Elliptic curves in integer factorisation

Jennifer Chen, Nicholas Giannoulis, Sharvil Kesarwani,
Hrishikesh Masurkar, Thomas Newham

May 2020

Abstract

We investigated the various methods used to factorise large numbers and have provided our own implementations of the algorithms. Specifically, we focused on Pollard's $p-1$ method and Lenstra's Elliptic Curve Method and ran experiments to determine the time efficiency of our implementations. Our investigation also highlights the strengths and weaknesses of each method.

1 Introduction

Before we begin our investigation, we present some definitions:

Definition 1.1. *Let* $\gcd(a, b)$ *denote the greatest common divisor of two integers* a *and* b*.* $\forall d \in \mathbb{N}$, $d \mid a$ *and* $d \mid b \implies d \leq \gcd(a, b)$

Definition 1.2. *Denote by* $\varphi(n)$ *Euler's Totient Function, the number of positive integers less than or equal to* n *to which* n *is coprime. That is,* $\varphi(n) = |\{i \mid \gcd(i, n) = 1, 0 < i \leq n\}|$

Definition 1.3. *An **elliptic curve** is defined over* $\mathbb{Z}/n\mathbb{Z}$ *as a projective equation of the form:*

$$y^2t = x^3 + axt^2 + bt^3$$

where $(x : y : t)$ *are the projective coordinates, and* a *and* b *are elements of* $\mathbb{Z}/n\mathbb{Z}$ *such that* $4a^3 + 27b^2$ *is invertible modulo* n*. [1]*

Definition 1.4. *Let* B *be a positive integer. A positive integer* n *is said to be* B***-smooth*** *if all the prime divisors of* n *are less than or equal to* B*.* n *is said to be* B***-powersmooth*** *if all prime powers dividing* n *are less than or equal to* B*.*

1.1 Public Key Encryption

Factorising large numbers in fast and efficient ways is becoming increasingly significant for the field of cryptography, brought upon by the creation of public key cryptography in the 1980's. Public key encryption utilises a 'public key' to encrypt messages. These messages can then only be decrypted by a corresponding 'private key'. Whilst everyone has access to this public key, only someone with the private key can read the encrypted message, allowing messages to be sent securely.

Public key encryption is reliant on the concept of one way mathematical functions. The property that every composite number can be expressed as a product of primes is hence significant. Although it is relatively easy to generate large numbers by multiplication, there is no efficient method to work backwards from large composite numbers to find its prime factors; large numbers may even take months to factorise. Secure online communication relies on this difficulty. Hence, it's important to test how secure systems are against the best cryptography algorithms.

1.2 RSA encryption

RSA encryption is an example of Public Key Cryptography. The procedure for generating an RSA encryption scheme is provided below.

- Choose two primes p and q. For reasons described later, we ensure they're approximately the same size.
- Compute $n = pq$.
- Compute $\varphi(n)$. Since the Euler totient function is multiplicative and p and q are prime, $\varphi(n) = (p-1)(q-1)$ (The totient is multiplicative by **Lemma** 8.3.5).
- Choose an integer e such that; $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$.
- Compute d, the modular inverse of $e \pmod{\varphi(n)}$.
- Distribute the public key as a tuple (n, e).

Assuming one has a protocol for converting messages into integers less than n, a converted message m is encrypted as the remainder of $m^e \pmod{n}$. The original message m can be recovered by raising the residue to the power d and taking the residue of the resulting product mod n.

Recall that d and e are modular inverses mod $\varphi(n)$. Hence $ed = \varphi(n)k + 1$ for some positive integer k. By Euler's theorem (**Theorem** 8.2), if a is coprime to n, then $a^{\varphi(n)} \equiv 1 \pmod{n}$. Hence, assuming $m \neq p, q$:

$$\begin{aligned}(m^e)^d &= m^{ed} \\ &= m^{k\varphi(n)+1} \\ &= \left(m^{\varphi(n)}\right)^k m \\ &\equiv m \pmod{n}\end{aligned}$$

If one derives p and q and can hence derive $\varphi(n)$, then the Extended Euclidean Algorithm can be used to compute d, the inverse of $e \pmod{\varphi(n)}$, in $O(\log(n))$ time. The encryption is then at most as difficult to break as it is difficult to factorise the modulus n.

2 Factoring techniques

Naïvely one can attempt to factorise n by means of trial division with worst case complexity in $O(\sqrt{n})$. Since the factors are prime, we can reduce our search space. By the prime number theorem, the complexity of this approach is (asymptotically) $O\left(\frac{\sqrt{n}}{\log(n)}\right)$. However, the ease of access to large prime numbers (often with hundreds of digits) for use in RSA encryption schemes makes such methods untenable. In practice, a host of probabilistic methods have proven considerably more efficient.

2.1 Pollard's $p-1$ Method

Certain RSA moduli are more insecure than they first appear. Pollard observed that if the factor p minus 1 is a product of small primes, then it can be factored quickly by the $p-1$ method.

Suppose p is a prime divisor of n. Choose a positive integer a less than n. First determine if a is coprime to n. If not, then we have produced a factor of n. Otherwise, initialise an integer $B := 2$. Compute $G = \operatorname{lcm}(1, 2, \ldots, B)$. If $p-1$ is B-powersmooth, then it must divide G. By Fermat's little theorem (a corollary of **Theorem 8.2**), $a^{p-1} \equiv 1 \pmod{p}$. Hence $a^G \equiv 1 \pmod{p}$ and thus $a^G - 1$ is divisible by p. If $\gcd(a^G - 1, n) > 1$, then we have discovered a factor of n. If the factor is precisely n then the scheme fails and the procedure can be repeated for an alternative choice of a. Otherwise a non-trivial factor of n is returned. If the integers are in fact coprime, we may increment B and repeat the test.

Unfortunately, Pollard's method only works for primes p where $p-1$ is B-powersmooth. This means it is limited in its use, and primes can be chosen to avoid this potential insecurity. However, the algorithm remains incredibly useful in factoring certain semiprime integers.

3 Elliptical Curves

Elliptic curve cryptography is another type of public key cryptography, and the elliptic curve method is one of the most efficient ways currently known to factorise large numbers.

Elliptic curves are Diophantine equations of polynomial degree 3 in the form

$$y^2 = x^3 + ax^2 + bx + c$$

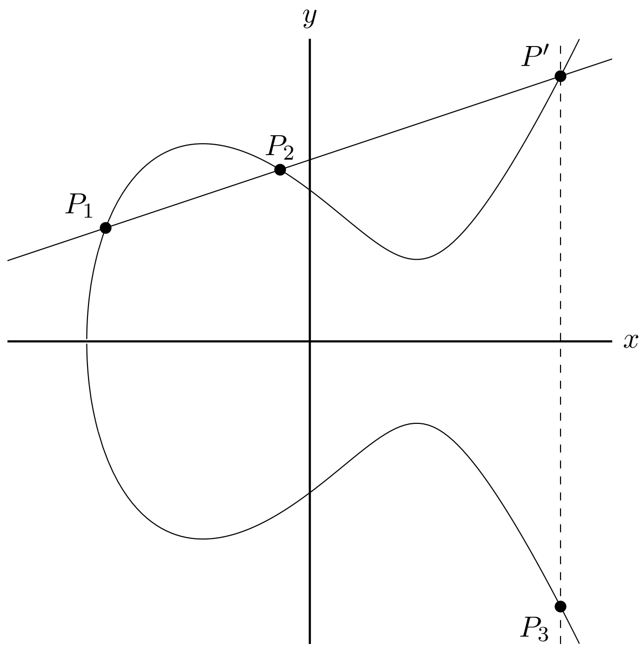
with fixed values a, b, c. [2] For our purposes when using the elliptic curve method, we use the projective form

$$y^2t = x^3 + axt^2 + t^3.$$

3.1 Point Addition on Elliptic Curves

Given 2 distinct points P_1 and P_2, point addition on elliptic curves is defined as the intersection between the elliptic curve and a straight line intersecting points P_1 and P_2.

$$P_1 + P_2 = P_3$$



where P_3 is a new point on the elliptic curve found using addition.

