Finding n-th permutation without computing others

Given an array of N elements representing the permutation atoms, is there an algorithm like that:

```
function getNthPermutation( $atoms, $permutation_index, $size )
```

where `$atoms` is the array of elements, `$permutation_index` is the index of the permutation and `$size` is the size of the permutation.

For instance:

```
$atoms = array( 'A', 'B', 'C' );
// getting third permutation of 2 elements
$perm = getNthPermutation( $atoms, 3, 2 );

echo implode( ', ', $perm )."\n";
```

Would print:

```
B, A
```

Without computing every permutation until $permutation_index ?

I heard something about factoradic permutations, but every implementation i've found gives as result a permutation with the same size of V, which is not my case.

Thanks.

php algorithm math permutation

edited Jun 21 '12 at 8:40

Alix Axel
74.4k 49 260 369

asked Oct 27 '11 at 16:00

Simone Margaritelli
1,641 4 24 57

what do you mean the index of the permutation? – galchen Oct 27 '11 at 16:04

imagine you print every permutation of N elements with its iteration counter (permutation 0, permutation 1, permutation 2, ...) ... i want the n-th permutation. – Simone Margaritelli Oct 27 '11 at 16:08

but what determines the order of the permutation? i mean, permutation with index 0 can be any of the forms – galchen Oct 27 '11 at 16:09

1 i don't care about the sorting of the permutations, any will do the job :) – Simone Margaritelli Oct 27 '11 at 16:15

6 if you don't care about the order, you can just pick ANY permutation of the size $size that you like. do you want to call this function several times each time with a different index? – galchen Oct 27 '11 at 16:20

6 Answers

As stated by RickyBobby, when considering the lexicographical order of permutations, you should use the factorial decomposition at your advantage.

From a practical point of view, this is how I see it:

- Perform a sort of Euclidian division, except you do it with factorial numbers, starting with `(n-1)!`, `(n-2)!`, and so on.
- Keep the quotients in an array. The `i`-th quotient should be a number between `0` and `n-i-1` inclusive, where `i` goes from `0` to `n-1`.
- This array *is* your permutation. The problem is that each quotient does not care for previous values, so you need to adjust them. More explicitly, you need to increment every value as many times as there are previous values that are lower or equal.

The following C code should give you an idea of how this works (`n` is the number of entries, and `i` is the index of the permutation):

```
/**
 * @param n The number of entries
 * @param i The index of the permutation
 */
void ithPermutation(const int n, int i)
{
   int j, k = 0;
   int *fact = (int *)calloc(n, sizeof(int));
   int *perm = (int *)calloc(n, sizeof(int));

   // compute factorial numbers
   fact[k] = 1;
   while (++k < n)
      fact[k] = fact[k - 1] * k;

   // compute factorial code
   for (k = 0; k < n; ++k)
   {
      perm[k] = i / fact[n - 1 - k];
      i = i % fact[n - 1 - k];
   }

   // readjust values to obtain the permutation
   // start from the end and check if preceding values are lower
   for (k = n - 1; k > 0; --k)
      for (j = k - 1; j >= 0; --j)
         if (perm[j] <= perm[k])
            perm[k]++;

   // print permutation
   for (k = 0; k < n; ++k)
      printf("%d ", perm[k]);
   printf("\n");

   free(fact);
   free(perm);
}
```

For example, `ithPermutation(10, 3628799)` prints, as expected, the last permutation of ten elements:

```
9 8 7 6 5 4 3 2 1 0
```

edited May 25 at 19:54

Kowser
4,249 2 18 45

answered Oct 27 '11 at 17:26

FelixCQ
1,301 7 9

+1 thx Felix for the implementation :) – Ricky Bobby Oct 27 '11 at 17:41

That was exactly the implementation i was searching for, the 'n' argument is the key ... thanks soooo much :) – Simone Margaritelli Oct 27 '11 at 22:15

The method used here to get the factoradic / lehmer code (makes use of calculated factorials and stores quotients not remainders) is different from the one described in the Wikipedia page of Factoradic just a little above the Examples section. The output as I've tested is the same however I find the latter method simpler. Nevertheless your example also helped me understand the concept better. – konsolebox Feb 12 at 15:00

It depends on the way you "sort" your permutations (lexicographic order for example).

