CS 630, Fall 2024, Homework 5 Solutions
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.

   Solution. The probability of being successful in one iteration is $2p(1-p)$. Because the probability of getting HT is $p(1-p)$, getting $TH = (1-p)p$.

   Let $k = 2p(1-p)$, then the probability of being successful in iteration t is $(1-k)^{t-1}k$. Let X be the random variable that indicates the number of iterations until success. That is, X can take on values $X = 1, 2, \ldots, \infty$ Note that this is a geometric distribution, hence we compute the expected value of the geometric distribution. The expected number of iterations, i.e. t is

   $$E[X] = \sum_{t=1}^{\infty} t(1-k)^{t-1}k = \frac{1}{k}$$

   Substituting back p we get $E[X] = \frac{1}{2p(1-p)}$

2. 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?)

   Solution.

   Algorithm. We will use a similar approach - waiting for combinations of coin-toss events that are equally likely - but now tailored to 3 possible outcomes. Specifically, we generate three outputs using 2-WayFair(p). If the three are identical, i.e. all are H or all are T, then we repeat the three calls. Otherwise, there are 3 options based on which outcome is different from the other two:

   - the first outcome is different: HTT or THH $\rightarrow$ return 0
   - the second outcome is different: THT or HTH $\rightarrow$ return 1
   - the third outcome is different: TTH or HHT $\rightarrow$ return 2

[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!

Algorithm 1: 3-WayFair(p)

```
1  while True do
2      c1 ←2-WayFair(p);
3      c2 ←2-WayFair(p);
4      c3 ←2-WayFair(p);
5      if c1 ≠ c2 OR c2 ≠ c3 then
           /* the three outcomes are not identical                    */
6          if c2 == c3 then
7              return 0
8          else if c1 == c3 then
9              return 1
10         else
11             return 2
```

uniform outcome. We have to show that the probability of returning $0, 1$ or 2 has equal probability. We know that 2-WayFair(p) returns H or T with probability 0.5.

We can compute the probability that 3-WayFair(p) returns one of $0, 1, 2$ in iteration t: The probability of returning one specific sequence of of 3 is $\frac{1}{2^3}$ since H and T are equally likely. Two of those sequences correspond to the 0 outcome, thus $p(0) = 2 \cdot \frac{1}{2^3} = \frac{1}{4}$, two correspond to each of the other cases, thus $p(0) = p(1) = p(2) = \frac{1}{4}$. We can make the same argument for any iteration of 3-WayFair(p).

expected number of iterations. *Note that since we only ask for the expected number of iterations, this is much easier than computing the expected number of total coin tosses.*

We consider one iteration of 3-WayFair(p) to be the 3 calls to 2-WayFair(p). The probability of being successful in any specific iterations is $\frac{3}{4}$. The probability of being successful in iteration t is $(1-\frac{3}{4})^{t-1}\frac{3}{4}$. We can use the exact same computation as before. Let X be the random variable that indicates the number of iterations until success. Then $E[X] = \sum_{t=1}^{\infty} t(1-\frac{3}{4})^{t-1}\frac{3}{4} = \frac{1}{\frac{3}{4}} = \frac{4}{3}$.

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?)

   Solution. *This solution is very inefficient but will be accepted for full credit.* The generalization of the previous approach is to generate k values such that only one is different from the others. The downside of this algorithm is that the probability of success in any iteration is going to be less and

less as k is increasing.

Algorithm 2: k-WayFair(p)

```
while True do
    for i = 0 to k − 1 do
        c_i ←2-WayFair(p);
    if c_1 + c_2 + ... + c_k == 1 OR c_1 + c_2 + ... + c_k == k − 1 then
        /* exactly one of the values is 1 or exactly one is 0 */
        for i = 0 to k − 3 do
            if c_i ≠ c_{i+1} AND c_i ≠ c_{i+2} then
                /* index i is different from the others */
                return i
            else if c_{k−2} ≠ c_{k−3} AND c_{k−2} ≠ c_{k−1} then
                return k − 2
            else
                return k − 1
```

uniform outcome. The probability of returning one specific sequence of k 0/1 values is $\frac{1}{2^k}$. Two sequences apply to each returned value i, thus the probability of returning i is $\frac{1}{2^k} \cdot 2 = \frac{1}{2^{k-1}}$.

expected number of iterations. One iteration of k-WayFair(p) consists of k calls to 2-WayFair(p). The probability of being successful in any specific iteration is $p(success) = k \cdot \frac{1}{2^{k-1}}$. The probability of being successful after t iterations is $(1 - p(success)^{t-1} p(success)$. Let X be the random variable for the number of iterations until success. Then the expected value can be computed as $E[X] = \sum_{t=1}^{\infty} t(1-p(success))^{t-1} p(success) = \frac{1}{p(success)} = \frac{1}{\frac{k}{2^{k-1}}} = \frac{2^{k-1}}{k}$. (This is a lot of iterations... but will be accepted for full credit)

Solution. **The efficient solution.** Here we generate the integers $0 \ldots k-1$ as binary numbers.

case where k is a power of 2. For ease of explanation we first assume that k is a power of 2, i.e. $k = 2^\ell$. Then any number $0 \ldots k-1$ can be expressed on $\log k = \ell$ bits. In our algorithm we call 2-WayFair(p) ℓ-times to generate 0 or 1 for each bit at random.

