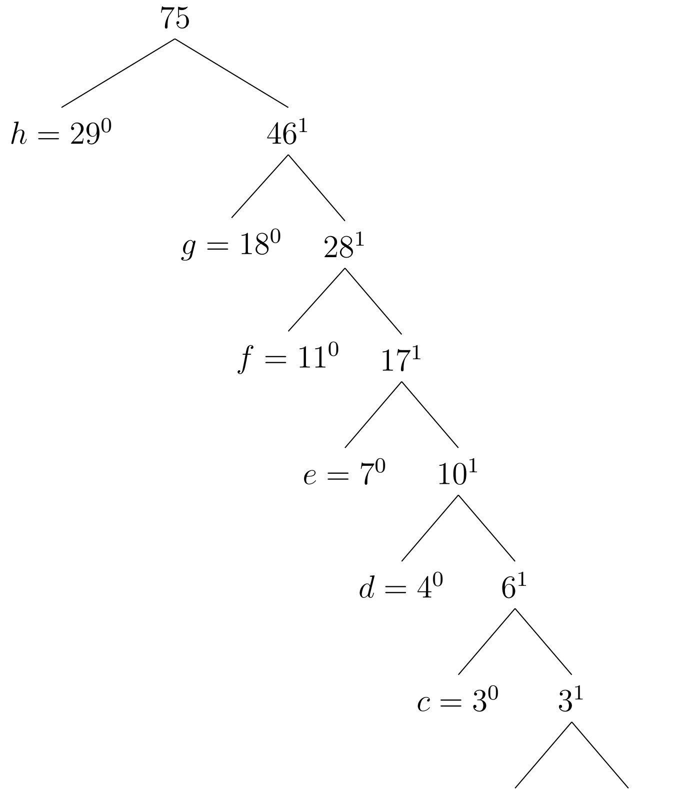
CSCI 3104: Algorithms **Oscar Delfado**
Problem Set 4 **Summer 2019, CU-Boulder**

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.*

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



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

   This works because everytime we we up the Lucas number it will never make a seperate tree that would added to the original. This is because when we want to add the n Lucas

(This is for part a but for some reason wouldn't compile correctly)

a	1111111
b	1111110
c	111110
d	11110
e	1110
f	110
g	10
h	0

Optimal Code

a	11,...,11 (n-1)
b	11,..11 (n-2) + 0
c	11,...,11(n-3) + 0
.	.
.	.
.	.
z	0

number, the the n+1 Lucas number would be greater than ΣL_{n-1} so the n Lucas number and and the root would make children of the new root with their frequencies added. Therefor the tree left childrens will all be letters and there for going up the tree it would be the amount of "1" decreases while ending with zero with the expection of a.

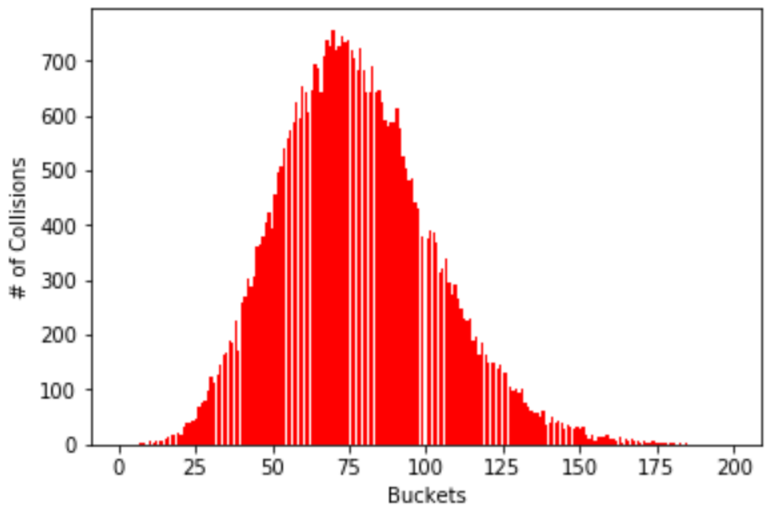
3. *(45 points) A good hash function $h(x)$ behaves in practice very close to the uniform hashing assumption analyzed in class, but is a deterministic function. That is, $h(x) = k$ each time x is used as an argument to $h()$. Designing good hash functions is hard, and a bad hash function can cause a 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 $\Sigma = [\texttt{A}, \ldots, \texttt{Z}]$, 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 $h(x) = ([\sum_{i=1}^{m} f(x_i)] \bmod \ell)$, 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.



This histogram shows that there is a lot of collisions in the 50-90 range of the buckets, overloading them pretty severely (close to 800 collisions). Overall it seems that h(x) is making the names cluster towards the middle buckets, which makes sense because last names tend to be of medium size, and are usually made of letters not found in the back of the alphabet, producing numbers values that are in the 50-90 range.

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

1. The distribution of hash locations that h(x) outputs is not nearly spread evenly enough to be a good uniform hashing function.
2. h(x) does not account for the letters positions in each name. That means that two different names that contain all the same letters will be hashed to exactly the same location, resulting in a lot of collisions.
3. Last names have a general limit to how long they are going to be, and because h(x) is partly based off of the length of a name, the hash locations are limited toa certain possible number. This number is not nearly as large as the amount of names, and thus is a bad way to create hash locations.
4. Because of the range of letters in names, and how the function of h(x) works, it is entirely possible for two names with contrasting letters to have the same hash location. For example the name Aug $(1 + 21 + 7 = 28)$ and the name But $(2 + 21 + 6 = 28)$ will be hashed to the same location. This is so possible in last names that collisions are bound to happen.

(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.

4. *(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.

Bugs:
1. On line 2 is should be d[0...r-1] as the indexing seems to start at 0. 2. On line 4 (k=1) should be k=r-1 because it should start with the largest coins first and should also be before the first while loop as it won't restart with the largest coin in the beginning of 1st loop.
3. On line 5 (k¡r) is unessasary and messes up the loop so it is better removed and (v[k]¿n) should be v[k]¡=n like this once the coin is unable to fit it will move on to the next smalles coin.
4. On line 6 (k==r) should be k==-1 as this means its gone through all the coins and still couldn't hit the desire amount (n).
5. On line 7 the else is unessary as the if statements returns and also it needs to add all number of coins used in d[k] so we can add d[k]++ if this wasn't there it would do the work but return 0's.
6 lastly k++ should not be done inside the second loop and should be done on the end of the first while loop so the same can can be subtracteds from n as much as it could. earlier it would only do it once then move on to the next and so not

subtracting correctly.

(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.)*

One subset of Euro USA denominations in which the greedy algorithm fails to provide the optimal solution is the subset v = (1, 10, 25) for the n = 34 cents. Firstly it would get the largest coin 25 and one to it, when it does this the remaingin is 9 so it skips coin(10) and goes to coin(1) which he adds 9 making a total of 10 coins used $< 9, 0, 1 >$. The optimal solution would be $< 4, 3, 0 >$ $(1(4) + 10(3) = 34)$ as you can see the best solution only used 7 coins compare to 10 that greedy used. Firstly it doesn' have Optiumal Substructure as $30 \leq n \leq 33$ which are subsolution to 34 are not optimal("Best" Solutions) with the greedy choice property as when we choose n=30 the greedy choice would give $< 5, 0, 1 >= 6$ when optimal $< 0, 3, 0 >= 3$, so the greedy choice property fails to pick best solution.

(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 $c = 2$.

(Answer)

[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.

5. *(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).*