Problem Set 2

PAWS 2025

Nathaniel Hurst

1 The order of $2 \in \mathbb{F}_{71}^*$ is 35. Charlie uses the subgroup generated by $g = 2$, his public key is $g_c = 29$. Use the baby-step giant-step algorithm to compute an integer c such that $g_c \equiv g^c \pmod{71}$.

First we compute $m = \lfloor \sqrt{35} \rfloor + 1 = 6$. Then we compute the sequence

$$g_0 = 1 \quad g_1 = 2 \quad g_2 = 4 \quad g_3 = 8 \quad g_4 = 16 \quad g_5 = 32$$

for our baby steps and the the sequence

$$A_0 = 29 \quad A_1 = 29 \cdot g^{-6} = 6 \quad A_2 = 29 \cdot \left(g^{-6}\right)^2 = 60 \quad A_3 = 29 \cdot \left(g^{-6}\right)^3 = 32$$

For our giant steps (note we stopped at the third giant step as we found a collision $g_5 = A_3$). Thus our discrete log is given by $g^{6 \cdot 3 + 5} = 29$, or $\operatorname{dlog}_g(29) = 6 \cdot 3 + 5 = 23$.

2 We have seen in Remark 2.12 (of the lecture notes) that one may reduce the size of the required memory at the cost of increasing the overall runtime of the algorithm. In this exercise, the goal is to achieve the opposite: decreasing the runtime at the cost of increasing the required memory.

(a) We have shown that Algorithm 1 (of the lecture notes) requires $O(\sqrt{q})$ multiplications. More concretely, show that the average runtime is given by $T = \frac{3}{2}\sqrt{q}$ (if the exponent a is chosen uniformly at random).

Now consider a variant of Algorithm 1, where the baby steps and giant steps are computed in parallel, and all values g_i, A_i for $0 \leq i \leq n \leq m$ are stored until the match is found for some n.

(b) What is the average runtime of this variant of Algorithm 1? What is the required memory?

Hint: You can use (or prove if you are familiar with probability theory) that for two integers i, j uniformly chosen at random from $(0, ..., m)$, the expected value of $\max(i, j) \approx \frac{2}{3}m$.

(a) Observe that we must always do $m = \lfloor \sqrt{q} \rfloor + 1$ multiplications to generate our baby steps. Then we need to do at maximum another $m = \lfloor \sqrt{q} \rfloor + 1$ giant steps, at each point checking for a collision in our baby and giant steps. The expected number of giant step computations we will have to do is $\frac{\sqrt{q}}{2}$ before we hit a collision (if a is chosen uniformly at random), and combining that with the guaranteed $\sqrt{q}$ computations we have that the average runtime of this algorithm is $\frac{3}{2}\sqrt{q}$.

(b) Using the hint we can calculate the expected time before the first collision. To do so we fix two integers $i, j \in \{0, ..., m\}$ such that $\text{baby\_step}_i = \text{giant\_step}_j$ is the first collision. Then we will hit this collision in our computations after $\max\{i, j\} = \frac{2}{3}m$ steps (the computations are running in parallel, so $\frac{4}{3}m$ computations total). Thus the average runtime is $\frac{2}{3}m \approx \frac{2}{3}\sqrt{q}$ and the required memory is $\frac{4}{3}m \approx \frac{4}{3}\sqrt{q}$.

3

Implement the baby-step giant-step algorithm and use it to solve the DLP instances from Exercise 5 of the first exercise set (copied again here). How does the running time compare to the log function in SageMath? Which algorithm is used in SageMath to solve the DLP?

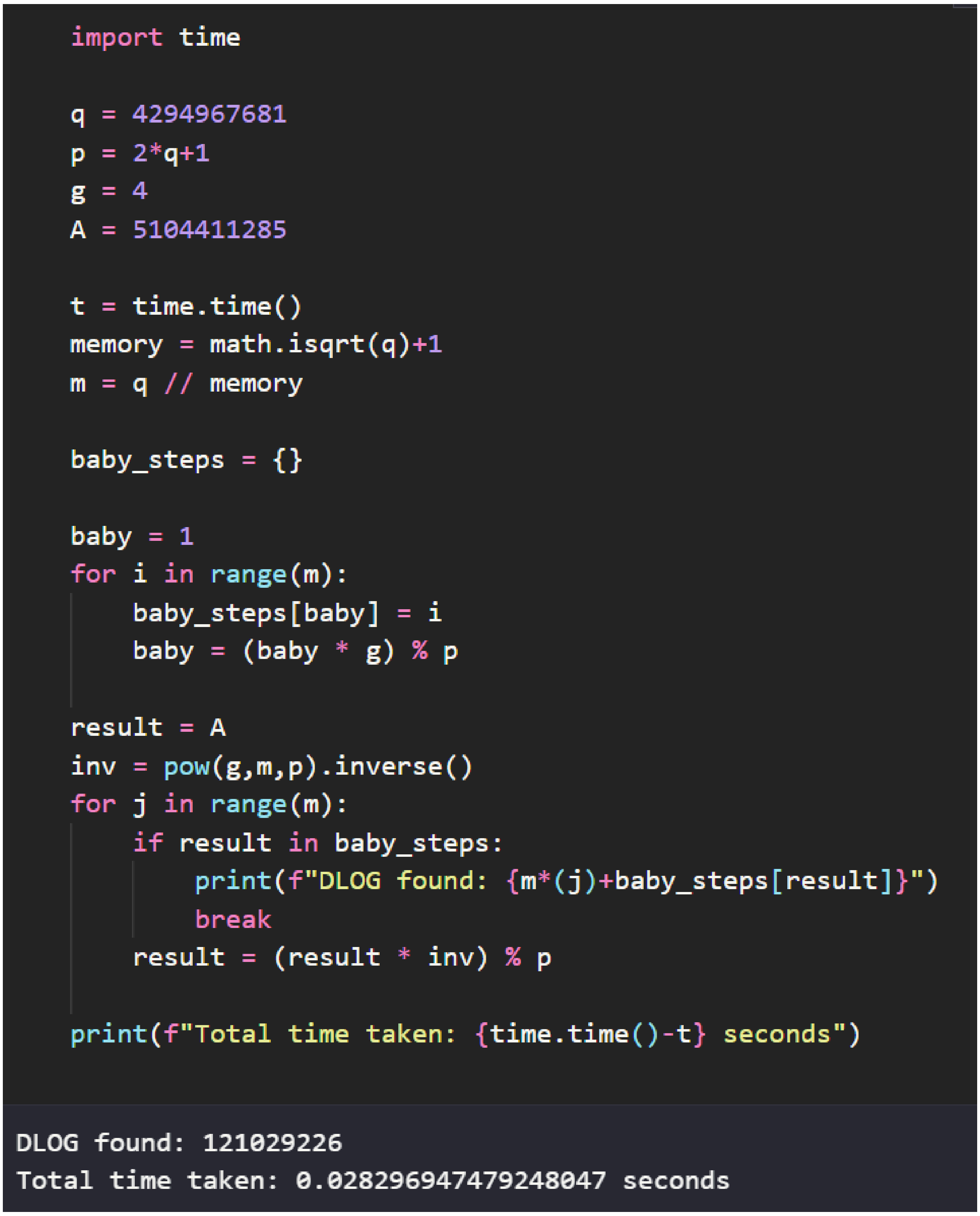
In all of these, the public parameters are a prime $p = 2q + 1$, and the element $g = 4 \in \mathbb{F}_p^*$ with order q:

