Homework 4, CSE 232

Due September 24 Note: On most of the problem sets through the semester, I'll put a horizontal line with "Optional" under it. Any problems below this section are encouraged - I think they're interesting and will help you learn the subject - but not necessary to complete in order to get credit for the homework.

Problem 1

This week's problem is GCJ 1B 2013 - Garbled Email.

Here's the problem statement:

> Gagan just got an email from her friend Jorge. The email contains important information, but unfortunately it was corrupted when it was sent: all of the spaces are missing, and after the removal of the spaces, some of the letters have been changed to other letters! All Gagan has now is a string S of lower-case characters.
>
> You know that the email was originally made out of words from the dictionary described below. You also know the letters were changed after the spaces were removed, and that the difference between the indices of any two letter changes is not less than 5. So for example, the string "code jam" could have become "codejam", "dodejbm", "zodejan" or "cidejab", but not "kodezam" (because the distance between the indices of the "k" change and the "z" change is only 4).
>
> What is the minimum number of letters that could have been changed?

Dictionary

> In order to solve this problem, you'll need an extra file: a special dictionary that you can find at https://code.google.com/codejam/contest/static/garbled_email_dictionary.txt. It is not a dictionary from any natural language, though it does contain some English words. Each line of the dictionary contains one word. The dictionary file should be 3844492 bytes in size, contain 521196 words, start with the word "a", and end with the word "zymuznh".
>
> When you're submitting the code you used to solve this problem, you shouldn't include the dictionary. As usual, however, you must submit all code you used to solve the problem.
>
> Note that if you are using Windows and want to look at the dictionary file, you should avoid Notepad, and instead use WordPad or another piece of software, or else all the words might appear on the same line.

Input

> The first line of the input gives the number of test cases, T. T test cases follow. Each test case consists of a single line containing a string S, consisting of lower-case characters a-z.

Output

> For each test case, output one line containing "Case #x: y", where x is the case number (starting from 1) and y is the minimum number of letters that could have been changed in order to make S.

Limits

> S is valid: it is possible to make it using the method described above.

Small dataset

> $1 \leq T \leq 20$.
>
> $1 \leq$ length of S ≤ 50.

Large dataset

1 ≤ T ≤ 4.

1 ≤ length of S ≤ 4000.

In class today, we talked about how to solve a related problem: given S, determine whether it can be constructed by concatenating together words from our dictionary. Let's solve an instance of this related problem. Given the word list
`a`, `aac`, `acb`, `baa`, `bbc`, `bc`, `caba`, `caab`
and a test string $S =$ `aacaabbc`, we first want to construct a trie containing all the words in our word list.

a) How many terminator nodes will our trie contain? What will be the children of the root node?

A trie contains a terminator node for each word in the input word list, so it will have eight total terminator nodes. The children of the root node will be the first characters of the words in our list, `a`, `b`, and `c`.

b) Write a function that takes the root node of a trie and a new word and inserts the word into the trie. Don't forget the terminator nodes. Use this function to write code constructing the trie containing the word list given above. Attach your code to the homework.

```
In []: TERMINATOR = ''

       def add_branch(tree, word):
           for char in word:
               if not char in tree:
                   tree[char] = {}
               tree = tree[char]
           tree[TERMINATOR] = True

       tree = {}
       wordlist = ['a', 'aac', 'acb', 'baa', 'bbc', 'bc', 'caba', 'caab']
       for word in wordlist:
           add_branch(tree, word)
```

c) We talked about how to solve this problem in class using dynamic programming to fill in an array of booleans the length of S denoting whether each suffix of S can be constructed from words in our dictionary. Construct this array by hand for $S =$ `aacaabbc` *and give your justification for each choice.*

`[T, T, T, T, T, T, T, F]`
From the end to the beginning:
The last suffix, `c`, is not a word, and so is marked as False.
The next suffix, `bc`, is a word, and is marked as True.
Similarly for `bbc`.
`abbc` is not a word, but can be split into the words `a` and `bbc`, and so is True.
`aabbc` can similarly be split into `a`, `a`, and `bbc`, and so is True.
`caabbc` can be split into `caab` and `bc`, and so is True.
`acaabbc` can be split into `a`, `caab`, and `bc`, and so is True.
`aacaabbc`, the full string, can then be split into `a`, `a`, `caab`, and `bc`, and so is True.

d) Suppose that instead we asked how many characters of S must be changed in order for S to be composed of words in our dictionary. Construct a dynamic programming solution to this problem by replacing our array of booleans as in part c) with an array of integers denoting how many characters of each suffix of S must be changed in order for that suffix to be composed of words in our dictionary. Test your code on the sample input and output below. Attach both your code and the output of your code on the test input to your homework.

