Number Theory

For computing contests

Topics

- Floor/ceiling function
- Modular arithmetic
- Fermat’s little theorem
- Binary exponentiation (fast exponentiation for non-negative integer exponents)
- Binomial coefficients
- Euclidean algorithm (fast GCD)

Floor/ceiling functions

- floor(x) = the largest integer smaller than x
  - For non-negative integers this can be thought of as truncating everything after the decimal point
- ceil(x) = the smallest integer greater than x

Examples

- floor(1.5) = 1
- floor(1) = 1
- floor(1.999) = 1
- floor(-1.5) = -2
- ceil(1.5) = 2
- ceil(2) = 2
- ceil(1.001) = 2
- ceil(-1.5) = -1

Useful facts

- Floor division of two numbers, a and b, is floor(a/b)
- In c++ and java, all integer division is floor division
- Python has a // operator for floor division (I will use this operator for floor division later in the presentation)
- If a and b are integers, then (a+b-1)//b = ceil(a/b)
- If you have floor(x+n), where n is an integer, you can replace it with n+floor(x)
- a//b has at most O(sqrt(a)) unique values

Modular arithmetic

Imagine you're given this problem:

Given three integers $1 \le a,b,c \le 10^{18}$ find the remainder of ab when divided by c

Observations

- The number will be too big if we naively multiply (about 128 bits)
- The remainder of a divided by c (a%c) is a - c(a//c)
- (a+cx)%c = a%c
  Proof: (a+cx)%c = a+cx - c((a+cx)//c) = a+cx - c((a//c) + x) = a - c(a//c) = a%c
- (x+y)%c = (x%c+y%c)%c
  Proof: (x%c+y%c) = x-c(x//c) + y-c(y//c) = x+y - c((x+y)//c). (x+y-c((x+y)//c))%c = (x+y)%c

Observations

- This means that we can add and take the remainder after each step
- 10^{18} additions is still too slow
- We can do it in $O(\log_2(b))$ additions using doubling
  Let x_0 = a. $x_n = (x_{n-1}+x_{n-1})\%c$. This makes $x_n=(a2^n)\%c$. Now, if we break b up into powers of two, we can sum these terms of x and get (ab)%c
- Breaking b into powers of two just means looking at its digits in binary

Pseudocode

```
def multiply_and_get_remainder(a, b, c)
    x = a
    out = 0
    for i in 1:bit_length(b)
        if b.bit_at(i) == 1
            out += x
            out %= c
        x = (x+x)%c
    return out
```

Python and c++ implementations

Note: subtraction is much faster than modulo, so always use it instead of possible

Also, python can store arbitrarily large integers, so this code is useless

```python
def multiply_and_get_remainder(a, b, c):
    x = a
    out = 0
    while b:
        if b&1:
            out += x
            if out >= c: out -= c
        x += x
        if x >= c: x -= c
        b >>= 1
    return out
```

```cpp
using ll = long long;
ll multiply_and_get_remainder(ll a, ll b, ll c) {
    ll x = a, out = 0;
    while (b) {
        if (b&1) {
            out += x;
            if (out >= c) out -= c;
        }
        x += x;
        if (x >= c) x -= c;
        b >>= 1;
    }
    return out;
}
```

Modular arithmetic

- This example is a bit contrived, but it showed that if you're adding numbers together, you can take a modulo anywhere and it won't change the answer
- This also works for multiplication
  Proof: ((x%b) * (y%b)) = (x-b(x//b))*(y-b(y//b)) = xy - (something times b).
  (xy - (something times b))%b = (xy)%b

Modular arithmetic terminology

- “a is congruent to b mod c” if a%c = b%c
    - This is written as $a \equiv b \pmod{c}$
- A residue is the result of the operation a mod c

Modular exponentiation

- Because you can modulo after each operation when multiplying, you can use similar logic to the previous problem to solve $a^b \% c$
- Instead of doubling, square
- Instead of adding, multiply

Python and c++ implementation

```python
# This method is built in to Python
def modPow(a, b, c):
    return pow(a, b, c)
```

```cpp
using ll = long long;
ll modPow(ll a, ll b, ll c) {
    ll out = 1, x = a;
    while (b) {
        if (b&1) out = (out*x)%c;
        x = (x*x)%c;
        b >>= 1;
    }
    return out;
}
```

Fermat's little theorem/modular inverse

- From this point on I will only talk about prime mods
- Fermat's little theorem:
    - $a^{p-1} \equiv 1 \pmod{p}$
- In other words:
    - $a^{p-2} * a \equiv 1 \pmod{p}$
- We can use this for division!

