Enumerating bit strings with a specific number of bits set (binomial coefficients strike again)

devblogs.microsoft.com/oldnewthing/20140616-00

June 16, 2014

Raymond Chen

Today's Little Program prints all bit strings of length n subject to the requirement that the string have exactly k 1-bits. For example, all bit strings of length 4 with 2 bits set are 0011, 0101, 0110, 1001, 1010, and 1100. Let's write BitString(n, k) to mean all the bit strings of length n with exactly k bits set.

Let's look at the last bit of a typical member of BitString(n, k). If it is a zero, then removing it leaves a string one bit shorter but with the same number of bits set. Conversely every BitString($n - 1$, k) string can be extended to a BitString(n, k) string by adding a zero to the end.

If the last bit is a one, then removing it leaves a bit string of length $n - 1$ with exactly $k - 1$ bits set, and conversely every bit string of length $n - 1$ with exactly $k - 1$ bits set can be extended to a bit string of length n with exactly k bits set by adding a one to the end.

Therefore, our algorithm goes like this:

- Handle base cases.
- Otherwise,
  - Recursively call BitString($n - 1$, k) and add a 0 to the end.
  - Recursively call BitString($n - 1$, $k - 1$) and add a 1 to the end.

This algorithm may look awfully familiar. The overall recursive structure is the same as enumerating subsets with binomial coefficients; we just decorate the results differently.

That's because this problem is the same as the problem of enumerating subsets. You can think of the set bits as selecting which elements get put into the subset.

It's not surprising, therefore, that the resulting code is identical except for how we report the results. (Instead of generating arrays, we generate strings.)

```
function repeatString(s, n) {
 return new Array(n+1).join(s);
}
function BitString(n, k, f) {
 if (k == 0) { f(repeatString("0", n)); return; }
 if (n == 0) { return; }
 BitString(n-1, k, function(s) { f(s + "0"); });
 BitString(n-1, k-1, function(s) { f(s + "1"); });
}
```

If `k` is zero, then that means we are looking for strings of length `n` that contain no bits set at all. There is exactly one of those, namely the string consisting of `n` zeroes.

If `k` is nonzero but `n` is zero, then that means we want strings of length zero with some bits set. That's not possible, so we return without generating anything.

Finally, we handle the recursive case by generating the shorter strings and tacking on the appropriate digits.

Since this is the same algorithm as subset generation, we should be able to write one in terms of the other:

```
function BitString(n, k, f) {
 Subsets(n, k, function(s) {
  var str = "";
  for (var i = 1; i <= n; i++) {
   str += s.indexOf(i) >= 0 ? "1" : "0";
  }
  f(str);
 });
}
```

Raymond Chen

Follow