Finding Maximal Prime Gaps

Oliver Tan   Lucas Teoh   Andrew Wang

July 20, 2020

1 Introduction

A prime number, p, is a positive integer that is only divisible by 1 or p (itself). In this paper, we examine a method, using Python code, of finding the maximal prime gap, that is, the largest difference between two consecutive prime numbers p_n and p_{n+1}, such that $p_n, p_{n+1} < 10^x, x \in N$. Section 2 explores the basic outline of such a function and the fundamental steps required in developing an algorithm to find prime gaps. Section 3 explores the Miller-Rabin primality test and demonstrates how it can be implemented to increase efficiency. Section 4 explores how the Logarithmic Integral function is used in conjunction with various other functions to simplify the search for a maximal prime gap. Section 5 displays the final algorithm and discusses further potential optimisations and limitations.

2 Development of the Algorithm

In this section we demonstrate the development of the base outline of a Prime Gap finding algorithm, as well as the further alterations that were made to increase its versatility and efficiency.

2.1 Constructing a base formula

To begin, the root steps in finding a prime gap less than 10^x needed to be outlined, and they were as follows:

1. Generate a list of primes less than 10^x
2. Check the difference between each successive pair of primes and record the largest difference
3. Return the largest difference

The premise of this initial method was not to immediately generate the perfect method, but rather to set a base algorithm to build from. From these three steps, we constructed, using Python code, the first algorithm.

```
def GetPrime(n):
    primeList = [2]
    num = 3
    while len(primeList) < n:
        for p in primeList:
            if num % p == 0:
                break
        else:
            primeList.append(num)
        num += 2
    return primeList[-1]

def PrimeGap(n):
    return(GetPrime(n+1)-GetPrime(n))

q = 1
t = 0
while GetPrime(q) < 10**x:
    if PrimeGap(q)>PrimeGap(t):
        t = q
    q += 1
```

Despite the inefficiency of this initial method, it provided a solid foundation on which to build on. As there were only two main steps - the prime finding and the prime gap finding, these were to be the main focus for optimisation and improvements.

2.2 Improving prime finding

The immediate shortcomings of this method lay within the fact that as x increased, the number checks would increase exponentially, resulting in a very quick growth in computing time.

```
def GetPrime(n):
    primeList = [2]
    num = 3
    while len(primeList) < n:
        for p in primeList:
            if num % p == 0:
                break
        else:
            primeList.append(num)
        num += 2
    return primeList[-1]
```

This initial method of obtaining prime numbers involved shovelling through every odd number and searching for a divisor in the list of all primes. This involved largely numerous checks per number, and so it was an incredibly slow method that was only able to efficiently produce a list of primes up to 10^5. As such, a significantly better method was needed.

This method came to light as the Miller-Rabin primality test, which in simple terms, is more complex yet optimised prime determiner. The Miller-Rabin test will be explored in further detail in Section 3, however its most notable and beneficial trait was that the number of checks required to determine whether any given number was prime or not, remained constant. This allowed for a more improved algorithm, capable of determining the primality of even larger given numbers without increasing the number of checks required. Having now found a time-effective method of obtaining prime numbers, the next step was to improve the method for finding prime gaps.

2.3 Improving prime gap finding

In the same way that proved a hindrance in obtaining the initial set of primes, the number of primes still remained fairly substantial for higher values of x.

```
def PrimeGap(n):
    return(GetPrime(n+1)-GetPrime(n))

q = 1
t = 0
while GetPrime(q) < 10**x:
    if PrimeGap(q)>PrimeGap(t):
        t = q
    q += 1
```

This code was incredibly inefficient and wasteful, as not only did it have to comb through every single pair of primes to find the maximal gap, but with each pair of primes, it would generate a completely new list to pull the primes from. This long and arduous process could however, be removed, with the introduction of a new and significantly improved prime finding algorithm allowing for more options to be easily implemented in conjunction with the Miller-Rabin test.

