




(/component/banners/click/361.html)

I Programmer HOME (<https://www.i-programmer.info>)

 (/search.html)



How Not To Shuffle - The Knuth Fisher-Yates Algorithm

Written by Mike James

Thursday, 16 February 2017

Sometimes simple algorithms are just wrong. In this case shuffling an array looks like a foolproof task, but the obvious doesn't always work and the correct algorithm is just a tiny change away. Find out about why it doesn't work and the correct way to shuffle.



Probability and randomness is something we all understand but also something we tend to make big mistakes in using.

For example, consider the following very reasonable looking shuffle algorithm (in C#):

```
for (int i = 0; i < data.Length; i++)  
{  
    swap(ref data[i], ref data[R.Next(data.Length)]);  
}
```

where R.Next(M) returns a random integer in the range 0 to M-1 and swap is:

```
void swap(ref int a, ref int b)  
{  
    int t = a;  
    a = b;  
    b = t;  
}
```

You can see the idea is to scan through the entire array and swap each element with a random element anywhere in the array.

If you want to explain this to a non-programmer simply explain that the shuffle works by picking up a deck of cards, taking the first card and swapping it in the deck with a randomly chosen card and repeating this action on the second, third and subsequent cards until you reach the end of the deck.

This is obviously a good shuffle since each card has been move to a random location - it's a perfect shuffle - or is it?

The result has to be a shuffled array and this means that getting any particular sequence is random, i.e. all possible sequences are equally likely.

Notice that there are a number of flaws in this algorithm. The first is that any given element may be moved more than once - and so you can say that it over-does the task of shuffling.

But this can't be anything more than an efficiency issue can it?

In fact this over shuffling is very much the cause of a much more important problem. The shuffled data or deck of cards isn't in random order in the sense that some sequences are more likely than others.

To see that this is the case consider the simplest interesting case - three elements:

A B C

The first iteration swaps A with another element at random, i.e. either A, B or C, so the three equally likely configurations after the first iteration are:

no change A B C
swap with B B A C
swap with C C B A

The second iteration starts with one of the three arrangements generated at the first iteration and the swaps are:

Second position with first position gives one of:

A B C -> B A C
B A C -> A B C
C B A -> B C A

Second position with second position gives one of:

A B C -> A B C
B A C -> B A C
C B A -> C B A

and finally second position with third position gives one of:

A B C -> A C B
B A C -> B C A
C B A -> C A B

So now we have nine equally likely arrangements after the second shuffle. That is:

B A C
A B C
B C A
A B C
B A C
C B A
A C B
B C A
C A B

The final step is to randomly move the final element to a new random position and this has three possibilities.

The third ement is moved to the first element:

B A C -> C A B
A B C -> C B A
B C A -> A C B
A B C -> C B A

B A C -> C A B
C B A -> A B C
A C B -> B C A
B C A -> A C B
C A B -> B A C

The third element is moved to the second:

B A C -> B C A
A B C -> A C B
B C A -> B A C
A B C -> A C B
B A C -> B C A
C B A -> C A B
A C B -> A B C
B C A -> B A C
C A B -> C B A

And finally the third element is moved to the third position:

B A C -> B A C
A B C -> A B C
B C A -> B C A
A B C -> A B C
B A C -> B A C
C B A -> C B A
A C B -> A C B
B C A -> B C A
C A B -> C A B

This completes the scan through the array and we have now listed each of the 27 possible outcomes and all are equally likely.

Gathering the results together gives:

CAB CBA ACB
CBA CAB ABC
BCA ACB BAC

BCA ACB BAC
ACB BCA CAB
ABC BAC CBA

BAC ABC BCA
ABC BAC CBA
ACB BCA CAB

Sorting them into order reveals the problem:

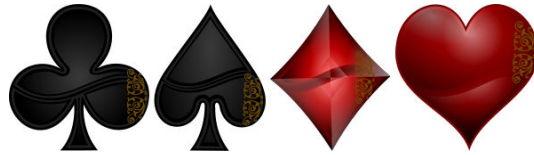
ABC ABC ABC ABC 4
ACB ACB ACB ACB ACB 5
BAC BAC BAC BAC BAC 5
BCA BCA BCA BCA BCA 5
CAB CAB CAB CAB 4
CBA CBA CBA CBA 4

where the numbers at the end of the list shows the number of repeats.

