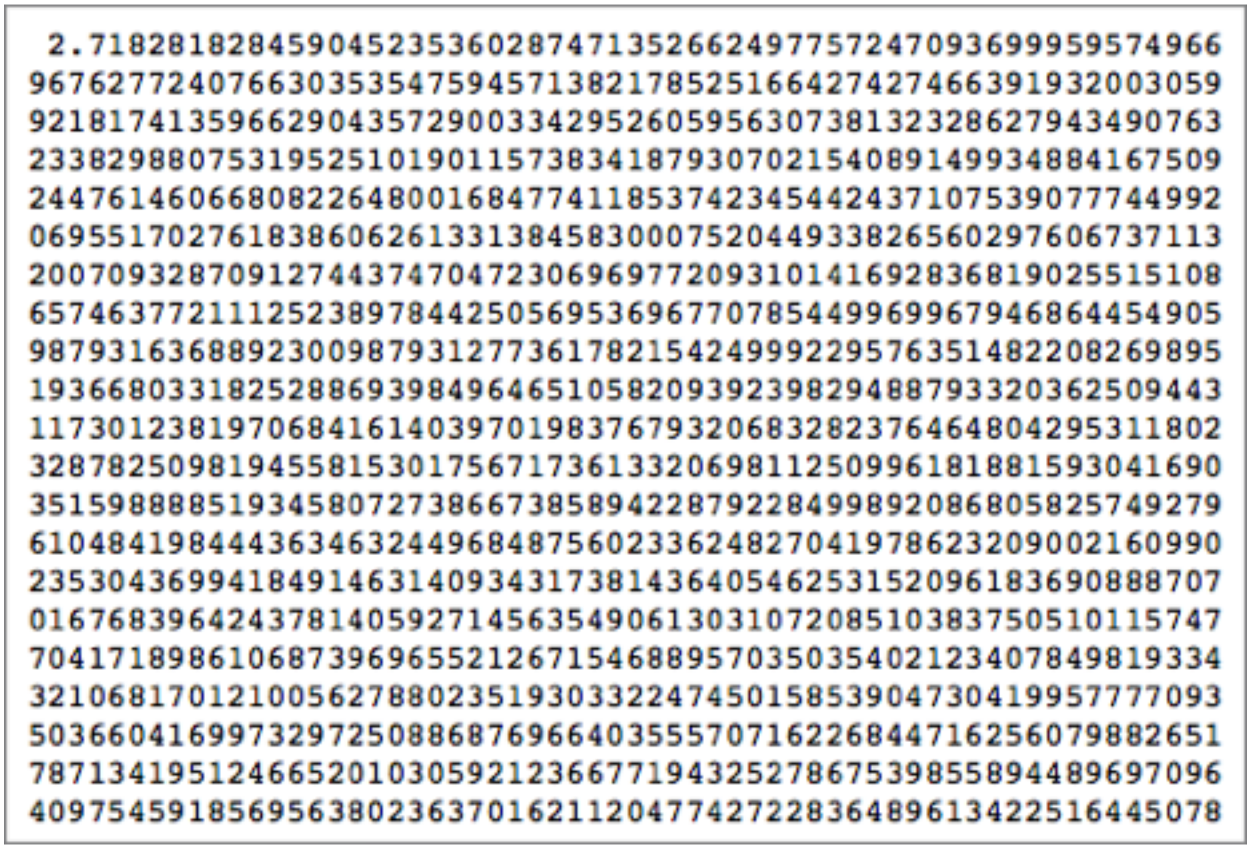
INTERNAL ASSESSMENT MATHEMATICS HIGHER LEVEL



```
 2.7182818284590452353602874713526624977572470936999595749669
9676277240766303535475945713821785251664274274663919320030599
2181741359662904357290033429526059563073813232862794349076323
3829880753195251019011573834187930702154089149934884167509244
7614606680822648001684774118537423454424371075390777449920695
5170276183860626133138458300075204493382656029760673711320070
9328709127443747047230696977209310141692836819025515108657463
7721112523897844250569536967707854499699679468644549059879316
3688923009879312773617821542499922957635148220826989519366803
3182528869398496465105820939239829488793320362509443117301238
1970684161403970198376793206832823764648042953118023287825098
1945581530175671736133206981125099618188159304169035159888851
9345807273866738589422879228499892086805825749279610484198444
3634632449684875602336248270419786232090021609902353043699418
4914631409343173814364054625315209618369088870701676839642437
8140592714563549061303107208510383750510115747704171898610687
3969655212671546889570350354021234078498193343210681701210056
2788023519303322474501585390473041995777709350366041699732972
5088687696640355570716226844716256079882651787134195124665201
0305921236677194325278675398558944896970964097545918569563802
3637016211204774272283648961342251644507818244235294863637214
17402388934412479635743702637552
```

