Non-uniform Discrete Distribution (Continued)

In this lesson, we will continue modifying our non-uniform discrete distribution and also learn about rejection sampling.

WE'LL COVER THE FOLLOWING

In the previous lesson, we sketched out an $O(log\ n)$ algorithm for sampling from a weighted distribution of integers by implementing a discrete variation on the "inverse transform" method where the "inverse" step involves a binary search.

Modification of Non-Uniform Discrete Distribution

Can we do better than $O(logn)$? Yes — but we'll need an entirely different method. We will sketch out two methods, and implement one this time, and the other in the next lesson.

Again, let's suppose we have some weights, say 10, 11, 5. We have three possible results, and the highest weight is 11. Let's construct a 3 by 11 rectangle that looks like our ideal histogram; we'll put dashes to indicate the "spaces":

```
0|**********-
1|***********
2|*****------
```

Here's an algorithm for sampling from this distribution:

1. Uniformly choose a random row and column in the rectangle; it's easy to choose a number from 0 to 2 for the row and a number from 0 to 10 for the column.
2. If that point is a `*`, then the sample is the generated row number.
3. If the point is a `-`, try again.

Basically, we're throwing darts at the rectangle, and the likelihood of hitting a valid point on a particular row is proportional to the probability of that row.

In case that's not clear, let's work out the probabilities. To throw our dart, first we'll pick a row, uniformly, and then we'll pick a point in that row, uniformly. We have a $\frac{1}{3} \times \frac{10}{11} = \frac{10}{33}$ chance of hitting a star from row 0, an $\frac{1}{3} \times \frac{11}{11} = \frac{11}{33}$ chance of hitting a star from row 1, a $\frac{1}{3} \times \frac{5}{11} = \frac{5}{33}$ chance of hitting a star from row 2, and that leaves a $\frac{7}{33}$ chance of going again. We will eventually pick a `*`, and the values generated will conform to the desired distribution.

Let's implement it. We'll throw away our previous attempt and start over.

```
private readonly List<IDistribution<int>> distributions;
private WeightedInteger(List<int> weights)
{
  this.weights = weights;
  this.distributions =
    new List<IDistribution<int>>(weights.Count);
  int max = weights.Max();

  foreach(int w in weights)
    distributions.Add(Bernoulli.Distribution(w, max - w));
}
```

All right, we have three distributions; in each, a zero is a success and a one is a failure. In our example of weights $10, 11, 5$, the first distribution is "10 to 1 odds of success", the second is "always success", and the third is "5 to 6 odds of success". And now we sample in a loop until we succeed. We uniformly choose a distribution and then sample from that distribution until we get success.

```
public int Sample()
{
  var rows = SDU.Distribution(0, weights.Count - 1);
  while (true)
  {
    int row = rows.Sample();
    if (distributions[row].Sample() == 0)
      return row;
  }
}
```

We do two samples per loop iteration; how many iterations are we likely to do? In our example, we'll get a result on the first iteration $\frac{26}{33}$ of the time, because we have 26 hits out of 33 possibilities. That's 79% of the time. We get a result after one or two iterations 95% of the time. The vast majority of the time we are going to get a result in just a handful of iterations.

But now let's think again about pathological cases. Remember our distribution from last time that had 1000 weights: $1, 1, 1, 1, \ldots, 1, 1001$. Consider what that histogram looks like.

```
  0|*------...---  (1 star, 1000 dashes)
  1|*------...---  (1 star, 1000 dashes)
           ...
           ...
           ...
998|*------...---  (1 star, 1000 dashes)
999|*******...***  (1001 stars, 0 dashes)
```

Our first example has 26 stars and 6 dashes. In our pathological example, there will be 2000 stars and 999000 dashes, so the probability of exiting the loop on any particular iteration is about one in 500. We are typically going to loop hundreds of times in this scenario! This is far worse than our $O(log n)$ option in pathological cases.

Rejection Sampling

This algorithm is called "*rejection sampling*" (because we "reject" the samples that do not hit a `*`) and it works very well if all the weights are close to the maximum weight, but it works extremely poorly if there is a small number of high-weight outputs and a large number of low-weight outputs.

Fortunately, there is a way to fix this problem. We are going to reject *rejection sampling* and move on to our third and final technique.

Let's re-draw our original histogram, but I'm going to make two changes. First, instead of stars, we are going to fill in the number sampled, and make it a 33 by 3 rectangle, and triple the size of every row.

```
0|000000000000000000000000000000---
1|111111111111111111111111111111111
2|222222222222222------------------
```

This is logically no different; we could “throw a dart”, and the number that we hit is the sample; if we hit a dash, we go again. But we still have the problem that 21 out of 99 times we’re going to hit a dash.

Our goal is to get rid of all the dashes, but we are going to start by trying to get 7 dashes in each row. There are 21 dashes available, three rows, so that’s seven in each row.

To achieve that, we are going to first pick an “excessive” row (too many numbers, too few dashes) and “deficient row” (too few numbers, too many dashes) and move some of the numbers from the excessive row to the deficient row, such that the deficient row now has exactly seven dashes. For example, we’ll move eleven of the 0s into the 2^{nd} row, and swap eleven of the 2^{nd} row’s dashes into the 0^{th} row.

```
0|0000000000000000000--------------
1|111111111111111111111111111111111
2|22222222222222200000000000-------
```

We’ve achieved our goal for the 2^{nd} row. Now we do the same thing again. We are going to move seven of the 1s into the 0^{th} row:

```
0|00000000000000000001111111-------
1|11111111111111111111111111-------
2|22222222222222200000000000-------
```

We’ve achieved our goal. And now we can get rid of the dashes without changing the distribution:

```
0|00000000000000000001111111
1|11111111111111111111111111
2|22222222222222200000000000
```

Now we can throw darts and never get any rejections!

The result is a graph where we can again, pick a column, and then from that column sample which results we want; there are only ever one or two possibilities in a column, so each column can be represented by a *Bernoulli* or *Singleton distribution*. Therefore we can sample from this distribution by doing two samples: a uniform integer to pick the row, and the second one from the distribution associated with that row.

> You might wonder why we chose to move eleven 0s and then seven 1s. We didn't have to! I also could have moved eleven 1s into the 2^{nd} row, and then four 0s into the 1 row. Doesn't matter; either would work. As we'll see in the next lesson, as long as you make progress towards a solution, we'll find a solution.

This algorithm is called the **alias method** because some portion of each row is *"aliased"* to another row. But for now, let's have a look at the changes we have made in this lesson.

Implementation

In this lesson, we update the `WeightedInteger.cs` file. The rest of the code remains the same as the previous lesson.

- Program.cs
- Bernoulli.cs
- BetterRandom.cs
- Distribution.cs
- Episode07.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 Episode07
  {
    public static void DoIt()
    {
      Console.WriteLine(WeightedInteger.Distribution(10, 11, 5).Histogram());
    }
  }
}
```

In the next lesson, we will implement the alias method.