CS 49/249: Randomized Algorithms (Spring 2021) Coding Avenues

This is an alive document which will refresh frequently. Check every weekend for newer problems.

Can be done in groups of size ≤ 2.

Instructions

- *Credit Statements:* At the beginning of each problem you must write who all you discussed this problem set with (including the teaching staff). This is ***important.*** Even if you did not talk with anyone about any of the problems, you need to mention "No one". *Without a credit statement, you may be awarded* 0 *for the p-set.*
- All coding assignments need to be submitted using **Jupyter Notebooks as .ipynb files**. You should use your notebooks to (a) describe what you are doing in text, (b) writing code which is commented well, and (c) showing plots, test-runs, etc. You should produce quality stuff which you would be happy to show to the public. Indeed, after the course is done, you should proudly put them up on GitHub.
- These assignments can give you up to 20 of the "Engagement Points." The points near each problem is my (possibly bad) estimate of the time taken (in person-hours) to do the question.

Problem 1 ("Experimental Analysis" of QUICKSORT)**.** (**4 points**)

In class, we saw that the expected number of comparisons made by the QuickSort algorithm is at most $2nH_n$ where $H_n := 1 + \frac{1}{2} + \cdots + \frac{1}{n}$. In this assignment, the goal is to experimentally figure out the "shape" of the "running time random variable." Here are the to-do's for this assignment.

a. First implement Quick Sort. You can google and find out the command to randomly choose an element among a set of elements.

b. In your implementation, maintain a **global** counter which counts the *number of comparisons* QUICKSORT ever makes.

c. For $n = 100$ and 1000, do the following. Fix array A_n by taking a random permutation of $\{1, 2, \ldots, n\}$. Next run QUICKSORT on A_n for 100 iterations, storing the global number of comparisons for each of these 100 iterations. Plot a histogram of these using MATPLOTLIB to see the "shape" of the random variable.

d. Finally, for every n going from 100 to $10,000$ in jumps of 100, do the following:

- Fix an array A_n obtained as a random permutation of 1 to n.
- Run QUICKSORT on A_n, $T = 100$ times, and take the average of all the comparisons made using the global counter. Call this average $\hat{T}_n$.
- Set $\theta_n := \frac{\hat{T}_n}{2nH_n}$

Plot θ_n as a function of n. Does practice match theory?

e. How many person-hours did it take to do this? Give an honest estimate.

Problem 2 ("Seeing" the Chernoff Bound.)**.** (**3 points**)

In this exercise, you actually take samples of $X = \sum_{i=1}^{n} X_i$ where each X_i is a Bernoulli random variable and **see** that X really stays near its expectation. Do this for the following two "extremes". Repeat the following for $n = 10, 100, 1000$.

a. Sample n Bernoullis $X_1, \ldots, X_n$ where each $X_i \sim \{0, 1\}$ with probability $\frac{1}{2}$. Set $X = \sum_{i=1}^{n} X_i$. Repeat this $10,000$ times and plot the histogram. Experimentally evaluate what is $\mathbf{Pr}[X \geq 1.1\,\mathbf{Exp}[X]]$ for the three n's.

b. Repeat the above except each X_i is now a Bernoulli which is 1 with probability $\frac{1}{n}$. For this one, also experimentally evaluate $\mathbf{Pr}[X \geq 10]$ for these three n's.

Problem 3 (Estimating the digits of π)**.** (**3 points**)

Consider the following Monte-Carlo algorithm to generate the digits of π. Let S be the square with four coordinates $[\pm\frac{1}{2}, \pm\frac{1}{2}]$ of side length 1. Let C be the circle centered at the origin of diameter 1 which is completely contained in S. Sample a random point $p := (x, y)$ in S by sampling each coordinate independently and uniformly in the range $[-\frac{1}{2}, +\frac{1}{2}]$, and check if $p \in C$ or not (how will you do this?). Consider a random variable $\widehat{\mathsf{est}}$ which is set to 0 if $p \notin C$ and otherwise set to some constant A. What should A be such that $\mathbf{Exp}[Z] = \pi$.

