CSCI 3104 Spring 2017 **Problem Set 4**

Matthew Niemiec (06/24)
Mehmet Karaoglu (07/30)
Amulya Srivastava (09/02)

---

1. (10 pts) *Given the red-black 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.*

   The algorithm is going to compare m numbers. So in this case, for each T1 number, O(log n) operations are going to have to run to be able to search for the number in T2. Hence T(n) = O(m * log n) which is then T(n) = O(n * 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.* If we traverse T1 and are trying to make H1, that process will take about O(m) time. Meanwhile, in T2, it usually takes about O(n) time to first traverse and then compare the values of T1 and T2. So given that, T(n) = O(m+n) which is then T(n) = O(n). So the asymptotic running time for this algorithm in the average case would be O(n)

2. (30 pts) *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.*

   $L_{0-7} = [2, 1, 3, 4, 7, 11, 18, 29]$

   (b) *Recall that in the Huffman algorithm, we may specify a simple convention that determines the way a pair of dequeued symbols are arranged in the coding tree relative to their parent. How many optimal Huffman codes could you have provided to Voldemort for the set of frequencies in part (**??**)? Justify your answer.*

   Hint: (i) symmetries and tie-breaking, and (ii) not every optimal code tree can be produced by a simple convention for Huffman.

   There can be 8 different combinations of optimal Huffman codes that could be provided to Voldemort because if you look at the tree, when you flip a and b, you will get two different trees, you can flip the 0's and 1's in the first tree so you get the first 2 combinations from there and then there will be a tie-break for which you can flip the nodes so that will be another 2 combinations and, when you flip the 1 and 0's for the second new tree, then you get 4 new combinations. So when you add all the combinations, you get 8 total combination of optimal Huffman codes for Voldemort.

