Karel Klein Cardeña

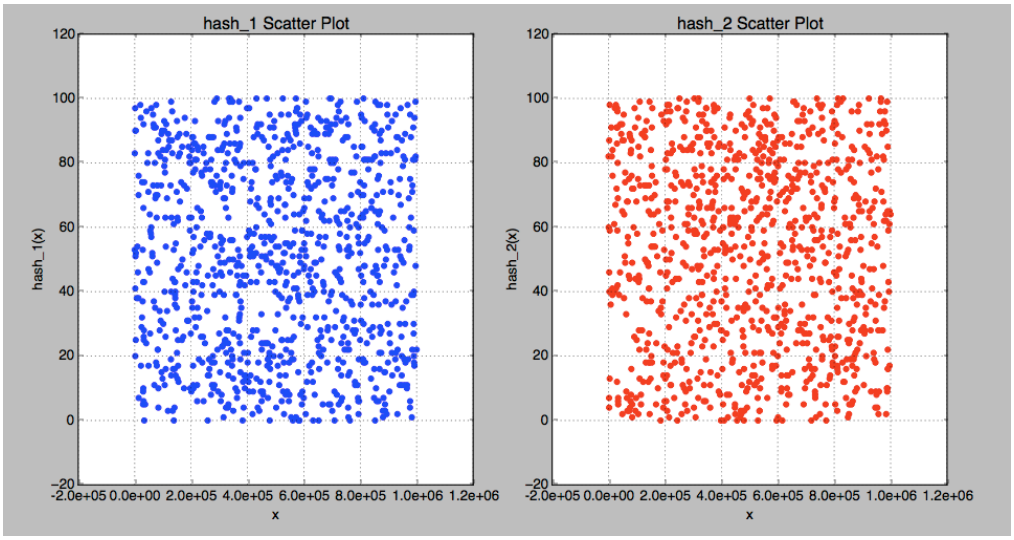
Project 1: Bloom Filters

Part 1 - Hash Functions

For this implementation of bloom filters, two hash functions were tested. The first hash function, $h_i(x) = r(s_i+x)$, takes an input x and generates a random integer by seeding with the sum of a random seed and x. The second hash, $h_i(x) = (a_i\, x + b_i)\ mod(n)$, uses two randomly generated integers a and b to modify x and output an index for the hash table.

To ensure both hashes compute random ouputs, we plot 1000 randomly chosen points from the universe $U=\{0, 1, \ldots, N-1\}$ and let $N = 1e6$. The y-axis will display the values of the table each x is mapped to; their distribution across the plot should reveal no pattern or correlation if the hashes output a uniformly random index.

Figure 1
$N = 1e6$
$n = 101$
$m = 1000$



As the plot for each hash function shows, the values mapped to are uniformly random across the input x and seem to be uniformly distributed across each bin in the hash table. To ensure this is the case, we find the average max and min loads calculated by repeating this experiment 1000 times (with $n = 1009$). For `hash_1`, the average min and average max loads were found to be *1.000* and *5.483*, respectively. `hash_2` respective statistics were *1.00* and *5.484*. Based on these numbers alone, there appears to be no advantage in using one hash over the other. Both map to uniformly random integers and have nearly the same average min and max load sizes.

Part 2 - Bloom Filter Implementation

To implement the bloom filter, I utilized Python and created a set of functions to handle each component of the filter. Two initialization functions were created, one per hash tested. `init()` creates a hash table of size n and creates an array of k random seeds. To assure random seeds, I utilized `time.clock()`, which returns the current processor time as a floating point number. In `init2()`, two arrays of k random integers are created with

the `random.random` generator. The `add(x)` and `contains(x)` functions utilize `hash_1`, as Part 3 experiments found it to be the slightly better function. Lastly, a `main()` function is created to allow for a user to input elements to add and query.

Part 3 - False Positive Rate Analysis

To test for optimal *k* and *c* values, we can use each hash function and plot varying *k* values versus the false positive rate achieved. This is repeated for various *c* values.

