Assignment Overview

In this assignment you are writing an interactive program that displays information on the popularity of any given name from 1880-2016. The program will read the name popularity data for each 4 year period (quadrennium) from 1880 to 2016 from pre-formatted data files. It will then prompt the user to input commands to print statistics and graph results for the input name.

The purpose of this programming assignment is to gain more practice with C, dynamic memory management, pointers, and file I/O.

We will try not to do too much hand-holding in this PA. You have some experience now and you need to learn how to look up information yourself. You will not be given nearly as much help in the upper-division classes, so now is the time to start doing more on your own.

Make sure you start early! This assignment will be significantly more difficult than the previous assignments. If you do not start early, you will not finish this assignment on time!

Grading

- **README: 10 points** - See README Requirements [here](http://cseweb.ucsd.edu/~ricko/CSE30READMEGuidelines.pdf) and questions below
- **Compiling: 5 points** - Using our Makefile; no warnings. If what you turn in does not compile with the given Makefile, you will receive 0 points for this assignment. **NO EXCEPTIONS!**
- **Style: 10 points** - See Style Requirements [here](http://cseweb.ucsd.edu/~ricko/CSE30StyleGuidelines.pdf)
- **Correctness: 75 points**
  - **Milestone (15 points)** - To be distributed across the Milestone functions (see below)
    - Make sure you have all files tracked in Git.
- **Extra Credit: 5 points** - View Extra Credit section for more information.
- **Wrong Language:** You will lose 10 points for each module in the wrong language, C vs. Assembly or vice versa.

**NOTE:** If what you turn in does not compile with given Makefile, you will receive 0 points for this assignment.
Getting Started

Follow these steps to acquire the starter files and prepare your Git repository.

Gathering Starter Files:
The first step is to gather all the appropriate files for this assignment. Connect to pi-cluster via ssh.

```
$ ssh cs30xyz@pi-cluster.ucsd.edu
```

Create and enter the pa3 working directory.

```
$ mkdir ~/pa3
$ cd ~/pa3
```

Copy the starter files from the public directory.

```
$ cp -r ~/../public/pa3StarterFiles/* ~/pa3/
```

Copy over the following files from PA2. You are responsible for making sure these functions have good style and work correctly as they are required for this assignment.

```
$ cp ~/pa2/isInRange.s ~/pa3
$ cp ~/pa2/myRemainder.s ~/pa3
```

**Starter files provided:**

- pa3.h
- pa3Strings.h
- pa3Globals.c
- test.h
- testpopulateNameData.c
- Makefile
- filenameCompare.c

**Data sets provided:**

- /data
- /testData/reallySmallData
- /testData/smallData

**NOTE:** There are two directories included in these starter files: data and testData. These contain the quadrennium files for testing. **DO NOT** modify the contents of these directories, they have been setup to specifically work with this assignment.

Preparing Git Repository:
You are required to use Git with this and all future programming assignments. Refer to the PA0 writeup for how to set up your local git repository.
Example Input

A sample stripped executable provided for you to try and compare your output against is available in the public directory. Note that you cannot copy it to your own directory; you can only run it using the following command (where you will also pass in the command line arguments):

$ ~/./public/pa3test

NOTE:
1. The output of your program MUST match exactly as it appears in the pa3test output. You need to pay attention to everything in the output, from the order of the error messages to the small things like extra newlines at the end (or beginning, or middle, or everywhere)!

2. **We are not giving you any sample outputs, instead you are provided some example inputs.** You are responsible for trying out all functionality of the program; the list of example inputs is not exhaustive or complete. It is important that you fully understand how the program works and you test your final solution thoroughly against the executable.

Example input that has error output:

```
cs30xyz@pi-cluster-001:pa3$ ./pa3 -s StartEarly
```

```
cs30xyz@pi-cluster-001:pa3$ ./pa3 -d
```

```
cs30xyz@pi-cluster-001:pa3$ ./pa3 -s
```

```
cs30xyz@pi-cluster-001:pa3$ ./pa3 -s 31415
```

```
cs30xyz@pi-cluster-001:pa3$ ./pa3 -s -d
```

Example input that has normal output:

```
cs30xyz@pi-cluster-001:pa3$ ./pa3 -h
```

```
cs30xyz@pi-cluster-001:pa3$ ./pa3 -s 20000 -h
```

```
cs30xyz@pi-cluster-001:pa3$ ./pa3
```

Example output from the help message:

```
cs30xyz@pi-cluster-001:pa3$ ./pa3 -h
```

Usage: ./pa3 [-d dirname] [-s tableSize] [-h]

Popular Name Data Search.

