Lab 4 - Second Moment Estimation & Load Balancing

Due: Thursday April 20th 11.59PM in PDF form via Websubmit.

1 What to submit (MANDATORY)

1. PDF file ((lab4_fisrtname_lastname.pdf))including plots and a snapshot of the code used to answer the questions.
   - Names of the collaborators.
   - Number of late days for this assignment.
   - Number of late days so far.
   - References used
2. Python script (lab4_firstname_lastname.py) with the code used.

Failing to meet any of the above requirements will cause a decrease of your grade.

2 Load Balancing

2.1 Background

Simply put, load balancing is the methodology of distributing workload on to multiple workers. In computing this can mean many things: balancing workload on computer cores, or providing Internet service from multiple servers. Load balancing isn't limited to computers, though. Many branches of mathematics use load balancing - queuing theory, for instance, is full of load balancing.

In this lab we are going to see a couple of different load balancing techniques which are improved by randomization.

2.2 Deterministic Load Balancing

Imagine a supermarket at around 6:00 on a weeknight. Everyone in town is buying groceries for dinner, right? Naturally, the lines are extremely long. You have been hired to stand in the front of the store and assign customers to a new line one at a time. This is a typical load balancing problem, you have a large workload (customers) and many workers (cashiers), and you need to distribute the work as evenly as you can. Remember, you have to assign customers to lines as quickly as you can, otherwise the aisles get full of angry customers (some are waiting and others still trying to shop!)

Every question in the lab assumes, for simplicity, that we have n shoppers and n lines.

1. Consider a simple deterministic algorithm: each time a customer arrives put him in the shortest line.

   Model the problem by keeping an array of counters, just add one to the count if you want to send a customer to that line. Assume that another process is decreasing the size of the count in between runs of your algorithm (to simulate customers leaving the store after paying), so you can never know the count at any given line until you check. Let n be a parameter to your function for the number of lines:

   ```
   def deterministicLoadBalancer(n):
   ```

   Submit your code.

   Hint: If it helps, imagine you have balls and bins, instead: You need to put balls in bins by evenly distributing them. Sometimes the balls fall out of the bins, though, so you can't know how many balls are in a given bin unless you look.

2. What is the running time of your deterministic algorithm in terms of the number of lines you have to look at per customer? (i.e. given n customers, how many lines do you look at, total?)

2.3 Randomized Load Balancing

Imagine using the deterministic load balancing procedure you wrote in practice. You would have to run up and down the store in between every customer, checking for the shortest line. The algorithm might be great for your daily cardio, but definitely would aggravate your boss.

It's time for a new approach, then. You're a little too lazy to want to run around the store, so how can you randomize your algorithm so you don't have to move from the front of the line of customers looking to be directed?

The answer is actually very simple, just randomly and uniformly assign customers to lines. That's right, your job can be replaced with a screen that uniformly draws numbers from $[1, ..., n]$. It doesn't have to know anything about the current state of the lines.

The real question is, how well is the load balanced with this algorithm? We use the same assumption as earlier, n shoppers and n lines.

Chapter 5 of your Mitzenmacher and Upfal book has a comprehensive discussion of this exact algorithm. We haven't quite introduced all the concepts needed for proof, but it turns out that with this random and uniform assignment there is a high probability, $1 - \frac{1}{n}$, there are at most $\frac{3 \ln n}{\ln \ln n}$ shoppers in any line.

3. Write a Python simulation for the randomized load balancer. Submit the code. Suppose you have n customers and n lines. Your function should randomly assign customers to lines and return the length of the longest line. Call the function loadBalancer(n) where n is the number of customers and lines. **Submit your code.**
4. What is the running time of this algorithm? Does it run faster than the deterministic algorithm from section 2?
5. As with all randomized algorithms, there is a tradeoff here. What is it?
6. What is the worst case of this randomized load balancing algorithm (i.e. what is the maximum length of any given line after n insertions)? What is the probability that the worst case happens?
7. Now it's time to plot some data. Plot maximum line length versus n with $n = 3, ..., 500$. Use the average of at least 1000 simulations (i.e. 1000 independent runs of loadbalancer()) for every data point. Also, on the same plot, plot a line of theoretically maximum line length using Mitzenmacher and Upfal's bound from above. Does your empirical data ever get higher than your theoretical bound? **Submit your code and plot.**

It's time for one more beautiful and very important result: Mitzenmacher wrote his PHD thesis on it so we will spare you the math. It's actually this result that got him his first job at Harvard! (Just in case the paper he published is titled: 'The Power of Two Choices in Randomized Load Balancing' and the proof is also in the textbook, chapter 14.)

