CSE100 - Winter 2018

Programming Project 2

Checkpoint Deadline: 11:59pm on Tuesday, 2/13 (not slip day eligible)
Final Deadline: 11:59pm on Tuesday, 2/20 (slip day eligible)

Overview
In this project, you will be asked to complete two major 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 part 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. And please 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

Part 1:
For PA2, you are given the top level interface for the data structure 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:

```
Autocomplete.hpp        Makefile    grade_pa2_checkpoint.py
AutocompleteUtils.hpp   TestCases   main.cpp
```

Part 2:
If you first ran “getprogram2” after part 2 was released, you do not need to do anything else to retrieve the part 2 code.

If you first ran “getprogram2” before part 2 was released, then do the following: run “getprogram2 <newDirectory>”. That will get you all the files for both parts of project 2 from scratch. Copy the new files for part 2 into the directory you’ve been using for part 1. Here’s a list of the files you should copy over:

- DocumentGenerator.hpp
- main2.cpp
- PresidentialSpeeches
- Grade_pa2_post_check.py
- Makefile** (We’ve heavily edited the Makefile from what you got in PA1 - it now compiles part 1 more cleanly in addition to compiling part 2. Even though you may have edited the Makefile for part 1, start with this new one we’ve given you, and port your changes over.)
- AutocompleteUtils** (only change was to add imports <vector> and <iostream> to meet newer, more rigorous compilation requirements)

Part 1: Implementing a string autocompleter

This part of the project will be due at the checkpoint.

In the first part of the project you will write the data structure for a string autocompleter. The idea here is that 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 underlies the autocompletion 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 the Autocomplete.hpp file using a either Multiway Tries or Ternary Tries. Read the Autocomplete.hpp file carefully to understand what is required. You don't need to look at any other files just yet, and you should never need to look at (and shouldn't ever modify) the main.cpp, grade_pa2_checkpoint.py files or the GradingScriptFiles directory.

We have only provided you with the interface you need to implement. You will need to use a Trie to implement this interface. You may use the following parts of the STL:

- vector
- unordered_map (which STL implements as a HashMap),
- pair
- The sort method, from <algorithm>
- All of the queues and stacks

Outside datastructures not listed above are not allowed in the project. In particular, you may not use std::map. Using a disallowed data structure qualifies as an academic integrity violation. If you're not sure, ask. Also - you are not required to use these tools, not all good solutions will need them.

See below for advice on programming the data structure.

Know that we've not provided you with any reasonable tests. You should write many of your own. The grading script will not be sufficient for testing your code. As a number of students were clearly running the grading script rather than writing their own tests, **tutors will not help you debug your code based on grading script output without clear evidence of substantial testing on your own part.** So if you have not written

tests and come to the tutors for help, they will tell you to write tests before continuing to help you. See below for advice on testing.

This assignment is, in part, designed to challenge you to write a functioning piece of software from end to end, unlike previous assignments (such as the BST) which have given you detailed outlines to work from. This is what lots of programming is like, so we want to help you get comfortable with that now. Good luck.

Part 1: Trie data structure

We recommend you implement the autocompleter using a trie. 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.

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 .cpp and .hpp files from your directory when grading your code. **One obvious caveat, you have to have written, from start to end, any code included in your turn in.** If you add additional files, you will need to modify the `Makefile` to compile those files too.

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.

There are two differences between the trie you need to implement to solve this problem and the tries you've seen in class. The first 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 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 every node in a traditional multiway trie contains an array of pointers of a large fixed size -- if every node contains space for a pointer for each digit just in case that letter in that node is followed by a digit -- you may run out of space or your performance will suffer. If you want to use a multiway trie, each node will instead need to use a dynamic array or hash map of pointers to its children which grows as the node acquires new children. This way only the nodes which are followed with an apostrophe have allocated space for a pointer to an apostrophe.

The other difference is that you must store word frequencies in your trie. The simplest way to do this is to, at every word node, store not just an indicator that the node is the end of a word, but also store a count of how many times that word appeared in your training corpus.

Part 1: Testing

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 version control system like git. We will not be able to help students who accidentally delete your code. Enough said.