3.2 Group Theory for Elliptic Curves

With addition on elliptic curves now defined, the points on an elliptic curve satisfy the conditions of a group operator.

- Existence of an associative, addition operation which is closed: The addition operation is considered finding a single point P_3 from two inputs, P_1 and P_2, as mentioned in the previous section, denoted as $+$. This process is closed as P_3 will always be on the curve. For this report, we assume the operation is associative.

- Existence of inverse for each value P on the curve: The inverse of point $P(x, y)$ is $P(x, -y)$, as $P + (-P) = I$

- Existence of identity for each value P on the curve: Since there is no 'natural choice' for an identity, it is defined as a point of infinity. This satisfy $P + I = P$ for all P, the identity is I.

4 ECM Method

The ECM method, developed by H. W. Lenstra, utilises ideas of the $p - 1$ method to factorise large numbers. ECM acts on the group of points in $\mathbb{Z}/n\mathbb{Z}$ on an elliptic curve. It differs from $p-1$ and other methods as the efficiency and time taken relies on the size of the smallest prime divisor of the number rather than the size of the number itself. Although the probability of finding prime factors is relatively low for an individual curve, successively trialing many different curves increases the probability with time. ECM is also used in combination with other methods, such as with the $p - 1$ method to remove smaller factors first. The method is generally split into 2 stages, with stage 2 increasing the reach of the stage 1 method.

4.1 Stage 1

The main idea of the ECM method is to generate a large number of elliptic curves with known points, and then trial point addition and look at the properties of new points for these curves. Given two points on an elliptic curve such that $P_1 = (X_1, Y_1), P_2 = (X_2, Y_2)$, it follows that m, the slope of P_1 and P_2, will equal $\left(\frac{Y_1 - Y_2}{X_1 - X_2}\right)$ if $P_1 \neq P_2$ (if $P_1 = P_2$, consider the tangent instead). Considering the elliptic curve in $\mathbb{Z}/N\mathbb{Z}$, when $X_1 - X_2 > 0$ is not invertible, we have found a non-trivial factor of N.

The method is as follows:

1. Choose a bound B_1, and calculate all primes up to B_1. Consider the list of primes $(p_1, \ldots, p_k)$.
2. We initialise our curve, $y^2t = x^3 + axt^2 + t^3$. We also set $a = 0$.
3. We start with the trivial solution $(x : y : t) = (0 : 1 : 1)$. We also set $i = 0$
4. We increment $i \to i + 1$. If we have reached the end of our list of primes ($i > k$), we increment $a \to a + 1$, go back to step 3, and generate another elliptic curve. Otherwise, we proceed to the next step.
5. Set q to the prime we trial $q = p_i$, then set $q = q_1$. We also set $l = \left\lfloor \frac{B}{q} \right\rfloor$.
6. While $q_1 \leq l$, we perform normal multiplication $q_1 \to q \cdot q_1$. Then, using this new value of q_1, we calculate $q_1 \cdot x$ using elliptic curve addition. If $q_1 \cdot x$ is not part of the set of special points or the $n - 1$ part of E, go to step 3.
7. If the previous step fails, then this means we have found a non-invertible value.

4.2 Stage 2

Stage 2 lengthens our search space to include primes up to the square of our prior bound B_1. Denote this new bound B_2 with $B_1 < B_2 \leq B_1^2$. The method is as follows:

1. Let k_1 be the index of the greatest prime less than or equal to B_1, and k_2 that less than or equal to B_2. Construct an array d of size k_2 with $d[i] = p_{k_1+i} - p_{k_1+i-1}$.
2. Initialise x to be the final point obtained by Stage 1. For each difference D in the array d, compute $D \cdot x$, the iterative sum of x composed with itself D times under the group operation.
3. Initialise the following variables: $b = x$, $c = 0$, $P = 1$, $i = 0$, $j = i$, $y = x$.
4. Increment i until $i > k_2$, at which stage one proceeds to the final step. For each increment, set x to be its composition with b's group product with d_i (pre-computed two steps prior), set P to be its product with the third (projective) coordinate of x, and increment c. If $c > 50$, proceed to the next step.
5. Compute the gcd of P and N. If they are coprime, set $c \to 0$, $j \to i$ and $y \to x$. Otherwise proceed to the next step.
6. Set $i \to j$ and x to y's group product with d_i (again, pre-computed several steps earlier). Proceed with incrementing i. For each increment compute the gcd of the third (projective) coordinate of x and N. If they are coprime, continue incrementing. If the gcd is precisely N, Stage 2 has failed and the algorithm can be reset (from Stage 1) with the incremented coefficient. Otherwise, return the non-trivial factor obtained.

7. If the primes below B_2 are exhausted, compute the gcd of the third (projective) coordinate of x and N. If they are coprime, Stage 2 has failed to find a factor and as in the previous step the algorithm may be tried anew. Otherwise, regress to the step immediately prior to this.

5 Results and Discussion

During the testing of our own implementations of the factorisation methods, we observed that Pollard's $p-1$ method was significantly slower than the ECM methods. This result aligns with the theoretical run times of each algorithm. Specifically, Pollard's $p-1$ method runs in $O(B\log(B)\log^2(n))$, where n is the integer to factorise and B is the bound chosen for the algorithm. On the other hand, the first phase of the ECM algorithm takes $O\left(\exp(\sqrt{(2+o(1))\log(n)\log(\log(n))}\right)$, where $o(1)\to 0$ as $n\to 0$.

We also optimised constant parameters in the algorithm using an experimental process. Evaluating the times it took to compute factors for various length semiprimes, we were able to develop a fairly optimised algorithm that performed well within the bounds we had. Unfortunately, we could not extensively test this due to lack of experience and being unable to access higher-order optimisations (such as accessing the microprocessor). Table 1 in **Appendix 8.1** displays our results on checking gcd of t (the third coordinate of a point) and the number we are factorising, after c iterations of sums.

Comparison of our implementation with open-source software

To test performance of our program, we compared the time taken for the algorithm to factorise numbers with existing open-source software. We mainly used Yafu [4], although we also used GMP-ECM [3]. Our algorithm was, as expected, much slower than our open source program tests. This is because these programs have been much more optimised, and often use other techniques in addition to the ECM method, such as the quadratic sieve method in the case of Yafu. At best, our algorithm is a factor of 10 slower than the source, a suitable value considering our lack of experience with factorisation. The data for this can be seen in Table 2 in **Appendix 8.1**.

6 Conclusion

Having explored the area of cryptography, we can see the importance of examining factorisation. The evolution of factorising methods, from Pollard's $p-1$ method to the elliptical curve method showcase the progress and optimisations made in the field of factoring. Whilst there are faster algorithms out there, the ECM algorithm we produced still runs rather quickly up to over 60 digits. Further optimisations and a greater computing power would lead to even faster results for larger amounts of digits. However, in spite of even the best cryptography algorithms, public key encryption remains secure.

7 References

1. Cohen, H., 2011. A Course In Computational Algebraic Number Theory. Berlin: Springer, pp.486, 487.

2. Silverman, Joseph H., 2013. A Friendly Introduction to Number Theory (4th ed). Upper Saddle River, N.J: Pearson

3. Zimmerman.P , 2017-07-04, GMP-ECM (Elliptic Curve Method for Integer Factorization) (7.0.4), https://gforge.inria.fr/projects/ecm/

4. Ben, 2013-03-06, Yafu (1.34), https://sourceforge.net/projects/yafu/

5. Apostol, Tom M., 1976, Introduction to Analytic Number Theory,. Pasadena, California: Springer

8 Appendices

8.1 Tables

Table 1: Average time taken to run through semiprimes of various lengths (s)

