Math 590 Extra Credit Fall 2019

Due: **Tuesday**, November 26

1. **Corrupted Text**

   You are given a string of $n$ characters $S[1 \ldots n]$, which you believe might be a corrupted text document in which all of the punctuation, spaces, and formatting (capital letters) has been removed. For example, the string could look something like:

   `'duringthewholeofadulldarkandsoundlessdayintheautumnoftheyear...'`

   You wish to reconstruct the original document using a dictionary or report that this is not possible. The dictionary is available in the form of a Boolean function `dict(w)` that will return `True` if `w` is a valid word and `False` otherwise, with each call to `dict(w)` taking $O(1)$ constant time.

   **Before discussing the solutions, one important note:** This is actually a very common interview question, but we have altered it very slightly for this course. Usually, you would be given a dictionary list of valid words. If you just look up words in that list, a lookup operation would take $O(n)$ time. To get the best performance, you would start by inserting the dictionary list into a hash table, which then gives you the desired $O(1)$ amortized lookup times.

   (a) [2pt] Describe a **recursive** use-it or lose-it algorithm for determining if the input string $S$ is a valid text document, i.e., is a sequence of valid words.
   (*Hint*: try asking whether a specific substring is in the dictionary. What two pieces of information are needed to specify a substring?)

   ANSWER: We will track two indices, $i$ and $j$, which will define the current substring we are looking at. We will then check if this substring can be considered a word. If it can, the use-it case means treat the substring like a word and remove it from the string. Meanwhile, the lose-it case says to assume that this substring is not a word (but might be part of a larger word), so extend the substring.

   Note: there are many different ways to set up this problem. Here, we start $i$ and $j$ at 0, and increase them as we look through the string. It would have been possible to start them both at the end of the string, and decrease them to search through the string.

   Pseduo-code on next page.

```
def text(S, i, j):
    # Takes in a string S and two indices i and j.
    # Returns True if S[i:] can be made into a sentence, else False.

    # Base Case: j has reached the end of S.
    # This substring is either a word, or it isn't.
    if j == len(S)-1:
        return dict(S[i:j+1])

    # Check if S[i:j+1] is a word and set useIt accordingly.
    if dict(S[i:j+1]):
        useIt = text(S, j+1, j+1) # Cut out the entire word.
    else:
        useIt = False

    # Set loseIt value by extending j.
    loseIt = text(S, i, j+1)
```

(b) [2pt] Describe, in clear English or pseudo-code, how you would convert your recursive algorithm into a Dynamic Programming solution. Indicate clearly how the DP table is set up and in what order the cells should be filled.

ANSWER: We will store our solution in a 2D table. I will take $i$ to be the row, and $j$ to be the column.

First note that we are considering the substring from $i$ to $j$, so it must be that $i \leq j$. So we only need to consider the upper right half of the table.

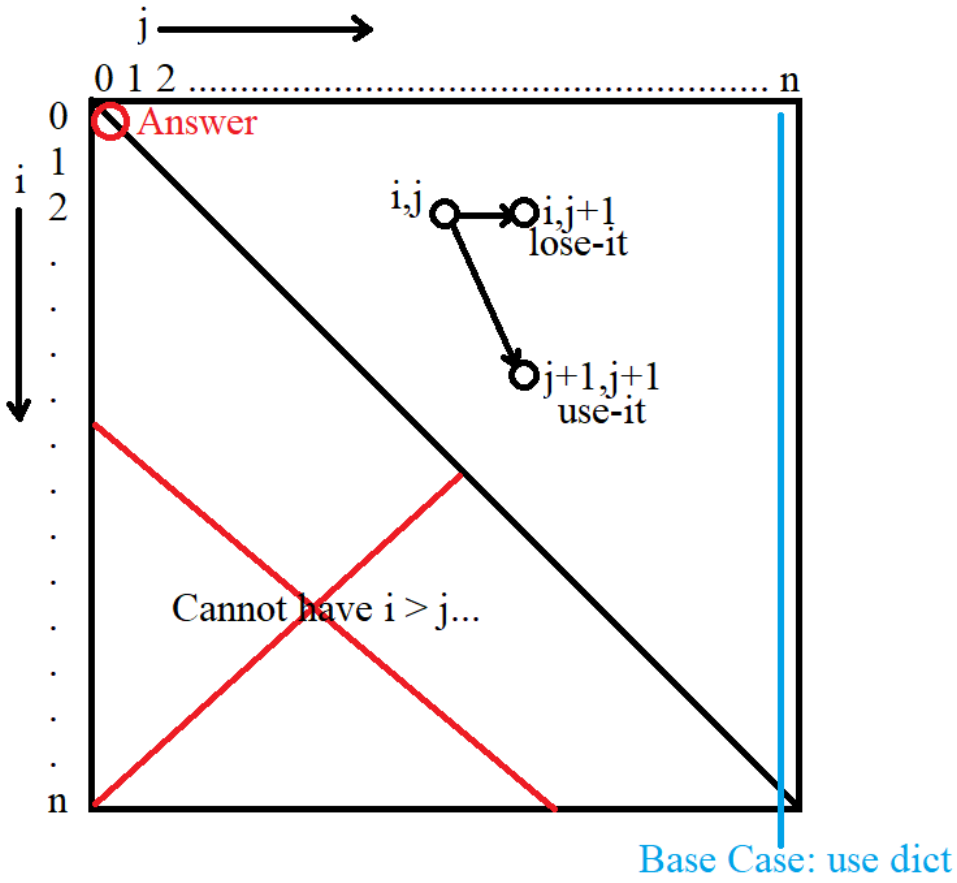
Now notice that the recursive calls for $(i, j)$ come from $(i, j+1)$ and $(j+1, j+1)$. Since $i \leq j$, this means that the recursive calls must always come from the column to the left, potentially at a lower row.

