Kattis

Number Theory

Atli FF

Árangursrík forritun og lausn verkefna

School of Computer Science

Reykjavík University

Mathematical introduction

Important point

$$\text{Computer Science} \subset \text{Mathematics}$$

- Problems often require mathematical analysis to be solved efficiently.
- Using a bit of math before coding can also shorten and simplify code.

Pattern finding

- Some problems have solutions that form a pattern.

Pattern finding

- Some problems have solutions that form a pattern.
- By finding the pattern, we solve the problem.
- Could be classified as mathematical ad-hoc problem.
- Requires mathematical intuition.

Pattern finding

- Some problems have solutions that form a pattern.
- By finding the pattern, we solve the problem.
- Could be classified as mathematical ad-hoc problem.
- Requires mathematical intuition.
- Useful tricks:
    - Solve some small instances by hand.
    - See if the solutions form a pattern.

Pattern finding

- Some problems have solutions that form a pattern.
- By finding the pattern, we solve the problem.
- Could be classified as mathematical ad-hoc problem.
- Requires mathematical intuition.
- Useful tricks:
    - Solve some small instances by hand.
    - See if the solutions form a pattern.
- Does the pattern involve some overlapping subproblem?

Pattern finding

- Some problems have solutions that form a pattern.
- By finding the pattern, we solve the problem.
- Could be classified as mathematical ad-hoc problem.
- Requires mathematical intuition.
- Useful tricks:
  - Solve some small instances by hand.
  - See if the solutions form a pattern.
- Does the pattern involve some overlapping subproblem? We might need to use DP.

Pattern finding

- Some problems have solutions that form a pattern.
- By finding the pattern, we solve the problem.
- Could be classified as mathematical ad-hoc problem.
- Requires mathematical intuition.
- Useful tricks:
  - Solve some small instances by hand.
  - See if the solutions form a pattern.
- Does the pattern involve some overlapping subproblem? We might need to use DP.
- Knowing reoccurring identities and sequences can be helpful.

Arithmetic progression

- Often we see a pattern like

$$2, 5, 8, 11, 14, 17, 20, \ldots$$

Arithmetic progression

- Often we see a pattern like

$$2, 5, 8, 11, 14, 17, 20, \ldots$$

- This is called an arithmetic progression.

$$a_n = a_{n-1} + c$$

Arithmetic progression

- Depending on the situation we may want to get the n-th element

$$a_n = a_1 + (n-1)c$$

- Or the sum over a finite portion of the progression

$$S_n = \frac{n(a_1 + a_n)}{2}$$

Arithmetic progression

- Depending on the situation we may want to get the n-th element

$$a_n = a_1 + (n-1)c$$

- Or the sum over a finite portion of the progression

$$S_n = \frac{n(a_1 + a_n)}{2}$$

- Remember this one?

$$1 + 2 + 3 + 4 + 5 + \ldots + n = \frac{n(n+1)}{2}$$

Geometric progression

- Other types of pattern we often see are geometric progressions

$$1,\ 2,\ 4,\ 8,\ 16,\ 32,\ 64,\ 128,\ \ldots$$

Geometric progression

- Other types of pattern we often see are geometric progressions

$$1,\ 2,\ 4,\ 8,\ 16,\ 32,\ 64,\ 128,\ \ldots$$

- More generally

$$s,\ sr,\ sr^2,\ sr^3,\ sr^4,\ sr^5,\ sr^6,\ \ldots$$

$$a_n = sr^{n-1}$$

Geometric progression

- Sum over a finite portion

$$\sum_{i=0}^{n} sr^i = \frac{s(1-r^n)}{(1-r)}$$

Geometric progression

- Sum over a finite portion

$$\sum_{i=0}^{n} sr^i = \frac{s(1-r^n)}{(1-r)}$$

- Or from the m-th element to the n-th

$$\sum_{i=m}^{n} sr^i = \frac{s(r^m - r^{n+1})}{(1-r)}$$

Quick note on logarithms

- Sometimes doing computation in logarithm can be an efficient alternative.