- **-d <dirname>** Name of the directory to find data; 
  Defaults to ./data if unspecified.

- **-s <tableSize>** The size of the hashtable to be used; 
  Must be a decimal value in the range [100 - 20001]. 
  Defaults to 421 if unspecified.

- **-h** Print the long usage.

```
cs30xyz@pi-cluster-001:pa3$
```

Back to Top
One example of how the program is run with the name "Terry":
(Note: to stop the interactive portion you press ctrl-d which is denoted using "^d" below)

```
cs30xyz@pi-cluster-001:pa3$ ./pa3
```

Commands are:
- **graph [name]**
  Display a graph showing a name's rank over time.
- **stats [name]**
  List ranks by quadrennium for a name.
- **help**
  Display this message.

Please enter a command: **graph terry**
```
1 |
M M M M M M
40 | M M
80 | M F F F M M
120 | F M M
160 | M F F
200 | M
240 | F F
280 | M
320 | M M
360 | F
400 | F M
440 | M
480 | M F M
520 | M M M M M F
560 | M M F
600 | M M M M F M
640 | M M F
680 | F
720 | M
760 | F
800 |
840 |
880 | F
920 |
960 |
>1000 | F F F F F F F F F F F F F F F F
```

Please enter a command: **stats terry**
```
Quadrennium  Gender  Rank  Gender  Rank
1880        F       -      M      736
1884        F       -      M      551
1888        F       -      M      614
```
1892        F     -     M     622
1896        F     -     M     553
1900        F     -     M     518
1904        F     -     M     589
1908        F     -     M     524
1912        F     -     M     578
1916        F     -     M     621
1920        F     796     M     530
1924        F     607     M     467
1928        F     595     M     351
1932        F     503     M     189
1936        F     411     M     102
1940        F     265     M     54
1944        F     171     M     27
1948        F     121     M     29
1952        F     95      M     28
1956        F     92      M     33
1960        F     114     M     32
1964        F     172     M     39
1968        F     252     M     52
1972        F     360     M     66
1976        F     556     M     87
1980        F     706     M     106
1984        F     915     M     121
1988        F     -     M     154
1992        F     -     M     215
1996        F     -     M     281
2000        F     -     M     356
2004        F     -     M     424
2008        F     -     M     510
2012        F     -     M     608
2016        F     -     M     665

Terry as a female name was most popular in 1956 with a rank of 92
Terry as a male name was most popular in 1944 with a rank of 27

Please enter a command: ^d
cs30xyz@pi-cluster-001:pa3$
Detailed Overview

The function prototypes for the various C and Assembly functions are as follows.

**C routines:**
- int main( int argc, char * argv[] );
- long convertStr( char * str, int * errorFlag );
- int populateNameData( namesTable_t * table, char * dirName );
- int filenameCompare( const void * p1, const void * p2 );
- nameData_t * nameQuest( char * name, tableEntry_t * te );
- void handleUserInput( namesTable_t * table );
- int parseCmd( const char * const cmdString, const char * const commands[] );
- void printStats( nameData_t * namePtr );
- void generateGraph( nameData_t * name, char arr[GRAPH_ROWS][GRAPH_COLS] );
- void drawGraph( char arr[GRAPH_ROWS][GRAPH_COLS] );

**Assembly routines:**
- int getTableIndex( char const * name, size_t tableSize );
- long myRemainder( long dividend, long divisor );
- long isInRange( long num, long min, long max );
- int hash( const char * str );

Note that there are no function descriptions for `filenameCompare()` because we have provided that function for you, and there are no function descriptions for `isInRange()` and `myRemainder()` because those functions are directly copied over from PA2 (see the Getting Started section).

**For the Milestone, you will need to complete:**
- hash.s
- getTableIndex.s
- parseCmd.c
- nameQuest.c
- convertStr.c
- populateNameData.c

**Process Overview:**
The following is an explanation of the main tasks of the assignment, and how the individual functions work together to form the whole program.

