Data Structures and Algorithms

Dynamic Programming

Introduction

Dynamic programming is simply the process of turning recursive calls in to table lookups. If a recursive function does redundant computations, the time complexity of the dynamic programming version of the algorithm can be greatly improved over the recursive version.

To implement a dynamic programming solution to a problem, start out by writing a simple recursive implementation. Once completed and debugged, apply the following steps to convert the function:

(1) count how many formal parameters are changing in the recursive call - this is the dimensionality of your table

(2) look at the range of the values sent to the recursive function - these ranges are the sizes of your table

(3) build a table of the correct dimensionality and size

(4) convert the recursion to table operations

(5) fill out the table in the direction indicated by the recursive calls

(6) move the base cases

(7) retrieve the desired output from the table

An Example

Here is an example recursive function that makes redundant computations:

```
function fib(n)
    {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
    }
```

Step 1: counting the changing parameters

There is one formal parameter of *fib* and you can see that the value changes in the recursive calls. So we will need to build a one-dimensional table.

Step 2: finding the parameter ranges

The range of the formal parameter is from 0 to n, meaning the table size has to be $n+1$.

Step 3: building the table

Lets assume the table is built, with variable t pointing to the newly built table.

Step 4: converting the recursions

Converting the recursive calls requires four things: changing the function being called to a call to a lookup function, removing those arguments that do not change, changing the formals that do change to loop variables, and changing the return to a table update. For example:

```
return fib(n-1) + fib(n-2);
```

becomes:

```
    t[i] = getTable(i-1) + getTable(i-2);
```

Note (and this is important) that the changing from n to i happens everywhere in the function body, including the base cases. The table lookup function is initially defined as:

```
    function getTable(i)
        {
        return t[i];
        }
```

Step 5: filling the table

In the recursive calls to *fib*, we note that the value of n being sent is smaller in both cases (this is not always true). So our loop for filling out the table needs to run from smaller n to larger n.

We start our loop at the smallest value of n (which is zero) and end the loop after we reach the maximum value, which is the original value of n:

```
    for (i = 0; i <= n; ++i)  //note that n is included
        ???
```

The recursive calls that were converted to table operations now become the body of the loop:

```
    for (i = 0; i <= n; ++i)
        t[i] = getTable([i-1) + getTable(i-2);
```

Step 6: moving the base cases

The base cases in the original function now serve as base cases in the table lookup function:

```
    function getTable(i)
        {
        if (i == 0)
            return 0;
        else if (i == 1)
            return 1;
        else
            return t[i];
        }
```

Step 7: retrieving the output

In this case, the desired output resides at index n in the table. Usually, this location corresponds to the initial arguments to the recursive function.

```
    return t[n];
```

The revised function

Putting it all together yields:

```
    function fib(n)
        {
        var t = makeTable(n+1);
        for (var i = 0; i <= n; ++i)
            t[i] = getTable(n-1) + getTable(n-2);
        return t[n];
        }
```

While the original *fib* function runs in exponential time and linear space, the dynamic programming version runs in linear time and constant space.

Another Example

Consider determining how many possible ways you can make change using a set of coins. For example, if you need to give back 11 cents in change, there are four ways you can do this:

- one dime and one penny
- two nickels and one penny
- one nickel and six pennies
- eleven pennies

Here is a function that does this. The first argument to the function is how much change you need to give back. The second argument is an index into the coin array; it signifies the coin you are going to use. The last argument is the array of coin values.

```
function makeChange(amount,index,coins)
    {
    if (amount == 0) return 1; //this is a legal solution
    if (amount < 0) return 0; //not a legal solution
    if (index == coins.size) return 0; //not a legal solution

    //find the number of combinations using the current coin
    var with = makeChange(amount-coins[index],index,coins);

    //find the number of combinations without using the current coin
    var without = makeChange(amount,index+1,coins);

    return with + without
    }
```

Like *fib*, this function also performs many redundant calculations.

Step 1: counting the changing parameters

There are three formal parameters, two of which change in the recursive calls. So we will need to build a two-dimensional table.

Step 2: finding the parameter ranges

The range of the formal parameter *amount* is from 0 to *amount*, meaning one dimension of the table has to be size *amount*+1. The range of the formal parameter *index* is from 0 to *coins.size*-1, yielding a dimension with magnitude *coins.size*.

