Computer Science I
Hiding Information

CSCI141
Lecture (2/2)

08/26/2013

1 Problem Statement

For this problem, we have a series of text documents that contain information we wish to hide. We want to remove a name (or possibly other identifying words) from reports that will be distributed publicly.

2 Solution Design and Analysis

2.1 Hiding Algorithm

Let's start with the hiding algorithm. We simply need to examine every word in a file, and if it matches the word we wish to hide, we will do so by displaying three dashes (e.g. 'Maria' becomes '---').

```
def hide(textFile, hiddenWord):
    for currentWord in textFile:
        if currentWord == hiddenWord:
            print '---'
        else:
            print currentWord
```

2.2 From Algorithm to Implementation

To simplify matters we will assume that a file has no more than one word per line. Suppose we have a file (called `text1.txt`) that contains the following text. (Spaces are displayed explicitly.)

```
␣word1
␣␣word2
␣␣␣word3
␣␣word4
␣word5
word6

word7
```

How can we read in this file? It turns out that the `for` loop we've seen can used to loop through the lines in a file. But how do we supply the file? If we just supply the file name to the `for` loop we'll merely loop through that string. We need a new Python function: `open`. It converts a string to a file, which we can then use in the `for` loop.

```
for line in open('text1.txt'):
   print(line)
```

We see the following.

```
word1

␣␣word2

␣␣␣word3

␣␣word4

␣word5

word6



word7
```

Notice that we see all the spaces from the original. Also notice that there are blanks bewteen each line. That's because `print` generates a new-line, and the new-lines in the file are also being displayed. Another way to see this is to put all the lines into a single string. Try it!

We can make the printing look nicer by telling `print` not to generate a new-line as follows `print(..., end='')`; however, this observation also has consequences for the solution to our problem. Even if we require no spaces next to a word in the file, there will still be a new-line at the end, which will make it a different string from the string we are looking for.

```
for line in open('text1.txt'):
   print(line == 'word1')
```

We see the following.

```
False
False
False
False
False
False
```

```
False
False
```

We need one more feature from Python: `strip`. It removes all white space (including new-lines) from the string. The way we invoke this function is a little different from the usual way. We write s`.strip()`, where s denotes a string.

```
for line in open('text1.txt'):
   print(line.strip() == 'word1')
```

We see the following, as expected.

```
True
False
False
False
False
False
False
False
```

Putting these ideas together, we can write the Python function `hide` and a helper function[1] that takes a file name rather than a file.

```
def hide(textFile, hiddenWord):
    """hide: File * String -> NoneType
       Effect: Display the file with modifications.
    """
    for currentWord in textFile:
       if currentWord.strip() == hiddenWord:
           print('---')
       else:
           print(currentWord, end='')

def hideUsingFileName(textFileName, hiddenWord):
    """hideUsingFileName: String * String -> NoneType
       Effect: Display the file with modifications.
    """
    hide(open(textFileName), hiddenWord)
```

2.3 Linear Search and Time Complexity

Although we originally framed this problem as a problem of hiding a word, it is quite reasonable to re-frame it as *searching* for a particular word. Above we search for every

1. A helper function is introduced for two reasons: 1) The `hide` function more closely resembles the pseudo-code. 2) More importantly, it is good practice when writing functions that operate on a kind of object, to, in fact, operate on that kind of object rather than on a representation of that kind of object.

occurrence; however, in many other problems we search for only the first occurrence, and stop if it is found. We will assume this modified scenario involving stopping at the first occurrence for the rest of this sub-section.

Linear search is the process of visiting each element in a sequence in order (e.g. each character in a string, each word in a file, etc.), stopping when either 1) we find what we are looking for, or 2) we have visited every element in the sequence, and not found a match.

Often we are interested in characterizing how much time an algorithm uses. For better or for worse a measurement in, say, seconds, depends on the hardware, the operating system, etc. We'd like to avoid such details, and give a hardware and software independent measure. How can that be done? We focus on the *size* of the input, N, and characterize the time spent using a function of N. This function is called the *time complexity* of the algorithm.

For linear search, we quickly see that different inputs of size N can lead to radically different behaviors and times. Suppose what we're searching for is the first element, then we find it right away; it's fast. But if what we're searching for is last, we must look at every single element; it's slow. (If N is ten, even the slow case is not so bad; but if N is a billion we will start to notice.) Frequently we are interested in how bad it can get and do a *worst-case* analysis.

So let's consider a worst-case analysis of linear search. To find what we're looking for, for each element, we must spend some constant amount of time checking to see if we've found it, plus some fixed start up and shut down time. Since there are N elements, we can bound the (worst-case) time linear search spends by the function $y = kN + c$, where k and c are constants. What are k and c? In fact, very often we simply don't care! Rather, it's enough to know that the time complexity is a linear function with non-zero slope. The mathematical way of ignoring these details is to say that the worst-case time complexity of linear search is $O(N)$ (which is pronounced "Order N").

Constant-time operations, such as comparing a pair of characters, have time complexity $O(1)$. There are also algorithms with time complexity $O(N^2)$. Notice that this notation makes it relatively easy to compare algorithms. We'd prefer an $O(1)$ algorithm to an $O(N)$ algorithm, and we'd prefer and $O(N)$ algorithm to an $O(N^2)$ algorithm. We might also wonder if there are time complexities in between...

3 Testing

In the approach shown in `hiding.py`, we run the function `hideUsingFileName` several times: once when the word being sought is not there; once when it occurs one time; and once when it occurs more than one time. Verification that the output is correct must be done manually.