Homework 9: Algorithm Design and Analysis

Due Thursday, December 12, 11:59 pm

Worth 20 points

Read this first. A few things to bring to your attention:

1. Start early! If you run into trouble installing things or importing packages, it's best to find those problems well in advance, not the night before your assignment is due when we cannot help you!
2. **Make sure you back up your work!** I recommend, at a minimum, doing your work in a Dropbox folder or, better yet, using `git`, which is well worth your time and effort to learn.
3. **Be careful to follow directions!** Remember that Python is case sensitive. If we ask you to define a function called `my_function` and you define a function called `My_Function`, you will not receive full credit.
4. **A note on grading:** overly complicated solutions or solutions that suggest an incomplete grasp of key concepts from lecture will not receive full credit.

Instructions on writing and submitting your homework.

Failure to follow these instructions will result in lost points. Your homework should be written in a jupyter notebook file. I have made a template available on Canvas, and on the course website at `http://www-personal.umich.edu/~klevin/teaching/Fall2019/STATS507/hw_template.ipynb`. You will submit, via Canvas, a `.zip` file called `yourUniqueName_hwX.zip`, where `X` is the homework number. So, if I were to hand in a file for homework 3, it would be called `klevin_hw3.zip`. Contact the instructor or your GSI if you have trouble creating such a file.

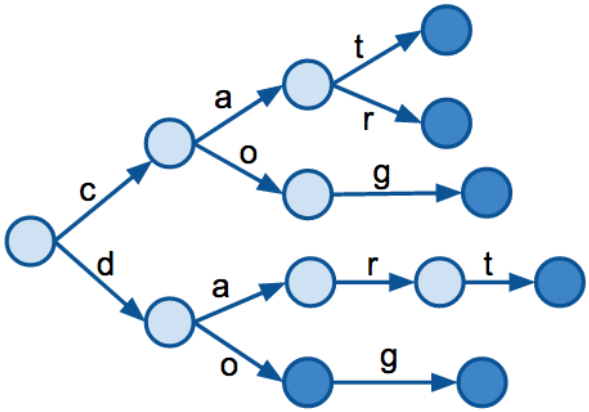
Please include all code necessary to run all problems in the homework in your submission unless instructed otherwise. Please include in your notebook file a list of any and all people with whom you discussed this homework assignment as well as estimates of how many hours you spent on each of the sections of the assignment.

Detailed instructions on submitting your homework can be found on the course web page at `http://www-personal.umich.edu/~klevin/teaching/Fall2019/STATS507/hw_instructions.html`. Please direct any questions to either the instructor or your GSI.

1 Implementing a Basic Data Structure (12 points)

In this problem you'll get a small taste of the kind of problem solving typical of software engineering.

1. A *trie* (rhymes with *tree*, confusingly) is a tree-like data structure for storing collections of array-like data. We will confine our attention to strings for the sake of this problem, but bear in mind that tries can of course be used to represent much more general data than that. In a trie, each path from the root node to a leaf corresponds to a string in the collection, and every string in the collection has a corresponding path from the root to a node (not necessarily a leaf). Each edge from a node to one of its children is annotated with a letter, as in this example, which represents the collection of strings `{cat, car, cog, dart, do, dog}`.



Notice that only certain nodes in the trie are shaded. These correspond precisely to the nodes that represent strings in the collection. In passing, note that tries have certain advantages and disadvantages when compared against other look-up structures like hash tables. Refer to any textbook on data structures (or Wikipedia) for a discussion.

It should be more or less clear how one might represent this data structure as a Python dictionary, but let's walk through it. We will represent a trie by a series of nested dictionaries, with each node in the trie having a dictionary that maps a single character to another dictionary. Thus, for example, the trie in the figure above is, at its outer-most layer, a dictionary, say, `my_trie`, with two keys, `'c','d'`. `my_trie['c']` is itself a dictionary with two keys, `'a','o'` and `my_trie['d']` is a *separate* dictionary with the same two keys. (that is, `my_trie['c']` and `my_trie['d']` are equivalent, but they are not identical). Note that one tricky thing here is to make sure we have a system for representing which leaves (i.e., dictionaries) correspond to the ends of strings (i.e., shaded nodes in the figure above). To solve this, we will use the empty string (`''`) as a sort of special "end-of-word" character. Thus, for example, `my_trie['d']['o']` would itself be a dictionary with the keys `'','g'`. The first key corresponds to the fact that `'do'` is a string in the collection, while the second corresponds to the string `'dog'` (which also happens to be in the collection, so that the dictionary `my_trie['d']['o']['g']` will have the empty string as its only key. We will, by convention, give the end-character symbol `''` the value `None` (note that this is again an arbitrary choice, but one which I am enforcing for the sake of uniformity and ease of grading).

Now, this is all well and good, but the natural (and more principled) thing to do is to wrap this structure in a class. Define a class called a `Trie`, which supports the following methods:

- `__init__(self)` : takes no arguments. Initializes an empty dictionary called `root` as the only instance attribute of the `Trie` object. This dictionary will serve as the data structure described in the previous paragraph.
- `add(self,s)` : takes a string `s` as its only argument. Adds `s` to the trie represented by `self.root` according to the procedure described in the paragraph above.
- `contains(self,s)` : takes a string `s` as its only argument and returns a boolean, which is `True` if and only if the string `s` is represented in the trie.
- `__repr__(self)` : takes no arguments. Returns a string representing the object. When you try to print an object, Python calls this method (if it exists). See `https://docs.python.org/3.6/reference/datamodel.html#object.__repr__` for more information. In our case, let's just say that calling the `Trie.__repr__` method just returns the string representation of the dictionary, which you'll recall is stored as `self.root`.

