Introduction

This week's lab will be centered around helping you better understand code and the more challenging problems in computer science.

Lab Objectives

By successfully completing today's lab, you will be able to:

- Better read code for understanding of its functionality
- Conduct a basic analysis of sorting algorithms
- Implement a C program that reads values in from a file.
- Practice creating a multifile application

Prior to Lab

- You should be familiar with functions (zyBooks chapter 6), file input (zyBooks chapter 9), pointers (zyBooks chapter 8), recursion (zyBooks chapter 10), be able to look up sorting algorithms (zyBooks chapter 11), and know how to write a multifile application (zyBooks chapter 14).

Deadline and Collaboration Policy

- This assignment is due by 11:59 PM on Friday (12/6/2019) via Canvas.
    - More instructions on what and how to submit are included at the end of this document.
- You should write your solutions to this lab by yourself. In this lab, **you should not talk about specific code to anyone but a course instructor or lab teaching assistant. If you speak with a TA at the TA help desk, you should document the TA's name in the academic honesty code header.**
- Your zyBooks chapters and lecture slides are available resources you can use to assist you with this lab.

Lab Instructions

The Problem:

Computer science is no more about computers than astronomy is about telescopes. – Edsger Dijkstra

In Computer Science, there are often many problems that are difficult to solve. For example, you may recall with our Magic Squares lab, that checking the answers to a problem is often very easy, but finding that answer is very hard, because of the sheer number of possible combinations. Another difficult problem in computing is sorting.

While humans can very easily take a look at a list of numbers (like *1, 2, 3, 15, 4, 200, 5, 6, 7*) and immediately determine that they're out of order and what the proper order would be, it is more difficult for computers to do this. It requires the programmer to explicitly write out the logic of determining if each number is in the correct place, and then finding the correct place. Even once this is done, there is often not a perfect solution to the problem as you work with larger and larger sets of input data. If you want an algorithm to sort numbers, it often involves various tradeoffs (such as if you want it to work very quickly if the list is already in the correct order, or

if you want it to work at a consistent speed[1]), so computer scientists have developed a number of proven algorithms already[2].

Today, we'll be learning more about them and conducting our own analysis to compare them to one another.

In order to do that, however, we'll need to write a program that can read input and output to files on our computer. Previously, we used redirection to do this, but there's only so much we can do with that.

In today's lab, you will conduct some experiments using four provided data set files. Each file will include a different number of randomly generated numbers. You will write a program that will ask the user which file to read in, read in the appropriate file, and then use different sorting algorithms to sort each file and save it out to a file called sorted.algorithm.size.output. Then, you'll write a one-page summary that discusses the results that you found running each sorting algorithm on each set of numbers and use the Excel template to graph your results.

Sample Output

```
./a.out P4

Welcome the CPSC 1011 sorting simulator.
Which data file would you like to sort?
  [1] Small file (10 items)
  [2] Medium file (100 items)
  [3] Large file (1,000 items)
  [4] Jumbo file (10,000 items)

Enter your choice:  4

Sorting with selection sort...done after 180 ms.
Sorting with insertion sort...done after 200 ms.
Sorting with bubble sort...done (after 600 ms)
Sorting with quick sort... (after 50 ms).

Finishing the program...
```

This week's lab doesn't have a planning document required, but it will still be helpful to discuss the provided sorting algorithms with those around you.

Lab 13 Structure	
Tuesday	Thursday
20 minutes: work on understanding the various sorting algorithms. 30 minutes: Begin coding your solution.	50 minutes: Finish coding your solution and conduct your analysis in your lab report. Submit to Canvas by 11:59PM on Friday.

[1] These various speed traits are referred to as an algorithm's **time complexity**.
[2] You'll learn a lot more in-depth about these sorting algorithms and more in CPSC 2120, the Algorithms and Data Structures course, including how to write them yourself, but for now we'll keep our analysis relatively simple.

Lab Exercise

1. Implement the algorithms and functionality
   - Write a main (and save it in file sorting_algoritms.c) that prompts the user for their choice of data size and opens and reads the data the appropriate file from the local directory.
   - Write the function prototypes and definitions for each of the various sorting algorithms, then use them to sort the input. (HINT: there will be 4 function prototypes and 4 function definitions…and the only difference should be the name of the sort).
     - Create a .h file with function prototypes for the sorting functions.
     - Create a .c file for the function implementations of the sorting algorithms.
     - The code for the algorithms is provided in the starter code on Canvas.
     - The sample input files are provided on Canvas.
     - The starter code for the sorting functions are provided on Canvas.
   - Your program output should look like the sample output above.
   - Add functionality to time how long each algorithm takes (in milliseconds) to sort the input.
     - Hint: functions included in time.h would work well as a starting point for this.
   - Use the provided code to write your sorted output to the appropriate file. This file name should match the pattern sorted.algorithm.size.output.
     - For example, if I just ran the program using the medium size file and am outputting the sorted output for selection sort, my file name should be sorted.selection.medium.output
2. Comment the code for the sorting algorithms to the best of your ability with how the code operates and accomplishes its goal. If you choose to use outside resources to do this, verify with your TAs that your understanding is correct.
3. Once you've run all of the sorting algorithms on all file sizes, plot the results using the Excel template. Clearly label the axes and the data points. Include a brief description of the results you found. Export your graph as a PDF and submit this as lab_report.pdf along with your code.

Be sure to read the submission instructions thoroughly, as there are different requirements for this week.

Working with multiple files and make has great benefit to code clarity and modularity and is something you'll need to be very familiar with in later computer science courses. Don't forget that you need to include all of your .c files in your gcc command to get the system to compile, unless you complete the bonus exercise and create a makefile.

