COSC 311: ALGORITHMS
HW1: SORTING AND ASYMPTOTIC ANALYSIS

Due Friday, September 20 in class

1 Submission and Collaboration

Type up your responses to the questions below. I recommend using LaTeX, which is a typesetting language that makes it easy to make math look good. If you're not already familiar with it, I encourage you to practice! You are also welcome to use your favorite word processor, provided that all equations and formulas are readable.

This assignment is due on Friday, September 20 at the start of class; bring a hard copy of your typed responses to submit in class. **Each problem must be on a separate sheet of paper, with your name and section number at the top of the page.** If you use more than one sheet of paper for an individual problem, please staple together those pages. There is no need for you to rewrite the problems on your submission.

In accordance with this course's intellectual responsibility policy, you are welcome to discuss the problems with other students who are currently taking the course, but you must write up your solutions individually. Please note at the top of your submission with whom you discussed each problem.

2 Problems

Problem 1: Asymptotic analysis.
(a) Suppose you know that $f(n) \in O(n^2)$ (and you don't know anything else about f). Are each of the following claims **true** for all possible choices of f, **false** for all possible choices of f, or **sometimes true** (i.e., true for some specific choices of f and false for others)? Briefly explain your reasoning. You do not need to give any formal proofs.

(i) $f(n) \in \Omega(n)$

(ii) $f(n) \in \Theta(n^2)$

(iii) $f(n) \in O(n^3)$

(iv) $f(n) \in O(n \lg n)$

(v) $f(n) \in \Theta(n^2 \lg n)$

(vi) $f(n) \in \Omega(n^4)$

(vii) $f(n) \in O(14n^2 + 74)$

(b) Let $f(n) = 3n^2 + n$. Prove that $f(n) \in \Theta(n^2)$.

(c) Let $f(n) = 100n - 7$. Is $f(n) \in O(n \lg n)$? Is $f(n) \in \Omega(n \lg n)$? Prove or disprove each.

Problem 2: Analyzing heapsort. Here is the heapsort algorithm that we wrote together in class:

```
1 Heapsort(A) // A is an unsorted array of length n
2    A.heapify() // turn A into a maxheap
3    for i = A.length-1 to 0
4       swap(A, 0, i)
5       A.siftDown(0)
```

Your job in this problem is to analyze the asymptotic runtime of heapsort, using the steps below to help you. You may find that you need to review the asymptotic analysis of heaps!

(a) In class we wrote the following iterative `heapify` method:

```
10 A.heapify()
11    for i = (n-1)/2 to 0
12       A.siftDown(i)
```

We'll first analyze the runtime of `heapify`. Let j denote the depth of a node in the heap, where the root is at depth 0.

(i) If a node in position i in the heap array is at depth j, at most how many swaps can be performed during the call to `siftDown` on line 12 (in terms of j)?

(ii) Over the course of the for loop in line 11, how many calls to `siftDown` are made from depth j?

(iii) If the heap has n elements, what values can j take on (you can assume that n is one less than a power of 2)?

(iv) Put (i)-(iii) together to write a summation that captures the total number of swaps performed during a call to `heapify`.

(v) Find a closed-form expression (i.e., an expression that does not contain a summation) that gives an upper bound on the number of swaps performed during `heapify`. (Hint: you may find it useful to know that $\sum_{j=0}^{\infty} \frac{j}{2^j} = 2$.)

(b) Based on your answer to part (a) and any other analysis you need, what is the asymptotic (big-O) runtime of heapsort? Give your answer in the simplest form possible, i.e., omitting constant factors and lower-order terms. Briefly explain your reasoning (you might find it helpful to refer to the line numbers in the pseudocode above).

Problem 3: What do we learn from asymptotic analysis? In this problem you will explore the value and limitations of asymptotic analysis by experimenting with three sorting algorithms: insertion sort, mergesort, and a new algorithm called bubble sort.

```
BubbleSort(A) // A is an unsorted array of length n
   for i = 0 to n-1
      for j = 1 to n - 1 - i
         if A[j-1] > A[j]
            swap(A, j-1, j)
```

(a) Explain in a short paragraph what bubble sort is doing (i.e., what is the sequence of steps it follows?) and why it works. Your goal is to help another CS undergrad understand what bubble sort does and convince them that this algorithm will in fact result in a sorted array.

(b) What is the asymptotic runtime of bubble sort? Justify your answer in a sentence or so.

(c) Based on your answer to (b), how much wall-clock time do you expect bubble sort to take relative to mergesort? What about insertion sort? ("Wall-clock" time is the amount of time that passes between when you start running your program and when it finishes. This is not the same as the number of operations the computer does while running your code, it instead has to do with the user experience of waiting for your code to finish running.)

(d) Download my `Sort.java` code from:
https://kgardner.people.amherst.edu/courses/f19/cosc311/hw/hw1/Sort.java.
The file contains implementations of three sorting algorithms: insertion sort, mergesort, and bubble sort. Your job is to run an experiment to compare their performance. Specifically, do the following:

- Fill an array of length n with randomly generated ints between 0 and 10^6.
- Sort the array using each of insertion sort, mergesort, and bubble sort. For each algorithm, time how long the algorithm takes by recording the system time before and after running the sorting method. You can do this as follows:

```
long start_time = System.nanoTime();
// whatever code you want to time
long end_time = System.nanoTime();
long time_my_code_took = end_time - start_time;
```

Repeat the above for each of the three sorting algorithms for many values of n. You will want to try some small values of n (less than 1000), some large values of n (greater than 100000), and some in between. Be sure to re-fill your array with a new set of random numbers in between each sorting algorithm—otherwise you will be sorting data that is already in sorted order, which might lead to inaccurate results.

Graph your results: make one graph with all three algorithms, with n on the x-axis and time on the y-axis (you may find that it is easier to see what's going on if you also make separate graphs for small n and large n).

Discuss your findings. Do your experimental results match up to your predictions in part (b)? Is the asymptotic analysis equally predictive at small n and large n? Are there things you learn about the relative performance of the three algorithms from the experimental results that you don't learn from the asymptotic analysis? Any other interesting observations?

Note: you **do not** need to submit your code for this problem.