The Change Problem

"The Change Store" was an old SNL skit (a pretty dumb one...) where they would say things like, "You need change for a 20? We'll give you two tens, or a ten and two fives, or four fives, etc."

If you are a dorky minded CS 2 student, you might ask yourself (after you ask yourself why those writers get paid so much for writing the crap that they do), "Given a certain amount of money, how many different ways are there to make change for that amount of money?"

Let us simplify the problem as follows:

Given a positive integer n, how many ways can we make change for n cents using pennies, nickels, dimes and quarters?

Recursively, we could break down the problem as follows:

To make change for n cents we could:
1) Give the customer a quarter. Then we have to make change for n-25 cents
2) Give the customer a dime. Then we have to make change for n-10 cents
3) Give the customer a nickel. Then we have to make change for n-5 cents
4) Give the customer a penny. Then we have to make change for n-1 cents.

If we let T(n) = number of ways to make change for n cents, we get the formula

$T(n) = T(n-25)+T(n-10)+T(n-5)+T(n-1)$

Is there anything wrong with this?

If you plug in the initial condition T(1) = 1, T(0)=1, T(n)=0 if n<0, you'll find that the values this formula produces are incorrect. (In particular, for this recurrence relation T(6)=3, but in actuality, we want T(6)=2.)

So this can not be right. What is wrong with our logic? In particular, it can been seen that this formula is an OVERESTIMATE of the actual value. Specifically, this counts certain combinations multiple times. In the above example, the one penny, one nickel combination is counted twice. Why is this the case?
The problem is that we are counting all combinations of coins that can be given out where ORDER matters. (We are counting giving a penny then a nickel separately from giving a nickel and then a penny.)

We have to find a way to NOT do this. One way to do this is IMPOSE an order on the way the coins are given. We could do this by saying that coins must be given from most value to least value. Thus, if you "gave" a nickel, afterwards, you would only be allowed to give nickels and pennies.

Using this idea, we need to adjust the format of our recursive computation:

To make change for n cents using the largest coin d, we could

1)If d is 25, give out a quarter and make change for n-25 cents using the largest coin as a quarter.
2)If d is 10, give out a dime and make change for n-10 cents using the largest coin as a dime.
3)If d is 5, give out a nickel and make change for n-5 cents using the largest coin as a nickel.

4)If d is 1, we can simply return 1 since if you are only allowed to give pennies, you can only make change in one way.

Although this seems quite a bit more complex than before, the code itself isn't so long. Let's take a look at it:

```
public static int makeChange(int n, int d) {

   if (n < 0)
      return 0;
   else if (n==0)
      return 1;
   else  {
      int sum = 0;
      switch (d)  {
         case 25: sum+=makeChange(n-25,25);
         case 10: sum+=makeChange(n-10,10);
         case 5: sum += makeChange(n-5,5);
         case 1: sum++;
       }
       return sum;
    }
}
```

There's a whole bunch of stuff going on here, but one of the things you'll notice is that the larger n gets, the slower and slower this will run, or maybe your computer will run out of stack space. Further analysis will show that many, many method calls get repeated in the course of a single initial method call.

In dynamic programming, we want to AVOID these reoccuring calls. To do this, rather than making those three recursive calls above, we could store the values of each of those in a two dimensional array.

Our array could look like this

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4
10	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6
25	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6

Essentially, each row label stands for the number of cents we are making change for and each column label stands for the largest coin value allowed to make change.

(Note: The lightly colored squares with 1, 2 and 1 are added to calculate the lightly colored square with 4, based on the recursive algorithm.)

Now, let us try to write some code that would emulate building this table by hand, from left to right.

```
public static int makeChangedyn(int n, int d) {

    // Take care of simple cases.
    if (n < 0)
       return 0;
    else if ((n>=0) && (n < 5))
       return 1;

     // Build table here.
     else {

         int[] denominations = {1, 5, 10, 25};
         int[][] table = new int[4][n+1];
```

```
        // Initialize table
        for (int i=0; i<n+1;i++)
            table[0][i] = 1;
        for (int i=0; i<5; i++) {
            table[1][i] = 1;
            table[2][i] = 1;
            table[3][i] = 1;
        }
        for (int i=5;i<n+1;i++) {
            table[1][i] = 0;
            table[2][i] = 0;
            table[3][i] = 0;
        }

        // Fill in table, row by row.
        for (int i=1; i<4; i++) {
            for (int j=5; j<n+1; j++) {
                for (int k=0; k<=i; k++) {
                    if ( j >= denominations[k])
                        table[i][j] += table[k][j - denominations[k]];
                }
            }
        }
        return table[lookup(d)][n];
    }
}
```

An alternate way to code this up is to realize that we DON'T need to add many different cases up together. Instead, we note that the number of ways to make change for n cents using denomination d can be split up into counting two groups:

1) The number of ways to make change for n cents using denominations LESS than d

2) The number of ways to make change for n cents using at least ONE coin of denomination d.

The former is simply the value in the table that is directly above the one we are trying to fill.

The latter is the value on the table that is on the same row, by d spots to the left.

Visually, consider just adding two values from our previous example:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4
10	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6
25	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6

(Also note that the lightly colored three was computed by adding the 1 and 2 that were lightly colored in the previous example.)

Here is the code to implement this slight change, just substitute this line for the for loop with k in the previous code:

```
if ( j >= denominations[i])
      table[i][j] = table[i-1][j] + table[i][j - denominations[k]];
else
      table[i][j] = table[i-1][j]
```