§2 — "SSH" (windows/mac)

SSH stands for **S**ecure **Sh**ell. Abstractly, you can think of it as a means to access one computer from another.[1] By now you've already had exposure to the shell of a computer. If you're coming from a Windows background, you likely know this as **command prompt**, that black window shown in Figure 1. If you're coming from Mac, then you know it as **terminal**, as shown in Figure 2.

Figure 1: **Window's Command Prompt**, which has its own set of commands for interacting with the shell.

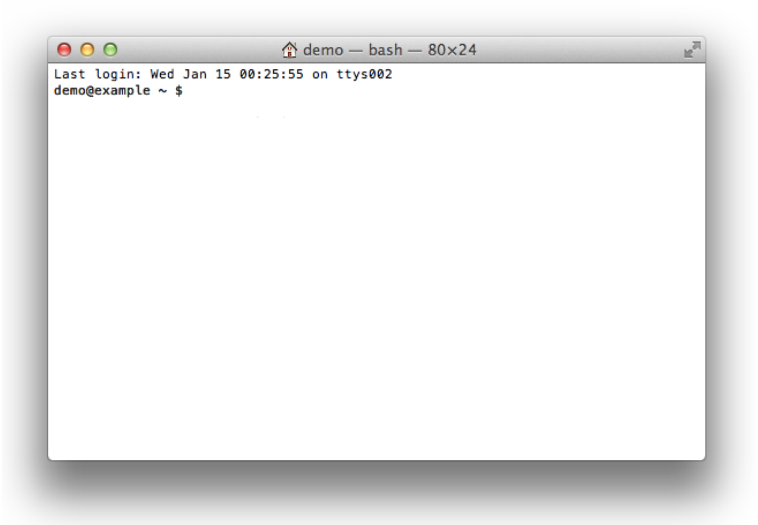

Figure 2: **Apple's Terminal**, which follows the standard UNIX/POSIX format. All of the following specifications (commands etc.) should work in here.

[1] In reality, it's far more complex than this and offers a bunch of cool functionality.

From this box (which gives you access to the computer in the form of a **shell**), you have the ability to run programs, view files, and edit configurations—essentially everything that you can do with any computer. The idea behind SSH is to give you access to the shell of another computer. For our purposes, we can think of it as a program that redirects input from your comptuer to the remote computer and redirects output from the remote computer to yours. Thus, when you type a command into the SSH program on your side, the shell of the remote computer receives it and when the remote computer tries to print something to its screen, the message shows up on the screen of your SSH program.

SSH is a remarkably powerful tool.

As a note in semantics, SSH is termed *secure* shell because of the way it transfers messages between your local machine and the remote one. Specifically, it encrypts your passwords (among other data) before transmitting them. This makes it impossible for someone who's watching your connection to read your password.[2]

P.1 "Gaining Access" — This section is going to be slightly different depending upon what machine you're coming from. I'll start off with the section on Unix-like machines (Mac / Linux / Solaris varieties) and then have a section for those coming from windows.

Either way, though, you'll some info before you can access the remote machine. You'll need:

- hostname – The "address" of the remote machine. This is normally in the form of either a recognizable URL (like `ssh.example.edu`) or an IPv4 address (like `192.168.1.1`).[3]
- port – The "door" at the address given by the hostname. Each address may have many doors with different services waiting behind different doors. This will be in the form of a number (between 1 and 65535) and is normally (defaulted) 22.
- username – The name of the user on the remote machine that you want to access the machine as. Just as you must "log on" to your own machine, you too must log on to the remote one.
- password – The password associated with the user.[4]

unix (mac)
Unix-based operating systems have SSH built into their terminals. Accessing a remote

[2]I use the term *impossible* liberally here.

[3]For the sake of future-proofing, I should include the possibility for a hostname to be an IPv6 address, that looks something like `2001:0db8::8a2e:0001:::7334`

[4]In some cases, it is possible that you don't need a password; however, that's advanced topics and is overlooked at this point in the interest of simplicity.

computer, then, is as easy at opening up the terminal and typing `ssh username@hostname -p port` at which point you will be prompted for a password, which you should enter. As a security measure, when you type in your password, no characters will be printed to the screen. Hit enter when you're done.

