Problem 1: Word Representations

As described in class, RadixSort runs in $O(d(n+b))$ time, where $n$ is the number of elements, $b$ is the base, and $d$ ("digits") is the max length of the elements.

(a) (2 pts.) Suppose we want to use RadixSort to sort the following list of words $W$, each of which comes from the uppercase English alphabet, into alphabetical order:

$$W = [\text{HOW}, \text{VEXINGLY}, \text{QUICK}, \text{DAFT}, \text{ZEBRAS}, \text{JUMP}]$$

Before sorting, we left-pad all words with ! characters (assume ! comes before a alphabetically) to make them the same length.

Provide numerical values for the three variables $n$, $b$, and $d$. No explanation required.

Your answer goes here!

(b) (2 pts.) Now you happen to have the date (the day, month, year, all as base-10 numbers) that each word in $W$ was first published in an English dictionary. (Assume that all such dates are within the last 500 years.) You want to sort first by date (going forward in time), then use alphabetical ordering to break ties. You will do this by converting each of the original words in $W$ into words with date information (digits) prepended, appended, or inserted somewhere in the string, in an order that you choose.

Provide the string that you would use to represent the word ZEBRAS (first published, say, May 4, 1604) so that it will be correctly sorted by RadixSort for the given objective, regardless of what the dates for the other words in $W$ are. No explanation required.

Your answer goes here!

(c) (2 pts.) Hereafter we will leave the list $W$ behind. You have a new list $V$ representing the text of a document (just the words, all consisting only of uppercase English alphabet letters). $V$ has $n$ words in it in all, where $n > 2$, and there are at least two distinct words (but words may be repeated within the list).

You decide to save space and simply represent the first distinct word in $V$ with the binary value 0, the second distinct word with 1, the third distinct word with 10,
etc., continuing to increment by one each time in binary. All subsequent instances of a word you have already seen get the same number. So, for instance, the text [CS, ONE, SIXTY, ONE, CS, IS, FUN] would become [0, 1, 10, 1, 0, 11, 100].

Now you want to sort your list $V$ (an arbitrary list, not the example above!) using RadixSort. Before running RadixSort, you left-pad the strings with 0s as needed to make them equal-length.

Give the time complexity of RadixSort on this list, as a simplified big-O expression in terms of $n$. No explanation required.

Your answer goes here!

(d) (2 pts.) Now, assume the same setup as in the previous part, but because you don’t want to mess with binary conversions, you decide instead to represent the distinct words in $V$ with "one-hot" vectors (vectors with all 0’s except for a single ‘1’ in a position corresponding to a particular word). For example, the [CS, ONE, SIXTY, ONE, CS, IS, FUN] example from before has five distinct words, and it would become [10000, 01000, 00100, 01000, 10000, 00010, 00001].

As before, give the time complexity of RadixSort on this list, as a simplified big-O expression in terms of $n$. No explanation required.

Your answer goes here!

Problem 2: Proof Planning

Waverly has gotten into the habit of running proofs by Terry. Terry is preoccupied with a lot of work, but they still want to help Waverly as much as possible. Waverly has an estimate of how long it takes to run each proof by Terry. So she sends an array $A = [a_1, ..., a_n]$ of these time lengths (measured in minutes) to Terry. Terry has put aside a total of $L$ minutes to help Waverly today, and they want to go over as many proofs as possible.

(a) (6 pts.) Design an algorithm, $MaximumProofs$, that calculates the maximum number of proofs that fit within $L$ minutes and runs in time $O(n)$, where $n$ is the length of $A$. You may assume that all time lengths in $A$ are distinct. You may ignore divisibility issues (such as floors and ceilings) throughout this problem. You may use any algorithm we have learned in class as a function.

For this problem, please provide pseudocode (we don’t care about the exact format), or, if you prefer, code in a language like Python or C++. We think the solution, though not super complicated, has enough details that it’s worth writing out the algorithm.
rigorously. However, you do not need to prove correctness or justify the runtime – just the algorithm is sufficient.

**Input:** Array $A$ of $n$ distinct positive integers indicating the time it takes to verify each proof, and number $L \geq 0$ indicating how much time Terry has put aside.

**Output:** The maximum number of proofs that Terry can verify in $\leq L$ minutes.

Here are some sample inputs and the correct outputs below.

- For $A = [5, 1, 6, 2, 3]$ and $L = 3$, the correct output is 2.
- For $A = [20, 10, 30]$ and $L = 5$, the correct output is 0.
- For $A = [4, 1, 2, 3]$ and $L = 1000$, the correct output is 4.

Your answer goes here!

(b) (2 pts.) Now prove the runtime of the $MaximumProofs$ algorithm which you have just designed. That is, argue using a recurrence relation, or otherwise, that your algorithm satisfies the time bound of $O(n)$. (There is a way to do this pretty briefly.)

Your answer goes here!

**Problem 3: Randomized Algorithms**

In this exercise, we’ll explore different types of randomized algorithms. Recall from class 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.

(8 pts.) Fill in the following table. You may use asymptotic notation for the running times. For the probability of returning a majority element, give the **tightest bound** that you can given the information provided. No explanations required.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Monte Carlo or Las Vegas?</th>
<th>Expected running time</th>
<th>Worst-case running time</th>
<th>Probability of returning a majority element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithm 1: findMajorityElement1

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

Algorithm 2: findMajorityElement2

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

Algorithm 3: findMajorityElement3

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

Algorithm 4: isMajority

Input: A population $P$ of $n$ elements and a element $p \in P$
Output: True if $p$ is a member of a majority species
count $\leftarrow 0$;
for $q \in P$ do
    if $p = q$ then
        count $\leftarrow$ count + 1;
if count $> n/2$ then
    return True;
else
    return False;

Your answer goes here!
Problem 4: What Happens If I...?

Some algorithms have constants or properties that seem magical at best and arbitrary at worst, and it’s not clear why they’re there. In this problem, we’ll tweak a couple of the algorithms from this unit to see what breaks.

(a) (5 pts.) Show that if the median-of-medians part of the k-Select algorithm is modified to use groups of 3 instead of groups of 5, the argument from lecture that the overall k-Select algorithm is $O(n)$ does not work. To do this, first derive a recurrence $T(n)$ for the running time of k-Select, and then show that there is no way to choose a constant $c$ such that the recurrence is $O(n)$.

We recommend closely following what we did in Lecture 3, and seeing what breaks. You do not need to reiterate the entire argument from Lecture 3, but you do need to show where the differences arise. Also, to make your life easier:

- Assume that $n \geq 9$, and that all the list elements are distinct.
- Be very precise about the number of elements that are guaranteed to be less than the estimated median of medians, because the analysis is very tight. You do not need to argue about the equivalent greater-than case.
- You do not need to address the possible issue of the length of the list (or subsequent sublists) not being divisible by 3, which we also glossed over in lecture.
- You may feel free to leave the combined cost of the call to Partition and the work of finding the medians of the groups of 3 as $\Theta(n)$.
- Don’t worry about the base case of the inductive argument here, about choosing an $n_0$, or about $T(1)$.
- You don’t need to definitively prove that this change makes k-Select not $O(n)$. Just show that the argument used in lecture falls apart.

Your answer goes here!

(b) Brutus thinks the edge contraction step in Karger’s Algorithm is weird, and so he has invented his own variant that just deletes edges. Like Karger’s Algorithm, it operates on a connected graph with undirected edges and $n$ vertices, and we will assume $n$ is large (say, at least 10). Because Brutus is huge, he calls his new version Larger’s Algorithm. Unfortunately, Larger’s Algorithm currently only works for finding min cuts of a known exact size $k$. It works as follows:

- While there are still more than $k$ edges left, repeatedly choose and delete an edge chosen uniformly at random.
- Check whether the remaining $k$ edges are actually a cut in the original graph. If so, return that answer. Otherwise, start over.
Brutus checks that Larger’s Algorithm seems to work for $k = 2$. That is enough for him, and he returns to the swamp.

But now Indy is excited about the idea, because he came up with the following proof for $k = 2$, and he thinks it might be possible even to extend the idea.

- **Claim:** If we run Larger’s Algorithm over and over on a connected graph with $n$ vertices, it takes an expected $\frac{n(n-1)}{2}$ iterations to find a particular min-cut $M$ of size 2, just like for Karger’s Algorithm.

- **Proof:** In order for the algorithm to find $M$, both edges of $M$ must survive the entire procedure. The chance that both edges still exist after the first deletion is $\frac{n-2}{n}$. The chance that both edges still exist after the first and second deletions is $\frac{n-2}{n} \cdot \frac{n-3}{n-1}$. And so on down to the last three edges:

$$\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \cdots \cdot \frac{2}{4} \cdot \frac{1}{3}$$

But after we perform all cancellations, this reduces to $\frac{2}{n(n-1)}$. Therefore, again just as in the argument in class, it takes an expected $\frac{n(n-1)}{2}$ iterations to find $M$.

(i) (1 pt.) Indy’s proof does look awfully similar to the Karger’s Algorithm argument in class. But there is a **big** problem/misunderstanding with it. Briefly explain why his argument is incorrect. (Peeking ahead at the next part might help if you get stuck.)

(ii) (1 pt.) Indy can fix his argument by replacing all instances of $n$ with some other value $m$ that is a function of $n$. Now what is a (tight) big-O bound, in terms of $n$ (not $m$), for the number of iterations needed for Larger’s Algorithm to find a particular min cut of size 2? (Hint: How many edges can there be in the graph, as a big-O function?) No justification needed.

(iii) (1 pt.) With that fix in place, Indy tries to extend the proof to work for arbitrary values of $k$. Now his expression becomes

$$\frac{m-k}{m} \cdot \frac{m-k-1}{m-1} \cdot \cdots$$

He first checks what happens for $k = 3$. Now what is a tight big-O bound, in terms of $n$ (not $m$) and $k$, for the number of iterations needed for Larger’s Algorithm to find a particular min cut of size 3? No justification needed.
Problem 5: Distributed Median-Finding

Sisi the systems gator points out that we live in an age in which our data might not all fit on one machine. What if we want to find the median of a huge number of numbers?

Suppose that Sisi has \( n \) positive integers (each of which uses \( \log_2 n \) bits), which are spread out evenly across \( m \) “storage machines”. Each machine only has access to its own \( \frac{n}{m} \) integers. Moreover, there is a “central machine” that does not store any of the integers itself, but can send and receive messages to and from all of the storage machines. The storage machines cannot talk to each other. However, all machines can carry out whatever calculations might be needed for an algorithm (including using numbers with more than \( \log_2 n \) bits).

In this model, one round of computation consists of the following four phases, which do not overlap in time.

- **Phase 1**: Each storage machine performs some computations. All of these happen in parallel, so the time taken by this phase equals the maximum time used by any one storage machine – in this problem, we assume (somewhat unrealistically) that all storage machines that are performing the same calculations always take the same amount of time.

- **Phase 2**: Each storage machine sends its own message of size \( O(\log n) \) bits to the central machine; the \( O(\log n) \) limit reflects the fact that we can send only a constant number of values (each of which is \( O(\log n) \)). The central machine receives the messages. The sending occurs in parallel, but the receiving does not, so the time taken by this phase is determined by how long it takes for the central machine to receive and read all the messages.

- **Phase 3**: The central machine performs some computations.

- **Phase 4**: The central machine either returns a final answer, or sends a message of size \( O(\log n) \) bits (not necessarily the same message) to each storage machine. The receiving occurs in parallel, but the sending does not, so the time taken by this phase is determined by how long it takes the central machine to create and send all the messages.

The total running time of a round is the sum of the time taken by these four phases. The total running time of the algorithm is the sum of the running times of the rounds; they do not overlap in time.

For convenience, in this problem, assume that \( m \) and \( n \) are both odd, and that \( m \) evenly
divides \( n \). Also assume that the values are all distinct integers between 1 and \( n^2 \), and that the integers on each machine are already sorted.

(a) (6 pts.) Design and describe an algorithm that takes \( O\left(\frac{n}{m} \log n + m \log^2 n\right) \) time and finds the exact median. (Remember that \( \log^2 n \) is the same as \( (\log n)^2 \).)

This big-O expression might look daunting, but sometimes these runtime requirements provide useful hints! For instance, notice that you can factor out a \( \log n \) – what does that make you think of? But we think that you’ll know the nice approach to this problem when you see it, and then it’ll turn out to match that runtime, so it might be better not to worry about it up front.

We are expecting:

- A clear high-level description of the algorithm. You do not need to provide pseudocode. Your description can refer to known algorithms without providing the exact implementation details. We will basically be grading the description on whether you have the right overall idea, so in this part, don’t stress about off-by-one issues and that kind of thing.
- A justification of the running time: give and justify the number of rounds, and the running time for each phase of each round. (In the solution we have in mind, all rounds run pretty much the same way.) Use big-O expressions.
- You do not need to prove or justify correctness.