The result is remarkably intuitive. What if, instead of choosing one random line, we chose two and picked the shorter of the two? This way, it's impossible to ever add to the longest line. In his paper, Mitzenmacher bounded the maximum line length (again with high probability) for two choices, and generalized the bound to the case where you choose from d lines and pick the shortest.

In general, with d choices, the maximum line length is now $\frac{\ln \ln n}{\ln d} + 2, d \geq 2$ where d is the number of choices.

This result might even get you your job back - you need to do some comparisons, but in this case it's always a fixed and possibly small number!

8. Modify loadBalancer to accept another parameter, d, and to use a randomized d-choice algorithm. Again, it should return the length of the longest line. **Submit your code.**
9. What happens to this algorithm when $d = n$?

10. Lets look at how this bound improves with higher values of d. Plot d on the x-axis and the maximum line length on the y, with $n = 1000$, for $d = [2, 3, ..., 10]$. Again, each data point should be the average of 1000 trials. Plot Mitzenmacher and Upfal's bound on the same plot. Is the empirical data ever higher than the theoretical data? Use values of $d \geq 2$ because that is what our bound is limited to. **Submit your code and plot.**

11. Does your empirical data line seem to flatten out very quickly? Why is this the case?

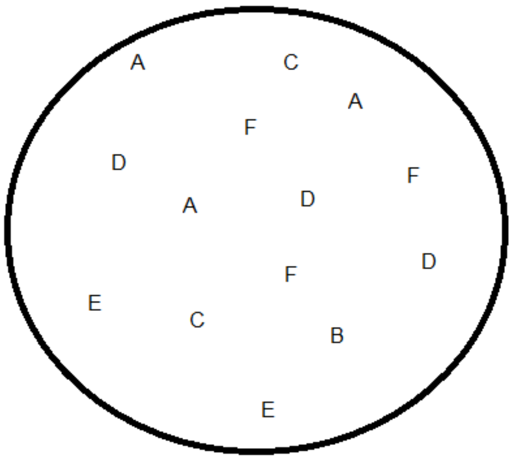
3 Second moment estimation

3.1 Background

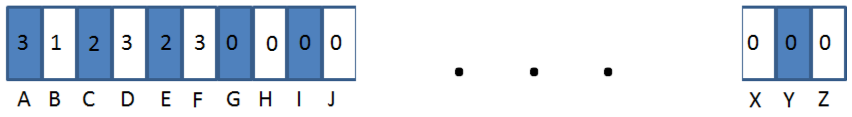
This part of the lab is about an interesting way to estimate the Second Moment of a data set (which we will define) using 4-wise independent hashes. The method and proof are taken from the appendix to a paper called "Tabulation Based 4-Universal Hashing with Applications to Second Moment Estimation" by Mikkel Thorup and Yin Zhang.

As usual, we will work through the Math and then verify with code.

First, lets define what we mean by data set. As with every set of data, there is a Universe, U, that the data comes from. If the data is "letters of the English alphabet" then the U is simply all 26 characters. The sets of data we will be working with are multi-sets, meaning we can have repeats of the same data from U. For example, a sample data set, u, of alphabet data might look like this:



This is obviously not a great way to represent the data - especially if we want to use it in mathematical and programming constructs. Instead, we can map this data set to a vector in the following way:



Notice two things: the length of $u =$ the number of objects in the universe (we didn't choose the variable by accident), and $u[i] =$ # of times item i appears in the set (in our case, A is the first element and it appears 3 times in our data, B the second and it appears once, and so on).

Now that we can represent the data, lets define what the moments of data sets are. At a very high level, the different moments are different methods for assigning a single value to an entire data set. Moments are frequently used in statistics and other fields. There is a very specific formula for computing the different moments $(F_1, F_2, .., F_n)$. Consider data set $S = \{a_1, a_2, ...a_s\}$ where each a_i is a what we will call a key (the letter in our previous example).

The First Moment (we will call it F_1 from here on) is given by the following formula:

$$F_1 = \sum_i |u[i]|$$

Similarly, the Second Moment, F_2:

$$F_2 = \sum_i (u[i])^2$$

Intuitively, F_1 is the number of all items in a set of data. The Second Moment is the sum of squared counts. The Second Moment is particularly useful because its square root gives us some interesting information about the data. First, the square root of the second moment can give us a good approximation of the amount of data in the data set. Comparing the square root to the first moment also gives us some good information - if we know there are 20 items in the data set and the second moment is exactly 20^2 then its clear that one counter was exactly 20 - or that they were all the same item! Thus it is a rough measure of duplicates in the data.

