Homework 1: BokChain

Due: Sept. 11, 2018 at 6:00 PM (early, 5% bonus)
Sept. 14, 2018 at 6:00 PM (on time)
Sept. 16, 2018 at 6:00 PM (late, 20% penalty)

The written portion of this homework must be typed in Latex and handed in via Gradescope. See `http://cs.brown.edu/courses/cs157/content/docs/CS157LatexLinks.pdf` to get started with Latex, and see the Gradescope post on Piazza, `http://piazza.com/brown/fall2018/cs157`. **Homework will *not* be accepted after the late deadline.** Solutions to different problems may be turned in by different deadlines.

This homework must be *written together in pairs*, and *you must understand and be able to answer questions about the entirety of the joint submission.* You are encouraged, as always, to discuss the problems with other pairs or consult online resources. Make sure to hand in one solution per problem for both you and your partner, and do not forget to put both Banner IDs on it. Note that future pair assignments must be done with a different partner, unless otherwise indicated.

See the Google doc emailed to the course list for your assigned partner. **If you do not have a partner, or have not been getting course emails, email cs1570headtas@lists.brown.edu immediately.**

Working in pairs will give you an opportunity to improve your thinking, communication, and writing skills. If something you write requires a verbal explanation for your partner to understand it, consider this a valuable sign that this explanation should be included in your writeup. In particular, you are responsible for *everything* you and your partner submit. It is an academic code violation to sign your name to something that is not yours.

Please ensure that your solutions are complete, concise, and communicated clearly: use full sentences and plan your presentation before you write. Except in the rare cases where it is indicated otherwise, consider every problem as asking you to prove your result.

Problem 1

(20 points)

Welcome to BokChain! We're devoted to using genetic information to determine your ancestral relationship to bok choy.[a] To keep your genetic information secure, we transmit all of our data on the blockchain, with advanced artificial intelligence techniques. We're seeking senior algorithm developers to help us efficiently compare human and bok choy DNA sequences.

[a] `https://en.wikipedia.org/wiki/Bok_choy`

Definitions

- A **sequence** is an ordered collection of elements $[a_i]$.
- A **subsequence** is a sequence that can be formed from another sequence by deleting elements, thus preserving their order.
- A **prefix** of length k is the first k elements of a sequence.

Examples

- $[9, 2, 5, 3, 7]$ is a sequence.
- $[1, 1, 1, 1]$ is a sequence.
- $S = [1, 1, 2, 3, 5, 8, \ldots]$ is a sequence.
- $[1, 3, 5]$ is a subsequence of S.
- $[2, 1, 5]$ is **not** a subsequence of S.
- $[1, 1, 2, 3, 5]$ is a 5-prefix of S.
- $[]$ is a 0-prefix of all sequences.

Given two sequences $A = [4, 5, \mathrm{v}, 3, \mathrm{D}, 1]$ and $B = [5, 4, 7, \mathrm{D}, 1, 3, 120]$, we see that the length of the longest common subsequence is 3, and is either $[4, \mathrm{D}, 1]$ or $[5, \mathrm{D}, 1]$. We can find these solutions efficiently via dynamic programming. To solve the above problem we will use the following algorithm:

1. Before we have examined any of the characters in either string, the length of the longest common subsequence is zero.
2. We consider each pair of entries, A_i and B_j, where A_i denotes the i^{th} element of A, and B_j denotes the j^{th} element of B. There are two possibilities: either $A_i = B_j$ or $A_i \neq B_j$.
   - In the first case, then the length of the longest common subsequence of the i-prefix of A and j-prefix of B is one more than the longest common subsequence of the $(i-1)$-prefix of A and the $(j-1)$-prefix of B.
   - Otherwise, if the i^{th} element of A is different from the j^{th} element of B, then we have not discovered a new longest common subsequence, and can instead say that the longest common subsequence up to (A_i, B_j) is the *maximum* of the longest common subsequences up to either (A_{i-1}, B_j) or (A_i, B_{j-1}).

   Having analyzed both cases, we can thus recursively define the longest common subsequence in terms of the solution to a smaller problem.

Your task in this problem is to understand and flesh out this algorithm.

1. Describe a simple recursive or brute-force solution (not dynamic programming) to find the length of the longest common subsequence — pseudocode is not necessary here; an informal description is fine. Find the runtime and explain why it is correct. If it requires a proof, include one, however keep it short; you should be able to explain this concisely. *Note: the brute-force and recursive solutions should yield some form of exponential runtime.*

2. Explicitly write out the recurrence that both a recursive and a dynamic programming solution would use. Namely, how can the length of the longest common subsequence of two strings be expressed in terms of smaller versions of the same problem?

3. What is the size of the table a dynamic programming algorithm would use to solve the longest common subsequence problem?

