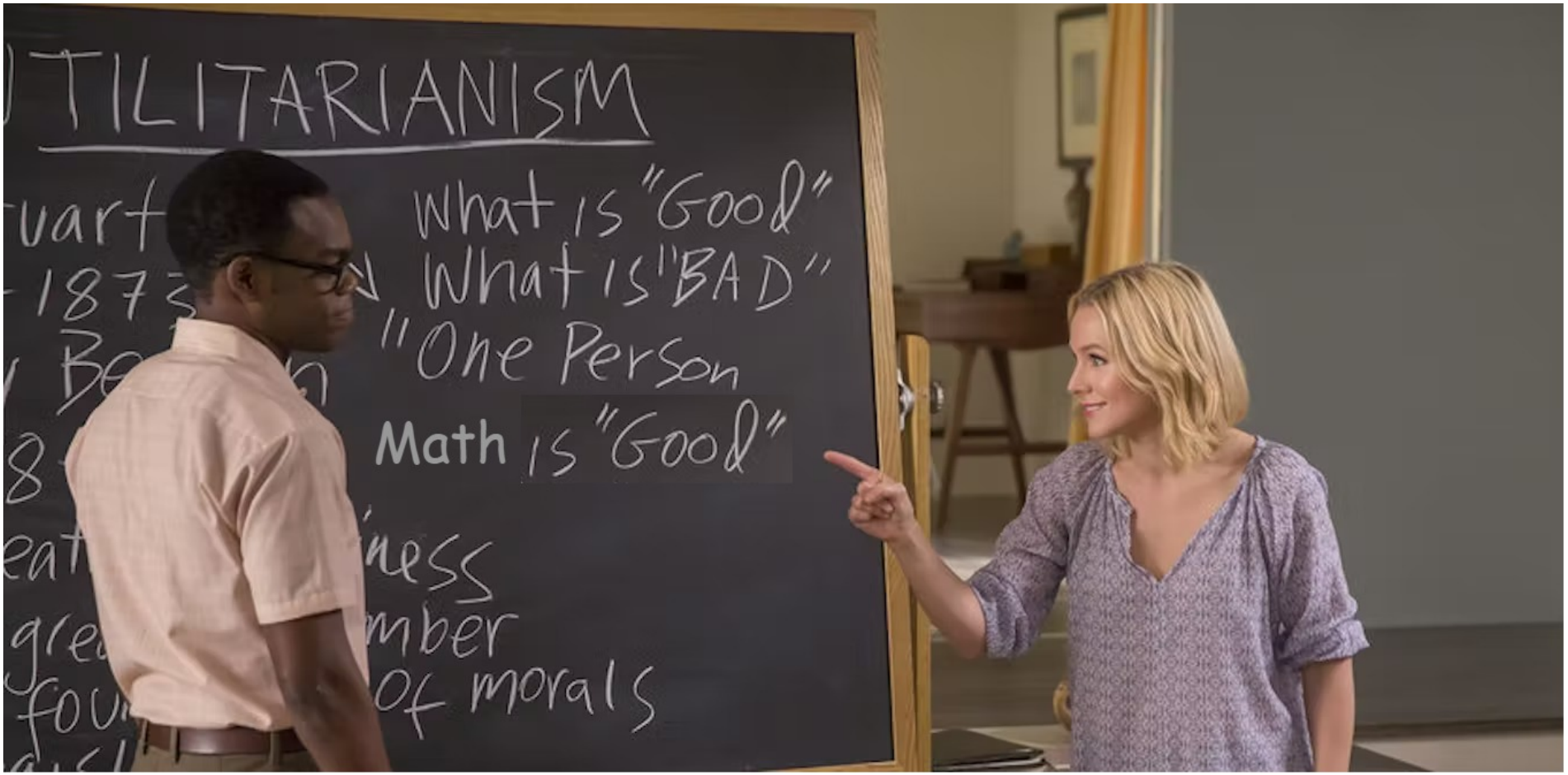
inzva ACWC'23 | Math Notes

Prepared with ♥, just for you...

