

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.

```

1: function ONEINTHREE
2:   if FAIRCOIN = 0 then
3:     return 0
4:   else
5:     return 1 − ONEINTHREE
6:   end if
7: end function

```

- Prove that **ONEINTHREE** returns 1 with probability $\frac{1}{3}$.
- What is the *exact* expected number of times that this algorithm calls **FAIRCOIN**?
- 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.
- 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:

$$\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \dots + \frac{1}{2^{2n}} = \sum_{k=1}^{\infty} \frac{1}{2^{2k}} \quad (1)$$

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:

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

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 $E[T(n)]$:

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

Part C

```

1: function FAIRCOIN
2:    $c_1 \leftarrow \text{ONEINTHREE}$ 
3:    $c_2 \leftarrow \text{ONEINTHREE}$ 
4:   if  $c_1 = 1$  and  $c_2 = 1$  then
5:     return FAIRCOIN
6:   else
7:     if  $c_1 = 1$  or  $c_2 = 1$  then
8:       return 1
9:     else
10:      return 0
11:    end if
12:  end if
13: 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:

```
1: function RANDOMMIN( $A[1..n]$ )
2:    $min \leftarrow \infty$ 
3:   for  $i \leftarrow 1$  to  $n$  in random order do
4:     if  $A[i] < min$  then
5:        $min \leftarrow A[i]$  ▷ (★)
6:     end if
7:   end for
8:   return  $min$ 
9: end function
```

- (a) In the worst case, how many times does **RANDOMMIN** execute line (★)?
- (b) What is the probability that line (★) is executed during the n th iteration of the for loop?
- (c) What is the *exact* expected number of executions of line (★)?

Solution

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.

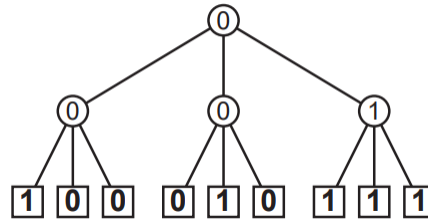


Figure 1: A majority tree with depth $n = 2$.

- (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