

Best way to randomize an array with .NET

Asked 16 years, 3 months ago Modified 7 months ago

Viewed 228k times



198

What is the best way to randomize an array of strings with .NET? My array contains about 500 strings and I'd like to create a new `Array` with the same strings but in a random order.



Please include a C# example in your answer.



c#

.net

algorithm

sorting

random

Share

Improve this question

Follow

edited Oct 24, 2015 at 16:19



[Ruben Bartelink](#)

61.8k ● 31 ● 194 ● 260

asked Sep 20, 2008 at 17:33



[Mats](#)

14.8k ● 33 ● 82 ● 113

1 Duplicate of [Randomize a List<T>](#) – [Herohtar](#) Aug 9, 2020 at 5:12

**283**

The following implementation uses the [Fisher-Yates algorithm](#) AKA the Knuth Shuffle. It runs in $O(n)$ time and shuffles in place, so is better performing than the 'sort by random' technique, although it is more lines of code. See [here](#) for some comparative performance measurements. I have used `System.Random`, which is fine for non-cryptographic purposes.*

```
static class RandomExtensions
{
    public static void Shuffle<T> (this Random rng, T[]
    {
        int n = array.Length;
        while (n > 1)
        {
            int k = rng.Next(n--);
            T temp = array[n];
            array[n] = array[k];
            array[k] = temp;
        }
    }
}
```

Usage:

```
var array = new int[] {1, 2, 3, 4};
var rng = new Random();
rng.Shuffle(array);
rng.Shuffle(array); // different order from first call
```

* For longer arrays, in order to make the (extremely large) number of permutations equally probable it would be

necessary to run a pseudo-random number generator (PRNG) through many iterations for each swap to produce enough entropy. For a 500-element array only a very small fraction of the possible 500! permutations will be possible to obtain using a PRNG. Nevertheless, the Fisher-Yates algorithm is unbiased and therefore the shuffle will be as good as the RNG you use.

Share Improve this answer

edited Jun 26, 2019 at 12:34


Follow

community wiki

15 revs, 3 users 98%

Matt Howells

7 Wouldn't it be better to change the parameters and make the usage like `array.Shuffle(new Random());` .. ?
– Ken Kin May 10, 2019 at 17:13

3 You can simplify the swap using Tuples as of framework 4.0 -
> `(array[n], array[k]) = (array[k], array[n]);` – dynamichael Jun 17, 2019 at 0:31 

2 @Ken Kin: No, this would be bad. The reason is that `new Random()` is initialized with a seed value based on the current system time, which only updates every ~16ms.
– Matt Howells Jun 26, 2019 at 11:20

In some quick tests of this vs the list removeAt solution, there is a small difference at 999 elements. The difference gets drastic at 99999 random ints, with this solution at 3ms and the other at 1810ms. – galamdring Nov 30, 2019 at 12:04

great solution I had used for years. BTW, just found out, in .NET 8, Shuffle has become a built-in function with similar

implementation. Thus the extension is hidden, thus better removed. – ZZZ Aug 4 at 22:18



First, you should use the most upvoted answer to this question so far, which is

218

<https://stackoverflow.com/a/110570/1757994>.



You can shuffle using LINQ.



```
var myShuffledArray = myArray.Shuffle().
```



Add the following code as a file called

`EnumerableExtensions.cs` to your project:



```
public static class EnumerableExtensions
{
    public static IList<T> Shuffle<T>(this IEnumerable
    {
        return sequence.Shuffle(new Random());
    }

    public static IList<T> Shuffle<T>(this IEnumerable
    randomNumberGenerator)
    {
        if (sequence == null)
        {
            throw new ArgumentNullException("sequence")
        }

        if (randomNumberGenerator == null)
        {
            throw new ArgumentNullException("randomNum

        T swapTemp;
        List<T> values = sequence.ToList();
        int currentlySelecting = values.Count;
```

```

        while (currentlySelecting > 1)
        {
            int selectedElement =
randomNumberGenerator.Next(currentlySelecting);
            --currentlySelecting;
            if (currentlySelecting != selectedElement)
            {
                swapTemp = values[currentlySelecting];
                values[currentlySelecting] = values[selectedElement];
                values[selectedElement] = swapTemp;
            }
        }

        return values;
    }
}

```

Adapted from <https://github.com/microsoft/driver-utilities/blob/master/src/Sarif.Driver/EnumerableExtensionS.cs>

It is a bad idea to use informally authored shuffle algorithms, since they are hard to analyze for flaws.

