Title: Searching a nucleotide sequence for binding sites

Introduction:

One of the primary goals of Bioinformatics is to find patterns in DNA sequences. These include start and stop codons for ORF prediction in prokaryotes, exon\intron splice junctions for protein coding region identification in DNA of Eukaryotes, and the identification of transcription factor binding sites, to name a few!

A transcription factor binding site is where proteins, known as transcription factors, bind to the DNA molecule. These sites are usually found in the vicinity of promoters of genes. The proteins that bind to these sites influence gene regulation; they can either act as inducers causing the gene to be transcribed, or act as suppressors causing the gene to be deactivated. Hence finding these sites is critical to understanding gene regulation and gene networks (how genes interact with each other).

These sites are only 5-10 bp long, and could be several hundred to several thousand bp away from the transcription start site of a gene. It's amazing how something so small can have such a huge impact so far away from a gene. This is why transcription factors and their binding sites have been studied so intensely by Biologists.

As a Bioinformatician working in a molecular biology lab you will most likely be given a sequence (or an accession number) and a list of transcription factors (names or actual sites) by your boss and asked to see if they exist in the sequence or not! There are searchable databases that you can turn to for this, some you've already learned about in your intro to Bioinformatics class, such as the Transfac database found at http://www.gene-regulation.com/pub/databases.html. BUT let's assume you want to impress your boss by writing your own script to do this! Writing your own programs allows you to customize it the way you want, plus it frees you from the shackles of web sites and their restrictions. It also allows you to automate something to run in batch mode. For instance Transfac might be ideal if you were given one sequence, and one transcription site for search against, but what if you were given thousands of sequences and hundreds of sites? Here is where the power of scripting in a language like Perl or Phyton comes into play.

In this demonstration we will search a DNA sequence for a binding site. We will keep this simple by assuming we have ONE sequence and are looking for exact matches of a binding site given to us by the user. Once we've learned how to do this, we can then easily modify it to work on 1000s of sequences and\or 100s of binding sites.

Learning Demonstration:

Open your favorite IDE or editor, and create a new .py file for your program (call it bindingSiteSearch.py or something like that).

As you recall in our discussions in class, all programs are broken down into 3 main sections, input, algorithm and output. Before we start to write a program we need to ALWAYS think about these 3 things AHEAD of time! That is what pseudocode allows you do to. So on a piece of paper, jot down what you think are the inputs, how you think the program should work to find the binding sites (i.e. the algorithm) and what the outputs should be (i.e. what is going to be displayed back to the user?).

1- **Gathering inputs:** We know we need at least one binding site to search for, so that is an input. We also know that we need at least one sequence to search against, so that's another input.

   a. Let's ask the user what binding site he\she has in mind. In Python we use the input() function to do this. This function collects what a user types on the keyboard and stores it in a variable of your choosing:

      ```
      bindingSite = input("Please enter a binding site:")
      ```

      As you recall we always have to display a message to the user so he\she knows what needs to be typed on the keyboard. In this case the user will type in a binding site, such as TGACCT and press enter. The string will be stored in the variable `bindingSite`.

      Note: Always use descriptive names for variables! I could have called my variable x, but a future me or someone else reading this code many years from now will not have the faintest idea what I was trying to do!

   Now let's get the sequence, from a file, and store it in memory as well. Opening files are easy in Python. We just have to call the open function and Python handles the rest! First though we need to ask the user what the file name is:

   ```
   DNAfileName = input("Please type in the file name containing a
   nucleotide sequence:")
   ```

   Once we have the file name, we can use the open function. We'll have to test to make sure Python can open the file or not (i.e. make sure the file actually exists and that the user typed the name correctly). There are many ways to do this, one way is to use the try…except block:

   ```
   try:

       f = open(DNAfileName)

       # Do something with the file

   except IOError:

       print("File not accessible")

   finally:
   ```

```
 f.close()
```

f here is known as the file handler, it will basically represent the file in our program. Programming languages do not allow us to interact with files directly (security risk, plus they are afraid we might mess things up! ☺) and so always allow you access through a representative it trusts!

