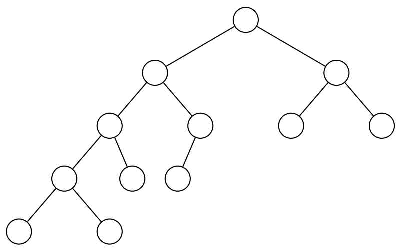
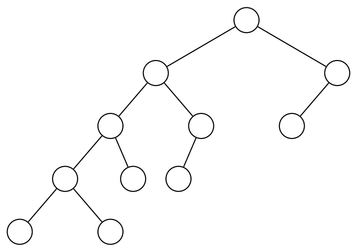
CS 161 **Problem Set 3**
Winter 2019 **Due:** Friday, February 8, 2019, at 3pm on Gradescope

Exercises

Please do the exercises on your own.

1. **(2 pt.)** For each of the following examples, if the nodes can be colored red or black to make a legitimate red-black tree, then give such a coloring. If not, then say that they cannot.

   (For reference, the LaTeXcode to make these trees is included at the end of the PSET, as well as instructions about how to color the nodes red or black. If you use this code, make sure you include `\usepackage{tikz}` before the `\begin{document}` command. You are also welcome to take a screenshot and color the trees in MSPaint, or make a hand-drawn copy and take a photo, or...)

   [**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.*]

2. **(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 (that is, whenever it returns, it returns the right answer), 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 returns a sorted array, but it might be slow if we get very unlucky.

   The penguin population from PSET 2 is back, and you want to solve the same problem as you did on PSET 2 using randomized algorithms. As a reminder, there are n penguins on an island, of many different species. You can't tell the penguins apart, but there's an expert penguinologist who can, and she can answer queries to `isTheSame(p1,p2)` which returns True if `p1` and `p2` are penguins of the same species, and False otherwise. Your goal is to find a member of the majority species (assuming there is one). Consider the three algorithms seen in Figure 1, all of which call the subroutine isMajority (also seen in Figure 1), and all of which attempt to find a majority penguin.

   Fill in the table below, and justify your answers. If it is helpful, the LaTeXcode for the table is copied at the end of the problem set.

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

   [**We are expecting:** *Your filled in-table, and a short justification for each entry of the table. You may use asymptotic notation for the running times; for the probability of returning a majority penguin, give the tightest bound that you can given the information provided.*]

```
Algorithm 1: findMajorityPenguin1
  Input: A population $P$ of $n$ penguins
  while true do
      Choose a random $p \in P$;
      if isMajority($P, p$) then
          return $p$;
```

```
Algorithm 2: findMajorityPenguin2
  Input: A population $P$ of $n$ penguins
  for 100 iterations do
      Choose a random $p \in P$;
      if isMajority($P, p$) then
          return $p$;
  return $P[0]$;
```

```
Algorithm 3: findMajorityPenguin3
  Input: A population $P$ of $n$ penguins
  Put the penguins in $P$ in a random order.;
  /* Assume it takes time $\Theta(n)$ to put the $n$ penguins in a random order      */
  for $p \in P$ do
      if isMajority($P, p$) then
          return $p$;
```

```
Algorithm 4: isMajority
  Input: A population $P$ of $n$ penguins and a penguin $p \in P$
  Output: True if $p$ is a member of a majority species
  count $\leftarrow 0$;
  for $q \in P$ do
      if isTheSame($p,q$) then
          count ++;
  if count $> n/2$ then
      return True;
  else
      return False;
```

Figure 1: Three randomized algorithms for finding a majority penguin

3. **(3 pt.)** This exercise references the IPython notebook `HW3.ipynb`, available on the course website.

   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) **(1 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. 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?

   i. Does using bucketSort always work correctly?
   ii. Does using quickSort always work correctly?
   iii. Does using mergeSort (with merge1) always work correctly?
   iv. Does using mergeSort (with merge2) always work correctly?

