COSC4315/COSC6345: Programming Languages
Functional programming: Finding the k most frequent integers and k most frequent reals in a file

1 Introduction

You will write a program to find the most frequent numbers in a text file: integers or reals. This is the main mechanism to find documents on the web in a search engine. There will be an input integer $k \geq 1$ as parameter.

The input file can contain words and numbers as well as other symbols, but words and other symbols can be ignored. The input files can be large (having 1000s of numbers, mixed). Since this homework requires functional programming, you should use recursive functions and lambda expressions to solve the problem. You should not worry about memory efficiency and time complexity within reason.

2 Program and input specification

The main program should be called "freqnumber" (exactly like this, in lower case, avoid other names).

Call syntax (assuming your main program is freqnumber.py):

```
python3 freqnumber.py "k=3;input=input.txt;output=freqnumber.txt"
```

A number is a regular expression of digits, perhaps having a decimal part. A number with only digits and sign is an integer. A number having a decimal point will be considered real. Scientific notation will not be used (exponent+mantissa). Therefore, strings mixing letters+numbers can be ignored. Notice that words can be followed by punctuation signs, they can be capitalized.

2.1 Input

Input: A plain text file mixing numbers and words. The input file name will not necessarily be the same every time. Therefore, your program will have to parse the input file name.

The input is one text file, with words and numbers, separated by spaces, punctuation symbols and any other common text symbols. Repeated spaces should be considered as one space. You can assume integers fit in 32 bits (i.e. bounded) and real numbers as double (5 decimals max, 64 bits). Numbers can have a negative sign.

2.2 Output

The output should be sent to the output file *exactly* in the format below with each number and its frequency, one per line for automated testing. There will be two lists: one for integers and one for reals. Avoid changing the output format or adding any other messages in the output file since this will prevent automated testing. Any messages should appear on the screen or a different file.

Output example with $k = 3$ and most frequent numbers in two lists:

```
integer:
1  1000
-1  500
0   500
real:
-2.1 100
3.1416 5
1.5 3
```

3 Requirements

- Programming language: Python, version 3 (any).
- Functional programming is the main requirement, as detailed below. Basically, this means all functions should be recursive, there should be no mutation and all computations will happen via lists as input and lists as output. That is, functional programming is a stronger requirement than recursive programming.
- Functional expressions: Your program can use map()/reduce()/filter() functional constructs to compute frequencies when possible. Your program should preferably use lambda expressions to convert strings to integers/reals, instead of built-in functions.
- Lambda expressions: it is acceptable to name and reuse lambda expressions. It is acceptable to define nameless lambda expressions, inside a function (used once). It is recommended you use a lambda expression instead of a recursive function for one-liner/elegant expressions (i.e. length, insert, count). That is, aim to develop elegant lambda expressions.
- Lists (as available in the language) are required to store numbers: input, temporary and output. You should use primitive operators provided by the language. Python: Array-based lists (provided by the language) are encouraged.

  You cannot use existing Python libraries with numeric arrays (e.g. numpy, scipy), dictionary (nlp) or hash tables data structures. Therefore, you cannot use their associated functions either. Unfortunately, these popular Python libraries defeat a functional programming approach, do all the list algorithmic work and there is too much example code on the Internet that you can copy/paste (i.e. you learn nothing). In other words, you are expected to develop recursive algorithms based on lists like a functional programming like ML, Haskell, Scala.
- No mutation: make sure lists are not mutated (overwritten), especially for inserting and sorting. Mutation in a few temporary list variables is acceptable, but it can be avoided with numbered variables. Mutation in a growing list to read the input file is acceptable, but it can be avoided. Mutation to insert or delete elements from lists should be avoided if possible (explain why not). Notice that in functional programming it is the job of the compiler/interpreter to reuse space/variables.
- Efficiency: It is important to emphasize that efficiency (saving space) is less important than correctness and functional style. Specifically, do not worry about space, other than not exceeding the stack limit (recursion depth). It is important to know $O()$ for every function you develop. It is acceptable to have $O(n^2)$, but $O(n \log(n))$ is preferred when possible.
- We will follow a "strongly typed" language approach. Lists will be strongly typed: one list for integers and one list for real numbers. Therefpore, it is a bad idea to have a single "mixed" list of reals/integers since you are required to find the most frequent integers.
- Recursive functions are required to process all lists. It is unacceptable to have loops (while/for) to process the list(s). Using loops to print() the list is unacceptable. Loops: while/for loops are acceptable only for I/O to read the input file into a list, but recursive I/O functions are encouraged (do not worry about speed). Using split() is discouraged, as it does all the work.
- Search and sort algorithms: Lists should have recursive sort and recursive search functions, without mutation. It is acceptable to have an $O(n^2)$ sort algorithm, but $O(n \log(n))$ is encouraged; specify which sort algorithm you are using. Search can also be sequential or binary, with binary search being preferred. Keep in mind a recursive sort algorithm will use significant RAM. Important: you cannot use built-in sort/search functions since most of them produce variable mutations. Therefore, you must build your own sort and search functions.
- Optional: for +10 points, up to 100 (cannot be applied to other homeworks).

  Rare/extreme cases should be considered. Rank ties should be considered (i.e. integers or reals having the same frequency, rare but feasible). If there are ties: this is the problem "find all integers (or reals) that have the k highest frequencies". For instance, if $k = 2$ if there are 2 numbers with the highest frequency, 3 with the 2nd largest frequency the output would be 5 numbers. Second example, the file has only number 2, repeated 1000 times and $k = 3$. The output is only "2 1000". Third example, a file without numbers, just words: output is empty. Fourth example, many negative numbers (integers and reals). Fifth, very wide lines with up to 100 numbers.

  In your README file explain if you handle ties, rare cases.

- Big Data, +20 points!!: files of size in the order of GBs, which cannot fit in RAM. A correctness/speed comparison with an SQL engine is required. This extra work can be considered at the end of the course.

  You can send me a personal email if you solve this problem.

- Correctness is the most important requirement: TEST your program with many input files, especially large files. Your program should not crash or produce exceptions.

- A list needs to be passed as a function argument and/or function return value, but cannot be a global variable.

- Create a README file explaining how to run your program and explaining any issues or incomplete features.

- Your program will be tested for originality, comparing it to other students source code, source code students from past semesters and top source code sites from the Internet (e.g. github, stackoverflow). You must disclose any source code that was not written by you. Keep in mind several students may get source code from the same site, especially if it comes as a top hit in Google.

- Your program should display a small "help" if not input parameters are provided and must write error messages to the screen. Your program should not crash, halt unexpectedly or produce unhandled exceptions. Consider empty input, blank lines and so on.

- Unix system (UH Linux server). You can develop and test your code on any computer or operating system (e.g. Windows, Mac), BUT your program will be tested only on our course server. Therefore, test it carefully on the UH server before the deadline.

- Test cases:

  Your program will be tested with 10 test cases, going from easy to difficult. You can assume 70% of test cases will be clean, valid input files. The remaining 30% will test correctness and efficiency with harder input.

- Grading:

  A program not submitted by the deadline is zero points (any exception must be discussed with the instructor at least 2 days before the deadline). A non-working program is worth 10 points (i.e. not producing ouput for the easiest test case). A program that does some computations correctly, but fails several test cases (especially the easy ones) is worth 50 points. Only programs that work correctly with most input files that produce correct results will get a score of 60 or higher. In general, correctness is more important than speed.