Algorithm 3: k-WayFair.v.2(p)

```
ℓ ← log k;
val ← 0;
for j = 0 to ℓ do
    /* generate ℓ bits and convert their sum to a decimal */
    bit ← 2-WayFair(p);
    val ← val + 2^j · bit;
return val
```

uniform outcome. Since each bit is equally likely to be 0/1 every binary number on ℓ bits is equally likely to be generated. There are k such numbers on ℓ bits, thus each output has probability $1/k$.

expected number of iterations. We consider one iteration to be either the generation of one bit or ℓ bits. In either case, since 2-WayFair only returns with a successful outcome, the number of iterations is fixed (either ℓ or 1 depending on our interpretation).

general k. Now we apply this solution to any integer k. If k is not a power of 2, then let $\ell = \lfloor \log k \rfloor$ and we have $2^\ell \leq k \leq 2^{\ell+1}$. We use the same algorithm as before; we generate a binary number, this time on $\ell + 1$ bits. If the value is $0 \dots k-1$ we return it, otherwise we generate a new number.

Algorithm 4: k-WayFair.v.2.1(p)

```
1 ℓ ← ⌊log k⌋;
2 val ← k + 1;
3 while val ≥ k do
4 |   val ← 0;
5 |   for j = 0 to ℓ + 1 do
  |   |   /* generate ℓ bits and convert their sum to a decimal   */
6 |   |   bit ← 2-WayFair(p);
7 |   |   val ← val + 2^j · bit;
8 return val
```

uniform outcome. Since each bit is equally likely to be 0/1 every binary number on $\ell + 1$ bits is equally likely to be generated (no matter whether they are $\leq k - 1$.

expected number of iterations. We consider one iteration to be the generation of one $\ell + 1$ bit integer. We know that in worst case $k = 2^\ell + 1$, in which case half (-1) the generated numbers are rejected by the algorithm. Thus we have $p(success) > \frac{1}{2}$ in any particular iteration. Let X be the random variable of how many iterations are needed until success. Then we have
$E[X] \leq \sum_{t=1}^{\infty} t \left(\frac{1}{2}\right)^t = \frac{1}{\frac{1}{2}} = 2$.

Problem 2 *Uniform sample (7 points)*

UPDATE: you should use a random number generator that generates a number between 0 and 1, NOT an integer to k.

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( )` to generate a random number in $[0, 1]$.

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.*)

Solution. **algorithm.** Initially we add the first 3 numbers that arrive to S. Suppose that the nth number x has just arrived. With probability $\frac{3}{n}$ we add x to S, otherwise we reject it. If we add x, then we have to reject one of the current numbers in S, we select the number to be rejected with equal probability of 1/3 each.

complexity analysis. This algorithm onlu uses space to store S, which is constant. The running time is also constant per iteration as it requires the generation of two random numbers.

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

   Solution. We will prove this by induction on the number n of numbers that have arrived.

   base case. When we have 3 numbers they are all in the sample with probability $1 = 3/3 = 3/n$.

   inductive hypothesis. Suppose that for any $\ell = 3 \dots n-1$ in iteration k each number is in S with probability $3/e \ll$.

   prove for n. Before n arrives by the inductive assumption the three items in S are there with probability $\frac{3}{n-1}$. The new item is in the sample with probability $\frac{3}{n}$ by design of the algorithm. Let y be one of the items in S. The probability that y is *not* rejected from the sample is $1 - \frac{3}{n} \cdot \frac{1}{3} = 1 - \frac{1}{n} = \frac{n-1}{n}$. Since y was on the sample with probability $\frac{3}{n-1}$, hence it is on the sample after the arrival of x with probability $p(\text{y } \textit{in sample}) = \frac{k}{n} \cdot \frac{n-1}{n} = \frac{k}{n}$.

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.

   Solution. The algorithm is the same as for 3 except to replace the number 3 in the random choices by k. The proof of probability is also identical with the use of k instead of 3.

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?*

   Solution. Since duplicate values are allowed, the pseudocode merely needs to hash the new key x and prepend it at the beginning of the linked list. This takes $O(1)$ time. Appending is also acceptable if the linked list maintains a pointer to the end of the linked list.

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.

   Solution. The key detail here is that this code just needs to confirm that x was inserted into H, and it does not need to access all copies of x in H. So it just needs to scan the bucket of $h(x)$ until it finds the first copy of x, and then it returns true. The number of copies of x in that bucket does not matter since it stops at the first one. If it gets all the way to the end of the bucket, then it returns false. The average number of items besides x in that bucket is bounded by $\alpha = m/n$, so this takes $O(1+\alpha)$ time on average.

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

   Solution. Like the previous problem, this algorithm stops if it finds a copy of y in the bucket, but returns false if y is found and true if the whole bucket is scanned without finding y. This algorithm also stops at the first match to y, so the average time is also $O(1+\alpha)$.