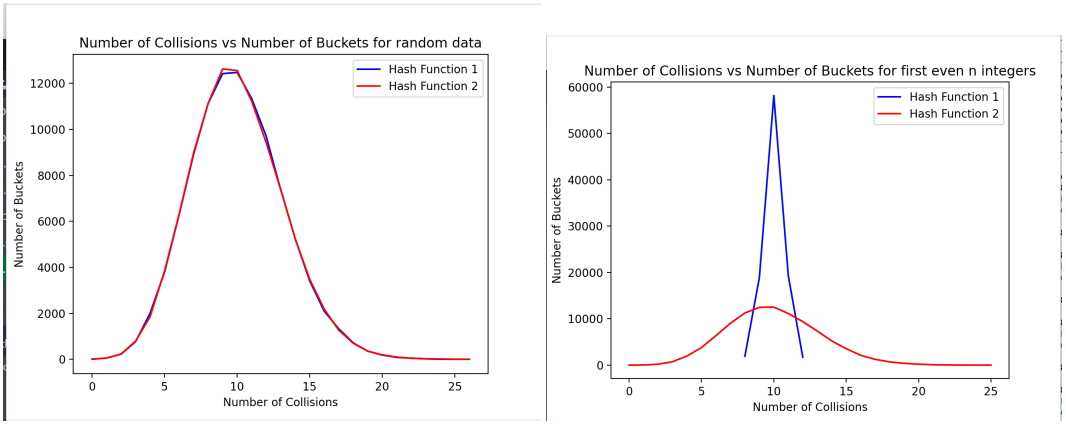
Bloom Filter Report

1 Introduction

A Bloom filter is a data structure based on hashing that can determine, with high probability, whether an element is in the table of size m. It works by hashing the inserted value using k different hash functions and setting the corresponding table slots to 1. While false positives can occur, false negatives cannot, as the presence of an element is determined by checking whether all the slots for the corresponding hash functions are set to 1. This report analyzes the collisions of two different hash functions (one based on a linear function and the other on a random number generator) and compares their false positive rates to theoretical false positive rates.

