```
======================================================================
  CS 50011          INTRODUCTION TO SYSTEMS II          Summer 2017
                              Lab #1
======================================================================
```

Name: ______________________________________

```
======================================================================
```

Part 0: Introduction to your VM

```
----------------------------------------------------------------------
```

1. Log into your virtual machine. The instructor will assign you a port number in the range 2000-2010. You may access your virtual machine via ssh:
   $ ssh -p 20XX root@endor.cs.purdue.edu

   The default password is “cs50011PW”. Change it immediately by executing passwd.

   ***** IMPORTANT ***** Course staff DO NOT HAVE ACCESS to your password. If you forget it, you will have to ask the instructor to reset it.

   A more secure, and often convenient, way to access your system is to use an SSH key. Instructions for this can be found here:
   https://www.digitalocean.com/community/tutorials/how-to-set-up-ssh-keys--2

   This is optional.

   Note that after you create the .ssh subdirectory and authorized_keys file, you should execute:
   [root]$ restorecon -R .ssh

   This is because the SELinux context is not properly set upon creation.

   On a final note, typically remote root access is disabled on systems as part of a defense in depth approach to security. For educational purposes, we’re going to violate this standard practice.

2. Some portions of this lab should be run as the root user. Others should be done unprivileged in a “personal” account. Commands that should be executed as root will have a prompt prefixed by [root]. Create the unprivileged account now:
   [root]$ useradd yourusername

3. Become that user:
   [root]$ su – yourusername
   $

4. Create a Lab01 directory inside your home directory. All solutions to the programming exercises should be stored here.

5. Copy this week’s lab files (all files in ~jeff/labfiles/Lab01) into your Lab01 subdirectory.

Part 1: Data Forensics Fun (25 points)

One of the files included with this week’s lab files is named “sdcard.img”. This is a block-level copy of an sdcard used in a camera. You can mount this image and inspect its contents as root.

To return to your root shell, type **exit** or **ctrl-D**.

```
[root]$ cd ~yourusername/Lab01
[root]$ losetup loop0 ./sdcard.img     # Create a loopback device

# Note how there is a single partition
[root]$ fdisk -l /dev/loop0

[root]$ mount /dev/loop0p1 /mnt        # Mount it
[root]$ cd /mnt
```

[10 points] Now you can explore the files that are present there. How many files do you see? What are their name(s)? Note that there are no image files.

Unmount the disk.

```
[root]$ cd; umount /mnt
[root]$ losetup -d /dev/loop0
```

Now we’re going to use a program that is part of the “testdisk” package. This is a disk recovery utility. It specifically includes a program named “photorec” for recovering deleted photographs.

Let’s run it as an unprivileged user:

```
[root]$ su – yourusername
$ cd; cd Lab01
$ photorec ./sdcard.img
```

Choose the default options until you get to the destination selection. Then press “C” to choose the current directory. This will create a “recup_dir” subdirectory in your current working directory. The contents of that directory includes any recovered images.

Copy the images to your workstation. From a workstation terminal:

```
$ scp -P 20XX
root@endor.cs.purdue.edu:~yourusername/Lab01/recup_dir.1/*.jpg .
```

[15 points] What is the message in the picture?

```
======================================================================
Part 2: Basic BASH (25 points)
----------------------------------------------------------------------
SCRIPT NAME:      hailstone
PURPOSE:          to calculate and display a Hailstone sequence
PARAMETERS:       two, the starting value and the limit
RETURN VALUE:     zero on success, one if invalid arguments are given
```

"Hailstone sequences" are one of mathematics' unsolved problems in that it has yet to be proved that every starting value in a Hailstone sequence will eventually degenerate to the sequence 4, 2, 1, 4, 2, 1, ... In other words, does there exist a sequence that never settles to a repeating cycle? A Hailstone sequence is defined in the following manner:

Starting with any positive integer n, form a sequence in the following manner:
- If n is even, divide it by 2 to give n' = n/2.
- If n is odd, multiply it by 3 and add 1 to give n' = 3 * n + 1.

Take n' as the new starting number and repeat.

For example, n = 5 results in the following sequence: 5, 16, 8, 4, 2, 1, 4, 2, 1, ...
n = 11 produces the following: 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...
These are called "Hailstone sequences" because they go up and down in a similar manner to hailstones in a cloud before they fall to Earth.