Modular inverse

- Imagine you're asked to find x given that $7x \equiv 3 \pmod{19}$
- Start by "dividing" both sides by 7
  - Dividing by seven is the same as multiplying by $7^{17} \pmod{19}$
  - $7^{17} \equiv 11 \pmod{19}$
- Now we know that $x \equiv 3*11 \pmod{19}$

  $\equiv 33 \equiv 14$
- $x \equiv 14 \pmod{19}$

Binomial coefficients

- Because we have a way to do division mod p, we can now evaluate expressions like n choose k
- Calculate n!, (n-k)!, and k! mod p
- Take the inverse of (n-k)!k!
- Multiply them together

Solving a problem

Let's use these tools to solve this problem: https://cses.fi/problemset/task/1715

You're given a string of N letters and asked: how many different strings can be made by rearranging all of these letters? The answer should be given mod 1e9+7

Example:

If the string is aba, then we can create 3 strings:
aab, aba, baa

Observations

- There are N! ways to rearrange the letters, but this counts some strings more than once
- Swapping two ‘a’s in a string doesn’t change it
- We can remove these extra strings N! by dividing out the number of ways to arrange each letter individually
    - aba has two ‘a’s and one ‘b’. The number of arrangements are 3!/(2!*1!)
- We can use modular inverse for this

C++ implementation

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

const ll base = 1000000007;

// Fast modular exponentiation
ll modPow(ll n, ll k) {
	ll out = 1, x = n;
	while (k) {
		if (k&1) out = (out*x)%base;
		x = (x*x)%base;
		k >>= 1;
	}
	return out;
}

// Get the inverse of n mod base
ll modInverse(ll n) {
	return modPow(n, base-2);
}

// Get n! mod base
ll factorial(ll n) {
	ll out = 1;
	for (ll i = 1; i <= n; i++) out = (out*i)%base;
	return out;
}

int counts[26];
int main() {
	string s; cin >> s;
	// Count the occurrences of each letter
	for (int i = 0; i < s.size(); i++) counts[s[i]-'a']++;

	ll out = factorial(s.size());
	for (int i : counts) {
		out = (out*modInverse(factorial(i)))%base;
	}
	cout << out << endl;
}
```

Euclidean algorithm

- Euclidean algorithm is a method of finding the greatest common divisor of two numbers efficiently
- $\gcd(12, 8) = 4$
- Also, $x|y$ means that x divides y, or x is a factor of y
- If $(a+b)|y$, and $a|y$, then $b|y$

  Proof: $a \equiv 0 \pmod{y}$. $a + b \equiv 0 \pmod{y}$. Therefore $b \equiv 0 \pmod{y}$

Euclidean algorithm

- It’s based on the following fact: gcd(a, b) = gcd(b, a%b)
  Proof: gcd(b, a%b) = gcd(b, a-b*(a//b)). Since gcd(a,b) divides a and b, this proves that gcd(b, a%b) ≥ gcd(a,b). Since b is a multiple of gcd(b,b%a), and a-b(a//b) is also a multiple of gcd(b,b%a), we know that gcd(b,b%a)|a. Since it divides a and b, and gcd(b,b%a) ≥ gcd(a,b), it must equal gcd(a,b).

Euclidean algorithm

- We can repeatedly apply this identity. Here is an example for gcd(27, 15):
  gcd(27, 15)
  = gcd(15, 27%15)
  = gcd(15, 12)
  = gcd(12, 15%12)
  = gcd(12, 3)
  = gcd(3, 12%3)
  = gcd(3, 0) = 3
- The gcd of anything with zero is itself (everything divides zero)

Implementation

This algorithm runs in O(log(min(a,b)))

```
ll gcd(ll a, ll b) {
        if (b == 0) return a;
        return gcd(b, a%b);
}
```

Factorization

- Factoring is well known as a problem that is difficult to do efficiently
- This is the basis for a lot of cryptography
- There is a very simple method that runs in O(sqrt(N))
- If there is a factor, f, greater than sqrt(N), then N/f < sqrt(N)
- Iterate from 1 to sqrt(N) and check if anything divides N

Implementation

```
vector<int> factors(int N) {
    vector<int> out;
    for (int i = 1; i*i <= N; i++) {
        // Test if i divides N
        if (N%i == 0) {
            out.push_back(i);
            // Don't add N/i as a factor if i == sqrt(N)
            if (i*i != N) out.push_back(N/i);
        }
    }
    return out;
}
```

Prime detection

- Two ways of finding primes:
- Find factors in O(sqrt(N)) and check if there are only two
- Use a sieve, like Sieve of Eratosthenes

Sieve of Eratosthenes

- Create an array that stores at the ith index whether or not i is prime
- All numbers default to true
- Start looping at 2, and every time a prime number is encountered set all of its multiples to false
- Once the loop is done all the composite numbers will be set to false
- $O(n \log \log n)$ complexity

Implementation

```
const int N = 1000000;
bitset<N+1> prime;
void sieve() {
    // Default all numbers to true
    prime.set();
    // 1 is not prime
    prime[1] = false;
    for (int i = 2; i <= N; i++) {
        if (prime[i]) {
            // Set all multiples to false
            for (int j = 2*i; j <= N; j+=i) {
                prime[j] = false;
            }
        }
    }
    // Now only prime indices are set to true
}
```

Practice problems

- https://dmoj.ca/problem/ccc02s2
- https://dmoj.ca/problem/multimadness
- https://dmoj.ca/problem/qccp3
- https://dmoj.ca/problem/phantom3
- https://dmoj.ca/problem/dmopc21c5p1
- https://dmoj.ca/problem/dmopc21c5p2
- https://dmoj.ca/problem/dmopc21c5p3
- https://dmoj.ca/problem/dmopc21c5p4
- https://cses.fi/problemset/task/2209
- https://cses.fi/problemset/task/2182
- https://dmoj.ca/problem/aac4p2
- https://cses.fi/problemset/task/1081
- https://cses.fi/problemset/task/2417/