CSCI 3104 Spring 2018 Problem Set #5

Alison Ostlund (05/22)

Worked with: Kyle Ward, Matt Donovan, Jason Lubrano, Hannah Haines

---

1. (15 pts) Bellatrix Lestrange is writing a secret message to Voldemort and wants to prevent it from being understood by meddlesome young wizards and Muggles. She decides to use Huffman encoding to encode the message. Magically, the symbol frequencies of the message are given by the Pell numbers, a famous sequence of integers known since antiquity and related to the Fibonacci numbers. The nth Pell number is defined as $P_n = 2P_{n1} + P_{n2}$ for $n > 1$ with base cases P0 = 0 and P1 = 1.

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

   Using Pells numbers:
   $P_n = 2P_{n1} + P_{n-2}$ for $n > 1$
   a=1
   b= 2(a) + 0 = 2
   c = 2(b) + a = 5
   d = 2(c) + b = 12
   e = 2(d) + c = 29
   f = 2(e) + d = 70
   g = 2(f) + e = 169
   h = 2(g) + f = 408
   Using the Tree below:
   a = 0000000
   b = 0000001
   c = 000001
   d = 00001
   e = 0001
   f = 001
   g = 01
   h = 1

```
                696
              0 /  \1
              288   h
             0/  \1
            119   g
           0/  \1
          49    f
        0/  \1
       20    e
      0/ \1
     8    d
   0/ \1
  3    c
0/ \1
a   b
```

(b) Generalize your answer to (1a) and give the structure of an optimal code when the frequencies are the first n non-zero Pell numbers.

In $\sum_{i=1}^{n}(a,b,c,d,e,f,g,h)$
n = size of the set, i = position
P(n) = n - i zeros and bits - base case
if i > 1,
n-i + 1 bits
n-1 zeros with + 1 added at the end

2. (30 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 hash function. Let U be the universe of strings composed of the characters from the alphabet $\sum = [A, ..., Z]$, and let the function f(xi) return the index of a letter $x_i \epsilon \sum$, e.g., f(A) = 1 and f(Z) = 26. Finally, for an m-character string x $\epsilon \sum^{m}$, define h(x) = $\sum_{i=1}^{m} f(x_i) mod(l)$, where l is the number of buckets in the hash table. That is, our hash function sums up the index values of the characters of a string x and maps that value onto one of the l buckets.

(a) The following list contains US Census derived last names: http://www2.census.gov/topics/genea
Using these names as input strings, first choose a uniformly random 50 percent of the name strings and then hash them using h(x). Produce a histogram showing the corresponding distribution of hash locations when l = 200. Label the axes of your figure. Briefly describe what the figure shows about h(x), and justify your results in terms of the behavior of h(x). Do not forget to append your code. Hint: the raw file includes information other than 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.