One way to do it is the factorial number system, it gives you a bijection between [0 , n!] and all the permutations.

Then for any number i in [0,n!] you can compute the ith permutation without computing the others.

This factorial writing is based on the fact that any number between [0 and n!] can be written as :

```
SUM( ai.(i!) for i in range [0,n-1]) where ai <i
```

(it's pretty similar to base decomposition)

for more information on this decomposition, have a look at this thread :
http://math.stackexchange.com/questions/53262/factorial-decomposition-of-integers

hope it helps

As stated on this wikipedia article this approach is equivalent to computing the lehmer code :

> An obvious way to generate permutations of n is to generate values for the Lehmer code (possibly using the factorial number system representation of integers up to n!), and convert those into the corresponding permutations. However the latter step, while straightforward, is hard to implement efficiently, because it requires n operations each of selection from a sequence and deletion from it, at an arbitrary position; of the obvious representations of the sequence as an array or a linked list, both require (for different reasons) about n2/4 operations to perform the conversion. With n likely to be rather small (especially if generation of all permutations is needed) that is not too much of a problem, but it turns out that both for random and for systematic generation there are simple alternatives that do considerably better. For this reason it does not seem useful, although certainly possible, to employ a special data structure that would allow performing the conversion from Lehmer code to permutation in O(n log n) time.

So the best you can do for a set of n element is O(n ln(n)) with an adapted data structure.

edited Oct 27 '11 at 19:52

answered Oct 27 '11 at 16:11

Ricky Bobby
5,364 1 22 38

i'm already aware of factorial number system, but i can't find an implementation where the size of the output permutation is not the same of the initial vector of items. – Simone Margaritelli Oct 27 '11 at 16:14

@SimoneMargaritelli What do you mean by ? you want a permutation of one subset of your original set of element ? – Ricky Bobby Oct 27 '11 at 16:39

You could actually do O(n lg lg U) using vEB trees, since U=n. I wonder what the lower bound is?? – dhruvbird Apr 4 at 7:54

Here's a solution that allows to select the size of the permutation. For example, apart from being able to generate all permutations of 10 elements, it can generate permutations of pairs among 10 elements. Also it permutes lists of arbitrary objects, not just integers.

This is PHP, but there's also JavaScript, and Haskell impementation.

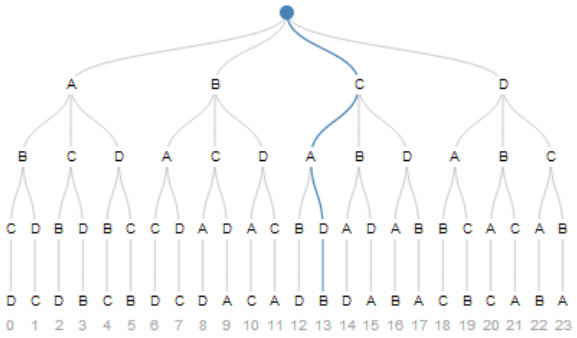
```
function nth_permutation($atoms, $index, $size) {
    for ($i = 0; $i < $size; $i++) {
        $item = $index % count($atoms);
        $index = floor($index / count($atoms));
        $result[] = $atoms[$item];
        array_splice($atoms, $item, 1);
    }
    return $result;
}
```

Usage example:

```
for ($i = 0; $i < 6; $i++) {
    print_r(nth_permutation(['A', 'B', 'C'], $i, 2));
}
// => AB, BA, CA, AC, BC, CB
```

How does it work?

There's a very interesting idea behind it. Let's take the list `A, B, C, D`. We can construct a permutation by drawing elements from it like from a deck of cards. Initially we can draw one of the four elements. Then one of the three remaining elements, and so on, until finally we have nothing left.

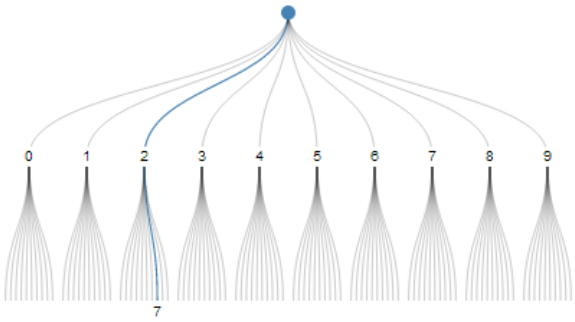


Here is one possible sequence of choices. Starting from the top we're taking the third path, then the first, the the second, and finally the first. And that's our permutation #13.

