COMP 4030/6030 -- Assignment 8

**Due: Tuesday 11/14/2017**

**This is a coding assignment. Mail your work to the TA:**

* The name your solution file should be the same as your UID, plus a .py extension. For example, if your UID is jsmith (i.e. your email is [jsmith@memphis.edu)](mailto:jsmith@memphis.edu)), then your solution file should be **jsmith.py**.
* In the file, put your full name, COMP 4030 or COMP 6030, and Assignment 8.
* Send your solution to the TA (Quang Tran, [qmtran@memphis.edu)](mailto:qmtran@memphis.edu)) with the subject line “**COMP 4030 Assignment 8**”.

The problems that you’ll work on in this assignment has to do with restoring texts after spaces in the texts were removed. In the simplest model, you are given a text like this: **thisisacat** and a dictionary like this **{ this, that, is, are, a, cat, dog }**. You will need to determine if the given text can be decomposed into a sequence of words in the dictionary. (We are not concerned with grammar in this model. And the answer depends on the content of the dictionary as well.)

In the example above, the answer is Yes, *the text is restorable*. However, if the text is **thesearecats**, then the answer is No, it is not restorable, given the same dictionary.

Problems like these are hard to solve by brute force because there are too many ways of decomposing (i.e. breaking into words separated by spaces) a text. We can use dynamic programming to solve this problem.

1. (40 Points) Implement the following strategy in Python: Find the first word (starting from the end of the input text) that is in the dictionary, and repeat the process.

For example, given “**thisisacat”**, and the same dictionary as above, your algorithm will start from the end of the text and find “**cat**” in the dictionary. Then, it will repeat the same process with **“thisisa”**. It will find “**a**”. And it will continue like that.

In other words, the logic of this strategy can be expressed like this. Given a text **t**, if we can write **t** as **uv**, where **v** is a word in the dictionary. Then **t** is restorable only if **u** is *restorable*. If **u** is not restorable, then **t** is not restorable either.

**Hint**: to determine if the input string is restorable, run an index **i** from n-1 to 0. Define **v** as the slice from **i** to the end of the string, and **u** as the slice from 0 to **i**. If **v** is in the dictionary, use the same strategy to determine if **u** is restorable. If the text is empty, you should return True. When do you return False?

1. (10 Points) Show an example of an input text for which the previous algorithm does not work correctly.
2. (10 Points) Write the running time equation of the function in problem 1. What is the running time in terms of Theta?
3. (40 Points) We will revise the strategy described in problem 1. The revised logic is this. If **t** is restorable, then it can be written as **uv**, where **v** is a word in the dictionary,and the text **u** is also restorable.

There is a subtle difference between this logic and the previous logic (described in problem 1). The previous logic does not consider all possibilities; it considers only one possibility.

The new logic in this problem suggests there are multiple possibilities. Therefore, the correct strategy must consider all possibilities.

Write a **recursive** Python function that implements this new logic.

**Hint:** study the hint in problem 1. How do you consider all possibilities, when one of them fail?

1. (40 points) In this problem, you will write an **iterative** Python function to solve this problem. You can start with this code:

**def is\_restorable(text, D):**

**R = [False] \* len(text)**

**for i in range(0, len(text)):**

**for j in range(?, ?):**

**u = ?**

**v = ?**

**# additional code here**

Actually, the logic of this iterative function is exactly like the logic described in question 4: If **text** is restorable, then it can be written as **uv**, where **v** is a word in the dictionary,and **u** is also restorable.

R has the following meaning: **R[i]** is true if **text[0: i+1]** is restorable. We initialize R to be all False. But your code should correctly compute **R[i]**.

Assuming **R** is correctly computed, the final answer we need is **R[ len(text) ].** Your function should return True or False.

In the code above, you will compute **R[i]**, for each value of i, iteratively.

You will compute **R[i]** by considering all possible ways of decomposing **text[0: i+1]** at all possible breaking points **j**.

You will need to figure out how to get the correct values of **u** and **v**. These variables **u** and **v** have the same meanings as in problem 4.