Your task for this part is simple. You are going to write a BASH script that generates a Hailstone sequence when provided with a starting value, *n* and a limit, *l*. The limit represents how many values in the sequence should be generated.

You should check that the number of parameters is correct, but do not worry about the user providing invalid start and/or limit values.

Recall that the modulus operator (%) can be used to determine whether a number is even or odd.

N % 2 = 0 if and only if N is even.

The following are a few sample runs of our solution. Your output should be identical to ours. **We use the UNIX diff utility to grade**. If we have to inspect your output you automatically lose 5 points.

```
$ hailstone
Usage: hailstone <start> <limit>
$

$ ./hailstone 5 9
5 16 8 4 2 1 4 2 1
$
```

==
Part 3: Looping with non-integer values (25 points)

SCRIPT NAME:	compile
PURPOSE:	to compile and test a series of C source files
PARAMETERS:	none
RETURN VALUE:	zero on success, one on error

Assuming that you executed the copy command at the beginning of this lab, you should have the following files in your Lab01 directory:
src1.c, src2.c, src3.c, input1.data, input2.data, input3.data

The script **compile** should attempt to compile all files beginning with the name "**src**" and ending with "**.c**". It should then print a message indicating whether the compilation was successful or not.

If compilation was successful, you should then run the generated executable (**a.out**) sequentially on all input files beginning with "**input**" and ending with "**.data**".

This means that you should have two nested loops.

Here is some psuedo-code:

```
For each .c file in current directory
  Compile .c file using gcc -Wall -std=c99 FileName
  Check gcc's return value - 0 = success, nonzero = failure (HINT:
      Remember bash's "special variables"!)
  Display appropriate message (see output below)
  If gcc succeeded
    for each input*.data file in current directory
      Display appropriate message (see output below)
      Execute generated a.out binary on each input file
        Eg, a.out < Input_File_Name
      Check return value of a.out, display appropriate message
```

Here is a sample run of our solution (please note that when we grade your program there will be other files in the directory and many more src and input files). Note that some of the output (output not in **bold**) is generated by running a.out (and sometimes gcc) and should not be explicitly generated by your script:

```
$ compile bob
Usage: compile

$ compile
C program: src1.c
Successfully compiled!
Input file: input1.data
The prime numbers between 3 and 5 are: 3
Run successful.
Input file: input2.data
The prime numbers between 3 and 10 are: 3 5 7
Run successful.
Input file: input3.data
Error...lower limit is larger than upper limit
Run failed on input3.data.
```

```
C program: src2.c
src2.c: In function `main':
src2.c:57: error: syntax error at end of input
Compilation of src2.c failed!

C program: src3.c
Successfully compiled!
Input file: input1.data
The prime numbers between 3 and 5 are: 3
Run successful.
Input file: input2.data
The prime numbers between 3 and 10 are: 3 5 7
Run successful.
Input file: input3.data
Error...lower limit is larger than upper limit
Run failed on input3.data.

$
```

```
========================================================================
Part 4: UNIX commands (25 points)
------------------------------------------------------------------------
SCRIPT NAME:       phonebook
PURPOSE:           to display directory information
PARAMETERS:        none
RETURN VALUE:      zero if match is found, one otherwise
```

Write a script that asks the user to enter part of a name to search for in the file **house_dir_tab.txt**. Once the search string is entered, you should use **grep** along with **wc**, **cut**, and **head** to generate output identical to the following:

Recall how you can use the pipe **|** to feed output from one command into the input of another.

```
$ phonebook bob
Usage: phonebook.ksh
$ phonebook
Welcome to MagicPhone!
Please enter part or all of a name to search for: Bob
I found 11 matches
You might want to be more specific.
The first of these matches is:
Name: Barr, Bob
State: GA
Phone: 225-2931
Search complete on Wed Aug 22 16:07:38 EDT 2007
$ phonebook
Welcome to MagicPhone!
Please enter part or all of a name to search for: Weiner
Match found!
Name: Weiner, Anthony D.
State: NY
Phone: 225-6616
Search complete on Wed Aug 22 16:07:48 EDT 2007
$ phonebook
Welcome to MagicPhone!
Please enter part or all of a name to search for: Turkstra
Sorry, I could not find that person.
$
```

```
======================================================================
Part 5: File I/O (25 points)
----------------------------------------------------------------------
SCRIPT NAME:      histogram
PURPOSE:          display statistics for a provided set of data
PARAMETERS:       one, input filename
RETURN VALUE:     zero on success, one on error
```

