Primality Testing

January 11, 2024

1 Trial Division

A program called ***Q1.py*** was used to perform the Trial Division primality test on arbitrary N. As with all programs, it is listed in Section 5.

It was used to compute primes within certain intervals shown in Table 1.

Interval	Primes
$[188000, 188200]$	188011, 188017, 188021, 188029, 188107, 188137, 188143, 188147, 188159, 188171, 188179, 188189, 188197
$[10^9, 10^9 + 200]$	$10^9 + 7$, $10^9 + 9$, $10^9 + 21$, $10^9 + 33$, $10^9 + 87$, $10^9 + 93$, $10^9 + 97$, $10^9 + 103$, $10^9 + 123$, $10^9 + 181$

Table 1: Primes in intervals

2 The Fermat Test

2.1 Implementation

A program called ***Q2.py*** has been written to perform the Fermat test on arbitrary number N and base a. It is listed in Section 5.

By applying the Fermat test in the same ranges as in Table 1, it is indeed true that all prime numbers that we found pass the Fermat test. Moreover, tabulated below in Table 2 are the found Fermat pseudoprimes along with the base at which they were detected up to $a = 13$.

Base	Fermat Pseudoprimes
2	188057
3	188191
4	188057, 188191
5	188113
7	188191
8	188057
9	188191
10	188191
12	188191

Table 2: Fermat pseudoprimes and their bases

In order to compute $a^b \bmod N$ remaining within size limits we first note that we can express b in the following form: $b = \sum_{i=0}^{k} \varepsilon_i 2^i$ and thus $a^b = \prod_{i=0}^{k} a^{\varepsilon_i 2^i}$ Now computing each $a^{2^i} \bmod N$ in turn and then multiplying it by some rolling total - again taken modulo N when appropriate - it becomes clear that N^2 is the maximum size of any integer, in any stage of computation.

This will suffice for Python. However, there is an even more memory efficient method, although it is much more intensive timewise. We note that the only time we can have numbers on the order of N^2 is when computing a^{2^i} which we get from $\left(a^{2^{i-1}}\right)^2$. Setting $c = a^{2^{i-1}}$ we can find c^2 by splitting this up in a similar fashion to before:

$$c^2 = c \cdot \sum_{i=0}^{k} \varepsilon_i 2^i = \sum_{i=0}^{k} \varepsilon_i (c \cdot 2^i)$$

Each of these $c \cdot 2^i \bmod N$ are computable with a maximal value in any given computation being $2N$.

2.2 Algorithmic Complexity

Using the formula discussed above:

$$a^b = \prod_{i=0}^{k} a^{\varepsilon_i 2^i}$$

We can compute the number of necessary operations it will take to produce an answer, assuming addition, multiplication, taking remainders and comparison statements all take one operation.

Firstly we must compute all values of $a^{2^i} \bmod N$ of which there will be at most $\lceil \log_2(b) \rceil$ values. Each value is computed by squaring the previous one and then taking a remainder. So we have $2\lceil \log_2(b) \rceil$ operations.

For each of the ε_i we must check if it is 0 or 1 and then multiply a running total by a^{2^i} if necessary and then take a remainder. So this yields us a maximum of $3\lceil \log_2(b) \rceil$

In total we have a theoretical maximum of $5\lceil \log_2(b) \rceil \leq 5\lceil \log_2(N) \rceil$ operations. Other necessary minor computations such as calculating each of the ε_i are tasks that are, at worst, $O(\log(N))$. With this we can conclude that performing the Fermat test is an $O(\log(N))$ time complexity algorithm.

2.3 Finding Carmichael Numbers

A program called ***Q3.py*** has been written to compute both Fermat pseudoprimes of a certain base and Carmichael numbers. It computes the Fermat test and then vets if it is indeed prime via the Trial Division test. It is listed in Section 5.

The results of this program are listed in Table 3.