In order to avoid the drawback of having to check every consecutive pair in the potentially hugely long list of primes, rough boundaries had to be made around areas where the maximal prime gap was most likely to be found. This is explored in further detail in Section 4, where a formula suggested by Marek Wolf is discussed. This formula utilises the Logarithmic Integral function and is able to produce an approximate number around which to look for such a gap, greatly reducing the range of numbers required to check.

3 The Miller-Rabin Primality Test

The Miller-Rabin primality test [5], constructed by Gary L. Miller in 1976 and refined by Michael O. Rabin in 1980, is an probabilistic algorithm used to approximate if a given number is prime or not. Using specific data points, this algorithm becomes deterministic below a certain threshold, one that is above any of x that we would need to equate.

3.1 Explaining the test

Lemma 3.1.1. *No square roots of 1 modulo p exist, such that p is prime and $p > 2$, other than those congruent to either 1 or -1 mod p.*

Proof. Suppose:

$$x^2 \equiv 1 \pmod{p}$$

It then follows, by the difference of perfect squares,

$$(x-1)(x+1) \equiv 0 \pmod{p}$$

By Euclid's Lemma, p must divide either $x-1$ or $x+1$, and so it follows that x is congruent to either 1 or -1 modulo p.

Now, for any prime number $n > 2$, it follows that $n-1$ is even. By taking the largest power of 2 from $n-1$, we can now say $n - 1 = 2^s d$, where d and s are both positive integers, and d is odd. We say that for any a in Z/n, either

$$a^d \equiv 1 \pmod{n}$$
$$\textit{or}$$
$$a^{2^r d} \equiv -1 \pmod{n}$$

where $0 \leq r \leq s-1$.
Fermat's Little Theorem states

$$a^{n-1} \equiv 1 \pmod{n}$$

for some prime number n. By Lemma 3.1.1., if we continually take square roots from a^{n-1}, we will end up with either 1 or -1. If we get -1, then we see that the second equality holds. If we do not get -1, then we are left with a^d, which will hold true to the first equality if n is indeed prime.

However, rather than checking every value of a to see if this holds for all, the Miller-Rabin test works with the opposite idea, that is, if we can find a value of a, such that

$$a^d \not\equiv 1 \pmod{n}$$
$$\text{and}$$
$$a^{2^r d} \not\equiv -1 \pmod{n}$$

then n is not prime.

3.2 Reliability

The overall correctness of this test on a large scale is reliant on the Riemann Hypothesis, which is at the time of writing, unproven. This is largely in part due to the existence of *strong liars*, which are values of a with which the equations hold, despite n being composite. However, there exist deterministic variants, which allow for only a specific set of a values to be tested, when n is within a certain limit, and these have been proven for n values up to $n < 3,317,044,064,679,887,385,961,981$, a 25 digit number. Given that the largest known prime gap is 20 digits long, using the Miller-Rabin algorithm for this specific task is acceptable, as the numbers being tested will not exceed its current known deterministic bounds.

3.3 Implementation into code

```
a_list = [2,3,5,7,11,13,17,19,23]

def MillerRabin(n):
    q = n-1
    r = 0
    while True:
       q = q>>1
       r += 1
       if q&1 == 1:
           break
    for a in a_list:
        check = True
        m = pow(a,q,n)
        if m == 1 or m == n-1:
            check = False
        else:
            for _ in range(c):
                m = pow(m,2,n)
                if m == n-1:
                    check = False
            if check:
                return False
    return True
```

The code begins with assigning q to the value $n-1$. As it moves into the `while` loop, q is converted into binary, and all powers of 2 are removed, leaving only the odd number, d. For each square root, the variable `r` (the r value) also increases. From here the code references a list, `a_list`. This list can alter based on the n value that needs to be checked, but since we will not be exceeding 10^{19} with any prime numbers, we can settle for the list shown, which has been proven to be enough to test $n < 3,825,123,056,546,413,051$. As the Miller-Rabin test looks for the inequalities, if the check variable is found to be false, for all a values in the set, the algorithm can output that n is indeed prime, but if the check variable remains true for just one a value, then n is known to be composite.

