Subject: Re: Randomize array order
Posted by Vince Hradil on Wed, 25 Jul 2007 19:59:14 GMT
View Forum Message <> Reply to Message

On Jul 25, 2:17 pm, Conor <cmanc...@gmail.com> wrote:
> Hi everyone!
>
> Anyone know an efficient way to randomize an array (I have a
> sorted array that I want unsorted). Initially, I tried something like
> this:
>
> array = findgen(1000000)
> unsort = array[sort(randomu(seed,1000000))]
>
> It works, but sorting on a million elements is rather slow. Anyone
> know a faster way?

Takes about 0.87 seconds on my machine? Is this too long?

Of course, if you want "with replacement" you could do:
unsort = array[long(1000000*randomu(seed,1000000))]

But you probably want a permutation, right? That'll be more difficult.

Subject: Re: Randomize array order
Posted by Conor on Thu, 26 Jul 2007 12:24:55 GMT
View Forum Message <> Reply to Message

On Jul 25, 3:59 pm, hradilv <hrad...@yahoo.com> wrote:
> On Jul 25, 2:17 pm, Conor <cmanc...@gmail.com> wrote:
>
>> Hi everyone!
>
>> Anyone know an efficient way to randomize an array (I have a
>> sorted array that I want unsorted). Initially, I tried something like
>> this:
>
>> array = findgen(1000000)
>> unsort = array[sort(randomu(seed,1000000))]
>
>> It works, but sorting on a million elements is rather slow. Anyone
>> know a faster way?
>
> Takes about 0.87 seconds on my machine? Is this too long?
>
> Of course, if you want "with replacement" you could do:
> unsort = array[long(1000000*randomu(seed,1000000))]

>
> But you probably want a permutation, right? That'll be more difficult.

It takes about 3 seconds on my machine (really crappy machine). The whole program takes about 6 seconds, so about half the program execution comes from the one sort statement. This is going to execute somewhere between 1000 and 10,000 times, so losing three seconds will save me between 50 minutes and 8 hours. Yeah, I need a permutation. Losing or repeating elements would be bad. It's always got to be the hard way, doesn't it :)

Subject: Re: Randomize array order
Posted by Conor on Thu, 26 Jul 2007 12:28:46 GMT
View Forum Message <> Reply to Message

On Jul 25, 3:59 pm, hradilv <hrad...@yahoo.com> wrote:
> On Jul 25, 2:17 pm, Conor <cmanc...@gmail.com> wrote:
>
>> Hi everyone!
>
>> Anyone know an efficient way to randomize an array (I have a
>> sorted array that I want unsorted). Initially, I tried something like
>> this:
>
>> array = findgen(1000000)
>> unsort = array[sort(randomu(seed,1000000))]
>
>> It works, but sorting on a million elements is rather slow. Anyone
>> know a faster way?
>
> Takes about 0.87 seconds on my machine? Is this too long?
>
> Of course, if you want "with replacement" you could do:
> unsort = array[long(1000000*randomu(seed,1000000))]
>
> But you probably want a permutation, right? That'll be more difficult.

You know, it's a shame that sort and histogram don't make use of the thread pool. If there were any algorithms that need to be multi-threaded, those would be it. Oh well :(

Subject: Re: Randomize array order
Posted by mattf on Thu, 26 Jul 2007 12:50:25 GMT
View Forum Message <> Reply to Message

On Jul 25, 3:17 pm, Conor <cmanc...@gmail.com> wrote:
> Hi everyone!
>
> Anyone know an efficient way to randomize an array (I have a
> sorted array that I want unsorted). Initially, I tried something like
> this:
>
> array = findgen(1000000)
> unsort = array[sort(randomu(seed,1000000))]
>
> It works, but sorting on a million elements is rather slow. Anyone
> know a faster way?

I think this is one of those vectorization cases where you have a trade-off between processor time and memory-- the one 'obvious' faster way that occurs to me is to pre-compute a sequence of a few billion random integers-- and then just select sub-sequences as needed. It's possible that there's some intermediate strategy that's not quite so extravagantly wasteful of memory.

Subject: Re: Randomize array order
Posted by Brian Larsen on Thu, 26 Jul 2007 13:04:19 GMT
View Forum Message <> Reply to Message

for the without replacement see http://www.dfanning.com/code_tips/randomindex.html
I have written a resample with and without replacement routine based on this and its works fine.

Cheers,

Brian

---------- ----------
Brian Larsen
Boston University
Center for Space Physics

Subject: Re: Randomize array order
Posted by Brian Larsen on Thu, 26 Jul 2007 13:06:21 GMT
View Forum Message <> Reply to Message

for the without replacement see http://www.dfanning.com/code_tips/randomindex.html
I have written a resample with and without replacement routine based on this and its works fine.

Cheers,

Brian

---------- ----------
Brian Larsen
Boston University
Center for Space Physics

Subject: Re: Randomize array order
Posted by Allan Whiteford on Thu, 26 Jul 2007 13:30:00 GMT
View Forum Message <> Reply to Message

Conor wrote:
> Hi everyone!
>
> Anyone know an efficient way to randomize an array (I have a
> sorted array that I want unsorted). Initially, I tried something like
> this:
>
> array = findgen(1000000)
> unsort = array[sort(randomu(seed,1000000))]
>
> It works, but sorting on a million elements is rather slow. Anyone
> know a faster way?
>

Conor,

Is it a million elements you want to do?

The following scales better:

```
pro shuffle,in
 b=long(n_elements(in)*randomu(seed,n_elements(in)))
        for i=0l,n_elements(in)-1 do begin
         tmp=in[i]
               in[i]=in[b[i]]
  in[b[i]]=tmp
        end
end
```

but on my machine, a million elements is around about where it starts to become as efficient as yours. For 10 million elements the above is a bit (17.05 seconds vs 12.92 seconds) but for 1 million elements they both come in at around 1.2 seconds (1.15 seconds vs 1.26 seconds). The above will scale as pretty much O(n) since it doesn't do any sorting but it

takes a hit in the practical implementation because of the loop in IDL-space. Your suggestion will scale worse than O(n) but it seems the overlap in the two methods is exactly where you want to work.

Maybe my loop can be made more efficient in practical terms but I don't think this is any better algorithm in terms of scaling (hard to imagine anything that could go faster than O(n) to randomise n things).

Probably not helpful but I thought it was interesting that the cross-over is exactly where you want to work. But, maybe I should get out more if I think that's especially interesting.