Notice that the open function call is inside a try statement, this is a conditional statement like the if statement. So in the code above Python will check to see if the file can be opened, and if it cannot, then it will print out an error message and exit the program.

b. Now that we have opened the file, it's time to read in the data. As you recall we can read in an entire file and store its contents into memory using one line of code in Python:

```
DNA = f.read()
```

Here we used the read() function to read in the entire file and stored its contents in the variable .

Assigning a file to an variable like we did here is good if you have only one sequence, but will slow your program down or crash it completely if you have a very large sequence or many of them stored in the file (remember the entire content of the file gets stored in memory!). A good programmer needs to think ahead, he\she needs to assume all possible scenarios, what if the user gives me a huge sequence(s) to work with for instance? A better way would be to read in the file line by line, thereby only storing a line at a time in memory instead of the entire file! This makes your program run faster and will prevent it from crashing if the user gives you a huge file.

So how do we go about doing that? Recall the for loop. We can use this structure to read a file line by line till there are no more lines left in the file:

```
for line in f:
        #do something with the lines
```

Here each line in the file gets assigned to line, one after the other, till there are no more lines left (i.e. till Python reaches the end of the file).

2- **Algorithm:** Now that we have our inputs, it's time to start with the brains of the program.

a. We already have the binding site stored in bindingSite and we have the lines of the file continuously being read and stored in line.
b. One might think of using an if…else statement coupled with a regular expression to test to see if the line being read contains the binding site or not:

```
      if re.search(bindingSite, line):
            print("The sequence has the binding site")
        else:
            print("Binding site NOT found")
```

Code here assumes you are using the re module and have imported it already. The regular expression re.search(bindingSite, line) is inside the if statement, if line contains bindingSite, then the condition is true, and the if block will execute, but if the condition is false the else statement is executed. Review conditional statements and regular expressions under content for more info.

c. What happens if we place that if…else statement above inside the loop? Will it solve the problem? Try it…. you'll see that it will print out "The sequence has the binding site" or "Binding site NOT found" many times, as many times as there are lines in the file in fact! But we don't want that, we just want to print one message to the user, whether or not the sequence has the binding site.

   We could employ a trick programmers use, knows as flag setting. We basically lift a flag, by assigning a Boolean variable the value true, when we found something. The way it works is as follows:

```
found = 0 #false
    for line in f:
        if re.search(bindingSite, line):
            found = 1 #true
```

   Notice I first assigned the value 0 to found before the loop. Inside the loop I test to see if line contains bindingSite, if it does I know the sequence contains at least one binding site. I can then set found to 1.

   After the loop I can test to see if found was changed, if it did then we must have hit the binding site when the loop ran, if it remained 0 however, it means we did not find the binding site in the sequence.

Output: The output for this program is simple, we will just print out a message to the user whether or not we found his\her binding site:

```
if (found): #alternatively I could also say if (found == 1)
   print("The sequence has the binding site")
else:
    print("Binding site NOT found")
```

3- **Testing and modification:** As we discussed in class, all software, including scripts go through the development life cycle (Requirement analysis -> Design -> Implementation -> Testing). And it's called a cycle because programs are usually modified and improved upon after users have had a chance to use it.

   a. Test your code, is it working properly? Ask yourself will my code work for ANY sequence? And ANY binding site?

b. Modify the code so it can look for binding site in more than one sequence.
c. Our code is looking for binding sites within each line of sequence, what if the binding site is split onto two or more lines? Will this code work then?
d. If not, how would you modify the code to fix it?
e. What if your users or yourself wanted to know the exact positions of the binding sites? How would you modify the code to get that?

Hints: Look up how to get multiple matches using re.finditer() and how to extract the matching positions using start() and end().

Deliverables:

1. Submit the .py file with the working code above; also submit your test sequence file.
2. Make sure to add comments to your code to explain briefly what each line or block code does.