When working on this quiz, recall the rules stated on the Academic Integrity statement that you signed. You can download the **q8helper** project folder (available for Friday, on the **Weekly Schedule** link) in which to write/test/debug your code. Submit your completed **q8solution** modules (**81**, **82**, **83**) and **empirical.doc** online by Thursday, 11:30pm. I will post my solutions to EEE reachable via the **Solutions** link on Friday morning.

1a. (6 pts) Write a script that uses the **Performance** class to generate data that we can use to determine empirically the complexity class of the **closest_2d** recursive function defined in the **nearestneighbor.py** module. This function takes a list of **(x,y)** coordinates and finds the two that are closest together and returns the distance between them. Call the **evaluate** and **analyze** functions on an appropriately constructed **Performance** class object (repeating each timing 5 times) for lists of random coordinates (from sizes 100 to 25,600 doubling the size each time). Hint: Use the **random.random** function to generate each coordinate: it returns **float** values in the range **[0,1)**. See the file **sample8.pdf** (included in the download) for what your output should look like: of course your times will depend on the speed of your computer.

1b. (3 pts) Fill in part 1b of the **empirical.doc** document (included in the download; edit/submit this file) with the data that you collect (or use the data in **sample8.pdf** if you cannot get your code to produce the correct results) and draw a conclusion about the complexity class of the **closest_2d** function by examining the run-time ratios for all the size-doubled problems.

2a. (6 pts) Write a script that uses the **cProfile** module to profile all the functions called when the **closest_2d** function is run on a random list with 25,600 coordinates. Generate the random list, and then call **CProfile.run** so that it runs **closest_2d** on that list; also specify a second argument, which is the file to put the results in (and the file on which to call **pstats.Stats**) to print the results.

For the first output, print the top 15 results, sorted decreasing by **ncalls**; for the second output, print the top 15 results, sorted decreasing by **tottime**. Hint: The notes show how to instruct the profiler put the profile information into a file and then show how to access that file and format the results it displays to the console. See the file **sample8.pdf** (included in the download) for what your output should look like: of course your times will depend on the speed of your computer.

2b. (3 pts) Answer the questions in part 2b of the **empirical.doc** document (included in the download) with the data that you collect (or the data in **sample8.pdf** if you cannot get your code to produce the correct results).

1b and 2b: If possible, after editing empirical.doc for parts 1b and 2b, convert it into a .pdf document and submit the document in that format on checkmate. The TAs must be able to read your document on their computers, so don't submit it in any other format than a Word Document or a .pdf file.

3. (7 pts) Define a **Test_Bag** class, derived from **unittest.TestCase** class, which performs the following tests to verify some (not all) of the methods in the **Bag** class (I have provided my solution from programming assignment #2) work as expected. I wrote about 80 lines of code. Remember to run you code via **Run as** and the **Python unit-test**, which should show all tests passed. Write the setup and test functions in the order specified below. Read information at the top relating to the imported methods **random.shuffle** and **random.randint**.

a) Setup: store an attribute named **alist** storing **['d','a','b','d','c','b','d']** in this order. Store another attribute named **bag** that is a **Bag** constructed using **alist**. Note the bag contains one **'a'**, two **'b'**s, one **'c'**, and three **'d'**s.

Write one test method for each of the following tests. Each method might contain multiple asserts, or single asserts executed repeatedly in loops. Write each test using the names shown below (**equals** for **==**).

b) Test **len**: check that the **len** of the constructed bag is initially 7, and then remove one value from the bag at a time (remove the values based on a random order of **alist** used in the setup), checking that the **len** decreases by 1 after each removal, and eventually becomes 0. My code was about 8 lines.

c) Test **unique**: check that the bag initially contains 4 unique values, and then remove one value from the bag at a time (remove the values based on a random order of **alist** used in the setup), checking that the number of unique values is the same as the **len** of the **count** dictionary stored in the Bag. My code was about 5 lines.

d) Test **contains**: check that the bag initially contains **'a'**, **'b'**, **'c'**, and **'d'**, but not **'x'**. My code was about 3 lines.

e) Test **count**: check that the bag initially contains 1 **'a'**, 2 **'b'**s, 1 **'c'**, 3 **'d'**s, and 0 **'x'**s, and then remove one value from the bag at a time (remove the values based on a random order of the list used in the setup), checking that the **sum of the counts** of **'a'**, **'b'**, **'c'**, and **'d'** decreases by 1 after each removal, and eventually becomes 0. My code was about 10 lines.

f) Test **==**: Ignore the setup. Construct a bag initialized by a list of 1,000 random integers in the range 1-10; construct another bag initialized with the same 1,000 random numbers, but in a different random order: check that the two bags are considered equal. Remove the first value in the list of random integers from one bag (it is certainly in the bag) and check that the bags are no longer equal. My code was about 7 lines.

g) Test **add**: Ignore the setup. Construct a bag initialized by a list of 1,000 random integers in the range 1-10; construct another bag that starts empty, then **add** each of the 1,000 random numbers to the bag one at a time, but in a different random order: check that the two bags are equal. My code was about 7 lines.

h) Test **remove**: Ignore the setup. Construct a bag initialized by a list of 1,000 random integers all in the range 1-10; test that removing the value 62 raises the **ValueError** exception. Then, construct another bag initialized with the same 1,000 random numbers in the same order; then **add** each of these same 1,000 random numbers to the second bag in a different random order; then remove each of these 1,000 random number from the second bag in the same order they were added the second time; finally check that the two bags are equal. By adding values twice and then removing them once to the second bag, we should be back to the original bag contents. My code was about 10 lines.

You should rewrite some **Bag** methods to be incorrect, and see that some of your test methods fail.