Number of Digits	$c = 1$	$c = 2$	$c = 5$	$c = 10$	$c = 20$
27	0.08015758	0.07939693	0.07902595	0.07973419	0.11152951
30	0.21809523	0.21713862	0.21824571	0.21542426	0.274051
33	3.1225028	3.06994976	3.06897962	3.08630598	3.4493613
36	1.32812167	1.27466422	1.28786484	1.42127248	1.63183646
38	0.61992311	0.60201616	0.60394815	0.75608246	0.79902613
41	4.81359401	4.95981157	4.4858182	4.42236209	4.41786652
44	10.2539276	11.3143479	9.46911647	9.34732125	9.33672123
46	15.1146262	14.6253772	13.7342625	13.6125052	34.2703061
49	5.93740816	5.76939225	5.44948943	5.38941472	5.38520081
51	45.0016973	43.1434744	41.5519542	41.1614151	41.2873801
Number of Digits	$c = 50$	$c = 100$	$c = 200$	$c = 500$	$c = 1000$
27	0.10959072	0.08935096	0.10044167	0.10225796	0.10611787
30	0.28888226	0.2244417	0.24273366	0.27882808	0.27692071
33	3.28818365	3.14290271	3.20873131	3.29039963	3.30412158
36	1.28661658	1.31558753	1.34502388	1.3096759	1.35426674
38	0.61689538	0.62900124	0.63755214	0.68369458	0.71252918
41	4.4231214	4.42383358	4.46619725	4.87429531	4.50085261
44	9.33119864	9.35819473	9.57617071	9.93858202	9.34642982
46	34.2726183	34.3186843	39.2926382	92.1007764	95.7536913
49	5.39258878	5.58025248	7.07249055	5.74680698	7.39974327
51	41.1831525	42.686522	47.3452537	54.7998222	84.4902343

Table 2: Comparison of our algorithm with various open source algorithms

Number	Factor	Time- Open Source Program	Our Time
47993094467069810000 7569	694206942013371 337, 691337	0.0123	0.107682943344116
55256537361946032470 2771457	888564021963695 017, 621863321	0.0309	0.209025144577026
92844654043344152913 572776253	694206942013371 337, 133742042069	0.0677	0.83380103111267
27223212146300374042 6227904199	4751461961, 5729 4391430992188559	0.0299	0.333078145980834
54432399757348512054 11645234644653525724 306138561	17305892679709063, 31453101417397440 5756131086000247	0.0936/18.845	6.1383421421051
15580928761561444176 40678493166800019820 516621834733	1178315166983119 2842968259132229 259, 132230571226 998887	0.1475*	44.9054200649261
48937707171184596312 03705326908842847315 60434462861233	59568447461339782 3, 821537395329052 53010243698591581 4671	0.4487	53.7072329521179
15386770741252012512 25976475835148653027 49460861942017267893	31636044111475582 94124221554385950 725953, 4863683552 5433636981	0.0568	175.471575021743
56483946555634522015 50997312414998989974 96991465859656296517 60909	43283596493851869 95641, 13049735033 83843549547761775 7325875364994549	10.7647*	830.567373991012
63134840925127838837 96707109367883239865 33189381436734250775 3125718037516787	17350136637536842 687756151, 3638867 07315815933986211 90513600386266866 5066459237	58.0104*	5063.68022322654
19487611285929535286 10606288376682022569 56751952656220196874 3760674455899140957	18256204121883436 01519402443240119 23497387688000039 27, 10674514349108 3286645046891	125.3498*	8198.74480319023

*: Indicates value was found using a combination of ECM and quadratic sieve method.

8.2 Proofs

8.2.1 The GCD

Lemma 8.0.1. *If a and b are positive integers with $a \geq b$. Then* $\gcd(a, b) = \gcd(a - b, b)$.

Proof. Let $g_1 := \gcd(a, b)$ and $g_2 := \gcd(a - b, b)$. We have

$$\begin{aligned} g_1 \mid a, b &\implies g_1 \mid a - b \implies g_1 \leq g_2 \\ g_2 \mid a - b, b &\implies g_2 \mid (a - b) + b \implies g_2 \leq g_1 \end{aligned}$$

Therefore, $g_1 = g_2$. The result follows. □

Lemma 8.0.2 (Bézout's Identity)**.** *For positive integers a and b, there exist integer solutions x and y to the equation $ax + by = \gcd(a, b)$.*

Proof. If $a = b = 1$, observe that $ax + by = \gcd(a, b)$ is trivially satisfied with $x = 1, y = 0$. Now, assume true for all positive integer pairs with greatest element less than or equal to k. Consider a positive integer pair $(k + 1, j)$ with $1 < j \leq k$. By **Lemma 8.0.1** $\gcd(k + 1, j) = \gcd(k + 1 - j, j)$. Hence, by our assumption we have $x(k + 1 - j) + yj = x(k + 1) + (y - x)j = \gcd(k + 1, j)$. The lemma is proved true by strong induction. □

Lemma 8.0.3 (Euclid's Lemma)**.** *Suppose p is a prime dividing ab. Then p divides at least one of a and b.*

Proof. Suppose p does not divide a. Then $\gcd(a, p) = 1$, and so by **Lemma 8.0.2** there are integer solutions x and y to the equation $ax + py = 1$.

$$\begin{aligned} ax + py &= 1 \\ abx + pby &= b \\ \frac{ab}{p}x + by &= \frac{b}{p} \end{aligned}$$

Since the left hand side is an integer, we conclude that b must be divisible by p also. □

8.2.2 The Fundamental Theorem of Arithmetic

Theorem 8.1. *Every positive composite integer may be expressed uniquely (up to permutation) as a product of primes.*

Proof. Let a composite positive integer n equal $p_1 p_2 ... p_k$. Suppose n were to admit a factorisation $p'_1, p'_2, ... p'_j$. Since the factorisations are equal, we may divide both by p_1. By Euclid's Lemma, p_1 must divide exactly one of $p'_1, p'_2, ..., p'_j$. For one prime to divide another, they must be equal, hence exactly 1 prime is removed from each factorisation. We proceed in the same fashion until one of the factorisations is exhausted.

If one is exhausted before the other, then we have that a product of primes is equal to 1, which is impossible. Hence, the factorisations must be exhausted simultaneously and must therefore be equal being composed of precisely the same primes.

In general we may express an integer n (uniquely) as $n = \prod_{p \in \mathbb{P}} p^{a_p}$ for some set of non-negative integers $\{a_p\}$. ☐

Lemma 8.1.1. *For two positive integers a and b with $a = \prod_{p \in \mathbb{P}} p^{a_p}$ and $b = \prod_{p \in \mathbb{P}} p^{b_p}$,*

$$\gcd(a, b) = \prod_{p \in \mathbb{P}} p^{\min\{a_p, b_p\}}$$

Proof. If the exponent of a prime p in the product form of the gcd exceeds the least of a_p or b_p then it will fail to divide either one or both of a or b. Hence the largest possible exponent for a prime dividing both a and b is $\min\{a, b\}$. ☐

Lemma 8.1.2. *If d divides both a and b, then d divides* $\gcd(a, b)$.

Proof. By definition, gcd is the largest number that divides a and b simultaneously. Hence, if $d \mid a, b$, then if $d \nmid \gcd(a, b)$, then there exists $n > 1$ dividing d that doesn't divide $\gcd(a, b)$. Hence, $n \mid a, b$, which implies $n \cdot \gcd(a, b) \mid a, b$. But $n \cdot \gcd(a, b) > \gcd(a, b)$, which is a contradiction.

Hence, $d \mid \gcd(a, b)$. ☐

8.2.3 Modular Multiplicative Groups

Theorem 8.2. *For a given n, the set $\{a \mid \gcd(a, n) = 1\}$ forms a group under multiplication modulo n.*

Proof. The existence of an identity element and the associativity are inherited trivially from the properties of real multiplication. It remains to prove that the group is closed and contains the inverse of each of its elements.

Let a and b (not necessarily distinct) be two elements of the set. Suppose the product ab was not coprime to n. Let p be a prime divisor of both ab and n. Then by **Lemma 8.0.3**, p divides at least one of a or b. But this is a contradiction since a and b by merit of their inclusion in the set are coprime with n. Hence the reisdue of ab must be a member of the set.

Consider an element of the set a. Since $\gcd(a, p) = 1$, by **Lemma 8.0.2** there exist integer solutions x and y to the equation $ax + ny = 1$. In particular, the product ax is congruent to one modulo n, and x is the inverse of a modulo n. ☐

8.2.4 Euler's Theorem

Theorem 8.3. *If a is coprime to n, then $a^{\varphi(n)} \equiv 1 \pmod{n}$.*

Proof. Denote the set $\{a, a^2, \ldots, a^{k-1}, 1\}$ of residues generated by a A, where k is the cardinality of the set and hence the smallest exponent for which a is congruent to 1. Let g and h be two residues mod n coprime to n. Let gA denote the set $\{g \cdot a^* \mid a^* \in A\}$.

