James Stell
ITCS 6114
Nov. 16, 2015

HASHING AND BLOOM FILTER

EXPERIMENT WITH DIFFERENT LOAD FACTORS

The variations of hashing algorithms implemented were done using 1,000,000 random integers ranging from 0 to 10,000,000. Multiple hashing algorithms and variations of them were tested using a 1.0 load factor for benchmarking.

Multiple hashing algorithms were tested with multiple variations of numbers within the hashing algorithms. All performed about the same with around 36% of the hash table empty and the longest linked list ranging from 5 to 8 elements. A little bit of memory is wasted but lookups and insertions are basically constant time with O(1) time complexity.

> IMPORTANT NOTE: When selecting a load factor or recommended hash table size, it is not guaranteed that you will get *exactly* this size. For performance benefits the requested table size or load factor will be rounded up to the nearest prime number larger than the requested number (unless the requested number is a prime number itself).

The experiments were also run with varying sizes of *n*, or input numbers. Ranging from 1,000 – 1,000,000. The hash table percent empty results were proportionally the same each time remaining at almost exactly 36%. The longest list however remained almost constant at 7 – 8 elements.

After settling down on one particular hash function I began testing different load factors, while varying the size of n.

INFORMATION ON RUNNING MY PROGRAM

- My program accepts command line arguments for specifying the size of the hash table as well as how many numbers to process from a given file.
- There are two ways to specify the size of the hash table.
  - Specify the size of the hash table as a number
    - Use the '-s' flag for using a specific size
  - Specify the size of the hash table as a load factor
    - Use the '-l' flag for specifying a load factor
- Syntax for executing program:
  - `Hashing <file> <# of elements to process> [<-s> <size of hash table> | <-l> <load factor>]`
- Example for reading 1,000 numbers from the file num.txt with a hash table size of 500
  - Hashing nums.txt 1000 -s 500
- Example for reading 1,000 numbers from the file num.txt with a load factor of .75
  - Hashing nums.txt 1000 -l .75

James Stell
ITCS 6114
Nov. 16, 2015

Load Factor Results

Size of n	Load Factor	% Empty	Longest List	Build Time (milliseconds)
100	0.1	90.49	2	5
	0.2	81.11	2	5
	0.4	66.13	2	5
	0.8	47.24	3	5
	1	38.61	4	5
	1.6	23.88	5	5
	3.12	5.41	7	4
	6.24	0.00	10	5
	10	0.00	19	4
1,000	0.1	90.44	3	21
	0.2	81.79	3	20
	0.4	67.20	4	21
	0.8	43.29	5	21
	1	36.57	6	21
	1.6	19.80	6	21
	3.12	4.83	9	21
	6.24	0.62	15	20
	10	0.00	18	21
10,000	0.1	90.52	4	171
	0.2	81.88	4	170
	0.4	67.06	5	172
	0.8	44.56	6	171
	1	37.21	7	171
	1.6	20.31	7	171
	3.12	3.99	11	170
	6.24	0.19	17	171
	10	0.00	23	172
100,000	0.1	90.48	4	1,566
	0.2	81.83	5	1,578
	0.4	67.03	6	1,565
	0.8	45.01	7	1,571
	1	36.92	7	1,575
	1.6	20.32	9	1,571
	3.12	4.47	14	1,572
	6.24	0.21	18	1,568
	10	0.00	25	1,568
1,000,000	0.1	90.49	5	15,790
	0.2	81.88	5	15,566
	0.4	67.04	6	15,458
	0.8	44.91	8	15,589
	1	36.74	8	15,572
	1.6	20.24	10	15,421
	3.12	4.39	14	15,480
	6.24	0.19	19	15,357
	10	0.00	26	15,600

ANALYSIS OF RESULTS

As one can see from the results the build times are almost exactly identical for a particular size of n, regardless of the load factor. This is the exact definition of O(n) or linear time to build a particular hash table. Retrieving an element from a table, regardless of load factor will be done in constant time, or O(1).

