

Anagrams assignment

Izaak van Dongen

December 12, 2017

Contents

1	Introduction and definitions	1
2	Suggested approach by sorting	1
3	Comparing letter frequencies	3
4	The fundamental theorem of arithmetic	4
5	Crossing off letters	5
6	Brute force	6
7	Testing	6
8	Source	8

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:

```
1  {$MODE OBJFPC}
2
3  program Sort;
4
5  uses SysUtils;
6
7  procedure swap_vars(var a, b: char);
8  var
9      t: char;
10 begin
11     t := b;
12     b := a;
13     a := t;
14 end;
15
16 procedure swap_vars(var a, b: integer);
17 var
18     t: integer;
19 begin
20     t := b;
21     b := a;
22     a := t;
23 end;
24
25 procedure find_minmax(plain: string; lower, upper: integer; out min, max: integer);
26 var
27     i: integer;
28 begin
```

```

29     min := lower;
30     max := upper;
31     if min > max then
32         swap_vars(min, max);
33     for i := lower to upper do
34         if plain[i] < plain[min] then
35             min := i
36         else if plain[i] > plain[max] then
37             max := i;
38 end;
39
40 procedure selection_sort(var plain: string);
41 var
42     lower, upper, min, max: integer;
43 begin
44     lower := 1;
45     upper := length(plain);
46     while upper > lower do begin
47         find_minmax(plain, lower, upper, min, max);
48         swap_vars(plain[max], plain[upper]);
49         if min = upper then
50             swap_vars(plain[max], plain[lower])
51         else
52             swap_vars(plain[min], plain[lower]);
53         lower := lower + 1;
54         upper := upper - 1;
55     end;
56 end;
57
58 function is_anagram(a, b: string): boolean;
59 begin
60     selection_sort(a);
61     selection_sort(b);
62     is_anagram := a = b;
63 end;
64
65 begin
66     writeln(is_anagram(ParamStr(1), ParamStr(2)));
67 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:

```

1 import System.Environment
2
3 set_item :: [a] -> Int -> a -> [a]
4 set_item (x:xs) 0 y = y:xs
5 set_item (x:xs) n y = x:set_item xs (n - 1) y
6
7 min_of_two :: (Ord a) => a -> a -> a
8 min_of_two x y | x < y = x | otherwise = y
9
10 min_item :: (Ord a) => [a] -> (a, Int)
11 min_item [a] = (a, 0)
12 min_item (x:xs) = let (alt, ind) = min_item xs in
13                     min_of_two (x, 0) (alt, ind + 1)
14
15 sel_sort :: (Ord a) => [a] -> [a]
16 sel_sort [] = []
17 sel_sort [a] = [a]
18 sel_sort xs = let (min_it, pos) = min_item xs in
19                 min_it:(sel_sort . tail . (set_item xs pos) . head) xs
20
21 is_anagram :: (Ord a) => [a] -> [a] -> Bool
22 is_anagram x y = (sel_sort x) == (sel_sort y)
23
24 main = do
25     [a, b] <- getArgs
26     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:

```

1 program RecSort;
2
3 function min(plain: string; a, b: integer): integer;
4 begin
5     if plain[a] < plain[b] then
6         min := a
7     else
8         min := b;
9 end;
10
11 function min_of_string(plain: string; i: integer): integer;
12 begin
13     if i = length(plain) then
14         min_of_string := i
15     else
16         min_of_string := min(plain, i, min_of_string(plain, i + 1));
17 end;
18
19 function swap_chars(plain: string; a, b: integer): string;
20 var
21     t: char;
22 begin
23     t := plain[a];
24     plain[a] := plain[b];
25     plain[b] := t;
26     swap_chars := plain;
27 end;
28
29 function _selection_sort(plain: string; i: integer): string;
30 begin
31     if length(plain) = i then
32         _selection_sort := plain
33     else
34         _selection_sort := _selection_sort(
35             swap_chars(plain, i,
36                 min_of_string(plain, i)),
37             i + 1);
38 end;
39
40 function selection_sort(plain: string): string;
41 begin
42     selection_sort := _selection_sort(plain, 1);
43 end;
44
45 function is_anagram(a, b: string): boolean;
46 begin
47     is_anagram := selection_sort(a) = selection_sort(b);
48 end;
49
50 begin
51     writeln(is_anagram(ParamStr(1), ParamStr(2)));
52 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, and I'm somewhat constrained for time, 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.

