CS 1313 010: Programming for Non-Majors with C, Spring 2025
Programming Project #5: Big Statistics
Due by Wednesday April 9 2025 9:50am Central Time

This fifth programming project will give you experience writing programs that involve `for` loops and arrays. This programming project will use the same development process as in Programming Projects #2, #3 and #4, and will be subject to the same rules and grading criteria, along with some new criteria.

This specification will be less detailed than for previous programming projects. **YOU ARE EXPECTED TO KNOW HOW TO PERFORM BASIC TASKS WITHOUT HAVING TO BE TOLD EXPLICITLY,** based on your experience with previous programming projects.

To get full credit on this programming project, you **MUST** use `for` loops and dynamically allocated arrays appropriately.

NOTE: The next programming project (#6) will be **BASED ON PP#5**, and you will use your program for PP#5 as the starting point. So it's important to do PP#5 well and completely.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!

For this project, you are **ABSOLUTELY FORBIDDEN** to have any user-defined functions.
(We'll use those in PP#6.)

I. PROJECT DESCRIPTION

You've been hired to develop statistics software. Specifically, your software will calculate the same statistics as in the statistics program in PP#3.

In each individual run of your software, you will input two lists of numbers, and these two lists will have the same length, a length that will be input (and idiotproofed) at runtime, just before allocating and then inputting the lists.

You will then calculate all the same values that you calculated in PP#3, but using these two lists of numbers.

Specifically, for each list: the arithmetic mean, the geometric mean, the root mean square and the harmonic mean.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
This programming project specification contains many small code examples. In most cases, these code examples will be extremely useful in your actual PP#5. **WE URGE YOU TO USE THEM.**

NOTE: Except where and as explicitly permitted in writing (for example, in a Programming Project specification, as above), you are **ABSOLUTELY FORBIDDEN** to **COPY EVEN A SINGLE CHARACTER** from, or to have **ANY** shared code with, **ANY** other entity, whether a human being, a text resource, a computing resource or anything else, whether in person, on a local computer, online or anywhere else. It's **INCREDIBLY EASY** for us to detect such copying, so **DON'T EVEN THINK ABOUT IT!**

HOW TO COPY-AND-PASTE IN PUTTY

In PP#5, you'll find that using copy-and-paste will be **EXTREMELY** helpful.

Here's how to copy-and-paste in PuTTY:

1. Using your mouse (or laptop touchpad), position the mouse cursor at the very left of the text that you want to copy.
2. Hold down the left mouse button.
3. To copy, while holding down the left mouse button, drag the mouse cursor over the text that you want to copy, which will highlight it. The text is now copied (so no need to Ctrl-C or anything).
4. Release the left mouse button.
5. Position the green text cursor where you want to paste.
6. To paste, (single) click the right mouse button.

NOTE: This method only works with PuTTY.

II. WHAT TO DO FIRST

Add the new program into your `makefile` in the usual way, as well as the example program (see below).

III. WHAT TO DO SECOND

For the example program in "Array Lesson 3," slides #18-25:

Type in, compile and run that example program, using the input values on slide #26 of the same lecture slide packet.

Then, comment that example program, and compile and run it again, with the same inputs.

Then script it in the usual way, with the same inputs.

IV. PROGRAM DESCRIPTION

Write a program to calculate statistics from input data. The program body **MUST** be broken into **THREE** subsections, rather than the usual four (there **WON'T** be a greeting subsection):

1. an input subsection;
2. a calculation subsection;
3. an output subsection.

Because of how data will be input (see below), **THERE WON'T BE A GREETING SUBSECTION.**

You are **ABSOLUTELY FORBIDDEN** to have:

- **ANY** calculations in the input subsection (and the only outputs should be idiotproofing error messages);
- **ANY** inputs or outputs in the calculation subsection;
- **ANY** inputs or calculations in the output subsection.

Thus, the subsections **MUST BE TOTALLY SEPARATE,** and **MUST BE CLEARLY LABELED.**

For this project, `if` blocks can occur in any subsection of the execution section (body) of the program, and the same is true of `for` loops.

A. ARRAY DECLARATIONS

You MUST use **DYNAMIC** memory allocation and deallocation for ALL arrays. (See "Array Lesson 3," slides #10-14, and the examples on slides #15-26.) Any statically allocated arrays will be **SEVERELY PENALIZED**.

