COMP 1405B
Fall 2019 – Practice Problems #7

Objectives

- Continue to practice using the programming concepts we have covered already
- Practice writing code that uses dictionaries
- Practice designing more complex algorithms and implementing more complex programs
- Begin to think about runtime and space complexity of algorithms/implementations

---

Remember to test your solutions to verify their correctness and discuss your solutions with both other students and TAs in the course. Verifying your solutions with TAs will be a good way of ensuring that you are on a path to success in the course.

Problem 1 (Counting Letters - **)

Write a function called **lettercount** that takes a single string argument. This function should return a dictionary containing the frequency of each letter within the string.

Problem 2 (Dictionaries and Caching - **)

An interesting application of dictionaries is their use in caching. In Computer Science, a cache stores data so future requests for that data are performed faster (i.e., they are optimized). Factorial calculations are one thing that can benefit from caching. Note the definition of a factorial is:

$$n! = n * (n-1) * (n-2) * \ldots * 3 * 2 * 1$$

When you calculate the factorial in the order specified in this equation (multiplying from highest to lowest), you can stop early if you know the factorial value of a number between n and 1. For example, if we know the value of 5! is 120, then we can calculate the value of 7! two possible ways:

$$7! = 7 * 6 * 5 * 4 * 3 * 2 * 1$$
$$7! = 7 * 6 * 5! = 7 * 6 * 120$$

To start, create a new Python file and write a function called factorial(int) that calculates and returns the factorial value for a given integer. Create a new function called cachedfactorial(int, dictionary), which takes a number to calculate the factorial value for and a dictionary that stores the cached values. The cachedfactorial function should

calculate the factorial, but should update the cache whenever a new factorial value is calculated and use the cache to stop early if possible.

If you have time, evaluate the performance of each function using an approach similar to the one discussed in lecture. Use the time module to record the start time, calculate the factorial values from 1-5000 using either the cached or regular factorial function, then record the end time and print out the time difference. Does it make a difference if you go from 1-5000 or from 5000-1? Why? Try using the random integer function to calculate factorial values of numbers between 1-500 many times.

Problem 3 (Morse Code Conversion - *)

If you have ever wanted to communicate with the spirit of a mid-19th century telegraph operator, but just weren't sure how, now is your chance! Morse code is used to represent letters with long and/or short signals of light or sound (really though, the same strategy you use for this problem could be applied to any other code). In this problem, you will create a Morse code translator. To start, copy and paste the Morse code dictionary from the morse_code.txt file included in the PP7-Resources.zip into your Python code file. You can use this dictionary to "look up" the Morse code representation of any character. You then need to write the rest of the program, which should do the following:

1) Use input to read in a message from the user.
2) Iterate over each character in the message and add its Morse code representation to a result string (note: only upper case characters are included in the dictionary, so you may have to pre-process the string in some way)
3) Print out the result string once completed.

Test your program with various input strings. Some examples are included below:

Hello → .... . .-.. .-.. ---

Secret Message → ... . -.-. .-. . - -- . ...- --. .

Hold up the train → .... --- .-.. -.. ..- .--. - .... . - .-. .- .. -.

(Trivia bonus marks for knowing what significant historical disaster the last example is from! Trivia bonus marks are just like regular bonus marks, except they're worth nothing in the course…)

Problem 4 (JSON and Dictionaries - **)

The previous set of practice problems included a problem that involved the creation of a

JSON parsing module using lists. If you did not complete that problem, you can look at it for some details required for this problem. Now that we have discussed dictionaries, you should realize that, since the keys in JSON are strings, dictionaries are a more logical choice for storing JSON information. Recreate your JSON parsing module from the previous practice problems using dictionaries instead of lists. You should find the implementation is much easier.

Problem 5 (Text Analysis - ***)

Create a Python file called **analysis.py** that will perform text analysis on files. For this question, assume that each space (“ “) in the document separates one word from the next. You can also assume there is no punctuation or other symbols present in the files – only words separated by spaces. If you want to see examples of the type of text, look in the testfile_.txt files included in the PP7-Resources.txt file.

