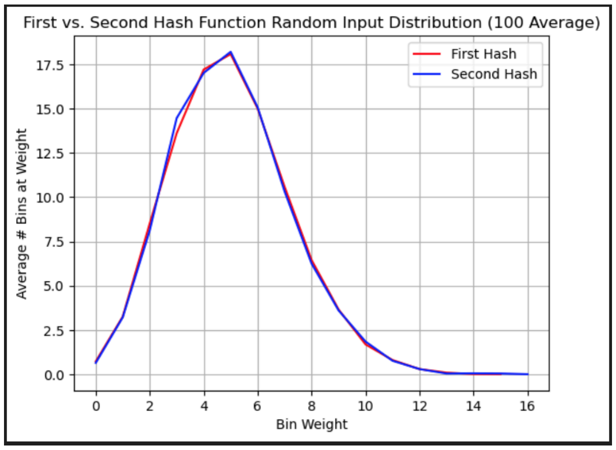
Dakota Barnes
CS 130A Algorithms
November 27,2023
1pm Friday Section

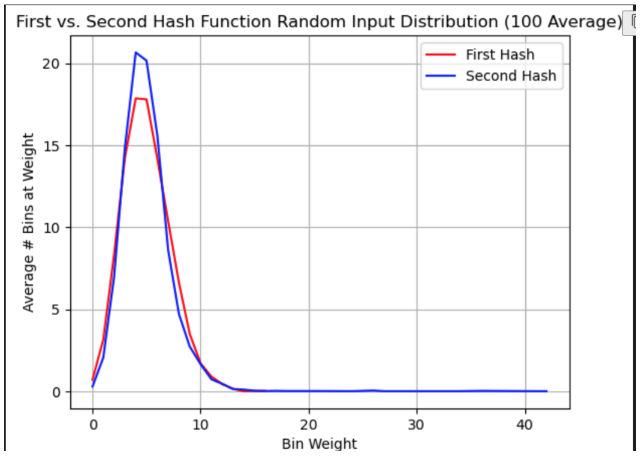
Bloom Filter Project

1. Designing Suitable Hash Function

I created two hash functions based on the instructions: First(Random Seed based) and Second(Prime Modulo Based). For 100 tests, I generated 2 new BloomFilters, one with the First Hash function, and the other with the Second Hash function with the same input data. The graphs display on the X axis the bin weight, or the number of items a bin contains. The Y axis displays the average number of bins with that specific bin weight across 100 tests.

Constants used for Visibility: $N = 1000000$, $n = 100$, $c = 1$, $m = 100$, $k = 5$





Graph #1: Average Random Input Graph #2: Average Linear Input

For Graph #1, the First and Second Hash Functions follow nearly the same bin Weight distribution curve when the input data is randomly generated. Notably, the largest Bin for the First Hash was 15 and the Second Hash being 16, and both Hash Functions had a minimum of 0. The majority of Bins had 5 items in it for both functions.

For Graph #2, the first and second Hash Functions consistently started to diverge in shape. Instead of randomly generated data, each datapoint was a factor of a number (in the above figure 2203). When I changed the factor number, the Second Hash Function would consistently have some outlier Bin Weight, in the range of 20-80. This means that with linear data, there is a low chance of a very filled bin, which is unideal for a Hash Function. The First Hash function consistently had a max bin weight of <16, and the curve always had a lower peak at a bin weight of 5 than the second Hash Function.

These graphs are based on test constants that I came up with, but the big takeaway is that when input data is correlated, the Second function with modulo can stack a bin very high with low probability. However, the second Function performs at a faster rate than the first function. This means that the First function performs more reliably and consistently with different data inputs, and with this test, is the preferred hash function for the Bloom Filter if speed is not a concern.

2. Implement a Bloom Filter

For my Bloom Filter, I have created a class that contains
Int m: The size of the Hash Table
Int k: The number of Hash Functions
Bool firstHash: Determines if first or second hash Function is used
Vector<pair<bool, int> T: Hash Table with each index containing the visited bool, and the number of unique elements at the index/bin. The count is mainly used for test purposes and can be removed when testing is complete for space/time efficiency, and T can be vector<int>.
Constructor: Initialized all variables and sets the Hash Table to visited = false and count = 0 for all elements of size m.
add(x): The function that adds a specific element to the Hash Table. In a loop for the number of Hash Functions used, the function calls the FirstHashFunction or SecondHashFunction (Based on Bool firstHash) to determine each index(curLoc) for the specified element and Hash Function number. At the curLoc index, the visited flag is set to true, and 1 is added to the curLoc bin count.

