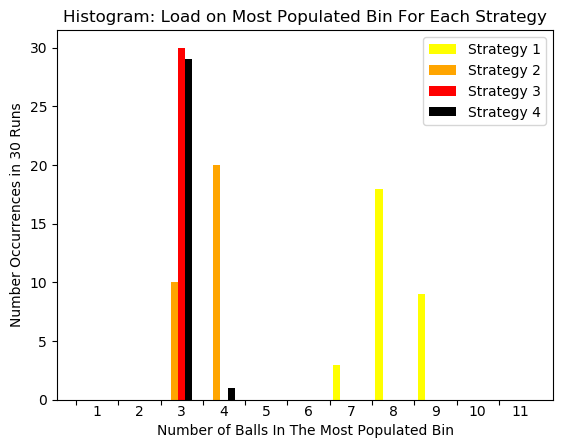
CS168 P1

Henry Lin

1. A. See code.py

B.



C.

Strategy 1 is the fastest to run because it just requires randomly choosing any bin. This does come with a cost though as you can observe the high relative collision rate in comparison to the other strategies. There is an improvement with the collision rate in Strategy 2, which is still fast, but both Strategy 3 and 4 seem to perform the best with few collisions on average compared to the others. Strategy 3 would be the slowest to run however, so Strategy 4 would be the sweet spot in terms of speed and number of collisions.

C. The N bins we are throwing into in our processes are equivalent to the N buckets in the hash table, and the N balls are equivalent to the N elements being hashed. Since good hash functions seem to be random, hashing N elements to N buckets in a hash table is the same as making a uniformly random bin choice for each of our N balls. X increases linearly with search time since the entire chain for a bucket may have to be traversed through in order to find an element.

D. If the length of chains is stored somewhere, insertion time is always O(1) because you can compare chain lengths for each strategy in O(1) time and put the element at the beginning of the linked list.

Strategy 2 is the equivalent of using 2 hash functions with the same range, Strategy 3 is the equivalent of using 3 hash functions with the same range, and Strategy 4 is the equivalent of using 2 hash functions where one function has a range covering half of the buckets, and one has a range covering the other half.

Search time will vary directly with X, as explained in part C. \*\* not sure if this is true bc in this case, Strategy 3 would be the best, but isn’t worse possible case having to search through all three chains anyway? Three short chains = 1 long chain?

1. A. See code.py

B. Since there are 87925 total elements in the data stream, any heavy hitter must appear at least 880 times. This means there are 21 heavy hitters because only elements occurring between 302 and 502 times inclusive occur at least 880 times.

C. It does not affect the estimated counts is because no matter the order of insertion, every relevant bucket in the table is incremented. Since the order of elements does not affect the hash function, the relevant bucket for a given element is always the same. Thus, when the same elements are fed in, the end counts will turn out to be equal.

|  |  |  |
| --- | --- | --- |
|  | Number Heavy Hitters Average | Frequency of 9050 Average |
| Forward | 403.6 | 3215.6 |
| Reverse | 403.6 | 3215.6 |
| Random | 403.6 | 3215.6 |

D. See code.py

E. The count-min sketch will never underestimate the count of a value because every instance of a value will be hashed to the same buckets consistently, and we increment every counter that has the minimum current count, rather than just one of the counters with the minimum count. Many times, the counters for a given value will have the same count. To see this, consider that they will all start at count 0, and the first occurrence of a given value will increment all of the relevant buckets to 1 since they have the same min count of 0, and this process will continue. This process will only differ when there are collisions, which only increases the count above the actual frequency. Therefore, there will never be an occurrence of underestimation.

F. It does affect the estimated counts. In this conservative implementation, not every relevant bucket is incremented and which bucket is incremented is contingent on the current count. This means that order of insertion will matter depending on which buckets fill up before others.

|  |  |  |
| --- | --- | --- |
|  | Number Heavy Hitters Average | Frequency of 9050 Average |
| Forward | 22.2 | 2864.9 |
| Reverse | 21.2 | 2500 |
| Random | 21.2 | 2500 |