```

1 program Freqs;
2
3 uses
4     SysUtils;
5

```

```

6  const
7      alphabet: set of char = ['a'.. 'z', 'A'.. 'Z'];
8
9  type
10     LetterFrequency = array ['a'.. 'z'] of integer;
11
12  function new_freq: LetterFrequency;
13  var
14     freqs: LetterFrequency;
15     i: char;
16  begin
17     for i := 'a' to 'z' do
18         freqs[i] := 0;
19     new_freq := freqs;
20  end;
21
22  function compare_freqs(a, b: LetterFrequency): boolean;
23  var
24     i: char;
25  begin
26     for i := 'a' to 'z' do
27         if a[i] <> b[i] then
28             exit(False);
29     exit(True);
30  end;
31
32  function get_freq(plain: string): LetterFrequency;
33  var
34     freqs: LetterFrequency;
35     i: integer;
36  begin
37     freqs := new_freq;
38     for i := 1 to length(plain) do
39         if plain[i] in alphabet then
40             inc(freqs[LowerCase(plain[i])]);
41     get_freq := freqs;
42  end;
43
44  function is_anagram(a, b: string): boolean;
45  begin
46     is_anagram := compare_freqs(get_freq(a), get_freq(b));
47  end;
48
49  begin
50     writeln(is_anagram(ParamStr(1), ParamStr(2)));
51  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 (this is a kind of Gödel coding). 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:

```

1  program Primes;
2
3  uses
4      SysUtils;
5
6  const
7      prime_table: array ['a'.. 'z'] of integer =
8          (5, 71, 37, 29, 2, 53, 59, 19, 11, 83, 79, 31, 43,
9           13, 7, 67, 97, 23, 17, 3, 41, 73, 47, 89, 61, 101);
10     alpha: set of char = ['a'.. 'z', 'A'.. 'Z'];
11
12  function prime_hash(plain: string): integer;

```

```

13 var
14     prod: integer = 1;
15     c: char;
16 begin
17     for c in plain do
18         if c in alpha then
19             prod := prod * prime_table[LowerCase(c)];
20     prime_hash := prod;
21 end;
22
23 function is_anagram(a, b: string): boolean;
24 begin
25     is_anagram := prime_hash(a) = prime_hash(b);
26 end;
27
28 begin
29     writeln(is_anagram(ParamStr(1), ParamStr(2)));
30 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.

```

1 program Slow;
2
3 uses
4     SysUtils;
5
6 function is_anagram(a, b: string): boolean;
7 var
8     available: array of boolean;
9     i: integer;
10    c: char;
11    found_match: boolean;
12 begin
13     if length(a) <> length(b) then
14         exit(False)
15     else
16         setLength(available, length(b));
17         for i := 0 to length(b) - 1 do
18             available[i] := True;
19         for c in a do begin
20             found_match := False;
21             for i := 0 to length(b) - 1 do
22                 if (not found_match)
23                     and available[i]
24                     and (b[i + 1] = c) then begin
25                     available[i] := False;
26                     found_match := True;
27                 end;
28             if not found_match then
29                 exit(False);
30             end;
31         exit(True);
32     end;
33
34 begin
35     writeln(is_anagram(ParamStr(1), ParamStr(2)));

```

```
36 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.

```
1 import itertools
2 import sys
3
4 def letters(text):
5     return "".join(filter(str.isalpha, text))
6
7 def is_anagram(a, b):
8     return tuple(letters(a)) in itertools.permutations(b)
9
10 if __name__ == "__main__":
11     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.

