CSCI 3104, Fall 2017 Problem Set 5
Adam Ten Hoeve (10/09)
adam.tenhoeve@colorado.edu

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.*

$$\mathcal{O}(m * \log n)$$

This runtime is because the algorithm has to go through every element of T_1, which is m, to compare to the elements in T_2. The algorithm does not need to look at every element in T_2 because it can search the sorted binary tree, which takes $\log n$ time. The orders are multiplied because T_2 gets searched from every node in T_1.

(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.*

$$\mathcal{O}(m) + \mathcal{O}(n)$$

The algorithm will traverse T_1 once to create the hash table, which takes $\mathcal{O}(m)$ time because every need needs to be traversed. Then the algorithm will traverse T_2 once to go through every node and compare their keys to the hash table, which takes $\mathcal{O}(n)$ time. Comparing the keys in the hash table runs at constant time because creating a hash key is constant so it does not need to be considered for this function. To get the final run time of the algorithm, we can add the two

run time together because they are independent of each other, they do not run within each other.

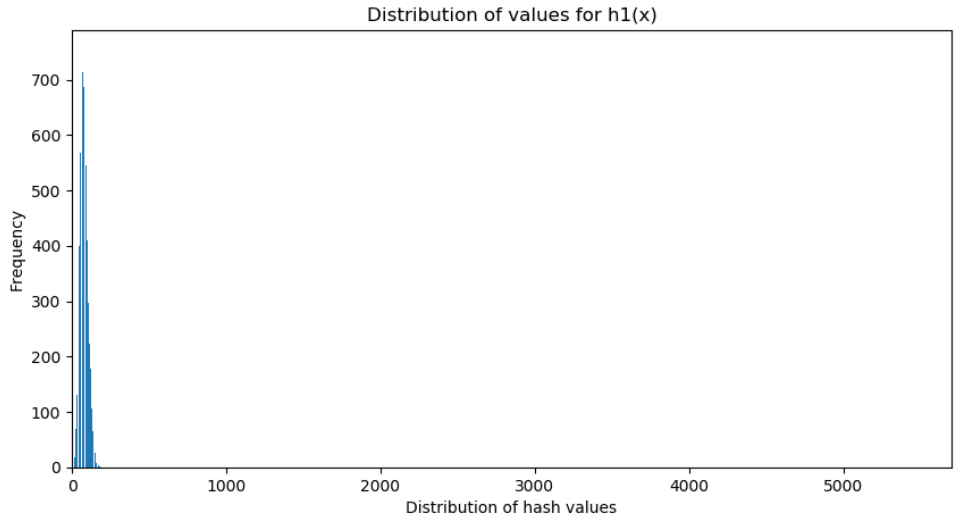
Problem 2

(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.

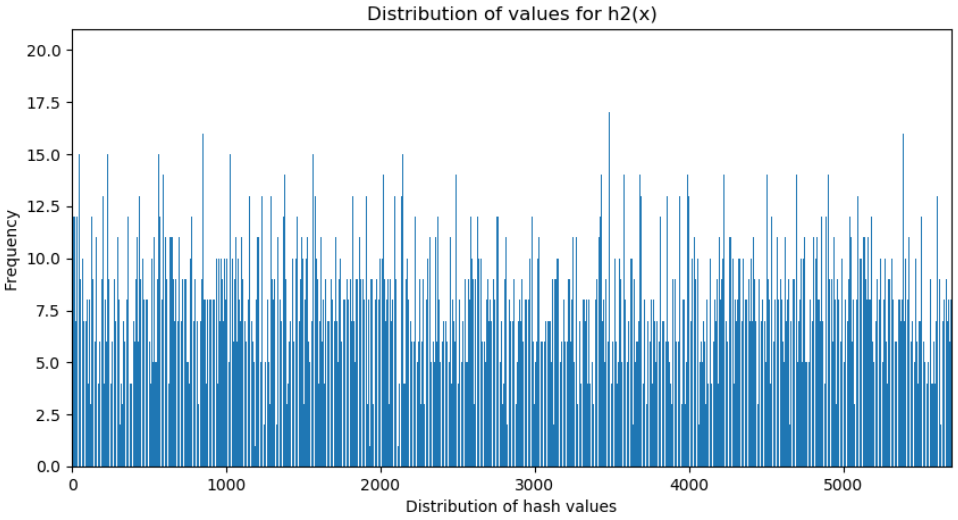
Consider the following two hash functions. Let U be the universe of strings composed of the characters from the alphabet $\sum = [\mathtt{A}, \cdots, \mathtt{Z}]$, and let the function $f(x_i)$ return the index of a letter $x_i \in \sum$, e.g., $f(\mathtt{A}) = 1$ and $f(\mathtt{Z}) = 26$. Finally, for 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 $h2(x)$.*

