SJSU Students
February 11,
2019

CS 255 HW 1

1. Suppose we have a sample space of n coin tosses. Give a construction of a set of $n+1$ events in this sample space which are n-wise independent. Prove your construction works.

Construct the first n events to be the event that the i-th coin toss is heads for i = 1 to n. Make the last n+1 event to be the event that the total number of heads is even.
The probability that any one of the coin toss is heads is 0.5.
The probability that the total number of heads is even is also 0.5 for n > 0.

For n = 1, Pr(Even) is 0.5. It is heads or tails, which is odd or even.
If the next coin flip is tails, the number of heads stays the same as Pr(Even) for k-1.
If the next coin flip is heads, the probability is the same as Pr(Odd) for k-1.
Adding these two events together is Pr(Tails) * Pr(Even) + Pr(Heads) * Pr(Odd) = 0.5

The probability of any n events in this set occurring together is 0.5^n which makes it n-wise independent. Each coin flip is independent, and the evenness of the total number of heads is independent of any combination of n-1 coin flips.

Event E_i: i the coin flip is heads
Event E: 1st, 2nd, ..., i - 1th, i + 1th, ..., nth coin flips are heads
Event F: 1st, 2nd, ..., i - 1th, i + 1th, ..., nth coin flips are heads, and the total number of heads are even.
P(F) = P(F| E) P(E) = 0.5 * 0.5 ^ (n - 1) = 0.5 ^ n.

2. An **injection** $f:[n]\rightarrow[m]$ (here [n] is the set $\{1,2,\ldots,n\}$) where $n\leq m$, is a function such that if $x\neq y$ then $f(x)\neq f(y)$. Let $F_{n,m}$ be the space of all such functions. Give pseudo-code for a randomized algorithm which generates elements of $F_{n,m}$ uniformly at random. So for example, the random permutation algorithms from class do this for the case $n=m$, I want you to figure out a way to do it for all $n\leq m$. Estimate the number of bits of randomness used by your algorithm and the run time of your algorithm in terms of n and m.

Injection Randomization for domain R of size n and Codomain S of size m where n <= m

```
for i in 1 to n:
  swap(S[i], S[Random(i,m)])
  R[i] = S[i]
```

Lemma: Prior to the ith iteration of the loop, for each possible (i-1)-permutation, the subarray A[1,i-1] contains this permutation with probability (n-i+1)!/n! (Slide 5 of Feb 4 Lecture)

At the beginning of the (n + 1) th iteration, we have that the subarray A[1... n] is a given n-permutation with probability (m - (n + 1) + 1)! / m! = (m - n)!/m!. To get an n-permutation from m objects, there are m!/(m - n)! ways. Thus, we can see that the algorithm produces a uniform random permutation.

The number of random bits used is lg(m-n)! - lgm! and has runtime O(n).

3. Consider the Hiring Problem from class. Candidate i had a $rank(i)$ which was a number from 1 to n. Each candidate had a distinct rank. Rather than using a rank suppose the candidates had a fitness number $fit(i)$ saying how good the candidate was. Suppose each candidate has a distinct fitness number between 1 and n, but that we only hire a candidate if they are 2 times better than the current best candidate. Calculate how this would affect the analysis of the expected number of candidates we hire.

Instead of hiring if rank(new) > rank(best), we hire if fit(new) >= 2* fit(best).

Since the problem looks to hire candidates that have at least 2x the fitness score of the currently best candidate, that means each candidate has about 1/(2i) chance of being hired. For each permutation of the order of candidates, if the first candidate has a score higher than the average score, no other candidates will be selected for hiring. This average case is O(log(log(n))).

At the worst case, the candidates arrive in an order where each candidate is exactly double the fitness of the previous one until no better candidates are available, then the rest of the candidates arrive in any order. In this case, the candidate with the fitness score that is double of the current hired one is always hired. Candidates with fitness scores of 1, 2, 4, 8, 16, and so on are hired up to n. The worst case is represented as log base 2 of n which is O(log n).

Coupon Collector Experiment

The purpose of this experiment is to determine whether the expected number of coupons scales with the equation b(ln(b)+O(1)) where n is the maximum number of coupons. In addition to checking whether the equation scales with the formula, we want to determine the value of the constant O(1) in this formula.

We suspect that the overall trend of the expected output of the coupon problem does in fact fit the equation b(ln(b)+O(1)). As for the value of O(1), we also suspect that this value is in fact the Euler-Mascheroni constant, which is a constant that appears commonly in harmonic series. The value of the constant is approximately 0.5772. To conduct our experiment and verify our hypothesis, we first write a program replicating the coupon collector problem. To summarize the problem, the coupon collector needs to draw a complete set of coupons in order to redeem a prize. The problem concerns with the expected number of draws the collector needs to perform in order to complete his set of coupons.

