Design and Analysis of Algorithms: Homework #4

Due in class on March 27, 2018

Professor Kasturi Varadarajan

Alic Szecsei

Problem 1

Consider the following randomized algorithm for generating biased random bits. The subroutine **FairCoin** returns either 0 or 1 with equal probability; the random bits returned by **FairCoin** are mutually independent.

```
function OneInThree
    if FairCoin = 0 then
        return 0
    else
        return 1−OneInThree
    end if
end function
```

(a) Prove that **OneInThree** returns 1 with probability $\frac{1}{3}$.

(b) What is the *exact* expected number of times that this algorithm calls **FairCoin**?

(c) Now suppose you are *given* a subroutine **OneInThree** that generates a random bit that is equal to 1 with probability $\frac{1}{3}$. Describe a **FairCoin** algorithm that returns either 0 or 1 with equal probability, using **OneInThree** as your only source of randomness.

(d) What is the *exact* expected number of times that your **FairCoin** algorithm calls **OneInThree**?

Solution

Part A

In order for **OneInThree** to return 1, **FairCoin** must return an odd number of 1s before returning a 0. We can begin to enumerate cases: first, that **FairCoin** returns a 1 and then a 0 is $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$. The probability that **FairCoin** returns 3 1s and then a 0 is $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{16}$. The probability that **FairCoin** returns 5 1s and then a 0 is $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{64}$, and so on. Our summation of all cases is then:

$$\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \ldots + \frac{1}{2^{2n}} = \sum_{k=1}^{\infty} \frac{1}{2^{2k}} \tag{1}$$

This is a geometric series with common ratio $r = \frac{1}{4}$; we can then use the formula $S = \frac{a_1}{1-r}$ to determine the summation. Using this, we can see that $S = \frac{\frac{1}{4}}{1-\frac{1}{4}} = \frac{\frac{1}{4}}{\frac{3}{4}} = \frac{1}{3}$. Therefore, **OneInThree** has a $\frac{1}{3}$ probability of returning 1.

Part B

Since **OneInThree** might never terminate, our expected value of $T(n)$ is:

$$E[T(n)] = \sum_{k=1}^{\infty} k \cdot \Pr[T(n) = k] \tag{2}$$

In order for **OneInThree** to terminate with exactly k calls to **FairCoin**, there must have been exactly $k-1$ times that **FairCoin** returned 1 (and **OneInThree** continued recursing), and 1 time it returned 0 (and **OneInThree** stopped recursing). Thus, the combined probability for **OneInThree** to terminate with exactly k calls to **FairCoin** is $\frac{1}{2^k}$. Plugging this back into our equation for $E[T(n)]$:

Problem 1 continued on next page...

$$
\begin{aligned}
E[T(n)] &= \sum_{k=1}^{\infty} k \cdot \Pr[T(n) = k] \\
&= \sum_{k=1}^{\infty} k \cdot \frac{1}{2^k} \\
&= 2
\end{aligned} \tag{3}
$$

Part C

```
function FairCoin
    c1 ← OneInThree
    c2 ← OneInThree
    if c1 = 1 and c2 = 1 then
        return FairCoin
    else
        if c1 = 1 or c2 = 1 then
            return 1
        else
            return 0
        end if
    end if
end function
```

The idea for this algorithm is that we break the probability space into 3 partitions: first, the probability that both calls to **OneInThree** return 1 is $\frac{1}{9}$. The remaining probability is $\frac{8}{9}$; we further divide this into two partitions. The probability that both calls to **OneInThree** returned 0 is $\frac{4}{9}$; thus, the remaining probability (that only one call to **OneInThree** returned 1) is $\frac{4}{9}$.

Now, the probability that any call to **FairCoin** which actually *returns* will return 0 is clearly $\frac{1}{2}$; the same probability applies to any returning call to **FairCoin** returning 1. Thus, **FairCoin** has an equal probability of returning either 0 or 1.

Part D

Since **FairCoin** might never terminate, our expected value of $T(n)$ is:

$$
E[T(n)] = \sum_{k=1}^{\infty} k \cdot \Pr[T(n) = k] \tag{4}
$$

In order for **FairCoin** to terminate with exactly $2k$ calls to **OneInThree**, there must have been exactly $k-1$ times that both calls of **OneInThree** returned 1 (and **FairCoin** continued recursing), and 1 time at least one call returned 0 (and **FairCoin** stopped recursing). Thus, the combined probability for **FairCoin** to terminate with exactly $2k$ calls to **FairCoin** is $\frac{1}{9^{k-1}} \cdot \frac{8}{9} = \frac{8}{9^k}$. Plugging this back into our equation for $E[T(n)]$:

$$
\begin{aligned}
E[T(n)] &= \sum_{k=1}^{\infty} 2k \cdot \Pr[T(n) = k] \\
&= \sum_{k=1}^{\infty} 2k \cdot \frac{8}{9^k} \\
&= \frac{9}{4}
\end{aligned} \tag{5}
$$

