# CS2043 Final Project

**Due:** Sunday, March 9, 2014 at 11:59PM EST

**Important Note:** The configuration of the CSUG Lab machines is our standard environment. Make sure that your scripts execute as intended in one of the CSUG lab machines or in a clone thereof installed on your computer as a Virtual Machine.

**General Instrictions:**

- You may form groups of **two** students to complete this assignment. Please form a group on CMS and submit only one solution per group.
- For each problem, you will write a script and save it to a file named with the problem label, e.g., **project1.sh**. Assume that the input is in the same directory as the script.
- Once you complete the assignment, make a compressed tarball using gzip named `submission.tgz`, containing all the scripts you have produced. Submit your compressed tarball to CMS (`http://cms.csuglab.cornell.edu`). Your tarball should contain no directories.

---

## Bash Scripting: What's in theaters?

- **project1.sh** : In this project we will build a command line app in Bash that fetches the list of the top 10 box office movies from the Web, presents a menu to the user, and allows them to choose one of the movies in the list to read its synopsis.

The interface looks like this

```
Movies.com Top 10 Box Office
1.  The LEGO Movie - $31.4M
2.  3 Days to Kill - $12.3M
3.  Pompeii - $10.0M
4.  Robocop - $9.4M
5.  The Monuments Men - $8.1M
6.  About Last Night - $7.4M
7.  Ride Along - $4.6M
8.  Frozen - $4.3M
```

```
9.  Endless Love - $4.3M
10.  Winter's Tale - $2.1M

Choose a movie (1-10) >
```

When the user chooses a number between 1 and 10 and presses enter, the app displays the corresponding movie's synopsis on the screen (or an empty synopsis if the user enters an invalid option).

```
Movies.com Top 10 Box Office
1.  The LEGO Movie - $31.4M
2.  3 Days to Kill - $12.3M
3.  Pompeii - $10.0M
4.  Robocop - $9.4M
5.  The Monuments Men - $8.1M
6.  About Last Night - $7.4M
7.  Ride Along - $4.6M
8.  Frozen - $4.3M
9.  Endless Love - $4.3M
10.  Winter's Tale - $2.1M

Choose a movie (1-10) > 8


Movie 8
Synopsis


"After the kingdom of Arendelle is cast into eternal winter by the
powerful Snow Queen Elsa (voice of Idina Menzel), her sprightly
sister Anna (Kristen Bell) teams up with a rough-hewn mountaineer
named Kristoff (Jonathan Groff) and his trusty reindeer Sven to
break the icy spell.  Chris Buck and Jennifer Lee co-directed this
Walt Disney Animation Studios production based on Hans Christian
Andersen's beloved fairy tale The Snow Queen."


Press enter to return to the main menu
```

When users finish reading the synopsis, they can press enter to return to the initial screen (the menu without the synopsis) and choose another movie.

Here is our strategy. First, we are going to use the command `curl` to fetch the data from the website `http://www.movies.com/rss-feeds/top-ten-box-office-rss`. The syntax is

```
curl http://www.movies.com/rss-feeds/top-ten-box-office-rss 2> /dev/null
```

The data we will retrieve contains multiple records, each with a `title` field, which stores the movie's title, and a `description` field, which stores the movie's synopsis. We are going to extract the movie titles from the data and print them on the screen. In each record, the movie titles are embedded in a line with the following format:

```
<title><![CDATA[1. The LEGO Movie - $31.4M]]></title>
```

From this line, we want to extract and print the string

1. The LEGO Movie - $31.4M

When we print the strings for all movies, we will produce a nice numbered menu.

The movie's corresponding synopsis is embedded in a description field. The line has the following format:

```
<description><![CDATA[Colin Farrell headlines this romantic...]]></description>
```

From this line, we want to extract the string

Colin Farrell headlines this romantic...

While we will print the titles directly, we are going to store the synopses in a Bash array. Then, we'll print them on request.

(Hint 1: To create an array from a set of strings separated by the newline character we will first redefine the default field separator for Bash as the newline character:

```
IFS=$'\n'
```

For example, if we have a file called `actors.txt` with the following lines

George Clooney
Brad Pit

The command `array=($(cat actors.txt))` will be equivalent to

```
array=([0]=George [1]=Clooney [2]=Brad [3]=Pit).
```

On the other hand, after resetting the default field separator to the newline character, this command will be equivalent to

```
array=([0]="George Clooney" [1]="Brad Pit")
```

(Hint 2: Write an infinite while loop to repeat each cycle, i.e., (1) fetch and print the titles, (2) ask the user for input, (3) show a synopsis on request, and repeat from (1) after the user presses enter.)

(Hint 3: The `sed` flag `-n` in combination with the function `p` (see `man sed`) might prove handy for this project, e.g., `sed -n 's/Movie/Flix/p`.)

(Style tip: Ideally, your script shouldn't produce any temporary files. Some of your users may not have permission to write on disk.)

