CSE100 - Winter 2018

Programming Project 2

Checkpoint Deadline: 11:59pm on Tuesday, 11/6 (not slip day eligible)
Final Deadline: 11:59pm on Tuesday, 11/13 (slip day eligible)

Overview
In this project, you will be asked to complete two text generation tasks. The first is to implement a string autocompleter, like what your browser gives you when you start to type in text. The second is to implement a document generator, which will create documents that mimic the style of specific authors. **Your first step is to *read these instructions carefully*.** We've given you a lot of advice here on how to succeed, please take it. Don't post a question on piazza asking something which is already answered here.

Pair Programming Partner Instructions

If you wish to work with a partner, please be sure to read the guidelines for Pair Programming in the syllabus carefully. Once you are both sure you can work together as a pair, please fill out this form.

NOTE!!: Even if you paired with a partner previously, you must **RESUBMIT** a partner pair form if you want to continue to pair for this assignment.

Should a relationship dissolve, you can fill out this form to request a break-up. But by doing so, you are agreeing to delete all the work you and your partner have completed. Any code in common between dissolved partnerships is a violation of the Academic Integrity Agreement.

Retrieving Starter Code

For PA2, you are given the top level interface for the data structures you need to implement, in addition to some helper files. To retrieve these files, you need to log into your CSE 100 account. (Use either your lab account or your user account. Log in to a linux machine in the basement or ssh into the machine using the command “ssh *yourAccount*@ieng6.ucsd.edu”). When logging in, you will be prompted for a password. You should already have run ‘‘prep cs100w” by now.

Once you are logged in, open up a linux terminal and run the command “getprogram2 <directory>” where <directory> is the name of a folder that you want to be created to hold the starter code. After this command is run, you should see that the directory you specified has been created if it previously did not exist, and the directory should contain the starting files:

```
create-submission-zip.sh      example_dg_datasets  src                  test
example_autocomplete_datasets Makefile             starter_pa2.tar.gz
```

And src will contain the files:

```
Autocomplete.hpp  main.cpp  Makefile
```

Do not modify `main.cpp` or `main2.cpp` - they are used to test your code.

The following make targets are provided for you:

- `main` - makes the autocompleter with debug info, provides the executable `build/main`
- `submit_main` - same as above, but fully compiler optimizes (-O3) instead of providing debug info. This is what’s used by the grading scripts
- `main2/submit_main2` - the same as above, but for the document generator
- `test_autocomplete` - makes the executable `build/test/test_autocomplete` from `test/test_autocomplete.cpp` as a convenience for you to start writing your own tests
- `test_document_generator` - same, except for the document generator

⚠ There is currently a bug in the `Makefile` where if you run two `make` commands in a row (e.g. `make main && make test_autocomplete`), the second call to make often fails due to a linking bug. You can modify the `Makefile` to fix that, but a simple workaround is to call `make clean` before the second call to `make` - this works fine.

Checkpoint: Implementing a string autocompleter

In the first part of the project you will write a string autocompleter. The idea here is that when someone starts typing in a string, let's say "tho", you provide all the possible words that could possibly complete that string, such as "thought", "those", "thoroughly", "though", with the most likely completion being returned before the less likely ones. This functionality is what you see when you're texting on your phone or searching on google. Your autocompleter will learn which completions to use by reading in many words from a real world dataset we've provided.

Your assignment is to implement the three methods in `Autocomplete.hpp` file using either Multiway Tries or Ternary Tries. Read `Autocomplete.hpp` file carefully to understand what is required. The `main.cpp` file we've provided runs your autocomplete code in a way that is usable by the grading scripts. Feel free to look at it, but you shouldn't need to modify it.

Your autocompleter will be graded not only on correctness, but on efficiency. It must handle large (50MB) inputs and many (thousands) of queries. Our testing advice below contains info about sample datasets. See below for advice on programming the data structure.

Some notes:

Hopefully you're excited about running your code on real data! Do not be too concerned about the efficiency requirements - you don't need to rely on any special tricks. Just make sure that you don't choose an inefficient design, and that you don't unnecessarily copy data.

