Lab 03 Specification – A hands-on exercise to practice variations of elementary sorting algorithms.
50 points

Lab Goals

- Think about how to solve the sorting problem by applying different approaches.
- Analyze sorting algorithms by taking a deeper look at the comparisons and swap procedures.
- Implement different sorting algorithms using a programming language such as Java.

Suggestions for Success

- Take a look at the suggestions for successfully completing the lab assignment, which is available at:
  `https://www.cs.allegheny.edu/sites/amohan/resources/suggestions.pdf`

Learning Assignment

If not done previously, it is strongly recommended to read all of the relevant ”GitHub Guides”, available at the following website:
`https://guides.github.com/`
that explains how to use many of the features that GitHub provides. This reading assignment is useful to understand how to use both GitHub and GitHub Classroom. To do well on this assignment, it is also recommended to do the reading assignment from the section of the course textbook outlined below:

- **Sedgewick chapter 02, 2,1**

Assignment Details

Now that we have discussed some basics of sorting algorithms and analyzed them together in the last few lectures, it is now time to do it practically. In this lab, students will practice developing a variety of algorithms to retain the knowledge from the class discussions so far. This also includes developing one or more code files to implement a series of variations of sorting algorithms using a programming language such as Java.

The lab has two parts. The first part is required to be completed by the end of the lab session. The second part is required to be completed within a one week time period. Please make sure to push your code file(s) and commit by the deadline provided in each of the following section(s).

It is required for all students to follow the honor code. Some important points from the class honor code are outlined below for your reference:

1. Students are not allowed to share code files and/or other implementation details. It is acceptable to have a healthy discussion with your peers. However, this discussion should be limited to sharing ideas only.

2. Submitting a copy of the other's program(s) is strictly not allowed. Please note that all work done during lab sessions will be an opportunity for students to learn, practice, and master the materials taught in this course. By doing the work individually, students maximize the learning and increase the chances to do well in other assessments such as skill tests, exams, etc.

At any duration during and/or after the lab, students are recommended to team up with the Professor and the Technical Leader(s) to clarify if there is any confusion related to the lab and/or class materials.

The Professor proof-read the document more than once, if there is an error in the document, it will be much appreciated if student(s) can communicate that to the Professor. The class will be then informed as soon as possible regarding the error in the document. Additionally, it is highly recommended that students will reach out to the Professor in advance of the lab submission with any questions. Waiting till the last minute will minimize the student chances to get proper assistance from the Professor and the Technical Leader(s).

Section 1: In-Lab Exercise (25 points)

Attendance is in general a required component in computer science class(es) and laboratory session(s). By attending these sessions regularly, students get an opportunity to interact with the Professor and the Technical Leaders(s). This interaction helps maximize the student's learning experience. Attendance is taken in the form of Mastery Quizze(s) and by checking the commit log of the student's In-Lab exercise submission. No student is allowed to take the Mastery Quizze(s) outside the laboratory room unless prior arrangements are done with the Professor. It is against the class honor code to violate the policy outlined above.

Part 01 - Attendance Mastery Quiz (5 points)

The Mastery Quiz is graded based on a Credit/No-credit basis. Students are required to complete the Mastery Quiz only in the laboratory room, which is in **Alden 109** before 3:15 PM. Submissions after 3:15 PM is not acceptable. Students who had completed the Mastery Quiz according to the policy outlined above will automatically receive (5) points towards their lab grade. The link to the Mastery Quiz is provided below:
`https://forms.gle/HRiBWf6AdGfiFhup8`

Part 02 - A Simple Exercise to design and develop a solution to the sorting problem using a Stack data structure. (20 points)

Students are required to complete this part only in the laboratory room, which is in **Alden 109**. It is required to complete the final commit by the end of the laboratory session, which is before **4:20 PM** on the same day that the lab is given to the students. There is late submission accepted for this part, according to the late policy outlined in the course syllabus.

We had seen in detail how to perform and analyze some elementary sorting algorithms such as Insertion and Selection so far, and just got started on the Quick Sort algorithm. In this part, let us solve the sorting problem using a Stack data structure. It is worth noting that all the sorting algorithms discussed in class so far used Arrays as the data structure. As a variation to this approach, let us use the Stack data structure to solve the sorting problem.

Let us suppose that you are given with a list of un-sorted weight(s) of different atheletes stored in a primary Stack data structure. The requirement is to sort the weights stored in the Stack in an ascending order. Assume that we are also provided with access to a temporary Stack data structures. The access to both Stack ADTs is restricted to only the following supporting operations:

1. push
2. pop
3. isEmpty()
4. peek

Start thinking towards understanding this variation of the sorting problem.