(a) $q = 4294967681 \approx 2^{32}$, $A = 5104411285$, $B = 7620748646$.

(b) $q = 18446744073709552109 \approx 2^{64}$, $A = 17485644247020728566$, $B = 17485644247020728566$.

(c) $q = 340282366920938463463374607431768219863 \approx 2^{128}$, $A = 158556695861572453782110953476057 06305$, $B = 64379118553030588585874013494652067 2205$

To save space and computing time we only reproduce the first discrete logarithm with the following baby-step giant-step algorith (code can be found in baby-step_giant-step.ipynb in the Scripts folder of my github repo:



```python
import time

q = 4294967681
p = 2*q+1
g = 4
A = 5104411285

t = time.time()
memory = math.isqrt(q)+1
m = q // memory

baby_steps = {}

baby = 1
for i in range(m):
    baby_steps[baby] = i
    baby = (baby * g) % p

result = A
inv = pow(g,m,p).inverse()
for j in range(m):
    if result in baby_steps:
        print(f"DLOG found: {m*(j)+baby_steps[result]}")
        break
    result = (result * inv) % p

print(f"Total time taken: {time.time()-t} seconds")
```

```
DLOG found: 121029226
Total time taken: 0.028296947479248047 seconds
```

The algorithm used by SageMath is defaulted to the baby-step giant-step but it has many algorithms you can switch to in order to tailor to your input.

4 Use Pollard's rho algorithm to compute c such that $2^c \equiv 29 \pmod{71}$, i.e., use Pollard's rho algorithm to do Exercise 1 in this Problem Set. What are T and L? How does the value of $T + L$ compare to the expected value given in Theorem 2.16?

We will use the function given by Pollard, that is $f(x) = x \cdot A$ if $0 < x < \frac{p}{3}$, $f(x) = x^2$ if $\frac{p}{3} < x < \frac{2p}{3}$, and $f(x) = x \cdot g$ if $\frac{2p}{3} < x < p$. Then we start with $x_0 = 1$, giving us the sequence

$$x_1 = f(x_0) = A = 29 \qquad x_2 = f(x_1) = A^2 = 60 \qquad x_3 = f(x_2) = A^2 \cdot g = 60 \cdot 2 = 49$$

$$x_4 = f(x_3) = A^2 \cdot g^2 = 49 \cdot 2 = 27 \qquad x_5 = f(x_4) = A^4 \cdot g^4 = 27^2 = 19$$

$$x_6 = f(x_5) = A^5 \cdot g^4 = 19 \cdot 29 = 54 \qquad x_7 = f(x_6) = A^5 \cdot g^5 = 37$$

$$x_8 = f(x_7) = A^{10} \cdot g^{10} = 37^2 = 20 \qquad x_9 = f(x_8) = A^{11} \cdot g^{10} = 20 \cdot 29 = 12$$

$$x_{10} = f(x_9) = A^{12} \cdot g^{10} = 12 \cdot 29 = 64 \qquad x_{11} = f(x_{10}) = A^{12} \cdot g^{11} = 64 \cdot 2 = 57$$

$$x_{12} = f(x_{11}) = A^{12} \cdot g^{12} = 57 \cdot 2 = 43 \qquad x_{13} = f(x_{12}) = A^{24} \cdot g^{24} = 43^2 = 3$$

$$x_{14} = f(x_{13}) = A^{25} \cdot g^{24} = 3 \cdot 29 = 16 \qquad x_{15} = f(x_{14}) = A^{26} \cdot g^{24} = 16 \cdot 29 = 38$$

$$x_{16} = f(x_{15}) = A^{52} \cdot g^{48} = 38^2 = 24 \qquad x_{17} = f(x_{16}) = A^{104} \cdot g^{96} = 24^2 = 8$$

$$x_{18} = f(x_{17}) = A^{105} \cdot g^{96} = 8 \cdot 29 = 19$$

Giving us our collision $A^4 \cdot g^4 = x_5 = x_{19} = A^{105} \cdot g^{96}$. Hence the discrete logarithm is given by $a \equiv (4 - 105)^{-1} \cdot (96 - 4) \equiv 23 \pmod{\operatorname{ord}(g)}$. Which is the same as in problem 1. In this case we see that $T = 5$ and $L = 19 - 5 = 14$, giving us $T + L = 19$. By Theorem 2.16 the expected value of $T + L$ is $E(T + L) \approx 1.2533 \cdot \sqrt{71} \approx 10.56$. Since this is a Las Vegas Model I am not suprised that we were not close to the expected value, we got unlucky.