inzva Team

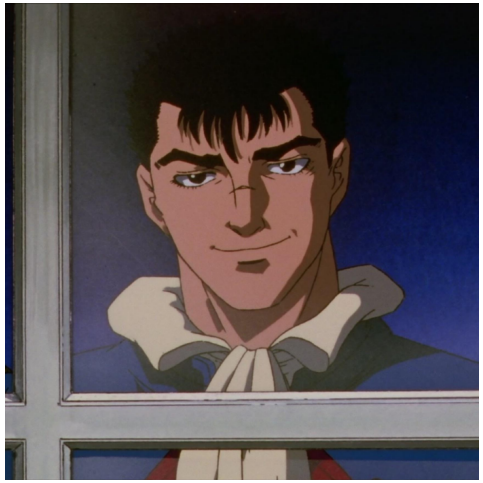
January 24, 2023

Has anyone watched “The Good Place”? :)



What do we get?

1. **Prime Factorization!** (Hey, anyone hating primes? Get the fork out!)
2. **Sieving Primes!** (Thanks Eratosthenes, much appreciated bro!)
3. **GCD & LCM!** (LCM is the elder brother)
4. **Modular Arithmetic!** (Starring: Mrs. Remainder)
5. **Fast Exponentiation!** (I mean, fast as fork)



Guts enjoying math (Berserk)

These notes don't dive into details that much. For more details, please take a look at the more detailed Notion version. :)

Divisors — who divides who?

- d divides $n \rightarrow n = kd$ (n is a multiple of d, d is a divisor of n)
- d doesn't divide $n \rightarrow n = kd + r$ ($0 < r < d$ is the remainder)
  - $r = 0 \rightarrow d$ is a divisor!
- Number of positive divisors is 2 (1 and itself) $\rightarrow n$ is a prime!
- Otherwise, n is a composite! (except 1 but never mind...)

Naruto using Shadow Clone Jutsu

Divisors — how to find?

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

int main() {
    int n = 13012023;
    int n_divisors = 0;

    // Iterate over d's
    for (int d = 1; d ⩽ n; d++) {
        // The remainder is zero → d is a divisor!
        if (n % d == 0) { // Do whatever you want...
            n_divisors++;
        }
    }

    // Hate endl, embrace "\n" :)
    cout << n_divisors << "\n";
}
```

Divisors — how to find?

```python
n = 13012023
n_divisors = 0

for d in range(1, n + 1):
    if n % d == 0:
        n_divisors += 1

print(n_divisors)
```

Divisors — how to find FASTER?

Lucky Luke shooting his own shadow before it shoots him back

Noticed?

$n = kd \rightarrow k$ is also a divisor!

Assume d is smaller $\rightarrow d \leq k \rightarrow d^2 \leq dk = n$

$$d^2 \leq n \rightarrow d \leq \sqrt{n}$$

If d is found, then k is $\frac{n}{d}$!

Term unlocked!

d and k are complementary divisors.

To find d's, check until $\sqrt{n}$!

Divisors — gimme the source!

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    ll n = 1234567ll * 1234567;
    int n_divisors = 0;
    for (ll d = 1; d * d ≤ n; d++) {
        if (n % d)
            continue;

        n_divisors++;
        ll k = n / d;
        if (d ≠ k) {
            n_divisors++;
        }
    }
    cout << n_divisors << "\n";
}
```

Divisors — gimme the source!

```python
n = 1234567**2
n_divisors = 0

for d in range(1, n + 1):
    if d * d > n:
        break
    if n % d:
        continue
    n_divisors += 1
    k = n // d
    n_divisors += d != k

print(n_divisors)
```

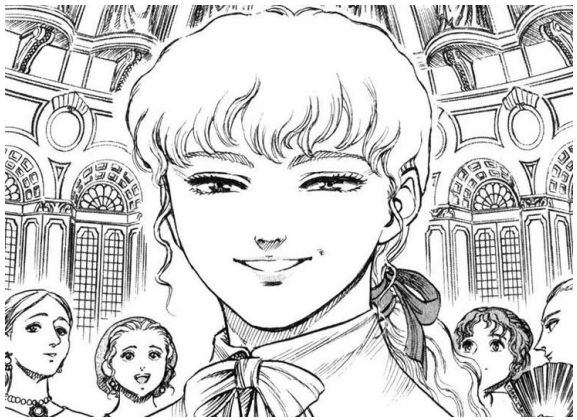
Primes — what's up with 'em?

Any positive integer n can be expressed as:

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

where each p_i is a distinct prime divisor. This is called **prime factorization**.

- Number of divisors: $(e_1 + 1)(e_2 + 1)(e_3 + 1) \ldots$
- n can be expressed with *(prime, exponent)* pairs! (`(unordered) map/array` can be used)
- Max. number of p_i's: $\log n$!



Griffith in his prime (Berserk)

Factorization — how?

1. Smallest divisor s of n is its smallest prime divisor.
   - If n isn't a prime, then $s^2 \leq n$!
2. Find s, then divide n by s as long as it's divisible.
   - To find s, begin the iteration with 2. Then, continue the iteration from where you left off because the next s is definitely larger than the previous.
3. Repeat step 2 until n becomes a prime or 1.

Achievement unlocked!

You factorized in $\mathcal{O}(\sqrt{n})$!

Factorization — how EXACTLY?

Let's take $n = 2^3 \cdot 3^2 \cdot 7^3 \cdot 43^2 \cdot 59$ (assume not already factorized :D):

n	s	Factorization
$2^3 \cdot 3^2 \cdot 7^3 \cdot 43^2 \cdot 59$	2 (divide 3 times)	2^3
$3^2 \cdot 7^3 \cdot 43^2 \cdot 59$	3 (divide 2 times)	$2^3 \cdot 3^2$
$7^3 \cdot 43^2 \cdot 59$	7 (divide 3 times)	$2^3 \cdot 3^2 \cdot 7^3$
$43^2 \cdot 59$	43 (divide 2 times)	$2^3 \cdot 3^2 \cdot 7^3 \cdot 43^2$
59	The condition $s^2 \leq n$ does not hold anymore. n is a prime. We're done.	$2^3 \cdot 3^2 \cdot 7^3 \cdot 43^2 \cdot 59$

Factorization — gimme the source!

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    // 2.694.111.336
    ll n = (ll)2 * 2 * 2 * 3 * 3 * 7 * 7 * 7 * 43 * 59;

    unordered_map<ll, int> exps;
    for (ll d = 2; d * d ⩽ n; d++) {
        if (n % d)
            continue;
        int exp = 0;
        while (n % d == 0) {
            n /= d;
            exp++;
        }
        exps[d] = exp;
    }

    if (n > 1) {
        exps[n] = 1;
    }

    int n_divisors = 1;
    for (auto &[prime, exp] : exps) {
        n_divisors *= exp + 1;
    }
    cout << n_divisors << "\n";
}
```

Factorization — gimme the source!

```python
n = 2**3 * 3**2 * 7**3 * 43**2 * 59
exps = {}  # May use dictionary (hashmap)!
for d in range(2, n + 1):
    if d * d > n:
        break
    if n % d:
        continue
    exp = 0
    while n % d == 0:
        n //= d
        exp += 1

    exps[d] = exp

if n > 1:
    exps[n] = 1

n_divisors = 1
for prime, exp in exps.items():
    n_divisors *= exp + 1

print(n_divisors)
```

Primes — we're forking back!

What if we wanted to detect all primes up to n?

- For every integer i, check if divisible by d s.t. $1 < d \leq \sqrt{i}$
  - Number of operations approaches $n\sqrt{n}$
  - What if $n = 10^7$? Meh...
- What about detecting composites?

Eleanor (The Good Place)

~~Primes~~ Composites

- For every integer $i > 1$ up to n, we can mark the multiples of i as composite: $2i, 3i, 4i, \ldots$
- Which integers would be left unmarked? Primes!
- The cost?

$$n\left(\frac{1}{2} + \frac{1}{3} + \cdots\right) \approx n \log n$$

  - For geeks among us, it's based on harmonic series :)
- Called Sieve of Eratosthenes!

Sieve — how to?

Let's sieve for $n = 20$:

1. Initial: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20$
2. Mark $2k$: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20$
3. Mark $3k$: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20$
4. Mark $4k$: Not needed, already marked $2k$
5. Mark $5k$: Not needed, already marked for $k < 5$. For $k \geq 5$, $5k > 20$.
6. We're actually done!

Sieve — how to FASTER?

1. When i isn't a prime, its multiples have already been marked as i is a multiple of its prime divisors.
   - Ignore composite i's!
   - Number of operations drops down to $n \log \log n$ (yes, $\log$ of $\log$)
2. For $k < i$, $k \cdot i$'s have already been marked. So, might start from $k = i$.
   - That's why it's enough to go up to $i = \sqrt{n}$
   - Not as game-changer as the previous optimization, optional to use

Sieve — "gieve" us the source!

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    int n = 1e7;
    // Initially, all are unmarked (marked means composite)
    vector<bool> is_prime(n + 1, 1);
    is_prime[0] = is_prime[1] = 1; // Not that necessary
    for (int i = 2; i <= n; i++) {
        if (!is_prime[i])
            continue;
        for (int k = 2 * i; k <= n; k += i) {
            is_prime[k] = 0;
        }
    }

    // Do whatever you want
    int n_primes = 0;
    for (int i = 2; i <= n; i++)
        n_primes += is_prime[i];
    cout << n_primes << "\n";
}
```

Sieve — “gieve” us the source!

```python
n = 10**7
is_prime = [1] * (n + 1)
n_primes = 0

for i in range(2, n + 1):
    if not is_prime[i]:
        continue
    n_primes += 1
    for k in range(2 * i, n + 1, i):
        is_prime[k] = 0

print(n_primes)
```

Sieve — factorize even faster!

Notice that while factorizing, we unnecessarily iterate over non-primes to reach the next smallest prime divisor (s) eventually. We can actually overcome this issue by storing primes in an array. Then, we can just iterate over this array and stop whenever the current prime's square exceeds n.
This method is useful even if n is large because the largest prime required to find is at most $\sqrt{n}$.

Primes are much fewer!

The number of primes up to n is $\approx \frac{n}{\log n}$.

Achievement unlocked!

You factorized in $O\left(\frac{\sqrt{n}}{\log \sqrt{n}}\right)$!

Sieve — factorize even faster!

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    int n = 1e7;
    vector<bool> is_prime(n + 1, 1);
    vector<int> primes;
    for (int i = 2; i <= n; i++) {
        if (!is_prime[i])
            continue;
        primes.push_back(i);
        for (int k = 2 * i; k <= n; k += i) {
            is_prime[k] = 0;
        }
    }
    ll i = (ll)1234567 * 1234567;
    unordered_map<ll, int> exps;
    for (int p : primes) {
        if ((ll)p * p > i)
            break;
        if (i % p)
            continue;
        int exp = 0;
        while (i % p == 0) {
            i /= p;
            exp++;
        }
        exps[p] = exp;
    }
    if (i > 1) { // i might be a prime up to 10**14!
        exps[i] = 1;
    }
}
```

Sievish — factorize fast as fork!

Sieving isn't only the method that uses this fantastic concept of *iterating over multiples*.

Remember, remember the 10^{th} page!

An integer n has at most $\log n$ prime divisors.

There is another must-know sieve-like algorithm: **finding a prime divisor of each integer up to** n. Notice that if we could precalculate them fast enough, we could factorize any integer i in $\log i$ steps by repeatedly dividing the current number by its prime divisor until we're left with 1. In fact, we're able to do that precalculation! It's best to just examine the implementation on the next page.

Achievement unlocked!

You factorized in $\log n$!

Fastest, but...

You can't really use this method for huge n's, because notice that this doesn't involve any $\sqrt{n}$ stuff!

Sievish — factorize fast as fork!

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    int n = 1e7;
    vector<int> prime_divisor(n, 0);
    // Still n * log log(n)!
    for (int i = 2; i <= n; i++) {
        if (prime_divisor[i]) // i isn't a prime, it has a smaller prime divisor
            continue;
        for (int k = 2 * i; k <= n; k += i) {
            prime_divisor[k] = i;
            // Might also do this to keep only the smallest prime divisor:
            // if(!prime_divisor[k]) prime_divisor[k] = i;
        }
    }
    // Factorize i in log(i) steps!
    int i = 987654;
    int n_divisors = 1;
    while (i > 1) {
        int p = prime_divisor[i];
        int exp = 0;
        while (i % p == 0) {
            i /= p;
            exp++;
        }
        n_divisors *= exp + 1;
    }
}
```

GCD — greatest of common divisors!

