COSC2430 HW3: Search, Sorting, Recursion and Time Complexity
A Spell Checker (searching words)

1 Introduction

You will create a C++ program that implements a spell-checker, using efficient/inefficient sorting and efficient search algorithms. Algorithms will require recursion and you need to analyze O().

In essence a spell checker works by comparing all the words in a document to words in a ”dictionary”. If a word is not found in the dictionary, Then the program assumes that it is an error (misspelled word). In order to efficiently spell check a document you need to store the dictionary in a sorted array, and you need an efficient way of determining if a word can be found or not in the dictionary array. Therefore, you must use an efficient search algorithm (Binary Search) to determine if each word of the input file is present in the dictionary.

You will read the dictionary file into a linked list (STL acceptable in case you failed previous homeworks). The dictionary file may have errors, meaning duplicate words or numbers. The dictionary file must be processed to eliminate errors and sort the words to enable efficient search. Once the file is processed, you should allocate an array and use it as the dictionary You must delete the original linked list to free memory.

You will use the dictionary to check spelling of an input file (e.g. some document). The input file to be checked can either be loaded into a linked list, sorted and stored in a more efficient manner, OR it can be processed one word at the time (but keep in mind we need word frequencies). The second method is less efficient, since there could be repeated words in the input documents, but easier. In either case you must use an efficient search algorithm (binary search) to determine if each word of the input file is present in the dictionary. You should produce a list with all the words that are misspelled in the document.

The major goals of the homework are to master linear recursion and understand big O().

2 Program and output specification

The main program should be called ”spellchecker”. Call syntax example:

```
spellchecker input=f01.txt;dictionary=dict.txt;recursive=y;sort=slow/fast
```

When asked to output the list if incorrect words to a file you must write one word per line followed by a space and the frequency (number of occurrences).

3 Examples

INPUTS

```
dictionary.txt
---------------------------------------------
```

```
alpha beta gamma delta epsilon zeta eta theta
iota kappa lambda mu nu xi omicron pi
rho sigma tau upsilon phi chi psi omega
<EOF>

input1.txt
----------------------------------------------
alfa alpha halpha beta veta kapa alfa
<EOF>

input2.txt
----------------------------------------------
alphA

mega omega

<EOF>

input3.txt
----------------------------------------------
alpha beta delta zeta epsilon
theta
mu
nu<EOF>
```

RESULTS

```
out1.txt
----------------------------------------------
alfa 2
halfa 1
veta 1
kapa 1
<EOF>


out2.txt
----------------------------------------------
mega 1
<EOF>

out3.txt
----------------------------------------------
<EOF>
```

TIME COMPLEXITY

```
time.txt
```

```
------------------------------------
BINARY SEARCH

T(n)= 5*n^2 + 10*n + 7
O(n^2) c=9.999 n_0=100

     n  #operations  T(n)  O(f(n))
================================-
   100          99     99     999
  1000          99     99     999
 10000
100000
..

<EOF>
```

4 Requirements

- Input file: it may have repeated words. There can be numbers that must be ignored in the search process.

  Dictionary file: unsorted, may have numbers, maybe it has repeated words

- Search: You must search the "clean/sorted" dictionary only with an efficient algorithm: only $O(log(n))$.

- Sort: Your program must sort the dictionary with one fast and one slow algorithm (your choice). Efficient means an $O(n \log n)$ algorithm, slow means $O(n^2)$.

- Recursion: you must sort words and search with recursive algorithms. There must be search and sort recursive functions of the algorithm. Search: It is expected search algorithms will be purely recursive. Sort: OK, to combine loops and recursion in the default textbook algorithm, more difficult to develop purely recursive; a purely recursive sort algorithm can receive 20% extra credit (no loops at all).

- Time complexity: you must count the number of operations (1 per C++ line, including comparisons, assignments and function calls), adding a counter to the source code line by line, analytically derive $T(n)$ and then derive a tight bound $O(f(n))$ for $T(n)$. You will add these two functions to your program passing n as an argument. That is, these functions are derived with pencil/paper and they are practically copied/pasted into your program. It is expected that the difference between $T(n)$ and $O(f(n)$ will get smaller as n grows. That is, you produce the $O()$ table in log scale. In your README file you must explain $T(n)$ and its $O(f(n))$ in a short paragraph making clear which are your c and n_0. You should append the table with a new analysis every time you test an algorithm (do not overwrite the time.txt file).

  This O() analysis must be done ONCE before you actually process the entire files, once per algorithm with large arrays. Your program must append the file time.txt with your O() analysis produced by your program for each algorithm. You must produce $O()$ analysis tables comparing search and sorting number of operations, $T(n)$ and $O(f(n))$ for large n (10,100,1000,10000,100000,..) populating the array with random digits converted to letters (e.g. '012215' becomwes 'ABCCBF'). That is, do not run this analysis for every call, but instead you produce one $O()$ table per algorithm. The goal is to understand the time complexity as n grows BEFORE you execute the spell check (testing efficiency of your algorithms): $T(n)$ gets closer to $O(f(n))$.

- You should also use an efficient searching algorithm, such as Binary Search, in order to find out if a word is misspelt. There will be no punctuation signs in the input files, only letters and numbers. Since it is possible that some letters will be capitalized (upper case), you should convert all words to lower case.
- Write incorrect words (not found) in alphabetical order, followed by their frequency.
- Phases

  Phase 1: non-recursive linear search (not tested); recursive linear search (not tested); non-recursive binary search; recursive binary search, any sort algorithm (can be non-recursive and it can be slow),

  O() analysis tables of binary search (two versions, delayed to Phase 2).

  Phase 2: recursive fast and slow sort algorithms (i.e. two recursive algorithms), O() analysis tables of sort algorithms.
- The program should not halt when encountering errors in the script. It should just send a message to the log file and continue with the next line. The only error that is unrecoverable is a missing script file or a missing dictionary.
- Do not use search/sort array algorithms in the STL library: you are expected to write your own algorithms, even if taken from the textbook or the web.
- Your program should write error messages to a log file (and optionally to the screen). Your program should not crash, halt unexpectedly or produce unhandled exceptions. Consider empty input, zeroes and inconsistent information. Each exception will be -10.
- GRADING

  Test cases: Your program will be tested with 10 test scripts, going from easy to difficult, counting 5 points each. You can assume 80% of test cases will be clean, valid input files.

  Recursion: non-recursive solution will receive 75% credit on Phase 1. non-recursive solutions will receive 50% credit on Phase 2.

  Time complexity: time complexity tables in the time file will count 50 points altogether (3 tables).
- A program not submitted by the deadline is zero points. A non-working program is worth 10 points. 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 80 or higher. In general, correctness is more important than speed.