1. (15 points) Shadow is writing a secret message to Harry and wants to prevent it from being understood by Thormund. 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.

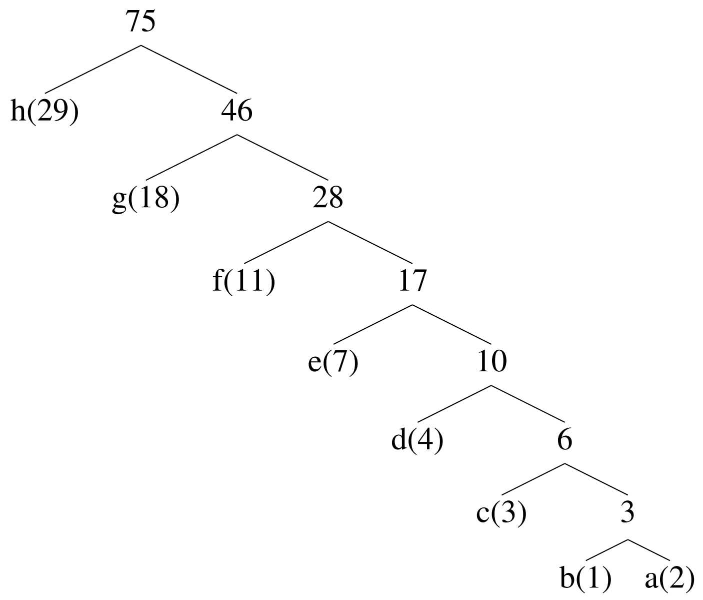
   (a) For an alphabet of $\Sigma = \{a, b, c, d, e, f, g, h\}$ with frequencies given by the first $|\Sigma|$ Lucas numbers, give an optimal Huffman code and the corresponding encoding tree for Shadow to use.

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

(a):

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

Using the first 8 letters of the alphabet, our inital que size will be 8 and we need 7 merge steps to buildour tree. The letter is paired with a sequence of the edge labels on the path from the root to the letter.



h:							
g:	0						
f:	1	0					
e:	1	1	0				
d:	1	1	1	0			
c:	1	1	1	1	0		
b:	1	1	1	1	1	0	
a:	1	1	1	1	1	1	0

(b):

We can see from the tree is tail heavy tree with n= n leaves hanging off the right. And this is true for Lucas weights in general, because the Lucas the recurrence implies that

$$L_{n+2} = \sum_{i=0}^{n} L_i + 1$$

We can prove this by induction. The numbers 2,1,3,4 provide a sufficient base. We assume the equality holds for all Lucas numbers smaller than L_{n+2}. Step: We prove correctness for L_{n+2}:

$$L_{n+2} = L_{n+1} + L_n = \sum_{i=0}^{n-1} L_i + 1 + L_n = \sum_{i=0}^{n} L_i + 1$$

Therefore $L_{n+2} < \sum_{i=0}^{n-1} L_{i+1}$ and $L_{n+2} < L_{n+1}$ so L_{n+2} is after all smaller Lucas numbers have been merged into one single tree.

2. (45 points) 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 $\Sigma = [\mathtt{A}, \ldots, \mathtt{Z}]$, and let the function $f(x_i)$ return the index of a letter $x_i \in \Sigma$, e.g., $f(\mathtt{A}) = 1$ and $f(\mathtt{Z}) = 26$. Finally, for an m-character string $x \in \Sigma^m$, define $h(x) = ([\sum_{i=1}^{m} f(x_i)] \mod \ell)$, 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 $\ell = 200$. 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.

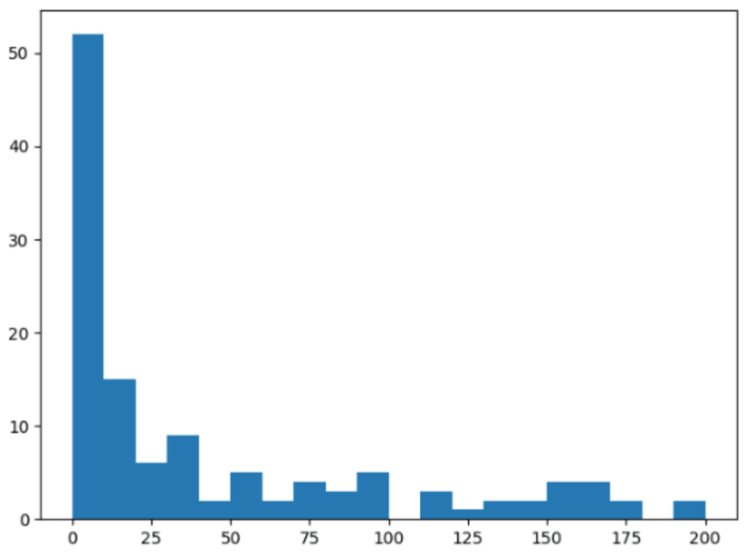
   Hint: the raw file includes information other than name strings, which will need to be removed; and, think about how you can count hash locations without building or using a real hash table.

