Exercises

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

1. **(2 pt.)** Siggi and Ollie are playing the following game of chance. Siggi will roll a fair six-sided die. If Siggi rolls a 1, 2, 3, or 4, then Ollie will flip a fair coin. If the coin comes up heads, Siggi pays Ollie a dollar. Otherwise, Ollie pays Siggi a dollar. On the other hand, if Siggi rolls a 5 or a 6, then Ollie pays Siggi a dollar immediately.

   (a) How much money does Siggi make in expectation? How much money does Ollie make in expectation?

   (b) What is the probability that Ollie makes money? What is the probability that Siggi makes money?

   (c) What is the probability that both Ollie and Siggi make money?

   (d) Suppose you know that Siggi made money. Conditional on that event, what is the probability that Siggi rolled a 1?

   SOLUTION:

   (a) Siggi makes 1/3 dollars in expectation, and Ollie makes $-1/3$ dollars (aka loses 1/3 dollars). To see this, we compute:

   $$\mathbb{E}[O] = \frac{2}{3} \cdot \left(\frac{1}{2} \cdot (-1) + \frac{1}{2} \cdot (+1) \right) + \frac{1}{3} \cdot (-1) = -1/3,$$

   $$\mathbb{E}[S] = \frac{2}{3} \cdot \left(\frac{1}{2} \cdot (-1) + \frac{1}{2} \cdot (+1) \right) + \frac{1}{3} \cdot (+1) = 1/3,$$

   where O is Ollie's earnings and S is Siggi's earnings.

   (b) The probability that Ollie makes money is $\frac{2}{3} \cdot \frac{1}{2} = \frac{1}{3}$. The probability that Siggi makes money is $\frac{2}{3} \cdot \frac{1}{2} + \frac{1}{3} = \frac{2}{3}$.

   (c) The probability that both make money is zero. Either Ollie pays Siggi or Siggi pays Ollie.

   (d) The probability that Siggi made money *and* rolled a 1 is $\frac{1}{6} \cdot \frac{1}{2}$. (A 1/6 chance of the 1, and then a 1/2 chance that Siggi made money after rolling a 1). So using the definition of conditional probability,

   $$\mathbb{P}\{\text{S rolled 1} \mid \text{S made money}\} = \frac{\mathbb{P}\{\text{S made money and rolled 1}\}}{\mathbb{P}\{\text{S made money}\}} = \frac{\frac{1}{6} \cdot \frac{1}{2}}{\frac{2}{3}} = \frac{1}{8}.$$

2. **(4 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).

Consider the following task. The population of n tricky and trustworthy toads from HW1 is back. As before, you are guaranteed that there are strictly more than $n/2$ trustworthy toads in the population. Your goal is to find a single trustworthy toad. However, this time you've arrived on the island with a toad expert. The expert can determine whether a given toad is tricky or trustworthy, but she takes time $\Theta(n)$ to do it.

The algorithms given in Figure 1 all attempt to identify a single trustworthy toad. Fill in the chart below. You may use asymptotic notation for the running times; for the probability of returning a truthful toad, give the tightest bound that you can.

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

If it helps in writing up your solution, the LaTeXcode for the table (which you can copy and paste and fill in) has been reproduced at the end of this problem set. [**We are expecting: Your answers, and for each algorithm, a short informal justification of your answers.**]

Algorithm 1: findTrustworthyToad1

Input: A population of n toads
while *true* **do**
  Choose a random index $i \in \{0, \ldots, n-1\}$;
  Ask the expert if toad i is trustworthy;
  `/* Asking the expert takes time` $\Theta(n)$ `*/`
  if *the expert says toad i is trustworthy* **then**
    return *toad i*

Algorithm 2: findTrustworthyToad2

Input: A population of n toads
for *100 iterations* **do**
  Choose a random index $i \in \{0, \ldots, n-1\}$;
  Ask the expert if toad i is trustworthy;
  `/* Asking the expert takes time` $\Theta(n)$ `*/`
  if *the expert says toad i is trustworthy* **then**
    return *toad i*
return *toad 0*

