A puzzle involving dynamic programming, or maybe it doesn't, episode 2

devblogs.microsoft.com/oldnewthing/20160118-00

January 18, 2016

Raymond Chen

Here's a programming challenge:

> Given an array of numbers, count the number of ways the array can be partitioned into three contiguous parts such that each part has the same sum. For example, given the array `{ 1, 1, -1, 0, -1, 2, 2, 0, -2, 1 }`, the answer is seven:
>
> 1. 1 = 1 = (−1) + 0 + (−1) + 2 + 2 + 0 + (−2) + 1
> 2. 1 = 1 + (−1) + 0 + (−1) + 2 = 2 + 0 + (−2) + 1
> 3. 1 = 1 + (−1) + 0 + (−1) + 2 + 2 + 0 + (−2) = 1
> 4. 1 + 1 + (−1) = 0 + (−1) + 2 = 2 + 0 + (−2) + 1
> 5. 1 + 1 + (−1) = 0 + (−1) + 2 + 2 + 0 + (−2) = 1
> 6. 1 + 1 + (−1) + 0 = (−1) + 2 = 2 + 0 + (−2) + 1
> 7. 1 + 1 + (−1) + 0 = (−1) + 2 + 2 + 0 + (−2) = 1
>
> Hint: Start with a solution to the partition problem, and use dynamic programming.

We will assume for now that empty parts are legal. (This is significant only if the array sum is zero.)

The hint to use dynamic programming is yet another red herring.

The first thing you notice is that partitioning without rearrangement is much easier than the general partition problem, since you have to study only $C(N+1, 2)$ subsets rather than 2^N. You can calculate sums of contiguous subsets very easily by keeping a running total and taking the difference between the endpoint and the starting point.

If you keep a parallel array with the running totals, you can pick out the subarrays quickly: They are the partitions where the left partition is at the point where the running total is one third of the array sum, and the right partition is at the point where the running total is two thirds of the array sum.

In our example above:

1	1	−1	0	−1	2	2	0	−2	1
1	2	1	1	0	2	4	4	2	3

Since the array total is 3, we want to find points where the running total is one third and two thirds of the total, or 1 and 2. I've highlighted them in the table above. Any time we see a 1 to the left of a 2, we have a successful partition: The first part is everything up to and including the 1; the middle part is everything after the 1 up to and including the 2; the last part is everything after the 2. For example,

1		1	−1	0	−1	2		2	0	−2	1
1		2	1	1	0	2		4	4	2	3

Therefore, to count the number of successful partitions, we need only count how many 1's come before 2's.

```
function equalthirds(a)
{
  var i;
  var total = 0;
  for (i = 0; i < a.length; i++) {
   total += a[i];
  }
  var third = total / 3;

  var firstpart = 0;
  var partitions = 0;

  var partial = 0;
  for (i = 0; i < a.length; i++) {
    partial += a[i];
    if (partial == third) firstpart++;
    if (partial == third * 2) partitions += firstpart;
  }
  return partitions;
}

equalthirds([ 1, 1, -1, 0, -1, 2, 2, 0, -2, 1 ]);
```

First, we get the total for the entire array, then calculate one third of the total. That is the target value for the partitions.

Next, we make another pass through the array tracking the running totals. Each time the partial sum reaches one third of the total, we increment the number of "successful first segments at or before this point". And each time the partial sum reaches two thirds of the

total, that means that we found a valid spot for the second segment (and therefore also the third segment), so we count all of the successful first segments so far toward our partition count.

The running time of this algorithm is $O(N)$ and the space requirements are constant. This is optimal: You must read every element of the array, because if you skipped one, then an adversary could modify it and alter the totals. (And you can't do better than constant space.)

Exercise: What changes would be necessary if zero-length parts are not legal?

Raymond Chen

Follow