best way to pick a random subset from a collection?

Asked 16 years, 3 months ago Modified 2 months ago Viewed 39k times

70

I have a set of objects in a Vector from which I'd like to select a random subset (e.g. 100 items coming back; pick 5 randomly). In my first (very hasty) pass I did an extremely simple and perhaps overly clever solution:

```
Vector itemsVector = getItems();

Collections.shuffle(itemsVector);
itemsVector.setSize(5);
```

While this has the advantage of being nice and simple, I suspect it's not going to scale very well, i.e. Collections.shuffle() must be O(n) at least. My less clever alternative is

```
Vector itemsVector = getItems();

Random rand = new Random(System.currentTimeMillis()); // would make this static
to the class

List subsetList = new ArrayList(5);
for (int i = 0; i < 5; i++) {
     // be sure to use Vector.remove() or you may get the same item twice
     subsetList.add(itemsVector.remove(rand.nextInt(itemsVector.size())));
}
```

Any suggestions on better ways to draw out a random subset from a Collection?

java algorithm collections random subset

Share
Improve this question
Follow

edited Sep 25, 2008 at 22:49
Alexander
9,380 ●2 ●28 ●23

asked Sep 25, 2008 at 22:02
Tom
803 ●1 ●7 ●5

Strictly speaking, your code assumes you're dealing with a list/vector. If you dealt with an arbitrary collection, you'd first have to extract all of its items into a list/vector/array which could be quite expensive. This is because the usual shuffling algorithms work only on lists/arrays. – Alexander Sep 25, 2008 at 22:55

2 I've found Floyd's algorithm to provide a provably uniform distribution over all subsets, so I highly recommend Eyal Schneider's answer, which links to a post detailed it, incl. proof and implementation. – Jean-Philippe Pellet Jun 10, 2011 at 9:59

1 itemsVector.remove is O(n) docs.oracle.com/javase/7/docs/api/java/util/…. I think O(k) running time is possible. – Kunukn Oct 18, 2013 at 19:16

10 Answers

Sorted by: Highest score (default)

12

Jon Bentley discusses this in either 'Programming Pearls' or 'More Programming Pearls'. You need to be careful with your N of M selection process, but I think the code shown works correctly. Rather than randomly shuffle all the items, you can do the random shuffle only shuffling the first N positions - which is a useful saving when N << M.

Knuth also discusses these algorithms - I believe that would be Vol 3 "Sorting and Searching", but my set is packed pending a move of house so I can't formally check that.

Share
Improve this answer
Follow

edited Nov 5, 2008 at 15:02

answered Sep 25, 2008 at 22:35
Jonathan Leffler
752k ●145 ●946 ●1.3k

+1 for beating me to the answer. I was also writing about performing the random shuffle for the first five steps: choose random number from 1 to M, swap the first element with the element at that index, choose a random number from 2 to M, swap second element, and so forth. – Alexander Sep 25, 2008 at 22:46

Thanks to everybody for providing all the great info. While they all had great things to add, I'm picking this because it's probably the way I'll refactor the code: * set i = 0 * grab random element r from i to n * swap element @ i with element @ r * i++ * repeat until I've got the ones I want – Tom Sep 30, 2008 at 9:52

9

@Jonathan,

I believe this is the solution you're talking about:

```
void genknuth(int m, int n)
{    for (int i = 0; i < n; i++)
         /* select m of remaining n-i */
         if ((bigrand() % (n-i)) < m) {
             cout << i << "\n";
             m--;
         }
}
```

It's on page 127 of Programming Pearls by Jon Bentley and is based off of Knuth's implementation.

EDIT: I just saw a further modification on page 129:

```
void genshuf(int m, int n)
{    int i,j;
     int *x = new int[n];
     for (i = 0; i < n; i++)
         x[i] = i;
     for (i = 0; i < m; i++) {
         j = randint(i, n-1);
         int t = x[i]; x[i] = x[j]; x[j] = t;
     }
     sort(x, x+m);
     for (i = 0; i< m; i++)
         cout << x[i] << "\n";
}
```

This is based on the idea that "...we need shuffle only the first *m* elements of the array..."

Share

Improve this answer

Follow

edited Nov 28, 2017 at 11:59

too honest for this site
12.3k ● 4 ● 32 ● 53

answered Sep 25, 2008 at 22:57

daniel
2,626 ● 1 ● 26 ● 32

5

If you're trying to select k distinct elements from a list of n, the methods you gave above will be O(n) or O(kn), because removing an element from a Vector will cause an arraycopy to shift all the elements down.

Since you're asking for the best way, it depends on what you are allowed to do with your input list.