For both the autocompleter and the document generator, because we're giving you real world data, you'll run into accented characters and random non-alphabetic punctuation. We've sanitized the input data in both cases so that you can read it in as you normally do, as ASCII chars, and don't have to deal with Unicode. But make sure you programs can handle characters such as 'á'.

You **may not** use `std::map` from STL in this assignment - that will be considered cheating. (This is STL's ordered map implementation). `std::unordered_map` is allowed. You may use any other code in STL, and if you desire (though it may not be helpful) your code from the previous assignment.

Our grader reads what `main.cpp` writes to standard out - if you print any debugging info to standard out, this will throw off our grader and cause it to fail your submission.

We've not provided you with any reasonable tests; the grading script produces very little output. You should write many tests of your own. Tutors will not help you debug your code based on grading script output without clear evidence of substantial testing on your own part.

Lastly, unlike the previous assignment, we've given you very little outline of how to layout your code. This is what lots of programming is like, so we want to help you get comfortable with that now. Good luck.

Trie data structure

You may implement either a ternary trie, or a modified form of the multiway trie, your choice. The input to your autocompleter has already been preprocessed (converted to lowercase, stripped of annoying punctuation, split into words) so you only need to worry about the data structure itself, and not the text processing.

To generate the completions for a prefix, you'll find the node in your trie that corresponds to the last letter of the prefix, and look through all of its descendants. You'll choose those descendants which are words, and among those words, which occur most frequently. You will then return the most frequent words in the input text which start with the prefix. See the notes in Autocomplete.hpp for more details.

The constructor `Autocompleter(const vector<string> words)` should build your data structure and do the bulk of the work. The `predictCompletions(const prefix)` method should only query the data structure you've already built, and not perform any modifications.

When writing your Autocomplete data structure, good coding practice dictates that you put the different classes you create in separate files (e.g. `Trie` and `TrieNode` among other possibilities). That's okay - we will collect all the files in your src/* directory when grading your code. **One obvious caveat, you have to have written, from start to end, any code included in your turn in.**

Because we aren't using templates in this assignment, you should follow best practices and put your implementation in Autocomplete.cpp (or other .cpp files), not in the .hpp

file. If you use a .cpp, you'll need to add it to the targets of the `Makefile`. E.g., if you use `Autocomplete.cpp`, you'll modify your `Makefile` like so:

```
main: init src_object.main src_object.Autocomplete
submit_main: init src_object_optimize.main src_object_optimize.Autocomplete
test_autocomplete: init test_object.test_autocomplete src_object.Autocomplete
```

The major difference between this trie you need to implement and the tries you've seen in class is that this trie must allow for a nonconstant range of characters. Most words in the corpus you've been provided will only contain alphabetic characters, but a small percentage will contain numbers, dashes, apostrophes and accents, among other things (e.g. *doesn't* will be a word in your input). A ternary trie can can handle this without any modification. However, a traditional multiway trie will be too memory inefficient to do so. If you want to use a multiway trie, each node, instead of holding a fixed array of pointers to its children, will need to use a dynamic map of pointers to its children which grows as the node acquires new children.

Testing the trie

Every step of the way you'll want to write small tests yourself. If you make a node class, write functions to test all of its methods before moving on. You'll want to test linking two nodes together before linking a full word together. If you write an insert method on your trie, you'll want to test inserting a single word before adding many words. Even though you may not use it for autocomplete, you'll probably want to implement find and/or print methods on your trie so you know everything's looking good. And you'll want to test those methods so you know they're working properly. You'll want to test corner cases - a word that's a subset of another word, two words that are nothing alike, etc. In other words, if you can think of it, you should test it. You can model the setup for your test code off of the `c++` test code in the BST project.

In addition to constantly writing tests, **you should be constantly backing up your code in a *private* repo in a version control system like git.** We will not be able to help students who accidentally delete their code. Enough said.

In `main.cpp` there is a function called `vector<string> getWords(const string fileLocation)` which reads in all the words in a file. Feel free to copy it into your tests, it will provide convenient testing input.

