M345SC 2019 Homework 1

Due date/time: 6/2/2019, 23:59 GMT

Clarifications (last edited 2/2/19)

2/2/19:

- **Guidance for Part 1**: There is no one algorithm which will work best for all possible inputs. Furthermore you are not expected to develop several different algorithms tuned for different kinds of input (e.g. a method specifically designed for k=2 or N-k=3). You should be able to come up with a few feasible approaches based on ideas from class and then you should choose one which works well for a broad range of inputs (e.g. for a few orders-of-magnitude of k). Your analysis and discussion should reflect why you chose your method, and in what sense it "works well". Comprehensive exact operation counts are not required, but you should go beyond simply estimating the big-O asymptotic behavior, and some discussion of the leading-order and lower-order terms should be included. There will not be any "speed tests" where your code is assigned marks based solely on timing results. Marking will consider the efficiency with which your method is implemented and the level-of-understanding reflected in your algorithm design and analysis. The test sequence from lab 3 is an example of a "very long" sequence.
- Your search codes (all code in or called from within *ksearch* and *nsearch*) should not utilize any external modules such as numpy, matplotlib, scipy, etc… They may be used in *nsearch_time* or when generating input to test your code. Please do not submit codes that use np.array, np.sort, np.where in *ksearch* or *nsearch*.
- Part 2: What is *ascending order*? The first P elements of a list are in ascending order if L[i]>=L[i-1] for i=1,2,..,P-1. Note that there may be repeated elements within both the sorted and unsorted portions of the input lists.

1/2/19:

- Part 1.2: Which point-x mutations are needed? For *each* "frequently occurring k-mer", you should find all point-x mutations in the sequence, *S*.
- Part 2.1: The numerical values in the unsorted portion of a partially sorted list are not constrained in any way by the numbers in the sorted portion.
- **Note above replaces this one** You may use numpy, matplotlib, time, timeit when testing and developing your code. They should not be used within *ksearch*, *nsearch* or within functions called by *ksearch* or *nsearch*.

28/1/19: You may use the *time* and *timeit* modules in *nsearch_time* along with *numpy* and *matplotlib*.

This project consists of two parts – in the first, you will be working with DNA sequences, and in the second you will be working with sequences of numbers.

Getting Started: Template files have been added to the course repo in the hw1/ directory. You can also download the files directly from here: | p1_template.py | p2_template.py

Place these files in a folder, and create copies of the template files named *p1_dev.py* and *p2_dev.py* in this new directory. Ultimately, the final files that you submit should be titled *p1.py* and *p2.py*. Browse through the *dev* file; there are a few function headers, and you will have to add code for each of these functions as described below. First, however, you should *add your name and 8-digit college id to the docstring at the top of each file.*

Part 1. (8 pts)

Here, you will work with strings corresponding to DNA sequences. Consider a DNA sequence, *S*, provided as input. The goal is to find: 1) frequently-occurring *k-mers* within *S*, 2) their positions in the input sequence, and 3) the number of point-x mutations of these k-mers that are in *S*. Note that a *k-mer* is simply a length-k string of consecutive nucleotides in *S*, and a point-x mutation of a k-mer, *s*, is another k-mer which is identical to *s* aside from the nucleotide in its *xth* position (with x=0 corresponding to the first nucleotide in a k-mer). So, the sequence "ACGTCG" contains 4 *3-mers* (ACG, CGT, GTC, TCG) and TCG is a point-0 mutation of ACG (and vice-versa).

1) Add code to the function, *ksearch* in *p1_dev.py* so that it finds all *k-mers* that occur at least *f* times within the the input sequence, *S* (and with *f* and *k* also provided as input). The function should return the locations of these frequent k-mers as well as the *k-mer* strings themselves.

2) Complete *ksearch* by computing the total number of k-mers which are point-x mutations of each "frequently-occurring" k-mer found in 1) with *x* provided as an input parameter.

Ultimately, the function should return three lists: L1 containing the strings corresponding to the frequently-occurring k-mers; L2 containing the locations of the frequent k-mers, specifically, the *ith* element in L2 should be a list of integers corresponding to the indices in *S* where the *ith* k-mer in L1 is found; L3 containing the number of point-x mutations for each k-mer in L1. If no frequent k-mers are found, simply return three empty lists.

3) Carefully analyze the running time of your code in *ksearch*. Your analysis should include: a) a concise description of your algorithm and b) a clear and concise discussion of the running time (including the leading-order running time) and how it depends on the input.

We are generally interested in cases where the length of *S* is very large though you may assume that a string containing all k-mers for *S* will fit in your computer's main memory.

Part 2. (12 pts)

1) Complete the function *nsearch* in *p2_dev.py* so that it efficiently searches for *all* occurrences of a target within *N* partially sorted lists of numbers of length *M*. Specifically, the first *P* numbers in each list are sorted in ascending order. If the target is found in one or more lists, the function should determine which list(s) the target is in and the location within each list where the target is located. If the target is not found, simply return an empty list. See the function documentation for more details on the structure of the function input and output. You may assume that *N*, *P*, and *M-P* are all large.

2) Carefully analyze the running time of *nsearch*. Your analysis should include: a) a concise description of your algorithm b) a clear and concise discussion of the running time and how it depends on the input, c) a concise explanation of if/why your algorithm can be considered to be efficient, d) one or more figures clearly illustrating key trends in the actual running time of the function, and e) a description of and explanation of the trends shown in the figure(s). The code that generates your figures should be placed in *nsearch_time*. Place your discussion in the docstring of *nsearch_time* as indicated in the template file. You should save your figure(s) as *png* files following the naming convention *fig1.png*, *fig2.png*, … and submit the figures along with your codes.

Notes:

1. Marking will consider both the correctness and efficiency of your code as well as the soundness of your analysis.

2. All figures created by your code should be well-made and properly labeled. The title of each figure should include your name and the name of the function which created it.

3. In order to assign partial credit, markers must be able to understand how your code is organized and what you are trying to do.

4. You are allowed to discuss general aspects of Python and the problem statement with others (e.g. What is a "k-mer"?), however you should not discuss your particular implementations with other students or show your code to other students.

5. Submitting this assignment commits you to the class.

6. You may use *numpy* and *matplotlib* in *nsearch_time*. Otherwise you should not import/use any python modules without consulting with the instructor.

Submitting the assignment

To submit the assignment for assessment, go to the course blackboard page, and click on the “Assignments” link on the left-hand side of the page, and then click on Homework 1. Click on “Write Submission” and add the statement: “This is all my own unaided work unless stated otherwise.”

Click on “Browse My Computer” and upload your final files, *p1.py*, *p2.py*, *fig1.png*, *fig2.png*, etc… . Finally, click “Submit”.