MPCS 55001 Algorithms Autumn 2023
Homework 2

Zhiwei Cao zhiweic
Collaborators: N/A

Instructions: Write up your answers in LaTeX and submit them to Gradescope. **Handwritten homework will not be graded.**

Collaboration policy: You may work together with other students. If you do, state it at the beginning of your solution: **give the name(s) of collaborator(s)** and the nature and extent of the collaboration. Giving/receiving a hint counts as collaboration. Note: you must write up solutions **independently without assistance**.

Internet sources: You must include the url of any internet source in your homework submission.

1 Flipping Coins (10 points)

Let us consider how we can use an unfair coin to simulate the behavior of a fair coin and vice versa.

(a) Suppose you had an unfair coin, represented by a function `UNFAIR`(), that returns `HEAD` with probability p and `TAIL` with probability $1-p$, where $0 < p < 1$. How could you use that coin to implement an algorithm `FairFromUnfair`() that returns `HEAD` with probability 1/2 and `TAIL` with probability 1/2?

(i) (2 points) Write **pseudocode** for a randomized algorithm to solve this problem. Is your algorithm Las Vegas or Monte Carlo? Explain.

Algorithm 1 Find Fair Probability From the UNFAIR() Function

```
function FairFromUnfair()
while (True) do
    cast1 = UNFAIR()
    cast2 = UNFAIR()
    if cast1 = HEAD & cast2 = Tail then
        return HEAD
    else if cast1 = Tail & cast2 = HEAD then
        return Tail
    end if
end while
```

The algorithm is Las Vegas.
Since the algorithm always provides the correct result (Head or Tail), the Monte Carlo's return result is probabilistic. However, the running time is not fixed in this algorithm, it depends on the UNFAIR() function, if two casts from the UNFAIR function could provide the sequence that FairFromUnFair() required, the algorithm could end quickly, otherwise not. In sum, the algorithm fits the idea of Las Vegas, which ensures the result, but not the running time.

(ii) (1 points) What is the expected number of calls to `UNFAIR` that `FairFromUnfair` will make before returning? Explain your answer.

In order to reach the if statement in the FairFromUnfair(), it is required to make sure that we can get the sequence of [HEAD, TAIL] or [TAIL, HEAD], which the probability of getting these two situations is $p*(1-p)$ and $(1-p)*p$. The rest of other cases' probability is $1-(p*(1-p))*2$, which is $1-2p*(1-p)$.

Then $E = 2 * 2p(1-p) + 4(1-2p(1-p)) * 2p(1-p) + 6(1-2p(1-p)^2) * 2p(1-p) + \ldots$, which follows the geometric distribution with parameter $2p(1-p)$. In sum, the expected value is $1/p(1-p)$ generally.

(b) Using a fair coin, represented by a function `FAIR()` that returns `HEAD` with probability 1/2 and `TAIL` with probability 1/2, give an algorithm that, when given a positive integer n, generates an integer chosen uniformly at random from 0 to n (inclusive).

(i) (2 points) Write **pseudocode** for a randomized algorithm to solve this problem.

Algorithm 2 Generate a random number bases on FAIR() function

```
function generateRandomNum(n)
Input: n: An integer n
Output: An integer that is randomly generated in the range from 0 to n.
bitLen ← getBitLenOfN(n)// a function that can find the bit length of an integer
res ← 0
for i in 0, ..., bitLen do
    coinRes ← FAIR()
    if coinRes ← HEAD then
        res ← res * 2 + 1
    else
        res ← res * 2
    end if
end for
if res <= n then
    return res
end if
```

(ii) (2 points) What is the expected number of calls to `FAIR()` your algorithm makes? Explain.

Based on the algorithm, we can acquire a number called *bitLen* at the beginning, which shows the number of bits required that could represent the input number n. As a result, based on the *bitLen*, the algorithm could provide a number in the range $[0, \ldots, n^{bitLen}]$. Then, the probability of getting the res in the range $[0, \ldots, n]$ is $[n/(2^{bitLne} - 1)]$, and the probability that res out of the range is $[1 - n/(2^{bitLne} - 1)]$. Furthermore, the expected value is $1/(n/(2^{bitLne} - 1))$, equal to $[(2^{bitLne} - 1)/n]$, based on the geometric distribution. Since the algorithm will call the $FAIR()$ function for *bitLen* time. Then the expected value of FAIR() calls is $[bitLne * (2^{bitLne} - 1)/n]$

(iii) (1 point) How would you do this if you had an unfair coin? What is the expected number of calls to `UNFAIR()` your adjusted algorithm would make?

