Enumerating the ways of choosing teams in a group of players

devblogs.microsoft.com/oldnewthing/20140804-00

August 4, 2014

Raymond Chen

Suppose you have a bunch of people, and you want to break them up into m teams of size n. (Therefore you have a total of nm people.) Today's Little Program will enumerate the ways this can be done.

Formally, let's say that you have a collection of size nm, and you want to enumerate the ways of partitioning the collection into m subsets, each subset of size n. The order of elements within each subset does not matter, and the order of the subsets doesn't matter. That's saying that a team of Alice and Bob is the same as a team of Bob and Alice, and Alice-Bob versus Charlie-David is the same as Charlie-David versus Alice-Bob.

The number of ways of doing this is $(nm)!/n!^m m!$. You can see this by first taking all permutations of the players, then dividing out by the things that cause us to overcount: The number of ways of ordering players within each team is $n!$, and there are m teams, and there are $m!$ ways of ordering the teams themselves. (Note that this is a cute way of expressing the result, but you shouldn't use it for computation. A slightly better way for computation would be $(\Pi_{1 \le k \le n} C(mk, m))/m!$.

Okay, but how do you generate the teams themeselves?

Let's first see how to generate the first team. Well, that's easy. You just select n players and call them *Team 1*.

This leaves you $n(m - 1)$ players with which to form $m - 1$ teams, which you can do recursively.

```
function Teams(n, m, f) {
 var a = [];
 for (var i = 1; i <= n * m; i++) {
  a.push(i);
 }
 if (m == 1) { f([a]); return; }
 Subsets(n * m, n, function(s) {
  var rest = a.filter(function(i) { return s.indexOf(i) < 0; });
  Teams(n, m - 1, function(t) {
    f([s].concat(t.map(function(team) {
        return team.map(function(i) { return rest[i-1]; });
       })));
  });
 });
}
Teams(2, 3, logToConsole);
```

The first part of this function builds an array of the form `[1, 2, 3, ..., n * m]`. If we are asking for only one team, then everybody is on the same team. Otherwise, for all possible choices of `n`-member teams, first see which people haven't yet been picked for a team. Then generate all remaining possible team arrangements for those leftovers, and combine them to form the final team rosters.

The combination step is tricky because the recursive call generates subsets in the range `[1, 2, 3, ..., n * (m-1)]`, and we need to convert those values into indices into the array of people waiting to be picked.

Note that this algorithm over-counts the possibilities since it generates both `[[1,2],[3,4]]` and `[[3,4],[1,2]]`. In other words, it assumes that team order is important (say, because the first team will wear red jerseys and the second team will wear blue jerseys). In the original problem statement, the order of the teams is not significant. (Maybe we'll let them pick their own jersey colors.)

To solve that, we impose a way of choosing one such arrangement as the one we enumerate, and ignore the rest. The natural way to do this is to select a representative player from each team in a predictable manner (say, the one whose name comes first alphabetically), and then arranging the representatives in a predictable manner (say, by sorting them alphabetically).

The revised version of our algorithm goes like this:

```
function Teams(n, m, f) {
 var a = [];
 for (var i = 1; i <= n * m; i++) {
  a.push(i);
 }
 if (m == 1) { f([a]); return; }
 a.shift();
 Subsets(n * m - 1, n - 1, function(s) {
  var firstTeam = [1].concat(s.map(function(i) { return i+1; }))
  var rest = a.filter(function(i) { return s.indexOf(i) < 0; });
  Teams(n, m - 1, function(t) {
    f([firstTeam].concat(t.map(function(team) {
      return team.map(function(i) { return rest[i-1]; });
     })));
  });
 });
}
Teams(2, 3, logToConsole);
```

The first part of the function is the same as before, but the recursive step changes.

We remove the first element from the array. That guy needs to belong to *some* team, and since he's the smallest-numbered guy, he will be nominated as the team representative of whatever team he ends up with, and since he's the smallest-numbered guy of all, he will also be the first team representative when they are placed in sorted order. So we pick him right up front.

We then ask for his `n - 1` teammates, and together they make up the first team. The combination is a little tricky because the `Subsets` function assumes that the underlying set is `[1, 2, ..., n-1]` but we actually want the subset to be of the form `[2, 3, ..., n]`; we fix that by adding 1 to each element of the subset.

We then find all the people who have yet to be assigned to a team and recursively ask for `m - 1` more teams to be generated from them. We then combine the first team with the recursively-generated teams. Again, since the recursively-generated teams are numbered starting from 1, we need to convert the returned subsets into the original values we saved away in the `rest` variable.

