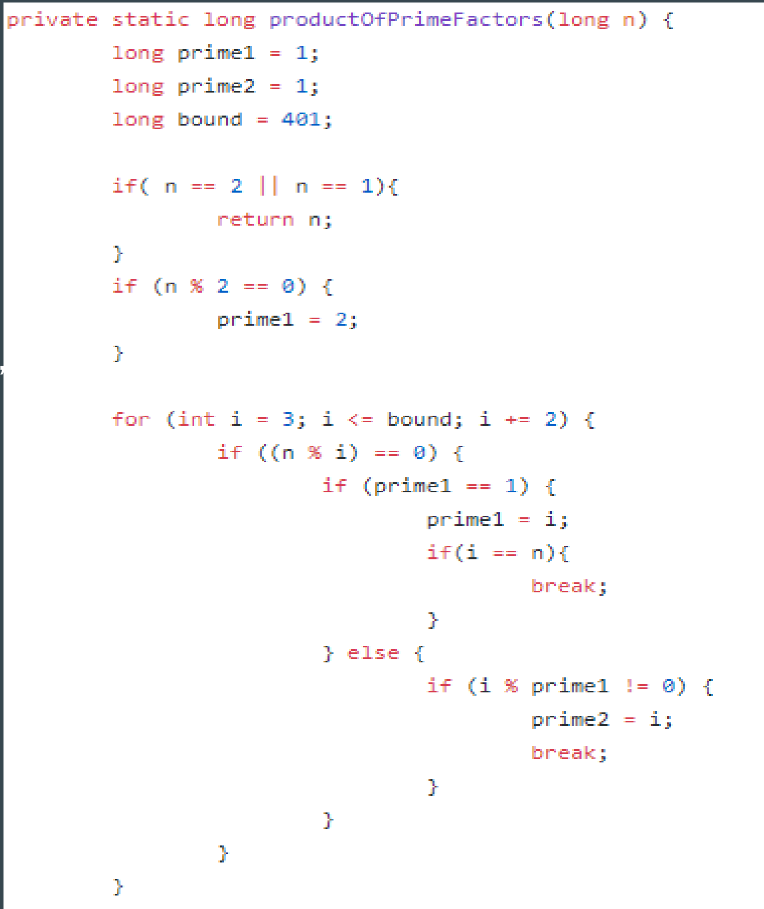
Group 1: Rocherno de Jongh and Tsz Hong Lau
3rd place in class, 5th place overall
Sorted Correctly

●●●

Description

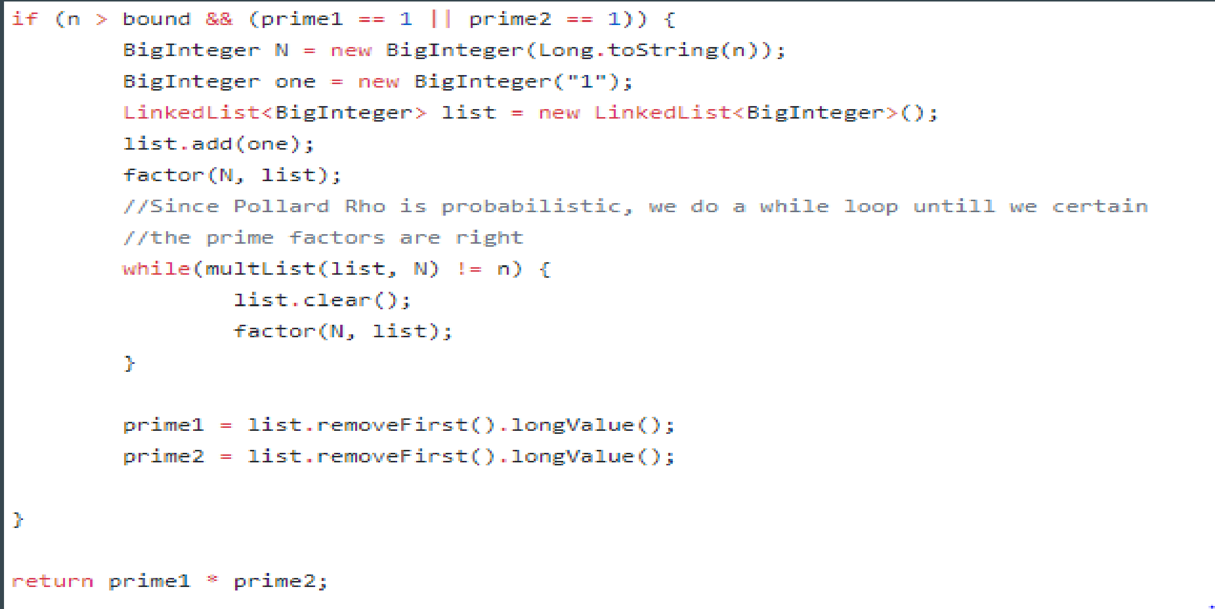
For our algorithm, we check how big the data is, and depending on the size we will either use quicksort or timsort. If it is bigger than or equal to 4000 we will use timsort, and if it’s less than we will use quicksort. These are comparison based sorting so we used a “PrimesComparator” for the comparator. When productOfPrimesFactors is called, it will try and find the first two primes using the following →



```
private static long productOfPrimeFactors(long n) {
        long prime1 = 1;
        long prime2 = 1;
        long bound = 401;

        if( n == 2 || n == 1){
                return n;
        }
        if (n % 2 == 0) {
                prime1 = 2;
        }

        for (int i = 3; i <= bound; i += 2) {
                if ((n % i) == 0) {
                        if (prime1 == 1) {
                                prime1 = i;
                                if(i == n){
                                        break;
                                }
                        } else {
                                if (i % prime1 != 0) {
                                        prime2 = i;
                                        break;
                                }
                        }
                }
        }
```

Description

If it did not find the two smallest primes within the given bound, it will call the pollard’s Rho algorithm. This algorithm is a probabilistic algorithm that finds all of the prime factors. We put all of the prime factors in a linkedlist, and get the two smallest ones.



