COSC 311: Algorithms
HW1: Sorting and Asymptotic Analysis
Solutions

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 choices of f and false for others)? Briefly explain your reasoning. You do not need to give any formal proofs.

(i) $f(n) \in \Omega(n)$
Solution: Sometimes true. For example, the function $f(n) = n$ is in $O(n^2)$ and $\Omega(n)$, but the function $f(n) = 3$ is in $O(n^2)$ but not $\Omega(n)$.

(ii) $f(n) \in \Theta(n^2)$
Solution: Sometimes true. For example, the function $f(n) = 3n^2$ is in $\Theta(n^2)$, but the function $f(n) = 97n$ is not.

(iii) $f(n) \in O(n^3)$
Solution: True. The function n^2 is asymptotically upper bounded by the function n^3, so any function f that is in $O(n^2)$ must also be in $O(n^3)$.

(iv) $f(n) \in O(n \lg n)$
Solution: Sometimes true. For example, the function $f(n) = n$ is in $O(n^2)$ and $O(n \ln n)$, but the function $f(n) = 4n^2$ is in $O(n^2)$ but not $O(n \lg n)$.

(v) $f(n) \in \Theta(n^2 \lg n)$
Solution: False. For a function to be in $\Theta(n^2 \lg n)$ it must be, in particular, lower bounded by $n^2 \lg n$. But for a function to be in $O(n^2)$ it must be upper bounded by n^2. It is not possible to satisfy these two properties simultaneously.

(vi) $f(n) \in \Omega(n^4)$
Solution: False. For a function to be in $\Omega(n^4)$ it must be lower bounded by n^4, which is not possible if it is upper bounded by n^2.

(vii) $f(n) \in O(14n^2 + 74)$
Solution: True. The function $14n^2 + 74$ is asymptotically equivalent to n^2; that is, $\Theta(14n^2 + 74) = \Theta(14n^2 + 74)$ (and so their big-O and big-Ω classes are also equivalent).

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

Solution: Let $c_1 = 3$, $c_2 = 4$ and $n_0 = 1$. We will show that for all $n \geq n_0$, $c_1 n^2 \leq f(n) \leq c_2 n^2$. We have:

$$f(n) = 3n^2 + \frac{\lg n}{n} \geq 3n^2 = c_1 n^2$$

since $\frac{\lg n}{n} > 0$. Thus the first inequality is satisfied. We also have:

$$f(n) = 3n^2 + \frac{\lg n}{n} \leq 3n^2 + n^2 = 4n^2 = c_2 n^2$$

since, when $n \geq n_0 = 1$, $\lg n \leq n^4$ so $\frac{\lg n}{n} \leq n^3$. Thus the second inequality is satisfied.
We have shown that there exist a c_1, c_2, and n_0 such that for all $n \geq n_0$, $c_1 n^2 \leq f(n) \leq c_2 n^2$. Hence $f(n) \in \Theta(n^2)$.

(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.

Solution: $f(n) \in O(n \lg n)$. To see this, let $c = 100$ and $n_0 = 1$. Then for all $n \geq n_0$ we have:

$$f(n) = 100n - 7 \leq 100n \leq 100n \lg n = cn \lg n.$$

On the other hand, $f(n) \notin \Omega(n \lg n)$. To see this, suppose there exists a c and an n_0 such that for all $n \geq n_0$ $cn \lg n \leq f(n)$. That is:

$$\begin{aligned} cn \lg n &\leq 100n - 7 \\ c \lg n &\leq 100 - \frac{7}{n} \\ c \lg n + \frac{7}{n} &\leq 100 \end{aligned}$$

which is a contradiction because the left-hand side approaches ∞ as n approaches ∞, but the right-hand side is a constant.

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

```
Heapsort(A) // A is an unsorted array of length n
   A.heapify()
   ans: a new array of length n
   for i = 0 to n-1
      ans[i] = A[0]
      A[0] = A[n-1-i]
      A.siftDown(0)
   for i = 0 to n-1
      A[i] = ans[i]
```

(a) The `heapify` that we wrote in class is a bottom-up, iterative method. Rewrite `heapify` as a top-down, recursive method (you may find that you need an extra parameter; this is fine).