You must implement and test the following functions inside of your analysis.py file:

1) load(str) – Takes a single string argument, representing a filename. The program must open the file and parse the text inside. This function should initialize the variables (e.g., lists, dictionaries, other variables) you will use to store the information necessary to solve the remainder of the problems. These variables should be created outside the function so that they can be accessed by the other functions you will define. This way, the file contents can be parsed once and the functions below can be executed many times without re-reading the file, which is a relatively slow process. This function should also remove any information stored from a previous file when it is called (i.e., you start over every time load is called).
2) commonword(list) – Takes a single list-type argument which contains string values. The function should operate as follows:
    a. If the list is empty or none of the words specified in the list occur in the text that has been loaded, the function should return None.
    b. Otherwise, the function should return the word contained in the list that occurs most often in the loaded text - or any one of the most common, in the case of a tie.
3) commonpair(str) – Takes a single string argument, representing the first word. This function should return the word that most frequently followed the given argument word (or one of, in case of ties). If the argument word does not appear in the text at all, or is never followed by another word (i.e., is the last word in the file), this function should return None.
4) countall() – Returns the total number of words found in the text that has been loaded. That is, the word count of the document.

5) countunique() – Returns the number of *unique* words in the text that has been loaded. This is different than the previous function, as it should not count the same word more than once.

You can use the analysistester.py file from the PP7-Resources.zip, along with the text files, to test your functions.

Problem 6 (Minimizing Runtime - ***)

This problem involves runtime analysis. If we have not discussed the topic in lecture yet, it is fine to come back to this problem later. Dictionaries are very nice because they let us insert, remove and determine if an item is present in O(1) time (i.e., the time does not depend on the number of items in the dictionary). They do not, however, allow us to maintain the order that the items are added, which may be important in certain applications, like when time/age of data is involved such as the 'storing sensor data' problem from a previous set of practice problems. Lists allow us to insert/remove in O(1) time (it actually depends on how they are implemented, but we will assume this is true here) and maintain the order of inserted elements, but determining whether an element is in an unsorted list takes O(n) time.

For this question, you will implement a solution that uses more memory space (i.e., stores more data), but allows us to insert, remove, and determine if an item is present in constant time, while also maintaining the order of item insertion. To start, get the listandhash.py file from the PP7-Resources.zip file on cuLearn. This file has add and remove functions that modify a list variable by adding an element to the end or removing an element from the start (this is called a queue, which is a common data structure). The provided solution maintains the order of items as they are inserted, but the containslinear search function will run in O(n) time. You must modify this file by adding an additional function, called containshash(dict, value), that will determine if an item exists in the list in O(1) time. The original list-based functionality should be maintained. This will require you to store additional information in a dictionary. Additionally, you will have to add additional arguments to your function to pass the dictionary variable as input. Your final functions should look like: addend(list, dict, value), removestart(list, dict), containslinear(list, value), and containshash(dict, value). Try solving this problem yourself, but if you need additional hints, look at the end of the problem for hints or ask a TA.

If you want to test your implementation, you can copy/paste the code from the listandhash-test1.txt and listandhash-test2.txt files into your program. The first file will randomly add/remove some values before printing out the list and hash. This will allow you to verify that things seem to be working. The second file will perform significantly more operations and verify that your containshash function always returns the same

result as the containslinear function. It will also compare the search time using the linear function to the search time using your hash-based function.

The following hints are ordered so they will give you progressively more information. Try reading as few as possible for you to solve the problem. After reading each hint, think about how you could possibly solve the problem and try to work out a solution. If you can't, move on to the next hint.

Hint #1: Note that answering the question 'does value x exist in the list?' is equivalent to answering the question 'is the frequency of x in the list greater than 0?'.

Hint #2: To accomplish the goal of the problem, then, you can add a dictionary that stores the frequency of each item inside the list. In this case, the values stored in the list are the keys of the dictionary and the values of the dictionary are the frequencies of those key values in the list.

Hint #3: The frequency of an item only changes during insertion/removal, so you just need to update the item frequencies in the dictionary inside of these functions and add the improved containshash(dict, value) function.

Hint #4: The containshash function can look up the frequency value in the dictionary. Assuming the counts are updated correctly on insertion/removal, if an item exists in the dictionary and has a count greater than 0, the item is in the list. This is an O(1) time operation.

