COMPSCI 260, Introduction to Computational Genomics **13 September 2019**

Problem Set 2

Due: 5pm on 27 September 2019

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

In addition to providing a Python implementation of your algorithms, also include both pseudo-code and a plain text description in your README.txt. 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 any algorithm you develop, you are additionally required to analyze its (asymptotic) running time.

Problem 1: The limits of force (15 points)

List of files to submit

1. `README.problem1.[txt/pdf]`

Plan

Often, one of the first steps in analyzing next-generation sequencing data is 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 as an exact substring somewhere within the genome.

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

Develop

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. Note you do not need to write code to implement this algorithm.

Reflect

c) Now imagine having to map k distinct reads to a reference genome. Especially if n is large (as is typical for many genomes), the naïve 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 as to whether it contains a specific read sequence in $\Theta(m \log n)$ time. If the data structure took $\Theta(n \log n)$ 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: Should we worry about an alien invasion? (35 points)

List of files to submit

1. `README.problem2.[txt/pdf]`
2. `collective_similarity.py`

Plan

While studying an organic sample taken from an asteroid, you isolate a nucleotide sequence X from what seems to be a mysterious alien organism. Bits of it seem to be similar to human DNA while other bits are quite dissimilar. To better understand what kind of threat the human race might be facing, scientists around the planet are rushing to your side to study this sequence.

One scientist has now managed to find all the genes in X and, for each one, has assessed how similar it is to its closest human counterpart, reporting a number where a more positive value indicates greater similarity and a more negative value indicates greater dissimilarity. She has compiled all these values into one long list, where the first element represents the human similarity for the first gene along the alien genome, the second element represents the human similarity for the second gene along the alien genome, and so on. A very short sample list might look like this:

$$[2, -3, -4, 4, 8, -2, -1, 1, 10, -5]$$

though the actual list for all the genes in the alien genome is quite a bit longer. You are now tasked with the job of identifying the region (contiguous segment) of the alien genome that possesses the greatest collective similarity to the human genome. More precisely, we define the "collective similarity" of a region of the alien genome to be the sum of all the similarity values of the genes contained within it, and your job is to maximize this score. In the sample list above, the region of maximal collective similarity is the sublist $[4, 8, -2, -1, 1, 10]$, whose collective similarity is 20. Note that you should return an empty region (i.e. collective similarity $= 0$) if that ends up being optimal.

To solve this task, you turn to algorithms, but since you're a strong student in COMPSCI 260, you know that there are many ways to design a correct algorithm. In this problem, you will design different algorithms for this task and see how they compare. For each algorithm we ask you to implement below, please write your implementation as a distinct function within the file `collective_similarity.py`. Each function should take in as input a list of gene similarity values as integers and return a single integer of the maximal collective similarity. This structure will be useful later when you compare how long it takes each implementation to run on larger and larger inputs.

As one test of the correctness of each of your implementations, you can provide the list above as an input, but please recognize that passing one test (or even a large suite of tests) does not prove correctness; it is true that failing a test proves incorrectness, but the converse doesn't hold.

Please note that for this problem, we only ask you to output the largest collective similarity score that is achievable. You do not need to output the region in the alien genome that achieves that score—though naturally you should try that on some or all of the implementations if you are looking for an extra challenge :-).

Develop

a) A brute force way to solve this problem would be to exhaustively consider every possible sublist of the input list, and compute the collective similarity for each sublist. However, even when you are constrained to use a brute force solution like this one, you can still design your algorithm to run faster. Specifically, how could you organize the order in which you exhaustively consider your sublists so as to speed up your computation? Analyze the complexity of your algorithm, and then implement it in Python as a function called `brute_force` within `collective_similarity.py`.

b) Describe a recursive divide-and-conquer algorithm that solves this problem in $\Theta(n \log n)$ time. Show your worst-case running time analysis. Then, implement your algorithm as a different function called `divide_and_conquer` within `collective_similarity.py`.

c) It turns out that there is a way to solve this problem in linear time. We can just scan the list once from left to right, keeping track of two things: the largest collective similarity of any sublist we've seen so far (let's call this `MAX_SO_FAR`) and also the largest collective similarity of any sublist that ends at the current position (let's call this `MAX_INCLUDING_HERE`). As you traverse the list one element at at time, you will need to determine how these values should be updated so that they maintain their meanings.

Explain how to update these values so that they maintain their meanings throughout the scan, and then explain how this method will guarantee you find the optimal (highest-scoring) region. *Hint:* Start by thinking about a list that contains only one positive score: What can you say about the optimal solution for such a list, and how will your method work on such a list? Now think about what happens if there is a slightly larger sublist whose score is positive. Now think about what happens if there are multiple sublists (of varying lengths) whose scores are positive.

Finally, once you understand this algorithm well, implement it in Python as a different function called `linear` in `collective_similarity.py`.