1. Parse the command-line arguments using `getopt()` and check for any invalid input. If an error occurs, print the corresponding error messages and return indicating failure.
2. Populate the namesTable struct by hashing each name obtained from several data files into the table. The data files are inside a directory. The general workflow is to open the directory file, get the filenames in the directory and store them for easy access. For each file, read in the data and populate the table accordingly.
3. Allow the user to interactively obtain information about an input name. They can request:
   a. Statistics about the name
   b. A graph that shows how the popularity of the name as a female name and as a male name has changed over time
*Any time an error occurs, always free all the allocated memory before exiting the program.

**Memory Layout of the Name Data Hashtable:**

```
struct namesTable {
    entryPtr entryPtr;
    size;
}
```

- `entryPtr`: Pointer to a fixed-size array of `tableEntry` structs. The size of the array is the size of the `namesTable`.
- `dataPtr`: Pointer to a dynamically expanding array of `nameData` structs where there is one `struct nameData` for each name whose hash key maps to that index in the `namesTable`.

**Clarifications on the above diagram:**

- `entryPtr`: Pointer to a fixed-size array of `tableEntry` structs. The size of the array is the size of the `namesTable`.
- `dataPtr`: Pointer to a dynamically expanding array of `nameData` structs where there is one `struct nameData` for each name whose hash key maps to that index in the `namesTable`. 
Milestone Functions to be Written

Listed below are the modules to be written for the milestone.

**hash.s**

```c
int hash( char const * str );
```

This function will be used to create the hash key of a string. This function creates an integer hash key from `str` with the following hashing algorithm. Your task is to translate this algorithm into assembly. Use the global variables defined in `pa3Globals.c` to access these constants in assembly. The return value can be negative because of overflow.

```c
    int hash = HASH_START_VAL;
    int strLen = strlen(str);
    for( int i = 0; i < strLen; i++ ) {
        hash = hash * HASH_PRIME + str[i];
    }
    return hash;
```

**Return Value:** The hash key of `str`.

**getTableIndex.s**

```c
int getTableIndex( char const * name, size_t tableSize );
```

This function will calculate the index into the hashtable for the string `name` in a hashtable of size `tableSize`. Please strictly follow the order of the steps below.

1. Calculate the hash key (using your `hash()` function) of the `name` string.
2. If the hash key is negative, make it positive (you may use the ARM `neg` instruction).
3. Map the hash key to a valid index in the table by modding the hash key by the `tableSize` (use your `myRemainder()` function).

**Return Value:** The index of `name` in the hashtable.

**parseCmd.c**

```c
int parseCmd( const char * const cmdString, const char * const commands[] );
```

This function checks to see if the `cmdString` is in the `commands` array (`man -s3 strncmp`). Make sure to have appropriate null checks for both of the pointers passed in. If either pointers are null, return -1.

**Return Value:** If `cmdString` is found, return the index of `cmdString` in the `commands` array. Otherwise, return -1 to indicate `cmdString` was not in the `commands` array.

**convertStr.c**

```c
long convertStr( char * str, int * errorFlag );
```

This function converts the input `str` to a long value using `strtol()`. If any errors occur during the conversion, set the `errorFlag` to 1, print out the appropriate error message, and return -1. If no error occurs...
during conversion, return the converted long value, and set errorFlag to 0. [Note that the errorFlag is a pointer, so “setting” the errorFlag to a certain value, means setting the value that errorFlag points to.] If either of the pointers passed in are NULL, return indicating failure.

**Reasons for error:**
When either type of error occurs, set the errorFlag to be 1.

- Input str is a number too large to be converted to a long: use snprintf() and perror() to print out the error message (STR_ERR_CONVERTING)
- Input str is not a valid number: print the error message (STR_ERR_NOTINT)

**Return Value:** Converted long on success, -1 on failure.

---

nameQuest.c
nameData_t * nameQuest( char * name, tableEntry_t * te );

This function performs a linear search through the array of nameData_t structs in the table entry te, looking for the struct containing the specified name (man -s3 strncmp). The function returns a pointer to the matching nameData struct, if it exists. If either of the pointers passed in are NULL, return NULL.

**Return Value:** If found, return a pointer to the struct containing name. Otherwise return NULL.

---

populateNameData.c
int populateNameData( namesTable_t * table, char * dirName );

This function will populate the table by reading in data from files inside the directory dirName (if either pointers passed in are NULL, just return indicating failure). You will need to open the directory, open each file inside the directory, populate the table with data from each file, and then close all of the files. Following the steps below should aid you in this process.