Take two integers:

$$x = p_1^{a_1} \cdot p_2^{a_2} \cdot p_3^{a_3} \cdots$$
$$y = p_1^{b_1} \cdot p_2^{b_2} \cdot p_3^{b_3} \cdots$$

- Common divisor z: Must both divide x and y.

$$z = p_1^{c_1} \cdot p_2^{c_2} \cdot p_3^{c_3} \cdots$$

  c_i must not exceed a_i or b_i. So, $c_i \leq \min(a_i, b_i)$.
- GREATEST common divisor $\gcd(x, y)$: $c_i = \min(a_i, b_i)$!
- Applicable to any number of integers, not just two!

GCD — Euclid's cool stuff

We can ignore factorization. Say $\gcd(x, y) = g$, and let $\gcd(x - y, y) = f$.

1. x and y are multiples of $g \rightarrow x = gm, y = gn$.
   - $\gcd(m, n) = 1$. Otherwise, $\gcd(x, y) > g$.
2. $x - y = g(m - n)$, so $f = g \cdot \gcd(m - n, n)$.
3. $\gcd(m - n, n) = k \rightarrow$ both m and n are multiples of k. So, $k = 1$.
4. $f = g$, voila!

So, subtracting one from another doesn't change the gcd.

Euclidean Algorithm

$$\gcd(x, y) = \gcd(x\%y, y) = \gcd(x, y\%x)$$

Can compute gcd in $O(\log \min(x, y))$!

GCD — gimme the source!

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

// Always keep the smaller one be y (or x) for simpler coding!

ll iterative_gcd(ll x, ll y) {
    // While both are non-zero, perform larger = larger % smaller
    while (x > 0 && y > 0) {
        if (x >= y)
            x %= y;
        else
            y %= x;
    }

    // Afterward, the non-zero one is the GCD!
    return (x ? x : y);
}

ll iterative_gcd_v2(ll x, ll y) {
    // While y is non-zero, perform (x, y) = (y, x % y)
    while (y > 0) {
        ll tmp = y;
        y = x % y;
        x = tmp;
    }

    // After y becomes 0, x is the GCD!
    return x;
}

ll recursive_gcd(ll x, ll y) {
    if (y == 0)
        return x;
    return recursive_gcd(y, x % y);
}
```

GCD — gimme the source!

```python
def iterative_gcd(x, y):
    while x and y:
        if x ⩾ y:
            x %= y
        else:
            y %= x
    return x if x else y


def iterative_gcd_v2(x, y):
    while y:
        x, y = y, x % y
    return x


def recursive_gcd(x, y):
    if not y:
        return x
    return recursive_gcd(y, x % y)
```

LCM — least of common multiples!

- Common multiple z: Must be divisible by both x and y.

$$z = p_1^{c_1} \cdot p_2^{c_2} \cdot p_3^{c_3} \cdots$$

  c_i must be not smaller than a_i or b_i. So, $c_i \geq \max(a_i, b_i)$.
- LEAST common multiple $\text{lcm}(x, y)$: $c_i = \max(a_i, b_i)$!
- Applicable to any number of integers, not just two!

Special case: 2 integers

We know the sum of the larger and the smaller of two integers equals the sum of these integers, that is, $\max(a_i, b_i) + \min(a_i, b_i) = a_i + b_i$. Therefore,

$$\gcd(x, y) \cdot \text{lcm}(x, y) = x \cdot y$$

As a result, $\text{lcm}(x, y) = \frac{x \cdot y}{\gcd(x, y)}$!
However, for n integers, $\text{lcm}(a_1, a_2, \ldots, a_n)$ might not be $\frac{a_1 \cdot a_2 \cdots a_n}{\gcd(a_1, a_2, \ldots, a_n)}$. Figure out why!

GCD & LCM — dealing with many integers

Itachi (LCM) & Sasuke (GCD)

Say we have n integers $a_1, a_2, \ldots, a_n$, and we want to find the $\gcd$ of all. If we consider the $\gcd$ operation as if we find the $\min$'s of primes' exponents, it's a bit clear that the following works:

1. Start with $g = 0$.
2. Perform $g := \gcd(g, a_i)$ for each i one by one.

Why?

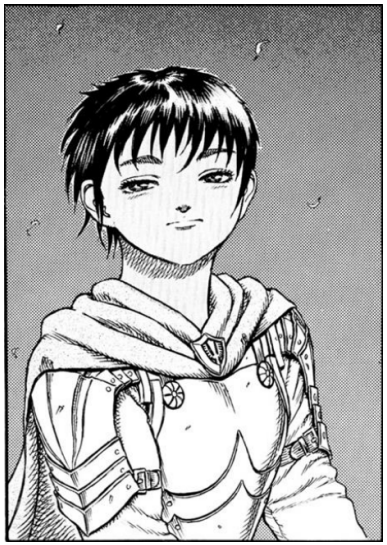
What $\gcd$ does to exponents is what $\min$ does to integers themselves. It's very similar to doing the same sequence of operations with $\min$ — to find the $\min$, we iterate over each integer one by one and update the result $m = \min(m, a_i)$.

As you might have guessed, the correspondence between lcm and $\max$ operations is **exactly** the same.

GCD & LCM — show us how to deal!