Suppose that for some indices i and j with $k \geq i > j$ $ga^i = ga^j$. Then $a^{i-j} \equiv 1$ with $i - j < k$, which is a contradiction since k was said to be minimal. Hence the cardinality of gA is exactly k.

Suppose now that for two indices i and j under the same conditions $ga^i = ha^j$. Then $h = ga^{i-j}$. In particular, h is a member of gA. Hence, $hA = gA$.

Since every set belongs to some unique coset, all of which have cardinality k, we conclude that k divides the number of residues coprime to n ($\varphi(n)$). Hence $kl = \varphi(n)$ for some integer l.

$$\begin{aligned} a^{\varphi(n)} &= a^{kl} \\ &= \left(a^k\right)^l \\ &\equiv 1^l \bmod n \\ &\equiv 1 \bmod n \end{aligned}$$

□

8.2.5 Euler's Totient Function

Lemma 8.3.1. *For positive integers $n > 1$, $\sum_{d|n} \varphi(d) = n$.*

Proof. Define the function f on the set of divisors of n such that $f(d) = |\{k \mid \gcd(k, n) = d, 1 < k < n\}|$. Then $\sum_{d|n} f(d) = n$. There are $\frac{n}{d}$ positive multiples of d less than or equal to n. There are then $\varphi\left(\frac{n}{d}\right)$ of these which do not share factors with n greater than d. Hence, $f(d) = \varphi\left(\frac{n}{d}\right)$. Since iterating over the complete set of quotients of an integers is equivalent to iterating over the complete set of its divisors, $\sum_{d|n} \varphi\left(\frac{n}{d}\right) = \sum_{d|n} \varphi(d) = n$. □

Definition 8.1. *Denote by $\mu(n)$ the Möbius function which signifies the 'prime parity' of a function for 'square free' integers greater than 1. If $\exists a \in \mathbb{Z}$ such that a^2 divides n, then $\mu(n) = 0$, otherwise $\mu(n) = (-1)^k$ where k is the number of distinct prime divisors of n Define $\mu(1)$ to be 1.*

Lemma 8.3.2. *For positive integers $n > 1$, $\sum_{d|n} \mu(d) = 0$*

Proof. Denote the prime factors of n, $p_1, p_2, ..., p_k$. The only divisors d of n for which $\mu(d) \neq 0$ are those which are products of some subset of the primes p_1 through p_k. There are $\binom{k}{i}$ products of exactly i of the prime factors. Hence;

$$\begin{aligned} \sum_{d|n} \mu(d) &= \sum_{i=0}^{k} \binom{k}{i} (-1)^k \\ &= (1-1)^k \\ &= 0 \end{aligned}$$

□

Lemma 8.3.3. *For positive integers $n > 1$, $\sum_{d|n} \mu(d) \frac{n}{d} = \varphi(n)$*

Proof. We may write $\varphi(n)$ as $\sum_{k=1}^{n} \left\lfloor \frac{1}{\gcd(k,n)} \right\rfloor$. Since $\mu(1) = 1$ and by **Lemma 8.3.2** the sum $\sum_{d|n} \mu(d)$ is equal to 0 for $n > 1$, we may further write $\varphi(n)$ as $\sum_{k=1}^{n} \sum_{d|\gcd(k,n)} \mu(d)$. This sum iterates over the common divisors of n and k for each positive integer less than or equal to n. For a given divisor d of n, $\mu(d)$

occurs in the sum precisely when k is a multiple of d. Hence, we may equivalently iterate through the divisors d of n adding $\mu(d)$ for each index k which is a positive multiple of d less than or equal to n.

$$\begin{aligned}\varphi(n) &= \sum_{k=1}^{n} \sum_{d|\gcd(k,n)} \mu(d) \\ &= \sum_{d|n} \sum_{i=1}^{\frac{n}{d}} \mu(d) \\ &= \sum_{d|n} \mu(d) \sum_{i=1}^{\frac{n}{d}} \\ &= \sum_{d|n} \mu(d) \frac{n}{d}\end{aligned}$$

□

Lemma 8.3.4. $\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$ *where p iterates over primes.*

Proof. Denote the distinct prime factors of n be $p_1, p_2, ..., p_k$. We may write our product instead as $n \prod_{i=1}^{k} \left(1 - \frac{1}{p}\right)$. Expanding this product one obtains n plus the reciprocal of every product of distinct prime factors of n multiplied by n and the mobius function of the product.

$$\begin{aligned}n \prod_{p|n} \left(1 - \frac{1}{p}\right) &= n + \sum_{i<k} \frac{n\mu(p_i)}{p_i} + \sum_{j<i<k} \frac{n\mu(p_i p_j)}{p_i p_j} + ... + \frac{n\mu(p_1 p_2, ...p_k)}{p_1 p_2, ...p_k} \\ &= \sum_{d|n} \mu(d) \frac{n}{d} \\ &= \varphi(n)\end{aligned}$$

□

Lemma 8.3.5. $\varphi(mn) = \varphi(m)\varphi(n) \frac{\gcd(m,n)}{\varphi(\gcd(m,n))}$

Proof. The primes which divide both m and n are precisely those which divide $\gcd(m,n)$. Hence (by **Lemma 8.3.4**),

$$\begin{aligned}\varphi(mn) &= mn \prod_{p|nm} \left(1 - \frac{1}{p}\right) \\ &= \frac{m \prod_{p|m} \left(1 - \frac{1}{p}\right) n \prod_{p|n} \left(1 - \frac{1}{p}\right)}{\prod_{p|\gcd(n,m)} \left(1 - \frac{1}{p}\right)} \\ &= \varphi(m)\varphi(n) \frac{\gcd(n,m)}{\left(\gcd(n,m) \prod_{p|\gcd(n,m)} \left(1 - \frac{1}{p}\right)\right)} \\ &= \varphi(m)\varphi(n) \frac{\gcd(m,n)}{\varphi(\gcd(m,n))}\end{aligned}$$

