Problem Set Five: Randomized Data Structures

This final problem set of the quarter 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 Thursday, May 28 at 2:30PM Pacific time.
Problem One: Cardinality Estimation

A *cardinality estimator* is a data structure that supports the following two operations:

- `ds.see(x)`, which records that the value `x` has been seen; and
- `ds.estimate()`, which returns an estimate of the number of *distinct* values we’ve seen.

Imagine that we are given a data stream consisting of elements drawn from some universe \( \mathcal{U} \). Not all elements of \( \mathcal{U} \) will necessarily be present in the stream, and some elements of \( \mathcal{U} \) may appear multiple times. We’ll denote the number of distinct elements in the stream as \( F_0 \).

Here’s an initial data structure for cardinality estimation. We’ll begin by choosing 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 \((0, 1)\). *(We’re aware that hashing to a uniformly-random real number poses some theoretical challenges; in practice, you’d use a slightly different strategy, but 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 \( \hat{F} = \frac{k}{h_k} \), where \( h_k \) denotes the \( k \)th smallest hash code seen.

### i. Explain, intuitively, why \( \hat{F} \) is likely to be close to the number of distinct elements.

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

### ii. Prove that \( \Pr[\hat{F} \geq (1+\epsilon)F_0] \leq \frac{2}{k^2\epsilon^2} \). This shows that by tuning \( k \), we can make it unlikely that we overestimate the true value of \( F_0 \).

As a hint, use the techniques we covered in class: use indicator variables and some sort of concentration inequality. What has to happen for the estimate \( \hat{F} \) 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 the properties of 2-independent hash functions.

Using a proof analogous to the one you did in part (ii) of this problem, we can also prove that \( \Pr[\hat{F} \leq (1-\epsilon)F_0] \leq \frac{2}{k^2\epsilon^2} \).

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 \( \pm \epsilon F_0 \) 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 \( \epsilon \in (0, 1) \) and \( \delta \in (0, 1) \) such that

- `see(x)` takes time \( \mathcal{O}(poly(\epsilon^{-1}, \log \delta^{-1})) \), where \( poly(x, y) \) denotes “something bounded from above by a polynomial in \( x \) and \( y \),” such as \( x^2y + y^2\log x \);
- `estimate()` takes time \( \mathcal{O}(poly(\log \delta^{-1})) \), and if \( C \) denotes the estimate returned this way, then \( \Pr[|C - F_0| > \epsilon F_0] < \delta \); and
  - the total space usage is \( \mathcal{O}(poly(\epsilon^{-1}, \log \delta^{-1})) \).

There are a number of really beautiful approaches out there for building cardinality estimators. Check out the impressive HyperLogLog estimator for an example of a totally different approach to solving this problem that’s used widely in practice!
Problem Two: Hashing IRL

Your task is to implement the following flavors of hash table:

- **Linear probing:** A popular open-addressing hashing strategy. Check [these slides from CS106B](#) for more information about linear probing.

- **Robin Hood hashing:** The modified version of linear probing that tries to distribute elements more “fairly” than linear probing. Again, check [these CS106B slides](#) for details.

- **Cuckoo hashing:** The dynamic perfect hashing scheme described in class.

We've provided a test harness that will test your hash tables with different load factors and different choices of hash functions. Each table cell contains three entries: the mean insert time (\(I\)), the mean time for a successful lookup (\(H\), for “hit”), and the mean time for an unsuccessful lookup (\(M\), for “miss”). Once you've finished implementing your solutions, turn on max optimization (\(-O\)fast \(-march=native\)), run our driver code, and submit a (brief) writeup answering the following questions:

i. The theory predicts that linear probing and cuckoo hashing degenerate rapidly beyond certain load factors. How accurate is that in practice?

ii. How does Robin Hood hashing compare to linear probing? Why do you think that is?

iii. In theory, cuckoo hashing requires much stronger classes of hash functions than the other types of hash tables we've covered. Do you see this in practice?

iv. In theory, cuckoo hashing’s performance rapidly degrades as the load factor approaches \(\alpha = \frac{1}{2}\). Do you see that in practice?

Starter files are available at `/usr/class/cs166/assignments/a5`. To receive full credit, your code should compile cleanly without warnings on the myth machines and have no memory errors. Some notes:

- Our driver code will provide the number of buckets to use as a parameter to the constructors of your hash table. **You should not resize your hash tables dynamically.** Our starter files will always leave at least a few slots empty in your tables, so, for example, you don’t need to handle the case where you have a linear probing hash table that’s at 100% capacity.

- The file comments for each of the hash tables contain information about specific implementation requirements. For example, we’d like you to implement deletions in Robin Hood hashing using backwards-shift deletion and deletions in linear probing tables using tombstone deletion.

- Hash functions are represented using the `HashFunction` type. If you have a `HashFunction` named `h`, you can invoke it by calling `h(key)`. The hash values are distributed over the range \([0, 2^{31}]\), so you will need to mod hash codes by your table sizes.

- You can assume that the keys you’re hashing will be nonnegative integers. Feel free to reserve negative integers as sentinel values.

- In Robin Hood hashing, remember that you can – and should – terminate searches early in many cases by looking at where the currently-scanned element is relative to where it should be.

- You are welcome to use C++ standard library types and functions, though the standard hash containers (`std::unordered_map`, `std::unordered_set`, etc.) are, understandably, off-limits.