CSCI 3104, Fall 2017 Problem Set 5
Austin Griffith (MM/DD)
augr4882@colorado.edu
Coding Language: Python

Problem 1

(10 pts) Given the balanced binary trees T_1 and T_2, which contain m and n elements respectively, we want to determine whether they have some particular key in common. Assume an adversarial sequence that placed the m and n items into the two trees.

(A) *Suppose our algorithm traverses each node of T_1 using an in-order traversal and checks if the key corresponding to the current node traversed exists in T_2. Express the asymptotic running time of this procedure, in terms of m and n.*

Since in-order traversal is an algorithm used on tree data structures we know the time complexity to travel through T1 will be $\Theta(m)$. Since this is a binary search tree you have to look at every node of the tree which is $\Theta(m)$. The running time it takes to search T2 to see if any values match in T1 is $\Theta(logn)$. The asymptotic running time of this procedure in terms of m and n will be $\Theta(mlogn)$ to check weather a corresponding key in T_1 is in T_2 with the in-order traversal of T1 and search of T2.

(B) *Now suppose our algorithm first allocates a new hash table H_1 of size m (assume H_1 uses a uniform hash function) and then inserts every key in T_1 into H_1 during a traversal of T_1. Then, we traverse the tree T_2 and search for whether the key of each node traversed already exists in H_1. Give the asymptotic running time of this algorithm in the average case. Justify your answer.*

The traversal of T_1 will still be $\Theta(m)$ time complexity and the insertion of every key in T_1 will be done in constant time $\Theta(1)$. The time complexity at worst case of traversing all nodes T_2 will also be $\Theta(n)$ and searching weather the key accessed in T_2 is in the hash table will be constant time $\Theta(1)$ so the total time complexity will be $\Theta(mn)$

Problem 2

(45 pts) A good hash function $h(x)$ behaves in practice very close to the uniform hashing assumption analyzed in class, but is a deterministic function. That is, $h(x) = k$ each time x is used as an argument to $h()$. Designing good hash functions is hard, and a bad hash function can cause a hash table to quickly exit the sparse loading regime by overloading some buckets and under loading others. Good hash functions often rely on beautiful and complicated insights from number theory, and have deep connections to pseudorandom number generators and cryptographic functions. In practice, most hash functions are moderate to poor approximations of uniform hashing.

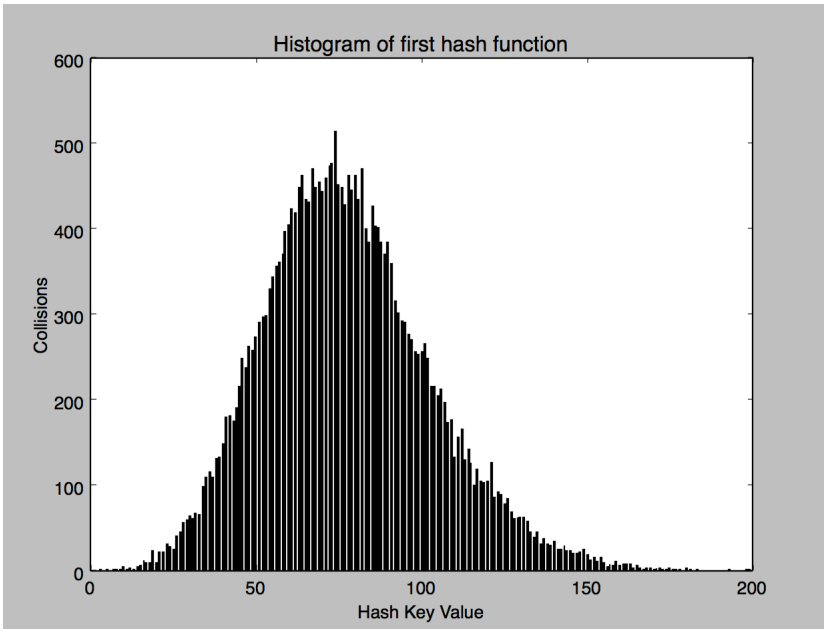
Consider the following two hash functions. Let U be the universe of strings composed of the characters from the alphabet $\sum = [\texttt{A}, \cdots, \texttt{Z}]$, and let the function $f(x_i)$ return the index of a letter $x_i \in \sum$, e.g., $f(\texttt{A}) = 1$ and $f(\texttt{Z}) = 26$. Finally, for an m-character string $x \in \sum^m$, define $h1(x) = ([\sum_{i=1}^{m} f(x_i)] \ mod \ l)$, where l is the number of buckets in the hash table. For the other hash function, let the function $f2(x_i, a_i)$ return $x_i * a_i$, where a_i is a uniform random integer, $x \in [0, \cdots, l-1]$, and define $h2(x) = ([\sum_{i=1}^{m} f(x_i, a_i)] \ mod \ l)$. That is, the first hash function sums up the index values of the characters of a string x and maps that value onto one of the l buckets, and the second hash function is a universal hash function.

