Number Theory

Atli Fannar Franklín, Arnar Bjarni Arnarson

Árangursrík forritun og lausn verkefna

School of Computer Science

Reykjavík University

Primes and factorization

Definitions that everybody should know

- Prime number is a positive integer greater than 1 that has no positive divisor other than 1 and itself.
- Greatest Common Divisor of two integers a and b is the largest number that divides both a and b.
- Least Common Multiple of two integers a and b is the smallest positive integer that both a and b divide.

Primality checking

- How do we determine if a number n is a prime?

Primality checking

- How do we determine if a number n is a prime?
- Naive method: Iterate over all $1 < i < n$ and check if i divides n.
  - $O(N)$

Primality checking

- How do we determine if a number n is a prime?
- Naive method: Iterate over all $1 < i < n$ and check if i divides n.
  - $O(N)$
- Better: If n is not a prime, it has a divisor $\leq \sqrt{n}$.
  - Iterate up to $\sqrt{n}$ instead.
  - $O(\sqrt{N})$

$\mathcal{O}(\sqrt{n})$ check

```cpp
template <typename T>
bool is_prime(T x) {
    if(x <= 1) return false;
    for(T i = T(2); i * i <= x; ++i)
        if(x % i == 0)
            return false;
    return true;
}
```

Modular arithmetic

Modulo

- Problem statements often end with the sentence

  "... and output the answer modulo n."

Modulo

- Problem statements often end with the sentence
  "... and output the answer modulo n."

- This implies that we can do all the computation with integers *modulo* n.

Modulo

- Problem statements often end with the sentence
  "... and output the answer modulo n."

- This implies that we can do all the computation with integers *modulo* n.
- But what does this mean? Taking an integer modulo n means taking the remainder of it when we divide by n.

Modulo

- Problem statements often end with the sentence
  "... and output the answer modulo n."

- This implies that we can do all the computation with integers *modulo* n.
- But what does this mean? Taking an integer modulo n means taking the remainder of it when we divide by n.
- Thus if we do everything modulo n we consider every number a multiple of n apart the same. So modulo 7 the numbers $1, 8, -6, 15, -13, \ldots$ are all the same, and so are $5, -2, 12, -9, 19, \ldots$.

Modulo

- In this system it's simplest if we pick one of the infinite set of equivalent numbers to be the one we use to represent them. We usually choose the representatives $0, 1, \ldots, n-1$ if we're working modulo n.

Modulo

- In this system it's simplest if we pick one of the infinite set of equivalent numbers to be the one we use to represent them. We usually choose the representatives $0, 1, \ldots, n-1$ if we're working modulo n.
- Then to do addition, subtraction and multiplication we just do it as usual, but add or subtract multiples of n afterwards so we end up back in $\{0, 1, \ldots, n-1\}$.

Modulo

- In this system it's simplest if we pick one of the infinite set of equivalent numbers to be the one we use to represent them. We usually choose the representatives $0, 1, \ldots, n-1$ if we're working modulo n.
- Then to do addition, subtraction and multiplication we just do it as usual, but add or subtract multiples of n afterwards so we end up back in $\{0, 1, \ldots, n-1\}$.
- This means this set, which we denote $\mathbb{Z}_n$, is a ring, for those familiar with that terminology.

Modulo

- Problem statements often end with the sentence
  "... and output the answer modulo n."

Modulo

- Problem statements often end with the sentence

  "... and output the answer modulo n."

- This implies that we can do all the computation in $\mathbb{Z}_n$.

Modulo

- Problem statements often end with the sentence
  "... and output the answer modulo n."

- This implies that we can do all the computation in $\mathbb{Z}_n$.
- This is often very useful, since the numbers never get too big and we don't generally have to worry about over/underflow.

Division

- What about division? Is it possible to divide?

Division

- What about division? Is it possible to divide? Not always!

Division

- What about division? Is it possible to divide? Not always!
- We could end up with a fraction!
- Division with k equals multiplication with the *multiplicative inverse* of k.

