Fisher–Yates shuffle

The **Fisher–Yates shuffle** (named after Ronald Fisher and Frank Yates), also known as the **Knuth shuffle** (after Donald Knuth), is an algorithm for generating a random permutation of a finite set—in plain terms, for randomly shuffling the set. A variant of the Fisher–Yates shuffle, known as **Sattolo's algorithm**, may be used to generate random cycles of length *n* instead. The Fisher–Yates shuffle is unbiased, so that every permutation is equally likely. The modern version of the algorithm is also rather efficient, requiring only time proportional to the number of items being shuffled and no additional storage space.

Fisher–Yates shuffling is similar to randomly picking numbered tickets (combinatorics: distinguishable objects) out of a hat without replacement until there are none left.

Fisher and Yates' original method

The Fisher–Yates shuffle, in its original form, was described in 1938 by Ronald A. Fisher and Frank Yates in their book *Statistical tables for biological, agricultural and medical research*.[1] Their description of the algorithm used pencil and paper; a table of random numbers provided the randomness. The basic method given for generating a random permutation of the numbers 1 through *N* goes as follows:

1. Write down the numbers from 1 through *N*.
2. Pick a random number *k* between one and the number of unstruck numbers remaining (inclusive).
3. Counting from the low end, strike out the *k*th number not yet struck out, and write it down elsewhere.
4. Repeat from step 2 until all the numbers have been struck out.
5. The sequence of numbers written down in step 3 is now a random permutation of the original numbers.

Provided that the random numbers picked in step 2 above are truly random and unbiased, so will the resulting permutation be. Fisher and Yates took care to describe how to obtain such random numbers in any desired range from the supplied tables in a manner which avoids any bias. They also suggested the possibility of using a simpler method — picking random numbers from one to *N* and discarding any duplicates—to generate the first half of the permutation, and only applying the more complex algorithm to the remaining half, where picking a duplicate number would otherwise become frustratingly common.

The modern algorithm

The modern version of the Fisher–Yates shuffle, designed for computer use, was introduced by Richard Durstenfeld in 1964 and popularized by Donald E. Knuth in *The Art of Computer Programming* as "Algorithm P". Neither Durstenfeld nor Knuth, in the first edition of his book, acknowledged the work of Fisher and Yates; they may not have been aware of it. Subsequent editions of *The Art of Computer Programming* mention Fisher and Yates' contribution.

The algorithm described by Durstenfeld differs from that given by Fisher and Yates in a small but significant way. Whereas a naive computer implementation of Fisher and Yates' method would spend needless time counting the remaining numbers in step 3 above, Durstenfeld's solution is to move the "struck" numbers to the end of the list by swapping them with the last unstruck number at each iteration. This reduces the algorithm's time complexity to $O(n)$, compared to $O(n^2)$ for the naive implementation. This change gives the following algorithm (for a zero-based array).

```
To shuffle an array a of n elements (indices 0..n-1):
  for i from n − 1 downto 1 do
       j ← random integer with 0 ≤ j ≤ i
       exchange a[j] and a[i]
```

If random number generator can return random integer $p \le j < q$ for specified parameters p, q then following version could be used:

```
To shuffle an array a of n elements (indices 0..n-1):
  for i from 0 to n − 1 do
       j ← random integer with i ≤ j < n
       exchange a[j] and a[i]
```

The "inside-out" algorithm

The Fisher–Yates shuffle, as implemented by Durstenfeld, is an *in-place shuffle*. That is, given a preinitialized array, it shuffles the elements of the array in place, rather than producing a shuffled copy of the array. This can be an advantage if the array to be shuffled is large.