Therefore, ALL arrays MUST be declared as **POINTERS.** For example:

```
float* list1_input_value = (float*)NULL;
```

WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#5!

B. INPUT SUBSECTION

B.1. INPUT THE SHARED ARRAY LENGTH

The program will take its input from a data file, rather than from a user typing live at the keyboard (see section VI, **INPUT DATA FILES**, below).

The input data will be in the following format:

1. a single length, which is shared by both of the lists of numbers (for example, if the length that is input is 22, then the first list will have length 22 and the second list will also have length 22);
2. for each element in the two lists:
   (a) the value of that element of the first list;
   (b) the value of that element of the second list, on the same line of input text.

Several such data files will be provided, each representing an individual run. **YOU** should determine how to input the data **BY EXAMINING THE INPUT DATA FILES** (see **HOW TO FIND AND EXAMINE THE INPUT DATA FILES**, below).

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!

Because of how the data will be input, **YOU WON'T PROMPT THE USER FOR THE INPUTS** (see **HOW THE DATA WILL BE INPUT**, below).

You MUST store the input data in appropriate one-dimensional arrays. You are **ABSOLUTELY FORBIDDEN** to use multidimensional arrays in PP#5.

C. IDIOTPROOFING

YOU MUST IDIOTPROOF ANY input that needs idiotproofing, to make sure that it has an appropriate value. **YOU** are responsible for figuring out all of the possible cases of idiocy that could come up. **ALL IDIOTPROOFING MUST BE COMPLETED BEFORE ANY CALCULATIONS ARE PERFORMED;** that is, idiotproofing belongs in the input subsection.

Note that, for this programming project, you are **ABSOLUTELY FORBIDDEN** to use `while` loops for your idiotproofing; that is, upon detecting idiocy, the program **MUST EXIT.**

Idiotproofing error messages MUST be clear, complete English sentences that **COMPLETELY AND UNAMBIGUOUSLY** state the nature of the error. Thus, **EACH ERROR MESSAGE MUST BE UNIQUE.** For example, an error message might be:

```
ERROR: You cannot have a list length of -3.
```

D. ALLOCATING ARRAYS

You **MUST** use **DYNAMIC** memory allocation and deallocation for **ALL** arrays. (See "Array Lesson 3," slides #10-14, and the examples on slides #15-26.) Any statically allocated arrays will be **SEVERELY PENALIZED**.

Therefore, **ALL** arrays **MUST** be declared as **POINTERS.**

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!

Note that ALL of the arrays MUST be allocated, at runtime, in the execution section, IMMEDIATELY AFTER INPUTTING AND IDIOTPROOFING THEIR SHARED LENGTH.

In other words, once you have input and idiotproofed the shared length of the arrays, you **MUST IMMEDIATELY** allocate and check each of the arrays.

After allocating any array, the program **MUST IMMEDIATELY** check whether that array was allocated successfully, and if not, the program **MUST** output a suitable, **UNIQUE** error message and then **MUST EXIT.**

For details on dynamically allocating and deallocating arrays, see the lecture slide packet "Array Lesson 3," slides #10-14, and the examples on slides #15-26.

E. CALCULATION SUBSECTION

In the calculation subsection, the program **MUST** calculate the following values, in the following order:

- for each list of numbers:
  - their arithmetic mean;
  - their geometric mean;
  - their root mean square;
  - their harmonic mean.

In any `for` loop in the calculation subsection, you **MUST** calculate **EXACTLY ONE** kind of result; that is, you are **ABSOLUTELY FORBIDDEN** to calculate multiple kinds of results in a single `for` loop.

For example, the `for` loop that calculates the artihmetic mean of the first list is **ABSOLUTELY FORBIDDEN** also to calculate the arithmetic mean of the second list.

However, within a particular `for` loop, you may choose to calculate temporary scalar variables representing various subexpressions.

F. OUTPUT SUBSECTION

In the program's output subsection, you **MUST** output the following:

- the (shared) length of the lists;
- all of the values of the first list, in order;
- all of the values of the second list, in order;
- each of the items calculated, in the same order as in the calculation subsection.