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

3. (30 pts total) *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 coin's value v_i is a positive integer, and where v_1 is the most valuable coin. 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. 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) :
   d[1 .. v.len()] = 0  // initial histogram of coin types in solution
   while n > 0 {
      k = r
      while ( v[k] > n and k >= 0 ) { k-- }
      if k==0 { return 'no solution' }
      else { d[k]++ }
   }
   return d
```

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

   The first bug with this algorithm is that the line **k = r** is inside the first **while** loop. This should instead be before the loop because we shouldn't go back to **v[r]** and check if **v[k] > n** from the beginning after we have decremented **k** because we won't be needing the greater **v[k]** values anymore. It might cause problems if we leave it in the loop.

   The second and third bugs in this algorithm are: (**k >= 0**) and (**if k==0 return 'no solution'**). These are going to cause the algorithm to run incorrectly because in the case that **v[k] > n** and **k = 0** (which means that we won't have a solution because we won't have smaller change to break up or our remaining money), it is going to decrement **k** so that **k = -1**. Then, we won't be able to print out **'no solution'** because we can only print that out when **k = 0**. A way to fix this would be to change **k >= 0** to **k > 0** so we don't decrement when **k = 0**. We would also want to add a **v[k] > n** to the **if** statement because we only want to return **'no solution'** if **v[k] > n** and **k = 0**.

The next thing that we have to fix about our algorithm is that we want to add **n = n - v[k]** after **d[k]++**. The reason we have to do this is because after we have used a coin, we have a smaller amount of money remaining to change. We have to subtract the value of the coin that we used from our original amount. Also, since we are inside a **while n > 0** loop, we might never exit this loop if we don't fix it and if **n** is always greater than 0.

In the end, a correct cashier's algorithm might look something like this:

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

(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.)*

Here is a set of coin denominations for which the greedy algorithm doesn't yield and optimal solution:

$$\{1, 5, 23, 40, 100\}$$

Let's look at the example of trying to convert \$1.46 to change by using the denominations stated above. The greedy algorithm will give the ouput of

100c	40c	23c	5c	1c
1	1	0	1	1

As shown above, it's going to divide the amount into one 100c, one 40c, zero 23c, one 5c, and one 1c. So there will be 4 coins in total. The reason why the algorithm gives us this output is because of it's greedy choice property: It only makes the choice that seems best at a particular moment and solves the remaining subproblems later. The choices that it makes never depend on its future choices and it never reconsideres the old choices that it has made. It has an optimal substructure which only allows it to select the optimal solution to each subproblem, but not necessarily the optimal solution for the whole problem. This algorithm doesn't give us an optimal solution because the optimal solution would look like this:

100c	40c	23c	5c	1c
1	0	2	0	0

The optimal way to change \$1.46 by using our denominations is to divide it into one 100c and two 23c's, where we have 3 coins instead of 5 coins.

(c) *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* $c^0, c^1, \ldots, c^\ell$ *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: consider the special case of $c = 2$.

Here, we will look at the special case where $c = 2$ and prove that the cashier's algorithm for the denominations mentioned above will always yield an optimal solution.The denominations when $c = 2$ will be:

$$2^0 < 2^1 < 2^1 < 2^2 < ... < 2^l \text{ or } 1 < 2 < 4 < 8 < ... < 2^l$$

First, we point out that there are a few things that must hold true for every optimal solution of every problem that uses coins denominated in powers of c. The maximum number of coins that an optimal solution can have for each denomination of coins c^l is 1 coin. This means that number of $c^0 \leq 1$, number of $c^1 \leq 1$, ... , number of $c^l \leq 1$. If the maximum amount of coins for each denomination $wasn't$ 1, we wouldn't have an optimal solution. For example, if we had 2 of a particular

c^l, this wouldn't be optimal because we could have just had 1 of $c^{l+1} = c \cdot c^l = 2 \cdot 2^l$ ($c = 2$) in this case.

Proof (by induction) that our denominations always yield optimal results:

Let's say we want to change an x amount of money where $2^l \leq x < 2^{l+1}$ (value of x is somewhere in between the values of two coin values 2^l and 2^{l+1}). We know that our greedy algorithm is going to pick 2^l and nothing lower. We also know that the optimal choice is also 2^l because if 2^l isn't the choice, we will have to pick more than one denomination from the set $2^0 < 2^1 < 2^1 < 2^2 < ... < 2^{l-1}$ when we could have just picked 2^l, which had a higher value than the sums of the values of the lower denominations.

The next step is to subtract 2^l from x, where we end up with anpther instance of $2^l \leq x < 2^{l+1}$. Our algorithm is also going to find the optimal solution for this x value, and the rest of the x values that come after this, until it finishes.

Idea for this proof is drawn from: https://www.cs.princeton.edu/courses/archive/spring13/cos42
2x2.pdf

4. (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 $\Sigma = [\texttt{A}, \ldots, \texttt{Z}]$, 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. 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.

   Hint: the raw file includes information other than the 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.

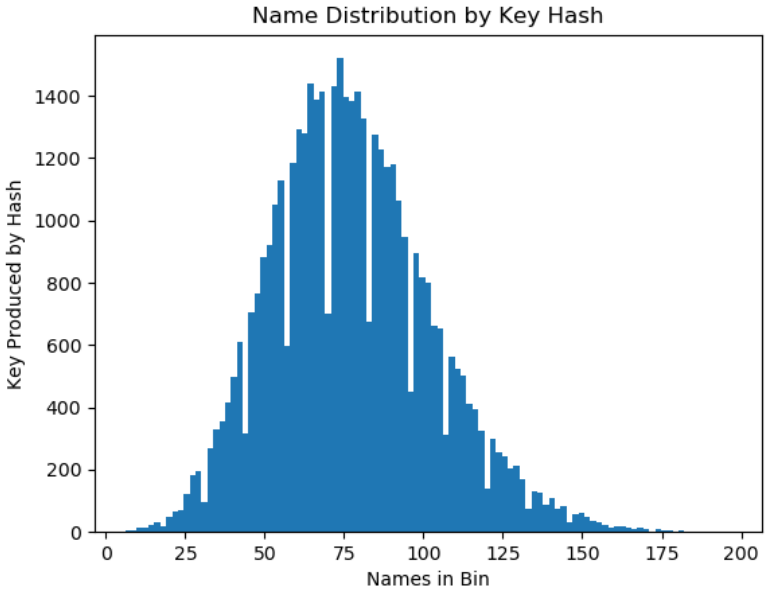

Figure 1: Histogram for Part a

Figure **??** (shown above) shows the distribution of the names by the hash function according to the algorithm provided. The code, on the next page, shows how we take a name, add up all of the characters using ASCII values, and then returns the result modulus 200. Next we read in the file of names, split it according to line breaks, and use the *random* module to rearrange the names in the list in a random order. Then we iterate through each of those, and "hash" every other one. However, we're not actually hashing it, just adding the key to a list, which will be read in by the histogram reader later. However, we see that the vast majority of the results are between 50 and 100, which means that only a quarter of the buckets are really getting used! This makes sense because if names are around 7 letters long, and 'M'=13 is the middle character in the alphabet, and $7 * 13 = 91$, which is not that far off from the true average, and that's just going off intuition.

```
import matplotlib.pyplot as plt
import random

# This is the hash function that returns the key of the hash table for the name
def hash_function(name):
    key = 0
    for char in name:
        key += ord(char)-64
    return key%200

# Read in the complete list of names. The other columns were previously removed
reading = open("LastNames.txt", 'r')
text = reading.read()
reading.close()
# Parse the names by line and then shuffle them to get them in a random order
text = text.split('\n')
random.shuffle(text)

# hist_counter keeps track of the number of each key (i.e. [3, 4, 3, 2])
hist_counter = []
choose_name = False
for name in text:  # Iterate through every name
# We are getting every other name to get an even sample of the random sample
    choose_name = not choose_name
    if choose_name:
        key = hash_function(name)
        hist_counter.append(key)