Problem 7 (Nesting Dictionaries - *****)

This is an interesting, albeit hard to explain, problem involving dictionaries. What this problem really does is implement something similar to a 'trie' data structure (https://en.wikipedia.org/wiki/Trie), which can be used to store string keys just like a dictionary. The trie structure has an added advantage in that it can easily find all keys that match a given prefix (e.g., the prefix "do" matches the keys "do", "dog", "dolt", "doing", etc.).

In this problem, you will develop a program that will store a large number of valid words and must be capable of checking if a given string is a valid word (e.g., like a spell-check program). An easy way to do this is to store all the words in a simple list and search through this list whenever you want to verify a given string is valid. This approach, however, will take up a large amount of memory and will take a relatively long time to determine whether a word is present or not, since you must search through a long list. An alternate approach would be storing the words in a dictionary, but this would also involve storing a larger amount of data than may be required with the approach implemented here.

Instead, you will implement a much more efficient solution. Consider the following words:

apple, application, apply, appetizer, apples, applies, applications

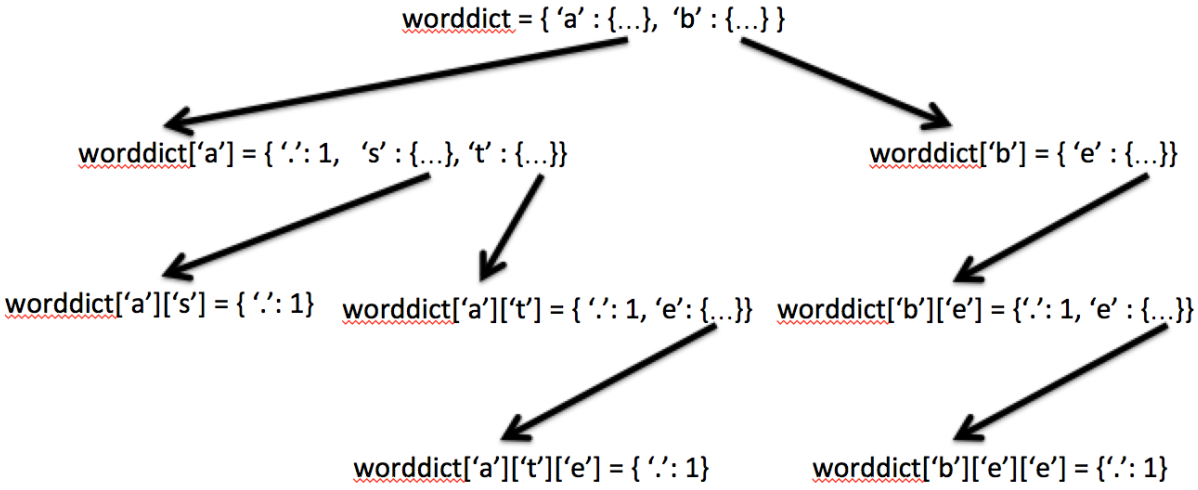
These words all share several of the same starting characters, which would be duplicated in a list or dictionary-based implementation of the program. You will remove this unnecessary reproduction of information by implementing a dictionary-based solution. Within your solution, there will be a single base dictionary, which will have keys representing characters of the alphabet. These keys will represent the first character of a word and the values associated with these keys will also be dictionaries. The values of this base dictionary, which are also dictionaries, can be used to store characters representing the second character of the word as keys with values that are dictionaries, and so on. If you want help getting started, check out the wordstorage-three.py file, which has a solution that works only for words of length 3. That file would need to be modified to use loops, so words of any size can be handled. Alternatively, if you are familiar with recursion, it is a great fit for this problem as well.