**Figure out what files we will collect data from:**

- First you will need to open the directory (man -s3 opendir). Then you will need to copy all of the filenames from that directory into an array of strings (man -s3 readdir). Note that you will need to store the path to the file. See the format string in pa3Strings.h and consider using snprintf(). Some dynamic memory allocation here would be useful. Note: you need to ignore the current directory and parent directory filenames ("." and "." -- use constants in pa3.h) as these will also be found by readdir().
- Next you will need to sort the array of filenames as readdir() does not guarantee any specific order of reading. You will need to use filenameCompare() and qsort() (man -s3 qsort)

**Read in the name data. Perform the following for each quadrennium file:**

- Read each line of the file one at a time (read up to MAX_LINE_LEN characters for each line). Continue to read lines until you reach the end of the file or MAX_RANK female and MAX_RANK male names have been read.
- For each line read from the file:
  - The name, gender, and rank need to be extracted from the line read (fgets reads a '\n' when reading each line. strchr() should help you here). Next, ensure the gender is an uppercase
character (man -s3 toupper). Then use convertStr() to convert the rank to an integer value.

- Now that each piece of data is separated, you need to store this information in the table.
  - Use getTableIndex() to determine what index in the table this name hashes to.
  - If the entry has no names, then calloc() a nameData struct, initialize the values, update the table entry, and read the next line from the file.
  - Otherwise, if the entry already has at least one name attached to it, you will need to determine if the name you read in already exists. Use nameQuest() here.
  - If the name was found, update the appropriate fields of the nameData struct and read the next line from the file.
  - Otherwise, you will need to dynamically allocate a new nameData struct at the end of the array. This is where you need to use realloc() (man -s3 realloc). Make sure you really understand how this function works before using it. If you don't use realloc() properly, you could introduce incredibly challenging bugs into your code that could take days to debug--don't do this to yourself! After you have allocated the memory, you need to initialize everything to zero inside of the new struct (man -s3 memset). After you have populated all of the fields in the struct, read in the next line from the file.

- Now that you've read in all the lines in the file (or 2,000 names), you need to close the file and continue on to the next file.

After all files have been read and the table has been populated, you need to make sure all files have been closed, the directory has been closed, and you free all dynamically allocated memory that is no longer needed.

Some notes about the directories and data files:
- Each file corresponds to one quadrennium. Each line of the file will be formatted as:
  name,gender,rank (e.g. Mary,F,1).
- You cannot assume any specific ordering within the file (i.e. it could be male, female, or random rank ordering), so keep this in mind when writing your parsing algorithm.
- We have provided you with a few test directories containing these quadrennium files. Some things that you can assume will always be the same: There will always be NUM_QUADRENNIUM files in a directory. The filenames will sort properly so that, if you use fileNameCompare(), the first file will correspond to the first possible quadrennium and they will increase in that order appropriately.
- When testing your function, you should always test on a full directory (like the ones we provided), otherwise, it will be difficult to properly populate your table.

Reasons for error:
There are three types of errors you could encounter in this function:
- Opening a file/directory fails. If this is the case, you need to call perror() on that file/directory name. Note: to match the public executable, print a newline to stderr before and after your perror() call.
- Error with dynamically allocating memory. You should call perror(), passing in a null pointer as the parameter, and perror() will print the correct error message.
- Error with the format of the data. More specifically, if there are not two commas in each line of the file, this is a error with the name data. Print out the appropriate error message (see pa3Strings.h).
- If any of the above errors are encountered, make sure you free all memory that you allocated in this file before returning.

Return Value: 0 on success, -1 on failure.
Post-Milestone Functions to be Written

Listed below are the modules to be written after the milestone functions are complete.

**main.c**

```c
int main( int argc, char * argv[] );
```

This is the main driver for your program. Its main tasks are to parse the command line arguments, build the hashtable of name data, and enter user interactive mode, allowing the user to examine the data for a specified name. The user will specify whether to display this information as a graph or as a list of statistics.

**Parsing command line arguments:**

We will use `getopt()` to parse the flags described in the long usage statement. See the man page for a very helpful example of how to do this (`man -s3 getopt`). Make sure to use the constants for argument parsing defined in `pa3.h`. The implementation details for the flags are listed below.

