CSC 505, Homework 2

Due date: Monday, September 30

Please submit homework **in printed form as a pdf** via Moodle. All assignments are due at 9 PM on 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.

All assignments for this course are intended to be individual work. Turning in an assignment that is not your work is cheating. The Internet is not an allowed resource! Copying text, code, or other content from the Internet (or other sources) is plagiarism. Any tool/resource must be approved 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 at least as severe as getting a 0 for the whole homework assignment for which the violation occurred. The case will be reported to the Office of Student Conduct.

General instructions: Please justify all answers. If not explicitly asked, answers consisting of a single number, equation, or time/space complexity will result in reduced marks.

Instructions about describing an algorithm (taken from Erik Demaine): Be concise, correct, and complete**. To avoid deductions,** you 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) argue the correctness of the algorithm; and (4) an analysis of the time complexity (and, if relevant, the space complexity) of the algorithm. Your overall goal is to communicate. If a grader cannot understand your solution, they cannot give you appropriate credit for it.

In this assignment, the function lg indicates the binary logarithm, ln indicates the natural logarithm.

Problem 1 *(12 points). Purpose: Implementing algorithms.* Implement insertion sort, merge sort, and heap sort, and count the number of comparisons they perform. Follow the description in our textbook in the following sections: Section 2.1, Page 19 (INSERTION-SORT) for insertion sort; Section 2.3, Pages 36 (MERGE) - 39 (MERGE-SORT) for merge sort; and Sections 6.2-6.4, Pages 165 (MAX-HEAPIFY) - 170 (HEAPSORT) 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 19), 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 36; iii) in heapsort, count the number of comparisons between the elements of array *A* (the second compares on line 3 and line 6 on page 165), not the compares between variables *l, r*, *A.heapsize*, and *largest*. (12 points, 4 points for each algorithm).

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 make sure 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 existing methods' names, and any existing code - points will be cut if you do. You can code this in either Java or Python. The 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.** We will check your code on the remote Linux server using the cases provided in SortingTest.py/java, plus some (unknown) cases. To avoid loss of marks, please ensure that all provided test cases pass on the remote-linux server using the test file. Instructions for the remote-linux server setup and test are in the document "HW2_Programming_Assignment_Setup.pdf".

Problem 2 (12 points). *Purpose: Practice solving recurrences.* For a-c, 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. Please assume that small instances need constant time c if not explicitly stated. Justify your answers, i.e., give the values of a, b, n^log_b(a), ε for case 3 of the Master Theorem also show that the regularity condition is satisfied. (3 points each)

(a) $T(n)=5T(n/5) + n/\lg(n)$.

(b) $T(n)=16T(n/2) + \text{sqrt}(n^9)$.

(c) $T(n)=T(n/27) + 2T(n/27) + n^{1/4} \lg(n)$.

(d) $T(n) =8T(n/4)+ sqrt(n^3\lg(n)^4)+n+1$.

Problem 3 (10 points). Recursive algorithms. Consider the following recursive algorithm F(n). You may assume n>0.

```
Alg: F(n)
{
        IF n=1 THEN
                RETURN 1;
        ELSE
                RETURN 2*n * F(n-1)
}
```

a) (2 points) Please compute F(i), i=1,…,4

b) (2 points) What function value does F(n) compute?

c) (3 points) Choose the multiplication '*' as the only basic operation. Provide a recurrence for the **running time** T(n) of algorithm F(n). Give the exact solution and a tight asymptotic bound for the worst-case running time of F.

d) (5 points) How do the answers to b) and c) change if, in the algorithm F(n) above, you replace the line "RETURN 2*n*F(n-1)" with the line "RETURN F(n-1)*F(n-1)"?

Problem 4 (16 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 timestamp 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. The timestamps in the stream are not in their correct order, but you know that every time stamp (integer) in the stream is at most a 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. In the worst-case, your algorithm should report the first n elements in the input stream in $\Theta(n)$. a) Solve the problem using a heap. b) Simplify the solution described in a). Memory consumption and asymptotic performance should stay the same. You should continue using an array, but **you are not allowed to use a heap**, i.e., your solution cannot use Buildheap and Heapify. Describe the modified algorithm and argue why memory and asymptotic performance remain the same.