Our base case is when `j = len(S)-1`, which is the rightmost column.

So fill down each column, starting with the rightmost column and moving to the left. Stop filling when we hit the main diagonal (since $i \leq j$ means anything below that diagonal cannot be filled). Our answer will be stored in the $i = 0$, $j = 0$ location at the top left corner.

Note: we are filling the table with True or False values.

An image of my DP table is on the following page.




(c) [1pt] Describe how you could construct the original document using your DP table.

ANSWER: Start at the $(0, 0)$ entry. If it is False, then return False. If it is True, then trace back through the use-it or lose-it cases. If lose-it, just keep tracing back. If use-it, the substring `S[i:j+1]` corresponded to a word. Remember that we are tracing forwards from the front of the string, so the first use-it is the first word, the second is the second word, etc.

If we want to report all possible solutions (not required for this assignment, but often asked as a follow-up in interviews), we will need to trace all possible paths. If there was a tie (i.e., if both use-it and lose-it were True), then either path would reconstruct a proper solution. In this case, we can use DFS to reconstruct all possible paths. We use depth-first because reaching a leaf means reaching a base case, i.e., when we reach the end of a brach, we have reconstructed one complete solution.

(d) [1pt] What is the runtime of your Dynamic Programming algorithm? Give a brief explanation.

ANSWER: We had to fill half the table, with each cell taking constant time to fill. So $O(n^2)$ work. *(a different approach continued on next page)*

Now, we could actually have solved this problem with a 1D approach.

Let's consider the single index $i$, so that we are looking at the substring `S[0:i+1]`. Now we make an observation: this substring can only be a proper sentence if there exists some $j < i$ such that `S[j:i+1]` is a word, and `S[0:j]` is a sentence.

So our 1D approach will consider a specific $i$ and loop over the possible values of $j$. We can only return True if one of those values of $j$ fit the above conditions.

The recursive approach is:

```
def text(S, i):
    # Considers the substring S[0:i+1].

    # Base Case: i < 0
    # Here S[0:i+1] is empty, so trivially is a sentence!
    if i < 0:
        return True

    # Set a running tally to False initially.
    isSentence = False

    # Loop over the possible j values.
    # Loops from i (inclusive) to -1 (exclusive) taking steps of size -1.
    for j in range(i, -1, -1):
        # Check if S[j:i+1] is a word.
        if dict(S[j:i+1]):
            # Check if S[0:j] is a sentence.
            if text(S, j-1):
                isSentence = True
                break # Not needed, but will slightly improve runtime.

    return isSentence
```

For our DP approach, we can store this in a 1D table and fill from left to right. The base case is that the 'entry' for $-1$ is True, i.e., if we ever ask for the entry at $-1$, we will instead automatically give True. From there, filling each entry will require a linear scan of the previous entries. Note that this still takes $O(n^2)$ time, but only requires $O(n)$ space instead of the 2D table.

To reconstruct the solution, we can simply start with $i$ at the right end of the table and scan backwards until we find a $j$ such that `S[j:i+1]` is a word, and the table entry at location $j$ is True.