There are 55 regular points available on this problem set.

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 $n$.*

   In-order traversals of a tree $T_1$ always take $m$ steps (the number of nodes in the tree), and for each node visited in the traversal, it costs $\Theta(\log n)$ time to search for that node's value in the other balanced binary tree $T_2$. Hence, the overall cost is the product of these two costs: $\Theta(m \log n)$.

   (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.*

   Any traversal of the tree $T_2$ will behave equivalently, and for each node visited, we hash its key into $H_1$ and check whether it is present.
   Because $H_1$ has size $m$, and we inserted $m$ elements into it, its load factor will be $\alpha = O(1)$. Under a uniform hash function, the cost of each insert is $\Theta(1 + \alpha) = \Theta(1)$. Hence, inserting all of the elements of $T_1$ into the hash table $H_1$ costs $\Theta(m)$. And, checking whether each element of $T_2$ is already in the hash table costs $\Theta(n)$. Since these operations are performed sequentially, the total cost is their sum: $\Theta(m + n)$.

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 $\Sigma = [\texttt{A},\ldots,\texttt{Z}]$, and let the function $f(x_i)$ return the index of a letter $x_i \in \Sigma$, e.g., $f(\texttt{A}) = 1$ and $f(\texttt{Z}) = 26$. Finally, for an $m$-character string $x \in \Sigma^m$, define $h1(x) = ([\sum_{i=1}^{m} f(x_i)] \bmod \ell)$, where $\ell$ is the number of buckets in the hash table. For the other hash function, let the function $f2(x_i, a_i)$ return $f(x_i) * a_i$, where $a_i$ is a uniform random integer, $a_i \in [\texttt{0},\ldots,\ell\texttt{-1}]$, and define $h2(x) = ([\sum_{i=1}^{m} f(x_i, a_i)] \bmod \ell)$. 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 $\ell$ 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 $h(x)$.*

(b) *Produce a histogram showing the corresponding distribution of hash locations when $\ell = 5701$. Label the axes of your figure. Brief description what the figure shows about $h(x)$; justify your results in terms of the behavior of $h(x)$. Do not forget to append your code.*

Solution for (a) and (b):
The data file includes some extraneous information that we need to strip out before we can perform our hashing. Similarly, we need to randomize the order in which we hash the strings, and use only half of that order. One useful insight is that we do not have to actually create or use a real hash table to study the behavior of $h(x)$. Instead, we can use $h(x)$ directly to increment the locations of the histogram, which we'll represent using an array.

Here is the pseudo code for creating the histogram:

```
x = textread('dist.all.last.txt', '%s')   % read the data
s = x[1:4:end]                            % extract the name strings

ell =  5701                     % number of locations
ht1  = [0]*ell          % our hash locations
sr  = s[randperm(length(s))]  % randomize the order we hash the names
for t in range(length(sr)/2):    % hash only half the names
```

```
    t = 1+mod(sum(double(sr[i])-64),ell)  % h1(x)
    ht1[t] = ht1[t]+1                      % increment a hit on h1(x)

ht2  = [0]*ell
for i in range(length(sr)/2):     % hash only half the names h2
       a = randi(ell-1)
        t = 1+mod(sum((double(sr[i])-64)*a),ell) % h2(x)
        ht2[t] = ht2[t]+1
```

The histogram for $h_1(x)$itself (see below) is strikingly non-uniform, indicating that $h_1(x)$ is a poor hash function relative to the ideal of uniform hashing.

The histogram for $h_2(x)$itself (see below) is uniform, indicating that $h(x)_2$ is a good hash function relative to the ideal of uniform hashing.

If $h_1(x)$ were ideal, it would place roughly an equal number of inputs in each bucket. The data file contained 88,799 strings, and because we are only hashing half of them, the ideal would be about 8 strings in each location. Instead, $h_1(x)$ places most strings in the range 50–100, and under-utilizes all the remaining locations. This happens because the average length of a string is only 6.64 characters, and since the average value of $f(x_i)$ is around 13 (do you see why?), the distribution of $h_1(x)$ will be strongly peaked around 85 or so.

