Algorithms Exercise Sheet: Probability Review and Coding

Instructions

In this assignment, you will gain experience thinking about probability theory. This will include some by-hand exercises as well as writing some code to define numerical experiments.

Basic Probability Exercises

1. **Basic Probability Rules:** Consider the problem of rolling a 6-sided die. Complete the following exercises:

   a. What is an appropriate sample space Ω for this problem?

   b. Given this sample space, write a set of probabilities for a *fair die*; i.e., the die is equally likely to come up on any side.

   c. Given this sample space, write a set of probabilities for a *biased die* which is an elongated cube: sides 2,3,4,5 all have the same probability; sides 1,6 have the same probability, but the probability of landing on side 1 is only 1/20.

   d. Compute Pr[rolling a 3] for both the biased and the fair die.

   e. Compute Pr[rolling an even number] for both the biased and the fair die.

   f. Compute Pr[rolling a number divisible by 3] for both the biased and the fair die.

2. **More interesting probability spaces:** Suppose you're playing a game where you have to flip a coin and also roll a 6-sided die. Use the same "fair" and "biased" definitions from Problem 1 for the die.

   a. Choose an appropriate sample space for this problem. Write out the sample space visually.

   b. In this part, assume that the coin flip and the die are *independent*. For a fair coin and die, what are the probabilities for each of the possible outcomes in the sample space?

   c. Again, assume the coin and die are independent. For a fair coin and Problem 1's biased die, what are the probabilities for each of the possible outcomes in the sample space?

   d. **CS 5080 Only:** Now, someone has done something funky to your coin and die. Maybe they weighted it? Maybe they magnetized it? Whatever happened, you do thousands of joint flip-roll experiments and determine that the probabilities are now given in Table 1. Are your

	1	2	3	4	5	6
Heads	0.08	0.08	0.08	0.14	0.08	0.1
Tails	0.08	0.08	0.08	0.02	0.08	0.1

Table 1: The probabilities of each flip/roll combination after some trickster got into your game cabinet.

   coin and die independent? If they are, you should be able to come up with the probabilities of each separate from the other. If they aren't, you should be able to show mathematically that there is no way to compute a probability of Head/Tail without knowing how the die roll came out (and vice versa).

3. **Expected Value:** All the following questions involve sequences of independent coin flips.

   a. Consider a game where you flip a fair coin 10 times in a row. You are paid \$1 for each Heads, and \$0 for each Tails. What is the expected payout of this game?

   Note: be careful with this. The "game" is 10 coin flips in a row. So it would technically be possible to define a sample space that encodes all possible sequences of 10 coin flips (this would have 2^{10} elements), then define your random variable on that sample space. Or, that leverages the fact that the coin flips are all independent from one another.

b. **CS 5080 only:** Now consider a different game: Now you get to flip the coin *up to* 10 times in a row, but there's a catch: you have to stop flipping when you get your first Tails. Thus, an outcome of this game is a sequence of k Head flips. If you get $k \geq 1$ Head flips, your payout is 2^k. I.e., \$2 for a single Head, \$4 for two Heads, \$8 for 3 Heads, and so on. What is the expected payout of this game?

c. **CS 5080 only:** Think again about the game that I described in part (b), but modify it so that the game didn't have a deadline. I.e., it doesn't stop automatically at 10 flips but you just keep flipping as long as you keep getting Heads. Would you pay a casino \$1000 to play this game? Justify your answer using rigorous mathematical reasoning.

Coding Exercise

4. **Linear Search Empirical Verification:** In lecture, we described 2 ways to compute probabilities of finding a search key K at any of the n indices of a random input array. In particular, the 2nd model had some more advanced methods of computation. Whenever you're working with probabilities, it can be very useful to write code to numerically verify that your calculations are correct. This exercise will walk you through this process for the lecture example.

The linear search problem:

Given array A of n elements and target value K, find the index of K in A or determine that K is not present. Simple pseudocode:

```
function search(A, n, K):
    for i from 0 to n-1:
        if A[i] == K:
            return i
    return -1
```

Throughout this assignment, we'll make the following assumption. There are M distinct array elements, all equally likely to be anywhere in the array. Let's write p_i to denote the probability that the earliest occurrence of the search key K is in index i of the array; write p_{-1} to denote the probability that the search key K is not in the array at all. Then we came up with the following probabilities:

- For $i \in \{0, \ldots, n-1\}$, it holds that $p_i = \left(1 - \frac{1}{M}\right)^i \frac{1}{M}$.
- Then, it holds that $p_{-1} = 1 - \sum_{i=0}^{n-1} p_i$.

In this exercise, you'll experimentally verify these formulas.

(a) Choose M. I would pick this number to be not too big, not too small; maybe something in the 5 to 20 range.

(b) Choose your alphabet. This should be M distinct symbols. For example, if $M = 5$, your alphabet could be {`A`,`B`,`C`,`D`,`E`}.

(c) *From your alphabet,* choose a search key. Make this the same throughout all of your experiments. In mine, I'm choosing `A`.

(d) Generate a dataset. You should pick several different values of n; say $n \in \{5, 10, 20, 50\}$. For each value of n, you should generate a large number of random arrays selected from your alphabet. For example, if $n = 5$ and your alphabet is a list called `alph`, you could use Python code like `numpy.random.choice(alph,5)` to generate a single random array of length 5. Overall, your dataset should have a large number of (different) random arrays for each value of n.

(e) Run every array in your dataset through the linear search function defined above. For each value of n, save the *count* of each distinct outcome. I plan to do this with a Python dictionary for simplicity: {`i:0 for i in range(-1,n)`}. For array length n, key i of the dictionary will store the *count* of length-n arrays for which the "search" function returned value i.

(f) From the results of the previous part, compute the *empirical frequencies*[1] of each outcome. For example, if there are 10,000 arrays with $n = 5$, you should compute frequencies for each of 6 possible outcomes: found in 0, found in 1,...,found in 5, or not found. The empirical frequency of an outcome is defined as the *count* of that outcome (computed in part (e)) divided by the number of total outcomes. In my example case, the number of total outcomes is 10,000 since there are 10,000 arrays for each n.

(g) Finally, generate plots which compare the theoretical probabilities (given above before part (a)) with the empirical frequencies computed in part (f).

Some considerations:

- When comparing small probabilities (i.e., probabilities that are very close to 0), it's often a good idea to use a *logarithmic* vertical axis. Pyplot achieves this with the command `plt.yscale('log')`.
- Your plots should be scatter plots (I recommend `matplotlib.pyplot.scatter`) or bar charts, not line plots.[2]
- How do you know that you've taken enough samples (i.e., enough random arrays)? There are strong scientific answers to this, but for our purposes you know you're taking enough samples when you can just barely tell the difference between the empirical frequencies and the theoretical probabilities. If there's a big difference between probabilities and frequencies, you either need to take more samples or your code has a bug. Or you derived the probabilities incorrectly.
- Depending on the size of your alphabet, it may take a very large number of samples for your frequencies to match your probabilities for the larger values of n; this is because with a small alphabet, you're extremely unlikely to ever find the search key late in an array, because you'll almost always find it early in the array.

[1] "Empirical frequency" is what we call a thing like a probability that we compute from data! If you take enough samples (and your random number generator has enough precision), you're guaranteed that the empirical frequencies of your experiments will eventually converge to the theoretical probabilities. This phenomenon is called the "law of large numbers."

[2] NEVER use a line plot if the horizontal axis is *categories* and not numerical. In this assignment, the "key not found" output will be a significant part of the results, but it's not numerically related to any of the other results so it should not be connected with a line.