Sample Input: abbbcbbc.

Correct Output: [1, 1, 0, 0, 1, 0, 0, 1]

```
In [ ]: import sys
        import numpy as np

        #Construct the trie
        TERMINATOR = ''

        def add_branch(tree, word):
            for char in word:
                if not char in tree:
                    tree[char] = {}
                tree = tree[char]
            tree[TERMINATOR] = True

        tree = {}
        wordlist = ['a', 'aac', 'acb', 'baa', 'bbc', 'bc', 'caba', 'caab']
        for word in wordlist:
            add_branch(tree, word)

        S = "abbbcbbc"
        MAXVAL = len(S)

        def match_words(prefix, index):
            suffix = S[index:]
            branch = tree
            for char in prefix:
                branch = branch[char]

            if suffix == '':
                if TERMINATOR in branch:
                    return 0
                else:
                    return MAXVAL

            best = MAXVAL

            char = suffix[0]
            for nextchar in branch:
                #Skip terminating branches
                if nextchar == TERMINATOR:
                    continue
                #Check if this is a mismatch
                mismatch = 1
                if nextchar == char:
                    mismatch = 0
                #If we end a word here, check the DP array and increment if this is a mismatch
                if TERMINATOR in branch[nextchar]:
                    best = min(best, mismatch + min_swaps[index + 1])
                #And then increment our prefix by this character and recurse to the next position
                best = min(best, mismatch + match_words(prefix + nextchar, index + 1))

            return best
```

```
    #Initialize the DP array
    min_swaps = np.zeros(len(S)+1, dtype=int)

    for i in range(len(S))[::-1]:
        min_swaps[i] = match_words('', i)

    print wordlist
    print S
    print min_swaps[:-1]
```

Optional

e) The Google Code Jam problem is very similar to part d) but has a constraint on how closely garbled characters are allowed to be - there must be at least four non-garbled characters between any two garbled characters. Construct a dynamic programming solution to the GCJ problem using a two-dimensional array of integers denoting how many characters must be garbled to construct the i^{th} suffix of S from words in the dictionary if the first j characters of the suffix are not allowed to be garbled. Test your code on the Small and Large input sets and check that it works.

```
In []: import sys
       import numpy as np

       #First, load the dictionary into a trie
       tree = {}
       END = ''

       def add_branch(tree, word):
           for char in word:
               if not char in tree:
                   tree[char] = {}
               tree = tree[char]
           tree[END] = True

       infile = open('./garbled_email_dictionary.txt')
       for line in infile.readlines():
           add_branch(tree, line.strip())
       infile.close()


       def match_words(prefix, index, required):
           suffix = S[index:]

           #Walk down the tree to find the right branch
           branch = tree
           for char in prefix:
               branch = branch[char]

           if suffix == '':
               if END in branch:
                   return 0
               else:
                   return -1
```

```
    def check(best, new, inc=0):
        if new == -1:
            return best
        if best is None:
            best = new + inc
        else:
            best = min(best, new + inc)
        return best

    best = None

    char = suffix[0]
    if char in branch: #valid match
        if END in branch[char]: #end of a word
            best = check(best, mincosts[index + 1, max(0, required - 1)])
        best = check(best, match_words(prefix + char, index + 1, max(0, required - 1)))
    if required == 0: #We're allowed to get this character wrong!
        for this in branch:
            if this == END or this == char:
                continue
            if END in branch[this]:
                best = check(best, mincosts[index + 1, 4], 1)
            if best is None or best > 1: #Don't bother checking if best is already good enough
                best = check(best, match_words(prefix + this, index + 1, 4), 1)

    if best is None:
        return -1
    else:
        return best

#Start input/output
infile = open("%s" % sys.argv[1], 'r')
outfile = open("%s.out" % sys.argv[1][:-3], 'w')
cases = int(infile.readline().strip('\n'))
for i in range(cases):
    S = infile.readline().strip()

    #Initialize our DP cost matrix and fill it in in the correct order
    mincosts = np.zeros((len(S) + 1, 5), dtype=int)

    for index in range(len(S))[::-1]:
        for required in range(5):
            mincosts[index, required] = match_words('', index, required)

    outfile.write('Case #%i: %s\n' % (i+1, mincosts[0,0]))
    print 'Case #%i: %s' % (i+1, mincosts[0,0])
infile.close()
outfile.close()
```