Alias Method

In this lesson we will be implementing the Alias method.

WE'LL COVER THE FOLLOWING

In the previous lesson, we sketched out the "*alias method*", which enables us to implement sampling from a weighted discrete distribution in constant time.

Implementing the Alias Method

The implementation of the Alias Method is slightly tricky, but we'll go through it carefully.

The idea is:

- If we have n weights then we're going to make n distributions.
- Every distribution is either a singleton that returns a particular number, or we use a projection on a Bernoulli to choose between two numbers.
- To sample, we uniformly choose from our n distributions, and then we sample from that distribution, so each call to `Sample` on the weighted integer distribution does exactly two samples of other, simpler distributions.

How are we going to implement it?

> The key insight is weights which are lower than the average weight "*borrow*" from weights which are higher than average.

Call the sum of all weights s. We start by multiplying each weight by n, which gives us $s \times n$ *weight points* to allocate; each of the n individual distributions will have a total weight of s points.

```
private readonly IDistribution<int>[] distributions;
private WeightedInteger(IEnumerable<int> weights)
{
  this.weights = weights.ToList();
  int s = this.weights.Sum();
  int n = this.weights.Count;
  this.distributions = new IDistribution<int>[n];
```

All right, we are going to keep track of two sets of "weight points":

1. elements where the unallocated "weight points" are lower than s, and
2. elements where it is higher than s. If we've got a weight that is exactly equal to the average weight then we can just make a singleton.

```
var lows = new Dictionary<int, int>();
var highs = new Dictionary<int, int>();
for(int i = 0; i < n; i += 1)
{
  int w = this.weights[i] * n;

  if (w == s)
    this.distributions[i] = Singleton<int>.Distribution(i);

  else if (w < s)
    lows.Add(i, w);

  else
    highs.Add(i, w);
}
```

If every element's weight was average, then we're done; the dictionaries are empty. (And we should not have created a weighted integer distribution in the first place!)

If not, then some of them must have been above average and some of them must have been below average.

What we're going to do is choose (based on no criteria whatsoever) one "*low*" and one "*high*", and we're going to transfer points from the high to the low. We'll start by grabbing the relevant key-value pairs and removing them from their respective dictionaries.

```
while(lows.Any())
{
  var low = lows.First();
  lows.Remove(low.Key);
  var high = highs.First();
  highs.Remove(high.Key);
}
```

The high has more than s points, and the low has less than s points; even if the low has zero points, the most we will transfer out of the high is s, so the high will still have some points when we're done. The low will be full up, and we can make a distribution for it:

```
int lowNeeds = s - low.Value;
this.distributions[low.Key] = Bernoulli.Distribution(low.Value, lowNeeds).
Select(x => x == 0 ? low.Key : high.Key);
```

Even if the low column value is zero, we're fine; `Bernoulli.Distribution` will return a `Singleton`. That's exactly what we want; there is zero chance of getting the low column number.

If the low is not zero-weighted, then we want the low-to-high odds to be in the ratio of the low points to the stolen high points, such that their sum is s; remember, we have $n \times s$ points to distribute, and each row must get s of them.

We just filled in the distribution for the "*low*" element. However, the "*high*" element still has points, remember. There are three possible situations:

1. the high element has exactly s points remaining, in which case it can become a singleton distribution
2. the high element has more than s points, so we can put it back in the high dictionary
3. the high element has one or more points, but fewer than s points, so it goes in the low dictionary

```
int newHigh = high.Value - lowNeeds;

if (newHigh == s)
  this.distributions[high.Key] = Singleton<int>.Distribution(high.Key);

else if (newHigh < s)
  lows[high.Key] = newHigh;

else
  highs[high.Key] = newHigh;
```

And we’re done.

Every time through the loop we remove one element from both dictionaries, and then we add back either zero or one element, so our net progress is either one or two distributions per iteration. Every time through the loop we account for either s or $2s$ points and they are removed from their dictionaries, and we maintain the invariant that the low dictionary is all the elements with fewer than s points, and the high dictionary is all those with more.

Therefore we will never be in a situation where there are lows left but no highs, or vice versa; they’ll run out at the same time.

Time and Space Complexity

We’ve spent $O(n)$ time (and space) computing the distributions; we can now spend $O(1)$ time sampling. First, we uniformly choose a distribution, and then we sample from it, so we have a grand total of two samples, and both are cheap.

```
public int Sample()
{
  int i = SDU.Distribution(0, weights.Count - 1).Sample();
  return distributions[i].Sample();
}
```

> We have just committed the sin we mentioned a few lessons ago, of sampling from an array by emphasizing the mechanism in the code

rather than building a distribution. Of course, we could have made a

“meta distribution” in the constructor with `distributions.ToUniform()` and then this code would be the perfect:

```
public int Sample() => this.meta.Sample().Sample();
```

What do you think? is it better, in this case, to emphasize the mechanism, since we’re in mechanism code, or do the double-sample?

We’ll see in a later lesson another elegant way to represent this double-sampling operation, so stay tuned.

Let’s try an example with zeros:

```
WeightedInteger.Distribution(10, 0, 0, 11, 5).Histogram()
```

Sure enough, the output will be:

```
0|************************************
3|****************************************
4|******************
```

The alias method is easy to implement and cheap to execute.

Remember, the whole point of this thing is to stop messing around with `System.Random` when you need randomness. We can now take any set of integer weights and produce a random distribution that conforms to those weights; this is useful in games, simulations, and so on.

> **Exercise:** We mentioned that we wanted to do all weights in integers rather than doubles. It is certainly possible to make a weighted integer distribution where the weights are doubles; doing so is left as an exercise. Give it some thought and then click on `Show Hint`.

Show Hint

Implementation

Let's have a look at the modifications made in our code up till now. Notice changes in the file `WeightedInteger.cs`

Program.cs

Bernoulli.cs

BetterRandom.cs

Distribution.cs

Episode09.cs

Extensions.cs

IDiscreteDistribution.cs

IDistribution.cs

Projected.cs

Pseudorandom.cs

Singleton.cs

StandardCont.cs

StandardDiscrete.cs

WeightedInteger.cs

```
using System;
namespace Probability
{
    static class Episode09
    {
        public static void DoIt()
        {
            Console.WriteLine(WeightedInteger.Distribution(10, 0, 0, 11, 5).Histogram());
        }
    }
}
```

There are (at least) two ways to apply "*conditional*" probabilistic reasoning; in the next lesson, we'll explore both of them.