Your job is to write a script that generates a histogram of grades when provided with an input file. The data file's format is as follows, with one or more space and/or tab separating the fields (see the file **ex02**):
<login> <score out of 100>

The script should determine the average score in addition to displaying the histogram. The histogram should have 11 bins:

```
== 100
>= 90 && < 100
>= 80 && < 90
>= 70 && < 80
>= 60 && < 70
>= 50 && < 60
>= 40 && < 50
>= 30 && < 40
>= 20 && < 30
>= 10 && < 20
< 10
```

There are many, many different ways to implement this script. It is ultimately up to you how you approach it. Spending a little time thinking about it now may significantly reduce the amount of code you have to write in the end. You are strongly encouraged to use arrays for this part.

Hint: What happens if you divide a given score by ten and then multiply it by 10 using integer division and multiplication?

When finished, your script's output should match the following exactly:

```
$ histogram
Usage: histogram <filename>

$ histogram noread
Error: noread is not readable!

$ histogram ex02
91 scores total...
100: =
 90: ==========================
 80: ==========================
 70: ================
 60: ===========
 50: ======
 40: =
 30: =
 20: =
 10: =
  0: =
Average: 78
$
```

Hint: To get the whitespace correct use printf instead of echo to display the bucket values (100, 90, 80, 70, etc).

==
Part 6: Exploring UNIX access control (25 points)

Alice would like to share a single file named “data” with Bob.

```
$ id alice
uid=1000(alice) gid=1010(users) groups=1010(users),200(shared)
$ id bob
uid=1001(bob) gid=1010(users) groups=1010(users),200(shared)
```

```
Assume:     the group ‘users’ contains many users
            the group ‘shared’ only contains alice and bob
            the file ‘data’ is located in /home/alice/private/data
```

What are the *most conservative* permissions for each element in the path, including the file, that would permit bob to access the file ‘data’?

Execute the following commands as root in your VM:

```
# Create two new users
[root]$ useradd userone
[root]$ useradd usertwo

# Become userone
[root]$ su – userone

# Create a file
$ echo “Hello” > myfile
$ ls -l myfile
```

The permissions that you see are the result of the default umask.

```
$ umask
```

Are they sufficient to prevent other users from accessing myfile? State any assumptions.

Suppose that userone wishes to permit usertwo, and only usertwo, access to myfile without changing its owner and group. Write the command(s) to make this possible:

As userone, try accessing the file located at ~jeff/dir1/dir2/fun. Is it accessible? Why are why not? You may use root privileges to arrive at a more complete answer.

If fun is inaccessible, what should be done to make it accessible?

Convert the following chmod commands from symbolic notation to numeric notation or vice versa:

```
chmod u+rwx,g+rx,o+rx filename
```

```
chmod u+rw filename
```

```
chmod u+rwx,o+x filename
```

```
chmod 711 filename
```

```
chmod 644 filename
```

```
chmod 4700 filename
```

==
Part 7: inodes and files (25 points)

Given the diagram on slide 28 of Lecture 1, a block size of 512 bytes, and an inode (in memory) for a regular file, how many blocks must be read to access byte 71680? Recall that bytes are addressed starting at 0. Assume that blocks are referred to by a 32-bit unsigned integer.

How many blocks are allocated for a file of size 524288 bytes?

Execute the following commands:

```
$ df -h
```

Note the available space for the root partition (/). What is it?

```
$ truncate -s 10G largefile
$ ls -lh largefile
```

How large is the file “largefile”?

Execute:

```
$ stat largefile
```

What is the number of allocated blocks?

Why might this be? Read up on “sparse files.” Can you come up with any scenarios where this could be used maliciously on a production system?

Hint: read about zip bombs.

==

Part 8: Endianness (25 points)

SOURCE NAME:	endian.c
PURPOSE:	determine and display system endianness
PARAMETERS:	none
RETURN VALUE:	zero on success, one on error

Write a C program that determines whether a system is big endian or little endian.

Implement the following functions:

```
int is_big_endian();
```

return 1 if the architecture on which we are executing is big endian, 0 otherwise.

main(): Call is_big_endian() and display either:
```
This system is big endian.
```
Or
```
This system is little endian.
```

Hint: How might one inspect a single byte of a multi-byte value in C?

What is the endianness of the following systems:

data.cs.purdue.edu:

lore.cs.purdue.edu: