RSA ENCRYPTION RUN-TIME ANALYSIS

Introduction

The strength of RSA encryption lies in the fact that factoring large numbers is "hard" for a computer. This means that, as the size of a number, n, gets larger, the amount of work required to factor n increases rapidly.

The relationship between the input to a program and the time it takes to run that program is the subject of *run-time analysis*. We've already studied algorithms with a variety of run-times. To refresh your memory, complete the table below, in which run-times are arranged from fastest to slowest.

Big-O run-time	Example algorithm	Example run-time calculation	Your run-time calculation
$O(1)$ (constant run-time)	Retrieving an element in an array	It takes 1 unit of time to retrieve the 675th element in an array.	How many units of time are required to retrieve the 1000th element in the array?
$O(n)$ (linear run-time)	Printing all elements in an array	It takes 50 units of time to print all 100 elements in an array.	How many units of time are required to print the first 50 elements of this array?
$O(\log n)$ (logarithmic run-time)	Conducting a binary search on an array	It takes 3 units of time to search for an item in an array containing 8 elements.	What's the greatest amount of time it could take to locate an item in an array containing 64 elements?
$O(n^2)$ (quadratic run-time, which is an example of polynomial run-time)	Computing all the pairs of elements in a set	It takes 49 units of time to compute all the pairs of elements in an array containing 7 elements.	How many units of time does it take to compute all the pairs of elements in an array containing 14 elements?
$O(2^n)$ (Exponential run-time)	Computing all the subsets of a given set (this is called computing the power set of a set)	It takes 32 units of time to compute all the subsets of an array containing 5 elements.	How many units of time does it take to compute all the subsets of an array containing 10 elements?

How long does factoring take?

The fastest run-times for factoring a number are polynomial. In the original RSA paper[1], the authors cite one factoring algorithm with run-time $O(n^{1/4})$.

The original paper also includes a table that shows how long it would take to factor numbers of different sizes. As you take a look at the table, keep in mind that the age of the universe is about 1.38×10^{10} years.

Base-10 length of the number to factor (number of digits)	Number of operations required	Time required
50	1.4×10^{10}	3.9 hours
75	9.0×10^{12}	104 days
100	2.3×10^{15}	74 years
200	1.2×10^{23}	3.8×10^{9} years
300	1.5×10^{29}	4.9×10^{15} years
500	1.3×10^{39}	4.2×10^{25} years

Check for understanding

In 1977, the authors of the original RSA paper, Rivest, Shamir, and Adleman, made the table above based on the assumption that a computer could perform 1 million operations per second. Let's assume that a computer can now perform 100 million operations per second. How long would it take to factor a number with 200 digits based on our new assumption? Give your answer in scientific notation as well as words.

Let's create a new unit of time called the univ. One univ is the current age of the universe.

1 univ = 1.38×10^{10} years.

Assuming 1 million operations per second, how many univs are required to factor a 300-digit number? Assuming 100 million operations per second?

Today's RSA encryption uses values for n that have about 600 digits. Based on the table above, about how long do you think it would take to factor a 600-digit number?

---

[1] Rivest, R.; Shamir, A.; Adleman, L. (February 1978). "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" (PDF). *Communications of the ACM*. **21** (2): 120–126. CiteSeerX 10.1.1.607.2677. doi:10.1145/359340.359342. S2CID 2873616

A method for finding prime factors

Here is a method that takes in an `int n` and returns an `ArrayList<Integer>` of `n`'s prime factors.

```
public static ArrayList<Integer> primeFactorizer(int n) {

      ArrayList<Integer> primeFactors = new ArrayList<Integer>();

      int[] primes = new int[]{2,3,5,7,11, … } // primes contains all

the primes less than 30,000.

      int i = 0;

      while ((n > 1) && (i < (int) Math.sqrt(n))) {

            while (n % primes[i] == 0) {

                  n = n / primes[i];

                  primeFactors.add(primes[i]);

            }

            i++;

      }

      if (n > 1) {

            primeFactors.add(n);

      }

      return primeFactors;

}
```