---

## Python Scripting: Six degrees of Kevin Bacon

Six degrees of separation is the theory that everyone is on average six steps away, by way of introduction, from any other person in the world, so that a chain of "a friend of a friend" statements can be made to connect any two people in a maximum of six steps.

This theory was made popular by the game "Six Degrees of Kevin Bacon" where the goal is to link any actor to Kevin Bacon through no more than six connections, where two actors are directly connected if they have appeared in a movie or commercial together. On September 13, 2012, Google made it possible to search for any given actor's 'Bacon Number' through their search engine (try querying "al pacino kevin bacon number").

In this exercise we are going to build the engine behind Google's Kevin Bacon Number feature in Python. Instead of actors, we will use the students in our familiar `restaurants.txt` dataset. We will break this task into two simpler steps.

**Step 1**: An adjacency list representation of a network is a collection of unordered lists, one for each vertex in the graph. Each list describes the set of neighbors of its vertex. The main operation performed by the adjacency list data structure is to report a list of the friends of a given actor. In other words, the total time to report all of the neighbors of an actor is proportional to its degree.

- **project2a.py** : write a Python script to build an adjacency list from the `restaurants.txt` file in Python using the `dyad 1` definition. The input file should be specified as an argument of `project2a.py` passed via the command line, i.e., `./project2a.py restaurants.txt`.

Here is an example. If your input, e.g., `restaurant.txt` file has the following lines:

```
A,B;R1
B,C;R2
C,A;R1
```

your script should produce the following "`Actor: Adjacent to`" lines:

```
A: B C
B: A C
C: A B
```

(Hint1: use nested dictionaries to build the adjacency list. For example, if "Bruno" and "Hussam" are friends, then you would have the entries `{"Bruno":{"Hussam": 1}, "Hussam":{"Bruno": 1}}` in your dictionary. If your dictionary is called `friends`, then `friends["Bruno"]["Hussam"]` returns the value 1 if Bruno and Hussam are friends. Otherwise, either `"Bruno"` is not a key of `friends` or `"Hussam"` is not a key of `friends["Bruno"]`, or vice-versa, and it returns a key error.)

(Hint2: If you want to access `friends["Bruno"]["Hussam"]`, remember to initialize both `friends` and `friends["Bruno"]` as dictionaries before its first use so as to avoid a key error. It is easy to initialize `friends`, but it can be challenging to initialize `friends["Bruno"]` as you don't know whether `"Bruno"` will be in the dataset. You avoid this problem by calling the function `friends.setdefault("Bruno", {})` before using the dictionary `friends["Bruno"]`. This function does nothing if `friends["Bruno"]` is already a dictionary, but initializes `friends["Bruno"]` as a dictionary if it is not.)

(Hint3: remember that the list of arguments is stored in the array `sys.argv`. You need to import the module `sys`.)

**Step 2**: Now, we are interested in computing the distance between a given actor and all other actors. To this end, we will use a network search algorithm called Breadth First Search (BFS).

BFS begins with a given actor, whom we call the root, and then inspects all of his or her friends. These friends have distance 1 from the root. Then for each of those friends in turn, it inspects their unvisited friends. These "friends of friends" have distance 2 from the root, and so on.

BFS can be implemented using an adjacency list and a queue. Here are the steps the algorithm performs:

1. Enqueue the root node, set the distance of root to zero, and mark root as visited.

2. Dequeue a node (let's call this node `n`).
   - enqueue all friends of `n` that have not yet been visited
   - set the friends distance to the distance of `n` plus one
   - mark the friends of `n` as visited
3. If the queue is empty, then the algorithm exits. Otherwise repeat from Step 2.

- **project2b.py** : write a Python script to compute the distance of all foodies in restaurants.txt who are reachable from a given root. The first argument of your script should be the name of the root, e.g., "Beula", and the second, the input file name, e.g., restaurants.txt. Your script should contain the code you've written in the previous step to build the adjacency list from `restaurants.txt`.

Here is an example. If your `restaurants.txt file` looked like

Bruno,Hussam;CTB
Harsh,Hussam;Subway
Harsh,Atheendra;CTB
Harsh,Sangha;Subway

Then, if you run `./project2b.py "Bruno" restaurants.txt` your output should be (not necessarily sorted by distance):

Bruno 0
Hussam 1
Harsh 2
Atheendra 3
Sangha 3

(Hint1: The restaurants.txt dataset contains a group of Cornell students. Therefore, their distance from one another is expected to be small.)

(Hint2: Use a list to implement the queue. You might find these two functions useful.

- `list.append(x)` : inserts element x at the end of a list.
- `list.pop(i)` : returns and deletes element at index i from list.)

(Hint3: Use a dictionary to keep track of the actors that have been visited by BFS and their distance from the root, i.e., `dist= {"Bruno": 0, "Hussam" : 1}`)