CSCI 239—Discrete Structures of Computer Science
Lab 11—Counting and Probabilities

This lab will explore ways to use Haskell to solve counting and probability problems.

Objectives:

- to gain more experience writing recursive Haskell functions
- to solve counting problems using a computer
- to solve an interesting probability problem

Preliminaries

Copy the `/usr/people/classes/CS239/labs/Lab11_CountingAndProbabilities` folder into your *CS239* folder. It contains a literate Haskell file that you will use at the end of this lab. Put any other Haskell files for this lab into this folder.

Part 1: Haskell methods for counting

In this part you will implement Haskell functions for arrangements and selections, i.e. the *p*(*n*, *r*) and *c*(*n*, *r*) functions.

A generalized factorial function
The p and c functions we want to implement are both defined using the factorial function. To make things more efficient, however, it would be helpful to have a function that computes the product of an arbitrary range of integers from *m* to *n*. We can use this function to compute the factorial of *n* by setting *m* to 1. We need to be careful, however, because we want zero to be factored in as one as well. The following recurrence relation will compute what we want:

$$fact(m,n) = \begin{cases} fact(m,-1) & n = 0 \\ 1 & m > n \\ n \cdot fact(m, n-1) & otherwise \end{cases}$$

Implement this recurrence relation as a curried Haskell function. (That is, it takes two parameters, not a single parameter that is a 2-tuple.) Be careful with the cases; you can use pattern matching for the first case, but you have to use the vertical bar selection for the next two; you may use the vertical bar selection for all three if you prefer.

Use the `:type` command to determine the type of the function. Test your function on enough small cases (ranges from negative to positive) to be confident it is working correctly.

A function to compute arrangements (permutations)
The function *p*(*n*, *r*) is defined as:

$$p(n,r) = \frac{n!}{(n-r)!}$$

Using a little algebra, we can simplify this to:

$$\begin{aligned} p(n,r) &= (n-r+1)(n-r+2)\dots(n-1)n \\ &= fact(n-r+1, n) \end{aligned}$$

Implement this last formula as a Haskell function, get its type, and test it. (Note that there is no more recursion here if you use the *fact* function.) We use the simplified formula to reduce the number of multiplications required by a factor of $2(n-r)$.

Two functions to compute selections
The function *c*(*n*, *r*) is defined as:

$$c(n,r) = \binom{n}{r} = \frac{n!}{r!\,(n-r)!}$$

Using simple algebra, we can simplify this to two equivalent formulas:

$$\begin{aligned} c(n,r) &= \frac{(r+1)(r+2)\ldots(n-1)n}{(n-r)!} &&= \frac{fact(r+1,n)}{fact(1,n-r)} \\ &= \frac{(n-r+1)(n-r+2)\ldots(n-1)n}{r!} &&= \frac{fact(n-r+1),n)}{fact(1,r)} \end{aligned}$$

We again want to use a simplified formula to reduce the number of multiplications, but which one is better? Whichever has fewer multiplications, of course! And which is that? It depends. If *r* > *n*/2 then the first formula will have fewer multiplications; otherwise, the second formula has fewer (or the same if *r* = *n*/2). Implement a curried Haskell function *c1* using these two formulas, choosing the formula that is more efficient depending on the value of *r* relative to *n*.

There is another, recursive definition for *c*(*n*, *r*), based on our definition of Pascal's triangle:

$$c(n,r) = \begin{cases} 1 & r = 0, r = n \\ c(n-1,r) + c(n-1,r-1) & 0 < r < n \\ 0 & otherwise \end{cases}$$

Implement a curried Haskell function *c2* using this definition. Notice that there are only additions, and no multiplications or divisions in this definition, so it should be faster in principle.

Test both functions for all values of *n* from 0 up to 8 and *r* from 0 to *n*. Make a table of your results. If you aren't getting the same values for both functions, at least one of them is not implemented correctly. Keep working on it until you get the same value for both functions.

Now try computing *c1* 20 10 and *c2* 20 10. Which is faster?

Use the *p* function and the faster of the *c* functions you developed to compute the answers to Exercises 10.4.2.b, 10.4.3.c, 10.5.2.d, 10.5.3.f, and 10.5.5 from the zyBooks textbook.

Use the *p* function and the faster of the *c* functions you developed to compute the answers to Exercises 12.1.5.c, 12.2.4.c, and 12.3.5 (compute the two sides of the final inequality).

Part 2: The Monty Hall problem

(Note: Monty Hall died on September 30, 2017, age 96. We do this problem to honor his life.)

In the 1960s and 70s (and maybe into the 80s), Monty Hall was the host of the TV game show *Let's Make a Deal*. (You can see reruns on the *Buzzr* and *Game Show* networks.) One of the games on the show captured the interest of statisticians everywhere. The game works like this: there are three giant doors on the stage labeled 1, 2, and 3; behind two of the doors are goats (UGH!!!); behind the other door is a ...

NEW
CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAR!!!!!!!!!!!!!!!!!

(Back in the 60s, a new car was pretty much the most fabulous prize anyone could imagine.)

Anyway, back to the description. Monty would offer the contestant a choice of one of the three doors. Then, since he knew everything, Monty would have one of the other two doors opened, always revealing a goat. (UGH!!!) At this point, one of the doors still hides a goat. (UGH!!!) And behind the other door hides the ....