Think about how, given this sequence of choices, you would get to the number thirteen algorithmically. Then reverse your algorithm, and that's how you can reconstruct the sequence from an integer.

Let's try to find a general scheme for packing a sequence of choices into an integer without redundancy, and unpacking it back.

One interesting scheme is called decimal number system. "27" can be thought of as choosing path #2 out of 10, and then choosing path #7 out of 10.



But each digit can only encode choices from 10 alternatives. Other systems that have a fixed radix, like binary and hexadecimal, also can only encode sequences of choices from a fixed number of alternatives. We want a system with a variable radix, kind of like time units, "14:05:29" is hour 14 from 24, minute 5 from 60, second 29 from 60.

What if we take generic number-to-string and string-to-number functions, and fool them into using mixed radixes? Instead of taking a single radix, like parseInt('beef', 16) and (48879).toString(16), they will take one radix per each digit.

```
function pack(digits, radixes) {
    var n = 0;
    for (var i = 0; i < digits.length; i++) {
        n = n * radixes[i] + digits[i];
    }
    return n;
}

function unpack(n, radixes) {
    var digits = [];
    for (var i = radixes.length - 1; i >= 0; i--) {
        digits.unshift(n % radixes[i]);
        n = Math.floor(n / radixes[i]);
    }
    return digits;
}
```

Does that even work?

```
// Decimal system
pack([4, 2], [10, 10]); // => 42

// Binary system
pack([1, 0, 1, 0, 1, 0], [2, 2, 2, 2, 2, 2]); // => 42

// Factorial system
pack([1, 3, 0, 0, 0], [5, 4, 3, 2, 1]); // => 42
```

And now backwards:

```
unpack(42, [10, 10]); // => [4, 2]

unpack(42, [5, 4, 3, 2, 1]); // => [1, 3, 0, 0, 0]
```

This is so beautiful. Now let's apply this parametric number system to the problem of permutations. We'll consider length 2 permutations of `A, B, C, D`. What's the total number of them? Let's see: first we draw one of the 4 items, then one of the remaining 3, that's `4 * 3 = 12` ways to draw 2 items. These 12 ways can be packed into integers [0..11]. So, let's pretend we've packed them already, and try unpacking:

```
for (var i = 0; i < 12; i++) {
    console.log(unpack(i, [4, 3]));
}
```

```
// [0, 0], [0, 1], [0, 2],
// [1, 0], [1, 1], [1, 2],
// [2, 0], [2, 1], [2, 2],
// [3, 0], [3, 1], [3, 2]
```

These numbers represent choices, not indexes in the original array. [0, 0] doesn't mean taking `A, A`, it means taking item #0 from `A, B, C, D` (that's A) and then item #0 from the remaining list `B, C, D` (that's B). And the resulting permutation is `A, B`.

Another example: [3, 2] means taking item #3 from `A, B, C, D` (that's D) and then item #2 from the remaining list `A, B, C` (that's C). And the resulting permutation is `D, C`.

This mapping is called Lehmer code. Let's map all these Lehmer codes to permutations:

```
AB, AC, AD, BA, BC, BD, CA, CB, CD, DA, DB, DC
```

That's exactly what we need. But if you look at the `unpack` function you'll notice that it produces digits from right to left (to reverse the actions of `pack`). The choice from 3 gets unpacked before the choice from 4. That's unfortunate, because we want to choose from 4 elements before choosing from 3. Without being able to do so we have to compute the Lehmer code first, accumulate it into a temporary array, and then apply it to the array of items to compute the actual permutation.

But if we don't care about the lexicographic order, we can pretend that we want to choose from 3 elements before choosing from 4. Then the choice from 4 will come out from `unpack` first. In other words, we'll use `unpack(n, [3, 4])` instead of `unpack(n, [4, 3])`. This trick allows to compute the next digit of Lehmer code and immediately apply it to the list. And that's exactly how `nth_permutation()` works.

