Assignment 5

Karan Sikka
ksikka@andrew.cmu.edu
E
March 1, 2012

Through the Rabbit Hole

Let X be a random variable which maps sequences of dice rolls which end in $6, 6$, to the ordinal of the last roll in the sequence.

Say you roll the dice. There are 2 outcomes: you roll a 6 or you don't roll a six. They happen with a probability of $\frac{1}{6}$ and $\frac{5}{6}$ respectively.

Case you don't roll a six:
The problem is exactly as it was when you started, except you increase the roll number by one.

Case you roll a six:
There are two cases. Either you roll a six or you don't:
– **Case you don't roll a six:**
The problem is exactly as it was when you started, except you increase the roll number by two.
– **Case you roll a six.**
Stop rolling! The value of X is two since the ordinal of the *second* roll is *two*.

We can call these events A, B, and C:
A is the event that you don't roll a six.
B is the event that you roll a six and then you don't roll a six.
C is the event that you roll two consecutive sixes. Because dice-rolls are independent, the events have the following probabilities.

$$Pr(A) = \frac{5}{6}$$
$$Pr(B) = \frac{1}{6} * \frac{5}{6} = \frac{5}{36}$$
$$Pr(C) = \frac{1}{6} * \frac{1}{6} = \frac{1}{36}$$

Because of the linearity of expectation, and the fact that A, B, and C are mutually exclusive:

$$E(X) = E(X|A) + E(X|B) + E(X|C)$$

Using our definitions of A B and C defined above,

$$\begin{aligned}
E(X) &= P(A)X(A) + P(B)X(B) + P(C)X(C) \\
E(X) &= \frac{5}{6}(1 + E(X)) + \frac{1}{6}\frac{5}{6}(2 + E(X)) + \frac{1}{6}\frac{1}{6}(2) \\
E(X) &= \frac{5}{6}(1 + E(X)) + \frac{5}{36}(2 + E(X)) + \frac{1}{36}(2) \\
36E(X) &= 30(1 + E(X)) + 5(2 + E(X)) + 2 \\
36E(X) &= 30E(X) + 5(2 + E(X)) + 32 \\
36E(X) &= 35E(X) + 42 \\
E(X) &= 42
\end{aligned}$$

Now it is clear that the the cat will be 42 minutes late on average.

Don't Eat the Mushroom

---

a.
```
x = Bernoulli(1/2)
y = Bernoulli(1/2)
z = Bernoulli(1/2)
return x + y*2 + z*4 + 1
```

b.
```
RandInt(10)%2
```

c.
```
if Bernoulli(2/5) == 1
  if Bernoulli(1/4) == 1
    return 2
  else
    return 1
else
  if B(1/12)==1
    return 3
  else
    return 4
```

d.
```
x = Bernoulli(1/2)
y = Bernoulli(1/2)
z = x + y
if(z == 0) return "FAIL"
if(z == 1) return 0
if(z == 2) return 1
```

Proof:

Let A be the event that the program returns 0
Let B be the event that the program returns 1
Let C be the event that the program returns "FAIL"

Note that since the events are mutually exclusive, $C^c = A \cup B$.

Now we derive the condition which we must maintain in our program:

$$
\begin{aligned}
& Pr[B|C^c] = \frac{1}{3} && \text{given} \\
\iff & Pr[B|A \cup B] = \frac{1}{3} && \text{substitution} \\
\iff & \frac{Pr[B \cap (A \cup B)]}{Pr[A \cup B]} = \frac{1}{3} && \text{def of conditional probability} \\
\iff & \frac{Pr[B]}{Pr[A \cup B]} = \frac{1}{3} && \text{A and B are mutually exclusive} \\
\iff & \frac{Pr[B]}{Pr[A] + Pr[B]} = \frac{1}{3} && \text{Rule of Addition for disjoint events} \\
\iff & \frac{3Pr[B]}{Pr[A] + Pr[B]} = 1 && \text{multiply by 3 on both sides} \\
\iff & 3Pr[B] = Pr[A] + Pr[B] && \text{multiply by Pr[A] + Pr[B] on both sides} \\
\iff & 2Pr[B] = Pr[A] && \text{subtract Pr[B] from both sides}
\end{aligned}
$$

Thus, the condition which must hold is that the probability of A is twice the probability of B.

In the program, A occurs if Bernoulli(1/2) + Bernoulli(1/2) = 1, and B occurs if Bernoulli(1/2) + Bernoulli(1/2) = 0.

The following are all the possible outcomes of x + y. Note that they are equally likely.

```
x | y  | x+y
--------------
0 | 0  |  0
0 | 1  |  1
1 | 0  |  1
1 | 1  |  2
```

It is clear that `x+y` is twice as likely to be 1 than it is to be 2.

