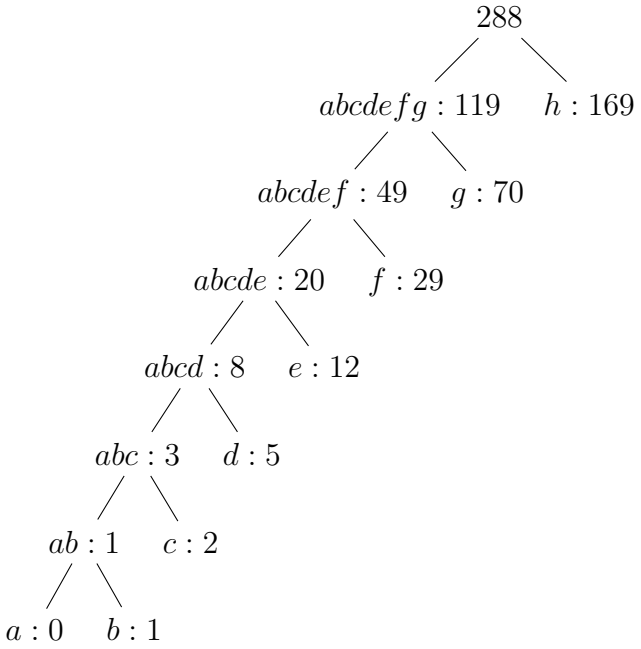

Problem 1

(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 ***n****th Pell number is defined as* $P_n = 2P_{n-1} + P_{n-2}$ *for* $n > 1$ *with base cases* $P_0 = 0$ *and* $P_1 = 1$.

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



Element	Huffman code
a	0000000
b	0000001
c	000001
d	00001
e	0001
f	001
g	01
h	1

(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.*

From (1a) we can see that if we have n elements, then the first element P_0 has $n-1$ zeros, the second element has $n-2$ zeros and 1 at the end [for example if $n=5$ then Huffman code for the second element is 0001].
The third element has $n-3$ zeros and 1 at the end, and so on until we reach the last element which Huffman code is 1.

```
In mathematical notation:
Let's say:
n:  the number of elements in the array(first n non-zero Pell numbers)
i:  the index of each element
[(n-i)0's]1 means:  starting from the left, there are n-i 0's and 1 at
the end(the first bit)
H(i):  Huffman code for the ith element
```

The first element is a special case, so $H(0) = (n-1)0's$
For $n \geq 1$, $H(i) = [(n-1-i)0's]1$

```
The reason:
```

These codes are prefix-free codes. The reason we have the above function for Huffman codes because all the elements are right children except the first element. We can see that the elements are in different levels: This explains the number of zeros.
And being right children explain the one at the end.

Problem 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 $\Sigma = [A, \ldots, Z]$, and let the function $f(x_i)$ return the index of a letter $x_i \in \Sigma$, e.g., $f(A) = 1$ and $f(Z) = 26$. Finally, for an $\boldsymbol{m}$-character string $x \in \Sigma^m$, define $h(x) = ([\sum_{i=1}^{m} f(x_i)] \bmod l)$, 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.