□

9 Code

9.1 Factoring and Primality Testing

```
import time
import math
import random
import numpy as np
from bisect import bisect

# extended euclidean algorithm
def bin_euclid(num_1, num_2):

    flag_1 = (num_1 < num_2)
    if flag_1:
        temp = num_1
        num_1 = num_2
        num_2 = temp

    if num_2 == 0:
        if flag_1:
            return (0, 1, num_1)
        else:
            return (1, 0, num_1)

    else:
        quotient, rem = divmod(num_1, num_2)
        num_1 = num_2
        num_2 = rem
        if num_2 == 0:
            if flag_1:
                return (1, 0, num_1)
            else:
                return (0, 1, num_1)

        else:
            power_of_two = 0
            while not (num_1 & 1) and not (num_2 & 1):
                power_of_two += 1
                num_1 >>= 1
                num_2 >>= 1

            flag_2 = not (num_2 & 1)
            if flag_2:
                temp = num_1
                num_1 = num_2
                num_2 = temp

            coeff_1 = 1
            gcd_num = num_1
            par_2_coeff_1 = num_2
```

```
par_2_coeff_3 = num_2

if num_1 & 1:
    par_1_coeff_1 = 0
    par_1_coeff_3 = -num_2

else:
    par_1_coeff_1 = (num_2 + 1) >> 1
    par_1_coeff_3 = num_1 >> 1
    while not (par_1_coeff_3 & 1):
        par_1_coeff_3 >>= 1
        if (par_1_coeff_1 & 1):
            par_1_coeff_1 = (par_1_coeff_1 + num_2) >> 1
        else:
            par_1_coeff_1 >>= 1

if par_1_coeff_3 > 0:
    coeff_1 = par_1_coeff_1
    gcd_num = par_1_coeff_3

else:
    par_2_coeff_1 = num_2 - par_1_coeff_1
    par_2_coeff_3 = -par_1_coeff_3

par_1_coeff_1 = coeff_1 - par_2_coeff_1
par_1_coeff_3 = gcd_num - par_2_coeff_3

if par_1_coeff_1 < 0:
    par_1_coeff_1 += num_2

while par_1_coeff_3 != 0:
    while not (par_1_coeff_3 & 1):
        par_1_coeff_3 >>= 1
        if (par_1_coeff_1 & 1):
            par_1_coeff_1 = (par_1_coeff_1 + num_2) >> 1
        else:
            par_1_coeff_1 >>= 1

    if par_1_coeff_3 > 0:
        coeff_1 = par_1_coeff_1
        gcd_num = par_1_coeff_3

    else:
        par_2_coeff_1 = num_2 - par_1_coeff_1
        par_2_coeff_3 = -par_1_coeff_3

    par_1_coeff_1 = coeff_1 - par_2_coeff_1
    par_1_coeff_3 = gcd_num - par_2_coeff_3

    if par_1_coeff_1 < 0:
        par_1_coeff_1 += num_2
```

```
            coeff_2 = (gcd_num - num_1 * coeff_1) // num_2
            gcd_num <<= power_of_two
            if flag_2:
                temp = coeff_1
                coeff_1 = coeff_2
                coeff_2 = temp

            coeff_1 -= coeff_2 * quotient

            if flag_1:
                return (coeff_1, coeff_2, gcd_num)
            else:
                return (coeff_2, coeff_1, gcd_num)

# modular inverse
def inverse(base, modulus):
    inv, fac, d = bin_euclid(base, modulus)
    invertible = (d == 1)
    if invertible:
        return invertible, inv % modulus
    else:
        return invertible, [base, modulus, d]


# bit-slicing
def inclusive_digit(number, index_1, index_2):
    return (((1 << (index_2 - index_1 + 1)) - 1) & (number >> index_1))

# converting numbers to binary for grouping
def flexible(number, base):
    odd=[]
    indices = []

    indices.append(int(math.log2(number & (~(number - 1)))))
    index = 0
    curr_idx = indices[0]

    odd.append(inclusive_digit(number, curr_idx, curr_idx + base - 1))

    new_number = (number >> (curr_idx + base))
    while new_number != 0:
        index += 1
        indices.append(int(math.log2(new_number & (~(new_number - 1)))) + base)

        curr_idx += indices[index]
        odd.append(inclusive_digit(number, curr_idx, curr_idx + base - 1))

        new_number = (number >> (curr_idx + base))
    else:
        return odd, indices, index

# grouping constant for sliding window
```

```python
vals = [(k + 1) * (k + 2) * pow(2, k-1) + 1 for k in range(1, 55)]
def group_cons_func(num, breakpoints=vals):
    if num <= (1 << 64):
        return bisect(breakpoints, num) + 1
    else:
        return 55

# efficient exponentiation
def power(base, index, modulus=0):
    if index == 0:
        return 1
    else:
        if index < 0:
            abs_index = -index
            new_base = 1/base
            if modulus:
                invertible, new_base = inverse(base, modulus)
                if not invertible:
                    raise ValueError(f"{base} is not invertible in mod {modulus}")

        else:
            abs_index = index
            new_base = base
            if modulus:
                new_base %= modulus

        logval = math.log2(abs_index)
        group_cons = group_cons_func(logval)

        odd, indices, len_val = flexible(abs_index, group_cons)
        curr_idx = len_val

        squared = new_base * new_base
        if modulus:
            squared %= modulus
        odd_powers = [new_base]
        for _ in range(3, (1 << group_cons), 2):
            next_val = odd_powers[-1] * squared
            if modulus:
                next_val %= modulus
            odd_powers.append(next_val)

        if curr_idx == len_val:
            prod = odd_powers[(odd[curr_idx]-1) >> 1]
        else:
            prod *= odd_powers[(odd[curr_idx]-1) >> 1]
            if modulus:
                prod %= modulus

        for _ in range(indices[curr_idx]):
            prod *= prod
            if modulus:
```

```
                prod %= modulus

        while curr_idx != 0:
            curr_idx -= 1
            if curr_idx == len_val:
                prod = odd_powers[(odd[curr_idx]-1) >> 1]
            else:
                prod *= odd_powers[(odd[curr_idx]-1) >> 1]
                if modulus:
                    prod %= modulus

            for _ in range(indices[curr_idx]):
                prod *= prod
                if modulus:
                    prod %= modulus
        else:
            return prod

# euler totient function
def euler_totient(n):
    value = n
    prime_dict = factorint(n)
    for p in prime_dict:
        value //= p
        value *= p-1
    return value

# order of an element in multiplicative group
def order(g, group):
    h = euler_totient(group)
    prime_dict = factorint(h)
    prime_factors = [(i, prime_dict[i]) for i in prime_dict]
    k = len(prime_factors)
    e = h
    i = 0
    while i < k:
        p_i = prime_factors[i][0]
        v_i = prime_factors[i][1]
        e //= power(p_i, v_i)
        g_1 = power(g, e, group)
        while g_1 != 1:
            g_1 = power(g_1, p_i, group)
            e *= p_i
        i += 1
    else:
        return e

# finds primitive root in mod p
def primitive_root(p):
    prime_dict = factorint(p-1)
    primes = [i for i in prime_dict]
    k = len(primes)
```

```
    i = 0
    a = 2
    while i < k:
        p_i = primes[i]
        e = pow(a, (p - 1) // p_i, p)
        if e != 1:
            i += 1
        else:
            a += 1
            i = 0
    else:
        return a

# checks if num is a QR in the mod
def QR_kronecker(num, mod):
    if mod == 0:
        if abs(num) != 1:
            return 0
        else:
            return 1
    else:
        if not (num & 1) and not (mod & 1):
            return 0
        else:
            powers_of_neg_1 = [0, 1, 0, -1, 0, -1, 0, 1]
            powers_of_two = 0
            while not (mod & 1):
                powers_of_two += 1
                mod >>= 1
            if not (powers_of_two & 1):
                output = 1
            else:
                output = powers_of_neg_1[num & 7]
            if mod < 0:
                mod = -mod
                if num < 0:
                    output = -output
            while num != 0:
                powers_of_two = 0
                while not (num & 1):
                    powers_of_two += 1
                    num >>= 1
                if (powers_of_two & 1):
                    output *= powers_of_neg_1[mod & 7]
                if (num & mod & 2):
                    output = -output
                rem = abs(num)
                num = mod % rem
                mod = rem
                if num > (rem >> 1):
                    num -= rem
            else:
```

```
                if mod > 1:
                    return 0
                else:
                    return output

# checks if num is QR in the mod
def bin_kronecker(num, mod):
    if mod == 0:
        if abs(num) != 1:
            return 0
        else:
            return 1
    else:
        if not (num & 1) and not (mod & 1):
            return 0
        else:
            powers_of_neg_1 = [0, 1, 0, -1, 0, -1, 0, 1]
            powers_of_two = 0
            while not (mod & 1):
                powers_of_two += 1
                mod >>= 1
            if not (powers_of_two & 1):
                output = 1
            else:
                output = powers_of_neg_1[num & 7]
            if mod < 0:
                mod = -mod
                if num < 0:
                    output = -output
            while num != 0:
                powers_of_two = 0
                while not (num & 1):
                    powers_of_two += 1
                    num >>= 1
                if (powers_of_two & 1):
                    output = powers_of_neg_1[mod & 7]
                rem = mod - num
                if rem > 0:
                    if (num & mod & 2):
                        output = -output
                    mod = num
                    num = rem
                else:
                    num = -rem
            else:
                if mod > 1:
                    return 0
                else:
                    return output

#sqrt in mod prime
def mod_sqrt(num, prime):
```

```
    if num == 0:
        return 0
    else:
        e = 0
        q = prime - 1
        while not (q & 1):
            e += 1
            q >>= 1

        n = random.randint(1, prime - 1)
        while QR_kronecker(n, prime) != -1:
            n = random.randint(1, prime - 1)
        z = pow(n, q, prime)

        y = z
        r = e
        x = pow(num, (q - 1) >> 1, prime)
        b = (num * x * x) % prime