Type	Pseudoprimes
Base 2	341 561 645 1105 1387 1729 1905 2047 2465 2701 2821 3277 4033 4369 4371 4681 5461 6601 7957 8321 8481 8911 10261 10585 11305 12801 13741 13747 13981 14491 15709 15841 16705 18705 18721 19951 23001 23377 25761 29341 30121 30889 31417 31609 31621 33153 34945 35333 39865 41041 41665 42799 46657 49141 49981 52633 55245 57421 60701 60787 62745 63973 65077 65281 68101 72885 74665 75361 80581 83333 83665 85489 87249 88357 88561 90751 91001 93961 101101 104653 107185 113201 115921 121465 123251 126217 129889 129921 130561 137149 149281 150851 154101 157641 158369 162193 162401 164737 172081 176149 181901 188057 188461 194221 196021 196093 204001 206601 208465 212421 215265 215749 219781 220729 223345 226801 228241 233017 241001 249841 252601 253241 256999 258511 264773 266305 271951 272251 275887 276013 278545 280601 282133 284581 285541 289941 294271 294409 314821 318361 323713 332949 334153 340561 341497 348161 357761 367081 387731 390937 396271 399001 401401 410041 422659 423793 427233 435671 443719 448921 449065 451905 452051 458989 464185 476971 481573 486737 488881 489997 493697 493885 512461 513629 514447 526593 530881 534061 552721 556169 563473 574561 574861 580337 582289 587861 588745 604117 611701 617093 622909 625921 635401 642001 647089 653333 656601 657901 658801 665281 665333 665401 670033 672487 679729 680627 683761 688213 710533 711361 721801 722201 722261 729061 738541 741751 742813 743665 745889 748657 757945 769567 769757 786961 800605 818201 825265 831405 838201 838861 841681 847261 852481 852841 873181 875161 877099 898705 915981 916327 934021 950797 976873 983401 997633
Absolute/ Carmichael Numbers	561 1105 1729 2465 2821 6601 8911 10585 15841 29341 41041 46657 52633 62745 63973 75361 101101 115921 126217 162401 172081 188461 252601 278545 294409 314821 334153 340561 399001 410041 449065 488881 512461 530881 552721 656601 658801 670033 748657 825265 838201 852841 997633

Table 3: Absolute and Fermat 2-pseudoprimes < 1 million

To check for Carmichael numbers, we can apply a crude test of checking each base in turn. However, we can speed up this process, as we only need to check base p where p is prime.

This is due to the fact that:

$$p^{N-1}q^{N-1} \equiv pq^{N-1} \not\equiv 1 \mod N$$

$$\Rightarrow p^{N-1} \text{ or } q^{N-1} \not\equiv 1 \bmod N$$

So if a composite base yields an issue, it would already have been flagged by one of its factors. Moreover if N is a pseudoprime for all prime numbers $< N$, it is immediate that N is a Carmichael number.

If a number is not a Carmichael number it must be flagged by by some p, the first such p we found is listed in Table 4. As such there are no Carmichael numbers listed; this is acceptable as we have all of them in Table 3.

Base	**Fermat 2-Pseudoprimes**
3	341 1387 2047 3277 4033 4369 4681 5461 7957 8321 10261 11305 13741 13747 13981 14491 15709 16705 19951 23377 30121 30889 31417 31609 34945 35333 39865 41665 42799 49981 57421 60701 60787 65077 65281 68101 74665 80581 85489 88357 91001 113201 121465 123251 129889 130561 137149 149281 150851 158369 162193 164737 181901 188057 194221 196021 196093 208465 215749 219781 220729 223345 233017 241001 249841 253241 256999 258511 264773 266305 271951 272251 275887 280601 284581 285541 294271 318361 323713 341497 348161 357761 367081 387731 390937 396271 401401 422659 423793 435671 443719 448921 452051 458989 464185 476971 481573 486737 489997 493697 493885 513629 514447 556169 574861 580337 582289 587861 588745 604117 611701 617093 625921 635401 642001 647089 657901 665333 665401 672487 679729 680627 683761 688213 710533 711361 722201 722261 729061 738541 741751 742813 743665 745889 757945 769567 769757 800605 818201 838861 841681 847261 852481 875161 877099 898705 916327 934021 950797 976873 983401
5	2701 4371 8481 18721 23001 25761 31621 33153 49141 83333 87249 88561 90751 93961 104653 129921 154101 157641 176149 204001 206601 212421 226801 228241 276013 282133 289941 332949 427233 526593 534061 563473 574561 622909 653333 665281 786961
7	645 1905 12801 18705 55245 72885 83665 107185 215265 451905 831405 873181
11	721801

Table 4: Base at which Fermat 2-pseudoprimes are detected

3 The Euler Test

3.1 Euler Pseudoprimes

A program called ***Q4.py*** has been written to compute the Jacobi symbol, perform the Euler test, compute Euler pseudoprimes, and absolute Euler pseudoprimes. It is listed in Section 5.

We note that there are no absolute Euler pseudoprimes. This was a result established during lectures. Due to this, Table 5, which gives the base that 2-pseudoprimes fail the Euler test, must contain every pseudoprime.

