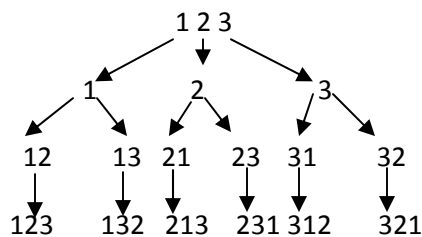


## The Fisher-Yates shuffle algorithm

Random permutations (or shuffles) are used very often world-wide both in computing and day-to-day life. Most common uses are in card shuffling, mathematics, cryptography, simulations. In this essay the topic of random permutations and random generators will be introduced. Then, the two shuffling algorithms, Fisher-Yates and Knuth, will be presented and their efficiency in terms of time and memory occupied will be discussed. Finally, problems occurring when the algorithms are badly implemented will be presented.

### Random permutations

Finding a random permutation can be done in a very simple way that doesn't require any particular algorithm. It requires finding all possible permutations of  $N$  objects and picking a random permutation. The best way of showing all possible permutations is by drawing a permutation tree.



This solution however is extremely inefficient as the total number of permutations of  $N$  objects is  $N!$ . Random permutations are often used to shuffle cards and in this case there would be a total of  $52!$  (or approximately  $8 \cdot 10^{67}$ ) permutations. Not only this would take a very long time to obtain a result, but it will also occupy a lot of memory, rendering this algorithm impossible to compute. There are, however, other methods of obtaining a random permutation that are extremely efficient. But these methods depend on the efficiency and accuracy of the pseudo-random number generators.

### Pseudo-random number generators

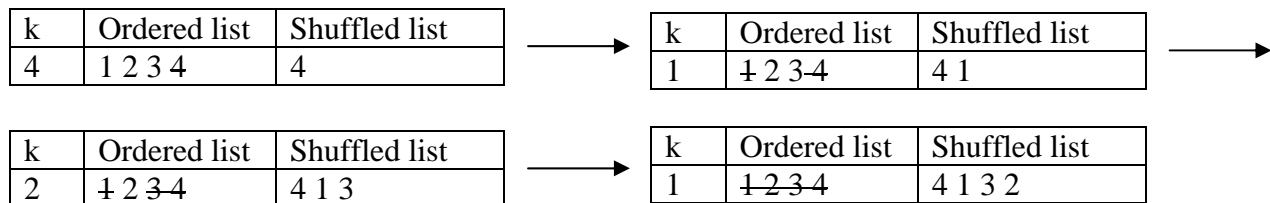
Most, if not all programming languages have libraries that include a pseudo-random number generator. This generator usually returns a random number between 0 and 1 (not including 1). In a perfect generator all numbers have the same probability of being selected but in the pseudo generators some numbers have zero probability. In order to build this generator a

series of numbers from 1 to N (usually  $2^{32}$  for a 32-bit processor) is used. A random number in that range is selected and then divided by N. For most computers this will provide a maximum of  $2^{32}$  (~ 4 billion) numbers between 0 and 1. For small data sets this range will produce a close enough result, but very large data sets the range is not nearly enough. If we take the problem described above (finding all possible random permutations for a 52-card deck and choosing a random permutation) and we assume that the problem is solvable from the perspective of space and time, we would need a random number in between 1 and  $10^{68}$ , but on regular PCs the maximum number of randomly generated numbers is  $10^9$ . We can conclude from this that only an extremely small number of possible permutations will be selected while most of them will never come up. So, in order to render the random number generators efficient data sizes smaller than  $2^{32}$  should be used. [4,5]

### Fisher-Yates algorithm

The original Fisher-Yates algorithm was described in *Statistical Tables for Biological, Agricultural and Medical Research* written by Ronald Fisher and Frank Yates in 1936. While discussing random numbers and their uses, the authors pose the following problem: how can you arrange 16 treatments, numbered 1 thru 16, in a random order and make sure that this is not biased in any way [1]. The solution they provide (generalized to N numbers to be ordered) is the following:

- pick a random number, k, from 1 to N
- count the  $k^{\text{th}}$  number, put it at the beginning of the random ordered list and cross it out
- subtract 1 from N and assign to N this new value
- repeat the above steps until all numbers have been crossed out



In 1936, in the absence of random number generators the authors composed and used a table comprised of random numbers between 0 and 99. This of course could have posed a problem since it really limits the potential of this algorithm. These random numbers also raised the issue of biasness. In the case they provide, with 16 items to be randomly ordered, a random number would be picked from the table and then the remainder of its division to 16 would be considered as the random number. The 100 numbers, however, don't divide evenly in sets of 16 numbers so numbers 97, 98, 99 will provide a bias towards its remainders (1,2 and 3). For this

reason these numbers should be excluded in order to make the process of arranging numbers in a random order completely unbiased.