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

   (b) **(2 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. You do not need to justify why each of the algorithms do or do not have the property.*]

Problems

You may talk with your fellow CS161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers 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.

4. **(6 pt.)** [**Duck.**] 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 co-operate: each operation costs one stale hot dog bun.[1] Also, you didn't bring a piece of paper or a pencil, so you can't write anything down and have to rely on your memory. Like many humans, you can remember up to seven[2] integers between 0 and $n-1$ at a time.

(a) **(4 pt.)** Design an algorithm to sort the ducks which costs $O(n)$ stale hot dog buns, and uses no extra memory other than storing at most seven[3] integers between 0 and n.

[**We are expecting:** *Pseudocode* ***AND*** *a short English description of your algorithm.*]

(b) **(2 pt.)** Justify why your algorithm is correct, and why it uses only $O(n)$ stale hot dog buns and stores at most seven integers at a time.

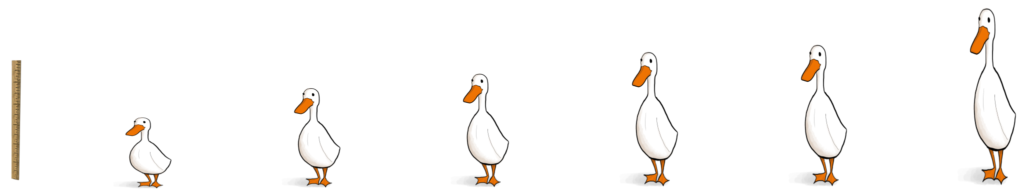
[**We are expecting:** *An informal justification which is both clear and convincing to the grader. If it's easier for you to be clear, you can give a formal proof of correctness, but you do not have to.*]

[1] Probably you shouldn't be feeding ducks stale hot dog buns, but that's all you have.

[2] see, e.g., `https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two`

[3] 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. **(5 pt.)** [**Duck.**] Suppose that n ducks 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 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 do is `compareToStick(j)`, where $j \in \{0, \ldots, n-1\}$, which returns `taller` if the j'th duck is taller than the stick, `shorter` if the j'th duck is shorter than the stick, and `the same` if the j'th duck is the same height as the stick. As in Problem 4, you have to pay a duck one stale hot dog bun in order to compare it to the stick.

   (a) **(2 pt.)** Give an algorithm which either finds a duck the same height as the stick, or else returns "No such duck," in the model above which uses $O(\log(n))$ stale hot dog buns.

   [**We are expecting:** *Pseudocode* ***AND*** *an English description of your algorithm. You do not need to justify the correctness or stale hot dog bun usage.*]

   (b) **(3 pt.)** Prove that any algorithm in this model of computation must use $\Omega(\log(n))$ stale hot dog buns.

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

6. **(4 pt.)** [**Goose!**] A goose comes to you with the following claim. They say that they have come up with a new kind of binary search tree, called `gooseTree`, even better than red-black trees!

More precisely, `gooseTree` is a data structure that stores data in a binary search tree. It might also store other auxiliary information, but the goose won't tell you how it works. The goose claims that `gooseTree` supports the following operations:

- `gooseInsert(k)` inserts an item with key k into the `gooseTree`, maintaining the BST property. It does not return anything. It runs in time $O(1)$.
- `gooseSearch(k)` finds and returns a pointer to node with key k, if it exists in the tree. It runs in time $O(\log(n))$, where n is the number of items in the `gooseTree`.
- `gooseDelete(k)` removes and returns a pointer to an item with key k, if it exists in the tree, maintaining the BST property. It runs in time $O(\log(n))$, where n is the number of items in the `gooseTree`.

The goose says that all these operations are deterministic, and that `gooseTree` can handle arbitrary comparable objects.

You think the goose's logic is a bit loosey-goosey. How do you know the goose is wrong?