(A) *There is a txt file on Moodle that contains US Census derived last names: Using these names as input strings, first choose a uniformly random 50% of these name strings and then hash them using $h1(x)$ and $h2(x)$.*

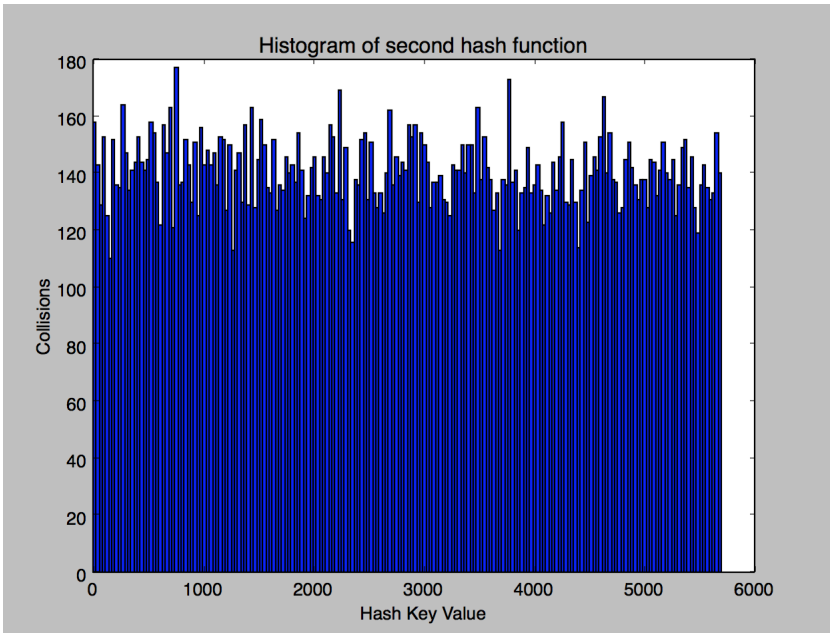
See python code.

(B) *Produce a histogram showing the corresponding distribution of hash locations for each hash function when $l = 5701$. Label the axes of your figure. Brief description what the figure shows about $h1(x)$ and $h2(x)$; justify your results in terms of the behavior of $h(x)$. Hint: the raw file includes information other than the name strings, which will need to be removed; and, think about how you can count hash locations without building or using a real hash table.*

After running python code from command line or terminal, Histogram for h1(x) and h2(x) will pop up after computation has finished.



1.png 7/Histogram 1.png



2.png 7/Histogram 2.png

(C) *Enumerate at least 4 reasons why $h1(x)$ is a bad hash function relative to the ideal behavior of uniform hashing.*

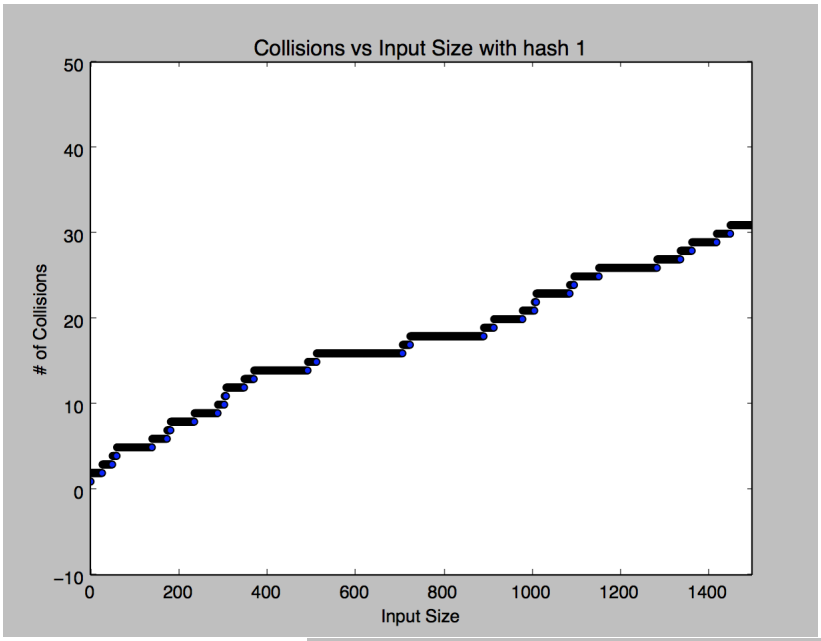
1) one reason h1(x) is a bad hash function when compared to that of a universal hashing function is because of the big clusters it creates while trying to assign each particular key to a particular bucket of l=5701.