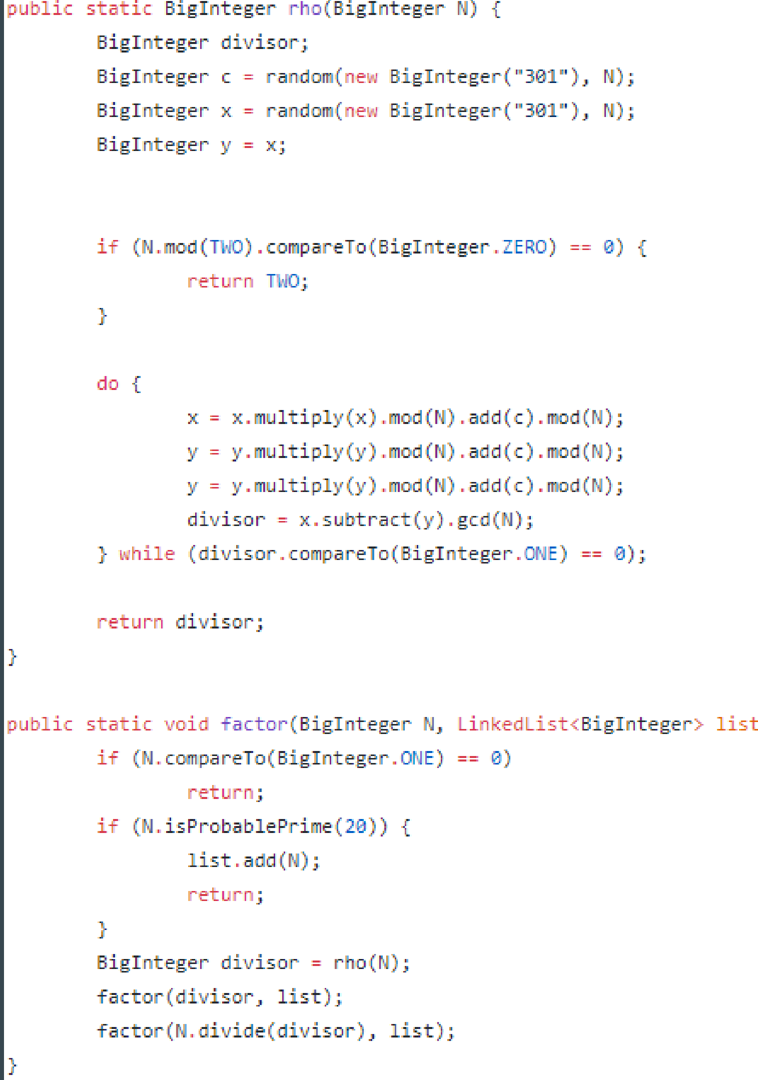
```
if (n > bound && (prime1 == 1 || prime2 == 1)) {
        BigInteger N = new BigInteger(Long.toString(n));
        BigInteger one = new BigInteger("1");
        LinkedList<BigInteger> list = new LinkedList<BigInteger>();
        list.add(one);
        factor(N, list);
        //Since Pollard Rho is probabilistic, we do a while loop untill we certain
        //the prime factors are right
        while(multList(list, N) != n) {
                list.clear();
                factor(N, list);
        }

        prime1 = list.removeFirst().longValue();
        prime2 = list.removeFirst().longValue();

}

return prime1 * prime2;
```

Efficiency

Running Time
We used two sorting algorithms, timsort and quicksort. We used quicksort for data that are less than 4000, and data greater than 4000 it uses timsort. So our theoretical worst case would be $O\left(n^2\right)$. Since we won't expect to find a sorted data easily, our expected running is $\theta(nlog(n))$, especially if we get big data, bigger than 4000, it will use timsort, which it's worst case is $O(nlog(n))$.

What we would have done differently is use the fact that every primes number over 3 lies next to a number divisible by six. We knew this fact, but we couldn’t implement it. If we could have implemented that, we could have gotten the prime factors much faster, especially for numbers that have relatively small primes for their first two primes factors.

Thank you!



```java
public static BigInteger rho(BigInteger N) {
        BigInteger divisor;
        BigInteger c = random(new BigInteger("301"), N);
        BigInteger x = random(new BigInteger("301"), N);
        BigInteger y = x;


        if (N.mod(TWO).compareTo(BigInteger.ZERO) == 0) {
                return TWO;
        }

        do {
                x = x.multiply(x).mod(N).add(c).mod(N);
                y = y.multiply(y).mod(N).add(c).mod(N);
                y = y.multiply(y).mod(N).add(c).mod(N);
                divisor = x.subtract(y).gcd(N);
        } while (divisor.compareTo(BigInteger.ONE) == 0);

        return divisor;
}

public static void factor(BigInteger N, LinkedList<BigInteger> list
        if (N.compareTo(BigInteger.ONE) == 0)
                return;
        if (N.isProbablePrime(20)) {
                list.add(N);
                return;
        }
        BigInteger divisor = rho(N);
        factor(divisor, list);
        factor(N.divide(divisor), list);
}
```

Credit:
https://introcs.cs.princeton.edu/java/99crypto/PollardRho.java.html