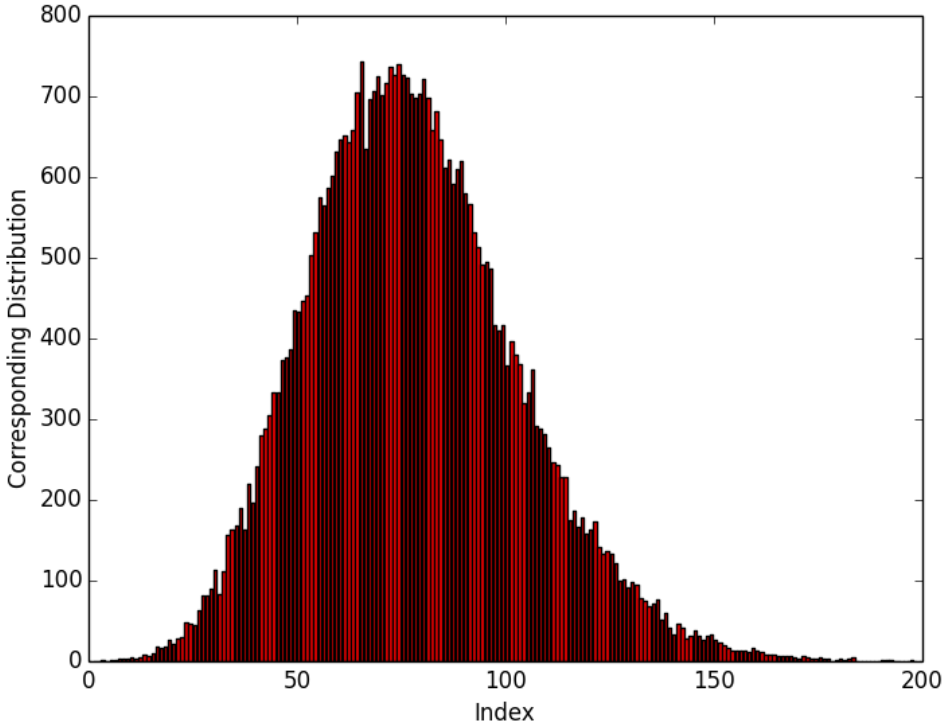
(a) *The following list contains US Census derived last names:*

`http://www2.census.gov/topics/genealogy/1990surnames/dist.all.last`

Using these names as input strings, first choose a uniformly random 50% of these 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.



In the histogram above, the values in the x axis are the indices or the buckets in the hash table, and the values in the y axis are the corresponding distribution or the frequencies in each bucket in the hash table.
In other words, this histogram shows how many strings are stored in each index(bucket) in the hash table.

We can see that elements in the indices from about 50 to 100 are repeated more than the elements in the indices less than 50 or more than 100.
The reason is because the average length of the strings is about 5 characters. When we calculate the sum of each character, the smallest possible value is 5, and largest possible value is 130. And the median value is about 67. And then we take the module of 200, which is the length itself.
Therefore, this is why the histogram takes this shape.

`NOTE: my code is appended at the end of the pdf file`

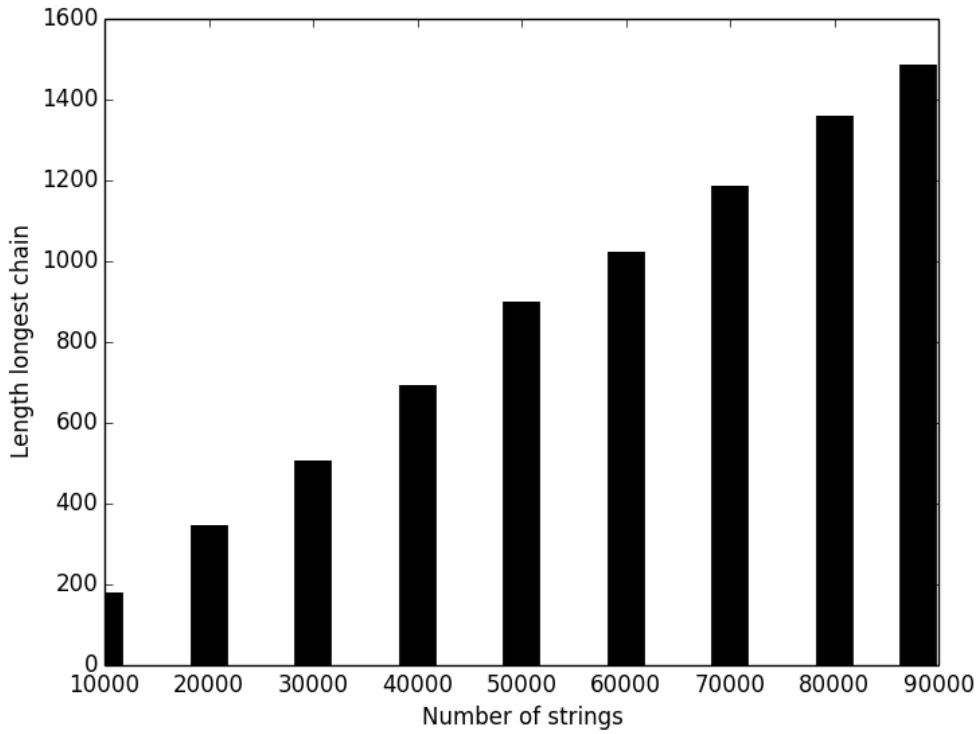
(b) *Enumerate at least 4 reasons why $h(x)$ is a bad hash function relative to the ideal behavior of uniform hashing.*

