**COMP 204 – Assignment #3**

**Due date: November 4, 23:59**

- Submit *one* Python program on MyCourses, which should contain all your functions. The file should be called medical_diagnostic.py
- Write your name and student ID at the top of the program
- For each question, complete the relevant function(s). **Do not change the functions' name or arguments.** You can create additional functions if needed.
- Do not use any modules or any pieces of code you did not write yourself.
- For each question, your function will be automatically tested on a variety of test cases, which will account for 75% of the mark.
- For each question, 25% of the mark will be assigned by the TA based on (i) Re-use of functions as appropriate; (ii) appropriate naming of variables; (iii) commenting of your program; (iv) simplicity of your program (simpler = better).
- **Important notes:** This assignment focuses a lot on software re-use. You will be writing several functions. Some of those functions should call other functions in order to re-use the code already written. Make sure to re-use your functions as much as needed!

**Background:** Your task in this assignment is to write a program that will help doctors make disease diagnostics based on a patient's symptom. This will be achieved by comparing a new patient's symptoms to a database of patients containing both their symptoms and their diagnostic. Given a patient X with a certain set of symptoms, your diagnostic help to the doctor will be obtained by identifying, among the patients in the database, those whose symptoms are most similar to those of patient X. This is actually a commonly used approach in artificial intelligence, called k-nearest neighbors classification, although everything you need to know about k-nearest neighbors classification is contained in this assignment.

**Download medical_diagnostic.py from MyCourses.**
The file contains several functions that you will need to complete. It also contains the my_test() function, which calls each of the functions you will write. The expected (correct) output for several examples is given in the file **my_test_output.txt**. *Use this to make sure that you understand what every function is expected to do and to test your own code*. Note: You will not be able to run the my_test() function before you complete all the functions it calls. In order for you to test your functions one at a time, comment out the portions of the my_test() function that call functions you have not yet written.

**Data representation:**
- Each patient is identified using an integer identifier (e.g. 56374).
- Each symptom is a string such as "headache" or "fever".
- Each diagnostic is a string such as "cold" or "meningitis"
- Symptoms for a given patient are stored in a tuple of two sets: the first set contains the symptoms that are present in the patient; the second set contains the symptoms that are observed *not* to be present (i.e. absent).
    - For example, a patient with coughing, runny nose, and sneezing, but no headache and no fever would be represented as:
      ({"coughing", "runny_nose", "sneezing"}, {"headache","fever"})

- Note that for a given patient, information may be missing about whether or not a patient has a symptom. For example, in the example above we don't know if the patient has a sore throat.

- The set of symptoms of all the patients in our database is represented using a dictionary, whose keys are the patient identifiers, and values are the tuples of symptoms. For example:

```
all_patients_symptoms = {
   45437: ({"coughing", "runny_nose"},{"headache","fever"}),
   16372: ({"coughing", "sore_throat"},{"fever"}),
   54324: ({"vomiting", "coughing","stomach_pain"},{"fever"}),
   35249: ({"sore_throat", "coughing","fever"},{"stomach_pain", "runny_nose"}),
   74821: ({"vomiting", "fever"},{"headache"}),
   94231: ({"stomach_pain","fever","sore_throat","coughing","headache"},{"vomiting"})
 }
```

- The diagnostic given to each patient in our database is stored in another dictionary, with keys equal to the patient identifiers and values corresponding to a string. For example:

```
all_patients_diagnostics= {45437: "cold", 56374:"meningitis", 54324:"food poisoning",
                           16372:"cold", 35249:"pharyngitis", 44274:"meningitis",
                           74821:"food poisoning", 94231:"unknown"}
```

**Good to know before you start:**

- Suppose you have a list of tuples and you want to sort the list based on the elements at a particular index of the tuple. Example: sort a list of tuples based on the values at index 1 of the tuples.