<table>
<thead>
<tr>
<th>Short Flag</th>
<th>Required Argument</th>
<th>How to Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h</td>
<td>N/A</td>
<td>Print out the long usage to <code>stdout</code> and return indicating success.</td>
</tr>
<tr>
<td>-s</td>
<td>table size</td>
<td>Convert the table size from a string to a long (using <code>convertStr()</code>) and save the converted value for later use.</td>
</tr>
<tr>
<td>-d</td>
<td>data directory name</td>
<td>Save the directory name for later use.</td>
</tr>
</tbody>
</table>

In the case where `getopt()` detects an invalid flag or missing flag argument, it will automatically print an error message for you. If this happens, you also need to print the short usage to `stderr` and return indicating failure.

If all flags are parsed without any errors, make sure you also check for extra arguments after `getopt()` completes. Then, use `isInRange()` to make sure the table size is in the range 100-20001 (exclusive on upper bound).

**Build the namesTable:**

From parsing the command line arguments, we should now know the directory name (which should be set to default directory if the user didn’t enter one -- see `pa3.h`) and the size of the hashtable (should be set to default size if the user didn’t enter one -- see `pa3.h` again). With that in mind, we now need to first allocate zero-filled memory for the array of `tableEntry` structs (as shown in blue in the memory layout diagram up above). If that was successful, create the `struct namesTable` itself and initialize its members appropriately. Then populate the namesTable by calling `populateNameData()`. Your namesTable should now be filled like the memory diagram up above.

**Enter Interactive Mode:**

After successfully populating the namesTable, enter user-interactive mode by calling `handleUserInput()`. After user interaction is finished, don’t forget to deallocate ALL the dynamically allocated memory before you return.
Reasons for error:
In this file, as soon as an error is detected, print the appropriate error message(s) to stderr (see pa3Strings.h), and return indicating failure.

- `convertStr()` detects an error when converting the hashtable's size (print short usage)
- `getopt()` indicates an invalid flag or missing flag argument (print short usage)
- Extra arguments detected after `getopt()` completes (STR_ERR_EXTRA_ARGS, short usage)
- Table size is not in range (STR_ERR_RANGE)
- Ran out of memory when allocating the array of TableEntry structs (STR_ERR_MEM_EXCEEDED)
- `populateNameData()` failed to populate the namesTable (NO error message required)
  - Don't forget to free ALL the dynamically allocated memory before returning!

Return Value: EXIT_SUCCESS on success, EXIT_FAILURE on failure.

handleUserInput.c

```c
void handleUserInput( namesTable_t * table );
```

This function allows the user to interactively search through the table for specific name. This function first prints out the list of possible commands (provided in pa3Strings.h: HELP_STR), then prompts the user to enter a command. This function re-prompts the user until the user enters ctrl-D (sends the EOF character).

An overview of the user-interactive process:

1. Read the commands and argument entered by the user (man -s3 fgets). Note that fgets() will include the newline character as part of the string read, so as soon as you get a new input, replace the newline character with the NUL character (man -s3 strchr).
2. Determine which command was entered by the user.
   - Use `strtok()` and the provided token separator string SEP to extract the command portion of the user's input from what was read by fgets().
     - If the string could not be tokenized, reprompt the user.
   - Check to see if the command entered was valid by calling `parseCmd()`. Note that you will need to define the commands array at the top of this file like so:
     ```c
     static char const * const COMMANDS_LIST[] = COMMANDS;
     ```
     - If it wasn't valid, print the ERR_CMD_NOT_FOUND error and reprompt the user.
3. If the "help" command was entered, print the HELP_STR and then reprompt the user.
4. Otherwise, tokenize the string again to get the name to search for.
   - If no argument was entered, print the ERR_NAME_NOT_ENTERED error and reprompt the user.
   - Otherwise, format the name so that the first character of the name is uppercase and the rest are lowercase.
   - Use `getTableIndex` to calculate the index of this name in the table. Then try to find the name in table entry at this index using `nameQuest()`.
     - If the name was not found, print STR_NAME_NOT_FOUND (to stdout).
     - Otherwise, check for commands in the following order.
5. If the command entered is stats, print out the stats by calling `printStats()`.
6. If the command is graph, print out the graph using `generateGraph()` and `drawGraph()`.
7. Reprompt the user, and repeat the above steps.
printStats.c
void printStats( nameData_t * namePtr );

This function prints out the ranking information for the given nameData struct.

