Introduction

Start out by writing the regular recursive implementation. Presumably, the recursive version performs redundant computations. Otherwise, there is no need to take a dynamic programming approach.

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. initialize the table with the base case values
5. convert the recursion to table operations
6. fill out the table in the direction indicated by the recursive calls
7. retrieve the desired output from the table

An Example

Here is an example recursive function that makes redundant computations:

```javascript
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: initializing the table

There are two base case values, zero and one, when n is zero and one, respectively. We initialize the the table, using those values of n as indices, with the associated base case values:

```javascript
t[0] = 0;
t[1] = 1;
```

Step 5: converting the recursion

Writing the recursive calls to fib as a recurrence, we get:

```javascript
fib(n) = fib(n-1) + fib(n-2);
```

Converted to table operations, we have:

```javascript
t[n] = t[n-1] + t[n-2];
```
Step 6: 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 larger than the base cases. We end the loop after we reach the maximum value, which is n:

```
for (i = 2; i <= n; ++i)
    ???
```

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

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

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

While the original fib function runs in exponential time, the dynamic programming version runs in linear time.

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 \( coins.size-1 \), meaning the other dimension of the table has to be size \( coins.size+1 \), or just \( coins.size \).

Step 3: building the table

Let’s 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.

\[
t = \text{makeTable}(amount+1,coins.size);
\]

Step 4: initializing the table

There are three base case values, one of which is legal and two of which fall out of range. We initialize the table with the legal base cases and note the illegal ones for later use.

Note that when \( amount \) is zero, the result is 1, regardless of the value of \( index \). Therefore, we need to initialize an entire row of the table, the row where \( amount \) corresponds to zero.

\[
\text{for } (c = 0; c < coins.size; ++c) \\
\quad t[0][c] = 1;
\]

Step 5: converting the recursions

Using just the parameters that change, we can write the recursive calls as this recurrence:

\[
\text{makeChange}(amount,index) = \text{makeChange}(amount-coins[index],index) + \text{makeChange}(amount,index+1)
\]

Converting this to table operations yields:

\[
t[amount][index] = t[amount-coins[index]][index] + t[amount][index+1];
\]

Step 6: filling the table

In the recursive calls to \( \text{makeChange} \), 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 call 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 \( amount \):

\[
\text{for } (r = 1; r <= amount; ++r) \\
\quad ???
\]

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

\[
\text{for } (r = 1; r <= amount; ++r) \\
\quad \text{for } (c = coins.size-1; c >= 0; ++c) \\
\quad \quad ???
\]

As with \( fib \), the body of our nested loops is the converted recurrence:

\[
\text{for } (r = 1; r <= amount; ++r) \\
\quad \text{for } (c = coins.size-1; c >= 0; ++c) \\
\quad \quad t[r][c] = t[r-coins[c]][c] + t[r][c+1];
\]

Now, the illegal base cases come into play. Both expressions \( r-coins[c] \) and \( c+1 \) may (or will) generate out-of-bounds array exceptions. We prevent this by adding in our illegal base cases and their return values.
for (r = 1; r <= amount; ++r)
    for (c = coins.size-1; c >= 0; ++c)
        {
            if (r-coins[c] < 0)
                t[r][c] = 0; //return value becomes table entry
            else if (c + 1 == coins.size)
                t[r][c] = 0; //return value becomes table entry
            else
                t[r][c] = t[r-coins[c]]+ t[r][c+1];
        }

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

            //initialize the table
            for (c = 0; c < coins.size; ++c)
                t[0][c] = 1;

            //fill out the table
            for (r = 1; r <= amount; ++r)
                for (c = coins.size-1; c >= 0; ++c)
                    {
                        if (r-coins[c] < 0)
                            t[r][c] = 0; //return value becomes table entry
                        else if (c + 1 == coins.size)
                            t[r][c] = 0; //return value becomes table entry
                        else
                            t[r][c] = t[r-coins[c]]+ t[r][c+1];
                    }

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

The original makeChange function runs in exponential time. In contrast, the dynamic programming version of makeChange runs in quadratic time.

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, recur 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;
            long 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);
                    long curr = left + right; //multiplications needed for break
                    if (curr < best)
                        best = curr;
                }
            return best;
        }

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

            //initialize the table
            for (c = 0; c < coins.size; ++c)
                t[0][c] = 1;

            //fill out the table
            for (r = 1; r <= amount; ++r)
                for (c = coins.size-1; c >= 0; ++c)
                    {
                        if (r-coins[c] < 0)
                            t[r][c] = 0; //return value becomes table entry
                        else if (c + 1 == coins.size)
                            t[r][c] = 0; //return value becomes table entry
                        else
                            t[r][c] = t[r-coins[c]]+ t[r][c+1];
                    }

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

Turning this recursive solution into a dynamic programming solution is not intuitive. 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:

```javascript
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.