Algorithm 3: findTrustworthyToad3

Input: A population of n toads
Put the toads in a random order ;
`/* Assume it takes time` $O(n)$ `to put the` n `toads in a random order */`
for $i = 0, \ldots, n-1$ **do**
  Ask the expert if toad i is trustworthy;
  `/* Asking the expert takes time` $\Theta(n)$ `*/`
  if *the expert says toad i is trustworthy* **then**
    return *toad i*

Figure 1: Three algorithms for finding a truthful toad

SOLUTION:

Algorithm	Monte Carlo or Las Vegas?	Expected running time	Worst-case running time	Probability of returning a truthful toad
Algorithm 1	LV	$\Theta(n)$	∞	1
Algorithm 2	MC	$\Theta(n)$	$\Theta(n)$	$1 - 1/2^{100}$
Algorithm 3	LV	$\Theta(n)$	$\Theta(n^2)$	1

- **Algorithm 1** is a Las Vegas algorithm because it will never return an untrustworthy toad. This also means that the success probability is 1.
  The expected number of iterations is no more than 2. To see this, notice that the probability of choosing a tricky toad at random is at most $1/2$. In general (as we proved in class), the number of independent draws you expect to take before seeing an event that happens with probability p is $1/p$. Thus, the expected number of iterations is at most $1/(1/2) = 2$. Thus, the total runtime is $\Theta(n)$, since it takes $\Theta(n)$ time to ask the expert.
  The worst-case runtime is infinite, because an adversary fixing the randomness would always choose untrustworthy toads to check.
- **Algorithm 2** is a Monte Carlo algorithm because it may return an untrustworthy toad. The number of iterations is no more than 100, and at least 1, hence $\Theta(1)$. Thus, the expected runtime is $\Theta(n)$, since for each of $\Theta(1)$ rounds, we ask the expert who takes $\Theta(n)$ time.
  The worst-case runtime is also $\Theta(n)$, because in the worst case the algorithm will run for $100 = \Theta(1)$ iterations and then return Toad 0. Each iteration requires $\Theta(n)$ time to ask the expert.
  The success probability (on worst-case input) is at least $1 - 1/2^{100}$, because the probability that we fail on a trial is at most $1/2$, and we do 100 independent trials.
- **Algorithm 3** is a Las Vegas algorithm because it iterates over all the toads, and eventually it will hit a trustworthy one. Thus, the success probability is 1.
  The expected number of iterations is still $O(1)$. To see this, let's compare the situation to that of **Algorithm 1**. The expected number of iterations of **Algorithm 3** is at most the expected number of iterations in **Algorithm 1**. Intuitively, this is true because the probability of seeing t tricky toads in a row is *less* likely in Algorithm 3, since we can't re-draw a tricky toad once we've already looked at it.
  To see this formally, let $X(m_1, m_2)$ be the number of iterations until we draw a truthful toad, given that there are m_1 truthful toads and m_2 tricky toads. Then

$$\mathbb{E}X(m_1, m_2) = \frac{m_1}{m_1 + m_2} + \frac{m_2}{m_1 + m_2}\left(1 + \mathbb{E}X(m_1, m_2 - 1)\right) \geq \frac{m_1}{m_1 + m_2} + \frac{m_2}{m_1 + m_2}\left(1 + \mathbb{E}X(m_1, m_2)\right).$$

  The first equality is from the definition of expectation: with probability $m_1/(m_1 + m_2)$, we draw a truthful toad first and we're done with 1 iteration. Otherwise, with probability $m_2/(m_1 + m_2)$, we draw a tricky toad. Then we definitely pay one iteration for that draw, and the number of iterations in the future is $X(m_1, m_2 - 1)$, since we've removed one tricky toad.
  The inequality follows because our expected time to drawing a truthful toad only goes down if we decrease the number of tricky toads in the population but keep the number of truthful toads. 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 $m_1 > (m_1 + m_2)/2$ truthful toads, we see

$$\mathbb{E}(\text{number of iterations}) \leq 2.$$

Then the total expected runtime is $\Theta(n)$ as before.
The worst-case runtime in $\Theta(n^2)$, because in the worst case we will look at $\lceil n/2 \rceil - 1$ tricky toads before we meet any truthful toads, and to test each tricky toad takes time $\Theta(n)$.