Division

- What about division? Is it possible to divide? Not always!
- We could end up with a fraction!
- Division with k equals multiplication with the *multiplicative inverse* of k.
- The *multiplicative inverse* of an integer a, is the element a^{-1} such that

$$a \cdot a^{-1} = 1 \pmod{n}$$

Division

- What about division? Is it possible to divide? Not always!
- We could end up with a fraction!
- Division with k equals multiplication with the *multiplicative inverse* of k.
- The *multiplicative inverse* of an integer a, is the element a^{-1} such that

$$a \cdot a^{-1} = 1 \pmod{n}$$

- Such an a^{-1} does not always exist, let's see how we can find it when it does exist though.

Euclidean algorithm

- The Euclidean algorithm is a recursive algorithm that computes the GCD of two numbers.

```
template<typename T>
T gcd(T a, T b){
    return b == T(0) ? a : gcd(b, a % b);
}
```

- Runs in $O(\log N)$.

Euclidean algorithm

- The Euclidean algorithm is a recursive algorithm that computes the GCD of two numbers.

```
template<typename T>
T gcd(T a, T b){
    return b == T(0) ? a : gcd(b, a % b);
}
```

- Runs in $O(\log N)$.
- Notice that this can also compute LCM

$$\text{lcm}(a,b) = \frac{a \cdot b}{\gcd(a,b)}$$

- See Wikipedia to see how it works and for proofs.

Extended Euclidean algorithm

- Reversing the steps of the Euclidean algorithm we get the Bézout's identity

$$\gcd(a, b) = ax + by$$

  which simply states that there always exist x and y such that the equation above holds.

Extended Euclidean algorithm

- Reversing the steps of the Euclidean algorithm we get the Bézout's identity

$$\gcd(a, b) = ax + by$$

  which simply states that there always exist x and y such that the equation above holds.

- The extended Euclidean algorithm computes the GCD and the coefficients x and y.
- Each iteration it add up how much of b we subtracted from a and vice versa.

Extended Euclidean algorithm

```
template <typename T>
T egcd(T a, T b, T& x, T& y) {
    if (b == 0) {
        x = T(1);
        y = T(0);
        return a;
    } else {
        T d = egcd(b, a % b, x, y);
        x -= a / b * y;
        swap(x, y);
        return d;
    }
}
```

Applications

- Essential step in the RSA algorithm.
- Essential step in many factorization algorithms.
- Can be generalized to other algebraic structures.
- Fundamental tool for proofs in number theory.
- Many other algorithms for GCD

Modular inverse

Back to modular inverse.

- Working modulo n often requires division (multiplication by inverse).

Modular inverse

Back to modular inverse.

- Working modulo n often requires division (multiplication by inverse).
- Given some $a \pmod{n}$, then the multiplicative inverse $a^{-1} \pmod{n}$ exists iff. a and n are coprime.

Modular inverse

Back to modular inverse.

- Working modulo n often requires division (multiplication by inverse).
- Given some $a \pmod{n}$, then the multiplicative inverse $a^{-1} \pmod{n}$ exists iff. a and n are coprime.
- It so happens that when we have from EGCD algorithm

$$ax + ny = \gcd(a, n) = 1$$

then

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

Modular inverse

```
template<typename T>
T mod_inv(T a, T m) {
    T x, y, d = egcd(a, m, x, y);
    return d == T(1) ? (x%m+m)%m : T(-1);
}
```

Discrete logarithm

- What about logarithm?

Discrete logarithm

- What about logarithm? YES!
    - But difficult.

Discrete logarithm

- What about logarithm? YES!
    - But difficult.
    - Basis for some cryptography such as elliptic curve, Diffie-Hellmann.
- Google "Discrete Logarithm" if you want to know more.

Chinese remainder theorem

What is the lowest number n such that when divided by

... 3 it leaves 2 in remainder.

... 5 it leaves 3 in remainder.

... 7 it leaves 2 in remainder.

Chinese remainder theorem

What is the lowest number n such that when divided by

... 3 it leaves 2 in remainder.

... 5 it leaves 3 in remainder.

... 7 it leaves 2 in remainder.

