This next problem set explores randomized data structures and the mathematical techniques useful in analyzing them. By the time you've finished this problem set, you'll have a much deeper appreciation for just how clever and powerful these data structures can be!

Due Tuesday, May 3rd at the start of lecture
**Problem One: Cardinality Estimation**

In this problem, you’ll design and analyze a cardinality estimator that works on a different principle than the HyperLogLog estimator we saw in lecture. For the purposes of this problem, let’s assume our elements are drawn from the set \( \mathcal{U} \) and that the true cardinality of the set seen is \( n \).

Here’s an initial data structure for cardinality estimation. Choose a hash function \( h \) uniformly at random from a family of 2-independent hash functions \( \mathcal{H} \), where every function in \( \mathcal{H} \) maps \( \mathcal{U} \) to the open interval of real numbers \((0, 1)\). (Hashing to a uniformly-random real number poses some theoretical challenges; in practice, you’d use a slightly different strategy. For the purposes of this problem, assume this is possible.)

Our data structure works by hashing the elements it sees using \( h \) and doing some internal bookkeeping to keep track of the \( k \)th-smallest hash code out of all the hash codes seen so far, ignoring duplicates. The fact that we ignore duplicate hash codes is important; we’d like it to be the case that if we call see(\( x \)) multiple times, it has the same effect as just calling see(\( x \)) a single time. (The fancy term for this is that the see operation is idempotent.) We’ll implement estimate() by returning the value \( \hat{n} = \frac{k}{h_k} \), where \( h_k \) denotes the \( k \)th smallest hash code seen.

i. Explain, intuitively, why \( \hat{n} \) is a somewhat reasonable guess for the actual number of elements.

Let \( \varepsilon \in (0, 1) \) be some accuracy parameter that’s provided to us.

ii. Prove that \( \Pr[\hat{n} \geq (1+\varepsilon)n] \leq \frac{2}{\varepsilon^2} \). This shows that by tuning \( k \), we can make it unlikely that we overestimate the true value of \( n \).

Use the techniques we covered in class: use indicator variables and some sort of concentration inequality. What has to happen for the estimate \( \hat{n} \) to be too large? As a reminder, your hash function is only assumed to be 2-independent, so you can’t assume it behaves like a truly random function and can only use properties of 2-independent hash functions.

As a hint, \( \hat{n} \) is **not** an unbiased estimator and computing \( E[\hat{n}] \) is extremely challenging – as in, we’re not sure how to do it! See if you can solve this problem without computing \( E[\hat{n}] \).

Using a proof analogous to the one you did in part (ii) of this problem, we can also prove that

\[
\Pr[\hat{n} \leq (1-\varepsilon)n] \leq \frac{2}{\varepsilon^2}.
\]

The proof is very similar to the one you did in part (ii), so we won’t ask you to write this one up. However, these two bounds collectively imply that by tuning \( k \), you can make it fairly likely that you get an estimate within \( \pm \varepsilon n \) of the true value! All that’s left to do now is to tune our confidence in our answer.

iii. Using the above data structure as a starting point, design a cardinality estimator with tunable parameters \( \varepsilon \in (0, 1) \) and \( \delta \in (0, 1) \) such that

\begin{itemize}
  \item see(x) takes time \( O(poly(\varepsilon^{-1}, \log \delta^{-1})) \), where \( poly(x, y) \) denotes “something bounded from above by a polynomial in \( x \) and \( y \),” such as \( x^3y + y^2\log x \);
  \item estimate() takes time \( O(poly(\log \delta^{-1})) \), and if \( C \) denotes the estimate returned this way, then
    \[ \Pr[ |C - n| > \varepsilon n ] < \delta; \]
  \item the total space usage is \( O(poly(\varepsilon^{-1}, \log \delta^{-1})) \).
\end{itemize}

You’ve just built a tunable cardinality estimator that just needs 2-independent hash functions. Nicely done!
Problem Two: Cuckoo Phase Transitions

In lecture, we discussed vanilla cuckoo hashing (one table of \( m \) elements with two hash functions) and saw how the performance degraded when the load factor approached 50%. We then discussed two strategies for improving the space usage:

- **Blocked cuckoo hashing**: Make a table of \( m / b \) slots, where each slot can hold \( b \geq 1 \) items.
- **\( d \)-ary cuckoo hashing**: Use \( d \geq 2 \) hash functions to select table locations.

A \((b, d)\) cuckoo hash table is one that combines these two strategies. Specifically, the table will be subdivided into \( m / b \) blocks of size \( b \), and we’ll have \( d \geq 2 \) hash functions indicating where an item may be placed. So, for example, vanilla cuckoo hashing is \((1, 2)\) cuckoo hashing, \((1, 3)\) cuckoo hashing is \(3\)-ary cuckoo hashing, and \((2, 2)\) cuckoo hashing is blocked cuckoo hashing with \( b = 2 \).

Your task is to determine, empirically, what the maximum load factor is for \((b, d)\) cuckoo hash tables for different choices of \( b \) and \( d \). Specifically, write simulation code in your Programming Language of Choice to fill in the following table with the maximum value of \( \alpha \) for which, empirically, the probability that \( \alpha m \) items can be inserted into a \((b, d)\) cuckoo hash table is at least 99%.

<table>
<thead>
<tr>
<th></th>
<th>( d = 2 )</th>
<th>( d = 3 )</th>
<th>( d = 4 )</th>
<th>( d = 5 )</th>
<th>( d = 6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b = 1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b = 2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b = 3 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b = 4 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b = 5 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You can code this up however you’d like, provided that the following are true:

- Your implementation of cuckoo hashing is your own – that is, you’re not using an existing library that implements cuckoo hashing.
- Your code obeys our normal coding conventions – it’s well-commented, easy to read, etc.
- Your code must be able to run in terminal mode on the myth machines.
- We should be able to execute your code by running this exact command:

  ```
  make && ./generate-table
  ```
- Your code outputs the final table of results to stdout in some format that a well-meaning member of the course staff could interpret without too much difficulty.
- It takes less than thirty minutes to generate the table.

There are many design decisions you’ll need to consider here when running these experiments, and we’re going to leave them to you to decide. You should document your design decisions in the file DESIGN.txt that you’ll submit alongside the rest of your submission. In particular, please address the following:

- What value of \( m \) did you pick? Why?
- How did you come up with the hash codes for the items in the table? Did you generate them randomly, or did you use a known hash function (e.g. Jenkins, shift-add-XOR, etc.)? Why?
- What procedure did you use to displace items from the table? How did you determine that an insertion failed? Why?

Check /usr/class/cs166/assignments/a3 for some sample Makefiles you can use, along with a DESIGN.txt doc you can use as a starting point.