like quicksort, which would take $O(n\log(n))$. However, this would in fact also be slower than another sorting-based approach, so I've opted not to do that. As text consists of discrete elements, we can apply an even faster class of sorting algorithm - the integer sort. These don't rely on comparisons, but instead use integer arithmetic, which is generally a lot faster. In this case, the most appropriate would be the counting sort, which has a linear runtime in the length of the list. Counting sort would be appropriate as it is histogram-based, and we have a good restriction on possible characters (ie letters). However, I actually won't implement counting sort either because having constructed a histogram, we can just *compare histograms*.

Comparing letter frequencies

This approach, for all its speed, is actually pretty simple to implement. A multiset, or histogram, of letters can be easily represented as an array of integers indexed by letters.

```
program Freqs;

uses
    SysUtils;

const
    alphabet: set of char = ['a'..'z','A'..'Z'];
```

```pascal

type
    LetterFrequency = array['a'..'z'] of integer;

function new_freq: LetterFrequency;
var
    freqs: LetterFrequency;
    i: char;
begin
    for i := 'a' to 'z' do
        freqs[i] := 0;
    new_freq := freqs;
end;

function compare_freqs(a, b: LetterFrequency): boolean;
var
    i: char;
begin
    for i := 'a' to 'z' do
        if a[i] <> b[i] then
            exit(False);
    exit(True);
end;

function get_freq(plain: string): LetterFrequency;
var
    freqs: LetterFrequency;
    i: integer;
begin
    freqs := new_freq;
    for i := 1 to length(plain) do
        if plain[i] in alphabet then
            inc(freqs[LowerCase(plain[i])]);
    get_freq := freqs;
end;

function is_anagram(a, b: string): boolean;
begin
    is_anagram := compare_freqs(get_freq(a), get_freq(b));
end;

begin
    writeln(is_anagram(ParamStr(1), ParamStr(2)));
end.
```

Listing 4: Basic letter frequencies in Pascal

As the number of letters is constant, the complexity of this program only depends on the length of the text, so has complexity $O(n)$.

The fundamental theorem of arithmetic

Interestingly, as a bit of fun, there is another way to represent a histogram. This is as an integer. We say that some histogram represents the series of frequencies U, from indices 1 to k. This histogram can be encoded as $\prod_{i=1}^{k} P(i)^{U_i}$, where P is the prime function. The fundamental theorem of arithmetic states that each integer corresponds to a unique prime factorisation. This means that each of this prime histogram products corresponds to a unique integer. We can then simply perform an integer comparison to test equality. Another benefit of this approach is that we can 'add' a letter to a histogram simply by multiplying it by the corresponding prime number. Here is the implementation:

```pascal
program Primes;

uses
    SysUtils;

const
    prime_table: array['a'..'z'] of integer =
                        (5, 71, 37, 29, 2, 53, 59, 19, 11, 83, 79, 31, 43,
                         13, 7, 67, 97, 23, 17, 3, 41, 73, 47, 89, 61, 101);
    alpha: set of char = ['a'..'z','A'..'Z'];

function prime_hash(plain: string): integer;
var
    prod: integer = 1;
```

```
    c: char;
begin
    for c in plain do
        if c in alpha then
            prod := prod * prime_table[LowerCase(c)];
    prime_hash := prod;
end;

function is_anagram(a, b: string): boolean;
begin
    is_anagram := prime_hash(a) = prime_hash(b);
end;

begin
    writeln(is_anagram(ParamStr(1), ParamStr(2)));
end.
```

Listing 5: Prime-number anagram checking in Pascal

It's also impressively short, especially considering that this is written in Pascal. A small modification made with regard to the original statement is that the correspondence of letters to primes isn't quite linear in the progression of primes. I have in fact mapped the most frequently occurring letters to the smallest primes. This doesn't have a theoretical effect on the algorithm, but it means that in theory the integers being used should remain a little smaller.

This approach does have a slight drawback. For programming languages with primarily finite integer types, it may cause integer overflow to occur (this is in fact highly likely for longer words, as the value of the integer is roughly exponential in length of text). This can lead to false positives. Interestingly, it cannot lead to a false negative - this is because really an overflowing integer system is a system of modular arithmetic, and multiplication is commutative in modular arithmetic, so a weaker version of the fundamental theorem still holds..