Solution: Here's another version of `heapify`. We have an extra parameter `root`, which is the index of the root of the subtree we're heapifying.

```
Heapify(A, root)
   if root > n/2: DONE // base case: leaves are already heapified
   else
      Heapify(A, 2*root + 1) // heapify the left subtree
      Heapify(A, 2*root + 2) // heapify the right subtree
      siftDown(A, root) // get the root into the right place
```

(b) Write a recurrence for $T(n)$, the runtime of your `heapify` method from part (a). You can assume that n is one less than a power of 2, i.e., all levels of the heap are full.

Solution: Our `heapify` involves heapifying two subtrees, each of which has half as many nodes as the original tree, and then sifting down from the root. If the original tree has n nodes, our two calls to `heapify` on the subtree will each take time $T(n/2)$. Siftdown takes time $O(\lg n)$. Putting it together, we have:

$$T(n) = 2T(n/2) + O(\lg n),$$

which (to make the $O(n)$ term more explicit) we can also write as:

$$T(n) \leq 2T(n/2) + c \lg n$$

where c is an (unknown) constant. We also need to specify a base case. If $n = 3$ then our tree has only three nodes so heapifying it requires at most one swap; we can thus say:

$$T(3) = c_0$$

where c_0 is an (unknown) constant.

(c) Prove by induction that $T(n) \in O(n)$. (Hint: show that $T(n) \leq c_1 n - c_1 \lg n$ for the appropriate choices of c_1 and n_0 (again, you can assume that n is one less than a power of 2). This likely will be somewhat easier to show than trying to show directly that $T(n) \leq c_1 n$.)

Solution: Let $c_1 = 3c + c_0$ and $n_0 = 3$. We will show that for all $n \geq n_0$, $T(n) \leq c_1 n - c_1 \lg n$.

Base case: $n = n_0 = 3$. Then we have:

$$T(n) = T(3) = c_0 \leq 3c + c_0 = c_1 \leq c_1(n - \lg n)$$

as desired.

Inductive step: Assume that $T(n) \leq c_1 n - c_1 \lg n$ for some $n \geq n_0$. We will show that $T(2n) \leq 2c_1 n - c_1 \lg(2n)$. We have:

$$\begin{aligned}
T(2n) &\leq 2T(n) + c\lg(2n) && (1)\\
&\leq 2(c_1 n - c_1 \lg n) + c\lg(2n) && (2)\\
&= 2c_1 n - 2c_1 \lg n + c\lg(2n) && (3)\\
&= 2c_1 n - (2c_1 \lg(2n) - 2c_1) + c\lg(2n) && (4)\\
&= 2c_1 n - c_1 \lg(2n) - (c_1 \lg(2n) - 2c_1 - c\lg(2n)) && (5)\\
&\leq 2c_1 n - c_1 \lg(2n). && (6)
\end{aligned}$$

Here line (1) follows from substituting $2n$ into the recurrence from part (b), line (2) follows from the inductive hypothesis (substituting $T(n) \leq c_1 n - c_1 \lg n$), lines (3)-(5) are algebraic, and line

(6) can be seen as follows: when $c_1 = 3c + c_0$ we have

$$\begin{aligned} c_1 \lg(2n) - 2c_1 - c\lg(2n) &= (3c + c_0)\lg(2n) - 2(3c + c_0) - c\lg(2n) && (7)\\ &= (2c + c_0)\lg(2n) - 6c - 2c_0 && (8)\\ &\geq 3(2c + c_0) - 6c - 2c_0 && (9)\\ &= c_0 && (10) \end{aligned}$$

where line (7) follows from substituting $c_1 = 3c + c_0$ and line (9) follows from the fact that $n \geq n_0 = 4$. This tells us that the extra subtracted term in line (5) is positive, so line (6) follows.

(d) Put it all together: What is the asymptotic 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). You may find that you need to review the asymptotic analysis of heaps!

Solution: The overall runtime of heapsort is $O(n \lg n)$. Here's why:

- Line 2: we've just seen in part (c) that `heapify` takes time $O(n)$.
- Lines 4-7: the loop runs n times. Lines 5 and 6 are constant time and the `siftDown` in line 7 takes time $O(\lg n)$, so in all we have $O(n \lg n)$ (number of iterations multiplied by time per iteration).
- Lines 8-9: the loop runs n times and does constant work on each iteration, so in all we have $O(n)$.