Check for understanding

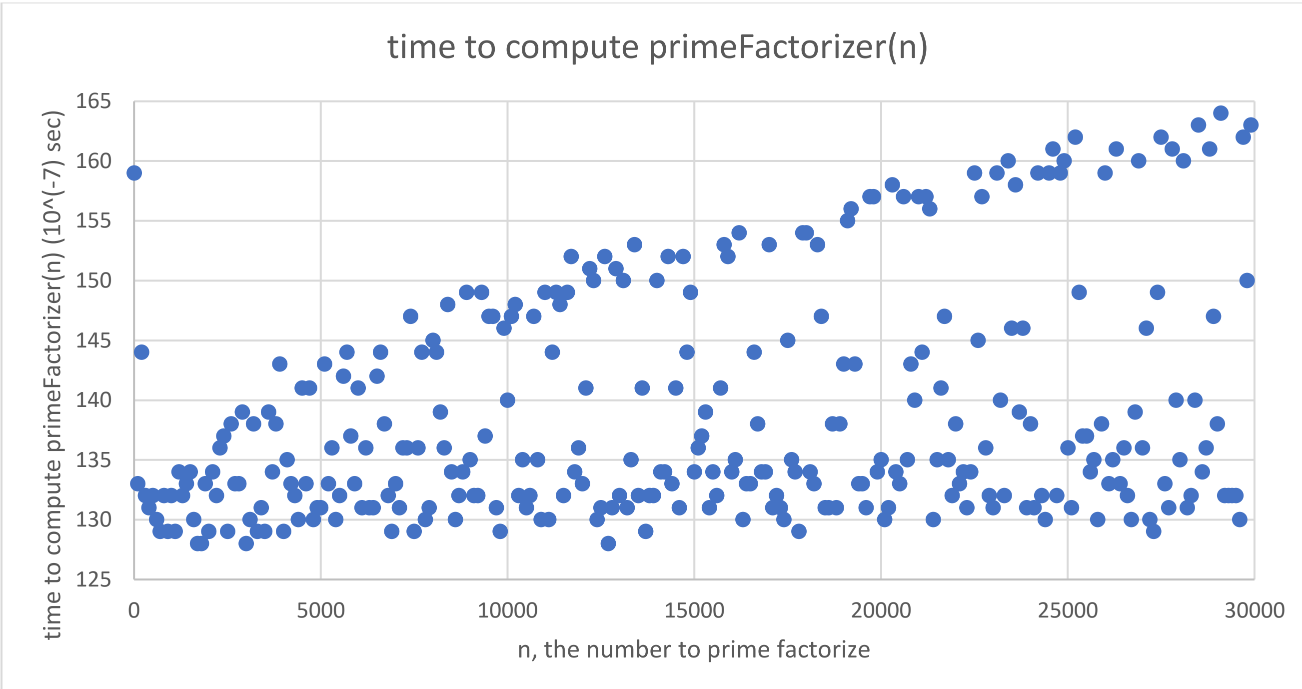
Why does this method contain the condition `(i < (int) Math.sqrt(n)))` in the outer `while` loop? To answer this, you'll need to bring in some of your mathematical knowledge about factors and multiples.

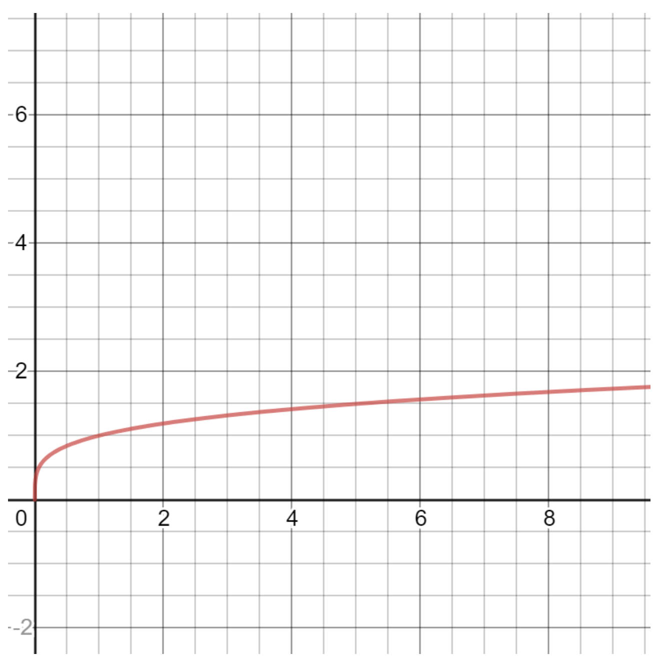
How would you describe in your own words what is meant by the condition `n % primes[i] == 0`?