4. What is the *meaning* of each entry of the table? That is, what does a given cell in your table represent in terms of the longest common subsequence problem? A corresponding explanation for the *edit distance algorithm* discussed in class is: "The edit distance algorithm fills in a table indexed by i and j, where each entry represents the cost of the best way to edit the first i characters of the first string to equal the first j characters of the second string."

5. Write out the table for these inputs, and find the longest common subsequence:

   Paul: $[\,\mathrm{A,G,C,G,A,G,A,G,A,G,T,G}\,]$ and

   Bok Choy: $[\,\mathrm{C,G,A,T,C,G,A,T,T,T,A,G}\,]$.

6. Prove the correctness of your algorithm. We suggest you follow the standard outline for proving correctness of a dynamic programming algorithm, where you must fill in the parts in square brackets (though ultimately, your proof must make sense to you; there are no "magic words" here; come to collab hours to figure out how to get to the essence of your proof):

   "We will show by induction on [you should name the variables indexing the table, traversed in the order of the algorithm] that our algorithm correctly computes [you should specify what it is supposed to compute].

   (Here, you'll need to state your inductive hypothesis. You should consider what you're inducting over. What should you assume is true for previous entries in the table?)

   As a base case for the induction, we note that [you should justify that the initialization steps of the algorithm correctly computes what it is supposed to for those cases].

   Now consider [you should describe an arbitrary input to the algorithm], and [describe an arbitrary table entry that is about to be filled out, essentially specifying a moment in time for the algorithm]. Consider an optimal [you should specify the form of a solution to the problem], and consider the last decision made in this optimal solution. There are [you should specify how many cases exist in the definition of your recurrence relation] cases.

   (The following is the most critical part of a dynamic programming correctness proof. If it is not clear what you should be proving, please come to collaboration hours.)

   [Go through each case and prove that, assuming that everything previously put into the table is correct, your algorithm will correctly fill out the current entry of the table. You should do this by showing that, for each possibility for the last choice made by the unknown longest common subsequence, there is a case in your recurrence relation that accurately describes this possibility (here is where you relate the possibilities in the real world to the cases covered by your code). Thus your recurrence relation will rely on the previous entries in the table to produce a solution that is at least as good as the optimal solution's choice.]"

Problem 2

(20 points)

Recall the definition of a *sequence* and a *subsequence* from the previous problem.

A palindrome is a sequence that reads the same backwards and forwards, such as `acegeca`.

1. What is the length of the longest palindromic subsequence of each of the following sequences? Please also write down the sequence, but there is no need to show your work.

   (a) `a`

   (b) `abcacb`

   (c) `dabcacbd`

   (d) `localareaman`

   (e) `toptensecretsofcs`

2. Design a dynamic programming algorithm that takes as input a sequence $x_0x_1 \ldots x_{n-1}$ and returns the length k of a longest palindromic subsequence. Its running time should be $O(n^2)$. If you are unsure how to write this up, consider the following steps:

   - Identify a table $T[\cdots]$ used for your dynamic programming algorithm. This means you should say how the table is indexed and what its content means. For example, if your array is a one-dimensional array, what does $T[i]$ mean?
   - Specify the order in which you will fill in T.
   - Explicitly define the recurrence relation used to fill in T.
   - Provide pseudocode (only a few lines should be needed) summarizing the above work.
   - It should be fairly clear from what you have already written why your algorithm runs in $O(n^2)$ time and is correct, so state and prove it now. (Again, we suggest using the outline given at the end of the previous problem.)

   (**Note:** It turns out that you can solve this problem using the algorithm of the previous problem. This is interesting, but proving this relationship is far more complicated than just solving this problem from scratch, using the standard dynamic programming proof. We recommend that you *do not* try to approach this problem via the previous problem, but use the previous problem only as inspiration, and follow the above outline, describing your dynamic programming approach from scratch.)

Problem 3

(15 points)

Your task here is to prove the correctness of the Depth-First Search (DFS) algorithm, implemented on a rooted tree. In pseudocode, the algorithm is:

```
DFS(node,val)
if node.key = val
    output node
    halt
for each child x of node
    DFS(x, val)
```

Specifically, given a tree of nodes, where each node has zero or more children, and each node stores data in a *key*, show that if a certain key is stored in some node in the tree, then running the above DFS algorithm *starting at the root* of the tree will find such a node. If you are unclear what a "tree" is, look up `http://en.wikipedia.org/wiki/Tree_data_structure`.

Hint: Prove this by induction. Think carefully about your induction hypothesis (what happens when you are inductively considering running your algorithm on a subtree that does *not* contain the key, and how should this affect your induction hypothesis?). Remember, induction works over the positive integers, so make sure there is a positive integer in your induction hypothesis. The challenge in this problem is that it is "obvious" that depth first search works; but you need to translate your intution into something concrete. Remember that the structure of your proof should often be inspired by the structure of the algorithm.

Problem 4

(30 points)

Now that we've figured out how to find all of the bok choy a person is related to, we need to display all of the information we've uncovered to our customers. We want to arrange the text in an aesthetically pleasing way.