Base	Euler 2-Pseudoprimes
3	1105 2047 2465 3277 4033 4681 6601 8321 16705 30121 34945 42799 65281 74665 80581 85489 88357 90751 104653 113201 130561 149281 158369 162401 164737 188057 196093 208465 220729 223345 233017 252601 253241 256999 266305 271951 278545 280601 323713 340561 348161 357761 390937 410041 448921 458989 476971 486737 489997 493697 514447 552721 580337 588745 625921 635401 647089 658801 665281 683761 711361 741751 745889 800605 818201 825265 838861 841681 852481 852841 875161 877099 916327 976873 983401
5	561 8481 25761 29341 33153 46657 49141 52633 87249 126217 129921 294409 314821 334153 427233 526593 670033 721801 748657 873181 997633
7	1905 10585 12801 18705 62745 75361 115921 215265
11	1729 172081 488881 530881
13	15841 399001 449065
17	41041 656601
29	838201

Table 5: Base at which Euler 2-pseudoprimes are detected

3.2 Algorithmic Complexity

We can compute the complexity working with the same assumptions as before.

For the first part of the Euler test we must compute $a^{\frac{N-1}{2}} \bmod N$, we already know from the Fermat Test that this is an $O(\log(N))$ task. For the second part we must compute the Jacobi symbol. Any operation which we already have a formula for, ie computing $\left(\frac{2}{a}\right)$, $\left(\frac{-1}{a}\right)$ and flipping $\left(\frac{a}{b}\right)$ are all $O(1)$ operations.

We note that when we flip the Jacobi symbol we apply one step of Euclid's algorithm (finding the remainder mod b) and then factor out by all powers of 2. If we begin with $\left(\frac{a}{N}\right)$ every number is bounded above by N. The worst scenario on any individual step is that the numerator is a power of 2 and so factoring it is an $O(\log(N))$ task. Then, for the worst case on the number of iterations we need to complete, we can look to Euclid's algorithm itself, which has a known complexity of $O(\log(\min(a, N)))$.

Combining these two results gives the Euler test a time complexity of $O\left(\log(N)^2\right)$. This however is grossly overcompensating as there is clearly a trade off between factoring powers of 2 and the remaining steps needed in the algorithm.

4 The Strong Test

4.1 Implementation

A program called ***Q5.py*** has been written to compute strong pseudoprimes with respect to a given base and by extension absolute strong pseudoprimes. It is listed in Section 5.

As with Euler pseudoprimes, discussed in Section 3, we established in lectures that there are no absolute strong pseudoprimes.

Base	Strong 2-Pseudoprimes
3	2047 3277 4033 4681 8321 15841 29341 42799 49141 52633 65281 74665 80581 85489 88357 90751 104653 130561 196093 220729 233017 252601 253241 256999 271951 280601 314821 357761 390937 458989 476971 486737 489997 514447 580337 635401 647089 741751 800605 818201 838861 873181 877099 916327 976873 983401

Table 6: Base at which strong 2-pseudoprimes are detected

We note that every single 2-pseudoprime under 10^6 is not a base 3 pseudoprime. However, we have no reason to believe that this pattern will continue onwards to arbitary N.

4.2 Algorithmic Complexity

Again, like before, we may compute the complexity of this algorithm with the assumptions that addition, multiplication, taking remainders, and comparison statements all take one operation.

To find the complexity of the strong test, we must first compute r, s from $N - 1 = 2^r s$, which is clearly an $O(\log(N))$ task. Additionally, we must compute a^t, again at worst $O(\log(N))$, followed by repeatedly squaring a^{t2^k} until we terminate early or do it r times. This is an $O(r)$ task or at worst an $O(\log(N))$ task.

Finally, by composition, we see the strong test has time complexity $O(\log(N))$.

4.3 Relation Between Tests

A program called ***Q6.py*** has been written to compute the number of pseudoprimes with each test over given intervals. It also yields the pseudoprimes which pass both the base $a = 2$ and 3 cases. It is listed in Section 5.

We tabulated the number of pseudoprimes in each test along with the count of true primes in certain intervals in Table 7. In Table 8 we note the pseudoprimes that passed the tests for both bases 2 and 3.

Interval	Fermat	Euler	Strong	Primes
$[10^5, 2 \cdot 10^5]$	28	13	3	8393
$[10^6, 10^6 + 10^5]$	16	9	4	7217
$[10^7, 10^7 + 10^5]$	6	4	0	6242
$[10^8, 10^8 + 10^5]$	1	0	0	5412
$[10^9, 10^9 + 10^5]$	0	0	0	4833

Table 7: Number of pseudoprimes detected

Fermat	Euler	Strong
101101 104653 107185 115921 126217 162401 172081 176149 188461 1024651 1033669 1050985 1082809 10024561 10084177 100017223	115921 126217 172081 1050985 1082809 10024561 10084177	∅

Table 8: Composite numbers passing both base 2 and 3 tests

We note that the Fermat test is the weakest test, followed by Euler then the strong test. This agrees with the established theory that: N a strong pseudoprime $\Rightarrow$ N an Euler pseudoprime $\Rightarrow$ N a Fermat pseudoprime.