        x = (num * x) % prime

        flag = 0
        while b % prime != 1 and not flag:
            m = 1
            while pow(b, 1 << m, prime) != 1:
                m += 1
            if m == r:
                flag = 1
            else:
                t = pow(y, 1 << (r - m - 1), prime)
                y = (t * t) % prime
                r = m % prime
                x = (x * t) % prime
                b = (b * y) % prime
        else:
            if flag:
                raise ValueError(f"square root of {num} doesn't exist mod {prime}")
            else:
                return x

# x^2 + dy^2 = p
def prime_pell(d, p):
    if d >= p or d <= 0:
        raise ValueError
    else:
        k = QR_kronecker(-d, p)
        if k == -1:
            raise ValueError
        else:
            x_0 = mod_sqrt(-d, p)
            if x_0 < (p >> 1):
                x_0 = p - x_0
            a = p
```

```
            b = x_0
            l = int(np.sqrt(p))
            while b > l:
                r = a % b
                a = b
                b = r
            if (p - b * b) % d != 0 or (c := (p - b * b) // d) != (p - b * b) / d:
                raise ValueError(f"x^2 + {d}y^2={p} has no solutions")
            else:
                return (b, np.sqrt(c))

# x^2 + |d|y^2 = 4p
def mod_prime_pell(d, p):
    if p == 2:
        if int(np.sqrt(d + 8)) == np.sqrt(d + 8):
            return (np.sqrt(d + 8), 1)
        else:
            raise ValueError(f"x^2 + |{d}|y^2=4{p} has no solutions")
    else:
        k = QR_kronecker(d, p)
        if k == -1:
            raise ValueError
        else:
            x_0 = mod_sqrt(d, p)
            if not ((x_0 & 1) ^ (d & 1)):
                x_0 = p - x_0
            a = p << 1
            b = x_0
            l = int(2 * np.sqrt(p))
            while b > l:
                r = a % b
                a = b
                b = r
            else:
                if ((p << 2) - b * b) % d != 0 or (c := ((p << 2) - b * b) // d) != (p - b * b) /
                    raise ValueError(f"x^2 + |{d}|y^2=4{p} has no solutions")
                else:
                    return (b, np.sqrt(c))


#############################################################

# Generate small primes up to a million for small stuff
very_small_file = open('very_small_primes.txt', 'r')
lines = very_small_file.readlines()
very_small_primes = list(map(int, [line.strip().split(', ') for line in lines][0]))

# Generate primes up to a million for "small cases"
small_file = open('small_primes.txt', 'r')
lines = small_file.readlines()
small_primes = list(map(int, [line.strip().split(', ') for line in lines][0]))
```

```
# Generate prime differences up to a million
prime_diff_file = open('prime_diffs.txt', 'r')
lines = prime_diff_file.readlines()
prime_diffs = list(map(int, [line.strip().split(', ') for line in lines][0]))

# Generate primes up to a 2 million for big stuff
big_file = open('primes_under_2mil.txt', 'r')
lines = big_file.readlines()
big_primes = list(map(int, [line.strip().split(', ') for line in lines][0]))

# Factorising numbers with small primes
def small_factors(n, factors=[]):
    if n == 1:
        factors.append(1)
        return factors
    else:
        i = 0
        prime_found = False
        while i < len(small_primes) and not prime_found:
            prime = small_primes[i]
            if n % prime == 0:
                k = 0
                while n % prime == 0:
                    n //= prime
                    k += 1
                factors.append((prime, k))
                prime_found = True
                break
            else:
                i += 1
        if prime_found:
            return small_factors(n, factors)
        else:
            factors.append(n)
            return factors

# For Miller-Rabin Primality Testing
def try_composite(a, d, n, s):
    if pow(a, d, n) == 1:
        return False
    for i in range(s):
        if pow(a, (1 << i) * d, n) == n-1:
            return False
    return True

# Miller-Rabin Primality Testing, deterministic for up to 2^64
def miller_rabin(n, precision_for_huge_n=16):
    if any((n % p) == 0 for p in small_primes) or n in (0, 1):
        return False
    d, s = n - 1, 0
    while not (d & 1):
        d, s = d >> 1, s + 1
```

```
    if n < (1 << 64):
        if n == 299210837:
            return True
        else:
            return not any(try_composite(a, d, n, s) for a in (2, 325, 9375, 28178, 450775, 97805
    else:
        return not any(try_composite(a, d, n, s) for a in small_primes[:precision_for_huge_n])

# Pollard p-1 method
def pollard(N, B=1000000):
    if B != 1000000:
        prime_list = list(sieve.primerange(1, B))
    else:
        prime_list = small_primes
    k = len(prime_list)

    x = 2 # set x = 3 if last while loop fails
    y = x
    c = 0
    i = 0
    j = i

    i += 1
    backtrack_flag = False
    while not backtrack_flag:
        if c < 20:
            if i > k:
                g = math.gcd(x-1, N)
                if g == 1:
                    raise ValueError
                else:
                    i = j
                    x = y
                    backtrack_flag = True
            else:
                q = prime_list[i - 1]
                q_1 = q
                l = B // q
                while q_1 <= l:
                    q_1 *= q
                x = pow(x, q_1, N)
                c += 1
        else:
            g = bin_euclid(x-1, N)[2]
            if g == 1:
                c = 0
                j = i
                y = x
            else:
                i = j
                x = y
                backtrack_flag = True
```

```
    finished_flag = False
    while not finished_flag:
        i += 1
        q = prime_list[i - 1]
        q_1 = q
        x = pow(x, q, N)
        g = math.gcd(x-1, N)
        backtrack2_flag = False
        while not backtrack2_flag:
            if g == 1:
                q_1 *= q
                if q_1 <= B:
                    x = pow(x, q, N)
                    g = math.gcd(x-1, N)
                else:
                    backtrack2_flag = True
            else:
                if g < N:
                    backtrack2_flag = True
                    finished_flag = True
                    return g
                else:
                    raise ValueError

# medium_file = open('primes_under_2mil.txt', 'r')
# lines = small_file.readlines()
# b1_primes = [line.strip().split(,) for line in lines][0]
# bigfile = open('primes_under_2^32.txt', 'r')
# lines = small_file.readlines()
# b2_primes = [line.strip().split(,) for line in lines][0]
# b2_diffs = [b2_primes[i] - b2_primes[i-1] for i in range(1, len(b2_primes))]


def pollard_factor(n, factors=[]):
    if factors == []:
        factors = small_factors(n)
    if factors[-1] == 1:
        return factors
    else:
        try:
            num = factors.pop()
            p = pollard(num)
        except ValueError:
            factors.append(num)
            return factors
        k = 0
        while num % p == 0:
            num //= p
            k += 1
        factors.append((p, k))
        factors.append(num)
        return pollard_factor(num, factors)
```

```python
# y^2 t = x^3 + a x t + t^3
# ECM Addition using projective points, without any division
def ECM_sum(point_1, point_2, a, mod):
    x1, y1, t1 = point_1
    x2, y2, t2 = point_2
    if point_1 == point_2:
        if y1 == 0:
            return False, (x1, y1, 0)
        else:
            T = (3 * x1**2 + a * t1**2) % mod
            U = (y1 * t1 << 1) % mod
            V = (U * x1 * y1 << 1) % mod
            W = (T**2 - 2 * V) % mod
            x3 = (U * W) % mod
            y3 = (T * (V - W) - 2 * (U * y1)**2) % mod
            t3 = (U**3) % mod
            invertible = (t3 != 0)
            if invertible:
                return invertible, (x3, y3, t3)
            else:
                return invertible, (point_1, point_2, a, mod)
    else:
        if (x1 * t2 - x2 * t1) % mod == 0:
            return (False, (x1, y1, 0))
        else:
            T0 = (y1 * t2) % mod
            T1 = (y2 * t1) % mod
            T = (T0 - T1) % mod
            U0 = (x1 * t2) % mod
            U1 = (x2 * t1) % mod
            U = (U0 - U1) % mod
            U2 = (U**2) % mod
            U3 = (U * U2) % mod
            V = (t1 * t2) % mod
            W = (T**2 * V - U2 * (U0 + U1)) % mod
            x3 = (U * W) % mod
            y3 = (T * (U0 * U2 - W) - T0 * U3) % mod
            t3 = (U3 * V) % mod
            invertible = (t3 != 0)
            if invertible:
                return invertible, (x3, y3, t3)
            else:
                return invertible, (point_1, point_2, a, mod)


# y^2 t = x^3 + a x t + t^3
def ECM_prod(point_1, n, a, mod):
    if n == 1:
        return point_1
    else:
```

```
logval = math.log2(n)
group_cons = group_cons_func(logval)

odd, indices, len_val = flexible(n, group_cons)
curr_idx = len_val

invertible, temp_doubled = ECM_sum(point_1, point_1, a, mod)
if not invertible:
    return invertible, (point_1, point_1, a, mod)
else:
    doubled = temp_doubled
odd_sums = [point_1]
for _ in range(3, (1 << group_cons), 2):
    invertible, temp_next_val = ECM_sum(odd_sums[-1], doubled, a, mod)
    if not invertible:
        return invertible, (odd_sums[-1], doubled, a, mod)
    else:
        next_val = temp_next_val
    odd_sums.append(next_val)

if curr_idx == len_val:
    prod = odd_sums[(odd[curr_idx]-1) >> 1]
else:
    invertible, temp_prod = ECM_sum(prod, odd_sums[(odd[curr_idx]-1) >> 1], a, mod)
    if not invertible:
        return invertible, (prod, odd_sums[(odd[curr_idx]-1) >> 1], a, mod)
    else:
        prod = temp_prod

for _ in range(indices[curr_idx]):
    invertible, temp_prod = ECM_sum(prod, prod, a, mod)
    if not invertible:
        return invertible, (prod, prod, a, mod)
    else:
        prod = temp_prod

while curr_idx != 0:
    curr_idx -= 1
    if curr_idx == len_val:
        prod = odd_sums[(odd[curr_idx]-1) >> 1]
    else:
        invertible, temp_prod = ECM_sum(prod, odd_sums[(odd[curr_idx]-1) >> 1], a, mod)
        if not invertible:
            return invertible, (prod, odd_sums[(odd[curr_idx]-1) >> 1], a, mod)
        else:
            prod = temp_prod

    for _ in range(indices[curr_idx]):
        invertible, temp_prod = ECM_sum(prod, prod, a, mod)
        if not invertible:
            return invertible, (prod, prod, a, mod)
        else:
```

```
                    prod = temp_prod

        else:
            return True, prod

def lenstra_ECM_S2(large_num, point, coeff):
    temp_point = point
    y_point = point
    g = 0
    P = 1
    i = 0
    j = 0
    c = 0
    point_diffs = []
    used_diffs = {}
    for k in range(0, len(prime_diffs)):
        if prime_diffs[k] in used_diffs:
            point_diffs.append(used_diffs[prime_diffs[k]])
        else:
            invertible, new = ECM_prod(point, prime_diffs[k], coeff, large_num)
            if not invertible:
                # print(f"Points {new[0], new[1]} not summable for a={new[2]} in mod {large_num}
                point_1 = new[0]
                point_2 = new[1]
                if point_1 != point_2:
                    check_1, inverse_1 = inverse(point_1[2], large_num)
                    check_2, inverse_2 = inverse(point_2[2], large_num)
                    if check_1 & check_2:
                        t = (point_1[0] * point_2[2] - point_2[0] * point_1[2]) % large_num
                        if t != 0:
                            notFound = False
                            return math.gcd(t, large_num)
                        else:
                            return -1
                    else:
                        if not check_1:
                            return inverse_1[2]
                        else:
                            return inverse_2[2]
                else:
                    check_1, inverse_1 = inverse(point_1[2], large_num)
                    if check_1:
                        t = (2 * point_1[1] * inverse_1) % large_num
                        if t != 0:
                            notFound = False
                            print(t)
                            return math.gcd(t, large_num)
                        else:
                            return -1
                    else:
                        return inverse_1[2]
            else:
```

```
            used_diffs[prime_diffs[k]] = new
            point_diffs.append(new)
    invertible, point = ECM_prod(point, very_small_primes[-1], coeff, large_num)
    while i < len(prime_diffs):
        invertible, temp_point_check = ECM_sum(temp_point, point_diffs[i], coeff, large_num)
        if not invertible:
            # print(f"Points {temp_point_check[0], temp_point_check[1]} not summable for a={temp_
            point_1 = temp_point_check[0]
            point_2 = temp_point_check[1]
            if point_1 != point_2:
                check_1, inverse_1 = inverse(point_1[2], large_num)
                check_2, inverse_2 = inverse(point_2[2], large_num)
                if check_1 & check_2:
                    t = (point_1[0] * point_2[2] - point_2[0] * point_1[2]) % large_num
                    if t != 0:
                        notFound = False
                        return math.gcd(t, large_num)
                    else:
                        return -1
                else:
                    if not check_1:
                        return inverse_1[2]
                    else:
                        return inverse_2[2]
            else:
                check_1, inverse_1 = inverse(point_1[2], large_num)
                if check_1:
                    t = (2 * point_1[1] * inverse_1) % large_num
                    if t != 0:
                        notFound = False
                        print(t)
                        return math.gcd(t, large_num)
                    else:
                        return -1
                else:
                    return inverse_1[2]
        else:
            temp_point = temp_point_check
        P *= temp_point[2]
        c += 1
        i += 1
        if c >= 50:
            g = math.gcd(P, large_num)
            if g == 1:
                c = 0
                j = i
                y_point = temp_point
            else:
                i = j
                temp_point = y_point
                while True:
                    invertible, temp_point_check = ECM_sum(temp_point, point_diffs[i], coeff, lar
```

```
                    if not invertible:
                        # print(f"Points {temp_point_check[0], temp_point_check[1]} not summable
                        point_1 = temp_point_check[0]
                        point_2 = temp_point_check[1]
                        if point_1 != point_2:
                            check_1, inverse_1 = inverse(point_1[2], large_num)
                            check_2, inverse_2 = inverse(point_2[2], large_num)
                            if check_1 & check_2:
                                t = (point_1[0] * point_2[2] - point_2[0] * point_1[2]) % large_n
                                if t != 0:
                                    notFound = False
                                    return math.gcd(t, large_num)
                                else:
                                    return -1
                            else:
                                if not check_1:
                                    return inverse_1[2]
                                else:
                                    return inverse_2[2]
                        else:
                            check_1, inverse_1 = inverse(point_1[2], large_num)
                            if check_1:
                                t = (2 * point_1[1] * inverse_1) % large_num
                                if t != 0:
                                    notFound = False
                                    print(t)
                                    return math.gcd(t, large_num)
                                else:
                                    return -1
                            else:
                                return inverse_1[2]
                    else:
                        temp_point = temp_point_check
                    i += 1
                    g = math.gcd(temp_point[2], large_num)
                    if g > 1:
                        if g == large_num:
                            return -1
                        else:
                            return g
    g = math.gcd(P, large_num)
    if g == 1 or g == large_num:
        return -1
    else:
        return g

# Lenstra's Elliptic Curves Method
def lenstra_ECM(large_num, B=12000, c=20, stage=False):
    primes = [i for i in small_primes if i < B]
    num_primes = len(primes)
    coeff = 0
    counter = 0
```

```
notFound = True
while notFound:
    point = (0, 1, 1)
    curr_prime_idx = 0
    if curr_prime_idx >= num_primes:
        print("Stage 2 Required!")
        stage_2 = lenstra_ECM_S2(large_num, point, coeff)
        if stage_2 == -1:
            coeff += 1
            counter = 0
            point = (0, 1, 1)
            curr_prime_idx = 0
        else:
            return f"2.4. Factor of {large_num}: {stage_2}"
    else:
        prime = primes[curr_prime_idx]
        prime_power = prime
        l = B // prime
        while prime_power <= l:
            prime_power *= prime
        invertible, point = ECM_prod(point, prime_power, coeff, large_num)
        while invertible:
            counter += 1
            curr_prime_idx += 1
            if curr_prime_idx >= num_primes:
                if stage:
                    print("Stage 2")
                    stage_2 = lenstra_ECM_S2(large_num, point, coeff)
                    if stage_2 == -1 or stage_2 == -2:
                        coeff += 1
                        counter = 0
                        point = (0, 1, 1)
                        curr_prime_idx = 0
                    else:
                        return f"2.4. Factor of {large_num}: {stage_2}"
                else:
                    coeff += 1
                    counter = 0
                    point = (0, 1, 1)
                    curr_prime_idx = 0
            else:
                prime = primes[curr_prime_idx]
                prime_power = prime
                l = B // prime
                while prime_power <= l:
                    prime_power *= prime
                invertible, point = ECM_prod(point, prime_power, coeff, large_num)
                if invertible and counter % c == 0:
                    d = math.gcd(point[2], large_num)
                    if d != 1:
                        return f"1.1. Factor of {large_num}: {d}"
        else:
```

```
if not invertible:
    # print(f"Points {point[0], point[1]} not summable for a={point[2]} in mod {l
    point_1 = point[0]
    point_2 = point[1]
    if point_1 != point_2:
        check_1, inverse_1 = inverse(point_1[2], large_num)
        check_2, inverse_2 = inverse(point_2[2], large_num)
        if check_1 & check_2:
            t = (point_1[0] * point_2[2] - point_2[0] * point_1[2]) % large_num
            if t != 0:
                notFound = False
                return f"1.2. Factor of {large_num}: {math.gcd(t, large_num)}"
            else:
                coeff += 1
                counter = 0
        else:
            if not check_1:
                return f"1.3. Factor of {large_num}: {inverse_1[2]}"
            else:
                return f"1.4. Factor of {large_num}: {inverse_2[2]}"
    else:
        check_1, inverse_1 = inverse(point_1[2], large_num)
        if check_1:
            t = (2 * point_1[1] * inverse_1) % large_num
            if t != 0:
                notFound = False
                print(t)
                return f"1.5. Factor of {large_num}: {math.gcd(t, large_num)}"
            else:
                coeff += 1
                counter = 0
        else:
            return f"1.6. Factor of {large_num}: {inverse_1[2]}"
```

