CSS 390a - Autumn 2017 Midterm Exam Study Questions

Here are some study questions for the midterm. They are not calibrated for time or difficulty, but will help you exercise the skills that will be tested.

Solve the following problems using any technique. For best results, give multiple solutions using different combinations of shell script(s), sed, awk, Perl, or Python (if already familiar). Use your imagination to come up with variations and similar problems.

Some problems can be solved by sorting or using dictionaries and/or lists. Try it both ways.

1. Download a randomly-selected C or C++ project from github or similar repository, e.g. Linux kernel, Bash, nethack (game). For each **`.h`** file, list the header and source files that **`#include`** it.

2. Write a script that creates a directory tree of paths with spaces in them (use **`mkdir`** to create the directories and **`touch`** to create a zero-length file. Both directories and files should have spaces in them.

a) Write a script to rename the paths to remove spaces or replace them with underscores. Make sure no file gets "overwritten" by another on (e.g. both "**`ab c`**" and "**`a bc`**" map to **`abc`**). Make sure your data generator script has cases that that test this.

b) Write a script that generates a script containing the required **`mv`** commands with the control logic to avoid clobbering files. Verify the generated script works correctly.

3: Using the **`http://courses.washington.edu/css390/2017-q4/midterm/ICAO_airports.txt`** TSV data file,

a) Find all the large airports (ident, name, municipality, and ISO region) in the U.S. How many are there?

b) Display the large airports grouped by state.

c) Reformat to list the state (ISO region code), followed by the airports of the state (ident, name, municipality) one per line, followed by a blank line (to separate states).

d) Same as c, but instead of a blank line to separate states, indent the airports and leave the state at the left margin.

e) Using the output from (c) or (d), refomat to give, for each state, the state ISO region code, followed by a comma-separated list of the ident the large airports of the state as a comma-separated list, on the same line. Sort by state. e.g.

```
US-AK   [PAFA, PANC]
US-AL   [KBHM, KHSV, KMGM, KMOB]
US-AR   [KFSM, KLIT]
US-AZ   [KLUF, KPHX, KTUS]
US-CA   [KBAB, KEDW, KLAX, KOAK, KONT, KSAN, KSFO, KSJC, KSMF, KSNA, KSUU, KVBG]
...
```

f) Using the output from (e), refomat to go back to TSV, or the format of (c) or (d).

4: The data file **http://courses.washington.edu/css390/2017-q4/midterm/kern.log.txt** contains log messages regarding packets dropped by firewall rules. The **/etc/services** file shows the services assigned to internet ports.

a) Extract the protocol (TCP or UDP) and source/destination ports (PROTO, SPT, DPT) from the log file. Display the **services** entry for each port/protcol combination in the log. Collect the entries for each service to give a count of the number of packets.

b) Collect and count the source and destination IP addresses.

5: You have route data for an airline consisting of a file containing one airport per line, followed by an n-length string consisting of 0s and 1s, where 1 in position *i* means there is a route from that airport to destination airport *i*. You wish to convert this to a file consisting of pairs of airport A B (tab-separated) where there is a flight from A to B.

HINT: if you split a string into an array of single-character strings, you can use the array index to get the destination city, e.g. **$city[2]** is BLI.

Your logic could look something like this:

```
for each line
    append field 1 to city array
    append field 2 to destination array
for i from 0 to size of city array - 1
    for j in from 0 to length of destination[i] - 1
```

```
            if character j in destination[i] is "1"
                print city[i] city[j]
```

Sample input:

```
SEA    011111
PDX    101011
BLI    110000
GEG    110000
YVR    100001
SFO    110110
```

Sample output:

```
SEA    PDX
SEA    BLI
SEA    GEG
SEA    YVR
SEA    SFO
PDX    SEA
PDX    BLI
PDX    YVR
PDX    SFO
BLI    SEA
BLI    PDX
GEG    SEA
GEG    PDX
YVR    SEA
YVR    SFO
SFO    SEA
SFO    PDX
SFO    GEG
SFO    YVR
```

6: Your eccentric writer friend is working on his *magnum opus*: a treatise on the larch tree (okay, he's really cribbing from wikipedia, but he's mostly harmless). He is typing the text in raw HTML, with one file per chapter, imaginatively named **`chapter01.html`**, **`chapter02.html`**, etc.

