```
/**
 * Returns true if this BigInteger is probably prime,
 * false if it's definitely composite.
 *
 * @param  t specifies the number of tests (Fermat or Miller-Rabin)
 *         this BigInteger has to pass in order to be regarded as
 *         probably prime.
 * @return true if this BigInteger is probably prime,
 *       false if it's definitely composite.
 */
public boolean myIsProbablePrime(int t) {
 Random random = new Random();
 BigInteger a;
 for(int i = 0; i < t; i++){
  a = new BigInteger(this.bitCount(),random);
  while(this.gcd(a).intValue() != 1){
   a = new BigInteger(this.bitCount,random);
  }
  if(a.modPow(this.subtract(BigInteger.ONE),this).intValue() != 1){
   return false;
  }
 }
 for(int i:tableOfPrimes){
  if(this.mod(BigInteger.valueOf(i)).intValue() == 0){
   return false;
  }
 }
 return true;
}

/**
 * After a call to createTableOfPrimes, this array contains all the prime
 * numbers less or equal than the value max.
 */
private static int[] tableOfPrimes;

/**
 * Returns an array of all the primes less or equal than max.
 * A copy of the array is also stored in the private static array
 * tableOfPrimes.
 */
public static int[] createTableOfPrimes(int max) {
 boolean[] isPrime = new boolean[max];
 Arrays.fill(isPrime, true);
 isPrime[0] = false;
 isPrime[1] = false;

 for (int i = 2; i < Math.sqrt(max); i++){
  if (isPrime[i]){
```

```
    for (int y = i*i; y < max; y+=i){
      isPrime[y] = false;
    }
   }
  }

  List<Integer> primeList = new ArrayList<>();

  for (int i = 0; i < isPrime.length; i++){
   if (isPrime[i]){
    primeList.add(i);
   }
  }

  tableOfPrimes = primeList.stream().mapToInt(Integer::intValue).toArray();
  return tableOfPrimes;
}
```