Lab 04 Specification – A hands-on exercise to practice representation and implementation of advanced sorting algorithms.
50 points

Lab Goals

- Think about how to solve the sorting problem by applying different approaches.
- Analyze quick sort and merge sort algorithms by developing a different representation of the solution,
- Implement another variation of the sorting algorithm 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, 2,2, 2.3**

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 a different representation of Quick and Merge sort and implement another variation of sorting algorithm 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/U6gjq9zPwVF8nCyF8`

Students are required to complete both part 02 and 03 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.

Part 02 - A Simple Exercise to design and develop a solution to the Quick sort algorithm. (10 points)

We had seen in detail how to perform and analyze the Quick sort algorithm and to develop a recursive tree to solve this classic sorting algorithm. The Quick Sort algorithm has two parts namely:

1. the main algorithm
2. the partition procedure

The key learning objectives are summarized below:

1. Filling out the swapping traces correctly indicates a leaners algorithmic understanding level on the partition procedure.
2. Filling out the levels, QS calls, and the q value [key player in the game] correctly indicate a learner's understanding level on how the QS main algorithm work, how the recursive call works, and most importantly the process of building the recursive tree itself.

It is recommended to build the recursive tree first as we did in all our in-class examples, in order to generate the table effectively.

An example of applying the Quick Sort algorithm on the input un-sorted array [9,6,5,0,8,2,4,7] is shown in next page:

Level	QS calls	q	A* [swapping trace]
1	**Root:** QS(0,7)	5	A[0];A[1] A[1]; A[2] A[2]; A[3] A[3]; A[5] A[4]; A[6] A[5]; A[7]
2	QS(0,4) QS(6,7)	2 7	A[0]; A[2] A[1]; A[3] A[2]; A[4] No swaps
3	QS(0,1) QS(3,4) QS(6,6) QS(8,7)	1 4 X X	No swaps
4	QS(0,0) QS(2,1) QS(3,3) QS(5,4)	X X X X	No swaps

Start developing the solution to this part by including the details outlined below:

1. For simplicity, a starter document with a file named `quick-template` is provided. To give an example, the starter document has the sample solution in a google doc format. Copy the document to your google drive and start making edits to the table.

   `https://docs.google.com/document/d/1gz2JMEV71b_Yso50v7HtbDG8TJW5jYC2FgBS8qR9dJ0/edit?usp=sharing`

2. Let us suppose that an input un-sorted array [8,1,6,5,10,7,4,2,9,3] is provided.

3. Start developing the recursive tree to sort the array provided in the previous step based on your understanding of class discussions. Refer to the lecture 06 slides and your notes.

4. Develop the tabular solution using the template provided. Take a screenshot (clear) and name the file as `quick-solution`.

5. **ADD** the following statement to all your submission files including the file named `quick-solution`. Add the honor code on the google doc before taking the screenshot.

   This work is mine unless otherwise cited - Student Name

Part 02 - Answer the following question to record your understanding of the basic ideology of Quick Sort. (10 points)

1. **ADD** the following statement to all your submission files including the file named `quick-analysis`. The file can be developed using a mark down file and/or PDF file.

   This work is mine unless otherwise cited - Student Name

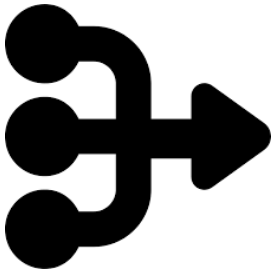
2. Let us suppose that there are n elements in the un-sorted array. Answer the following?

   - q1: What is the worst, best, and average case running time of the Quick Sort algorithm.
   - q2: What is the base case for Quick Sort to break?
   - q3: What is the recursive case for Quick Sort?
   - q4: How many recursive calls are made at each step in QS main algorithm?
   - q5: Does QS partition always create equal splits?

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 **21st 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.

Part 01 - A Simple Exercise to design and develop a solution to the Merge sort algorithm. (10 points)



We had seen in detail how to perform and analyze the Merge sort algorithm and to develop a recursive tree to solve this classic sorting algorithm. The Merge Sort algorithm has two parts namely:

1. the main algorithm
2. the merge procedure

The key learning objectives are summarized below:

1. Filling out the sub-array traces correctly indicates a learner's algorithmic understanding level on the merge procedure.
2. Filling out the levels, MS calls, and the m value [key player in the game] correctly indicate a learner's understanding level on how the MS main algorithm work, how the recursive call works, and most importantly the process of building the recursive tree itself.

It is recommended to build the recursive tree first as we did in all our in-class examples, in order to generate the table effectively.