Assuming we want to access a machine at `ssh.example.edu` with username `demo` and port `1022`, we would type the following:

```
~ $ ssh demo@ssh.example.edu -p 1022
demo@ssh.example.edu's password:
```

Because port 22 is the default, ssh assumes you want it if you do not specify a port number. So for example, if we want to access a machine at `23.19.17.13` on port 22 with username *sample*, we would type

```
~ $ ssh sample@23.19.17.13
sample@23.19.17.13's password:
```

. . . and with that, you should be in!

windows

Unfortunately, Windows does not really like playing nice with POSIX and SSH and does not have any built-in tools to handle opening SSH connections to remote computers.

Luckily, there are several tools that make using SSH possible on Windows. One such is called **PuTTY** and is available online at `http://www.chiark.greenend.org.uk/~sgtatham/putty/`. I endorse and will assume its use for this section.[5]

Download the most recent PuTTY tool for "Windows on Intel x86". You only need the `putty.exe` executable for our purposes. Save it somewhere accessible.

Running the file should present you with a new window that looks something like Figure 3. Simply put your hostname in the "Host Name (or IP address)" box, edit the port if necessary and click "Open".

You'll be prompted for a username and then a password. As a security measure, when you type in your password, no characters will be printed to the screen. Hit enter when you're done.

With that, you should be successfully make a connection. The PuTTY window will now act as your interface with the remote computer.

[5]The linked website features a prominent disclaimer about the legality of encryption. Chances are you are fine; however, if you think the government of your country may take issue with your remotely accessing machines, look it up. SSH was perfectly allowed in the United States and the United Kingdom at the time this document was written.

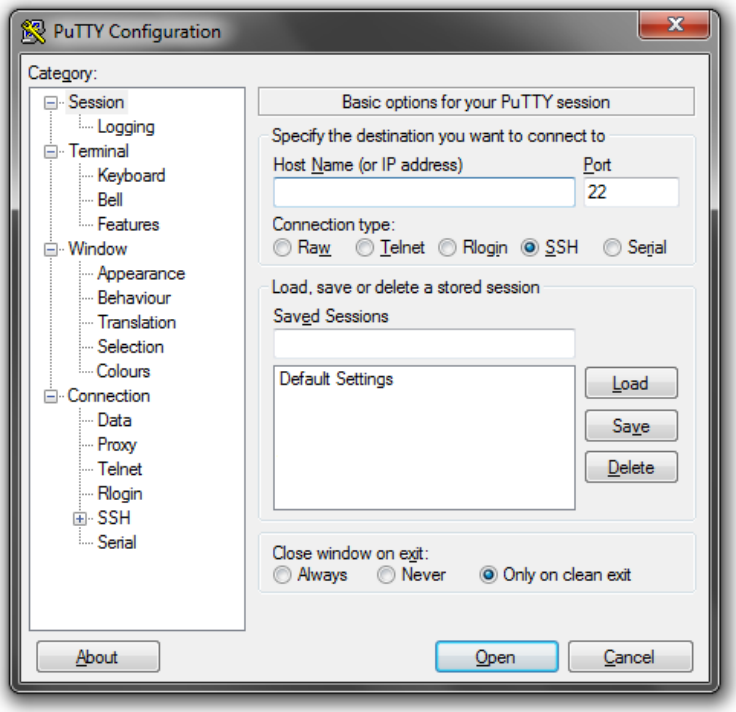

Figure 3: **PuTTY's Start/Configuration** From here you can specify things like hostname and port as well as do things like set up saved sessions/default settings.

One of the most important skills of any programmer is interaction with the computer. Arguably, the most powerful way to talk to a computer is through the shell. In this section, we'll familiarize ourselves with various ways to talk to the shell. Because of its widespread use, we will assume and use the POSIX standard, the set of instructions and specifications most often used for UNIX systems.

P.2 "Navigating the Filesystem" — Pages upon pages of information could be written about how files and other data are stored on a computer, and in fact, they have been. In the interest of brevity, I present an abstracted version of how everything. While this model is conceptually useful, I suggest you bear in mind the fact that there is more going on "under the hood".

We can think of the storage system of a computer as a group of boxes and pieces of paper. Boxes may contain pieces of paper or other boxes. Boxes may be empty. Consider Figure 4.

