Dynamic Programming

We have looked at several algorithms that involve recursion. In some situations, these algorithms solve fairly difficult problems efficiently, but in other cases they are inefficient because they recalculate certain function values many times. The example given in the text is the fibonacci example. Recursively we have:

```
public static int fibrec(int n) {

        if (n < 2)
                return n;
        else
                return fibrec(n-1)+fibrec(n-2);
}
```

The problem here is that lots and lots of calls to Fib(1) and Fib(0) are made. It would be nice if we only made those method calls once, then simply used those values as necessary. In fact, if I asked you to compute the 10th Fibonacci number, you would never do it using the recursive steps above. Instead, you'd start making a chart:

$F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, $F_8 = 21$, $F_9 = 34$, $F_{10} = 55$.

First you calculate F_3 by adding F_1 and F_2, then F_4, by adding F_3 and F_4, etc.

The idea of dynamic programming is to avoid making redundant method calls. Instead, one should store the answers to all necessary method calls in memory and simply look these up as necessary.

Using this idea, we can code up a dynamic programming solution to the Fibonacci number question that is far more efficient than the recursive version:

```
public static int fib(int n) {

	int[] fibnumbers = new int[n+1];
	fibnumbers[0] = 0;
	fibnumbers[1] = 1;

	for (int i=2; i<n+1;i++)
		fibnumbers[i] = fibnumbers[i-1]+fibnumbers[i-2];

	return fibnumbers[n];
}
```

The only requirement this program has that the recursive one doesn't is the space requirement of an entire array of values. (But, if you think about it carefully, at a particular moment in time while the recursive program is running, it has at least n recursive calls in the middle of execution all at once. The amount of memory necessary to simultaneously keep track of each of these is in fact at least as much as the memory the array we are using above needs.)

Usually however, a dynamic programming algorithm presents a time-space trade off. More space is used to store values, but less time is spent because these values can be looked up.

Can we do even better (with respect to memory) with our Fibonacci method above? What numbers do we really have to keep track of all the time?

```
public static int fib(int n) {

        int fibfirst = 0;
        int fibsecond = 1;

        for (int i=2; i<n+1;i++) {
                fibsecond = fibfirst+fibsecond;
                fibfirst = fibsecond - fibfirst;
        }
        return fibsecond;
}
```

So here, we calculate the nth Fibonacci number in linear time (assuming that the additions are constant time, which is actually not a great assumption) and use very little extra storage.

To see an illustration of the difference in speed, I wrote a short main to test this:

```
public static void main(String[] args) {

    long start = System.currentTimeMillis();
    System.out.println("Fib 30 = "+fib(30));
    long mid = System.currentTimeMillis();
    System.out.println("Fib 30 = "+fibrec(30));
    long end = System.currentTimeMillis();

    System.out.println("Fib Iter Time = "+(mid-start));
    System.out.println("Fib Rec Time = "+(end-mid));
  }
// Output:
// Fib Iter Time = 4
// Fib Rec Time = 258
```