Crossing off letters

For completeness, I thought I should implement the suggested 'crossing off' approach. I decided to try and implement it in some semblance of optimality, so didn't perform any deletions (I suspect these would be very slow, as they require a section of memory to be 'shifted'. Instead, I also created a boolean array to signify the 'crosses'. This is perhaps a nice example of space vs time complexity, as the linear auxiliary space here offers a good increase in time performance. Despite this, it will still have approximately $O(n^2)$ complexity due to the linear number of linear passes.

```
program Slow;

uses
    SysUtils;

function is_anagram(a, b: string): boolean;
var
    available: array of boolean;
    i: integer;
    c: char;
    found_match: boolean;
begin
    if length(a) <> length(b) then
        exit(False)
    else
        setLength(available, length(b));
        for i := 0 to length(b) - 1 do
            available[i] := True;
        for c in a do begin
            found_match := False;
            for i := 0 to length(b) - 1 do
                if  (not found_match)
                    and available[i]
                    and (b[i + 1] = c) then begin
                    available[i] := False;
                    found_match := True;
                end;
            if not found_match then
                exit(False);
        end;
    exit(True);
end;

begin
    writeln(is_anagram(ParamStr(1), ParamStr(2)));
```

```
end.
```

Listing 6: ‘Crossing off’ approach

Brute force

In my journey from the suggested approach to the linear, histogram based approach, I’ve encountered quite a couple of complexities. I thought that to make selection sort feel better, I might implement something even slower. That is, brute-forcing permutations.

```python
import itertools
import sys

def letters(text):
    return "".join(filter(str.isalpha, text))

def is_anagram(a, b):
    return tuple(letters(a)) in itertools.permutations(b)

if __name__ == "__main__":
    print(is_anagram(*sys.argv[1:]))
```

Listing 7: Brute force

We can now add $O(n!)$ to the collection. Unfortunately, exponential complexity remains elusive.

Testing

I, again, wrote a Python script to systematically test my programs.

```python
"""
Integration test a program that should determine if things are anagrams
"""

import argparse
import string
import subprocess
import time
import re
import sys

from random import randrange, choice, shuffle, sample

AN_TRUE = [("OLYMPIAD", "OLYMPIAD"),
           ("LEMON", "MELON"),
           ("COVERSLIP", "SLIPCOVER"),
           ("TEARDROP", "PREDATOR"),
           ("ABBCCCDDD", "DDDCCCBBA"),
           ]

AN_FALSE = [("I", "A"),
            ("FORTY", "FORT"),
            ("ONE", "SIX"),
            ("GREEN", "RANGE"),
            ("ABBCCCDDD", "AAABBBCCD"),
            ]

def strip_suffix(script):
    return re.match(r"(.*)\.pas", script).group(1)

def get_args():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("files", nargs="*", type=str,
                                        help="Pascal files to test")
    return parser.parse_args()

def get_anagrams(length, num):
    yield from AN_TRUE
    for _ in range(num):
        word = [choice(string.ascii_uppercase) for _ in range(length())]
        copy = word[:]
        shuffle(copy)
        yield map("".join, [word, copy])

def get_nonanagrams(length, num):
    yield from AN_FALSE
```

```
47      for _ in range(num):
48          word = [randrange(len(string.ascii_uppercase)) for _ in range(length())]
49          copy = word[:]
50          shuffle(copy)
51          copy[randrange(len(copy))] += randrange(1, len(string.ascii_uppercase))
52          yield ("".join(string.ascii_uppercase[i % len(string.ascii_uppercase)]
53                                                for i in j) for j in (word, copy))
54
55  def test_correct(script, generator, expected):
56      exec_path = "bin/{}".format(strip_suffix(script))
57      comp = subprocess.Popen(
58                  ["fpc", script, "-o{}".format(exec_path), "-Tlinux"],
59                              stdout=subprocess.PIPE,
60                              stderr=subprocess.PIPE)
61      out, err = comp.communicate()
62      if comp.returncode:
63          sys.exit("\ncritical {}".format(err))
64      passes, fails = 0, 0
65      timings = [0] * 7
66      for length in range(1, 8):
67          for a, b in generator(
68                      lambda: randrange(1 << length, 2 << length), 1000):
69              proc = subprocess.Popen([exec_path, a, b],
70                                      stdout=subprocess.PIPE,
71                                      stderr=subprocess.PIPE)
72              start = time.time()
73              out, err = proc.communicate()
74              timings[length - 1] += time.time() - start
75              if err:
76                  sys.exit("critical\n{}".format(err))
77              if out.decode().strip().lower() == expected:
78                  passes += 1
79              else:
80                  fails += 1
81                  sys.exit("\nfailed on input {!r}, {!r} ({})".format(a, b, out))
82              print(
83          "\rtesting {:13}, length {:3}-{:3}, passes {:5}, fails {:5}".format(
84                  script, 1 << length, 2 << length, passes, fails).ljust(80, " "),
85                    end="")
86      return "\n{0}\n{3:13} passed {1:.2%} in {2}\n{0}".format(
87                          "*" * 80, passes / (passes + fails), " ".join("{:6.4f}".format(t) for t in timings
    ),
88                          script)
89
90  def main():
91      args = get_args()
92      pos_results = []
93      neg_results = []
94      for script in args.files:
95          pos_results.append(test_correct(script, get_anagrams, "true"))
96          neg_results.append(test_correct(script, get_nonanagrams, "false"))
97      print("\npos:\n{}\nneg:\n{}".format("\n".join(pos_results),
98                                          "\n".join(neg_results)))
99
100 if __name__ == "__main__":
101     main()
```

