CMPT 103 – Lab #7

General Information

Python version and IDE:	Python 3.3 / Wing IDE 101
Allocated lab time:	2 hrs and 50 min
Due date:	At the end of the lab period
Lab weight:	3%

Topics

- ✓ File I/O and dictionaries

Submission

- ✓ **All code file (.py) should be submitted electronically** to your Lab Blackboard site.
- ✓ A portion of the total marks (20%) will be allocated for the programming style. For example, functions should be small; avoid writing duplicate code; names should be meaningful and descriptive; naming conventions should be followed consistently; code should be formatted properly; and comments should be accurate and well written.
- ✓ Comments are **required** for:
  - EACH program indicating the student name and program name.
  - EACH function indicating the function purpose, syntax (example usage of the function), parameters, and return value
  - Any block of code for which the purpose may be unclear (Note: you should always try to write clean code that can be understood easily without comments).

Assignment

For this lab, please put all functions into a file called `Lab7your_initials.py` (e.g., `Lab7FL.py` where F and L are the first letter of your first name and last name). Please feel free to write helper functions if necessary.

1. [20 marks] Write a function named `intersect` that takes two dictionaries and that returns a new dictionary containing only the key/value pairs that exist in both of the given dictionaries. Note that the order of key/value pairs in a dictionary does not matter.

```
>>> dict1 = {'Janet': 87, 'Logan': 62, 'Jeff': 88, 'Sylvia': 95}
>>> dict2 = {'Logan': 62, 'Sylvia': 87, 'Brian': 60, 'Jeff': 88, 'Kim': 83}
>>> intersection = intersect(dict1, dict2)
>>> print(intersection)
{'Jeff': 88, 'Logan': 62}
```

2. [20 marks] Write a function named `is_submap` that takes two dictionaries and that returns `True` if every key in the first dictionary is also contained in the second dictionary and maps to the same value in the second map. The empty dictionary is a submap of any dictionary.

```
>>> is_submap({}, {'Smith': '765-9382', 'Marty': '637-8274'})
    True
>>> is_submap({'Marty': '637-8274'}, {'Smith': '765-9382', 'Marty': '637-8274'})
    True
>>> is_submap({'Marty': '637-8274'}, {'Smith': '765-9382', 'Marty': '777-8274'})
    False
>>> is_submap({'Marty': '637-8274', 'Smith': '765-9382'}, {'Marty': '637-8274'})
    False
```

3. [60 marks] The edit distance between two strings is the minimum number of operations that are needed to transform one string into the other. For this question, an operation is a substitution of a single character, such as from “brisk” to “brick” or from “dog” to “log”. The edit distance between the words “dog” and “cat” is three, following the chain of “dot”, “cot”, and “cat” to transform “dog” into “cat”. When you compute the edit distance between two words, each intermediate word must be an actual valid word. Edit distances are useful in applications that need to determine how similar two strings are, such as spelling checkers.

   In this question, you’re going to implement two helper functions that can be used towards computing edit distances:

   a. [20 marks] Write a function named `is_neighbour` that takes two strings and that returns `True` if the given strings are immediate neighbour (e.g., “brisk” and “brick”), or `False` otherwise. To be immediate neighbour, strings must have the same length.

```
>>> is_neighbour('dog', 'log')
    True
>>> is_neighbour('brisk', 'brick')
    True
>>> is_neighbour('dog', 'dogs')
    False
>>> is_neighbour('dog', 'cat')
    False
```

   b. [40 marks] Write a function named `build_neighbour_map` that takes a filename (i.e., the dictionary text file – “dict.txt”) and that returns a dictionary that maps every word in the file to its immediate neighbours (store these neighbours in a list). This function should return `None` if there is an error while processing the dictionary text file. Use the `is_neighbour` function above to implement this function.

```
>>> map = build_neighbour_map('dict-test.txt')
>>> len(map)
    25
>>> map['search']
    ['starch']
>>> map['house']
    ['douse', 'horse', 'louse', 'mouse', 'rouse']
>>> map['humble']
    ['bumble', 'fumble', 'jumble', 'mumble', 'rumble', 'tumble']
```

Notes: For this lab, you don't have to implement fully a program that can compute edit distances. However, for informational purpose, here is a description of how to do it. Once a neighbour map is built (by calling `build_neighbour_map` with a filename of the dictionary text file), you can walk it to find paths from one word to another. A good way to process paths to walk the neighbour map is to use a list of words to visit, starting with the beginning word, such as "dog". Your algorithm should repeatedly remove the front word of the list and add all of its neighbours to the end of the list, until the ending word (such as "cat") is found or until the list becomes empty, which indicates that no path exists between the two words.

Acknowledgements: The lab specification is adapted from Building Java Programs (2nd Edition) by S. Reges and M. Stepp.