CS 161 **Homework 1**
Summer 2018 **Due:** Tuesday 7/10 at 5 p.m. on Gradescope

Style guide and expectations: Please see Homework 0 solutions and Tutorial 2 solutions for guidance on what we are looking for in homework solutions. We will grade according to these standards.

Exercises

Exercises should be completed **on your own**.

---

0. **(1 pt.)** Have you thoroughly read the course policies on the webpage?

   [We are expecting: The answer yes.]

1. **(1 pt.)** See the Jupyter notebook `hw1.ipynb` for Exercise 1. Modify the code to generate a plot that convinces you that $T(x) = O(g(x))$.

   [We are expecting: Your choice of c, n_0, the plot that you created after modifying the code in Exercise 1, and a short explanation of why this plot should convince a viewer that $T(x) = O(g(x))$.]

2. **(3 pt.)** See the Jupyter notebook `hw1.ipynb` for Exercise 2.

   (a) (0.5 pt.) Referencing Fig 2.1-3, for what input types (random, reverse-sorted, or sorted) is insertion sort faster than mergesort? And for what input types is mergesort faster than insertion sort? If you did not know the input type a priori, which of the two algorithms should you prefer?

   [We are expecting: 1-2 sentence answers to these questions.]

   (b) (0.5 pt.) Referencing Fig 2.4-5, how does the variability of runtimes of insertion sort over different input types compare to the variability of runtimes of mergesort over different inputs types? Given what you know about insertion sort and mergesort, offer an explanation of why this is the case.

   [We are expecting: 1-2 sentence answers to these questions.]

   (c) (0.5 pt.) Modify the code in `hw1.ipynb` to generate a plot that convinces you (and the grader) of the runtime of insertion sort on a sorted list (tight-bound on the best-case runtime).

   [We are expecting: Your choice of c_1, c_2, n_0, and $g(n)$; the plot that you created after modifying the code in Exercise 2c; and a short explanation of why this plot should convince a viewer that the best-case runtime of insertion sort is what you claimed it was.]

   (d) (0.5 pt.) Modify the code in `hw1.ipynb` to generate a plot that convinces you (and the grader) of the runtime of insertion sort on a reverse-sorted list (tight-bound on the worst-case runtime).

   [We are expecting: Your choice of c_1, c_2, n_0, and $g(n)$; the plot that you created after modifying the code in Exercise 2d; and a short explanation of why this plot should convince a viewer that the best-case runtime of insertion sort is what you claimed it was.]

   (e) (1 pt.) How much time do you think it would take to sort a worst-case input of size $n = 10^{10}$ using insertion sort? Using mergesort?

   [We are expecting: Your answer (in whichever unit of time makes the most sense) with a brief justification, that references the runtime data you used in the earlier parts. You don't need to do any fancy statistics, just a reasonable back-of-the-envelope calculation.]

3. **(4 pt.)** Using the definition of Big-O, Big-Ω, and Big-Θ, formally prove the following statements.

   (a) (1 pt.) $10^n = O(n!)$

   (b) (1 pt.) $\log_a(n) = \Theta(\log_b(n))$ for all $a, b \geq 2 \in \mathbb{N}$

   (c) (1 pt.) $n^2 = \Omega(n \log(n))$

   (d) (1 pt.) n^2 is **not** $O(n \log(n))$

   [We are expecting: For each part, a rigorous (but short) proof, using the definition of Big-O, Big-Ω, and Big-Θ.]

4. **(4 pt.)** Solve the following recurrence relations; i.e. express each one as $T(n) = O(f(n))$ for the tightest possible function $f(n)$, and give a short justification.

   [To see the level of detail expected, we have worked out the first one for you.]

   (z) $T(n) = 6T(n/6) + 1$. We apply the Master theorem with $a = b = 6$ and with $d = 0$. We have $a > b^d$, so the runtime is $O(n^{\log_6(6)}) = O(n)$.

   (a) (1 pt.) $T(n) = 2T(n/2) + 3n$

   (b) (1 pt.) $T(n) = 2T(n/3) + n^c$, where $c \geq 1$ is a constant (that is, it doesn't depend on n).

   (c) (1 pt.) $T(n) = 4T(n/2) + n\sqrt{n}$

   (d) (1 pt.) $T(n) = 3T(n/3) + 6n$

Problems

You can collaborate with your classmates about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your solutions yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

---

1. **(Not Tinder) (6 pt.)** Distressed by the impending reality in which technology companies facilitate all first-encounters between potential friends, thereby reducing the nuanced dance of social courtship to a phone swipe, Vivian, the entrepreneurial crusader, opens a coffee and bagel shop called **Not Tinder** and invites guests to spontaneously meet other guests.