Based on the question (a), I can know that it requires $1/(2p(1-p))$ calls to produce a fair result in calling the $UNFAIR()$ function. As a result the expected number of calls to $UNFAIR()$ should be $1/(2p(1-p)) * [bitLne * (2^{bitLne} - 1)/n]$

(c) Using a fair coin, represented by the same `FAIR()` function as above, give an algorithm that, when given a **rational** number $p = a/b$ where a and b are integers and $0 < p < 1$, returns `HEAD` with probability p and `TAIL` with probability $1 - p$.

(i) (2 points) Write **pseudocode** for a randomized algorithm to solve this problem.

Algorithm 3 Find the HEAD probability p

function findHEADPr(a,b)
Input: a: An integer a
Input: b: An integer a, which fulfill $p = a/b$ which $0 < p < 1$
Output: The p, which is the probability of the HEAD
$bitLen \leftarrow getBitLenOfN(a, b)$// a function that can find the bit length of the rational number a/b
for i in $0, \ldots,$ $bitLen$ **do**
  $cast1 = UNFAIR()$
  $cast2 = UNFAIR()$
  if $cast1 = HEAD$ & $cast2 = Tail$ **then**
    return HEAD
  else if $cast1 = Tail$ & $cast2 = HEAD$ **then**
    return Tail
  end if
end for
return HEAD

Algorithm 4 Find the bit length of the rational number a/b

function getBitLenOfN(n)
Input: a: An integer a
Input: b: An integer a, which fulfill $p = a/b$ which $0 < p < 1$
Output: Bit length of the rational number a/b
$bitLen \leftarrow 0$
while $a > 0$ **do**
  $a \leftarrow a * 2$
  if $a \geq b$ **then**
    $a \leftarrow a - b$
    $bitLen \leftarrow bitLen + 1$
  end if
end while
return bitLen

2 Reservoir Sampling (10 points)

A data stream is a sequence of objects (or events) seen one at a time (e.g., requests to a web server, or cars passing as you stand on the side of a highway). Typically we may look at each object in the stream one at a time, and then never again (it cannot be replayed). A stream may have infinite length, but even if it is finite, its length is usually not known in advance. Assume all objects in the stream are distinct.

Going forward (on all future problems unless we tell you otherwise), you may use any standard functions to generate random numbers in your pseudocode.

(a) (2 points) Suppose that you want to choose a single object uniformly at random from a finite stream with unknown length. The number of objects may be very large, so you cannot write all of them down and then pick at random later on.

Write **pseudocode** for an algorithm `Sample(S)` which takes as input a stream S and returns a single object chosen uniformly at random from the stream.

You should assume that you can iterate over the stream (e.g., `for item in S`) exactly one time. Your algorithm must use $O(1)$ space and only spend $O(1)$ time processing each object.

Algorithm 5 Find single object chosen uniformly at random from the stream

function Sample(S)
Input: S: a stream S
Output: returns a single object chosen uniformly at random from the stream
$res \leftarrow None$
for i in $0, \ldots, S.length$ **do**
  $prob \leftarrow random(1, i+1)$ // random choose a number in the range from 1 to i+1
  if $prob = 1$ **then**
    $res \leftarrow S[i+1]$
  end if
end for
return res

(b) (2 points) Prove that your algorithm is correct, i.e., if S has n objects, it will return each object in S with probability $1/n$. Remember: n is not an input to the algorithm, but we can still use it for analysis.

Prove the algorithm by the induction.
Base Case: n=1, then the probability is 1/1 for the base case
Assume that after k steps, the probability for the k-th object is $1/k$
Inductive Step: Prove that for k+1-th object, the probability is $1/(k+1)$.
When the algorithm reaches k+1 iteration. There must be a random object o that can be selected after k-th. As a result, the probability is 1/k. Meanwhile, if the algorithm will not select k+1 item in k+1-th iteration, the probability of non-select is $(k+1-1)/(k+1)$ which equals $k/k+1$. At the end, the probability of k+1-th object is $1/k+1$.

(c) (3 points) Write **pseudocode** for an algorithm `Sample(S, k)` which returns an array of k distinct objects sampled from the stream S instead of just one. You may assume that $k \leq n$, where n is the (unknown) number of objects in the stream.

Your algorithm must return each of the $\binom{n}{k}$ possibilities with equal probability. The order of your output array does not matter. It must use $O(1)$ time to process each object, and $O(k)$ space in total.