5 Explain why $f(x) = x^2$ is a bad (inefficient) choice of function for Pollard's rho algorithm (say, with initial value $x_0 = g \cdot A$, where the order of g is odd).

The first thing that makes this function inefficient is that it is not psuedorandom, and so we cannot apply the birthday paradox to get its time complexity. Second if $\operatorname{ord}(g) = 2^n - 1$ for some n then it is possible that the algorithm will have no "tail" as it could continuously loop back to the starting value x_0.

6 **Pollard's ρ for Integer Factorization**. You already know that Pollard's ρ algorithm can be used to solve the **discrete logarithm problem** by exploiting cycle detections. The same idea can be adapted to **factor composite integers**, i.e. find prime numbers $p_1, ..., p_r$ such that $n = p_1 ... p_r$.

Let n be a composite integer. Consider the sequence

$$x_{k+1} \equiv \pi(x_k) := x^2 + 1 \pmod{n}$$

Because there are only n residues, the sequence eventually repeats. If a prime $p \mid n$ causes two values to collide modulo p before they collide modulo n, then

$$\gcd(|x_i - x_j|, n)$$

can reveal a non-trivial factor of n.

(a) Show that if $x_i \equiv x_j \pmod{p}$, then $x_{i+k} \equiv x_{j+k} \pmod{p}$ for all $k \geq 0$.

(b) Implement Pollard's ρ method of factorization in Sagemath and use it to factor the following integers:

$$n_1 = 1007 \quad n_2 = 8051 \quad n_3 = 10403 \quad n_4 = 5545419598547562675200$$
$$n_5 = 6263601980743976967475 2 \quad n_6 = 19783 \quad n_7 = 16310011$$

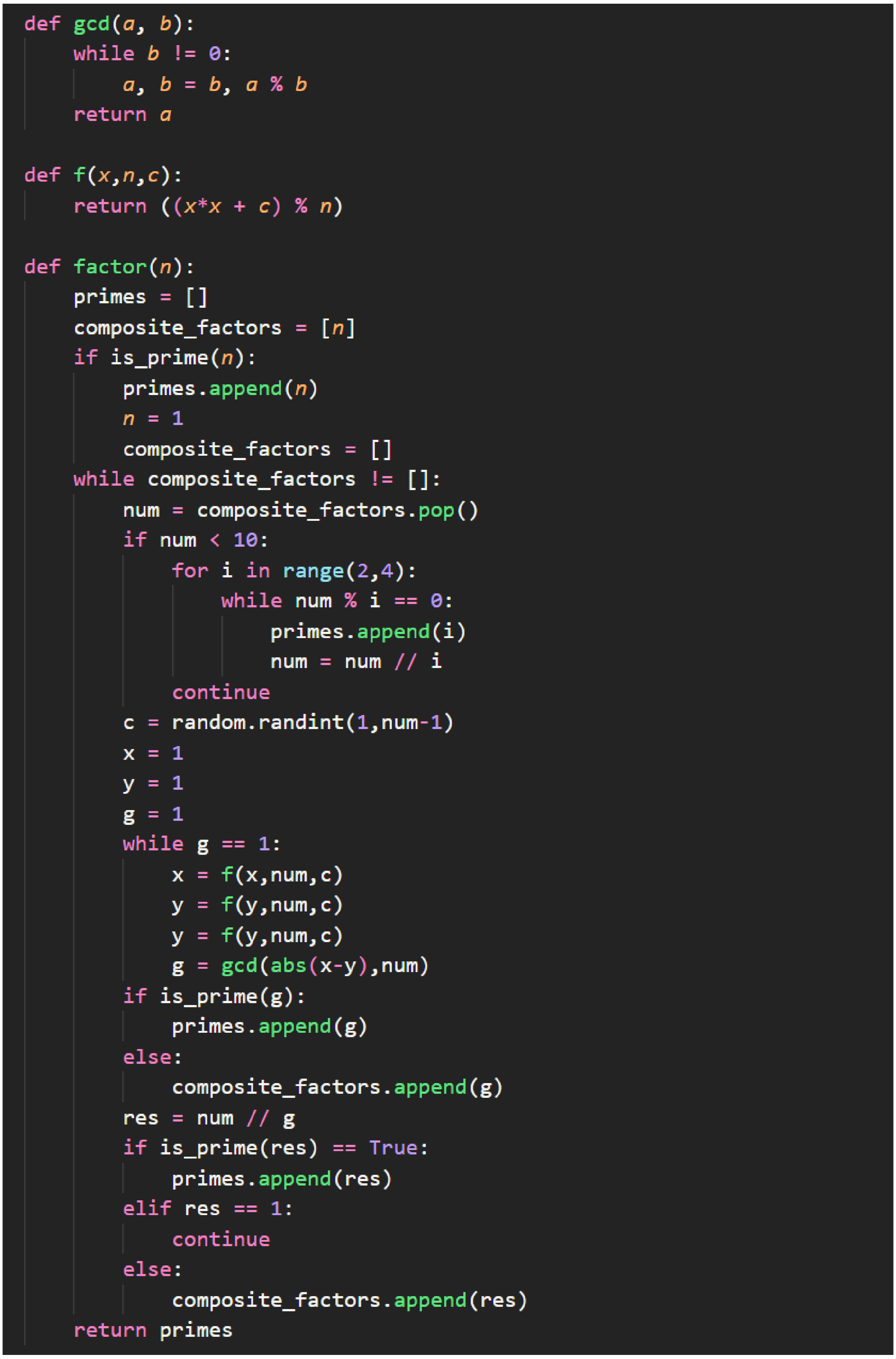
(Yes, they are ordered from easy to hard :), do the next one if you want to understand why)

(c) Use Theorem 2.16 to estimate the expected number of steps needed to factor n, assuming π to be a random permutation. Note: it may be helpful to use the Theorem on a function different from π, since you are looking for a collision modulo one of the prime factors of n.