If you're using C# 7.0 or higher, this other approach is slow for many elements but works correctly in all environments C# is used:

```

var rnd = new Random();
var myRandomArray = myArray
    .Select(x => (x, rnd.Next()))
    .OrderBy(tuple => tuple.Item2)
    .Select(tuple => tuple.Item1)
    .ToArray();

```

This is almost as verbose as the Fisher-Yates shuffle, which is much faster for large numbers of elements. This

creates random numbers for each element, then sorts by the random number.

This answer used to say something along the lines of

```
.OrderBy(x => random()).
```

Experienced developers will correctly know that

`.OrderBy(x => random())` alone is wrong. So if you commit code that contains it, you will look bad.

Why is it wrong? `OrderBy` expects the `keySelector`, the function that it is passed, to be pure. That means the `keySelector` should return the same answer every time it is called and cause no side effects.

C# implementations like Unity's do not cache the result of the selector passed to `OrderBy`. Games tend to shuffle things. It is hard to generalize across all CLR implementations, so it's better to just use the correct thing. In order to not look bad with code you author, you should use Fisher-Yates instead.

For Visual Basic and versions of C# earlier than 7.0, use Fisher-Yates. A correct implementation using only LINQ is more verbose than implementing Fisher-Yates, so you will have written more code to do a worse implementation.

Share Improve this answer

Follow

edited Aug 18, 2023 at 17:04



DoctorPangloss

3,073 ● 1 ● 20 ● 23

answered Sep 20, 2008 at 17:40



mdb

52.8k ● 11 ● 66 ● 62

-
- 1 two notes: 1) `System.Random` is not thread-safe (you've been warned) and 2) `System.Random` is time based, so if you use this code in a heavily concurrent system, it's possible for two requests to get the same value (i.e. in webapps) – [therealhoff](#) Sep 20, 2008 at 21:37
-
- 3 Just to clarify the above, `System.Random` will seed itself using the current time, so two instances created simultaneously will generate the same "random" sequence..`System.Random` should only be used in toy apps – [therealhoff](#) Sep 20, 2008 at 21:41
-
- 11 Also this algorithm is $O(n \log n)$ and biased by the Qsort algorithm. See my answer for an $O(n)$ unbiased solution. – [Matt Howells](#) Sep 21, 2008 at 9:02
-
- 11 Unless `OrderBy` caches the sort keys internally, this also has the problem of violating the transitive property of ordered comparisons. If there is ever a debug mode verification that `OrderBy` produced correct results, then in theory it could throw an exception. – [Sam Harwell](#) Oct 16, 2009 at 16:22
-
- 7 See this: blogs.msdn.com/b/ericlippert/archive/2011/01/31/... and en.wikipedia.org/wiki/... – [Gregor Slavec](#) Feb 1, 2011 at 15:11 ✎
-



You're looking for a shuffling algorithm, right?

20



Okay, there are two ways to do this: the clever-but-people-always-seem-to-misunderstand-it-and-get-it-wrong-so-maybe-its-not-that-clever-after-all way, and the dumb-as-rocks-but-who-cares-because-it-works way.



Dumb way



- Create a duplicate of your first array, but tag each string should with a random number.
- Sort the duplicate array with respect to the random number.

This algorithm works well, but make sure that your random number generator is unlikely to tag two strings with the same number. Because of the so-called [Birthday Paradox](#), this happens more often than you might expect. Its time complexity is $O(n \log n)$.

Clever way

I'll describe this as a recursive algorithm:

To shuffle an array of size n (indices in the range $[0..n-1]$):

if $n = 0$

- do nothing

if $n > 0$

- (*recursive step*) shuffle the first $n-1$ elements of the array
- choose a random index, x , in the range $[0..n-1]$

- swap the element at index $n-1$ with the element at index x

The iterative equivalent is to walk an iterator through the array, swapping with random elements as you go along, but notice that you cannot swap with an element **after** the one that the iterator points to. This is a very common mistake, and leads to a biased shuffle.

Time complexity is $O(n)$.

