Enumerating set partitions with Bell numbers and Stirling numbers of the second kind

devblogs.microsoft.com/oldnewthing/20140324-00

March 24, 2014

Raymond Chen

Just for fun, today's Little Program is going to generate set partitions. (Why? Because back in 2005, somebody asked about it on an informal mailing list, suggesting it would be an interesting puzzle, and now I finally got around to solving it.)

The person who asked the question said,

> Below we show the partitions of [4]. The periods separate the individual sets so that, for example, 1.23.4 is the partition {{1},{2,3},{4}}.

1 blocks:	1234						
2 blocks:	123.4,	124.3,	134.2,	1.234,	12.34,	13.24,	14.23
3 blocks:	1.2.34,	1.24.3,	1.23.4,	14.2.3,	13.2.4,	12.3.4	
4 blocks:	1.2.3.4						

I replied with a hint, saying, "This page explains what you need to do, once you reinterpret the Stirling recurrence as an enumeration." Only now, writing up this post, did I realize that I linked to *the same page they were quoting from*.

The key is in the sentence, "They satisfy the following recurrence relation, *which forms the basis of recursive algorithms for generating them*." Let's reinterpret the Stirling recurrence combinatorially. No wait, we don't need to. Wikipedia did it for us. Go read that first. If you didn't follow Wikipedia's explanation, here's another angle:

Suppose you have a set of *n* elements, and you want to generate all the partitions into *k* blocks. Well, let's look at element *n*. What happens when you delete it from the partition?

One possibility is that n was in a block all by itself. After you remove it, you are left with a partition of $n - 1$ elements into $k - 1$ blocks. Therefore, to generate the partitions where n is in a block all by itself, you generate all the partitions of $n - 1$ elements into $k - 1$ blocks, and then tack on a single block consisting of just element n.

On the other hand, if element n was not in a block by itself, then removing it leaves a partition of $n - 1$ elements into k blocks. (Removing n did not decrease the number of blocks because there are still other numbers keeping that block alive.) Now, element n could have belonged to any of the k blocks, so you have k possible places where you could reinsert element n.

Therefore, our algorithm goes like this:

- Handle base cases.
- Otherwise,
    - Recursively call $S(n - 1, k)$ and add element n separately into each of the k blocks.
    - Recursively call $S(n - 1, k - 1)$ and add a single block consisting of just n.

Now that we spelled out what we're going to do, actually doing it is a bit anticlimactic.

```
function Stirling(n, k, f) {
 if (n == 0 && k == 0) { f([]); return; }
 if (n == 0 || k == 0) { return; }
 Stirling(n-1, k-1, function(s) {
  f(s.concat([[n]])); // append [n] to the array
 });
 Stirling(n-1, k, function(s) {
  for (var i = 0; i < k; i++) {
   f(s.map(function(t, j) { // append n to the i'th subarray
    return t.concat(i == j ? n : []);
   }));
  }
 });
}
```

You can test out this function by logging the results to the console.

```
function logToConsole(s) {
  console.log(JSON.stringify(s));
}
Stirling(4, 2, logToConsole);
```

Let's walk through the function. The first test catches the vacuous base case where you say, "Please show me all the ways of breaking up nothing into zero blocks." The answer is, "There is exactly one way of doing this, which is to break it into zero blocks." Math is cute that way.

The second test catches the other base cases where you say, “Please show me all the ways of breaking up `n` elements into zero blocks” (can’t be done if `n > 0`, because you will always have elements left over), or you say, “Please show me all the ways of breaking up 0 elements into `k` blocks” (which also can’t be done if `k > 0` because there are no elements to put in the first block).

After disposing of the base cases, we get to the meat of the recurrence. First, we ask for all the ways of breaking `n - 1` elements into `k - 1` blocks, and for each of them, we add a single block consisting of just `n`.

Next, we ask for all the ways of breaking `n - 1` elements into `k` blocks, and for each of them, we go into a loop adding `n` to each block. The goofy `map` creates a deep copy of the array and adds `n` to the `i` th block.

Here’s a walkthrough of the goofy `map`:

- The `concat` method creates a new array consisting of the starting array with the `concat` parameters added at the end. If a parameter is an array, then its elements are added; otherwise the parameter itself is added. For example, `[1].concat(2, [3, 4])` returns `[1, 2, 3, 4]`. The `concat` method creates a new array, and a common idiom is `s.concat()` to make a shallow copy of an array.
- The `map` method calls the provided callback once for each element of the array `s`. The return values from all the callbacks are collected to form a new array, which is returned. For example, `[1,2,3].map(function (v) { return v * 2; })` returns the new array `[2, 4, 6]`.
- The `map` callback is called with the subarray as the first parameter and the index as the second parameter. (There is also a third parameter, which we don’t use.)
- Therefore, if all we want to do was create a deep copy of `s`, we could write `s.map(function (t, j) { return t.concat(); })`.
- But we don’t want a perfect deep copy. We want to change the `i` th element. Therefore, we check the index, and if it is equal to the `i`, then we append `n`. Otherwise, we append `[]` which appends nothing.
- After building the array (with modifications), we pass it to the callback function `f`.

This pattern is common enough that maybe we should pull it into a helper function.

```
function copyArrayOfArrays(array, indexToEdit, editor) {
 return array.map(function(e, i) {
  return i === indexToEdit ? editor(e) : e.concat();
 });
}
function Stirling(n, k, f) {
 if (n == 0 && k == 0) { f([]); return; }
 if (n == 0 || k == 0) { return; }
 Stirling(n-1, k-1, function(s) {
  f(s.concat([[n]])); // append [n] to the array
 });
 Stirling(n-1, k, function(s) {
  for (var i = 0; i < k; i++) {
   f(copyArrayOfArrays(s, i, function(e) { return e.concat(n); }));
  }
 });
}
```

The `copyArrayOfArrays` function abstracts the goofy `map`: You give it an array of arrays, and optionally an index to edit and the function that does the editing. It copies the array of arrays and calls your editor on the element you want to edit.

To reduce the number of memory allocations, the recursion could also be done destructively. You’re then counting on the callback not modifying the result, since you’re going to use it again.

```
function Stirling(n, k, f) {
 if (n == 0 && k == 0) { f([]); return; }
 if (n == 0 || k == 0) { return; }
 Stirling(n-1, k, function(s) {
  for (var i = 0; i < k; i++) {
   s[i].push(n);
   f(s);
   s[i].pop();
  }
 });
 Stirling(n-1, k-1, function(s) {
  s.push([n]);
  f(s);
  s.pop();
 });
}
```

The original question was about enumerating all partitions (Bell numbers), and that’s easy to put together from the Stirling numbers.

```
function Bell(n, f) {
 for (var i = 1; i <= n; i++) {
  Stirling(n, i, f);
 }
}
```

Raymond Chen

Follow