```
1 """
2 Integration test a program that should determine if things are anagrams
3 """
4
5 import argparse
6 import string
7 import subprocess
8 import time
9 import re
10 import sys
11
12 from random import randrange, choice, shuffle, sample
13
14 AN_TRUE = [( "OLYMPIAD", "OLYMPIAD" ),
15            ( "LEMON", "MELON" ),
16            ( "COVERSLIP", "SLIPCOVER" ),
17            ( "TEARDROP", "PREDATOR" ),
18            ( "ABBCCDDDD", "DDDDCCBBA" ),
19            ]
20
21 AN_FALSE = [( "I", "A" ),
22             ( "FORTY", "FORT" ),
23             ( "ONE", "SIX" ),
24             ( "GREEN", "RANGE" ),
25             ( "ABBCCDDDD", "AAABBBCCD" ),
26             ]
27
28 def strip_suffix(script):
29     return re.match(r"(.*)\.pas", script).group(1)
30
31 def get_args():
32     parser = argparse.ArgumentParser(description=__doc__)
33     parser.add_argument("files", nargs="*", type=str,
34                         help="Pascal files to test")
35     return parser.parse_args()
36
37 def get_anagrams(length, num):
38     yield from AN_TRUE
39     for _ in range(num):
40         word = [choice(string.ascii_uppercase) for _ in range(length)]
41         copy = word[:]
42         shuffle(copy)
43         yield map("".join, [word, copy])
44
45 def get_nonanagrams(length, num):
46     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} ({})"
82                        .format(a, b, out))
83             print(
84                 "\rtesting {:13}, length {:3}-{:3}, passes {:5}, fails {:5}"
85                 .format(script, 1 << length, 2 << length, passes, fails).ljust(80, " "),
86                 end="")
87     return "\n{0}\n{3:13} passed {1:.2%} in {2}\n{0}"
88     .format(
89         script,
90         "*" * 80, passes / (passes + fails), " ".join("{:6.4f}".format(t) for t in timings)
91     ),
92     script)
93
94 def main():
95     args = get_args()
96     pos_results = []
97     neg_results = []
98     for script in args.files:
99         pos_results.append(test_correct(script, get_anagrams, "true"))
100        neg_results.append(test_correct(script, get_nonanagrams, "false"))
101    print("\npos:\n{}\nneg:\n{}".format("\n".join(pos_results),
102                                       "\n".join(neg_results)))
103
104 if __name__ == "__main__":
105     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:

```

1 $ python aux/test-anagrams.py Freqs.pas RecSort.pas Slow.pas Sort.pas
2 testing Sort.pas           , length 128-256, passes 7035, fails 0
3 pos:
4
5 *****
6 Freqs.pas      passed 100.00% in 0.1383 0.1399 0.1419 0.1438 0.1410 0.1443 0.1472
7 *****

```

```

8
9 *****
10 RecSort.pas    passed 100.00% in 0.1056 0.1082 0.1111 0.1135 0.1358 0.3291 1.2897
11 *****
12
13 *****
14 Slow.pas       passed 100.00% in 0.1403 0.1410 0.1410 0.1456 0.1461 0.1748 0.2377
15 *****
16
17 *****
18 Sort.pas       passed 100.00% in 0.1384 0.1422 0.1460 0.1450 0.1491 0.1684 0.2100
19 *****
20 neg :
21
22 *****
23 Freqs.pas      passed 100.00% in 0.1416 0.1426 0.1433 0.1439 0.1445 0.1421 0.1484
24 *****
25
26 *****
27 RecSort.pas    passed 100.00% in 0.1063 0.1085 0.1120 0.1138 0.1361 0.3252 1.2903
28 *****
29
30 *****
31 Slow.pas       passed 100.00% in 0.1393 0.1416 0.1417 0.1424 0.1480 0.1736 0.2273
32 *****
33
34 *****
35 Sort.pas       passed 100.00% in 0.1417 0.1428 0.1441 0.1441 0.1478 0.1686 0.2135
36 *****

```

Listing 9: Actual testing

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

```

1 $ python aux/test_anagrams.py Primes.pas
2 testing Primes.pas , length 128–256, passes 6048, fails 0
3 failed on input 'NKYCEOCHALDDNLKPECZREYNQCLZYUFNPZZQBLAAXJDENZOVCCQVPXBZMKESUKZQXUVLHTDCLJVAEQFHWWVCDPJWSIOKIKUVJ
    ', 'QEECHKPEIWKDANJLXRQCZNUBZCTMOUCREIETLMCZETROPOGWIZLQZQUFHNJTPYHDOGAYPXBBSSETCULSVCSKJBAYQAFYLQXECVQKDTVXA
    ' (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 L^AT_EX document, can be found at <https://github.com/elterminad0r/anagrams>.