Thanks,

Allan

Subject: Re: Randomize array order
Posted by Conor on Thu, 26 Jul 2007 13:40:56 GMT
View Forum Message <> Reply to Message

On Jul 26, 9:30 am, Allan Whiteford
<allan.rem...@phys.remove.strath.ac.remove.uk> wrote:
> Conor wrote:
>> Hi everyone!
>
>> Anyone know an efficient way to randomize an array (I have a
>> sorted array that I want unsorted). Initially, I tried something like
>> this:
>
>> array = findgen(1000000)
>> unsort = array[sort(randomu(seed,1000000))]
>
>> It works, but sorting on a million elements is rather slow. Anyone
>> know a faster way?
>
> Conor,
>
> Is it a million elements you want to do?
>
> The following scales better:
>
> pro shuffle,in
> b=long(n_elements(in)*randomu(seed,n_elements(in)))
> for i=0l,n_elements(in)-1 do begin
> tmp=in[i]
> in[i]=in[b[i]]
> in[b[i]]=tmp

> end
> end
>
> but on my machine, a million elements is around about where it starts to
> become as efficient as yours. For 10 million elements the above is a bit
> (17.05 seconds vs 12.92 seconds) but for 1 million elements they both
> come in at around 1.2 seconds (1.15 seconds vs 1.26 seconds). The above
> will scale as pretty much O(n) since it doesn't do any sorting but it
> takes a hit in the practical implementation because of the loop in
> IDL-space. Your suggestion will scale worse than O(n) but it seems the
> overlap in the two methods is exactly where you want to work.
>
> Maybe my loop can be made more efficient in practical terms but I don't
> think this is any better algorithm in terms of scaling (hard to imagine
> anything that could go faster than O(n) to randomise n things).
>
> Probably not helpful but I thought it was interesting that the
> cross-over is exactly where you want to work. But, maybe I should get
> out more if I think that's especially interesting.
>
> Thanks,
>
> Allan

Thanks for the suggestions guys! I'll have to play around and see
what works best.

Subject: Re: Randomize array order
Posted by Vince Hradil on Thu, 26 Jul 2007 14:58:19 GMT
View Forum Message <> Reply to Message

On Jul 26, 8:40 am, Conor <cmanc...@gmail.com> wrote:
> On Jul 26, 9:30 am, Allan Whiteford
>
>
>
> <allan.rem...@phys.remove.strath.ac.remove.uk> wrote:
>> Conor wrote:
>>> Hi everyone!
>
>>> Anyone know an efficient way to randomize an array (I have a
>>> sorted array that I want unsorted). Initially, I tried something like
>>> this:
>
>>> array = findgen(1000000)
>>> unsort = array[sort(randomu(seed,1000000))]
>

>>> It works, but sorting on a million elements is rather slow. Anyone
>>> know a faster way?
>
>> Conor,
>
>> Is it a million elements you want to do?
>
>> The following scales better:
>
>> pro shuffle,in
>> b=long(n_elements(in)*randomu(seed,n_elements(in)))
>> for i=0l,n_elements(in)-1 do begin
>> tmp=in[i]
>> in[i]=in[b[i]]
>> in[b[i]]=tmp
>> end
>> end
>
>> but on my machine, a million elements is around about where it starts to
>> become as efficient as yours. For 10 million elements the above is a bit
>> (17.05 seconds vs 12.92 seconds) but for 1 million elements they both
>> come in at around 1.2 seconds (1.15 seconds vs 1.26 seconds). The above
>> will scale as pretty much O(n) since it doesn't do any sorting but it
>> takes a hit in the practical implementation because of the loop in
>> IDL-space. Your suggestion will scale worse than O(n) but it seems the
>> overlap in the two methods is exactly where you want to work.
>
>> Maybe my loop can be made more efficient in practical terms but I don't
>> think this is any better algorithm in terms of scaling (hard to imagine
>> anything that could go faster than O(n) to randomise n things).
>
>> Probably not helpful but I thought it was interesting that the
>> cross-over is exactly where you want to work. But, maybe I should get
>> out more if I think that's especially interesting.
>
>> Thanks,
>
>> Allan
>
> Thanks for the suggestions guys! I'll have to play around and see
> what works best.

Here's a table of results from my machine. All times are in seconds.
PC single processor, WinXP, IDL6.4

i	Niter	Rand-meth	Loop-meth
0	100000	0.0929999	0.110000
1	166810	0.0779998	0.0940001

2	278256	0.140000	0.157000
3	464158	0.297000	0.297000
4	774263	0.578000	0.562000
5	1291549	1.09400	0.890000
6	2154435	2.06300	1.48400
7	3593812	3.84400	2.56300
8	5994841	7.09400	4.31300
9	10000000	13.0470	7.29800

Here's my code:

```
function runsort, array
  na = n_elements(array)
  return, array[sort(randomu(seed,na))]
end

function lunsort, array
  na = n_elements(array)
  rarray = array

  b = long(na*randomu(seed,na))
  for i=0l, na-1 do begin
    tmp = rarray[i]
    rarray[i] = rarray[b[i]]
    rarray[b[i]] = tmp
  endfor

  return, rarray
end

pro test_unsort, randi=randi, loopi=loopi, nel=nel

  n = 10l
  nlo = 5l
  nhi = 7l
  fndx = findgen(n)/float(n-1)
  nel = long(10^( (nhi-nlo)*fndx + nlo ) )

  randi = fltarr(n)
  loopi = fltarr(n)
  for i=0l, n-1 do begin
    array = findgen(nel[i])
    t = systime(1)
    unsort = runsort(array)
    randi[i] = systime(1)-t

    t = systime(1)
    unsort = lunsort(array)
```

```
    loopi[i] = systime(1)-t

    print, i, nel[i], randi[i], loopi[i]
  endfor

  return
end
```

Subject: Re: Randomize array order
Posted by Vince Hradil on Thu, 26 Jul 2007 15:13:36 GMT
View Forum Message <> Reply to Message

On Jul 26, 9:58 am, hradilv <hrad...@yahoo.com> wrote:
> On Jul 26, 8:40 am, Conor <cmanc...@gmail.com> wrote:
>
>
>
>> On Jul 26, 9:30 am, Allan Whiteford
>
>> <allan.rem...@phys.remove.strath.ac.remove.uk> wrote:
>>> Conor wrote:
>>>> Hi everyone!
>
>>>> Anyone know an efficient way to randomize an array (I have a
>>>> sorted array that I want unsorted). Initially, I tried something like
>>>> this:
>
>>>> array = findgen(1000000)
>>>> unsort = array[sort(randomu(seed,1000000))]
>
>>>> It works, but sorting on a million elements is rather slow. Anyone
>>>> know a faster way?
>
>>> Conor,
>
>>> Is it a million elements you want to do?
>
>>> The following scales better:
>
>>> pro shuffle,in
>>> b=long(n_elements(in)*randomu(seed,n_elements(in)))
>>> for i=0l,n_elements(in)-1 do begin
>>> tmp=in[i]
>>> in[i]=in[b[i]]
>>> in[b[i]]=tmp
>>> end
>>> end

>
>>> but on my machine, a million elements is around about where it starts to
>>> become as efficient as yours. For 10 million elements the above is a bit
>>> (17.05 seconds vs 12.92 seconds) but for 1 million elements they both
>>> come in at around 1.2 seconds (1.15 seconds vs 1.26 seconds). The above
>>> will scale as pretty much O(n) since it doesn't do any sorting but it
>>> takes a hit in the practical implementation because of the loop in
>>> IDL-space. Your suggestion will scale worse than O(n) but it seems the
>>> overlap in the two methods is exactly where you want to work.
>
>>> Maybe my loop can be made more efficient in practical terms but I don't
>>> think this is any better algorithm in terms of scaling (hard to imagine
>>> anything that could go faster than O(n) to randomise n things).
>
>>> Probably not helpful but I thought it was interesting that the
>>> cross-over is exactly where you want to work. But, maybe I should get
>>> out more if I think that's especially interesting.
>
>>> Thanks,
>
>>> Allan
>
>> Thanks for the suggestions guys! I'll have to play around and see
>> what works best.
>
> Here's a table of results from my machine. All times are in seconds.
> PC single processor, WinXP, IDL6.4
>

	i	Niter	Rand-meth	Loop-meth
>	0	100000	0.0929999	0.110000
>	1	166810	0.0779998	0.0940001
>	2	278256	0.140000	0.157000
>	3	464158	0.297000	0.297000
>	4	774263	0.578000	0.562000
>	5	1291549	1.09400	0.890000
>	6	2154435	2.06300	1.48400
>	7	3593812	3.84400	2.56300
>	8	5994841	7.09400	4.31300
>	9	10000000	13.0470	7.29800

