Homework 8
Practical Regexes and Synthesis of Rich Commands

Assigned: Friday, October 28, 11:00AM

Due: Saturday, November 5, 10:00PM (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.

1 Playing with words

By default, Ubuntu ships with a few dictionaries. We can find them in the `/usr/share/dict` directory. If we head into that directory, we can see (`wc -l *`) that these are not small lists. American English comes in just shy of 100,000 words.

An interesting file is `cracklib-small`. This is a word list of around 50,000 common passwords. We can use the `grep` utility to search through this file quickly to see if our password is in the file. For example if my password is "password", then `grep password cracklib-small` tells me that I've picked a bad one, but `grep sup3rs3cure cracklib-small` tells me that may be a better choice.

Remember `ed`? Wikipedia tells us that `grep`'s name, "comes from the ed command g/re/p (**g**lobally search a **re**gular expression and **p**rint)". Let's give that a go eh? Run `ed cracklib-small` and try the command `g/password/p`. Look familiar?

Thus far, with `ed` in lecture and `sed` in the previous homework, we've only used simple regular expressions, namely things that match the whole string we're searching for. Regular expressions can be far more powerful, however.

Try the following commands:

```
> grep password cracklib-small
> grep pass cracklib-small
> grep ^pass cracklib-small # That's a caret, shift+6
> grep pass$ cracklib-small
> grep ^pass$ cracklib-small
> grep pass^ cracklib-small
> grep '$pass' cracklib-small # Why do we need quotes here?
> grep p.ss cracklib-small
> grep ^..th$ cracklib-small
> # Play with some others of your own design
```

Aside:
Programs that want to check your spelling will use the file `/usr/share/dict/words`. Notice, however, that when we type `ls` in this directory, the `words` file shows up in teal, indicating that it's a symlink. Recall that a symbolic link is a way to make something that looks like a real file, but is actually just a pointer to another file.

We can follow this pointer, however, to see what the actual file is. Type `ls -l words` to see what it actually points to, in this case `/etc/dictionaries-common/words`. It turns out that this too is a symlink however! If we then type `ls -l /etc/dictionaries-common/words` we see that it points back to `/usr/share/dict/american-english`, which is finally a real file.

These little circles come up sometimes for compatibility reasons. Some programs expect to find the word list at `/usr/share/dict/words` while other programs expect it at `/etc/dictionaries-common/words`. Using symlinks we can make all of these point to the same file. We can also easily change the language used for spellchecking by all programs simply by changing what the last symlink points to.

To shortcut this whole operation, we can use the `readlink` utility. Try the command `readlink words`. Now try `readlink -f words`. Does what each of these commands are doing make sense?

Simple primitives build powerful features.

What does a carat (ˆ) mean in a regular expression?

What does a dollar sign ($) mean in a regular expression?

What does a period (.) mean in a regular expression?

Sometimes it's more interesting to know how many matches there are, rather than the exact matches themselves. `grep` provides the `-c` (count) flag for this case. For example `grep -c password cracklib-small` tells us there are 3 lines that have "password" in them, but there are (`grep -c pass cracklib-small`) 60 lines with "pass".

There are 810 lines in cracklib-small that are exactly 3 characters long. Give a command you could run to learn that number:

A "count" flag is a very common flag and very useful flag shared by many utilities, not just grep.

Groups are another powerful feature. Try

```
> grep ^p[ao]ss cracklib-small
> grep ^p[aeo]ss cracklib-small
> grep ^p[a-z]ss cracklib-small
```

There are 766 lines in cracklib-small that are exactly 3 *letters* long. Give a command you could run to learn that number:

Command-line tools really start to shine when you string them together. Try running the following command:

```
> for vowel in a e i o u; do echo -e "$(grep -c ^$vowel cracklib-small) \t $vowel"; done | sort -rn
```

Try playing around with this command a bit. Remove the flags to `sort`, remove `sort` altogether, replace the body of the for loop (everything between `do` and the `;`) with just `echo $vowel`.

In plain English, what is this command doing?

