Big Integer (c++)

const int BASE\_DIGITS = 9;

const int BASE = 1000000000;

struct BigInt {

int sign;

vector<int> a;

// -------------------- Constructors --------------------

// Default constructor.

BigInt() : sign(1) {}

// Constructor from long long.

BigInt(long long v) {

\*this = v;

}

BigInt& operator = (long long v) {

sign = 1;

if (v < 0) {

sign = -1;

v = -v;

}

a.clear();

for (; v > 0; v = v / BASE)

a.push\_back(v % BASE);

return \*this;

}

// Initialize from string.

BigInt(const string& s) {

read(s);

}

// -------------------- Input / Output --------------------

void read(const string& s) {

sign = 1;

a.clear();

int pos = 0;

while (pos < (int) s.size() && (s[pos] == '-' || s[pos] == '+')) {

if (s[pos] == '-')

sign = -sign;

++pos;

}

for (int i = s.size() - 1; i >= pos; i -= BASE\_DIGITS) {

int x = 0;

for (int j = max(pos, i - BASE\_DIGITS + 1); j <= i; j++)

x = x \* 10 + s[j] - '0';

a.push\_back(x);

}

trim();

}

friend istream& operator>>(istream &stream, BigInt &v) {

string s;

stream >> s;

v.read(s);

return stream;

}

friend ostream& operator<<(ostream &stream, const BigInt &v) {

if (v.sign == -1 && !v.isZero())

stream << '-';

stream << (v.a.empty() ? 0 : v.a.back());

for (int i = (int) v.a.size() - 2; i >= 0; --i)

stream << setw(BASE\_DIGITS) << setfill('0') << v.a[i];

return stream;

}

// -------------------- Comparison --------------------

bool operator<(const BigInt &v) const {

if (sign != v.sign)

return sign < v.sign;

if (a.size() != v.a.size())

return a.size() \* sign < v.a.size() \* v.sign;

for (int i = ((int) a.size()) - 1; i >= 0; i--)

if (a[i] != v.a[i])

return a[i] \* sign < v.a[i] \* sign;

return false;

}

bool operator>(const BigInt &v) const {

return v < \*this;

}

bool operator<=(const BigInt &v) const {

return !(v < \*this);

}

bool operator>=(const BigInt &v) const {

return !(\*this < v);

}

bool operator==(const BigInt &v) const {

return !(\*this < v) && !(v < \*this);

}

bool operator!=(const BigInt &v) const {

return \*this < v || v < \*this;

}

// Returns:

// 0 if |x| == |y|

// -1 if |x| < |y|

// 1 if |x| > |y|

friend int \_\_compare\_abs(const BigInt& x, const BigInt& y) {

if (x.a.size() != y.a.size()) {

return x.a.size() < y.a.size() ? -1 : 1;

}

for (int i = ((int) x.a.size()) - 1; i >= 0; --i) {

if (x.a[i] != y.a[i]) {

return x.a[i] < y.a[i] ? -1 : 1;

}

}

return 0;

}

// -------------------- Unary operator - and operators +- --------------------

BigInt operator-() const {

BigInt res = \*this;

if (isZero()) return res;

res.sign = -sign;

return res;

}

// Note: sign ignored.

void \_\_internal\_add(const BigInt& v) {

if (a.size() < v.a.size()) {

a.resize(v.a.size(), 0);

}

for (int i = 0, carry = 0; i < (int) max(a.size(), v.a.size()) || carry; ++i) {

if (i == (int) a.size()) a.push\_back(0);

a[i] += carry + (i < (int) v.a.size() ? v.a[i] : 0);

carry = a[i] >= BASE;

if (carry) a[i] -= BASE;

}

}

// Note: sign ignored.

void \_\_internal\_sub(const BigInt& v) {

for (int i = 0, carry = 0; i < (int) v.a.size() || carry; ++i) {

a[i] -= carry + (i < (int) v.a.size() ? v.a[i] : 0);

carry = a[i] < 0;

if (carry) a[i] += BASE;

}

this->trim();

}

BigInt operator += (const BigInt& v) {

if (sign == v.sign) {

\_\_internal\_add(v);

} else {

if (\_\_compare\_abs(\*this, v) >= 0) {

\_\_internal\_sub(v);

} else {

BigInt vv = v;

swap(\*this, vv);

\_\_internal\_sub(vv);

}

}

return \*this;