Renumbering elements is turning into a bit of a bother, so let's tweak our original `Subsets` function. For example, we would prefer to pass the set explicitly rather than letting `Subsets` assume that the set is `[1, 2, 3, ..., n]`, forcing us to convert the indices back to the original set members. It's also convenient if the callback also included the elements that are not in the subset.

```
function NamedSubsets(a, k, f) {
 if (k == 0) { f([], a); return; }
 if (a.length == 0) { return; }
 var n = a[a.length - 1];
 var rest = a.slice(0, -1);
 NamedSubsets(rest, k, function(chosen, rejected) {
  f(chosen, rejected.concat(n));
 });
 NamedSubsets(rest, k-1, function(chosen, rejected) {
  f(chosen.concat(n), rejected);
 });
}
function takeAndLeave(chosen,rejected) {
 console.log("take " + chosen + ", leave " + rejected);
}
NamedSubsets(["alice", "bob", "charlie"], 2, takeAndLeave);
```

The `NamedSubsets` function takes the last element from the source set and either rejects it (adds it to the "rejected" parameter) or accepts it (adds it to the "chosen" parameter).

With the `NamedSubsets` variant, we can write the `Teams` function much more easily.

```
function Teams(a, m, f) {
 var n = a.length / m;
 if (m == 1) { f([a]); return; }
 var p = a[0];
 NamedSubsets(a.slice(1), n - 1, function(teammates, rest) {
  var team = [p].concat(teammates);
  Teams(rest, m - 1, function(teams) {
    f([team].concat(teams));
  });
 });
}
Teams([1,2,3,4,5,6], 3, logToConsole);
```

Assuming we're not in one of the base cases, we grab the first person `p` so he can be captain of the first team. We then ask `NamedSubsets` to generate his teammates and add them to `p`'s team. We then recursively generate all the other teams from the people who haven't yet been picked, and our result is our first team plus the recursively-generated teams.

There is a lot of potential for style points with the `NamedSubsets` function. For example, we can avoid generating temporary copies of the `a` array just to remove an element by instead passing slices (an array and indices marking the start and end of the elements we care about).

```
function NamedSubsetsSlice(a, begin, end, k, f) {
 if (k == 0) { f([], a.slice(begin, end)); return; }
 if (begin == end) { return; }
 var n = a[end - 1];
 NamedSubsetsSlice(a, begin, end - 1, k, function(chosen, rejected) {
  f(chosen, rejected.concat(n));
 });
 NamedSubsetsSlice(a, begin, end - 1, k-1, function(chosen, rejected) {
  f(chosen.concat(n), rejected);
 });
}
function NamedSubsets(a, k, f) {
 NamedSubsetsSlice(a, 0, a.length, k, f);
}
```

We could use an accumulator to avoid having to generate closures.

```
function AccumulateNamedSubsets(a, begin, end, k, f, chosen, rejected) {
 if (k == 0) { f(chosen, rejected.concat(a.slice(begin, end))); return; }
 if (begin == end) { return; }
 var n = a[begin];
 AccumulateNamedSubsets(a, begin + 1, end, k-1, f, chosen.concat(n), rejected);
 AccumulateNamedSubsets(a, begin + 1, end, k, f, chosen, rejected.concat(n));
}
function NamedSubsetsSlice(a, begin, end, k, f) {
 AccumulateNamedSubsets(a, begin, end, k, f, [], []);
}
function NamedSubsets(a, k, f) {
 NamedSubsetsSlice(a, 0, a.length, k, f);
}
```

For bonus style points, I recurse on the start of the range rather than the beginning so that the results are in a prettier order.

We can also get rid of the temporary accumulator objects by manipulating the accumulators destructively.

```
function AccumulateNamedSubsets(a, begin, end, k, f, chosen, rejected) {
 if (k == 0) { f(chosen, rejected.concat(a.slice(begin, end))); return; }
 if (begin == end) { return; }
 var n = a[begin];
 chosen.push(n);
 AccumulateNamedSubsets(a, begin + 1, end, k-1, f, chosen, rejected);
 chosen.pop();
 rejected.push(n);
 AccumulateNamedSubsets(a, begin + 1, end, k, f, chosen, rejected);
 rejected.pop();
}
```

And then we can take advantage of the accumlator version to pre-select the first player when building teams.

```
function Teams(a, m, f) {
 var n = a.length / m;
 if (m == 1) { f([a]); return; }
 AccumulateNamedSubsetsSlice(a, 1, a.length, n - 1, function(team, rest) {
  Teams(rest, m - 1, function(teams) {
    f([team].concat(teams));
  });
 }, [a[0]], []);
}
```

There is still a lot of potential for improvement here. For example, you can switch to the iterative version of `Subsets` to avoid the recursion on subset generation. You can use an accumulator in `Teams` to avoid generating closures. And if you are really clever, you can eliminate many more temporary arrays by reusing the elements in the various recursively-generated arrays by shuffling them around. But I've sort of lost interest in the puzzle by now, so I won't bother.

Raymond Chen

Follow