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

devblogs.microsoft.com/oldnewthing/20140331-00

March 31, 2014

Raymond Chen

Here's a programming challenge:

> Evaluate the following recurrence relation efficiently for a given array $[x_0, \ldots, x_{n-1}]$ and integer k.
>
> = 3″>= 3″>
>
> $$f([x_0, x_1], \quad k) = (x_0 + x_1) / 2$$
>
> $$f([x_0, \ldots, x_{n-1}], \quad k) = (x_0 + x_{n-1}) / 2 + \quad f([x_0, \ldots, x_{n-2}], k + 1) / 2 + \quad f([x_1, \ldots, x_{n-1}],$$
>
> $$f([x_0, \ldots, x_{n-1}], \quad k) = \quad f([x_0, \ldots, x_{n-2}], k + 1) / 2 + \quad f([x_1, \ldots, x_{n-1}],$$
>
> Hint: Use dynamic programming.

In words:

- If the array has only two elements, then the result is the average of the two elements.
- If the array has more than two elements, then then the result is the sum of the following:
  - Half the value of the function evaluated on the array with the *first* element deleted and the second parameter incremented by one.
  - Half the value of the function evaluated on the array with the *last* element deleted and the second parameter incremented by one.
  - If the second parameter is even, then also add the average of the first and last elements of the original array.

The traditional approach with dynamic programming is to cache intermediate results in anticipation that the values will be needed again later. The naïve solution, therefore, would have a cache keyed by the vector and k.

My habit when trying to solve these sorts of recurrence relations is to solve the first few low-valued cases by hand. That gives me a better insight into the problem and may reveal some patterns or tricks.

$$
\begin{aligned}
f([x_0, x_1, x_2], k_{\text{even}}) &= (x_0 + x_2) / 2 + f([x_0, x_1], k_{\text{even}} + 1) / 2 + f([x_1, x_2], k_{\text{even}} + 1) / 2 \\
&= (x_0 + x_2) / 2 + f([x_0, x_1], k_{\text{odd}}) / 2 + f([x_1, x_2], k_{\text{odd}}) / 2 \\
&= (x_0 + x_2) / 2 + (x_0 + x_1) / 4 + (x_1 + x_2) / 4 \\
&= \tfrac{3}{4}x_0 + \tfrac{1}{2}x_1 + \tfrac{3}{4}x_2
\end{aligned}
$$

Even just doing this one computation, we learned a lot. (Each of which can be proved by induction if you are new to this sort of thing, or which is evident by inspection if you're handy with math.)

First observation: The function is linear in the array elements. In other words,

$$
\begin{aligned}
f(\boldsymbol{x} + \boldsymbol{y}, k) &= f(\boldsymbol{x}, k) + f(\boldsymbol{y}, k), \\
f(a\boldsymbol{x}, k) &= a\, f(\boldsymbol{x}, k).
\end{aligned}
$$

To save space and improve readability, I'm using vector notation, where adding two vectors adds the elements componentwise, and multiplying a vector by a constant multiples each component. The long-form version of the first of the above equations would be

$$f([x_0 + y_0, \ldots, x_{n-1} + y_{n-1}], k) = f([x_0, \ldots, x_{n-1}], k) + f([y_0, \ldots, y_{n-1}], k)$$

Since the result is a linear combination of the vector elements, we can work just with the coefficients and save ourselves some typing. ("Move to the dual space.") For notational convenience, we will write

$$\langle a_0, \ldots a_{n-1} \rangle = a_0\, x_0 + \cdots + a_{n-1}\, x_{n-1}$$

Second observation: The specific value of k is not important. All that matters is whether it is even or odd, and each time we recurse to the next level, the parity flips. So the second parameter is really just a boolean. That greatly reduces the amount of stuff we need to cache, as well as increasing the likelihood of a cache hit. (The naïve version would not have realized that $f(\boldsymbol{x}, k)$ can steal the cached result from $f(\boldsymbol{x}, k + 2)$.)

