Week 1

Q1

Suppose you've written your own version of the cat command that, when called, outputs this to the terminal:

```
\    /\
 )  ( ')
(  /  )
 \(__)|
```

Assume that the full path name for the the directory where the executable for your personal cat command is stored is:

`/u/cs/ugrad/bruin/better_commands/animals`

a) Suppose you're working on lnxsrv06 and decide this version of the command is good enough to be the default version of 'cat' for yourself. Write a one line command that will ensure that when you invoke the 'cat' command, you get back the personal version above.

   export PATH=/u/cs/ugrad/bruin/better_commands/animals:$PATH

b) Your friends aren't nearly as impressed by your cat command as you thought they'd be, so you now decide to retire the command. You do *not* delete the executable; that is, the file stored in the better_commands/animals folder still exists. Write a command that will ensure that when you now invoke the 'cat' command, **after** the events of part a, you get back the actual Linux concatenate command.

   Different answers possible (you can use stream editing or grep to filter out the first first path) but easiest is:

   export PATH=/usr/local/cs/bin:$PATH

   Note the question does *not* ask to necessarily delete the invalid path, so it's perfectly reasonable to just overwrite it with the path of where Linux cat actually is.

Week 3

Q1

You and your friend decide that you've had enough of seeing USC on the news, and you'd like to create a spam filter to filter out the unneeded acronym. Write an extended regular expression that matches with non-empty strings that consist of the uppercase letters U and S and C, but with the following restrictions:

- The strings must have a length of six
- The strings must have exactly two copies of each letter (two U, two S, two C)
- The strings must start with the letter U and end with the letter C
- Given that the string contents can be thought of being split into two groups, each consisting of three letters U, S, and C, then each group must follow this restriction:
    - U ← S ← C (U must come before S within the string and S must come before C within the string)

Here are some examples of valid matches (one on each line):

USCUSC
USUSCC
UUSSCC

Hint: Try writing out the combinations as a tree that will help in deriving the regular expression

U(S(CUSC|U(CSC|SCC))|US(CSC|SCC)) or anything similar (can be found by writing out the combinations as a tree)

Q2

1- Write a Bash command that implements a configuration of the Caesar cipher (a very simple substitution cipher). The command should comply with the following specifications: (2pt)

- Input should be from stdin
- Output should be to stdout
- Only substitute lowercase and uppercase ASCII letters
- Use a right rotation of three places i.e. substitute 'a' with 'd', 'b' with 'e', 'w' with 'z', 'x' with 'a', and so on. Similarly, do the same with uppercase letters: substitute 'A' with 'D', 'B' with 'E', 'W' with 'Z', 'X' with 'A', and so on.

For example:

Input: afC2xA

Output: diF2aD

```
tr ‘a-zA-Z’ ‘d-za-cD-ZA-C’
```

2- Write another Bash command such that the previous specifications apply, except:

- Substitute lowercase letters ‘x’ through ‘z’ with the ‘@’ character instead of their previous substitutions.
- Substitute uppercase letters ‘A’ through ‘W’ with the ‘@’ character instead of their previous substitutions.

For example:

Input: afC2xA
Output: di@2@@

```
tr ‘a-zA-Z’ ‘d-z[@*]A-C’
```

Week 4

Q1

The following is a patch for a project you're working on.

```
--- a/src/coolors.c    2020-05-31 09:14:41.000000000 -0700
+++ b/src/coolors.c    2020-05-31 09:14:37.000000000 -0700
@@ -1,12 +1,17 @@
+/* Checks that an RGB parameter is valid */
+int is_valid_rgb(int val) {
+    return val >= 0 && val <= 255;
+}
+
 /* Convert RGB color to HEX format */
 int rgb2hex(int r, int g, int b) {
-    if (r >= 0 & r <= 255) {
+    if (is_valid_rgb(r)) {
         fprintf(stderr, "Invalid red value");
         return -1;
-    } else if (g >= 0 & g <= 255) {
+    } else if (is_valid_rgb(g)) {
         fprintf(stderr, "Invalid green value");
         return -1;
-    } else if (b >= 0 & b <= 255) {
+    } else if (is_valid_rgb(b)) {
         fprintf(stderr, "Invalid blue value");
         return -1;
     } else {
```

1. Explain what this patch does at a high level
   Adds a new function `is_valid_rgb()` used in `rgb2hex()` to validate rgb values.

