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
```

(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

```
1:  function FairCoin
2:      c_1 ← OneInThree
3:      c_2 ← 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 ← ∞
3:     for i ← 1 to n in random order do
4:         if A[i] < min then
5:             min ← 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 $(\star)$?

(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

Our algorithm selects $\sqrt{n}$ random pivots in the linked list, and searches them to find the greatest lower bound on our target value. If our target value is less than all of the pivots, it selects the largest pivot. We then walk through the linked list until we either find our target or exceed its value (with suitable edge cases handled if we need to wrap around).

First, we would like to show that the expected size S of any of the $\sqrt{n}$ partitions our algorithm generates is, itself, $\sqrt{n}$. As an example, we will have 3 partitions of an array of length 9. We randomly select an element of the array r and two pivots, p_1 and p_2; without loss of generality, assume $p_1 < p_2$. The probability that r is between the two pivots is $\frac{1}{3}$, as the other two equiprobable alternatives are that $r < p_1$ and $r > p_2$. The probability that any value is in one of the partitions is $\frac{S}{L}$, where L is the length of the array and S is the partition size; thus, the expected length of each partition must be $\frac{1}{3} = \frac{S}{L} \Rightarrow \frac{1}{3} = \frac{S}{9} \Rightarrow S = 3$.

We can generalize this for any array length L and number of partitions P; the expected partition size must be $\frac{L}{P}$. Since we have $\sqrt{n}$ partitions and an array of size n, our expected partition size is $\frac{n}{\sqrt{n}} = \sqrt{n}$.

Our algorithm uses $O(\sqrt{n})$ runtime to generate our pivot list and determine which partition contains x; it then linearly traverses the relevant partition. Since our expected partition size is *also* $\sqrt{n}$, we can determine our final expected runtime to be $O(\sqrt{n})$.

```
1: function RandomContains(X[1..n], N[1..n], x)
2:     T ← an array of √n random numbers between 1 and n
3:     hasLowerBound ← False   ▷ We need to be able to handle the case where we do not select any elements less than x
4:     for i ← 1, √n do
5:         if X[T[i]] < x then
6:             hasLowerBound ← True break
7:         end if
8:     end for
9:     if hasLowerBound then
10:        minDist ← ∞
11:        for i ← 1, √n do
12:            if X[T[i]] < x and x − X[T[i]] < minDist then
13:                minDist ← x − X[T[i]]
14:                l ← T[i]
15:            end if
16:        end for
17:    else
18:        l ← T[1]
19:        for i ← 2, √n do
20:            if X[T[i]] > X[l] then
21:                l ← T[i]
22:            end if
23:        end for
24:    end if                  ▷ We now have the "closest" element before x (if it exists) on our linked list
25:    while True do
26:        if X[l] = x then
27:            return True
28:        else
29:            prev ← X[l]
30:            l ← N[l]
31:            if X[l] < prev then
32:                if hasLowerBound then
33:                    return False                    ▷ We wrapped around before we found x
34:                else
35:                    hasLowerBound ← True            ▷ We wrapped around, so we should have a lower bound
36:                end if
37:            end if
38:            if X[l] > x and hasLowerBound then
39:                return False        ▷ We had a lower bound, but now we've passed where x should have been
40:            end if
41:        end if
42:    end while
43: end function
```

Problem 4

A *majority tree* is a complete ternary 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.

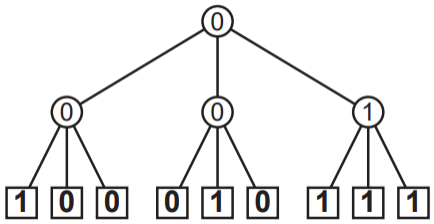

Figure 1: A majority tree with depth $n = 2$.

(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

Part A

In order to determine the value of a node, a deterministic algorithm can use 1, 2, or 3 of its children's values. If an algorithm uses only one child, it has the obvious potential to be wrong (for example, one in which an algorithm only examines a child of value **1**, while the other two children both have values of **0**). If an algorithm only examines 2 of its children's values, it can encounter a tie (in which the algorithm examines children of values **0** and **1**). It cannot determine which of these values to correctly return *unless* the algorithm also references the third child (as the third child may be either **0** or **1**, acting as a tie-breaker). Therefore, for any non-leaf node, the algorithm must examine all 3 of the node's children. The algorithm must be called on every child of the root, and every grandchild, and so on. Since the root has all leaf nodes as its descendents, the algorithm will examine every descendent of the root, which includes all leaf nodes.

Part B

The base case of our algorithm is straightforward; if a node is a leaf (represented by a list containing only 1 element), its value can simply be returned.

The recursive case is somewhat more interesting; we retrieve two random children (sub-lists) of the node and examine their values. If they are the same, we can immediately return this value; no matter what the third child's value is, the majority is already known. However, if the values are *not* the same, one must be a 0 and one must be a 1; the third child acts as a tiebreaker, and we can simply return its value.