Look at the `example_autocomplete_datasets` for example documents to draw words from. They will not be useful for small unit tests, but should help you test at scale.

In particular, this contains the file `LargeCorpus.txt`, which is a 50Mb subset of this public dataset of Amazon product reviews. This is exciting! You'll be autocompleting using text that real people have generated - typos, slang and all. However, despite the vetting of Amazon and Kaggle, there may be foul language and/or inappropriate content in this dataset. We do not endorse any inappropriate language or content and have given you this dataset to accurately reflect real world data.

Final turn in: Document Generation

In this part of the project you will generate documents to spoof the style of an author. To motivate this, we've provided you with speeches from Trump and Obama (scraped from the weekly addresses recorded at this website). Using those speeches, we want you to generate a new speech for each president using their own (or their speech writers') words!

The idea is as follows: suppose the word "foo" occurs 8 times across all of Obama's speeches. 4 times its followed by the word "bar", 3 times by "baz", and 1 time by a period. When you're generating a speech for Obama and need to follow up "foo" with another word, 4 times out of 8 choose the word "bar", 3 times out of 8 choose the word "baz", and 1 time out of 8 "foo" should just be the last word in the sentence. To generate a full document, you'll simply repeat this process many times until you've the document reaches the specified length.

This algorithm admittedly does not generate very realistic documents. Still, implementing it should give you a feeling for how to take a cool idea and turn it into reality. And, this algorithm has been used as the basis for more complex and effective solutions, so it is a good starting point.

For this part of the project, your requirement is to implement the methods in `DocumentGenerator.hpp`. Read the comments there for more specifics, then continue reading here.

⚠ The comments in `DocumentGenerator.hpp` are *very specific*. Follow them to the letter, so that you do not confuse the autograder.

A few notes:

- Like the Autocomplete assignment, we will be testing for efficiency in addition to correctness.

- Like that Autocomplete assignment, if you print extraneous content to `stdout` you will throw off the grader.
- For the document generator, you'll be required to do more text processing than you did for the autocompleter. Again, that's been described in `DocumentGenerator.hpp`
- Take a look at `tokenize()` and `getWords()` in `main.cpp` - you may want to copy, modify, and use them. I would not suggest modifying `main.cpp` directly as that may break your Autocomplete implementation.
- You may want `dirent.h` for opening directories and you may want the random number generators in `stdlib.h`

You've been provided with sample presidential speeches in the directory `example_dg_datasets` directory which you can use to test your code. In that directory you've also been provided two sample fabricated speeches, one from each president, to give you an example of what your output should look like. Can you guess which example is generated from which president?

As before, you are in control of the design of your code - as long as it passes our tests, you're good. Unlike that assignment, we're not expecting you to use a specific data structure to solve this problem - solve it using whatever data structure(s) you'd like. You may use your code from part 1 of the assignment if you think that's appropriate, but you do not have to.

Document Generator Testing

All the warnings about testing your own code for the autocompleter still apply. Especially the warning that your tests should start small and build up - if your first test uses the large datasets we've given you, that test is way to complex.

There's one additional trick to testing the document generator - your code will need to use randomness, and randomness is hard to test. One approach is to build your tests to run your code many times, aggregate the results, and make sure that they are roughly distributed as you'd expect.

Another approach is to set a *seed* for your random number generator. The generator uses the seed when creating the random numbers - if you start with the same seed (and the same generator function), then the generator will generate the same "random" numbers, in the same order. This way, once you've shown that your code can produce

a specific output with a specific seed, you can write tests to confirm that if you reuse that seed you get exactly the same output.

This type of test is good for ensuring continued functionality of your code. Once you've written the test, you can keep developing your code, and so long as your code keeps passing the test, you know it is still working as you expect it to. Just be careful - if you use seeds during your own testing, make sure your code doesn't use seeds during production - otherwise your code won't actually be random.