Once you figure this out, do the following. Take $N = 10^6$ iid estimates of $\widehat{\mathsf{est}}$. With these perform the following:

- Find Z_1 which is simply the average of *all* million of the $\widehat{\mathsf{est}}$'s.
- Find Z_2 as follows. Divide the 10^6 samples into 10 groups of 10^5. For each find the average. Take the median of these 10 averages.
- Find Z_3 and Z_4 as in the previous part, where instead of breaking into 10 groups of 10^4, you break (for Z_3) into 100 groups of 10^4, and (for Z_4) into 1000 groups of 1000.

For each of these four estimates, look at the real π and report to how many digits do they agree.

Problem 4 (Balls and Bins Plots)**.** (**10 points**)

In the balls-and-bins problem, one throws m balls into n bins resulting in a load vector $(\mathsf{L}_1^{(m)}, \ldots, \mathsf{L}_n^{(m)})$. In class, we looked at some theoretical bounds. In this exercise, you are being asked to see them empirically. There are lots of plots to be made here. So make sure you do a good job in annotating it in your Jupyter notebook. Ideally, anyone reading this should really appreciate it.

a. (Birthday Paradox) In class we calculated the ***exact*** probability of there existing a bin with at least two balls in it (see notes). For $n = 100, 1000, 10000, 100000, 1000000$, create plots with m on the x-axis and the probability of collision on the y-axis. For each of these, also "zoom" into the region $m = \frac{1}{3} \cdot \lceil \sqrt{n} \rceil$ to $m = 3\lceil \sqrt{n} \rceil$. Do you see "sharp jumps" here? Is there some empirical inference you can make in this zone?

b. (Balls-and-Bins "Simulator".) For most questions on balls and bins, we don't have the above luxury of an exact probability (or at least, not with such ease). So, build a simulator as follows. Given the pair (m, n), do the balls-and-bins experiment: create n counters and run m for loops where in each for loop one of the counters is randomly picked and incremented. Now, check if the final counter values satisfy your event. Repeat this T times, and take the average to get an estimate of the probability. Let's call this the (m, n, T) simulation of this event.

   (a) (Birthday Paradox Redux.) Suppose the event you are interested in is : "at least one bin has at least **three** balls" (three people share a birthday). For a fixed $n = 100, 1000, 10000$, do the following. Set $T = 1000$. For m starting at $\lceil \sqrt{n} \rceil$ and going to n in adequate number of jumps (you choose), run the (m, n, T) simulation for this event to get an estimate p_m. Plot p_m versus m. This will give you three plots. Can you now guess what m should look like, as a function of n, for $p_m \approx 1/2$? If interested, go ahead and do the same for more n's.

   (b) (Coupon Collector.) Suppose the event you are interested in is : "every bin has at least one ball", that is the coupon collector problem. For $n = 10, 100$, do the following. Set $T = 10000$. Keep running the (m, n, T) simulation till the event occurs. Note the m-values, and plot the histogram. This should give you a good picture of the distribution of the random variable X we found the expectation of in class (and which you are asked to theoretically analyze a bit more in the problem set).

   (c) (The Load Vector). For $n = 10, 100, 1000$. Set $T = 1000$. Let $m = n$. Obtain the load vector (the counters) and **sort** it in decreasing order. Plot various things.

   - For the largest entry, plot the histogram over the 1000 iterations. This is the distribution of $\max_i \mathsf{L}_i^{(n)}$.

- Take the **average** of the T sorted load vectors. Show the first $\lceil \sqrt{n} \rceil$ entries of this average. At what point does this average become < 2?

If you are up for it, create a "sliding scale" where people can play with the m and n's in your balls and bins simulator. I don't think I have seen such a thing — but it will be super educational to "know" what these distributions really look like. And if you have other ideas of visualizations, let your creativity flow!

Problem 5 (Count Sketch and Count Min)**. (15 points)**
In this exercise, you are asked to implement COUNT-MIN and COUNT-SKETCH, and test their performance on `descartes.txt` provided to you on Canvas (see "Tex+Data Files" module.)