In order to construct an expected runtime, we will assume that the worst-case problem is given; namely, that each possible decision contains a mixed distribution of **1**s and **0**s. Then, to determine expected runtime, we want to examine the expected number of recursive calls to MAJORITY. There's a $\frac{2}{3}$ chance of selecting one of the duplicate values

```
1:  function MAJORITY(N[1..n])
2:      if n = 1 then
3:          return N[1]
4:      else
5:          p_1 ← N[1..n/3]
6:          p_2 ← N[n/3 + 1..2n/3]
7:          p_3 ← N[2n/3 + 1..n]
8:          children ← a shuffled list of p_1, p_2, and p_3
9:          a ← MAJORITY(children[0])
10:         b ← MAJORITY(children[1])
11:         if a = b then
12:             return a
13:         else
14:             return MAJORITY(children[2])
15:         end if
16:     end if
17: end function
```

for the first selection. After this, there is a $\frac{1}{2}$ chance of selecting the other duplicate value, for a total probability of $\frac{2}{6} = \frac{1}{3}$. In this case, we would perform 2 recursive calls. Conversely, there must be a $\frac{2}{3}$ probability that the first two calls to **Majority** returned differing values; in this case, we would perform 3 recursive calls. Therefore, the expected number of recursive calls is $2 \cdot \frac{1}{3} + 3 \cdot \frac{2}{3} = \frac{8}{3}$.

We can use this to determine our runtime; namely, $T(n) = \frac{8}{3}T(n-1)$, as we run an expected $\frac{8}{3}$ recursions, and each child has depth of $n-1$. Using annihilators,

$$\begin{aligned} T(n) &= \frac{8}{3}T(n-1) \\ T(n+1) &= \frac{8}{3}T(n) \\ T(n+1) - \frac{8}{3}T(n) &= 0 \\ \left(\mathbf{E} - \frac{8}{3}\right) T(n) &= 0 \end{aligned} \tag{8}$$

Our annihilator is thus $(\mathbf{E} - \frac{8}{3})$. This gives us the generic solution $T(n) = \alpha\frac{8}{3}^n$ for some unknown constant α; therefore, this algorithm runs in $O(\frac{8}{3}^n)$ time.

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

Part A

```
1:  function SelectEdge(G)
2:      t ← 0
3:      for all e in G.edges do
4:          t ← t + e.weight                          ▷ Get total edge weights
5:      end for
6:      r ← a random number between 0 and t
7:      i ← 0                          ▷ i accumulates edge weights until we pass our random number
8:      for all e in G.edges do
9:          i ← i + e.weight
10:         if i > r then
11:             return e
12:         end if
13:     end for
14: end function
```

Part B

Assume without loss of generality that we are given a graph G with m edges, which has a unique min cut consisting of k edges with weights $w_1, w_2, ..., w_k$. Let the weight of the min cut be W_C, and the weight of the graph be $\sum_{e \in E} w_e = W_G$.

We can define the probability that we choose an edge in the min cut as $\frac{W_C}{W_G}$. Since W_C is weight of the min cut, we know that every possible vertex-isolating cut of G must have a weight of at least W_C; that is, for all vertices v, the sum of the weights of all edges connected to v must be greater than W_C.

For each vertex, we can obtain a vertex-isolating cut; which transforms our inequality into:

$$\begin{aligned} \sum_{v \in V} \sum_{e=(v,v')} w_e &\geq \sum_{v \in V} W_C \\ \sum_{v \in V} \sum_{e=(v,v')} w_e &\geq n \cdot W_C \end{aligned} \tag{9}$$

The handshake lemma says that the sum of the degrees of all vertices is equal to $2m$, so our inequality can further be transformed into $2 \sum_{e \in E} w_e = 2 \cdot W_G \geq n \cdot W_C$. Simple algebra suffices to transform this into:

$$\frac{W_C}{W_G} \leq \frac{2}{n} \tag{10}$$

The event that we arrived at an optimal cut means that none of the edges within the cut was selected by our algorithm; the probability that no edge in the min cut was selected during an iteration is $1 - \frac{W_C}{W_G} \geq 1 - \frac{2}{n}$.

This matches the probability for the unmodified GuessMinCut algorithm, so by the same logic as is used to determine that probability of success, the probability of success of our modified GuessMinCut algorithm will *also* be $\Omega(1/n^2)$.

Part C

In order to determine the runtime of the modified GuessMinCut algorithm, we need to first determine the runtime of our SelectEdge algorithm. As this simply iterates twice over the number of edges in the graph, it has a $O(n^2)$ runtime complexity, since there are at most $\frac{n(n-1)}{2}$ edges in a graph.

The GuessMinCut algorithm requires that we iterate over each vertex in G, picking random edges and collapsing them. Picking a random edge, as we just determined, requires $O(n^2)$ time, while collapsing an edge only requires $O(n)$ time. Therefore, the costly operation is picking a random edge, which must happen once for each vertex; the total runtime of the modified GuessMinCut algorithm is thus $O(n^3)$.