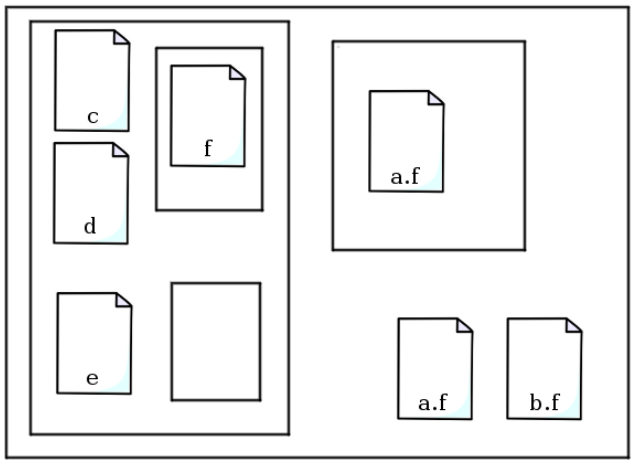

Figure 4: **Illustration of Box/Paper Filesystem Analogy** Papers are named above while boxes are not. In the following translation, you should be able to figure out which box is which directory.

Each piece of paper is called a file and is represented by a name, as in `myfile`. Often, filenames include **extensions**—groups of letters following a dot, as in `myfile.ext`. These extensions help to identify the type of file, for example `.txt` for text files, `.html` for web pages encoded in HTML, `.mp3` for MP3 audio files. While useful for the purpose of identification, extensions do not fundamentally change the nature of a file's contents: they only change its name.

Each box is called a directory (or folder) and is represented by a name followed by a forward slash, as in `mydirectory/` . The largest box is known as the "root" of the filesystem and has no name in its representation. Sometimes, directory names will not appear to include the trailing slash; however, you can be sure it is there, just hidden.

The location of every file and directory on a storage device can be given relative to the root. The list below gives the locations of each of the files and directories in Figure 4 according to this schema.

```
/
/a.f
/b.f
/foo/
/foo/c
/foo/d
/foo/e
/foo/bar/
/foo/baz/
```



```
/foo/baz/f
/baz/a.f
```

When using the shell, at all times we have a **working directory** that represents where we *are* within the filesystem. We can see this information by typing `pwd` for "print working directory". The text below demonstrates running this command:

```
demo@example $ pwd
/home/demo
demo@example $
```

To change directories, we use the command `cd` for "change directory", followed by an argument signifying where we want to go. For the argument, we can give an absolute path (the exact location of the directory with respect to the location of the root) or we can give a relative path (how to get to the directory from our current location). When using relative paths, using `..` allows you to go "up".[6] See the following examples of both absolute and relative navigation.

```
demo@example $ pwd
/home/demo
demo@example $ cd /home/
demo@example $ pwd
/home
demo@example $ cd demo/foo
demo@example $ pwd
/home/demo/foo
demo@example $ cd ../bar
demo@example $ pwd
/home/demo/bar
demo@example $
```

Of course, navigating by itself isn't all that useful. We also want to be able to see what is in a certain directory.[7] For this purpose, we use the command `ls` for "list". The command can be followed by the path to a directory (relative or absolute) we want to list the contents of or it can be used on its own to list the contents of the current directory.

```
demo@example $ pwd
/home/demo
demo@example $ ls
bar file1 foo
demo@example $ ls foo
file2
```

[6] Going "up" means going to the directory in which your current directory resides. It's also known as the **parent directory**. Unsurprisingly, going "down" will bring you to a **child directory**.

[7] Well *I want to* anyway. I'm not sure what you're doing.

```
demo@example $ ls /home/demo/foo
file2
demo@example $ cd foo
demo@example $ ls
file2
demo@example $
```

We can also ask `ls` for more information than it would normally give by adding `-l` or `-a` before the path. The `-l` flag will provide the "long" information for each element of the directory. The `-a` flag effectively shows hidden directories and files.[8] The two flags can be combined, as in `-la`.[9]

```
demo@example $ ls -l
total 11
drwxrwxr-x 2 demo demo 4096 Jan 14 21:06 bar
-rw-rw-r-- 1 demo demo  186 Jan  6 13:06 file1
drwxrwxr-x 3 demo demo 4096 Dec 15 08:06 foo
demo@example $ ls -a
.secretfile .hidden_folder foo bar file1
demo@example $
```

P.3 "Running programs" — Some files are known as "executable" files—files that may be run (i.e. executed). Their functionality may range from something as simple as printing the words `Hello world!` to the screen to something as complex as keeping all of Google's backend up and running.[10] To run a file you simply prepend a dot-slash to the **RELATIVE** path of the file. If you wanted to run the (executable) file `myProgram`, you could do the any of the following write:

```
demo@example $ ./myProgram
output of myProgram!
demo@example $ pwd
/home/demo/bar
demo@example $ cd ..
demo@example $ pwd
demo@example $ ./bar/myProgram
output of myProgram!
demo@example $
```

[8] Technically `-a` prints information for files and directories begin with a dot. Because UNIX filesystems normally hide these folders and files, I said the offending statement.