1. The hash function sums the values of the characters, and the location doesn't matter. Therefore, the order of the letters doesn't matter. Different strings can have the same index if they have the same letters but the order is different.
2. The length of the strings doesn't matter. Two different strings can have the same

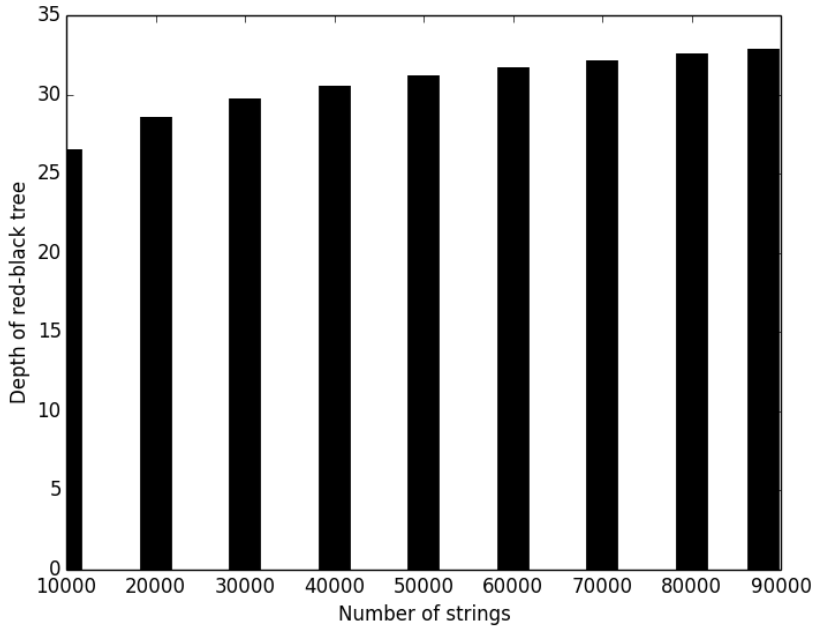
index while the length of one is greater than the other.
3. The frequencies of the letters differ. For example, we use the vowels more than other letters, such as x. And since some letters appear more than others, then same indices will occur more than others.
4. The alphabet array, which gives a value for each letter (1 for a, 2 for b, etc), is one of the reasons that $h(x)$ is a bad hash function. As I mentioned before that the frequencies differ from each letter and other, the value for each letter should be related to how much it occurs in general. 5. The value of l also in this particular problem can be related. If we choose l to be the median value, let's say 80, then the elements will be more distributed.

(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 cn*
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.