  [ (""A",8), ("B",4),("C",7),("D",2),("E",3) ] **->** [ (""D",2), ("E",3),("B",4),("C",7),("A",8) ]

  The List type has a sort() function, but how to tell the sort function what element of the tuple to look at? First, define a small function that takes an as argument a tuple and returns element at index 1 of the tuple:

```
def get_key1(item):
    return item[1]
```

  Then, tell the sort() function to use the get_key1() function to select the keys to base the sorting upon. This is done by passing the get_key1 function as a keyword argument to the sort function. (Yes, functions can be passed as arguments to other functions!).

```
someList.sort(key=get_key1)
```

  To sort in reverse order:

```
someList.sort(key=get_key1, reverse=True)
```

Note: The medical_diagnostic.py file already contains the get_key1 function. It is slightly different from the one presented here, for a reason that is a bit complicated to explain (essentially, it allows us to break ties in a deterministic fashion). Please use that function when sorting, and do not change it.

**Question 1 (10 Points)**
Complete the symptom_similarity() function, which measures the similarity between the symptoms of two patients. See below for an explanation of how the similarity is computed, and see my_test() function for examples.

```
def symptom_similarity(symptoms_tuple_A, symptoms_tuple_B):
        """
        Args:
                symptoms1: tuple of a set of symptoms present and a set of symptoms absent
                symptoms2: tuple of a set of symptoms present and a set symptoms absent
        Returns:
                present_present + absent_absent - present_absent - absent_present
                where   common_present is the number of symptoms present in both patients
                        absent_absent is the number of symptoms absent in both patients
                        present_absent is the number of symptoms present in patientA and absent in patientB
                        absent_present is the number of symptoms absent in patientA and present in patientB
                """
```

**Question 2 (10 Points)**
Complete the similarity_to_patients() function, which measure the similarity between a symptom tuple and the entire set of patients in our database. See below for an explanation of exactly what is expected, and see my_test() function for examples.

```
def similarity_to_patients(my_symptoms, all_patients):
    """
    Args:
        my_symptoms: tuple of symptoms present and absent
        all_patients: dictionary of patients IDs (key) and associated tuple of
                      present and absent symptoms
    Returns:
       List of tuples. Each tuple is of the form: (patientID, similarity), with one tuple per patient
       in all_patients. For each patient in all_patients, similarity is the symptom similarity between
       my_symptoms and the patient's symptoms. The list should be sorted in decreasing order of
       similarity.
     """
```

**Question 3 (15 Points)**
Write the most_similar_patients() function, which identifies the patients that have symptoms that are the most similar to those of a new patient. See below for an explanation of exactly what is expected, and see my_test() function for examples.

```
def most_similar_patients(my_symptoms, all_patients, n_top):
    """
    Args:
        my_symptoms: tuple of a set of symptoms present and absent
        all_patients: dictionary of patients IDs (key) and associated tuple of
                      present and absent symptoms
        n_top: Maximum number of patients to return
    Returns:
        The set of up to n_top patient IDs from all_patients
        with the highest similarity to my_symptoms
    """
```

**Question 4 (15 Points)**
Write the count_diagnostics() function, which counts, among a set of patients, what fraction exhibit each possible diagnostic. See below for an explanation of exactly what is expected, and see my_test() function for examples.

```
def count_diagnostics(patient_set, diagnostic_by_patient):
    """
    Args:
        patient_set: A set of patient IDs
        diagnostic_by_patient: A dictionary with key = patient_ID and values = diseases
    Returns:
        A dictionary with keys = diagnostic and
        values = fraction of patients in patient_set with that diagnostic
    """
```

**Question 5 (10 Points)**
Write the diagnostics_from_symptoms () function, which assesses the probability of different diagnostics based on a patient's symptoms. It does so by using the most_similar_patients() and count_diagnostics() functions. See below for an explanation of exactly what is expected, and see my_test() function for examples.

```
def diagnostics_from_symptoms(my_symptoms, all_patients_symptoms,
                              all_patients_diagnostics, n_top):
    """
    Args:
        my_symptoms: tuple of symptoms present and absent
        all_patients_symptoms: dictionary of patients IDs (key) and associated symptoms
        all_patients_diagnostics: dictionary of patients IDs (key) and associated
                                  diagnostic
        n_top: Number of most similar patients to consider.
    Returns:
        A dictionary with keys = diagnostic and values = fraction of the n_top most
        similar patients with that diagnostic
  """
```