(c) *Enumerate at least 4 reasons why $h_1(x)$ is a bad hash function relative to the ideal behavior of uniform hashing.*

The reasons are many; here are several. The key consideration is how a uniform hash function treats different kinds of inputs, and how $h_1(x)$ behaves differently.

- For $h_1(x)$, the location of the letters within $x$ doesn't matter, only their individual values. Thus, any two strings that are anagrams (permutations) of each other will have the same hash value, e.g., $x =$ `SAUCE` and $y =$ `CAUSE` will have the same hash value.
- We can also add an arbitrary value to one character and subtract it from another character, and then rearrange the letters to get a different string with the same score as the original string. For instance, add 1 to `C` to make a `D`, then subtract 1 from `U` to make a `T`, and finally permute the letters to convert `CAUSE` to `DATES`.

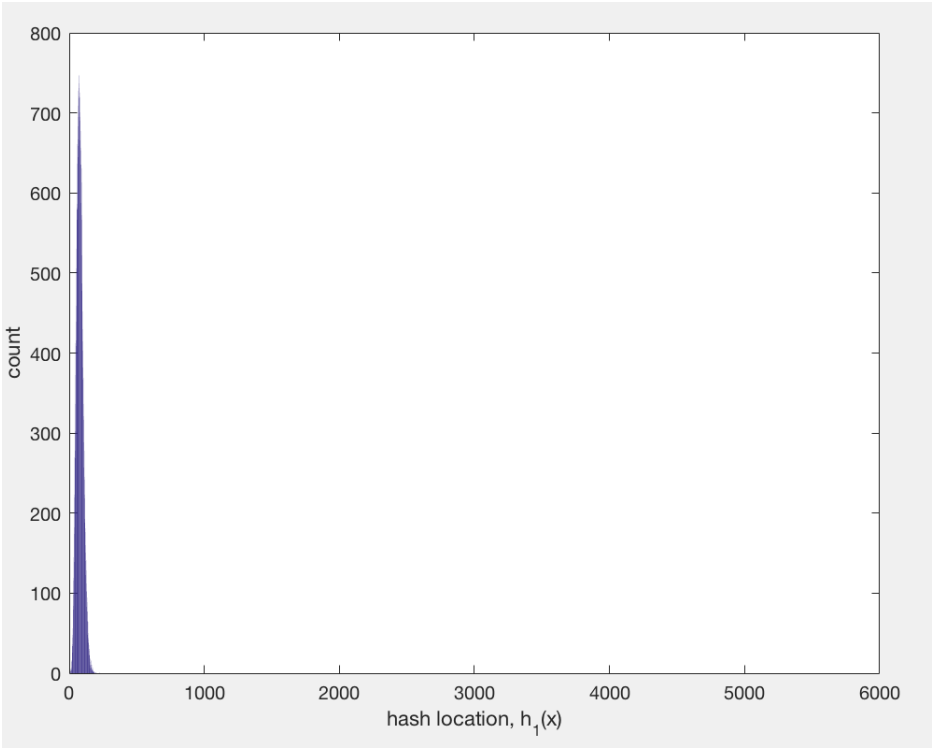



Figure 1: 2(a) The histogram for $h_1$.

- We can repeat the previous attack in various combinations to get a string that contains none of the original characters, e.g., `GIDDY`.
- In fact, strings of different lengths can also have the same hash value, e.g., `FRY` and `HUT`, or even `ABDICATED`.
- The last four points are ways we can demonstrate that $h_1(x)$ does not behave like a uniform hash; if it did, then the probability that a string $x$ would hash to the same location as some $y$ is $1/\ell$, meaning that "nearby" strings should not hash to nearby (or the same) location, and neither should "distant" strings.
- For strings with a typical length, like surnames or English words, some hash values will be improbable, in contrast to a uniform hash. For instance, only one string has a value of 1 (`A`), two have values of 2 (`AA` and `B`), four with value 3 (`AAA`, `AB`, `BA`, `C`), etc. (ignoring the mod function).
- Even if we focus only on strings with a fixed length, some characters are simply more common than others (the most common letters in English are `etaoin`