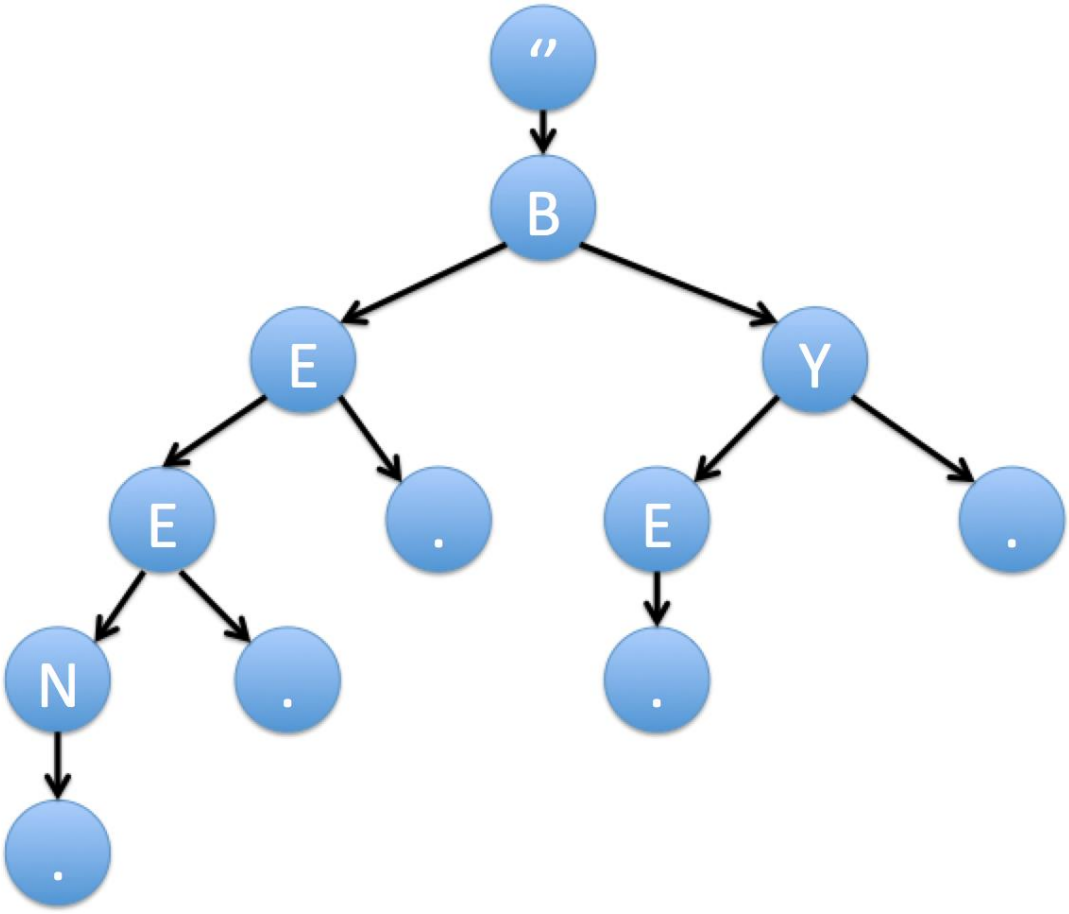
Write a program that has a single base dictionary variable to store words and the following three functions:
1) addword(dic, str): This function will add the word contained in the variable **str** to the storage dictionary **dic**. You will have to iterate over each character of the word and move deeper into the word storage dictionary (i.e., into the internal dictionaries associated with each character index), ensuring that the necessary key/values exist or are created. Note: to signify a word has ended, add an entry in the dictionary value associated with the final character of the word that has a key of '.' and a value of 1.
2) checkword(dic, str): Returns true if the word contained in the variable **str** is contained in the word storage dictionary **dic**. You can use a similar process to search for a word as you do when adding a word. If the necessary key does not exist for a certain character in the word, you know that the word has not been added (otherwise, you would have added that key during the add process). In addition, if you can find keys for each character in the word, and the key '.' exists in the final character's dictionary, you know this is a valid word. Otherwise, it is not a valid word.
3) initialize(dic, filename): Takes a string representing a filename. Each line in the specified file will contain a single word. This function will open the specified file and read each word, adding it to the dictionary **dic** using the addword function.