If it's acceptable to modify the input list, as in your examples, then you can simply swap k random elements to the beginning of the list and return them in O(k) time like this:

```
public static <T> List<T> getRandomSubList(List<T> input, int subsetSize)
{
    Random r = new Random();
    int inputSize = input.size();
    for (int i = 0; i < subsetSize; i++)
    {
        int indexToSwap = i + r.nextInt(inputSize - i);
        T temp = input.get(i);
        input.set(i, input.get(indexToSwap));
        input.set(indexToSwap, temp);
    }
    return input.subList(0, subsetSize);
}
```

If the list must end up in the same state it began, you can keep track of the positions you swapped, and then return the list to its original state after copying your selected sublist. This is still an O(k) solution.

If, however, you cannot modify the input list at all and k is much less than n (like 5 from 100), it would be much better not to remove selected elements each time, but simply select each element, and if you ever get a duplicate, toss it out and reselect. This will give you O(kn / (n-k)) which is still close to O(k) when n dominates k. (For example, if k is less than n / 2, then it reduces to O(k)).

If k not dominated by n, and you cannot modify the list, you might as well copy your original list, and use your first solution, because O(n) will be just as good as O(k).

As others have noted, if you are depending on strong randomness where every sublist is possible (and unbiased), you'll definitely need something stronger than `java.util.Random`. See `java.security.SecureRandom`.

Share
Improve this answer
Follow

edited Sep 26, 2008 at 16:22

answered Sep 25, 2008 at 23:26
David L
44.6k ● 11 ● 64 ● 62

4

I wrote an efficient implementation of this a few weeks back. It's in C# but the translation to Java is trivial (essentially the same code). The plus side is that it's also completely unbiased (which some of the existing answers aren't) - a way to test that is here.

It's based on a Durstenfeld implementation of the Fisher-Yates shuffle.

Share
Improve this answer
Follow

edited Jun 10, 2011 at 13:28

answered Sep 25, 2008 at 23:06
Greg Beech
136k ● 45 ● 209 ● 250

Great article. One takeaway that I think can be used to improve the code in the original question is to swap elements instead of removing them. This saves the performance penalty from having to collapse the list when the element is removed. – qualidafial Sep 25, 2008 at 23:32

This is close to a link-only answer - could someone please update this with the relevant code? – Aaron Hall ♦ Jul 31, 2017 at 1:43

1 The link is broken. – Antoine Jan 25, 2019 at 18:38

Thanks to the power of the WaybackMachine, you can find a copy here – Jayrassic Jun 23, 2020 at 14:44

2

Your second solution of using Random to pick element seems sound, however:

- Depending on how sensitive your data is, I suggest using some sort of hashing method to scramble the random number seed. For a good case study, see How We Learned to Cheat at Online Poker (but this link is 404 as of 2015-12-18). Alternative URLs (found via a Google search on the article title in double quotes) include:
  - How We Learned to Cheat at Online Poker — apparently the original publisher.
  - How We Learned to Cheat at Online Poker
  - How We Learned to Cheat at Online Poker
- Vector is synchronized. If possible, use ArrayList instead to improve performance.

Share
Improve this answer
Follow

edited Dec 18, 2015 at 17:38
Jonathan Leffler
752k ● 145 ● 946 ● 1.3k

answered Sep 25, 2008 at 22:10
qualidafial
6,812 ● 3 ● 30 ● 38

Thanks for the tip on using a better seed; I'll check out the link you've posted. Completely agree about using ArrayList vs. Vector; however, this is a 3rd-party library returning the Vector and I have no control over the datatype being returned. Thanks! – Tom Sep 25, 2008 at 22:12

LOL, I need to fix my shuffle code now...I was using System.nanoTime() as my seed as well! Thanks for the great article. – Pyrolistical Sep 25, 2008 at 22:59

It's sound, but not the best way to do it. It is slower than it needs to be. – David L Sep 26, 2008 at 3:42

1

This is a very similar question on stackoverflow.

To summarize my favorite answers from that page (furst one from user Kyle):

- **O(n) solution**: Iterate through your list, and copy out an element (or reference thereto) with probability (#needed / #remaining). Example: if k = 5 and n = 100, then you take the first element with prob 5/100. If you copy that one, then you choose the next with prob 4/99; but if you didn't take the first one, the prob is 5/99.
- **O(k log k) or O(k^2)**: Build a sorted list of k indices (numbers in {0, 1, ..., n-1}) by randomly choosing a number < n, then randomly choosing a number < n-1, etc. At each step, you need to recallibrate your choice to avoid collisions and keep the probabilities even. As an example, if k=5 and n=100, and your first choice is 43,

your next choice is in the range [0, 98], and if it's >=43, then you add 1 to it. So if your second choice is 50, then you add 1 to it, and you have {43, 51}. If your next choice is 51, you add *2* to it to get {43, 51, 53}.

Here is some pseudopython -

```
# Returns a container s with k distinct random numbers from {0, 1, ..., n-1}
def ChooseRandomSubset(n, k):
  for i in range(k):
    r = UniformRandom(0, n-i)                 # May be 0, must be < n-i
    q = s.FirstIndexSuchThat( s[q] - q > r )  # This is the search.
    s.InsertInOrder(q ? r + q : r + len(s))   # Inserts right before q.
  return s
```

I'm saying that the time complexity is O(k^2) **or** O(k log k) because it depends on how quickly you can search and insert into your container for s. If s is a normal list, one of those operations is linear, and you get k^2. However, if you're willing to build s as a balanced binary tree, you can get out the O(k log k) time.

