Solutions for HW3 - CS 6033 Fall 2021

Q1 → Hashing

1. Suppose you are given the following keys: $112, 2542, 9992, 5502$ and the following hash function $h(x) = x \mod 10$.

   Hash the keys using the hash function. How many keys collide?

   Choose a random[1] hash function, $h_{a,b}$ from $H_{10007,10}$. Include your random choice of $h_{a,b}$ with your answer to this problem. Before you rehash the numbers with the new hash function, determine the probability that the keys 112 and 2542 collide when hashed with $h_{a,b}$? Hash each of the keys $112, 2542, 992, 502$ with $h_{a,b}$. How many keys collided?

   If you had 1000 keys (all keys were positive integers less than 10000) inserted into a hash table of size 2000 using a hash function, $h_{a,b}$, randomly chosen from $H_{10007,2000}$ (the family of universal hash functions we defined in class). What is the expected number of collisions you would have if you inserted a new key, x, into the hash table?

We have keys: 112, 2542, 9992, 5502, and hash function h(x)= x mod 10.

Now we hash those keys into table slots with h(x), h(112)=2, h(2542)=2, h(9992)=2, h(5502)=2

Therefore all four keys collide at table slot 2.

Before I rehash the numbers with new hash function, the probability that the keys 112 and 2542 collide is 1/m=1/10. I pick a to be 5, and b to be 7. p=10007, m=10

Now we have universal family $H_{pm} = \{h_{a,b}(x) = ((ax + b) \bmod p) \bmod m\}$

$H_{5,7}(112) = ((3*112 +7) \bmod 10007) \bmod 10=3$

$H_{5,7}(2542)=9$

$H_{5,7}(992)=1$

$H_{5,7}(502)=7$

There are 0 key collided this time.

If I had 1000 keys, n=1000, the table size is 2000, the expected number of collisions is n/m=1000/2000=1/2

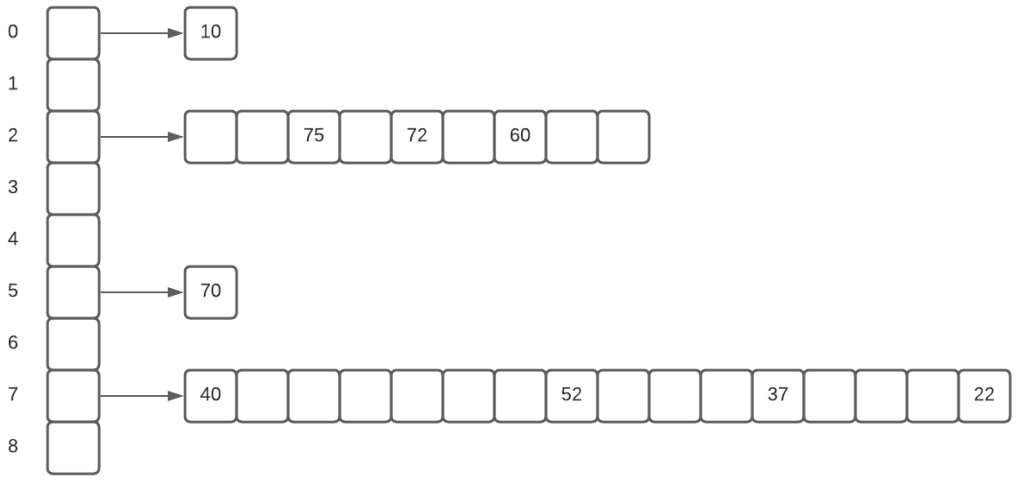
Q2 → Perfect Hashing - Structure

2. Using the technique for perfect hashing that we discussed in class, store the set of keys:

$$\{10, 22, 37, 40, 52, 60, 70, 72, 75\}.$$

For your first hash function ("outer hash" function) use $h_{a,b}(k) = ((ak + b) \mod p) \mod m$, where $a = 3, b = 42, p = 101, m = 9$.

