EECS 340, Breakout Session Notes, November 18, 2019

1. As usual, begin by handing out the quiz and collect it quiz at 3:40.

2. Place the students into groups and give them this problem: Many programs that you enter text on (messaging apps, a word processors) compare the words you enter to a dictionary. If the word is not found, the program assumes it is a typographical error and presents the user with a list of possible alternatives. The goal is to find words that are “close” to the word the user entered. One way to define “close” is with *edit distance.*

   The edit distance between two strings $s1$ and $s2$ is the minimum number of changes that you must make to $s1$ for it to become $s2$. There are three types of changes:

   - You can remove a character of $s1$.
   - You can add a character to $s1$.
   - You can replace a character of $s1$ with a different character.

   Design an algorithm that takes two strings, $s1$ and $s2$, $s1$ had n characters and $s2$ has m characters, and gives the (minimum) edit distance between them.

3. The first thing the students should ask is if it feels like an NP-hard problem or a polyomial time problem. One thing that should jump out at them is that we are minimizing a metric and we have choices. That suggests dynamic programming.

4. The next question is whether we want the minimum subproblem from each choice. To answer that, we have to think about what is the subproblem.

5. A useful hint is to think about the change needed to match that last character of $s1$ to the last character of $s2$.

6. They should hopefully start to realize that there are three choices, and depending on the choice, we have a different subproblem. That subproblem can be solved optimally, and we then do the last step to match the strings.

7. The next step is to use that insight to create the recurrence relation. What information do we need to know at each step? Which prefix of $s1$ we need to match to which prefix of $s2$.

8. Here is a possible solution and proof:

   Create a table $T[i, j]$ to store the minimum edit distance between the first i characters of $s1$ and the first j characters of $s2$.

   $T[i, j] = \min\{T[i-1, j-1] + c(i, j), T[i-1, j] + 1, T[i, j-1] + 1\}$ where $c(i, j)$ is 0 if $s1[i] = s2[j]$ and 1 if $s1[i] \neq s2[j]$. We initialize the table with $T[0, 0] = 0$.

   Proof by induction. The base case is trivial.

   Assume $T[i-1, j-1]$ stores the minimum edit distance between the first $i-1$ characters of $s1$ and the first $j-1$ characters of $s2$.
   Assume $T[i, j-1]$ stores the minimum edit distance between the first i characters of $s1$ and the first $j-1$ characters of $s2$.
   Assume $T[i-1, j]$ stores the minimum edit distance between the first $i-1$ characters of $s1$ and the first j characters of $s2$.

   Consider $T[i, j]$. We have four choices for our “last” conversion to get $s1[1..i]$ to be equal to $s2[1..j]$

   - We match the ith character of $s1$ with the jth character of $s2$ (only if they are the same). In this case, we had to previously convert the $i-1$ characters of $s1$ to equal the $j-1$ characters of $s2$.

- We replace the ith character of $s1$ with the jth character of $s2$. In this case, we had to previously convert the $i-1$ characters of $s1$ to equal the $j-1$ characeters of $s2$.
- We remove the ith character of $s1$. In this case, we had to previously convert the $i-1$ characters of $s1$ to the j characters of $s2$.
- We add the jth character of $s2$ to $s1$. In this case, we had to previously conver the first i characters of $s1$ to the first $j-1$ characters of $s2$.

For each of these choices, it does not matter how we match the resulting prefixes because those choices do not affect our last choice. Therefore, to get the shortest edit distance, we should match them optimally. By the induction hypotheses, $T[i-1, j-1]$, $T[i, j-1]$, and $T[i-1, j]$ store the minimum edit distance for each of the possible prefix cases above. That completes the proof.

How do we get the solution? For this proboem, we only want the minimum edit distance between $s1$ and $s2$, and that will be the value stored in $T[n, m]$

The running time is $\Theta(nm)$ because we have to fill a $n \times m$ table, and it takes constant time for each entry.