CISC/CMPE-365
Fall 2019

Week 9 Lab Problem: Comparing Strings

Not to be handed in for grading

Consider these two bits of code:

```
let A[1..n] and B[1..n] be arrays of integers
for i = 1 to n:
        if A[i] == B[i]:
                print(i)
```

and

```
let A[1..n]  and B[1..n] be arrays of strings
for i = 1 to n:
        if A[i] == B[i]:
                print(i)
```

Each of these code pieces contains a loop that executes n times – inside the loop is a comparison and a print statement. The first one is in O(n), but the second is not.

Why not? Because comparing two strings is ***not*** a constant time operation. Comparing two strings takes time proportional to the length of the shorter of the two strings. Basically, we have to compare the strings character by character. Only when we find a mismatch can we stop without comparing the rest of the characters.

But comparing strings is an important operation in many applications (such as search engines, for example!) so it would be nice if there were some way to accelerate the process.

It turns out there is ... and in this lab you will experiment with some variations on the technique.

The basic idea is that if we can represent every string by an integer, then instead of comparing the strings we could just compare the integers – which would take constant time. The problem of course is that there are far more possible strings than the range of possible integers available on any actual computer. We can't do a 1-to-1 mapping from strings to integers. The best we can hope for is a many-to-1 mapping in which not too many strings get mapped to any one integer.

Let $f(s)$ be a many-to-1 function that maps strings to integers. Since f is a function, we know that if $s_1 = s_2$, then $f(s_1) = f(s_2)$. This is equivalent to saying that if $f(s_1) \neq f(s_2)$ then $s_1 \neq s_2$

However because f is many-to-1 it may the case that $s_1 \neq s_2$ and $f(s_1) = f(s_2)$

Here is how we can use this. Suppose we want to know if strings s_a and s_b are equal.

```
We compute $f(s_a)$ and $f(s_b)$.
if $f(s_a) \neq f(s_b)$:
    print("they are not equal")
else:
    if $s_a == s_b$:        #character by character comparison
        print("they are equal")
    else:
        print("they are not equal")
```

This is an improvement on the BFI comparison ... maybe. It pays off if it is the case that most of the time $f(s_a) \neq f(s_b)$. We just compare these integers and because they are unequal we know the strings are different.

We also have to think about the time required to compute $f(s_a)$ and $f(s_b)$ ... these computations are certainly going to look at all the characters in the strings so they will bring the complexity right back to where it was.

We can address the second issue by computing $f(s)$ as soon as the string s is created. Then we pay the cost of computing $f(s)$ just once, no matter how many other strings s is eventually compared to. Since creating a string takes time proportional to its length, if we can compute $f(s)$ in the same time it doesn't cost us anything in terms of computational complexity.

We can address the first issue by trying to find a good function $f(s)$ that keeps the number of "false positives" (situations where $f(s_a) = f(s_b)$ but $s_a \neq s_b$) relatively low. And that, at long last, is the task to be explored in this lab.

Experiment 1:

One of the simplest functions for computing an integer to represent a string is this:

```
def f1(s):
        sum = 0
        for c in s:
                sum += ord(c)
        return sum
```

In other words we just sum the ASCII values of the characters in s. (This is generally regarded to be a poor function for this purpose, but it was actually recommended in a famous text-book – bonus points to anyone who can name the text.)

The file **words1ASCII.txt** (used in the last lab) contains several thousand words, all different. Write a program that will calculate `f1(s)` for all the words in this file, and report the number of times two different words have the same `f1` value (these would be the false positives if we were testing these strings for equality). Compute the ratio of the number of false positives to the number of comparisons made.

Here's the real challenge: let n be the number of words in the file. Compute the number of false positives in O(n) time.

Experiment 2:

Consider this function for mapping strings to integers:

```
def f2(s):
        result = 0
        for c in s:
                result = 2*result + ord(c)
        return result
```

Test this function on **words1ASCII.txt** just as you did for `f1`. How does the ratio of false positives to the number of comparisons for `f2` compare to the ratio for `f1`?

Experiment 3:

There is a problem with `f2` that `f1` is not really prone to. If `s` is a long string, `f2(s)` may well exceed the maximum integer we can represent. For this reason it is common to use a variation that uses modular arithmetic to keep the integers within the legal range.

In this variation we choose two integer constants `a` and `b` to define the function.

```
def f2mod(s):
    result = 0
    for c in s:
        result = (a*result + ord(c) ) % b
    return result
```

For example we might let `a = 7` and `b = 100000` ... the function would look like

```
def f2mod(s):
    result = 0
    for c in s:
        result = (7*result + ord(c)) % 100000
    return result
```

Since this possible integer overflow is only likely to be an issue with long strings, the **words1ASCII.txt** file is not a particularly good test for this function – but try it out anyway.

For this experiment, use the file **TheSunderingFloodASCII.txt**

Apply `f1` to this file and determine the false positive ratio.

Apply `f2mod` to this file and determine the false positive ratio. Experiment with different `a` and `b` values to see if the false positive ratio is sensitive to the values of `a` and `b`. Note: this file is a book written in 1897 and now in the public domain. This book contains some duplicate lines – these will generate the same function values but they are not false positives! This means that when two lines have matching function numbers you will have to compare them to see if they actually match.

Experiment 4:

There is a simple way to reduce the number of false positives even further: compute and save the value of more than one function for each string. In this case the only way to get a false positive is if **all** the functions match on the two strings, even though the strings are different.

For example we might decide to use two functions f(s) and g(s), each of which is a many-to-1 function for mapping strings to integers. To compare strings s_1 and s_2 we would do this:

```
if $f(s_1) \neq f(s_2)$ or $g(s_1) \neq g(s_2)$:
        print("not equal")
else:
        if $s_1 == s_2$:                # character by character comparison
                print("equal")
        else:
                print("not equal")
```

For this experiment choose any two functions you like. You can use `f1`, different versions of `f2mod` (with different `a` and `b` values), or any other function you find or create. The literature is replete with potential functions – try searching for "hashing functions for strings". Try to find two functions that when used together reduce the false positive ratio to almost 0.

Would using 3 functions make a significant difference? Would there be any downside of using 10 functions, or 100?