(d) Explain why Pollard's ρ is particularly effective when n has many small prime factors, even if n itself is large.

(a) Observe that if $x_i \equiv x_j \pmod{p}$ then $x_{i+1} \equiv (x_i)^2 + 1 \equiv (x_j)^2 + 1 \equiv x_{j+1} \pmod{p}$, and so repeating inductively we see that $x_{i+k} \equiv x_{j+k} \pmod{p}$ for all $k \geq 0$.

(b) We use the following code to compute the factorization (which can be found in the file pollard_integer_factorization.ipynb in my github repo scripts folder):



```python
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def f(x,n,c):
    return ((x*x + c) % n)

def factor(n):
    primes = []
    composite_factors = [n]
    if is_prime(n):
        primes.append(n)
        n = 1
        composite_factors = []
    while composite_factors != []:
        num = composite_factors.pop()
        if num < 10:
            for i in range(2,4):
                while num % i == 0:
                    primes.append(i)
                    num = num // i
            continue
        c = random.randint(1,num-1)
        x = 1
        y = 1
        g = 1
        while g == 1:
            x = f(x,num,c)
            y = f(y,num,c)
            y = f(y,num,c)
            g = gcd(abs(x-y),num)
        if is_prime(g):
            primes.append(g)
        else:
            composite_factors.append(g)
        res = num // g
        if is_prime(res) == True:
            primes.append(res)
        elif res == 1:
            continue
        else:
            composite_factors.append(res)
    return primes
```

Figure 2: Code (we use trial division to factor numbers less than 10 as the algorithm gets stuck in a meaningless loop in this case)

Which yields:

$$n_1 = 1007 = 19 \cdot 53$$

$$n_2 = 8051 = 83 \cdot 97$$

$$n_3 = 10403 = 101 \cdot 103$$

$$n_4 = 5545419598547562675200 = 2^{10} \cdot 5^2 \cdot 7 \cdot 19 \cdot 1628706414047099$$

$$n_5 = 62636019807439769674752 = 2^{10} \cdot 3^4 \cdot 755160346829665433$$

$$n_6 = 19783 = 73 \cdot 271$$

$$n_7 = 16310011 = 67 \cdot 243433$$

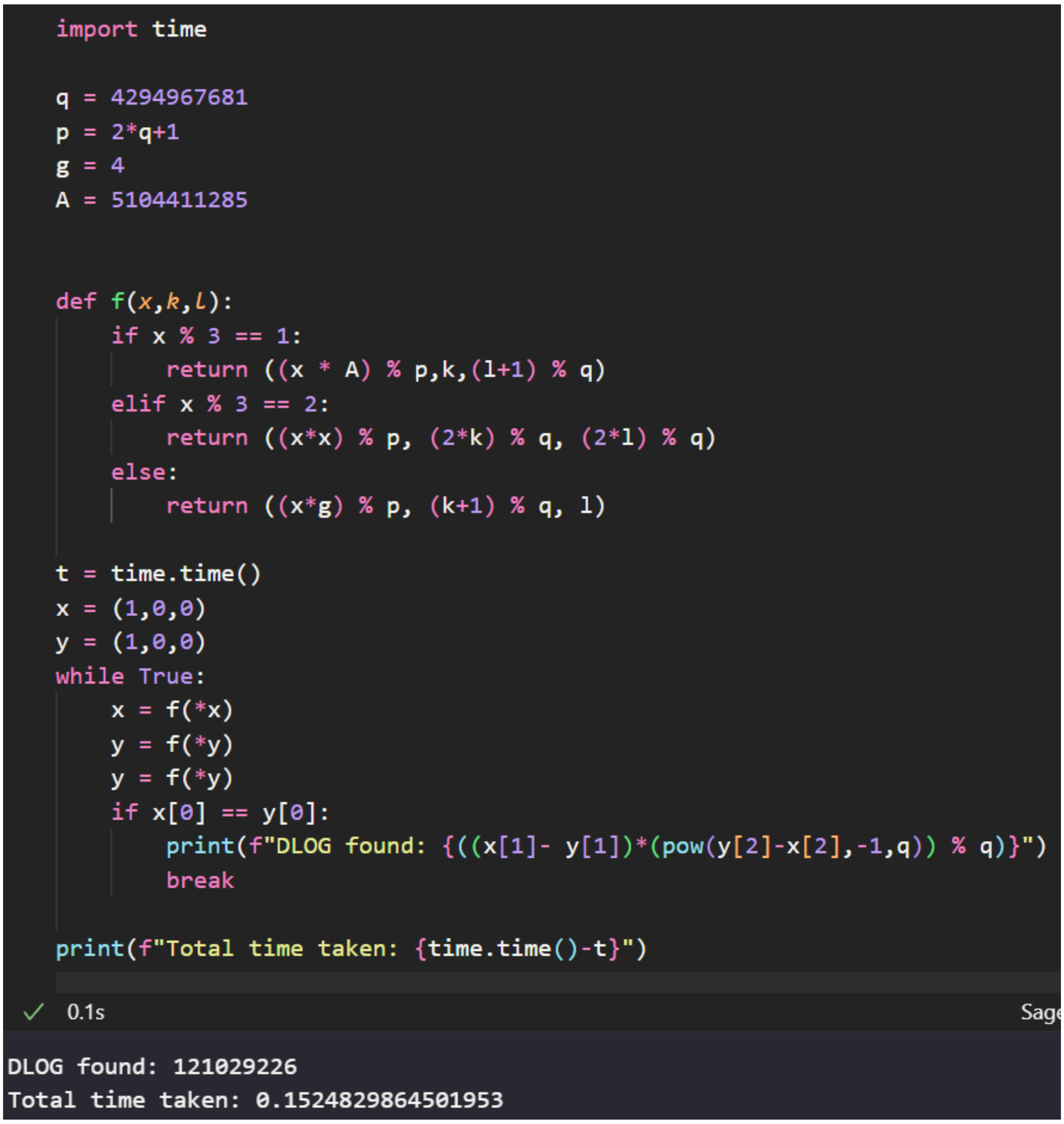
The algorithm runs astonishingly quickly, with factorization time for the hardest example (n_7) being only 0.4 ms.