# Using matplotlib to make the actual histogram
plt.hist(hist_counter, bins='auto')
plt.title("Name Distribution by Key Hash")
plt.xlabel("Names in Bin")
plt.ylabel("Key Produced by Hash")
plt.show()
```

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

1) As we found from the histogram in part A, it has a poor hashing distribution. Many of the names' keys end up between 50 and 100, which does not make

for very efficient storing or retrieving of data. Too many items are stored in a few buckets, and virtually no items are stored in others. We see that negligibly few names were hashed above 180, which means that the top 20 buckets were basically unused. Also, due to the wrap-around nature of it, it is even more unlikely that a name will be placed in the first few bins. For example, the only string of key less than 200 that can be placed in bucket 1 is the character 'A', which we know is not a last name. So the lower buckets are also unused. This can also be attributed to a poor choice for ℓ, which is very very large given the spread.

2) The hashing distribution is not random - similar names are together. Not only are many buckets unused, but similar names are grouped similarly, not randomly. In order to have an efficient hashing algorithm, the hashing should be as random as possible, yet we can know where the last names with 5 characters are being stored, for example.

3) It takes longer depending on the size of the input, taking really long for long names. We know that there aren't any extraordinarily long names being entered in this list. However, taking an adversarial approach, we could be handed a very long string and then just hashing would take $O(n)$ time instead of the promised $O(1)$.

4) There are significant dips in the data. We briefly mentioned earlier the lack of randomness in the hashing, but it really deserves its own point. Notice that there are several "dips" in the data, where some of the bucket groups are. This is not a coincidence. When randomly running the experiment, the same dips always show up in the exact same places, which again just means that the distribution wasn't random enough.

(c) (10 pts extra credit) *Produce a plot showing how the length of the longest chain (were we to use chaining for resolving collisions) grows as a function of the number n of these strings that we hash into a table with $\ell = 200$ buckets. Comment on this trend using the language of the asymptotic growth of functions, worst-case scenarios, and loading factors of hash tables.*

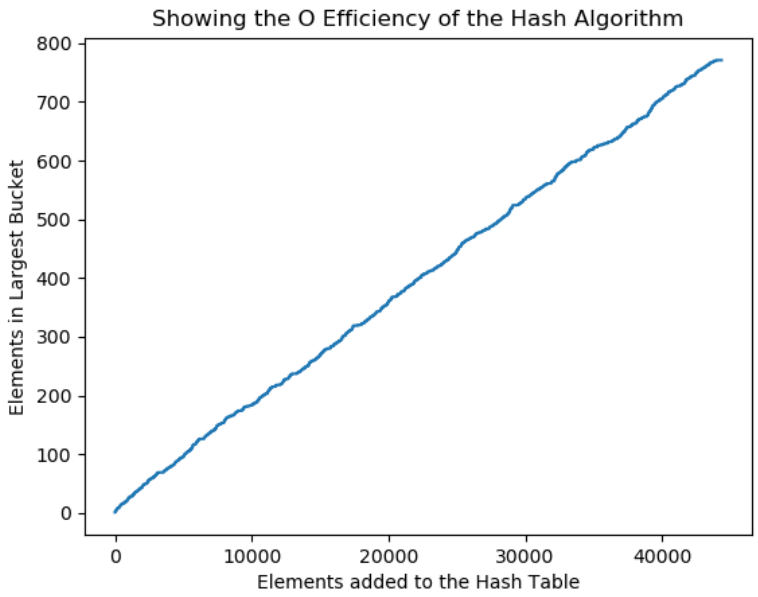

Figure 2: Histogram for Part c

Figure **??** (shown above) shows the progression of the largest number of elements in a single bucket as each name is added to the table. We can see the results are disastrous! The number of items in a bucket grows with $O(n)$, which is asymptotically equivalent to a simple linked list! Bad hash function! Instead we're hoping to see something more flat, which closely resembles $O(1)$, but there are too few buckets for too many data points, and too few buckets are being used.

```
import matplotlib.pyplot as plt
import random

def hash_function(name):
    key = 0
    for char in name:
        key += ord(char)-64
    return key%200

reading = open("LastNames.txt", 'r')
text = reading.read()
reading.close()
text = text.split('\n')
random.shuffle(text)

# We need to keep track of the number of times each key is hashed in key_counter
key_counter = {}
max_key = 0  # The largest number in key_counter
point_adder = 0
plot_points = []  # Every iteration we add another point to our plot
choose = False
for name in text:
    choose = not choose
    if choose:  # Again getting every other name
        key = hash_function(name)
        try:
            key_counter[key] += 1
        except KeyError:
            key_counter[key] = 1
        max_key = max(key_counter[key], max_key)
        plot_points.append(max_key)

# Plot the linear graph
plt.plot(plot_points)
plt.title("Showing the O Efficiency of the Hash Algorithm")
plt.ylabel('Elements in Largest Bucket')
plt.xlabel("Elements added to the Hash Table")
plt.show()
```

5. (20 pts extra credit) *Let A and B be arrays of integers. Each array contains n elements, and each array is in sorted order (ascending). A and B do not share any elements in common. Give a $O(\lg n)$-time algorithm which finds the median of $A \cup B$ and prove that it is correct. This algorithm will thus find the median of the $2n$ elements that would result from putting A and B together into one array. (Note: define the median to be the average of the two middle values of a list with an even number of elements.)*