Figure 2: *Analysis for* `hash_1()`

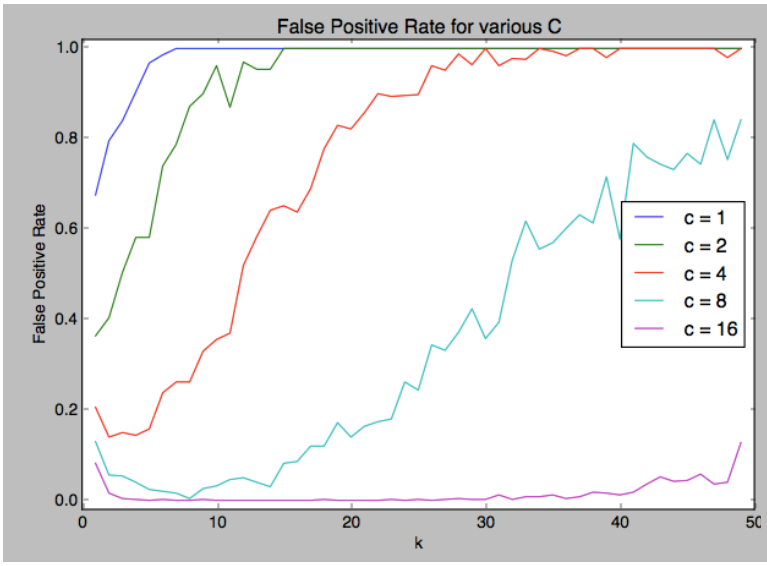
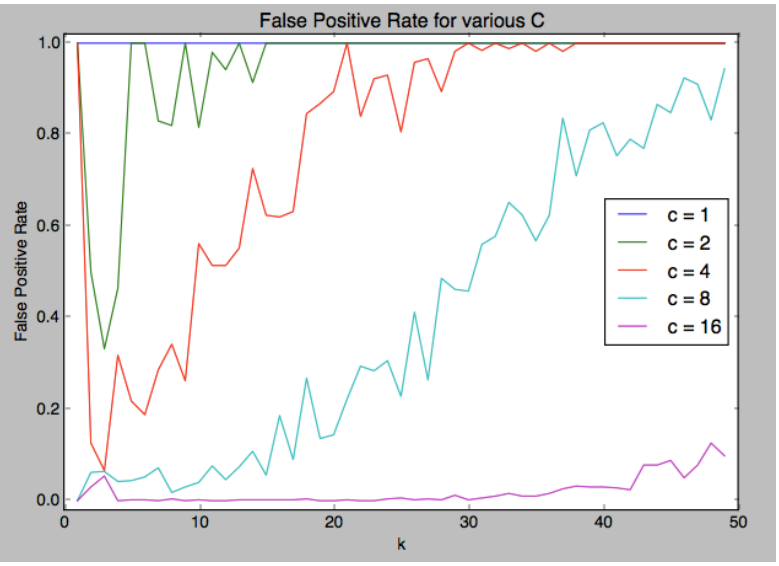


Figure 3: *Analysis for* `hash_2()`



From these plots, two general trends are immediately recognizable. The first is that as *c* is increased, the false positive rate noticeably decreases for every *k*. This makes intuitive sense: since the hash table size is $n=mc$, the size of the table grows proportionally to the size of *c*. Values mapped to will be increasingly sparse in the table and the likelihood of two *x* values sharing all $h_i(x)$ indices reduces. Given a fixed number of elements added *m*, queried *t*, and hashes *k*, the theoretical probability $Pr(\text{false positive}) = (1 - e^{-k/c})^k$ decreases as *c* increases, matching the above results.

Secondly, each curve forms a parabola between the interval $k=[0,10]$ within which it reaches a minimum, and after which it keeps an increasing trend. Considering the `hash_1` curve for $c=4$, we can calculate the theoretically optimal value for *k*: $k=c\ln 2=2.773$. This corresponds to the minimum found on the curve, which lies between $k = 2$ and $k = 3$. In the case of `hash_2` for $c=2$, the theoretical optimum happens at $k=1.386$, which the curve approximates well with a minimum at $k=1.476$. Given the proximity of optimal *k* values and the resemblance of the curves to the theoretical $Pr(fp)$, the experimental results strongly affirmed the theoretical predictions.