COMPSCI 260, Introduction to Computational Genomics **16 September 2022**

Problem Set 2

Due: 5pm on 30 September 2022

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

Where you are asked to create Python implementations of your algorithms in this problem set, you must first describe them clearly and concisely in plain English sentences within your README *before coding.* We ask you to do this because having the algorithm clear in your head before you write any code will help you write your code much more quickly. In your plain English description, 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. You may also use pseudo-code if it helps, but pseudo-code should still be accompanied by a plain English description. Finally, for any algorithm you develop in this problem set, you are additionally required to analyze its (asymptotic) running time.

Problem 1: The limits of force (10 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’. We will talk about this in more detail later in the course, but for now, we can consider a simple version of the problem to begin thinking about it. Suppose you are given the complete nucleotide sequence of a reference genome of length n. Suppose you are also given the nucleotide sequence of a single read of length m, which is a different nucleotide sequence that may or may not exist within the reference genome. Consider the mapping problem: determining whether or not the read can be found (as an exact substring) somewhere within the reference genome (you can ignore the other strand), and if so, determining all the locations in the genome where it occurs.

a) Assuming a uniform nucleotide distribution for both the genome and the read, what is the expected number of times a read sequence of length m will occur 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 implementing this algorithm; you just need to describe it and analyze its running time.

Reflect

c) Now imagine having to map k distinct reads to a single 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, because the total running time will grow linearly in k. Suppose, however, that you could pre-process the genome to create a data structure that will allow you to query the genome as to whether it contains a specific read sequence in only $\Theta(m \log n)$ time. The downside is that it will take your code $\Theta(n \log n)$ time to build this data structure. *Reason about the number of reads you would need to map in order for it to have been worth it for you to adopt the approach of pre-processing the genome to produce this data structure. What, ultimately, are the trade-offs between the brute force approach and the pre-processing approach?*

Problem 2: Alien invasion! (20 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, Dr. Jean Hunter, has managed to use an ORF-finder to find all the genes in X. After translating those ORFs into peptide sequences, she has assessed, for each one, 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 (owing to the fact that these aliens may possess super-intelligence). Dr. Hunter has worked all through the night to produce this list and is about to crash from lack of sleep, so she tasks you with the job of identifying the region (contiguous segment) of the alien genome that possesses the highest collective similarity to the human genome. More precisely, she defines the "collective similarity" of a region of the alien genome to be the sum of all the similarity values of the genes contained within that region; your mission (should you choose to accept it) is to maximize this score. For example, in the sample list above, the region of highest collective similarity is the sublist $[4, 8, -2, -1, 1, 10]$, whose collective similarity is 20. Note that you should return an empty region (whose collective similarity is 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 their run times compare empirically. 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 representing the highest collective similarity of any region in the alien genome. This common function signature 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 an algorithm's correctness; it is true that failing a test proves incorrectness, but the converse doesn't hold.

Please note that we only ask you to output the highest collective similarity score that is achievable. You do not need to output the specific region in the alien genome that achieves that score—though naturally you should try that on some or all of your implementations if you are looking for an extra challenge :-). [*Note:* If you do try it, it's probably wise to report the indices of the region rather than the region itself, because the latter could be quite long.]

Develop

a) A brute force way to solve the maximal collective similarity problem would be to exhaustively consider every possible sublist of the input list, and simply compute the collective similarity for each sublist, keeping track of the largest such value. Interestingly, even given the constraint of solving the problem with this kind of brute force approach, sometimes one can still design algorithms that are more or less efficient! As an example of an inefficient design, imagine an algorithm with an outer loop that exhaustively enumerates all possible sublists (a new sublist is chosen each time through the loop), and then within the loop (once a specific sublist has been chosen), a sum function is called to add up all the values in the sublist. *How long will it take this inefficient brute force algorithm to run?*

There is a way to solve the maximal collective similarity problem that is still brute force, but is asymptotically faster. To see this, first observe that the inefficient strategy above would always run in the same total time no matter what order the sublists are chosen within the outer loop. *Claim*: One can more thoughtfully choose the order in which to iterate over the sublists that enables one to compute the sum of the values in each sublist more quickly. Can you design an algorithm that intelligently orders the sublists in such a way that facilitates a more rapid computation of their sums? Analyze the complexity of your algorithm. Finally, implement this more efficient version of a brute force algorithm 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`.

