Cody Hegwer and Tyler Nevell
Elizabeth Stade
CSCI 2824
7 April 2018

Programming Project: RSA decryption

The basis of RSA encryption is finding a composite number consisting of only two prime factors, "n". The factors we will refer to as p and q. An e value is chosen that is relatively prime to n, or their greatest common divisor is 1. A private key consisting of the original n and a new value "d" is generated only for the user to know. They are all related by the formula $de == 1 \pmod{\Phi(n)}$. The private key is hard to decrypt as "n" becomes very large and takes a lot of computational time to find its factors. The following describes our process of refining our factoring algorithm, a complexity analysis of each, and a few examples of decrypted messages.

Factoring Algorithms Used

The following algorithm utilizes the brute force method of testing all numbers up to the number n. All following "factorize" functions will use this function as a baseline but with further improvement for faster computations.

```
def factorize(n):
    for i in range(2, n-1):
        if n % i == 0:
            return i
    assert False, 'You gave me a prime number to factor'
    return -1
```

Factorize1 builds upon the original factorize, by skipping all even numbers, except for 2, increasing the step size by 2. Additionally, and here on out, we limit to checking only numbers less than the square root of n. This is based on the idea that pairs of factors of composite numbers are always "centered" around the square root of that composite number. By centered, we mean, if a number is composite, it has a factor below it's square root and a corresponding factor above the square root. This however doesn't affect the computing time for our case where we know the number we are testing is

composite, as we still arrive to the factor at the same computational pace. If it were prime, it would obviously have less numbers to check.

```
def factorize1(n):
    if n % 2 == 0:
        return 2

    i = 3
    nSquareRoot = math.sqrt(n)
    while i <= nSquareRoot:
        if n % i == 0:
            return i
        else:
            i = i + 2
    assert False, 'You gave me a prime number to factor'
    return -1
```

Factorize2 adds onto the previous algorithm by skipping all multiples of 3, once we've tested the number is not divisible by 3. The idea was to find a pattern where we can constantly skip all these multiples of 3, increase our step size, and consequently reduce our computational time.

```
def factorize2(n):
    if n % 2 == 0:
        return 2
    if n % 3 == 0:
        return 3

    i = 3
    nSquareRoot = math.sqrt(n)
    while i <= nSquareRoot:
        # skip n % i == 0:

        if n % (i+2)== 0: #i + 2*1
            return i+2
        if n % (i+4)== 0: #i + 2*2
            return i+4

        #skip n % (i+2*3) == 0:

        i = i + 6

    assert False, 'You gave me a prime number to factor'
    return -1
```

Factorize3 skips all multiples of 5 in a similar fashion to factorize2. We actually find this to run slower than factorize2 however in our complexity analysis.

```
def factorize3(n):
    if n % 2 == 0:
        return 2
    if n % 3 == 0:
        return 3
    if n % 5 == 0:
        return 5

    i = 5
    nSquareRoot = math.sqrt(n)
    while i <= nSquareRoot:
        # skip if n % i == 0:

        if n % (i+2)== 0:
            return i+2
        if n % (i+4)== 0:
            return i+4
        if n % (i+6)== 0:
            return i+6
        if n % (i+8)== 0:
            return i+8

        #skip n % (i+2*5) == 0:

        i = i + 10

    assert False, 'You gave me a prime number to factor'
    return -1
```