Prime numbers in the digits of e

Naren Sivagnanadasan
Candidate ID:0804-0234
May 2014

PRIME NUMBERS IN THE DIGITS OF e

A solution to Google's problem and an examination of the distribution of large primes in the digits of e

Introduction

Google being a leading company in the technology industry in order to remain competitive needs to find the best talent in computer science, mathematics and the hard sciences. However pure recruitment does not always find the best and the brightest. In an effort to attract driven and curious people to work for Google, the company released this billboard ad.

Personally, this problem is of interests as it is an seemingly simple computer science problem, yet it is used by a company with high standards to screen talent. Therefore it is a good problem to measure ones skills . Though it seems simple, with closer inspection, this problem begins to examine not only a mathematical curiosity but with a little pursuit of elegance also examines number theory, the frequency of primes within a set of digits and efficiency of algorithms. Computer science itself is an interesting extension of a variety of fields including mathematics and engineering. Computer science has enabled mathematics, scientists, engineers and even high school students to explore the curiosities and patterns of mathematics and science and to create new things. Algorithms in computer science start to become very similar to fields in mathematics like combinatorics, probability and number theory. While looking at algorithms, it is interesting to see where the two fields overlap. For instance, for loops and recursive methods in many programming languages are comparable to summations and factorials.

```
public static int factorial(int num) {
      if (num == 0){
            return 1;
            }
      else{
      //recursive (calls itself again)
            num * factorial((num -1));
      }
      return temp;
 }
```

$$\equiv \ n!$$

At first glance, a brute force method of checking if each ten digit number x is divisible by any of the numbers from 2 to x-1 should not take that long. However, It is unclear though how far into the digits of e the program would have to go. Even though, it may take one iteration 10 seconds, 100 iterations could take 10000 and so on. Therefore a more mathematical approach to this problem may be in order.

Discussion of the Problem In-depth

From a high level this problem has two parts. Part one is to develop a sufficiently precise definition of e to test for the prime in question. The odds are extremely low that the answer would occur in the standard definition of e of most programming languages. Part two is to develop a algorithm that will test numbers for primality since a brute force method is not scalable.

e as it is defined in Java's java.lang.Math class is 2.718281828459045. Java is a object oriented programming language that is meant to be compatible across different computers.The decision to use Java will be explained later In the majority of computer science problems, this definition is sufficient, however as this problem is to find the first ten digit prime, in this definitions there are six ten digit possible numbers to test. The likelihood that the prime exists in these digits are fairly low and therefore it would be useful to generate a more precise definition. e can be calculated in a variety of ways, the first of which would be the method used by Bernoulli:

$$\lim_{n\to\infty} \left(1+\frac{1}{n}\right)^n$$

This is a method that is really easy to understand and is therefore the first definition of e many students learn, yet is not very practical in a computational sense as limits are not very easily implemented. Others issues like the lack of control of the precision of the number since the number's precision does not change linearly. It is therefore hard to predict a point at which the number is sufficiently large and the program can terminate.

Another definition of e is the continued fraction:

$$e = 2+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{4+\cfrac{1}{1+\cfrac{1}{1+\ddots}}}}}}} = 1+\cfrac{1}{0+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{4+\cfrac{1}{1+\cfrac{1}{1+\ddots}}}}}}}}},$$

A recursive method could easily take care of the continual dividing, however the main issue with this definition is the integer being added to the next fraction is not easily describable in a method. The definition is also not very clear, which for the sake of learning is not helpful.