Therefore, the function returns 0 twice as often as it returns 1.

The condition required by the problem holds.

Note that it doesn't matter what the P of failure is. If you change P of failure, the condition still holds.

e.

```
exp = 0
sum = 0
while(exp < 200) {
   if(Bernoulli(1/2) == 1) sum = sum + 2^exp
   exp = exp + 1
if(sum == 0) return 'FAIL'
```

```
n = sum % 3
if(n == 0) return 0
else return 1
```

Proof:
The program generates a random 200 bit integer, where each integer is equally likely to be generated. The loop iterates through the 200 bit-positions, and each time it picks 1 or 0 with equal probability (of $\frac{1}{2}$) for the bit. If the bit is 1, it adds the appropriate base 10 representation of that bit. The result `sum` is an integer from 0 to $2^{200} - 1$.

On of these equally likely numbers is 0. This occurs when all the bits are 0, which happens with a probability of a half, 200 times. Also the Bernoulli calls are independent. Therefore the function returns FAIL with a probability of $(1/2)^200$. That condition is satisfied.

You are left with $2^{200} - 1$ numbers. Note that $2^{200-1} = (2^2)^{100} - 1 \equiv (1)^{100} - 1 \equiv 0 \bmod 3$. Therefore the number of integers, $2^{200} - 1$, is divisible by 3. Also, since the integers from 1 to $2^{200} - 1$ are consecutive, exactly 1 out of every 3 integers will be divisible by 3. Therefore, there is a $\frac{1}{3}$ probability that a generated number (excluding 0) will be equivalent to 0 mod 3. Thus, there is a $\frac{2}{3}$ probability that the integer will not be equivalent to 0 mod 3. Therefore the program perfectly simulates Bernoulli(1/3).

f. 2^k is not divisible by 3, since 3 never occurs in its prime factorization. If there are 2^k equally likely outcomes, where $k \in \mathbb{N}, k \leq 200$, there is no way to split the outcomes evenly into three subsets.

A probability of $\frac{1}{3}$ is only attainable if there are 3 equally likely outcomes to choose from. We have proved that this is not the case using 2^k bernoulli calls.

DumDumDeeDum

Claim:
For all $T \in \mathbb{N}$, $T \geq 3$, the probability of k Dee's dancing is $\frac{1}{T-1}$.

Base case
For $(T = 3)$ We are given that after $T = 2$, there is 1 Dee and 1 Dum. There is a 50% chance that the Dee will replicate, and a 50% chance that the Dum will replicate.

Therefore there are two equally outcomes: 2 Dees and 1 Dum, and 1 Dee and 2 Dums. Therefore, the probability of there being 2 Dees is 1/2 and the probility of there being 1 Dee is 1/2. Then the probability of k Dee's dancing is 1/2, which also happens to be $\frac{1}{T-1} = \frac{1}{3-} = \frac{1}{2}$. The claim holds for the base case.

Inductive Hypothesis
Assume for some $q \in \mathbb{N}, q \geq 3$, that the probability of k Dee's dancing is $\frac{1}{q-1}$.

Inductive Step
Consider a situation with k Dee's in step $q + 1$. There are two ways in which this situation could have arisen:

`Way 1:`

```
       Step Q                Dees replicate               Step Q+1
Dees = k-1 , Dums = Q-(k-1)   -------->          Dees = k, Dums = Q-(k-1)

                                   OR

Way 2:
       Step Q                Dums replicate               Step Q+1
Dees = k , Dums = Q-k         -------->          Dees = k, Dums = Q-k+1
```

Thus, the event that there are k Dees in step $Q+1$ is the event that there are k Dees in step Q and a Dum duplicates, or the event that there are $k-1$ Dees in step Q and a Dee duplicates.

The probability that there are k Dees in step Q is $\frac{1}{Q-1}$ by the I.H. Call the number of Dees is k, the probability that a Dum duplicates is the number of Dums divided by the total number of dancing Dee/Dums, which is $\frac{Q-k}{Q}$. The probability of both these independent events happening is the product of their probabilities:

$$(\frac{1}{Q-1})(\frac{Q-k}{Q}) = \frac{Q-k}{Q(Q-1)}$$

The probability that there are $k-1$ Dees in step Q is $\frac{1}{Q-1}$ by the I.H. (unless $k-1 \leq 1$ but we'll get there later.) The probability that a Dee duplicates is the number of Dees divided by the number of dancing Dee/Dums, which is $\frac{k}{Q}$. The probability of both these independent events happening is the product of their probabilities:

$$(\frac{1}{Q-1})(\frac{k}{Q}) = \frac{k}{Q(Q-1)}$$

The sum of their probabilities is

$$\frac{k}{Q(Q-1)} + \frac{Q-k}{Q(Q-1)} = \frac{1}{Q}$$

Edge Case k=1:
We will show that the inductive step still works when k=1. In this case, the only way to get 1 Dee at step $Q+1$ is to have had 1 Dee at step Q and have Dum replicate. The probability of having 1 Dee at step Q is $\frac{1}{Q-1}$ by the IH, and the probability of selecting a Dum to replicate is $\frac{Q-1}{Q}$. Their product is $\frac{1}{Q}$.