Quick note on logarithms

- Sometimes doing computation in logarithm can be an efficient alternative.
- In both C++(<cmath>) and Java(`java.lang.Math`) we have the natural logarithm

```
double log(double x);
```

Quick note on logarithms

- Sometimes doing computation in logarithm can be an efficient alternative.
- In both C++(<cmath>) and Java(`java.lang.Math`) we have the natural logarithm

```
double log(double x);
```

and logarithm in base 10

```
double log10(double x);
```

Quick note on logarithms

- Sometimes doing computation in logarithm can be an efficient alternative.
- In both C++(<cmath>) and Java(`java.lang.Math`) we have the natural logarithm

  ```
  double log(double x);
  ```

  and logarithm in base 10

  ```
  double log10(double x);
  ```

- And also the exponential

  ```
  double exp(double x);
  ```

Example

- For example, what is the first power of 17 that has k digits in base b?

Example

- For example, what is the first power of 17 that has k digits in base b?
- Naive solution: Iterate over powers of 17 and count the number of digits.

Example

- For example, what is the first power of 17 that has k digits in base b?
- Naive solution: Iterate over powers of 17 and count the number of digits.
- But the powers of 17 grow exponentially!

$$17^{16} > 2^{64}$$

- What if $k = 500$ ($\sim 1.7 \cdot 10^{615}$), or something larger?

Example

- For example, what is the first power of 17 that has k digits in base b?
- Naive solution: Iterate over powers of 17 and count the number of digits.
- But the powers of 17 grow exponentially!

$$17^{16} > 2^{64}$$

- What if $k = 500$ ($\sim 1.7 \cdot 10^{615}$), or something larger?
- Impossible to work with the numbers in a normal fashion.
- Why not $\log$?

Example

- Remember, we can calculate the length of a number n in base b with $\lfloor \log_b(n) \rfloor + 1$.

Example

- Remember, we can calculate the length of a number n in base b with $\lfloor \log_b(n) \rfloor + 1$.
- But how do we do this with only $\ln$ or $\log_{10}$?

Example

- Remember, we can calculate the length of a number n in base b with $\lfloor \log_b(n) \rfloor + 1$.
- But how do we do this with only $\ln$ or $\log_{10}$?
- Change base!

$$\log_b(a) = \frac{\log_d(a)}{\log_d(b)} = \frac{\ln(a)}{\ln(b)}$$

- Now we can at least count the length without converting bases

Example

- We still have to iterate over the powers of 17, but we can do that in log

$$\ln(17^{x-1} \cdot 17) = \ln(17^{x-1}) + \ln(17)$$

Example

- We still have to iterate over the powers of 17, but we can do that in log

$$\ln(17^{x-1} \cdot 17) = \ln(17^{x-1}) + \ln(17)$$

- More generally

$$\log_b(xy) = \log_b(x) + \log_b(y)$$

- For division

$$\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y)$$

Example

- We can simplify this even more.
- The solution to our problem is in mathematical terms, finding the x for

$$\log_b(17^x) = k - 1$$

Example

- We can simplify this even more.
- The solution to our problem is in mathematical terms, finding the x for

$$\log_b(17^x) = k - 1$$

- One more handy identity

$$\log_b(a^c) = c \cdot \log_b(a)$$

Example

- We can simplify this even more.
- The solution to our problem is in mathematical terms, finding the x for

$$\log_b(17^x) = k - 1$$

- One more handy identity

$$\log_b(a^c) = c \cdot \log_b(a)$$

- Using this identity and the ones we've covered, we get

$$x = \left\lceil (k-1) \cdot \frac{\ln(10)}{\ln(17)} \right\rceil$$

Base conversion

- Speaking of bases.

Base conversion

- Speaking of bases.
- What if we actually need to use base conversion?

Base conversion

- Speaking of bases.
- What if we actually need to use base conversion?
- Simple algorithm

```
vector<int> toBase(int base, int val) {
    vector<int> res;
    while(val) {
        res.push_back(val % base);
        val /= base;
    }
    return val;
}
```

- Starts from the 0-th digit, and calculates the multiple of each power.

Working with doubles

- Comparing doubles, sounds like a bad idea.

Working with doubles

- Comparing doubles, sounds like a bad idea.
- What else can we do if we are working with real numbers?

Working with doubles

- Comparing doubles, sounds like a bad idea.
- What else can we do if we are working with real numbers?
- We compare them to a certain degree of precision like in binary search.