More details: Single Intel 1.86GHz, 2Gb RAM

Other machine: Sun Blade 2500 - Solaris 9, IDL 6.3 - Dual processor,
2Gb RAM

i	Niter	Rand-meth	Loop-meth
0	100000	0.112775	0.218330
1	166810	0.194601	0.370555

2	278256	0.369679	0.621675
3	464158	0.700207	1.05355
4	774263	1.32646	1.74441
5	1291549	2.42519	2.95356
6	2154435	4.38822	4.91093
7	3593812	8.63800	8.35843
8	5994841	15.6409	13.9243
9	10000000	28.9150	23.6173

Interesting, there's a crossover at ~ 3,000,000 where the loop method
starts to win.

Subject: Re: Randomize array order
Posted by Allan Whiteford on Thu, 26 Jul 2007 15:49:13 GMT
View Forum Message <> Reply to Message

hradilv wrote:
> On Jul 26, 9:58 am, hradilv <hrad...@yahoo.com> wrote:
>
>> On Jul 26, 8:40 am, Conor <cmanc...@gmail.com> wrote:
>>
>>
>>
>>
>>> On Jul 26, 9:30 am, Allan Whiteford
>>
>>> <allan.rem...@phys.remove.strath.ac.remove.uk> wrote:
>>>
>>>> Conor wrote:
>>>>
>>>> >Hi everyone!
>>
>>>> > Anyone know an efficient way to randomize an array (I have a
>>>> >sorted array that I want unsorted). Initially, I tried something like
>>>> >this:
>>
>>>> >array = findgen(1000000)
>>>> >unsort = array[sort(randomu(seed,1000000))]
>>
>>>> >It works, but sorting on a million elements is rather slow. Anyone
>>>> >know a faster way?
>>
>>>> Conor,
>>
>>>> Is it a million elements you want to do?
>>
>>>> The following scales better:

>>
>>>> pro shuffle,in
>>>> b=long(n_elements(in)*randomu(seed,n_elements(in)))
>>>> for i=0l,n_elements(in)-1 do begin
>>>> tmp=in[i]
>>>> in[i]=in[b[i]]
>>>> in[b[i]]=tmp
>>>> end
>>>> end
>>
>>>> but on my machine, a million elements is around about where it starts to
>>>> become as efficient as yours. For 10 million elements the above is a bit
>>>> (17.05 seconds vs 12.92 seconds) but for 1 million elements they both
>>>> come in at around 1.2 seconds (1.15 seconds vs 1.26 seconds). The above
>>>> will scale as pretty much O(n) since it doesn't do any sorting but it
>>>> takes a hit in the practical implementation because of the loop in
>>>> IDL-space. Your suggestion will scale worse than O(n) but it seems the
>>>> overlap in the two methods is exactly where you want to work.
>>
>>>> Maybe my loop can be made more efficient in practical terms but I don't
>>>> think this is any better algorithm in terms of scaling (hard to imagine
>>>> anything that could go faster than O(n) to randomise n things).
>>
>>>> Probably not helpful but I thought it was interesting that the
>>>> cross-over is exactly where you want to work. But, maybe I should get
>>>> out more if I think that's especially interesting.
>>
>>>> Thanks,
>>
>>>> Allan
>>
>>> Thanks for the suggestions guys! I'll have to play around and see
>>> what works best.
>>
>> Here's a table of results from my machine. All times are in seconds.
>> PC single processor, WinXP, IDL6.4
>>
>> i Niter Rand-meth Loop-meth
>> 0 100000 0.0929999 0.110000
>> 1 166810 0.0779998 0.0940001
>> 2 278256 0.140000 0.157000
>> 3 464158 0.297000 0.297000
>> 4 774263 0.578000 0.562000
>> 5 1291549 1.09400 0.890000
>> 6 2154435 2.06300 1.48400
>> 7 3593812 3.84400 2.56300
>> 8 5994841 7.09400 4.31300
>> 9 10000000 13.0470 7.29800

>
>
> More details: Single Intel 1.86GHz, 2Gb RAM
>
> Other machine: Sun Blade 2500 - Solaris 9, IDL 6.3 - Dual processor,
> 2Gb RAM
>
> i Niter Rand-meth Loop-meth
> 0 100000 0.112775 0.218330
> 1 166810 0.194601 0.370555
> 2 278256 0.369679 0.621675
> 3 464158 0.700207 1.05355
> 4 774263 1.32646 1.74441
> 5 1291549 2.42519 2.95356
> 6 2154435 4.38822 4.91093
> 7 3593812 8.63800 8.35843
> 8 5994841 15.6409 13.9243
> 9 10000000 28.9150 23.6173
>
> Interesting, there's a crossover at ~ 3,000,000 where the loop method
> starts to win.
>

Here's what I get on a dual core 3GHz Pentium 4 with 2GB of RAM running
Linux (FC4) using IDL6.2:

i	Niter	Rand-meth	Loop-meth
0	100000	0.0818000	0.120713
1	166810	0.140054	0.205111
2	278256	0.255531	0.340111
3	464158	0.462941	0.572567
4	774263	0.835279	0.973762
5	1291549	1.53649	1.71803
6	2154435	3.08281	2.83829
7	3593812	5.27431	4.71084
8	5994841	10.6316	7.85549
9	10000000	17.4706	13.6622

kind of annoying that your 1.8GHz machine running windows goes faster
than my 3GHz running Linux. Not as bad as how slow the Sun goes though.

Incidentally, previously I was quoting raw CPU times rather than the
wall clock times which your routine prints out.

Thanks,

Allan

Subject: Re: Randomize array order
Posted by Conor on Thu, 26 Jul 2007 16:22:27 GMT
View Forum Message <> Reply to Message

On Jul 26, 11:49 am, Allan Whiteford
<allan.rem...@phys.remove.strath.ac.remove.uk> wrote:
> hradilv wrote:
>> On Jul 26, 9:58 am, hradilv <hrad...@yahoo.com> wrote:
>
>>> On Jul 26, 8:40 am, Conor <cmanc...@gmail.com> wrote:
>
>>>> On Jul 26, 9:30 am, Allan Whiteford
>
>>>> <allan.rem...@phys.remove.strath.ac.remove.uk> wrote:
>
>>>> >Conor wrote:
>
>>>> >>Hi everyone!
>
>>>> >> Anyone know an efficient way to randomize an array (I have a
>>>> >>sorted array that I want unsorted). Initially, I tried something like
>>>> >>this:
>
>>>> >>array = findgen(1000000)
>>>> >>unsort = array[sort(randomu(seed,1000000))]
>
>>>> >>It works, but sorting on a million elements is rather slow. Anyone
>>>> >>know a faster way?
>
>>>> >Conor,
>
>>>> >Is it a million elements you want to do?
>
>>>> >The following scales better:
>
>>>> >pro shuffle,in
>>>> > b=long(n_elements(in)*randomu(seed,n_elements(in)))
>>>> > for i=0l,n_elements(in)-1 do begin
>>>> > tmp=in[i]
>>>> > in[i]=in[b[i]]
>>>> > in[b[i]]=tmp
>>>> > end
>>>> >end
>
>>>> >but on my machine, a million elements is around about where it starts to
>>>> >become as efficient as yours. For 10 million elements the above is a bit
>>>> >(17.05 seconds vs 12.92 seconds) but for 1 million elements they both
>>>> >come in at around 1.2 seconds (1.15 seconds vs 1.26 seconds). The above
>>>> >will scale as pretty much O(n) since it doesn't do any sorting but it