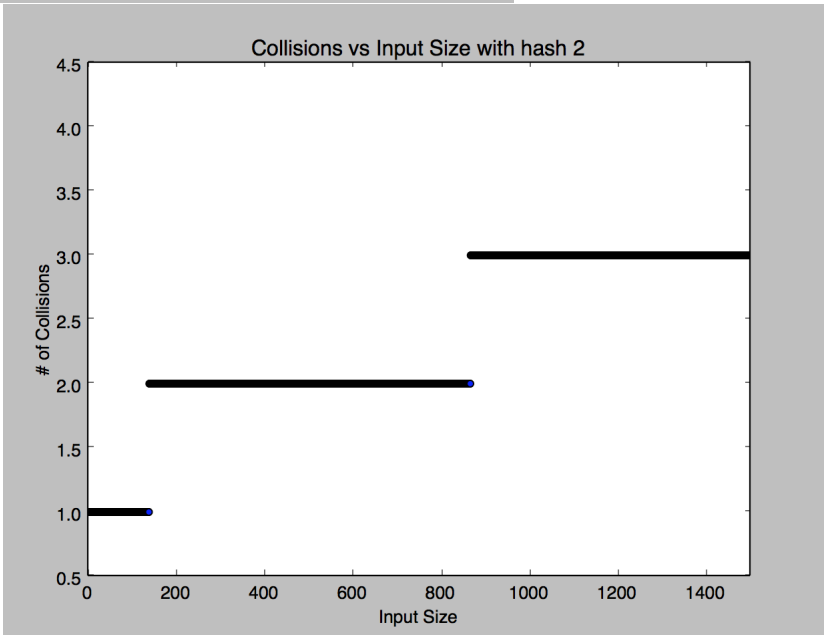
2) within the histogram created for h1(x) it can be noted that most and nearly all of the clustering is happening near the median of the bell curve on the graph showing that many collisions are happening. This is bad when comparing it to a uniform hashing function that will have much much more even distribution, if any, of collisions of key values to a given amount of buckets within the table.

3) h1(x) is also a bad hashing function because the probability of getting a collision while trying to fill a bucket is no where near equal for each bucket hence the collisions. A universal hash functions need to have equal probability that each bucket gets filled where h1(x) does not. A uniform hashing function also has a better way of handling collisions because of its randomness factor of allocating buckets for specific keys values to avoid collisions.

4) Since our data set involves strings of English letters (characters) the frequency of certain vowels is going to make collisions unavoidable for h1(x) function on the given data set where a uniform hashing function would take care of this problem much better.

(D) *Produce a plot showing (i) the length of the longest chain (were we to use chaining for resolving collisions) as a function of the number n of these strings that we hash into a table with $l = 5701$ buckets. Produce another plot showing the number of collisions as a function of l. Choose prime numbers for l and comment on how collisions decrease as l increases.*