To simultaneously initialize and shuffle an array, a bit more efficiency can be attained by doing an "inside-out" version of the shuffle. In this version, one successively places element number i into a random position among the first i positions in the array, after moving the element previously occupying that position to position i. In case the random position happens to be number i, this "move" (to the same place) involves an uninitialised value, but that does not matter, as the value is then immediately overwritten. No separate initialization is needed, and no exchange is performed. In the common case where *source* is defined by some simple function, such as the integers from 0 to $n − 1$, *source* can simply be replaced with the function since *source* is never altered during execution.

```
To initialize an array a of n elements to a randomly shuffled copy of source, both 0-based:
  for i from 0 to n − 1 do
      j ← random integer with 0 ≤ j ≤ i
      if j ≠ i
          a[i] ← a[j]
      a[j] ← source[i]
```

The inside-out shuffle can be seen to be correct by induction. Assuming a perfect random number generator, every one of the $n!$ different sequences of random numbers that could be obtained from the calls of *random* will produce a different permutation of the values, so all of these are obtained exactly once. The condition that checks if $j \ne i$ may be omitted in languages that have no problems accessing uninitialized array values, and for which assigning is cheaper than comparing.

Another advantage of this technique is that the algorithm can be modified so that even when we do not know "n", the number of elements in *source*, we can still generate a uniformly distributed random permutation of the *source* data. Below the array a is built iteratively starting from empty, and a.length represents the current number of elements seen.

```
To initialize an empty array a to a randomly shuffled copy of source whose length is not known:
  while source.moreDataAvailable
      j ← random integer with 0 ≤ j ≤ a.length
      if j = a.length
          a.append(source.next)
      else
          a.append(a[j])
          a[j] ← source.next
```

Examples

Pencil-and-paper method

As an example, we'll permute the numbers from 1 to 8 using Fisher and Yates' original method. We'll start by writing the numbers out on a piece of scratch paper:

Range	Roll	Scratch	Result
		1 2 3 4 5 6 7 8	

Now we roll a random number *k* from 1 to 8—let's make it 3—and strike out the *k*th (i.e. third) number (3, of course) on the scratch pad and write it down as the result:

Range	Roll	Scratch	Result
1–8	3	1 2 ~~3~~ 4 5 6 7 8	3

Now we pick a second random number, this time from 1 to 7: it turns out to be 4. Now we strike out the fourth number *not yet struck* off the scratch pad—that's number 5—and add it to the result:

Range	Roll	Scratch	Result
1–7	4	1 2 ~~3~~ 4 ~~5~~ 6 7 8	3 5

Now we pick the next random number from 1 to 6, and then from 1 to 5, and so on, always repeating the strike-out process as above:

Range	Roll	Scratch	Result
1–6	5	1 2 ~~3~~ 4 ~~5~~ 6 ~~7~~ 8	3 5 7
1–5	3	1 2 ~~3~~ 4 ~~5~~ 6 ~~7~~ 8	3 5 7 4
1–4	4	1 2 ~~3~~ 4 ~~5~~ 6 ~~7~~ ~~8~~	3 5 7 4 8
1–3	1	~~1~~ 2 ~~3~~ 4 ~~5~~ 6 ~~7~~ ~~8~~	3 5 7 4 8 1
1–2	2	~~1~~ 2 ~~3~~ 4 ~~5~~ 6 ~~7~~ ~~8~~	3 5 7 4 8 1 6
		~~1~~ ~~2~~ ~~3~~ 4 ~~5~~ 6 ~~7~~ ~~8~~	3 5 7 4 8 1 6 2

Modern method

We'll now do the same thing using Durstenfeld's version of the algorithm: this time, instead of striking out the chosen numbers and copying them elsewhere, we'll swap them with the last number not yet chosen. We'll start by writing out the numbers from 1 to 8 as before:

Range	Roll	Scratch	Result
		1 2 3 4 5 6 7 8	

For our first roll, we roll a random number from 1 to 8: this time it's 6, so we swap the 6th and 8th numbers in the list:

Range	Roll	Scratch	Result
1–8	6	1 2 3 4 5 **8** 7	**6**

The next random number we roll from 1 to 7, and turns out to be 2. Thus, we swap the 2nd and 7th numbers and move on:

Range	Roll	Scratch	Result
1–7	2	1 **7** 3 4 5 8	**2** 6

The next random number we roll is from 1 to 6, and just happens to be 6, which means we leave the 6th number in the list (which, after the swap above, is now number 8) in place and just move to the next step. Again, we proceed the same way until the permutation is complete:

Range	Roll	Scratch	Result
1–6	6	1 7 3 4 5	**8** 2 6
1–5	1	**5** 7 3 4	**1** 8 2 6
1–4	3	5 7 **4**	**3** 1 8 2 6
1–3	3	5 7	**4** 3 1 8 2 6
1–2	1	**7**	**5** 4 3 1 8 2 6

At this point there's nothing more that can be done, so the resulting permutation is 7 5 4 3 1 8 2 6.