Answer: Your answer might change depending on the inner hash functions used.



Q3 → Perfect Hashing - Space Analysis

3. What is the probability that you use at most $16n$ space for the secondary hash tables? What is the probability (in terms of k) that you use at most kn space (for some constant k)?

Using corollary 11.10(or last page of the slide Lec3), we know that: $E[\sum_{j=0}^{m-1}(n_j)^2] < 2n$
And according to Markov's inequality: $Pr[X \geq t] \leq \frac{E[X]}{t}$
We can regard $\sum_{j=0}^{m-1}(n_j)^2$ as the random variable X and plug into Markov's inequality, then we have:
Pr[using more than 16n space]= $Pr[\sum_{j=0}^{m-1}(n_j)^2 \geq 16n] < 2n/16n = 1/8$
Pr [using at most 16n space] = 1 - 1/8 = 7/8

Similarly,
Pr[using more than kn space]= $Pr[\sum_{j=0}^{m-1}(n_j)^2 \geq kn] < 2n/kn$
Pr [using at most 16n space] = 1 - 2n/kn

Q4 → Perfect Hashing - Collision

4. (10 points) How many times must you attempt to construct one of a perfect hash table's sub-tables so you have one with probability greater than 99.99999%? This is a failure rate of 1 out of one ten million.

For a fixed set of keys, S, and a table size of $m = |S|^2$.
If we randomly choose a function from the universal family - the probability that the function has at least one collision on S is $<\frac{1}{2}$ (thus the probability this function has no collusion on the set of keys is $>=\frac{1}{2}$).
If we randomly choose two functions from the universal family, the probability that both two functions have at least one collision on the set (where we try them separately) is < 1/4 If we randomly choose n functions from the universal family, the probability that all of them would have at least one collision on the set is less than $(\frac{1}{2})^n$ (thus the probability that at least one of n functions has no collision is 1 - $(\frac{1}{2})^n$)

Now, in this question if we want the probability that there is no collision to be greater than 99.99999%:
$1 - (\frac{1}{2})^n \Rightarrow (\frac{1}{2})^n < 10^{-7} \Rightarrow log(\frac{1}{2})^n < log10^{-7} \Rightarrow -nlog2 < -7log10 \Rightarrow n > 7log10 \approx 23.25 \approx 2$

Q5 → Application of Hashing - Find unique elements

5. You keep thinking back to the question for homework assignment 1. You are sure you could have solved this faster on average. Here was the question and the new running time you think you can achieve:

> Trying to finance your education, you decide to invest your D dollars in the stock market. The problem is you don't know which stocks to invest in. You decide ask your friends for stock tips and invest in every stock any of your friends suggests. If you receive n unique stock names you will invest D/n in each of the n stocks. Since some of your friends have given you the same stock name, you need to find a way to remove the duplicates.

Design an efficient algorithm to perform this task. You may call any algorithm as a subroutine that was presented in lectures 1-3. Your algorithm must run in $O(n)$ average case time where n is the number of names given to you by your friends.

Demonstrate that your algorithm runs in $O(n)$ average case time.

State your improved average and worst case running time is of your algorithm in big-Oh notation. Justify your reasoning.[2]

Answer: One solution is to use a universal hash-function with chaining. Assuming `stock_list` is the list of suggestions received,

```
remove_duplicates(stock_list):
    unique_stock_list = [ ]
    for stock in stock_list:
        if not CHAINED-HASH-SEARCH(stock):
            CHAINED-HASH-INSERT(stock)
            unique_stock_list.append(stock)
    return unique_stock_list
```

Our runtime is bound by the number of stocks in `stock_list`. Time complexity would be n times the runtime of each iteration of the **`for`** loop. In the average case, both `CHAINED-HASH-SEARCH(stock)` and `CHAINED-HASH-INSERT(stock)` take $O(1)$ time as the load factor α would be 1. So, the average case runtime of this subroutine is $O(n)$. The worst case is when some elements hash to the same index, where `CHAINED-HASH-SEARCH(stock)` might take $O(n)$ time, which gives $O(n^2)$ time. Some implementations that have used Direct-Addressing hashing or an equivalent such as dictionaries or maps have received credit if they have answered $O(n)$ time for the worst-case scenario.