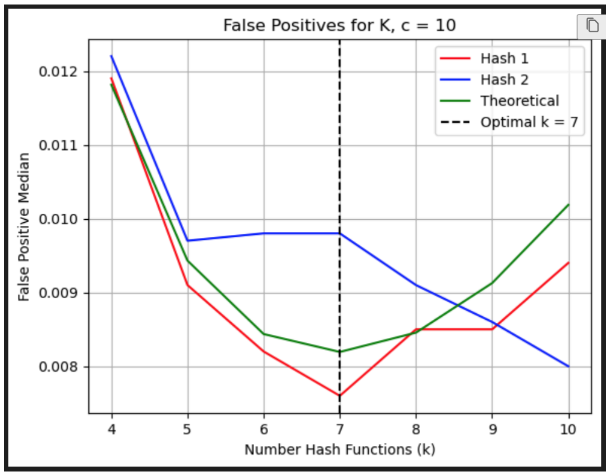
FirstHashFunction(i, x): Gets the specified generated seed for the Hash Function number and adds x to get the curSeed. The specified generated seed array for the Hash Functions is created randomly with mt19937 at the beginning of the program in the function **GenerateSeeds()** and stored in global int hashSeeds[k]. The curSeed is then placed into the random number generator mt19937 with range of (0, m-1), and the index is returned to the Add function as curLoc;

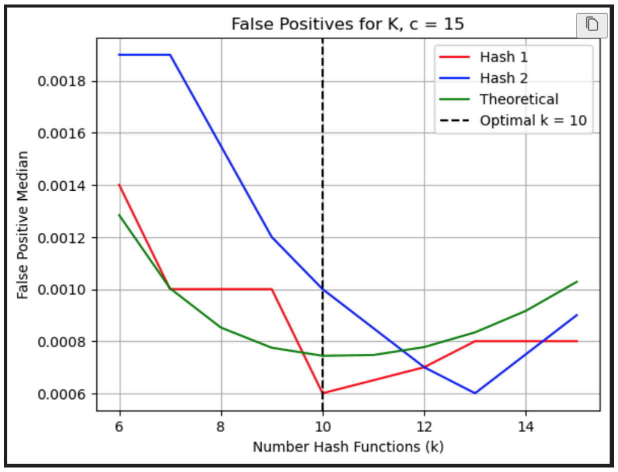
SecondHashFunction(i, x): Gets the specified int prime, and randomly generated hashA[x] and hashB[x] which are stored in global variables. Prime is created in the function **GeneratePrime()** which takes the smallest Mersenne Prime number greater than N, and HashA[k]/HashB[k] are created in the function **GenerateAB()** which creates different mt19937 random constants for A and B for each hash function.
contains(x): For each of the Hash Functions, if firstHash is true, contains(x) computes the location with **FirstHashFunction(i, x)**, and if false, uses **SecondHashFunction(i,x)**, where i is the current Hash function being computed. If at any iteration the current location of x has the visited flag set to false, the function returns false. If all Hash Functions are visited, return true for the BloomFilter containing x.
getBinCounts(): Used to graph the bin distribution for the Hash Values. Outputs a map<int, float> with the bin count for T in the int, and the number of bins with that bin count in the float.

3. Analyze False Positive Rate for Different Values k

Next I tested the false positive rate between varying parameters. I first kept the constant c at 10, and calculated k from equation $k = (0.4 \ to \ 1.0) * c$ in intervals of 0.1. For the second graph I used the same formula for k, but kept the constant c at 15. For non-whole values of k they are rounded to the nearest whole number. For each of the graphs, I plotted the First Hash function in red, the Second Hash function in blue, the theoretical false positive rate of a hash function in green $(1 - e^{-k/t})^k$, and the theoretical optimal k value from the equation $c ln(2)$. Each point on the graph is the calculated median of the false positive rate for 10 uniquely generated BloomFilters.

Constants used for Visibility: N = 10000000, n = 10000





Graph #3: False Positive Rate c = 10   Graph #4: False Positive Rate c = 15