CS 630, Fall 2024, Homework 4
Due Wednesday, November 6, 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 *Fair numbers from biased coins. (6 points)*

In class we saw the following problem: we are given a biased coin, that with probability p will flip to heads. Design a method – using this biased coin – to generate heads or tails with equal ($p = 1/2$) probability. We will call the algorithm from class 2-WayFair(p). This is a function that takes as input a probability p, and returns the values 0 or 1 with equal probability.

In this problem you have to think about how to generalize this approach.

1. One iteration of the discussed algorithm consists of two coin flips. Compute the expected number of iterations it takes for 2-WayFair(p) to return an answer.

2. *(UPDATE: clarified the text on how to use 2-WayFair.)* Design the algorithm 3-WayFair(p) that makes calls to 2-WayFair(p). Your algorithm can only use randomness by making calls to 2-WayFair(p). It should take as input the probability p and return $0, 1$ or 2 with equal probability. Aim to use a minimum number of calls to 2-WayFair(p) [1].

   Write your algorithm, then prove that the values are returned with uniform probability. Compute the expected number of iterations of your algorithm (what should constitute of one iteration?)

3. Generalize this algorithm to k-WayFair(p) to generate a uniform random number $0, 1, \ldots k-1$ by making calls to 2-WayFair(p).

   Write your algorithm, then prove that the values are returned with uniform probability. Compute the expected number of iterations of your algorithm (what should constitute of one iteration?)

Problem 2 *Uniform sample (7 points)*

Suppose there are n numbers. A *uniform sample* of size k is a subset S of k numbers, such that each of the n numbers is equally likely - with probability $\frac{k}{n}$ – to be in S.

In this problem we are working with a *stream* of numbers. This means that numbers arrive one at a time (with no specific frequency), for infinite amount of time. Note that this also implies that we don't know how many numbers will eventually come.

You can use the notation `stream.next` to retrieve the next number in the stream. You can use the notation `random(k)` to generate a random integer in $1, ...k$.

1. Design an algorithm that continuously maintains a sample S of size 3. That is, at any given time, if so far n numbers have arrived, then the probability of any given number to be in S should be $3/n$.

   Your algorithm should only use constant amount of space and take constant time per iteration, i.e. arrival of new number. (*In this part only write your algorithm, either (few) English words or pseudocode. Also analyze the overall space and per-iteration time complexity.*)

2. Prove that in your algorithm in part (1.) after n numbers have arrived, each has probability $\frac{3}{n}$ to be in S.

[1] In fact you can create a fair random generator to return $0, 1, 2$ by flipping coins in groups of 3. You can think of this just for fun!

3. Describe a Generalization of your algorithm to maintain a sample of size k instead. (*This description should take you 1-2 sentences.*) Prove that numbers are in S with uniform probability.

Problem 3 *hash (7 points)*

Consider a hash table with chaining that allows multiple values with the *same key* to be added – instead of replacing the previous value. (Such a data structure is sometimes called a "multimap".)

Answer the following questions about this modified data structure using the variables m and n to respectively represent the total number of items inserted and the number of slots in the hash table. Be sure to explicitly describe any necessary algorithmic changes from the version presented in class to support your claims for the modified data structure.

1. Write pseudocode to efficiently insert a new key x into the hash table H and analyze its running time. *Hint: what is different updating a multimap instead of a normal hash table?*

2. Suppose that H already contains some keys, and some have been inserted many times. You are presented a key x and told that it was already inserted in H. Write pseudocode to confirm that x was inserted into H at least once, and analyze its *average* running time assuming it actually was inserted into H. The performance of your code should not depend on the number of times x was inserted into H - even if this number is large, e.g. $m/2$ times.

3. Analyze the average runtime of verifying that some other key y is *not* in H.