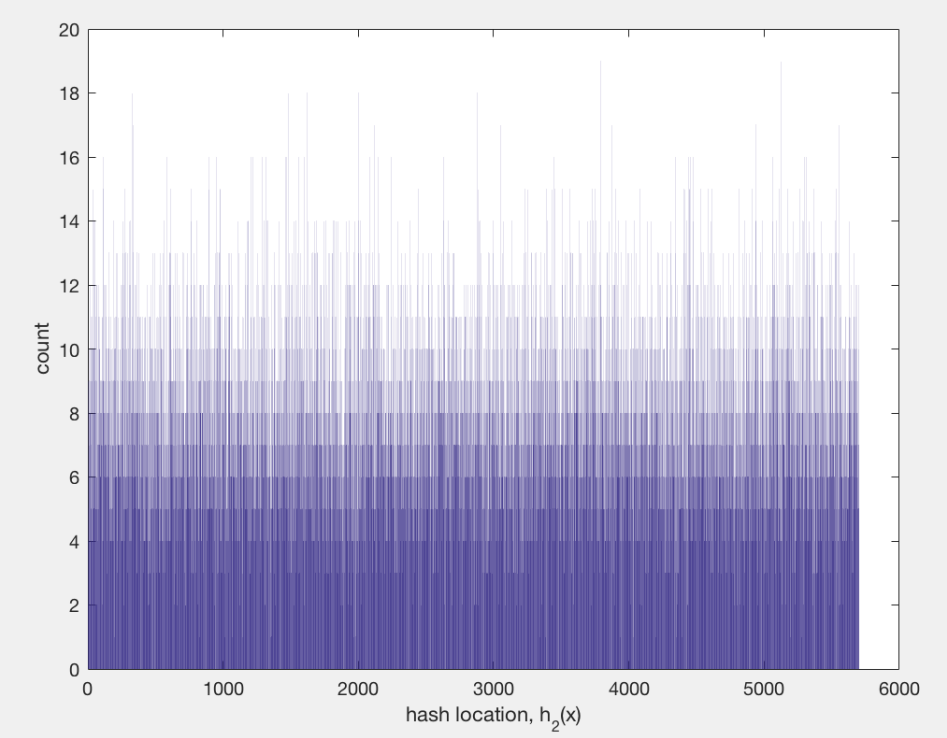



Figure 2: 2(a) The histogram for $h_2$.

`shrdlu`), and this will cause $h_1(x)$ will favor certain values over others.

- Strings that have a low "distance" from each other (having only 1 or 2 characters different) will hash to similar locations, while an ideal hash function would choose two random locations for a pair of inputs regardless of how close or far they are from each other.
- Etc.

(d) *(i) Produce a plot showing 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.*
*(ii)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.*

(i)The target function is simply the maximum value of the histogram, as a function

of the $n$ strings we hash from our data file. Here is the pseudo code that does this:

```
ell =  5701                  % number of locations
ht  = [0]*ell        % our hash locations
sr  = s[randperm(length(s))] % insert strings randomly
mx  = [0]*length(s)     % max hash count vs. n items hashed
for n in range(length(sr)):
    t = 1+ mod(sum(double(sr[n])-64),ell) % h1(x)
    ht[t] = ht[t]+1                      % increment a hit on h1(x)
    mx[n] = max[ht]                      % longest chain after n hashes
```

```
ell =  5701                 % number of locations
ht  = [0]*ell         % our hash locations
sr  = s[randperm(length(s))]  % insert strings randomly
mx2  = [0]*length(s)        % max hash count vs. n items hashed
for n in range(length(sr)):
     a = randi(ell-1)
      t = 1+mod(sum((double(sr[i])-64)*a),ell)  % h2(x)
     ht2[t] = ht2[t]+1                      % increment a hit on h(x)
      mx2[n] = max[ht2]                      % longest chain after n hashes
```

Then, plotting `mx` and `mx2` as a function of `n` yields the desired function (see Figures below).