Let me give some guidance on how to proceed. First, let us try to figure out an experiment to even do. The file is (hopefully) not that large, and you can actually find the **exact** frequencies of all words in the file. This is the "ground truth" and you should figure this out first, and also measure how much space this takes in memory (there should be a python call for that). Once you are done with this, figure out the top 30 common words and note down their **true frequencies** $\mathbf{f}_i$. Let this set of 30 common words be called W. This is what we will compare the two algorithms with.

Let us now come to implementation. There are a few things to decide on before you begin coding.

- Where will we get our hash functions from? Note that for both algorithms, we need t different **independent hash functions** $h : [n] \to [k]$. Here $[n]$ is the domain, but k is something we need to decide on. In the next bullet point we talk about k. Let's talk about h. There are a couple of ways one can do this.
  - One can actually go ahead an implement the Carter-Wegman functions. This is, in itself, a separate coding assignment. Rather, I recommend you doing the next.
  - The second is to use some hash functions out there. One example in Python (thanks to Sam Lensgraf for pointing this out to me) is `murmurhash3` or simply `mmh3`.
    * You have to first install this on your system. I use anaconda, so for me the following worked: `conda install -c conda-forge mmh3`. I must confess: `pip` didn't work for me, but these are precisely the problems Google can solve.
    * What `mmh3` allows one to do is you can pass a *seed*. So, for instance `mmh3.hash("apple", seed=0)` returns something completely different from `mmh3.hash("apple", seed=1)`. Thus, having different seeds can mimic independent hash functions.
- What should k be? Here one can do two experiments.
  - Run both algorithms with the same values of k; so $k = \{100, 1000, 10000\}$, and then compare the errors $|\widehat{\mathbf{f}}_i - \mathbf{f}_i|$ for $i \in W$, the two 30 words. Note once we fix value of k, the space required for both algorithms is similar. Plot space versus error averaged over the 30 words. Make sure you clearly spell out what you do in the text of your Jupyter notebook.
  - Pick the word $i \in W$ with largest $\mathbf{f}_i$. Keep raising k such that you get (averaged over 10 runs, say) error $|\widehat{\mathbf{f}}_i - \mathbf{f}_i| \leq 0.01\mathbf{f}_i$. What is this k for COUNT-MIN and COUNT-SKETCH. Report with the $i \in W$ with the smallest $\mathbf{f}_i$ as well.
- What should δ be? This is the easiest to answer, let's just set it to $1\% = 0.01$. This will fix the value of t in bullet point 1.

Bonus (5 points): Run COUNT-MIN and COUNT-SKETCH on the much bigger `shakespeare-all.txt` file (see canvas "Tex+Data Files" module). Is there any word which appears more with relative frequency ($\mathbf{f}_i/F_1$) more than 0.001? How about 0.0001? This is the heavy-hitters question.

Problem 6 (Estimating Distinct Items)**.** (**10 points**)
DeepC: This assignment is completely due to Samuel Lensgraf with very minor tweaks in the wording from my end. Thanks, Sam!

Our goal is to understand how diverse the vocabulary of a set of works of classic literature is. Download the file `shakespeare-all.txt` (see canvas "Tex+Data Files" module). We want to understand how many different words are used in this dataset. We do this by implementing a naive distinct element counter and Flajolet-Martin. We will test both the murmurhash implementation of hashing (see the exercise above) and python's own standard 'hash' function. We will need to implement the following:

a. Implement a function `get-trailing-zeroes` which counts the number of trailing zeroes in the binary representation of its argument.

b. Implement the naive ground truth element counter which simply keeps a python dictionary of counts for each element.

c. Implement the FLAJOLET-MARTIN algorithm from Lecture 19. Your Flajolet-Martin implementation should be general to the hash function (ie they accept the hash function as an argument). This will allow you to test many different hash functions!

Use the naive ground truth algorithm on both datasets and report the results. Using murmurhash and python's standard hash function compare the results of Flajolet-Martin against the ground truth. Do they meet the theoretical quality guarantees from class? For the Flajolet-Martin algorithm track how many elements it takes to increment k to each value. Plot this value. We would expect this function to look something like 2^k. Is that the case?

Addition by DeepC (for an extra 5 points) Implement BJKST-BASIC algorithm and compare estimate with FLAJOLET-MARTIN.