With a million elements the largest list is 26, which takes no time at all to scan through for a computer. Since the results of the operations on a hash table are the same regardless of load factor, the best load factor really comes down to memory management.

I think a load factor of 3 to 10 is ideal for best performance. Therefore the hash table will be anywhere from 4% to 0% empty – most spots in the hash table full – will the longest list ranging from 14 to 26. So this will save memory for the table allocation and still allow operations in constant time of O(1) – really O(longest list), however in a particular instance this is constant and therefore becomes O(1).

BLOOM FILTER

The bloom filter was implemented with Java's BitSet object. This was a preferred class over using an integer array due to its smaller size per element. An int in java takes up 4 bytes – or 32 bits – whereas the BitSet data structure uses one single bit for each element. Therefore an int array of size 100 is 400 bytes – or 1,600 bits – whereas a BitSet object of size 100 is 100 bits! So using a BitSet to set a 0 or 1 flag takes up 32 times less space than the same sized int array!

To keep the bloom filter under 1.1 MB in size so it can fit in cache means it cannot take up more than 2^{20} bytes, or roughly 1,000,000 bytes. So a max size of 8 million was used for the bloom filter for testing.

RESULTS

Testing different variations of m, k and n was done in four main stages. M was tested at sizes 1,000 / 10,000 / 100,000 / and 1,000,000. For each size of m I tested three different sizes of k – 1, 2 and 3. Then finally for each size of k I tested 6 sizes of n – 100 / 1,000 / 10,000 / 100,000 / 1,000,000 / and 10,000,000.

The results are as follows:

(NEXT PAGE)