   Vivian wants to track how many distinct pairs of guests ever simultaneously occupy her shop at the same time, so she can make more intelligent business decisions to challenge her soulless competitors. For each guest $i = 1, \ldots, n$, guest i enters the shop at time a_i and leaves at time $b_i \geq a_i$. Vivian is interested in the question: how many distinct pairs of guests are ever in her shop at the same time? (Here, the pair (i, j) is the same as the pair (j, i)).

   For example, suppose there are 5 guests with the following entering and leaving times:

Guest	Enter time	Leave time
1	1	4
2	2	5
3	7	8
4	9	10
5	6	10

   Then, the number of distinct pairs of guests who are in the shop at the same time is three: these pairs are $(1, 2)$, $(4, 5)$, $(3, 5)$. (Drawing the intervals on a number line may make this easier to see).

   (a) (2 pt.) Given input $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ as above in no particular order (i.e. not sorted in any way), there is a straightforward algorithm that takes about [1] n^2 time to compute the number of pairs of guests who are ever on the shop at the same time. Give this algorithm and explain why it takes time about n^2.

   [We are expecting: Either an English description or pseudocode of the algorithm, and an English description of why it takes time about n^2.]

   (b) (4 pt.) Give an $O(n \log(n))$-time algorithm to do the same task and analyze its running time.

   [We are expecting: Either an English description or pseudocode of the algorithm, and an English description of why it takes $O(n \log(n))$-time.]

[1] Formally, "about" here means $\Theta(n^2)$, but you can be informal about this.

2. **(Counting currencies) (9 pt.)**

(a) (2 pt.) Working as a cashier at a local supermarket, Vin wants to be able to efficiently report the k^{th}-smallest denomination of US currency occupying his cash register at the end of each day. Over the course of the day, customers pay in coins—pennies, nickels, dimes, quarters, half-dollars (but not dollar coins)—as well as cash—\$1, \$2, \$5, \$10, etc. bills, so that at the end of each day, his cash register might have any number of each denomination of US currency, but the value of each denomination is distinct. Because of the large volume of transactions he processes per day, he has found it to be most efficient to store information about the money in his cash register in a compressed format as an unsorted list of `(denomination, count)` pairs. For example, his dataset might resemble the following:

`($2, 8), ($1, 16), ($10, 2), ($5, 4), ($20, 1), (50¢, 32)`

In this example, the $k = 0$-smallest denomination (i.e. the smallest one) is `50¢`, the $k = 32$-smallest denomination is `$1`, and the $k = 60$-smallest denomination is `$10`.

Let m be the number of denominations of US currency that Vin will handle. Considering that these m denominations and their relative ordering are known a priori, use an existing **linear-time sorting algorithm** as a subroutine to design an $O(m)$-time algorithm for finding the k^{th}-smallest denomination of US currency in Vin's cash register at the end of the day.

[We are expecting: Either an English description or pseudocode of the algorithm, and an English justification of why it takes $O(m)$-time.]

(b) (3 pt.) Prove formally, using induction, that your answer to part (a) is correct. You do not need to prove the correctness of the linear-time sorting algorithm that you used as a subroutine.

[We are expecting: A formal argument by induction. Make sure you explicitly state the inductive hypothesis, base case, inductive step, and conclusion.]

(c) (4 pt.) Vin moves to a little-known country called Floop and again finds himself working as a cashier. He still wants to be able to efficiently report the k^{th}-smallest denomination of currency occupying his cash register at the end of each day and stores information about this money in the same compressed format as before. However, now the relative ordering of the m denominations of Floop's currency is not known a priori.

For example, his dataset might resemble the following:

`(boop, 8), (noop, 16), (loop, 2), ...`

Fortunately, given two denominations of Floop's new currency, Vin's cashier coworker in the adjacent aisle can tell him which one is more valuable. In this example, suppose Vin's coworker tells him that `boops` are less valuable than `loops`. Then his coworker tells him that `loops` are less valuable than `noops`. Assuming all other denominations are more valuable than `noops`, the $k = 0$-smallest denomination (i.e. the smallest one) is `boop`, the $k = 9$-smallest denomination is `loop`, and the $k = 20$-smallest denomination is `noop`.

Using a modified version of the **linear-time select algorithm**, design an $O(m)$-time algorithm for finding the k^{th}-smallest denomination of Floop's currency in Vin's cash register at the end of the day. In your algorithm, feel free to use `compare(d1, d2)`, which accepts two denominations of Floop's currency as input and returns which one is less valuable as output.

[We are expecting: Either an English description or pseudocode of the algorithm, and an English justification of why it takes $O(m)$-time.]