CSc 352 (Fall 2025): Assignment 5

Due Date: 11:59PM Friday, Oct 10

The purpose of this assignment is to provide you with additional experience using the functions malloc or calloc and getline. It also will give you further practice with pointers and parsing strings.

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. This includes protecting against buffer overflows whenever you read strings.
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.
5. To get full points your code should compile without warnings or errors when the **-Wall** flag is set in **gcc**
6. Anytime you input a string you must protect against a buffer overflow. Review slides 72 – 77 of the basic C deck.
7. Any time you call a system function like malloc() or calloc() that might fail, you must check the return value to ensure that it succeeded.

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/fall25`

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.

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:

```
assg5
   prob1
      count2.c
      Makefile
   prob2
      lineSum.c
      Makefile
```

To submit your solutions, go to the directory containing your assg5 directory and use the following command:

turnin cs352f25-assg5 assg5

prob1: count2

Write a C program, in a file **count2.c,** and a Makefile that creates an executable called **count2** that reads in a sequence of integers and counts the number of times each integer value appears in the sequence, as specified below. (This is almost exactly the same as the **count** problem from the previous assignment, the only difference being that you are not told, ahead of time, how many integers will appear on the input.)

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

  A_1 A_2 ... A_N

- **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:** You are not allowed to just create a big array and resize it when needed. Instead **you must store the input in a linked list**. The memory for this list should be dynamically allocated as the input is read in. Solutions that don't follow this restriction will receive a 0.

- **Error Cases:** A non-integer value in the input stream should cause the program to print an error message to stderr and exit immediately with an exit code of 1. There will be nothing written to stdout in this case.

- **Makefile:**

  In addition to your source files, you should submit a make file named **Makefile** that supports at least the following functionality:

  make count2
  Compiles the C source code to create an executable named **`count2`**. The compiler options used should include **`-Wall`**.

- **Example:** Input:

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

Output:

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

The input is only shown on one line for brevity. The input may have any amount of white space (including newlines) separating the numbers.

prob2: lineSum

Write a C program, in a file **lineSum.c**, and a Makefile that creates an executable called **lineSum** which reads in input a line at a time, parses out the integers, and prints out the sum as specified below.

- **Input:**

  A sequence of lines to be read in from `stdin`, each line consisting of a non-empty sequence of non-negative decimal integers, with adjacent numbers separated by whitespace. (Thus, leading negative signs are not allowed.)

- **Output:**

  For each line read in, the sum of the numbers in that line, printed to `stdout` using the following statement:

  ```
  printf("%d\n", sum)
  ```

  where `sum` is the number being printed out.

- **Error Conditions:**

  Negative integer or non-whitespace values in the input and empty lines (lines containing only whitespace) are both errors. In each case, print an error message to `stderr`, skip the culprit line (i.e., don't print any value for it), and continue processing. Use the exit status of your program to indicate whether any errors were encountered during processing.

- **Suggested Approach:**

  Reading in the input numbers using `scanf` won't work. Instead, read in each line using `getline()`. Iterate over the line thus read in using `sscanf()` to read the numbers one after another. (Read the man page on sscanf)

  Note that in order for this to work, once a number is read from the string, the string that is passed into `sscanf()` the next time should be the "rest of the string" so that you don't read the same number repeatedly. You can do this by using a pointer to keep track of where you are in the string and moving this pointer past each number that is read.

- **Restrictions:**

  One of the goals of this assignment is to teach you to move a pointer through a string while parsing. Therefore, you are NOT allowed to use a library function like strtok() to process the input. See the functionsList document on Piazza or D2L for which functions

are allowed.

- **Assumptions:**

  You may assume no integer on the input line is too long to fit into a variable of type int. Likewise, you may assume the sum is not larger than an int.

- **Makefile:**

  In addition to your source files, you should submit a make file named **Makefile** that supports at least the following functionality:

  make lineSum
  Compiles the C source code to create an executable named **`lineSum`**. The compiler options used should include **`-Wall`**.

  Note: the 'S' in **lineSum** is uppercase. All the other letters are lowercase.

- **Example:**

  Suppose the input consists of the following lines:

```
12  128   23   97
   34   23   19835   257   87
176
982   83
```

  Then the output should be

```
260
20236
176
1065
```

If the input was coming from the keyboard, the output would be printed after each line was read in. In other words, you should get a line, process it, print the output, and then get the next line. This problem gives you more practice using **getline** and processing the string it returns. You do not need to save more than one line at a time in memory.