>>>> >takes a hit in the practical implementation because of the loop in
>>>> >IDL-space. Your suggestion will scale worse than O(n) but it seems the
>>>> >overlap in the two methods is exactly where you want to work.
>
>>>> >Maybe my loop can be made more efficient in practical terms but I don't
>>>> >think this is any better algorithm in terms of scaling (hard to imagine
>>>> >anything that could go faster than O(n) to randomise n things).
>
>>>> >Probably not helpful but I thought it was interesting that the
>>>> >cross-over is exactly where you want to work. But, maybe I should get
>>>> >out more if I think that's especially interesting.
>
>>>> >Thanks,
>
>>>> >Allan
>
>>>> Thanks for the suggestions guys! I'll have to play around and see
>>>> what works best.
>
>>> Here's a table of results from my machine. All times are in seconds.
>>> PC single processor, WinXP, IDL6.4
>

>>>	i	Niter	Rand-meth	Loop-meth
>>>	0	100000	0.0929999	0.110000
>>>	1	166810	0.0779998	0.0940001
>>>	2	278256	0.140000	0.157000
>>>	3	464158	0.297000	0.297000
>>>	4	774263	0.578000	0.562000
>>>	5	1291549	1.09400	0.890000
>>>	6	2154435	2.06300	1.48400
>>>	7	3593812	3.84400	2.56300
>>>	8	5994841	7.09400	4.31300
>>>	9	10000000	13.0470	7.29800

>
>> More details: Single Intel 1.86GHz, 2Gb RAM
>
>> Other machine: Sun Blade 2500 - Solaris 9, IDL 6.3 - Dual processor,
>> 2Gb RAM
>

>>	i	Niter	Rand-meth	Loop-meth
>>	0	100000	0.112775	0.218330
>>	1	166810	0.194601	0.370555
>>	2	278256	0.369679	0.621675
>>	3	464158	0.700207	1.05355
>>	4	774263	1.32646	1.74441
>>	5	1291549	2.42519	2.95356
>>	6	2154435	4.38822	4.91093
>>	7	3593812	8.63800	8.35843

>> 8 5994841 15.6409 13.9243
>> 9 10000000 28.9150 23.6173
>
>> Interesting, there's a crossover at ~ 3,000,000 where the loop method
>> starts to win.
>
> Here's what I get on a dual core 3GHz Pentium 4 with 2GB of RAM running
> Linux (FC4) using IDL6.2:
>
> i Niter Rand-meth Loop-meth
> 0 100000 0.0818000 0.120713
> 1 166810 0.140054 0.205111
> 2 278256 0.255531 0.340111
> 3 464158 0.462941 0.572567
> 4 774263 0.835279 0.973762
> 5 1291549 1.53649 1.71803
> 6 2154435 3.08281 2.83829
> 7 3593812 5.27431 4.71084
> 8 5994841 10.6316 7.85549
> 9 10000000 17.4706 13.6622
>
> kind of annoying that your 1.8GHz machine running windows goes faster
> than my 3GHz running Linux. Not as bad as how slow the Sun goes though.
>
> Incidentally, previously I was quoting raw CPU times rather than the
> wall clock times which your routine prints out.
>
> Thanks,
>
> Allan

Here's what I get running it on my super old computer:

0	100000	0.231639	0.266472
1	166810	0.429814	0.450388
2	278256	0.768671	0.777250
3	464158	1.40014	1.29011
4	774263	2.55367	2.15114
5	1291549	4.66570	3.60980
6	2154435	8.48878	6.04430
7	3593812	15.3753	10.1437
8	5994841	29.2131	20.1072
9	10000000	52.2718	29.7969

Subject: Re: Randomize array order
Posted by David Streutker on Thu, 26 Jul 2007 16:57:31 GMT

How about a Knuth shuffle?

(Disclaimer: I'm not a statistician; I just found it on Wikipedia.)

```
function kunsort, array
  na = n_elements(array)
  random = randomu(seed,na) * (na - array - 1) + array
  for i=0L,na-2 do array[i] = array[random[i]]
  return, array
end
```

Added to Vince's code (last column):

```
      0     100000    0.0619998    0.0780001    0.0320001
      1     166810     0.125000     0.125000    0.0780001
      2     278256     0.282000     0.218000     0.141000
      3     464158     0.547000     0.406000     0.235000
      4     774263      1.07800     0.657000     0.390000
      5    1291549      1.93700      1.09400     0.703000
      6    2154435      3.51500      1.89100      1.20300
      7    3593812      6.34400      3.11000      1.98400
      8    5994841      11.5470      5.21800      3.36000
      9   10000000       20.3750      8.67200      5.60900
```

Windows XP, dual 2.66 GHz, 3 GB RAM, IDL 6.3

Subject: Re: Randomize array order
Posted by Vince Hradil on Thu, 26 Jul 2007 20:32:39 GMT

On Jul 26, 11:57 am, David Streutker <dstreut...@gmail.com> wrote:
> How about a Knuth shuffle?
>
> (Disclaimer: I'm not a statistician; I just found it on Wikipedia.)
>
> function kunsort, array
> na = n_elements(array)
> random = randomu(seed,na) * (na - array - 1) + array
> for i=0L,na-2 do array[i] = array[random[i]]
> return, array
> end
>
> Added to Vince's code (last column):
>
> 0 100000 0.0619998 0.0780001 0.0320001

```
>          1     166810     0.125000     0.125000     0.0780001
>          2     278256     0.282000     0.218000      0.141000
>          3     464158     0.547000     0.406000      0.235000
>          4     774263      1.07800     0.657000      0.390000
>          5    1291549      1.93700      1.09400      0.703000
>          6    2154435      3.51500      1.89100      1.20300
>          7    3593812      6.34400      3.11000      1.98400
>          8    5994841      11.5470      5.21800      3.36000
>          9   10000000      20.3750      8.67200      5.60900
>
>  Windows XP, dual 2.66 GHz, 3 GB RAM, IDL 6.3
```

I'm not sure, but I think that will give you "with replacement".

Subject: Re: Randomize array order
Posted by David Streutker on Thu, 26 Jul 2007 21:27:33 GMT
View Forum Message <> Reply to Message

On Jul 26, 2:32 pm, hradilv <hrad...@yahoo.com> wrote:
> On Jul 26, 11:57 am, David Streutker <dstreut...@gmail.com> wrote:
>
>
>
>> How about a Knuth shuffle?
>
>> (Disclaimer: I'm not a statistician; I just found it on Wikipedia.)
>
>> function kunsort, array
>> na = n_elements(array)
>> random = randomu(seed,na) * (na - array - 1) + array
>> for i=0L,na-2 do array[i] = array[random[i]]
>> return, array
>> end
>
>> Added to Vince's code (last column):
>
```
>>          0     100000    0.0619998    0.0780001    0.0320001
>>          1     166810     0.125000     0.125000    0.0780001
>>          2     278256     0.282000     0.218000     0.141000
>>          3     464158     0.547000     0.406000     0.235000
>>          4     774263      1.07800     0.657000     0.390000
>>          5    1291549      1.93700      1.09400     0.703000
>>          6    2154435      3.51500      1.89100      1.20300
>>          7    3593812      6.34400      3.11000      1.98400
>>          8    5994841      11.5470      5.21800      3.36000
>>          9   10000000      20.3750      8.67200      5.60900
>
```

>> Windows XP, dual 2.66 GHz, 3 GB RAM, IDL 6.3
>
> I'm not sure, but I think that will give you "with replacement".

