384. Shuffle an Array

Nov. 8, 2017 | 100.7K views

Average Rating: 4.74 (42 votes)

Shuffle a set of numbers without duplicates.

Example:

```
// Init an array with set 1, 2, and 3.
int[] nums = {1,2,3};
Solution solution = new Solution(nums);

// Shuffle the array [1,2,3] and return its result. Any permutation of [1,2,3] must eq
solution.shuffle();

// Resets the array back to its original configuration [1,2,3].
solution.reset();

// Returns the random shuffling of array [1,2,3].
solution.shuffle();
```

Initial Thoughts

Normally I would display more than two approaches, but shuffling is deceptively easy to do *almost* properly, and the Fisher-Yates algorithm is both the canonical solution and asymptotically optimal.

A few notes on randomness are necessary before beginning - both approaches displayed below assume that the languages' pseudorandom number generators (PRNGs) are sufficiently random. The sample code uses the simplest techniques available for getting pseudorandom numbers, but for each possible permutation of the array to be truly equally likely, more care must be taken. For example, an array of length n has $n!$ distinct permutations. Therefore, in order to encode all permutations in an integer space, $\lceil lg(n!) \rceil$ bits are necessary, which may not be guaranteed by the default PRNG.

Approach #1 Brute Force [Accepted]

Intuition

If we put each number in a "hat" and draw them out at random, the order in which we draw them will define a random ordering.

Algorithm

The brute force algorithm essentially puts each number in the aforementioned "hat", and draws them at random (without replacement) until there are none left. Mechanically, this is performed by copying the contents of `array` into a second auxiliary array named `aux` before overwriting each element of `array` with a randomly selected one from `aux`. After selecting each random element, it is removed from `aux` to prevent duplicate draws. The implementation of `reset` is simple, as we just store the original state of `nums` on construction.

The correctness of the algorithm follows from the fact that an element (without loss of generality) is equally likely to be selected during all iterations of the `for` loop. To prove this, observe that the probability of a particular element e being chosen on the kth iteration (indexed from 0) is simply $P(e \text{ being chosen during the } k\text{th iteration}) \cdot P(e \text{ not being chosen before the } k\text{th iteration})$. Given that the array to be shuffled has n elements, this probability is more concretely stated as the following:

$$\frac{1}{n-k} \cdot \prod_{i=1}^{k} \frac{n-i}{n-i+1}$$

When expanded (and rearranged), it looks like this (for sufficiently large k):

$$(\frac{n-1}{n} \cdot \frac{n-2}{n-1} \cdot (\ldots) \cdot \frac{n-k+1}{n-k+2} \cdot \frac{n-k}{n-k+1}) \cdot \frac{1}{n-k}$$

For the base case ($k = 0$), it is trivial to see that $\frac{1}{n-k} = \frac{1}{n}$. For $k > 0$, the numerator of each fraction can be cancelled with the denominator of the next, leaving the n from the 0th draw as the only uncancelled denominator. Therefore, no matter on which draw an element is drawn, it is drawn with a $\frac{1}{n}$ chance, so each array permutation is equally likely to arise.

Java | Python — Copy

```
class Solution:
    def __init__(self, nums):
        self.array = nums
        self.original = list(nums)

    def reset(self):
        self.array = self.original
        self.original = list(self.original)
        return self.array

    def shuffle(self):
        aux = list(self.array)

        for idx in range(len(self.array)):
            remove_idx = random.randrange(len(aux))
            self.array[idx] = aux.pop(remove_idx)

        return self.array
```

Complexity Analysis

- Time complexity : $\mathcal{O}(n^2)$.

  The quadratic time complexity arises from the calls to `list.remove` (or `list.pop`), which run in linear time. n linear list removals occur, which results in a fairly easy quadratic analysis.

- Space complexity : $\mathcal{O}(n)$.

  Because the problem also asks us to implement `reset`, we must use linear additional space to store the original array. Otherwise, it would be lost upon the first call to `shuffle`.