2. How many lines are added and/or removed by the patch?
   8 lines added, 3 lines removed.

3. Write a shell command that applies the patch. Assume `coolors.c` is in `src/coolors.c` and the patch file, named `cool.patch`, is in the root of the project directory.

   if pwd in project root
   `patch -p1 < cool.patch`

```
if pwd in src/
patch -p2 < ../cool.patch
```

Q2

Zoom is working on a Python tool that extracts info from password-protected meeting links that follow this format:

- Links start with the base URL "https://zoom.us/j/" or "https://SUBDOMAIN.zoom.us/j/", where SUBDOMAIN is a non-empty string of lowercase letters
- The base URL is followed by a 9-digit conference ID (e.g., 356209714)
- The conference ID is followed by "?pwd=", which is followed by a 32-character alphanumeric password (e.g. JZ9u3x6FVBumItd2PxOrbAApyCv860i3)

Here are some examples that match the format above:

https://zoom.us/j/356209714?pwd=JZ9u3x6FVBumItd2PxOrbAApyCv860i3
https://ucla.zoom.us/j/701495623?pwd=5915EqMgkRtaeCyrSTXouXeRnaQlSeCm

(a) Write a regular expression that matches with a valid base URL.

```
https://([a-z]+\.)?zoom\.us/j/
```

(b) Write a regular expression that matches with a valid 9-digit conference ID.

```
\d{9}
[0-9]{9}
[[:digit:]]{9}
```

(c) Write a regular expression that matches with a valid 32-character alphanumeric password.

```
[A-Za-z0-9]{32}
[[:alnum:]]{9}
```

(d) Write a Python function `parse_zoom(link)` that takes a valid meeting link as a string argument, extracts the conference ID and password using regex, and returns the values as a tuple. The outputs for the above examples would be:

```
('356209714', 'JZ9u3x6FVBumItd2PxOrbAApyCv860i3')
('701495623', '5915EqMgkRtaeCyrSTXouXeRnaQlSeCm')
```

Python's standard library conveniently provides the `re` module for regex operations. To search for a regular expression match in a string, use `re.search(pattern, string)`. You can retrieve matched substrings in a regular expression with the help of capture groups and the `group()` function. For example,

```
import re
s = "hello world"
match = re.search("hello (.*)", s) # search for pattern in s
match.group(1) # first capture group result, i.e., "world"
```

```
import re

def parse_zoom(link):
     match = re.search('(\d{9})\?pwd=([A-Za-z0-9]{32})', link)
     return match.groups()
```

Alternate solutions included: using match.group() to get the entire match, having multiple regex, etc.

Week 5

Q1

In the space available below, explain the differences between the two different 2D arrays declared below, arr1 and arr2. Focus on the underlying memory structure, any performance implications, and use-cases.

```
const int ROWS = 5
const int COLS = 4

//arr1
int** arr1 = malloc(ROWS * sizeof(int*));
for (int i = 0; i < ROWS; i ++)
    arr1[i] = malloc(COLS * sizeof(int));

//arr2
int arr2[ROWS][COLS]
```

Many answers possible, below are some examples. Diagrams are useful too.

Arr1 was declared entirely dynamically. That means there is actually a 1D array of size ROWS, where each element is a pointer to another array of size COLS. That means that in memory, each row of the array is stored contiguously but there is no guarantee that neighboring rows are contiguous with each other.

Arr2 was declared statically, so it is all a single block of contiguous memory that is known about at compile time.

Performance wise, arr2 would actually be better than arr1 since it is a known quantity and contiguous memory is faster to read from.

Use cases for arr2 are more limited though. If we ever need an array that is variably sized or is resizable (based on user input, or application needs) then we need to use arr1.

Q2

Do the lines of code below make sense? Why or why not?
Assume arr2 below is the same as the one declared in part A above

```
    int** arr3 = arr2;
    printf("%p", arr3[2][3]);
```

It does not make sense. Arr2 was declared statically, so its type is not actually an int pointer to an int pointer. So line 1 has a type mismatch. Line 2 has an issue where the value being printed and the variable itself also has a type mismatch.

Week 6

Q3

On one running instance, the code below returns 10. What are 2 possible reasons why?
Is either of those a concern and why?

```
  return read(STDIN_FILENO, buffer, 50);
```

1. There are only 10 characters left in STDIN.
2. read() was interrupted

The second issue is more of a concern, because we assumedly wanted to continue reading before being interrupted.

Q5

One day you decide that it is annoying and confusing that EOF is not a real character. So you decide to make a new language encoding, ASCII-with-EOF, where EOF is a real character. You do this by replacing ASCII character 28, the file separator, with EOF since the file separator control character is very rarely used in any programming/files. You decide you won't miss it.

Now getChar() actually can return a Char and read() will also fill in EOF as a character in the buffer.

What are some cons of this idea? Would you want to permanently use ASCII-with-EOF for all of your future work?