We must define one more concept, which we have already used a couple of times intuitively.

$$\text{Let } v_a = \sum_{i:a_i=a} 1$$

which says that v_a is a count of the number of times a shows up in the data.

Answer this easy problem to get a feel for the concepts.

Let S contain data about letters in the alphabet. Further, let S be a multi-set (keys can be repeated). We'll use a really small S so it will be easy to calculate the moments.

$$\{a, a, b, b, c, d, d, d\}$$

12. What is v_d?

13. What is F_2 of S, or what is Second Moment of S?

3.2 Estimating the Second Moment

This section (and the rest of the lab) is about estimating the second moment of a particular set of data. Consider a real world example - routing. Imagine writing the code to estimate the Second Moment of the packets that come through a router in one day. The first concern is that routers do not have access to all the packets they will ever have to transmit at once, so how would you keep track of data if it was given to you in a stream (not all at once, but a little at a time) like a router? You would keep an array of counters and just increment the correct count by 1 every time you saw new data.

Let u = an array of counters where $u[i] = c_i$ = the current count at i. The new formula for the second moment at any given time is :

$$F_2 = \sum_i (u[i])^2$$

The next (and rather more difficult) problem: packets have 3500 bits, so how long would your array of counters need to be? The answer is 2^{3500}, as that is the number of unique packets. This is an impossibly large array to store anywhere, let alone in the limited space that a router has. So, instead of calculating the exact F_2, we can estimate it! How, you ask? Hashing!

Instead of keeping one counter for every possible packet, we will hash all the packets as they come in to one of m buckets and keep only m counters - where every time an item hashes to a bucket, we increment the counter by one. Lets call this array of counters w and let w_i be the number of items that hash to bucket i.

In this lab we will propose and prove a method of using these m counters whose *expected value* is exactly the Second Moment of the original data.

Some notation: Our hash function is $h : [u] \to [m]$ which means we take anything from the universe of objects and hash it to a number between 1 and m. We use a family of 4-wise independent hash functions.

Definition. (k-wise independence). Let $X_1, X_2, ..., X_n$ be a random variables. The random variables are k-wise independent if for any $i_1, i_2, ...i_k$, and $x_{i_1} \in Range(X_{i_1})$ ... $x_{i_k} \in Range(X_{i_k})$ it follows that

$$\Pr[X_{i_1} = x_{i_1} \cap X_{i_2} = x_{i_2} \cap ... \cap X_{i_1} = x_{i_k}] = \Pr[X_{i_1} = x_{i_1}] \cdot \Pr[X_{i_2} = x_{i_2}] \cdot ... \cdot \Pr[X_{i_k} = x_{i_k}]$$

This means that the probability that any 4 independently hashed items all hash to the same bucket is $(\frac{1}{m})^4$.

Remember our final goal, we would like to estimate the Second Moment of an input stream. Keeping in mind the machinery we have put into place already, we propose to estimate it in the following way:

$$X_{estimate} = \frac{m}{m-1} \sum_{i \in [m]} w_i^2 - \frac{1}{m-1} (\sum_{i \in [m]} w_i)^2 \tag{1}$$

So, once we have all the data collected - in this case the array of m counters, we can use this $X_{estimate}$ to estimate F_2. Why does this work? We haven't proven it yet, but your goal during this section of the lab is to prove that $E[X_{estimate}] = F_2$ of the original data.

Let's introduce some notation which will make it easy to convert between the counters and the original data: Let $a \sim b$ denote $h(a) = h(b)$, and $a \nsim b$ denote $h(a) \neq h(b)$. Finally, let $a \ncong b$ denote $a \sim b$ but $a \neq b$.

Finally, we introduce the multi-sum notation:

$$\sum_{a,b} f(a) f(b) = \sum_a f(a) \sum_b f(b)$$

Now we can rewrite $X_{estimate}$ in terms of the original data stream:

3.3 Step 1: Go from equation (1) to equation (2)

$$X_{estimate} = \frac{m}{m-1}\sum_{a\sim b} v_a v_b - \frac{1}{m-1}\sum_{a,b} v_a v_b \tag{2}$$

Take a hard look at this new equation, can you see why it is exactly equal to the original equation we gave for $X_{estimate}$? Lets prove it one term at a time, so first we will prove that the first term of each of those equations are equal.

Prove: $\frac{m}{m-1}\sum_i c_i^2 = \frac{m}{m-1}\sum_{a\sim b} v_a v_b$

The first thing we note is the constant term $\frac{m}{m-1}$ on each side, we will ignore it in the proof since it is constant on both sides.