Step 3: building the table

Lets assume the table is built, with variable *t* pointing to the newly built table. We will use *amount* for the rows and *index* for the columns.

```
t = makeTable(amount+1,coins.size);
```

Step 4: converting the recursions

Using just the parameters that change, we can rewrite this:

```
with = makeChange(amount-coins[index],index)
without = makeChange(amount,index+1)
return with + without;
```

as this:

```
with = getTable(a-coins[i],i);
without = getTable(a,i+1);
t[a][i] = with + without;
```

Step 5: filling the table

In the recursive calls to *makeChange*, we note that the value of *amount* being sent is the same or smaller in both cases. So our loop for filling out the rows of the table needs to run from smaller amounts to larger amounts. On the other hand, the value of index in the recursive class stays the same or becomes larger. Therefore, our loop over the columns of the table needs to run from larger indices to smaller indices.
We start our outer loop at the smallest value larger than the base cases. We end the loop after we reach the maximum value, which is *amount*:

```
    for (a = 1; a <= amount; ++a)
        ???
```

We start the inner loop at the largest legal index and end it at the smallest legal index:

```
    for (a = 1; a <= amount; ++a)
        for (i = coins.size-1; i >= 0; --i)
            ???
```

As with *fib*, the body of our nested loops is the converted recurrence:

```
    for (a = 1; a <= amount; ++a)
        for (i = coins.size-1; i >= 0; --i)
            {
            with = getTable(a-coins[i],i);
            without = getTable(amount,i+1);
            t[a][i] = with + without;
            }
```

Step 6: moving the base cases

Moving the base cases to the table lookup function yields:

```
    function getTable(a,i)
        {
        if (a == 0) return 1;          //this is a legal solution
        if (a < 0) return 0;           //not a legal solution
        if (i == coins.size) return 0; //not a legal solution
        return t[a][i];
        }
```

Note that we are referencing the coins array in the third base case, so we need to pass the coins array to *getTable*:

```
    function getTable(a,i,coins)
        {
        if (a == 0) return 1;          //this is a legal solution
        if (a < 0) return 0;           //not a legal solution
        if (i == coins.size) return 0; //not a legal solution
        return t[a][i];
        }
```

We also need to update the calls to *getTable* to pass in this extra parameter.

Step 7: retrieving the output

In this case, the desired output resides at row *amount* and column *index* in the table.

```
    return t[amount][index];
```

The revised function

Putting it all together yields:

```
    function makeChange(amount,index,coins)
        {
        var a,i;

        //build the table
        var t = makeTable(amount+1,coins.size);

        //fill out the table
        for (a = 1; a <= amount; ++a)
            for (i = coins.size-1; i >= 0; --i)
                {
                with = getTable(a-coins[i],i,coins);
                without = getTable(amount,i+1,coins);
                t[a][i] = with + without;
                }

        //return the desired result
        return t[amount][index];
        }
```

The original *makeChange* function runs in exponential time and $\Theta(a + s)$ space, where a is the maximum amount and s is the maximum number of coins. In contrast, the dynamic programming version of *makeChange* runs in $\Theta(as)$ time and constant space.

Memoization

Computing the minimal number of multiplications needed for a chain of matrix multiplications is one of those funny recursive routines in that we recur inside a loop (making permutations is another one of these funny routines). The basic idea is we find the optimal place to break a subchain of matrices to be multiplied, recurring on the left to find the minimal number of multiplications to process the left side of the break and then doing the same on the right side. For any break, we also have to calculate the number of multiplications for multiplying the resulting left-side matrix and the resulting right-side matrix:

```
    function mm(rows,cols,lo, hi)
        {
        var i;
        if (lo == hi-1) return 0;
        var best = INFINITY;
        for (i = lo+1; i < hi; ++i) //try all split points
            {
            var left = mm(rows,cols,lo,i);
            var right = mm(rows,cols,i,hi);
            //calculate muls for left-side matrix times right-side matrix
            var last = rows[lo]*cols[i-1]*cols[hi-1];
            var total = left + right + last;
            if (total < best) best = total;
            }
        return best;
        }
```

Because of the interleaved looping and recursion, turning this recursive solution into a dynamic programming solution is a bit involved. A more straightforward approach is *memoization*. With memoization, we store subproblem solutions in a table. After the base case checks are performed, we simply look to see if we have solved this problem before. If so, we return the previous solution. Otherwise, we compute the solution and store it in the table for future use:

```
    function mm2(rows,cols,lo,hi,table)
        {
        var i;
        if (lo == hi-1) return 0;
        if (table[lo][hi] != EMPTY) return table[lo][hi]; //memoized!
        var best = INFINITY;
        for (i = lo+1; i < hi; ++i)
            {
            var left = mm2(rows,cols,lo,i,table);
            var right = mm2(rows,cols,i,hi,table);
            var last = rows[lo]*cols[i-1]*cols[hi-1];
            var muls = left + right + last;
            if (muls < best) best = muls;
            }
        table[lo][hi] = best; //take a memo
        return best;
        }
```

The table that is passed into the memoized version of the function is constructed in the same way as in dynamic programming: the dimensionality is the number of changing formal parameters and the extent of a dimension is the range of values for the corresponding formal.

While the non-memoized solution takes exponential time, the memoized solution takes quadratic time. Since memoization is so simple to implement, it has supplanted dynamic programming as the *go to* method for dealing with redundant computations.