COMPSCI 260, Introduction to Computational Genomics **16 September 2016**

Problem Set 2

Due: 5pm on 30 September 2016

In this problem set you will be designing and analyzing algorithms to solve various tasks.

Your algorithms should be presented clearly and concisely in English and/or pseudo-code. If you wish, you can illustrate your algorithm using an accompanying example, but please note that an example does not suffice as a clear and concise description of an algorithm.

For every algorithm you present, you should:

- argue briefly that it is correct and
- analyze its (asymptotic) running time.

Problem 1: The limits of force (15 points)

One of the first steps in analyzing next-generation sequencing data is often mapping the reads generated during the sequencing process against a reference genome. Suppose you are given a reference genome of length n and a single read of length m. Consider the problem of determining whether or not the read can be found somewhere in the genome.

a) Assuming a uniform nucleotide distribution, what is the expected number of occurrences for a read of length m in a genome of size n?

b) Describe a brute force algorithm which can accomplish the task of mapping a read to a reference genome and analyze its worst case running time.

c) Now imagine having to map k reads to a reference genome. Especially if n is large (as is typical for many genomes), the naive brute force approach will become very slow as k grows. Suppose, however, that you have access to a data structure which allows you to query the genome for a specific sequence in $O(m \log n)$ time. If the data structure took $O(n \log n)$ time to build, then for what number of reads will it have been worth it for you to adopt the approach which pre-processes the genome? What, ultimately, are the trade-offs between the brute force approach and the pre-processing approach?

Problem 2: Sorry, this array is out of order (35 points)

In an unsorted array $A = [a_1, \ldots, a_n]$ of distinct integers, we say that a pair of elements a_i and a_j represent an *in-order pair* if $i > j$ and $a_i > a_j$. For example, the following array of integers

$$[15, 8, 20, 4, 7, 10, 1]$$

contains 6 in-order pairs; they are $(15, 20), (8, 20), (8, 10), (4, 7), (4, 10)$ and $(7, 10)$.

Your task is to efficiently determine the number of in-order pairs in a given input array.

a) A brute force way to solve this problem would be to take the first number and compare it with the remaining $n-1$ numbers, then take the second number and compare it with the remaining $n-2$ numbers, and so on. Analyze the running time of this method.

b) Describe a divide-and-conquer algorithm that solves this problem in $O(n \log n)$ time. Show your worst case running time analysis.

c) Implement both of the above algorithms in the file `in_order_pairs.py`. For the divide-and-conquer algorithm, *evaluate* its running time on sets of random inputs of length $n \in \{10^2, 10^3, 10^4, 10^5, 10^6, 10^7\}$ and *estimate* its running time for inputs of length $n = 10^8$. For the brute force approach, *evaluate* its running time on sets of random inputs of length $n \in \{10^2, 10^3, 10^4, 10^5\}$ and *estimate* its running time for inputs of length $n \in \{10^6, 10^7, 10^8\}$. To do this, you should make use of the Python `timeit` module. We have already placed a line in `in_order_pairs.py` that loads this module: `import timeit`. To learn how to use this for timing your code, you can visit the Python documentation for the module here: http://docs.python.org/2/library/timeit.html. The very bottom of the webpage shows how you can use the module to time a function you have written. Here is an simple example: suppose you wrote a function called `brute_force(random_list)` that takes a Python list called `random_list` as the only argument. To get its running time, use the following code:

```
timeit.timeit("brute_force(randomlist)", setup="from __main__ import *", number=1)
```

How do their running times seem to scale as the problem size grows? How do their running times compare to each other? If there are differences between what you observe and what you might expect from the asymptotic analysis of the previous subproblems (a) and (b), why might those differences arise?

