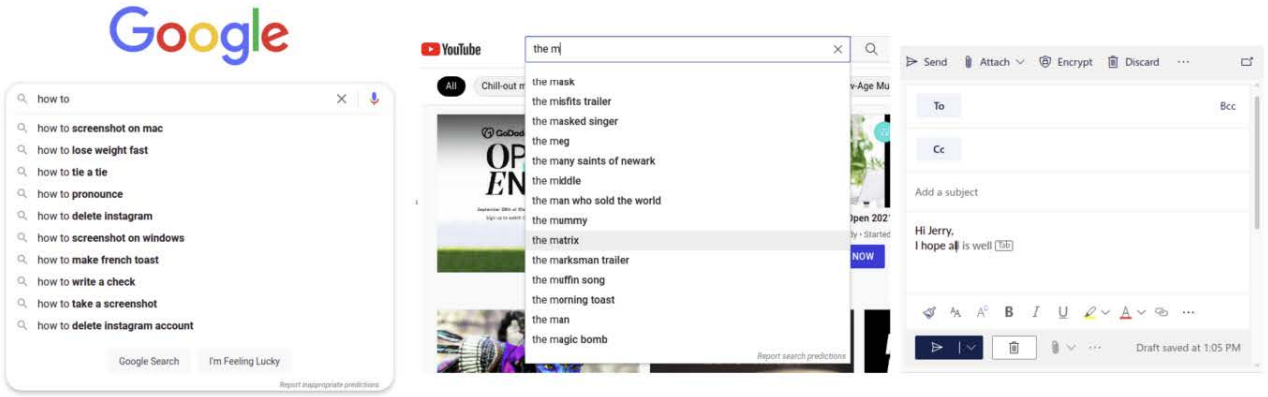
2 AUTOCOMPLETE

Autocomplete, or word completion, is a common feature that we interact with in our everyday life in which an application predicts the rest of a word you are in. You must have seen it being used in websites like Google (or other search engines), Youtube and Outlook/G-mail.



Autocomplete is everywhere!

One key characteristic of an autocomplete program is to suggest the most relevant query, which can reasonably be defined as the most frequently encountered, as the user starts typing in the initial parts of a query. When typing

into, say, the Google search bar, it is very likely that the underlying algorithm has already seen billions of queries so far and knows how frequently each query is searched. Hence, when the query is being typed, the algorithm searches for the most relevant queries (most frequently searched in the past) from its knowledge base and lists the top ones as suggestions.

The performance of autocomplete is crucial. First, it needs to provide suggestions for a query faster than a user would type it in such that the suggestion should make sense. Why? No one would wait for an autocomplete feature that takes seconds to do the job. Second, we also need to consider that this algorithm will be running again and again for each keystroke by the user. This is bound to cause a huge computational burden on the server if the algorithm is inefficient. Third, the top suggestions provided by the algorithm should be the most relevant to the query. This means that the utility should order the suggestions based on the frequency of the queries in the preexisting knowledge database.

3 Problem

In this assignment, you will write a C program that will implement an autocomplete utility. Your program will read a text file that contains query words along with weights (represented by a string and integer pair in each line) where larger weights indicate a higher likelihood of observing each query.
Your program will read a second file that contains a list of substrings that would represent new queries that are partially written, and print out query words from your knowledge base in sorted order that matches the provided substring (sorted by frequency) for each of the substrings in that second file. To be more specific, you need to consider the following four problems for this assignment:

1. **Sort 1:** When loading the knowledge base file, you should sort and store the word list in memory in such a way that we can efficiently search the list for suggestions later down the line.
2. **Search:** You will need a search algorithm that will efficiently search for the given query. Keep in mind that, your query will most probably be incomplete words. Thus, you will need to search for terms in the knowledge base for entries whose initial characters match the query word. But the query can be an exact match as well.
3. **Sort 2:** If your search leads to more than one entry as output, you should sort them according to their frequencies and print the suggestions from the most frequently encountered to the least frequently encountered.
4. **Efficiency:** Your program needs to run fast. To give you a concrete reference point: your program should not take more than twice the run time of the executable that is provided in the starter package.

IMPORTANT: Do not use available C libraries (such as qsort() function) which does sorting. Write sort function of your own. You can use C library function rand() if you need a random number generator.

4 Starter Package

- Along with this PDF document, you have been provided a starter C code file, an executable, a knowledge base file, and a query list file. In more detail,
  - The **starter code** handles reading the input files and the command line arguments.
  - You can use the **executable** to compare your program's output to match input/output specifications.
  - You can use the knowledge base file and the query list files as references as to how you handle input/output.