First print the header for the stats by using the provided STATS_HEADER. Then go through the arrays of name ranks for both male and female, printing out their ranking information for each quadrennium:

- If the rank of the name as a female name is smaller than MIN_RANK, only print out the rank for male using STATS_FMT_NO_FEMALE (use NO_RANK_CHAR for the rank of the female name).
- If the rank of the name as a male name is smaller than MIN_RANK, only print out the rank for female using STATS_FMT_NO_MALE (use NO_RANK_CHAR for the rank of the male name).
- If neither rank for male nor female is greater than MIN_RANK, print out the stats using STATS_FMT_NEITHER (use NO_RANK_CHAR for both the male and female ranks).
- If both ranks are valid (greater than MIN_RANK), print out the ranks for both female and male using STATS_FMT_BOTH.

After printing out the ranking information for the name in each quadrennium, you also need to print out a message indicating whether the name in a certain quadrennium has the highest rank over its ranking history from 1880 to 2016 (you should keep track of such information as you print the ranks above):

- If the name as a female name does have a highest record over the years, print out the rank as well as the quadrennium in which the rank is achieved (using the provided format string F_HIGH_RANK). Do the same if the name as a male name has a highest record (M_HIGH_RANK).
- If the name as a female name was never in the top 1000 female names from 1880 to 2016, then print out a message indicating that (using the provided format string F_NO_HIGH_RANK). Do the same if the name as a male name was never in top 1000 (M_NO_HIGH_RANK).

generateGraph.c
void generateGraph( nameData_t * name, char arr[GRAPH_ROWS][GRAPH_COLS] );

This function will populate arr, a 2d-array that contains the graph for the name data within name. This graph will be printed later, and therefore must be initialized in such a way to make printing the graph trivial. To initialize arr, this function should go through the male and female ranks for each quadrennium and set the marker at the appropriate index. Please refer to the reference executable for further clarification on the format of these graphs.

Things to consider:

- Make considerations for the fact that rank != index. For example, a name that has rank 50 for a particular quadrennium should be placed at the 40-79 range, which would be index 1.
- The last two characters in each row should be the newline character and the null character.
- If an index does not have any markers, then a space character should be at that index.
- If the male and female ranks are both at the same index for a given quadrennium, the intersection marker should be placed at the index instead of the male or female marker.

drawGraph.c
void drawGraph( char arr[GRAPH_ROWS][GRAPH_COLS] );

Prints a graph that has been previously initialized by generateGraph() to stdout. It is recommended that you iterate through the rows of arr in order to print each line of the graph. Note that you must figure out how to print out the labels for each row and column, as well as the lines for the axes (see pa3Strings.h). Also note that the last row of the graph is for unranked data, and has its own label string that includes a '>".

Back to Top
Unit Testing

You are provided with only one basic unit test file for the Milestone function, `testpopulateNameData()`. This file only has minimal test cases and is only meant to give you an idea of how to write your own unit test files. **You must write unit test files for each of the Milestone functions, as well as add several of your own thorough test cases to all of the unit test files. You need to add as many unit tests as necessary to cover all possible use cases of each function. You will lose points if you don’t do this!** You are responsible for making sure you thoroughly test your functions. Make sure you think about boundary cases, special cases, general cases, extreme limits, error cases, etc. as appropriate for each function. The Makefile includes the rules for compiling these tests. Keep in mind that your unit tests will not build until all required files for that specific unit test have been written. These test files **will be collected with the Milestone**, they must be complete before you submit your Milestone.

**Unit tests you need to complete:**
- `testpopulateNameData.c`
- `testconvertStr.c`
- `testgetTableIndex.c`
- `testhash.c`
- `testnameQuest.c`
- `testparseCmd.c`

**To compile:**

```
$ make testpopulateNameData
```

**To run:**

```
$ ./testpopulateNameData
```

(Replace “testpopulateNameData” with the appropriate file names to compile and run the other unit tests)

README Questions

1. [AI] Why are professional engineers expected to act with integrity?
2. [Vim] How do you increment the next number under or to the right of the cursor?
3. [Vim] What is the command to turn a lowercase character into an uppercase character and vice versa?
4. [Unix] From the command line, how can you find all instances of the word “bar” in the file foo.txt, with line numbers?
5. [Vim] What is the command to repeat the previously executed command?

Extra Credit

There are 5 points total for extra credit on this assignment.

