A special office hour for this assignment will be held on Friday Nov 13, 2020, 4:00pm-6:00pm in Zoom (Meeting ID: 939 4775 0254 Passcode: 522257)
Feel free to contact me at lab192@case.edu with any questions.

Written Section (50 points)

Solutions to this section should be submitted as a single PDF file.

Problem 1: InsertionSort on a linked list **(14 pts total)**

a) Write pseudocode for InsertionSort on a singly linked list. Your code should use constant space and its worst-case runtime complexity should be no worse than O(n^2). **(8 pts)**
b) What is the best-case runtime of your code? What type of input guarantees this runtime? **(3 pts)**
c) What is the worst-case runtime of your code? What type of input guarantees this runtime? **(3 pts)**

Problem 2: BucketSort **(12 pts total)**

Consider the following pseudocode[1] for BucketSort. You may assume that the hash function enforces the property that all the values in a bucket are greater than all values in preceding buckets.

```
BUCKET-SORT($A$)
1  $n = A.length$
2  let $B[0..n-1]$ be an empty array
3  for $i = 0$ to $n-1$
4      make $B[i]$ an empty list
5  for $i = 0$ to $n-1$
6      insert $A[i]$ into list $B[hash(A[i])]$
7      // Hashing function determines which bucket element goes into
8  for $i = 0$ to $n-1$
9      sort list $B[i]$ with insertion sort
10 concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order
```

a) What is the best-case runtime of this algorithm on an array of size n? Explain your answer. **(4 pts)**
b) What is the worst case runtime of this algorithm on an array of size n? When does this case occur? **(4 pts)**
c) Replacing InsertionSort with a more efficient sort would result in a better worst-case. Explain why it wouldn't make sense to do this. **(4 pts)**

[1] Adapted from Cormen et al. Introduction to Algorithms 3ed.

Problem 3: MergeSort **(12 pts total)**

a) Consider a modified version of MergeSort that uses InsertionSort on the subarrays once they are small enough. Specifically, once their size is $\leq k$. In terms of Big-O notation, what is the run-time of this sorting algorithm? Justify your answer. (*hint*: consider how many times insertion-sort will run) **(8 pts)**

b) Give rationale for such an implementation **(4 pts)**

Problem 4: Interesting Inputs **(12 pts total)**

Characteristics of data will be specified below. Please state which of *(InsertionSort, MergeSort)* has a better worst-case complexity for large inputs. Justify your choice and briefly explain the worst-case Big-O complexity of your choice on the data. Don't forget that MergeSort's complexity is independent of input characteristics, and make your decisions based on the version of InsertionSort given in class (Lecture 17).

a) A sorted array of integers **(3 pts)**

b) A completely random array of integers **(3 pts)**

c) An array of integers with the property that the ith smallest integer comes before index $i+30$ **(3 pts)**

d) An array of zeros and ones **(3 pts)**

(OPTIONAL) Extra Problem : SaladSort **(5 bonus pts)**

Consider the following sorting algorithm:

Step 1: Check if the array is sorted. If so, we are done.

Step 2: Randomly rearrange the elements *as though* you were mixing a salad in a salad bowl.

Step 3: Return to step 1

a) What is the best-case runtime of SaladSort? **(1 pt)**

b) You are given an array of 4 unique elements. You choose to sort it using SaladSort. How many times do you expect to perform step 2? Justify your answer. **(2 pts)**

c) Let k be a positive constant. Consider an array of n elements where k of those n elements are unique and the rest are 0. Assume that steps 1-3 *magically* run in constant time. In terms of n and k, what is the Big-O runtime complexity of SaladSort? Justify your answer with math, but there is no need to construct an airtight proof. **(2 pts)** (*Note:* You may assume we are working with arrays such that n>k)

Programming Section (50 points)

There are many different sorting algorithms, each with their pros and cons. In this assignment, you'll be implementing MergeSort, QuickSort, and InsertionSort. You'll also be comparing these various algorithms. Your programming project should be submitted as a single zip file. Optionally, you can include a readme.

Create a class named **Sort.java** including the following methods:

- **void MergeSort(int[] A)**: Takes an input array A and sorts it according to the MergeSort algorithm
- **void QuickSort(int[] A)**: Takes an input array A and sorts it according to the QuickSort algorithm
- **void InsertionSort(int[] A)**: Takes an input array A and sorts it according to the InsertionSort algorithm
- **int[] RandomArray(int n, int a, int b)**: Generates a random array of n integers in the interval [a,b] and returns it. This will be used for comparing the sorting algorithms.

There are no explicitly required methods other than these. That being said, feel free to use any helper methods and classes.

You will need to time your sorting algorithms across varying input size and create a data table. Perform time comparisons of the algorithms for random arrays of input sizes

- n = 5, 10, 15, 20, 1000, 2000, 4000, and 8000

where n is the number of elements. Feel free to perform tests on more n-values. It will suffice to test on random **arrays of integers in the interval [-10000,10000]**. In your submitted zip file, include your data table as well as **brief** answers to the following questions:

1. Using your data, compare MergeSort and QuickSort. Which is faster for large n? **(2 pts)**
2. Is your answer to the previous question consistent with theory? **(2 pts)**
3. Consider your data for MergeSort. Looking at large n, are your results consistent with the theoretical runtime complexity? **(2 pts)**
4. Answer the previous question for QuickSort. **(2 pts)**
5. If ever, when is InsertionSort preferable? **(2 pts)**

***Time comparison tips & details:**

- Record all of your data on the same computer. Your data will probably not make any sense if you don't do this. In general, try to standardize your testing as much as possible.
- If InsertionSort is taking too long to run past a certain n-value, you may omit these data points (just for insertion sort). If everything is taking too long to run, contact me (lab192@case.edu).
- I would recommend storing your data table in a google sheet.
- Averaging the times across multiple trials is a *very* good idea.
- Don't fudge your data. There's no incentive to -- your written responses will be graded in the context of your results.

Programming Section Rubric:

- MergeSort implementation **(10 pts)**
- QuickSort implementation **(10 pts)**
- InsertionSort implementation **(6 pts)**
- RandomArray Implementation **(4 pts)**
- Style
  - Comments should sufficiently explain your code**(2 pts)**
  - Variable names should convey meaning **(1 pts)**
  - The code should be well-designed. Are there lines that never execute? Inelegant constructions? **(2 pts)**
- Data table **(5 pts)**
- Written analysis questions **(10pts total)**

*** Your sorting algorithms will be tested (by me) for validity. Half of the awarded points will come from testing. Make sure that your sorting algorithms are valid for and don't crash on any input.**