```cpp
// Observe the similarity between min and gcd

ll min_of_many(vector<ll> &integers) {
    // Assume at least one element exists, or use LLONG_MAX as the initial value
    ll overall_min = integers[0];
    for (ll &i : integers) {
        overall_min = min(overall_min, i);
    }
    return overall_min;
}

ll gcd_of_many(vector<ll> &integers) {
    ll overall_gcd = 0;
    for (ll &i : integers) {
        overall_gcd = __gcd(overall_gcd, i);
    }
    return overall_gcd;
}

// Observe the similarity between max and lcm

ll max_of_many(vector<ll> &integers) {
    // Assume at least one element exists, or use LLONG_MIN as the initial value
    ll overall_max = integers[0];
    for (ll &i : integers) {
        overall_max = max(overall_max, i);
    }
    return overall_max;
}

ll lcm_of_many(vector<ll> &integers) {
    ll overall_lcm = 1; // Notice that 1 the minimum initial value for max'ing (all exps are zero)
    for (ll &i : integers) {
        // Notice how we first divided, then multiplied
        // It's just to prevent overflow!
        overall_lcm = overall_lcm / __gcd(overall_lcm, i) * i;
    }
    return overall_lcm;
}
```

Modular Arithmetic — help us Mrs. Remainder!



Casca (Berserk)

In competitive programming, we don't really deal with BIG integers that don't even fit in 64 bits. Instead, we use remainders: actual result *modulo* m where m is usually chosen as $10^9 + 7$, which is a prime number. Notice that we can perform basic arithmetic operations (summation and multiplication) on two values less than m using 64-bit integers.

Modular Arithmetic — summation & multiplication

Take an expression $a_1 \star a_2 \star a_3 \star \cdots \star a_n$ where every a_i is an integer, and each $\star$ can be either + or $\cdot$. We want to compute it modulo m.

1. Instead of a_i, we can take $a_i \mod m$. The equivalent of this operation is adding $k \cdot m$ to the whole expression, where k is an integer. This isn't going to change the result modulo m, i.e., the remainder.
2. After performing one of the operations $a_i \star a_{i+1}$, the general structure of the expression stays the same.
3. These are why we can perform the modulo operation whenever/wherever we want!

Modular Arithmetic — division

Division is the annoying one. There might be a lot to discuss, but generally, we use Fermat's little theorem to take care of divisions.

Fermat's little gift

If m is a prime and a isn't a multiple of m, the following holds:

$$a^{m-1} \equiv 1 \pmod{m}$$

Using this gift, instead of $\frac{p}{q}$, we multiply it by q^{m-1} and so, calculate $p \cdot q^{m-2}$. Notice how it turned into multiplication! However, note that q must **not** be divisible by m. You won't encounter the opposite most of the time, though.

New term unlocked!

In fact, q^{m-2} has its own name — q's modular inverse! It's because multiplying it by q results in 1 modulo m.

Fast Exponentiation — fast as fork!

Did you know that we can calculate $a^b \mod m$ REALLY fast?

Do we have to see you everywhere, Mr. Binary?

b can be represented as the sum of at most $k = \lfloor \log b \rfloor$ distinct powers of 2:

$$b = c_0 \cdot 2^0 + c_1 \cdot 2^1 + c_2 \cdot 2^2 + \cdots + c_k \cdot 2^k$$

where each c_i is either 0 or 1.

We can find c_i's by continuously performing the following operations:

1. Initially, $i = 0$.
2. $c_i = b \bmod 2$.
3. $b := \left\lfloor \frac{b}{2} \right\rfloor$, increment i.
4. Repeat steps 2-3 until b becomes 0.

It works because each time, b will be in the form of $c_i + 2k$. Also, it's clear that this only requires $\lfloor \log b \rfloor$ iterations!

Fast Exponentiation — fast as fork!

What to do after finding c_i's?

- $a^b = a^{c_0 \cdot 2^0 + c_1 \cdot 2^1 + c_2 \cdot 2^2 + \cdots + c_k \cdot 2^k}$ $(k = \lfloor \log b \rfloor)$
- Notice that $a^{2^{i+1}} = a^{2^i} \cdot a^{2^i}$
  - $a^2 = a \cdot a$
  - $a^4 = a^2 \cdot a^2$
  - $a^8 = a^4 \cdot a^4$
  - $\ldots$

We can continuously take the square of a^{2^i}, and whenever $c_i = 1$, we multiply the result by a^{2^i}. I.e., we can gather the powers of 2 we need for the exponent.

Achievement unlocked!

You've just calculated $a^b \bmod m$ in $\log b$ steps!

Remember Fermat's!

Can take $b \bmod (m-1)$ instead of b if m is a prime and a isn't divisible by m.

Fast Exponentiation — gimme the source!

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll fast_exp(ll a, ll b, int m) {
    // If m is a prime and a ≠ m*k, can do the following:
    // b %= m-1;

    // Can do this without any restrictions!
    a %= m;

    ll result = 1;
    while (b) { // Find c_i's while b > 0
        bool c_i = b % 2;
        if (c_i) {
            result = result * a % m;
        }
        b /= 2;

        // a becomes a^(2^i)
        a = a * a % m;
    }

    return result;
}

int main() {
    ll a = 12345678910111211;
    ll b = 1e18;
    int m = 1e9 + 7;
    cout << fast_exp(a, b, m) << "\n";
}
```

