Homework 1

Due: Sept. 13, 2016 at 6:00 PM (early, 5% bonus)
Sept. 16, 2016 at 6:00 PM (on time)
Sept. 18, 2016 at 6:00 PM (late, 20% penalty)

The written portion of this homework must be completed and handed in through the CS157 hand-in bin, which is located on the CIT 2nd floor between the Fishbowl and the locker. Each problem must be stapled, labeled (with the problem number and hand-in time), and handed in separately. The programming portion of the homework, when applicable, must be handed in electronically via the hand-in script. 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 division of labor is prohibited. 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 cs157headtas@cs.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

(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)
1   if node.key = val
2       output node
3       halt
4   for each child x of node
5       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. 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 intuition into something concrete. Remember that the structure of your proof should often be inspired by the structure of the algorithm.

**Problem 2**

(20 points)

Winnie the Pooh would like to have some hunny for his eleven o’clock snack. However, Tigger has put a contraption on Winnie’s favorite hunny pot to prevent him from opening it. Winnie needs your help. Help him find the longest common subsequence between two sequences in order to get him some hunny.

**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 \(A = [4, 5, v, 3, D, 1]\) and \(B = [5, 4, 7, D, 1, 3, 120]\), we see that the length of the longest common subsequence is 3, and is either \([4, D, 1]\) or \([5, D, 1]\). We can find these solutions efficiently via dynamic programming. To solve the above problem we will use the following algorithm. Before we have examined any of the characters in either string, the length of the longest common subsequence is zero. 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
1. Describe a simple recursive or brute-force solution (not dynamic programming) to this problem—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:


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:

   “We will show by induction on [the variables indexing the table, traversed in the order of the algorithm] that our algorithm correctly computes [what it is supposed to]. As a base case for the induction, we note that [the initialization steps of the algorithm correctly compute whatever it is supposed to for those cases]. Now consider [an arbitrary input to the algorithm], and an arbitrary [i, j, k, whatever, specifying a table entry that is about to be filled out, essentially specifying a moment in time for the algorithm]. Consider an optimal [solution to our problem], and consider the last decision made in this optimal solution. There are [some small number of] cases: [go through each case, typically these will be cases of the recurrence at the center of the algorithm, and show how, assuming (by the induction hypothesis) that everything previously put in the table is correct, you can conclude that no matter which of the cases the last step of the unknown arbitrary optimal solution does, your algorithm can do at least as well.]”
Problem 3

(15 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) dabacbd
   (d) tiggerandpiglet
   (e) poohandhishunnypot

2. Design a dynamic programming algorithm that takes as input a sequence \( x_0 x_1 \ldots x_{n-1} \) 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 4

(10 points)
Instructions: To turn in this problem electronically, please open up a terminal and run the command cs157_handin hw1-p4 from within the directory containing your code, which will copy and submit the entire directory. (Please note that this script will not recursively search for files in subdirectories.)
The following recurrence computes the edit distance between two words $A$ and $B$. See Wikipedia, a textbook, or your memory from lecture for more details.

$$S(i, j) = \begin{cases} S(i-1, j-1) & \text{if } A[i] = B[j] \\ \min(S(i-1, j), S(i, j-1), S(i-1, j-1)) + 1 & \text{if } A[i] \neq B[j] \end{cases}$$

Fill in the method `editDistance()` that calculates the edit distance between any two input words. You can find a stencil in `/course/cs157/pub/stencils/hw1-p4/EditDistance.java`. Be sure to use this stencil, or else your TAs will be sad. Furthermore, please do not print anything to standard output in the body of `editDistance()`.

**Note:** Your code will be graded on correctness. You only need to comment your code and make it look nice if your code is wrong and you are hoping for partial credit.

Later in the course we will reconsider edit distance from the perspective of optimizing speed and memory, so if you are feeling ahead of the game you might want to start thinking about these issues.

**Problem 5**

(25 points) A.A. Milne is typing up a brand new Winnie the Pooh story. He finds that his word processor’s greedy approach to word wrapping is not aesthetically pleasing enough for him. An easy way to do word wrapping is: 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. However, perhaps a more aesthetically pleasing result might be to minimize the sum for every line except the last line of the square of the distance between the end of the line and the margin, allowing the text to go over the margin. However, the last line cannot go over the margin. For example, if A.A. Milne wants to write “Life is a journey to be experienced, not a problem to be solved” with margins that allow 16 characters between them, the greedy solution would do this:

<table>
<thead>
<tr>
<th>Life is a</th>
<th>: 9 characters, 7 characters remaining ($7^2 = 49$ penalty)</th>
</tr>
</thead>
<tbody>
<tr>
<td>journey to be</td>
<td>: 13 characters, 3 characters remaining ($3^2 = 9$ penalty)</td>
</tr>
<tr>
<td>experienced, not</td>
<td>: 16 characters, no penalty</td>
</tr>
<tr>
<td>a problem to be</td>
<td>: 15 characters, 1 character remaining ($1^2 = 1$ penalty)</td>
</tr>
<tr>
<td>solved.</td>
<td>: 7 characters, 9 characters remaining (0 penalty — last line)</td>
</tr>
</tbody>
</table>

This has a total penalty of $49 + 9 + 0 + 1 = 59$. However, the optimal solution has only 5 penalty:

|Life is a journey | : 17 characters, 1 character over ($1^2 = 1$ penalty) |
|to be experienced,| : 18 characters, 2 characters over ($2^2 = 4$ penalty) |
|not a problem to  | : 16 characters, no penalty                              |
|be solved.        | : 10 characters, 6 characters remaining (0 penalty — last line) |

1. Design a dynamic programming algorithm to achieve this beauty in documents. 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.
Problem 6

(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 \(\omega\) (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 \(\omega\), \(\omega^2\), and \(\omega^3\). 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 \(\omega\).)

6. For the \(n^{th}\) roots of unity, what is the complex conjugate of \(\omega^j\) 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, \(\omega\) will represent the first fourth root of unity.

8. Rewrite \(\omega^5\) and \(\omega^7\) with an exponent between 0 and 3 where \(\omega\) is the first fourth root of unity.

9. What is \(\sum_{j=0}^{3} \omega^j\)?

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 \(n^{th}\) roots of unity.