Share Improve this answer

edited Sep 20, 2008 at 18:04

Follow

answered Sep 20, 2008 at 17:54



Pitarou

2,239 ● 18 ● 25



8



This algorithm is simple but not efficient, $O(N^2)$. All the "order by" algorithms are typically $O(N \log N)$. It probably doesn't make a difference below hundreds of thousands of elements but it would for large lists.

```
var stringlist = ... // add your values to stringlist

var r = new Random();

var res = new List<string>(stringlist.Count);

while (stringlist.Count > 0)
{
    var i = r.Next(stringlist.Count);
    res.Add(stringlist[i]);
}
```

```
stringlist.RemoveAt(i);  
}
```

The reason why it's $O(N^2)$ is subtle: [List.RemoveAt\(\)](#) is a $O(N)$ operation unless you remove in order from the end.

Share Improve this answer

edited Mar 16, 2017 at 12:42

Follow

answered Sep 20, 2008 at 17:39



Sklivvz

31.1k ● 24 ● 118 ● 174

3 This has the same effect as a knuth shuffle, but it's not as efficient, since it involves depopulating one list and repopulating another. Swapping items in place would be a better solution. – [Nick Johnson](#) Sep 20, 2008 at 19:06

2 I find this elegant and easily understandable and on 500 strings it doesn't make a bit of a difference... – [Sklivvz](#) Sep 20, 2008 at 19:41



[New simplified approach in .Net 8](#) is the `Random Shuffle method:

6



Without creating a duplicate array (will shuffle the array in place):



```
string[] employees = ["User 1", "User 2", "User 3", "U  
Random.Shared.Shuffle(employees);
```

If you definitely want a duplicate array, you can copy the array first before shuffling it:

```
string[] employees = ["User 1", "User 2", "User 3", "U  
var duplicateOfEmployees = employees.ToArray();  
Random.Shared.Shuffle(duplicateOfEmployees);
```

Share Improve this answer

answered Dec 5, 2023 at 14:55

Follow



Ryan Gaudion

925 ● 1 ● 12 ● 29



2



Just thinking off the top of my head, you could do this:

```
public string[] Randomize(string[] input)
{
    List<string> inputList = input.ToList();
    string[] output = new string[input.Length];
    Random randomizer = new Random();
    int i = 0;

    while (inputList.Count > 0)
    {
        int index = r.Next(inputList.Count);
        output[i++] = inputList[index];
        inputList.RemoveAt(index);
    }

    return (output);
}
```

Share Improve this answer

answered Sep 20, 2008 at 17:42

Follow



user19357



1



Randomizing the array is intensive as you have to shift around a bunch of strings. Why not just randomly read from the array? In the worst case you could even create a wrapper class with a getNextString(). If you really do need to create a random array then you could do something like

```
for i = 0 -> i= array.length * 5
    swap two strings in random places
```

The *5 is arbitrary.

Share Improve this answer

answered Sep 20, 2008 at 17:38

Follow



[stimms](#)

44k ● 31 ● 101 ● 151

-
- 1 A random read from the array is likely to hit some items multiple times and miss others! – [Ray Hayes](#) Sep 20, 2008 at 17:41
-
- 1 The shuffle algorithm is broken. You would have to make your arbitrary 5 very high indeed before your shuffle is unbiased. – [Pitarou](#) Sep 20, 2008 at 18:12
-
- 1 Make a Array of the (integer) indexes. Shuffle the indexes. Just use the indexes in that random order. No duplicates, no shuffling around of string references in memory (wich may each trigger interning and what not). – [Christopher](#) Aug 27, 2017 at 5:36
-



0

Generate an array of random floats or ints of the same length. Sort that array, and do corresponding swaps on your target array.



This yields a truly independent sort.



Share Improve this answer

answered Sep 20, 2008 at 17:37



Follow



[Nick](#)

13.4k ● 17 ● 66 ● 100



0

```
Random r = new Random();  
List<string> list = new List(originalArray);  
List<string> randomStrings = new List();
```



```
while(list.Count > 0)
{
    int i = r.Random(list.Count);
    randomStrings.Add(list[i]);
    list.RemoveAt(i);
}
```

Share Improve this answer

answered Sep 20, 2008 at 17:43

Follow



nullDev

11.5k ● 8 ● 37 ● 53



0



Jacco, your solution using a custom `IComparer` isn't safe. The Sort routines require the comparer to conform to several requirements in order to function properly. First among them is consistency. If the comparer is called on the same pair of objects, it must always return the same result. (the comparison must also be transitive).

Failure to meet these requirements can cause any number of problems in the sorting routine including the possibility of an infinite loop.