Finally, the third method and the one chosen was the Taylor series definition of e:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

The Taylor series definition is easy to understand and the most elegant to implement as it is simply a while loop continually adding the next element to the sum of the previous ones. An implementation is below:

```
public static double e_taylor_series() {
      double temp = 1;
      int iteration = 1;
      while(1 == 1){
            temp = temp + (1/factorial(iteration))
                  //see factorial function above
            iteration++;
      }
      return temp;
}
```

Of course this method would create and infinite loop, but it would calculate the true value of e. Simply changing the while loop to a for loop and selecting a number of iterations allows the program to be terminated at the desired level of precision:

```
public static double e_taylor_series() {
      double temp = 1;
      for(int i = 1; i <= 500; i++){
            temp = temp + (1/factorial(i))

            //see factorial function above
      }
      return temp;
}
```

However a new issue arises that has not to do with mathematics but with computer science. The primitive double which is used by computers to store large decimals as defined by IEEE is a primitive with 52 bits available for fraction storage and 11 bits for exponents.[1] This means that the smallest number that can stored in a double 4.94065645841246544e-324 with 18 digits of significant figures. If e was calculated using a double primitive the program could only test 7 possible values which does not improve the test set very much over the language's definition. This is where the choice of the correct programming language can make the program very simple or very complicated. The run time of the program is very important since the precise location of the first prime is not yet known. If this program is to be scalable the runtime should not require hours. Using a language like C which is an extremely low level language will increase speed significantly since C is run without very many peripheral activities, but a solution in C because of the lack of data types other than the standard float and double for decimals would require parsing decimals into large integer arrays and adding those together in the right locations which is very difficult to implement without errors and a large amount of time committed to debugging. Meanwhile, a higher level language like Java may be slightly slower since all of its commands are run through essentially another computer called the Java Virtual Machine which increases compatibility (also a positive for writing scalable code). However, defined in the java.lang library is a class bigDecimal that allows for more digits to be stored solving this issue easily. The algorithm for e is as follows:

[1] "IEEE Standard for Binary Floating-Point Arithmetic."

```
public static BigDecimal calc_e(){
                  BigDecimal e = BigDecimal.ONE;
      //the definition of e, big decimal object is required to hold the value
                  BigDecimal factorial = BigDecimal.ONE;
      //factorial object, must be the same type as e to be able to use the big decimal
      // methods add and divide
                for(int i = 1; i < 500; i++) {
      //summation of the taylor series definition of e for the first 500 iterations /
      // 1/1+1/1+1/2….+1/499+1/500
                  factorial = factorial.multiply(new BigDecimal(i * 1.0 + ""));
                  e = e.add(new BigDecimal(1.0 + “”).divide(
                        factorial, new MathContext(10000)));
                }
                /*String eString = (e + "");
                  //supposed to print out e, computing resources run out of memory
                        long eDef = Long.parseLong(eString);
                System.out.println("e = " +eDef);*/
                return e;
             }
```

Now that a sufficiently precise value of e has be generated, it is now necessary to develop a way to test numbers for their primality. A computer could easily given sufficient time test for a number n every number from 2 to n-1 for divisibility. A slightly more efficient method would be to check each number from 2 to √n as shown below:

```
public static boolean isPrime (long n) {
                if (n <= 1){
      //if the number is less than 1 it cannot be prime
                  return false;
                }
                if (n % 2 == 0) {
      //if the number is divisable by 2 then is cannot be prime
                  return false;
                }
                long m = (long) Math.sqrt(n);
      //there is no need to test past the square root of a number
      //since the square root is always a divisor of a number

                for (long i = 3; i <= m; i += 2)
      //tests every number from three to the square root of n
                  if (n % i == 0)
                    return false;
                return true;
      }
```

However, this is not scalable. A successful program should be able to find the first 20 digit or 40 digit prime number, (provided enough memory) in a reasonable time. Therefore it is necessary to examine some other methods of determining primality of a number. A method that is quite popular is Wilson's Theorem. Wilson's Theorem states that

$$(n-1)! \equiv -1 \pmod{n}$$ [2]

or that a number n is prime if and only if the factorial of (n-1) is exactly 1 less that a multiple of n. This is an extremely powerful statement and one that is extremely easy to implement in code.

```
public static boolean prime(int n){
            int test = factorial(n-1) + 1;
            //Calculate the test number
            if (test % n == 0){
            // check divisibility
                  return true;
            }
            else{
                  return false;
            }
      }
```

Wilson's theorem is very easy to prove and is as follows:

Proof of Wilson's Theorem

The when p = 2, 3 the result is obvious, therefore p is an odd prime, p > 3. If p is a composite number then its divisors are in the set {1,2,3,…, p-1}and ((p-1)! ,p) > 1, therefore (p-1)! = -1 (mod p) for a composite number. If p is prime, then each integers in the set relatively prime to p. Each of these integers a there is another b such that ab = 1 (mod p). It is important to note that this b is unique modulo p, and that since p is prime, a = b if and only if a is 1 or p-1. Now if we omit 1 and p-1, then the others can be grouped into pairs whose product is one showing

[2]Hildebrand, A. (Director) (2011, February 16). Math 453: Definitions and Theorems

$$2*3*4*\ldots*(p-2) = 1 \pmod{p}$$

or

$$(p-2)! = 1 \pmod{p}$$

This equation multiplied by (p-1) proves the theorem.[3]

Though the implementation of the theorem was quite simple, an issue that arises is that the theorem requires factorials and the factorial of a ten digit number let a 30 or 50 digit one requires some serious computational power and this program should not depend on access to a supercomputer to successfully run.

There is a a similar theorem to Wilson's Theorem called Fermat's Little Theorem.

$$a^{p-1} \equiv 1 \bmod p \text{ s.t. } (a,p) = 1$$

Fermat's Little Theorem states that for a prime number p and any number a where the greatest common divisor is 1 that a to the power of p-1 is 1 more than a multiple of p.[4]
A corollary to this theorem is

$$a^p \equiv a \bmod p$$

where a is any number and p is a prime. The proof of this theorem is as follows:

Proof of the Fermat's Little Theorem

Proof By Induction

P0. $0^p \equiv 0 \pmod{p}$ is true for all integers.

Lemma. For any prime p,

[3]Cadwell, C. (n.d.). A proof of Wilson's Theorem.

[4]Hildebrand, A.

$$(x+y)^p \equiv x^p + y^p \pmod{p}.$$

or

$$(x+y)^p = x^p + y^p$$

for any *x* and *y* in the finite field **GF**(*p*).

To be proved later.

P1 Assume *kp* ≡ *k* (mod *p*), and consider (*k*+1)*p*. Using the lemma

$$(k+1)^p \equiv k^p + 1^p \pmod{p}.$$

By the assumption, *kp* ≡ *k* (mod *p*); => 1*p* = 1. Thus

$$(k+1)^p \equiv k+1 \pmod{p},$$

Proof of lemma. Using binomial theorem, the binomial coefficients are all integers and when 0 < *i* < *p*, neither of the terms in the denominator includes a factor of *p*, leaving the coefficient itself to possess a prime factor of *p* which must exist in the numerator, implying that

$$\binom{p}{i} \equiv 0 \pmod{p}, \qquad 0 < i < p.$$

Modulo *p*, this eliminates all but the first and last terms of the sum on the right-hand side of the binomial theorem for prime *p*. ∎

The primality of *p* is essential to the lemma; otherwise, we have examples like

$$\binom{4}{2} = 6,$$

which is not divisible by 4.[5]

Fermat's Little Theorem also has a simple programatic implementation. By using 2 for a the resources necessary for the program are minimized. *note. an extra library is required: java.lang.math*

```
public static boolean prime(int n){
        int test = Math.pow(2,n) - 2;
        //Calculate the test number
        if (test % n == 0){
        // check divisibility
            return true;
        }
        else{
            return false;
        }
    }
```

[5]Golomb, S.W., (n.d.). Combinatorial Proof of Fermat's "Little" Theorem

This solution would be quite elegant if it were not for one caveat of Fermat's Little Theorem. Fermat's Little Theorem in addition to revealing primes reveals pseudoprimes or composite numbers that have a quality that primes have, namely fulfilling Fermat's Little Theorem. Though specific bases have specific pseudoprimes, a pseudoprime that is a pseudoprime for all values of base a is called a Carmichael number. Carmichael numbers are the set of all numbers that fulfill the following equation.

$$\left(\sum_{a=1}^{p-1} a^{p-1}, p\right) = 1$$

Meaning the greatest common divisor of the summation from a to p-1 of a^{p-1} and p is 1. While it would be easy to check for Carmichael numbers in the code, it would be difficult to check for all the other pseudoprimes not confined to the set of Carmichael numbers.