}

BigInt operator -= (const BigInt& v) {

if (sign == v.sign) {

if (\_\_compare\_abs(\*this, v) >= 0) {

\_\_internal\_sub(v);

} else {

BigInt vv = v;

swap(\*this, vv);

\_\_internal\_sub(vv);

this->sign = -this->sign;

}

} else {

\_\_internal\_add(v);

}

return \*this;

}

// Optimize operators + and - according to

// https://stackoverflow.com/questions/13166079/move-semantics-and-pass-by-rvalue-reference-in-overloaded-arithmetic

template< typename L, typename R >

typename std::enable\_if<

std::is\_convertible<L, BigInt>::value &&

std::is\_convertible<R, BigInt>::value &&

std::is\_lvalue\_reference<R&&>::value,

BigInt>::type friend operator + (L&& l, R&& r) {

BigInt result(std::forward<L>(l));

result += r;

return result;

}

template< typename L, typename R >

typename std::enable\_if<

std::is\_convertible<L, BigInt>::value &&

std::is\_convertible<R, BigInt>::value &&

std::is\_rvalue\_reference<R&&>::value,

BigInt>::type friend operator + (L&& l, R&& r) {

BigInt result(std::move(r));

result += l;

return result;

}

template< typename L, typename R >

typename std::enable\_if<

std::is\_convertible<L, BigInt>::value &&

std::is\_convertible<R, BigInt>::value,

BigInt>::type friend operator - (L&& l, R&& r) {

BigInt result(std::forward<L>(l));

result -= r;

return result;

}

// -------------------- Operators \* / % --------------------

friend pair<BigInt, BigInt> divmod(const BigInt& a1, const BigInt& b1) {

assert(b1 > 0); // divmod not well-defined for b < 0.

long long norm = BASE / (b1.a.back() + 1);

BigInt a = a1.abs() \* norm;

BigInt b = b1.abs() \* norm;

BigInt q = 0, r = 0;

q.a.resize(a.a.size());

for (int i = a.a.size() - 1; i >= 0; i--) {

r \*= BASE;

r += a.a[i];

long long s1 = r.a.size() <= b.a.size() ? 0 : r.a[b.a.size()];

long long s2 = r.a.size() <= b.a.size() - 1 ? 0 : r.a[b.a.size() - 1];

long long d = ((long long) BASE \* s1 + s2) / b.a.back();

r -= b \* d;

while (r < 0) {

r += b, --d;

}

q.a[i] = d;

}

q.sign = a1.sign \* b1.sign;

r.sign = a1.sign;

q.trim();

r.trim();

auto res = make\_pair(q, r / norm);

if (res.second < 0) res.second += b1;

return res;

}

BigInt operator/(const BigInt &v) const {

return divmod(\*this, v).first;

}

BigInt operator%(const BigInt &v) const {

return divmod(\*this, v).second;

}

void operator/=(int v) {

assert(v > 0); // operator / not well-defined for v <= 0.

if (llabs(v) >= BASE) {

\*this /= BigInt(v);

return ;

}

if (v < 0)

sign = -sign, v = -v;

for (int i = (int) a.size() - 1, rem = 0; i >= 0; --i) {

long long cur = a[i] + rem \* (long long) BASE;

a[i] = (int) (cur / v);

rem = (int) (cur % v);

}

trim();

}

BigInt operator/(int v) const {

assert(v > 0); // operator / not well-defined for v <= 0.

if (llabs(v) >= BASE) {

return \*this / BigInt(v);

}

BigInt res = \*this;

res /= v;

return res;

}

void operator/=(const BigInt &v) {

\*this = \*this / v;

}

long long operator%(long long v) const {

assert(v > 0); // operator / not well-defined for v <= 0.

assert(v < BASE);

int m = 0;

for (int i = a.size() - 1; i >= 0; --i)

m = (a[i] + m \* (long long) BASE) % v;

return m \* sign;

}

void operator\*=(int v) {

if (llabs(v) >= BASE) {

\*this \*= BigInt(v);

return ;

}

if (v < 0)

sign = -sign, v = -v;

for (int i = 0, carry = 0; i < (int) a.size() || carry; ++i) {

if (i == (int) a.size())

a.push\_back(0);

long long cur = a[i] \* (long long) v + carry;

carry = (int) (cur / BASE);

a[i] = (int) (cur % BASE);