Regarding the solutions that associate a random numeric value with each entry and then sort by that value, these lead to an inherent bias in the output because any time two entries are assigned the same numeric value, the randomness of the output will be compromised. (In a "stable" sort routine, whichever is first in the input will be first in the output. `Array.Sort` doesn't happen to be stable, but there is still a bias based on the partitioning done by the Quicksort algorithm).

You need to do some thinking about what level of randomness you require. If you are running a poker site where you need cryptographic levels of randomness to protect against a determined attacker you have very different requirements from someone who just wants to randomize a song playlist.

For song-list shuffling, there's no problem using a seeded PRNG (like `System.Random`). For a poker site, it's not even an option and you need to think about the problem a lot harder than anyone is going to do for you on stackoverflow. (using a cryptographic RNG is only the beginning, you need to ensure that your algorithm doesn't introduce a bias, that you have sufficient sources of entropy, and that you don't expose any internal state that would compromise subsequent randomness).

Share Improve this answer

answered Sep 21, 2008 at 13:36

Follow



Andrew



0



This post has already been pretty well answered - use a Durstenfeld implementation of the Fisher-Yates shuffle for a fast and unbiased result. There have even been some implementations posted, though I note some are actually incorrect.



I wrote a couple of posts a while back about [implementing full and partial shuffles using this technique](#), and (this second link is where I'm hoping to add value) also [a follow-up post about how to check whether your](#)

[implementation is unbiased](#), which can be used to check any shuffle algorithm. You can see at the end of the second post the effect of a simple mistake in the random number selection can make.

Share Improve this answer

answered Sep 21, 2008 at 17:37

Follow



Greg Beech

136k ● 45 ● 209 ● 250

1 Your links are still broken :/ – [Wai Ha Lee](#) Jan 4, 2019 at 20:36



0

Ok, this is clearly a bump from my side (apologizes...), but I often use a quite general and cryptographically strong method.



```
public static class EnumerableExtensions
{
    static readonly RNGCryptoServiceProvider RngCrypto
    RNGCryptoServiceProvider();
    public static IEnumerable<T> Shuffle<T>(this IEnum
    {
        var randomIntegerBuffer = new byte[4];
        Func<int> rand = () =>
        {
            RngCryptoServiceProvider.GetBytes(randomIntegerBuffer)
                return
            BitConverter.ToInt32(randomIntegerBuffer, 0);
        };
        return from item in enumerable
            let rec = new {item, rnd = rand()}
            orderby rec.rnd
            select rec.item;
```



```
}  
}
```

Shuffle() is an extension on any IEnumerable so getting, say, numbers from 0 to 1000 in random order in a list can be done with

```
Enumerable.Range(0, 1000).Shuffle().ToList()
```

This method also won't give any surprises when it comes to sorting, since the sort value is generated and remembered exactly once per element in the sequence.

Share Improve this answer

answered Oct 24, 2012 at 15:37

Follow



[jlarsson](#)

154 ● 3

This is biased (so unsuitable for generic cryptographic purposes, and potentially unsuitable for all purposes that need fair randomness with sufficiently large (but not that large – see birthday problem) numbers of elements) because of collisions. You should use Fisher–Yates with a CSPRNG.

– [Ry-](#) ♦ Oct 12 at 4:12



You don't need complicated algorithms.

0

Just one simple line:



```
Random random = new Random();  
array.ToList().Sort((x, y) => random.Next(-1, 1)).ToAr
```





Note that we need to convert the `Array` to a `List` first, if you don't use `List` in the first place.

Also, mind that this is not efficient for very large arrays! Otherwise it's clean & simple.

Share Improve this answer

answered Apr 13, 2015 at 20:43

Follow



[bytecode77](#)

14.8k ● 31 ● 116 ● 148

-
- 1 Error: Operator '.' cannot be applied to operand of type 'void'
– [usefulBee](#) Apr 13, 2016 at 15:33
-

This is clean, simple, and wrong. People were already explaining on this question that you can't pass a non-transitive comparison function back in 2008. – [Ry-](#) ♦ Oct 12 at 4:14



0



```
int[] numbers = {0,1,2,3,4,5,6,7,8,9};
List<int> numList = new List<int>();
numList.AddRange(numbers);

Console.WriteLine("Original Order");
for (int i = 0; i < numList.Count; i++)
{
    Console.Write(String.Format("{0} ", numList[i]));
}

Random random = new Random();
Console.WriteLine("\n\nRandom Order");
for (int i = 0; i < numList.Capacity; i++)
{
    int randomIndex = random.Next(numList.Count);
    Console.Write(String.Format("{0} ", numList[randomIndex]));
    numList.RemoveAt(randomIndex);
}
```

```
}  
Console.ReadLine();
```

