CS 630, Fall 2024, Homework 6 Solutions
Due Wednesday, November 13, 2024, 11:59 pm EST, via Gradescope

Homework Guidelines

Collaboration policy Collaboration on homework problems, with the exception of programming assignments and reading quizzes, is permitted, but not encouraged. If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 5 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (including generative AI tools or anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must provide:

1. pseudocode and, if helpful, a precise description of the algorithm in English. As always, pseudocode should include
   - A clear description of the inputs and outputs
   - Any assumptions you are making about the input (format, for example)
   - Instructions that are clear enough that a classmate who hasn't thought about the problem yet would understand how to turn them into working code. Inputs and outputs of any subroutines should be clear, data structures should be explained, etc.

   If the algorithm is not clear enough for graders to understand easily, it may not be graded.
2. a proof of correctness,
3. an analysis of running time and space.

You may use algorithms from class as subroutines. You may also use facts that we proved in class.

You should be as clear and concise as possible in your write-up of solutions. A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

Problem 1 *Bloom Filter Transformations (5 points)*

The Bloom filter B is of size m (number of bits), where m is a power of 2. So far n items have been inserted into B.

In this problem we create a new Bloom filter B' that is half the size of B, thus has $m/2$ bits. We "copy" the current content of B to B' in the following way: we denote the first $m/2$ bits of B by B_1, the second $m/2$ bits is denoted by B_2. Then we take the bit-wise Boolean *or* (inclusive-or) of B_1 and B_2, thus $B' = B_1 \vee B_2$.

Suppose $h_1(x), \ldots h_r(x)$ are the hash functions originally used by B. We will use the same functions for B' with the modification that $h_i(x)$ mod $(m/2)$ for each i.

1. Show that for every element $x \in B$ the look up in B' will return positive.

   Solution. **long answer.** We show that any x that gets a positive response when queried in B also gets a positive response in B'.

   Suppose that $h_i(x) = z$ in B. If z is an index in the range of B_1 (thus, in the first $\frac{m}{2}$ bits of the array) then it's hash value when using mod $(m/2)$ remains unchanged. Hence, it's hashed to the same index in B' as in B_1 and is represented in the bitwise or.

   Suppose that z is in the range of B_2. Then we can write z as $z = \frac{m}{2} + (z - \frac{m}{2}$. The second part of this is the index of z in the subarray B_2. In B' the item x maps to $zmod(m/2) = z - \frac{m}{2}$. This is the same index that corresponds to z in the bitwise or.

   In conclusion, every bit that x is hashed to in B is also represented in B', hence it would solicit a positive response.

   short answer. let k be the index of a bit in B. If $k \leq m/2$ then the location of k in B' is the same. Hence if $B[k] == 1$, then $B'[k] = 1$ too. If $k > m/2$ then bit k gets copied to k mod $(m/2)$. Any hash value that originally mapped to k will now map to k mod $(m/2)$, hence the corresponding bit value is 1 both in B and B'.

2. Compute and explain the false negative and false positive rates in B' for items that

   Solution. We know that for Bloom filters in general false negatives don't exist by design. Hence the false negative rate is 0 in all three cases below.

   (a) were in B prior to creating B'

   Solution. From part 1. we know that any item x in B is also in B' by the design of B'. Thus, for these items $Check(x)$ will correctly return True and there are no false positives nor false negatives.

   (b) were not in B

   Solution. Suppose B has n keys inserted. There are r hash functions Then we know from lecture that the expected number of 0 bits in B is $n(1 - 1/m)^{rn}$. Following the analysis from the slides we get that the false positive probability is $f \approx (1 - e^{-rn/m})^r$.
   Now let's apply this formula to B'. It has the same number of inserted keys as B, the only difference is it's size. If we plug that into the formula we get $f \approx (1 - e^{-2rn/m})^r$.

(c) now $O(n)$ ~~few~~ more items are inserted into B' what are the false negative and false positive rates now?

Solution. The answer to this is almost the same as (b). Now the number of keys is $O(n)$ instead of n. If we write $O(n)$ as cn where c is a constant. Then we get $f \approx (1 - e^{-rcn/m})^r$.

Problem 2 *Hash Evaluations for Bloom Filters. (5 points)*

In this problem, you will be asked to analyze the number of invocations of the hash function of a Bloom filter for different queries. The Bloom filter has m bits and n items were inserted using $k = \lfloor \ln 2(m/n) \rfloor$ hash functions.