You're right, I wasn't swapping. Corrected, and in the form of
Allan's method:

```
function kunsort, array
  na = n_elements(array)
  rarray = array

  b = randomu(seed,na-1) * (na - lindgen(na-1) - 1) + lindgen(na-1)
  for i=0L,na-2 do begin
    tmp = rarray[i]
    rarray[i] = rarray[b[i]]
    rarray[b[i]] = tmp
  endfor

  return, rarray
end
```

With the change, it's slightly slower than Allan's. However, for what
it's worth, there are claims this is a less biased method. (Again, I
am no expert. But the recent poker craze seems to have revived
interest in the probabilities of shuffling.)

Subject: Re: Randomize array order
Posted by Allan Whiteford on Fri, 27 Jul 2007 09:33:30 GMT
View Forum Message <> Reply to Message

David Streutker wrote:
> On Jul 26, 2:32 pm, hradilv <hrad...@yahoo.com> wrote:
>
>> On Jul 26, 11:57 am, David Streutker <dstreut...@gmail.com> wrote:
>>
>>
>>
>>
>>> How about a Knuth shuffle?
>>
>>> (Disclaimer: I'm not a statistician; I just found it on Wikipedia.)
>>
>>> function kunsort, array
>>> na = n_elements(array)
>>> random = randomu(seed,na) * (na - array - 1) + array
>>> for i=0L,na-2 do array[i] = array[random[i]]
>>> return, array

>>> end
>>
>>> Added to Vince's code (last column):
>>
>>> 0 100000 0.0619998 0.0780001 0.0320001
>>> 1 166810 0.125000 0.125000 0.0780001
>>> 2 278256 0.282000 0.218000 0.141000
>>> 3 464158 0.547000 0.406000 0.235000
>>> 4 774263 1.07800 0.657000 0.390000
>>> 5 1291549 1.93700 1.09400 0.703000
>>> 6 2154435 3.51500 1.89100 1.20300
>>> 7 3593812 6.34400 3.11000 1.98400
>>> 8 5994841 11.5470 5.21800 3.36000
>>> 9 10000000 20.3750 8.67200 5.60900
>>
>>> Windows XP, dual 2.66 GHz, 3 GB RAM, IDL 6.3
>>
>> I'm not sure, but I think that will give you "with replacement".
>
>
> You're right, I wasn't swapping. Corrected, and in the form of
> Allan's method:
>
> function kunsort, array
>   na = n_elements(array)
>   rarray = array
>
>   b = randomu(seed,na-1) * (na - lindgen(na-1) - 1) + lindgen(na-1)
>   for i=0L,na-2 do begin
>   tmp = rarray[i]
>   rarray[i] = rarray[b[i]]
>   rarray[b[i]] = tmp
>   endfor
>
>   return, rarray
> end
>
> With the change, it's slightly slower than Allan's. However, for what
> it's worth, there are claims this is a less biased method. (Again, I
> am no expert. But the recent poker craze seems to have revived
> interest in the probabilities of shuffling.)
>

Yours is doing a bit more than mine was in that it's creating a copy of the original data rather than in-place swapping so that would make yours a bit slower (but probably more useful). You can probably also get a speed up by converting "b" to a long at creation time rather than implicitly ever time you use it.

Looking over what a Knuth Shuffle is supposed to to, it seems that you're only supposed to swap with an element you've not already passed over; my code didn't do this but yours does. I guess Knuth is smarter than me :). Although, I tend to not believe anything which appears on Wikipedia.

However, in your code, it looks like the "na-1" in the creation of "b" and the "na-2" in the loop will mean that the last element of the array never gets swapped.

Thanks,

Allan

Subject: Re: Randomize array order
Posted by Allan Whiteford on Fri, 27 Jul 2007 12:03:34 GMT
View Forum Message <> Reply to Message

Conor wrote:
> Hi everyone!
>
> Anyone know an efficient way to randomize an array (I have a
> sorted array that I want unsorted). Initially, I tried something like
> this:
>
> array = findgen(1000000)
> unsort = array[sort(randomu(seed,1000000))]
>
> It works, but sorting on a million elements is rather slow. Anyone
> know a faster way?
>

Slightly different point and probably a bit academic:

If you have a million elements then you have 1000000! (i.e. one million factorial) different ways to re-order the data. However, your seed is a 4 byte integer which can only take 2^32 different values.

Some messing about suggests that:

1000000! =~ 10^5568636

which means there are ~ 10^5568636 different ways to re-arrange your elements as opposed to the 4 x 10^9 values your seed can take.

Thus, using any of the algorithms suggested you're only going to sample

10^-5568625 %

of the possible values. This is a really small number. It means that no matter how hard you try and how many times you do things you'll never be able to access anything but a tiny number of the possibilities without doing multiple shufflings - I think it's something like 618737 sub-shufflings (i.e. 5568636 / 9) but that could be wrong. However, that requires producing 618737 seeds per major-shuffle (and you can't use a generator based on a 4 byte seed to produce these seeds).

But, since you're only going to be running the code 1000-10,000 times (which is much smaller than 4e9) I guess everything will be ok. I don't know if anyone has studied possible correlations of results as a function of the very small number of seeds (compared to the data), whatever random number generator is used and the shuffling method. Presumably they have and presumably everything is ok. Does anyone know?

Thanks,

Allan

Subject: Re: Randomize array order
Posted by Paolo Grigis on Fri, 27 Jul 2007 13:59:08 GMT
View Forum Message <> Reply to Message

Allan Whiteford wrote:
> Conor wrote:
>> Hi everyone!
>>
>> Anyone know an efficient way to randomize an array (I have a
>> sorted array that I want unsorted). Initially, I tried something like
>> this:
>>
>> array = findgen(1000000)
>> unsort = array[sort(randomu(seed,1000000))]
>>
>> It works, but sorting on a million elements is rather slow. Anyone
>> know a faster way?
>>
>
> Slightly different point and probably a bit academic:
>
> If you have a million elements then you have 1000000! (i.e. one million
> factorial) different ways to re-order the data. However, your seed is a
> 4 byte integer which can only take 2^32 different values.
>

> Some messing about suggests that:
>
> 1000000! =~ 10^5568636
>
> which means there are ~ 10^5568636 different ways to re-arrange your
> elements as opposed to the 4 x 10^9 values your seed can take.
>
> Thus, using any of the algorithms suggested you're only going to sample
>
> 10^-5568625 %
>
> of the possible values. This is a really small number. It means that no
> matter how hard you try and how many times you do things you'll never be
> able to access anything but a tiny number of the possibilities without
> doing multiple shufflings - I think it's something like 618737
> sub-shufflings (i.e. 5568636 / 9) but that could be wrong. However, that
> requires producing 618737 seeds per major-shuffle (and you can't use a
> generator based on a 4 byte seed to produce these seeds).
>
> But, since you're only going to be running the code 1000-10,000 times
> (which is much smaller than 4e9)

I am by no means an expert, but I think that in general, common sense
suggests not to use more random numbers for one project than the cycle
length of the generator (that is, the length at which the numbers start
to repeat themselves). From the docs is hard to guess how long the cycle
is, but it can be at most 2^32 for a generator using long ints. So I
wouldn't suggest doing more than 2^32/1d6 runs of the code, if one is
using 1d6 numbers per run.

