CSE 442 HW1

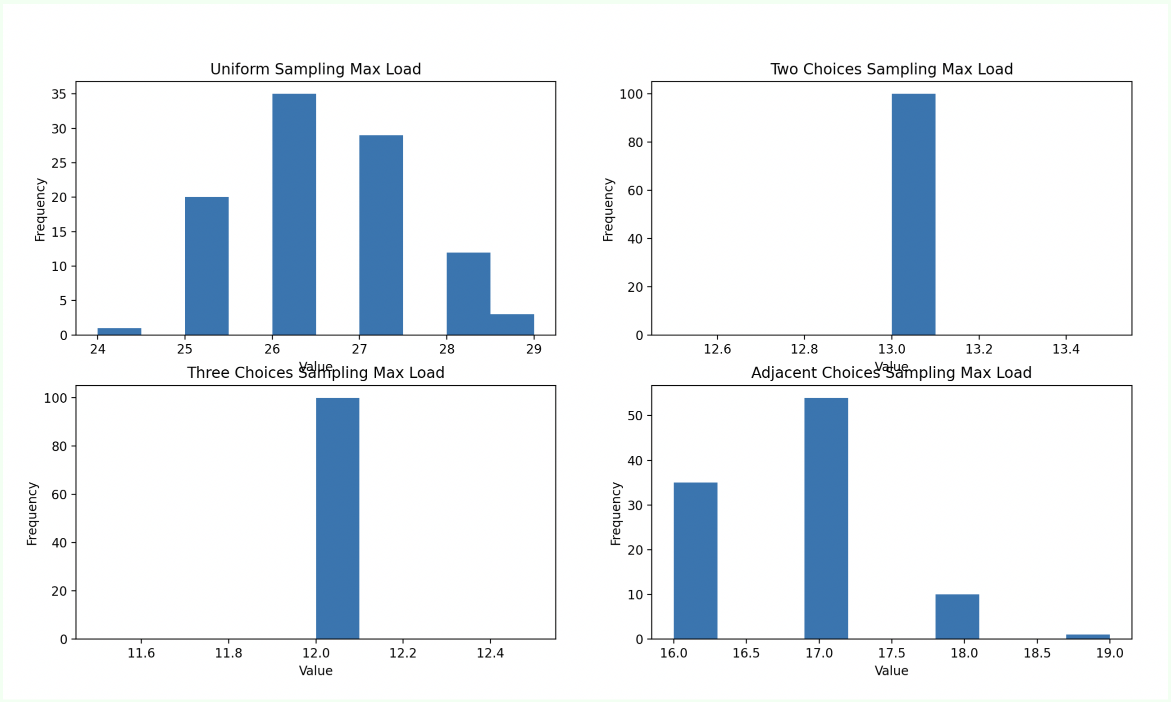
Sam Tacheny, Sophie Robertson

Monday 13th January, 2025

Question 1:

(a) Write code to simulate strategies 1-4. For each strategy, there should be a function which takes as input m, n, and outputs the maximum number of balls in any bin.

(b) Let m = 1, 000, 000, n = 100, 000 and simulate the four strategies 100 times. For each strategy, plot the histogram of the maximum loads. Discuss the pros and cons of each strategy.



Sampling one bin uniformly at random is the fastest by far. We don't have to compare the current load of any of the bins. Additionally, we don't have to worry about sampling without replacement. However, it also clearly has the largest discrepancies in max load comparing it to any of the other three strategies. The two choices approach is naturally faster than the three choices, but we found three choices to perform slightly better in terms of max load. The fourth, adjacent bin, strategy works better than sampling uniformly at random, but does not work better than selecting a second choice at random. We believe this is because by choosing the second bin as the adjacent one, we are removing the randomness that led to better load balancing.

(c) Recall the consistent hashing problem from lecture. Model consistent hashing as a balls-into bins problem, and discuss how you may use insights from this homework problem to augment the consistent hashing scheme discussed in lecture, and what tradeoffs different strategies will give.

To model the consistent hashing problem as a balls-into-bins problem, we consider the balls to be the M data points we want to cache, and the bins to be the N machine on which to cache them. We equate picking uniformly at random with computing a hash function over all bins. We use the strategies discussed in this problem to augment the scheme from lecture to include load balancing.

