#pragma once

#include"polynomial.h"

template<typename P>polynomial<P> polynomial<P>::operator>>(const uint64\_t power)const

{

polynomial result;result.newsize(deg + power);

for (uint64\_t i = 0; i < deg; i++)

{

result.ptr[i + power] = ptr[i];

}

return result;

}

template<typename P>polynomial<P> polynomial<P>::operator<<(const uint64\_t power)const

{

polynomial result; result.newsiz(deg - power);

for (uint64\_t i = 0; i <= (deg - power); i++)

{

result.ptr[i] = ptr[i + power];

}

return result;

}

template<typename P>bool polynomial<P>::operator==(const polynomial<P>& other)const

{

if (deg != other.deg)return false;

else

{

for (uint64\_t i = 0; i < deg; i++)

{

if (ptr[i] != other.ptr[i])return false;

}

}

return true;

}

template<typename P>bool polynomial<P>::operator==(int64\_t zero)const

{

if (zero != 0) {

return 0;

}

for (uint64\_t i = 0; i < this->deg; i++)

{

if (ptr[i] != 0)

{

return 0;

}

}

return 1;

}

template<typename P>bool polynomial<P>::operator!=(const polynomial<P>& other)const

{

if (deg != other.deg)return true;

else

{

for (uint64\_t i = 0; i < deg; i++)

{

if (ptr[i] != other.ptr[i])return true;

}

}

return false;

}

template<typename P>polynomial<P>::polynomial(P number) {

outm\_E = default\_output\_mode;

deg = 1;

ptr = new P[deg];

ptr[0] = number;

}

template<typename P>void polynomial<P>::newsize(uint64\_t size)

{

deg = size;

if (size) {

delete[] ptr;

ptr = new P[size];

}

else

ptr = nullptr;

for (uint64\_t i = 0; i < size; i++)

ptr[i] = 0;

}

template<typename P>polynomial<P>::polynomial(const polynomial<P>& other)

{

outm\_E = default\_output\_mode;

deg = other.deg;

ptr = new P[deg + 1];

for (uint64\_t i = 0; i < deg; ++i)

{

ptr[i] = other.ptr[i];

}

}

template<typename P>polynomial<P>::~polynomial()

{

delete[] ptr;

}

template<typename P>polynomial<P>& polynomial<P>::operator=(const P other)

{

delete[] ptr;

deg = 1;

ptr = new P[deg];

ptr[0] = other;

return \*this;

}

template<typename P>polynomial<P>& polynomial<P>::operator=(const polynomial<P>& other)

{

if (this != &other)

{

delete[] ptr;

deg = other.deg;

ptr = new P[deg];

for (uint64\_t i = 0; i < deg; ++i)

{

ptr[i] = other.ptr[i];

}

}

return \*this;

}

template<typename P>std::istream& operator>>(std::istream& in, polynomial<P>& plnm)

{

uint64\_t new\_size;

in >> new\_size;

polynomial<P> temp; temp.newsize(new\_size);

for (uint64\_t i = 0; i < new\_size; ++i)

{

in >> temp.ptr[i];

}

plnm = temp;

return in;

}

template<typename P>void polynomial<P>::output\_mode\_set(std::string newmode) {

switch (newmode)

{

case "FULL": outm\_E = output\_mode::FULL; break;

case "ABBREVIATED":outm\_E = output\_mode::ABBREVIATED;break;

case "SHORT": outm\_E = output\_mode::SHORT; break;

default:break;

}

}

template<typename P>void polynomial<P>::output\_mode\_set(uint64\_t newmode) {

switch (newmode)

{

case 0: outm\_E = output\_mode::FULL; break;

case 1: outm\_E = output\_mode::ABBREVIATED; break;

case 2: outm\_E = output\_mode::SHORT; break;

default:break;

}

}

template<typename P>void polynomial<P>::output\_mode\_set(output\_mode new\_outm\_E) {

switch (new\_outm\_E)

{

case output\_mode::FULL: outm\_E = output\_mode::FULL; break;

case output\_mode::ABBREVIATED: outm\_E = output\_mode::ABBREVIATED; break;

case output\_mode::SHORT: outm\_E = output\_mode::SHORT; break;

default:break;

}

}

template<typename P>std::ostream& operator<<(std::ostream& out, const polynomial<P>& plnm)

{

if (output\_mode::FULL == plnm.outm\_E) {

out << "Degree: " << plnm.deg << ", Coefficients: ";

}

for (uint64\_t i = 0; i < plnm.deg; ++i)

{

if (plnm.ptr[i] < 0 && (output\_mode::FULL == plnm.outm\_E || output\_mode::ABBREVIATED == plnm.outm\_E)) {

out << "(" << plnm.ptr[i] << ")";