This algorithm was implemented in Python and the built in pseudo-random number generator was used to generate the random numbers. In order to determine its efficiency, this algorithm was used on different list sizes (10,000 to 100,000) and timed. From the obtained results it can be seen that this algorithm is  $O(n^2)$ .

Python implementation:

```
while len(A)!=0:
    k = int(random()*n) + 1
    B.append(A[k-1])
    A.remove(A[k-1])
    n = n-1
```

### Knuth/Durstenfeld algorithm

This version of the shuffle algorithm was published in 1963 by L.E. Moses and R.V. Oakford [2] and in 1964 by Richard Durstenfeld[3], but it is most widely known as the Knuth shuffle as it was published in volume 2 of *The Art of Computer Programming* in 1969[4].

The way to shuffle cards or to compute a random permutation of N objects described by Knuth is the following:

- generate a random number, u in between 0 and 1
- let k be equal to the integer value of  $N*u$  plus 1
- exchange the  $k^{\text{th}}$  number with the  $N^{\text{th}}$
- decrease N by 1
- repeat the first four steps until  $N = 1$

	Step 1	Step 2	Step 3	Step 4
k		2	1	1
List	1 2 3 4	1 4 3   2	3 4   1 2	4   3 1 2

This algorithm was implemented in Python and the built in pseudo-random number generator was used to generate the random numbers. In order to determine its efficiency, this algorithm was used on different list sizes (10,000 to 1,000,000) and timed. From the obtained results it can be seen that this algorithm is  $O(n \log(n))$ .

When compared to the original Fisher-Yates algorithm, the Knuth shuffle is not only more efficient, but it also has the advantage of not occupying as much memory. In the Fisher-Yates shuffle two lists are used instead of one, which will occupy twice as much space. This represents a problem for very large data sizes as it will cause overflow and inaccurate results.

Python implementation:

```
while n>1:  
    k = int(random()*n) + 1  
    A[k-1], A[n-1] = A[n-1], A[k-1]  
    n=n-1
```

### Common uses of the Shuffle Algorithm and common problems

The most common applications in which these algorithms are used can be found in online card games. Providers of online games and other virtual versions of poker, blackjack, solitaire, etc. use one of these shuffle algorithms. Since shuffling cards only involves 52 cards or objects to be randomly permuted (a very small data size) the efficiency of the algorithm isn't as important and so either one of the two algorithms can be used.

The most important aspect that must be considered when using these algorithms is correct implementation. Even the smallest difference in the algorithm can produce a biased permutation and can affect many players who choose to gamble online. Let's take for example a modified version of Knuth (this version was actually used by ASF Software, Inc [5]):

```
for i in range(52):  
    k = int(random()*51) + 1  
    A[k-1], A[i] = A[i], A[k-1]
```

The following permutation tree shows all possible permutations that result from the above algorithm.



It can be seen that there are 312, 321 and 123 come up 4 times while 231, 213, 132 come up 5 times and so it is more likely to have 231, 213, 132. This problem can be detrimental in online gambling as knowing which card will be more likely to appear can help an experienced player cheat at these games.

Another problem that appears in the above algorithm is that k will be a random number from 1 to 51. Since there are 52 elements, all permutations resulted from the algorithm will have number 52 at the end.

The shuffle (random permutation) is often used in many fields and hence providing fast and accurate algorithms, such as the Fisher-Yates and Knuth algorithms, is essential in finding optimal solutions in cryptography, simulations and games.

References:

[1] Fisher R., Yates F. [1938] (1943) *Statistical Tables for Biological, Agricultural and Medical Research*, 2<sup>nd</sup> edition, London, Oliver and Boyd, p. 23-24

[2] Moses L.E., Oakford R.V.[1963] *Tables of Random Permutations*, Stanford University Press

[3] Durstenfeld R.[1964] *Communications of the ACM*, vol. 7, issue 7(July)

[4] Knuth D.[1969] (1981) *The Art of Computer Programming*, 2<sup>nd</sup> edition, Addison-Wesley, pp. 139-140

[5] [http://www.cigital.com/papers/download/developer\\_gambling.php](http://www.cigital.com/papers/download/developer_gambling.php)