4.4 An Efficient Primality Test

Our goal is to create an efficient primality test. To this end we begin by comparing the speed of computing one instance of the Fermat, Euler and strong tests and completing the trial division test. We do this by applying each test to some randomly chosen prime base and testing how long it takes to perform the test for 100, 000 randomly chosen odd numbers in $[10^k, 10^{k+1})$.

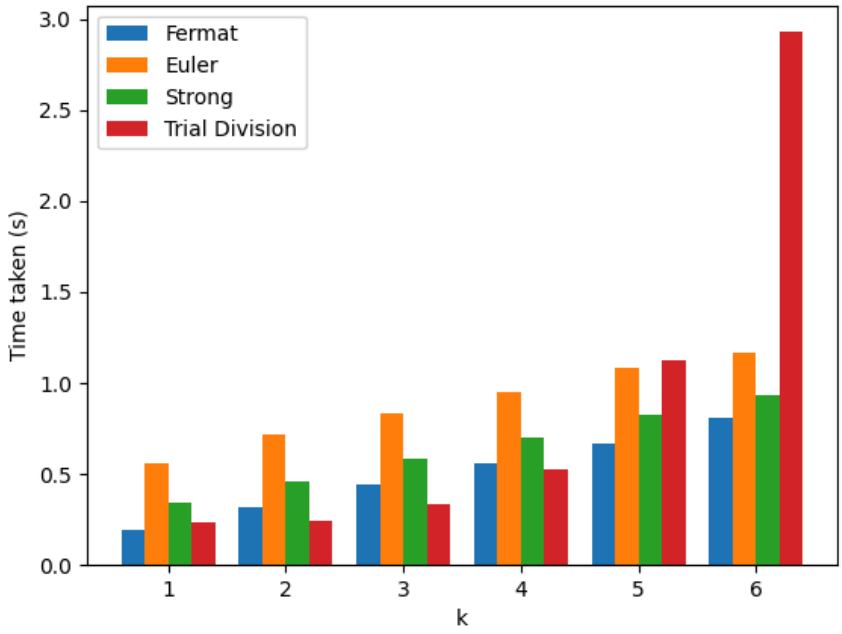

Figure 1: Time taken to perform each test over intervals $[10^k, 10^{k+1})$

Observing the data, we see that trial division is by far the most efficient method for small numbers $(n < 10^5)$. This is due to how easy remainders are to compute. However for sufficiently large numbers $(n > 10^8)$ it becomes incredibly slow relative to other tests due to the exponential complextity with respect to k.

The Fermat, Euler, and strong tests are all approximately the same speed for these numbers. However, the strong test is by far the best test because all strong pseudoprimes are Euler and Fermat pseudoprimes, so any real benefits to those two tests are overshadowed. Moreover, thanks to *R. Crandall and C. Pomerance, Prime Numbers: A Computational Perspective*, we have the sequence ***A014233*** on *oeis.org*

which informs us that using the strong test base the first five prime numbers yields no pseudoprimes $< 2 \cdot 10^{12}$.

Using this result, it becomes possible to create an algorithm which decides which is the best method to test primality of a number $N < 10^{10}$; it has been called the *quick test*. We compared it against trial division over intervals $\left[10^k, 10^{k+1}\right)$.

The quick test algorithm, and the testing code are all in ***Q7.py***, listed in Section 5.