4 Estimating Prime Gaps

This section mainly explores some conjectures by Marek Wolf [14] and Daniel Shank which allow us to find an approximate range in which the maximal prime gap less than 10^x can be found. We will also explore the numerous other well established mathematical functions utilised in this approximation, and we will demonstrate how these are then implemented into the algorithm as code.

4.1 Wolf's conjecture

Wolf conjectures [2] that the first occurring largest prime gap $G(n)$ such that both primes are less than n occurs at approximately

$$G(n) \sim \frac{n}{\pi(n)}(2\ln(\pi(n)) - \ln(n) + \ln(2c_2))$$

where $\pi(n)$ is the prime counting function, and $c_2 \approx 0.660$ is the twin primes constant.

4.2 Shank's conjecture

Let $p(n)$ be the first prime of a pair of consecutive primes with a gap n. Shank conjectures [13] that

$$p(n) \sim exp(\sqrt{n})$$

where $exp(n) = e^n$. In combination with Wolf's conjecture, Shank's conjecture can be used to find the approximate location of the maximal prime gap found using Wolf's conjecture.

4.3 Mathematical Functions of Note

4.3.1 The Prime Counting Function

The Prime Counting Function[6][12], $\pi(n)$, is a function that gives the number of primes that exist less than or equal to n. It has many approximated formulas, however one of its most significant approximations is stated by the prime counting theorem:

$$\pi(n) \sim li(x)$$

where li(x) is the logarithmic integral function.

4.3.2 The Logarithmic Integral Function

The Logarithmic Integral Function [4][11], $li(x)$, is defined as

$$li(x) = \int_0^x \frac{dt}{\ln t}$$

for all $\{x|R_+^*\backslash 1\}$.

There exists a unique constant, $\mu = 1.4513692348...$ called Soldner's constant [11] where $li(x) = 0$. This allows us to rewrite the function as

$$li(x) = \int_{\mu}^{x} \frac{dt}{\ln t}$$

for $x > \mu$. It is this function that we will apply into our Python code of Wolf's conjecture, instead of the prime counting function, as this still works within out range of numbers, and is a simpler, less programmatically expensive system that produces a very similar result.

4.4 Implementation into code

```
def li(x):
    n=10000
    t=0
    d=0
    dx=(x-2)/n
    for k in range(n):
        d=2+k*dx
        t+=dx*(1/math.log(d))
    return t + 1.05

po = pow(10,x)
G = math.floor(po/li(po)*(2*math.log(li(po))-math.log(po)+0.277))

g = math.floor(math.exp(math.sqrt(G)))
```

To calculate the area of an integral, we know to add numerous slices of minimal width. By assigning the variable n to be 10000, we have now set the number of rectangular slices we will use to calculate this area. We also see that the value dx, which would usually be calculated as simply x/n, has now been changed to $(x - 2)/n$. This is done in order to avoid encountering any asymptotes or singularities which would otherwise disrupt the code. Ideally, Soldner's constant would be subtracted, however it would be unnecessarily exact so we round up to 2. Later, within the `for` loop, we see that we add 2 onto the variable d. As can be seen in the line of code following this, we will be taking the log of d, and so we add the 2 back on in order to avoid taking the log of ∞ or 1. The `for` loop allows us to take the sum of the areas of all the slices, and so we end the function by returning this value. We also add on $1.05 \approx li(2)$ to make up for the area lost initially when we subtracted 2 from x in order to avoid asymptotes.

We now translate Wolf's conjecture into code, using the logarithmic integral function as a substitute for the prime counting function. We use 0.277 as an approximation for $ln(2c_2)$, which is acceptable as we are not looking for an exact value from this test. We also apply the floor function to round to a whole number.

Finally we apply Shank's conjecture, and end up with an approximate number to which we can start searching. Because Shank's conjecture is an underestimate, there is no chance of us skipping over a large prime gap.