If you would like more practice with regular expressions, check out this online tutor. One tricky thing, there are many different "dialects" of regular expressions. Standard `grep` is very fast but trades off speed for limited features. The tutor teaches what `grep` calls "Extended Regular Expressions". For example, Lesson 6 teaches quantifiers, `grep .z{2}` will not work, but `grep -E .z{2}` will.

We have one more tool we need to learn about before with can get to the grand finale, and that's `uniq`. First, run this command:

```
> for i in $(seq 20); do echo $(($RANDOM % 5)) >> /tmp/numbers; done
             |                 |   |
             |                 |   \- this is a built-in bash "variable"
             |                 |      try just "echo $RANDOM", what happens?
             |                 |
             |                 \- $(( ... )) lets us do math in bash, in this
             |                    case we're doing a mod as the range of
             |                    $RANDOM is quite large (0 - 32767)
             |
             \- seq is a program to print a sequence of numbers. Try just
                typing "seq 10", "seq 10 20", and "seq 10 2 20"

                You can also use the built-in bash range idiom:
                > for i in {1..10}; do echo $i; done
                to accomplish the same task, but I find that harder to remember the syntax
```

Check out the contents of the file `/tmp/numbers`. Do you understand what that command did? Now try running `uniq /tmp/numbers`. What does the `uniq` command do? Not sure? Try looking at the output side-by-side:[1]

```
> diff -y /tmp/numbers <(uniq /tmp/numbers)
```

The man page for `uniq` is short and simple. It is a good man page. Try giving it a read and playing with some of the other options for `uniq`.

[1] The `<( ... )` redirect lets you run a command and use its output for something that expects a file, similar to how `$( ... )` lets you run a command and use its output as text. It lets you skip creating a temporary file. Otherwise you could run

```
> uniq /tmp/numbers > /tmp/uniq-output
> diff -y /tmp/numbers /tmp/uniq-output
```

and the result would be the same.

Now, for the hard part: Solving EECS 281 problems in 100 characters or less.

We're going to combine everything we've learned so far to answer some powerful queries. For each question, you need to come up with a string of commands stuck together with pipes that will return the answer to the question. The correct answer is also given for each question so that you can check your work.

*Hint: While we often use the `cat` utility to just print the contents of a single file, it's real purpose is for con**cat**enating multiple files. How might concatenating files be useful in conjunction with these other utilities?*

Hint: The way to approach this problem (and many problems) is to build it up from small pieces. String together two commands until they give you want you want, then add a third, etc.

How many words are in only the `british-english` word list or the `american-english` word list but are not in both? [Answer: 3481 words not in common]

If for some reason there is no `british-english` file, run `sudo apt-get install wbritish`

How many entries in the easily crackable password list (`cracklib-small`) are English words (are in `american-english`)? [Answer: 40599 words in common]

Optional Related Readings

Quick and light, I particularly recommend the first one. It's a good anecdote for software and system design.

More shell, less egg – This is a fun blog story of when even Donald Knuth sometimes gets things wrong.

"Why GNU grep is so fast" – This is a nice example of the importance of efficient algorithm design and how an implementation can benefit from a deep understanding of the underlying system.

2 Pulling Some Pieces Together

Head back to the git repository you created for week 2's homework.

Week 2's homework had you blindly run the command

```
> grep ';' p2.h | grep -v ' \*' >> p2.cpp
```

Now try running

```
> grep ';' p2.h | grep -v ' \*'
> grep ';' p2.h | grep ' \*'
> grep ';' p2.h | grep -v ' \*' | grep filter
> grep ';' p2.h | grep -v ' \*' | grep -v filter
```

In plain English, dissect the command `grep -v ' \*'` (notice the space)

(Run `make` if you haven't already)

Suppose you wanted to change the `insert_list` function, so you wanted to find all of the places it's called. One approach would be:

```
> grep insert_list *
```

which searches all files for the string "insert_list". Compare that search, however, to

```
> git grep insert_list
```

Do you see how the two searches differ?

Does 'git grep' search untracked files? How do you know?

Does 'git grep' search new files that have been staged but not committed? What test could you quickly run to find out?