### Submitting your project

Submit the file

- `autocomplete.c`

on Gradescope.

The file will be compiled using

```
gcc -Wall -std=99 autocomplete.c main.c -o autocomplte
```

The file `autocomplete.h` that is given to you will be present in the same folder. You must make sure that your code passes the tests on Gradescope `and` that it runs correctly on ECF.

### Clarifications and discussion board

Important clarifications and/or corrections to the project, should there be any, will be posted on the ESC190 Piazza. You are responsible for monitoring the announcements there.

### Hints & tips

- Start early. Programming projects always take more time than you estimate!
- Do not wait until the last minute to submit your code. You can overwrite previous submissions with more recent ones, so submit early and often—a good rule of thumb is to submit every time you get one more feature implemented and tested.
- Write your code incrementally. Don't try to write everything at once, and then compile it. That strategy never works. Start off with something small that compiles, and then add functions to it gradually, making sure that it compiles every step of the way.
- Read these instructions and make sure you understand them thoroughly before you start—ask questions if anything is unclear!
- Inspect your code before submitting it. Also, make sure that you submit the correct file.
- Seek help when you get stuck! Check the discussion board first to see if your question has already been asked and answered. Ask your question on the discussion board if it hasn't been asked already. Talk to your TA during the lab if you are having difficulties with programming. Go to the instructors' office hours if you need extra help with understanding the course content.

  At the same time, beware not to post anything that might give away any part of your solution—this would constitute plagiarism, and the consequences would be unpleasant for everyone involved! If you cannot think of a way to ask your question without giving away part of your solution, then please drop by office hours or ask by private Piazza message instead.
- If your message to the TA or the instructor is "Here is my program. What's wrong with it?", don't expect an answer! We expect you to at least make an effort to start to debug your own code, a skill which you are meant to learn as part of this course. And as you will discover for yourself, reading through someone else's code is a difficult process—we just don't have the time to read through and understand even a fraction of everyone's code in detail.

  However, if you show us the work that you've done to narrow down the problem to a specific section of the code, why you think it doesn't work, and what you've tried to fix it, it will be much easier to provide you with the specific help you require and we will be happy to do so.

### How you will be marked

We will mark your project for the correctness of the functions that you are required to write. **Make sure that you follow the specifications exactly.**

### Correctness

Note that **your functions must be implemented precisely according to the project specifications.** Their signatures should be exactly as in the project handout, and their behaviour should be exactly as specified. In particular, make sure that functions do not print anything unless the project specifications specifically demand that, and that the functions return exactly what the project handout is asking for.

In this project, you will write a fast implementation of the autocomplete functionality.

You will work with files that contain a large number of terms, along with the importance weight associated with those terms.

The autocomplete task is to quickly retrieve the top terms that match a query string. For example, if the query string is `"Eng"`, your matches might be `"English"`, `"Engineering"`, and `"EngSci"`. In the project, matches are case-sensitive, and only the beginning of the string is matched. `"Eng"` matches `"EngSci"` but not `"engaged"` and `"Sci"` matches `"Science"` but not `"EngSci"`.

To accomplish the task, you will:

- Read in the entire file of terms, and sort the terms in lexicographic ordering. This sorted list will be reused for multiple queries
- Use binary search to find the location of the first term in lexicographic ordering that matches the query string, as well as the last term in lexicographic ordering that matches the query string
- Extract the terms that match the query string, and sort them by weight
- Extract the top matching terms by weight

Throughout, you will use the following `struct`:

```
typedef struct term{
  char term[200]; // assume terms are not longer than 200
  double weight;
} term;
```

## Part 1.

Write a function with the signature

```
void read_in_terms(term **terms, int *pnterms, char *filename);
```

The function takes in a pointer to a pointer to `term`, a pointer to an `int`, and the name of a file that is formatted like `cities.txt`.

The function allocates memory for all the terms in the file and stores a pointer to the block in `*terms`. The function stores the number of terms in `*pnterms`. The function reads in all the terms from `filename`, and places them in the block pointed to by `*terms`.

The terms should be sorted in non-descending lexicographic order. You must use `qsort` to accomplish this. You can assume that `strcmp` correctly returns a negative number of the first argument comes before the second number in lexicographic order, and a positive number if the first argument comes after the second argument in lexicographic order. (Note that `strcmp` assumes that the inputs are C strings that consist of one-byte characters; the actual input you are given is encoded using UTF-8, but you can ignore this).

## Part 2.

Write a function with the signature

```
int lowest_match(term *terms, int nterms, char *substr);
```

The function returns the index in `terms` of the first term in lexicographic ordering that matches the string `substr`.

This function must run in $\mathcal{O}(\log(\text{nterms}))$ time, where `nterms` is the number of terms in `terms`.

You can assume that `terms` is sorted in ascending lexicographic order.

## Part 3.

Write a function with the signature

```
int highest_match(term *terms, int nterms, char *substr);
```

The function returns the index in `terms` of the last term in lexicographic order that matches the string `substr`.

This function must run in $\mathcal{O}(\log(\text{nterms}))$ time, where `nterms` is the number of terms in `terms`.

You can assume that `terms` is sorted in increasing lexicographic order.

## Part 4.

Write a function with the signature

```
void autocomplete(term **answer, int *n_answer, term *terms, int nterms, char *substr);
```

The function takes terms (assume it is sorted lexicographically in increasing order), the number of terms `nterms`, and the query string `substr`, and places the answers in `answer`, with `*n_answer` being the number of answers. The answers should be sorted by weight in non-increasing order.

**Sorting with `qsort`**

See here: `https://www.tutorialspoint.com/c_standard_library/c_function_qsort.htm`

You **must** use `qsort` for this question.

*Credit: the assignment was designed by Kevin Wayne, and adapted to C by Michael Guerzhoy*