Problem Set

Please complete this problem set by September 14, 2021 at 11:59PM.

Problem 0 – Collaboration Policy (10%)

Read and sign the Collaboration Policy in the Course Content folder on HuskyCT. Hand in your signed form with your homework submission. We cannot grade your work without a signed form.

Problem 1 – Introductions (10%)

The goal of this problem is to introduce yourself to the course staff and to think about what you want to get out of the course. Write at most two paragraphs about yourself. This may include your academic and non-academic interests, preferred programming languages, courses that you are excited to take in the future, or areas of computer science that you find the most interesting. Finish by describing what do you want to learn by taking this course. Think about what would be most beneficial to your prospective career.

Problem 2 – Preliminaries (10%)

This problem tests your retention of prerequisite materials. If some of these problems are difficult, please go back and review material from the prerequisites.

1. How many permutations are there of the set of n numbers?
2. How many subsets of three numbers are there from a set of n numbers?
3. How many subsets of S are there when $|S| = n$?
4. Express the vector $\boldsymbol{a} = [-3, 6]$ as a linear combination of $\boldsymbol{b} = [1, 2]$ and $\boldsymbol{c} = [3, 0]$.
5. Explain why this is possible using the term "basis". You may also consider reviewing linear independence and the span of a set of vectors.

Problem 3 – Complexity (20%)

1. For functions A and B and constant $c > 1$, indicate which of $\{A = O(B), A = \Omega(B), A = \Theta(B)\}$ holds. Here, we use the notation that $log_2 = lg$. *Hint: for these problems it is useful to know some mathematical identities. See CLRS section 3.2 or **standard_notations_and_common_functions.pdf** in HuskyCT Lecture 2 course notes.*

   (a) $A = n^c$, $B = c^n$

   (b) $A = n^{lg(c)}$, $B = c^{lg(n)}$

   (c) $A = lg(n!)$, $B = lg(n^n)$. For this problem, you will find Stirling's approximation useful: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Omega\left(\frac{1}{n}\right)\right)$.

   (d) $A = 3^{3^n}$, $B = 3^{n^2}$

2. Place the following functions in increasing order of growth rate. I.e. if g follows f then $f = O(g)$ is true.

   (a) n^e,

   (b) $n^{\lg(n)}$

   (c) n^π,

   (d) $\lg(n!)$

   (e) $2^{\sqrt{\lg(n)}}$

   (f) n

Problem 4 – Vinder (30%)

Vinder is a new smartphone app designed to make it easier for everyone to eat their vegetables. Vinder compares your genetic information with a large database of vegetable genetic sequences to match you with the right vegetable based on sequence similarity. Specifically, it computes the longest common subsequence between your DNA sequence and the DNA sequences of thousands of vegetables.

An n-length *DNA sequence*, A, is an ordered collection of elements $\in [A, C, G, T]$, or $A \in [A, C, G, T]^n$. A *subsequence* is a sequence formed by deleting elements from another sequence, thus preserving the original order. A k-length *prefix* of sequence A is the first k elements of A. For example, $A = [A, C, C, G, G, A, A, T, C]$ is a sequence. $[A, C, A, T]$ and $[C]$ are subsequences of A but $[T, A]$ is not. $[A, C, C, G]$ is a 4-prefix of A and $[]$ is the 0-prefix.

Our goal is to find the longest common subsequence (LCS) of A, the human DNA sequence, and sequences $B^k \in D$ where D is the database of vegetable DNA sequences and $k = 1, ..., |D|$.

This solution can be found efficiently using dynamic programming. At a high-level, the algorithm is:

1. For each sequence $B^k \in D$
2. Initialize the length of the LCS to 0.
3. Let A_i denote the i^{th} element of A. Consider each pair of A_i and B^k_j; either $A_i = B^k_j$ or $A_i \neq B^k_j$.
   - If $A_i = B^k_j$, then the length of the LCS of the i-prefix of A and j-prefix of B^k is one more than the LCS of the $i-1$-prefix of A and $j-1$-prefix of B^k.
   - If $A_i \neq B^k_j$, then we cannot extend the LCS. Instead, we conclude that the LCS up to (A_i, B^k_j) is the maximum of the LCS up to (A_{i-1}, B^k_j) or (A_i, B^k_{j-1}).