Our previous hand computation can now be written as

$f([x_0, x_1, x_2], \text{even})$	$= \langle \&\text{frac12}, 0, ½\rangle + f([x_0, x_1], \text{odd}) / 2 + f([x_1, x_2], \text{odd}) / 2$
	$= \langle ½, 0, ½\rangle + \langle ½, ½, 0\rangle / 2 + \langle 0, ½, ½\rangle / 2$
	$= \langle ½, 0, ½\rangle + \langle ¼, ¼, 0\rangle + \langle 0, ¼, ¼\rangle$
	$= \langle ¾, ½, ¾\rangle$

Now that we are working with coefficients, we don't need to deal with $\boldsymbol{x}$ any more! All that matters is the length of the vector. This makes our recurrences much simpler:

$f(2,$	$k)$	$=$	$\langle ½, ½\rangle$			f
$f(n,$	even)	$=$	$\langle ½, 0, \ldots, 0, ½\rangle +$	$\langle f(n{-}1, \text{odd}) / 2, 0\rangle +$	$\langle 0, f(n{-}1, \text{odd}) / 2\rangle$	f
$f(n,$	odd)	$=$		$\langle f(n{-}1, \text{even}) / 2, 0\rangle +$	$\langle 0, f(n{-}1, \text{even}) / 2\rangle$	f

Now we can carry out a few more hand computations.

$f(3, \text{odd})$	$= \langle f(2, \text{even}) / 2, 0\rangle + \langle 0, f(2, \text{even}) / 2\rangle$
	$= \langle ¼, ¼, 0\rangle + \langle 0, ¼, ¼\rangle$
	$= \langle ¼, ½, ¼\rangle$
$f(4, \text{even})$	$= \langle ½, 0, 0, ½\rangle + \langle f(3, \text{odd}) / 2, 0\rangle + \langle 0, f(3, \text{odd}) / 2\rangle$
	$= \langle ½, 0, 0, ½\rangle + \langle ⅛, ¼, ⅛, 0\rangle + \langle 0, ⅛, ¼, ⅛\rangle$
	$= \langle ⅝, ⅜, ⅜, ⅝\rangle$
$f(4, \text{odd})$	$= \langle f(3, \text{even}) / 2, 0\rangle + \langle 0, f(3, \text{even}) / 2\rangle$
	$= \langle ⅜, ¼, ⅜, 0\rangle + \langle 0, ⅜, ¼, ⅜\rangle$
	$= \langle ⅜, ⅝, ⅝, ⅜\rangle$

The interesting thing to observe here is that the recursion does not branch. When we reduce the size of the vector by one element, the recursive calls are basically identical. We have to shift the coefficients differently in order to build up the results, but the recursive call itself is unchanged. This means that we need to perform only $n{-}2$ recursive steps to compute $f(n, k)$.

Okay, now that we've studied the problem a bit, we can write the code. I'll write three versions of the function. The first computes according to the recurrence relation as originally written. We use this to verify our calculations.

```
function f1(x, k) {
 if (x.length == 2) {
  return (x[0] + x[1]) / 2;
 }
 var term = 0;
 if (k % 2 == 0) {
  term = (x[0] + x[x.length - 1]) / 2;
 }
 return term +
        f1(x.slice(0,-1), k + 1) / 2 +
        f1(x.slice(1), k + 1) / 2;
}
Immediate window:
f1([1,2,3], 0)
= 4.0
```

Okay, that matches our hand calculations, ¾·1 + ½·2 + ¾·3 = 4.

Now let's do the straight recursive version.

```
function dotproduct(a, x)
{
 var total = 0;
 for (var i = 0; i < a.length; i++) total += a[i] * x[i];
 return total;
}
function f2a(n, k)
{
 if (n == 2) return [1/2, 1/2];
 var c = new Array(n);
 for (var i = 0; i < n; i++) c[i] = 0;
 if (k % 2 == 0) {
  c[0] = 1/2;
  c[n-1] = 1/2;
 }
 var inner = f2a(n-1, k+1);
 for (var i = 0; i < n - 1; i++) {
  c[i] += inner[i] / 2;
  c[i+1] += inner[i] / 2;
 }
 return c;
}
function f2(x, k)
{
 return dotproduct(f2a(x.length, k), x);
}
Immediate window:
f2([1,2,3], 0)
= 4.0
```

Notice that there is no dynamic programming involved. **The hint in the problem statement was a red herring!**