Factorize4 is a combination of factorize2 and factorize3 as it skips both multiples of 3 and 5, (and 2). We started out with a list of odd numbers and then proceeded to cross them out if they were multiples of 3 or 5. We demonstrate it with the following, where green is a multiple of 3, red is a multiple of 5, and purple is a multiple of both: 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45. Examining the list closely, you can begin to see a pattern. It repeats after 15 odd numbers, (30 if you include the evens we left out). It turns out 30 is also the least common multiple of 2, 3, and 5.

```
def factorize4(n):
    if n % 2 == 0:
        return 2
    if n % 3 == 0:
        return 3
    if n % 5 == 0:
        return 5

    i = 5
    nSquareRoot = math.sqrt(n)
    while i <= nSquareRoot:
        # skip if n % i == 0:

        if n % (i+2)== 0:
            return i+2
      #  if n % (i+2*2)== 0:
       #     return i+4
        if n % (i+6)== 0:
            return i+6
        if n % (i+8)== 0:
            return i+8
       # if n % (i+2*5)== 0:
        #    return i+10
        if n % (i+12)== 0:
            return i+12
        if n % (i+14)== 0:
            return i+14
        #if n % (i+2*8)== 0:
         #   return i+16
        if n % (i+18)== 0:
            return i+18
       # if n % (i+2*10)== 0:
        #    return i+20
      #  if n % (i+2*11)== 0:
       #     return i+22
        if n % (i+24)== 0:
            return i+24
        if n % (i+26)== 0:
            return i+26
        #if n % (i+2*14)== 0:
         #   return i+28
        #if n % (i+2*15)== 0:
         #   return i+30

        i = i + 30

    assert False, 'You gave me a prime number to factor'
    return -1
```

Algorithms you want to work and the rabbit hole of pursuing them.

Complexity Analysis

We foresaw issues with the process described in the project requirements and we proceeded with some slight changes. The first thing we noticed is that we are supposed to generate random numbers. We saw this as an issue as every other number is divisible by 2, every third number is divisible by 3, and every fifth number is divisible by 5. It would be hard to find a good examples of a large number to test consistently. We thought it would be beneficial to use composite numbers with two prime factors as it tends to apply to the overarching theme of the project. The next issue was the enormous amount of tests we are asked to do. We ended up doing one number for the 20 digit test, as it ended up taking ~241 seconds for factorize1 to finish. We are instructed to do this 10^(20/2) times, or 10,000,000,000 times, or 2,410,000,000,000 seconds of computation time, or 4,585,236 years of computation time. Ultimately we decided to use smaller amounts of better test examples.

We started by creating lists of random numbers that consisted of a certain number of digits in length. We accomplished this by utilizing the functions, generateRandomPrimes() and is_probable_prime(), from the rsaKeyGen python file we were given. We had observed that when generating the two random prime numbers with the length of "x" digits and "y" digits, the corresponding large composite number will typically be "x+y" digits long. An example of this is shown in the following code segment.