When stated mathematically, find n where

$$\begin{aligned} n &\equiv 2 \pmod 3 \\ n &\equiv 3 \pmod 5 \\ n &\equiv 2 \pmod 7 \end{aligned}$$

Chinese remainder theorem

The Chinese remainder theorem states that:

- When the moduli of a system of linear congruences are pairwise coprime, there exists a unique solution modulo the product of the moduli.

Chinese remainder theorem

The Chinese remainder theorem states that:

- When the moduli of a system of linear congruences are pairwise coprime, there exists a unique solution modulo the product of the moduli.

Let $n_1, n_2, \ldots, n_k$ be pairwise coprime positive integers, and let x be the solution to the system of linear congruences

$$\begin{aligned} x &\equiv b_1 \pmod{n_1} \\ x &\equiv b_2 \pmod{n_2} \\ &\vdots \\ x &\equiv b_k \pmod{n_k} \end{aligned}$$

Chinese remainder theorem

- The Chinese remainder theorem only states that there exists a solution and it is unique modulus the product of the moduli.
- To obtain the solution x

$$x \equiv b_1 c_1 \frac{N}{n_1} + \ldots + b_k c_k \frac{N}{n_k}$$

  where $N = n_1 n_2 \cdots n_k$.

- The coefficients c_i are determined from

$$c_i \frac{N}{n_i} \equiv 1 \pmod{n_i}$$

  (the multiplicative inverse of $\frac{N}{n_i}$ modulus n_i)

- Use EGCD to compute c_i.

Faster primality check?

- Can we determine primality faster?

Faster primality check?

- Can we determine primality faster?
- Sort of.

Faster primality check?

- Can we determine primality faster?
- Sort of.
- We can use probabilistic prime testing, a function that either says the input is *probably* prime or *definitely* not.

Faster primality check?

- Can we determine primality faster?
- Sort of.
- We can use probabilistic prime testing, a function that either says the input is *probably* prime or *definitely* not.
- This may sound shaky, but this program can be run a dozen times.

Faster primality check?

- Can we determine primality faster?
- Sort of.
- We can use probabilistic prime testing, a function that either says the input is *probably* prime or *definitely* not.
- This may sound shaky, but this program can be run a dozen times.
- The probability of the program being wrong every time is so vanishingly small.

Faster primality check?

- Can we determine primality faster?
- Sort of.
- We can use probabilistic prime testing, a function that either says the input is *probably* prime or *definitely* not.
- This may sound shaky, but this program can be run a dozen times.
- The probability of the program being wrong every time is so vanishingly small.
- You would spend your time better worrying about space rays flipping your bits while you run the program.

Miller-Rabin concept

- Let us first note that if $x^2 = 1 \ (mod \ p)$ this can be factored as $(x-1)(x+1) = 0 \ (mod \ p)$ and since p is prime this means $x = \pm 1 \ (mod \ p)$.

Miller-Rabin concept

- Let us first note that if $x^2 = 1\ (mod\ p)$ this can be factored as $(x-1)(x+1) = 0\ (mod\ p)$ and since p is prime this means $x = \pm 1\ (mod\ p)$.
- Now take some $p > 2$ and $a < p$. Write $p - 1 = 2^s d$ s.t. d is odd. Then by taking the square root on each side of the equation $a^{p-1} = 1\ (mod\ p)$ (which we know is true) then either the right side will at some point equal -1 and we have to stop, or we eventually divide out all powers of two in a. This either $a^d = 1\ (mod\ p)$ or $a^{2^r d} = -1\ (mod\ p)$ for some $0 \leq r \leq s-1$.

Miller-Rabin concept

- Let us first note that if $x^2 = 1\ (mod\ p)$ this can be factored as $(x-1)(x+1) = 0\ (mod\ p)$ and since p is prime this means $x = \pm 1\ (mod\ p)$.
- Now take some $p > 2$ and $a < p$. Write $p - 1 = 2^s d$ s.t. d is odd. Then by taking the square root on each side of the equation $a^{p-1} = 1\ (mod\ p)$ (which we know is true) then either the right side will at some point equal -1 and we have to stop, or we eventually divide out all powers of two in a. This either $a^d = 1\ (mod\ p)$ or $a^{2^r d} = -1\ (mod\ p)$ for some $0 \leq r \leq s-1$.
- Thus to prove that n is not prime we try to find $a < n$ s.t. $a^d \neq 1\ (mod\ n)$ and $a^{2^r d} \neq -1\ (mod\ n)$ for all $0 \leq r \leq s-1$.

