6.00 Introduction to Computer Science and Programming
Fall 2008

6.00: **Introduction to Computer Science and Programming**

Problem Set 3

Handed out: Tuesday, September 16, 2008.
Due: 11:59pm, Tuesday, September 23, 2008.

Introduction

This problem set will introduce you to using functions and recursion, as well as string operations in Python.

Collaboration

You may work with other students. However, each student should write up and hand in his or her assignment separately. *Be sure to indicate with whom you have worked.* For further detail, please review the collaboration policy as stated in the syllabus.

Submission

This problem set, and future ones, will be graded by a test harness. The test harness program will expect your files to include just function definitions, with no executable code outside the function definitions (besides what's already in the template). So remember to comment out your testing code. (And *do* test your code thoroughly!).

Strings and string searching

As we have seen in lecture, strings are a common data type in many programming languages, and are used to represent textual information. You have already seen common examples of string searching. For example, finding words or phrases in documents involves searching one sequence of characters (i.e., the document) to find instances of another sequence of characters (the word or phrase to be found). Similarly, for Web searches such as Google, one needs to count instances of key words in documents, in order to rank pages.

Matching strings: a biological perspective

String matching is also is very valuable in less obvious settings, such as biology. A common problem in modern biology is to understand the structure of DNA molecules, and the role of specific structures in determining the function of the molecule. A DNA sequence is commonly represented as a sequence of one of four nucleotides – adenine (A), cytosine (C), guanine (G), or thymine (T) –and hence a DNA molecule or strand is represented by a string composed of elements from an alphabet of only four symbols, for example, the string AAACAACTTCGTAAGTATA represents a particular strand of DNA.

One way to understand the function of a particular strand of DNA (or even a sub-strand of DNA) is to match that strand against a library of known DNA sequences – that is, sequences whose function and structure is known – with the idea that similar structure tends to imply similar function. Simple organisms such as bacteria may have millions of nucleotides in their DNA sequence, and the human chromosome is believed to have on the order of 246 million bases, so any matching scheme must be very efficient in order to be useful.

In this problem set, we won't ask you to build a practically useful tool, but hope to give you a sense of some of

the issues involved, by exploring some simple matching schemes.

To get started, we are going to use some built-in Python functions. To use these functions, include the statement

```
from string import *
```

at the beginning of your file. This will allow you to use Python string functions. In particular, if you want to find the starting point of the first match of a keyword string `key` in a target string `target` you could use the find function.

Try running find on some examples, such as find("atgacatgcacaagtatgcat","atgc")

Note how it returns the index of the first instance of the key in the target. Also note that if no instance of the key exists in the target, e.g, find("atgacatgcacaagtatgcat","ggcc") it returns the value -1.

We are going to explore some ideas in matching strings by looking at successively more complex tasks. **NOTE:** the solutions you are going to write will be tested on examples from DNA strings, but you should **not** assume that your solutions will only apply to DNA strings, i.e. do not assume that the strings consist of only 4 different characters, but rather that the strings could contain an arbitrary number of different characters.

Let's start with a fairly simple problem. Suppose we want to count the number of times that a key string appears in a target string. We are going to create two different functions to accomplish this task: one iterative, and one recursive. For both functions, you can rely on Python's find function – you should read up on its specifications to see how to provide optional arguments to start the search for a match at a location other than the beginning of the string. For example,

find("atgacatgcacaagtatgcat","atgc")

returns the value 5, while

find("atgacatgcacaagtatgcat","atgc",6)

returns the value 15, meaning that by starting the search at index 6, the next match is found at location 15.

For the recursive version, you will want to think about how to use your function on a smaller version of the same problem (e.g., on a smaller target string) and then how to combine the result of that computation to solve the original problem. For example, given you can find the first instance of a key string in a target string, how would you combine that result with invocation of the same function on a smaller target string. You may find the string slicing operation useful in getting substrings of a string.

Problem 1.

Write two functions, called countSubStringMatch and countSubStringMatchRecursive that take two arguments, a key string and a target string. These functions iteratively and recursively count the number of instances of the key in the target string. You should complete definitions for

def countSubStringMatch(target,key):

and

def countSubStringMatchRecursive (target, key):

Place your answer in a file named `ps3a.py`

For the remainder of this problem set, we are going to explore other substring matching ideas. These problems can be solved with either an iterative function or a recursive one. You are welcome to use either approach, though you may find iterative approaches more intuitive in these cases of matching linear structures.

The next thing we want to do is write a function that generalizes `find` so that it returns a tuple of all starting points of a match of a key string in a target string, not just the first instance.

Problem 2.