To help you test, we've provided a function in `AutocompletionUtils.hpp` called `vector<string> getWords(const string fileLocation)` which you can use to read in all the words in a file. This will provide you convenient testing input for your Autocomplete constructor. Look at the `TestCases/*Corpus.txt` files for examples to draw words from. A few notes:

- The files already in `TestCases` aren't necessarily the most friendly for your testing. You may want to create your own files for testing.

- Please *do not* modify the files in `TestCases`, as that will throw off the results you see when you run the grading script.

Your code will need to work at scale. The final challenge for your code is to efficiently operate on a 50 MB text document (called `LargeCorpus.txt`). We've used a 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 Amazon does, there may be foul language and/or content in this dataset. We neither endorse nor take responsibility for that.

One last piece about testing. Although we've given you the grading script we'll use, we haven't given you the inputs we'll use when grading. They will not be sufficiently larger than the Amazon data set, but they will be different. This means the onus on testing still falls on you. You'll want to be quite confident your code is correct before submitting and that means making your own tests. Our *strongest recommendation* for you is to do all your testing yourself and, only when highly confident it is all correct, use the grading script. Should the grading script succeed, you'll feel all the more confident your code is correct. In contrast, using the grading script before testing your own code annuls the utility of the grading script and makes it all the more likely you'll receive a different grade than expected.

Part 2: 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. Then when you're generating a speech for Obama, if you need to follow up "foo" with another word, 4 times out of 8 it should be "bar", 3 times out of 8 it should be "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 filled the document reaches the specified length.

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.

A few notes:

- Your code must build with the `make part2` command and must clean with the `make clean` command. We've provided you with a new and improved makefile to get you started.
- In this part of the project, you'll be required to do a bit more text processing than you did in part 1. Again, that's been described in `DocumentGenerator.hpp`
- You may use anything from STL for this part of the project. Specifically, you may want `dirent.h` for opening directories and you may want the random number generators in `stdlib.h`

You've also been provided with sample speeches in the directory `PresidentialSpeeches` which you can use to test your code. Please don't change these files - they're used in the grading script. 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? You are not required to use either of those resources for completing the project.

Like in part 1, you are in control of the design of your code - as long as it passes our tests, you're good. Unlike in part 1, 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.

Part 2: Testing

All the warnings about testing your own code from part 1 still apply. 'Nuff said.

There's one additional trick to testing in part 2 - 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 part 1 of the project 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 a tiny corpus
- 6 points for working correctly on a corpus of size 50KB
- 7 points for working correctly on a corpus of size 50MB
- 5 points for responding quickly to repeated queries quickly on the large corpus.
- If you miss points on the checkpoint, you can gain ½ of them back at the final submission. You can get all the bonus points at the final submission, that does not need to be implemented by the checkpoint.

The code for part 2 of the project 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 documents correctly.

Run the grading scripts (`grade_pa2_checkpoint.py` and `grade_pa2_post_check.py`) to see a sample score. If your code is nondeterministic, then we cannot guarantee that the score you see will be the score you get. The final grading will use the same logical tests, but will be run on different datasets than the

ones given to you in `TestCases` and `PresidentialSpeeches`, with different expected outputs.

Lastly, we will be running tests not seen in the grading script to look for the use of unauthorized libraries or data structures. In other words, if you try to solve this using another data structure from the STL or from the web and not build your own, you will receive a zero on the assignment and possibly face an Academic Integrity Violation.

Your files that will be collected for grading are: the `Makefile`, and any `*.cpp` and `*.hpp` files in your project's directory except for `main.cpp` and `main2.cpp`. Everything else will be ignored.

- 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).

Be sure to test your code on ieng6. That's where it will be run and graded.

Submitting Your Checkpoint and Final PA

Submission command: `bundlePA2`
If you are submitting with a partner, make sure that you two have registered that you two are working together.
You can submit as many times as you like before the deadline: only your last time running will be counted. If you submit the assignment within 24 hours after the deadline (even a minute after the deadline) you will be charged a slip day. If it is more than 24 but within 48 hours, you will be charged 2 slip days. If you've already used some of your slip days for the previous PA, you won't be able to use those slip days now.

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, Dylan McNamara, Niema Moshiri, Christine Alvarado, and Paul Kube for creating the base on which this assignment is built.