Discrete Distribution on C# Objects

In this lesson, we will learn how to apply discrete distribution on C# objects, and write a wrapper class for it.

WE'LL COVER THE FOLLOWING

In the previous lesson, we showed how we could implement more of the standard, straightforward probability distributions, like the Bernoulli distribution.

Uniformly Choosing Between Objects

Typically when we are working on a line-of-business program, we're working on objects that are meaningful in that line of business, and not necessarily on integers. Just to pick an example, suppose we want to uniformly choose between a cat, a dog, and a goldfish:

```
using SDU = StandardDiscreteUniform;
...
var animals = new List<Animal>()
  { new Cat(), new Dog(), new Goldfish() };
```

Traditionally, as we noted a while back, we'd write some mechanism-heavy code like:

```
var choice = animals[random.Next(0, animals.Length)];
```

Now, we sincerely hope you're not satisfied with:

```
var choice = animals[SDU.Distribution(0, animals.Length-1).Sample()];
```

The problem here is that we've failed to push the "mechanism" code into a helper class, so let's do that:

```
// This is wrong!
public sealed class DiscreteUniform<T> : IDiscreteDistribution<T>
{
  private readonly SDU sdu;
  private readonly List<T> support;
  public static DiscreteUniform<T> Distribution(IEnumerable<T> items) => n
ew DiscreteUniform<T>(items);

  private DiscreteUniform(IEnumerable<T> items)
  {
    this.support = items.ToList();
    this.sdu = SDU.Distribution(0, support.Count - 1);
  }

  public IEnumerable<T> Support() => support;
  public T Sample() => support[sdu.Sample()];
  public int Weight(T t) => this.support.Contains(t) ? 1 : 0;
  public override string ToString() =>
    $"DiscreteUniform[{string.Join(",", support)}]";
}
```

> **Exercise:** That code is wrong in a number of ways; what are they? Give it some thought and then click on **Show Hint**.

Show Hint

Fixing Problems

We'll fix all these problems in our final implementation at the bottom of this lesson.

Suppose we've managed to fix the problems identified above. We now create a helper method:

```
public static IDiscreteDistribution<T> ToUniform<T>(this IEnumerable<T> it
ems) =>
  DiscreteUniform<T>.Distribution(items);
```

And now our proposed code is much nicer:

```
var animals = new List<Animal>()
  { new Cat(), new Dog(), new Goldfish() };

var choice = animals.ToUniform().Sample();
```

That looks much nicer, but... it still does not look perfect. The call site is better with a readable fluent extension method but we have built yet another *single-purpose* wrapper class. This wrapper class is only good for making a uniform distribution of an arbitrary sequence. Suppose we have a non-uniform distribution of integers from 0 to n — say, a binomial distribution — and we want to map a list of animals to that distribution; am I going to have to write a "binomial distribution on sequence" class that duplicates almost all the logic of the wrapper class above?

> **Exercise:** Implement the binomial distribution; this is the extension of the Bernoulli distribution where, again, we have weights for a possibly-unfair coin, but we also have a parameter n which is the number of coins flips to make. The result is the total number of heads. Sampling is straightforward, but can you see how to compute the weights?

More Sophisticated Wrapper Class

This seems like an opportunity for a more sophisticated wrapper class, parameterized in the underlying distribution on integers. Let's tweak the class we just wrote:

```
public sealed class IntegerDistributionWrapper<T> : IDiscreteDistribution<
T>
{
  private readonly IDiscreteDistribution<int> d;
  private readonly List<T> items;
```

```
  public static IntegerDistributionWrapper<T> Distribution(
    IDiscreteDistribution<int> d, IEnumerable<T> items)

  {
    // some code here
  }

  private IntegerDistributionWrapper(
    IDiscreteDistribution<int> d, IEnumerable<T> items)
  {
     // some code here
  }

  public T Sample() => items[d.Sample()];

// and so on
```

This is an improvement. We're getting better, but we want **more generality for less cost**. What are we doing here really?

1. *Sampling* an *integer* from a distribution.
2. *Projecting* that integer onto a value taken from a list.

Why does the projection have to be "value taken from a list"? And for that matter, why does the value sampled from the underlying distribution have to be an integer, to begin with? We can write almost the same code, but make it far more general:

```
public sealed class Projected<A, R> : IDiscreteDistribution<R>
{
  private readonly IDiscreteDistribution<A> underlying;
  private readonly Func<A, R> projection;
  private readonly Dictionary<R, int> weights;

  public static IDiscreteDistribution<R> Distribution(IDiscreteDistributio
n<A> underlying, Func<A, R> projection)
  {
    var result = new Projected<A, R>(underlying, projection);
    if (result.Support().Count() == 1)
      return Singleton<R>.Distribution(result.Support().First());

    return result;
  }

  private Projected(IDiscreteDistribution<A> underlying, Func<A, R> projec
```

```
tion)
  {
    this.underlying = underlying;
    this.projection = projection;
    this.weights = underlying.Support().
      GroupBy(
        projection,
        a => underlying.Weight(a)).
      ToDictionary(g => g.Key, g => g.Sum());
  }

  public R Sample() => projection(underlying.Sample());
  public IEnumerable<R> Support() => this.weights.Keys;
  public int Weight(R r) => this.weights.GetValueOrDefault(r, 0);
}
```

Study this implementation and make sure it makes sense to you; the crux of the whole thing is in the constructor, where we compute the weights of the resulting distribution.