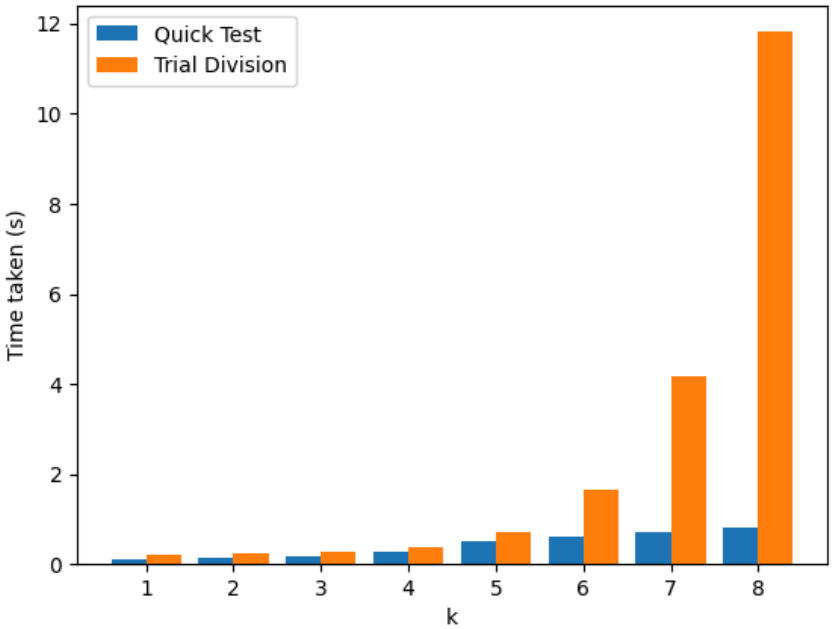

Figure 2: Time taken to perform each test over intervals $\left[10^k, 10^{k+1}\right)$

We note that it is marginally faster than trial division for $k < 5$ as this is due to the algorithm being almost trial division exactly. The small discrepancy could be realised from the quick test algorithm having an immediate check of divisibility by 2, as opposed to computing $\sqrt{N}$ first. Considering half of all quick tests stop after just one operation, it makes sense that there is an decrease in time taken. Of course for large n, when the strong test comes into play, the quick test is overwhelmingly superior.

As N becomes large, this algorithm breaks down as we are not sure of how many iterations of the strong test are needed to prove primality. However, suppose we did: Suppose $f(N)$ is a function defined as the number of iterations of the strong test needed to prove primality for all numbers $< N$, then it would be clear that the complexity of the quick test for large N would be $O(f(N)\log(N))$, having $f(N)$ iterations of the strong test. By observing the data we can see in ***A014233***, a reasonable f may be $f(N) = \log_{10}(N)$ - so the quick test may be $O(\log^2(N))$.

It is very important to note that if multiplication were not a constant time operation, for instance having complexity based in the bits required to represent a number $O(\log(N))$, the algorithm would be more complex. Indeed, this restriction would turn the quick test into an $O(\log^3(N))$ algorithm.

4.5 Effectiveness of the Strong Test

A program called ***Q8.py*** has been written to perform the strong test on random numbers. It has been used for numbers between 2^{k-1} and $2^k - 1$, and shows approximately how likely it is that t iterations of the strong test still yield a pseudoprime. It is shown in Section 5, with results in Table 9.

k	Primes	Strong Test Passes				P(prime|passes t strong tests)			
		1	2	3	4	1	2	3	4
15	19675	19744	19676	19675	19675	0.9965053	0.9999492	1.0	1.0
16	18492	18533	18492	18492	18492	0.9977877	1.0	1.0	1.0
17	17545	17575	17546	17545	17545	0.9982930	0.9999430	1.0	1.0
18	16529	16544	16529	16529	16529	0.9990933	1.0	1.0	1.0
19	15587	15600	15588	15587	15587	0.9991667	0.9999358	1.0	1.0
20	14742	14754	14742	14742	14742	0.9991867	1.0	1.0	1.0
21	13888	13894	13888	13888	13888	0.9995682	1.0	1.0	1.0
22	13278	13282	13278	13278	13278	0.9996989	1.0	1.0	1.0
23	12806	12806	12806	12806	12806	1.0	1.0	1.0	1.0
24	12313	12315	12313	12313	12313	0.9998376	1.0	1.0	1.0
25	11716	11718	11717	11716	11716	0.9998293	0.9999147	1.0	1.0
26	11429	11430	11429	11429	11429	0.9999125	1.0	1.0	1.0
27	11064	11065	11064	11064	11064	0.9999096	1.0	1.0	1.0
28	10401	10401	10401	10401	10401	1.0	1.0	1.0	1.0

Table 9: Applying the strong test t times to 10 thousand randomly chosen numbers

In observing the data, we note there is a very high probability that given a number passes the strong test it is prime. Indeed, a number N passing even two cases of the strong test gives an incredibly slim chance that N is not prime.