Share Improve this answer

answered Feb 6, 2017 at 18:01

Follow



[Nitish Katare](#)

1,165 ● 3 ● 12 ● 22



Could be:

0



```
Random random = new();  
  
string RandomWord()  
{  
    const string CHARS = "abcdefghijklmnopqrstuvwxyz";  
    int n = random.Next(CHARS.Length);  
    return string.Join("", CHARS.OrderBy(x => random.N  
}
```

Share Improve this answer

answered Dec 2, 2021 at 16:43

Follow



[Sith2021](#)

3,706 ● 1 ● 34 ● 23



Although Random's Next() method is much faster, you can also use this technique:

0



```
string[] myArray = <insert code that defines your arra  
var newArray = myArray.OrderBy(x => Guid.NewGuid()).To
```

There might be some odd reason why you'd use a GUID rather than use an Int for randomization... I think most

practical purposes would rule out this solution, but added since it's a technically valid approach.

Share Improve this answer

answered Jul 28, 2023 at 15:01

Follow



CokoBWare

474 ● 3 ● 13



0



This is a generic solution for IList to IEnumerable shuffle extension:

```
public static class ListShuffler
{
    public static IEnumerable<T> Shuffle<T>(this IList
    {
        var indexes = Enumerable.Range(0, list.Count).
        while (indexes.Count > 1)
        {
            int pos = random.Next(indexes.Count);
            yield return list[indexes[pos]];
            indexes.RemoveAt(pos);
        }
        yield return list[indexes[0]];
    }
}
```

Share Improve this answer

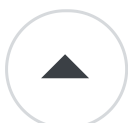
answered May 6 at 9:20

Follow



altair

121 ● 1 ● 6



-1

Here's a simple way using LINQ:

```
// Input array
List<String> lst = new List<string>();
for (int i = 0; i < 500; i += 1) lst.Add(i.ToString())
```



```
// Output array
List<String> lstRandom = new List<string>();

// Randomize
Random rnd = new Random();
lstRandom.AddRange(from s in lst orderby rnd.Next(100)
```

Share Improve this answer

answered Sep 20, 2008 at 19:03

Follow



Seth Morris

45 ● 3

Why 100? The general approach is already not a great idea, but an explicit example where 500 elements are assigned only 100 distinct values is just weird. – Ry- ♦ Oct 12 at 4:18



-1



```
private ArrayList ShuffleArrayList(ArrayList source)
{
    ArrayList sortedList = new ArrayList();
    Random generator = new Random();

    while (source.Count > 0)
    {
        int position = generator.Next(source.Count);
        sortedList.Add(source[position]);
        source.RemoveAt(position);
    }
    return sortedList;
}
```

Share Improve this answer

edited Oct 11, 2015 at 19:18

Follow



Rizier123

59.6k ● 17 ● 104 ● 164

answered May 1, 2012 at 12:00

Himalaya Garg

1,589 • 18 • 23

To me it feels like you could increase both efficiency and readability by instead of trying to shuffle an Array by declaring a second Array, you better try to convert to a List, Shuffle and back to an Array: `sortedList = source.ToList().OrderBy(x => generator.Next()).ToArray();` – T_D Feb 21, 2016 at 11:43



-1



This is a complete working Console solution based on [the example provided in here](#):

```
class Program
{
    static string[] words1 = new string[] { "brown", "quick" };

    static void Main()
    {
        var result = Shuffle(words1);
        foreach (var i in result)
        {
            Console.Write(i + " ");
        }
        Console.ReadKey();
    }

    static string[] Shuffle(string[] wordArray) {
        Random random = new Random();
        for (int i = wordArray.Length - 1; i > 0; i--)
        {
            int swapIndex = random.Next(i + 1);
            string temp = wordArray[i];
            wordArray[i] = wordArray[swapIndex];
            wordArray[swapIndex] = temp;
        }
    }
}
```

```
        return wordArray;  
    }  
}
```

Share Improve this answer

answered Apr 13, 2016 at 16:45

Follow



usefulBee

9,670 ● 10 ● 52 ● 95



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.