You've probably noticed that we have fixed all the problems identified above:

1. We return a singleton when possible.
2. We don't need to worry about the distribution being empty because presumably, the underlying distribution is not empty.
3. The support set is an immutable deduplicated keyset.
4. We compute the weights upfront.

> **Exercise:** Many optimizations could be made here for the common case where the number of "colliding" elements in the support are small or zero, to reduce the amount of extra storage. Try implementing some of them; see if you get a win.

Rewriting Helper Method

Now that we have a projection wrapper, we can delete our discrete uniform wrapper and integer wrapper classes entirely. That means we have to rewrite our helper method:

```
public static IDiscreteDistribution<T> ToUniform<T>(this IEnumerable<T> it
```

```
ems)
{

  var list = items.ToList();
  return Projected<int, T>.Distribution(
    SDU.Distribution(0, list.Count - 1), i => list[i]);
}
```

But again, this seems like I'm missing some generality.

Hmmm...let's try a further modification.

Maybe we should also write an extension method on distributions that makes a projected distribution. It's an easy helper method after all:

```
public static IDiscreteDistribution<R> MakeProjection<A, R>(
    this IDiscreteDistribution<A> d,
    Func<A, R> projection) =>
  Projected<A, R>.Distribution(d, projection);
```

Wait a minute; this looks very familiar. What code does this remind you of?

Show Hint

And now this helper can be rewritten into a call to `Select`:

```
public static IDiscreteDistribution<T> ToUniform<T>(this IEnumerable<T> it
ems)
{
  var list = items.ToList();
  return SDU.Distribution(0, list.Count - 1)
    .Select(i => list[i]);
}
```

This means we have a `Select` method with the right signature, which means we can use our beloved query comprehension syntax. The last line could be

```
return
  from i in SDU.Distribution(0, list.Count - 1)
  select list[i];
```

And now we know why a few lessons back we said that we were not going to

make `IDistribution<T>` an extension of `IEnumerable<T>`; doing so just causes confusion between the `Select` operation on distributions and the `Select` operation on sequences.

> If you've been following along you would have noticed that this implementation takes a list, then turns it into a list, and then turns it into a list again, and then turns it into a dictionary. This seems *"not cheap"*. And indeed it is not, but remember, this is pedagogical code, not industrial code.

The idea here is to sketch out what could be done, not exactly how we'd do it. When we think about performance in this course, we are going to be concentrating on things like eliminating arbitrarily expensive loops and not on pragmatic but mundane practicalities like eliminating sources of collection pressure.

Finally, let's just make sure that everything is working as expected:

```
var cat = new Cat();
var dog = new Dog();
var fish = new Goldfish();
var animals = new List<Animal>() { cat, dog, dog, fish };
Console.WriteLine(animals.ToUniform().Histogram());
Console.WriteLine(animals.ToUniform().ShowWeights());
```

Sure enough, we get the correct distribution:

```
Cat|******************
Dog|****************************************
Goldfish|******************
Cat:1
Dog:2
Goldfish:1
```

Interesting! We started by trying to make a uniform distribution and ended up with a correct non-uniformly-weighted distribution. Can we do a better job of building such a distribution without having to make an array that has duplicates? Let's explore this in later lessons.

Implementation

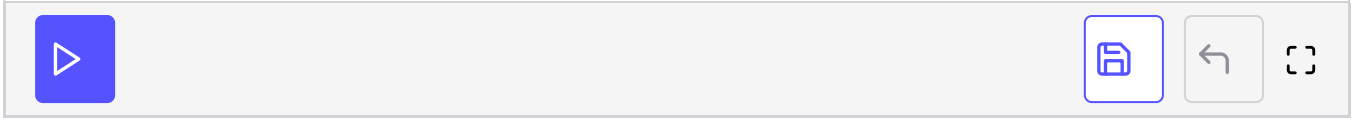
The code for this lesson is as follows:

Program.cs
Bernoulli.cs
BetterRandom.cs
Distribution.cs
Episode06.cs
Extensions.cs
IDiscreteDistribution.cs
IDistribution.cs
Projected.cs
Pseudorandom.cs
Singleton.cs
StandardCont.cs
StandardDiscrete.cs

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

abstract class Animal { }
sealed class Cat : Animal { }
sealed class Dog : Animal { }
sealed class Goldfish : Animal { }

namespace Probability
{
    static class Episode06
    {
        public static void DoIt()
        {
            var cat = new Cat();
            var dog = new Dog();
            var fish = new Goldfish();
            var animals = new List<Animal>() { cat, dog, dog, fish };
            Console.WriteLine(animals.ToUniform().Histogram());
            Console.WriteLine(animals.ToUniform().ShowWeights());
        }
    }
}
```



In this lesson, we've seen how to represent a "fair die roll" distribution and a Bernoulli distribution where we give the "odds" that either zero or one happens as a ratio of integers.

In the next lesson, we'll extend those concepts to any number of weighted outcomes, and discuss what data structure we can use to efficiently sample that distribution.