COMPSCI 514: Problem Set 1

Released: 8/30.

Due: 9/11 by 8:00pm in Gradescope.

Instructions:

- You are allowed to, and highly encouraged to, work on this problem set in a group of up to three members.
- Each group should **submit a single solution set**: one member should upload a pdf to Gradescope, marking the other members as part of their group in Gradescope.
- You may talk to members of other groups at a high level about the problems but **not work through the solutions in detail together**.
- You must show your work/derive any answers as part of the solutions to receive full credit.

1. Probability Practice (14 points + 4 bonus)

1. (3 points) Prove the union bound using Markov's inequality. That is, prove that for any events $\mathbf{B}_1, \ldots, \mathbf{B}_t$, $\Pr[\mathbf{B}_1 \cup \ldots \cup \mathbf{B}_t] \leq \sum_{i=1}^{t} \Pr[\mathbf{B}_i]$.

2. (2 points) Recall that Markov's inequality only applies to *non-negative* random variables. Consider a random variable $\mathbf{X}$, that always takes values greater than some floor f, which may be negative or positive. Give as tight an upper bound as you can on $\Pr(\mathbf{X} \geq t)$ for any $t > f$ (in terms of $\mathbb{E}[\mathbf{X}]$, f, and t).

3. **Bonus:** (2 points) For any $t > f$, describe a distribution on $\mathbf{X}$ for which your bound in part (2) is tight (i.e., the upper bound on the probability that $\mathbf{X} \geq t$ is equal to the true probability.

4. (4 points) Let $\mathbf{X}_1, \ldots, \mathbf{X}_n$ be independent, identically distributed random variables with mean μ and variance σ^2. Let $\boldsymbol{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} \mathbf{X}_i$ be the sample mean. Suppose one estimates the variance using the sample mean rather than the true mean, i.e.,

$$\boldsymbol{\sigma}_n^2 = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{X}_i - \boldsymbol{\mu}_n)^2.$$

   Prove that $\mathbb{E}[\boldsymbol{\sigma}_n^2] = \frac{n-1}{n} \cdot \sigma^2$, and thus, to have an unbiased estimate of the variance one should have divided by $n-1$ rather than n. **Hint:** Start by expanding $\mathbf{X}_i - \boldsymbol{\mu}_n = \mathbf{X}_i - \boldsymbol{\mu} + \boldsymbol{\mu} - \boldsymbol{\mu}_n$. Later use that $\boldsymbol{\mu}_n - \boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{X}_i - \boldsymbol{\mu})$.

5. (5 points) Let $\mathbf{Z}$ be distributed uniformly on $[0, 1]$.

(a) What is $\Pr(\mathbf{Z} \geq 3/4)$?

(b) Give an upper bound on $\Pr(\mathbf{Z} \geq 3/4)$ using Markov's inequality.

(c) Apply Markov's inequality to $\mathbf{Z}^2$ to give a tighter upper bound.

(d) What happens when you try to give a bound using higher moments? I.e., apply Markov's to $\mathbf{Z}^r$ for r up to 5 and describe what you observe.

(e) **Bonus:** (2 points) Exhibit a monotonic function g so that applying Markov's to $g(\mathbf{Z})$ gives as tight a bound on $\Pr(\mathbf{Z} \geq 3/4)$ as you can.

2. How Random is Your Hash Function? [1] (17 points + 6 bonus)

In Python, hash tables are known as dictionaries. Until 2012, a single deterministic hash function was used for *all dictionaries*. Hash values that collided in one Python program would do so for every other program. To avoid denial of service attacks, in which malicious users attack an application by choosing adversarial inputs that cause a large number of hash collisions, Python implemented randomized hashing.

If you run Python 2 with a $-R$ flag, a hash function that will be used throughout that session will be chosen randomly at the beginning of the session. If you run Python 3, a hash function is chosen randomly by default. To see this for yourself try running `python2 -R -c 'print(hash("a"))'`. Try running also without the `-R` flag. And with Python 3: `python3 -c 'print(hash("a"))'`.

The random hash implementation in Python 2 is unfortunately broken. This question will explore what is going on.

1. (3 points). Python 2's hash function maps any input to a 64-bit integer. Assume that for any input x, $\mathbf{h}(x)$ is equally likely to be any 64-bit integer (i.e., $\mathbf{h}$ is 1-universal). Consider running Python 2 with the `-R` flag n different times and computing `hash("a")` (in each run, a different random hash function will be applied to the string `"a"`.). Give an upper bound on the probability that we will see some hash value more than once. **Hint:** First compute the expected number of duplicate hash values we will see.

2. (3 points) Actually run the two different Python versions (Python 2 with the `-R` flag and Python 3). For each, report how many repeated values of `hash("a")` you see over $n = 2000$ runs. Do your results make sense in light of part (1)? **Hint:** You might want to call Python using Bash or another scripting language.

