CS 161 **Problem Set 3**
Fall 2025 **Due:** Friday October 24 at 11:59pm on Gradescope.

Style guide and expectations: Please see the top of the "Homework" page on the course webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards.

Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Collaboration policy: You may do the HW in groups of size up to three. Please submit one HW for your whole group on Gradescope. (Note that there is an option to submit as a group). See the "Policies" section of the course website for more on the collaboration policy.

LLM policy: Check out the course webpage for best practices on how to productively use LLMs on homework, if you use them at all.

Exercises

We recommend you do the exercises on your own before collaborating with your group. The point is to check your understanding.

---

1. **(6 pt.)** In this exercise, we'll explore different types of randomized algorithms. We say that a randomized algorithm is a **Las Vegas Algorithm** if it is always correct, but the running time is a random variable. We say that a randomized algorithm is a **Monte Carlo Algorithm** if there is some probability that it is incorrect. For example, QuickSort (with a random pivot) is a Las Vegas algorithm, since it always produces a sorted array (but if we get very unlucky QuickSort may be slow).

   Suppose that there is a population of n ducks. Half of the ducks are green, and half are brown, but it's dark outside and you can't tell the difference until you catch one and look at it with a flashlight. Assume that catching a random duck and looking at it takes time $O(1)$.

   The algorithms given in Figure 1 all attempt to find a single green duck. Fill in the chart below. You may use asymptotic notation for the running times; give the best big-Oh bound that you can. For the probability of returning a correct duck, do not use asymptotic notation; give the best bound that you can.

Algorithm	Monte Carlo or Las Vegas?	Expected running time	Worst-case running time	Probability of returning a green duck
Algorithm 1				
Algorithm 2				
Algorithm 3				

   Note that the LaTeX code for the table is available in the source file.

[**Hint:** *Remember (see Lecture 5 and the pre-lecture exercise) that the expectation of a geometric random variable with probability p is equal to $\frac{1}{p}$*]

[**We are expecting:** *A filled-in table. No justification is required.*]

Algorithm 1: findGreenDuck1

Input: A flock of n ducks
while *true* **do**
    Choose a random duck from the flock;
    if *That duck is green* **then**
        return *that duck*
    else
        Release the duck back into the flock

Algorithm 2: findGreenDuck2

Input: A flock of n ducks
for *100 iterations* **do**
    Choose a random duck from the flock;
    if *That duck is green* **then**
        return *that duck*
    else
        Release the duck back into the flock
Choose a random duck from the flock;
return *That duck*

Algorithm 3: findGreenDuck3

Input: A flock of n ducks
Put the ducks in a line, in a random order ;
/* Assume it takes time $O(n)$ to put the n ducks in a random order; and assume that they stay put once ordered. */
for $i = 0, \ldots, n-1$ **do**
    if *Duck i is green* **then**
        return *Duck i*

Figure 1: Three algorithms for finding a green duck

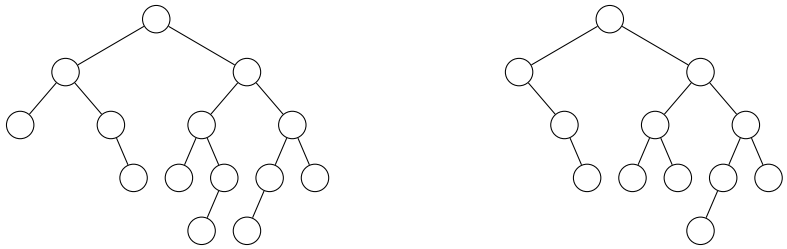
2. **(4 pt.)** [**Coloring RB Trees.**]

   Can you color in the nodes of the trees below to make legitimate red-black trees? For each tree, either color the nodes to make a valid red-black tree, or say that no such coloring exists.

   Note: the LaTeX code to make these trees is included in the template. If you'd like to use this code, you can color a node by editing it like:

   `\node[draw,circle,fill=red]` or `\node[draw,circle,fill=black]`.

   Or you can just color the nodes in your favorite drawing program and include the image with `\includegraphics{my_image.png}`.



   [**We are expecting:** *For each tree, either an image of a colored-in red-black tree or a statement "No such red-black tree." No justification is required.*]

3. **(5 pt.)** This exercise references the IPython notebook `HW3.ipynb`, available on the course website along with this problem set.