A more robust form of Fermat's Little Theorem called the Lucas Primality Test is probably the most scalable version of a primality test. The test states:

$$a^{n-1} \equiv 1 \pmod{n}$$
$$a^{(n-1)/q} \not\equiv 1 \pmod{n}$$

If both of the above statements are true, where q is each prime factor of (n-1), the number is prime. This test gets rid of the false positives of Fermat's little theorem of making it not necessity to include a test for Carmichael numbers and running the risk of getting a pseudoprime returned. and could be implemented to check the output. Below is the code implementation. The code implementation of the Lucas primality test is below.

```
public class prime {
      public static boolean prime(int n){
            int test = Math.pow(2, (n - 1)) - 1;
            //Calculate the test number
            if (test % n == 0){
            // check divisibility (for find_prime_factors() see below)
                  int[] prime_factors = find_prime_factors(n);
                  for(int i = 0; i <= 40; i++ ){
                        if(prime_factors[i] != 0){
                        //make sure not to divide by zero
            int carmichael =  Math.pow(2,((n-1)/
                        prime_factors[i])) - 1;
                              if(carmichael % n == 0){
                                    return false;
                              }}}
                  return true;
            }
            else{
                  return false;
            }
      }

      private static int[] find_prime_factors(int n) {
                  int[] factors = new int[40];
                  int x = 0;
                // #2 that divide n
                while (n % 2 == 0)
                {
                   factors[x] = 2;
                    x++;
                    n = n / 2;
                }
                // n is odd a. can skip one test (Note i = i +2)
                for (int i = 3; i <= Math.sqrt(n); i = i + 2)
                {
                    // While i divides n, print i and divide n
                    while (n % i == 0)
                    {
                        factors[x] = i;
                        x++;
                        n = n / i;
                    }
                }
                // final prime
                if (n > 2){
                  factors[x] = n;
                }
                return factors;
            }}}
```

Computer Science Solution

With Lucas's Primality Test all of the algorithms should be designed to find the first ten digit prime number in the digits of e. First a Taylor series will calculate a value of e through 500 iterations of the series. Then using the Lucas Primality Test each ten consecutive digits will be sampled to test for primality. It is hard to tell if the arbitrary settings of 500 iterations and 40 prime factors will be sufficient to complete the program but since those are easily changeable, that is not a significant issue.

There will be two programs included in this paper, one use the brute force method and the other the Lucas Test to test numbers for primality, both are included in Appendix A. This will check the answers against each other. Both programs work by calculating a value for e and storing it in a bigDecimal object. This is then parsed into a String removing the first 2 and the decimal point for simplicity. Afterwards 10 character chunks are concatenated into a single integer and tested through the two prime tests. If the number is prime, the system prints it out with the number prime it is in the set and the digit after the decimal point at which the first digit appears. After running through the two programs I found that the first prime in the digits of e is 7427466391 starting at digit 99 of the string or the 100th digit. This is a fairly surprising result since it is at such a significant digit. And, when navigating to the website described on the billboard, it turns out the website was taken down. Yet, many other sources have reported the same answer.

Application of this Problem

Prime numbers are not just of interest because they are different from other numbers. Prime numbers are a mathematical singularity and therefore are of interest for research. In Pure Maths fields like Number Theory prime numbers play a huge role. Great figures in Mathematics devoted a lot of time to the pursuit of prime numbers such as Fermat, Euler and Wilson. But this problem is not just a novelty or just a point of interest. The frequency of prime numbers in irrational ones is of mathematical interest. Prime numbers are vital to the cryptography and security industry as prime numbers are used to seed encryption algorithms. Prime numbers are the basis of the up incoming currency called Bitcoins where computers solving an algorithm using prime numbers to seed the problem generate a currency. It is valued at over $200 per Bitcoin. Fermat's Theorem itself is used to validate RSA's encryption methods. Prime numbers are also a competitive endeavor. An extension of Lucas's Primality Test is the Lucas–Lehmer primality test which is used to calculate larger primes that follow the form:

$$M_n = 2^n - 1$$

These are called Mersenne primes. These primes are calculated as tests for new supercomputers and are part of a competition to calculate the largest prime number since it is easier to calculate one of these primes than any given one. [6] The most recent one was calculated January 25, 2013 and had 17,425,170 digits. e is also used to test computer systems, NASA itself calculated 5 million digits with their system[7] The algorithms outlined in this paper are standard in the realm of computer science as not only benchmark tests and competitions but as key parts of security, privacy and the economy.