Cons are that the EOF symbol should never be stored in the same data type as what the file itself is made of. That is because we could accidentally store and then read a EOF character which would stop us from reading the rest of the file.

We would not use ASCII-with-EOF in the future. It works better for the file system if we just keep track of the length of the file, and then send alerts/alternate values (like read() has a dedicated int return value).

Week 7

Q1

You are a developer at a startup working on a project written in C. Apart from the C standard library, the project also uses a custom library you wrote called coolors. Here's a code snippet from the project:

```c
#include "coolors.h"
#include <stdlib.h>
...
/* Return a random hex color */
int random_hex_color() {
      int red = random_value(0, 256);
      int green = random_value(0, 256);
      int blue = random_value(0, 256);

      // Call function in coolors
      int hex = rgb2hex(red, green, blue);
      return hex;
}
...
```

1. You need to send the compiled executable to quality assurance (QA) for testing. Which linking method(s) would you use here to build the program and why? Assume that QA testers use the same architecture and Linux distribution as you.

   Multiple answers may be accepted here. Statically link the custom library since QA might not have it on their machine. Dynamically linking may be accepted (students might assume QA working on the same project has access to all libraries, or simply send the library alongside it). Dynamic linking is advantageous in case QA finds a bug and we don't want to recompile the library.

2. Write a gcc command to compile the coolors source code, located in `coolors.c`, into a shared library, `libcoolors.so`.

   ```
   gcc coolors.c -fPIC -shared -o libcoolors.so
   ```

3. Rewrite the code snippet to dynamically load the coolors shared library solely to be used within this function. If there is a dynamic loading error and the function doesn't have a random hex color to return, the function should return a value of -1.

```
#include <stdlib.h>
#include <dlfcn.h>
...
/* Return a random hex color */
int random_hex_color() {
      int red = random_value(0, 256);
      int green = random_value(0, 256);
      int blue = random_value(0, 256);

      void *handle = dlopen("libcoolors.so", RTLD_NOW);

      if (!handle)
            return -1;

      int (*rgb2hex)(int, int, int) = dlsym(handle, "rgb2hex");

      if (dlerror() != NULL)
            return -1;

      // Call function in coolors
      int hex = rgb2hex(red, green, blue);

      dlclose(handle);

      return hex;
}
```

Week 8

Q1

You are working on a software project that uses Git as the dedicated version control system. You clone the remote repository to a local directory on your computer. Your local repository has the following commit history:

C1 ← C2 (master, origin/master)
Where C1 and C2 are commits and C2 is pointed to by the 'master' and 'origin/master' branch references.

The remote repository also has the following commit history:

C1 ← C2 (master)

The following questions depend on each other and should be answered sequentially.

1- You commit a new snapshot to your local 'master' branch (assume the new commit is C3). What does your local commit history look like now? Make sure to draw the commit history (you can use the format above). And explicitly show the updated branch references and generated commits, if any.

C1 ← C2 (origin/master) ← C3 (master)

2- Unfortunately, someone on your team has pushed new changes to the remote 'master' branch before you could push your local changes. The remote repository now looks like this:

C1 ← C2 ← C4 (master)

You pull from the remote repository. What does your local commit history look like now? Make sure to draw the commit history (you can use the format above). And explicitly show the updated branch references and generated commits, if any.

$$\begin{tikzcd}
 & & \text{C4 (origin/master)} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{C1} & \text{C2} \arrow[l] & \text{C3} \arrow[l] & \text{M1 (master)} \arrow[l]
\end{tikzcd}$$

3- Once again, before you could push your local changes, someone on your team pushes their work to the remote repository. (Your team really needs to work on communication.) The remote repository now looks like this:

C1 ← C2 ← C4 ← C6 (master)

After pulling from the remote repository, what does your local commit history look like now? Make sure to draw the commit history (you can use the format above). And explicitly show the updated branch references and generated commits, if any.

```
           C4   ←  C6 (origin/master)
         /      \      \
C1 ← C2 ← C3 ← M1 ← M2 (master)
```

Q2

Assume your local commit history looks like this:

```
      C2 (feature)
    /
C1 ← C3 (master)
```

Where C1, C2, C3 are commits and ‘master’, ‘feature’ are branch references.

1- You are currently on the ‘feature’ branch. You run ‘git rebase master’. What does your local commit history look like now? Make sure to draw the commit history (you can use the format above). And explicitly show the updated branch references and generated commits, if any.

C1 ← C3* (master) ← C2* (feature)

Note that C3* and C2* are new commits

2- You now checkout the master branch and call ‘git merge feature’. What type of merge occurs?

Fast-forward merge (or 2 way merge).

