Lexicon of Notation

$\mathcal{S}$	Sample of population generated by algorithm
$n = \|\mathcal{S}\|$	Sample size desired (**not gauranteed by all algorithms**)
$\mathcal{A}$	A population to sample from **with random access**, think $\mathcal{A}$rray
$\mathcal{P}$	A population to sample from **without random access**
$m = \|\mathcal{P}\|, m = \|\mathcal{A}\|$	Size of population (**not always known for $\mathcal{P}$**)
$a[i], a_i, a_j \ldots$	An element of $\mathcal{A}$ or $\mathcal{P}$
i, j	Loop indices
`Get(&var,`$\mathcal{P}$`,loc)`	retrieval from a sequential access collection
`Put(&var,`$\mathcal{S}$`,loc)`	store into a sequential access collection

- The `&` of `Get` and `Put` is an "address of" operator borrowed from C, C++
- The `loc` argument of `Get` and `Put` is technically unneeded, think "get next element" and a "push" or "append" on a simple linked list.

Algorithm 6.5.1

Fisher-Yates Shuffling for Arrays

(You've probably seen this before. . .)

Algorithm 6.5.1

- The intuitive way to generate a random shuffle of $\mathcal{A}$ is:
  - draw the first element a_0 at random from $\mathcal{A}$
  - draw the second element a_1 at random from $A - \{a_0\}$
  - draw the third a_2 at random from $\mathcal{A} - \{a_0, a_1\}$, etc.
- An *in place* algorithm:

Algorithm 6.5.1

```
for(i = 0; i < m - 1; i++){
    j = Equilikely(i, m - 1);
    hold = a[j];
    a[j] = a[i]; /* swap a[i] and a[j]*/
    a[i] = hold;
}
```

The algorithm assumes $\mathcal{A}$ is stored in array a

- Algorithm 6.5.1 is an excellent example of the elegance of simplicity.

Algorithm 6.5.3

Use Fisher-Yates for sampling
without replacement into array `x[]`.

Example 6.5.3

- Suppose the population and sample are stored as arrays $a[0], a[1], \ldots, a[m-1]$ and $x[0], x[1], \ldots, x[n-1]$ respectively
- Algorithm 6.5.3 is equivalent to

Example 6.5.3

```
for(i = 0; i < n; i++){
  j = Equilikely(i, m - 1);
  x[i] = a[j];
  a[j] = a[i];
  a[i] = x[i];
}
```

- Algorithm 6.5.3 is a simple extension of Algorithm 6.5.1

Algorithm 6.5.4

Sampling without replacement with **unknown population size**.

Population $\mathcal{P}$ **without** random access.

Algorithm 6.5.4

```
i = 0; j = 0;
while( more data in 𝒫){
    Get(&z, 𝒫, j); /* sequential access */
    j++;
    if (Bernoulli(p)){
        Put(z, 𝒮, i);
        i++;
    }
}
```

- Order is preserved
- Each element of $\mathcal{P}$ is selected, independently, with probability p
- Algorithm 6.5.4 does not make use of either m or n explicitly
  - m may be unknown
  - Sample size n is a *Binomial*(m, p) random variate

Algorithm 6.5.5

Sampling without replacement with
known population size $|\mathcal{P}|$ and
a fixed sample size.

Population $\mathcal{P}$ **without** random access.

Offered without proof… not that hard to work out independently.

Algorithm 6.5.5

Algorithm 6.5.5

```
i = 0; j = 0;
while(i < n){
    Get(&z, 𝒫, j); /* sequential access */
    p = (n - i) / (m - j);
    j++;
    if (Bernoulli(p)){
        Put(z, 𝒮, i);
        i++;
    }
}
```

- $m = |\mathcal{P}|$ is known. Order is preserved.
- The key is that a_j is selected with a probability $(n - i)/(m - j)$
- Number of possible samples is $m!/(m - n)!n!$
- All samples are equally likely to be generated

Algorithm 6.5.6: Reservoir Sampling

It seems that when sampling from a sequential stream ($Get(\&z, \mathcal{P}, j)$), we need to:

- Know the size of the population ($|\mathcal{P}|$) (alg 6.5.6), or
- accept a variable sized sample (alg 6.5.4)

Algorithm 6.5.6: Reservoir Sampling

It seems that when sampling from a sequential stream ($Get(\&z, \mathcal{P}, j)$), we need to:

- Know the size of the population ($|\mathcal{P}|$) (alg 6.5.6), or
- accept a variable sized sample (alg 6.5.4)

Not so!

- ***Reservoir Sampling* techniques sample without replacement**,
- for a **fixed sample size**.
- **Population $\mathcal{P}$ without** random access and **unknown size**.

Algorithm 6.5.6

- Sequential random sampling without replacement, unknown m

Algorithm 6.5.6

```
for (i = 0; i < n; i++){
    Get(&z, 𝒫, i); /* sequential access */
    Put(z, 𝒮, i);
}
j = n;
while (more data in 𝒫){
    Get(&z, 𝒫, j); /* sequential access */
    j++;
    p = n / j;
    if (Bernoulli(p)){
        i = Equilikely(0, n -1);
        Put(z, 𝒮, i);
    }
}
```