Working with doubles

- Comparing doubles, sounds like a bad idea.
- What else can we do if we are working with real numbers?
- We compare them to a certain degree of precision like in binary search.
- Two numbers are deemed equal if their difference is less than some small epsilon.

```
const double EPS = 1e-9;

if(abs(a - b) < EPS) {
...
}
```

Working with doubles

- Less than operator:

```
if(a < b - EPS) {
...
}
```

- Less than or equal:

```
if(a  < b + EPS) {
...
}
```

- The rest of the operators follow.

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 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

```
bool is_prime(int x) {
    if(x <= 1) return 0;
    for(int i = 2; i * i <= x; ++i)
        if(x % i == 0)
            return false;
    return true;
}
```

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

```
bool is_prime(int x) {
    if(x <= 1) return 0;
    for(int i = 2; i * i <= x; ++i)
        if(x % i == 0)
            return false;
    return true;
}
```

- We will also see a faster (ish) way later on.

GCD

- But how do you calculate GCD?

GCD

- But how do you calculate GCD?
- Say we're considering two numbers $a \geq b$ with a greatest common divisor g. Then g must also divide $a - b$ since it divides both a, b. Say there exists some $g' > g$ that divides b and $a - b$. Then g' also divides $a - b + b = a$, which contradicts the fact that g is the greatest common divisor of a and b.

GCD

- But how do you calculate GCD?
- Say we're considering two numbers $a \geq b$ with a greatest common divisor g. Then g must also divide $a - b$ since it divides both a, b. Say there exists some $g' > g$ that divides b and $a - b$. Then g' also divides $a - b + b = a$, which contradicts the fact that g is the greatest common divisor of a and b.
- What does this all say? Well it means $\gcd(a, b) = \gcd(b, a - b)$. Repeating this logic it gives $\gcd(a, b) = \gcd(b, a\%b)$.

Euclidean algorithm

- This gives us the Euclidean algorithm which a recursive algorithm that computes the GCD of two numbers.

```
int gcd(int a, int b){
    return b == 0 ? a : gcd(b, a % b);
}
```

- Runs in $O(\log N)$ (proof left as exercise).

Euclidean algorithm

- This gives us the Euclidean algorithm which a recursive algorithm that computes the GCD of two numbers.

```
int gcd(int a, int b){
    return b == 0 ? a : gcd(b, a % b);
}
```

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

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

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 $1, 8, -6, 14, -13, \ldots$ 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. The most common example people use is to say this is like using $0, 1, \ldots, 23$ for the hours on a digital clock.

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. The most common example people use is to say this is like using $0, 1, \ldots, 23$ for the hours on a digital clock.
- 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. The most common example people use is to say this is like using $0, 1, \ldots, 23$ for the hours on a digital clock.
- 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.

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.
- We just do the calculations normally and then do `x%n`. If we had a negative number we might have to do `(x%n+n)%n`.

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.

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

```
int egcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    } else {
        int 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.

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 (share no prime factors).

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 (share no prime factors).
- It so happens that when we have from EGCD algorithm

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

then

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

Modular inverse

```
int mod_inv(int a, int m) {
    int x, y, d = egcd(a, m, x, y);
    return d == 1 ? (x%m+m)%m : -1;
}
```

Exponentiation

- Say we want to find $a^e \pmod{n}$. This should be no problem since the answer won't be too big.

Exponentiation

- Say we want to find $a^e \pmod{n}$. This should be no problem since the answer won't be too big.
- We use ideas from the divide and conquer week, calculating $a^{e/2}$ to take the calculation time down from $\mathcal{O}(e)$ to $\mathcal{O}(\log(e))$.

Modular inverse