(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.*



This is the histogram for the distribution of the keys for the h1(x) hash function. As shown in the graph, most of the values are within 5% of the total hash table with a single key having over 700 collisions. Because of this, the algorithm is very inefficient and does not sort out data well.



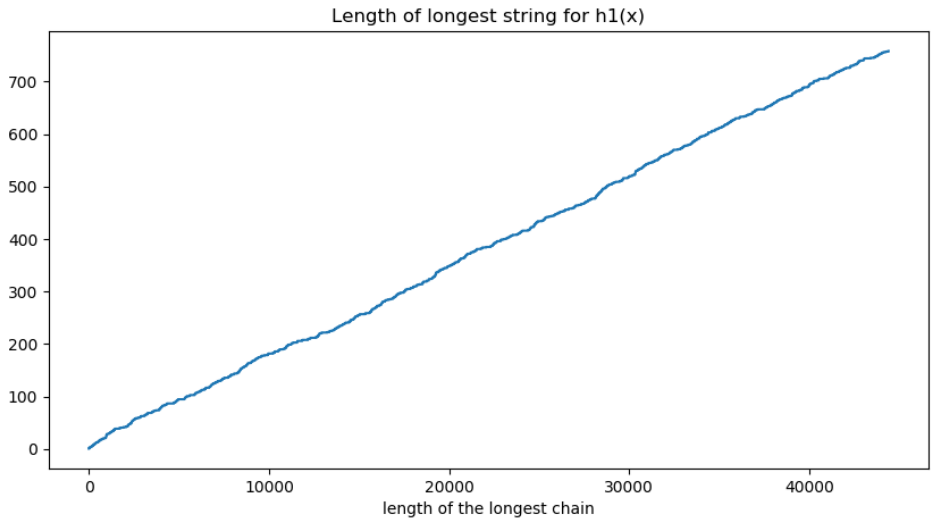
This histogram displays the distribution of keys when we use a universal hashing function. The keys are much more spread out through the data which is much better for efficiency because there are fewer collisions. The maximum number of keys in a position is 15, which is much less than the 700 of the previous hash function. Therefor, hash function h2(x) is much more effective.

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

behavior of uniform hashing.

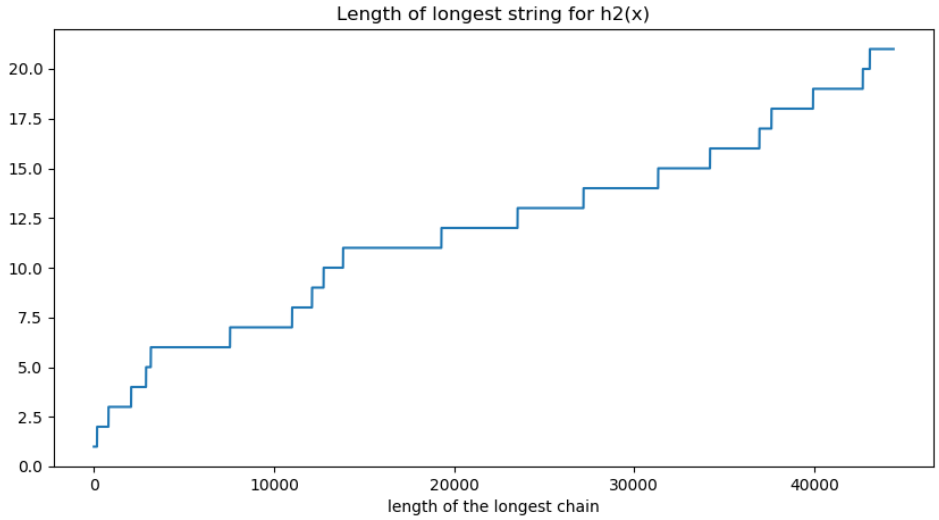
1. With the first hash function, there is no way to separate two strings that have the same hash value, they will always cause a collisions. However, the universal hash function can give them separate hash values because of the random multiple.
2. Most of the strings we are hashing are short so they end up having around the same hash value, which ends up stacking lots of values into the first section of the hash table and leaving the rest empty. The universal hash function does a better job of spreading the data around.
3. When searching the first hash table, almost every hash key position has multiple elements in it. This means that in order to find the correct element, it would take longer than constant time to traverse whereas the universal hash function is much more spread out so it takes less time to traverse.
4. The first hash function only uses 5% of the hash table, which causes more collisions because it is not a near uniform distribution. The second hash function is much closer to uniform hashing so it there are much fewer collisions.