5 The Final Algorithm

```
import math
def li(x):
    n=10000
    t=0
    d=0
    dx=(x-2)/n
    for k in range(n):
        d=2+k*dx
        s+=dx*(1/math.log(d+dx))
        t+=dx*(1/math.log(d))
    return t + 1.05
a_list = [2,3,5,7,11,13,17,19,23]
def MillerRabin(n):
    q = n-1
    c = 0
    while q&1 != 1:
       q = q>>1
       c += 1
    for a in a_list:
        check = True
        m = pow(a,q,n)
        if m == 1 or m == n-1:
            check = False
        else:
            for _ in range(c):
                m = pow(m,2,n)
                if m == n-1:
                    check = False
            if check:
                return False
    return True
def nextPrime(u):
    y = u
    isPrime1, isPrime2 = False, False
    while y%6 != 0:
        y -= 1
    while not isPrime1 and not isPrime2:
        y+=6
        if y-1 != u:
            isPrime1 = MillerRabin(y-1)
        isPrime2 = MillerRabin(y+1)
    if isPrime1:
        return(y-1,y-1-u)
    return(y+1,y+1-u)
x = 6
po = pow(10,x)
if x == 1:
    prevHigh = 2
    placement = 5
else:
    G = math.floor(po/li(po)*(2*math.log(li(po))-math.log(po)+0.277))
    if G&1 == 1:
        G += 1
    g = math.floor(math.exp(math.sqrt(G)))
    while not MillerRabin(g):
        g += 1
    initPrime = g
    prevHigh = 0
    placement = 0
    while initPrime < po:
        d = nextPrime(initPrime)
        if d[1] > prevHigh:
            prevHigh = d[1]
            placement = d[0]
        initPrime = d[0]
print(prevHigh)
print(placement)
print(placement-prevHigh)
```

5.1 Putting it all together

Displayed at the beginning of Section 5 is the full and final version of our prime gap finding algorithm (an annotated version will be attached on the end of this document). It begins by defining the various functions discussed previously, but as we approach the latter half of the code, we begin to see how these functions are all utilised in conjunction with one another. There is also another function introduced, the `nextPrime(u)` function. This function operates on the idea that every prime is either 1 more or 1 less than a multiple of 6. It takes the number u, and checks numbers 1 greater than or 1 less than all multiples of 6 greater than u. The purpose of this function is simply to output the next prime after u, and so as soon as it finds a prime, it will return the value of that prime, as well as the difference between u and that prime.

The immediate first line of code after defining all of the functions is to choose our x value, that is, the number that defines the range of primes in which to look, 10^x. In this example we have used 10^6, as this is the highest value of x for which the algorithm can run with maximal efficiency. We then set the variable, `po`, to be 10^x. Following this, we take into account the case of $x = 1$, which fails to work with the rest of the code as it is designed to compute larger sets of primes with larger numbers in them. As the $x = 1$ case can be easily calculated, the values are simply assigned, and the remainder of the code is ignored.

With the case of $x = 1$ dealt with, we can now explore the remainder of the code, which begins by applying Wolf's conjecture. We then add 1 to the value G if it is odd, as the prime gap can only be even. With this value, we now apply Shank's conjecture to arrive at an estimated location for the prime at which to start looking.

This can be applied in combination with the function, `nextPrime(u)`, as it offers a lower bound to the prime gap by a considerable margin, and one which has been tested and proven for all $10^x, x < 15$ so that the maximal gap will not occur at a number less than g. However in order for this to work effectively, the input u for the `nextPrime(u)` function should be prime, and so we increase the value of g until we reach a prime number. From here we set the value of `initPrime` to g, and create the variables `prevHigh` and `placement`, which will track the length of the gap itself, and its position.

With the setup complete, the code can now run its course as it continues to loop until it exceeds the upper limit, checking every subsequent prime number following the initial g value. The final output of this algorithm will be the length of the gap, and the pair of consecutive primes that form the gap.

5.2 Results

Our final code was able to produce results up to $x = 9$, however, this took several minutes. $x = 6$ was where the program took less than a second to run.