Week 9

Q1

After working for too long on Assignment 9 for CS 35L, one of your friends decides that the layout of git is just awful, and they simply don't like it. In particular, they state the following:

- The .git/object file should have the entire SHA-1 hash as its file name, instead of the first two characters being used as folder names and the remaining characters used as a file name inside of the folder.
- The files inside of .git/object should be uncompressed, so they don't have to take the useless extra step of decompressing files.
- Keeping all SHA-1 hashes for all git objects together inside of the .git/objects folder is too confusing; they should be separated into three different folders - .git/objects/commits, .git/objects/trees, .git/objects/blobs.

For each of the three points above, argue why your friend may be correct, or why your friend may be incorrect. Write only 1 - 2 sentences per bullet point.

Many answers possible as long as sufficiently backed up. One example:

- Dividing the .git/object folder into subdirectories with the first two characters as the subdir is a good decision. Since it speeds up access time for searching a file (less files per folder)
- Keep files compressed is a good thing. Git has to keep full copies of each commit, and if it is a large repo that can mean a lot of storage requirements.
- Based on the hash alone we don't know what type of object it is (commit vs tree vs blob) so it is useful to only have to search one folder for that object.

Q2

For the following questions, use the commit graph below. Assume that c1 is the oldest commit, and that each commit points to its child commits.

$$\begin{tikzcd}
\text{c1} \arrow[r] & \text{c2} \arrow[r] & \text{c3} \arrow[r] \arrow[d] & \text{c10} & & \\
 & & \text{c4} \arrow[r] \arrow[dr] & \text{c7} \arrow[r] & \text{c8} \arrow[r] & \text{c9} \\
 & & & \text{c5} \arrow[r] & \text{c6} \arrow[u] &
\end{tikzcd}$$

a. (2 pts) Give a valid topological ordering of this graph. You do **not** need to list branch names, or use the sticky start/end notation from Assignment 9.

Many answers possible, one example:
c1, c2, c3, c10, c4, c7, c5, c6, c8, c9

b. (2 pts) Suppose I pull a copy of commit c7 to my own local machine, and I use git log to read through commit messages for this project. Of the 10 commits, which commit messages will I *not* be able to read from my copy?

   c5, c6, c8, c9, c10

Week 2

Suppose you are worried that someone has broken into your SEASnet GNU/Linux server and installed both a packet sniffer and a file sniffer. Both sniffers are malware in your server's operating system. The packet sniffer snoops on all data sent to or from the network, and the file sniffer snoops on all data being read from or written to files. While the sniffers are installed, you login into the server via SSH and do assignment 1.

a. Can the attacker now get a copy of your solution to Assignment 1. Briefly explain:

   Yes the attacker can get a copy with the file sniffer. Since the file at rest on the server is unencrypted.

b. Can the attacker now pretend to be you to log into the same server via SSH. Briefly explain.

   No, the attacker cannot pretend to be you. Passwords over SSH are sent encrypted, and the user's private key never leaves their machine. Since the compromised server at max should only have access to a user's public key.

c. Can the attacker now setup a fake SSH server of his own, one that you can mistakenly log into? Briefly explain.

   Note - multiple answers are possible if you clearly state the starting assumptions

   Yes, if they have compromised the server so they potentially have access to its public key and private key. Which they can use to impersonate its identity.

   No if we assume that the private key of the server was encrypted or password protected. Or if mentioned that there is still a name mismatch which could cause a problem for someone who has the identity already recorded in their known hosts. Or that it would be hard to register the impersonating server with the same name online as the real SEAS host.

d. Suppose instead you are worried that someone has broken into your SEASnet GNU/Linux server and installed a pipe sniffer. That is a piece of malware that snoops on data going through pipes. You are worried that the pipe sniffer will get a copy of the data flowing through the pipe in the following shell script:

   ```
   #/bin/sh
   Sed 's/a/b/; /lambda/d; s/x/yy/g' | uniq -c
   ```

   Your friend says "Don't worry; you can easily modify your script to use ssh so that the pipe sniffer will see only encrypted data." Does your friends remark make sense? If so, show how to modify the script along the lines that your friend suggested, so that it is as efficient as possible. (For example, it works well even if a lot of data passes through the pipe); if not, explain why not.

Two possible answers depending on stated assumptions:
Option 1: If you have another trusted machine, you can rewrite the command so that you SSH to another machine first and than run the desired sed command on that machine, which returns the result back to you. Then provide one such example command

Option 2: If you don't have another machine, then there is no safe way to do this. SSH only protects network traffic, so any attempt to run that piped command on the local machine will be sniffed. You can't encrypt it locally either, since at some point that encryption command will take in plaintext input.