Algorithm 6 Find an array of k distinct objects sampled from the stream S instead of just one

```
function Sample(S,k)
Input: S: a stream S
Input: k: distinct objects
Output: an array of k distinct objects sampled from stream S instead of just one
res ← an array with size k
for i in 0, . . . , S.length do
    if i < k then
        res[i] ← S[i]
    else
        prob ← random(1, i + 1) // random choose a number in the range from 1 to i+1
        if prob < k then
            res[prob − 1] ← S[i]
        end if
    end if
end for
return res
```

(d) (3 points) Prove that your algorithm in part (c) is correct: if there are n objects in S, then your algorithm returns each subset of size k with probability $1/\binom{n}{k}$.
Base case: The probability after $k-th$ object is $k/k+1$, since for the first $k-th$ object, the probability of the object stored in the array is 1.
Then, assume that after m objects which $k < m < n$, and the algorithm will return each subset of size m with the probability $1/\binom{m}{k}$.
Inductive Step: Already known that, after $k-th$ objects, the probability of the object can be stored in the array is $k/k+1$. At the same time, the probability that it can replace an object in an array is $1/k$.

3 Probabilistic Testing and Primality (15 points)

In this problem we discuss probabilistic testing and then use that technique to find large primes used in cryptography.

3.1 An Introductory Example (from Nick Harvey)

Answer questions (a)–(j) in a sentence or two.

Suppose we are given an **even-length** vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ and need to differentiate between two cases:

- **All-zero case.** All entries of α are 0 ("Yes" case).
- **Half-zero case.** Exactly half of the entries in α are 0 and half are 1 ("No" case). However, we do not know in advance *which* bits are 0.

The trivial *deterministic* algorithm is to check $n/2+1$ bits and if all of them are zero, return "Yes"; otherwise return "No".

Algorithm 7 Deterministic Solution

```
function IsAllZero(bit vector α = (α_1, α_2, . . . , α_n))
for i = 1 . . . n/2 + 1 do
    if α_i = 1 then
        return "No"
    end if
end for
return "Yes"
```

(a) (1 point) Give a brief argument that Algorithm 7 **always** returns the correct answer.

Since the YES case is strict which requires all of the elements to be 0, as long as there are any elements equal to 1, it automatically turns to No case.

3.2 Different Types of Error

In computer science, problems with Yes/No answers are called decision problems. The above problem is an example of that class. Let's consider the possible outcomes that can occur in randomized algorithms for decision problems.

Correct Output \ Algorithm's Output	Yes	No
Yes	true positive	false negative
No	false positive	true negative

Table 1: The possible outcomes and their conventional names.

- A ***false positive*** is when the correct output is No, but the algorithm outputs Yes. In statistical hypothesis testing this is also called a "Type I error".
- A ***false negative*** is when the correct output is Yes, but the algorithm outputs No. In statistical hypothesis testing, this is also called a "Type II error".

In the current terminology of machine learning, the table itself is called a confusion matrix.

Consider the following randomized algorithm for identifying an all-zero vector that only looks at a single bit.

Algorithm 8 Constant Time Randomized

```
function ConstRandIsAllZero(bit vector α = (α_1, α_2, ..., α_n))
Pick X ∈ [n] uniformly at random
if α_X = 1 then
    return "No"
end if
return "Yes"
```

(b) (1 point) Is Algorithm 8 a Las Vegas or Monte Carlo type algorithm? Why?

It belongs to Monte Carlo. Since it cannot make sure that the output is always correct. For example, if the algorithm picks an element is 0 (return YES case), but there there half 0 and half 1 in the vector (No case).

(c) (1 point) If $\alpha_1 = \alpha_2 = \ldots = 0$ (the **all-zero** case), what is the probability of a false negative? Briefly explain.

0, it cannot find any 1 at all.

(d) (1 point) If half of the α-vector is 0 (the **half-zero** case), what is the probability of a false positive? Briefly explain.

1/2, since there are half-zero and half-one. It is the No case, so he probability that the algorithm checks 0 vectors is 1/2

3.3 Types of Randomized Algorithms

So far you have characterized an algorithm as "good" or "bad" depending on the resources it needs, such as running time, memory, or network messages sent. However, in randomized algorithms we also care about false positive and false negative rate.

The first type of algorithms are the ones that for *every* input whose correct output is `No` they always return `No` and for *every* input whose correct output is `Yes`, the algorithm must output `Yes` with probability at least 1/2. These are called **randomized polynomial time** or **RP-type** algorithms.

The second type of algorithms are the ones that *every* input whose correct output is Yes they always return Yes and for *every* input whose correct output is No, the algorithm must output No with probability at least 1/2. These are called **coRP-type** algorithms.