[9] There are a great many more flags that can be passed to `ls`. I provided the two I use most often. Typing `man ls` will give you a complete list.

[10] **Backend** meaning the part of their service that you do not directly interact with, including (but not limited to) actually performing your searches.

To pass arguments to a program, you follow the initial call with the arguments, separated by spaces:

```
demo@example $ ./myProgram arg1 arg2 arg3
output of myProgram!
you passed me three arguments: [arg1, arg2, arg3]
demo@example $
```

It's important to realize that not all files are executable. The computer only knows how to run files that are in a certain format: you can't run a random text file or image file, for example.[11] Often, executable files are called **binaries**, after the fact that they're written in a binary language that the computer understands.

In the last section, you encountered three commands, `pwd`, `cd`, and `ls`. Each of these is actually a program that you're running when you type its respective command. "... but wait!" you might exclaim, "where is the preceding `./`? And further, I don't see them in my current directory—where are they?"

This is part of the shell's dynamic; when you give it a command without the dot-slash prefix, it looks in some special directories for the program. These directories are given by the **path** environmental variable, which can be displayed by typing:

```
demo@example $ echo $PATH
/usr/local/bin:/usr/local/bin
demo@example $
```

The relevant directories will be displayed separated by colons.

P.4 "Editing Files" — Most modern computer users are used to editing files using some sort of graphical interface (e.g. Notepad, TextEdit, Sublime Text, Word); however, console-based editors must be used to edit files on remote computers, because the normal graphical programs can only access (read/write) to local files.[12] Several of the most popular programs for this purpose include vim, nano, and emacs.

I will focus on vim (pronounced "vim") because it is better than the other editors.[13] Vim is the successor to the vi (pronounced "vee-eye") text editor, originally released in 1976. Vim is run by typing `vim filename` at the command line.

Vim is based in the concept of having two modes, one for commands and another for direct input. Command mode allows you to do things like delete blocks of text, search for strings, save the file, run macros, and quit the program. Input mode allows you to

[11] For UNIX-type systems, the standard format of executable file is ELF, standing for Executable and Linkable Format.

[12] I'm a liar. Read up on X11 forwarding if you're interested.

[13] People might disagree with me. They are wrong. See `http://en.wikipedia.org/wiki/Editor_war`

directly type text to the file's contents. Simple navigation is possible in either mode by using the arrow keys.

On starting vim, you will be in command mode. Among others, the following commands are available to you. After entering a command, the enter key should be pressed.

- `i` change to insert mode with the cursor in place
- `a` change to insert mode moving cursor to next character
- `A` change to insert mode moving the cursor to the last character of the line
- `/string-of-text` to search the file for the phrase `string-of-text`
- `11G` to move the cursor to the 11th line
- `0G` go to the last line of the file
- `$` move the cursor to the end of the current line
- `0` move the cursor to the beginning of the current line
- `:w` to write the file to memory (i.e. save the file)
- `:w!` write the file no matter what
- `:q` quit vim
- `:q!` quit vim no matter what (e.g. even if you haven't saved the file)

In input mode, pretty much everything you type will be directly added to the screen. To return to command mode from input mode, simply press the `ESC` key.

P.6 "Miscellaneous" — Some useful commands you can use in the command line environment follow.

- `cat` – (**cat**enate) Allows you to print contents of a file directly to the screen.
- `grep` – (**g**lobally search a **r**egular **e**xpression and **p**rint) An immensely useful tool that allows you to search input for strings or other regular expressions.
- `find` – (*obvious*) Locates certain files and (optionally) performs an operation on them.
- `which` – (*no clue*) Prints the location (from the `$PATH`) of the command given as first argument.
- `chmod` – (**ch**ange **mod**e) Allows you to change the mode of file given as first argument. Can be used to deny read/write permissions
- `|` – ("pipe") Pipes output from preceding command to input of following command.

- `sudo` – (**s**uper **u**ser **do**) Runs the command that follows as the **super user** (with associated permissions), giving you a terrifying amount of power and the ability to completely screw everything up.

The complete functionality (and more) of each of the above commands can be found by typing `man command` at the command line, where `command` is the command you wish to learn more about. As you might guess, `man` is a program itself. It looks within the "manual pages" for an entry corresponding to the program given as first argument.