```
let base = 4;;

type nat = int list;;
type z = {signe: int; nat: nat};;
type dya = { m : z ; e : int } ;;
type ldb = {
lg : int ; g : dya list ;
ld : int ; d : dya list };;

(*Ex1 ~*)
let cons_nat (c :int) (n :nat) :nat =
let rec convert c b =
(convert (c mod b) b)@[c / b]
in
let rec add (n1 :nat) (n2 :nat) :nat =
match n1,n2 with
| [],[] -> []
| [],x::xs -> (x)::(add [] xs)
| x::xs, [] -> (x)::(add [] xs)
in
add (convert c base) n;;

(*Ex2*)
let add_nat (n1 :nat) (n2 :nat) :nat =
let rec add (n1 :nat) (n2 :nat) (r :int) =
match n1,n2 with
| [],[] -> if r = 0 then [] else [r]
| [],x::xs -> (x+r)::(add [] xs 0)
| x::xs, [] -> (x+r)::(add [] xs 0)
| x1::xs1 , x2::xs2 ->
if (x1 + x2 + r) > (base - 1)
then ((x1 + x2 + r) mod base):: (add xs1 xs2 ((x1 + x2 + r) / base))
else (x1 + x2 + r)::(add xs1 xs2 0)
in add n1 n2 0;;

(*Ex3*)
let cmp_nat (n1 :nat) (n2 :nat) :int =
let rec reverse (h :nat) (t :nat) (l :int)=
match t with
| [] -> (h,l)
| hd :: tl -> reverse (hd :: h) tl (l+1)
in
let rec compare (i :nat) (j :nat) =
match i,j with
| [], [] -> 0
| x1::xs1, x2::xs2 ->
if x1 > x2 then 1 else
if x2 > x1 then -1 else
compare xs1 xs2
in
let (a, l1) = reverse [] n1 0 in
```

```ocaml
let (b, l2) = reverse [] n2 0 in
if l1 > l2 then 1 else
if l2 > l1 then -1 else
compare a b;;

(*Ex4 ~*)
let sous_nat (n1 :nat) (n2 :nat) :nat =
let rec sous (n1 :nat) (n2 :nat) (r :int) =
match n1,n2 with
| [],[] -> []
| [],x::xs -> failwith "n2 > n1"
| x::xs, [] -> if (x+r) < 0 then (base + x +r)::(sous xs [] 0) else (x +r)::(sous xs [] 0)
| x1::xs1 , x2::xs2 ->
if (x1 - x2 + r) < 0
then (base + x1 - x2 + r):: (sous xs1 xs2 1)
else (x1 - x2 + r)::(sous xs1 xs2 0)
in
sous n1 n2 0;;

(*Ex5*)
let div2_nat (n :nat) :(nat * int)=
let reste2 (n :nat ) =
match n with
| [] -> 0
| x::xs -> x mod 2
in
let rec reverse (h :nat) (t :nat) =
match t with
| [] -> (h)
| hd :: tl -> reverse (hd :: h) tl
in
let rec quotient2 (n :nat) (q :nat) (r :int) :nat =
match n with
| [] -> q
| x::xs ->
if (x mod 2) = 1 then quotient2 xs ((x/2 + r)::q) 5 else quotient2 xs ((x/2 + r)::q) 0
in
let a = reverse [] n in
((quotient2 a [] 0),(reste2 n));;

(*Ex6*)
let neg_z (n :z) :z =
{signe = (n.signe * -1) ; nat = n.nat};;

(*Ex7 -> problème lié à sous_nat*)
let add_z (n1 :z) (n2 :z) :z =
if(n1.signe = 1 && n2.signe = 1) then {signe = 1 ; nat = add_nat n1.nat n2.nat}
else if(n1.signe = 1 && n2.signe = -1) then {signe = (cmp_nat n1.nat n2.nat) ; nat =
sous_nat n1.nat n2.nat}
else if(n1.signe = -1 && n2.signe = 1) then {signe = (cmp_nat n2.nat n1.nat) ; nat =
sous_nat n2.nat n1.nat}
```

```
else {signe = -1 ; nat = add_nat n1.nat n2.nat};;

(*Ex8*)
let mul_puiss2_z (p :int) (z1 :z) :z =
let rec mult n s r=
match n with
| [] -> if r > 0 then [r] else []
| x::xs ->
if (s * x + r) > (base -1) then (((s * x + r) mod base))::(mult xs s ((s * x + r) / base))
else (s * x + r)::(mult xs s 0)
in
let s = int_of_float(2. ** float_of_int(p)) in
{signe = z1.signe ; nat = (mult z1.nat s 0)};;

(*Ex9 -> problème sur fonction "puissance2"*)
let decomp_puiss2_z z =
let rec puissance2 (z1 :nat) (n :int) :(nat * int) =
let (a, b) = div2_nat z1 in
if ((cmp_nat a [0;0]) = 1) then (puissance2 a n+1)
else (a,n)
in
let c,d = puissance2 z.nat 0 in
({signe = z.signe ; nat = c},d);;

(*2 - Nombres dyadiques*)

(*Ex10*)
let div2_dya (d :dya) :dya =
{m = d.m ; e = d.e - 1};;

(*Ex11*)
let rec add_dya (d1 :dya) (d2 :dya) :dya =
if(d1.e > d2.e) then {m = (add_z d2.m (mul_puiss2_z (d1.e - d2.e) d1.m)) ; e = d2.e}
else add_dya d2 d1;;

(*Ex12*)
let sous_dya (d1 :dya) (d2 :dya) :dya =
if(d1.e > d2.e) then {m = (add_z (mul_puiss2_z (d1.e - d2.e) (neg_z d1.m)) d2.m) ; e =
d2.e}
else {m = (add_z (mul_puiss2_z (d2.e - d1.e) (neg_z d1.m)) d2.m) ; e = d1.e};;

(*Ex13*)
let ldb_est_vide (l :ldb) :bool =
if(l.ld = 0 && l.lg = 0) then true
else false;;

(*Ex14*)
let premier_g (l :ldb) :dya =
let rec reverse (h :dya list) (t :dya list) :dya list =
match t with
| [] -> h
```

```
| hd :: tl -> reverse (hd :: h) tl
in
match l.g with
| x::xs -> x
| _ -> (
l.g = reverse [] l.d;
l.ld = 0;
match l.g with
| x::xs -> x
| _ -> failwith "ldb vide"
);;

(*Ex15*)
let inverse_ldb (l :ldb) :ldb =
let rec reverse (h :dya list) (t :dya list) :dya list =
match t with
| [] -> h
| hd :: tl -> reverse (hd :: h) tl
in
{lg = l.ld ; g = (reverse [] l.d) ; ld = l.lg ; d = (reverse [] l.g)};;

(*Ex16*)

(*Ex17*)
let ajoute_g (d :dya) (ldb1 :ldb) :ldb =
{lg = ldb1.lg ; g = d::ldb1.g ; ld = ldb1.ld ; d = ldb1.d};;

(*Ex18*)
let enleve_g (d :dya) (ldb1 :ldb) :ldb =
let rm list =
match list with
| x::xs -> xs
in
invariant_ldb;
{lg = ldb1.lg ; g = (rm ldb1.g) ; ld = ldb1.ld ; d = ldb1.d};;

(*Ex19*)
(*
On suppose que c = 3.
Alors, une opération sur 3 nécessite un "invariant_ldb".
On a donc (1/3)*(3*n) = n opérations en moyenne
Les complexités de "enleve_g" et "ajoute_g" sont quand à elles similaires, et dépendent de
n.
Leur complexité est donc en n²
*)
```