CS417 Programming Assignment 4

- Due: Friday October 9th.
- Late penalty: Sat/Sun/Mon 5%, Tue 10%, Wed 20%, Thu 50%, Fri 100%.

Overview

For this assignment, you must write three programs.

- `letter_frequencies.py` analyzes a text file, and produces a table of letter frequencies.
- `make_words.py` uses the letter frequencies to produce "English-like" words.
- `search_times.py` will search for the words in a dictionary, and measures the search times.

Getting Started

Before you start, download the following data files:

- `A_Wasted_Day.txt` : a short story. Use in question 1 to find word frequencies
- `valid_words.txt` : a list of valid English words. Use in question 3 to check if a probe word is real.

Your Tasks

1. Write a program `letter_frequencies.py` which reads an ordinary text file, and prints a table of letter frequencies. The text file's name is a command-line argument.

   Call `print()` to output 26 lines of text. Each line should show a letter, and its frequency in the text (a number from 0 to 1.0). There should be 26 lines of output, like this (your numbers may be slightly different, but each line should have a letter, a space, and a number).

   ```
   a 0.07736986787266117
   b 0.012325973219827969
   c 0.023986875942183204
   d 0.0521415270018622
   e 0.12822559191274274
   ...
   ```

Details:

- Open the file, or complain if the command-line argument was absent, with a usage message:

```
usage: python letter_frequencies.py <filename>
```

- Create a list or `dict` of 26 counters, one for each letter in the alphabet.
- Read each line in the file, and convert upper case to lower case.
- Examine each character in the line, and ignore any letter that is not `a` to `z`. The module `string` includes the variable `string.ascii_lowercase`. Use it.
- Update the appropriate counter.
- When done, obtain a frequency (`0.0` to `1.0`) for each letter, and print it.

  To send your `print()` output into a file, use `>` on the command line to *redirect stdout*:

```
python letter_frequencies.py A_Wasted_Day.txt > freqs.txt
```

---

2. Write a program `make_words.py` which reads a list of 26 letter frequencies, and generates random 5-letter strings that have those frequencies. It expects these command-line arguments:
   - `sys.argv[1]`: a file where each line is a letter, a space, and a frequency.
   - `sys.argv[2]`: how many words to generate (an `int`).

   For each word generated, just `print()` it. You can send the output to a file:

```
python make_words freqs.txt 10000 > probe_words.txt
```

---

3. Write a program `search_times.py` which uses two files, whose names are command-line arguments:
   - `sys.argv[1]`: A file with N valid English words, one per line (`N` is 42869).
   - `sys.argv[2]`: A file with 5-letter strings (produced by `make_words.py`).

What your program should do:

- Measure how long, on average, it takes to search for one probe word in a subset of the valid words. The subset will have size `n`, for several values of `n`. You

are testing the following hypotheses:
    - if the subset is searched with linear search, the cost is proportional to `n`
    - if the subset is searched with binary search, the cost is proportional to log(`n`)

Details:

- Let `n` be in `[100,200,500,1000,2000,5000,10000,20000,N]` (`N` is 42869), and pick the first `n` valid words as your subset. Then, for each probe word, search for it in the subset.

- Store the words in a list, and simply search it using

  ```
  if word in subset:
  ```

  This does linear search.

- Sort the list, and *then* use the `bisect.bisect_left` method to do binary search. Count the time for searching, but **don't count** the sorting time. You will have to `import bisect`, of course.

  Alternatively, you can call your own `bsearch` function. See the sample code for an implementation.

- use `time.time()` to get the current (wall clock) time, before and after your searches. Make *TWO* calls to `time.time()`:
    - call it before `for word in probe_words:` loop, and
    - call it after the loop.

  Subtract the two times, to get the total search time, in seconds.

- divide the search time by `len(probe_words)`, to get the search time per word. This will be a small number. Multiply it by 1000000, to get the microseconds per search.

Required Output

Your program should print *four* numbers per line, separated by spaces. They are:

- `n`, the subset size

- `n_found`, the number of probe words that are valid

- `t_linsearch`, the average time of one linear search

- `t_bsearch`, the average time of one binary search

All times should be in microseconds per search. On my laptop, with 10,000 probe

words, I get numbers like these:

```
100 0 1.219797134399414 0.3392934799194336
200 0 2.4194955825805664 0.3712892532348633
500 0 6.033611297607422 0.5995035171508789
1000 0 24.963808059692383 0.5069971084594727
2000 2 24.272894859313965 0.5053997039794922
5000 2 61.2396001815796 0.6020069122314453
10000 6 126.40860080718994 0.5838871002197266
20000 10 255.22160530090335 1.8100976943969727
42869 20 584.6425294876099 0.6741046905517578
```

Turn in Your Work

When you are done, go to `mycourses.unh.edu`, find CS417, assignment 4, click "Submit", and upload the three files: `letter_frequencies.py`, `make_words.py`, and `search_times.py` .