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. Modify the code for radixSort to use each one of these four algorithms as an inner sorting algorithm. Which ones work?
     - Does using bucketSort work correctly?
     - Does using quickSort work correctly?
     - Does using mergeSort (with merge1) work correctly?
     - Does using mergeSort (with merge2) 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 matter? (You may wish to play around with `HW3.ipynb` to "debug" the incorrect cases.)[1]

     Make sure that the algorithms that you observed to work do have the property, and that those that you observed not to work don't have the property.

     [**We are expecting: A clear definition of the special property and a short but convincing explanation of why it matters. You do not need to justify why each of the algorithms do or do not have the property but you should understand why they do or do not.**]

     SOLUTION:

     (a) bucketSort works, quickSort doesn't, mergeSort1 doesn't work and mergeSort 2 does work.

     (b) The special things 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.

---

[1] **Note:** Yes, we talked about this a bit in class—that's why it's an exercise and not a problem!

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.

---

1. **(4 pt.)** Suppose that n flamingos are standing in a line.

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

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

However, in order to do either operation, you need to pay the flamingos to co-operate: each operation costs one brine shrimp.[2] 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[3] integers between 0 and $n-1$ at a time.

Design an algorithm to sort the flamingos which costs $O(n)$ brine shrimp, and uses no extra memory other than storing at most seven[4] integers between 0 and $n-1$.

[We are expecting: Pseudocode with a short explanation of the idea of your algorithm; an informal justification that it works, uses only $O(n)$ brine shrimp and not too much memory.]

SOLUTION:

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

```
sortFlamingos( flamingos ):
   i = 0
   # first, go through all the flamingos and put the left-leaning ones on the left.
   for j = 0, ..., n-1:
      if poll(j) == "left":
          swap( i, j )
          i += 1
```

[2] According to Wikipedia, flamingos eat brine shrimp. Yum!

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

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

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

This takes two passes through the flamingos, 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 $4n$ brine shrimp. Moreover, we only need to remember i and j, so we just need 2 integers worth of memory.

To see that it works, consider what happens in the first for loop. We'll maintain the loop invariant that flamingos $0, \ldots, i-1$ are left-leaning, and none of flamingos $i, \ldots, j-1$ 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 flamingo is left-leaning, and we swap it with the flamingo in the i'th position. By assumption, flamingo i was not left-leaning, so it is true that none of the flamingos $i+1, \ldots, j$ are left-leaning. Also since flamingo j (before we swapped) was left-leaning, we now have flamingos $0, \ldots, i$ 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, flamingos $0, \ldots, i-1$ are left-leaning, and there are no left-leaning flamingos left among $i, \ldots, n-1$. The logic for the second loop is exactly the same, and it puts all of the centrist flamingos to the left of the right-leaning flamingos. Thus, at the end of the day the flamingos are sorted.

2. **(4 pt.)** Suppose that n flamingos 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 flamingo which is the same height as the stick, or else report that there is no such flamingo. The only operation you are allowed to do is `compareToStick(f)`, where f is a flamingo, which 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. As in Problem 1, 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 1, you have to pay a Flamingo a brine shrimp in order to compare it to the stick.

(a) **(1 pt.)** Give an algorithm which either finds a flamingo the same height as the stick, or else returns "No such flamingo," in the model above which uses $O(\log(n))$ brine shrimp.
[We are expecting: Pseudocode and an English description. You do not need to justify the correctness or brine shrimp usage.]

(b) **(3 pt.)** Prove that any algorithm in this model of computation must use $\Omega(\log(n))$ brine shrimp.
[We are expecting: A short but convincing argument.]