Miller-Rabin concept ctd.

- Finding such an a sounds far fetched, but it turns out that a large percentage of numbers will work as the choice of a if n is not prime.

Miller-Rabin concept ctd.

- Finding such an a sounds far fetched, but it turns out that a large percentage of numbers will work as the choice of a if n is not prime.
- Thus the Miller-Rabin algorithm works by choosing random a and seeing if it excludes the possibility of n being prime.

Miller-Rabin concept ctd.

- Finding such an a sounds far fetched, but it turns out that a large percentage of numbers will work as the choice of a if n is not prime.
- Thus the Miller-Rabin algorithm works by choosing random a and seeing if it excludes the possibility of n being prime.
- Thus the algorithm is such that if it says p is not prime, this is definitely true. If it says p is a prime, it really means "I couldn't exclude the possibility that p is prime, but it could be non-prime".

Miller-Rabin concept ctd.

- Finding such an a sounds far fetched, but it turns out that a large percentage of numbers will work as the choice of a if n is not prime.
- Thus the Miller-Rabin algorithm works by choosing random a and seeing if it excludes the possibility of n being prime.
- Thus the algorithm is such that if it says p is not prime, this is definitely true. If it says p is a prime, it really means "I couldn't exclude the possibility that p is prime, but it could be non-prime".
- If we test many a the odds are in our favor. Thus we let the program take a variable `k` saying how often it should run. This runs in $\mathcal{O}(k \log(n)^3)$ for large n.

Miller-Rabin implementation

```
template <typename T>
bool is_probably_prime(T n, int k) {
    if (n % 2 == 0) return n == T(2);
    if (n <= 3) return n == T(3);
    T d = n - 1, r = T(0);
    while (d % 2 == 0) d >>= 1, r++;
    for (int i = 0; i < k; ++i) {
        T a = (n - 3) * rand() / RAND_MAX + 2;
        T x = modpow(a, d, n);
        if(x == T(1) || x == T(n - 1)) continue;
        bool ok = false;
        for(T j = 0; j < r - 1; ++j) {
            x = (x * x % n + n) % n;
            if(x == T(1)) return false;
            if(x == T(n - 1)) { ok = true; break; }
        }
        if(!ok) return false;
    }
    return true;
}
```

Bulk discount

- If we want to generate primes, using a primality test is very inefficient.

Bulk discount

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.

Bulk discount

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
    - For all numbers from 2 to $\sqrt{n}$:

Bulk discount

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
    - For all numbers from 2 to $\sqrt{n}$:
    - If the number is not marked, iterate over every multiple of the number up to n and mark them.

Bulk discount

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
  - For all numbers from 2 to $\sqrt{n}$:
  - If the number is not marked, iterate over every multiple of the number up to n and mark them.
  - The unmarked numbers are those that are not a multiple of any smaller number.
  - $O(N \log \log N)$

Eratosthenes example

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Eratosthenes example

		2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Eratosthenes example

		2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Eratosthenes example

		2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Eratosthenes example

	2	3	5	7	9
11		13	15	17	19
21		23	25	27	29
31		33	35	37	39
41		43	45	47	49
51		53	55	57	59
61		63	65	67	69
71		73	75	77	79
81		83	85	87	89
91		93	95	97	99

Eratosthenes example

	2	3	5	7	9
11		13	15	17	19
21		23	25	27	29
31		33	35	37	39
41		43	45	47	49
51		53	55	57	59
61		63	65	67	69
71		73	75	77	79
81		83	85	87	89
91		93	95	97	99

Eratosthenes example

2	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

Eratosthenes example

	2	3		5		7		
11		13				17		19
		23		25				29
31				35		37		
41		43				47		49
		53		55				59
61				65		67		
71		73				77		79
		83		85				89
91				95		97		