Approach #2 Fisher-Yates Algorithm [Accepted]

Intuition

We can cut down the time and space complexities of `shuffle` with a bit of cleverness - namely, by swapping elements around within the array itself, we can avoid the linear space cost of the auxiliary array and the linear time cost of list modification.

Algorithm

The Fisher-Yates algorithm is remarkably similar to the brute force solution. On each iteration of the algorithm, we generate a random integer between the current index and the last index of the array. Then, we swap the elements at the current index and the chosen index - this simulates drawing (and removing) the element from the hat, as the next range from which we select a random index will not include the most recently processed one. One small, yet important detail is that it is possible to swap an element with itself - otherwise, some array permutations would be more likely than others. To see this illustrated more clearly, consider the animation below:

Java | Python — Copy

```
class Solution:
    def __init__(self, nums):
        self.array = nums
        self.original = list(nums)

    def reset(self):
        self.array = self.original
        self.original = list(self.original)
        return self.array

    def shuffle(self):
        for i in range(len(self.array)):
            swap_idx = random.randrange(i, len(self.array))
            self.array[i], self.array[swap_idx] = self.array[swap_idx], self.array[i]
        return self.array
```

Complexity Analysis

- Time complexity : $\mathcal{O}(n)$.

  The Fisher-Yates algorithm runs in linear time, as generating a random index and swapping two values can be done in constant time.

- Space complexity : $\mathcal{O}(n)$.

  Although we managed to avoid using linear space on the auxiliary array from the brute force approach, we still need it for `reset`, so we're stuck with linear space complexity.

Rate this article:

Previous Next

Comments: 25

Sort By

Type comment here... (Markdown is supported)

Preview Post

tsiguang ★ 116 June 8, 2018 11:37 PM

Don't abuse fancy words. Both solutions are Fisher-Yates Algorithm. One original is used by pen and paper, and one is improved for computer use. See Fisher-Yates Algorithm and Knuth Shuffle.

30 Share Reply

kremebrulee ★ 52 August 19, 2018 7:34 AM

I'm getting wrong answer 9/10 passed when I copy paste the second algorithm into the solution.

23 Share Reply

SHOW 4 REPLIES

shshlomy ★ 20 November 7, 2018 5:18 PM

in python why not using random.shuffle(array)?

19 Share Reply

SHOW 3 REPLIES

tifa1379 ★ 61 December 4, 2019 3:03 PM Report

For anyone wondering, in the Python code, the utility derived in creating a new List object with each assignment is the creation of a **Deep Copy** of the array.

```
self.original = list(nums)
self.original = listSet(original)
```

Read More

16 Share Reply

SHOW 1 REPLY

yy106 ★ 17 June 19, 2018 3:55 AM

randRange(i, array.length), what's the advantage of this versus randRange(0, array.length)? does it impact chances of each elements swapping?

10 Share Reply

SHOW 1 REPLY

calvincharkiF ★ 2984 April 23, 2019 10:39 AM

the second approach is very similar to what python does

Here is the implementation of the built-in `random.shuffle()`

```
def shuffle(self, x, random=None):
```

Read More

9 Share Reply

SHOW 1 REPLY

YIMUPRG ★ 330 October 10, 2018 7:39 PM

In reset method(): why we can't just return original? Since we already make original = nums.clone(),

8 Share Reply

SHOW 2 REPLIES

omsrisagar ★ 6 March 13, 2018 12:14 AM

In the second solution for reset, shouldn't we return array instead of original? We did that in the first solution, but why differently in second solution?

6 Share Reply

poorva2808 ★ 2 September 25, 2019 5:08 AM

Can someone please explain the need for line 24: original = original.clone(): in the second approach? why not just return array.

2 Share Reply

SHOW 1 REPLY

fallenranger ★ 76 December 22, 2019 2:01 PM Report

What if I modify approach 1 so that when I generate random index, I swap it with last element of the array and pop? This way time complexity will be O(n).

1 Share Reply

SHOW 2 REPLIES

1 2 3