**Question 6 (15 Points)**
Write the pretty_print_diagnostics() function, which prints in a nicely formatted manner the frequency of diagnostics of the patients contained in the set it receives as argument. See below for an explanation of exactly what is expected, and see my_test() function for examples.

```
def pretty_print_diagnostics(diagnostic_freq):
    """
    Args:
        diagnostic_freq: A dictionary with key = diagnostic and value = frequency
    Returns:
        Nothing
    Prints:
        A table of possible diagnostics, sorted by frequency, expressed as percentages.
        Only diagnostics with non-zero percentages should be printed.
        If a diagnostic is longer than 10 characters, it should be truncated to 10
        characters.
        Frequencies should be expressed as percentages, rounded to the nearest percent.
    """
```

**Question 7 (25 Points)**

When the diagnostic for a patient is unclear based on the symptoms that are present/absent, a doctor may want to ask for additional information about the symptoms of the patient. For example, for a patient who has a headache but no stomach ache, the doctor may want to enquire whether the patient has a stiff neck, which would suggest he/she may have meningitis. But which symptom should the doctor ask about? They should be asking about the symptom for which the answer would be the most informative with respect to the diagnostic. Suppose that we define the clarity of a list of diagnostic frequencies as the probability of the most likely diagnostic. For example, the clarity of the diagnostic { "meningitis":0.8, "cold":0.1, "flu":0.1 } is 0.8. Suppose a patient has a certain set *P* of symptoms known to be present and a certain set *A* of symptoms known to be absent. Then, the value of asking the question "Do you have symptom *X?*" (where X is not already in *P* or *A*) is defined as

Value( $X$ ) = 0.5 * clarity ( diagnostics_from_symptoms ( ($P \cup \{ X \}$, $A$), all_patients_symptoms, all_patients_diagnostics, n_top ) ) +
0.5 * clarity ( diagnostics_from_symptoms ( $P$ , $A \cup \{ X \}$ ), all_patients_symptoms, all_patients_diagnostics, n_top ) ) )

In other words the value of asking about symptom *X* is the average of the clarity of the diagnostic we would obtain if the symptom X is present and the clarity of the diagnostic we would obtain is the symptom X is absent.

Write the recommend_symptom_to_test() function, that recommends the maximum value symptom to enquire about. See below for an explanation of exactly what is expected, and see my_test() function for examples.

```
def recommend_symptom_to_test(my_symptoms, all_patients_symptoms,
                              all_patients_diagnostics, n_top):
    """
    Args:
        my_symptoms: tuple of symptoms present and absent
        all_patients_symptoms: dictionary of patients IDs (key) and associated symptoms
        all_patients_diagnostics: dictionary of patients IDs (key) and associated
                                  diagnostic
        n_top: Number of most similar patients to consider.
    Returns:
        A string describing the best symptom to enquire about in order to clarify the
        diagnostic.
    Explanation:
        The best symptom to test for is one that:
        (i) has been tested at least once among the patients in all_patients_symptoms
        (ii) is not already in the new_patient_symptoms, and
        (iii) yields the maximum value (see text of question for definition of value).
  """
```

**Just for fun ( Nothing to submit for this).**

The approach you have implemented works best when the number of patients in the database is large. We have created a database of 1000 (fake) patients, with their symptoms and diagnostics. This is contained in the file medicalData.txt . We also provide you with the function read_data_from_file(), which will read the data contained in the file and build the symptoms and diagnostics dictionaries.

```
def read_data_from_file(filename):
    """
    args:
        filename: Name of file containing medical data
    Returns:
        Tuple of a dictionary of symptoms and a dictionary of diagnostics
    """
```

See our my_test() function to see the results of the different functions on this larger patient database. Compare those results to yours to ensure your functions work properly.