Consistent hashing considers two main goals, minimizing reallocation cost and an efficient way to lookup data points that are stored on our N machines. If we keep the circular representation of our hashes, and the same way of hashing our servers (with or without variance reduction), then the reallocation guarantee will be the same.

We propose introducing the two choices strategy to consistent hashing. We do this by having two separate hash functions, and as each data point is inserted into memory we compute both. This is equivalent to uniformly selecting two points on the circle at random, and consequentially two bins at random. We would then need to determine which bin these locations have mapped to, the same as in the original algorithm except done twice. We then access the current sizes of each bin (this must be stored and updated in some auxiliary data structure), and insert the element into the one with a lower count. Then, we update the size of that bin by one. It is possible that the two hash functions map to the same bin, so we are sampling with replacement, in which case it is a trivial solution of inserting the ball into that bin. At the time of a query, an element can be located by checking the 2 locations that the hash functions map to.

The original algorithm of using just one hash function is equivalent to strategy one, uniformly sampling at random.

If we were to implement 3 choices, we believe that we would not see as much improvement in load bearing compared to the extra computation required to do another hash function and check the extra bins.

The fourth strategy or adjacent choice, is harder to implement because there is no clear choice as to what the adjacent bin would be in the circular representation. We would compute one hash function and have to walk around the circle to find the next open spot that corresponds to a different bin. Then, there is no easy way to go back and access which bin the data point is on when the data point is next queried. This is just a less efficient implementation in a lot of ways.

(d) Some of these strategies are specific instances of a more general class of strategies. Imagine placing the n bins onto the vertices of a graph G. When a ball arrives, sample a uniformly random edge of this graph, and let B1, B2 be the vertices of this edge, then place the ball into whichever of B1 and B2 has fewer balls so far (breaking ties arbitrarily).

Which of these four strategies can be cast as an instance of this more general class? For each of these strategies, identify the associated graph.

The above description could align with strategy four, the adjacent bins strategy, if each node had exactly two neighbors. Here, each node represents a bin, and an edge between them represents them being adjacent. The graph would be one big cycle where each node, or bin, is connected to the bins adjacent in index. Thus, the cycle is in index order. We randomly sample some edge and decide between the two nodes that are connected by this edge, meaning they are adjacent.

A fully connected graph would represent the two choices strategy. We would sample a random edge, which has a uniform probability to connect any two random nodes. Both of these nodes would represent bins, and we would then check which bin had a lower volume of balls before inserting our current ball.

(e) Why might one graph perform better than another for this task?

In this construction, there will be no better result in load balancing than the complete fully connected graph described in part d. This is because this is the only instance which allows for uniform sampling across all pairs of bins. If the number of edges is decreased at all, then there is no longer an equal probability for any two pairs of nodes to be selected.

However, we can decrease the computational complexity while still retaining a good load balancing scheme. In a fully connected graph we sample from $\frac{n(n-1)}{2}$ different edges. Instead of connecting each node with every other node, we can connect it to some other node with probability p. This will reduce the number of edges by a factor of p, meaning there is less work required to randomly sample. This should be more efficient while performing worse load-balancing than the two choices approach, because we no longer choose a truly random second bin, but better than uniform choice and adjacent choices.

(f) Imagine you could design a graph for this problem, but each node could only have degree at most 3. What kind of graph might be good?

A good choice for a graph is one in which the maximum amount of nodes have degree 3, since this mimics the randomness present in strategy (2) where we use a complete graph. In other words, we want to create a random graph that is connected as much as possible.

One way to construct such a graph is to start with a cycle of vertices in a randomly indexed order. From there, we will choose pairs of non-adjacent vertices at random and connect them with an edge if both have degree less than 3. This ensures that (a) all nodes have degree 3 if there are an even number of bins, or (b) all but one node have degree 3 if there are an odd number of bins. This construction is more efficient than sampling all possible edges at random until the conditions above are met, since we essentially get two random edges for each node by the construction of the cycle.

Question 2: Count-min sketch

(a) Implement a data structure that for the algorithm described above.