Eratosthenes example

	2	3		5		7		
11		13				17		19
		23		25				29
31				35		37		
41		43				47		49
		53		55				59
61				65		67		
71		73				77		79
		83		85				89
91				95		97		

Eratosthenes example

	2	3		5		7			
11		13				17		19	
		23		25				29	
31				35		37			
41		43				47		49	
		53		55				59	
61				65		67			
71		73				77		79	
		83		85				89	
91				95		97			

Eratosthenes example

	2	3		5		7			
11		13				17		19	
		23						29	
31						37			
41		43				47		49	
		53						59	
61						67			
71		73				77		79	
		83						89	
91						97			

Eratosthenes example

	2	3		5		7			
11		13				17		19	
		23						29	
31						37			
41		43				47		49	
		53						59	
61						67			
71		73				77		79	
		83						89	
91						97			

Eratosthenes example

	2	3		5		7			
11		13				17		19	
		23						29	
31						37			
41		43				47		49	
		53						59	
61						67			
71		73				77		79	
		83						89	
91						97			

Eratosthenes example

	2	3		5		7			
11		13				17		19	
		23						29	
31						37			
41		43				47			
		53						59	
61						67			
71		73						79	
		83						89	
						97			

	2	3		5		7			
11		13				17		19	
		23						29	
31						37			
41		43				47			
		53						59	
61						67			
71		73						79	
		83						89	
						97			

Sieve of Eratosthenes

```
template <typename T>
vector<T> eratosthenes(T n){
    vector<bool> isMarked(n+1, false);
    vector<T> primes;
    T i = T(2);
    for(; i*i <= n; i++)
        if (!isMarked[i]) {
            primes.push_back(i);
            for(T j = i; j <= n; j += i)
                isMarked[j] = true;
        }
    for (; i <= n; i++)
        if (!isMarked[i])
            primes.push_back(i);
    return primes;
}
```

Factoring

The fundamental theorem of arithmetic states that

- Every integer greater than 1 is a unique product of primes, up to order.

Factoring

The fundamental theorem of arithmetic states that

- Every integer greater than 1 is a unique product of primes, up to order.

$$n = p_1^{e_1} p_2^{e_2} p_3^{e_3} \cdots p_k^{e_k}$$

Factoring

The fundamental theorem of arithmetic states that

- Every integer greater than 1 is a unique product of primes, up to order.

$$n = p_1^{e_1} p_2^{e_2} p_3^{e_3} \cdots p_k^{e_k}$$

We can therefore store integers as lists of their prime powers.

Factoring

The fundamental theorem of arithmetic states that

- Every integer greater than 1 is a unique product of primes, up to order.

$$n = p_1^{e_1} p_2^{e_2} p_3^{e_3} \cdots p_k^{e_k}$$

We can therefore store integers as lists of their prime powers.
To factor an integer n:

- Use the sieve of Eratosthenes to generate all the primes up $\sqrt{n}$
- Iterate over all the primes generated and check if they divide n, and determine the largest power that divides n.

Factoring code

```
template <typename T>
map<T, T> factor(T N) {
    vector<T> primes;
    primes = eratosthenes(static_cast<T>(sqrt(N+1)));
    map<T, T> factors;
    for(const auto prime : primes)
        T power = 0;
        while(N % prime == T(0)){
            power++;
            N /= prime;
        }
        if(power > T(0)){
            factors[prime] = power;
        }
    }
    if (N > T(1)) {
        factors[N] = T(1);
    }
    return factors;
}
```

Faster?

- This is a very good way of factoring numbers, but can we do it faster?

Faster?

- This is a very good way of factoring numbers, but can we do it faster?
- Again the answer is sort of.

Faster?

- This is a very good way of factoring numbers, but can we do it faster?
- Again the answer is sort of.
- We can use the birthday paradox to our advantage.

Faster?