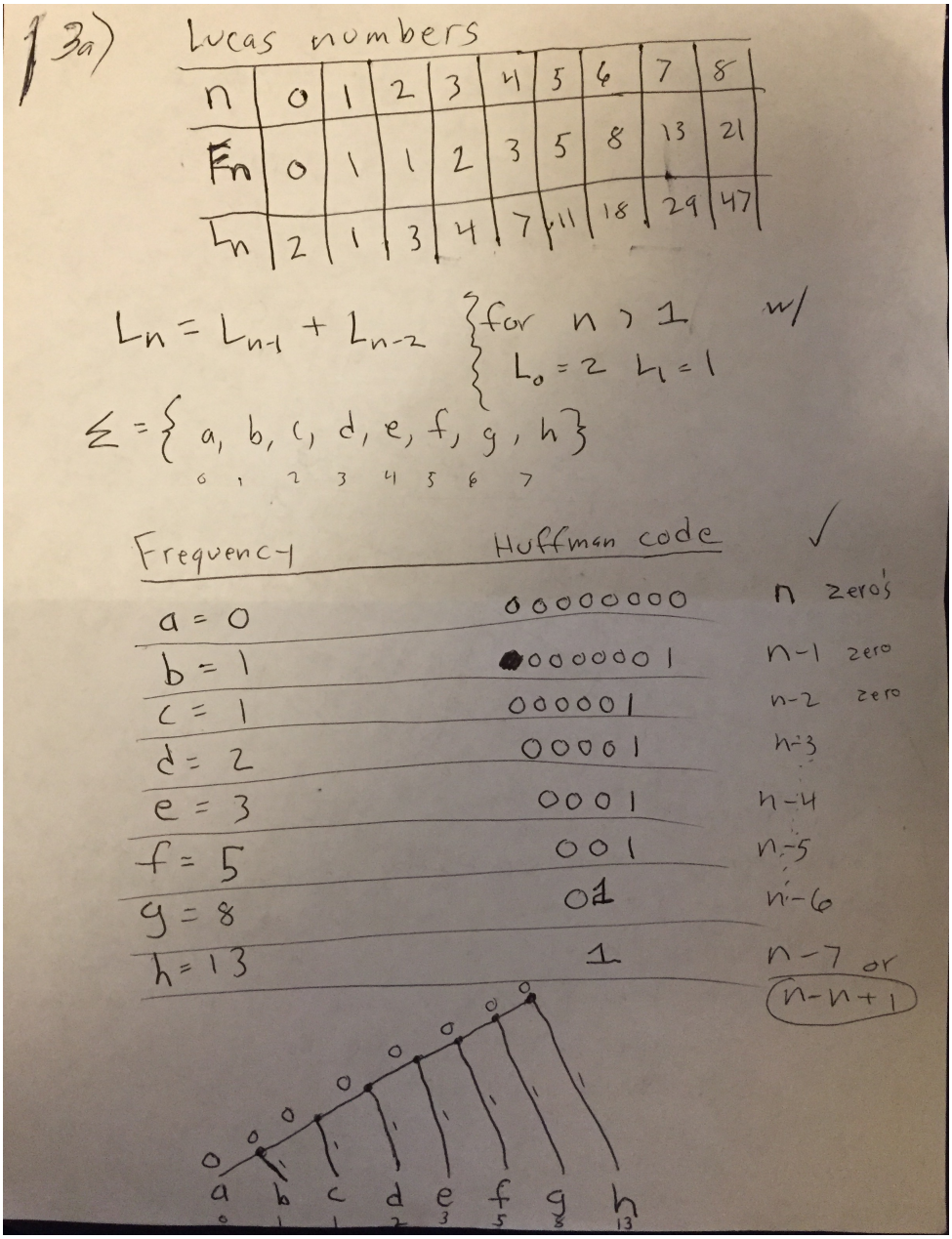


2.png 7/Plot 2.png

Problem 3

(15 pts) Voldemort is writing a secret message to his lieutenants and wants to prevent it from being understood by mere Muggles. He decides to use Huffman encoding to encode the message. Magically, the symbol frequencies of the message are given by the Lucas numbers, a famous sequence of integers discovered by the same person who discovered the Fibonacci numbers. The nth Lucas number is defined as $L_n = L_{n1} + L_{n2}$ *for* $n > 1$ *with base cases* $L_0 = 2$ *and* $L_1 = 1$.

(A) *For an alphabet of* $\sum = a, b, c, d, e, f, g, h$ *with frequencies given by the first* $|\sum|$ *Lucas numbers, give an optimal Huffman code and the corresponding encoding tree for Voldemort to use.*



3a)

Lucas numbers

n	0	1	2	3	4	5	6	7	8
Fn	0	1	1	2	3	5	8	13	21
Ln	2	1	3	4	7	11	18	29	47

$L_n = L_{n-1} + L_{n-2}$ for $n > 1$ w/ $L_0 = 2$ $L_1 = 1$

$\Sigma = \{a, b, c, d, e, f, g, h\}$

0 1 2 3 4 5 6 7

Frequency	Huffman code	✓
a = 0	00000000	n zeros
b = 1	00000001	n-1 zero
c = 1	0000001	n-2 zero
d = 2	000001	n-3
e = 3	00001	n-4
f = 5	0001	n-5
g = 8	001	n-6
h = 13	1	n-7 or n-n+1

(B) *Generalize your answer to* `(3a)` *and give the structure of an optimal code when the frequencies are the first n Lucas numbers.*

When the frequencies are the first n Lucas numbers we can see that the first element to be encoded such as "a" in this case will either have n 0's or n 1's. The huffmancode for the first Lucas number will be the longest such as above "a" is encoded with 8 zeros since they're are 8 letters in the sequence given. There will always be n number of zero's or 1's for n elements of Lucas numbers and n-n +1 for the last element. Overall the most visited element which has the highest frequency should have the least number of bits so it can be accessed the quickest and the least visited element should have the most bits.