Dynamic Programming

See Section 7.6,

pages 329-333 of Weiss

Question: There are some situations in which recursion can be massively inefficient. For example, the standard Fibonacci recursion

$$\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2)$$

computes the same values over and over. How many times do you think the calculation of Fib(40) computes the fact that Fib(4) = 3?

A. 36
B. 35+34
C. Hundreds of times
D. Thousands of times

Actually, the calculation of Fib(40) computes Fib(4) over 24,000,000 times.

Sometimes we can keep the advantages of recursion without this duplicated effort by just making a table of results; if the recursive function sees that a result has already been computed, it returns this value instead of recursing.

This technique goes by many names -- "function caching" (i.e, creating a cache for the recursive function), "memo-izing" (teaching the recursion to write itself memos) and Dynamic Programming, which is the preferred modern term.

The following function assumes that we have an array called Values. Since the Fibonacci numbers are all non-negative, I initialized all of the entries of Values to -1. Any non-negative entry indicates an actual value of the Fib function.

```
public static int Fib(int n) {
        if (Values[n] >= 0)
                return Values[n];
        else if (n == 0)
                return 0;
        else if (n == 1)
                return 1;
        else {
                int t = Fib(n-1) + Fib(n-2);
                Values[n] = t;
                return t;
        }
}
```

Here is another example. Joe, because he didn’t hand in his labs on time, is now working in a warehouse stuffing widgets into boxes. Because his company doesn't want to waste money on packing material, the boxes must be filled exactly to capacity.   The available boxes have capacities 25, 21, 10, 5, 1. Joe can use as many of each size as necessary. How can he choose boxes to minimize the number of boxes needed for a given order?

Question: We have capacities 25 21 10 5 1. Do we get the minimum number of boxes by always taking the largest box we can? For example 30=25+5 and that is better than 10+10+10 or 21+5+1+1+1+1.

A. Yes, taking the biggest box you can at each step always gets the minimum number of boxes.
B. No; order size 20 is an example where that doesn't work.
C. No; order size 43 is an example where that doesn't work.
D. No; we wouldn't be talking about it if the answer is yes.

Suppose someone orders 43 widgets. Joe could pack them into

6 boxes, of sizes 25, 10, 5, 1, 1, 1

or 3 boxes of sizes 21, 21, 1.

The last of these is the best choice and that doesn't use a box of size 25.

Naturally, we want to write a function that takes in an order size and computes both the minimum number of boxes needed and which boxes should be used.

I am going to assume that we have a box of size 1, so there is always a solution. It is easy to make the function slightly more complex to handle the no-solution option if you don't like that assumption.

We will solve this problem in a sequence of steps:

A. First we will find an easy recursion that answers the question: How many boxes are needed?
B. This recursion will be very inefficient, like the Fibonacci function. We will turn it into a Dynamic Program to eliminate redundant calculations.
C. Finally, we will add a function that prints the actual box sizes to use.

Suppose we have 15 items to pack and box sizes of 20, 10, 7, and 1. We can't use the largest box because it is too big, so we really have only 3 options: use the box of size 10, and find a way to pack the remaining 5 items efficiently; use a box of size 7 and find a way to pack the remaining 8 items efficiently; or use a box of size 1 and then pack the remaining 14 items.

Recursion can take the problem from there: to find the minimum number of boxes for any number of items, we loop through the possible sizes for the first box, and then recurse to find the minimum number for the remaining items. We return 1 plus the value of the recursive call for the best choice.

What is our base case? We can't do better than 1 box, so if we get a number of items that fits exactly into one of the boxes, we return 1 without recursing.

Here is our code for the first step:

```
public static int NumBoxes( int items) {

        for (int i = 0; i < BoxSizes.length; i++)
                if (BoxSizes[i] == items)
                        return 1;
        int min = items;
        for (int j = 0; j < BoxSizes.length; j++) {
                if (BoxSizes[j] < items) {
                        int t = 1+ NumBoxes(items-BoxSizes[j]);
                        if (t < min)
                                min = t;
                }
        }
        return min;
}
```

It might look like NumBoxes only recurses once, but that is because the recursive call is inside a loop. Each call to NumBoxes recurses once for each box size. If there are 10 different boxes, this could be even more inefficient than the Fibonacci function.

We will turn this into a Dynamic Program the same way we handled the Fibonacci function -- keeping an array (which we'll call *Counts*) that holds values of NumBoxes as we find them.

We can use the Java initialization of 0 since any assignment we will make to this array will be strictly positive.

We will start our call to NumBoxes by checking whether the argument has a non-zero entry in Counts. We can avoid the base cases if we just initialize the Counts entry for each box size to 1, since we can't do any better than putting the items into one box.

Here is the resulting function:

```
public static int NumBoxes( int items) {
        if (Counts[items] > 0)
                return Counts[items];
        int min = items;
        for (int j = 0; j < BoxSizes.length; j++) {
                if (BoxSizes[j] < items) {
                        int t = 1+ NumBoxes(items-BoxSizes[j]);
                        if (t < min)
                                min = t;
                }
        }
        Counts[items] = min;
        return min;
}
```

The only step left is finding the actual boxes to use to achieve the minimum we have calculated. We could build up a string or list of the right boxes as we go, but the code for that becomes obtuse. An easier solution is to just store the last box we used to achieve its minimum. For example, if we have 15 items to pack and we just chose to use a box that holds 7 of these items, then we must have already found a solution to the problem of packing 8 boxes. So we print our size 7, then go down to the last box we chose to pack 8 items. This was also a 7, so we print that and go down to the entry for 1 item, which of course is a box of size 1. Altogether we print 7 7 1, only storing one box for each number of items.

Here is the final version of NumBoxes:

```
public static int NumBoxes( int items) {
        if (Counts[items] > 0)
                return Counts[items];
        int min = items;
        int bestBox = 1;
        for (int j = 0; j < BoxSizes.length; j++) {
                if (BoxSizes[j] < items) {
                        int t = 1+ NumBoxes(items-BoxSizes[j]);
                        if (t < min) {
                                min = t;
                                bestBox = BoxSizes[j];
                        }
                }
        }
        Counts[items] = min;
        LastBox[items] = bestBox;
        return min;
}
```

Here's how we print a solution after the LastBox array is filled out:

```
public static void Print( int items ) {
        while (items > 0) {
                System.out.printf( "%d ", LastBox[items]);
                items -= LastBox[items];
        }
        System.out.println();
}
```

The initialization code is

```
Counts = new int[50];  // or maximum problem size
LastBox = new int[50];
for (int i = 0; i < BoxSizes.length; i++ ) {
            int size = BoxSizes[i];
            Counts[size] = 1;
            LastBox[size] = size;
}
```