# CSc 352 (Spring 2019): Assignment 3

**Start Date:** Sat Feb 9, 2019
**Due Date:** 11:59PM Saturday, Feb 16

The purpose of this assignment is to continue to work with strings and arrays, get some practice working with char variables and ASCII encodings, and to begin using dynamic memory allocation via malloc() and calloc().

## General Requirements

1. Your C code should adhere to the coding standards for this class as listed in the Documents section on the Resources tab for Piazza.
2. Your programs should indicate if they executed without any problems via their *exit status*, i.e., the value returned by the program when it terminates:

| Execution | Exit Status |
|---|---|
| Normal, no problems | **0** |
| Error or problem encountered | **1** |

3. Under *bash* you can check the exit status of a command or program ***cmd*** by typing the command "**echo $?**" immediately after the execution of ***cmd***. A program can exit with status *n* by executing "`exit(`*n*`)`" anywhere in the program, or by having `main()` execute the statement "`return(`*n*`)`".
4. Remember your code will be graded on lectura using a grading script. You should test your code on lectura using the `diff` command to compare your output to that of the example executable.

## Testing

Example executables of the programs will be made available. You should copy and run these programs on lectura to test your program's output and to answer questions you might have about how the program is supposed to operate. Our class has a home directory on lectura which is:

**`/home/cs352/spring19`**

You all have access to this directory. The example programs will always be in the appropriate **`assignments/assg#/prob#`** subdirectory of this directory. They will have the same name as the assigned program with "ex" added to the start and the capitalization changed to maintain

camelback. So, for example, if the assigned program is **`theBigProgram`**, then the example executable will be named **`exTheBigProgram`**. You should use the appropriate UNIX commands to copy these executables to your own directory.

Your programs will be graded by a script. This will include a timeout for all test cases. There must be a timeout or programs that don't terminate will cause the grading script to never finish. This time out will never be less than 10 times the time it takes the example executable to complete with that test input and will usually be much longer than that. If your program takes an exceedingly long time to complete compared to the example code, you may want to think about how to clean up your implementation.

# Makefiles

You will be required to include a Makefile with each program. Running the command:

**`make`** ***`progName`***

should create the executable file *progName*, where *progName* is the program name listed for the problem. The gcc command in your Makefile must include the **`-Wall`** flag. Other than that, the command may have any flags you desire.

# Submission Instructions

Your solutions are to be turned in on the host **lectura.cs.arizona.edu**. Since the assignment will be graded by a script, it is important you have the directory structure and the names of the files exact. Remember that UNIX is case sensitive, so make sure the capitalization is also correct. For all our assignments the directory structure should be as follows: The root directory will be named **assg#**, where # is the number of the current assignment. Inside that directory should be a subdirectory for each problem. These directories will be named **prob#** where # is the number of the problem within the assignment. Inside these directories should be any files required by the problem descriptions. For this assignment the directory structure should look like:

```
assg3
   prob1
      count.c
      Makefile

   prob2
      rotate.c
      Makefile
```

To submit your solutions, go to the directory containing your assg3 directory and use the following command:
**turnin cs352s19-assg3 assg3**

# Problems

# prob1: count

Write a C program, in a file **count.c**, that reads in a sequence of integers and counts the number of times each integer value appears in the sequence, as specified below. (Remember you also need to include a **Makefile** that compiles **count.c** to the executable **count**.)

- **Input:** The input will consist of a sequence of integers read from **`stdin`**:

  $N$
  $A_1$ $A_2$ ... $A_N$

  (The values $N$ and $A_i$ are shown on separate lines here primarily for clarity: in the actual test inputs, they need not actually be on separate lines. If you use **scanf** to read your inputs, this should not be a problem.) Notice that unlike previous homework problems, this time you need to gather all the input before you produce your output.

  The first value, $N$, is a positive (greater than 0) integer specifying the number of integers to be processed; this is followed by a sequence of $N$ integers that are processed as described below.

- **Output:** Your program should print out the number of times each distinct integer value appears in the input sequence. These should be printed out in ascending order of the integer value using the statement

  ```
  printf("%d %d\n", input_value, count);
  ```

  where **`input_value`** is a number occurring in the input and **`count`** is the number of times that value occurs in the input sequence.

  Each distinct value in the input sequence should be printed out exactly once along with the count of the number of times it occurs.

- **Restrictions:** Your program must dynamically allocate the appropriate amount of storage for the numbers to be processed using **malloc**() or **calloc**(). **Solutions that do not do this will not receive credit**.

- **Error Cases:** Cannot read at least one integer from **`stdin`**. The first integer value is not positive. Fewer integers available to read than indicated by the initial number. (Note it is **not** an error if more than N integers are supplied. The extra integers are just ignored.)

- **Example:** Input:

```
7   12   31   -5   31   12   7   12
```

Output:

```
-5 1
7 1
12 3
31 2
```

# prob2: rotation

Write a C program, in a file **rotate.c**, that rotates a vector of integers as specified (see below) and prints out the rotated vector. (Once again, don't forget the **Makefile** which creates the executable **rotate**)

- **Input:** The input to your program will be read from **stdin**. It will consist of a sequence of integers, as follows:

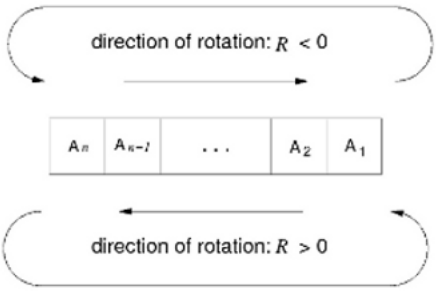
  $N$
  $A_1$ ... $A_{N-1}$ $A_N$
  $R$

  (The values $N$, $A_i$, $R$ are shown on separate lines here primarily for clarity: in the actual test inputs, they need not actually be on separate lines. If you use **scanf** to read your inputs, this should not be a problem.)

  The numbers read in have the following significance:

  - $N$ is a positive integer that specifies the size of the vector.
  - $A_1$ ... $A_{N-1}$ $A_N$ are integers (positive, zero, or negative) that constitute the elements of the vector.
  - $R$ is an integer (positive, zero, or negative) that specifies the amount by which the vector is to be rotated. The sign of $R$ specifies the direction of rotation, while the magnitude of $R$ specifies the amount of rotation.

- **Behavior:** The vector of integers read in should be rotated by the amount specified and the result printed out to **stdout** as specified below. The directions of rotation for positive and negative values of $R$ are as shown below. Positive values of $R$ cause values to be shifted "up", so with $R = 1$, $A_1$ goes to $A_2$, $A_2$ goes to $A_3$, etc.; values that are "shifted out" at the top are rotated into the bottom of the vector. The situation is reversed for negative values of $R$, so with $R = -1$, $A_3$ goes to $A_2$, $A_2$ goes to $A_1$, etc.




- **Output:** Print out the elements of the rotated vector on one line with the numbers separated by a single space.

- **Error Cases:** Cannot read at least one integer from `stdin`. The first integer value is not positive. Fewer integers available to read than indicated by the initial number or some value that cannot be interpreted as an integer is input before the indicated number of integers. If too many integers are input, the extras should be ignored.

- **Restrictions:** Your program should dynamically allocate the appropriate amount of storage for the vector using **malloc**() or **calloc**(). **Solutions that do not do this will not receive credit**.

- **Example:** If the following is input:

  5
  0 1 2 3 4
  2

  Then the output would be:

  3 4 0 1 2