Homework 3
Editors

Assigned: Friday, January 22, 11:00AM

Due: Friday, January 29, 11:00AM (Hard Deadline)

Submission Instructions

Submit this assignment on Gradescope. You must submit every page of this PDF. We recommend using the free online tool PDFescape to edit and fill out this PDF. You may also print, handwrite, and scan this assignment.

Even if there are not questions to answer on the first page, please still submit every page of this PDF.

There may multiple answers for each question. If you are unsure, state your assumptions and your reasoning for why you think your answer makes sense.

Readings

These readings are not required, not graded, and I will not ask any questions on them. They are simply articles that I think may be worth your time.

Interrupting Developers and Makers vs Managers

In lecture we spent a little time talking about the mental model of programmers. As computer science grows, there is a growing body of research and wisdom in how to maximize the efficacy of individual programmers. One of the key revelations is that programming is a creative process. It requires time to "page in" what you are working on. As a result, even small interruptions (read: text messages) can be far more costly than you may realize.

`http://thetomorrowlab.com/2015/01/why-developers-hate-being-interrupted/`
Why developers hate being interrupted, by Derek Johnson at The Tomorrow Lab.
`http://www.paulgraham.com/makersschedule.html`
Maker's Schedule, Manager's Schedule, by Paul Graham, co-founder of Y Combinator.

Once you are in a job environment and meetings become a regular thing, one of the best tricks I learned is to schedule a few 4-hour meetings with yourself throughout the week. It both prevents others from interrupting you (your calendar shows you as busy) and lets you mentally prepare for a good, reliable work session. (A similar alternative, some find pomodoro helpful)

Software Engineering in the Real World

`http://blogs.msdn.com/b/peterhal/archive/2006/01/04/509302.aspx`
What Do Programmers Really Do Anyway?, by Peter Hallam, then-Microsoft Developer

> *Why is 5 times more time spent modifying code than writing new code? The answer is that new code becomes old code almost instantly. Write some new code. Go for coffee. All of sudden you've got old code.*

This post gives a nice perspective on what enterprise software engineering is really like. School projects are misleading. Rarely in life will you be faced with a spec and a blank text file. Far more often you are building on or improving something that came before you and will exist long after you.

`http://www.joelonsoftware.com/articles/fog0000000069.html`
Things You Should Never Do, Part I, by Joel Spolsky

> *It's harder to read code than to write it.*

Building on the previous, rarely in life are you handed a spec and a blank text file, often in life you are handed a spec and a pile of (seemingly) spaghetti code that does at least some of it. This article discusses the hidden value of the built-up institutional knowledge and the high value of working code.

When was the last time you saw a hunt-and-peck pianist? -Jeff Atwood, co-founder of Stack Overflow and Discourse.
Some extras on the value of being a good typist and ditching the mouse for the keyboard.

1 The Basics

To kick off lecture, we talked about one of the earliest text editors, `ed`. While `ed` is rarely used today, its cousin `sed` ("stream editor"), can often be a useful tool for doing quick, simple (or remarkably complex) substitutions on files.

`sed` is a pretty powerful tool, with a lot of features. By far its most used feature, however, is "s commands".

Below is a transcript of a terminal session. Fill in the blanks to generate the output as shown:

```
[1]: cp input.txt backup.txt
[2]: cat input.txt
one two three
one two three three two one
One more here
What about ONE oNe oneone oneoneone oNeOnEoNe?

[3]: sed ________________________________ input.txt
zero two three
zero two three three two one
One more here
What about ONE oNe zeroone oneoneone oNeOnEoNe?

[4]: cat input.txt  # Notice that the input file hasn't changed
one two three
one two three three two one
One more here
What about ONE oNe oneone oneoneone oNeOnEoNe?

[5]: sed ________________________________ input.txt
zero two three
zero two three three two one
zero more here
What about zero oNe oneone oneoneone oNeOnEoNe?

[6]: sed ________________________________ input.txt
zero two three
zero two three three two zero
zero more here
What about zero zero zerozero zerozerozero zerozerozero?

[7]: sed ________________________________ input.txt # Careful, two things change here
[8]: cat input.txt
zero two three
zero two three three two zero
One more here
What about ONE oNe zerozero zerozerozero oNeOnEoNe?
```

Explaining a very annoying "gotcha"

Instead of command `[7]`, you might be tempted to write something like:

```
    sed [...] input.txt > input.txt
```

However, you will be disappointed by the outcome. Let's play with `cat` because it's a little easier:

```
[1]: echo one > 1 ; echo two > 2
[2]: cat 1 2 | cat > 3
[3]: cat 1 2 | cat > 1
```

What is the contents of 3? What is the contents of 1? Give a reasonable guess for why the contents of 1 and 3 are different.

2 Text-based editors (Basics)

By default, Ubuntu ships with a minimal `vim` and no `Emacs`. If you haven't already, start by installing each:

```
sudo apt-get install vim-gnome emacs
```

One recurring theme in this class is "neither is better". For text editors, this means you invariably will encounter both `vim` and `Emacs` environments in your career. While there is little benefit in becoming an expert in both, it's very valuable to have basic skills in the most common editors as you will encounter them.

This question simply asks you to list how to do what I consider the minimum skills in each editor. While I cannot force you to open a text editor and try each of these to build some muscle memory, I can say that having these basics at your fingertips is good, and in a setting such as a final exam, it would be perfectly reasonable to expect you to recall all of these correctly.

> Just starting out with these? Never used one before? Try the built-in tutorial programs. Run the program `vimtutor` for vim, or open Emacs and then hit `Ctrl-h` followed by `t` for Emacs.