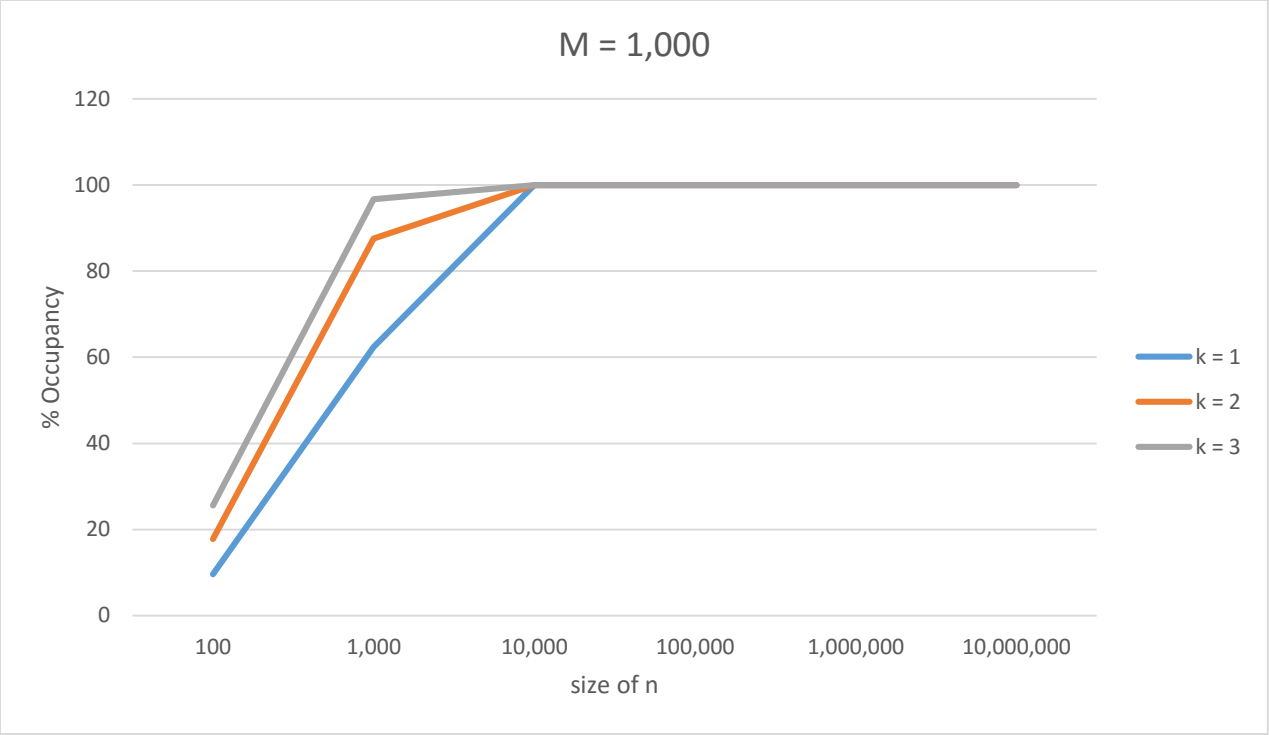
James Stell
ITCS 6114
Nov. 16, 2015

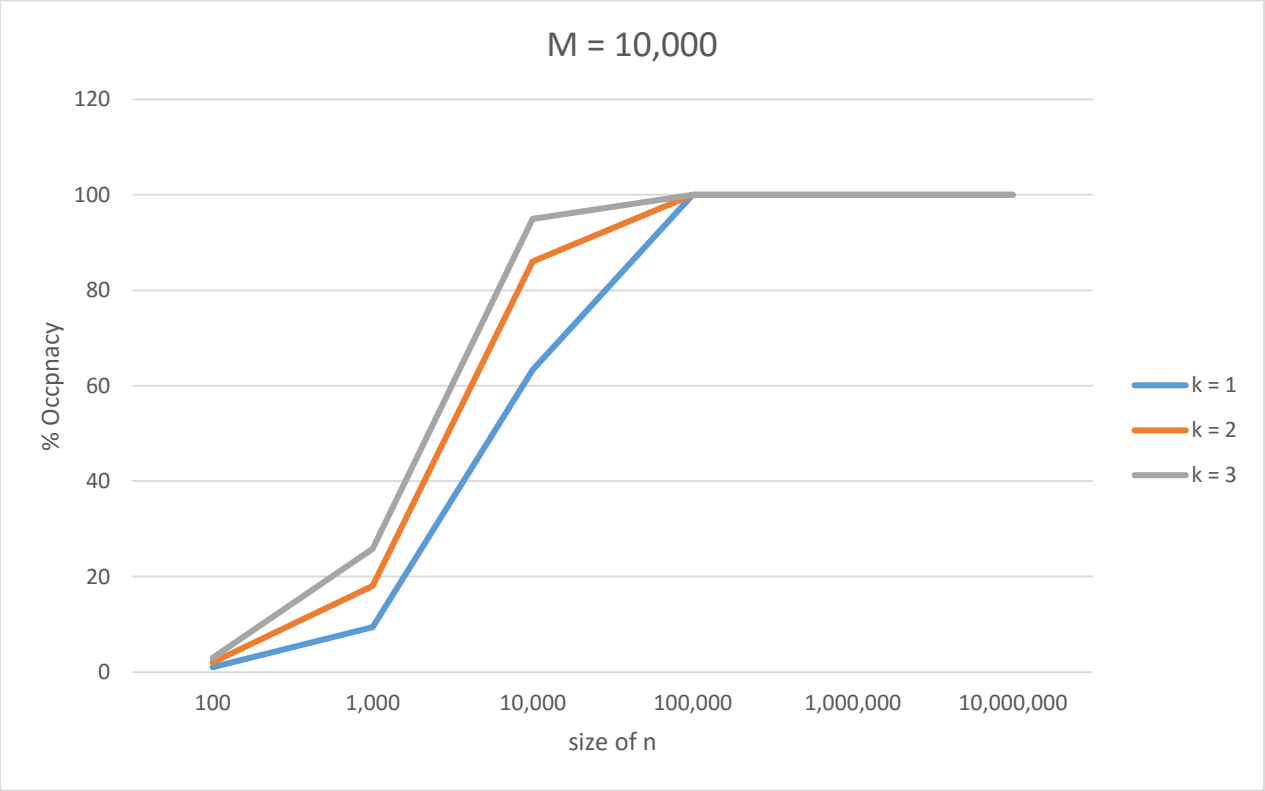
Bloom Filter

m	k	n	% Occupancy	m	k	n	% Occupancy
1,000	1	100	9.6	10,000	1	100	1
		1,000	62.4			1,000	9.39
		10,000	100			10,000	63.33
		100,000	100			100,000	99.99
		1,000,000	100			1,000,000	100
		10,000,000	100			10,000,000	100
	2	100	17.8		2	100	2
		1,000	87.6			1,000	18.04
		10,000	100			10,000	86.01
		100,000	100			100,000	100
		1,000,000	100			1,000,000	100
		10,000,000	100			10,000,000	100
	3	100	25.6		3	100	2.95
		1,000	96.7			1,000	25.8
		10,000	100			10,000	94.96
		100,000	100			100,000	100
		1,000,000	100			1,000,000	100
		10,000,000	100			10,000,000	100
100,000	1	100	0.1	1,000,000	1	100	0
		1,000	0.991			1,000	0
		10,000	9.496			10,000	0.99
		100,000	63.316			100,000	9.5
		1,000,000	99.99			1,000,000	63
		10,000,000	100			10,000,000	99.99
	2	100	0.2		2	100	0
		1,000	1.97			1,000	0.19
		10,000	18.173			10,000	1.97
		100,000	86.483			100,000	18.11
		1,000,000	100			1,000,000	86.2
		10,000,000	100			10,000,000	100
	3	100	0.3		3	100	0
		1,000	2.95			1,000	0.29
		10,000	26			10,000	2.9
		100,000	95			100,000	25.9
		1,000,000	100			1,000,000	94.92
		10,000,000	100			10,000,000	100

