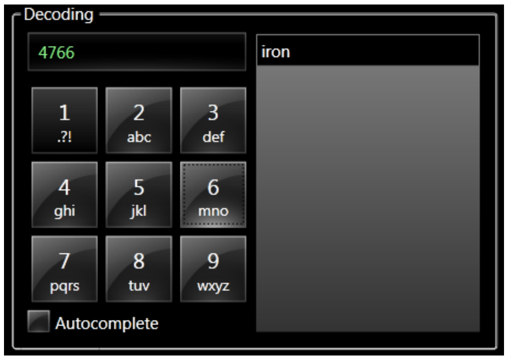
T9 in Python



Please read up a bit on the Web about the T9 technology. You may visit the following to start with: T9 Wikipedia page

Basically T9 technology belongs to the era when there were no smartphones with QWERT keyboards. So, SMSs needed to be typed using a numeric keypad. The traditional approach to typing messages was to press each key in quick succession the required number of times to get the desired character on the screen. For example, to type 'f', you need to press '3' thrice. To type a word 'key', the following sequence of keypresses is needed:

1. 5 twice
2. 3 twice
3. 9 thrice

A three letter word, thus, needed 7 keystrokes above. This was tedious; T9 helped. For example, with T9 enabled, you can type 'key' by just pressing 5 once, 3 once and 9 once. The catch is that there could be other words corresponding to this sequence of keypresses. For example, to type 'jew', you have the same sequence. Further, the above sequence could correspond to a prefix to a longer word. For example, if you wish to type 'lewd', the first three keypresses are the same as above. Therefore, T9 has to always keep track of the keypresses, and maintain a list of words that sequence corresponds to. Additionally, it has to designate one of those combinations as the current word, but should allow the user to choose another combination as the current word if desired.

Your objective is to simulate the working of T9 system using object-oriented concepts. We have provided two identical implementations of T9 (in files `t9slow.py` and `t9fast.py`) with the following features:

- Attributes:
    1. `keypresses`: A list of keys which correspond to the current set of prefixes
    2. `currentIndex`: The index of the the current prefix in the `currentPrefixes` list

3. currentPrefixes : The list of prefixes keypresses corresponds to

- Methods:

  1. inputKey(k): Given a key k, this method updates the instance attributes as per the value of k. The values that k may have are: numbers 2 to 9; `'r'` causes T9 to reset; `'b'` means backspace which deletes the last character in the keypresses; `'n'` makes the next prefix in the currentPrefixes the current word; and `'p'` makes the previous prefix in the currentPrefixes the current word;
  2. inputString (k)

The above classes make use of another class Dictionary (as its attribute dictionary) which represents the English language dictionary. t9slow uses dictionary1 while t9fast uses dictionary2 . The dictionary initialises by reading words off the dictionary file linux .words.txt provided herewith. The interface of the Dictionary class is as follows:

Feel free to implement additional attributes and methods if required.

- Attributes: As required

- Methods:

  1. findPrefix : Given a word w, this function tells if w exists in the dictionary.

Given: Class dictionary1 . Dictionary (dictionary1 .py)

In the first implementation of Dictionary class (named dictionary1 . Dictionary) is implemented using a simple list of words. The method findPrefix is merely a linear search through the word-list.

To Do: dictionary2 . Dictionary (dictionary2 .py)

Convince yourself that the above implementation dictionary1 . Dictionary is extremely inefficient. It wastes space in storing the words in a list, and it wastes enormous amount of time in searching for them in the list. In fact, you may find it to be so slow as to be unusable in practice. dictionary2 . Dictionary works towards removing this inefficiency. For this, dictionary2 . Dictionary uses the concept of suffix trees.

Suffix Tree

A suffix tree is an efficient data-structure for storing a set of words which may have common prefixes. It is also very efficient in searching for words. Hence, it is an ideal data-structure for use in our dictionary implementation. Figure 1 shows a suffix tree for a limited dictionary with the words: BACK, BAN, BIN, BINARY, BAT, BANTER, BRACKET and BRAG. Each word is represented by the nodes along a path from root node to a leaf (labelled with a $). While every path from the root to a leaf represents a complete word in the dictionary, every path from the root to any node represents a prefix of a work in it. If two words have a common prefix, the two words are represented by a common sequence of nodes till the last common letter, after which they bifurcate. For example, BACK, BAN, BAT and BANTER have a common prefix BA. So, they share their

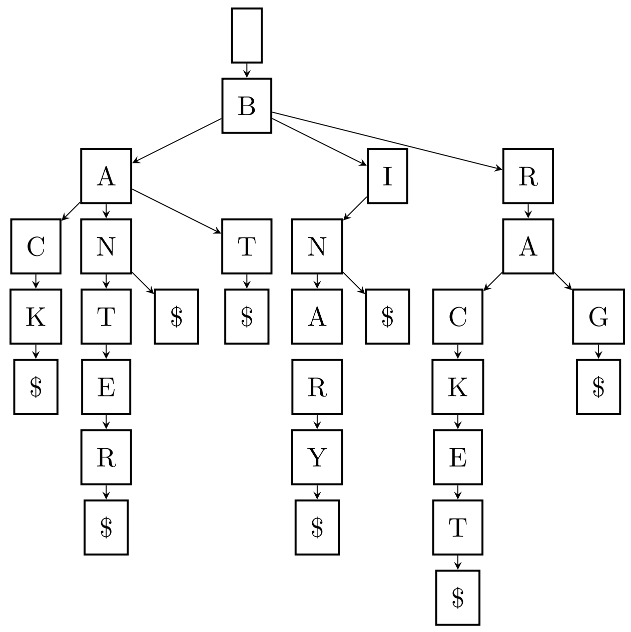


Figure 1: A suffix tree for a dictionary containing the words BACK, BAN, BIN, BINARY, BAT, BANTER, BRACKET, BRAG

initial two nodes. On the other hand, BAN AND BANTER share their first three letters, namely BAN.

In the current application, the distinction between incomplete prefixes and complete words is not important. Therefore, it is not necessary to have the special node with $ label.

Read more about suffix trees here.

`dictionary2.Dictionary` reads the provided dictionary file and loads it into its internal suffix tree. Once loaded, the suffix tree can be used very efficiently to search for words.

Note that `dictionary2.Dictionary` has an identical interface as `dictionary1.Dictionary`. However, it is very much more efficient.

You are expected to modify the implementation of `dictionary2.Dictionary` using a suffix tree. Please write your code only in `dictionary2.py` file. **Do not edit any of the other files provided.** You may add additional files (e.g. to implement the suffix tree) or you may implement all your code in `dictionary2.py`. It is your choice. If you execute the test file `test.py` on the command line, you should see all `True`s getting printed. Any other output indicates that your implementation is not correct. However, `test.py` is just a *smoke test harness*: it just checks that the basic functionality is correctly implemented. In order to come up with a working implementation, you must additionally test your code with additional test cases.

Submission

Submission date is **December 5, 2014 (Friday)**.

Assessment

You will work in teams which will be shortly announced by the TAs. On the submission day, you will be required to demonstrate your program as a team. There will be collective credits for getting the code to successfully run on test cases provided by the TAs. The TAs will quiz each team member on their understanding of the code and its underlying concepts. Individual credits will be given for that.