Solutions for CS174 Homework 1

Solution 1. $\Pr[Y=1]=1/3, \Pr[Y=1|X=1]=2/3$, so $\Pr[Y=1|X=1]\neq\Pr[Y=1]$. Therefore X and Y are not independent.

$\mathrm{E}[XY]=1\cdot\Pr[X=1,Y=1]+0\cdot(\Pr[X=1,Y=0]+\Pr[X=0,Y=1]+\Pr[X=0,Y=0])=1/3$.

Solution 2. Divisibility by 2 and 3 are fully determined by the value of an integer mod 6. So let $n=6k+i$, where k and i are integers and $0\leq i\leq 5$. To compute the probabilities, we need to do a case analysis. First notice that the equality $\Pr[Y=1]=\Pr[Y=1|Z=1]$ is enough for independence. If this equality holds, then by complement it follows that $\Pr[Y=0]=\Pr[Y=0|Z=1]=1-\Pr[Y=1]$. You can also show from the definition of conditional probability that

$$\Pr[Y=1]=\Pr[Y=1|Z=1] \implies \Pr[Y=1]=\Pr[Y=1|Z=0]$$

that is, if a probability is unchanged when restricted to a subspace (in this case $Z=1$), then it is also unchanged when restricted to the complement of that subspace. In general if $\Pr[Y=1]=\Pr[Y=1|Z=1]$, we have equality between all the conditional probability conditions needed for independence.
Now back to the case analysis, if $i=0$, i.e. $n=6k$ we have

$$\Pr[Y=1]=\frac{3k}{6k}=\frac{1}{2} \quad \text{while} \quad \Pr[Y=1|Z=1]=\frac{k}{2k}=\frac{1}{2}$$

as long as $k\geq 1$, so $n=6k$ gives independence. Now for $i=1$:

$$\Pr[Y=1]=\frac{3k}{6k+1} \quad \text{while} \quad \Pr[Y=1|Z=1]=\frac{k}{2k}=\frac{1}{2}$$

which doesnt match, so $n=6k+1$ doesnt work (there is a sole exception at $k=0$ where some denominators vanish). Now for $i=2$

$$\Pr[Y=1]=\frac{3k+1}{6k+2}=\frac{1}{2} \quad \text{while} \quad \Pr[Y=1|Z=1]=\frac{k}{2k}=\frac{1}{2}$$

which matches. Surprisingly enough, $n=6k+2$ works as well. For $i=3$:

$$\Pr[Y=1]=\frac{3k+1}{6k+3} \quad \text{while} \quad \Pr[Y=1|Z=1]=\frac{k}{2k+1}$$

which fails to match and $n = 6k + 3$ is no good. For $i = 4$:

$$\Pr[Y = 1] = \frac{3k+2}{6k+4} = \frac{1}{2} \quad \text{while} \quad \Pr[Y = 1|Z = 1] = \frac{k}{2k+1}$$

which fails, and finally $i = 5$ gives:

$$\Pr[Y = 1] = \frac{3k+2}{6k+5} \quad \text{while} \quad \Pr[Y = 1|Z = 1] = \frac{k}{2k+1}$$

which fails again. So the only numbers for which Y and Z are independent are 1, 2 or $6k$ or $6k + 2$ for $k = 1, 2, \ldots$.

Solution 3. $\Pr[Y = 1|X = 1] = 0, \Pr[Y = 1] = 1/52$, so X and Y are not independent.

Solution 4. Consider pairs of coin tosses. We discard HH or TT, and if HT appears, count it as 1, otherwise when TH appears count it as 0. Because HT and TH occurs equally likely, it represents a fair coin.

Solution 5. Consider an array of elements $A[i] = i, 1 \leq i \leq n$. The simplest implementation of randomized bubblesort would be:

```
for i =1:n-1,
  for j=1:n-i,
    swap A[j] and A[j + 1] with probability 1/2
  end
end
```

Such an algorithm will generate random permutations but not with uniform distribution. For example the first element will have a higher probability to stay in the first half of the array than go to the second half of the array. To get a uniform distribution, we would need to do something that simulates the probabilities of bubble sort making swaps when it is sorting a random permutation as input.

Suppose we ran bubblesort on a random permutation, i.e. $A[i]$ is initially $A_{\pi(i)}$ for a random π. Let p_{ij} be the probability that bubblesort does a swap as a function of the algorithm indices i and j. Suppose first that $i = 1$ and $j = 1$. The algorithm is comparing two random values so $p_{11} = 1/2$. Now consider the general case where $i = 1$ and j is arbitrary. Through its bubbling process, bubblesort guarantees that $A[j]$ contains the maximum of $A[1], \ldots, A[j]$. No swap will happen with $A[j + 1]$ if the value of $A[j + 1]$ (which is random because we havent seen it before) is bigger than $A[1], \ldots, A[j]$. In other words, no swap happens iff $A[j + 1]$ is the maximum of $A[1], \ldots, A[j + 1]$. From our analysis of random permutations we know that that probability that a given value occurs last (has largest value in this case) in a random permutation of $j + 1$ values is $1/(j + 1)$. That is the probability of not swapping $A[j]$ and $A[j + 1]$. So the probability that there is a swap when $i = 1$ is $p_{1j} = j/(j + 1)$.

After the first iteration i, bubblesort has bubbled the maximum element into $A[n]$ the other elements are in $A[1], \ldots, A[n - 1]$, and in the same order as in the original data. In

other words, they are in random order. So the second iteration ($i = 2$) of the main loops swaps elements with the same probability $p_{2j} = j/(j+1)$. The same argument applies for larger values of i, so we conclude that p_{ij} in fact only depends on j, so we write it as p_j and it has the form:

$$p_j = j/(j+1)$$

So to generate uniformly distributed random permutations, we can use the following algorithm: Start with an array of elements $A[i] = i, 1 \leq i \leq n$.

```
for i =1:n-1,
  for j=1:n-i,
    swap A[j] and A[j+1] with probability j/(j+1)
  end
end
```

Does this really generate a random permutation? We can prove it by determining the probability that an element initially in position $A[k]$ gets bubbled all the way to the end. That happens iff $A[k]$ is not swapped with $A[k-1]$, but it is swapped with all later elements. The probability of this is

$$(1 - p_{k-1})p_k p_{k+1} \cdots p_n = \frac{1}{k}\frac{k}{(k+1)}\frac{(k+1)}{(k+2)} \cdots \frac{(n-1)}{n} = \frac{1}{n}$$

which is a "telescoping" product, where all but the end terms cancel. Thus every element has probability $1/n$ of being swapped to the end on the first iteration $i = 1$. By similar reasoning, on the $i = 2$ iteration, every element has probability $1/(n-1)$ of being swapped to the end. These are exactly the probabilities needed to generate a random permutation. In fact, you should be able to see a close connection now between the bubblesort random generator and the "selection sort" random generator we did in class. Both have probability of $1/k$ of moving any element in a subarray of k elements to the end.

You may be concerned that this generator is not "consistent". It may not swap two elements (e.g. $A[1]$ and $A[2]$) on several passes, but then swap them on a later iteration, so that it is not exactly simulating bubblesort's swaps on some real input. It doesnt matter. Because it always places a random element of the current subarray in the last position, it is guaranteed to generate random permutations.

The running time is easily seen to be $O(n^2)$ in all cases.