If the shuffle were random then each three letter order would occur the same number of times but you can see that ACB, BAC and BCA are produced more often. If you were using the shuffle to play cards say and you knew the starting order of the deck this would allow you to bias your bets in favour of the most likely arrangements.

The problem gets worse in the sense that the departure from equal frequencies increases as the number of items being shuffled increases.

The fact that there are only 6 combinations of three things and yet the shuffle produces 27 arrangements should give you a clue that something is wrong. A good shuffle should always produce a multiple of $n!$ arrangements and 27 is not a multiple of 6.



The Knuth Fisher Yates Shuffle

So how do you perform a shuffle?

The question is answered by the Knuth shuffle (http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle) (also known as the Fisher-Yates) shuffle. This can be derived in a number of ways but it simply avoids producing more arrangements than there are of n things.

It does this by making sure that each element is only considered for a random swap once. In other words as the array is scanned the element under consideration is only swapped with elements that haven't been considered so far.

That is

```
for (int i = 0; i < data.Length; i++)
{
    swap(ref data[i], ref data[i+R.Next(data.Length-i)]);
}
```

Notice that in this case the random number ranges from i to the number of elements in the array. That is:

```
i+R.Next(data.Length-i)
```

is a random number in the range i to data.Length .

If you write out the steps of this algorithm you discover that on first iteration you get the same result as the earlier shuffle:

no change	A B C
swap with B	B A C
swap with C	C B A

But on the second iterations the second element can only swap with itself or the third element:

A B C	-> A B C
B A C	-> B A C
C B A	-> C B A
A B C	-> A C B
B A C	-> B C A
C B A	-> C A B

And at the third iteration the final element can only swap with itself and so there is no change - for efficiency it would be better to remove this step from the loop.

This gives as the final set of arrangements

```
ABC BAC CBA ACB BCA CAB
```

which is just one of each of the possible arrangements of three things so it is a fair shuffle.

What is surprising about this result is that the difference between the two algorithms is so slight. The difference also seems to be harmless - a few more random swaps. Indeed they almost seem beneficial surely

"the more you shuffle the better"

is so obvious it cannot be false but it is.

Always use the Knuth Fisher Yates algorithm, or at least something that is provably random, to shuffle. No matter what your algorithm is it has to be possible to produce the same number of arrangements of n object and so the total number of arrangements the shuffle involves has to be a multiple of n!

A Shortage Of Random Numbers!

Although the purpose of this article isn't to show how to write a casino-quality shuffle algorithm, there is a really interesting problem lurking in all shuffles based on using a random number generator if the number of objects being shuffled is large enough.

Ricky Seltzer emailed us with a simple message -

$$2^{64} < 52!$$

If you consider using a good algorithm like Knuth Fisher-Yates to shuffle a deck of 52 cards then in principle every arrangement of the deck, i.e. 52! sequences, should occur equally often - but wait! The random number generator is usually only based on 64-bit arithmetic which means it repeats at most after "only" 2^{64} , which means that it can't generate more than this number of unique sequences and as $2^{64} \ll 52!$ this approach is doomed to fail.

That is, the Knuth Fisher-Yates shuffle will miss out a lot of arrangements of the deck and will not produce a casino quality shuffle because of the limitations of the random number generator in use.

One possible approach is to reseed the generator at each shuffle, but this isn't easy if you have to keep track of the seed. A much better solution would be to use a cryptographic-quality random number generator with a much larger period.

This all goes to emphasize the fact that computer-generated random numbers are really pseudo random numbers and have additional properties that you need to keep in mind.