Apply

d) Now, *empirically evaluate* (i.e., measure) the running times of each of your algorithm implementations on randomly generated lists of different pre-specified lengths, as described below. The generated lists will contain integers drawn randomly from the set $\{-10, \ldots, 10\}$, but your algorithms need to work for lists of arbitrary values.

A code snippet using the Python `timeit` package is provided in `collective_similarity.py` for you as an example of how you can empirically evaluate 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 your README in table format (you can create this table by hand, or you can have your code output it for you and then paste it in: your choice). *Extra Challenge:* A minor optimization that won't affect your `timeit` measurements but could save a little overall run time is to think of a way to generate the random inputs for all of the above tests more efficiently than just doing it separately for each test.

Hint: Among all the input/algorithm combinations above, the slowest one should take a few minutes (we've seen as little as one minute and as many as ten, depending on the speed of a single core in your computer's CPU). So if you've written code that runs on any of the inputs above in markedly more than ten minutes, there's probably something not quite right.

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 the algorithm, the list length n, and the running time?

Problem 3: Pebble beach (20 points)

List of files to submit:

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

Plan

Imagine that you are given a game board arranged as 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; you need not use all the pebbles. 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 where pebbles have 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) Let us consider the overall problem of finding the maximum value of a valid placement of pebbles on the board. We say that two patterns are *compatible* (with one another) if they can be placed on adjacent rows when forming a valid placement. It follows that a valid placement will consist of a succession of compatible patterns moving down the grid. The question becomes: Among all of those valid placements, what is the highest possible value you can get?

We can first try to break this big problem down into smaller problems by considering subproblems of size $k \in \{0, \ldots, n\}$, in which we consider only the first k rows of the grid. While breaking the problem down this way is a good step, it turns out that it will not be quite enough to solve the overall problem. It will very much help your intuition to try it for a few minutes on a small problem and see why this doesn't quite work.

As it happens, to solve this problem we will need to consider multiple subproblems of the same size k that differ according to their *pattern type*, which is the pattern that occurs in the last (k^{th}) row. Putting this all together, the set of subproblems we will need to solve can now be "named" based on both their size and their pattern type.

With the above hints and descriptions in place, and using the notions of pattern type and pattern compatibility, describe an $O(n)$ dynamic programming algorithm for computing the maximum value of a valid placement on the grid.

Develop

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

Apply

d) A sample grid with random integers between 1 and 120 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 placing no pebbles 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 here 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 (10 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 that the goal of the Activity Selection Problem is to maximize the number of selected activities, not their total length.

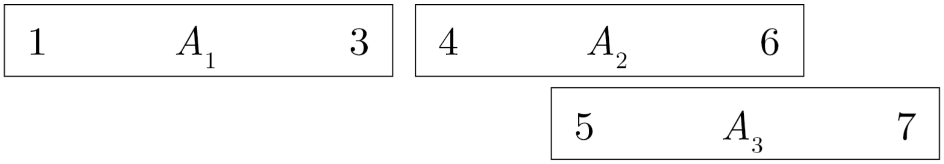

Figure 1: *Example diagram of three activities with equal duration. Activity A_1 begins at time 1 and ends at time 3. Activity A_2 begins at time 4 and ends at time 6. Activity A_3 begins at time 5 and ends at time 7. Activities A_2 and A_3 overlap and thus are mutually exclusive.*

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. `(counterexample diagrams if they are not part of your README)`
3. `FLAG.txt`

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. The overlaps must be recomputed after each round of activity selection, so that activities incompatible with the selections so far are omitted. Again, 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 will need to unambiguously "force the algorithm's hand").

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

List of files to submit:

1. `README.problem5.[txt/pdf]`

A 260-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 in the basement and on the roof. 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 at one end of the conduit to determine whether they have been 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 260 floors of 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 traverse 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.