Conclusion

This paper set out to find the first 10 digit prime number in the digits of e. This seems to be a trivial problem when one thinks about the steps. It is only to test each set of 10 consecutive digits for primality until the first one is found. However, issues with the inherent capabilities of computers force a more complex solution than expected. A more precise e required to begin testing which itself requires special classes to be implemented. Then the actually testing can be done with brute force, yet a solution like that is not scalable to find longer and longer primes in a set. Therefore tests like Wilson's Theorem, Fermat's Little Theorem, and the Lucas Primality Test need to be examined for possible implantation. Ultimately a solution was found in a singularly surprising spot. At the 100th digit the prime 7427466391 was found. This problem shows the overlap between computer science and mathematics and gives an introduction into Number Theory and the practices of the cryptography and security. It also was a challenging programming challenge that can help companies like Google discover new talent.

[6]Spencer, A. (Director) (2013, February 7). Adam Spencer: Why I fell in love with monster prime numbers. *TED2013*

[7]Nemiroff, R. (n.d.). *e* =. Retrieved from http://apod.nasa.gov/htmltest/gifcity/e.2mil

Appendix A

A.1
```
/** A program to find the first ten digit prime number in the digits of e
 * credit to Babji Prashanth, Chetty for help on creating long digit numbers
 *and primitive conversion algorithms.
 *
 * This program takes advantage of taylor series to develop a definition of e
 * A brute force method was used to test for primes
 * @author Naren Sivagnanadasan
 *
 */

import java.math.BigDecimal;
import java.math.MathContext;

public class first_e_prime {

        public static void main(String[] args) {
            BigDecimal eDefinition = BigDecimal.ONE;

            eDefinition = calc_e();
            //Receive the e definition from the below method

            String test_set = (eDefinition + "").substring(2);
            //remove the first 2 characters of e: "2, ."

            int x = 1;
            //set up to iterators to report digit number and number prime
            int y = 1;

            for(int i = 0; i < test_set.length() - 10; i++) {
            // iterate from the 0th digit to the 10th last digit
                  long test_number = Long.parseLong(test_set.substring(i, i +
10));             //remove the digits up to i and the digits 10 digits after
i
                  if(isPrime(test_number)) {
                        System.out.println("#" +x+ " 10 digit prime: " +
test_number+ " @ digit:" + y);
                        x = x + 1;
                        //increase the count by one if a prime is found
                  }
                  y = y + 1;
                  //increase the digit count
            }
        }
        public static BigDecimal calc_e(){
                 BigDecimal e = BigDecimal.ONE;
                 //the definition of e, big decimal object is required to
                 //hold the value
                 BigDecimal factorial = BigDecimal.ONE;
                 //factorial object, must be the same type as e to be able
                 //to use the big decimal methods add and divide
                for(int i = 1; i < 500; i++) {
                //summation of the taylor series definition of e for the
                //first 500 iterations 1/1+1/1+1/2....+1/499+1/500
                factorial = factorial.multiply(new BigDecimal(i * 1.0 + ""));
                 e = e.add(new BigDecimal(1.0 + "").divide(factorial, new
MathContext(10000)));
                }
                 /*String eString = (e + "");
            //supposed to print out e, computing resources run out of memory
                 long eDef = Long.parseLong(eString);
                System.out.println("e = " +eDef);*/
                return e;
             }

        public static boolean isPrime (long n) {
                if (n <= 1){
                  //if the number is less than 1 it cannot be prime
                  return false;
                }
                if (n % 2 == 0) {
                  //if the number is divisable by 2 then is cannot be prime
                  return false;
                }
                long m = (long) Math.sqrt(n);
                  //there is no need to test past the square root of a number
                  //since the square root is always a divisor of a number
                for (long i = 3; i <= m; i += 2)
                  //tests every number from three to the square root of n
                  if (n % i == 0)
                    return false;
                return true;
       }
}
```