The first engineer we hired ordered all of the words in this way: as you go along, greedily put as many words as possible on the current line until you hit the margin, then go to the next line.

Our Chief Aesthetics Officer (CAO) decided this would be not aesthetic enough and fired the engineer in question. The CAO has now hired you to devise a more elegant word-arranging scheme.

You decide that you'd like to somehow arrange the text such that you minimize the **penalty** on each line of text. **Penalty** is defined to be the square of the difference (in number of characters) between the last letter in a line of text and the actual line cutoff length. Thus, your algorithm is to minimize the total **penalty** over all lines of text *except the last line*. Further, the last line cannot go over the margin. Note that on any other line, your algorithm may arrange words such that they go over the line cutoff length, if this minimizes the total **penalty**.

For example, if you want to write: "Your closest bok choy relation is Vanessa The Great, who was eaten by Barack Obama in 2005." with margins that allow 16 characters between them, the greedy solution would do this:

`|Your closest bok|` : 16 characters, 0 characters remaining ($0^2 = 0$ penalty)
`|choy relation is|` : 16 characters, 0 characters remaining ($0^2 = 0$ penalty)
`|Vanessa The     |` : 11 characters, 5 character remaining ($5^2 = 25$ penalty)
`|Great, who was  |` : 14 characters, 2 characters remaining ($2^2 = 4$ penalty)
`|eaten by Barack |` : 15 characters, 1 character remaining ($1^2 = 1$ penalty)
`|Obama in 2005.  |` : 14 characters, 2 character remaining (0 penalty — last line)

This has a total penalty of $0+0+25+4+1+0 = 30$. However, the optimal solution has only 5 penalty:

`|Your closest bok|` : 16 characters, 0 characters remaining ($0^2 = 0$ penalty)
`|choy relation is|` : 16 characters, 0 characters remaining ($0^2 = 0$ penalty)
`|Vanessa The Grea|t,` : 18 characters, 2 characters over ($2^2 = 4$ penalty)
`|who was eaten by|` : 16 characters, 0 characters remaining ($0^2 = 0$ penalty)
`|Barack Obama in |` : 15 characters, 1 character remaining ($1^2 = 1$ penalty)
`|2005.           |` : 5 characters, 11 characters remaining (0 penalty — last line)

1. Design a dynamic programming algorithm to achieve the minimum penalty, when input a sequence of words. Your algorithm should both find the minimum possible penalty as well as the line splitting scheme that yields this penalty. As a first step, or for partial credit, consider using the same penalty for the last line as for all the others.

2. Prove the correctness of your algorithm.
3. What is the runtime of your algorithm? Justify your answer.

Hint: The point of a dynamic programming recurrence is to optimally make a choice about the "last action", making use of a table that represents information about how to optimally solve subproblems. How can you set things up (the "subproblem structure") so that you will be able to cleanly make optimal choices based on optimal solutions to smaller versions of the same problem?

Problem 5

(15 points)

The n^{th} roots of unity are the n complex values whose n^{th} power is one. These numbers exhibit interesting behavior in the complex plane and find many applications in algorithms, including the Fourier Transform, which we will see soon in class and on the next homework.

Given a fixed n, the greek letter ω (omega) is used to denote the complex number with radius 1 and angle $2\pi/n$, namely the first n^{th} root of unity, from which all the others can be derived.

1. Find the third roots of unity, and express them in regular coordinates (real and imaginary parts), polar coordinates (radius and angle), and as powers of e.
2. For $n = 3$, label the above roots of unity as ω, ω^2, and ω^3. Also, compute the squares of these three values and describe how they relate to the roots of unity.
3. Compute 1 divided by each root of unity and compare these answers to the roots of unity.
4. Compute the complex conjugate of each root of unity and compare these answers to the roots of unity.
5. Cube each root of unity. Show your work. (The answer should be obvious, this is an attempt to elucidate the behavior of ω.)
6. For the n^{th} roots of unity, what is the complex conjugate of ω^j where j is an integer between 0 and $n-1$? Express your answer in terms of a *positive* power of the n^{th} root of unity.
7. Write down the *fourth* roots of unity. For the remainder of this problem, ω will represent the first fourth root of unity.
8. Rewrite ω^5 and ω^7 with an exponent between 0 and 3 where ω is the first fourth root of unity.
9. What is $\sum_{j=0}^{3} w^j$?
10. For the next problem, consider the following matrix M:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{pmatrix}$$

    (a) Find the result of $M \times M$.
    (b) Find the result of $M \times \overline{M}$, where $\overline{M}$ denotes the element-wise complex conjugate of M.
    (c) Comment on anything interesting you notice. Simplify the result as much as you can using what you have learned in previous parts. As we will see when we look at Fourier transforms, everything we have seen here generalizes from 3rd and 4th roots to all nth roots of unity.