[**We are expecting:** *A formal proof that the goose must be wrong. (It is okay if it is a short proof). You may use results or algorithms that we have seen in class without further justification.*]

[**HINT:** *Since the goose won't tell you how* `gooseTree` *works, you cannot assume anything about how* `gooseInsert`*,* `gooseSearch` *or* `gooseDelete` *might be implemented.*]

[**HINT:** *Because* `gooseTree` *stores a BST, you can still do* `inOrderTraversal` *in time* $O(n)$*.*]

Feedback

This part is not worth any points, but it is quick, painless, and anonymous, and we'd really appreciate it if you help us out by giving us feedback!

1. **(0 pt.)** Please fill out the following poll, which asks your opinion on the homework:

   `https://goo.gl/forms/eSssHWRPgRUva6bA2`

Helpful LaTeXcode!

Here is some code to generate the red-black trees; we've shown how to color one of the nodes green, you can use this example to color them red or black. You are also welcome to take a screenshot and color in your favorite paint program, or include a picture of a hand-drawn image, or any other way you can think of getting a picture of a colored red-black tree into your PSET.

```
\begin{center}
\begin{tikzpicture}[xscale=0.7]
\begin{scope}
\node[draw,circle,fill=green](b) at (0,0) {};
\node[draw,circle](b0) at (-2,-1) {};
\node[draw,circle](b1) at (2,-1) {};
\node[draw,circle](b00) at (-3,-2) {};
\node[draw,circle](b01) at (-1,-2) {};
\node[draw,circle](b10) at (1,-2) {};
\node[draw,circle](b11) at (3,-2) {};
\node[draw,circle](b000) at (-4,-3) {};
\node[draw,circle](b001) at (-2.5,-3) {};
\node[draw,circle](b010) at (-1.5,-3) {};
\node[draw,circle](b0000) at (-5,-4) {};
\node[draw,circle](b0001) at (-3,-4) {};
\draw (b) -- (b0);
\draw (b) -- (b1);
\draw (b0) -- (b00);
\draw (b0) -- (b01);
\draw (b1) -- (b10);
\draw (b1) -- (b11);
\draw (b00) -- (b000);
\draw (b00) -- (b001);
\draw (b01) -- (b010);
\draw (b000) -- (b0000);
\draw (b000) -- (b0001);
\end{scope}

\begin{scope}[xshift=10cm]
\node[draw,circle](b) at (0,0) {};
\node[draw,circle](b0) at (-2,-1) {};
\node[draw,circle](b1) at (2,-1) {};
\node[draw,circle](b00) at (-3,-2) {};
\node[draw,circle](b01) at (-1,-2) {};
\node[draw,circle](b10) at (1,-2) {};
\node[draw,circle](b000) at (-4,-3) {};
\node[draw,circle](b001) at (-2.5,-3) {};
\node[draw,circle](b010) at (-1.5,-3) {};
\node[draw,circle](b0000) at (-5,-4) {};
\node[draw,circle](b0001) at (-3,-4) {};
\draw (b) -- (b0);
\draw (b) -- (b1);
\draw (b0) -- (b00);
\draw (b0) -- (b01);
\draw (b1) -- (b10);
\draw (b00) -- (b000);
\draw (b00) -- (b001);
\draw (b01) -- (b010);
\draw (b000) -- (b0000);
\draw (b000) -- (b0001);
\end{scope}

\end{tikzpicture}
\end{center}
```

Here is the code that generated the table for the penguin exercise:

```
\begin{center}
\begin{tabular}{|c|p{3cm}|p{2cm}|p{2cm}|p{4cm}|}
\hline
Algorithm & Monte Carlo or Las Vegas? & Expected running time
& Worst-case running time & Probability of returning a majority penguin \\
\hline
\textbf{Algorithm 1} &  &  &  & \\
\hline
\textbf{Algorithm 2} &  &  &  & \\
\hline
\textbf{Algorithm 3} &  &  &  & \\
\hline
\end{tabular}
\end{center}
```