Listing 8: Testing script

This hardcodes the prescribed test cases, and also provides function to compute random anagrams, and random non-anagrams. Anagrams are relatively easy - generate a random list and apply a Fisher-Yeates shuffle. Non-anagrams can be generated by doing something similar to with palindromes - generate an anagram and then apply a mutation to one letter, which guarantees that the result will not be an anagram. The program tests these categories separately.

Another fun feature is the use of the carriage return (line 83), to erase the previous line of output, meaning it provides a constantly changing status. Also of interest, at line 57, the script actually compiles the file manually.

Here we test the first couple of programs:

```
$ python aux/test_anagrams.py Freqs.pas RecSort.pas Slow.pas Sort.pas
testing Sort.pas      , length 128-256, passes  7035, fails     0
pos:

********************************************************************************
Freqs.pas     passed 100.00% in 0.1383 0.1399 0.1419 0.1438 0.1410 0.1443 0.1472
********************************************************************************
```

```

****************************************************************************
RecSort.pas    passed 100.00% in 0.1056 0.1082 0.1111 0.1135 0.1358 0.3291 1.2897
****************************************************************************

****************************************************************************
Slow.pas       passed 100.00% in 0.1403 0.1410 0.1410 0.1456 0.1461 0.1748 0.2377
****************************************************************************

****************************************************************************
Sort.pas       passed 100.00% in 0.1384 0.1422 0.1460 0.1450 0.1491 0.1684 0.2100
****************************************************************************
neg:

****************************************************************************
Freqs.pas      passed 100.00% in 0.1416 0.1426 0.1433 0.1439 0.1445 0.1421 0.1484
****************************************************************************

****************************************************************************
RecSort.pas    passed 100.00% in 0.1063 0.1085 0.1120 0.1138 0.1361 0.3252 1.2903
****************************************************************************

****************************************************************************
Slow.pas       passed 100.00% in 0.1393 0.1416 0.1417 0.1424 0.1480 0.1736 0.2273
****************************************************************************

****************************************************************************
Sort.pas       passed 100.00% in 0.1417 0.1428 0.1441 0.1441 0.1478 0.1686 0.2135
****************************************************************************
```

Listing 9: Actual testing

Here I've tested all the programs other than 'Primes'. When I test primes, I get the following:

```
$ python aux/test_anagrams.py Primes.pas
testing Primes.pas    , length 128-256, passes  6048, fails     0
failed on input 'NKYCEOCHALDDNLKPECZREYNQCLZYUFNPZZQBLAAXJDENZOVCQVPXBZMKESUKZQXUVLHTDCIJVAEQFHHWVVCDPJWSIOKIKUVJ
    ', 'QEECHKPEIWKDANJLXRQCZNUBZCTMOUCREIETLMCZEZROPOGWIZLQZQUFHNJTPYHDOGAYPXBBNSETCULSVCSKJBAYQAFYLQXECVQKDTVXA
    ' (b'TRUE\n')
```

Listing 10: Testing 'Primes'

It first successfully passes all the 'positive' tests, then passed the negative tests until the tests started to happen in the larger length range of 128-256, just as I predicted.

Source

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