1. How many times are the hash functions invoked when inserting an item into the Bloom filter?

   Solution. The hash functions are always invoked k times for insertions into the Bloom filter. **1 point.**

2. What is the average number of invocations of the hash functions when checking an item that was inserted into the Bloom filter?

   Solution. The hash functions are always invoked k times when checking items that were inserted into the Bloom filter. **1 point.**

3. What is the average number of invocations of the hash functions when checking an item that was not inserted into the Bloom filter? *Hint: do not forget that the check can stop at the first zero bit found.*

   Solution. The fraction of bits set to one, p, has expectation $1/2$ or less due to rounding down k to an integer. The first hash function is always invoked. The second hash function is invoked with probability p. The third hash function is invoked with probability p^2 when the first two hash functions both find one bits. The kth hash function is invoked with probability p^{k-1} when the previous $k-1$ hash functions all found one bits. So the number of hash function invocations is $\sum_{i=0}^{k-1} p^i = \frac{1-p^k}{1-p}$. Within the range of expectations, this is maximized at $p = \frac{1}{2}$, yielding at most $2 - (\frac{1}{2})^{k-1}$ hash invocations. **3 points.**

Problem 3 *Exploiting Weak Hash Functions (10 points)*

In this problem, you will be given "blackbox" access to a hash function used by a service. With this blackbox, you may invoke the hash function h with input x and the output bucket index $h(x)$ of the input will be returned. All you know about the hash function $h(x)$ is that it takes in integer inputs and has the form $h(x) = ax + b \pmod{2^k}$ where a, b, and k are unknown non-negative integers. To be clear, you know that the result you get is really just the hash function, and is not affected by collisions in the table.

1. Write and justify an algorithm to determine a, b assuming $k = 64$ with as few invocations of $h(x)$ as possible. To avoid ambiguity, return the smallest non-negative values for a and b that work.

Solution.

Invoking $h(0)$ returns b. **1 point.**

Invoking $h(1)$ returns $a + b \pmod 6 4$. So subtract $h(1) - h(0)$ and take the remainder mod 64 to recover a. **2 points.**

2. Later, you are hired as a consultant by this company after they notice weird performance of their hash table. Their architecture uses hashing with separate chaining, and the same linear hash function that you previously analyzed. Upon inspection, you note that only ~~half~~ the buckets with odd indexes have any data, so those odd buckets have twice the expected load factor. And after further testing with many random keys, you are unable to find any items that are into even buckets. What can you infer about the current hash function $h(x) = ax + b \pmod{2^k}$? How should we modify a and b to avoid this problem?

   Solution. This scenario can only occur if a is even and b is odd. If a is odd and $h(x)$ is odd, then $h(x+1)$ would be even. If a is even and b is even, then $h(x)$ is always even. If a is even and b is odd, then $h(x)$ is always odd. (The modulus is a power of two so it doesn't change the reasoning about parity.) **2 points for identifying which choices of a and b that trigger the issue. There should be enough detail to make it clear even a values are the problem. -1 point if the case with even a and even b is not mentioned or described as a better choice.**

   Since even a always fixes the parity of $h(x)$ to match b, a should always be chosen to be an odd integer. Since the modulus is a power of two, the specific odd a is not particularly important. The value of b is not important as it just rotates the item placement and does not change the distribution of bucket loads. **1 point for identifying that the fix is to always pick odd a.**

3. The company has noticed that 1/1024 of the buckets have significantly more load than average. They suspect that an adversary – who knows the structure of the hash function – is deliberately sending data to cause bucket collisions.

   Identify a set of keys that could be chosen to cluster in 1/1024 of the keys only knowing that the hash function the form $h(x) = ax + b \pmod{2^k}$ where a, b, and k are unknown non-negative integers.

   After identifying the problem, suggest a minimal change to the choice of the hash function design to prevent this problem recurring.

   Solution. This situation can be forced by always picking different x values which are multiples of 1024. As long as k is at least 10 to get at least 1024 buckets, then their hash functions will be equivalent to b modulo 1024. That will yield the observed imbalances. There are other ways to drive up the load on that fraction of buckets, but that is the most trivial one. **2 points for this attack or something similarly easy. Full credit for trying lots of random keys and filtering to 1/2024th. Deduct 1 point if the attack is clearly more expensive than filtering random keys.**

   There are many fixes to this problem. Two possible fixes are changing the modulus to not be a power of two and hash function to not be linear. **2 points for any fix as long as it addresses the load problem from just picking multiples of 1024.**