Assignment for Class Computer Science for Big Data D2

Binary Search and Building an Index, Points: 11

Python has a built-in function to sort lists, which will allow faster search. Read about the built-in sort() and sorted() functions: https://docs.python.org/3/howto/sorting.html, their arguments and returns and sort the list of names. The `find_name()` function will produce the same result as in C1.

1.) Read about deep and shallow copies, https://docs.python.org/3/library/copy.html, and explain the different outputs for cases a., b. and c. [1 Pts]

```
#Case a.
name_list = ns.us_names_top1000.sort()
print(' ' + str(ns.us_names_top1000), '\n', str(name_list))

==>
 ['Abbott', 'Acevedo', 'Acosta', 'Adams', 'Adkins', 'Aguilar', ...
 None

#Case b.
name_list = ns.us_names_top1000
name_list.sort()
print(' ' + str(ns.us_names_top1000), '\n', str(name_list))

==>
 ['Abbott', 'Acevedo', 'Acosta', 'Adams', 'Adkins', 'Aguilar', ...
 ['Abbott', 'Acevedo', 'Acosta', 'Adams', 'Adkins', 'Aguilar', ...

#Case c.
name_list = sorted( ns.us_names_top1000 )
print(' ' + str(ns.us_names_top1000), '\n', str(name_list))

==>
 ['Smith', 'Johnson', 'Williams', 'Jones', 'Brown', 'Davis', 'Miller', ...
 ['Abbott', 'Acevedo', 'Acosta', 'Adams', 'Adkins', 'Aguilar', ...
```

2.) Implement an elegant, recursive Python function that takes advantage of a sorted name list as input:

```
def find_name_fast(_name: str, _sortedlist: []) -> int:
```

The fastest algorithm to search for an element *e* in a sorted list is called *Binary Search* and works by comparing *e* with the middle element of the list. If *e* is "smaller" (<) than the middle element, continue search in the lower half. If *e* is "larger" (>), continue search in the upper half. Repeat the process until *e* is found or return -1 if *e* is not in the list. Avoid splitting/slicing the list (hint: define an inner function that passes upper and lower search bounds in the searched list).

For names as strings, alphabetical order is used to decide "smaller" and "larger" cases. For example, `'Adams' < 'Adkins'`. To test your function, run (names, name_freq are from C1(6)):

```
sorted_list = sorted(ns.us_names_top1000)

for n in names:
    i = find_name_fast(n, sorted_list)
    print(str(i) + '\t<-- ' + str(n), end='')
    print('\t\t--> freq: ' + str(name_freq[i]) if i >= 0 else "")
```

Output shows name indices from the sorted list that are different from to the original list ('Abbott' is now the first entry in the sorted list with index 0 compared to index 462 before): [2 Pts]

```
==>
854     <-- Smith                --> freq: 0.014        ; should be: 1.006
617     <-- Mendoza              --> freq: 0.019        ; should be: 0.043
785     <-- Rodriguez            --> freq: 0.015        ; should be: 0.229
0       <-- Abbott               --> freq: 1.006        ; should be: 0.025
-1      <-- Blokes
99      <-- Brewer               --> freq: 0.083        ; should be: 0.042
918     <-- Vang                 --> freq: 0.013        ; should be: 0.012
999     <-- Zimmerman            --> freq: 0.012        ; should be: 0.026
28      <-- Bailey               --> freq: 0.192        ; should be: 0.115
```

3.) What is the "time-complexity" of this "binary" search function (comparisons are needed for *n*-elements in the input list):

a. for the best case (minimal number of comparisons needed for a list with *n*-elements),

b. the worst case (maximal number of comparisons) and

c. on average? [1 Pts]

4.) How many steps are needed for the list of 1000 names for cases:

	on avg	worst case
Linear search in unsorted list:		
Binary search in sorted list:		

[1 Pts]

5.) Frequency values are displayed incorrectly because they are being looked up with wrong indices in `us_name_freq_top1000[]`. To fix the problem, a common approach is to create an *index data structure*. The index here will represent the sorted name list and store a reference to also locate the corresponding elements in the other lists (such as the list with name frequencies).

Consider a memory-efficient data structure to store this information under the assumption that the index is only built once (name and other lists are read-only).

Describe your data structure in few statements. [1 Pts]

6.) What is the "space complexity" of that data structure for *n* names? [1 Pts]

7.) Implement your approach by a function that builds and returns the index:

```
def build_index(_list: []) -> <<your_index_data_structure>>:
    #code ...

index = build_index(ns.us_names_top1000.sort)
```

[1 Pts]

8.) Demonstrate that correct name frequencies appear as output. [1 Pts]

9.) Would Python dictionaries be an alternative for creating an index that would work for binary search? Find out based on which data structure dictionaries are implemented. Briefly describe this data structure and its time complexity (rough estimates). [2 Pts]

(see: https://stackoverflow.com/questions/327311/how-are-pythons-built-in-dictionaries-implemented)

Answers:

C1.2 time complexity, linear search in unsorted list of *n* elements:

a. best case: 1
b. worst case: 1000
c. average: 500 (n/2)

C2.3 time complexity, binary search in sorted list of *n* elements:

a. best case: 1
b. worst case: 10, da $2^{10} = 1024$ → binary search complexity is: $\log_2 n$, n = 1000 → 10
c. average: ~ 9 - 10

C2.4 How many steps are needed for the list of 1000 names for cases:

	on avg	worst case
Linear search in unsorted list:	500	1000
Binary search in sorted list:	9 - 10	10

[1 Pts]