Fast Exponentiation — gimme the source!

```python
def fast_exp(a, b, m):
    a %= m
    result = 1
    while b:
        if b % 2:
            result = result * a % m
        b //= 2
        a = a * a % m
    return result


a = 123456789101112
b = 10**18
m = 10**9 + 7

print(fast_exp(a, b, m))
print(pow(a, b, m))  # Built-in fast exponentiation!
```

Using our new modular toys — combinatorics!

```cpp
const int mod = 1e9 + 7;
const int N = 1e6 + 5;

// f[i] = i!
// inv_f[i] = 1/i!
int f[N], inv_f[N]; // defining as ll will consume 2x memory!

// a^(mod-2) (1/a)
ll modular_inverse(ll a) {
    a %= mod;
    int exp = mod - 2;
    ll res = 1;
    while (exp) {
        if (exp % 2)
            res = res * a % mod;
        exp /= 2;
        a = a * a % mod;
    }
    return res;
}

// Fill in the factorial arrays
void setup() {
    f[0] = 1;
    for (int i = 1; i < N; i++) {
        f[i] = 1ll * f[i - 1] * i % mod;
    }
    inv_f[N - 1] = modular_inverse(f[N - 1]);
    for (int i = N - 2; i >= 0; i--) {
        inv_f[i] = 1ll * inv_f[i + 1] * (i + 1) % mod;
    }
}
```

Using our new modular toys — combinatorics!

```cpp
// C(n, k) binomial coefficient
// Slowest version → 2 * log(mod)
ll nCk_v1(int n, int k) {
    return (ll)f[n] * modular_inverse(f[k]) % mod * modular_inverse(f[n - k]) % mod;
}
// A bit faster version → log(mod)
ll nCk_v2(int n, int k) {
    return (ll)f[n] * modular_inverse((ll)f[k] * f[n - k] % mod) % mod;
}
// Fastest version → O(1)
ll nCk_v3(int n, int k) {
    return (ll)f[n] * inv_f[k] % mod * inv_f[n - k] % mod;
}
```

Thanks for listening!

Satoru & Kayo (Erased)