All results were checked and verified to those given by the Wolfram Mathworld page on prime gaps (excluding $x = 1$, as while Wolfram describes the prime gap beneath a number to be beneath if the beginning of the gap is underneath, the document which outlined problem 5 described the whole gap as being underneath the number, leading to a slight inconsistency in values in that instance). Although we did not manage to make it to $x \geq 10$, the gaps from $x \geq 12$ were achieved by others using more optimised algorithms, more time on their hands, and supercomputers; overall, we did not too badly. Consequentially, the challenge set for the generation of prime gaps from $x > 15$ would be either require immensely different code from what we have currently, or be extraordinarily time consuming.

5.3 Limitations and Further Optimisations

Though significant improvements to the algorithm were made during the process of its construction and development, there still remained some key flaws.

The most outstanding flaw of this code lies with its limited range, as it can only output values up to $x = 8$ within a reasonable time. While this could potentially be increased if it was run on a computer with higher processing power, it would still take far too long to reach any numbers within the range of $15 \leq x \leq 20$. The code itself, while greatly optimised, still likely has room for further improvements and adjustments that could reduce computing time. For example, other similar tests have used some form of Euler's sieve, an old method of producing a list of primes quickly. It has been shown that there are faster prime determining functions available to quantum computers, however, efficient and commercially available ones will likely not be around for some time yet. The mathematics in this area is rich and developing, with world famous mathematicians like Terence Tao being highly involved. We have no doubt that in years to come, there will be further progression in this area, and consequently, further optimisation of the finding of maximal prime gaps.

References

[1] Marek Wolf. "First occurrence of a given gap between consecutive primes". In: (Apr. 1997).

[2] Marek Wolf. "Some Conjectures on the Gaps Between Consecutive Primes". In: (Sept. 1998).

[3] *"Prime gap"*. Wikipedia. URL: `https://en.wikipedia.org/wiki/Prime_gap`.

[4] *Logarithmic integral function.* Wikipedia. URL: `https://en.wikipedia.org/wiki/Logarithmic_integral_function`.

[5] *Miller–Rabin primality test.* Wikipedia. URL: `https://en.wikipedia.org/wiki/Miller-Rabin_primality_test`.

[6] *Prime-counting function.* Wikipedia. URL: https://en.wikipedia.org/wiki/Prime-counting_function.

[7] *Table of Known Maximal Gaps.* PrimePages. URL: https://primes.utm.edu/notes/GapsTable.html.

[8] Terence Tao. *Small and large gaps in the primes.* URL: https://terrytao.files.wordpress.com/2015/07/lat.pdf.

[9] *The Gaps Between Primes.* PrimePages. URL: https://primes.utm.edu/notes/gaps.html?id=research&month=primes&day=notes&year=gaps.

[10] Uni. SoloLearn. URL: https://code.sololearn.com/ciMlRPgrK8DH/#py.

[11] Eric Weisstein. *Logarithmic Integral.* URL: https://mathworld.wolfram.com/LogarithmicIntegral.html.

[12] Eric Weisstein. *Prime Counting Function.* URL: https://mathworld.wolfram.com/PrimeCountingFunction.html.

[13] Eric Weisstein. *Prime Gaps.* URL: https://mathworld.wolfram.com/PrimeGaps.html.