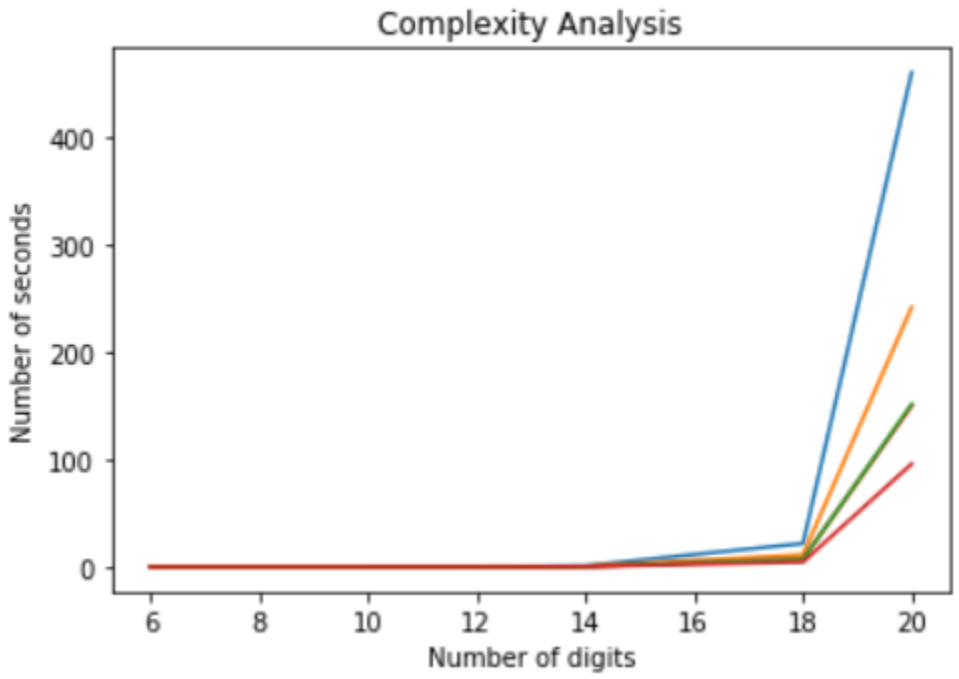
Originally we used math.ceil(numberofDigits/2) and math.floor(numberofDigits/2) to return the floor and ceiling integer values if our number of digits was odd. For example, a 7 digit number, would be produced by multiplying a 3 digit and 4 digit factor. We quickly recognized a major flaw with this approach. The computation times for finding a 6 digit number and a 7 digit number were nearly the same. (As well as an 8 digit number and a 9 digit number.) This is because our algorithms were finding the

```
numberOfDigits = 6
while numberOfDigits < 21:
    print("Current number of digits being tested:", numberOfDigits)
    tempDigitList = []
    for i in range(100): #create a list of 100 numbers of numberOfDigits long
        tempLargeComposite = generateRandomPrimes(numberOfDigits/2) * generateRandomPrimes(numberOfDigits/2)
        while len(str(tempLargeComposite)) != numberOfDigits:
            tempLargeComposite = generateRandomPrimes(numberOfDigits/2) * generateRandomPrimes(numberOfDigits/2)

        tempDigitList.append(tempLargeComposite)
```

smaller factor first and returning out of the function. Therefore it makes sense that a similar computation time was taken, as the functions were looking for a 3 digit number in both cases. This floor and ceiling method was scrapped as well as trying to create numbers of odd digit lengths.

Once we had created a sufficient amount of numbers in our tempDigitList, all consisting of the same length of digits, we tested each number with each function, and recorded the longest time for the given tempDigitList. The results produced the following plot, created using the matplotlib library in python3. (Important note; we skipped odd digits as described before)



LEGEND

Blue = factorize, **Gold** = factorize1, **Green** = factorize2 and factorize3, **Red** = factorize4

You will quickly realize that factorize2 and factorize3 are both represented by the green line. Their computation time's were nearly the same for all the tests. (If you look closely, you can barely see a line being obscured by the green line.)

We don't find the plot to be that easy to read or understand. We now try to describe the complexity of each of the functions in a different approach. Let's begin with our brute force algorithm, factorize(). Since we are checking every number up to a given number, n, we can state the complexity of the algorithm is in fact O(n). However, we specifically know the numbers we are checking are composite, and the function will terminate at a maximum of square root of n computations. Therefore our complexity can be represented by O(√n). Additionally, all of our functions can be represented by O(√n). Since we skip every even number, we essentially cut our numbers being checked in half with factorize1. Therefore we'd expect the computational time to be ½ that of factorize. In a similar fashion with factorize2, we skip a third of the numbers, leaving ⅔ that of factorize1 to check. Therefore we'd expect the computational time to be ⅔ that of factorize1. So far from what we've observed, that tends to be true. A surprise we can't seem to explain however is that factorize3, a function that computes ⅙ the amount of numbers than factorize1, computes at the same pace as factorize2, (approximately ⅔ the computational time of factorize1). Combining all the functions, factorize1, 2, and 3, we find that factorize4 computes at (½ * ⅔ * ⅔) or ~2/9 the total time compared to the original factorize.

Decoded Messages

message-1: good job. Yoda passed level 1.

message-2: Go to Papa John today and ask them for the discreet pizza structures.