3. (2 points) Consider running Python 2 with the `-R` flag n different times and computing `hash("a")-hash("b")` on each run. If Python 2's random hash function were pairwise independent, give an upper bound on the probability that we will see some difference more than once. E.g. that `hash("a") - hash("b")` $= 1001$ on run 10 and also on run 20. **Hint:** Start by giving an upper bound on the probability that `hash("a") - hash("b")` $= k$ for any value k and then use a similar approach to part (2).

4. (2 points) Actually run the two different Python versions (Python 2 with the `-R` flag and Python 3). For each, report how many repeated differences you get over $n = 2000$ runs. How do your results compare to the bound of part (3) and the results of parts (1)-(2)? What do they indicate about Python 2's random hash function implementation?

[1] This problem is due to Eric Price at UT Austin.

5. (4 points) **Bonus:** Give an example of a random hash function $\mathbf{h} : U \to [m]$, such that $\mathbf{h}(x)$ is equally likely to be any element of $[m]$ (i.e., $\mathbf{h}$ is 1-universal) but $\mathbf{h}$ is not 2-universal. How would your hash function perform in the tests done in parts (1)-(2) and parts (3)-(4)? For simplicity, you may assume that U is just the set of integers from 1 to $|U|$.

6. (4 points) The same differences in `hash("a") - hash("b")` appearing more often than expected might not be a problem in itself, but it is a sign that something is wrong. Consider the strings "8177111679642921702" and "6826764379386829346", which hash to the same value in the non-randomized version of Python 2. You would like to estimate the probability that these two keys hash to the same value in the randomized version of Python 2. How many trials n must you run to ensure that your estimate is within .05 of the true probability, with confidence at least 95%? Use a Bernstein bound to give an upper bound on the required n.

7. (3 points) Actually run the test described above to estimate the probability that these two keys hash to the same value in the randomized version of Python 2. You may want to run for larger n than what is given in (8) to get higher accuracy. Overall, do you think Python 2 is properly implementing a pairwise independent hash function?

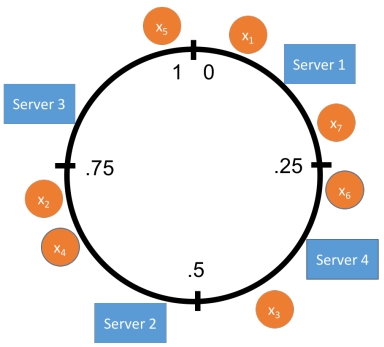
8. (2 points) **Bonus:** Can you find any pair of strings x, y such that: `hash(x) - hash(y)` repeats even more frequently than `hash("a") - hash("b")` or such that `hash(x) = hash(y)` occurs at an even higher rate than `hash("8177111679642921702") = hash("6826764379386829346")` in Python 2.

For full credit, include any code used in your pdf submission. We will not run the code, but will sanity check it.

3. Randomized Load Balancing Meets Scalability (10 points)

Consider a large scale distributed database, storing items coming from some universe U. A random hash function $\mathbf{h} : U \to [m]$ is used to assign each item x to one of m servers. When that item is queried in the future, the hash function is used to identify which server it is stored on. In many applications, the number of servers scales dynamically, depending on the storage load, availability, etc. If a new server is added to the current set of m, since $\mathbf{h}$ maps only to $[m]$, we will have to pick a new hash function, rehash and move all the stored items.

Consider the following solution: pick a random hash function $\mathbf{h}$ which maps both items and servers to values chosen independently and uniformly at random in the range $[0, 1]$. Each item is assigned to the first server to its right, with wrap-around. I.e., item x is assigned to the server with the smallest hash value larger than $\mathbf{h}(x)$. If there are no servers with a larger hash value, the item is assigned to the server with the smallest hash value. When a new server is added to the system it is hashed to $[0, 1]$ and any items that should be assigned to it are moved.



1. (3 points) If we have n items, originally stored on m servers, and we add a new server using this method, what is the expected number of items that will be moved to that new server? **Hint:** First show that the expected size of the hash range assigned to any server when there are m servers is $\frac{1}{m}$. Then use that the $(m+1)^{st}$ server is hashed in exactly the same way as the previous servers.

2. (4 points) Show that with probability $\geq 9/10$, with m servers in total no server is assigned a hash range of width greater than $\frac{10 \ln m}{m}$. **Hint:** Fix a single server and then think about how the remaining $m-1$ servers must be hashed so that its hash range has size $\geq \frac{10 \ln m}{m}$.

3. (3 points) Show using a concentration bound that with probability $\geq 9/10$ the maximum load on a server is $\leq \frac{20n \ln m}{m}$.

You may assume that m and $\frac{n}{m}$ are both large, say > 30 to get your bounds to hold. You may want to use the helpful inequality: for any $x > 0$ and $c > 0$ with $\frac{c}{x} \leq 1$, $\left(1 - \frac{c}{x}\right)^x \leq e^{-c}$.