9.2 RSA Encryption Suite

9.2.1 Decryption

```
primes = []
bprimes = []
def mod(n, a):
    A = n
    terms = [0,1]
    while n != 1 and a !=1:
        if n>a:
            terms.append(n//a)
            n = n%=a
        else:
            terms.append(a//n)
            a = a%=n
    inv = terms[1]
    inV = terms[0]
    k = 2
    while k <len(terms):
        inv, inV = inV - (terms[k]*inv), inv
        k+=1
    return inv%=A


c = 2
t = 116
def primer(t, C):
    new = []
    while C<100*t:
        prime = True
        k = 0
        sq = C**0.5
        while k<len(primes):
            if C%=primes[k] == 0:
                prime = False
            k+=1
            if k>sq:
                break
        if prime:
            primes.append(C)
            new.append(C)
            if C>11358:
                bprimes.append(C)
        C+=1
    return (C, new)

c, ignore = primer(t,c)

print("Welcome.")
print("To begin, pick two distinct primes from this list and enter them separated by a space.
```

```
If you'd like bigger primes enter \"more primes\".")
print(bprimes)
vinp = False
while vinp is False:
    command = input("Input: ")
    if command == "more primes":
        print ("Coming up.")
        t+=2
        c, new = primer(t,c)
        print(new)
    else:
        B = False
        commands = command.split(" ")
        if len(commands) <2:
            print("Too few inputs.")
        elif len(commands) >2:
            print("Too many inputs.")
        elif (not commands[0].isnumeric) or (not commands[1].isnumeric()):
            try:
                p = float(commands[0])
                q = float(commands[1])
                print("Please enter whole numbers.")
            except:
                print("Please enter numbers or ask for more primes.")
        else:
            p = int(commands[0])
            q = int(commands[1])
            if (p not in bprimes) or (q not in bprimes):
                print("Please pick from the list.")
            elif p == q:
                print("Please pick distinct primes.")
            else:
                vinp = True

n = p*q
phi = (p-1)*(q-1)
print("Your \"n\" value is " + str(n) + " and its totient (often called phi)
is " + str(phi) +  ".")
print("You'll now need to choose an \"e\" value. This number needs to be coprime to " +
str(phi) + ", which is to say they have no common factors other than 1.")
es = []
for x in range (2, phi):
    if math.gcd(x,phi) == 1:
        es.append(x)
    if len(es) == 100:
        break
print("Here's a list to choose from.")
print(es)
vinp = False
while not vinp:
    command = input("Input: ")
    if not command.isnumeric():
```

```
            print("Please pick an item from the list.")
        else:
            e = int(command)
            if e not in es:
                print("Please pick an item from the list.")
            else:
                vinp = True


d = mod(phi, e)
print("Your private key \"d\" is " + str(d) +  ", the unique number less than " + str(phi)
+ " such that the remainder when its product with e is divided by " + str(phi) + " is 1.")
print("We've now established a valid RSA encryption system. Share e and n publicly
to allow third parties to encrypt messages addressed to you.")
print("The program will now decrypt encrypted ascii sequences structured as python lists of
integers e.g. \"[1, 2, 3]\". Simply copy and paste one into the console to decrypt it.")
print("Enter \"values\" to see the values of n and e again
or \"quit\" to terminate the program.")


def modde(m, dee, en):
    x = str(bin(dee)).lstrip("0b")
    x = x[::-1]
    b = []
    for i in range(0, len(x)):
        b.append(int(x[i]))
    sq = [m]
    for i in range(1, len(x)):
        sq.append((sq[i-1]**2)%=en)
    cu = 1
    for i in range(0,len(x)):
        if b[i] == 1:
            cu = (cu*sq[i])%=en

    #M = m
    #for i in range(0,dee -1):
    #    M = (M*m)%=en
    return cu



vinp = False
while vinp is False:
    command = input("Input: ")
    if command == "quit":
        exit()
    elif command == "values":
        print("n is " + str(n) + " and e is " + str(e) + ".")
    elif command[0]!= "[" or command[-1]!= "]":
        print("Invalid input")
    else:
        command = command.lstrip("[")
```

```
        command = command.rstrip("]")
        seq = command.split(", ")
        out = ""
        succ = True
        for x in seq:
            if not x.isnumeric() or int(x)>=n:
                print("Invalid list")
                succ = False
                break
            else:
                N = modde(int(x),d,n)
                N = str(N)
                while len(N)!=9:
                    N = "0" + N
                a = int(N[0:3])
                b = int(N[3:6])
                g = int(N[6:9])
                thr = [a,b,g]
                for i in thr:
                    if i>128:
                        print("Invalid sequence")
                        succ = False
                        break
                    else:
                        out += chr(i)
        if succ:
            print(out)
```

9.2.2 Encryption

```
def encrypt(s, e, n):
    out = []
    while len(s)%=3!=0:
            s+="#"
    for i in range(0,int(len(s)/3)):
        st = s[3*i:3*i+3]
        a,b,c = str(ord(st[0])),str(ord(st[1])),str(ord(st[2]))
        while len(b)!=3:
            b = "0" + b
        while len(c)!=3:
            c = "0" + c
        k = int(a + b + c)
        out.append(modde(k,e,n))
    return(out)

def modde(m, dee, en):
    x = str(bin(dee)).lstrip("0b")
    x = x[::-1]
    b = []
    for i in range(0, len(x)):
        b.append(int(x[i]))
```

```
    sq = [m]
    for i in range(1, len(x)):
        sq.append((sq[i-1]**2)%=en)
    cu = 1
    for i in range(0,len(x)):
        if b[i] == 1:
            cu = (cu*sq[i])%=en

    #M = m
    #for i in range(0,dee -1):
    #    M = (M*m)%=en
    return cu

a = True

while a:
    kom = input("Enter a message here (or enter \"quit\" to quit): ")
    N = input("Enter an n value: ")
    while not N.isnumeric():
        print("Please enter an integer value")
        N = input("Enter an n value: ")
    N = int(N)
    E = input("Enter an e value: ")
    while not E.isnumeric():
        print("Please enter an integer value")
        E = input("Enter an e value: ")
    E = int(E)
    print(encrypt(kom, E, N))
```