- This is a very good way of factoring numbers, but can we do it faster?
- Again the answer is sort of.
- We can use the birthday paradox to our advantage.
- If we have n items we are expected to receive a duplicate once we have picked $\mathcal{O}(\sqrt{n})$ from the collection at random.

Faster?

- This is a very good way of factoring numbers, but can we do it faster?
- Again the answer is sort of.
- We can use the birthday paradox to our advantage.
- If we have n items we are expected to receive a duplicate once we have picked $\mathcal{O}(\sqrt{n})$ from the collection at random.
- We will use this to factor n. But first a small side step.

Floyd cycle finding

- If we have a function f, how do we find whether $f^{[n+m]}(x) = f^{[m]}(x)$ for some n, m?

Floyd cycle finding

- If we have a function f, how do we find whether $f^{[n+m]}(x) = f^{[m]}(x)$ for some n, m?
- This is often a useful thing to be able to do quickly, and to this end we use Floyd's cycle finding algorithm. It is also known as the tortoise-hare algorithm.

Floyd cycle finding

- If we have a function f, how do we find whether $f^{[n+m]}(x) = f^{[m]}(x)$ for some n, m?
- This is often a useful thing to be able to do quickly, and to this end we use Floyd's cycle finding algorithm. It is also known as the tortoise-hare algorithm.
- The trick is that i is a multiple of the cycle length of f iff $f^{[i]}(x) = f^{[2i]}(x)$.

Floyd cycle finding

- If we have a function f, how do we find whether $f^{[n+m]}(x) = f^{[m]}(x)$ for some n, m?
- This is often a useful thing to be able to do quickly, and to this end we use Floyd's cycle finding algorithm. It is also known as the tortoise-hare algorithm.
- The trick is that i is a multiple of the cycle length of f iff $f^{[i]}(x) = f^{[2i]}(x)$.
- Thus we only have to consider that equation when trying to find the cycle length. When that is done we can go back to find where the cycle began and check its size.

Floyd implementation

```
#include <bits/stdc++.h>
using namespace std;

template <typename T, typename F>
pair<int, int> floyd(F&& f, T x0) {
    T t = f(x0), h = f(f(x0));
    while(t != h) {
        t = f(t);
        h = f(f(h));
    }
    int length = 0;
    t = x0;
    while(t != h) {
        t = f(t);
        h = f(h);
        length++;
    }
    int start = 1;
    T h = f(t);
    while(t != h) {
        h = f(h);
        start++;
    }
    return {start, length}
}
```

Pollard rho factorization

- Let $g(x) = x^2 + 1 \ (mod\ n)$ and create the sequence $x_1 = x, x_2 = g(x), x_3 = g(g(x)), \ldots$ where x is chosen randomly.

Pollard rho factorization

- Let $g(x) = x^2 + 1 \ (mod\ n)$ and create the sequence $x_1 = x, x_2 = g(x), x_3 = g(g(x)), \ldots$ where x is chosen randomly.
- Check if $\gcd(x_i - x_j, n) > 1$.

Pollard rho factorization

- Let $g(x) = x^2 + 1 \ (mod\ n)$ and create the sequence $x_1 = x, x_2 = g(x), x_3 = g(g(x)), \ldots$ where x is chosen randomly.
- Check if $\gcd(x_i - x_j, n) > 1$.
- The sequence has begun repeating not just modulo n but modulo d where d divides n.

Pollard rho factorization

- Let $g(x) = x^2 + 1 \ (mod\ n)$ and create the sequence $x_1 = x, x_2 = g(x), x_3 = g(g(x)), \ldots$ where x is chosen randomly.
- Check if $\gcd(x_i - x_j, n) > 1$.
- The sequence has begun repeating not just modulo n but modulo d where d divides n.
- If $\gcd(x_i - x_j, n) = 1$ for all values then either n is not prime or we just didn't manage to find a divisor.

Pollard rho factorization

- Let $g(x) = x^2 + 1\ (mod\ n)$ and create the sequence $x_1 = x, x_2 = g(x), x_3 = g(g(x)), \ldots$ where x is chosen randomly.
- Check if $\gcd(x_i - x_j, n) > 1$.
- The sequence has begun repeating not just modulo n but modulo d where d divides n.
- If $\gcd(x_i - x_j, n) = 1$ for all values then either n is not prime or we just didn't manage to find a divisor.
- Test a few starting values of x before giving up. Usually good for $n > 2^{32}$.