(c) Let $n = p_1 ... p_k$ be the prime factorization of n, where we allow for repeats in the primes p_i. Then the number of steps before we get a collision modulo p_i is (by Theorem 2.16) $\approx 1.2533 \cdot \sqrt{p_i}$. Now we must get a collision for each prime power p_i, and so combining all the steps gives us that the total number of expected steps to factor n is about $1.2533\left(\sum_{i=1}^{k} \sqrt{p_i}\right)$, where p_i are the primes dividing n (with repeats if it appears more than once in the prime factorization).

(d) Pollard's ρ for integer factorization is particularly effective when n has many small prime factors as when a prime $p \mid n$ is small the number of equivalence classes mnodulo p is small, and hence we are more likely to get a collision modulo p. Take for example $p = 2$, we will get a collision modulo 2 as soon as both x_i and x_{2i} are both even or both odd, which will happen often.

7 Implement the Pollard rho algorithm in Sagemath and use it solve the DLP instances from Exercise 3. Compare the run times with your baby-step giant-step implementation.

This was my implementation, code can be found in the scripts folder in my github repo under pollard_rho.ipynb:


```python
import time

q = 4294967681
p = 2*q+1
g = 4
A = 5104411285


def f(x,k,l):
    if x % 3 == 1:
        return ((x * A) % p,k,(l+1) % q)
    elif x % 3 == 2:
        return ((x*x) % p, (2*k) % q, (2*l) % q)
    else:
        return ((x*g) % p, (k+1) % q, l)

t = time.time()
x = (1,0,0)
y = (1,0,0)
while True:
    x = f(*x)
    y = f(*y)
    y = f(*y)
    if x[0] == y[0]:
        print(f"DLOG found: {((x[1]- y[1])*(pow(y[2]-x[2],-1,q)) % q)}")
        break

print(f"Total time taken: {time.time()-t}")
```

```
✓ 0.1s                                                    Sag

DLOG found: 121029226
Total time taken: 0.1524829864501953
```

This took longer to run than the baby-step giant-step algorithm, and while they are both exponential in time complexity, I hypothesize (with no evidence) that the baby-step giant-step algorithm works faster when the numbers are relatively small.

8 Use index calculus to compute an integer c such that $2^c \equiv 29 \pmod{71}$ (i.e., use index calculus to solve 1 on this problem set). Use factor base $\mathcal{P}_B = \{2, 3, 5\}$ and the sequence of integers $e = 5, 13, 32, 19$.

First we use the sequence $e = 5, 13, 32, 19$ and compute the prime factorization of $\widehat{g^e/A} \in \mathbb{Z}$. We continue this until we have the following 4 relations (since $\#B = 3$):

$$\widehat{g^5/A} = 6 = 2 \cdot 3 \qquad \widehat{g^{13}/A} = 45 = 3^2 \cdot 5 \qquad \widehat{g^{32}/A} = 15 = 3 \cdot 5 \qquad \widehat{g^{19}/A} = 40 = 2^3 \cdot 5$$

Giving us the linear system in $\mathbb{Z}/70\mathbb{Z}$:

$$5 = 1x_2 + 1x_3 + 0x_5 + a$$
$$13 = 0x_2 + 2x_3 + 1x_5 + a$$
$$32 = 0x_2 + 1x_3 + 1x_5 + a$$
$$19 = 3x_2 + 0x_3 + 1x_5 + a$$

Now solving this system using SageMath we get the answer $a \equiv 23 \pmod{35}$, which is the same as in problem $\#1$.

9 When looking at Example 2.19 from the lecture notes, can you also solve for x_2, x_3 and x_5? Explain your observations.

It is possible to also solve for x_2, x_3, and x_5 in this system, as from linear algebra we know that if we were unable to solve for x_2, x_3 and x_5 we would also not be able to solve for a. Now once we have a we can continue solving the system for the other variables, and doing exactly this gives us that $(x_1, x_2, x_5, a) \equiv (1, 16, 28, 23) \pmod{35}$.