5 Programs

5.1 Q1.py

```
import numpy as np

def TrialDivisionTest(n):
    if (n <= 1): return False
    for i in range(2, int(np.sqrt(n)) + 1):
        if (n % i == 0):
            return False
    return True

if __name__ == "__main__":
    for i in range(188000, 188201):
        if (TrialDivisionTest(i)):
            print(i, end=" ")

    print("")
    k = 1000000000
    for i in range(k, k + 201):
        if (TrialDivisionTest(i)):
            print("10^{9}+" + str(i-k), end=" ")
```

5.2 Q2.py

```
import numpy as np

def EvaluateBasePowerMod(a, M, N):
    # reduce to simpler case
    b = a % N
    pow = 0
    mod = 1
    i = 0
    while (M > 0):
        c = M & 1
        M -= c
        pow += c * (1 << i)
        M = M >> 1
        if (c):
            mod = (mod * b) % N

        # May have overflow in this ase
        b = (b * b) % N

        # Here is memory efficient case

        #prod = 0
        #d = b
        #while (d > 0):
            #if (d % 2 == 1):
                #prod = prod + b % N
            #b = 2*b % N
            #d /= 2

        #b = prod


        if (b == 1):
            break

        i += 1

    return (mod % N)


def FermatTest(N, a):
    return (EvaluateBasePowerMod(a, N-1, N) == 1)



if __name__ == "__main__":

    primeset = {188011, 188017, 188021, 188029, 188107, 188137, 188143, 188147, 188159,
188171, 188179, 188189, 188197}
    for j in range(2, 14):
        testPass = set()
        print(j, end=" ")
        for i in range(188000, 188201):
            if (FermatTest(i, j)):
                testPass.add(i)
```

```
        print(primeset.issubset(testPass), testPass.difference(primeset))

    print("")

    k = 1000000000
    primeset = {1000000007, 1000000009, 1000000021, 1000000033, 1000000087, 1000000093,
1000000097, 1000000103, 1000000123, 1000000181}
    for j in range(2, 14):
        testPass = set()
        print(j, end=" ")
        for i in range(k, k+201):
            if (FermatTest(i, j)):
                testPass.add(i)
        print(primeset.issubset(testPass), testPass.difference(primeset))
```

5.3 Q3.py

```
import numpy as np
from Q2 import FermatTest
from Q1 import TrialDivisionTest

def FindFermatPseudoPrimesBase(a, upperBound=1000000, lowerBound=2):
    output = set()
    for i in range(lowerBound, upperBound + 1):
        if (FermatTest(i, a)):
            if (not TrialDivisionTest(i)):
                output.add(i)

    return output




def FindCarmichaelFermat (upperBound=1000000, lowerBound=2):
    twoPseudoPrimes = sorted(FindFermatPseudoPrimesBase(2, upperBound, lowerBound))
    #print(twoPseudoPrimes)

    carmichaelNumbers = set()

    for n in twoPseudoPrimes:
        isCarmichael = True
        for a in range(3, min(n, 20)):
            if (np.gcd(a,n) != 1):
                continue
            if (not TrialDivisionTest(a)):
                continue
            if (FermatTest(n, a)):
                continue
            # Uncomment this print to get data for when we found a base which breaks
            #print(n, a)
            isCarmichael = False
            break

        if (isCarmichael):
            carmichaelNumbers.add(n)


    return carmichaelNumbers


if __name__ == "__main__":
    print(sorted(FindFermatPseudoPrimesBase(2, 1000000)))
    print(sorted(FindCarmichaelFermat(1000000)))
```

5.4 Q4.py

```
import numpy as np
from Q1 import TrialDivisionTest
from Q2 import EvaluateBasePowerMod


def EvaluateJacobi (a, b):
    a = a % b
    result = 1
    while (a > 1):

        ## Factor out 4s
        while (a % 4 == 0):
            a = a >> 2

        # Check last 2
        if (a % 2 == 0):
            a = a >> 1
            if (int((b*b - 1)/8) % 2 == 1):
                result *= -1

        # Flip upsidedown and modulo out
        if (a != 1):
            c = b
            b = a
            a = c

            if (int((a-1)*(b-1) / 4) % 2 == 1):
                result *= -1

            a = a % b

    return result * a


def EulerTest (N, a):

    if (np.gcd(N,a) == 1 and np.gcd(N,2) == 1):
        return ((EvaluateJacobi(a, N) % N) == (EvaluateBasePowerMod(a, int((N-1) / 2),
N) % N))
    return False


def FindEulerPseudoPrimesBase(a, upperBound=1000000, lowerBound=2):
    output = set()
    for i in range(lowerBound, upperBound + 1):
        if (EulerTest(i, a)):
            if (not TrialDivisionTest(i)):
                output.add(i)

    return output

def FindAbsoluteEulerPseudoPrimes(upperBound=1000000, lowerBound=2):

    twoPseudoPrimes = sorted(FindEulerPseudoPrimesBase(2, upperBound, lowerBound))
```

```
    absolutePseudoPrimes = set()

    for n in twoPseudoPrimes:
        isAbsPseudo = True
        for a in range(3, n):
            if (np.gcd(a,n) != 1):
                continue
            if (EulerTest(n, a)):
                continue
            # Uncomment this print to get data for when we found a base which breaks
            #print(n, a)
            isAbsPseudo = False
            break

        if (isAbsPseudo):
            absolutePseudoPrimes.add(n)

    return absolutePseudoPrimes


if __name__ == "__main__":
    print(sorted(FindEulerPseudoPrimesBase(2, 1000000)))
    print(FindAbsoluteEulerPseudoPrimes(1000000))
```

