Enumerating integer compositions (the return of the binomial coefficients)

devblogs.microsoft.com/oldnewthing/20140714-00

July 14, 2014

Raymond Chen

In number theory, a composition of an integer is an ordered sequence of positive integers which sum to the target value. For example, the value 3 can be written as 3, 1+2, 2+1, or 1+1+1.

You can think about the target number as a string of stars, and a composition is a way of breaking the stars into groups. For example, here are the compositions of 3:

* * *	3
| *	1+2
* *|*	2+1
||*	1+1+1

How would you generate all compositions of a particular length? In the above example, the compositions of length 2 would be 1+2 and 2+1. Let's take a look at the last star in the composition. If it is immediately preceded by a space, then removing it results in a string one star shorter, but with the same number of groups (but the last group is one star smaller). In other words, what's left behind is a composition of $n - 1$ of length k. You can recover the original string by adding a star at the end.

On the other hand, if the last star is immediately preceded by a vertical line, then removing it deletes an entire group, so what remains is a string one star shorter with one fewer group. In other words, what's left behind is a composition of $n - 1$ of length $k - 1$. You can recover the original string by adding a separator and a star at the end.

Therefore, our algorithm goes like this:

- Handle base cases.

- Otherwise,
    - Recursively call Compositions($n - 1$, k) and add a star to the end. (*I.e.*, increment the last term.)
    - Recursively call Compositions($n - 1$, $k - 1$) and add a vertical line and a star to the end. (*I.e.*, add a `+1` to the end.)

```
function Compositions(n, k, f) {
 if (n == 0) { return; }
 if (k == 1) { f([n]); return; }
 Compositions(n-1, k, function(s) {
  f(s.map(function(v, i) { // increment the last element
    return i == s.length - 1 ? v + 1 : v;
  }));
 });
 Compositions(n-1, k-1, function(s) {
  f(s.concat(1)); // append a 1
 });
}
Compositions(5, 3, function(s) { console.log(s.join("+")); });
```

Once again, this algorithm should look awfully familiar, because we've seen it twice before, once in the context of enumerating subsets with binomial coefficients, and again when Enumerating bit strings with a specific number of bits set. All we're doing is decorating the results differently.

Here's a way to see directly how compositions are the same as subset selection. Let's ignore the stars and instead look at the gaps between them.

```
* * * * *
 ^ ^ ^ ^
```

Each of the gaps can hold either a space or a vertical line. Breaking n into k pieces is the same as drawing $k - 1$ vertical lines in the $n - 1$ gaps. In other words, you have $n - 1$ locations and you want to choose $k - 1$ of them: Ta da, we converted the problem into generating subsets of size $k - 1$ from a collection of size $n - 1$. (In mathematics, this visualization is known as stars and bars.)

Therefore, we could have made the `Subsets` function do the work:

```
function Compositions(n, k, f) {
 Subsets(n-1, k-1, function(s) {
  s.push(n);
  f(s.map(function(v, i) { return v - (s[i-1]||0); }));
  s.pop();
 });
}
```

The callback merely calculates the differences between adjacent elements of the subset, which is the number of stars between each line. There is a little extra playing around in order to create a virtual vertical bar at the beginning and end.

Since there is an incremental way of enumerating subsets, there should be an incremental way of enumerating compositions. If you look at how the incremental subset enumerator works, you can see how it maps to incremental composition enumeration: Incrementing an index is the same as moving a bar to the right, which maps to incrementing one term and decrementing the subsequent term. Resetting subsequent indices to the minimum values corresponds to setting the corresponding term to 1. The only trick is maintaining the value of the final term, which gathers all the values squeezed out of earlier terms.

```
function NextComposition(s) {
 var k = s.length;
 for (var i = k - 1; i >= 1; i--) {
  if (s[i] > 1) {
   s[i]--;
   s[i-1]++;
   for (; i < k - 1; i++) { s[k-1] += s[i] - 1; s[i] = 1; }
   return true;
  }
 }
 return false;
}
```

Exercise: If you wanted to generate all compositions of any length, you could do it by generating all compositions of length 1, then compositions of length 2, and so on up to length *n*. What's the easier way of doing it?

Bonus chatter: If you want to generate all partitions (which is like compositions, except that order doesn't matter), you can use this recursive version or this iterative one.

Raymond Chen

Follow