An example of the resulting dictionary structure after the words be, bee, been, by, and bye have been inserted, is shown below:

```
{'b': {'y': {'.': 1, 'e': {'.': 1}}, 'e': {'.': 1, 'e': {'.': 1, 'n': {'.': 1}}}}}
```

Two visual examples that may also help:





Problem 8 (Trivia Game Progressive Problem Part 3 - ***)

Now that we are familiar with dictionaries, we can modify the trivia game to utilize a webservice that provides questions in JSON format. This will allow you to request random questions from the webservice and include them in your game. Open Trivia DB (https://opentdb.com) is a free service that allows you to request trivia questions through a specific URL pattern. Information on how the URLs are formatted can be found at https://opentdb.com/api_config.php, or you can use the following URL to request 10 multiple choice questions https://opentdb.com/api.php?amount=10&type=multiple. Try opening the URL in the web browser first, you will see that it gives you back a JSON string. The PP7-Resources.zip file contains a webdev.py module. This module will allow you to call the readurl(string) function to request a specific URL and get a string value containing the contents of that URL. This way, you can request a question URL from Open Trivia DB, receive a string in response, and use that within your trivia game to generate questions.

Once you have the JSON string result from the URL request, you will need to convert it into a dictionary so you can use it in your program. There are 3 possible ways to do this listed below:

1. You can use the json module (import json) and then call the function json.loads(str). This function accepts a JSON string and gives you back a dictionary, assuming the string is formatted correctly.
2. Use Python's eval(str) function, which takes some string input and evaluates it as if it is Python code. Note: this function can have security issues – it would be worth reading https://www.programiz.com/python-programming/methods/built-in/eval if you plan to use it.
3. Create your own JSON processing module/functions. We did most of this in the previous practice problems, but the results in this case can also contain nested lists/arrays/dictionaries, which your code would need to be updated to handle. If you are feeling adventurous though…

Once you have converted the JSON string to a dictionary format, the result should always have the same format. The dictionary you receive will have two keys: "response_code" and "results". The response_code key is useful for ensuring the request was successful, but we can ignore it for this application. The "results" key has a value that contains a list of questions, each of which is represented by a dictionary. You can iterate over the list and extract each question's information from it. The dictionary used to represent each question has the following keys:

1. "category": a string representing the category of the question
2. "type": a string indicating whether the question is multiple choice or true/false
3. "difficulty": a string representing the difficult of the question

4. “question”: a string representing the question. Some questions have formatting characters inside them which may look funny, which you can try replacing/removing.
5. “correct_answer”: a string representing the correct answer
6. “incorrect_answers”: a list of strings with three incorrect answers (or 1 in the case of true/false questions)

Modify your trivia program to use questions that you extract from the OpenTriviaDB results. Each time you run the program or request the URL, you will get different questions back. You can convert your game to a multiple choice format or insist that the user types in the correct answer exactly.

Problem 9 (Search Engine Progressive Problem 3 - ****)

This problem involves runtime analysis. If we have not discussed the topic in lecture yet, it is fine to come back to this problem later. If you completed Part 2 of this problem in the previous practice problem sets, you should have a web crawler that can save all the pages and contents it finds to your local drive. If we consider runtime complexity, this design still has a lot of areas where we can maximize efficiency. For starters, let’s assume there are N pages, which may have K words in each. If you are given a set of search words and needed to go through all these N pages with K words to count how often the search word occurs, this would be an O(NK) operation. Additionally, the page information is stored in files currently, which are slow to read. While this doesn’t affect the number of operations we perform, it will certainly impact the time it takes for the operations to be completed. In this part, we will optimize the search procedure using dictionaries. Furthermore, we will implement some more analysis of the network structure to improve our page recommendations. To start, if you haven’t already, separate your search engine code into two separate parts: the web crawler (responsible for collecting the information) and the search engine (responsible for generating search results). This isn’t necessary but might help simplify the code a bit.

First, we will modify the web crawler to more efficiently store the data we find. Currently, we save all a page’s text into a file. What we are really interested in, though, is the number of times words appear on the page. So, change your code to create a dictionary for each page that has the words as keys and their frequency as values. Store this data into the file instead of all of the text (format is up to you). This should result in significantly smaller files and a more scalable solution. We can store signifcantly more pages, as they take less space.