5.5 Q5.py

```
import numpy as np
from Q1 import TrialDivisionTest
from Q2 import EvaluateBasePowerMod

def StrongTest(N, a):
    if (np.gcd(N,a) != 1):
        return False

    s = N - 1
    r = 0
    while (s & 1 == 0):
        r += 1
        s = s >> 1

    k = EvaluateBasePowerMod(a, s, N)
    if (k == 1 or k == (-1 % N)):
        return True
    for i in range(1, r):
        k = (k * k) % N
        if (k == (-1 % N)):
            return True

    return False


def FindStrongPseudoPrimesBase(a, upperBound=1000000, lowerBound=2):
    output = set()
    for i in range(lowerBound, upperBound + 1):
        if (StrongTest(i, a)):
            if (not TrialDivisionTest(i)):
                output.add(i)

    return output

def FindAbsoluteStrongPseudoPrimes(upperBound=1000000, lowerBound=2):
    twoPseudoPrimes = sorted(FindStrongPseudoPrimesBase(2, upperBound, lowerBound))
    #print(twoPseudoPrimes)

    absolutePseudoPrimes = set()
    breakingPoints = []

    for n in twoPseudoPrimes:
        isAbsPseudo = True
        for a in range(3, n):
            if (np.gcd(a,n) != 1):
                continue
            if (StrongTest(n, a)):
                continue
            # If needed
            breakingPoints.append([n,a])
            isAbsPseudo = False
            break

        if (isAbsPseudo):
            absolutePseudoPrimes.add(n)
```

```
    return absolutePseudoPrimes, breakingPoints


if __name__ == "__main__":
    print(sorted(FindStrongPseudoPrimesBase(2,1000000)))
    absPseudo, breakingPoints = FindAbsoluteStrongPseudoPrimes(1000000)

    print(sorted(absPseudo))
    print(breakingPoints)
```

5.6 Q6.py

```
import numpy as np
from Q1 import TrialDivisionTest
from Q3 import FindFermatPseudoPrimesBase
from Q4 import FindEulerPseudoPrimesBase
from Q5 import FindStrongPseudoPrimesBase
from Q5 import StrongTest


def FindPrimesWithStrongBase2(upperBound=1000000, lowerBound=2):
    output = {2}
    for i in range(lowerBound, upperBound + 1):
        if (StrongTest(i, 2)):
            if (TrialDivisionTest(i)):
                output.add(i)

    return output


def GetPseudoPrimesBase(a, upperBound = 10000, lowerBound=2):
    fermat = FindFermatPseudoPrimesBase(a, upperBound, lowerBound)
    euler = FindEulerPseudoPrimesBase(a, upperBound, lowerBound)
    strong = FindStrongPseudoPrimesBase(a, upperBound, lowerBound)

    return fermat, euler, strong

if __name__ == "__main__":

    fermatOverlap = [[] for i in range(5)]
    eulerOverlap = [[] for i in range(5)]
    strongOverlap = [[] for i in range(5)]

    for k in range(5, 10):
        low = int(np.power(10, k))
        upp = low + 100000
        ferm2, eul2, strong2 = GetPseudoPrimesBase(2, upp, low)
        ferm3, eul3, strong3 = GetPseudoPrimesBase(3, upp, low)

        primes = sorted(FindPrimesWithStrongBase2(upp, low))

        print(len(ferm2), len(eul2), len(strong2), len(primes))

        fermatOverlap[k-5] = sorted(ferm2.intersection(ferm3))
        eulerOverlap[k-5] = sorted(eul2.intersection(eul3))
        strongOverlap[k-5] = sorted(strong2.intersection(strong3))


    print(fermatOverlap)
    print(eulerOverlap)
    print(strongOverlap)
```