   In our implementation of radixSort in class, we used bucketSort to sort each digit. Why did we use bucketSort and not some other sorting algorithm? There are several reasons, and we'll explore one of them in this exercise.

   (a) **(2 pt.)** One reason we chose bucketSort was that it makes radixSort work correctly! In `HW3.ipynb`, we've implemented four different sorting algorithms—bucketSort, quickSort, and two versions of mergeSort—as well as radixSort.

   Note: The IPython notebook is long, but just because it implements many different sorting algorithms. Don't get scared!

   There is a `TODO` statement in the IPython notebook where you can change the code to use different sorting algorithms; you just have to make sure that the sorting algorithm you want to use is the one that is not commented out. No programming necessary!

   Modify the code for radixSort to use each one of these four algorithms within radixSort, and test it out on the examples suggested.

   Which sorting algorithms seem to be correct as "inner sorting algorithms" for radixSort?

   - Does using bucketSort always work correctly?
   - Does using quickSort always work correctly?
   - Does using mergeSort (with merge1) always work correctly?
   - Does using mergeSort (with merge2) always work correctly?

   [**We are expecting:** *Yes or no for each part.*]

(b) **(3 pt.)** Explain what you saw above. What was special about the algorithms which worked? Why does this special thing matter? (You may wish to play around with `HW3.ipynb` to "debug" the incorrect cases.)

[**We are expecting:** *A clear definition of the special property that the correct algorithms have, and a few sentences explaining why it matters. A minimal example of what might go wrong is a great way to explain why this property matters.*

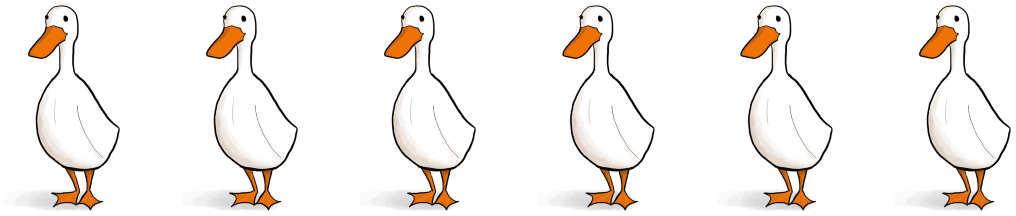
You do not need to justify why each of the algorithms do or do not have the property.]

Note: yes, this property is alluded to in the reading. That's why this is an exercise and not a problem! To get the most out of this exercise, play around with the examples in the code and make sure you really understand what's going on.

Problems

---

4. **(6 pt.)** **[Ducks.]** Suppose that n ducks are standing in a line.



Each duck has a political leaning: left, right, or center. You'd like to sort the ducks so that all the left-leaning ones are on the left, the right-leaning ones are on the right, and the centrist ducks are in the middle. You can only do two sorts of operations on the ducks:

Operation	Result
`poll(j)`	Ask the duck in position j about its political leanings
`swap(i,j)`	Swap the duck in position j with the duck in position i

However, in order to do either operation, you need to pay the ducks to cooperate: each operation costs one piece of duckweed. Also, you didn't bring a piece of paper or a pencil (or your smartphone or tablet or tablet or whatever you use to take notes) so you can't write anything down and have to rely on your memory! Like many humans, your memory is limited, and you can only remember up to seven[1] integers between 0 and $n-1$ at a time (i.e. you can use at most seven integer-valued variables at a time in your algorithm).

Design an algorithm to sort the ducks, which costs $O(n)$ pieces of duckweed, and uses no extra memory other than storing at most seven[2] integers between 0 and $n-1$.

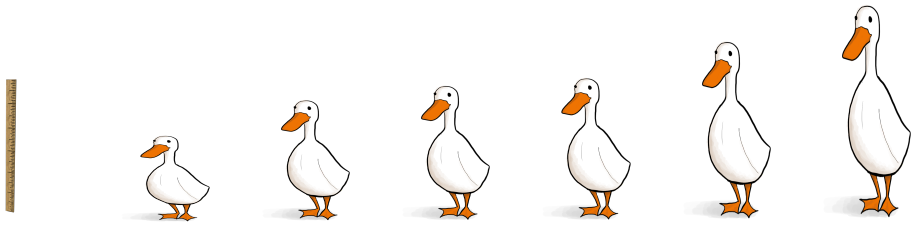
[**Hint:** *Does this task look like anything we've seen in class?*]

[**We are expecting:** *Pseudocode* ***AND*** *a short English description of your algorithm;* ***AND*** *a short explanation of why it uses only $O(n)$ pieces of duckweed and never uses more than seven numbers of memory.*]

[1] `https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two`

[2] You don't need to use all seven storage spots, but you can if you want to. Can you do it with only two?