Our web crawler and search engine currently use ONLY word frequencies when ranking pages. The word frequencies are useful for measuring the relatedness of the page to a set of search terms. If the search terms appear a lot, the page is related to what we are

searching for. Another component we may be interested in is the popularity of a page. There are many ways to define the popularity of a page, with the implementations having varying degrees of difficulty. A simple way we can measure a page's popularity, though, is to count the number of referrals to that page. In this case, a referral is simply another page that links to the page being considered. In this case, linking to a page is considered an endorsement of that page being good quality. Think about how you could implement this before reading the description that follows.

We want to remember a piece of information for each page (how many pages link to it). To do this, we can use a dictionary, with the page name (URL) being the key and the value being how many pages link to it. We also identify links in pages while crawling the web. If a page Y has a link to a page X, we need to increase the number of links that point to page X. Add code so that your web crawler records this information. You should also figure out a way to save it somewhere so you can use it in your search engine. You could add more information to the page's file, or create a separate file to remember this information. If you are interested in more complex methods of measuring page popularity/quality, you can Google the PageRank algorithm, which happens to be the algorithm that Google was founded around.

Once your web crawler has been updated, you now should have a more condensed version of each page's contents, as well as a measure of each page's popularity, stored somewhere in files. The size of the problem (i.e., the number of pages) that we are working on is small enough that we can load all this information into RAM without worrying about running out of memory. This will allow us to search through the page information significantly faster than reading the files. Modify your search engine to load all the page information from the files when the program starts. The information could be stored in many ways. If you may want to access the information about a specific page quickly, given the URL, it may be worth storing everything in a dictionary that uses the URLs as keys. If you will only ever iterate over all pages, it may be easier to store the same information in a list. Either way, for each page you will want to be able to retrieve the word frequency information (dictionary) and the popularity value (integer).

Once the search engine has loaded, it should repeatedly ask the user to enter a search request (a string, with words separated by spaces). Previously, the search engine would then search through all the files/contents. Now we will be able to find the same information using dictionaries stored in RAM. As we don't need to read all K words on each page anymore, this is now an O(N) operation instead of O(NK). Now, we will update the search engine algorithm to use both the relatedness measure (word frequencies) and the popularity measure (incoming links) to select the best page result.

Implement a function called similarity(page, words), which will accept a page name (URL) and a list of search words. This function should return the total number of times the search words appear on the given page. One issue with this basic implementation is that it may be biased toward pages with many words, even if most of them aren't related to the search words. Why? Make a small update to the code so that the similarity function returns the percent of all words on the page that match the search words. This is now less biased by the overall number of words. It will also give you a number that is between 0 and 1 (or 0 and 100, depending on whether you left the percentage as a decimal or not). Be careful how you implement this. You need to calculate the total number of words on the page. Should you calculate this every time you want to measure similarity? Or should you calculate it once, store that information, and recall it for each similarity computation? Which one would be more efficient?

Implement a second function called popularity(page). Like the similarity function above, this function will return a value between 0 and 1. In this case, calculate the number of referrals (incoming links) that this page has as a percentage of all referrals. Again, this will involve summing up all the referrals for all pages. It would be best to implement this in a way that doesn't require the sum to be executed each time. You could modify your web crawler to calculate the popularity in this way before saving the popularity value. Note that summing them each time would make the popularity function run in O(N) time, while having the percentage pre-computed would make it run in O(1) time. Customer satisfaction aside, if you are paying the power bills to run billions of queries, you would probably like to decrease the amount of computation.

Now that you can measure the popularity and relatedness of a given page, you can do so for all pages when a search is requested by the user. A simple way of combining the two measures is to add them together for each page and use that sum as a measure of the page's quality for this set of search terms. Remember the highest value that a page had and recommend that page to the user. Alternatively, try to remember the highest X pages and give the user a list to choose from. Another modification you can make is to weight the popularity and relatedness measures a certain way. Create two more variables (pop_weight and related_weight) and set their values so that they sum to 1 (0.5 and 0.5, 0.8 and 0.2, etc.). Multiply the relatedness/popularity result you get for each page by the appropriate weight variable before computing the sum. This now gives you an easy way to change the behaviour of your search engine to favor relatedness or popularity. You can also allow the user to specify these properties if you want.