If more random deviates are needed, it would be a good idea to use a
random generator with a longer cycle.

Ciao,
Paolo

I guess everything will be ok. I don't
> know if anyone has studied possible correlations of results as a
> function of the very small number of seeds (compared to the data),
> whatever random number generator is used and the shuffling method.
> Presumably they have and presumably everything is ok. Does anyone know?

>
> Thanks,
>
> Allan

Subject: Re: Randomize array order
Posted by David Streutker on Fri, 27 Jul 2007 14:54:55 GMT
View Forum Message <> Reply to Message

On Jul 27, 6:03 am, Allan Whiteford
<allan.rem...@phys.remove.strath.ac.remove.uk> wrote:
> Conor wrote:
>> Hi everyone!
>
>> Anyone know an efficient way to randomize an array (I have a
>> sorted array that I want unsorted). Initially, I tried something like
>> this:
>
>> array = findgen(1000000)
>> unsort = array[sort(randomu(seed,1000000))]
>
>> It works, but sorting on a million elements is rather slow. Anyone
>> know a faster way?
>
> Slightly different point and probably a bit academic:
>
> If you have a million elements then you have 1000000! (i.e. one million
> factorial) different ways to re-order the data. However, your seed is a
> 4 byte integer which can only take 2^32 different values.
>
> Some messing about suggests that:
>
> 1000000! =~ 10^5568636
>
> which means there are ~ 10^5568636 different ways to re-arrange your
> elements as opposed to the 4 x 10^9 values your seed can take.
>
> Thus, using any of the algorithms suggested you're only going to sample
>
> 10^-5568625 %
>
> of the possible values. This is a really small number. It means that no
> matter how hard you try and how many times you do things you'll never be
> able to access anything but a tiny number of the possibilities without
> doing multiple shufflings - I think it's something like 618737
> sub-shufflings (i.e. 5568636 / 9) but that could be wrong. However, that
> requires producing 618737 seeds per major-shuffle (and you can't use a
> generator based on a 4 byte seed to produce these seeds).
>
> But, since you're only going to be running the code 1000-10,000 times
> (which is much smaller than 4e9) I guess everything will be ok. I don't
> know if anyone has studied possible correlations of results as a
> function of the very small number of seeds (compared to the data),
> whatever random number generator is used and the shuffling method.

> Presumably they have and presumably everything is ok. Does anyone know?
>
> Thanks,
>
> Allan

I'm not sure that I agree. Where in any of our algorithms are we unable to access a (theoretically) possible outcome? As long as we are able to randomly select any element of the array in each step, it should work, right? (I.e., as long as the input array has fewer than 2^32 elements.) In your analysis, shouldn't we be using (2^32)^n for the maximum possible number of randomly generated combinations, where n is the number of steps/elements?

Also, in the Knuth method, the final element may or may not be swapped, depending on whether it is randomly selected for one of the previous swaps.

-David

Subject: Re: Randomize array order
Posted by James Kuyper on Fri, 27 Jul 2007 16:05:27 GMT
View Forum Message <> Reply to Message

David Streutker wrote:
> On Jul 27, 6:03 am, Allan Whiteford
...
>> If you have a million elements then you have 1000000! (i.e. one million
>> factorial) different ways to re-order the data. However, your seed is a
>> 4 byte integer which can only take 2^32 different values.
>>
>> Some messing about suggests that:
>>
>> 1000000! =~ 10^5568636
>>
>> which means there are ~ 10^5568636 different ways to re-arrange your
>> elements as opposed to the 4 x 10^9 values your seed can take.
>>
>> Thus, using any of the algorithms suggested you're only going to sample
>>
>> 10^-5568625 %
>>
>> of the possible values. This is a really small number. It means that no
>> matter how hard you try and how many times you do things you'll never be
>> able to access anything but a tiny number of the possibilities without
>> doing multiple shufflings - I think it's something like 618737
>> sub-shufflings (i.e. 5568636 / 9) but that could be wrong. However, that

>> requires producing 618737 seeds per major-shuffle (and you can't use a
>> generator based on a 4 byte seed to produce these seeds).
>>
>> But, since you're only going to be running the code 1000-10,000 times
>> (which is much smaller than 4e9) I guess everything will be ok. I don't
>> know if anyone has studied possible correlations of results as a
>> function of the very small number of seeds (compared to the data),
>> whatever random number generator is used and the shuffling method.
>> Presumably they have and presumably everything is ok. Does anyone know?
>>
>> Thanks,
>>
>> Allan
>
> I'm not sure that I agree. Where in any of our algorithms are we
> unable to access a (theoretically) possible outcome? As long as we
> are able to randomly select any element of the array in each step, it
> should work, right? (I.e., as long as the input array has fewer than
> 2^32 elements.) In your analysis, shouldn't we be using (2^32)^n for
> the maximum possible number of randomly generated combinations, where
> n is the number of steps/elements?

No, because the entire sequence of numbers is uniquely determined by
initial internal state of the generator. If you knew the algorithm
used, and the internal state, that's all the information you'd need to
predict, precisely, the entire sequence of numbers generated, no
matter how long that sequence was. If the internal state is stored in
a 32 bit integer, that means there's only 2^32 possible different
sequences.

> From that fact, it can also be shown that every possible sequence must
start repeating, exactly, with a period that is less than 2^32. If one
of the possible sequences has starts repeating with a period T, then
at least T-1 of the other possible sequences generate that same repeat
cycle, with various shifts.

There's a reason why these things are called PSEUDO-random number
generators.

Subject: Re: Randomize array order
Posted by James Kuyper on Fri, 27 Jul 2007 16:29:49 GMT
View Forum Message <> Reply to Message

Allan Whiteford wrote:
> David Streutker wrote:
>> On Jul 26, 2:32 pm, hradilv <hrad...@yahoo.com> wrote:
>>

>>> On Jul 26, 11:57 am, David Streutker <dstreut...@gmail.com> wrote:
>>>
>>>
>>>
>>>
>>>> How about a Knuth shuffle?
>>>
>>>> (Disclaimer: I'm not a statistician; I just found it on Wikipedia.)
>>>
>>>> function kunsort, array
>>>>   na = n_elements(array)
>>>>   random = randomu(seed,na) * (na - array - 1) + array
>>>>   for i=0L,na-2 do array[i] = array[random[i]]
>>>>   return, array
>>>> end
>>>
>>>> Added to Vince's code (last column):
>>>
>>>>   0   100000   0.0619998   0.0780001   0.0320001
>>>>   1   166810   0.125000   0.125000   0.0780001
>>>>   2   278256   0.282000   0.218000   0.141000
>>>>   3   464158   0.547000   0.406000   0.235000
>>>>   4   774263   1.07800   0.657000   0.390000
>>>>   5   1291549   1.93700   1.09400   0.703000
>>>>   6   2154435   3.51500   1.89100   1.20300
>>>>   7   3593812   6.34400   3.11000   1.98400
>>>>   8   5994841   11.5470   5.21800   3.36000
>>>>   9   10000000   20.3750   8.67200   5.60900
>>>
>>>> Windows XP, dual 2.66 GHz, 3 GB RAM, IDL 6.3
>>>
>>> I'm not sure, but I think that will give you "with replacement".
>>
>>
>> You're right, I wasn't swapping. Corrected, and in the form of
>> Allan's method:
>>
>> function kunsort, array
>>   na = n_elements(array)
>>   rarray = array
>>
>>   b = randomu(seed,na-1) * (na - lindgen(na-1) - 1) + lindgen(na-1)
>>   for i=0L,na-2 do begin
>>   tmp = rarray[i]
>>   rarray[i] = rarray[b[i]]
>>   rarray[b[i]] = tmp
>>   endfor
>>