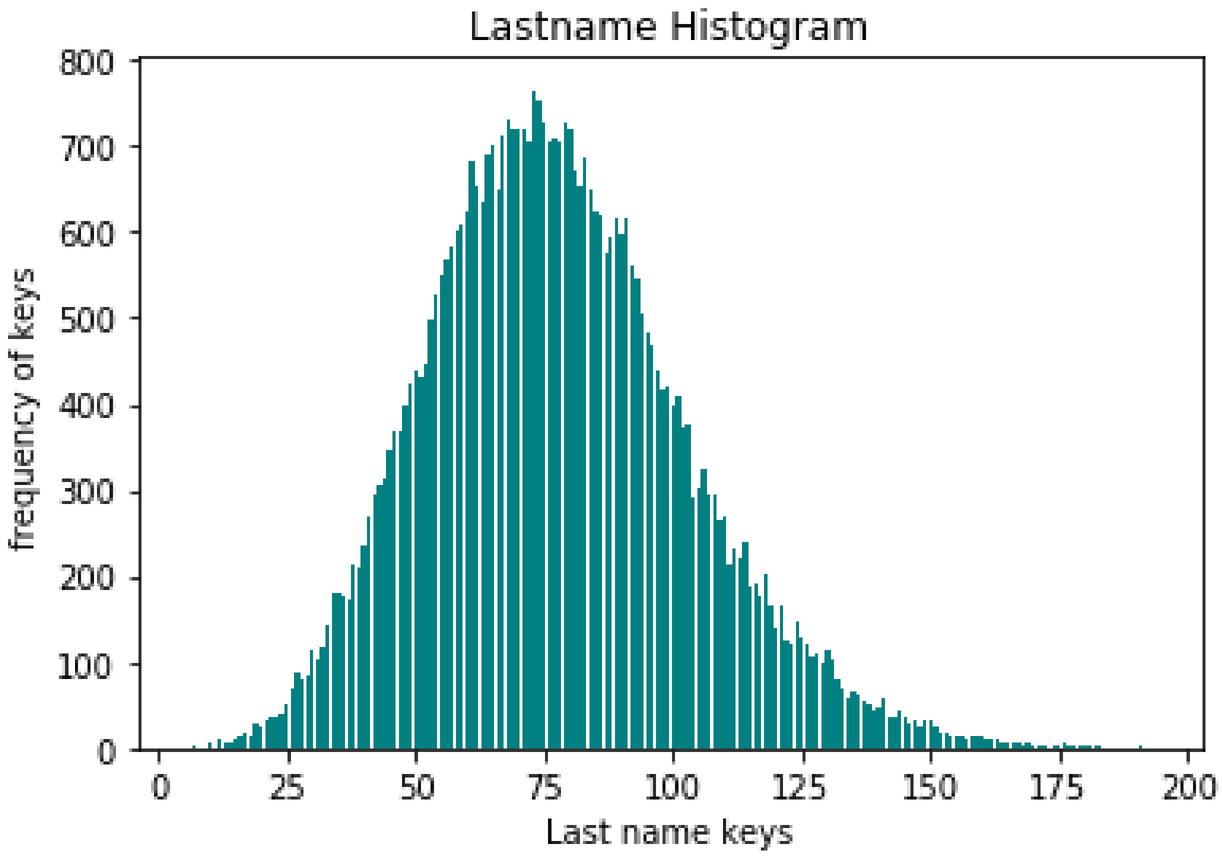
```
lastnames = []
hash = {}
length = []
largest = 0
alphabet = {'A':1,'B':2,'C':3,'D':4,'E':5,'F':6,'G':7,'H':8,'I':9,'J':10,'K':11

with open('lastnames.txt') as file:
    for line in file:
        lastnames.append(line.split()[0])

random.shuffle(lastnames)
split = int(len(lastnames)/2)
splitnames = lastnames[:split]


for name in splitnames:
    count = 0
    for x in name:
        count += alphabet[x]
    key = count % 200
    if key in hash:
        h_names[key] +=1
        if hash[key] > largest:
            largest = hash[key]
    else:
        hash[key] = 1
        if hash[key] > largest:
            largest = hash[key]
    length.append(largest)

 -Plotting the Histogram
plt.bar(hash.keys(), hash.values(), color='teal')
plt.title('Lastname Histogram')
plt.xlabel('Last name keys')
plt.ylabel('Frequency of keys')
plt.show()
```



This figure shows that h(x) is a bad representation of a histogram because the length of the last names are not evenly distributed creating a bell curve rather than the ideal form of a histogram which would be uniform. For an ideal hash function, we would want the ”boxes” to be evenly filled.

(b) Enumerate at least 4 reasons why h(x) is a bad hash function relative to the ideal behavior of uniform hashing.

h(x) is a bad hash function to an ideal uniform hash function:
1. There are some letters that occur more frequently than others, resulting in similar keys
2.Some last names are similar sizes so we are adding up a similar size of numbers giving a similar sum, normalizing the key.
3. Certain groups of letters occur more frequently together in the English language

than others.
4. The number of bins is too small for the input size, 200 bins for 45,000 names. The more bins there is the less inputs per bin there will be, ultimately creating a uniform histogram.