Check

d) Now, *empirically evaluate* the running time of each of your algorithm implementations on randomly generated lists of different pre-specified lengths, as described below. Each list will contain integers drawn randomly from the set $\{-10, \ldots, 10\}$, but your algorithms should work for lists of arbitrary values.

A code snippet using the Python `timeit` package is provided in `collective_similarity.py` for you to use in evaluating the running time of your code.

- For the brute force algorithm, *evaluate* its running time on sets of random inputs of length $n \in \{10^2, 10^3, 10^4, 10^5\}$.
- For the divide-and-conquer algorithm, *evaluate* its running time on sets of random inputs of length $n \in \{10^2, 10^3, 10^4, 10^5, 10^6, 10^7\}$.
- For the linear algorithm, *evaluate* its running time on sets of random inputs of length $n \in \{10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8\}$.

Present your results in table format.

Reflect

e) Now, *estimate* the running times as described below.

- For the brute force algorithm, *estimate* its running time for inputs of length $n \in \{10^6, 10^7, 10^8, 10^9\}$.
- For the divide-and-conquer algorithm, *estimate* its running time for inputs of length $n \in \{10^8, 10^9\}$.
- For the linear algorithm, *estimate* its running time for inputs of length $n = 10^9$.

Present your results in table format.

f) What are some observations you can make about the relationships between algorithm, list length, and running time?

Problem 3: Pebble beach (35 points)

1. `README.problem3.[txt/pdf]`
2. `pebbles.py`

Plan

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 all integers written on the squares with pebbles.

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.

Develop

c) Write a Python program to implement your algorithm. Your program should take an $n \times 4$ grid as input and then output the maximum value of a valid placement.

Check

d) 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.

Reflect

e) Can you think of an example grid where the optimal solution is obtained by not placing anything on at least one of the rows? Give a short grid example and state what condition can lead to an optimal solution that has at least one empty row (you do not need to write code for this subproblem, though you could use the short grid example you come up with to test the correctness of your earlier code). *Hint:* Sometimes it might be better to wait and not do anything in anticipation of a much better reward in the future, right?

Problem 4: Greed doesn't always pay (15 points)

In class, we discussed the Activity Scheduling Problem as a canonical example of a problem that lends itself to a greedy solution, but only if the right strategy is chosen: not any old greedy strategy will do. Recall

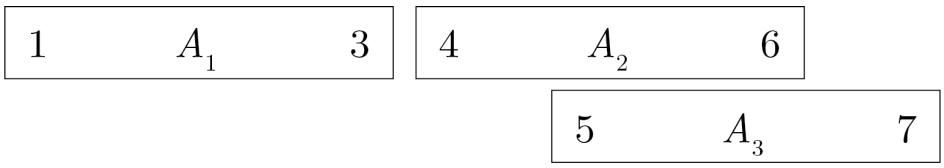

Figure 1: *Example diagram of three activities with equal duration. Activity A_1 begins at time 1 and ends at time 3. Activity A_2 begins at time 4 and ends at time 6. Activity A_3 begins at time 5 and ends at time 7. Activities A_2 and A_3 overlap and thus are mutually exclusive.*

that the goal of the Activity Selection Problem is to maximize the number of selected activities, not their total length.

In this problem, you will be asked to construct counterexamples to prove that certain greedy approaches will not work in every scenario. For each problem, you are required to provide a diagram that illustrates your counterexample following the general structure shown in Figure 1. This diagram can be drawn by hand or computer and can be either embedded in your `README.problem4.[txt/pdf]` or uploaded as a separate file at time of submission.

Note: A counterexample diagram is necessary, but not sufficient, to receive full credit; you would also need to explain how your counterexample forces the greedy algorithm to produce a suboptimal solution.

List of files to submit

1. `README.problem4.[txt/pdf]`
2. `FLAG.txt`
3. `(counterexample diagrams if they are not part of your README)`

a) Consider a greedy algorithm that tries to build up the set of activities by always selecting the activity of the shortest duration from those that are compatible with the ones already selected (if there is a tie, you cannot assume anything about how the algorithm will break the tie). Construct a counterexample to prove that this approach does not always yield a correct solution.

b) Consider a greedy algorithm that tries to build up the set of activities by always selecting the activity that overlaps the fewest activities remaining unselected (if there is a tie, you cannot assume anything about how the algorithm will break the tie). Construct a counterexample to prove that this approach does not always yield a correct solution either. The counterexample in this case is a little more complicated than the one in the previous case, so it will require a bit more thought. Remember: the goal is to construct a situation where enacting the greedy strategy leads to a choice being made that later precludes the possibility of doing better. You know exactly what choices the strategy will make, so set up a situation where it falls into a trap (because you cannot guarantee what the algorithm will do in a tie, you need to "force the algorithms hand" unambiguously).

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: 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 and two terminals, where the buzzer sounds when the circuit across the two terminals of the tester is closed. In particular, if two wires were to be connected by electrical tape at their far ends, 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.