For each of the following, assume the editor was just opened (e.g. you have just typed `vim hello.c`).

vim	Emacs
Save file	Save file
Quit **without** saving edits	Quit **without** saving edits
Save and quit (two commands fine)	Save and quit (two commands fine)
Add the text "Hello World" at the current location	Add the text "Hello World" at the current location
Find the text "TODO"	Find the text "TODO"
Find the next "TODO"	Find the next "TODO"

3 Text-based editors (The Fun Stuff)

People often complain about how difficult the simple things are using text-based text editors. Indeed, Notepad can do everything from the previous question (and Notepad is much easier to use). In this question we explore some of the things that Notepad (and nano, and gedit) cannot do.

I encourage you to play around with things as you work through this question. The intent is to expose you to features you may not have been aware of along with some context for why they are useful.

There are many things to try here. **For credit on the homework, you only need to choose any 5.** That said, I encourage you to try all of these.

For this question, you may choose `vim` or `Emacs`, whichever you are more comfortable with.

To start, grab copies of two files full of code:

```
# Examples from http://web.mst.edu/~price/cs53/code_example.html
wget http://web.mst.edu/~price/cs53/fs11/workDecider.cpp
wget http://web.mst.edu/~price/cs53/KatsBadcode.cpp
```

1. Sometimes you will come across some ugly code. Your editor can help make it better. Open `KatsBadcode.cpp`. Among other issues, the really bad tabbing makes this hard to read.
   Describe how to automatically fix the whitespace for the whole file.

2. Editing one file is nice, but often it's really useful to compare files side-by-side (source and headers, spec and implementation).
   Describe the command sequence to create another window side-by-side with your current window, switch to it, and then open a file in that window.

   Try playing around with these two views, copy/paste code between them, bind them so they scroll together, resize them, add more splits.

3. Editors also have pretty nice integration with compilation tools. With `KatsBadcode.cpp` open, try typing `:make KatsBadcode` in `vim` or `M-x compile<enter>make KatsBadcode` in `Emacs`.[1]

   > First cool thing that happened: Yes, you can run `make` *without* any Makefile anywhere. We'll cover how that happened during build-system week.

   Building `KatsBadcode.cpp` will fail with several errors.
   Describe how to navigate between compilation errors automatically.

4. While C-style languages support `/* block comments */`, others such as Python have no block comments and require you to put a `#` at the beginning of every line.
   Describe how to efficiently comment out a large block of Python code.

[1] Descriptions of `Emacs` commands are usually written as `C-c` or `M-x`, which mean "Ctrl+C" or "Meta+x" respectively. Meta is often mapped to the escape and/or alt key, `M-x` means press Escape and then x or hold alt and press x.

5. (This one is actually about the terminal emulator): The normal keyboard shortcuts to copy/paste are Ctrl-C and Ctrl-V. These do not work in the terminal, however.
   Explain what Ctrl-C does in a terminal.

   Describe how to copy/paste using the keyboard in a terminal.

6. The default cut/copy/paste(/put/yank/kill) operations do not put text in the system clipboard (that is you cannot copy/paste into/from other applications). **Describe how to copy/paste a region of text to/from the system clipboard using your text editor.**

7. Many times you have a repetitive task that you need to do say 10 or 20 times. It's too short to justify writing a script or anything fancy, but it's annoying to type the same thing over and over again. As example, reformatting the staff list to a nice JSON object:

```
                                staff = [
Pannuto,Pat                        {"first": "Pat", "last": "Pannuto"},
Darden,Marcus                      {"first": "Marcus", "last": "Darden"},
Smith,Max                          {"first": "Max", "last": "Smith"},
Snider,David                       {"first": "David", "last": "Snider"},
Khan,Waleed                        {"first": "Waleed", "last": "Khan"},
Terwilliger,Matt                   {"first": "Matt", "last": "Terwilliger"},
Chojnacki,Alex                     {"first": "Alex", "last": "Chojnacki"},
Hussein,Mo                         {"first": "Mo", "last": "Hussein"}
                                ]
```

   Editors support the record and reply of *macros*. With macros, you can start recording, puzzle out how to turn one line into the other, and then simply replay it for the rest. As a general rule, do not bother trying to be optimal or efficient, just find anything that works. Try starting with the column on the left and devising a macro to turn it into the column on the right.
   Describe how to record and replay a macro. *(you do not need to include the contents of your macro)*

8. Similar to how ~/.bashrc configures your shell, a ~/.vimrc or ~/.emacs file[2] can configure how your editor behaves. Here are some excerpts from the staff's configuration files:

```
.vimrc:
set number     " double-quote means comment in vimscript; this turns on line numbers
map ; :        " this line and the next makes ; act like : so that you don't have to
noremap ;; ;   " hit shift all the time, typing ";;" will act as ";" used to

.emacs:
; line numbers - in Emacs, ; means a comment
(global-linum-mode t)
(setq linum-format "%d ")
; Changes all yes/no questions to y/n type
(fset 'yes-or-no-p 'y-or-n-p)
```

Add something useful not listed above to your editor's configuration file.
Describe what you added and why.

9. Some final little things

   (a) Editors understand balanced ()'s and {}'s. **Describe how to jump from one token to its match, such as from the opening { on line 29 of KatsBadcode to its partner } on line 58.**

   (b) Now imagine if you are editing code and you determine you can remove an if block, for example removing the if on line 28 of KatsBadcode and running its body unconditionally. You need to fix the indentation level of all 20 lines of the body.
   Describe how to change the indentation level of a block of code.

   (c) Sometimes it's useful to run quick little commands. For example, your code needs to read from a file, but you can't remember if it's called sample_data.txt or sampleData.txt.
   Describe how to run 'ls' directly from within your editor.

[2] These files do not exist by default, you have to create them.