You are of course free to implement methods additional to these if you wish. **Hint:** you may find it helpful to write a "helper" method for use in the `contains` method, that takes a string and a dictionary as its two arguments, say, `contains_helper(s,d)`. If the first character of the string is not in the dictionary, you can safely return `False`. If the first character *is* in the dictionary, we can recurse, `contains_helper(s[1:],d[s[0]])`. Be careful of the base case, where `s` is the empty string! A similar trick will also help with the `add` method.

2. Write a function called `wordlist2trie` that takes a list of strings as input and returns a `Trie` object representing the collection of strings in the input list. Your function should perform error checking to verify that the input is indeed a list and that all its elements are strings. Note that we could just as well have implemented a method in the `Trie` class to do this, if we wanted.

3. Download the word list from `http://www.greenteapress.com/thinkpython/code/words.txt` (or read it directly using `urlib` or `requests`), and use `wordlist2trie` to build a trie representing the words in the word list. Save the result in a variable called `big_trie`.

2 Comparing Sorting Algorithms (8 points)

A point that we touched upon in lecture is the distinction between worst-case and average-case run time. That is, an algorithm may require, say, $O(n)$ time to run on most inputs, but certain inputs require, say, $O(n^2)$ time. In this problem, you'll see an illustration of this phenomenon in the context of sorting algorithms.

1. We are going to compare the performance of three popular sorting algorithms: merge sort, quicksort and bubble sort. Their respective Wikipedia pages, each of which includes pseudocode, which you are free to consult. Some of these pages display pseudocode for multiple variants of the algorithms (e.g., implementing certain small optimizations). You are not required to implement any of these speedups, but you are welcome to do so if you wish. You are free, if you wish, to simply copy the implementation of quicksort from the lecture slides. **Note:** looking up solutions to this problem from any resource other than these Wikipedia pages and the course

materials will be considered academic misconduct and will be referred to Rackham's academic integrity office accordingly.

```
https://en.wikipedia.org/wiki/Merge_sort
https://en.wikipedia.org/wiki/Quicksort
https://en.wikipedia.org/wiki/Bubble_sort
```

Implement functions called `mergesort`, `quicksort` and `bubblesort`, each of which takes as input a list of numbers (ints and/or floats) and returns the list with elements sorts in non-decreasing order. Your sorting functions should perform error checking to ensure that the input is a list and that the elements of the list are all numeric (i.e., ints and/or floats). Your `quicksort` should use the last element of the input list as the pivot.

2. The Wikipedia page on sorting algorithms includes information about best-, average- and worst-case runtimes for a number of sorting algorithms: `https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms`. Note that our three algorithms differ in some of these. For example, merge sort and quicksort both have $O(n \log n)$ average-case runtime, while bubble sort is $O(n^2)$ runtime in both the average and worst case. Write a function called `run_timing_expt`, which takes a list of numbers (ints and/or floats) as its only input. Calling this function should run each of the three sorting algorithms implemented in the previous subproblem on the given list, timing each call. The function should return a tuple of floats `(t_merge,t_quick,t_bubble)` corresponding to the times that it took to sort the input list with merge sort, quicksort and bubble sort, respectively.

3. Now, we will use the functions defined in the previous two subproblems to explore the behavior of these three sorting algorithms on different kinds of input lists. First, let's see what happens when we pass an already-sorted list. For $n = 500, 1000, 1500, 2000, 2500$, call `run_timing_expt(list(range(n)))` twenty times. Make a plot that shows the average runtime of the three sorting algorithms as a function of the input size n. Please include error bars in your plot indicating 2 standard errors of the mean. Give your plot an appropriate title, and label the axes. Save your plot as `ascending.pdf` and include it in your submission. Note that if you call quicksort on an already-sorted list of size much larger than 2500 you will encounter a recursion depth error (assuming that you are running with the default maximum recursion depth of 5000). Why is this the case? Under more realistic conditions, this would cause quicksort to be slow. In this case, it simply causes quicksort to be unusable for even moderately large inputs.

4. Run the same experiment, but this time with the input being a list sorted in *descending* order. That is, you should run twenty trials of

```
run_timing_expt(sorted(list(range(n)), reverse=True))
```

for $n = 500, 1000, 1500, 2000, 2500$. Again make a plot summarizing the results and including an appropriate title and axis labels. Please include error bars in your plot indicating 2 standard errors of the mean. Save your plot as `descending.pdf` and include it in your submission.

5. Run the same experiment, but this time on random inputs. That is, for each value of n in the previous subproblem, run 20 independent trials of

   ```
   run_timing_expt([random.random() for _ in range(n)])
   ```

   Again make a plot summarizing the results and including an appropriate title and axis labels. Please include error bars in your plot indicating 2 standard errors of the mean. Save your plot as `random.pdf` and include it in your submission.