Variants

Sattolo's algorithm

A very similar algorithm was published in 1986 by Sandra Sattolo for generating uniformly distributed cycles of (maximal) length n. The only difference between Durstenfeld's and Sattolo's algorithms is that in the latter, in step 2 above, the random number j is chosen from the range between 1 and $i–1$ (rather than between 1 and i) inclusive. This simple change modifies the algorithm so that the resulting permutation always consists of a single cycle.

In fact, as described below, it's quite easy to *accidentally* implement Sattolo's algorithm when the ordinary Fisher–Yates shuffle is intended. This will bias the results by causing the permutations to be picked from the smaller set of $(n–1)!$ cycles of length N, instead of from the full set of all $n!$ possible permutations.

The fact that Sattolo's algorithm always produces a cycle of length n can be shown by induction. Assume by induction that after the initial iteration of the loop, the remaining iterations permute the first $n - 1$ elements according to a cycle of length $n - 1$ (those remaining iterations are just Sattolo's algorithm applied to those first $n - 1$ elements). This means that tracing the initial element to its new position p, then the element originally at position p to its new position, and so forth, one only gets back to the initial position after having visited all other positions. Suppose the initial iteration swapped the final element with the one at (non-final) position k, and that the subsequent permutation of first $n - 1$ elements then moved it to position l; we compare the permutation π of all n elements with that remaining permutation σ of the first $n - 1$ elements. Tracing successive positions as just mentioned, there is no difference between π and σ until arriving at position k. But then, under π the element originally at position k is moved to the final position rather than to position l, and the element originally at the final position is moved to position l. From there on, the sequence of positions for π again follows the sequence for σ, and all positions will have been visited before getting back to the initial position, as required.

As for the equal probability of the permutations, it suffices to observe that the modified algorithm involves $(n–1)!$ distinct possible sequences of random numbers produced, each of which clearly produces a different permutation, and each of which occurs—assuming the random number source is unbiased—with equal probability. The $(n–1)!$ different permutations so produced precisely exhaust the set of cycles of length n: each such cycle has a unique cycle notation with the value n in the final position, which allows for $(n–1)!$ permutations of the remaining values to fill the other positions of the cycle notation.

A sample implementation of Sattolo's algorithm in Python is:

```
from random import randrange

def sattoloCycle(items):
    i = len(items)
    while i > 1:
        i = i - 1
        j = randrange(i)  # 0 <= j <= i-1
        items[j], items[i] = items[i], items[j]
    return
```

Comparison with other shuffling algorithms

The Fisher–Yates shuffle is quite efficient; indeed, its asymptotic time and space complexity are optimal. Combined with a high-quality unbiased random number source, it is also guaranteed to produce unbiased results. Compared to some other solutions, it also has the advantage that, if only part of the resulting permutation is needed, it can be stopped halfway through, or even stopped and restarted repeatedly, generating the permutation incrementally as needed.

An alternative method assigns a random number to each element of the set to be shuffled and then sorts the set according to the assigned numbers. The sorting method has the same asymptotic time complexity as Fisher-Yates: although general sorting is $O(n \log n)$, numbers are efficiently sorted using Radix sort in $O(n)$ time. Like the Fisher–Yates shuffle, the sorting method produces unbiased results. However, care must be taken to ensure that the assigned random numbers are never duplicated, since sorting algorithms typically don't order elements randomly in case of a tie. Additionally, this method requires asymptotically larger space: $O(n)$ additional storage space for the random numbers, versus $O(1)$ space for the Fisher-Yates shuffle. Finally, we note that the sorting method has a simple parallel implementation, unlike the Fisher-Yates shuffle, which is sequential.