Problem 2

Consider the following algorithm for finding the smallest element in an unsorted array:

function RandomMin($A[1..n]$)
  $min \leftarrow \infty$
  for $i \leftarrow 1$ to n in random order **do**
    if $A[i] < min$ **then**
      $min \leftarrow A[i]$ ▷ $(\star)$
    end if
  end for
  return min
end function

(a) In the worst case, how many times does **RandomMin** execute line $(\star)$?

(b) What is the probability that line $(\star)$ is executed during the ith iteration of the for loop?

(c) What is the *exact* expected number of executions of line $(\star)$?

Solution

Part A

In the worst case, **RandomMin** retrieves the elements in descending order, so it must re-assign min (in line $(\star)$) a total of n times.

Part B

If line $(\star)$ is executed during the ith iteration of the for loop, this means that the randomly selected element is smaller than the previous $i-1$ randomly selected elements. Essentially, we have a set of i elements, and we want to know the probability that a specific element (the most recently added element) is the smallest element in that set. We have a single minimal element, and a $\frac{1}{i}$ chance of choosing that minimal element; therefore, the probability that the most recently added element is the minimal element of the set is $\frac{1}{i}$. We can thus conclude that the probability that line $(\star)$ is executed during the ith iteration of the for loop is also $\frac{1}{i}$.

Part C

We know that line $(\star)$ has a $\frac{1}{i}$ probability of being executed during the ith iteration of the for loop, so the expected value at each iteration of the for loop is:

$$E[T(n)]_{\text{iter}} = 0 \cdot \frac{i-1}{i} + 1 \cdot \frac{1}{i} = \frac{1}{i} \tag{6}$$

We can sum all of these expected values together to determine the expected number of executions of line $(\star)$ during the entire algorithm:

$$E[T(n)] = \sum_{k=1}^{n} \frac{1}{k} = H_n \tag{7}$$

Problem 3

Suppose we have a circular linked list of numbers, implemented as a pair of arrays, one storing the actual numbers and the other storing successor pointers. Specifically, let $X[1..n]$ be an array of n distinct real numbers, and let $N[1..n]$ be an array of indices with the following property: If $X[i]$ is the largest element of X, then $X[N[i]]$ is the smallest element of X; otherwise, $X[N[i]]$ is the smallest among the set of elements in X larger than $X[i]$. For example:

i	1	2	3	4	5	6	7	8	9
$X[i]$	83	54	16	31	45	99	78	62	27
$N[i]$	6	8	9	5	2	3	1	7	4

Describe and analyze a randomized algorithm that determines whether a given number x appears in the array X in $O(\sqrt{n})$ expected time. ***Your algorithm may not modify the arrays*** X ***and*** N***.***

Solution

Problem 4

A *majority tree* is a complete binary tree with depth n, where every leaf is labeled either 0 or 1. The *value* of a leaf is its label; the *value* of any internal node is the majority of the values of its three children. Consider the problem of computing the value of the root of a majority tree, given the sequence of 3^n leaf labels as input. For example, if $n = 2$ and the leaves are labeled 1, 0, 0, 0, 1, 0, 1, 1, 1, the root has value 0.

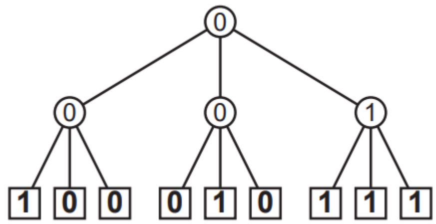

Figure 1: A majority tree with depth $n = 2$.

(a) Prove that *any* deterministic algorithm that computes the value of the root of a majority tree *must* examine every leaf. *[Hint: Consider the special case $n = 1$. Recurse.]*

(b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 3$. *[Hint: Consider the special case $n = 1$. Recurse.*

Solution

Problem 5

Suppose you are given a graph G with weighted edges, and your goal is to find a cut whose total weight (not just number of edges) is smallest.

(a) Describe an algorithm to select a random edge of G, where the probability of choosing edge e is proportional to the weight of e.

(b) Prove that if you use the algorithm from part (a), instead of choosing edges uniformly at random, the probability that **GuessMinCut** returns a minimum-weight is still $\Omega(1/n^2)$.

(c) What is the running time of your modified **GuessMinCut** algorithm?

Solution