5. **[Ducks.] (6 pt.)** Suppose that n ducks of distinct heights are standing in a line, ordered from shortest to tallest.



You have a measuring stick of a certain height, and you would like to identify a duck which is the same height as the stick, or else report that there is no such duck. The only operation you are allowed to perform is `compareToStick(f)`, where f is a duck (that is, you cannot directly access the heights of each duck). `compareToStick(f)` returns `taller` if f is taller than the stick, `shorter` if f is shorter than the stick, and `the same` if f is the same height as the stick. You've still forgotten to bring a paper and a pencil and so you can only store up to seven integers in $\{0, \ldots, n-1\}$ at a time. And you have to pay a duck one piece of duckweed every time you perform `compareToStick` on it.

(a) **(2 pt.)** Give an algorithm in this model of computation which either finds a duck the same height as the stick, or else returns "No such duck," and uses $O(\log(n))$ pieces of duckweed.

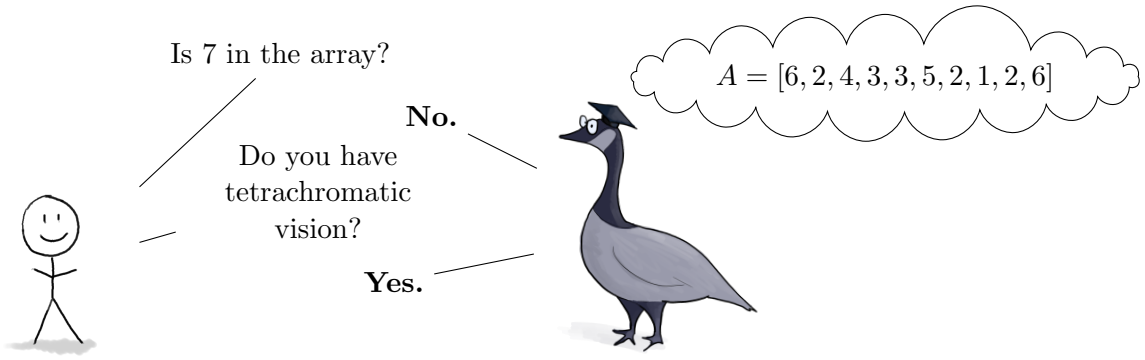
[**We are expecting:** *Pseudocode* ***AND*** *an English description. You do not need to justify the correctness or duckweed usage.*]

(b) **(4 pt.)** Prove that any algorithm in this model of computation must use $\Omega(\log(n))$ pieces of duckweed.

[**We are expecting:** *A short but convincing argument.*]

6. **[Goose!] (5 pt.)** A wise goose has knowledge of an array A of length n, such that $A[i] \in \{1, \ldots, k\}$ for all i. (Note that the elements of A are not necessarily distinct). You don't have direct access to A, but you can ask the wise goose *any* yes/no questions about it. For example, you could ask "If I remove $A[5]$ and swap $A[7]$ with $A[8]$, would the array be sorted?" or "can some geese fly as high as 29,000ft?"

   Unlike in the previous problems, this time you did bring a paper and pencil, and your job is to write down all of the elements of A in sorted order.[3] The wise goose charges one piece of duckweed per question.[4]



   (a) **(5 pt.)** Give a procedure that outputs a sorted version of A which uses $O(k \log(n))$ pieces of duck weed. You may assume that you know n and k, although this is not necessary.

   [**We are expecting:** *Pseudocode AND a short English description of your algorithm;* ***AND*** *a brief explanation of why it uses $O(k \log(n))$ pieces of duckweed.*]

   (b) **(1 BONUS pt.)** Prove that any procedure to solve this problem must use $\Omega(k \log(n/k))$ pieces of duckweed.

   [**We are expecting:** *Nothing; this part is not required.*]

[3] Note that you don't have any ability to change the array A itself, you can only ask the wise goose about it.
[4] Despite the name, it turns out that geese also eat duckweed.