>> return, rarray
>> end
>>
>> With the change, it's slightly slower than Allan's. However, for what
>> it's worth, there are claims this is a less biased method. (Again, I
>> am no expert. But the recent poker craze seems to have revived
>> interest in the probabilities of shuffling.)
>>
>
> Yours is doing a bit more than mine was in that it's creating a copy of
> the original data rather than in-place swapping so that would make yours
> a bit slower (but probably more useful). You can probably also get a
> speed up by converting "b" to a long at creation time rather than
> implicitly ever time you use it.
>
> Looking over what a Knuth Shuffle is supposed to to, it seems that
> you're only supposed to swap with an element you've not already passed
> over; my code didn't do this but yours does. I guess Knuth is smarter
> than me :). Although, I tend to not believe anything which appears on
> Wikipedia.
>
> However, in your code, it looks like the "na-1" in the creation of "b"
> and the "na-2" in the loop will mean that the last element of the array
> never gets swapped.

As I understand it, I think that neither program correctly implements
Knuth's algorithm. Here's my (minimally tested) attempt. I wrote it as
an in-place shuffle, to save space:

```
PRO knuth_shuffle, array
   na = N_ELEMENTS(array)
   b = LONG((na+2-LINDGEN(na-1))*RANDOMU(seed, na-1))
   FOR i=na-1, 1, -1 DO BEGIN
      temp = array[b[i-1]]
      array[b[i-1]] = array[i]
      array[i] = temp
   ENDFOR
END
```

Subject: Re: Randomize array order
Posted by Conor on Fri, 27 Jul 2007 16:44:53 GMT
View Forum Message <> Reply to Message

On Jul 27, 12:05 pm, kuyper <kuy...@wizard.net> wrote:
> David Streutker wrote:
>> On Jul 27, 6:03 am, Allan Whiteford
> ...

>>> If you have a million elements then you have 1000000! (i.e. one million
>>> factorial) different ways to re-order the data. However, your seed is a
>>> 4 byte integer which can only take 2^32 different values.
>
>>> Some messing about suggests that:
>
>>> 1000000! =~ 10^5568636
>
>>> which means there are ~ 10^5568636 different ways to re-arrange your
>>> elements as opposed to the 4 x 10^9 values your seed can take.
>
>>> Thus, using any of the algorithms suggested you're only going to sample
>
>>> 10^-5568625 %
>
>>> of the possible values. This is a really small number. It means that no
>>> matter how hard you try and how many times you do things you'll never be
>>> able to access anything but a tiny number of the possibilities without
>>> doing multiple shufflings - I think it's something like 618737
>>> sub-shufflings (i.e. 5568636 / 9) but that could be wrong. However, that
>>> requires producing 618737 seeds per major-shuffle (and you can't use a
>>> generator based on a 4 byte seed to produce these seeds).
>
>>> But, since you're only going to be running the code 1000-10,000 times
>>> (which is much smaller than 4e9) I guess everything will be ok. I don't
>>> know if anyone has studied possible correlations of results as a
>>> function of the very small number of seeds (compared to the data),
>>> whatever random number generator is used and the shuffling method.
>>> Presumably they have and presumably everything is ok. Does anyone know?
>
>>> Thanks,
>
>>> Allan
>
>> I'm not sure that I agree. Where in any of our algorithms are we
>> unable to access a (theoretically) possible outcome? As long as we
>> are able to randomly select any element of the array in each step, it
>> should work, right? (I.e., as long as the input array has fewer than
>> 2^32 elements.) In your analysis, shouldn't we be using (2^32)^n for
>> the maximum possible number of randomly generated combinations, where
>> n is the number of steps/elements?
>
> No, because the entire sequence of numbers is uniquely determined by
> initial internal state of the generator. If you knew the algorithm
> used, and the internal state, that's all the information you'd need to
> predict, precisely, the entire sequence of numbers generated, no
> matter how long that sequence was. If the internal state is stored in
> a 32 bit integer, that means there's only 2^32 possible different

> sequences.
>
>> From that fact, it can also be shown that every possible sequence must
>
> start repeating, exactly, with a period that is less than 2^32. If one
> of the possible sequences has starts repeating with a period T, then
> at least T-1 of the other possible sequences generate that same repeat
> cycle, with various shifts.
>
> There's a reason why these things are called PSEUDO-random number
> generators.

It shouldn't really be a problem for me, fortunately. I'm running
this a couple thousand times, but everytime it is on a different set
of values. The only thing I would have to worry about is it repeating
within one set of values, which won't happen for 1,000,000 elements.

Of course, worse comes to worse there's always a true random number
generator:

www.random.org

Subject: Re: Randomize array order
Posted by David Streutker on Fri, 27 Jul 2007 17:54:07 GMT
View Forum Message <> Reply to Message

On Jul 27, 10:05 am, kuyper <kuy...@wizard.net> wrote:
> David Streutker wrote:
>> On Jul 27, 6:03 am, Allan Whiteford
> ...
>>> If you have a million elements then you have 1000000! (i.e. one million
>>> factorial) different ways to re-order the data. However, your seed is a
>>> 4 byte integer which can only take 2^32 different values.
>
>>> Some messing about suggests that:
>
>>> 1000000! =~ 10^5568636
>
>>> which means there are ~ 10^5568636 different ways to re-arrange your
>>> elements as opposed to the 4 x 10^9 values your seed can take.
>
>>> Thus, using any of the algorithms suggested you're only going to sample
>
>>> 10^-5568625 %
>
>>> of the possible values. This is a really small number. It means that no
>>> matter how hard you try and how many times you do things you'll never be

>>> able to access anything but a tiny number of the possibilities without
>>> doing multiple shufflings - I think it's something like 618737
>>> sub-shufflings (i.e. 5568636 / 9) but that could be wrong. However, that
>>> requires producing 618737 seeds per major-shuffle (and you can't use a
>>> generator based on a 4 byte seed to produce these seeds).
>
>>> But, since you're only going to be running the code 1000-10,000 times
>>> (which is much smaller than 4e9) I guess everything will be ok. I don't
>>> know if anyone has studied possible correlations of results as a
>>> function of the very small number of seeds (compared to the data),
>>> whatever random number generator is used and the shuffling method.
>>> Presumably they have and presumably everything is ok. Does anyone know?
>
>>> Thanks,
>
>>> Allan
>
>> I'm not sure that I agree. Where in any of our algorithms are we
>> unable to access a (theoretically) possible outcome? As long as we
>> are able to randomly select any element of the array in each step, it
>> should work, right? (I.e., as long as the input array has fewer than
>> 2^32 elements.) In your analysis, shouldn't we be using (2^32)^n for
>> the maximum possible number of randomly generated combinations, where
>> n is the number of steps/elements?
>
> No, because the entire sequence of numbers is uniquely determined by
> initial internal state of the generator. If you knew the algorithm
> used, and the internal state, that's all the information you'd need to
> predict, precisely, the entire sequence of numbers generated, no
> matter how long that sequence was. If the internal state is stored in
> a 32 bit integer, that means there's only 2^32 possible different
> sequences.
>
>> From that fact, it can also be shown that every possible sequence must
>
> start repeating, exactly, with a period that is less than 2^32. If one
> of the possible sequences has starts repeating with a period T, then
> at least T-1 of the other possible sequences generate that same repeat
> cycle, with various shifts.
>
> There's a reason why these things are called PSEUDO-random number
> generators.