//asm("divl %%ecx" : "=a"(carry), "=d"(a[i]) : "A"(cur), "c"(base));

/\*

int val;

\_\_asm {

lea esi, cur

mov eax, [esi]

mov edx, [esi+4]

mov ecx, base

div ecx

mov carry, eax

mov val, edx;

}

a[i] = val;

\*/

}

trim();

}

BigInt operator\*(int v) const {

if (llabs(v) >= BASE) {

return \*this \* BigInt(v);

}

BigInt res = \*this;

res \*= v;

return res;

}

// Convert BASE 10^old --> 10^new.

static vector<int> convert\_base(const vector<int> &a, int old\_digits, int new\_digits) {

vector<long long> p(max(old\_digits, new\_digits) + 1);

p[0] = 1;

for (int i = 1; i < (int) p.size(); i++)

p[i] = p[i - 1] \* 10;

vector<int> res;

long long cur = 0;

int cur\_digits = 0;

for (int i = 0; i < (int) a.size(); i++) {

cur += a[i] \* p[cur\_digits];

cur\_digits += old\_digits;

while (cur\_digits >= new\_digits) {

res.push\_back((long long)(cur % p[new\_digits]));

cur /= p[new\_digits];

cur\_digits -= new\_digits;

}

}

res.push\_back((int) cur);

while (!res.empty() && !res.back())

res.pop\_back();

return res;

}

void fft(vector<complex<double> > & a, bool invert) const {

int n = (int) a.size();

for (int i = 1, j = 0; i < n; ++i) {

int bit = n >> 1;

for (; j >= bit; bit >>= 1)

j -= bit;

j += bit;

if (i < j)

swap(a[i], a[j]);

}

for (int len = 2; len <= n; len <<= 1) {

double ang = 2 \* 3.14159265358979323846 / len \* (invert ? -1 : 1);

complex<double> wlen(cos(ang), sin(ang));

for (int i = 0; i < n; i += len) {

complex<double> w(1);

for (int j = 0; j < len / 2; ++j) {

complex<double> u = a[i + j];

complex<double> v = a[i + j + len / 2] \* w;

a[i + j] = u + v;

a[i + j + len / 2] = u - v;

w \*= wlen;

}

}

}

if (invert)

for (int i = 0; i < n; ++i)

a[i] /= n;

}

void multiply\_fft(const vector<int> &a, const vector<int> &b, vector<int> &res) const {

vector<complex<double> > fa(a.begin(), a.end());

vector<complex<double> > fb(b.begin(), b.end());

int n = 1;

while (n < (int) max(a.size(), b.size()))

n <<= 1;

n <<= 1;

fa.resize(n);

fb.resize(n);

fft(fa, false);

fft(fb, false);

for (int i = 0; i < n; ++i)

fa[i] \*= fb[i];

fft(fa, true);

res.resize(n);

long long carry = 0;

for (int i = 0; i < n; ++i) {

long long t = (long long) (fa[i].real() + 0.5) + carry;

carry = t / 1000;

res[i] = t % 1000;

}

}

BigInt mul\_simple(const BigInt &v) const {

BigInt res;

res.sign = sign \* v.sign;

res.a.resize(a.size() + v.a.size());

for (int i = 0; i < (int) a.size(); ++i)

if (a[i])

for (int j = 0, carry = 0; j < (int) v.a.size() || carry; ++j) {

long long cur = res.a[i + j] + (long long) a[i] \* (j < (int) v.a.size() ? v.a[j] : 0) + carry;

carry = (int) (cur / BASE);

res.a[i + j] = (int) (cur % BASE);

}

res.trim();

return res;

}

typedef vector<long long> vll;

static vll karatsubaMultiply(const vll &a, const vll &b) {

int n = a.size();

vll res(n + n);

if (n <= 32) {

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

res[i + j] += a[i] \* b[j];

return res;

}

int k = n >> 1;

vll a1(a.begin(), a.begin() + k);

vll a2(a.begin() + k, a.end());

vll b1(b.begin(), b.begin() + k);

vll b2(b.begin() + k, b.end());

vll a1b1 = karatsubaMultiply(a1, b1);

vll a2b2 = karatsubaMultiply(a2, b2);

for (int i = 0; i < k; i++)

a2[i] += a1[i];

for (int i = 0; i < k; i++)

b2[i] += b1[i];

vll r = karatsubaMultiply(a2, b2);

for (int i = 0; i < (int) a1b1.size(); i++)

r[i] -= a1b1[i];

for (int i = 0; i < (int) a2b2.size(); i++)

r[i] -= a2b2[i];

for (int i = 0; i < (int) r.size(); i++)

res[i + k] += r[i];

for (int i = 0; i < (int) a1b1.size(); i++)

res[i] += a1b1[i];

for (int i = 0; i < (int) a2b2.size(); i++)

res[i + n] += a2b2[i];

return res;