A variant of the above method that has seen some use in languages that support sorting with user-specified comparison functions is to shuffle a list by sorting it with a comparison function that returns random values. However, *this is an extremely bad method*: it is very likely to produce highly non-uniform distributions, which in addition depends heavily on the sorting algorithm used. For instance suppose quicksort is used as sorting algorithm, with a fixed element selected as first pivot element. The algorithm starts comparing the pivot with all other elements to separate them into those less and those greater than it, and the relative sizes of those groups will determine the final place of the pivot element. For a uniformly distributed random permutation, each possible final position should be equally likely for the pivot element, but if each of the initial comparisons returns "less" or "greater" with equal probability, then that position will have a binomial distribution for $p = 1/2$, which gives positions near the middle of the sequence with a much higher probability for than positions near the ends. Randomized comparison functions applied to other sorting methods like merge sort may produce results that appear more uniform, but are not quite so either, since merging two sequences by repeatedly choosing one of them with equal probability (until the choice is forced by the exhaustion of one sequence) does not produce results with a uniform distribution; instead the probability to choose a sequence should be proportional to the number of elements left in it. In fact no method that uses only two-way random events with equal probability ("coin flipping"), repeated a bounded number of times, can produce permutations of a sequence (of more than two elements) with a uniform distribution, because every execution path will have as probability a rational number with as denominator a power of 2, while the required probability $1/n!$ for each possible permutation is not of that form.

In principle this shuffling method can even result in program failures like endless loops or access violations, because the correctness of a sorting algorithm may depend on properties of the order relation (like transitivity) that a comparison producing random values will certainly not have. While this kind of behaviour should not occur with sorting routines that never perform a comparison whose outcome can be predicted with certainty (based on previous

comparisons), there can be valid reasons for deliberately making such comparisons. For instance the fact that any element should compare equal to itself allows using them as sentinel value for efficiency reasons, and if this is the case, a random comparison function would break the sorting algorithm.

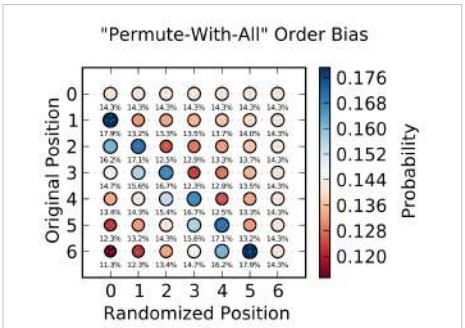
Potential sources of bias

Care must be taken when implementing the Fisher–Yates shuffle, both in the implementation of the algorithm itself and in the generation of the random numbers it is built on, otherwise the results may show detectable bias. A number of common sources of bias have been listed below.

Implementation errors

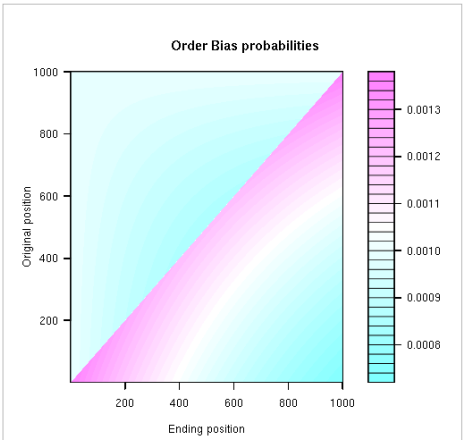
A common error when implementing the Fisher–Yates shuffle is to pick the random numbers from the wrong range. The flawed algorithm may appear to work correctly, but it will not produce each possible permutation with equal probability, and it may not produce certain permutations at all. For example, a common off-by-one error would be choosing the index j of the entry to swap in the example above to be always strictly less than the index i of the entry it will be swapped with. This turns the Fisher–Yates shuffle into Sattolo's algorithm, which produces only permutations consisting of a single cycle involving all elements: in particular, with this modification, no element of the array can ever end up in its original position.