Finally, we can eliminate the recursion by iterating `n` up from 2.

```
function f3(x, k)
{
 var c = new Array(x.length);
 for (var i = 0; i < x.length; i++) c[i] = 0;
 c[0] = 1/2;
 c[1] = 1/2;
 for (var n = 3; n <= x.length; n++) {
  var carry = 0;
  for (var i = 0; i < n; i++) {
   var nextcarry = c[i];
   c[i] = (carry + c[i]) / 2;
   carry = nextcarry;
  }
  if ((k + n + x.length) % 2 == 0) {
   c[0] += 1/2;
   c[n-1] += 1/2;
  }
 }
 return dotproduct(c, x);
}
```

I pulled a sneaky trick here in the place we test whether we are in the even or odd case. Officially, the test should be

```
  if ((k + (x.length - n)) % 2 == 0) {
```

but since we are interested only in whether the result is even or odd, we can just add the components together, because subtraction and addition have the same effect on even-ness and odd-ness. (If I really felt like micro-optimizing, I could fold `x.length` into `k`.)

Okay, now that we have our code, let's interpret the original problem.

The expression $\langle f(n, k) / 2, 0 \rangle + \langle 0, f(n, k) / 2 \rangle$ takes the vector $f(n, k)$ and averages it against a shifted copy of itself. (The word *convolution* could be invoked here.) If you think of the coefficients as describing the distribution of a chemical, the expression calculates the effect of diffusion after one time step according to the simple model "At each time step, each molecule has a 50% chance of moving to the left and a 50% chance of moving to the right." (Since the length of the vector varies with n, I'll visualize the vector drawn with center-alignment.)

The base case $\langle ½, ½ \rangle$ describes the initial conditions of the diffusion, where half of the chemicals are on the left and half are on the right. This is one time step after one unit of the chemical was placed in the center. Let's get rid of the extra term in the recurrence and focus just on the diffusion aspect:

= 3″>

$$d(2) = \langle ½, ½ \rangle$$

$$d(n) = \langle d(n-1) / 2, 0 \rangle + \langle 0, d(n-1) / 2 \rangle \quad \text{for } n \geq 3.$$

If you compute these values, you'll see that the results are awfully familiar. I've pulled out the common denominator to avoid the ugly fractions.

1 1	entire row divided by 2
1 2 1	entire row divided by 4
1 3 3 1	entire row divided by 8
1 4 6 4 1	entire row divided by 16
1 5 10 10 5 1	entire row divided by 32

Yup, it's Pascal's Triangle.

Notice that the sum across the row equals the amount we are dividing by, so that the sum of the row is always 1. (This is easy to see from the recurrence relation, since the base case establishes the property that the sum of the coordinates is 1, and the recurrence preserves it.)

This means that we can calculate the coefficients of $d(n)$ for any value of n directly, without having to calculate any of coefficients for smaller values of n. The coefficients are just row n of Pascal's triangle, which we know how to compute in $O(n)$ time.

Now we can also interpret the extra term we removed at the even steps. That term of the recurrence simulates adding a unit of chemical to the solution at every other time step, half a unit at the left edge and half at the right edge. And we can calculate these directly in the same way that we calculated the diffusion coefficients, since they basically *are* the diffusion coefficients, just with a time and location adjustment.

I pulled a fast one here. Maybe you didn't pick up on it: I'm assuming that the two parts of the recurrence unrelated to the diffusion behavior (the base condition and the extra term at the even steps) are independent and can simply be added together. You can show this by noting that given the generalized recurrence

$$f_G(2, k) = G(2, k)$$

$$f_G(n, k) = G(n, k) + \langle f_G(n-1, k + 1) / 2, 0 \rangle + \langle 0, f_G(n-1, k + 1) / 2 \rangle \text{ for } n \geq 3.$$

then $f_{G+H} = f_G + f_H$. (As before, induct on the recurrence relations.) Therefore, we can solve each of the pieces separately and just add the results together.

If I had the time and inclination, I could work out the solution for

$$C(n, i) + \Sigma_{k \text{ even}, 2 < k \le n} C(k, i) / 2^k$$

or something similar to that. Like I said, I ran out of time and inclination. But if I could come up with an efficient way to compute that value for all values of i for fixed n (and I believe there is, I'm just too lazy to work it out), then I would have an $O(n)$ solution to the original recurrence.

(Okay, so the "If I had the time" bit is a cop-out, but I sort of lost interest in the problem.)

Raymond Chen

Follow