```
namespace NumRepr {

template<type_traits::allowable_base_type_c UINT_T, UINT_T B>
    requires (type_traits::suitable_base<UINT_T,B>())
struct dig_t {
private:
    UINT_T m_d;
public:
    // SIG_UINT_T(uchint) -> usint
    using SIG_UINT_T        = typename type_traits::sig_UInt_for_UInt_t<UINT_T>;
    // SIG_SINT_T(uchint) -> ssint
    using SIG_SINT_T        = typename type_traits::sig_SInt_for_UInt_t<UINT_T>;

    using uintspair         = std::array<UINT_T,2>;

    template<binop_e op>
    using resbinop_t        = typename auxiliary_types::resbinop_t<dig_t,op>;

public:

    explicit operator UINT_T() const;
    const UINT_T& get() const;
    explicit operator SIG_UINT_T() const;
    UINT_T operator()() const;

public:
    ////////////////////////////////////
    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();
    ////////////////////////////////////


public:

    template<type_traits::integral_c Int_t>
    static UINT_T normaliza(Int_t);
    ///< CONSTRUCTORES
    consteval dig_t() ;
    template<type_traits::integral_c Int_t>
    dig_t(Int_t);
    dig_t(const dig_t&)  = default;
    dig_t(dig_t &&)  = default;

    /// ASIGNACIONES OPERATOR=()
    template<type_traits::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<type_traits::unsigned_integral_c UIntType>
const dig_t & operator ^= (UIntType);

template<type_traits::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<type_traits::integral_c Int_t>
bool operator != (Int_t) const;
template<type_traits::integral_c Int_t>
bool operator >= (Int_t) const;
template<type_traits::integral_c Int_t>
bool operator >  (Int_t) const;
template<type_traits::integral_c Int_t>
bool operator <= (Int_t) const;
template<type_traits::integral_c Int_t>
bool operator <  (Int_t) const;

template<type_traits::integral_c Int_t>
std::weak_ordering operator <=> (Int_t) const;

///   ARITMETICOS CON ASIGNACION

 const dig_t & operator +=(dig_t arg);

template<type_traits::integral_c Int_t>
 const dig_t & operator +=(Int_t arg);
 const dig_t & operator -=(dig_t arg);

template<type_traits::integral_c Int_t>
 const dig_t & operator -=(Int_t arg);
 const dig_t & operator *=(dig_t arg);
template<type_traits::integral_c Int_t>
 const dig_t & operator *=(Int_t arg);
 const dig_t & operator /=(dig_t arg);
template<type_traits::integral_c Int_t>
 const dig_t & operator /=(Int_t arg);
 const dig_t & operator %=(dig_t arg);
template<type_traits::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<type_traits::integral_c Int_type>
    dig_t operator + (Int_type arg) const;

    template<type_traits::integral_c Int_type>
    dig_t operator - (Int_type arg) const;

    template<type_traits::integral_c Int_type>
    dig_t operator * (Int_type arg) const;

    template<type_traits::integral_c Int_type>
    dig_t operator / (Int_type arg) const;

    template<type_traits::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
};

///< DEFINICION DE template<uint128_t Radix> digito_t{};
template<uint128_t B>
using digit_t =
        dig_t<
            type_traits::TypeFromIntNumber_t<B>,
            static_cast<type_traits::TypeFromIntNumber_t<B>>(B)
        >;
```

```
/// ISTREAM Y OSTREAM

/// TODO :
///             ESTA VERSION +
///             VERSION CON TRATAMIENTO DE ERRORES RUNTIME
template<type_traits::allowable_base_type_c UINT_T,UINT_T Base>
    requires (type_traits::suitable_base<UINT_T,Base>())
std::istream & operator >> (std::istream &,dig_t<UINT_T,Base> &);
template<type_traits::allowable_base_type_c UINT_T,UINT_T Base>
    requires (type_traits::suitable_base<UINT_T,Base>())
std::ostream & operator << (std::ostream &,dig_t<UINT_T,Base>);
}
```