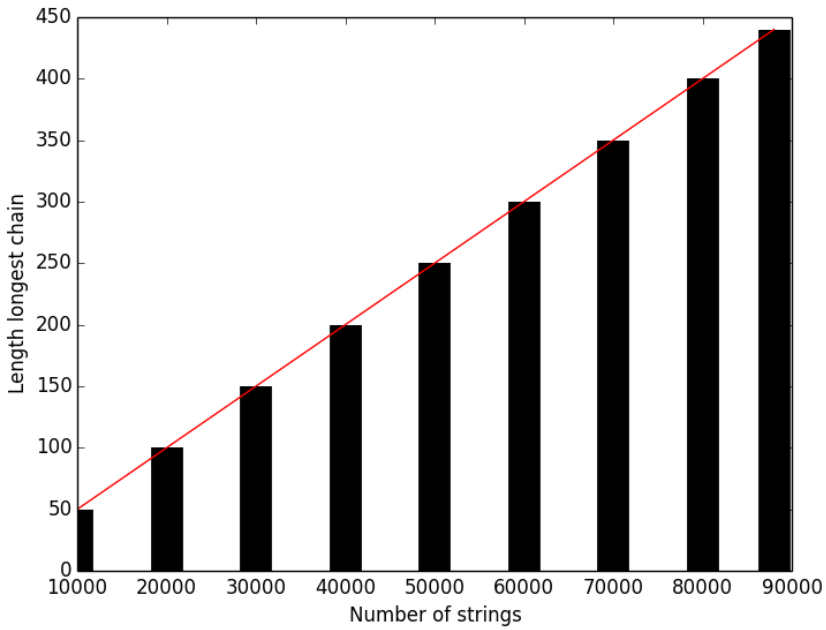
`(i) The plot is below, and the code is at the end of the pdf file.`



(ii) The plot is below, and the code is at the end of the pdf file.



(iii) The plot is below, and the code is at the end of the pdf file.



Since $l = 200$, then $cn = \frac{1}{200}n$

COMMENTS:

(i): If we compare the two graphs in (i) and (iii), we can see that the longest chain under a uniform hash function is almost 1/4 of the longest chain under h(x).

(ii): The upper bound of the depth of a red-black tree with n elements is $2\log_2(n+1)$. Therefore, when n becomes bigger and bigger, red-black trees become more efficient than h(x).

Problem 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 $v_1 < v_2 < \cdots < v_r$ for r coins types, where each coins value v_i 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:*

```
0    wizardChange(n,v,r) :
1      d[1 .. r] = 0
2      // initial histogram of coin types in solution
3     while n > 0 {
4        k = 1
5        while ( k < r and v[k] > n ) { k++ }
6        if k==r { return no solution }
7        else { n = n - v[k] }
8      }
9      return d
```

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

`line 5:` The condition in the while loop should be ($k < r$ and $v[k+1] \leq n$).
(i) '<' because we want to find the maximum element in the while loop that is less than n.
(ii) '$v[k+1]$' because if the next element is larger than n, then we want to stop and not increase the value of k.

`line 7:` We should increment the value of $d[k]$ by 1 because we are returning d and we want to modify it during the while loop.

(b) *Sometimes the goblins at Gringotts Wizarding Bank run out of coins 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.)*

If we have $v = [0.01, 0.10, 0.25]$ and $n = 0.30$.
When we go through the algorithm, it will choose 0.25 in the first iteration, and then choose 0.01 five times. The total number of coins will be 6, but it is not optimal. The optimal number of coins actually 3. [Choosing 0.10 three times].

(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., denominations of c^0, c^1, ..., c^l for some integers $c > 1$ and $l \geq 1$. (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 $c = 2$.

Considering the case $c = 2$ and from the examination of 3(b), we can say that if we have a sorted array of coins and the last element is the largest and if k_n is the last element, then the algorithm will give us an optimal solution if $k_n = 2k_{n-1}$
If $k_n > 2k_{n-1}$, same to what we saw in part b, then we will not get the optimal solution in many cases.
From this fact and assuming $c = 2$ we can use proof of induction to prove the algorithm will give us optimal solution if the denomination are $2^0, 2^1, \ldots, 2^l$.