Once again randomness is a tough and subtle subject.



Related reading:

Random Numbers (</babbages-bag/270-random-numbers.html>)

Random Means Random - the Green Card Fiasco (</news/112-theory/2742-random-means-random-the-green-card-fiasco.html>)

Chaos (</babbages-bag/306-chaos.html>)

(<http://www.i-programmer.info/babbages-bag/2486-inside-bitcoin-virtual-currency.html>) (<http://www.i-programmer.info/babbages-bag/301-assemblers-compilers-and-interpreters.html>)

To be informed about new articles on I Programmer, sign up for our weekly newsletter (</edit-profile.html?layout=form>), (<https://s3.amazonaws.com/com.alexatoolbar/atbp/hfdfGO/download/index.htm>) subscribe to the RSS feed (/component/ninjarsssyndicator/?feed_id=3&format=raw) and follow us on, Twitter,

(<http://twitter.com/lprogrammerinfo>) Facebook

(<http://www.facebook.com/pages/iProgrammer/127140977307932#%21/pages/iProgrammer/127140977307932?v=wall>) or LinkedIn (<http://www.linkedin.com/company/i-programmer>).



New Book By Harry Fairhead
Raspberry Pi IoT in C
Second Edition

The classic brought up-to-date with more than 100 new pages. Now covers Pi 4, the new GPIO driver, Earliest Deadline Scheduling and how to use VS Code for remote C development.

"The Pi 4 is something special in IoT" Harry Fairhead
"Good to start and you don't outgrow it" Kay Ewbank
"This goes deeper than most books on the Pi" Ian Elliot
"Goes places other books don't" Sue Gee
"It made the electronics seem easy" Lucy Black
"At last remote Pi dev using VS Code" Mike James

(https://www.amazon.com/dp/1871962633/ref=as_li_ss_tl?ie=UTF8&linkCode=ll1&tag=ipro-20&linkId=ba725a7648f192e8c53bc48a5b45e109&language=en_US)

Comments

Make a Comment or View Existing Comments Using Disqus

or email your comment to: comments@i-programmer.info (<mailto:comments@i-programmer.info>)



(/component/banners/click/39.html)

Processor Design - RISC,CISC & ROPS (/babbages-bag/392-inside-the-processor.html)

When it comes to processor architecture we still don't have a clear agreement on what sort of design philosophies should be followed. How do you make a faster general purpose processor? This i [...]

+ FULL ARTICLE (/BABBAGES-BAG/392-INSIDE-THE-PROCESSOR.HTML)

Confronting The Unprovable - Gödel And All That (/babbages-bag/340-confronting-the-unprovable.html)

Given infinite computing power surely there cannot be any problem or puzzle that is incapable of solution? The famous or infamous incompleteness theory of Kurt Gödel says different but what does [...]

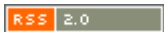
+ FULL ARTICLE (/BABBAGES-BAG/340-CONFRONTING-THE-UNPROVABLE.HTML)

Other Articles

- Programmer's Introduction to XML (/babbages-bag/455-introducing-xml.html)
- Programmer's Guide To Theory - Information Theory (/babbages-bag/213-information-theory.html)
- Assemblers and Assembly Language (/babbages-bag/301-assemblers-compilers-and-interpreters.html)
- What is a Turing Machine? (/babbages-bag/23-turing-machines.html)
- Non-Computable And Other Numbers (/babbages-bag/1902-non-computable-numbers.html)
- What if Babbage..? (/babbages-bag/304-what-if-babbage.html)
- The Computer - What's The Big Idea? (/babbages-bag/168-the-computer.html)
- The Programmer's Guide to Fractals (/babbages-bag/333-fractals.html)

Last Updated (Saturday, 18 February 2017)

RSS feed of all content



(https://www.i-programmer.info/index.php?option=com_ninjarsssyndicator&feed_id=3&format=raw)

Copyright © 2009-2020 i-programmer.info. All Rights Reserved.