(b) Find a general expression, as a function of n, for what elements of this dataset are heavy hitters.

The total number of elements in the stream is given by:

$$N = \sum_{i=1}^{n} i^2 + (n^2 - n) = \frac{n(n+1)(2n+1)}{6} + (n^2 - n) = \frac{2n^3 + 9n^2 - 5n}{6}$$

If $N \leq 100$ (e.g. $n \leq 5$), then all elements are heavy hitters since each occurs with frequency at least 1. Otherwise, if any exist, the heavy hitters are all integers $i \in \{j, \ldots, n\}$, where

$$j = \underset{\text{i} \in \{1,\ldots,n\}}{\min} \quad \text{such that} \quad j^2 \geq \frac{2n^3 + 9n^2 - 5n}{600} = \frac{N}{100}$$

Solving for j,

$$j = \lceil \sqrt{\frac{2n^3 + 9n^2 - 5n}{600}} \rceil$$

Since this expression grows faster than linear, we have $j > n$ for all $n > 295$, so there are no heavy hitters for these values of n. All together, the set of heavy-hitters as a function of n is given by:

$$HH(n) := \begin{cases} \{1, \ldots, n^2\} & 1 \leq n \leq 5 \\ \{\lceil \sqrt{\frac{2n^3+9n^2-5n}{600}} \rceil, \ldots, n\} & 6 \leq n \leq 295 \\ \emptyset & n \geq 296 \end{cases}$$

(c) For each of the 3 data streams, feed it into your count-min sketch, and compute the following quantities averaged over 10 trials.

1. The sketch's estimate of the frequency of the number 100
2. The sketch's estimate for the total number of heavy hitters in the stream

Does the order of the stream affect the estimated counts? Explain what the data says, and propose a plausible explanation.

Heavy first:

- On average, the frequency of the number 100 is 10082.
- There are an average of 43.4 heavy hitters in the stream.

Heavy last:

- On average, the frequency of the number 100 is 10082.
- There are an average of 43.4 heavy hitters in the stream.

Random:

- On average, the frequency of the number 100 is 10082.
- There are an average of 43.4 heavy hitters in the stream.

No, the order of the stream does not affect the estimated count since each counter is updated the same number of times, regardless of the order in which these updates happen.

(d) Implement the following conservative update optimization. When you are updating the five counters, only increment the subset of counters for which the current count is smallest (if a few are tied for smallest, increment all of those).

(e) Argue that, even with the conservative update rule, the count-min sketch never underestimates the count.

We can prove this claim by considering the following invariant for each number n read in the stream: all six counters for n are at least the true count for n over all data points previously read.

This is clearly true at initialization since we have read no data points and all counters are set to 0. Assume it is true at a given point, and consider reading the next number n. When running $Inc(n)$, we update the minimum of the six counters using the conservative update rule. Since this value is at least the true count (by our invariant), it stays the true count after the update. The counters that we do not increment must be at least one greater than the minimum counter, and therefore are also at least the true count after the update step.

(f) Repeat part (c) with conservative updates (including the discussion). If the outcome seems different, offer a plausible explanation

Heavy first:

- On average, the frequency of the number 100 is 10000.
- There are an average of 43.2 heavy hitters in the stream.

Heavy last:

- On average, the frequency of the number 100 is 10033.
- There are an average of 43.2 heavy hitters in the stream.

Random:

- On average, the frequency of the number 100 is 10000.
- There are an average of 43.2 heavy hitters in the stream.

In this case, the order of the stream does impact the estimated frequency counts. This can occur when multiple numbers increment the same counter in the count-min sketch.

As a toy example, consider a stream with m copies of a number n, and 6 other numbers $x_1, \ldots, x_6$ that, when inputted into the sketch, overlap with a unique counter for the number n but have all other counters independent of each other. If we put read all copies of n before the numbers x_i, then the conservative update rule will ensure that the counters for n are not updated when reading the x_i (since the other counters for the x_i will be 0). However, if we read all copies of the x_i before the copies of n, then the counters for n will all start at 1 instead of 0 and our frequency estimate will be 1 greater.

This toy example can explain why the frequency count for the number 100 is larger when reading the stream in a heavy-last order as opposed to a heavy-first order.