A.2
```
/** A program to find the first ten digit prime number in the digits of e
 * credit to Babji Prashanth, Chetty for help on creating long digit numbers
 * and primitive conversion algorithms.
 *
 * This program takes advantage of taylor series to develop a definition of e
 * A Lucas Primality Test was used to test for primes
 * @author Naren Sivagnanadasan
 *
 */

import java.math.BigDecimal;
import java.math.MathContext;
import java.lang.Math;

public class first_e_prime {

        public static void main(String[] args) {
          BigDecimal eDefinition = BigDecimal.ONE;

         eDefinition = calc_e();
            //Receive the e definition from the below method

          String test_set = (eDefinition + "").substring(2);
            //remove the first 2 characters of e: "2, ."

          int x = 1;
            //set up to iterators to report digit number and number prime
          int y = 1;

          for(int i = 0; i < test_set.length() - 10; i++) {
            // iterate from the 0th digit to the 10th last digit
            long test_number = Long.parseLong(test_set.substring(i, i + 10));
            //remove the digits up to i and the digits 10 digits after i
            if(isPrime(test_number)) {
              System.out.println("#" +x+ " 10 digit prime: " + test_number+ "
@ digit:" + y);
              x = x + 1; //increase the count by one if a prime is found
            }
            y = y + 1; //increase the digit count
          }
        }

        public static BigDecimal calc_e(){
                  BigDecimal e = BigDecimal.ONE;
      //the definition of e, big decimal object is required to hold the value
                  BigDecimal factorial = BigDecimal.ONE;
            //factorial object, must be the same type as e to be able to use
            //the big decimal methods add and divide
                for(int i = 1; i < 500; i++) {
            //summation of the taylor series definition of e for the first
            //500 iterations 1/1+1/1+1/2....+1/499+1/500
                  factorial = factorial.multiply(new BigDecimal(i * 1.0 +
""));
                  e = e.add(new BigDecimal(1.0 + "").divide(factorial, new
MathContext(10000)));
                }
                /*String eString = (e + "");    //supposed to print out e,
computing resources run out of memory
                        long eDef = Long.parseLong(eString);
                System.out.println("e = " +eDef);*/
                return e;
             }

        public static boolean isPrime (long n) {
                  boolean prime = prime(n);
                  return prime;
        }

        public static boolean prime(long n){
                    double test = Math.pow(2,(n-1)) - 1;
      //Calculate the test number
                    if (test % n == 0){
            // check divisibility
                  int[] prime_factors = find_prime_factors(n);

                  for(int i = 0; i <= 40; i++ ){
                        if(prime_factors[i] != 0){
                        //make sure not to divide by zero
                              double carmichael =  Math.pow(2,((n-1)/
prime_factors[i])) - 1;
                              if(carmichael % n == 0){
                                    return false;
                              }}}
                  return true;
            }
            else{
                  return false;
            }
      }


      private static int[] find_prime_factors(long n) {
                  int[] factors = new int[40];
                  int x = 0;
                //  #2 that divide n
                while (n%2 == 0)
                {
                   factors[x] = 2;
                    x++;
                    n = n/2;
                }
                // n is odd a. can skip one test (Note i = i +2)
                for (int i = 3; i <= Math.sqrt(n); i = i+2)
                {
                    // While i divides n, print i and divide n
                    while (n%i == 0)
                    {
                        factors[x] = i;
                        x++;
                        n = n/i;
                    }
                }
                // final prime
                if (n > 2){
                  factors[x] = (int) n;
                }
                return factors;
            }
      }
```

Citations

Cadwell, C. (n.d.). A proof of Wilson's Theorem. *A proof of Wilson's Theorem*. Retrieved November 20, 2013, from http://primes.utm.edu/notes/proofs/Wilsons.html

Golomb, S.W., (n.d.). Combinatorial Proof of Fermat's "Little" Theorem. *JSTOR*. Retrieved November 19, 2013, from http://www.jstor.org/discover/10.2307/2309563?uid=3739568&uid=2134&uid=365564481&uid=2&uid=70&uid=3&uid=365564471&uid=3739256&uid=60&sid=21103017707643.

Hildebrand, A. (Director) (2011, February 16). Math 453: Definitions and Theorems. *Math 453. Lecture* conducted from UIUC, Urbana.

"IEEE Standard for Binary Floating-Point Arithmetic." *IEEE Xplore.* Version 2088. IEEE, n.d. Web. 20 Nov. 2013. &lt;http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=2355&gt;.

Nemiroff, R. (n.d.). *e* =. Retrieved from http://apod.nasa.gov/htmltest gifcity/e.2mil
Spencer, A. (Director) (2013, February 7). Adam Spencer: Why I fell in love with monster prime numbers. *TED2013*. Lecture conducted from TED, Long Beach, CA.