Edge Case k=Q:
We will show that the inductive step still works when k=Q. In this case, the only way to get Q Dees at step $Q+1$ is to have had Q-1 Dees at step Q and have Dee replicate. The probability of having Q-1 Dees at step Q is $\frac{1}{Q-1}$ by the IH, and the probability of selecting a Dee to replicate is $\frac{Q-1}{Q}$. Their product is $\frac{1}{Q}$.

For the probabilities that I've multiplied, are the events independent? Yes. If you switch the order, the probabilities still hold. Ie. the event that you have k Dee's and the event that Dum replicates (where there are k Dees) has the same probability that Dum replicates (where there are k Dees)

and that you have k Dee's. This is counter-intuitive because the probability that a Dee or Dum replicates is in terms of k.

Therefore, the probability of k dees at step Q+1 is $\frac{1}{Q}$ and the claim holds for all $T \in \mathbb{N}, T \geq 3$ by induction.

Never Trust a Talking Cat

Let X be the random variable which maps a cat to the number of streaks of length $log_2 n + 1$ it has.

Let X_i be an indicator which is 1 if the i^{th} stripe up to the $i + log_2 n + 1^{th}$ stripe is of the same color, because this defines a streak. Note that i is an integer between 1 and $n - log_2 n$, because when i is $n - log_2 n$, the indicator checks the stripes from i to n. Any larger i will check the $n+1$th stripe which doesn't exist.

Therefore, we can say,

$$X = \sum_{i=1}^{n-log_2 n} X_i$$

Then, using linearity expectation with indicators,

$$\mathbb{E}(X) = \sum_{i=1}^{n-log_2 n} Pr[X_i = 1]$$

To find the closed form, we need to find $Pr[X_i = 1]$. Since there is a $\frac{1}{2}$ probability of a stripe being of a specific color, and the stripes are colored independently, there is a $(\frac{1}{2})^{log_2 n+1}$ probability of $log_2 n + 1$ stripes being the same color.

$$\begin{aligned} Pr[X_i = 1] =& (\frac{1}{2})^{log_2 n+1} \\ =& \frac{1}{2^{log_2 n+1}} \\ =& \frac{1}{2^{log_2 n} 2} \\ Pr[X_i = 1] =& \frac{1}{2n} \end{aligned}$$

Then we can simplify the summation using algebra:

$$\begin{aligned}
\mathbb{E}(X) &= \sum_{i=1}^{n-log_2 n} Pr[X_i = 1] \\
&= \sum_{i=1}^{n-log_2 n} \frac{1}{2n} \\
&= (n - log_2 n)\frac{1}{2n} \\
\mathbb{E}(X) &= \frac{1}{2} - \frac{log_2 n}{2n}
\end{aligned}$$

Therefore, the expectated number of streaks of length $log_2 n + 1$ where n is the number of stripes that the cat has, is

$$\mathbb{E}(X) = \frac{1}{2} - \frac{log_2 n}{2n}$$

Time for Tea with Turtles

We want to show that the average load across all n server turtles is at most $\frac{5}{12}$. We can show this by maximizing average load, and finding an upper bound to how much you can maximize it.

First, note that for an arbitrary but fixed number of tables m, we maximize average load by minimizing the number of servers n. In other words, we want to see what happens to the load as we try to have as few turtles as possible.

We are given that $Pr[L \leq \frac{1}{8}] \geq \frac{1}{5}$. To maximize load, we want as few turtles to have $L \leq \frac{1}{8}$. A minimum $\frac{1}{5}$ of these turtles will have the property $L \leq \frac{1}{8}$.

Then we can partition the tables based on whether or not their servers have this property or not. Gather $\frac{1}{5}$ of the tables in partition A, and $\frac{4}{5}$ of the tables in partition B. Given that $P[L \leq \frac{1}{8}] \geq \frac{1}{5}$, we can say we can say that tables in A tables have the servers with $L \leq \frac{1}{8}$. To maximize their load, say that those turtles have $L = \frac{1}{8}$. Likewise, we maximize the load of turtles who serve tables in B by saying that their $L = 1$. We examine the first partition to start. With tables in A, given that $L = \frac{1}{8}$ at max, we try to minimize the number of servers.