[14] Marek Wolf. URL: http://pracownicy.uksw.edu.pl/mwolf/.

```
#this is a fully documented version of our final code for problem 5, essentially step by step

import math #importing in one of python's inbuilt libraries, in order to avoid outsourcing one

def li(x): #li(x) is the logarithmic integral function, approximately equal to the prime counting function which we will use later
    #we will solve this integral by adding together small rectangles of the area underneath
    n=10000 #this is the number of rectangles we will use
    t=0 #this will be our output
    d=0 #this is the number which will be taken log of
    dx=(x-2)/n #this is the number which will be multiplied by to give the area. note that it is reduced by 2. this is in order to avoid
the asymptotes in li(x)
    for k in range(n): #repeating n times...
        d=2+k*dx #defining d based on x. we add the 2 back on here to avoid taking the log of infinity or 1
        t+=dx*(1/math.log(d)) #here we add the area of the rectangle onto t
    return t + 1.05 #once all the rectangles are added on, we have t, then we add on ~li(2) to make up for the area we did not
use in order to avoid the asymptotes

a_list = [2,3,5,7,11,13,17,19,23] #here we have the list of numbers used in the Miller-Rabin prime check

def MillerRabin(n): #here we begin to define the Miller-Rabin prime check
    q = n-1 #q becomes one less than n
    c = 0 #set c
    while q&1 != 1: #while the last bit of q is 0
        q = q>>1 #bitshift q by 1 to the right
        c += 1 #each time this happens add 1 to c
        #this produces n-1 = 2^c*q
    for a in a_list: #now all the numbers in a_list are checked
        check = True #used to check if conditions for primality are met
        m = pow(a,q,n) #now we set m to be a^q mod n
        if m == 1 or m == n-1: #if m meets the conditions outlied further on the document
            check = False #check is set to false; it is not composite yet
        else: #in the case that it could be composite
            for _ in range(c): #repeating c times
                m = pow(m,2,n) #m = m^2 mod n
                if m == n-1: #again, m is checked for potential primality, outlined further in the document
                    check = False #check is set to false; it is not composite yet
            if check: #if check was unchanged, this would come into effect
                return False #and return that it was composite
    return True #after all the cycles are done and it was never proven composite, return True

def nextPrime(u): #define a function that spits out the next prime, and the gap between them
    y = u #we set y to be u
    isPrime1, isPrime2 = False, False #both variables used to detect a prime are set to False
    while y%6 != 0: #while y is not divisible by 6
        y -= 1 #subtract 1
        #y is now a lower multiple of 6
    while not isPrime1 and not isPrime2: #while a prime has not been found
        y+=6 #6 is added to y
        if y-1 != u: #as long as y-1 was not the original number
            isPrime1 = MillerRabin(y-1) #check if y-1 is prime
        isPrime2 = MillerRabin(y+1) #check if y-2 is prime
    if isPrime1: #once a prime is found, if isPrime1 was the prime number
        return(y-1,y-1-u) #print the number and the gap
    return(y+1,y+1-u) #otherwise do the same for isPrime2

x = 5 #here we define x, as an example, 5
po = pow(10,x) #here we set 10^x
if x == 1: #because nextPrime(u) is not compatible with small numbers, we have to define x = 1 elsewhere
    prevHigh = 2 #as such we define the numbers that would work for x = 1
    placement = 5
else:
    G = math.floor(po/li(po)*(2*math.log(li(po))-math.log(po)+0.277)) #here we use Wolf's conjecture to estimate the gap of the
prime we are looking for
    if G&1 == 1: #if it is an odd number (which a gap cannot be)
        G += 1 #add 1
    g = math.floor(math.exp(math.sqrt(G))) #here we use Shank's Conjecture to estimate the number of which to start on by using
```

```
our estimated prime gap. we are able to combine this with nextPrime(u) because it offers a lower bound to the prime gap by a fair
margin, one which has been tested for all 10^x x<15
    while not MillerRabin(g): #in order to use nextPrime(u) effectively, u should be a prime number. here we repeat until g is
prime
        g += 1 #add 1 to g
    initPrime = g #here we initialise initPrime to our estimate of g
    prevHigh = 0 #this racks our previous high gap
    placement = 0 #this tracks the number the gap finished at
    while initPrime < po: #while initPrime is less than 10^x
        d = nextPrime(initPrime) #find the next prime after initPrime
        if d[1] > prevHigh: #if the gap between the two primes is now the largest gap
            prevHigh = d[1] #set the new highest gap to prevHigh
            placement = d[0] #get the placement of the gap
        initPrime = d[0] #initPrime is now the new prime

print(prevHigh) #once this is all done, print the highest gap
print(placement) #print the placement
print(placement-prevHigh) #print the placement the gap started at

#done!
```