(ii) Here is the pseudo code that does this:

```
p = primes(100000);% pick prime number less than 100000

maxl = 100
counts1 = [0]*maxl
for ell in range(maxl):
    counts1(ell) = 0             % counts the number of collisions
    ht  = [0]*ell          % our hash locations
    sr  = s[randperm(length(s))]  % insert strings randomly
    for n in range(length(sr)):
        t = 1+mod(sum(double(sr[n])-64),ell); % h1(x)
        if ht[t] != 0:
            counts1[ell]= counts1[ell]+1; %number of
```

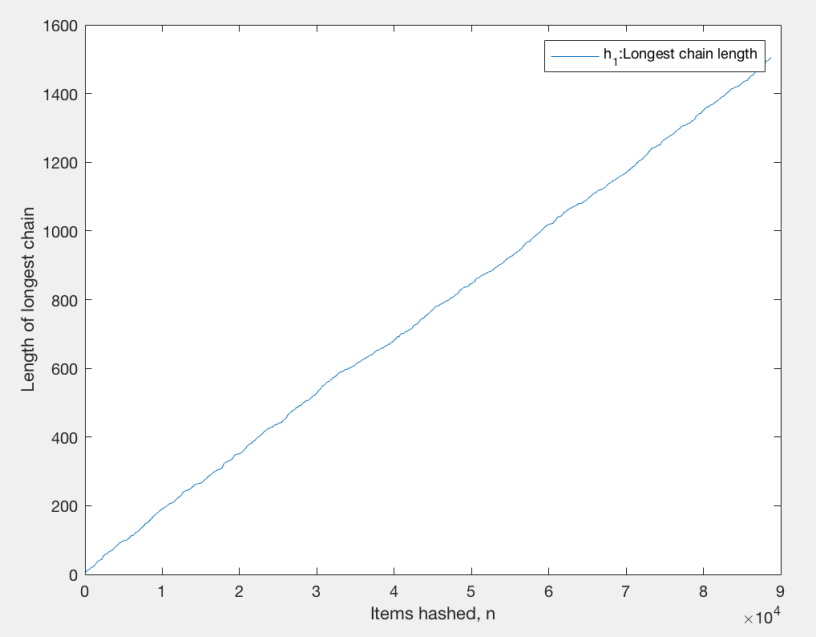



Figure 3: 2(d)i longest chain for $h_1$.

```
        ht[t] = ht[t]+1 % increment a hit on h(x)
```

```
maxl2 = 100
counts2 = [0]*maxl2
for ell in range(maxl2):
    counts2[ell] = 0             % counts the number of collisions
    ht2  = [0]*p[ell]           % our hash locations
    sr  = s[randperm(length(s))]  % insert strings randomly

   for n in range(length(sr)):
        a = randi(ell)
        t = 1+mod(sum((double(sr[i])-64)*a),p[ell]) % h2(x)
        if ht2[t] != 0:
            counts2[ell]= counts2[ell]+1
        ht2[t] = ht2[t]+1
        % increment a hit on h(x)
```

Then, plotting `counts1` and `counts2` as a function of `n` yields the desired function

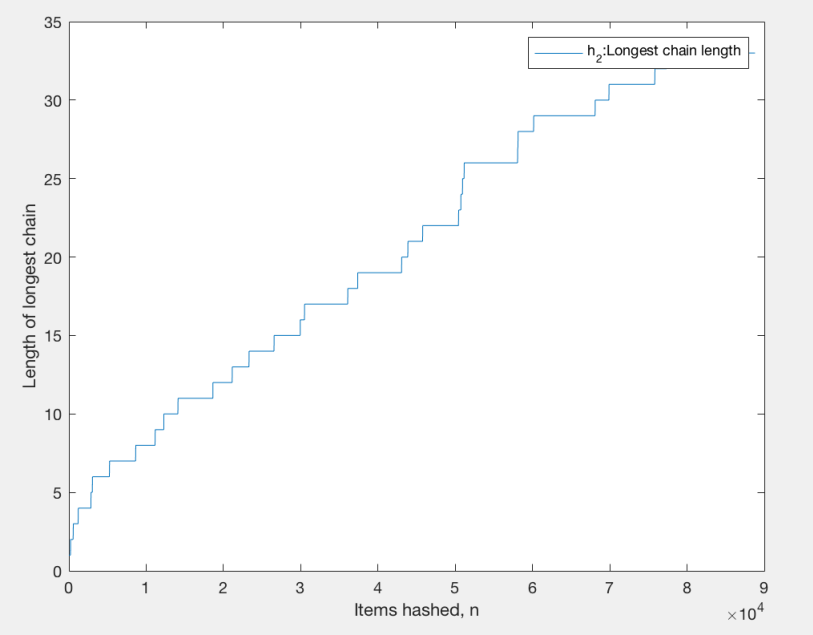



Figure 4: 2(d)i longest chain for $h_2$.

