namespace NumRepr {

using namespace type\_traits;

using namespace auxiliary\_types;

template<allowable\_base\_type\_c UINT\_T, UINT\_T B>

requires (suitable\_base<UINT\_T,B>())

struct dig\_t {

private:

UINT\_T m\_d;

public:

// SIG\_UINT\_T(uchint) -> usint

using SIG\_UINT\_T = typename sig\_UInt\_for\_UInt\_t<UINT\_T>;

// SIG\_SINT\_T(uchint) -> ssint

using SIG\_SINT\_T = typename sig\_SInt\_for\_UInt\_t<UINT\_T>;

using uintspair = std::array<UINT\_T,2>;

template<binop\_e op>

using resbinop\_t = typename resbinop\_t<dig\_t,op>;

explicit operator UINT\_T() const;

const UINT\_T& get() const;

explicit operator SIG\_UINT\_T() const;

UINT\_T operator()() const;

static consteval bool is\_prime();

/////////////////////////////////////

static consteval dig\_t dig\_max();

static consteval dig\_t dig\_submax();

static consteval dig\_t dig\_Bm1();

static consteval dig\_t dig\_Bm2();

static consteval dig\_t dig\_0();

static consteval dig\_t dig\_1();

/////////////////////////////////////

static consteval UINT\_T ui\_max();

static consteval UINT\_T ui\_submax();

static consteval UINT\_T ui\_Bm1();

static consteval UINT\_T ui\_Bm2();

static consteval UINT\_T ui\_0();

static consteval UINT\_T ui\_1();

/////////////////////////////////////

static consteval SIG\_UINT\_T sui\_B();

static consteval SIG\_UINT\_T sui\_max();

static consteval SIG\_UINT\_T sui\_submax();

static consteval SIG\_UINT\_T sui\_0();

static consteval SIG\_UINT\_T sui\_1();

/////////////////////////////////////

static consteval SIG\_SINT\_T ssi\_B();

static consteval SIG\_SINT\_T ssi\_max();

static consteval SIG\_UINT\_T ssi\_submax();

static consteval SIG\_UINT\_T ssi\_0();

static consteval SIG\_UINT\_T ssi\_1();

template< integral\_c Int\_t>

static UINT\_T normaliza(Int\_t);

///< CONSTRUCTORES

consteval dig\_t() ;

template< integral\_c Int\_t>

dig\_t(Int\_t);

dig\_t(const dig\_t&) = default;

dig\_t(dig\_t &&) = default;

/// ASIGNACIONES OPERATOR=()

template< integral\_c Int\_t>

const dig\_t & operator = (const Int\_t &);

dig\_t & operator = (const dig\_t &) = default;

dig\_t & operator = (dig\_t &&) = default;

/// FUNCIONES PARA CONOCER EL CARRY

static dig\_t sum\_carry(dig\_t arg\_1,dig\_t arg\_2);

/// OPERADORES & &= | |=

/// FUNCIONAN COMO MAX Y MIN

dig\_t operator & (const dig\_t &) const;

const dig\_t & operator &= (dig\_t);

dig\_t operator | (const dig\_t &) const;

const dig\_t & operator |= (dig\_t);

/// OPERADORES \*B^n \*^=B^n

/// FUNCIONAN COMO Power(\*,n) y n = Power(\*,n)

/// DONDE n ES NATURAL

template<unsigned\_integral\_c UIntType>

const dig\_t & operator ^= (UIntType);

template<unsigned\_integral\_c UIntType>

dig\_t operator ^ (UIntType) const;

/// OPERADORES COMPARACION

bool operator == (dig\_t) const;

bool operator != (dig\_t) const;

bool operator >= (dig\_t) const;

bool operator > (dig\_t) const;

bool operator <= (dig\_t) const;

bool operator < (dig\_t) const;

std::strong\_ordering operator <=> (dig\_t) const;

template<type\_traits::integral\_c Int\_t>

bool operator == (Int\_t) const;

template<integral\_c Int\_t>

bool operator != (Int\_t) const;

template<integral\_c Int\_t>

bool operator >= (Int\_t) const;

template<integral\_c Int\_t>

bool operator > (Int\_t) const;

template<integral\_c Int\_t>

bool operator <= (Int\_t) const;

template<integral\_c Int\_t>

bool operator < (Int\_t) const;

template<integral\_c Int\_t>

std::weak\_ordering operator <=> (Int\_t) const;