What do you predict would be the output for `primeFactorizer(48)`?

Analyzing the data for run-time

The first graph below shows the time required to prime factorize some of the numbers between 2 and 30,000. The second graph shows the function $y = x^{1/4}$.





Check for understanding

What do you notice or wonder about the upper graph?

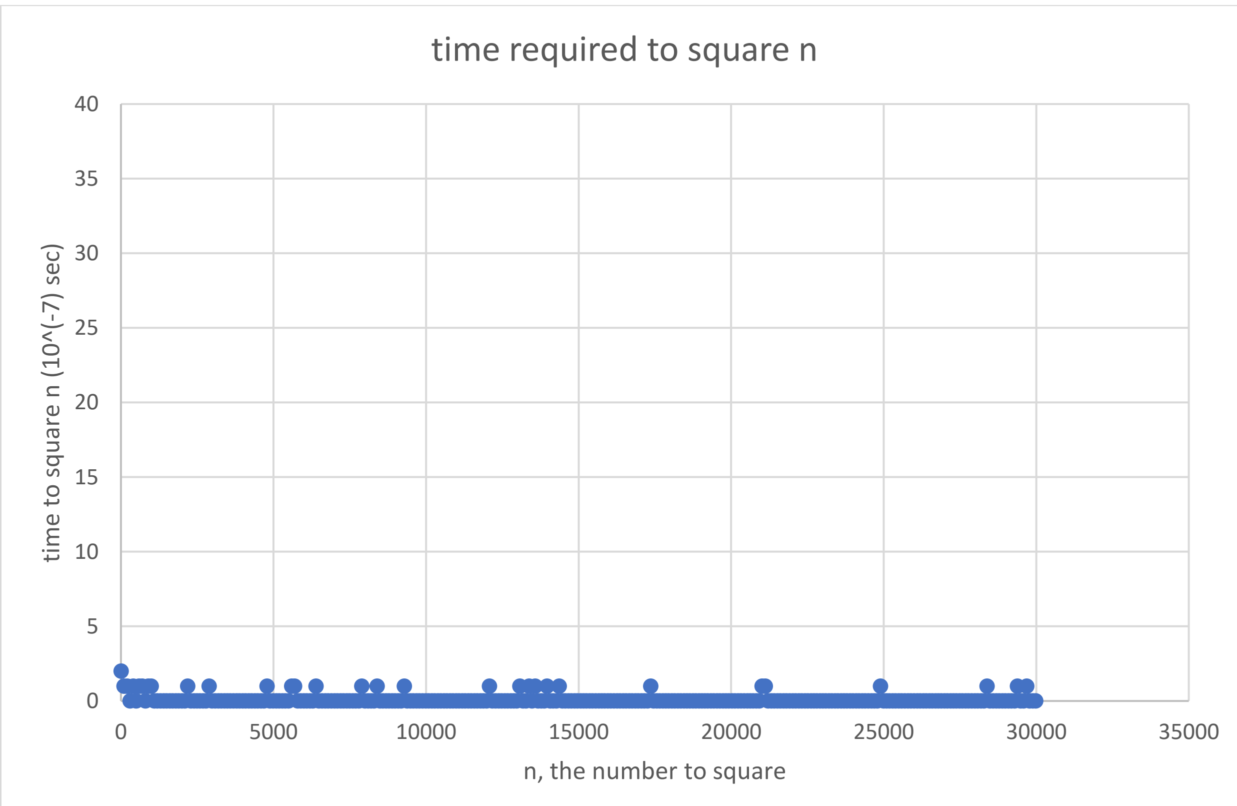
Do you see any patterns in how the data in the upper graph group together?

What is similar between the two graphs? How are they different?

Choose one difference you identified in the question above. How can you explain this difference using what you know about math and computer science?

Challenge: Write some code to collect your own data on the run-time for `primeFactorizer(n)`. How can you predict which values of `n` will require run-times along the upper curve of this graph? What values of `n` will require the shortest run-times?

The graph below shows the time required to square some of the numbers between 2 and 30,000.



Check for understanding

How does the time required to find the prime factors of a number compare to the time required to square that number?

How do the three graphs above help explain why RSA encryption is so secure?