Q6 → Application of Hashing - Stock Tips Reward

6. Your strategy worked and you have earned 1/2 of your fall tuition on the stocks you invested in. NYU is full of smart people (or you just got lucky...)

   Most of your friends gave you multiple stock tips. You decide to take out to lunch any friend whose stock tips earned you more than 20% return (i.e. if friend 1 suggested AMD, TSLA, and GME - by following your friends advice you made 22%)[3]

   Let n be the number of stock tips you received, and f is the number of friends, and $m \leq 5$ is the maximum number of stock suggestions any one of your friends gave you. In $O(n + f)$ average case time, determine who you will be taking to lunch. You have an array A, where $A[i]$ holds the stock name and the people who suggested you buy the stock and how much money you have made from owning this stock.

   Design an efficient algorithm to perform this task. You may call any algorithm as a subroutine that was presented in lectures 1-3. Your algorithm must run in $O(n + f)$ average case time where m is the number of names given to you by your friends.

   Justify that your algorithm runs in $O(n + f)$ average case time.

   if you wrote a clever algorithm, see if you can justify that your algorithm runs in $O(n+f*m)$ average case time where now $m \leq n$. If you can do this, you will receive a small amount of extra credit,

The pseudo-code for this algorithm is listed below:

For each stock:

For each person who suggested the stock:

If the person was not in the hash table, insert them into the hash table T and add the stock name/stock return information as satellite information to go with the name.

If a person was already in the hash table, add the stock name/stock return as satellite information to go with the name.

(I would keep the satellite information as a linked list associated with the person)

This takes O(n+ f*m) time.

If we calculated the running time of the inner loop separately from the rest of the algorithm, it would take O(f*m) (This is for the entire algorithm, not just one time through the loop.)

The time it takes to compute the outer for loop without including the time it takes for the inner for loop O(n).

The next part of the algorithm is easier to compute the running time for

Looping through T to calculate the average return for each person would take O(f*m) time.

Therefore the overall time complexity for this algorithm is O(n+ f*m) average case time.

Q7 → Application of Hashing - Newsletter

7. People started to hear about your talent in the stock market (perhaps you shouldn't have posted your gains on r/wallstreetbets) and they have started asking you for advice.

   You are never one to not spot an opportunity, and you decided to write a weekly newsletter and charge people for access. This took off (perhaps it helped that your cousin posted about your success and your for pay newsletter on reddit...)

   However your mass email provider has been having problems and sometimes the newsletters are not sent to your subscribers. Not wanting to get a bad reputation, you want to quickly determine if a users complaint is legitimate.

   The problem is you haven't gotten a new computer and you don't have that much memory to store all your subscribers, so you need to come up with an alternative method to know (within a reasonable probability) if the request is legitimate. If it is legitimate, you quickly forward them the newsletter (you have their email address from the customer support request.)

   Here is your plan:

   - You create an array S and initialize all the values to 'n'.
   - You then take all your subscribers and hash their email address, and set $S[hash(email\_address)] =$'s'.

     If a new person subscribes, you hash their email address, and set $S[hash(email\_address)] =$'s'.
   - If someone emails you saying they didn't get your newsletter, you quickly hash their name. If $S(hash(email\_address)] =$'s', you send them your newsletter, otherwise you send them your sign up page on your website.

   For this question, assume the simple uniform hashing assumption.

   (a) What is the probability a subscriber doesn't get their newsletter and when you checked to see if they were a subscriber you found that $S[hash(email\_address)] =$'s'? Assume they used the correct email address.

   (b) What is the probability a non-subscriber doesn't get their newsletter and when you checked to see if they were a subscriber you found that $S[hash(email\_address)] =$'s'?