The text looks like this:

```
<a name="ch1">

<h1 class="chapter">Chapter 1: Species and Classification</h1>

<p>There are 10–15 species; those marked with an asterisk (*) in the list below are not
accepted as distinct species by all authorities...</p>

<a name="ch1_1">

<h2 class="section">1.1 Eurasian</h2>

<a name="ch1_1_1">

<h3 class="subsection">1.1.1 Northern, Short-Bracketed</h2>

<p>In this section we list 2 species from the northern Eurasian region blah blah</p>

<a name="ch1_1_2">

<h3 class="subsection">1.1.2 Southern, Long-Bracketed</h2>

<p>More long-winded verbiage</p>

<p>etaoin shrdlu<p>

<a name="chap1_2" />

<h2 class="section">1.2 North American</h2>

<p>We have three North-American Larch species...</p>
```

He wants you to generate the table-of-contents page, imaginatively named **`contents.html`**.

Sample output:

```
<ul class="contents">

<li><a href="chapter01.html#ch01">1. Species and Classification</a></li>

<li><a href="chapter01.html#ch01_01">1.1 Eurasian</a></li>

<li><a href="chapter01.html#ch01_01_01">1.1.1 Northern, Short-Bracketed</a></li>

<li><a href="chapter01.html#ch01_01_02">1.1.1 Southern, Long-Bracketed</a></li>

<li><a href="chapter01.html#ch01_02">1.2 North American</a></li>

...

</ul>
```

7. Using **wget** or **curl**, download the **cnn.com** web page and extract the links (the strings **href="***link***"** inside the **<a...>** anchor tags. Hint: it may be easier to break the file in to lines and keep only the lines that contains anchor tags.

8: Your favorite cat pictures web site, also your employer, collects log files for every request served. The logs are text files, with one record per line. To simplify things, records consist of tab-separated fields:

1. Unix time as a 64-bit number in hexadecimal format
2. IP address of the request
3. cookie (64-bit number in hexadecimal)
4. URI of the request (e.g. http://www.cutekittypictures.com/...)

The log files are stored under **$LOGROOT**/ and use the naming convention ***yyyy-mm-dd*.log** and there is one log file per day. There may be other non-log files in the directory too.

Your boss wishes to identify the top 100 down-loaders of distinct images (i.e. just the **.jpg** and **.png** files, not **.html** or other files) over the last three days for some special promotion. That is, someone who just downloads the same image over and over is not eligible. The cookie is considered sufficient to uniquely identify a user.

Write a script to perform the logs analysis. Assume the site gets up to 20,000 hits per day, which makes O(N log N) solutions tractable (i.e. sorting).

9: You are responsible for a web site that serves pictures of the feline variety. To verify that the site is up and running, you wish to implement a "black box probe". Write a script, called **prober** that takes a URI (web address) and a regular expression pattern as command-line arguments.

- The prober should download the web page and verify that it contains the pattern, every 5 minutes.
- For each attempt, **prober** should append to file **prober_log** the current date/time and whether the probe was a success or failure.
- If a probe fails, wait one minute and try again. If it fails a second time, send an email to **oncall@cutekittypictures.com** with the the subject line "ALERT" and message "Prober failed.".

- Continue probing at 1-minute intervals until you have a successful probe then go back to 5-minute intervals.

`prober` may call additional support scripts if necessary. [Note that a realistic black box prober would be a little more sophisticated, but your pointy-haired boss told you that you only had 20 minutes to get it done].

HINT: this problem essentially divides into sub-problems:

1. performing a single probe and determining whether it is successful
2. control logic for counting/tracking successes/failures and what to do about it

You don't require any counters if you organize your control logic like this:

```
loop forever
    while probe is successful
        log success
        wait 5 minutes
    log failure
    wait 1 minute
    if probe is failure
        log failure
        send email
        wait 1 minute
        while probe is failure
            log failure
            wait 1 minute
    log success
```

Alternatively, using a counter variable:

```
failure_count = 0
loop forever
```

```
if probe is successful
    log success
    failure_count = 0
    wait 5 minutes
else
    log failure
    increment failure_count
    if failure_count == 2
        send email
    wait 1 minute
```

If you do this as a Bash script, in addition to **`if`** and **`while`**, you might need:

```
until test_command(s) ; do
    body_command(s)
done
```

which is just a while statement with the condition negated (i.e. repeat while the test command is *not* true. The exit status of a pipeline or command sequence (separated by **`;`**) is the exit status of the last command.

Alternatively, you can write the control logic in awk or Perl using the **`system`** library function to invoke external commands.