1. Some Useful Inequalities (9 points)

There are a number of very useful inequalities that come up over and over again in randomized
algorithm analysis, and more generally in probabilistic analysis that are worth knowing well.

1. (1 point) Show that for any \( \epsilon \in [0, 1/2] \), \( \frac{1}{1+\epsilon} \leq (1 + 2\epsilon) \).

2. (1 point) Show that for any \( \epsilon \geq 0 \), \( \frac{1}{1+\epsilon} \geq (1 - \epsilon) \).

3. (1 point) Show that for any \( \epsilon \in [0,1] \), \( (1+\epsilon)^2 \leq 1 + 3\epsilon \).

4. (1 point) Show that for any \( \epsilon \in [0,1] \) and any integer \( t \geq 1 \), \( (1 - \epsilon)^t \geq 1 - t\epsilon \).

5. (2 points) Prove that for any \( x \), \( 1 + x \leq e^x \). Use this to show that for any \( x > 0 \) and \( c > 0 \) with \( \frac{c}{x} \leq 1 \), \( (1 - \frac{c}{x})^x \leq e^{-c} \).

6. (3 points) Given an investment with rate of return \( c\% \) per year, a common rule of thumb is that the value of the investment will double every \( \frac{70}{c} \) years. Give an upper bound and a lower bound on how much the investment will actually grow in \( \frac{70}{c} \) years, which is valid for all \( c \in [0, 70] \). You may assume for simplicity that \( 70/c \) is an integer, so you don’t need to think about fractions of years.

Plot your bounds for \( c \in [0, 70] \) vs. the actual growth for different values of \( c \). When is the rule of thumb accurate?
2. **Messing Around with Markov’s Inequality** (9 points + 2 bonus)

1. (3 points) Prove the union bound using Markov’s inequality. That is, prove that for any events \(A_1, \ldots, A_m\), \(\Pr[A_1 \cup \ldots \cup A_m] \leq \sum_{i=1}^{m} \Pr[A_i]\).

2. (1 point) For any \(t > 0\) exhibit a non-negative random variable \(X\) for which Markov’s inequality is tight. I.e., \(\Pr[X \geq t] = \frac{E[X]}{t}\).

3. (5 points) Let \(X\) be distributed uniformly on \([0, 1]\).
   
   (a) What is \(\Pr(X \geq 7/8)\)?
   
   (b) Give an upper bound on \(\Pr(X \geq 7/8)\) using Markov’s inequality.
   
   (c) Apply Markov’s inequality to \(X^2\) 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 \(X^r\) for \(r\) up to 10 and describe what you observe.
   
   (e) **Bonus:** (2 points) Exhibit a monotonic function \(f\) so that applying Markov’s to \(f(X)\) gives as tight a bound on \(\Pr(X \geq 7/8)\) as you can.

3. **Bloom Filters Meet Machine Learning** (10 points)

Consider applying Bloom filters to database optimization: you work for an online retailer and want to store a large table of \((customer, product)\) pairs, with information about if a customer bought a certain product, when they bought it, where they shipped it, how they rated it, etc. Most pairs in this table are empty since you sell a large number of products and any customer has only purchased a tiny fraction of them. So, whenever a customer buys a product, you insert information on that pair into the database and also add the pair to a bloom filter. Before you make a possibly expensive query to the the table, you check the bloom filter and only query the table if it returns a hit.

1. (3 points) You have \(s = 10\) billion \((customer, product)\) pairs with information stored in your database. You would like to store these pairs in a bloom filter with false positive rate of 10%. How large must your filter be to achieve this rate? I.e., how large must you set the bit array size \(m\) and how much storage in kBs, MBs, or GBs does it take? You may use the false positive rate calculation given in class of \(\left(1 - e^{-km}\right)^k\) – even though its classic derivation is incorrect, it is a good approximation. Throughout you may assume that you use the optimal number of hash functions \(k = \ln 2 \cdot \frac{m}{n}\).

2. (4 points) The machine learning team at your company has developed a classifier that is able to predict if a customer has bought a product with very good accuracy, without having to look at the customer-product database. The classifier is a function \(y: (customer, product) \rightarrow \{0, 1\}\) which returns 1 on 95% of \((customer, product)\) pairs where the customer has actually purchased the product and returns 0 on 95% of pairs where the customer hasn’t purchased the product.

