Permutation Rank Problem

The problem is as follows:

Given a permutation of the integers, 0, 1, …, n-1, find it's 0-based rank, in lexicographical order.

For example, for n = 3, here is the output for each possible permutation:

[0,1,2] → 0
[0,2,1] → 1
[1,0,2] → 2
[1,2,0] → 3
[2,0,1] → 4
[2,1,0] → 5

Let's try to solve this problem more generally, without actually having to list each permutation.

Let's look at an example with n = 8:

3, 6, 0, 5, 1, 4, 7, 2

From the beginning, we know that we must "pass over" all permutations that start with 0, 1 and 2.

How many of those are there?

Well, if we put a 0 in the first slot, there are 7! ways to fill in the other 7 slots. There's nothing special about 0, so the same answer is true if we were to put a 1 or 2 in the first slot.

Thus, since we know that we must skip over 3 choices for the first number, then we know that we are skipping over

3 x 7!

Permutations. It follows that we can add 3 x 7! to our rank.

Now, we have the values 0, 1, 2, 4, 5, 6 and 7 left to permute.

Our next value to place is 6. This means we are skipping over 0, 1, 2, 4 and 5 as the second item. So we must add to our rank the permutations that start as follows:

3, 0, __ , __ , __ , __ , __ , __
3, 1, __ , __ , __ , __ , __ , __
3, 2, __ , __ , __ , __ , __ , __
3, 4, __ , __ , __ , __ , __ , __
3, 5, __ , __ , __ , __ , __ , __

We can see that there are 6! ways to fill the slots on each row. So we must simply be able to calculate how many unused numbers are in our permutation, which in this case is 5. It follows that we'll add

5 x 6! to our rank.

We continue iteratively, through each slot, i, using this same logic:

1) Determine the number of unused values that are less than perm[i], call this number x.

2) Add x*(n-1-i)! to our rank, because we are skipping over x values to fill in slot i, and each of the future slots n – 1 – i slots can be filled in (n – 1 – i)! ways.

To implement #1 in Java, we keep an ArrayList of the unused integers and use the indexOf method, which will return the first index where a particular value is stored. This index, in turn, represents the number of items we skipped over to place our current value in the current slot.

Here is code that implements this algorithm. It assumes that enough of the values of the factorial are pre-computed and stored in an array called fact:

```
public static int rank(int[] perm) {

	int n = perm.length;

	ArrayList<Integer> unused = new ArrayList<Integer>();
	for (int i=0; i<n; i++)
		unused.add(i);

	int res = 0;
	for (int i=0; i<n; i++) {

		// cur number is perm[i] find its index in unused.
		int idx = unused.indexOf(perm[i]);

		// idx represents how many numbers we're skipping.
		res += idx*fact[n-1-i];

		// Remove the value at this index.
		unused.remove(idx);
	}

	return res;
}
```

Converting a Rank to a Permutation

Now, let's look at the reverse problem. Given a rank and a length of a permutation, n, determine which permutation is that rank, in lexicographical order.

Let's consider the following example:

n = 6, 0-based rank = 341

We know there are 5! ways to fill in slots 1 – 5. This means we can figure out how many values get skipped in slot 0 via integer division by 5!. Since 341/120 = 2, that means that all permutations start start with 0 and 1 are skipped (there are 240 of these), but that the 341st permutation must start with 2, since permutation with ranks 240 through 359 all start with 2.

2, __, __, __, __, __

Now we know that what remains to be permuted is 0, 1, 3, 4, 5, and that we've skipped over 240 items. Thus, we desire the permutation of 0, 1, 3, 4, 5 with rank 341 – 240 = 101. (This is the reverse process of what we did before which was adding the contribution of skipped values from a starting point of 0. Now, we start at the rank and subtract out the skipped values.)

Now, take 101/4! = 4. This means we'll skip 4 values from the list 0, 1, 3, 4, 5. If we store these in an ArrayList, this just means to look at index 4 in that ArrayList. So, we now have:

2, 5, __, __, __, __, with new rank 101 – 4 x 4! = 5

So we want the 5th ranked item of the permutations of 0, 1, 3, 4.

Take 5/3! = 0, so we want the item in index 0, which is 0:

2, 5, 0, __, __, __, with new rank = 5 – 0 x 3! = 5

We are permuting 1, 3, 4.

Take 5/2! = 2, so we want the item in index 2, which is 4:

2, 5, 0, 4, __, __, with new rank = 5 – 2 x 2! = 1

We are permuting 1, 3.

Take 1/1! = 1, which means we want the item in index 1 which is 3:

2, 5, 0, 4, 3, __, with new rank 1 – 1 x 1! = 0

The 0^{th} ranked item from the list of just 1 is the item in index 0, which is 1:

2, 5, 0, 4, 3, 1

is the permutation of 6 items with rank 341.

Here is code which implements this algorithm. Again, we assume we have access to an array of pre-computed factorials.

```
public static int[] getperm(int rank, int n) {

	int[] res = new int[n];

	ArrayList<Integer> unused = new ArrayList<Integer>();
	for (int i=0; i<n; i++)
		unused.add(i);

	for (int i=0; i<n; i++) {

		int idx = rank/fact[n-1-i];

		res[i] = unused.get(idx);

		unused.remove(idx);

		rank -= idx*fact[n-1-i];
	}

	return res;
}
```

Note: Even though the indexOf method is slow (O(n)), we use it because our n for permutation is relatively small. There are tricks to speed this up, but they are beyond the scope of this class and generally unnecessary for a vast majority of permutation type problems.