(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.*

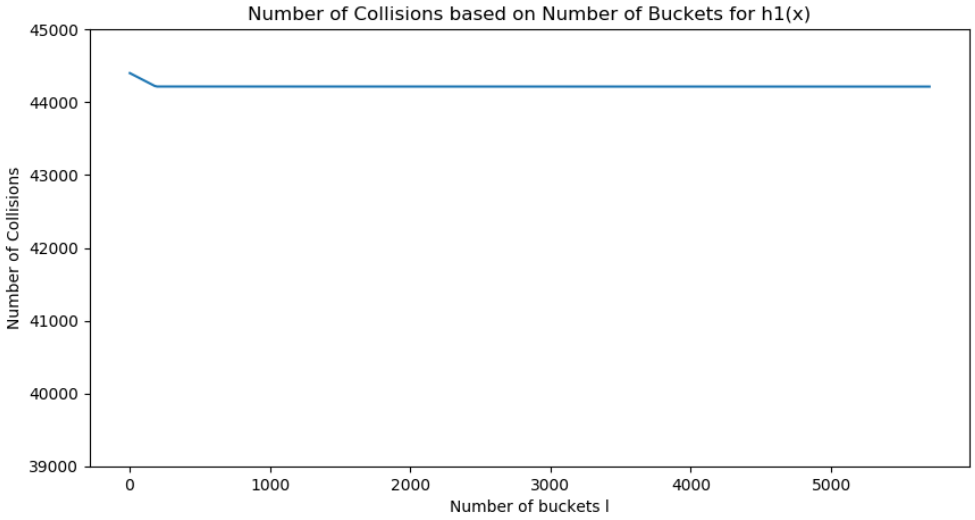


The plot above shows the length of the longest chain as the number of strings hashed using $h1(x)$ increases. The length of the longest chain increases near constantly to a max of over 700 because there are more collisions as the number

of inputs increases even though the number of buckets stays the same.