Pollard rho factorization

- Let $g(x) = x^2 + 1\ (mod\ n)$ and create the sequence $x_1 = x, x_2 = g(x), x_3 = g(g(x)), \ldots$ where x is chosen randomly.
- Check if $\gcd(x_i - x_j, n) > 1$.
- The sequence has begun repeating not just modulo n but modulo d where d divides n.
- If $\gcd(x_i - x_j, n) = 1$ for all values then either n is not prime or we just didn't manage to find a divisor.
- Test a few starting values of x before giving up. Usually good for $n > 2^{32}$.
- The time complexity is an open question, but it's conjectured to be the square root of the largest factor of N.

Pollard rho factorization

- Let $g(x) = x^2 + 1\ (mod\ n)$ and create the sequence $x_1 = x, x_2 = g(x), x_3 = g(g(x)), \ldots$ where x is chosen randomly.
- Check if $\gcd(x_i - x_j, n) > 1$.
- The sequence has begun repeating not just modulo n but modulo d where d divides n.
- If $\gcd(x_i - x_j, n) = 1$ for all values then either n is not prime or we just didn't manage to find a divisor.
- Test a few starting values of x before giving up. Usually good for $n > 2^{32}$.
- The time complexity is an open question, but it's conjectured to be the square root of the largest factor of N.
- Slow for primes, but much faster for composite numbers.

Pollard rho factorization

- Let $g(x) = x^2 + 1 \ (mod\ n)$ and create the sequence $x_1 = x, x_2 = g(x), x_3 = g(g(x)), \ldots$ where x is chosen randomly.
- Check if $\gcd(x_i - x_j, n) > 1$.
- The sequence has begun repeating not just modulo n but modulo d where d divides n.
- If $\gcd(x_i - x_j, n) = 1$ for all values then either n is not prime or we just didn't manage to find a divisor.
- Test a few starting values of x before giving up. Usually good for $n > 2^{32}$.
- The time complexity is an open question, but it's conjectured to be the square root of the largest factor of N.
- Slow for primes, but much faster for composite numbers.
- Checking for primality first using Miller-Rabin can be useful.

Pollard rho implementation

```
template <typename T>
T rho(T n) {
    vector<T> seed = {
        T(2), T(3), T(4), T(5), T(7), T(11), T(13), T(1031)
    };
    for(auto s : seed) {
        T x = s, y = x, d = T(1);
        while(d == T(1)) {
            x = ((x * x + 1) % n + n) % n;
            y = ((y * y + 1) % n + n) % n;
            y = ((y * y + 1) % n + n) % n;
            d = gcd(abs(x - y), n);
        }
        if(d == n) continue;
        return d;
    }
    return -1;
}
```

Number theory functions

The prime factors can be quite useful.

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$$

Number theory functions

The prime factors can be quite useful.

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$$

- The number of positive divisors

$$\sigma_0(n) = \prod_{i=1}^{k} (e_i + 1)$$

Number theory functions

The prime factors can be quite useful.

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$$

- The number of positive divisors

$$\sigma_0(n) = \prod_{i=1}^{k} (e_i + 1)$$

- The sum of all positive divisors in x-th power

$$\sigma_x(n) = \prod_{i=1}^{k} \frac{(p_i^{(e_i+1)x} - 1)}{(p_i - 1)}$$

More number theory functions

- The Euler's totient function

$$\phi(n) = n \cdot \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right)$$

counts the numbers $1 \leq x < n$ such that $\gcd(x, n) = 1$

More number theory functions

- The Euler's totient function

$$\phi(n) = n \cdot \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right)$$

  counts the numbers $1 \leq x < n$ such that $\gcd(x, n) = 1$

- Euler's theorem, if a and n are coprime

$$a^{\phi(n)} = 1 \pmod{n}$$

  Fermat's theorem is a special case when n is a prime.