Enumerating subsets with binomial coefficients

devblogs.microsoft.com/oldnewthing/20140414-00

April 14, 2014

Raymond Chen

Inspired by the Little Program which enumerates set partitions, I decided to do the binomial coefficients this week. In other words, today's Little Program generates all subsets of size *k* from a set of size *n*.

As before, the key is to interpret a recurrence combinatorially. In general, when a recurrence is of the form $A + B$, it means that at the recursive step, you should do *A*, followed by *B*. In our case, the recurrence is $C(n, k) = C(n - 1, k) + C(n - 1, k - 1)$. The combinatorial interpretation of the recurrence is to look at how you can go from a set of size *n* to a set of size $n - 1$ by studying the effect of removing element *n*. If element *n* is not part of the subset, then what's left is a subset of size *k*. If it is part of the subset, then removing it leaves a subset of size $k - 1$.

Therefore, our algorithm goes like this:

- Handle base cases.
- Otherwise,
    - Recursively call $C(n - 1, k)$ and pass the results through.
    - Recursively call $C(n - 1, k - 1)$ and add element *n* to each of the results.

As usual, once we spelled out what we're going to do, actually doing it is pretty straightforward.

```
function Subsets(n, k, f) {
 if (k == 0) { f([]); return; }
 if (n == 0) { return; }
 Subsets(n-1, k, f);
 Subsets(n-1, k-1, function(s) {
   f(s.concat(n));
 });
};
```

The first test catches the vacuous base case where you say, "Please show me all the zero-sized subsets of a set of size `n`." The answer is "There is exactly one zero-sized subset, called the empty set."

The second test catches the other base cases where you say, “Please show me all the `k`-sized subsets[1] of the empty set.” This can’t be done if `k > 0`, because the only subset of the empty set is the empty set itself, and its size is not `k`.

The meat of the recurrence is pretty much what we said. First, we generate all the `k`-sized subsets from a set of size `n-1` and pass them through. Then we generate all the `k-1`-sized subsets from a set of size `n-1` and add the element `n` to them.

We can test out the function by logging the results to the console.

```
Subsets(5, 3, logToConsole);
```

For style points, we can accumulate the results in helper parameters. This records the pending work in parameters instead of closures, which makes the code easier to port to languages which don’t support closures. (And probably helps the efficiency a bit too.)

```
function AccumulateSubsets(n, k, f, chosen) {
 if (k == 0) { f(chosen); return; }
 if (n == 0) { return; }
 AccumulateSubsets(n-1, k, f, chosen);
 AccumulateSubsets(n-1, k-1, f, [n].concat(chosen));
};
function Subsets(n, k, f) {
 AccumulateSubsets(n, k, f, []);
}
```

(I prepend `n` to `chosen` for extra style points, since it causes the results to be enumerated in a prettier order.)

As with Stirling numbers, we can use a destructive recursion to reduce memory allocation, if we can count on the callback not modifying the result. I’ll leave that as an exercise, because I’ve got something even better up my sleeve: Getting rid of the recursion entirely!

Let’s consider the case of enumerating all the subsets of size *k* for a fixed *k* known at compile-time. Let’s say *k* is 3. You can structure this as a series of nested loops.

```
function Subsets3(n, f) {
 for (var i = 1; i <= n - 2; i++) {
  for (var j = i + 1; j <= n - 1; j++) {
   for (var k = j + 1; k <= n; k++) {
    f([i, j, k]);
   }
  }
 }
}
```

The outer loop chooses the first element, the middle loop chooses the second element, and the inner loop chooses the last element. This clearly generalizes to bigger subsets; you just need more loop variables.

With this interpretation, you can see how to get from one subset to the next subset: You increment the last element, and if that's not possible without violating the loop constraint, then you back out one level and try incrementing the next-to-last element (and restarting any inner loops), and so on, backing out until you finally find an index that can be incremented (or give up).

```
function NextSubsetSameSize(s, n) {
 var k = s.length;
 // look for an index that can be incremented
 for (i = k - 1; i >= 0; i--) {
  // can this index be incremented?
  if (s[i] < n - k + i + 1) {
   // increment it
   s[i]++;
   // reset all inner loops
   while (++i < k) s[i] = s[i-1] + 1;
   return true;
  }
 }
 return false;
}
```

The loop on `i` looks for the highest index that can be incremented. The loop bounds depend on which index you are studying, since lower indices need to leave enough room for higher indices, but can you figure out the formula by looking at the pattern in `Subset3`. Once we find an index with room, we increment it and reset all the subsequent indices to their initial values. If we can't find an index to increment, then we report failure.

```
// Enumerate all subsets of size 3 from a set of size 5
var s = [1, 2, 3]; // initial subset
do {
 console.log(JSON.stringify(s));
} while (NextSubsetSameSize(s, 5));
```

Note

[1] In math circles, the phrase *k-sized subsets* is typically abbreviated as *k-subsets*, but I chose to spell it out here because the shorthand takes some getting used to.

Raymond Chen

Follow