```
int mod_pow(int b, int e, int n) {
  int res = 1;
  while(e) {
    if(e & 1) res = (res*a%n+n)%n;
    a *= a;
    a %= n;
    e >>= 1;
  }
  return res;
}
```

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.

Chinese remainder theorem

- We won't test the chinese remainder theorem, but it can be very useful to know.
- Sometimes just knowing a solution exists is enough. But this can also be coded, even when the moduli aren't pairwise coprime. It's just a hassle.

Fancier factoring algorithms

Faster prime check?

- Can we check if a number is prime faster than $\mathcal{O}(\sqrt{n})$?
- Sort of.
- We can use probabilistic prime testing, a function that says the input is *probably* prime.
- This may sound shaky, but this program can be run a dozen times and at that point 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 $x = \pm 1 \ (mod \ p)$.
- 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.
- 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 `t` saying how often it should run. This runs in $\mathcal{O}(t \log(n)^3)$ for large n.

Miller-Rabin implementation

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

Bulk discount

- One way to get primes "faster" is getting a bulk discount.

Bulk discount

- One way to get primes "faster" is getting a bulk discount.
- If we want to get a lot of them, we can use the sieve of Eratosthenes.

Bulk discount

- One way to get primes "faster" is getting a bulk discount.
- If we want to get a lot of them, we can use the sieve of Eratosthenes.
    - For all numbers from 2 to $\sqrt{n}$:

Bulk discount

- One way to get primes "faster" is getting a bulk discount.
- If we want to get a lot of them, we can use 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

- One way to get primes "faster" is getting a bulk discount.
- If we want to get a lot of them, we can use 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		

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		

Sieve of Eratosthenes

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

Factoring

- What if we don't just want to check if a number is prime but to get its prime factorization?

Factoring

- What if we don't just want to check if a number is prime but to get its prime factorization?
- Well there is a straight forward way to do it in $\mathcal{O}(\sqrt{n})$ like before.

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

```
map<int,int> factor(int x) {
    map<int, int> primes;
    for(int i = 2; i * i <= x; ++i)
        if(x % i == 0) {
            primes[i] = 0;
            while(x % i == 0) {
                primes[i]++;
                x /= i;
            }
        }
    }
    if(x != 1) primes[x] = 1;
    return primes;
}
```

Factoring

But we can also use the same bulk discount idea as before.

- 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

```
map<int, int> factor(int N) {
    vector<int> primes;
    primes = eratosthenes(static_cast<int>(sqrt(N+1)));
    map<int, int> factors;
    for(int i = 0; i < primes.size(); ++i){
        int prime = primes[i], power = 0;
        while(N % prime == 0){
            power++;
            N /= prime;
        }
        if(power > 0){
            factors[prime] = power;
        }
    }
    if (N > 1) {
        factors[N] = 1;
    }
    return factors;
}
```

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?
- 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

```cpp
#include <bits/stdc++.h>
using namespace std;
pair<int, int> floyd(function<int(int)> f, int x0) {
    int t = f(x0), h = f(f(x0));
    while(t != h) {
        t = f(t);
        h = f(f(h));
    }
    int mu = 0; t = x0;
    while(t != h) {
        t = f(t);
        h = f(h);
        mu++;
    }
    int lam = 1; h = f(t);
    while(t != h) {
        h = f(h);
        lam++;
    }
    // cycle length, starting point of cycle
    return {lam, mu};
}
```

Pollard rho factorization

- The idea is now to let $g(x) = x^2 + 1 \ (mod \ n)$ and create the sequence $x, g(x), g(g(x)), \ldots$ where x is chosen randomly. Denote these numbers $x_1, x_2, \ldots$. We calculate these values and check if $\gcd(x_i - x_j, n) > 1$. Then the sequence has begun repeating not just modulo n but modulo d where d divides n.
- Ef $\gcd(x_i - x_j, n) = 1$ for all values then either n is prime or we just didn't manage to find a divisor. Thus we generally test a few starting values of x before giving up. This is usually only useful for quite big numbers since it doesn't pay off until $x > 2^{32}$ at least.
- The time complexity is an open question, but it's conjectured to be the square root of the largest factor of N. It is thus quite slow for primes, but much faster for composite numbers. Checking for primality first using Miller-Rabin can be useful.

Pollard rho implementation

```
ll rho(ll n) {
    vector<ll> seed = {2, 3, 4, 5, 7, 11, 13, 1031};
    for(ll s : seed) {
        ll x = s, y = x, d = 1;
        while(d == 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:

Number theory functions

The prime factors can be quite useful to calculate many other things.

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

Number theory functions

The prime factors can be quite useful to calculate many other things.

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

- 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 to calculate many other things.

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

- 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.