As a hint, we do not recommend that you try to directly adapt k-Select to this situation, or to otherwise have the storage machines send estimates of their medians. See if you can come up with a different strategy. Feel free to ask us for hints if you are stuck.

Your answer goes here!

(b) (2 pts.) Sisi can decide how many machines the system has. What value of \( m \) should Sisi choose, as a big-O function of \( n \), to make the final big-O expression in terms of \( n \) as small as possible? What is that final big-O expression? Show the math leading to your answers, but you do not otherwise need to explain.

(Hint: Try to make the terms equal each other. Feel free to ignore the fact that the big-O notation here might be hiding constant factors and other lower-order terms. Also notice that you can solve this without solving part (a)).

Your answer goes here!
Problem 6 (Coding): In-Place Partitioning

(NOTE: As mentioned in lecture, we had originally intended this problem to be about the Miller-Rabin Primality Test, which is a Monte Carlo algorithm. However, we felt that it would have ended up being just an implementation exercise, and the details of why it works get too far into number theory for this course. Also, working with large numbers in C++ is painful compared to Python. Apologies to anyone who was looking forward to finding huge primes!)

When we discussed the Partition steps of the k-Select and Quicksort algorithms in lecture, we spoke as if they create multiple separate new sublists. In reality, this would needlessly waste space. It is possible to partition in place efficiently, and this problem will illustrate the idea!

The Problem

Brutus likes to collect three kinds of objects from his swamp: logs, motorcraft (such as the speedboats he overturns), and rocks. Right now these objects are currently stored in a linear order along a river bank, from left to right. He has \( n \) such objects, and there is at least one of each type.

Brutus wants to arrange the objects such that all of the logs (denoted \( L \)) are at the left end of the line, all of the rocks (denoted \( R \)) are at the right end of the line, and all of the motorcraft (denoted \( M \)) are in the middle. (For your convenience, \( L, M, R \) conveniently match “L”eft, “M”iddle, “R”ight.) Put another way, Brutus wants all \( L \)s to come before all \( M \)s, which in turn must come before all \( R \)s. (All objects of a given type are indistinguishable – that is, your sort does not need to be stable.)

Brutus asked Terry to accomplish this task for him, and then swam off to look for more objects. But Terry is not strong enough to move the objects. The best they can do is use a teleporter that they invented, the Swamp Swapper, that instantly swaps the positions of any two objects. However, this is experimental technology, and Terry thinks it can only be used \( 3n \) times, where \( n \) is the number of objects. Moreover, if Terry takes too long overall looking at the objects, Brutus will come back and will be angry that they are not fully sorted.

Your Task

Implement an algorithm \texttt{SortCollection} that takes in an array \( A \) of characters, each of which is \( L, M, \) or \( R \), and sorts the list in place while obeying some special rules:

- Your code can only access the elements of \( A \) by calling the Access method which we provide in the starter code. It takes an index as input and returns the identity of the character (e.g., \( L \)) at that index.
• Your code can only alter $A$ by calling the Swap method which we provide in the starter code. It takes two indices as input and swaps the elements at those indices, and returns nothing.

• Your code should return $A$ when you are done sorting. (This makes it easier for us to grade)

• The Access and Swap methods count how many times they are called. On a test case of size $n$, if you try to call Access more than $6n$ times or Swap more than $3n$ times, this will cause your function to return $A$ immediately (thereby stopping your code).

• Your code is allowed to (and will have to!) maintain some variables, but it cannot try to store parts of the list in its own memory. That is, keeping track of a variable $i$ and checking whether the $i$-th element of the list is an L, for example, is fine. Storing the first half of $A$ in a new array that you create is not.

• You cannot modify the Access or Swap methods or do anything else that is against the spirit of solving the problem in the intended way (e.g., calling a sorting function). We will be looking at code to check this. (As usual, though, you will not be graded on coding style.) If you have any doubts, please check with us.

We are aware of solutions that keep track of only a very small number of variables and do not come close to the limits allowed for the Access and Swap functions.

**Testing Details**

• For each test case, $n$ is between 3 and 1000.

• Each test case includes at least one each of L, M, and R.