Each output **MUST** be accompanied by helpful explanatory text; for example, the **OUTPUT** might look like this:

```
The lists have 4 elements each.
...
The arithmetic mean of the first list is 275.375000.
...
```

You may output these quantities in any format that you like, as long as the meaning of the quantities is **CLEARLY EXPLAINED** in the outputs.

You are welcome to use format descriptors on your placeholders (for example, `"%10.5f"`), but you **AREN'T** required to use them.

G. DEALLOCATING ARRAYS

At the end of the program, after the output subsection, you **MUST** deallocate each of the arrays that were allocated in the input subsection, using a `free` statement for each, and then nullifying the pointer, like so:

```
free(list1_input_value);
list1_input_value = (float*)NULL;
```

WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#5!

The deallocations **MUST** occur in the **OPPOSITE ORDER** from the allocations; that is, whichever array was allocated first **MUST** be deallocated last, and so on.

For details, see the lecture slide packet "Array Lesson 3," slide #14 and the example on slide #25.

V. RUNS

Run this program several times, using the several different input files that are available (see below). The runs **MUST** be in alphabetical order according to the input file names.

The order of the runs in your script file **MUST** be:

- all `actual_` files, in alphabetical order, followed by
- all `idiot_` file(s), in alphabetical order.

See section VI.B, below, for how to do the runs.

VI. INPUT DATA FILES

A. HOW TO FIND AND EXAMINE THE INPUT DATA FILES

The input files for your runs can be found on `ssh.ou.edu` in the directory `~neem1883/CS1313pp5`. You can find the names of all of the data files using the `ls` command:

```
ls   ~neem1883/CS1313pp5
```

The directory contains several data files; some are actual data and some are idiotproofing test files. You **MUST** perform the runs in alphabetical order.

Actual (non-idiotproofing) test files have file names beginning with

`actual_`

Idiotproofing test files have file names beginning with

`idiot_`

You **SHOULD CLOSELY EXAMINE** (but not change) the contents of each of the data files using `nano`:

```
nano   ~neem1883/CS1313pp5/actual_2.txt
```

B. HOW THE DATA WILL BE INPUT (HOW TO DO THE RUNS)

For this programming project, **YOU WON'T PROMPT THE USER FOR THE INPUTS,** because there won't be a user as such. Instead, the inputs will come from a file. To get the inputs from the file, you'll use a command like this at the Unix prompt:

```
big_statistics < ~neem1883/CS1313pp5/actual_1.txt
```

This use of a file is referred to as *redirecting input*. The less than symbol < indicates that the input will come from the file named `actual_1.txt`. In other words, as far as the program is concerned, the file will appear to be a user typing at the keyboard, and the program will accept input from the file exactly as if that input were being typed at the keyboard by a real live user. Thus, you **MUST** write your `scanf` statements exactly as if a user were going to be typing the data at the keyboard, but without the user needing to be prompted.

However, because there isn't actually a real live user, it isn't necessary to greet the user nor to prompt for inputs; the data file won't understand the prompts anyway, so to speak.

Your run commands **MUST** look like this example:

```
big_statistics  <  ~neem1883/CS1313pp5/idiot_1.txt
```

This means, "Run the executable named `big_statistics`, redirecting input from the file named `idiot_1.txt` that's in the directory named `~neem1883/CS1313pp5`".

VII. ADDITIONAL GRADING CRITERIA

All grading criteria for Programming Projects #2, #3 and #4 apply. In addition:

1. **Declaration order:**
   In the declaration section, the order of declarations **MUST** be:
   (a) named constants: `float` scalars followed by `int` scalars;
   (b) variables, in the following order:
      i. arrays: `float` arrays followed by `int` arrays;
      ii. scalars: `float` scalars followed by `int` scalars.

2. **Block closes of `for` statements:**
   ALL block closes associated with `for` statements **MUST** be followed, on the same line, by a space, a comment open delimiter, a space, the keyword `for`, a space, the counter variable, a space, and a comment close delimiter. For example:

```
for (element = first_element; element < number_of_elements; element++) {
    scanf("%f %f",
        &list1_input_value[element], &list2_input_value[element]);
} /* for element */
```

   WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#5!