`Base Case:` if $l = 1$, then we will have two coins, $[2^0, 2^1] = [1, 2]$. And these will yield to optimal solution since a number can be even or odd. Also, $2^1 = 2 * 2^0$

`Induction step:` Assume $l = k$, then $2^k = 2^{k-1} * 2$.
If we have $l = k + 1$, then
$2^{k+1} = 2^k * 2$
$2^k * 2 = 2^{k-1} * 2 * 2$ (divide by 2)
$2^k = 2^{k-1} * 2$

`Conclusion:` We proved that for $l = k + 1$ the algorithm will give us an optimal solution. Therefore, the algorithm will give an optimal solution for all the cases when $c = 2$

Problem 2a

```
#Assume textfile.txt, which has the values from the link in the problem,
 is modified and in the same directory.

from random import shuffle
from matplotlib.pyplot import *

def returnIndex(letter):
 #find the ascii value of a letter
 #knowing that 'a' start from 97, we subtract 96 to find the index
 index = ord((letter).lower()) - 96
 return index

def hashFunction(string, numBuckets):
 total = 0
 for i in range(0, len(string)):
 total += returnIndex(string[i])

 index = total % numBuckets

 return index

def readFile(filename):
 '''
 Read the file and store all strings in a list and return it
 '''
 l = []
 file = open(filename, 'r')
 for line in file:
 l.append(line.replace('\n', ''))

 return l



###############
#####main######
###############

#create a list that has all the strings from the textfile
all_string = readFile("textfile.txt")

#create a list that has 200 elements, and set each element to zero
```

```
#use it instead of creating a hash table
all_zero = [0]*200

#1. shuffle the elements in all_string array randomly
#2. choose first 50% elements of the array
#3. modify all_zero array so that each element is the number of repetitions of that
    index from the hash function
#from this array we can create a histogram
shuffle(all_string)
num_of_elements = len(all_string)/2

for i in range(num_of_elements):
 #index returned from the hash table for each string
 index = hashFunction(all_string[i], 200)
 all_zero[index] += 1;


y = all_zero
x = []
for i in range(200):
 x.append(i)

bar(x, y, color='y')
ylabel('Frequencies')
xlabel('Index')
show()
```

Problem 2c(i)

```
#Assume textfile.txt, which has the values from the link in the problem,
#is modified and in the same directory.

from random import shuffle
from matplotlib.pyplot import *

def returnIndex(letter):
        #find the ascii value of a letter
        #knowing that 'a' start from 97, we subtract 96 to find the index
        index = ord((letter).lower()) - 96
        return index

def hashFunction(string, numBuckets):
        total = 0
        for i in range(0, len(string)):
                total += returnIndex(string[i])


        index = total % numBuckets

        return index

def readFile(filename):
        '''
        Read the file and store all strings in a list and return it
        '''
        l = []
        file = open(filename, 'r')
        for line in file:
                l.append(line.replace('\n', ''))

        return l


all_string = readFile("textfile.txt")

nVals = [10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 88000]
chains = []
for n in nVals:
        shuffle(all_string)
        all_zero = [0]*200
        for i in range(n):
```

```
                index = hashFunction(all_string[i], 200)
                all_zero[index] += 1;
        #append max value from all_zero to chains array, which is the
         length of longest chain
        chains.append(max(all_zero))


y = chains
x = nVals

bar(x, y, color='r', linewidth=10)
ylabel('Length of the longest chain')
xlabel('n, number of strings')
show()
```

Problem 2c(ii)

```
from matplotlib.pyplot import *
import math

nVals = [10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 88000]
logn = []
for n in nVals:
        val = 2*math.log(n+1, 2)
        logn.append(val)


y = logn
x = nVals

bar(x, y, color='r', linewidth=20)
ylabel('Depth of red-black tree')
xlabel('Number of strings')
show()
```

Problem 2c(iii)

```
from matplotlib.pyplot import *
import math

nVals = [10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 88000]
chains = []
for n in nVals:
        val = n/200
        chains.append(val)


y = chains
x = nVals

bar(x, y, color='r', linewidth=20)
ylabel('Length longest chain')
xlabel('Number of strings')
plot(nVals, chains, color='r')
show()
```

I worked with:
Omar Mohammed, Mahmoud Almansouri, Firas Al Mahrouky, Yazeed Almuqwishi, Absdullah Bajkhaif