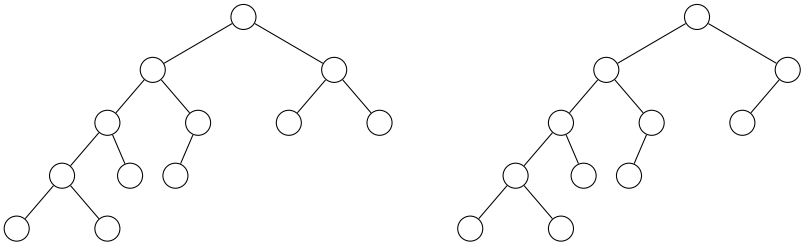
Exercises

Please do the exercises on your own.

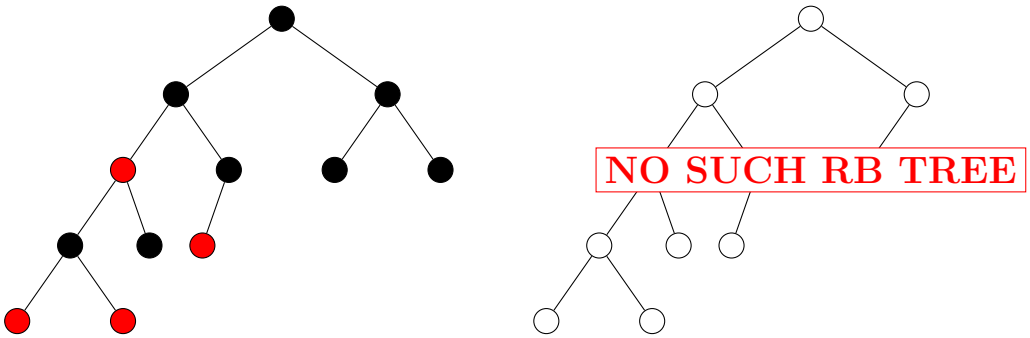
1. **(2 pt.)** For each of the following examples, if the nodes can be colored red or black to make a legitimate red-black tree, then give such a coloring. If not, then say that they cannot.

   (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.*]

   SOLUTION:



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, it returns the right answer with probability 1), 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 HW3.

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.*]

SOLUTION:

Algorithm	Monte Carlo or Las Vegas?	Expected running time	Worst-case running time	Probability of returning a majority penguin
Algorithm 1	Las Vegas	$O(n)$	∞	1
Algorithm 2	Monte Carlo	$O(n)$	$O(n)$	$\geq 1 - 1/2^{100}$
Algorithm 3	Las Vegas	$O(n)$	$O(n^2)$	1

- Alg. 1 is Las Vegas because if it returns, it will always return a Majority Penguin (MP). The expected number of iterations is no more than 2. To see this, notice that the probability of returning an MP is at least $1/2$. In general (as we said in class), the number of independent draws that you expect to take before seeing an event that happens with probability p is $1/p$. Thus, the expected number of iterations before finding an MP is at most $1/(1/2) = 2$. Since isMajority takes time $O(n)$, the running time is $O(n)$.
  The worst-case running time is ∞, because in the worst case we would always find a non-majority penguin.
  The probability of returning an MP is 1, because the algorithm will always return an MP if it returns, and it will return with probability 1. (More precisely, the probability that the algorithm hasn't returned by iteration t is at most $1/2^t$, so the probability that it never returns is at most $\lim_{t\to\infty} 2^{-t} = 0$).
- Alg. 2 is Monte Carlo because it might return $P[0]$, which may not be an MP. The number of iterations is at most $100 = O(1)$, and each iteration takes time $O(n)$, so the worst-case running time is $O(n)$. The expected running time is also $\Theta(n)$, since it's at most the worst-case running time, and at least n (the time to perform one iteration).
  The success probability (on a worst-case input) is at least $1 - 1/2^{100}$, because the probability of finding an MP on a single draw is at least $1/2$, so the probability that 100 random draws fail to find an MP is at most $1/2^{100}$.
- Alg. 3 is Las Vegas because it iterates over all the penguins, and so it will eventually hit a majority penguin, which it will return. Thus it is always correct.
  The expected number of iterations is $O(1)$, just like for Alg. 1. One way to see this is to compare the situation to that of Alg. 1. Let A be the majority species. The expected number of iterations in Alg. 3 is at most the number of iterations of Alg. 1. Intuitively, this is true because the probability of seeing t non-A penguins in a row is *less* likely in Alg. 3, since in Alg. 1 we can re-draw a non-A once we've looked at it. So since the expected number of iterations of Alg. 1 was $O(1)$, the same is true for Alg. 3.