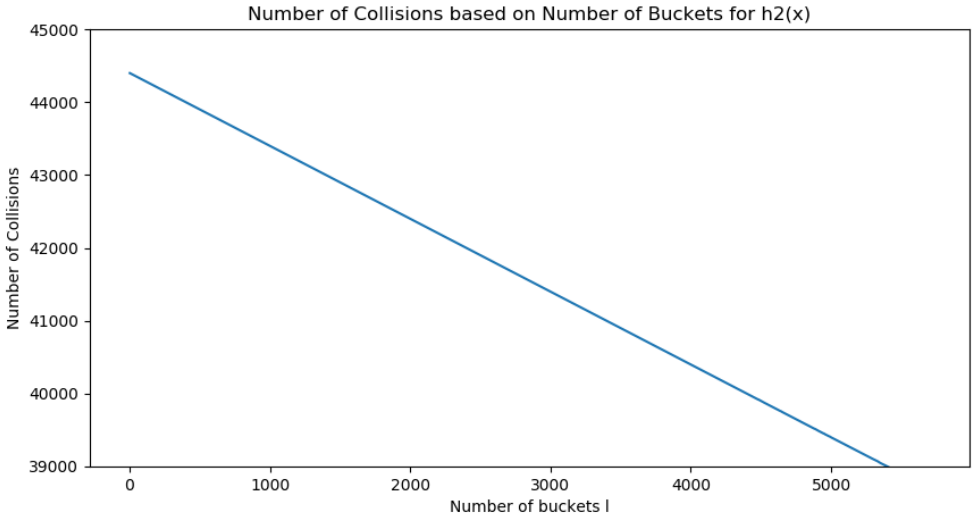


The plot above plots the length of the longest string as the number of strings hashed using $h2(x)$ increases. Similar to the first plot, it does increase as the number of plots increase. However, the graph maxes out at around 20 for the longest string where $h1(x)$ maxed out at 800. This means the values are more spread out through the hash table so there are less collisions.



The plot above shows the number of collisions depending on how many "buckets" are in the hash table for hash function h1(x). There is a small dip in the

beginning but soon after the amount of collisions is constant. This is because all of the strings are hashed into the very beginning of the hash table, so even as the size increases, the actual places that the strings are placed are not changing, so they will always cause collisions no matter how large the hash table.

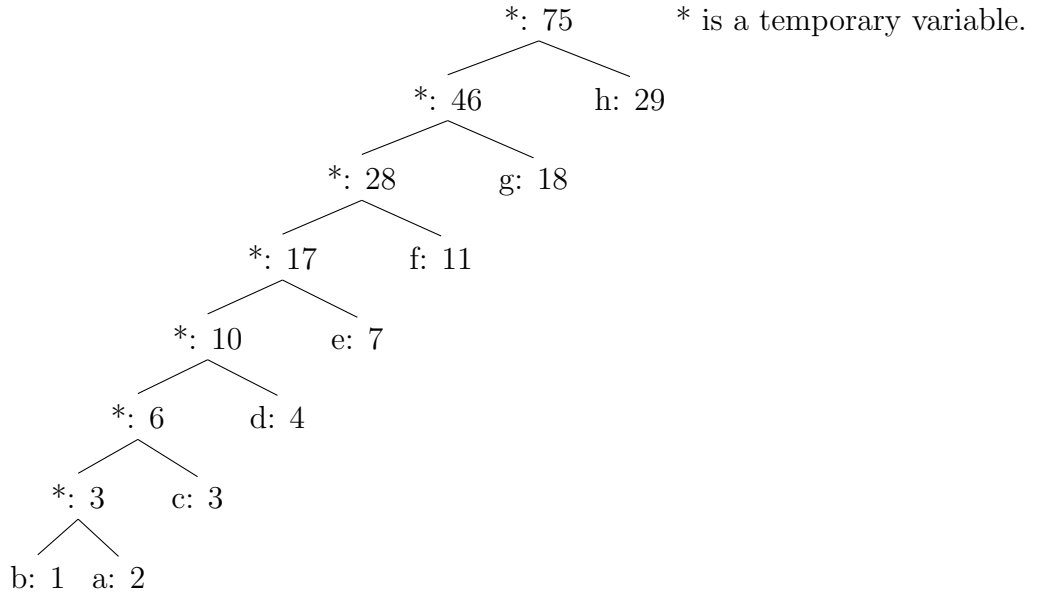


The plot shows the number of collisions depending on how many "buckets" are in the hash table for hash function h2(x). The number of collisions decreases linearly as the number of buckets is increased. This is because the hash table is near uniformly spread so, as the number of buckets increase (ℓ increases), there are more places to store the strings and there are less collisions overall. The more buckets, the more space there is to store everything.

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.*



Huffman Codes for each letter:
a: 0000001
b: 0000000
c: 000001
d: 00001
e: 0001
f: 001
g: 01
h: 1

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

The tree will continue to follow this trend as the number of Lucas numbers increases. The codes for all the values will add a zero for each Lucas number, and the maximum length code for b will have $|\sum| - 1$ zeroes. All codes for all letters besides b end in 1 so it will always be possible to decipher the codes. The first two Lucas Numbers (when n = 1 and n = 2) are located on the first level ($\ell = 1$). Every following number is located on the level beneath the number it is on (ℓ = n-1). Using this method, it's easy to find every code for all of the letters.