(see Figures below).

Both functions are decreasing as $l$ increases at first. But $h_1$ stops decreasing when $l$ is large enough while $h_2$ keeps decrease.

3. (15 pts total) *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* $L_n = L_{n-1} + L_{n-2}$ *for* $n > 1$ *with base cases* $L_0 = 2$ *and* $L_1 = 1$.

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

   Evaluating the Lucas number formula for up to $n = 8$ yields the frequencies of the symbols in $\Sigma$:

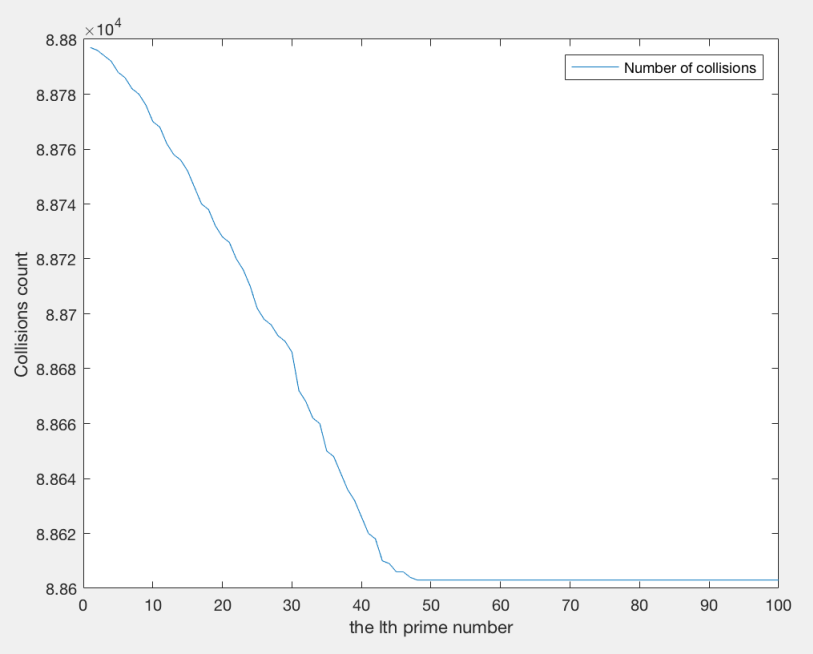



Figure 5: 2(d)ii collision count for $h_1$.

$a$:2 $b$:1 $c$:3 $d$:4 $e$:7 $f$:11 $g$:18 $h$:29

The Huffman algorithm constructs the encoding tree from the bottom up, iteratively merging the two symbols with the lowest two frequencies to produce a new symbol with their combined frequency. Applying this algorithm to the frequencies we obtained produces the following encoding tree:

An optimal code is then given by tracing each of the $n$ paths from root to a unique leaf and reading off the branch labels:

| a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|
| 0000000 | 0000001 | 000001 | 00001 | 0001 | 001 | 01 | 1 |

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

Visually, our encoding tree has a highly regular pattern to its structure: except for the first one, every codeword is a (possibly empty) sequence of 0s followed by

