Probability Distribution Monad with Round Numbers

In this lesson, we will learn an example of probability distribution monads by using rounded numbers.

WE'LL COVER THE FOLLOWING

In the previous lesson, we made a correct, efficient implementation of `SelectMany` to bind a likelihood function and projection onto a prior, and gave a simple example.

Example with Rounded Numbers

We deliberately chose "weird" numbers for all the weights; let's do that same example again but with more "nice round number" weights:

```
var prior = new List<Height>() { Tall, Medium, Short}.ToWeighted(60, 30, 1
0);
[...]

IDiscreteDistribution<Severity> likelihood(Height h)
{
  switch(h)
  {
    case Tall: return severity.ToWeighted(45, 55, 0);
    case Medium: return severity.ToWeighted(0, 70, 30);
    default: return severity.ToWeighted(0, 0, 1);
  }
}

[... projection as before ...]
Console.WriteLine(prior.SelectMany(likelihood, projection).ShowWeights());
```

This produces the output:

```
DoubleDose:270000
NormalDose:540000
HalfDose:190000
```

which is correct, but you'll notice how multiplying the weights during the `SelectMany` made for some unnecessarily large weights. If we then did another `SelectMany` on this thing, they'd get even larger, and we'd be getting into **integer overflow** territory.

Integer overflow is always possible in the system we've developed so far in this course, and we are deliberately glossing over this serious problem. A better implementation would either use doubles for weights, which have a much larger range, or arbitrary-precision integers, or arbitrary-precision rationals. We are using integers to keep it simple, but as with many aspects of the code in this course, that would become problematic in a realistic implementation.

One thing we can do to tame this slightly is to reduce all the weights when possible, in this case, we could divide each of them by 10000 and have the same distribution, so let's do that. And just to make sure, we are going to mitigate the problem in multiple places:

1. In `SelectMany` we could be taking the *least common multiple* (LCM) instead of the full product of the weights.
2. In the `WeightedInteger` factory, we could be dividing out all the weights by their *greatest common divisor* (GCD).

Let's have a look at the simplified implementation of *Euclid's Algorithm*:

```
public static int GCD(int a, int b) => b == 0 ? a : GCD(b, a % b);
```

We define the GCD of two non-negative integers a and b as:

- If both are zero, then the GCD is zero
- otherwise, if exactly one is zero, then the non-zero one is the GCD
- otherwise, the largest integer that divides both.

> **Exercise:** Prove that this recursive implementation meets the above conditions.

The problem we face though is that we have many weights and we wish to find the GCD of all of them. Fortunately, we can simply do an aggregation:

```
public static int GCD(this IEnumerable<int> numbers) => numbers.Aggregate(
GCD);
```

Similarly, we can compute the LCM if we know the GCD:

```
public static int LCM(int a, int b) => a * b / GCD(a, b);

public static int LCM(this IEnumerable<int> numbers) => numbers.Aggregate(
1, LCM);
```

And now we can modify our `WeightedInteger` factory:

```
public static IDiscreteDistribution<int> Distribution(IEnumerable<int> wei
ghts)
{
  List<int> w = weights.ToList();
  int gcd = weights.GCD();
  for (int i = 0; i < w.Count; i += 1)
    w[i] /= gcd;

[......]
```

And our `SelectMany`:

```
int lcm = prior.Support()
  .Select(a => likelihood(a).TotalWeight())
  .LCM();

[...and then use the lcm in the query...]
```

See the code for all the details. If we apply all these changes then our results look much better...

```
DoubleDose:27
NormalDose:54
```

```
HalfDose:19
```

... and we are at least a little less likely to get into an integer overflow situation.

> Of course, we can do the same thing to the `Bernoulli` class, and normalize its weights as well.

Implementation

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

Program.cs

Bernoulli.cs

BetterRandom.cs

Combined.cs

Distribution.cs

Episode15.cs

Extensions.cs

IDiscreteDistribution.cs

IDistribution.cs

Projected.cs

Pseudorandom.cs

Singleton.cs

StandardCont.cs

StandardDiscrete.cs

WeightedInteger.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace Probability
{
    using SCU = StandardContinuousUniform;
```

```
    public sealed class Bernoulli : IDiscreteDistribution<int>
    {
        public static IDiscreteDistribution<int> Distribution(int zero, int one)

        {
            if (zero < 0 || one < 0 || zero == 0 && one == 0)
                throw new ArgumentException();
            if (zero == 0)
                return Singleton<int>.Distribution(1);
            if (one == 0)
                return Singleton<int>.Distribution(0);
            int gcd = Extensions.GCD(zero, one);
            return new Bernoulli(zero / gcd, one / gcd);
        }
        public int Zero { get; }
        public int One { get; }
        private Bernoulli(int zero, int one)
        {
            this.Zero = zero;
            this.One = one;
        }
        public int Sample() => (SCU.Distribution.Sample() <= ((double)Zero) / (Zero + One))
        public IEnumerable<int> Support() => Enumerable.Range(0, 2);
        public int Weight(int x) => x == 0 ? Zero : x == 1 ? One : 0;
        public override string ToString() => $"Bernoulli[{this.Zero}, {this.One}]";
    }
}
```

We can use the gear we've created so far to solve problems in *Bayesian inference*; in the next lesson, we'll see how.