5.7 Q7.py

```
from Q1 import TrialDivisionTest
from Q2 import FermatTest
from Q4 import EulerTest
from Q5 import StrongTest
import numpy as np
import time
import matplotlib.pyplot as plt

def QuickPrimalityTest(N):
    ### Make sure N is odd
    if (N % 2 == 0):
        return False
    ### Is it better to use Trial Division?
    if (N < 100000):
        return TrialDivisionTest(N)

    if (not StrongTest(N, 2)): return False
    if (not StrongTest(N, 3)): return False

    # Does it need more tests
    if (N < 1373652): return True
    if (not StrongTest(N, 5)): return False

    # Does it need more tests
    if (N < 25326000): return True
    if (not StrongTest(N, 7)): return False

    # Does it need more tests
    if (N < 3215031750): return True
    if (not StrongTest(N, 11)): return False

    # Must be true now
    return True


def SpeedChecking():
    t = 100000
    primes = [3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
73, 79, 83, 89, 97]

    K = [i for i in range(1, 7)]
    fermatTimes = []
    eulerTimes = []
    strongTimes = []
    trialTimes = []
    for k in range(1, 7):

        print("$" + str(k) + "$, ", end= "")
        numbers = [ int(np.random.randint(10**k, 10**(k+1))) for i in range(t)]
        for i in range(t):
            if (numbers[i] % 2 == 0):
                numbers[i] = numbers[i] + 1

        start = time.time()
```

```
        for i in range(t):
            passesTest = FermatTest(numbers[i], primes[i % 24])

        fermatTimes.append(time.time() - start)
        start = time.time()

        for i in range(t):
            passesTest = EulerTest(numbers[i], primes[i % 24])

        eulerTimes.append(time.time() - start)
        start = time.time()

        for i in range(t):
            passesTest = StrongTest(numbers[i], primes[i % 24])

        strongTimes.append(time.time() - start)
        start = time.time()

        for i in range(t):
            prime = TrialDivisionTest(numbers[i])

        trialTimes.append(time.time() - start)


    ### Graph it

    X_axis = np.arange(len(K))

    plt.bar(X_axis - 0.3, fermatTimes, 0.2, label = 'Fermat')
    plt.bar(X_axis - 0.1, eulerTimes, 0.2, label = 'Euler')
    plt.bar(X_axis + 0.1, strongTimes, 0.2, label = 'Strong')
    plt.bar(X_axis + 0.3, trialTimes, 0.2, label = 'Trial Division')

    #plt.rcParams['text.usetex'] = True

    plt.xticks(X_axis, K)
    plt.xlabel("k")
    plt.ylabel("Time taken (s)")
    plt.title("")
    plt.legend()
    plt.show()



def QuickTrialComparison ():
    t = 100000

    K = [i for i in range(1, 9)]
    trialTimes = []
    quickTimes = []
    for k in range(1, 9):

        print("$" + str(k) + "$, ", end= "")
        numbers = [ int(np.random.randint(10**k, 10**(k+1))) for i in range(t)]
```

```
        start = time.time()

        for i in range(t):
            QuickPrimalityTest(numbers[i])

        quickTimes.append(time.time()- start)
        start = time.time()

        for i in range(t):
            TrialDivisionTest(numbers[i])

        trialTimes.append(time.time()- start)

    ### Graph it

    X_axis = np.arange(len(K))

    plt.bar(X_axis - 0.2, quickTimes, 0.4, label = 'Quick Test')
    plt.bar(X_axis + 0.2, trialTimes, 0.4, label = 'Trial Division')

    #plt.rcParams['text.usetex'] = True

    plt.xticks(X_axis, K)
    plt.xlabel("k")
    plt.ylabel("Time taken (s)")
    plt.title("")
    plt.legend()
    plt.show()

if __name__ == "__main__":
    SpeedChecking()
    print()
    QuickTrialComparison()
```

5.8 Q8.py

```
from Q1 import TrialDivisionTest
from Q5 import StrongTest
from Q7 import QuickPrimalityTest
import numpy as np

def TestStrongProbabilities(k, t, n):
    primeCount = 0
    passStrongCounts = [0 for j in range(t)]

    for i in range(n):
        # Get number
        num = np.random.randint(1 << k-1, 1 << k)
        if (num & 1 == 0):
            num += 1

        if (QuickPrimalityTest(num)):
            primeCount += 1

        passesStrong = [True for j in range(t)]
        for j in range(t):
            a = np.random.randint(2, num-1)
            if (not StrongTest(num, a)):
                for w in range(j, t):
                    passesStrong[w] = False
                break

        for j in range(t):
            if (passesStrong[j]):
                passStrongCounts[j] += 1

    print("$" + str(k) + "$, $" + str(primeCount) + "$, ", end="")

    for j in range(t):
        print("$" + str(passStrongCounts[j]) + "$, ", end="")
    for j in range(t):
        print("$" + str(primeCount/passStrongCounts[j]) + "$, ", end="")

    print("")



if __name__ == "__main__":

    for i in range(15, 28):
        TestStrongProbabilities(i, 4, 10000)
```