Problem 3: Pebble beach (35 points)

Imagine that you are given a grid with n rows and 4 columns. You are also given a set of $2n$ pebbles. Each pebble may be placed on at most one square of the grid, and at most one pebble may be placed on each square of the grid. Let us define a *valid placement* to be a placement of some or all of the pebbles on the board such that no two pebbles lie on horizontally or vertically adjacent squares (diagonal adjacencies are permitted). On each square of the grid is written a positive integer. Let us define the *value* of a placement to be the sum of the integers in all the squares on which a pebble has been placed.

a) Determine the number of distinct valid pebble placements that can occur in a single row. Henceforth, we shall call such a valid single-row placement a *pattern*. Describe/enumerate all the patterns.

b) We say that two patterns are *compatible* if they can be placed on adjacent rows to form a valid placement. Let us consider the big problem of finding a valid placement of maximum value. We can break down this big problem into subproblems of size $k \in \{0, \ldots, n\}$, where the subproblem of size k considers only the first k rows of the grid. We shall also assign each subproblem a *pattern type*, which is the pattern occurring in the last (k^{th}) row. So the subproblems we seek to solve can now be "named" based on their size and their pattern type. Using the notions of pattern compatibility and pattern type, design an $O(n)$ dynamic programming algorithm for computing a valid placement of maximum value on the original grid.

c) Write a Python program to implement your algorithm. Your program should take an $n \times 4$ matrix as input and then output the maximum value of a valid placement. A sample grid with random integers between 1 and 100 is provided to you in the file `grid.txt`. What is the maximum value of a valid placement for this sample grid?

Hint: Before running your program with the grid from `grid.txt` as input, you may want to test the program on smaller examples (grids with 1, 2, or 3 rows) for which you can compute the best solution by hand.

Problem 4: Greed, for lack of a better word, is (sometimes) good (15 points)

Eager to gain some research experience (and to earn a little work-study money), you begin serving as a technician in a lab on campus. Assigned to a project left behind by a recently graduated PhD student in the lab, one of your first tasks is to come up with a strategy for determining the sequence of a specific gene. However, with the technology available to you, you are only able to reliably sequence reads up to n bases in length. Since the length of the gene itself is much greater than n, it will be necessary for you to fragment the gene in order to generate reads which can be sequenced in their entirety. Good thing the lab manager just replenished the restriction enzyme stocks! While the sequence of the gene is obviously a mystery, it turns out that the graduate student whose work you've inherited had managed to figure out the locations of the restriction sites in the gene relative to one another (i.e., you know the distances between sites). Since you have plenty else keeping you busy both in and out of the lab, and you don't feel like performing any more restiction enzyme digests than necessary, you make it your goal to minimize the number of restriction sites used to generate manageable reads which span the length of the gene. Give an efficient method by which you can determine which sites to use. Remember to prove that your strategy yields an optimal solution, and give its running time.

Problem 5: Lazy, out of shape professor (extra challenge)

A 160-story building under construction contains a set of $n > 2$ indistinguishable wires running in a conduit from the basement up to the roof. Professor Alec Trician has been hired to label the extremities of the wires at both ends in such a way that the end labeled i in the basement is the same wire as the end labeled i on the roof. The conduit is not accessible, and thus, the professor has access only to the wire ends. He has electrical tape that he can use to connect wire ends together (two or more ends can all be connected together by tape, but the ends being connected must be either all in the basement or all on the roof; he can't connect a wire end in the basement to a wire end on the roof). The professor also has a pocket-sized continuity tester with two terminals that he can apply to two wire ends to determine whether they are connected at the other end of the conduit.[1]

Unfortunately, the elevator of the building is not yet functional, so the professor must climb up to the roof and get down to the basement by stairs, a chore that the professor finds far more onerous than doing electrical work. Describe an efficient algorithm by which the professor can label the wire ends as desired, where the efficiency of the algorithm is measured in terms of the number of times the professor must take the stairs.

Note: The more efficient your algorithm, the more extra credit you can receive. Also note, however, that you should only spend time on this problem if you've finished the others: this problem is mainly for the challenge of it, and is not especially relevant to the course apart from demonstrating how creative thinking can lead to more efficient solutions to problems.

[1] For concreteness, you can just imagine this to be a device with a battery and buzzer, where the buzzer sounds when a circuit is closed. In particular, if two wires are connected by electrical tape at the far end, connecting the terminals of the tester to their corresponding near ends will complete an electrical circuit and thus the tester's buzzer will go off. If the far ends of the wires are not connected the buzzer will remain silent.