An example of applying the Merge Sort algorithm on the input un-sorted array [5,2,4,1,0,3] is shown below:

Level	MS calls	m	**Sub Array* [traces]**
1	**Root:** MS(0,5)	2	[0,1,2,3,4,5]
2	MS(0,2) MS(3,5)	1 4	[0,1,2,3,4,5]
3	MS(0,1) MS(2,2) MS(3,4) MS(5,5)	0 X 3 X	[2,4,5] [0,1,3]
4	MS(0,0) MS(1,1) MS(3,3) MS(4,4)	X X X X	[2.5] [0,1]

Start developing the solution to this part by including the details outlined below:

1. For simplicity, a starter document with a file named `merge-template` is provided. To give an example, the starter document has the sample solution in a google doc format. Copy the document to your google drive and start making edits to the table.

   `https://docs.google.com/document/d/1qzV1TvUN9ZgqJfsS8kioRUMD_UtzSXAuyqRRuTosNls/edit?usp=sharing`

2. Let us suppose that an input un-sorted array [8,1,6,5,10,7,4,2,9,3] is provided.

3. Start developing the recursive tree to sort the array provided in the previous step based on your understanding of class discussions. Refer to the lecture 07 and 08 slides and your notes.

4. Develop the tabular solution using the template provided. Take a screenshot (clear) and name the file as `merge-solution`.

5. **ADD** the following statement to all your submission files including the file named `merge-solution`. Add the honor code on the google doc before taking the screenshot.

   This work is mine unless otherwise cited - Student Name

Part 02 - Implement a variation of the Quick Sort algorithm. (10 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 Quick Sort is the Randomized Quick Sort. A huge performance metric of the Quick Sort algorithm is the pivot selection. If the pivot is fixed to the last element as in the original Quick Sort, then this may lead to a **0 : n-1** split, which then leads to a $O(n^2)$ run time in the worst case. A randomized quick sort may take this equation out of the picture. The general idea is to simply randomize the pivot selection. The original Quick Sort and Randomized Quick Sort procedures are provided on the next page for easy access.

Algorithm - Partition(A, p, r)

Input: an n-element un-sorted array A of integer values, a lower bound p of the array A, and a pivot r in the array A.

Output: an n-element sorted array A of integer values.

```
x ← A[r]
i ← p − 1
for j = p to r-1 do
   if A[j] ≤ x then
      i ← i + 1
      swap A[i] and A[j]
   end if
end for
swap A[i+1] and A[r]
return i+1
```

Algorithm - QuickSort(A, p, r)

Input: an n-element un-sorted array A of integer values, a lower bound p of the array A, and a pivot r in the array A.

Output: an n-element sorted array A of integer values.

```
if p < r then
   q ← Partition(A,p,r)
   QuickSort(A, p, q-1)
   QuickSort(A, q+1, r)
end if
```

Characteristics of Randomized Quick Sort:

- Assume all elements are distinct.
- Partition around a random element.
- Consequently, all splits (1: n-1, 2: n-2, ... , n-1: n)
- Randomization is a general tool to improve algorithms with bad worst-case but good average case complexity.

Algorithm - Randomized Partition(A, p, r)

Input: an n-element un-sorted array A of integer values, a lower bound p of the array A, and a pivot r in the array A.

Output: an n-element sorted array A of integer values.

```
i ← Random(p, r)
swap A[r] and A[i]
return Partition(A, p, r)
```

Note: Partition algorithm is provided above.

Algorithm - Randomized QuickSort(A, p, r)

Input: an n-element un-sorted array A of integer values, a lower bound p of the array A, and a pivot r in the array A.

Output: an n-element sorted array A of integer values.

if p $<$ r **then**
  $q \leftarrow$ Randomized-Partition(A,p,r)
  Randomized-QuickSort(A, p, q-1)
  Randomized-QuickSort(A, q+1, r)
end if

Expected Running Time:

- Randomizing pivot selection
- Splits are going to be better than 0: n-1
- Take average for running time using different pivot selection.
  $O(n \times log(n))$

After thinking through the pseudocode provided above, implement the requirements outlined below:

1. First, read through the high-level outline of the randomized partition and randomized quick sort algorithm provided above. The challenge here is to think through a given algorithm and traslate it to implementation. A huge learning objective here is to find out the limitations of Quick Sort through practical activity. 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 `RQS.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 `RQS.java`.

   This work is mine unless otherwise cited - Student Name

Part 03 - Answer the following question to record your understanding of the basic ideology of Merge Sort. (5 points)

1. **ADD** the following statement to all your submission files including the file named `merge-analysis`. The file can be developed using a mark down file and/or PDF file.

   This work is mine unless otherwise cited - Student Name

2. Let us suppose that there are n elements in the un-sorted array. Answer the following?

   - q1: How is merge sort different from quick sort? q2: What is the split ratio in merge sort? q3: What is the worst-case/average-case/best-case running time of Merge Sort? q4: Why is the worst case running time of Merge sort O(n log n) always? q5: Why does Merge Sort use a static tree in the recursion process? It is worth noting that the Quick Sort use a dynamic tree.

Submission Details

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

1. “quick-analysis” document file.
2. “quick-solution” image file.
3. “merge-solution” image file.
4. “merge-analysis” image file.
5. Commented source code from the “RQS.java” program.
6. 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.