- **Early turnin:**
  - [2 Points] 48 hours before regular due date and time
  - [1 Point] 24 hours before regular due date and time
    (it’s one or the other, not both)

- [3 Points] Bar Graph

Bar Graph

To implement this extra credit you will be making modifications to how the graph appears. Make sure your non extra credit portion of this assignment has no issues before beginning the extra credit. You will not be making changes to the non extra credit portion, instead you will copy some of your existing files to new EC files and compile into a new executable.

Back to Top
Rather than giving you example input or output, use the public executable to see what the graph should look like (test it with a variety of names to see how it is different).

Getting started:
Copy over new files from the public directory.

```
$ cp ~/../public/pa3StarterFiles/Makefile ~/pa3
$ cp ~/../public/pa3StarterFiles/pa3EC.h ~/pa3
```

Copy your non extra credit files to extra credit files.

```
$ cp ~/pa3/generateGraph.c ~/pa3/generateGraphEC.c
$ cp ~/pa3/drawGraph.c ~/pa3/drawGraphEC.c
```

In the two files above, be sure to include `pa3EC.h` for the extra constants and strings.

Compiling:
To compile the extra credit:

```
cs30xyz@pi-cluster-001:pa3$ make pa3EC
```

Sample Executable:

```
cs30xyz@pi-cluster-001:pa3$ ~/../public/pa3ECtest
```

A note on function names: **Do not** change the names of the functions for `generateGraph()` or `drawGraph()`, only change the name of the file. The Makefile has been setup to use these extra credit files to create your extra credit executable, your program will not compile if you change the function names.

---

**generateGraphEC.c**

```c
void generateGraph( nameData_t * name, char arr[GRAPH_ROWS][GRAPH_COLS] );
```

This function will be very similar to the non extra credit function with a few changes. See the output from the public executable for how the graph should now look. For Male values we are now using '| ' to denote this in the graph, for Female values we are using '- ', and for intersections we are using '#' (use the constants in `pa3EC.h`). These characters should fill the whole column based on the Male and Female data (rather than just being a single data point at the top of the column--run the public executable to see what this looks like). It is up to you to determine the logic required to do this.

---

**drawGraphEC.c**

```c
void drawGraph( char arr[GRAPH_ROWS][GRAPH_COLS] );
```

You will only need to make one modification to this function, printing out the legend before the graph. This is needed since the characters to denote Male or Female are no longer obvious without explanation. Use the declared legend string in `pa3EC.h` and print it out before the graph.
**Turnin Summary**

See the turnin instructions [here](#). Your file names must match the below *exactly* otherwise our Makefile will not find your files.

**Milestone Turnin:**
**Due:** Wednesday night, May 16 @ 11:59 pm

**Files required for the Milestone (along with a unit test file for each):**

<table>
<thead>
<tr>
<th>hash.s</th>
<th>getTableIndex.s</th>
<th>parseCmd.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>nameQuest.c</td>
<td>convertStr.c</td>
<td>populateNameData.c</td>
</tr>
</tbody>
</table>

**Final Turnin:**
**Due:** Wednesday night, May 23 @ 11:59 pm

**Files required for the Final Turn-in:**

<table>
<thead>
<tr>
<th>getTableIndex.s</th>
<th>convertStr.c</th>
<th>pa3.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash.s</td>
<td>drawGraph.c</td>
<td>pa3Strings.h</td>
</tr>
<tr>
<td>isInRange.s</td>
<td>filenameCompare.c</td>
<td>test.h</td>
</tr>
<tr>
<td>myRemainder.s</td>
<td>generateGraph.c</td>
<td>Makefile</td>
</tr>
<tr>
<td>testconvertStr.c</td>
<td>handleUserInput.c</td>
<td>README</td>
</tr>
<tr>
<td>testgetTableIndex.c</td>
<td>main.c</td>
<td></td>
</tr>
<tr>
<td>testhash.c</td>
<td>nameQuest.c</td>
<td></td>
</tr>
<tr>
<td>testnameQuest.c</td>
<td>pa3Globals.c</td>
<td></td>
</tr>
<tr>
<td>testparseCmd.c</td>
<td>parseCmd.c</td>
<td></td>
</tr>
<tr>
<td>testpopulateNameData.c</td>
<td>populateNameData.c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>printStats.c</td>
<td></td>
</tr>
</tbody>
</table>

**Extra Credit Files:**

| drawGraphEC.c | generateGraphEC.c | pa3EC.h |

If there is anything in these procedures which needs clarifying, please feel free to ask any tutor, the instructor, or post on the Piazza Discussion Board.