The Technion – Faculty of Biology
Computational Approaches for Life Scientists - 134141
Spring 2015

Home assignment 2 – Due May 8th, 23:55

Submission instructions:

- Submission is in pairs (or singles after approval from the lecturer)
  Honor code:
  Each student is expected to be fully involved in solving all the questions. Furthermore, although you may discuss the solutions with other pairs / students, each pair must write the whole solution separately. In particular, it is not allowed to spread any piece of code / solution.

- The answers will be submitted in 2 files:

  1. A doc/docx file with the "dry" part: answers to all the questions and the required explanations.
  2. A py file with the "wet" part: any code you have written to answer the questions. The code in this file must support your answers and conclusions, such that when run yields the results provided in the dry part. The grade for this part will be given for a correct and reasonable code, which avoids unnecessary complications and bad styling.

- The assignment should be submitted via moodle. The 2 files should be uploaded to moodle by the deadline. You may submit late though, up to a maximum of 5 days for the whole semester. For that matter, submission after 23:55 is a day late.

- Note: while each question normally has a single correct answer, the code you'll write to answer them may be written in various ways. There is no single correct code.
  However, to get full score you must make an effort to write your code in a reasonable manner, in terms such as low complexity, meaningful names, etc.

  You are requested to comment (#) your code, and you may even leave old code that you have written and no longer use (commented out!). This way I will be able to track your thinking process, at least partially.

- Bonuses: up to 5 points may be given as a bonus to your grade, for interesting, **non-trivial** comments, mostly biological ones, that you find relevant. For example:
  - A relevant application for the question's topic that is beyond a trivial one
  - New information that sheds interesting light on the question, or puts it in an interesting biological context

  If you provide such comments, make sure you state your references.

ATG

Question 1 – common substring

a. In class, we saw several solutions to the *common substring problem*. We said that the naïve solution, implemented in the function common_substring_naive, runs in $O(k \cdot n_1 \cdot n_2)$ time. Explain why the time complexity of the improved, hash based solution, implemented in common_substring_hash, runs in $O(k\ (n_1+n_2))$ on average.

b. We implemented the function longest_common_substring_bsearch, which finds the longest common substring of two strings, using the binary search approach. However, our implementation has a small bug. For example:

```
>>> longest_common_ss_bsearch("A"*7, "A"*7)
1 found
2 found
4 found
8 was not found, but 4 was found
6 found
(6, 'AAAAAA')
```

Find the bug. In your answer, just describe what fix is needed and why.

Question 2 – common substring for 3 strings

In this question we will deal with the common substring problem, but for three strings instead of two (as we did in class). The input for this problem is 3 strings, s_1, s_2, s_3, and a positive integer k. The output is all the substrings of length k that appear in all three strings.

a. Look at the following naïve solution:

```
def common_3ss_naive(s1, s2, s3, k):
    """ find a common length k substring of s1 and s2 and s3 """

    s = set() #not really using the hash table
    for i in range(len(s1)-k+1):
         for j in range(len(s2)-k+1):
              if s1[i:i+k] == s2[j:j+k]:
                   for m in range(len(s3)-k+1):
                       if s1[i:i+k] == s3[m:m+k]:
                            s.add([s3[i:i+k]])

    return s
```

Note that the output is provided in a set type element. However, here the set is not used for efficient search (with the hash table mechanism), it is used merely for storing element. We could have used lists or any other collection instead.

We want to use this function to find all the common substrings of length k=50 of the three genomes of Mycobacterium tuberculosis, Salmonella enterica, and Mycobacterium leprae. Approximately how much time would it take? Estimate the running as we did in class.

b. Modify this solution, by using Python's 'in' operator. Call the new function `common_3ss_naive2`, and make it contain only a single for loop. By how much is this solution faster? Estimate the running time again.

c. Improve the solution by using Python's sets (not merely for storing, but this time in the context of efficient search, exploiting the hash table mechanism underneath Python's sets). Try optimizing your solution as much as possible, in terms of running time. Call your function `common_3ss_hash`.

d. How long did it take this time to find all the common substrings of length k=50 for the three genomes? How many such substrings are there?

e. Repeat the last section for k=1. What is the result? Is it surprising?

Now repeat it for each k between 2 and 20, and compare the result against the number of possible genomic strings of length k. Show the results in a table, and explain them.

f. Now we want to find the *longest* common substring for 3 given strings. We could check increasing values of k (k=1,2,3,…), but we want to make it more efficient, and use a binary search approach, as we did in class. Write a function `longest_common_3ss_bsearch` that works in this way. Like the function from class, this function will print the values of k along the way, to indicate the execution progress.

g. Of what length is the longest common substring of the three genomes of Mycobacterium tuberculosis, Salmonella enterica, and Mycobacterium leprae? In your answer, specify all the different values of k that your function checked in the computation by their order.

h. Suppose we wanted only to know *whether or not* 3 given sequences have a common substring of a given length *k*. In other words, instead of finding the actual common substrings, the output will merely be True if they do and False otherwise.

   Amir suggests the following idea: we already have a solution for 2 strings, which we saw in class (the function common_ss_hash). We will check if each pair of strings (s_1 and s_2, s_1 and s_3, s_2 and s_3) has a common substring of length *k*. If all 3 pair do, then we can return True, otherwise False.
   Is Amir's idea correct? Explain why it is, or give a counter example to show it is incorrect.

Question 3 – regular expressions

Suppose we have two restriction enzymes called AbcI and AbcII. The recognition site of AbcI is ANT-AATTTTT, and of AbcII is GCRW-TGGGGG (N means any base, R means A or G, W means A or T, and "-" indicates the position of the cut site).

a. Predict the fragment lengths that we will get, if we digest the genome of *Mycobacterium leprae* with AbcI. For example, if the genome was "CCCAATAATTTTTGGGATTAATTTTTGG", then AbcI would cut between the nucleotides underlined. We would get three fragments: "CCCAAT", "AATTTTTGGGATT", and "AATTTTTGG", whose lengths are 6, 13 and 9.

b. Repeat the previous section when the two enzymes digest the DNA sequence together.

In your answers, also provide a short explanation on how you solved the problem using Python.

TAG