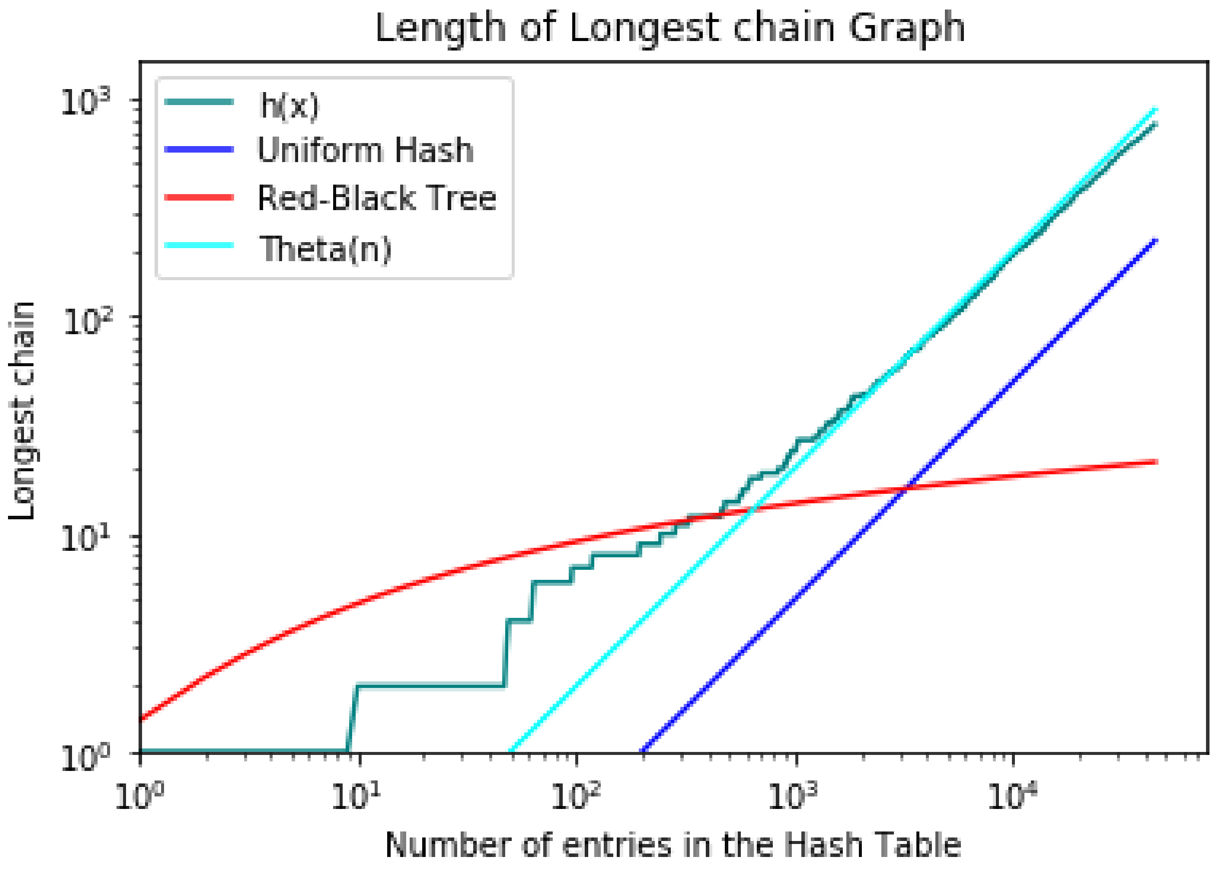
(c) Produce a plot showing (i) the length of the longest chain (were we to use chaining for resolving collisions under h(x)) as a function of the number n of these strings that we hash into a table with l = 200 buckets, (ii) the exact upper bound on the depth of a red-black tree with n items stored, and (iii) the length of the longest chain were we to use a uniform hash instead of h(x). Include a guide of c n Then, comment (i) on how much shorter the longest chain would be under a uniform hash than under h(x), and (ii) on the value of n at which the red-black tree becomes a more efficient data structure than h(x) and separately a uniform hash.

```
 -Plotting the Red Black, Theta, and Uniform Vs Hash Table
index = []
index.extend(range(1, 44400))

uniformHash =[]
redBlack =[]
theta = []

for i in index:
    if i/200 < 1:
        uniformHash.append(0)
    else:
        uniformHash.append(i/200)
    redBlack.append(2 * np.log(i+1))
    theta.append(i/50)

hash_table, = plt.loglog(index, length, color='teal', label='Hash Function')
uniformHash_table, = plt.loglog(index, uniformHash, color='b',label='Uniform Ha
redBlack_table, = plt.loglog(index, redBlack, color='r',label='Red-Black Tree')
thetan, = plt.loglog(index,theta,color='aqua',label='Theta(n)')
plt.title('Length of Longest chain Graph')
plt.legend([hash_table, uniformHash_table, redBlack_table, thetan], ['h(x)', 'U
plt.xlim(1)
plt.ylim(1)
plt.xlabel('Number of entries in the Hash Table')
plt.ylabel('Longest chain')
plt.show()
```