SOLUTION:

(a) This is just a simple application of binary search:

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

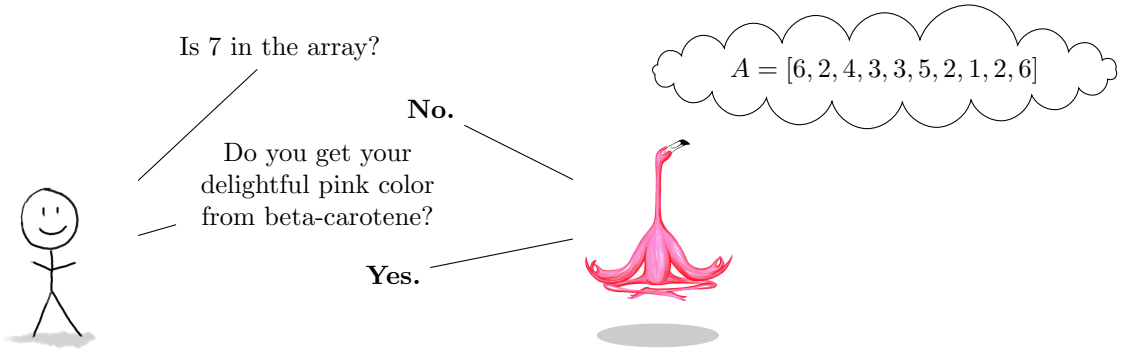
That is, at each step we divide the line in half. If the current flamingo is too short, we recurse on the taller (right) half; if the current flamingo is too tall, we recurse on the shorter (left) half. Since we divide the input in half each time, the total number of calls to `findFlamingo` is $\log(n)$,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

(b) We use the decision tree method. Any algorithm in this model works by comparing flamingos 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 flamingo 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 $\log_3(n+1) = \Omega(\log(n))$.

3. **(3 pt.)** A wise flamingo has knowledge of an array A of length n, so 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 the list, but you can ask the wise flamingo *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 "do molecular and morphological studies support a relationship between grebes and flamingos?"

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.[5] As usual, the wise flamingo charges one brine shimp per question.



(a) **(3 pt.)** Give a procedure to output a sorted version of A which uses $O(k \log(n))$ brine shrimp. You may assume that you know n and k, although this is not necessary.

[We are expecting: Pseudocode, with an English explanation of what it is doing, why it is correct, and why it only uses $O(k \log(n))$ brine shrimp.]

(b) **(1 bonus pt.)** Prove that any procedure to solve this problem must use $\Omega(k \log(n/k))$ brine shrimp.

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

SOLUTION:

(a) Our algorithm will be to do binary search on each of the numbers $\{1, \ldots, k\}$, to find out how many copies of it there are. Then we write down that many copies on our notepad.

```
for i = 1, ..., k:
    # binary search
    num_copies = flamingo_binary_search( 0, n-1, i )
    write down num_copies copies of i on your notepad

def flamingo_binary_search( low, high, k):
    mid = (low + high)/2
    ask the wise flamingo if there are exactly mid copies of k in the list.
    if so,
        return mid
    ask the wise flamingo if there are < mid copies of k in the list.
    if so,
        return flamingo_binary_search( low, mid, k )
    return flamingo_binary_search( mid, high, k )
```

(b) We use the decision tree method from class: imagine that each question we ask the flamingo is a node in the decision tree, and the leaves are the answers (the possible sorted lists). Consider the number of distinct sorted lists that the wise flamingo may be thinking of. Each list corresponds to a list of numbers $x_1, \ldots, x_k$, so that $\sum_i x_i = n$. Each such list means "There are x_1 copies of 0, x_2 copies of 1," etc. How many of these lists are there? It turns out there are

$$X = \binom{n+k-1}{n} = \binom{n+k-1}{k-1}$$

[5]Note that you don't have any ability to change the array A itself, you can only ask the wise flamingo about it.

of them. (This is the number of ways to put n balls into k bins). Thus, the decision tree has at least X leaves, and so the depth of the tree (number of questions asked) is at least $\log_2(X)$, which is

$$\log_2 \binom{n+k-1}{k-1} \geq \log_2 \left(\left(\frac{(n+k-1)}{k-1} \right)^{k-1} \right) = \Omega(k \log(n/k)).$$

In the last equality, we used the assumption that $n \geq k$.

Here is LaTeX code for the table in Exercise 3:

```
\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 truthful toad \\
\hline
\textbf{Algorithm 1} &  &  &  & \\
\hline
\textbf{Algorithm 2} &  &  &  & \\
\hline
\textbf{Algorithm 3} &  &  &  & \\
\hline
\end{tabular}
```