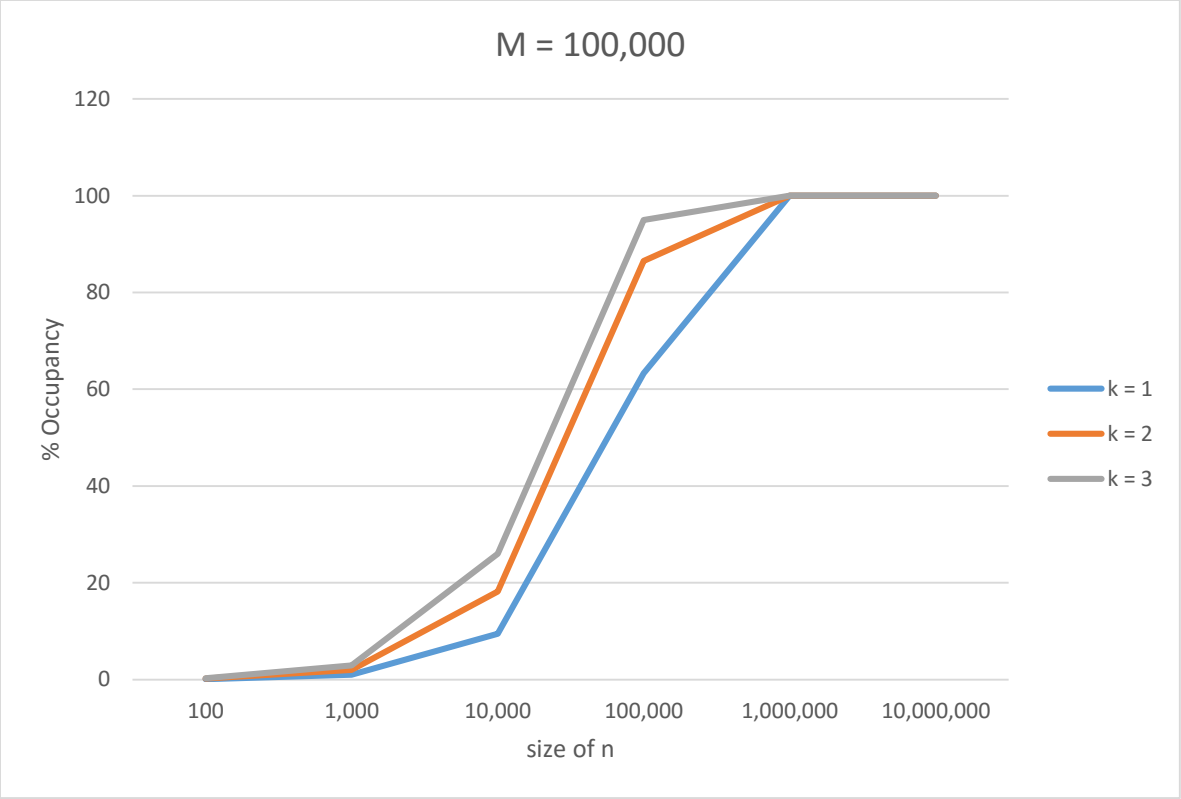
THE GRAPH OF THE RESULTS IS BELOW

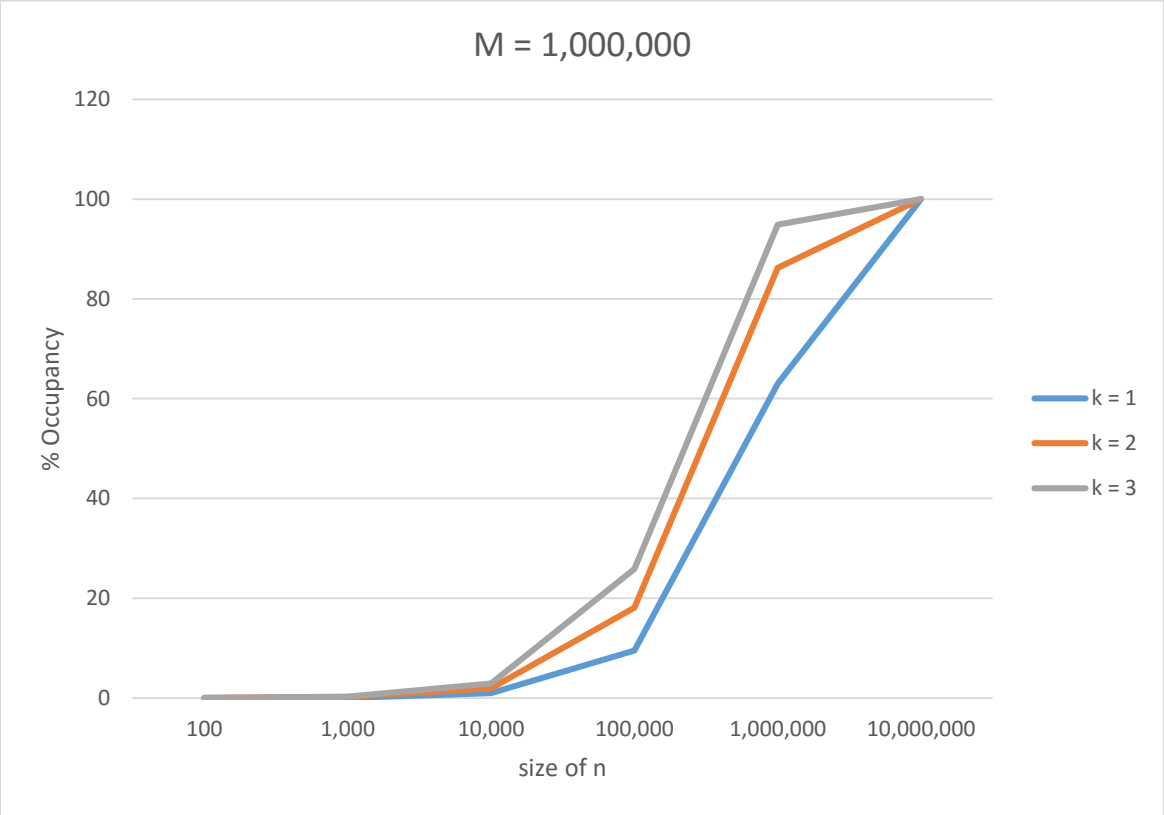
James Stell
ITCS 6114
Nov. 16, 2015

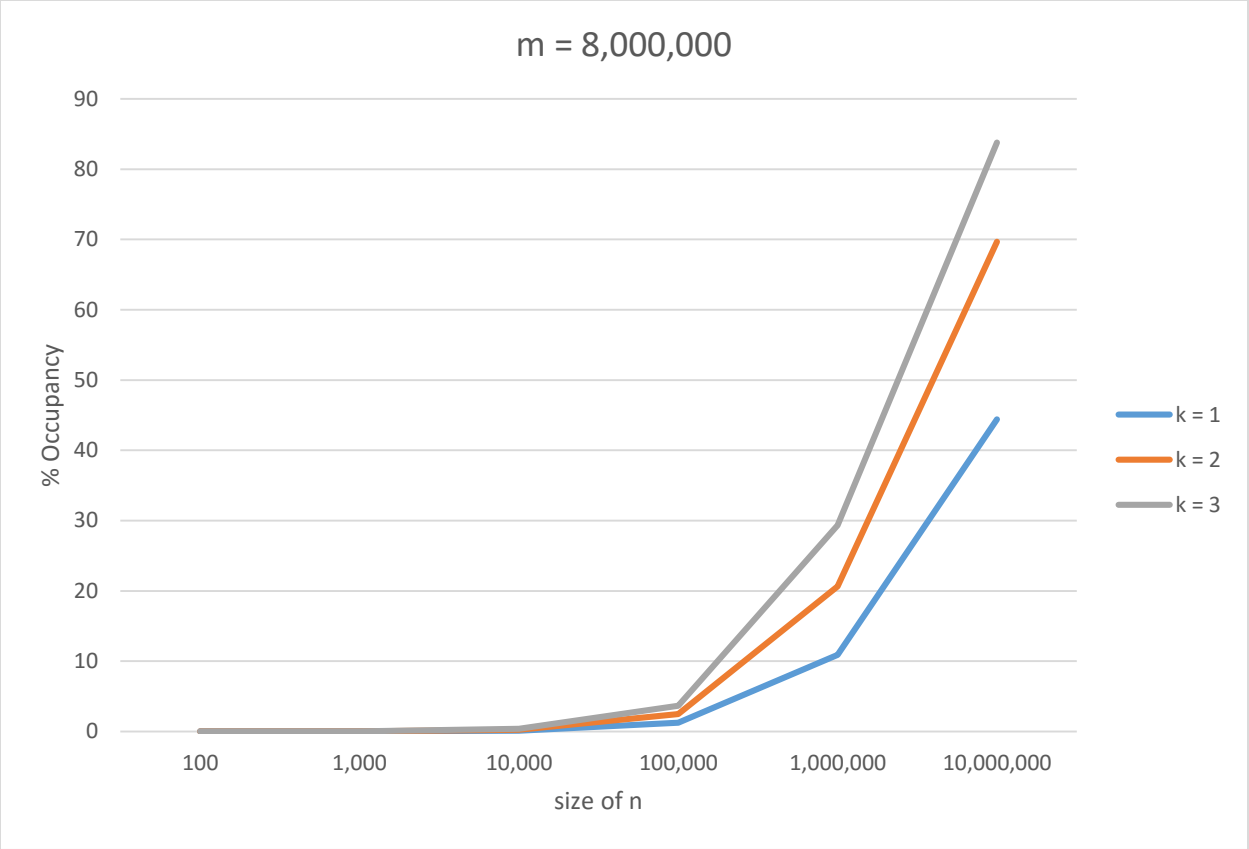




James Stell
ITCS 6114
Nov. 16, 2015





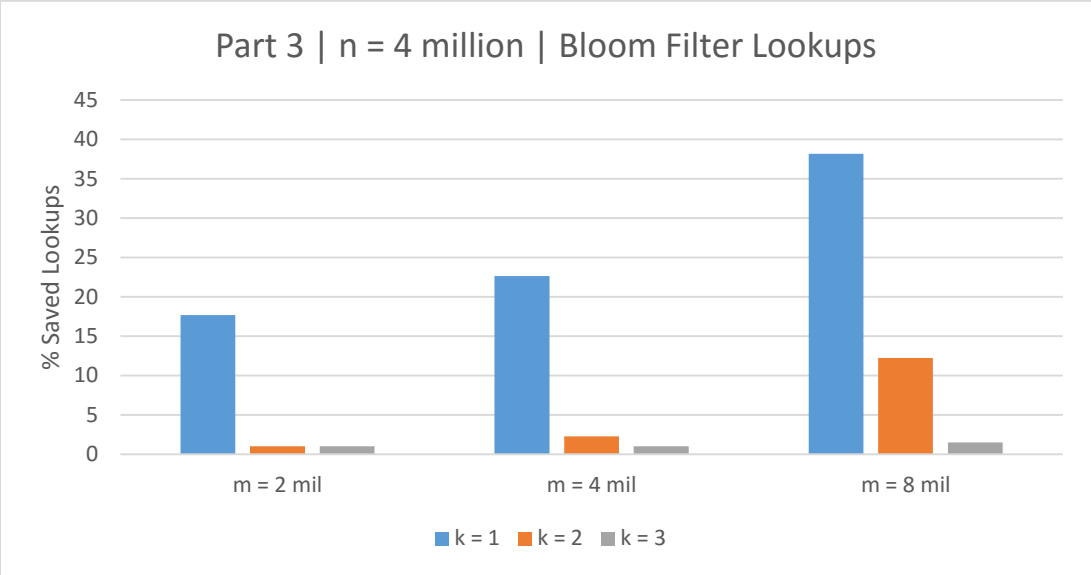


PART 3:

For part 3, three different sizes of m were used – 2, 4, and 8 million. Each bloom filter size was tested with three different k sizes – 1, 2, and 3. All tests were conducted with an n of size 4 million. Each test 4 million random numbers were inserted into an unsorted array after being hashed through the bloom filter. Then another three thousand random integers were 'looked up' – not *actually* looked for in the array, but rather checked against the bloom filter – to see if we were spared the expensive array search. At each iteration I capture the amount of times we knew for sure that the searched element was not in the array, because the bloom filter hashes for that number were 0. Therefore I have plotted as a percentage (out of 3,000 lookups) how many times we did not have to perform the expensive array search for an element when it is 100% not in the array.

THE RESULTS ARE AS FOLLOWS

Part 3 | n = 4,000,000		
m	k	Saved Lookup %
2,000,000	1	17.67
	2	0
	3	0
4,000,000	1	22.63
	2	2.26
	3	0.1
8,000,000	1	38.16
	2	12.2
	3	1.47



CONCLUSION

After doing multiple tests on data using bloom filters, it is apparent that k hash functions needs to be smaller to get better results. All my tests were done using 1, 2 and 3 different hash funtions, and the best results were captured when k was one. It is also better if the size of the bloom filter, m, is larger than your data set being entered into the bloom filter.

However, the actual size of the bloom filter did not change results, only the size of the bloom filter in relation to the data set, as a percentage of the data being entered. It is best to be of the same size or larger as the amount of data being entered. If m is at least the same size or twice as large as the data set then the best results will be obtained, especially if only using one hash function.