5 Input/Output Specifications

- The **executable program** generated from your C code is supposed to take two command line arguments, the name of the input knowledge base file and the name of the query list file. Thus, an example call to your program would be as follows. (assuming that you used the following command to compile your code that was written inside main.c: "gcc main.c -o auto", which will produce an executable named 'auto')

$./auto movieScripts.txt queries.txt

- The **input knowledge base file** that contains the frequency of various words used in English movies has been provided to you. We will test your program with other input knowledge base files as well which will have the same structure as this one.

```
CCI-FGG0ZQQ6LX:pa2 yo42$ more movieScripts.txt
you 28787591
i 27086011
the 22761659
to 17099834
a 14484562
's 14291013
it 13631703
and 10572938
that 10203742
't 9628970
of 8915110
is 7400675
in 7337058
what 6900164
we 6755687
me 6444985
this 5739788
he 5516364
for 5174060
```

```
CCI-FGG0ZQQ6LX:pa2 yo42$ more simpsons.txt
the 107946
you 98068
i 91502
a 79241
to 70362
and 47916
of 42175
it 36497
in 32503
my 32254
that 32083
is 31533
this 29902
me 28208
your 26781
for 25634
oh 25053
```

First few lines of sample input files!

  - Each line represents a single query in the knowledge base
  - The first column contains the query as a string. Don't worry about the punctuation marks in the queries. Search for whatever is in those strings, which might contain apostrophes, etc.
  - The second column contains the weight of the query word. You can read it as an *int*. The two columns are separated by a single whitespace character.
  - Note that the input file **may or may not** be in any sorted order.

- Your program is going to read the **query list file**, which consists of one substring per line, and search for each of these substrings in the knowledge base. Below is an example query list file consisting of three lines of substrings.

  class
  som
  noup

  - If the query is found, your program should print the list of terms that have the query as an initial substring in sorted order with respect to their frequencies. If there are many suggestions, you should print **only the top 10** suggestions with respect to their frequency as your output.

  - If the item is not found in the knowledge base, your program should output *"No suggestion!"*.

- The **output** should be printed to *stdout* and should be structured as follows for the queries provided above (which are presumably stored in queries,txt for this example).

```
$ ./auto simpsons.txt queries.txt
Query word:class
class 875
classic 192
classy 67
classroom 50
classical 41
Query word:som
some 6138
something 4109
```

someone 1564
somebody 529
sometimes 426
somewhere 318
someday 179
someone's 169
something's 146
sometime 124
Query word:noup
No suggestion!

- First, print "Query word:" followed by the word being queried and a new line.
- If the query is found, each of the following lines should be in <query> <weight> format, with a single space character between query and weight.
- If there are multiple outputs, print a newline character at the end of each line.
- If the query is not found, print *"No suggestion!"*.

- To measure **runtime** of your program, use the "time" command of Unix, that runs on tux. Below is an example run of the program with a query list file that contains only a single substring "class":

```
$ time ./auto simpsons.txt queries.txt
Query word:class
class 875
classic 192
classroom 50
classical 41

real 0m0.013s
user 0m0.002s
sys 0m0.002s
```

- Focus on "user" and "sys" portions of what is printed, as this shows how much time did the process take on CPU (user) and system calls such as memory allocation (sys) from its start to end.
- We will test performance of your program in addition to its correctness. To get full credit from a test case, runtime of your program should not take longer than at most twice what the provided executable takes.
- Note that the time portion printed above will be handled by the Unix command "time". So, your program should not print any time information.

- **Test cases and grading:** Your program will be tested across several test cases, and grading for each test case will be as follows:
  1. You get full credit from a test case if your output is both correct and is produced within the time limit mentioned above.
  2. You get half the credit from a test case if your output is correct but it took longer than the above mentioned time frame to produce the output.
  3. You get no credit from a test case if your output is wrong.

6 SUBMISSION

- Even if you develop it elsewhere, your code **must** run on **tux**. So, make sure that it compiles and runs on tux prior to submitting it.

- Your program should take two arguments from the command line, which will be the path to the two input files containing knowledge base file and the query list file (as explained above), and should print the output to standard output.
- Your submission will consist of two separate files (do not compress/zip the files).
  1. A single C file named "main.c", which includes your code for the assignment.
  2. A single filed named **self_grade**, without any file extension, which consists of two lines, such that:
     - first line consists of a single number in [0,30] range to indicate your self assessment for your effort/performance for the assignment.
     - second line is a short description of how much effort you put into the assignment, such as when you started, how many hours it took in total, whether you attended office hours etc.
- Submit your assignment by following the Gradescope link for the assignment through Blackboard Learn.

7 GRADING

- Assignment will be graded out of 100 points.
- You will provide your self-assessment score for your effort as a 30-point portion of the assignment.
- The remaining 70 points will be awarded according to how many test cases your program is able to pass.