10 Try to implement the Index Calculus algorithm (Algorithm 3 from the notes) in Sagemath. Here are some hints:

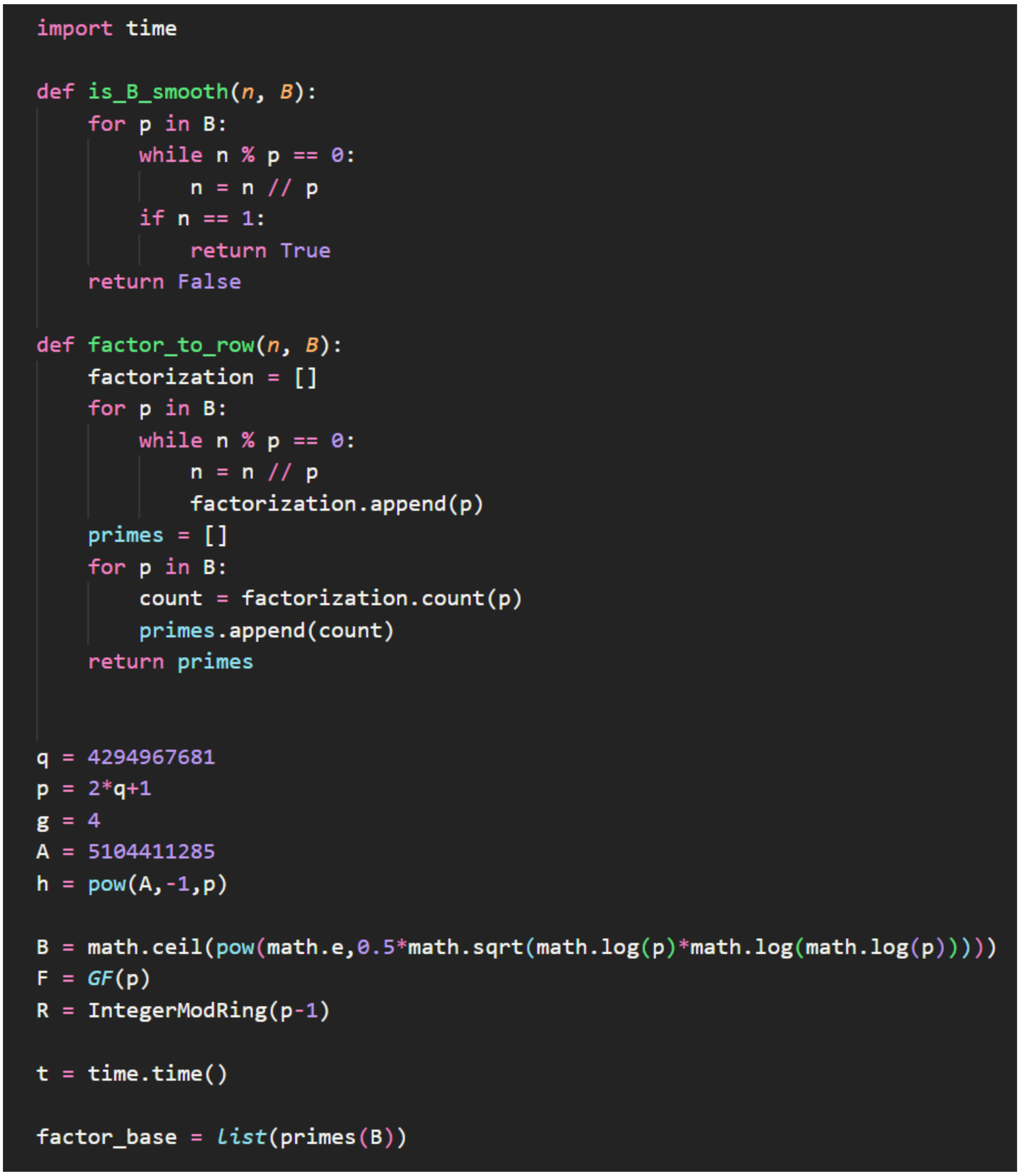
(i) **list(primes(B))** gives you a list of all primes up to B;

(ii) You can initialize the matrix to then the relations having b columns and $b + 1$ rows;

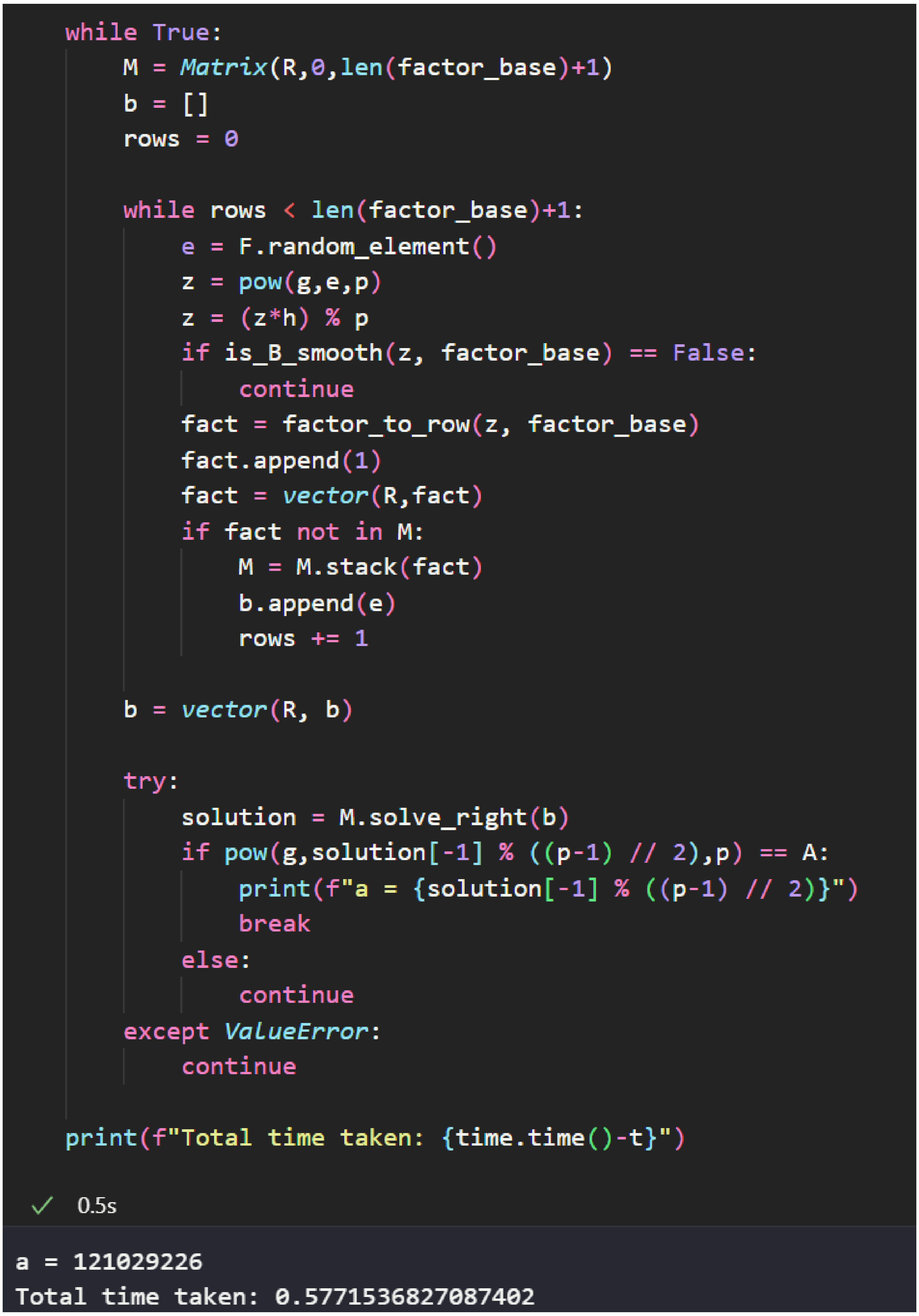
(iii) You do not need to factor completely, just divide out the primes in your factor base, then if you get 1 it means the number was completely factored;

(iv) You can use **A.solve_right(b)** to solve the linear system $Ax = b$ in $\mathbb{F}_q$.