3. Draco Malfoy is struggling with the problem of making change for n cents using the smallest number of coins. Malfoy has coin values of$v1 < v2 < \ < vr$for r coins types, where each coins value vi is a positive integer. His goal is to obtain a set of counts d_i, one for each coin type, such that $\sum_{i=1}^{r}(d_i = k)$ and where k is minimized.

(a) A greedy algorithm for making change is the cashiers algorithm, which all young wizards learn. Malfoy writes the following pseudocode on the whiteboard to illustrate it, where n is the amount of money to make change for and v is a vector of the coin denominations:

```
wizardChange(n,v,r) :
   d[1 .. r] = 0 // initial histogram of coin types in solution
   while n > 0 {
       k = 1
       while ( k < r and v[k] > n ) { k++ }
       if k==r { return no solution }
       else { n = n - v[k] }
       }
 return d
```

Hermione snorts and says Malfoys code has bugs. Identify the bugs and explain why each would cause the algorithm to fail.

Instead of v[k] > n, should be v[k]<n. Should check to see if $k > 0$. If it is, then there is no match. k ==r incorrect should return no solution if v[i] > r. Should add d[k] += 1 in else statement to decrement the coins.

(b) Sometimes the goblins at Gringotts Wizarding Bank run out of coins,1 and make change using whatever is left on hand. Identify a set of U.S. coin denominations for which the greedy algorithm does not yield an optimal solution. Justify your answer in terms of optimal substructure and the greedy-choice property. (The set should include a penny so that there is a solution for every value of n.)

Given n set of denominations (d,...,d_i)
n = amount wanted
d = index of greatest denomination
$(n - v[d]) < (v[d-1])$. The largest denomination index value is less than the wanted amount,n . If this is true, then the greedy algorithm will not work. This is the situation where there is a less smaller amount of coins for more money.

(c) (8 pts extra credit) On the advice of computer scientists, Gringotts has announced that they will be changing all wizard coin denominations into a new set of coins denominated in powers of c, i.e.,$c^0, c^1, ..., c^l$ denominations of c for some integers c > 1 and l ≥ 1. (This will be done by a spell that will magically transmute old coins into new coins, before your very eyes.) Prove that the cashiers algorithm will always yield an optimal solution in this case. Hint: first consider the special case of c = 2.

Suppose c = 2, then :
$2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8..., 2^l = 2 * 2^{l-1} = 2^l$

same holds for c =3:
$3^0 = 1, 3^1 = 3, 3^2 = 9, 3^3 = 27..., 3^l = 3 * 3^{l-1} = 3^l$

The greedy algorithm would always be correct. If $l + 3$ coin value is needed, then they would give $n * n * n^l$. When they could just give n^{l+3} coin denominations.