}

else {

out << plnm.ptr[i];

}

if (i == 1)

{

out << "x";

}

else if (i > 1) {

out << "x^" << i;

}

if (1+i< plnm.deg) {

if (output\_mode::FULL == plnm.outm\_E) {

out << " + ";

}

else if (output\_mode::ABBREVIATED == plnm.outm\_E) {

out << "+";

}

else if (output\_mode::SHORT == plnm.outm\_E)

{

out << " ";

}

}

}

if (plnm.deg == 0)

{

out << "0";

}

return out;

}

template<typename P>polynomial<P> polynomial<P>::operator+(const polynomial<P>& other) const

{

uint64\_t new\_size = std::max(deg, other.deg);

polynomial<P> result; result.newsize(new\_size);

for (uint64\_t i = 0; i < new\_size; i++)

{

if (i < deg && i < other.deg)

{

result.ptr[i] = ptr[i] + other.ptr[i];

}

else if (i < deg)

{

result.ptr[i] = ptr[i];

}

else if (i < other.deg)

{

result.ptr[i] = other.ptr[i];

}

}

return result;

}

template<typename P>polynomial<P> polynomial<P>::operator-(const polynomial<P>& other) const

{

uint64\_t new\_size = std::max(deg, other.deg);

polynomial<P> result; result.newsize(new\_size);

for (uint64\_t i = 0; i < new\_size; i++)

{

if (i < deg && i < other.deg)

{

result.ptr[i] = ptr[i] - other.ptr[i];

}

else if (i < deg)

{

result.ptr[i] = ptr[i];

}

else if (i < other.deg)

{

result.ptr[i] = -other.ptr[i];

}

}

return result;

}

template<typename P>polynomial<P> polynomial<P>::operator\*(const polynomial<P>& other)const

{

uint64\_t new\_size = deg + other.deg;

polynomial<P> result; result.newsize(new\_size);

for (uint64\_t i = 0; i < deg; i++)

{

for (uint64\_t j = 0; j < other.deg; j++)

{

result.ptr[i + j] = result.ptr[i + j] + ptr[i] \* other.ptr[j];

}

}

return result.cutbag();

}

template<typename P>polynomial<P> polynomial<P>::operator\*(const P& other)const

{

polynomial<P> result; result.newsize(deg);

for (uint64\_t i = 0; i < deg; i++)

{

result.ptr[i] = ptr[i] \* other;

}

return result;

}

template<typename P>polynomial<P> polynomial<P>::operator/(const polynomial<P>& other)const

{

polynomial<P> result(1), ans;

if (other == 0) {

throw std::exception("parametr 2 is zero");

}

if (deg < other.deg)

{

ans = 0;

}

else if ((\*this) == other) {

ans = 1;

return ans;

}

else

{

result = (\*this);

uint64\_t difference = deg - other.deg;

for (int64\_t i = difference; i >= 0; i--) {

ans = ans >> 1;

ans.ptr[0] = (((result.ptr[other.deg + i - 1] / other.ptr[other.deg - 1])));

result = result - (other >> i) \* (ans.ptr[0]);

}

}

return ans.cutbag();

}

template<typename P>polynomial<P> polynomial<P>::operator%(const polynomial<P>& other)const

{

polynomial<P> result(1), ans;

if (other == 0) {

result = (\*this);

}

else if (deg < other.deg)

{

result.ptr[0] = 0;

}

else

{

result = (\*this);

uint64\_t difference = deg - other.deg;

for (int64\_t i = difference; i >= 0; i--) {

ans = ans >> 1;

ans.ptr[0] = (((result.ptr[other.deg + i - 1] / other.ptr[other.deg - 1])));

result = result - (other >> i) \* (ans.ptr[0]);

}

}

return result.cutbag();

}

template<typename P>bool polynomial<P>::operator>(const polynomial<P>& other)const

{

return this->deg > other.deg;

}

template<typename P>bool polynomial<P>::operator<(const polynomial<P>& other)const

{

return this->deg < other.deg;

}

template<typename P>P& polynomial<P>::operator[](uint64\_t index) {

if (index >= deg) {

throw std::out\_of\_range("Index out of range");

}

return ptr[index];

}

template<typename P>polynomial<P> polynomial<P>::cutbag() const {

uint64\_t new\_deg = deg;

while (new\_deg > 0 && ptr[new\_deg - 1] == 0) {

new\_deg--;

}

polynomial<P> new\_poly;

new\_poly.newsize(new\_deg);

std::copy(ptr, ptr + new\_deg, new\_poly.ptr);

return new\_poly;

}