Grading

The grading for the autocompleter is out of 25 points, and is broken down into 5 categories:

- The code builds properly - if your code does not build and clean properly using your makefile, you will score 0 points.
- 3 points for no memory leak. Memory leaks are notoriously hard to debug if not caught immediately, so test your code frequently, and create checkpoints you can revert to regularly.
- 4 points for working correctly on empty/tiny corpuses
- 6 points for working correctly on a corpus of size 50KB
- 6 points for working correctly on a corpus of size 50MB
- 6 points for responding quickly to many queries on the large corpus.
- If you miss points on the checkpoint, you can gain ½ of them back at the final submission.

The code for the document generator is out of 20 points, and is broken down into 3 categories:

- The code builds properly - if your code does not build and clean properly using your makefile, you will score 0 points.
- 2 points for no memory leak. The same warning about memory leaks as above.
- 8 points for generating single words correctly
- 10 points for generating long documents correctly and efficiently.

The document generator tests use randomness. If your score fluctuates on different runs, assume that's because a bug in your code is being uncovered at random.

Please note that you do not need to have your name or PID in the turn in files. In fact, by adding your name/PID to these files, they will be exposed to a 3rd party server (so feel free to remove them).

Submitting Your Checkpoint and Final PA

As in the last PA, there is a `PA2Checkpoint` assignment on gradescope. The `PA2FinalAutocomplete` and `PA2FinalDG` assignments will be opened up closer to the final deadline.

⚠ **Make sure you submit the same zip file** to `PA2FinalAutocomplete` and `PA2FinalDG`. We will be grading them separately, but you need to hand in the same code.

⚠ **If you're doing pair programming** *Only* ***one*** of you should make a submission. Do not make separate individual submissions. Make sure to add your PA partner to your final submission. And make sure to submit the Pair Programming sign-up form, even if you partnered with the same person last PA.

Academic Integrity and Honest Implementations

We will hand inspect, randomly, a percentage of all submissions and will use automated tools to look for plagiarism or deception. **Attempting to solve a problem by other than your own means will be treated as an Academic Integrity Violation.** This includes all the issues discussed in the Academic Integrity Agreement, but in addition, it covers deceptive implementations. For example, if you use a library (create a library object and just reroute calls through that library object) rather than write your own code, that's seriously not okay and will be be treated as dishonest work.

Getting Help

Tutors in the labs are there to help you debug. TA and Professor OH are dedicated to homework and/or PA conceptual questions, but they will ***not*** help with debugging (to ensure fairness and also so students have a clear space to ask conceptual questions). Questions about the intent of starter code can be posted on piazza. Please do not post your code to piazza either publicly or privately - debugging support comes from the tutors in the labs.

Format of your debugging help requests

At various times in the labs, the queue to get help can become rather long (all the more reason to start early). To ensure everyone can get help, we have a 5 minute debugging rule for tutors in that they are not supposed to spend more than 5 minutes with you before moving onto a new group. Please respect them and this rule by not begging them to stay longer as you're essentially asking them to NOT help the next group in exchange for helping you more.

5 minutes?!

Yes, 5 minutes. The job of tutors is to help you figure out the *next step in the debugging process*, not to debug for you. So this means, if you hit a segfault and wait for help from a tutor, the tutor is going to say “run your code in gdb, then run bt to get a backtrace.” Then the tutor will leave as they have gotten you to the next step.

This means you should use your time with tutors effectively. Before asking for help, you will want to already have tried running your code in gbd (or valgrind, depending on the error). You should known roughly which line is causing the error and/or have a clear idea of the symptoms. When the tutor comes over, you should be able to say:
What you are trying to do. For example, “I’m working on Part 1 and am trying to get the insert method in the BST to work correctly.”
What’s the error. For example, “the code compiles correctly, but when I insert a child in my right subtree, it seems to lose the child who were there before.”
What you’ve done already. For example, “I added the method which prints the whole tree (pointers and all) and you can see here <point to screen of output before and after insert> that insert to the right subtree of the root just removes what was on the right subtree previously. But looking at my code for that method, it seems like it should traverse past that old child before doing the insert. What do you suggest I try next?”

Acknowledgements
Special thanks to Jonathan Margoliash, Sander Valstar, and Dylan McNamara for building this assignment.