1. For simplicity, a starter code with a file named `sorting.tex` is provided in the lab repository. To give an example, the starter code has the original formal insertion sorting algorithm discussed and practiced during class discussions.
2. A latex file may be developed using the Overleaf website link provided below:

   `https://www.overleaf.com/login`
3. Please note: The Overleaf website may prompt to register and log in before providing the options to compile latex files and generate pdf files. This should be a straight forward process. If there are any questions, students are encouraged to ask the Technical Leader(s) and the Professor.
4. An alternative approach is to install latex on your laptop using the web resources below:
   (a) **MAC**: `https://www.latex-project.org/get/`
   (b) **Ubuntu**: `https://milq.github.io/install-latex-ubuntu-debian/`
   (c) **Windows**: `https://www.youtube.com/watch?v=oI8W4MvFo1M`
5. An easier way to do Latex development is through Overleaf. To do the development without the internet, it is best to install latex on your laptop.
6. **ADD** the following statement to all your submission files including the file named `sorting.tex`.

   This work is mine unless otherwise cited - Student Name
7. Make necessary modification(s) to the file named `sorting.tex` to include the steps to formulate your algorithmic solution to the sorting problem.
8. The algorithm should be developed in a formal manner using a similar style as we did in the class examples. Please refer back to the lecture slides and your class notes to look at the examples of sorting algorithms discussed in class.
9. Analyze the algorithm proposed above and add a summary file named `sorting-analysis`. This file can be PDF or markdown format. In this file, analyze the running time of the algorithm and provide an asymptotic upper bound for the running time.

10. Implement the algorithm proposed above using the Java programming language. For simplicity, a starter code-named `Sorting.java` is provided. The input of the program is automatically generated within the starter-code using a randomized sequencing. The output of the program should be generated using the ascending order format.

11. A sample code file named `StackUsage.java` is provided in the repository. The sample code may be used as a tool to learn how to use the built-in Stack ADT in Java. It is worth to make a note that a Stack implementation was done in Lab 2.

12. For more details on the requirements for this program, please look at the comments in the source code file.

13. **ADD** the following statement to all your submission files including the file named `Sorting.java`.

    This work is mine unless otherwise cited - Student Name

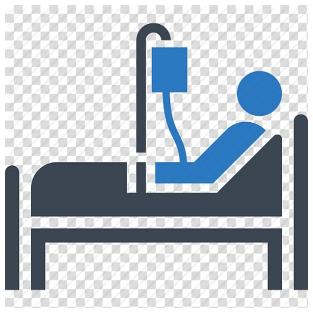
14. It is required as part of this lab submission for students to compile the latex files and generate a PDF version of the file. The PDF file should be named as `sorting.pdf` and uploaded to the repository. Both the **tex** and **pdf** files will be used for grading purposes.

15. Both the algorithm and the java implementation is worth 10 points each.

Section 2: Take Home Exercise (25 points)

Students are recommended to get started with this part in the laboratory room, by discussing ideas and clarifying with the Professor and the Technical Leader(s). All the part(s) in this section is due in **one week** time. It is acceptable to discuss high-level ideas with your peers, while all the work should be done individually. The deadline for submitting the part(s) in this section is **14th Feb 2020, 8:00 AM**. Late submission is accepted for the part(s) in this section, based on the late policy outlined in the course syllabus.

A variation of the Insertion Sort algorithm implementation (15 points)



Let us suppose that the patient details such as patient id's are given and the requirement is to sort the patients in ascending order based on their id's in ascending order.

A variation of the Insertion Sort is the binary insertion sort algorithm. The concept is driven by a simple idea. Insertion sort goes through the elements one by one to identify the correct placement of a given value in the sequence. A binary insertion sort in-contrast will simply divide the problem set into two parts. It is worth noting that we had already solved a divide and conquer problem earlier in the Defective coin problem. This approach is an extension of what was done in Lab 1. This process seems like a $O(n * log(n))$. But in reality, it is not. There is still a need to move the elements to the right in a linear time range.

The YouTube url below demonstrates this variation of insertion sort. Watch this video by clicking on the lin below:

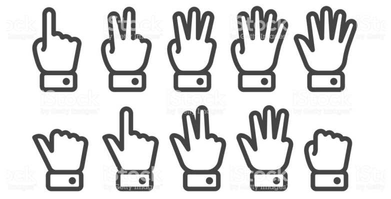
`https://www.youtube.com/watch?v=-OVB5pOZJug`

After watching the video, implement the requirements outlined below:

1. First, think through the solution discussed in the video. In order to challenge students to think through algorithms and formulating implementation-specific details, the step by step solution is purposefully left out from this specification. It is highly recommended to discuss and brainstorm ideas with your peers, the TL(s), and the Professor on how to implement this.
2. Implement the algorithm proposed above using the Java programming language. For simplicity, a starter code-named `BinarySort.java` is provided. The input of the program is automatically generated within the starter-code using a randomized sequencing. The output of the program should be generated using the ascending order format. Make necessary code modifications to this starter-code file. Read through the comments in the code file for additional information on the structure of the code.
3. **ADD** the following statement to all your submission files including the file named `BinarySort.java`.