Write the function subStringMatchExact. This function takes two arguments: a target string, and a key string. It should return a tuple of the starting points of matches of the key string in the target string, when indexing starts at 0. Complete the definition for

def subStringMatchExact(target,key):

For example,

subStringMatchExact("atgacatgcacaagtatgcat","atgc")

would return the tuple (5, 15). The file ps3_template.py includes some test strings that you can use to test your function. In particular, we provide two target strings:

target1 = 'atgacatgcacaagtatgcat'

target2 = 'atgaatgcatggatgtaaatgcag'

and four key strings:

key10 = 'a'

key11 = 'atg'

key12 = 'atgc'

key13 = 'atgca'

Test your function on each combination of key and target string, as well as other examples that you

create. Place your answer in a file named ps3b.py

The function you wrote in Problem 2 will find exact matches of a key string in a target string. It is often also useful to find near matches, for example, matches of a key string in a target string, where one of the elements of the key string is replaced by a different element. For example, if we want to match ATGC against ATGACATGCACAAGTATGCAT, we know there is an exact match starting at 5 and a second one starting at 15. However, there is another match starting at 0, in which the element A is substituted for C in the key, that is we match ATGC against the target. Similarly, the key ATTA matches this target starting at 0, if we allow a substitution of G for the second T in the key string.

We can build on your function from Problem 2 to solve this problem. In particular, consider the following steps. First, break the key string into two parts (where one of the parts could be an empty string). Let's call them key1 and key2. For each part, use your function from Problem 2 to find the starting points of possible matches, that is, invoke

starts1 = subStringMatchExact(target,key1)

and

starts2 = subStringMatchExact(target,key2)

The result of these two invocations should be two tuples, each indicating the starting points of matches of the two parts (key1 and key2) of the key string in the target. For example, if we consider the key ATGC, we could consider matching A and GC against a target, like ATGACATGCA (in which case we would get as locations of matches for A the tuple (0, 3, 5, 9) and as locations of matches for GC the tuple (7,). Of course, we would want to search over all possible choices of substrings with a missing element: the empty string and TGC; A and GC; AT and C; and ATG and the empty string. Note that we can use your solution for Problem 2 to find these values.

Once we have the locations of starting points for matches of the two substrings, we need to decide which combinations of a match from the first substring and a match of the second substring are correct. There is an easy test for this. Suppose that the index for the starting point of the match of the first substring is n (which would be an element of starts1), and that the length of the first substring is m. Then if k is an element of starts2, denoting the index of the starting point of a match of the second substring, there is a valid match with one substitution starting at n, if n+m+1 = k, since this means that the second substring match starts one element beyond the end of the first substring.

Problem 3.

Write a function, called constrainedMatchPair which takes three arguments: a tuple representing starting points for the first substring, a tuple representing starting points for the second substring, and the length of the first substring. The function should return a tuple of all members (call it n) of the first tuple for which there is an element in the second tuple (call it k) such that n+m+1 = k, where m is the length of the first substring. Complete the definition

def constrainedMatchPair(firstMatch,secondMatch,length):

To test this function, we have provided a function called subStringMatchOneSub, which takes two arguments: a target string and a key string. This function will return a tuple of all starting points of matches of the key to the target, such that at most one element of the key is incorrectly matched to the target. This function is provided for you in the file ps3_template.py and invokes the function you are to write.

Save your answers in a file named `ps3c.py`.

You will have noticed in your tests for Problem 3, that this approach will find matches with one substitution but will also find matches with no substitutions, that is, exact matches of the key to the target. Suppose we want to find only those matches with exactly one substitution. One easy way to do this is to use the functions from both Problem 2 and Problem 3. These functions will give you a tuple representing starting points for exact matches, and matches with up to one substitution, respectively. If we keep only those elements of the second tuple that don't occur in the first tuple, we will have the matches with exactly one substitution.

Problem 4.

Write a function, called subStringMatchExactlyOneSub which takes two arguments: a target string and a key string. This function should return a tuple of all starting points of matches of the key to the target, such that at exactly one element of the key is incorrectly matched to the target. Complete the definition

```
def subStringMatchExactlyOneSub(target,key):
```

Save your answers in a file named `ps3d.py`.

Hand-In Function

1. Save

Save your code in the specific file name indicated for each problem. *Do not ignore this step or save your file(s) with different names.*

2. Time and Collaboration Info

At the start of each file, in a comment, write down the number of hours (roughly) you spent on the problems in that part, and the names of the people you collaborated with. For example:

```
# Problem Set 3 (Part I)
# Name: Jane Lee
# Collaborators: John Doe
# Time: 1:30
#
... your code goes here ...
```