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 Tuesday, May 28 at 2:30PM.

Remember that the take-home exam goes out on that Tuesday, so while you're allowed to use late days if you'd like, we would strongly advise against doing so!

Problem One: Final Details on Count Sketches (3 Points)

In our analysis of count sketches from lecture, we made the following simplification when determining the variance of our estimate:

$$\text{Var}\left[\sum_{j \neq i} \boldsymbol{a}_j s(x_i) s(x_j) X_j\right] = \sum_{j \neq i} \text{Var}\left[\boldsymbol{a}_j s(x_i) s(x_j) X_j\right]$$

In general, the variance of a sum of random variables is not the same as the sum of their variances. That only works in the case where all those random variables are ***pairwise uncorrelated***, as you saw on Problem Set Zero.

Prove that any two terms in the above summation are uncorrelated under the assumption that both s and h are drawn uniformly and independently from separate 2-independent families of hash functions.

As a refresher, two random variables X and Y are uncorrelated if $\text{E}[XY] = \text{E}[X]\text{E}[Y]$.

Problem Two: Cardinality Estimation (12 Points)

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 $\mathscr{U}$. Not all elements of $\mathscr{U}$ will necessarily be present in the stream, and some elements of $\mathscr{U}$ may appear multiple times. We'll denote the number of distinct elements in the stream as $\boldsymbol{F_0}$.

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 $\mathscr{H}$, where every function in $\mathscr{H}$ maps $\mathscr{U}$ to the open interval of real numbers (0, 1).

Our data structure works by hashing the elements it `see`s using h and doing some internal bookkeeping to keep track of the kth-smallest unique hash code seen so far. The fact that we're tracking *unique* 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 kth smallest hash code seen.

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

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

ii. Prove that $\Pr[\hat{F} > (1+\varepsilon)F_0] \leq \frac{2}{k\varepsilon^2}$. 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

$$\Pr[\hat{F} < (1-\varepsilon)F_0] \leq \frac{2}{k\varepsilon^2}.$$

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

- `see`(*x*) takes time $O(\log \varepsilon^{-1} \cdot \log \delta^{-1})$;
- `estimate`() takes time $O(\log \delta^{-1})$, and if we let C denote the estimate returned this way, then

$$\Pr[\ |C - F_0| > \varepsilon F_0\ ] < \delta; \text{ and}$$

the total space usage is $\Theta(\varepsilon^{-2} \log \delta^{-1})$.

As a reminder, $\log \varepsilon^{-p} = \Theta(\log \varepsilon^{-1})$ for any constant p.

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 Three: Hashing IRL (10 Points)

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

- ***Chained hashing:*** The standard hash table usually taught in CS106B/X, CS107, and CS161. Chances are you've implemented one of these before, so hopefully this will just be a warm-up.
- ***Second-choice hashing:*** This is a variation on chained hashing. You'll maintain two hash functions h_1 and h_2. When inserting a key x, compute $h_1(x)$ and $h_2(x)$ and insert x into whichever bucket is less loaded. To do a lookup, search for x both in the bucket given by $h_1(x)$ and $h_2(x)$.
- ***Linear probing:*** The open-addressing scheme described in class.
- ***Robin Hood hashing:*** The modified version of linear probing from lecture where elements closer to home are displaced during insertions to make room for elements far from home.
- ***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 (`-Ofast -march=native`), run our driver code, and submit a (brief) writeup answering the following questions:

- The theory predicts that linear probing and cuckoo hashing degenerate rapidly beyond a certain load factor. How accurate is that in practice?
- How does second-choice hashing compare to chained hashing? Why do you think that is?
- How does Robin Hood hashing compare to linear probing? Why do you think that is?
- 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?
- In theory, cuckoo hashing's performance rapidly degrades as the load factor approaches $\alpha = ½$. Do you see that in practice?

Starter files are available at `/usr/class/cs166/assignments/ps5`. 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.