Adding these up, we have $O(n) + O(n \lg n) + O(n)$ which is equal to $O(n \lg n)$ since $O(n) \subset O(n \lg n)$.

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 <u>what</u> bubble sort is doing (i.e., what is the sequence of steps it follows?) and <u>why</u> 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.

Solution: Bubble sort "bubbles" larger elements up to the top (right side) of the array. The idea is to walk down the array, comparing each pair of consecutive elements and swapping them if they are out of order. The algorithm walks down the array n times, at the end of which all n elements in order.

To understand why bubble sort works, we make the following observation: on the `i`th iteration of the outer loop, the `i`th largest element ends up in the right place. Why? Consider the 0th iteration. If the largest element is not already in place, then it's in some position `k` in the middle of the array. ***Once the inner loop reaches position `k`, the largest element gets swapped with the element to the right of it on each remaining iteration of the inner loop. This brings the largest element to the right-most position.*** On the next iteration, the second-largest element bubbles up to the second-to-last position in the array for the same reason: it is larger than, and gets swapped with, every element it's compared to in the inner loop until it reaches the end of the array. Continuing with this line of reasoning, we see that after `i` iterations, the `i` largest elements are in sorted order at the right of the array. So after `n` iterations the entire array is sorted.

***These sentences are particularly important in explaining why bubble sort works. Many of you observed that the largest element ends up in the right-most position at the end of the first iteration, but didn't explain *how we know* that happens. It's important to go back to the algorithm and explain exactly why the steps the algorithm follows lead to the observed outcome.

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

Solution: Bubble sort runs in $O(n^2)$. Lines 4-5 take constant time. These are inside the loop on line 3, which runs for $O(n)$ iterations, so the entire loop in lines 3-5 takes $O(n)$. This is inside the loop on line 2, which runs for n iterations, so the entire loop in lines 2-5 (and hence the entire algorithm) takes time $O(n^2)$.

(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.)

Solution: Mergesort has asymptotic runtime $O(n \lg n)$. Since $n \lg n$ grows strictly slower than n, we'd expect mergesort to run faster than bubble sort. Both bubble sort and insertion sort are in $O(n^2)$, so based on the asymptotic analysis we'd expect them to have similar wall-clock runtimes.

(d) Download my `Sort.java` code from:
https://kgardner.people.amherst.edu/courses/f18/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 ant 10^6.
- Sort the array using {insertion sort, mergesort, bubble sort}. Time how long this takes by recording the system time before and after running the sorting method. You can do this as follows:

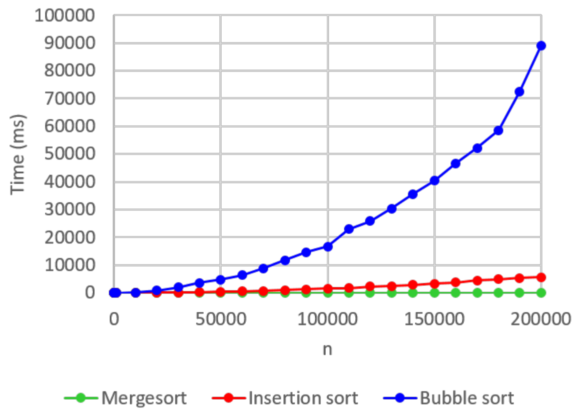
```
long start_time = System.nanoTime();
// whatever code you want to time
long end_time = System.nanoTime();
long time_my_code_took = end_time - start_time;
```

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?

Solution: The asymptotic analysis is a good predictor of the general shape of the runtime curve as a function of n, particularly as n gets large (see Figure 1). We can see from the graphs that both bubble sort and insertion sort have a quadratic shape (it helps to zoom in and remove bubble sort from the graph to see that insertion sort is quadratic), whereas the shape for mergesort matches the shape of the function $f(n) = n \lg n$. Since n^2 and $n \lg n$ have very different behavior as n gets large, the asymptotic analysis also does a good job predicting that mergesort will perform better than insertion sort and bubble sort. We can see from the graphs that this is indeed what happens.