Here is my implementation, code can be found in the scripts folder of my github repo under index_calculus.ipynb:



```python
import time

def is_B_smooth(n, B):
    for p in B:
        while n % p == 0:
            n = n // p
        if n == 1:
            return True
    return False

def factor_to_row(n, B):
    factorization = []
    for p in B:
        while n % p == 0:
            n = n // p
            factorization.append(p)
    primes = []
    for p in B:
        count = factorization.count(p)
        primes.append(count)
    return primes


q = 4294967681
p = 2*q+1
g = 4
A = 5104411285
h = pow(A,-1,p)

B = math.ceil(pow(math.e,0.5*math.sqrt(math.log(p)*math.log(math.log(p)))))
F = GF(p)
R = IntegerModRing(p-1)

t = time.time()

factor_base = list(primes(B))
```



```
    while True:
        M = Matrix(R,0,len(factor_base)+1)
        b = []
        rows = 0

        while rows < len(factor_base)+1:
            e = F.random_element()
            z = pow(g,e,p)
            z = (z*h) % p
            if is_B_smooth(z, factor_base) == False:
                continue
            fact = factor_to_row(z, factor_base)
            fact.append(1)
            fact = vector(R,fact)
            if fact not in M:
                M = M.stack(fact)
                b.append(e)
                rows += 1

        b = vector(R, b)

        try:
            solution = M.solve_right(b)
            if pow(g,solution[-1] % ((p-1) // 2),p) == A:
                print(f"a = {solution[-1] % ((p-1) // 2)}")
                break
            else:
                continue
        except ValueError:
            continue

    print(f"Total time taken: {time.time()-t}")
```

✓ 0.5s

```
a = 121029226
Total time taken: 0.5771536827087402
```

Figure 5: This implementation sees high variance in computation times (see graphs in problem 12) most likely due to the fact that gathering relations is based on random chance, and sometimes the system is not solvable with our relations, so we must restart the algorithm and look for new relations.

11

Let T_B be the expected number of trials of random integers modulo p until one is B-smooth, b be the number of primes up to B, r is the (prime) order of g and $M(t)$ be the number of bit operations required to multiply two t-bit integers.

Then, the expected running time of the Index Calculus algorithm (using naive trial division and specialized methods for the **sparse** linear system) is:

$$O\left(b^2 T_B M(\log(p)) + b^{2+o(1)} M(\log(r))\right) \text{ for } p \to \infty$$

Let $L_{p(\frac{1}{2},c)}$ be the subexponential function defined as

$$L_{p(\frac{1}{2},c)} = e^{c\sqrt{\log(p)\log(\log(p))}}$$

Assuming that $B = L_{p(\frac{1}{2},c)}$ for some constant $c > 0$ and $T_B = L_{p(\frac{1}{2},\frac{1}{2c}+o(1))}$, show that the optimal value for c is $\frac{1}{2}$. Then estimate the asymptotic complexity of the Index Calculus algorithm.

Since we are only estimating asymptotic complexity we can ignore the solving of the sparse linear system, as this is very easy compared to gathering relations. Observe then that $b \approx \frac{B}{\log(B)} \approx B = L_{p(c,\frac{1}{2})}$ (since $\log(B) = c\sqrt{\log(p)\log(\log(p))}$ is negligible when we talk about asymptotics). Then

$$b^2 T_B M(\log(p)) = B^2 T_B M(\log(p)) = L_{p(\frac{1}{2},2c)} L_{p(\frac{1}{2},\frac{1}{2c}+o(1))} M(\log(p)) =$$
$$= e^{\left(2c+\frac{1}{2c}+o(1)\right)\sqrt{\log(p)\log(\log(p))}}$$

Now as $p \longrightarrow \infty$ we have that $o(1)$ is negligible and so we just need to choose c such that $2c + \frac{1}{2c}$ is minimal. By calculus we know that $c = \frac{1}{2}$ is the positive c which minimizes $2c + \frac{1}{2c}$, showing that the optimal c is $\frac{1}{2}$. Then the asymptotic complexity of the Index Calculus algorithm is

$$O\left(b^2 M(T_B(\log(p)) + (\log(r)))\right) = O\left(M e^{2\sqrt{\log(p)\log(\log(p))}} \log(p) + e^{\sqrt{\log(p)\log(\log(p))}} \log(r)\right)$$
$$= O\left(L_{p(\frac{1}{2},2)} M(\log(p))\right)$$

Where we used the fact that the dominating term comes from gathering the relations, $b^2 T_B M(\log(p))$.

12

Use your implementation of the Index Calculus algorithm to run some experiments and find an optimal value for B for

$$p = 2 \cdot 386545163 + 1, \quad g = 4$$

Does your experimental optimal value for B agree with the asymptotic value you found in the previous exercise? How does it change? Try to use larger primes and find the optimal $B = L_{p(\frac{1}{2},c)}$ for your implementation experimentally.

By our above calculations the optimal B should be $\left\lceil e^{\frac{1}{2}\sqrt{\log(p)\log(\log(p))}} \right\rceil = 51$. We will now run some experiments to determine that experimentally, where we will run our Index Calculus implementation for every $B \in \{20, 21, ..., 175\}$ and time our algorithm, this is by no means rigorous, but for the proof of the optimal value see the previous problem.