2 Part 1

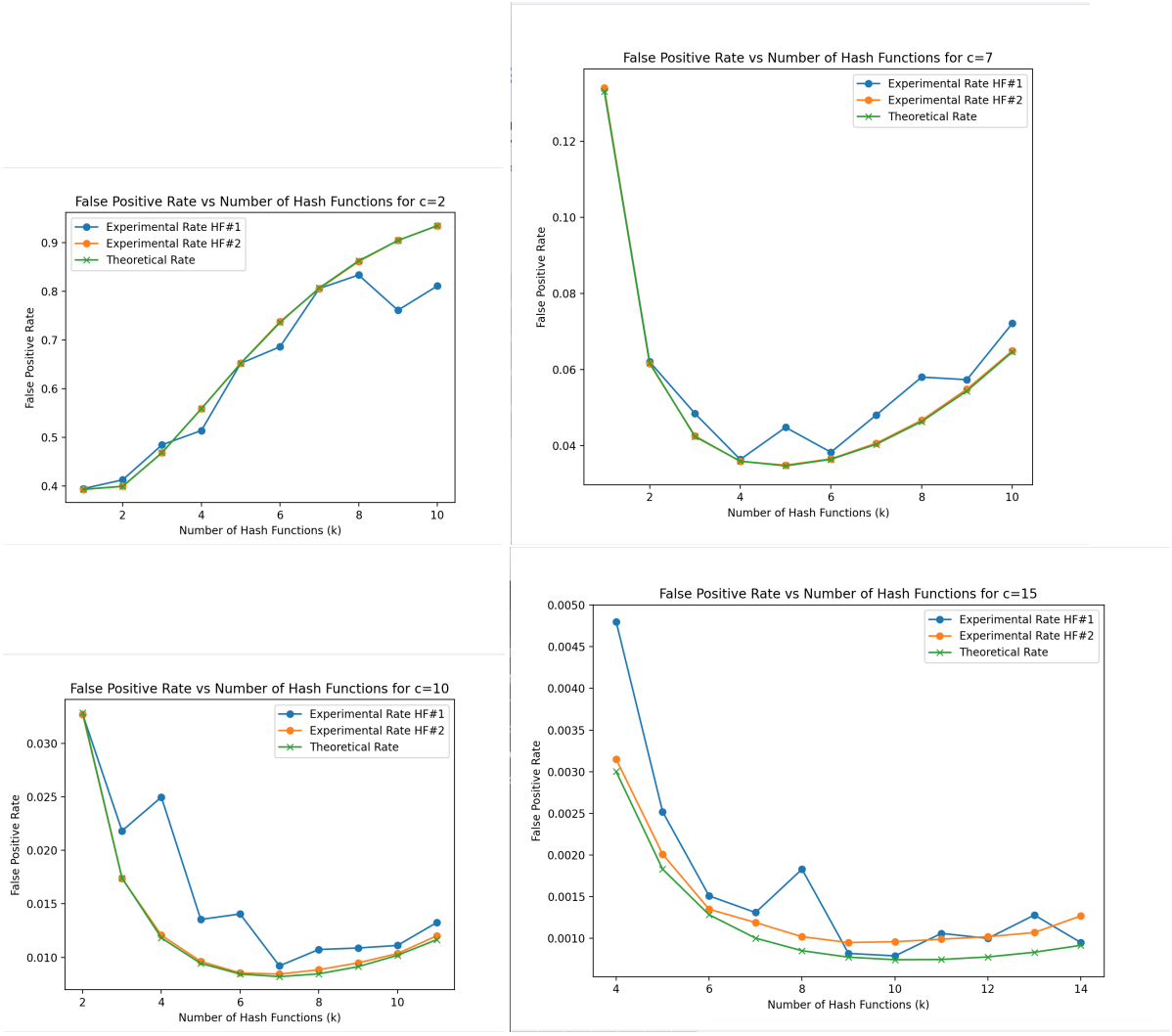


The first graph shows the number of collisions versus the number of buckets that had that number of collisions, with elements generated randomly for insertion. The observed normal distribution likely stems from the central limit theorem. As random data is hashed into buckets, the distribution of collisions tends to aggregate around a mean value because of the independent and uniform nature of the hash function. A bell curve is created by deviation from the mean caused by chance clustering, reflecting the expected behavior of random inputs across a finite set of bins. This result shows the hash function's capacity to spread elements evenly under random conditions. Both hash functions had a similar distribution indicating no difference given the insertion of randomly selected elements. To create the graph, I set m to 100,000 and n to 1,000,000 to ensure collisions and analyze how many collisions per bucket. For both hash functions, the expected amount of collisions for a bucket was 10 collisions. This is because the elements were inserted randomly, making each insertion independent. Thus, the expected amount of collisions is n/m=10. However, when the sequential data is inputted for the second graph (in this case, the first 2n even integers), hash function 1 exhibits a spike rather than a bell-shaped curve while hash function 2 retains its previous shape. This discrepancy could be because hash function 1 relies on a linear function, causing buckets to repeat themselves in a pattern that clusters elements into specific buckets. Despite this, for both hash functions, the expected number of collisions remains 10.

3 Part 2

I implemented the Bloom Filter as a class, so that a_i, b_i, and s_i would be different every time I instantiated a new Bloom Filter. I set p to $2^{31} - 1$ because it is the largest Mersenne prime that can be represented. I set n to 100,000 to test a large amount of inserts into the bloom filter. I set N to 2^{30} so that it's less than p, but still large. I created two different *add* functions and *contain* functions, each corresponding to a different hash function.

4 Part 3



I used the parameters in Part 2 and randomly inserted n elements from the universe into the bloom filter. I ran experiments for fixed values of c=2, c=7, c=10, and c=15. For k, I chose 10 values of k around k=cln(2) as that is the theoretical number of optimal hash functions. I then performed 100,000 queries, checking for values in the bloom filter. I calculated the false positive rate as

$(1 - e^{(-kn)/m})^k$ for every value of k and c. While the results of both hash functions differed slightly from the theoretical rates, likely due to statistical error, they roughly follow the same curve. The curve decreases quickly until it reaches the trough and then increases slowly. However, hash function 1 deviated more significantly, likely because it is not truly random for related inputs, as shown in Part 1. As expected, the lowest false positive rate occurs when k=cln(2) for both hash functions and for all values of c. As c increases, the theoretical minimum false positive rate (the trough) decreases due to an increasing table size leading to fewer collisions and the number of optimal hash functions (k) increases.

5 Conclusion

This report demonstrates the performance differences between two hash functions in a Bloom filter and how their characteristics influence collision rates and false positive rates. Hash function 1, which relies on a linear function, showed predictable patterns when inserting sequential data, leading to spikes in the collision vs frequency graph and deviations from theoretical expectations of false positive rates. In contrast, hash function 2, based on a random number generator, maintained a more consistent distribution across all types of data, aligning more closely with theoretical predictions. Despite these differences, both hash functions followed the expected trends for minimizing false positives, particularly when k=cln(2), and larger values of c reduced the overall false positive rates. These results highlight the importance of selecting hash functions carefully based on the type of data being inserted to optimize Bloom filter performance.