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

   (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 $\ell = 200$ 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.

(a):



Appended python code below:

```
import csv
import random
import numpy as np
from matplotlib import pyplot as plt

'''    a part   '''
def hash_names(person_name, bin):

    alphabetical_letters = {"A": 1, "B": 2, "C": 3, "D": 4, "E": 5, "F": 6, "G": 7, "H": 8, "I": 9, "J": 10, "K": 11, "L": 12, "M": 13, "N": 1

    sum_hash = 0
    for i in person_name:
        sum_hash = sum_hash + alphabetical_letters[i]

    return sum_hash % bin


person_list = list()
with open('D:\MyFile\dist.all.last.txt') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        name = row[0].split()
        person_list.append(name[0])

    fifty_percent = int(len(person_list)/2)
    person_samples = random.sample(person_list, fifty_percent)

l = 200
result_list = [0 for x in range(l)]  # N = size of list you want

for person in person_samples:
    result = hash_names(person, l)
    result_list[result] = result_list[result] + 1

plt.hist(result_list, bins=np.arange(0,200,10))
plt.title('Results of Hash Function')
plt.xlabel('Hash Bins')
plt.ylabel('Frequency')
plt.show()
```

(b):

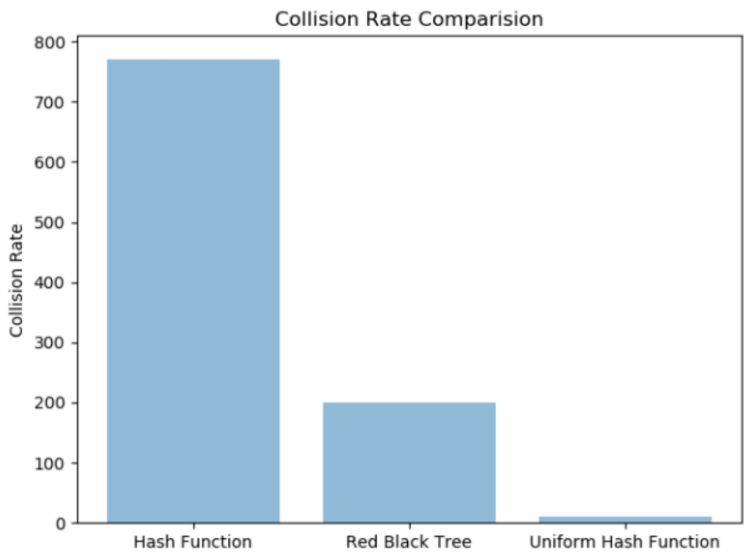
1) The h(x) is a bad hash function because it is not dividing the elements in a uniform way.

2) The hash function has been written in such a way that it gives an answer less than 30 most of the times, which result in a right skew histogram,which result in increasing the collision.

3) Using this hash function, increasing the sample data will not populate the higher less frequent indices, which is not good for a hash function. It should divide data symmetrically.

4) This hash function will enhance the time complexity to access an element upt o O(n)

(c):



```
complexity_of_red_black_trees = 1          # complexity of red black is O(n) = 200
complexity_of_uniform_hash_function = fifty_percent/200          # consider c = 10
longest_chain = max(result_list)
performance = list()
performance.append(longest_chain)
complexity_of_hash_function = longest_chain
performance.append(complexity_of_red_black_trees)
performance.append(complexity_of_uniform_hash_function)
objects = ('Hash Function', 'Red Black Tree', 'Uniform Hash Function')
y_pos = np.arange(len(objects))

plt.bar(y_pos, performance, align='center', alpha=0.5)
plt.xticks(y_pos, objects)
plt.ylabel('Collision Rate')
plt.title('Collision Rate Comparision')

plt.show()

print("Complexity of uniform hash function: ",complexity_of_uniform_hash_function)
print("Collision of longest chain: ",longest_chain)

'''
Comment:

The complexity of uniform hash function is almost 221 in this case, whereas the collision rate of
hash function is 771, so the hash function will needs to get 3.5x time efficient to compete the uniform hash
function.


'''
```