RP-type and **coRP-type** have **one-sided error**. When an algorithm has both, it is said to have **two-sided error**. For those algorithms a useful definition is **bounded-error probabilistic polynomial time** or **BPP-type** algorithms that have false positive and false negative error rate below 1/3.

Correct Output \ Algorithm's Output	RP-type		coRP-type		BPP-type	
	Yes	No	Yes	No	Yes	No
Yes	$\geq 1/2$ true positive	$\leq 1/2$ false negative	1 true positive	0 false negative	$\geq 2/3$ true positive	$\leq 1/3$ false negative
No	0 false positive	1 true negative	$\leq 1/2$ false positive	$\geq 1/2$ true negative	$\leq 1/3$ false positive	$\geq 2/3$ true negative

Table 2: Confusion matrix for different types of randomized algorithms.

(e) (1 point) What kind of algorithm is `ConstRandIsAllZero`?

coRP-type.

3.4 Boosting for one-sided error

Consider the following algorithm.

Algorithm 9 Boosted Randomized

function BoostRandIsAllZero(bit vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, integer k)
for $i = 1$ **to** k **do**
  if ConstRandIsAllZero(a) = "No" **then**
    return "No"
  end if
end for
return "Yes"

(f) (1 point) In the **all-zero** (correct output "Yes") case, what is the probability of a false negative? Briefly explain.

0, since the ConstRandIsAllZero will never return Yes, for the BoostRandIsAllZero function.

(g) (1 point) In the **half-zero** (correct output "No") case, what is the probability of a false positive? Briefly explain.

It depends on how many iterations the BoostRandIsAllZero can run. Before BoostRandIsAllZero detects the "No", ConstRandIsAllZero have probably of 1/2 to get the false positive situation. As a result, the probability of BoostRandIsAllZero to get the false positive situation is $(1/2)^{timesofiterations}$

3.5 Primality Testing

The public key cryptographic protocol RSA uses extremely long primes (at least 1024 bits long). As a result, we need to find very large primes efficiently. Fermat's little theorem states that $\forall a \in \mathbb{N}^+ \; \forall$ primes $p : a^{p-1} \equiv 1 \pmod{p}$. In addition, for any composite number q, at least half the integers a will be "witnesses," meaning $a^{q-1} \not\equiv 1 \pmod{q}$.

Algorithm 10 Fermat's Primality Test

function IsPrime(p)
Select $a \in [p-1]$
if $a^{p-1} \equiv 1 \pmod{p}$ **then**
  return "Yes"
end if
return "No"

(h) (1 point) Between RP-type, coRP-type, and BPP-type, which one describes `IsPrime`? Why?

coPR-type. Since if the p is a prime, the algorithm's output ensures that p may be prime (true composite = 1). However, if p is a composite, the algorithm may return that the p may be prime (YES) or not a prime (NO).

(i) (1 point) If p is prime, what is the probability of a false negative? Briefly explain.

Based on the table of coRP-type, the probability is 0. Based on the algorithm, it always returns the correct value for the prime number but may return YES (is the prime number) for the composite number.

(j) (1 point) If p is not prime, what is the probability of a false positive? Briefly explain.

Based on the table of coRP-type, the probability is $\leq 1/2$. Based on the algorithm, it may return YES (is the prime number) for the composite number.

(k) (5 points) Follow this Google Colab template link to implement Fermat primality testing and boosting. **Do not attempt to edit the linked file. Create your own copy instead by clicking File → Save a copy in Drive**. You do not need to write anything in your .pdf document for 3(k).

When you have completed the notebook, download the .ipynb file. **Important: you must submit this problem separately from .pdf file which contains the rest of your theory solutions. Submit the**

completed .ipynb file to the Gradescope assignment named HW 2 - Problem 3(k) Notebook.

4 Programming: Sortedness (15 points)

Follow this GitHub Classroom link to accept the assignment. Your code should be pushed to GitHub; do not include it here.

After completing the sortedness test in `sortedness.py`, complete the following.

- From `test_sortedness.py` use `generate_array(n, ϵ)` to create almost-sorted arrays of size $n = 1,000,000$.
- Values for ϵ should vary starting at 10^{-1} down to 10^{-5} by multiplying with $10^{-1/2}$ every time.
- On **each** of the generated arrays run your `isEpsilonSorted` function 100 times with the **same list** of ϵ-values as the one you used to generate the arrays and count how many times it returned sorted.
- For each array, plot the probability of `isEpsilonSorted` returning sorted and the ϵ-value you used. The ϵ-axis should be in logarithmic scale (use `semilogx`). What do you observe?
- Submit your plots in your GitHub Repository. Make sure that they are added to version control.