Similarly, always selecting j from the entire range of valid array indices on *every* iteration also produces a result which is biased, albeit less obviously so. This can be seen from the fact that doing so yields n^n distinct possible sequences of swaps, whereas there are only $n!$ possible permutations of an n-element array. Since n^n can never be evenly divisible by $n!$ when $n > 2$ (as the latter is divisible by $n-1$, which shares no prime factors with n), some permutations must be produced by more of the n^n sequences of swaps than others. As a concrete example of this bias, observe the distribution of possible outcomes of shuffling a three-element array [1, 2, 3]. There are 6 possible permutations of this array (3! = 6), but the algorithm produces 27 possible shuffles ($3^3 = 27$). In this case, [1, 2, 3], [3, 1, 2], and [3, 2, 1] each result from 4 of the 27 shuffles, while each of the remaining 3 permutations occurs in 5 of the 27 shuffles.



Order bias from incorrect implementation

The matrix to the right shows the probability of each element in a list of length 7 ending up in any other position. Observe that for most elements, ending up in their original position (the matrix's main diagonal) has lowest probability, and moving one slot backwards has highest probability.



Order bias from incorrect implementation - n = 1000

Modulo bias

Doing a Fisher–Yates shuffle involves picking uniformly distributed random integers from various ranges. Most random number generators, however—whether true or pseudorandom—will only directly provide numbers in some fixed range, such as, say, from 0 to $2^{32}-1$. A simple and commonly used way to force such numbers into a desired smaller range is to apply the modulo operator; that is, to divide them by the size of the range and take the remainder. However, the need, in a Fisher–Yates shuffle, to generate random numbers in every range from 0–1 to 0–n pretty much guarantees that some of these ranges will not evenly divide the natural range of the

random number generator. Thus, the remainders will not always be evenly distributed and, worse yet, the bias will be systematically in favor of small remainders.

For example, assume that your random number source gives numbers from 0 to 99 (as was the case for Fisher and Yates' original tables), and that you wish to obtain an unbiased random number from 0 to 15. If you simply divide the numbers by 16 and take the remainder, you'll find that the numbers 0–3 occur about 17% more often than others. This is because 16 does not evenly divide 100: the largest multiple of 16 less than or equal to 100 is 6×16 = 96, and it is the numbers in the incomplete range 96–99 that cause the bias. The simplest way to fix the problem is to discard those numbers before taking the remainder and to keep trying again until a number in the suitable range comes up. While in principle this could, in the worst case, take forever, the expected number of retries will always be less than one.

A related problem occurs with implementations that first generate a random floating-point number—usually in the range [0,1)—and then multiply it by the size of the desired range and round down. The problem here is that random floating-point numbers, however carefully generated, always have only finite precision. This means that there are only a finite number of possible floating point values in any given range, and if the range is divided into a number of segments that doesn't divide this number evenly, some segments will end up with more possible values than others. While the resulting bias will not show the same systematic downward trend as in the previous case, it will still be there.

Pseudorandom generators: problems involving state space, seeding, and usage

Size of PRNG seeds and the largest list where every permutation could be reached

seed bits	maximum list length
0	1
1	2
3	3
5	4
7	5
10	6
13	7
16	8
24	10
32	12
48	16
64	20
128	34
160	40
226	52
256	57
512	98
1024	170
1600	245
19937	2080

44497	4199

An additional problem occurs when the Fisher–Yates shuffle is used with a pseudorandom number generator or PRNG: as the sequence of numbers output by such a generator is entirely determined by its internal state at the start of a sequence, a shuffle driven by such a generator cannot possibly produce more distinct permutations than the generator has distinct possible states. Even when the number of possible states exceeds the number of permutations, the irregular nature of the mapping from sequences of numbers to permutations means that some permutations will occur more often than others. Thus, to minimize bias, the number of states of the PRNG should exceed the number of permutations by at least several orders of magnitude.