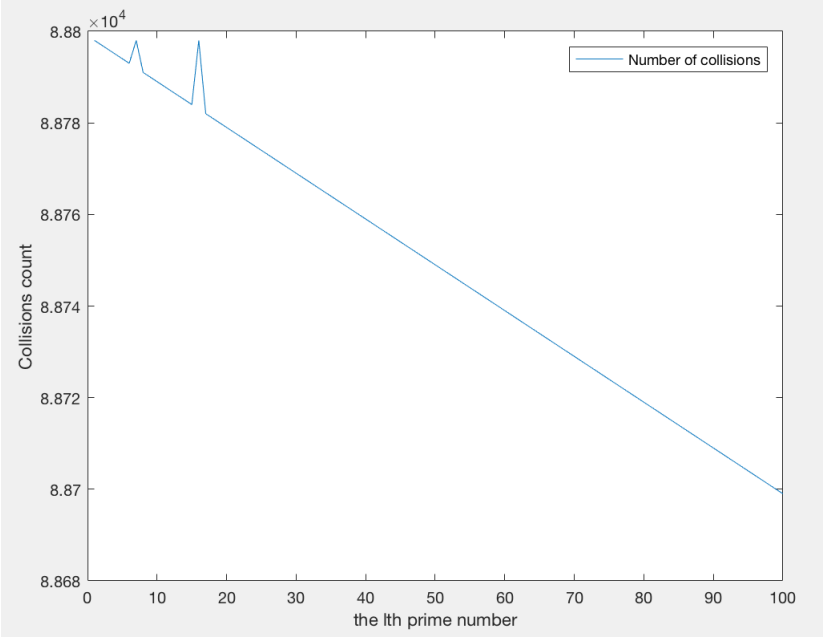
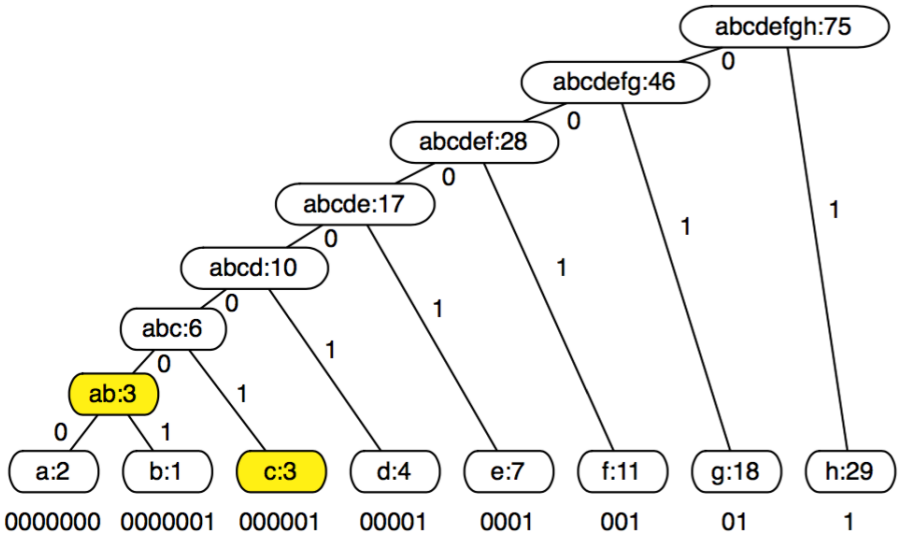



Figure 6: 2(d)ii collision count for $h_2$.



a single 1. Thus, we may generalize our encoding as follows.

Let $n$ be the number of input symbols, and let $\ell$ index the symbols in sorted order (least- to most-frequent); thus, $\ell = 3$ is "$c$" and $\ell = 5$ is "$e$". The $\ell = 1$th symbol is assigned the codeword $0^{n-1}$, where the string notation $x^y$ denotes the symbol $x \in \{0, 1\}$ repeated $y \geq 0$ times (not mathematical exponentiation). For all $\ell > 1$, the codeword assigned to the $\ell$th symbol is $0^{n-1-\ell}1$.

*(optional explanation)* This generalization holds because Lucas numbers $L_n$ have the property that $\sum_{i=1}^{n} L_n = L_{n+2} - 1$. (Fibonacci numbers also have this property.) Thus, the sum of the frequencies of the first $n$ Lucas numbers, i.e., the combined frequency of the first $n$ symbols, is always less than the frequency of the $(n+2)$th symbols. In the Huffman algorithm, this means we always merge the $n$th and $(n+1)$th symbols in the tree, producing a tree with depth exactly $n-1$ in which the path from the root to the $\ell$th symbol will traverse exactly $n-1-\ell$ left-hand branches, followed by either another left-hand branch (for $\ell = 1$) or a right-hand branch (for $\ell > 1$).