The two aforementioned cases sum up the possibilities at (A_i, B^k_j). We can use these to recursively define the LCS in terms of smaller problem solutions. Your goal is to describe the algorithm to do this.

1. Describe an algorithm to find the length of the LCS that does not use dynamic programming. You do not have to write pseudocode, simply describe the algorithm in enough detail so that we understand it. What is the runtime? Why is it correct? You may keep your answers informal. *Hint: a brute-force or recursive solution should have exponential runtime.*
2. Write out the recurrence relation used in the algorithm defined in the problems description (not part 1 of the answers). That is, write out how the LCS length can be expressed in terms of smaller instances of the problem.
3. Think about how to convert this into a dynamic program. What size table would you need for the LCS problem?
4. Interpret each table entry. That is, give your explanation for what each entry in the dynamic programming table represents. We discussed how to interpret the entries of the dynamic programming table for the similar problem of *edit distance*.
5. Write out the table for $Derek : [A, C, A, G, G, T, T, A, C]$ and $Asparagus$[1] $: [T, C, G, G, A, A, T, A, A]$.
6. Prove your LCS algorithm's (that you defined in 2-5) correctness. *Hint: Many dynamic programming proofs follow a simple structure. Consider the following template:*

[1] provably the best vegetable

(a) Describe what you will show. This should include the variables used to index the table, presented in the order in which your algorithm fills the table values. Also detail here what your algorithm is supposed to compute.

(b) State your induction hypothesis. This includes stating your base case and how your algorithm correctly computes the base cases. Then describe an arbitrary input and corresponding arbitrary entry in the table that will be computed. You can think of this as a *state* of the algorithm at one particular time point. Next, consider an optimal dynamic programming table for your particular problem; detail the last decision made in this optimal solution. This usually falls under a number of different cases based on your recurrence relation.

(c) Consider each case; assuming that everything in the dynamic programming table is correct, prove that your algorithm will correctly compute the current entry in the table. Show that your recurrence relation accurately describes each possibility for the last choice made by the unknown LCS. This is the part of your proof where you compare the real world possibilities to the cases in your code. By showing this, your recurrence relation and code rely on the previous entries in the table to produce a solution at least as good as the optimal LCS.

Problem 5 – Longest Path Problem (20%)

We are given a directed, unweighted graph $G = (V, E)$ where $v_i \in V$ for $i = 1, ..., n$. Additionally, G is ordered, meaning

- An edge can only originate from a vertex with lower index than its destination. Formally, edges may only have the form (v_i, v_j) where $i < j$.
- The only *absorbing* node is v_n. Absorbing nodes have out-degree 0.

Given an ordered, unweighted, directed graph $G(V, E)$, the *longest path problem* is to compute the longest path length from v_1 to v_n.

Algorithm 1: Longest path algorithm

```
def longest_path(v1):
    iter=v1
    length=0
    while iter.hasOutgoingEdge():
        iter=iter.getEdgeSmallestIndex()
        length+=1
    return length
```

1. Demonstrate that Alg. 1 does not correctly solve the problem by giving a counter-example as a graph. The function `iter.hasOutgoingEdge()` returns a boolean indicating whether or not the vertex `iter` has an outgoing edge. The function `iter.getEdgeSmallestIndex()` returns the smallest index of all vertices incident from `iter`. That is, $min_j\{(iter, v_j) \in E\}$. We prefer that your solution be a graph embedding either by hand-drawing and scanning or taking a picture, or drawing it digitally. If none of these options are available, you can simply list the set of edges in the graph.

2. Develop an $O(n^2)$ algorithm for solving the *longest path problem* in ordered, unweighted, directed graphs. Besides the functions defined above, you are free to use any function associated with graph data structure implementations.