You decide to use this classifier to reduce the load on your bloom filter. You only add a \((customer, product)\) pair \((c, p)\) to the filter if the customer has purchased the product and \(y(c, p) = 0\). When any pair \((c, p)\) is queried you first check if \(y(c, p) = 1\) – if so you query the pair in the your database. If not, you check your bloom filter, and if it returns a positive you then query the database.
What is the overall false positive rate on a random pair \((c, p)\) where \(c\) has not purchased \(p\) among the database size \(s\), the filter size \(m\), and the number of hash functions \(k\).

3. (3 points) How large must you set \(m\) to still have a false positive rate of 10%. How much storage have you saved using this ML approach compared to the standard approach in (1).

4. Network Size Estimation via Colliding Crawls (11 points)

You want to estimate the number of nodes \(n\) in a large network (the Facebook social network, the web, etc.), by randomly crawling the network. Assume that after enough steps of randomly hopping from node to node in the network, the final node you land at is sampled uniformly at random from all nodes in the network.

1. (1 point) Consider sending out \(t\) independent random crawls that end at \(t\) random nodes. Let \(C\) denote the number of pairwise collisions between these nodes. Prove that \(E[C] = \frac{\binom{t}{2}}{n}\).

2. (2 points) Let \(C_{i,j}\) be an indicator random variable which is 1 if crawls \(i\) and \(j\) collide (i.e., end at the same node) and 0 otherwise. Argue that the \(C_{i,j}\) random variables are pairwise independent. I.e., for any two unordered pairs \((i, j)\) and \((k, ℓ)\) that differ in at least one element and any \(s, t \in \{0, 1\}\), \(\Pr[C_{i,j} = s \cap C_{k,ℓ} = t] = \Pr[C_{i,j} = s] \cdot \Pr[C_{k,ℓ} = t]\).

3. (3 points) Prove that for any set of pairwise independent set of random variables \(X_1, \ldots, X_z\),

\[
\text{Var} \left[ \sum_{i=1}^{z} X_i \right] = \sum_{i=1}^{z} \text{Var}[X_i].
\]

This is, pairwise independence suffices for linearity of variance to hold. **Hint:** It may be useful to show that for any \(i \neq j\), \(E[X_i \cdot X_j] = E[X_i] \cdot E[X_j]\).

4. (1 point) Use the above fact to bound \(\text{Var}[C]\).

5. (2 points) Consider estimating the network size as \(\tilde{n} = \frac{\binom{t}{2}}{C}\). How large must you set \(t\) so that \(\Pr(|n - \tilde{n}| \geq \epsilon n) \leq \delta\) for \(\delta > 0\)? (Give an expression in terms of \(n, \epsilon, \text{ and } \delta\).) **Hint:** Start by showing how to pick \(t\) so that \(C\) is close to its expectation with good probability.

6. (1 point) Can you apply an exponential tail bound (Chernoff bound or Bernstein’s inequality) to effectively bound the accuracy the above estimation procedure? Why or why not?

7. (1 point) Why is the assumption that the random crawls land on uniformly random nodes unrealistic? How might this issue affect your estimator — would it tend to make it an overestimate or an underestimate? How might you combat it?

You may assume that the number of crawls \(t\) is less than the network size \(n\) in all bounds. You do not need to work out precise constants in any bounds (i.e., may use Big-O notation.)
5. How Random is Your Hash Function? ¹ (7 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: python2 -c ‘print(hash("a"))’.

1. (3 points) The random hash implementation in Python 2 is broken. To see this, consider running Python 2 with the -R flag n different times. We will look at the difference in hash values hash("a") - hash("b") on each run. If the random hash function were pairwise independent over the range of 64-bit integers, given n different runs (i.e., n different random hash functions), 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.

2. Bonus: (3 points) 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 you results compare to the bound of part (1)? What do they indicate about Python 2’s random hash function implementation? Hint: You might want to call Python using Bash or another scripting language.

3. (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 90%? Use a Bernstein bound to give an upper bound on the required n.

4. Bonus: (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 (3) to get higher accuracy. Given your estimate, do you think Python 2 is properly implementing a pairwise independent hash function? How many random hash functions do you think Python 2 is actually ‘choosing from’? Why does this still leave its dictionaries vulnerable to attacks?

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

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