For example, the built-in pseudorandom number generator provided by many programming languages and/or libraries may often have only 32 bits of internal state, which means it can only produce 2^{32} different sequences of numbers. If such a generator is used to shuffle a deck of 52 playing cards, it can only ever produce a very small fraction of the 52! ≈ $2^{225.6}$ possible permutations. It's impossible for a generator with less than 226 bits of internal state to produce all the possible permutations of a 52-card deck.

Also, of course, no pseudorandom number generator can produce more distinct sequences, starting from the point of initialization, than there are distinct seed values it may be initialized with. Thus, a generator that has 1024 bits of internal state but which is initialized with a 32-bit seed can still only produce 2^{32} different permutations right after initialization. It can produce more permutations if one exercises the generator a great many times before starting to use it for generating permutations, but this is a very inefficient way of increasing randomness: supposing one can arrange to use the generator a random number of up to a billion, say 2^{30} for simplicity, times between initialization and generating permutations, then the number of possible permutations is still only 2^{62}.

A further problem occurs when a simple linear congruential PRNG is used with the divide-and-take-remainder method of range reduction described above. The problem here is that the low-order bits of a linear congruential PRNG are less random than the high-order ones: the low n bits of the generator themselves have a period of at most 2^n. When the divisor is a power of two, taking the remainder essentially means throwing away the high-order bits, such that one ends up with a significantly less random value. This is an example of the general rule that a poor-quality RNG or PRNG will produce poor-quality shuffles.

Finally, it is to be noted that even with perfect random number generation, flaws can be introduced into an implementation by improper usage of the generator. For example, suppose a Java implementation creates a new generator for each call to the shuffler, without passing constructor arguments. The generator will then be default-seeded by the language's time-of-day (System.currentTimeMillis() in the case of Java). So if two callers call the shuffler within a time-span less than the granularity of the clock (one millisecond in the case of Java), the generators they create will be identical, and (for arrays of the same length) the same permutation will be generated. This is almost certain to happen if the shuffler is called many times in rapid succession, leading to an extremely non-uniform distribution in such cases; it can also apply to independent calls from different threads. A more robust Java implementation would use a single static instance of the generator defined outside the shuffler function.

References

[1] Note: the 6th edition, ISBN 0-02-844720-4, is available on the web (http://digital.library.adelaide.edu.au/coll/special/fisher/stat_tab.pdf), but gives a different shuffling algorithm by C. R. Rao.

External links

- http://bost.ocks.org/mike/shuffle/An Interactive example

Article Sources and Contributors

Fisher–Yates shuffle *Source*: http://en.wikipedia.org/w/index.php?oldid=602069219 *Contributors*: 1&only, 2knights2, Adrianwn, Alexhofsteede, AllTom, Andreas Kaufmann, Anonymous Dissident, Apokrif, Blonkm, BryanC, CRGreathouse, Cagdasgerede, Charbal, ChrisGualtieri, Civinext, Curly Turkey, David Eppstein, Eelvex, EmilJ, Fanf, Furrykef, Gang65, Garyzx, Giftlite, Gilliam, Glrx, Graue, GregorB, Hellclanner, Ilikestuffalot, Ilmari Karonen, JWilk, Jberryman, JumpDiscont, Knakts, Kostmo, Loadmaster, Luther93, MacMog, Marc van Leeuwen, Matteo.migliore, Mattstan, Melcombe, Michael Hardy, Nanite, Nbarth, NeonMerlin, Not-just-yeti, Oremj, Pgimeno, PhiLho, Qwertyus, Reardonj, Roxstar001, SESteve, Sligocki, Startswithj, Strait, Svick, Sytelus, Tbhotch, Thehebrewhammer, Thinking of England, Tkgd2007, Trusilver, Unbitwise, Wd.acgrs, Wolever, Woliveirajr, Wtuvell, 159 anonymous edits

Image Sources, Licenses and Contributors

Image:Probabilities7.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Probabilities7.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Kostmo

Image:Orderbias.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Orderbias.png *License*: Creative Commons Attribution 3.0 *Contributors*: Eelvex, Jason Davies, John of Reading

License