Interesting. I hadn't really thought it through before.

If there are only 2^32 possible sequences, then why is the internal
state characterized by a 36-element array?

IDL> test = randomu(seed)
IDL> help, seed
SEED LONG = Array[36]

Is it that there are only 2^32 possible sequences available during any
given session? With a new set being available in a different session?

Subject: Re: Randomize array order
Posted by Conor on Fri, 27 Jul 2007 18:09:20 GMT
View Forum Message <> Reply to Message

On Jul 27, 1:54 pm, David Streutker <dstreut...@gmail.com> wrote:
> On Jul 27, 10:05 am, kuyper <kuy...@wizard.net> wrote:
>
>
>
>> David Streutker wrote:
>>> On Jul 27, 6:03 am, Allan Whiteford
>> ...
>>>> If you have a million elements then you have 1000000! (i.e. one million
>>>> factorial) different ways to re-order the data. However, your seed is a
>>>> 4 byte integer which can only take 2^32 different values.
>
>>>> Some messing about suggests that:
>
>>>> 1000000! =~ 10^5568636
>
>>>> which means there are ~ 10^5568636 different ways to re-arrange your
>>>> elements as opposed to the 4 x 10^9 values your seed can take.
>
>>>> Thus, using any of the algorithms suggested you're only going to sample
>
>>>> 10^-5568625 %
>
>>>> of the possible values. This is a really small number. It means that no
>>>> matter how hard you try and how many times you do things you'll never be
>>>> able to access anything but a tiny number of the possibilities without
>>>> doing multiple shufflings - I think it's something like 618737
>>>> sub-shufflings (i.e. 5568636 / 9) but that could be wrong. However, that
>>>> requires producing 618737 seeds per major-shuffle (and you can't use a
>>>> generator based on a 4 byte seed to produce these seeds).
>
>>>> But, since you're only going to be running the code 1000-10,000 times
>>>> (which is much smaller than 4e9) I guess everything will be ok. I don't
>>>> know if anyone has studied possible correlations of results as a
>>>> function of the very small number of seeds (compared to the data),
>>>> whatever random number generator is used and the shuffling method.

>>>> Presumably they have and presumably everything is ok. Does anyone know?
>
>>>> Thanks,
>
>>>> Allan
>
>>> I'm not sure that I agree. Where in any of our algorithms are we
>>> unable to access a (theoretically) possible outcome? As long as we
>>> are able to randomly select any element of the array in each step, it
>>> should work, right? (I.e., as long as the input array has fewer than
>>> 2^32 elements.) In your analysis, shouldn't we be using (2^32)^n for
>>> the maximum possible number of randomly generated combinations, where
>>> n is the number of steps/elements?
>
>> No, because the entire sequence of numbers is uniquely determined by
>> initial internal state of the generator. If you knew the algorithm
>> used, and the internal state, that's all the information you'd need to
>> predict, precisely, the entire sequence of numbers generated, no
>> matter how long that sequence was. If the internal state is stored in
>> a 32 bit integer, that means there's only 2^32 possible different
>> sequences.
>
>>> From that fact, it can also be shown that every possible sequence must
>
>> start repeating, exactly, with a period that is less than 2^32. If one
>> of the possible sequences has starts repeating with a period T, then
>> at least T-1 of the other possible sequences generate that same repeat
>> cycle, with various shifts.
>
>> There's a reason why these things are called PSEUDO-random number
>> generators.
>
> Interesting. I hadn't really thought it through before.
>
> If there are only 2^32 possible sequences, then why is the internal
> state characterized by a 36-element array?
>
> IDL> test = randomu(seed)
> IDL> help, seed
> SEED   LONG   = Array[36]
>
> Is it that there are only 2^32 possible sequences available during any
> given session? With a new set being available in a different session?

That is a very interesting question. According to the online-manual:

The random number generator is taken from: "Random Number Generators:
Good Ones are Hard to Find", Park and Miller, Communications of the

ACM, Oct 1988, Vol 31, No. 10, p. 1192. To remove low-order serial correlations, a Bays-Durham shuffle is added, resulting in a random number generator similar to ran1() in Section 7.1 of Numerical Recipes in C: The Art of Scientific Computing (Second Edition), published by Cambridge University Press.

Hmm... It turns out that randomn is completely useless. It claims to use the Box-Muller method, which I happen to know is a simple variation on a regular random number generator, but with half the possible sequences. Therefore, it has: (2^32)/2 sequences = 1^32 sequences = 1 It repeats after generating only 1 random number!!! Yikes!!!! Someone should alert RSI!!!

(okay, okay, it was a bad joke. So sue me.) Anyway, back to reality. I wonder if RSI uses an array of size 36 to institute a "virtual" increase of variable size, allowing for more precise calculations??? Is such a thing possible? I don't know why else they would need an array to hold their seed, although I'm going to guess it is for another reason.

Subject: Re: Randomize array order
Posted by James Kuyper on Fri, 27 Jul 2007 21:29:17 GMT
View Forum Message <> Reply to Message

David Streutker wrote:
...
> If there are only 2^32 possible sequences, then why is the internal
> state characterized by a 36-element array?
>
> IDL> test = randomu(seed)
> IDL> help, seed
> SEED LONG = Array[36]

That is a VERY good question. I only looked at the documentation; I didn't bother to check. The documentation says:

> The state of the random number generator is contained in a long integer vector.

and also

> If the Seed argument is:
> * an undefined variable - the generic state is used and the resulting generic state array is stored in the variable.

Given that the resulting generic state appears to be a 36-element array, I don't think both statements can be true. An array of 36 longs

would be more than enough to generate all possible shuffles of a 52-card deck. More importantly, it's large enough to do large numerical simulations in quantum field theory without introducing spurious correlations. Therefore I hope that the second statement is the one that's correct.

Subject: Re: Randomize array order
Posted by Michael Galloy on Fri, 27 Jul 2007 22:35:13 GMT
View Forum Message <> Reply to Message

On Jul 27, 3:29 pm, kuyper <kuy...@wizard.net> wrote:
> The documentation says:
>
>> The state of the random number generator is contained in a long integer vector.
>
> and also
>
>> If the Seed argument is:
>> * an undefined variable - the generic state is used and the resulting generic state array is stored in the variable.
>
> Given that the resulting generic state appears to be a 36-element
> array, I don't think both statements can be true.

I think those statements are consistent. The state is stored in a long integer vector (a 36-element long integer array to be specific).

Mike
--
www.michaelgalloy.com