   This work is mine unless otherwise cited - Student Name

Part 02 - An implementation of the Counting Sort algorithm. (10 points)



Counting sort assumes that we have a known, reasonably small, range of values to be sorted. For simplicity, let's assume that all possible values in the array `a[0]`, `a[1]`, . . . , `a[n-1]` belong to the range $0, 1, 2, \ldots, M-1$ for some "reasonable" constant M that does not depend upon the array size n. ("Reasonable" means, for instance, something that is much smaller than n, or that does not exceed the maximum array size allowed by a particular language or system.)

Create an `int` array named `count` of size M, initialized to all zeros. Make one pass through the array `a` and, for each element `a[i]`, add 1 to the counter for the value `a[i]`. Then run through the values in the `count` array and, for each i from 0 through $M-1$, place `count[i]` copies of i into the array `a`. Here's a simple example with $n = 15$ and $M = 5$:

BEFORE:

`a` =

4	3	0	1	2	4	0	2	1	3	4	1	0	2	2

, `count` =

0	0	0	0	0

AFTER PASS THROUGH `a`:

`a` =

4	3	0	1	2	4	0	2	1	3	4	1	0	2	2

, `count` =

3	3	4	2	3

AFTER PASS THROUGH `count`:

`a` =

0	0	0	1	1	1	2	2	2	2	3	3	4	4	4

, `count` =

3	3	4	2	3

The running time here is $O(n)$, since we must examine each element of `a` just once. (We must also examine each element of `count`, but since its size M is a constant independent of n, this just adds a constant amount of time.)

Obviously, this method is useless when we don't know in advance what the range of values will be, or when the range is not easy to enumerate (think about sorting real numbers like 3.1540023, -0.0000003021, and 38827233.3772), or when the range is very large (we wouldn't want to create a `count` array of size 2^{32}, for instance, but there are 2^{32} possible values in a Java `int` variable).

1. First, understand thoroughly the algorithmic solution provided above. The description provided above may seem confusing. Please read the description a few times and try to work on a few examples using the process outlined. Discuss with your peers, the TL's and the Professor if the solution is not fully understood.
2. Implement the algorithm proposed above using the Java programming language. For simplicity, a starter code-named `CountingSort.java` is provided. The input of the program is automatically generated within the starter-code using a randomized sequencing. The output of the program should be generated using the ascending order format. Make necessary code modifications to this starter-code file. Read through the comments in the code file for additional information on the structure of the code.
3. **ADD** the following statement to all your submission files including the file named `CountingSort.java`.

   This work is mine unless otherwise cited - Student Name
4. The implementation is left out as an open-ended problem in this specification. This is simply because to give the student an opportunity to think through and come up with their own implementation ideas.

Submission Details

For this assignment, please submit the following to your GitHub repository by using the link shared to you by the Professor:

1. Commented source code from the "sorting.tex" program.
2. Commented "sorting.pdf" file.
3. Commented source code from the "Sorting.java" program.
4. Commented source code from the "BinarySort.java" program.
5. Commented source code from the "CountingSort.java" program.
6. A document containing the summary "sorting-analysis" file.
7. It is highly important, for you to meet the honor code standards provided by the college and to ensure that the submission is made before the deadline. The honor code policy can be accessed through the course syllabus. Make sure to add the statement "This work is mine unless otherwise cited." in all your deliverables such as source code and PDF files. In your summary file, please make sure to include your name and your partner's name.

Grading Rubric

1. There will be full points awarded for the lab if all the requirements in the lab specification are correctly implemented. Partial credits may be awarded if deemed appropriate.

2. Failure to upload the lab assignment code to your GitHub repository will lead to receiving no points given for the lab submission. In this case, there is no solid base to grade the work.

3. There will be no partial credit awarded if your code doesn't compile correctly. It is highly recommended to validate if the correct version of the code is being submitted before the due date and make sure to follow the honor code policy described in the syllabus. If it is a late submission, then it is the student's responsibility to let the professor know about it after the final submission in GitHub. In this way, an updated version of the student's submission will be used for grading. If the student did not communicate about the late submission, then automatically, the most updated version before the submission deadline will be used for grading purposes. If the student had not submitted any code, then, in this case, there are no points awarded to the student.

4. If a student needs any clarification on their lab grade, it is strongly recommended to talk to the Professor. The lab grade may be changed if deemed appropriate.