Aufgabe 3

Die Methode `pKR` berechnet die n-te Primzahl ($n > 1$) kaskadenartig rekursiv und äußerst ineffizient:

```
long pKR(int n) {
long p = 2;
if (n >= 2) {

p = pKR(n - 1); // beginne die Suche bei der vorhergehenden Primzahl
int i = 0;
do {

P++; // pruefe, ob die jeweils naechste Zahl prim ist, d.h. ...
for (1=1; i < n && p % pKR(i) != 0; i++) {
} // pruefe, ob unter den kleineren Primzahlen ein Teiler ist
} while (i != n); // ... bis nur noch 1 und p Teiler von p sind
}

return p;
}
```

Überführen Sie `pKR` mittels *dynamischer Programmierung* (hier also Memoization) und mit möglichst *wenigen Änderungen* so in die *linear* rekursive Methode `pLR`, dass `pLR(n, new long[n + 1])` ebenfalls die n-te Primzahl ermittelt:

```
private long pLR(int n, long[] ps) {
ps[1] = 2;
}

/**
 * 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 2, 3, 5, 7, 11, 13, 17, 19, 23, 29
 */
public class PrimzahlDP {

  static long pKR(int n) {
    long p = 2;
    if (n >= 2) {
      p = pKR(n - 1); // beginne die Suche bei der vorhergehenden Primzahl
      int i = 0;
      do {
        p++; // pruefe, ob die jeweils naechste Zahl prim ist, d.h. ...
        for (i = 1; i < n && p % pKR(i) != 0; i++) {
        } // pruefe, ob unter den kleineren Primzahlen ein Teiler ist
      } while (i != n); // ... bis nur noch 1 und p Teiler von p sind
    }

    return p;
  }

  private long pLR(int n, long[] ps) {
    ps[1] = 2;
    return ps[0];
  }

  static void debug(int n) {
    System.out.println(String.format("%d. Primzahl: %d", n, pKR(n)));
  }

  public static void main(String[] args) {
    for (int i = 1; i <= 10; i++) {
```

```
34            debug(i);
35          }
36        }
37    }
```