The assumption is that S.length = N and the number of subscribers = A.

(a) The question is asking about the probability of a hash collision occurring for a subscriber. The answer should be 1. A subscriber should have a hash collision happening as they were in the system and their email address was already hashed.

(b) The question is asking the probability of a hash collision occurring for a non-subscriber. The probability of a collision occurring for two keys to one slot is $\frac{1}{N}$. So, non-collision at that slot has the probability of $1 - \frac{1}{N}$.

Let the person be "x" and their slot in S be hash(email)=b, where $0 \leq b \leq N - 1$

$Pr[\text{“}x\text{” is not a subscriber but } S[b] = \text{“}s\text{”}] = Pr[\text{at least one subscriber collides with "}x\text{" at } b]$

$= 1 - Pr[\text{none of } A \text{ people collide with "}x\text{" at } b] = 1 - (Pr[\text{one of } A \text{ people doesn't collide with "}x\text{"}$

$= 1 - (1 - \frac{1}{N})^A$

Q8 → Application of Hashing - Newsletter Part 2

8. You started to realize your solution in question 7 was letting a few people have your newsletter without subscribing. You don't want to buy a new laptop (you are making too much money off the market) so instead you use two hash functions as follows.

   - You create an array S and initialize all the values to 'n'.
   - You then take all your subscribers and hash their email address, and set $S[hash_1(email\_address)] =$'s', and $S[hash_2(email\_address)] =$'s'.
     If a new person subscribes, you hash their email address, and set $S[hash_1(email\_address)] =$'s', and $S[hash_2(email\_address)] =$'s'.
   - If someone emails you saying they didn't get your newsletter, you quickly hash their name. If both $S[hash_1(email\_address)] =$'s' and $S[hash_2(email\_address)] =$'s', you send them your newsletter, otherwise you send them your sign up page on your website.

   For this question, assume the simple uniform hashing assumption.

   (a) What is the probability a subscriber doesn't get their newsletter and when you checked to see if they were a subscriber you found that $S[hash_1(email\_address)] =$'s' and $S[hash_2(email\_address)] =$'s',? Assume they used the correct email address.

   (b) What is the probability a non-subscriber doesn't get their newsletter and when you checked to see if they were a subscriber $S[hash_1(email\_address)] =$'s' and $S[hash_2(email\_address)] =$'s',?

The assumption is that S.length = N and the number of subscribers = A.

(a) The question is asking about the probability of a hash collision occurring for a subscriber. The answer should be 1. A subscriber should have a hash collision happening as they were in the system and their email address was already hashed.

(b) The question is asking the probability of a hash collision occurring for a non-subscriber. The probability of a collision occurring for two keys to one slot is $\frac{1}{N}$. So, non-collision at that slot has the probability of $1 - \frac{1}{N}$. For two slots, the probability is $(1 - \frac{1}{N})^2$.

Let the person be "x" and their slot in S be $hash_1(email) = b,\ hash_2(email) = c$, where $0 \leq b \leq N - 1\ and\ 0 \leq c \leq N - 1$.

$Pr[\text{“}x\text{” is not a subscriber but } S[b] = \text{“}s\text{” and } S[c] = \text{"}s\text{"}]$

$= Pr[\text{at least one subscriber collides with "}x\text{" at } b] * Pr[\text{at least one subscriber collides with "}x\text{" at } c]$

$= (Pr[\text{at least one subscriber collides with "}x\text{" at } b])^2$ *(Because b and c are a random slot in the table)*

$= (1 - Pr[\text{none of } A \text{ people collide with "}x\text{" at } b])^2$

$= (1 - (Pr[\text{one of } A \text{ people doesn't collide with "}x\text{" at } b])^A)^2$

$= (1 - ((1 - \frac{1}{N})^2)^A)^2$

$= [1 - (1 - \frac{1}{N})^{2A}]^2$