NEW CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAR!!!!!!!!!!!!!!!!!

Now comes the fun part: Monty tells the contestant she may either keep the door she chose in the first place, or she may switch to the other unopened door. Should she stick to her first choice, switch, or doesn't it matter? Your task is to compute the probability of winning when she doesn't switch and when she does to answer this question.

Before you go on, what does your intuition tell you? After Monty reveals a door hiding a goat (UGH!!!), two doors remain. Does switching matter, given that two doors remain, one with the fabulous prize and the other with the not-so-fabulous prize?

First, consider the parameters of the sample space: which door hides the ...

NEW
CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAR!!!!!!!!!!!!!!!!!

(thereby determining which doors hide the goats (UGH!!!)), which door the contestant chooses, which door Monty reveals (does this one affect the outcome?), and whether the contestant switches doors or not. How big is this sample space of four or three parameters? What determines the outcome? Note that if the contestant chooses the right door to start with, she wins only if she doesn't switch. If she chooses the wrong door at first, when does she win? In both cases, when does she lose? (Answering these questions correctly should confirm whether the door Monty reveals is relevant to the sample space. Assume that the goats (UGH!!!) are indistinguishable.)

Consider the sample space parameters in turn, and give your answers to each question below:

a) What are the ways to configure the “prizes” behind the three doors? There will be one

NEW
CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAR!!!!!!!!!!!!!!!!,

and two goats (UGH!!!), one behind each door. How many ways are there? What is the mathematical formula that explains why you got those answers?

b) What are the contestant’s choices? How many choices are there? Again, what is the mathematical formula that explains why you got those answers?
c) What are Monty’s choices when he reveals what’s behind one of the doors? Remember, he never reveals the door the contestant chose, and he never reveals the door with the

NEW
CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAR!!!!!!!!!!!!!!!!

Does he always have a choice? If not, does he ever have a choice? If so, when does he have a choice? In that case, does his choice affect the outcome? If not, you don’t have to include Monty’s choice in the sample space.
Do not go beyond this point until you’ve verified with the TA or lab instructor that you have the correct answer here.

d) The contestant always has the choice of whether to switch doors or not. As we have noted switching can affect the outcome positively or negatively depending on the state of the other choices.

Now identify the sample space. An outcome must include a) the placement of the prizes, b) the door the contestant chooses, c) possibly Monty's choice, and d) whether the contestant switches. Define an outcome to be a tuple of these three or four components, and the sample space to be the set of all valid tuples. (Which tuples, if any are not valid?)

What is the size of your sample space? Justify your answer.

Assume a uniform distribution for all outcomes, that is, that they are all equally likely. This may not be true, particularly if we determine it's better to switch or not to switch, but it doesn't matter for our calculations. (If you are a big fan of the show, and realize that they tend to put the best prize behind some doors more than others, knowing that could affect how you would play the game, but it's not part of the problem we're trying to solve. As far as I know, they tended to distribute the prizes pretty randomly.)

Define *SW* to be the event that the contestant switches doors. What is |SW|? Define *WIN* to be the event that the contestant wins the

NEW
CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAR!!!!!!!!!!!!!!!!!!

What is |WIN|? That is, how many of the outcomes in the sample space are winning outcomes? What is the probability *p*(WIN) of winning overall (not considering whether the contestant switches)?

What is |WIN ∩ SW|? That is, how many of the winning outcomes are ones where the contestant switched doors? What is the probability *p*(WIN | SW) that the contestant wins if she switches doors?

Are these two probabilities the same? That is, are WIN and SW independent events? Is it better to switch or not to switch? Does this match your intuition?

People who don't understand statistics (including some people with PhDs) continue to argue vehemently against the statistically correct solution. See the Wikipedia web page for the Monty Hall problem for more information.

Part 3: A Monte Carlo/Monty Hall Simulation

You can use the supplied Haskell file to help you write a simulation of the problem. Write a function *makeTuple* that construct an outcome as a tuple, using parameters that pertain to each outcome component. Now write a function *makeTupleList* that combines equal length lists of outcome components into a list of outcome tuples.

Next, write a function *isWin* that identifies whether an outcome tuple is a win or not, another function *isSwitch* that identifies whether the contestant switched doors in an outcome, and a third function *isWinWithSwitch* that identifies whether an outcome is a win with a door switch. Finally, write three functions that respectively count the number of winning outcomes (*countWins*), the number of door switches (*countSwitches*), and the number of winning outcomes with door switches (*countWinsWithSwitches*) in a list of outcomes.

Now use the supplied functions (*selectDoors* twice and *selectSwitches*) and the *makeTupleList* function you wrote to generate a list of a few thousand random outcomes and bind that list to an identifier. Apply the last three functions to that list. The ratio of total wins to total outcomes should be close to the value you calculated for *p*(WIN), and the ratio of wins with switches to total outcomes with switches should be close to the value you calculated for *p*(WIN | SW). If not, consult with the instructor or TA to figure out what's going on.

NOTE:

> The seed parameter for the two supplied random-list-building functions can be any integer. Every time you use the same seed, you get the same sequence of elements, even if the lists are different lengths. (You just get fewer with a smaller length.) Change the seed to get a different sample. Try your Banner ID (or the last four digits). Just use a different seed from others you're working with.