One last thing I want to mention is that `unpack(i, [4, 3])` is closely related to the factorial number system. Look at that first tree again, if we want permutations of length 2 without duplicates, we can just skip every second permutation index. That'll give us 12 permutations of length 4, which can be trimmed to length 2.

```
for (var i = 0; i < 12; i++) {
    var lehmer = unpack(i * 2, [4, 3, 2, 1]); // Factorial number system
    console.log(lehmer.slice(0, 2));
}
```

answered Jun 17 '14 at 7:25

Alexey Lebedev
5,735 1 18 31

Here's an algorithm to convert between permutations and ranks in linear time. However, the ranking it uses is not lexicographic. It's weird, but consistent. I'm going to give two functions, one that converts from a rank to a permutation, and one that does the inverse.

First, to unrank (go from rank to permutation)

```
Initialize:
n = length(permutation)
r = desired rank
p = identity permutation of n elements [0, 1, ..., n]

unrank(n, r, p)
  if n > 0 then
    swap(p[n-1], p[r mod n])
    unrank(n-1, floor(r/n), p)
  fi
end
```

Next, to rank:

```
Initialize:
p = input permutation
q = inverse input permutation (in linear time, q[p[i]] = i for 0 <= i < n)
n = length(p)

rank(n, p, q)
  if n=1 then return 0 fi
  s = p[n-1]
  swap(p[n-1], p[q[n-1]])
  swap(q[s], q[n-1])
  return s + n * rank(n-1, p, q)
end
```

The running time of both of these is O(n).

There's a nice, readable paper explaining why this works: Ranking & Unranking Permutations in Linear Time, by Myrvold & Ruskey, Information Processing Letters Volume 79, Issue 6, 30 September 2001, Pages 281–284.

http://webhome.cs.uvic.ca/~ruskey/Publications/RankPerm/MyrvoldRuskey.pdf

answered Oct 4 '14 at 21:59

Dave Galvin
965 4 10

This solution is likely the fastest because you don't have to do array splicing (or element removing) and there are no nested for loops +1. – James Jun 14 at 3:11

Here is a short and very fast (linear in the number of elements) solution in python, working for any list of elements (the 13 first letters in the example below) :

```
from math import factorial

def nthPerm(n,elems):#with n from 0
    if(len(elems) == 1):
        return elems[0]
    sizeGroup = factorial(len(elems)-1)
    q,r = divmod(n,sizeGroup)
    v = elems[q]
    elems.remove(v)
    return v + ", " + ithPerm(r,elems)
```

Examples :

```
letters = ['a','b','c','d','e','f','g','h','i','j','k','l','m']

ithPerm(0,letters[:])          #--> a, b, c, d, e, f, g, h, i, j, k, l, m
ithPerm(4,letters[:])          #--> a, b, c, d, e, f, g, h, i, j, m, k, l
ithPerm(3587542868,letters[:]) #--> h, f, l, i, c, k, a, e, g, m, d, b, j
```

Note: I give `letters[:]` (a copy of `letters`) and not letters because the function modifies its parameter `elems` (removes chosen element)

edited Aug 26 '14 at 20:00

lennon310
8,241 10 18 37

answered Aug 21 '14 at 21:33

ismax
21 3

If you store all the permutations in memory, for example in an array, you should be able to bring them back out one at a time in O(1) time.

This does mean you have to store all the permutations, so if computing all permutations takes a prohibitively long time, or storing them takes a prohibitively large space then this may not be a solution.

My suggestion would be to try it anyway, and come back if it is too big/slow - there's no point looking for a "clever" solution if a naive one will do the job.

answered Oct 27 '11 at 16:11

Chris Browne
901 9 20

3 sorry, my psychic powers must be failing me today - either that or you put that information in very small text in your question. – Chris Browne Oct 27 '11 at 16:16

3 +1 for giving Simone not the answer to the question he meant to ask, but the answer to the question he actually asked. – Patrick87 Oct 27 '11 at 16:58

3 i think it was kinda obvious since i stated '... Without computing every permutation ...' ... – Simone Margaritelli Oct 27 '11 at 17:02

3 You actually stated "without computing every permutation until $permutation_index", which is not the same as "without computing every permutation". That's the first time I've ever seen somebody quote *themselves* out of context! – Chris Browne Oct 27 '11 at 20:47

2 Can't resist. An algorithm that uses precomputed permutations does not compute any permutations. (I am only here because I found the question and the other responses useful). – dansalmo Dec 27 '13 at 17:10