Starter Code

Selection Sort

```
int i = 0; int j = 0; int temp = 0;
int index_smallest = 0;

for (i = 0; i < num i++){
   index_smallest = i;
   for (j = i + 1; j < num; j++){
      if (numberArray[j] < numberArray[index_smallest])
         index_smallest = j;
   }
   temp = numberArray[i];
   numberArray[i] = numberArray[index_smallest];
   numberArray[index_smallest] = temp;
}
```

Insertion Sort

```
int temp = 0; int i = 0; int j = 0;

for (i = 1; i < num; i++){
   temp = numberArray[i];
   j = i - 1;
   while (temp < numberArray[j] && j >= 0){
      numberArray[j+1] = numberArray[j];
      j--;
   }
   numberArray[j+1] = temp;
}
```

Bubble Sort

```
int i = 0; int j = 0; int temp = 0;

for (i = 0; i < num; i++){

   for (int j = 0; j < (num - i - 1); j++){
      if (numberArray[j]> numberArray[j+1]){
         temp = numberArray[j];
         numberArray[j] = numberArray[j+1];
         numberArray[j+1] = temp;
      }
   }
}
```

Quick Sort

```
int pivot = 0;
int i = 0;
int j = 0;
int temp = 0;

if (low < high){
   pivot = low;
   i = low;
   j = high;

   while (i < j){
      while (numberArray[i] <= numberArray[pivot] && i <= high){
         i++;
      }
      while (numberArray[j] > numberArray[pivot] && j >=low){
         j--;
      }
      if (i < j) {
         temp = numberArray[i];
         numberArray[i] = numberArray[j];
         numberArray[j] = temp;
      }
   }
   temp = numberArray[j];
   numberArray[j] = numberArray[pivot];
   numberArray[pivot] = temp;
   quickSort(low, j-1, numberArray);
   quickSort(j+1, high, numberArray);
}
```

Write Integer Array to Output File

```
void writeArrayToFile(int array[], int num) {
  FILE * pFile;

  pFile = fopen("rand" , "w+");
  if (pFile == NULL) perror ("Error opening file");
  else
  {
    int i;
    for(i = 0; i < num; i++) {
      fprintf(pFile, "%d, ", array[i]);
    }
    fclose (pFile);
  }
  return;
}
```

Tar File

This week's lab will use a different method for submitting your files! This week, you will submit your files as a tar archive. Tar is a UNIX command for creating and extracting archives and compressing files. You have probably seen file archives before most commonly as .zip files, which compress other files to save space and keep them bundled together. We'll be using these to keep all of your files in one nice format for submission.

See the demo below for an example on how to create an archive for this week's lab. In it, the -c flag specified we want to create an archive, -z specified we want to use .tar.gz as our format, and -f specifies that we want to name our archive something specific when it's created. If instead we wanted to remove files from an archive (for example, to check our submission to Canvas worked correctly), we would use -x to extract the archive.

```
prompt % ls
sorting_ algoritms.c      lab_report.pdf

prompt % tar -czf hyte.tar.gz sorting_ algoritms.c lab_report.pdf

prompt % ls
sorting_ algoritms.c      lab_report.pdf      hhyte.tar.gz
```

The archive created above (hyte.tar.gz) contains the two files specified (sorting_algorithms.c and lab_report.pdf) in a compressed format. You would now submit hyte.tar.gz to Canvas.

Submission Guidelines

- This week's lab will use a different method for submitting your files! This week, you will submit your files as a tar archive as described above. Submissions for this lab should be archived in the format last_name.tar.gz (where last_name is replaced with **your** last name).
- Your archive should contain:
  - lab_report.pdf
  - graph of your sorting results
  - ALL of your source code
    - sorting_algoritms.c
    - Sorting function files
    - makefile that may have created as part of the optional bonus exercise.
- Your archive should NOT contain:
  - a.out or any other compiled code
  - A folder, with any of your files inside of that folder.
  - ANY of the data files provided with this lab.
  - ANY of the sorted output files you created as part of this lab.
- Submit your last_name.tar.gz archive to Canvas assignment associated for this lab by 11:59PM on Friday (12/6/2019). You should verify within Canvas to make sure your submission was uploaded correctly and in its entirety. You must include the academic honesty header and *explicitly* list the names of TAs that you asked for help outside of lab.

Grading Rubric

- If you are not present and attending lab on Tuesday and Thursday, you will not receive credit for this lab.
- If you do not check in your code with a TA on Thursday, you will not receive credit for this portion of the lab.
- If your assignment is not submitted fully to Canvas by the due date, you will not receive credit for that portion of the lab.
- Your assignment will be graded out of 100 points. The approximate grading distribution is:
    - All files are properly archived according to submission instructions — 10 points
    - sorting_algoritms.c
        - User-interface
            - Properly prompts user, matching sample output — 10 points
            - Times and displays sorting algorithm runtimes in milliseconds — 10 points
        - Functionality requirements
            - Properly calls required functions — 10 points
            - Uses each algorithm correctly to sort user input — 10 points
            - Writes sorted output to appropriate file for each algorithm — 5 points
    - Proper code formatting, commenting, headers using good programming conventions — 5 points
    - lab_report.pdf
        - Correctly plots sorting algorithm runtimes in a clearly readable format — 10 points
        - Differences between algorithm runtimes are visible at all levels — 10 points
        - Includes description of differences observed — 10 points
        - breaking out the functions into .h and .c files — 10 points
    - Optional bonus exercise — up to +10 points
      +10 for creating a WORKING makefile to build your system

Additional Resources

- Pseudocode and details of commonly used sorting algorithms are publicly avaliable on the web.
- The time() command is briefly covered as part of zyBooks chapter 2.19.
- More details on the time functions can be found here: http://www.cplusplus.com/reference/ctime/