3. **Indenting** `for` statements and their associated block closes:
   For a given `for` loop, the `for` statement and its associated block close **MUST** be indented identically, and this indentation amount **MUST** be appropriate with respect to their position within the program.

4. **Indenting** inside `for` loops:
   For a given `for` loop, **ALL** statements **INSIDE** the `for` loop **MUST** be indented **FOUR SPACES** farther than the `for` statement and its associated block close. For example:

```
sum = initial_sum;
for (element = first_element; element < number_of_elements; element++) {
    sum = sum + list1_input_value[element];
} /* for element */
list1_arithmetic_mean = sum / number_of_elements;
```

   WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#5!

5. **Commenting** `for` loops:
   Each `for` loop **MUST** be preceded by a comment that describes what the loop as a whole does. For example:

```
/*
 * Calculate the arithmetic mean of the first list of input values.
 */
sum = initial_sum;
for (element = first_element; element < number_of_elements; element++) {
    sum = sum + list1_input_value[element];
} /* for element */
list1_arithmetic_mean = sum / number_of_elements;
```

   WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#5!

6. **Commenting inside** `for` loops:
   A statement inside a `for` loop **MUST** be preceded by a comment that describes what the statement does. The comment **MUST** be properly indented, so that the asterisks of the comment line up with the statement. For example:

```
sum = initial_sum;
for (element = first_element; element < number_of_elements; element++) {
   /*
    * Increase the first list's sum by the current element of
    * the first list.
    */
    sum = sum + list1_input_value[element];
} /* for element */
list1_arithmetic_mean = sum / number_of_elements;
```

   WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#5!

7. For PP#5, you are allowed to use **ONLY** the following kinds of programming constructs in your C source file, but **NO OTHER KINDS OF PROGRAMMING CONSTRUCTS**:
   - all programming constructs allowed in PP#2, PP#3 and PP#4;
   - dynamically allocatable array declarations;
   - dynamically allocatable array allocations (`malloc`);
   - `for` loops;
   - dynamically allocatable array deallocations (`free`).

   Use of any other kind of programming constructs in your C source file might result in **SEVERE PENALTIES**, at the instructor's sole discretion.

VIII. DEBUGGING VIA `printf` STATEMENTS

The best mechanism for debugging this program is to put in lots of `printf` statements that show where in the program the run currently is. For example:

```
for (element = first_element; element < number_of_elements; element++) {
    printf("About to input data for element #%d.\n", element);
    scanf("%f %f",
        &list1_input_value[element], &list2_input_value[element]);
    printf("Done inputting data for element #%d:\n", element);
    printf("  list1_input_value[%d]=%f, list2_input_value[%d]=%f\n",
        element, list1_input_value[element],
        element, list2_input_value[element]);
} /* for element */
```

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!

Once you've completed debugging, you **MUST** delete **ALL** debugging `printf` statements. **EXTRANEOUS OUTPUTS WILL BE SEVERELY PENALIZED.**

IX. WHAT TO SUBMIT

Upload to the Canvas PP#5 dropbox your summary essay (.docx, .doc or .pdf), your example script file (.txt), your big statistics C source file (.c), and your big statistics script file (.txt).

EXTRA CREDIT

HELP SESSION BONUS EXTRA CREDIT

You can receive an extra credit bonus of as much as 5% of the total value of PP#5 as follows:

1. Attend at least one regularly scheduled CS1313 help session for at least 30 minutes, through Wed Apr 9.
2. During the regularly scheduled help session that you attend, work on CS1313 assignments (ideally PP#5, but any CS1313 assignment is acceptable). **YOU CANNOT GET EXTRA CREDIT IF YOU DON'T WORK ON CS1313 ASSIGNMENTS DURING THE HELP SESSION.**

BONUS VALUE NOTICE: Through Tue Apr 1, the extra credit bonus will be worth **5%** of the total value of PP#5; from Mon Apr 7 through Wed Apr 9, the extra credit bonus will be worth **only 2.5%** of the total value of PP#5. That is, **YOU'LL GET TWICE AS MUCH EXTRA CREDIT DURING THE FIRST WEEK AS DURING THE FINAL WEEK.**

NOTE: This extra credit bonus **WON'T** be available on any other programming project unless explicitly stated so in the project's specification.