Algorithm 1: findMajorityPenguin1

```
Input: A population P of n penguins
while true do
    Choose a random p ∈ P;
    if isMajority(p) then
        return p;
```

Algorithm 2: findMajorityPenguin2

```
Input: A population P of n penguins
for 100 iterations do
    Choose a random p ∈ P;
    if isMajority(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 Θ(n) to put the n penguins in a random order */
for p ∈ P do
    if isMajority(p) then
        return p;
```

Algorithm 4: isMajority

```
Input: A population P of n penguins and a penguin p ∈ P
Output: True if p is a member of a majority species
count ← 0;
for q ∈ 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

We can make this formal as follows (this is not required for credit on this exercise). Let $X(m_1, m_2)$ be the number of iterations until we draw an A penguin, given that m_1 penguins of species A are left and m_2 penguins of other species are left. We have:

$$\begin{aligned}\mathbb{E}X(m_1, m_2) &= \frac{m_1}{m_1 + m_2} + \frac{m_2}{m_1 + m_2}(1 + \mathbb{E}X(m_1, m_2 - 1)) \\ &\leq \frac{m_1}{m_1 + m_2} + \frac{m_2}{m_1 + m_2}(1 + \mathbb{E}X(m_1, m_2)).\end{aligned}$$

The first equality is from the definition of expection: with probability $m_1/(m_1 + m_2)$, we draw an A-penguin and we're done with 1 iteration. Otherwise, with probability $m_2/(m_1 + m_2)$, we draw a non-A penguin. Then we pay one iteration for that draw, and then the number of iterations in the future is $X(m_1, m_2 - 1)$, since we've removed one non-A penguin.

The second inequality follows because our expected time to draw a penguin of species A only goes up if we keep the number of A penguins the same but increase the number of non-A penguins. Solving for $\mathbb{E}X(m_1, m_2)$, we find

$$\mathbb{E}X(m_1, m_2) \leq \frac{m_1 + m_2}{m_1}.$$

Instantiating this with $m_1 > (m_1 + m_2)/2$ (aka, the A penguins have a strict majority), we see $\mathbb{E}(\text{num. iterations}) \leq 2$. Thus the expected running time is $O(n)$, as before.

The worst-case running time is $O(n^2)$, because in the worst case we will look at $\lceil n/2 \rceil - 1$ non-majority penguins before we meet any majority penguins, and to test each penguin takes time $O(n)$.

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.

   - **(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?

     – 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.*]

   - **(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.*]

     SOLUTION:

     (a) bucketSort works, quickSort doesn't, mergeSort1 doesn't work and mergeSort2 does work.

     (b) The special thing is that these algorithms are **stable.** That is, if we have two items x and y with the same keys, and x comes before y in the original list, then x comes before y after the sort has been run. As we saw in class, this was crucial for the correctness proof of radixSort: once we had sorted on the first digit for example, we needed that work not to be un-done when we sorted on the second digit. More precisely, suppose that we are radixSorting $[12, 13]$. First we sort on the LSD, and the array doesn't change. Then we sort on the MSD, which is 1 for both 12 and 13. If the sort isn't stable, then we might switch the order of the elements in the case of a tie, and end up with $[13, 12]$ rather than $[12, 13]$.

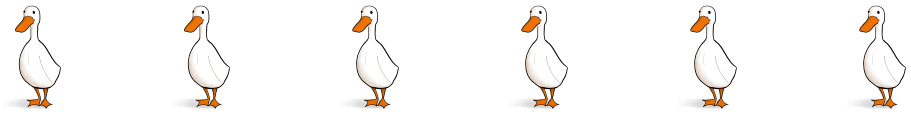
Problems

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 $n-1$.

[**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.*]

SOLUTION:

(a) We can use a variant of the algorithm which we used for QuickSort to partition elements in-place:

[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?