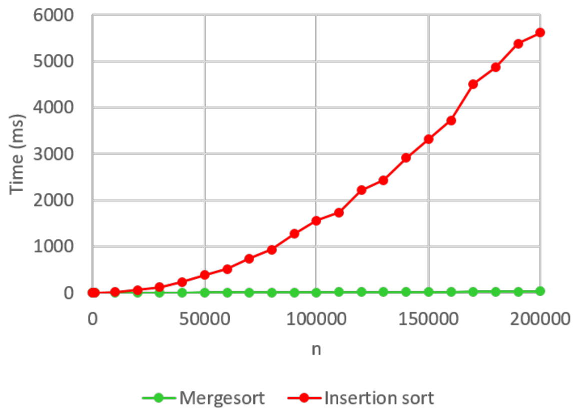
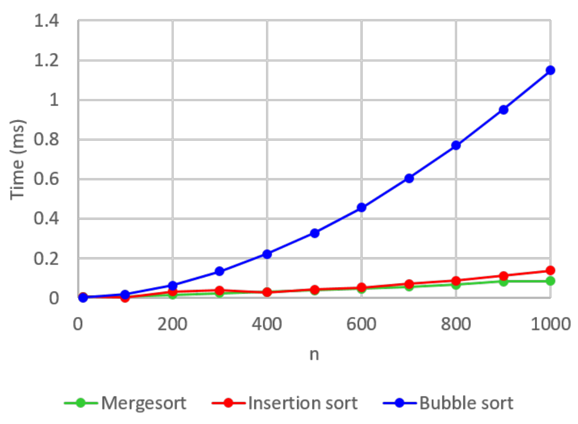


Figure 1: Comparing the runtimes of mergesort, insertion sort, and bubble sort. Left: all three sorting algorithms. Right: zooming in on just mergesort and insertion sort.

When n is low (see Figure 2), the asymptotic analysis is worse at predicting the relative performance of the three algorithms. We can still see in the graph on the left that bubble sort is worse than mergesort, and even that it looks to be quadratic, but we're no longer able to differentiate between insertion sort and mergesort, which have nearly identical performance when $n \leq 1000$. At even lower n (see the graph on the right for $n \leq 100$), there's too much noise for us to distinguish between the three algorithms at all. This is fine: asymptotic analysis is not really intended to tell us anything useful about what is going on at small values of n.



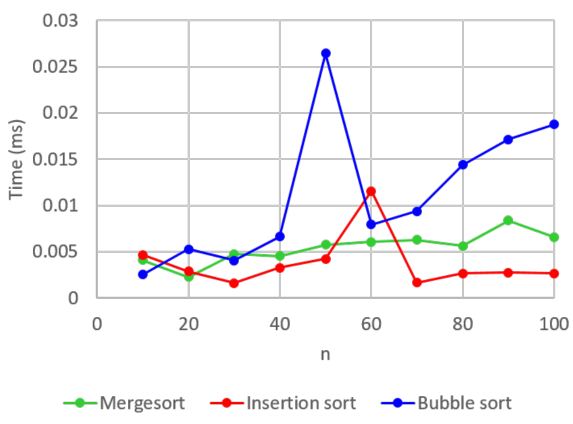


Figure 2: Comparing the runtimes of mergesort, insertion sort, and bubble sort at low n. Left: $n \leq 1000$. Right: zooming in further on $n \leq 100$.

What the asymptotic analysis is missing is the ability to differentiate between algorithms that have the same asymptotic performance. Insertion sort clearly takes much less time to run than bubble sort, even though they both have $O(n^2)$ runtime. Asymptotic analysis ignores lower-order terms and constant multiplicative factors, and in this case these factors make a big difference!

What this tells us is that while asymptotic analysis is useful for getting the big picture about how to order algorithms with respect to their runtime, it doesn't tell the full story. When you go out into the world and are asked to select an algorithm to perform a certain task, you can be reasonably confident that, at high enough n, an $O(n)$ algorithm will outperform, say, and $O(n^2)$ algorithm. But not all $O(n)$ algorithms are created equal, and we need different tools to evaluate the relative performance of algorithms that belong to the same asymptotic class.