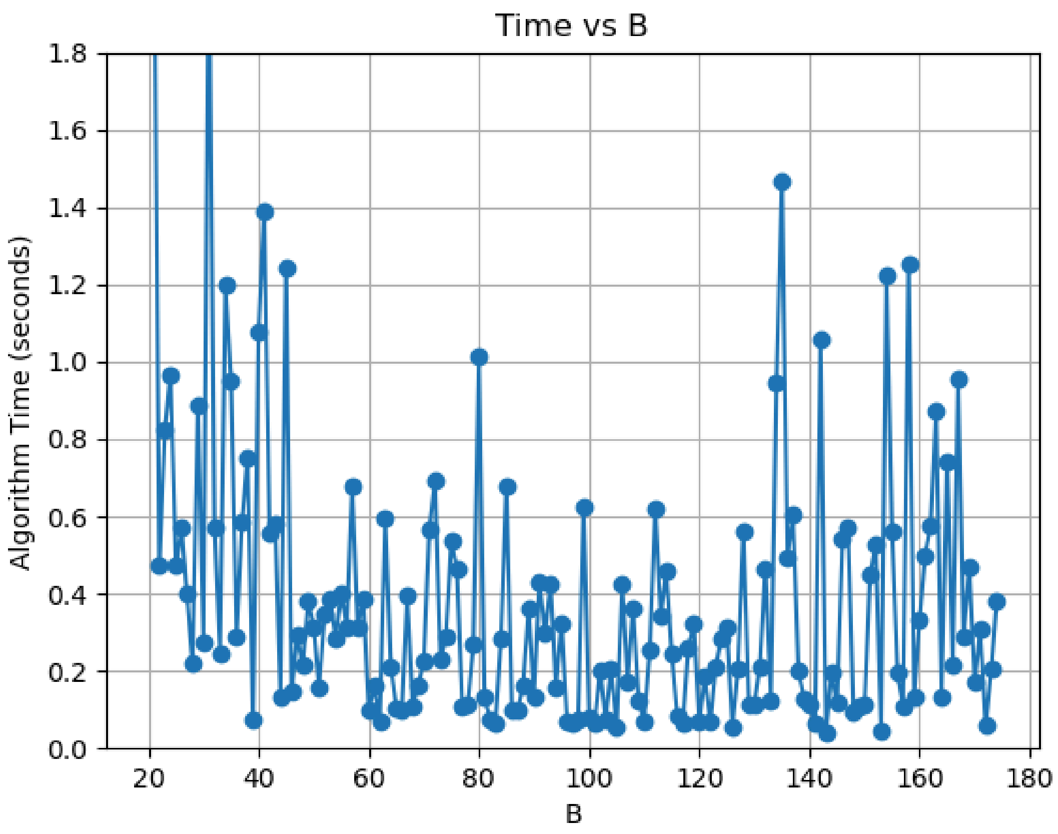


Figure 6: We see that the optimal value for B in this case is between $50 - 120$, with the exact being hard to pick out due to possible randomness in finding relations in order to solve the linear system.

For a test with a larger prime, we use the first prime in exersize #3, $p = 8589935363$. The expected optimal value of B is 69, now running our experiment again we get:

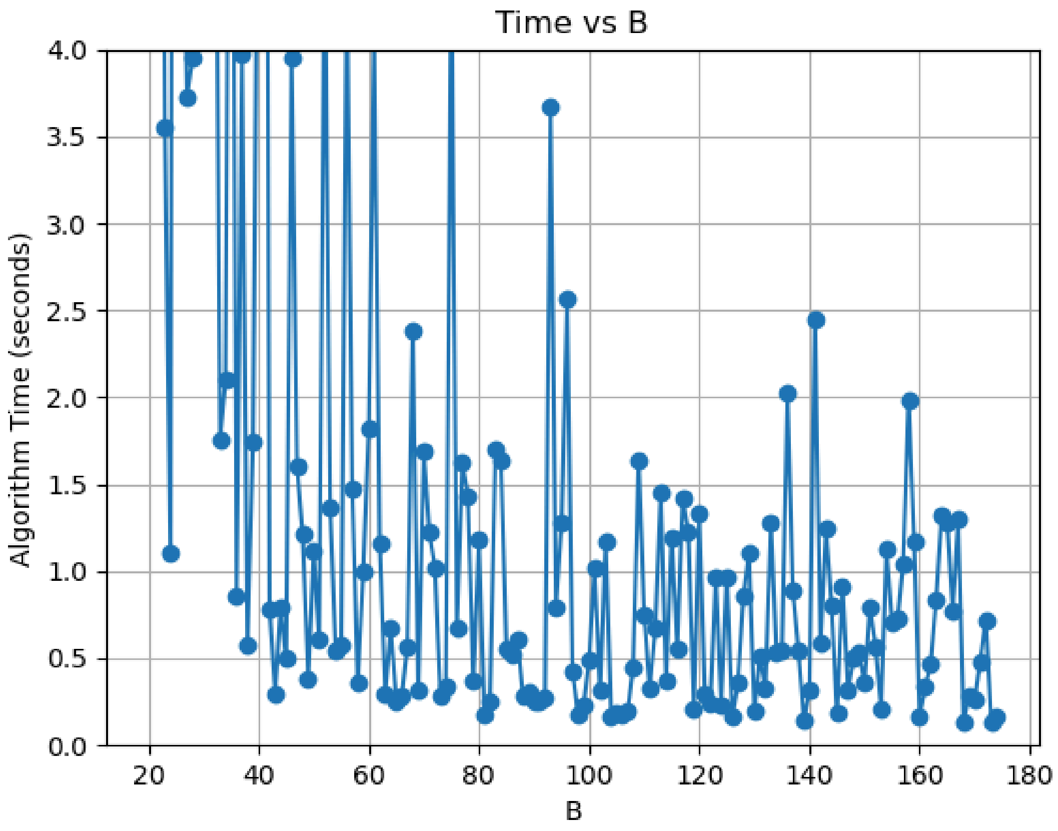


Figure 7: We see that the optimal value for B in this case is between $60 - 120$. From the large spikes before 60 it is clear that using the same B for larger primes is not optimal.