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

	0	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	1
5	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4
10	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6
25	1	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.

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++) { // Iterate through 3 rows
           for (int j=5; j<n+1; j++) { // Iterate through all cols
               for (int k=0; k<=i; k++) { // Iterate through all
                                          // coin possibilities.

                   // Only add if the coin isn't too big, also
                   // prevents array out of bounds errors.
                   if ( j >= denominations[k])
                       table[i][j] += table[k][j - denominations[k]];
               }
           }
       }
       return table[lookup(d)][n];
    }
}
```

Longest Increasing Sequence

Consider the following problem:

Given a sequence of numbers a_1, a_2, a_3, ... a_n, find the length of the longest subsequence of numbers b1, b2, ... bm such that a_{b1}, a_{b2}, a_{b3}, ... a_{bm}, is a strictly increasing sequence with b1 < b2 < b3 < ... < bm.

Here is an outline to a recursive solution:

Method prototype:

```
public static int maxincseq(int [] numbers, int index, int min);
```

This method should return the longest increasing sequence in the array numbers that starts from index index with all values greater than or equal to min.

For example, given the array numbers [8, 2, 4, 6, 12, 9, 5, 8], and index of 2 and a min of 5 as parameters, the method should return 2 because the longest increasing sequence with a minimum value greater than 5 in it starting from array index 2 is the sequence 6, 12. (This is tied with 6, 9, and 6, 8.)

Can you think of a way to solve this problem assuming we have a solution to the LCS problem?

Here we are given one sequence, but the LCS problem takes in two sequences as input. Can you think of a second sequence to generate along with the given sequence to input into the LCS problem? Why does this work?

The idea shown above is an example of a reduction. A reduction is where you show that one problem is solvable, given that you have a solution to another problem. (This is a simplification, but one that will do for introducing the idea.)

Now, let's consider solving the problem on its own using a recursive solution:

The maximum sequence either includes numbers[index] or does NOT include numbers[index]. (Note that it can only include numbers[index] if numbers[index] > min.)

If it does include numbers[index], then the maximum length sequence has the length

1 + maxincseq(numbers, index+1, numbers[index]).

If it does NOT include numbers[index], then the maximum length sequence has the length

maxincseq(numbers,index+1,min);

Now that we have this characterization, we can generate the following:

```
public static int maxincseq(int[] numbers, int index, int min) {

    if (index >= numbers.length)
        return 0;
    else if (numbers[index] <= min)
        return maxincseq(numbers, index+1, min);
     else
        return max(maxincseq(numbers, index+1, min),
              1+maxincseq(numbers, index+1, numbers[index]);
}
```

Use this recursive solution to come up with a dynamic programming solution to the problem. Your dynamic programming solution should only take in one parameter: the array.