CSC 505, Homework 2
Due date: Friday, September 24th, 9:00 PM

Homework should be submitted via Moodle in printed form. To avoid reduced marks, please do **NOT submit scanned writing or drawings**. All assignments are due on 9 PM of the due date. Late homework will be accepted only in circumstances that are grounds for excused absence under university policy. The university provides mechanisms for documenting such reasons (severe illness, death in the family, etc.). Arrangements for turning in late homework must be made by the day preceding the due date if possible.

All assignments for this course are intended to be individual work. Turning in an assignment which is not your own work is cheating. The Internet is not an allowed resource! Copying of text, code or other content from the Internet (or other sources) is plagiarism. Any tool/resource must be approved in advance by the instructor and identified and acknowledged clearly in any work turned in, anything else is plagiarism.

If an academic integrity violation occurs, the offending student(s) will be assessed a penalty that is at least as severe as getting a 0 for the whole homework for which the violation occurred, and the case will be reported to the Office of Student Conduct.

Instructions about how to "give/describe" an algorithm (taken from Erik Demaine): Try to be **concise, correct, and complete. To avoid deductions,** you should provide (1) a textual description of the algorithm, and, if helpful, flow charts and pseudocode; (2) at least one worked example or diagram to illustrate how your algorithm works; (3) a proof (or other indication) of the correctness of the algorithm; and (4) an analysis of the time complexity (and, if relevant, the space complexity) of the algorithm. **Remember that, above all else, your goal is to communicate.** If a grader cannot understand your solution, they cannot give you appropriate credit for it.

Here (and elsewhere), the function lg indicates the binary logarithm.

Problem 1. (12 points, 4 points for each algorithm). *Purpose: Implementing algorithms.* Implement insertion sort, merge sort, and heapsort, and count the number of comparisons they perform. Follow the description in our textbook in the following sections: Section 2.1, Page 18 for insertion sort; Section 2.3, Pages 31-34 for merge sort; and Sections 6.2-6.4, Pages 154-160 for heapsort. Pay attention to ties and special case considerations. Please only count comparisons between the input values, not between index variables. i) In insertion sort, only count the number of comparisons between elements in the array *A* (the second compare on line 5 page 18), not the compares that check if the index variable *i* is larger than zero; ii) in merge sort, count the number of comparisons on line 13 page 31; iii) in heapsort, count the number of comparisons between the elements of array *A* (the second compare on line 3 and line 6 on page 154), not the compares between variables *l, r*, *A.heapsize*, and *largest*.

Use the code frameworks Sorting.py or Sorting.java in the file Sorting_framework.zip. You don't need to print or return the results, but please make sure that they are stored in the following two instance variables: *sorting_array* (stores the sorted input)

and *comparison_count* (stores the number of comparisons performed) in Python, or *sortingArray* and *comparisonCount* in Java.

You can create additional methods if required but do not change the name of existing methods and any existing code - points will be cut if you do. You can code this in either Java or Python. Another file SortingTest.py/java contains test cases to check your code for various inputs**. Submit the file Sorting.py/java (not the test file!) via Moodle.** Your code will be checked on the remote-linux server by us automatically using the provided cases in SortingTest.py/java, plus some (unknown) cases. To avoid loss of marks please make sure that all provided test cases pass on the remote-linux server by using the test file. Instructions for the remote-linux server setup and test given in the document "HW2_Programming_Assignment_Setup.pdf".

2. *Purpose: Practice solving recurrences.* For each of the following recurrences, use the Master Theorem to derive asymptotic bounds for T(n), or argue why the Master Theorem does not apply. You don't have to solve the recurrence if the MT does not apply. If not explicitly stated, please assume that small instances need constant time c. Justify your answers, ie. give the values of a, b, n^log_b(a) f, ε, for case 3 of the Master Theorem also show that the regularity condition is satisfied. (3 points each)

(a) $T(n)=4T(n/2)+n^{2.1}$.

(b) $T(n)=3T(n/3) + 4T(n/3) + n^2$.

(c) $T(n) = 3T(n/2)+ n^{7/3}$.

3. *Purpose: Often, recursive function calls use up precious stack space and might lead to stack overflow errors. Tail call optimization is one method to avoid this problem by replacing certain recursive calls with an iterative control structure. Learn how this technique can be applied to QUICKSORT.* (2 points each) Solve Problem 7-4, a-c on page 188 of our textbook.

4. (9 points) *Purpose: Practice algorithm design and the use of data structures. This problem was an interview question! To avoid deductions, please follow Eric Demaine's instructions about how to "give/describe" an algorithm.* Consider a situation where your data is almost sorted—for example you are receiving time-stamped stock quotes and earlier quotes may arrive after later quotes because of differences in server loads and network traffic routes. Focus only on the time-stamps. Assume that each time-stamp is an integer, all time-stamps are different, and for any two time-stamps, the earlier time-stamp corresponds to a smaller integer than the later time-stamp. The time-stamps arrive in a stream that is too large to be kept in memory completely. The time-stamps in the stream are not in their correct order, but you know that every time-stamp (integer) in the stream is at most hundred positions away from its correctly sorted position. Design an algorithm that outputs the time-stamps in the correct order and uses only a **constant amount of storage**, i.e., the memory used should be independent of the number of time-stamps processed. Solve the problem using a heap.