```
sortDucks( ducks ):
   i = 0
   # first, go through all the ducks and put the left-leaning ones on the left.
   for j = 0, ..., n-1:
      if poll(j) == "left":
          swap( i, j )
          i += 1
   # next, go through and put the center-leaning
   # ducks to the left of the right-leaning ones.
   for j = i, ..., n-1:
       if poll(j) == "center":
          swap( i, j )
          i += 1
   # now we should be all done!
```

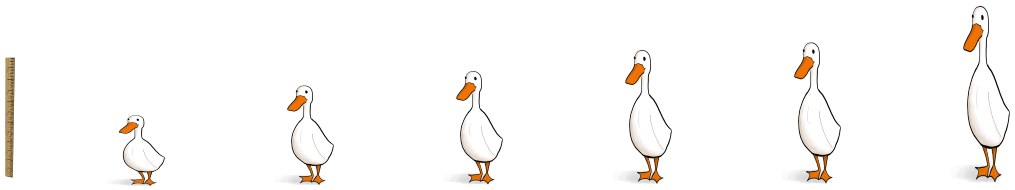
In the first loop, the algorithm puts the left-leaning ducks on the left and the rest of the ducks on the right. In the second loop, it separates the centrist ducks from the right-leaning ducks. We keep two counters, i and j. In the first loop, everything to the left of i is left-leaning, and there are no lefty ducks between i and j. When the second loop starts, all the lefty ducks are to the left of i. In the middle of the second loop, all of the ducks to the left of i are either lefty or centrist (and they are sorted), and none of the ducks between i and j are lefty or centrist. At the end of the day, the only ducks to the right of i are the right-wing ducks, and the ducks are sorted.

(b) **Hot dog buns and memory:** This takes two passes through the ducks, and in each pass, for each j, it does two operations: a `poll` operation and a `swap` operation. Thus, the total cost is at most $4n$ stale hot dog buns. Moreover, we only need to remember i and j, so we just need 2 integers worth of memory.

Correctness: To see that it works, consider what happens in the first for loop. We'll maintain the loop invariant that ducks $0, \ldots, i-1$ are left-leaning, and none of ducks $i, \ldots, j-1$ are left-leaning . For the base case, when $i = j = 0$, this is trivially true. Now assume it's true for $j-1$. We advance j until the j'th duck is left-leaning, and we swap it with the duck in the i'th position. By assumption, duck i was not left-leaning, so it is true that none of the ducks $i+1, \ldots, j$ are left-leaning. Also since duck j (before we swapped) was left-leaning, we now have ducks $0, \ldots, i$ are left-leaning. After incrementing i and j, this establishes the loop invariant for the next round.

Thus, at the end of the first loop, ducks $0, \ldots, i-1$ are left-leaning, and there are no left-leaning ducks left among $i, \ldots, n-1$. The logic for the second loop is exactly the same, and it puts all of the centrist ducks to the left of the right-leaning ducks. Thus, at the end of the day the ducks are sorted.

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 forgot to bring a piece of paper, so you can only store up to seven integers in $\{0, \ldots, n-1\}$ at a time. And, 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.*]

SOLUTION:

(a) This is an application of binary search:

```
findDuck( ducks ):
    if len( ducks ) == 0:
        return "no such duck."
    j = floor( len(ducks)/2 )
    if compareToStick( j ) == "the same":
        return ducks[j]
    else if compareToStick( j ) == "shorter":
        return findDuck( ducks[j+1:] )
    else if compareToStick( j ) == "taller":
        return findDuck( ducks[:j] )
```

That is, at each step we divide the line in half. If the current duck is too short, we recurse on the taller (right) half; if the current duck is too tall, we recurse on the shorter (left) half. Since we divide the input in half each time, the total number of calls to `findDuck` is $O(\log(n))$, and each call uses only one call to `compareToStick`, so the total number of stale hot dog buns used is $O(\log(n))$.

(b) We use the decision tree method. Any algorithm in this model works by comparing ducks to the stick, and thus can be written as a tri-nary decision tree: at each step, there are one of three possible answers, bigger, smaller, or the same. There are $n+1$ possible outcomes (each duck could be the right height, or none of them could be) so the decision tree has at least $n+1$ leaves. Thus it has height at least $\log_3(n+1) = \Omega(\log(n))$.

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 $O(\log(n))$.
- `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 $O(\log(n))$.

Above, n is the number of items stored 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.*]

SOLUTION: Suppose towards a contradiction that the untrustworthy goose is correct. Then consider the following sorting algorithm:

```
def gooseSort(A):
    G = gooseTree()
    for a in A:
        G.insert(a)
    print inOrderTraversal(G)
```

This algorithm runs in time $O(n)$, since it takes time $O(1)$ to insert each item from A into the gooseTree G, and there are n items in A; also `inOrderTraversal` runs in time $O(n)$. But then `gooseSort` also sorts an array of length n in time $O(n)$! Indeed, we saw in class that `inOrderTraversal` will output the elements stored in any BST (and in particular in a gooseTree) in sorted order. But then `gooseSort` is a sorting algorithm that runs in time $O(n)$ on arbitrary comparable objects, which we saw in class does not exist. This is a contradiction, so the goose must be wrong.

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}
```