/// ARITMETICOS CON ASIGNACION

const dig\_t & operator +=(dig\_t arg);

template<integral\_c Int\_t>

const dig\_t & operator +=(Int\_t arg);

const dig\_t & operator -=(dig\_t arg);

template<integral\_c Int\_t>

const dig\_t & operator -=(Int\_t arg);

const dig\_t & operator \*=(dig\_t arg);

template<integral\_c Int\_t>

const dig\_t & operator \*=(Int\_t arg);

const dig\_t & operator /=(dig\_t arg);

template< integral\_c Int\_t>

const dig\_t & operator /=(Int\_t arg);

const dig\_t & operator %=(dig\_t arg);

template<integral\_c Int\_t>

const dig\_t & operator %=(Int\_t arg);

/// PRE Y POST

/// CIRCULARES

const dig\_t& operator ++ ();

dig\_t operator ++ (int);

const dig\_t& operator -- ();

dig\_t operator -- (int);

/// OPERADORES ARITMETICOS

dig\_t operator + (dig\_t) const;

dig\_t operator - (dig\_t) const;

dig\_t operator \* (dig\_t) const;

dig\_t operator / (dig\_t) const;

dig\_t operator % (dig\_t) const;

template<integral\_c Int\_type>

dig\_t operator + (Int\_type arg) const;

template<integral\_c Int\_type>

dig\_t operator - (Int\_type arg) const;

template<integral\_c Int\_type>

dig\_t operator \* (Int\_type arg) const;

template<integral\_c Int\_type>

dig\_t operator / (Int\_type arg) const;

template<integral\_c Int\_type>

dig\_t operator % (Int\_type) const;

/// COMPLEMENTO BASE

/// Y BASE MENOS 1

dig\_t operator ! () const;

dig\_t operator - () const;

dig\_t C\_Bm1 () const;

dig\_t C\_B () const;

const dig\_t & mC\_Bm1 ();

const dig\_t & mC\_B ();

/// NULO Y MAXIMO

bool is\_0 () const;

bool is\_1 () const;

bool is\_0or1 () const;

bool is\_not\_1 () const;

bool is\_not\_0 () const;

bool is\_not\_0or1 () const;

bool is\_max () const;

bool is\_not\_max () const;

bool is\_Bm1 () const;

bool is\_not\_Bm1 () const;

bool is\_submax() const;

bool is\_maxorsubmax() const;

bool is\_Bm1orBm2() const;

bool is\_not\_maxorsubmax() const;

bool is\_not\_Bm1orBm2() const;

bool is\_not\_submax() const;

bool is\_Bm2() const;

bool is\_not\_Bm2() const;

bool is\_not\_maxormin() const;

bool is\_maxormin() const;

bool is\_far\_maxormin() const;

bool is\_near\_maxormin() const;

/// VARIOS CASTS

/// TIENE QUE DEVOLVER STD::STRING

private:

std::string num\_to\_string() const;

static std::string radix\_str();

public:

std::string to\_string() const;

/// TODO : CREAR UN LITERAL DESDE EL QUE CREAR UN DIGITO

};

///< DEFINCION DE template<uint128\_t Radix> digit\_t{};

template<uint128\_t B>

using digit\_t =

dig\_t<

TypeFromIntNumber\_t<B>,

static\_cast<TypeFromIntNumber\_t<B>>(B)

>;

/// ISTREAM Y OSTREAM

template<allowable\_base\_type\_c UINT\_T,UINT\_T Base>

requires (suitable\_base<UINT\_T,Base>())

std::istream & operator >> (std::istream &,dig\_t<UINT\_T,Base> &);

template<allowable\_base\_type\_c UINT\_T,UINT\_T Base>

requires (suitable\_base<UINT\_T,Base>())

std::ostream & operator << (std::ostream &,dig\_t<UINT\_T,Base>);

}

}