}

BigInt mul\_karatsuba(const BigInt &v) const {

vector<int> a6 = convert\_base(this->a, BASE\_DIGITS, 6);

vector<int> b6 = convert\_base(v.a, BASE\_DIGITS, 6);

vll a(a6.begin(), a6.end());

vll b(b6.begin(), b6.end());

while (a.size() < b.size())

a.push\_back(0);

while (b.size() < a.size())

b.push\_back(0);

while (a.size() & (a.size() - 1))

a.push\_back(0), b.push\_back(0);

vll c = karatsubaMultiply(a, b);

BigInt res;

res.sign = sign \* v.sign;

long long carry = 0;

for (int i = 0; i < (int) c.size(); i++) {

long long cur = c[i] + carry;

res.a.push\_back((int) (cur % 1000000));

carry = cur / 1000000;

}

res.a = convert\_base(res.a, 6, BASE\_DIGITS);

res.trim();

return res;

}

void operator\*=(const BigInt &v) {

\*this = \*this \* v;

}

BigInt operator\*(const BigInt &v) const {

if (a.size() \* v.a.size() <= 1000111) return mul\_simple(v);

if (a.size() > 500111 || v.a.size() > 500111) return mul\_fft(v);

return mul\_karatsuba(v);

}

BigInt mul\_fft(const BigInt& v) const {

BigInt res;

res.sign = sign \* v.sign;

multiply\_fft(convert\_base(a, BASE\_DIGITS, 3), convert\_base(v.a, BASE\_DIGITS, 3), res.a);

res.a = convert\_base(res.a, 3, BASE\_DIGITS);

res.trim();

return res;

}

// -------------------- Misc --------------------

BigInt abs() const {

BigInt res = \*this;

res.sign \*= res.sign;

return res;

}

void trim() {

while (!a.empty() && !a.back())

a.pop\_back();

if (a.empty())

sign = 1;

}

bool isZero() const {

return a.empty() || (a.size() == 1 && !a[0]);

}

friend BigInt gcd(const BigInt &a, const BigInt &b) {

return b.isZero() ? a : gcd(b, a % b);

}

friend BigInt lcm(const BigInt &a, const BigInt &b) {

return a / gcd(a, b) \* b;

}

friend BigInt sqrt(const BigInt &a1) {

BigInt a = a1;

while (a.a.empty() || a.a.size() % 2 == 1)

a.a.push\_back(0);

int n = a.a.size();

int firstDigit = (int) sqrt((double) a.a[n - 1] \* BASE + a.a[n - 2]);

int norm = BASE / (firstDigit + 1);

a \*= norm;

a \*= norm;

while (a.a.empty() || a.a.size() % 2 == 1)

a.a.push\_back(0);

BigInt r = (long long) a.a[n - 1] \* BASE + a.a[n - 2];

firstDigit = (int) sqrt((double) a.a[n - 1] \* BASE + a.a[n - 2]);

int q = firstDigit;

BigInt res;

for(int j = n / 2 - 1; j >= 0; j--) {

for(; ; --q) {

BigInt r1 = (r - (res \* 2 \* BigInt(BASE) + q) \* q) \* BigInt(BASE) \* BigInt(BASE) + (j > 0 ? (long long) a.a[2 \* j - 1] \* BASE + a.a[2 \* j - 2] : 0);

if (r1 >= 0) {

r = r1;

break;

}

}

res \*= BASE;

res += q;

if (j > 0) {

int d1 = res.a.size() + 2 < r.a.size() ? r.a[res.a.size() + 2] : 0;

int d2 = res.a.size() + 1 < r.a.size() ? r.a[res.a.size() + 1] : 0;

int d3 = res.a.size() < r.a.size() ? r.a[res.a.size()] : 0;

q = ((long long) d1 \* BASE \* BASE + (long long) d2 \* BASE + d3) / (firstDigit \* 2);

}

}

res.trim();

return res / norm;

}

};