3. (20 points) Grog is struggling with the problem of making change for n cents using the smallest number of coins for his purchase of a new great sword. Grog has coin values of $v_1 < v_2 < \cdots < v_r$ for r coin types, where each coin's 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 **cashier's algorithm**, which all young wizards learn. Harry 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
```

   Thormund snorts and says Harry's code has bugs. Identify the bugs and explain why each would cause the algorithm to fail.

   (b) Sometimes the dwarves at Rocky Mountain 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.)

   (c) On the advice of wizards specializing in electricity, Rocky Mountain Bank has announced that they will be changing all coin denominations into a new set of coins denominated in powers of c, i.e., denominations of $c^0, c^1, \ldots, c^\ell$ for some integers $c > 1$ and $\ell \geq 1$. (This will be done by a spell that will magically transmute old coins into new coins, before your very eyes.) Prove that the cashier's algorithm will always yield an optimal solution in this case.

   Hint: first consider the special case of $c = 2$.

[1] It's a little known secret, but dwarven pets like to *eat* the coins. It isn't pretty for the coins, in the end.

(a):

```
Bug 1: while ( k < r and v[k] > n ) { k++ }
```

The while condition should instead check that (v[k] <= n) which will check that the denomination of coin is less than the total number of change. The while loop will then continue until the denomination is greater than the coin.

```
Bug 2: if k==r { return no solution }
```

k being at the same values of r should still run the same function in the else statement, so we should remove the if/else statement and just have ”n = n - v[k] ”

```
Bug 3: d array is never updated
```

We need to update the d array in the loop before returning it at the end of the function.

(b):

Denominations: [1, 10, 25]

The greedy algorithm would not yield the optimal solution if the dwarves had to make change for 80 cents.

Greedy Solution: 25+25+25+[1+1+1+1+1] == 8 coins

Optimal Solution: 25+25+[10+10+10] == 5 coins (c):

IH:

For i = 0,1,....,k let a$_i$ be the number of coins of denominations and Let c^i be used in an optimal solution to the problem of making change for n cents. Then for i = 0,1,...,ℓ-1 we have a$_i$<c.

Proof by contradiction:

To show that the greedy solution is optimal, we show that any non-greedy solution is not optimal, Let the non-greedy solution use a$_i$ coins of denomination c^i , for $i = 0, 1, , j - 1$, and $i = 0, 1, , j - 1$
thus we have
$\sum_{i=0}^{j-1} a_i c^i = n$ since $n \geq c^j$ we have $\sum_{i=0}^{j-1} a_i C^i \geq c^j$
S'pose the non-greedy solution is optimal. By the IH:

$$\sum_{i=0}^{j-1} a_i c^i \leq \sum_{i=0}^{j-1} (c - i) c^i$$

$$= (c - i) \sum_{i=0}^{j-1} c_i$$

$$= (c-i)\tfrac{c^j - 1}{c - 1}$$

$$= c^j - i$$

$$< c^j$$

which contradicts our earlier assertion that $\sum_{i=0}^{j-1} a_i C^i \geq c^j$. We conclude the non-greedy solution is not optimal.

4. (20 points) We saw in the previous problem that the cashier's (greedy) algorithm for making change doesn't handle arbitrary denominations optimally. In this problem you'll develop a dynamic programming solution which does, but with a slight twist. Suppose we have at our disposal an arbitrary number of cursed coins of each denomination $d_1, d_2, \ldots, d_k$, with $d_1 < d_2 < \ldots < d_k$, and we need to provide n cents in change. We will always have $d_1 = 1$, so that we are assured we can make change for any value of n. The curse on the coins is that in any one exchange between people, with the exception of $i = 2$, if coins of denomination d_i are used, then coins of denomination d_{i-1} cannot be used. Our goal is to make change using the minimal number of these cursed coins (in a single exchange, i.e., the curse applies).

   (a) For $i \in \{1, \ldots, k\}$, $n \in \mathbb{N}$, and $b \in \{0, 1\}$, let $C(i, n, b)$ denote the number of cursed coins needed to make n cents in change using only the first i denominations $d_1, d_2, \ldots, d_i$, where d_{i-1} is allowed to be used if and only if $i \leq 2$ or $b = 0$. That is, b is a Boolean "flag" variable indicating whether we are excluding denomination d_{i-1} or not ($b = 1$ means exclude it). Write down a recurrence relation for C and prove it is correct. Be sure to include the base case.

   (b) Based on your recurrence relation, describe the order in which a dynamic programming table for $C(i, n, b)$ should be filled in.

   (c) Based on your description in part (b), write down pseudocode for a dynamic programming solution to this problem, and give a Θ bound on its running time (remember, this requires proving both an upper and a lower bound).

(a):

Base cases c(j) = 0 for all j, j>1

recrsive below

= 0 if j $\leq$ 0)

c(j)=

= 1 + $\min_{1 \leq i \leq \ell}$(c(j-d$_i$)) if j>1)

(b): ran out of time
(c): ran out of time