Note: $c_i = \sum_{h(a)=i} v_a$. Why is this true? The count at any individual bucket is equal to the sum of the counts of items that hash to that bucket.

$$\begin{aligned}
c_i^2 &= (\sum_{h(a)=i} v_a)^2 \\
&= (\sum_{h(a)=i} v_a)(\sum_{h(b)=i} v_b) \\
&= \sum_{a\sim b, h(a)=i} v_a v_b \\
\sum_i c_i^2 &= \sum_i \sum_{a\sim b, h(a)=i} v_a v_b \\
&= \sum_{a\sim b} v_a v_b
\end{aligned}$$

14. Now its your turn. Prove $\frac{1}{m-1}(\sum_{i\in[m]} c_i)^2 = \frac{1}{m-1}\sum_{a,b} v_a v_b$

3.3.1 Step 2: From equation (2) to equation (3)

$$X_{estimate} = F_2 + \sum_{a\ncong b} v_a v_b - \frac{1}{m-1}\sum_{a\nsim b} v_a v_b \tag{3}$$

15. Prove that $\frac{m}{m-1}\sum_{a\sim b} v_a v_b - \frac{1}{m-1}\sum_{a,b} v_a v_b = F_2 + \sum_{a\not\cong b} v_a v_b - \frac{1}{m-1}\sum_{a\nsim b} v_a v_b$. Hint: Notice that $\sum_{a,b} v_a v_b = \sum_a v_a^2 + \sum_{a\neq b} v_a v_b$ Which can be seen using a small example. $\sum_{a,b} v_a v_b = (v1v1 + v1v2 + v1v3 + v2v1 + ... + v3v3) = (v1v1 + v2v2 + v3v3) + (v1v2 + v1v3 + v2v1 + v2v3 + v3v1 + v3v2) = \sum_{a=a} v_a^2 + \sum_{a\neq b} v_a v_b$ Use this fact to break sums into smaller, component sums.

3.3.2 Step 3: From equation (3) to equation (4)

Now we define a variable $X_{a,b}$ in the following way $X_{a,b} = \begin{cases} 1 & \text{if } a \sim b \\ -\frac{1}{m-1} & \text{if } a \nsim b \end{cases}$

We can use $X_{a,b}$ in the equation we derived above to get:

$$X_{estimate} = F_2 + \sum_{a\not\cong b} v_a v_b * X_{a,b} + \sum_{a\nsim b} v_a v_b * X_{a,b} =$$

$$F_2 + \sum_{a\neq b} v_a v_b * X_{a,b} \tag{4}$$

16. In line 1 why can we add $X_{a,b}$ in the two places that we did?

17. Why can we make the jump from line 1 to line 2?

3.3.3 Step 4: Taking Expected Value

18. Compute $E[X_{a,b}]$.

    Hint: What is the probability a and b hash to the same bucket? Remember we are using 4-wise independent hashes

19. Explain why $E[(\sum_{a \neq b} v_a v_b) X_{a,b}] = (\sum_{a \neq b} v_a v_b) * E[X_{a,b}]$

20. Compute $E[X_{estimate}]$ using $E[X_{a,b}]$ that you computed above.

Success! You have proven that $E[X_{estimate}] = F_2$

21. Where did we use the independence of hash functions in the proof of $E[X_{estimate}] = F_2$?

3.4 Verify with Code

As in the other labs, lets write some code and verify our predictions. We have provided some skeleton code in SecondMoment.py We have provided.

- m: a class variable representing the same m from the proof - the length of the array you'll be hashing to
- counterArray: a class variable that we initialize to length m. Use this for your counters
- actualDataArray: a class variable we initialize to length 2^{14}. Use this to keep track of the actual data so you can compare your estimate to the actual F_2.
- packetGenerator(): call this and receive a random int (representing a packet) between 0 and 16384. Integer overflow is a very realistic possibility here, so we only use a small universe.
- hash_4i(key): call this for a 4-wise independent hash. It'll hash down to the range $[0, m]$

22. It is up to you to implement the other un-implemented functions. Introduce class variables as you see fit. **Submit your code.**
    - __init__(m): As with any constructor, intialize the object
    - dataStream(numPackets): Generate numPackets packets, hash, and count them. Remember to count using both the array for estimation and the array for the actual data.
    - estimateF2() : Use counterArray and the definition of $X_{estimate}$ to estimate F_2
    - actualF2() : Use actualDataArray to calculate the actual value of F_2

23. Now plot m vs $X_{estimate}$. Use 1000 packets and $m = [50, 100, ...2500]$ Plot actual F_2 on the same graph. **Submit your code and plot.**