Monty Hall Problem

In this lesson, we will be discussing an interesting problem called the Monty Hall Problem. It can be solved using the stochastic workflow technology.

WE'LL COVER THE FOLLOWING

Application of the Stochastic Workflow Technology

You are probably familiar with the famous "*Monty Hall*" problem, but if not, here it is:

Monty Hall

Problem

You're on a game show hosted by Monty Hall, the handsome Canadian fellow

pictured above. Before you, there are three closed doors. Behind a uniformly randomly selected door, there is a new car; behind the other two, there is nothing. You get to choose a door, and you get what is behind it as a prize: either a car or nothing. You randomly choose a door, again by some uniform process.

Monty — who knows where the car is — now always opens a door that meets two criteria: it does not have the car behind it, and it is not the door you chose.

To clarify: if you chose the door with the car, Monty chooses one of the remaining two doors by a uniform random choice. If you chose a door without the car, Monty only has one door he can open, and he opens that one. Monty gives you the opportunity to switch your choice to the other still-closed door. Assuming you wish to maximize your probability of winning the car, should you switch doors or stay with your original choice?

> We've tried to be very precise in our description of the game for the purposes of our analysis. In the real game as played on television, there were irrelevant details such as the "prizes" behind the other two doors were goats or other bizarre, undesirable items, and so on. But there were also germane differences between the real game and our model above; for example, in the real game Monty would sometimes offer choices like "do you want to switch your door, or forget about the doors entirely and take the prize that is in this box?" and it is not clear by what process Monty decided to offer those choices. In this simplified version of the game, we've removed all human agency from Monty; for our purposes, Monty is just a machine that is following an algorithm that involves generating random outcomes.

If you don't already know the solution, work it out. The answer is below.

Solution

You are two-to-one more likely to win the car if you switch than if you stay. But don't take our word for it. Let's solve the problem with computers, not by thinking!

Plainly the key to the problem is what is the distribution of Monty's choice?

Monty chooses a random door but is observed to not pick a door with a car or the door which you picked. We can represent that as a two-parameter likelihood function:

```
IDiscreteDistribution<int> doors = SDU.Distribution(1, 3);

IDiscreteDistribution<int> Monty(int car, int you) =>
    from m in doors
    where m != car
    where m != you
    select m;
```

There's no logical difficulty in adding more parameters to a likelihood function; think of the parameters as a tuple if that makes you feel better.

Now we can answer the question. Here's the probability distribution of winning if you do not switch:

```
var noSwitch1 =
  from car in doors
  from you in doors
  from monty in Monty(car, you)
  select car == you ? "Win" : "Lose";
Console.WriteLine(noSwitch1.ShowWeights());
```

And the output is:

```
Win:1
Lose:2
```

As predicted by thinking, you are twice as likely to lose if you do not switch. Computers for the win!

> **Exercise:** Wait a minute, we never even used the value of range variable monty in the query. How is it possible that adding a `from` clause to the query changes its outcome when the sampled value is not even used?

Show Hint

Okay smart person, if you thought that one was easy, take a look at this one. We have our likelihood function `Monty()` which is just a query comprehension, and our `noSwitch1` which is also just a query comprehension. We can make the program a little bit shorter by combining them together in an obvious way:

```
var noSwitch2 =
  from car in doors
  from you in doors
  from monty in doors
  where monty != car
  where monty != you
  select car == you ? “Win” : “Lose”;
```

And if we print out the weights of that one we get:

```
Win:1
Lose:1
```

We would have thought this program fragment to be logically the same as before, but this gives weights of $1:1$ when we know the correct answer is $1:2$.

Where did we go wrong?

Show Hint

Implementation

Let’s have a look at the code for this lesson:

Program.cs

Bernoulli.cs

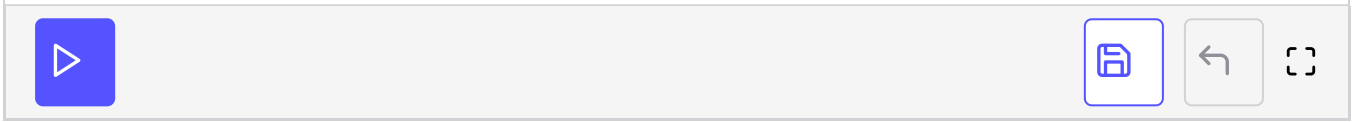
BetterRandom.cs

Bonus01.cs

Combined.cs

```
using System;
using System.Collections.Generic;

// Let's show how we can create Where out of SelectMany on sequences.
namespace Weird
{
    static class MyLinq
    {
        // Standard implementation of Select:
        public static IEnumerable<R> Select<A, R>(
            this IEnumerable<A> items,
            Func<A, R> projection)
        {
            foreach (A item in items)
                yield return projection(item);
        }

        // Standard implementation of SelectMany:
        public static IEnumerable<R> SelectMany<A, B, R>(
            this IEnumerable<A> items,
            Func<A, IEnumerable<B>> selection,
            Func<A, B, R> projection)
        {
            foreach (A a in items)
                foreach (B b in selection(a))
                    yield return projection(a, b);
        }

        public static IEnumerable<T> Single<T>(T t)
        {
            yield return t;
        }

        public static IEnumerable<T> Zero<T>()
        {
            yield break;
        }

        // Non-standard Where:
```

```
        public static IEnumerable<T> Where<T>(
                this IEnumerable<T> items,
                Func<T, bool> predicate) =>
            from a in items
            from b in predicate(a) ? Single(a) : Zero<T>()
            select b;
    }
}
namespace Probability
{
    // No using System.Linq.
    using Weird;
    static class Episode17
    {
        public static void DoIt()
        {
            Console.WriteLine("Episode 17");
            Console.WriteLine("Custom Where using only SelectMany");
            Console.WriteLine("aBcDe".Where(char.IsLower).CommaSeparated());
            Console.WriteLine("Delayed throw in enumerator block");
            var foo = Foo(null);
            Console.WriteLine("No throw yet!");
            try
            {
                foreach (int x in foo)
                    Console.WriteLine(x);
            }
            catch
            {
                Console.WriteLine("Now we throw.");
            }
        }

        static IEnumerable<int> Foo(string bar)
        {
            if (bar == null)
                throw new ArgumentNullException();
            yield return bar.Length;
        }
    }
}
```



In the next lesson, we will be implementing the zero value.