Let k be the number of servers serving tables in A.

$$\begin{aligned}
\frac{m}{5} &= k_{min} * (\text{number of tables each is serving}) \\
\frac{m}{5} &= k_{min} * 1000 * L_{max} \\
\frac{m}{5} &= k_{min} * 1000 * \frac{1}{8} \\
k_{min} &= \frac{m}{5000 L_{max}} = \frac{m}{625}
\end{aligned}$$

We see that we require at least $\frac{m}{625}$ servers, and if we decrease L, we add more servers. Therefore, $\frac{m}{625}$ is the minimum number of servers for this partition of tables.

Now we examine the second partition. With $\frac{4m}{5}$ tables, and a max load of 1,

$$k_{min} = \frac{4m}{5000L_{max}} = \frac{m}{1250}$$

If we have fewer servers, at least one will have $L > 1$, which is impossible. If we have decrease L, we add more servers. Therefore $\frac{m}{1250}$ is the minimum number of servers who serve tables in this partition.

We have logically found the situation which yields the maximum average load. Now we will show that it is at most $\frac{5}{12}$.

Note that in the below equations, L_a stands for the average load of the turtles serving tables in A , and L_b stands for the average load of the turtles serving tables in B.

$$\begin{aligned}
\text{Avg. Load} &= \frac{[(L_1) + (L_2) + ... + (L_n)]}{n} \\
&= \frac{[L_a \frac{m}{625} + L_b \frac{4m}{5000}]}{\frac{m}{625} + \frac{4m}{5000}} \\
&= \frac{[\frac{1}{8}\frac{m}{625} + 1\frac{4m}{5000}]}{\frac{5000m+2500m}{625*5000}} \\
&= \frac{\frac{m}{5000} + \frac{4m}{5000}}{5000m + 2500m} \\
&= \frac{\frac{5m}{5000} * 625 * 5000}{7500m} \\
&= \frac{3125m}{7500m} \\
\text{Avg. Load} &= \frac{5}{12}
\end{aligned}$$

We have shown that in the scenario where we maximize the load per turtle, the worst-case scenario for the average load is $\bar{L} = \frac{5}{12}$.

We have also shown that if we change the scenario in any way, we increase the number of servers, thereby decreasing average load.

Thus, we have shown that the maximum average load is at most $\frac{5}{12}$.

Off with Their Heads

(Part A)
The pseudo-code:

```
boolean containsAverage(int x[]) {
```

```
max = 0                              //holds the max value of x
n = x.length

for(i=0; i < n; i++)                 // O(n)
{
    if(x[i] > max) max = x[i]        // O(1)
}


hashTable = alloc_array(max+1)        // O(1)

//Initialize only n of the cells to 0.
//Note that we will only ever read from these cells.
for(i=0; i < n; i++)                 // O(n)
    hashTable[x[i]] = 0

//Get all the averages of elements in x.
for(i=0; i < n; i++)                 //O(n^2)
   for(j=0; j < n; j++)
     if(i != j) {                    //only do the following for 2 elements in the set
        average = (i+j)/2
        if (average - average.floor == 0) //check if the average is an integer
            hashTable[average] ++         //Increment the value in the array.
     }                                    // Note: If the average is in the set of numbers,
                                          // we are incrementing a non-garbage value.

for(i=0; i < n; i++)          //O(n)
    if(hashTable[x[i]] != 0) {      //it was initialized to 0, but may have been
      free(hashTable)               //disrupted in the n^2 process.
      return true
    }
}

free(hashTable)
return false

}
```

Proof:
Call the integer array x. The algorithm works like this:

1. Get the maximum integer in x. $O(n)$
2. Make an array 1 of size 1 larger than the maximum integer and call it "hashTable". $O(1)$
3. The "hash function" will be x, which takes an index and returns the corresponding value for the index in x. $O(1)$ read/write

4. We initialize the values for all the possible keys produced by the hash funciton to 0. $O(n)$ (since there are n values in x)

5. Compute all averages in a nested for-loop. If the average is an integer, it can potentially be in x, so increment the value in the hash table. $O(n^2)$

6. For each element in x, check if the value in the hashtable is no longer 0, as we had initialized it. If so, the average of 2 elements is that element. Return true. $O(n)$

7. If by the end of the process you haven't returned true, you must return false. $O(1)$

Clearly, the entire process is bounded by $O(n^2)$, due to step 5.

A few observations: There is no potential for collisions or invalid read/write in the hash table. Call m the maximum integer in x. The array is of size m+1, so it is valid for all indices from 0 to m. The "hash function" will only return natural numbers less than or equal to m.

What if `x[i] == x[j]` when $i \neq j$? Normally we would consider this a collision in the hash table, but we can ignore this because if x[i] or x[j] is the average of two elements, we should return true regardless of which one is the average.