Python was used to replicate the problem. Below is the code used to simulate a run of the collector's problem:

```
def coupon(n):
 array = [False] * n
 iterations = 0
 while False in array:
   i = random.randrange(0,n)
   print(i)
   array[i] = True
   iterations += 1
 print("All coupons collected in", iterations, "iterations.")
```

For an argument n, an array is created to track the collected coupon. A loop is then created to draw a random number between 0 and n. If the index position value of the array holds the value false, then it is set to true. The counter used to track the number of iterations done is also incremented. The loop terminates once the array does not hold any more false values.

The values 1-1000 were passed in to the method. For each integer, the experiment was ran 10 times and the average and standard deviation of the run were calculated. The average values of the run for the given integer were then plotted on Figure 1 in blue. The values for standard deviation were also plotted in Figure 2. .

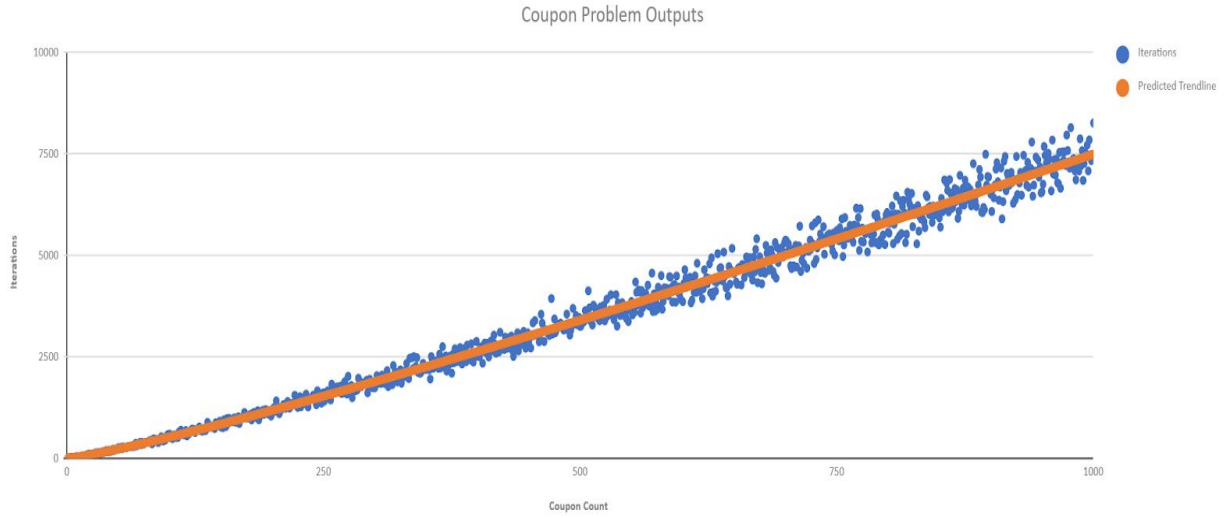

Figure 1. The plotted average outputs along with line of best fit

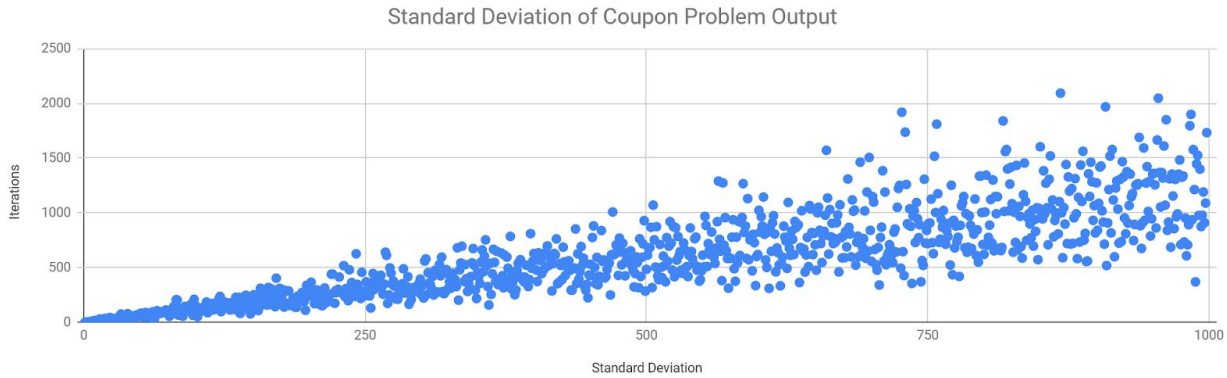

Figure 2. Standard deviation of the plotted average outputs of coupon problem

To determine the constant O(1) we used the sum of squared errors function and checked potential decimal values between 0 and 1 significant to the thousandths place. The data we gathered and the generated constant were then plugged into the coupon collector formula to calculate the sum of squared errors. Empirically, fitting the data to the curve using this method gave us the constant 0.552. Thus, the line of best fit for our simulated problem is $y = x(\ln(x) + 0.552)$. The line of best fit is plotted in Figure 1 as orange.

In conclusion, we determined that the formula $b(\ln(b)+O(1))$ does in fact scale with the coupon collector problem. Furthermore, we empirically determined that the constant O(1) is approximately 0.552. This value sits closely with our predicted value of 0.5772, which is the approximate value of the Euler-Mascheroni constant.