Share

Improve this answer

Follow

edited May 23, 2017 at 12:34

Community Bot
1 ● 1

answered Sep 26, 2008 at 1:26

Tyler
28.9k ● 12 ● 93 ● 108

These are decent, but not the best way. It can be done in O(k). – David L Sep 26, 2008 at 3:44

These don't mess with the original array. I haven't seen any solutions that do as well without manipulating the original array. – Tyler Sep 26, 2008 at 6:13

I've added such a solution above. So long as k is considerably less than n, you're better off just selecting random elements from the list, and throwing out dupes until you get k. – David L Sep 26, 2008 at 16:19

That is a practically useful algorithm esp if you use a hash set to check for collisions quickly. But from theoretical analysis, the worst-case is actually O(infinity) because you have no guaranteed limit on # of collisions; a nonhashed version still takes O(log k) per collision check=k log k total. – Tyler Sep 26, 2008 at 23:19

Indeed, you clearly should use a hashed set to check for collisions. Since we're dealing with a randomized algorithm, it's important to analyze the complexity for the worst case over the input, but the expected case over the random values. – David L Sep 27, 2008 at 3:31

0

How much does remove cost? Because if that needs to rewrite the array to a new chunk of memory, then you've done O(5n) operations in the second version, rather than the O(n) you wanted before.

You could create an array of booleans set to false, and then:

```
for (int i = 0; i < 5; i++){
   int r = rand.nextInt(itemsVector.size());
   while (boolArray[r]){
       r = rand.nextInt(itemsVector.size());
   }
   subsetList.add(itemsVector[r]);
   boolArray[r] = true;
}
```

This approach works if your subset is smaller than your total size by a significant margin. As those sizes get close to one another (ie, 1/4 the size or something), you'd get more collisions on that random number generator. In that case, I'd make a list of integers the size of your larger array, and then shuffle that list of integers, and pull off the first elements from that to get your (non-colliding) indeces. That way, you have the cost of O(n) in building the integer array, and another O(n) in the shuffle, but no collisions from an internal while checker and less than the potential O(5n) that remove may cost.

Share Improve this answer Follow

answered Sep 25, 2008 at 22:17
mmr
14.9k ● 29 ● 97 ● 148

O(5N) === O(N); that's the point of big-O notation. However, when you have two methods, both of O(N), then the constant multiplier and the constant addition terms become significant (and any relevant sub-linear terms). – Jonathan Leffler Sep 25, 2008 at 22:30

0

I'd personal opt for your initial implementation: very concise. Performance testing will show how well it scales. I've implemented a very similar block of code in a decently abused method and it scaled sufficiently. The particular code relied on arrays containing >10,000 items as well.

Share Improve this answer Follow

answered Sep 25, 2008 at 22:18
daniel
2,626 ● 1 ● 26 ● 32

0

```
Set<Integer> s = new HashSet<Integer>()
// add random indexes to s
while(s.size() < 5)
{
    s.add(rand.nextInt(itemsVector.size()))
}
// iterate over s and put the items in the list
for(Integer i : s)
{
    out.add(itemsVector.get(i));
}
```

Share

Improve this answer

Follow

edited Sep 25, 2008 at 23:56

answered Sep 25, 2008 at 23:05

Wesley Tarle

668 ●3 ●6

If would be great if it didn't have a probabilistic run-time, which increases a lot when n gets closer to the size of the collection… – Jean-Philippe Pellet Jun 10, 2011 at 10:00

0

two solutions I don't think appear here - the corresponds is quite long, and contains some links, however, I don't think all of the posts relate to the problem of choosing a subst of K elemetns out of a set of N elements. [By "set", I refer to the mathematical term, i.e. all elements appear once, order is not important].

Sol 1:

```
//Assume the set is given as an array:
Object[] set ....;
for(int i=0;i<K; i++){
randomNumber = random() % N;
    print set[randomNumber];
    //swap the chosen element with the last place
    temp = set[randomName];
    set[randomName] = set[N-1];
    set[N-1] = temp;
    //decrease N
    N--;
}
```

This looks similar to the answer daniel gave, but it actually is very different. It is of O(k) run time.

Another solution is to use some math: consider the array indexes as Z_n and so we can choose randomly 2 numbers, x which is co-prime to n, i.e. chhose gcd(x,n)=1, and another, a, which is "starting point" - then the series: a % n,a+x % n, a+2*x % n,...a+(k-1)*x%n is a sequence of distinct numbers (as long as k<=n).

Share

Improve this answer

Follow

edited Jun 4, 2013 at 12:38

guhou
1,732 ● 13 ● 32

answered Mar 3, 2012 at 22:09

user967710
1,997 ● 3 ● 35 ● 62