```
(* ricorsione e iterazione (fattoriale e prodotto) - iterazione (mcd) *)
let rec fatt n =
    match n with
    0 -> 1
    | _ when n > 0 -> n * fatt (n - 1)
    | _ -> invalid_arg "argomento negativo";;

let fatt n =
    let rec aux acc n =
        match n with
        0 -> acc
        | _ when n > 0 -> aux (n * acc) (n - 1)
        | _ -> invalid_arg "argomento negativo"
    in
    aux 1 n;;


let rec prodotto x y =
    match y with
    0 -> 0
    | _ when y > 0 -> x + prodotto x (y - 1)
    | _ -> invalid_arg "y negativo";; (* per potenza usare 0 -> 1 e "x * " *)

let prodotto x y =
    let rec aux acc y =
        match y with
        0 -> acc
        | _ when y > 0 -> aux (x + acc) (y - 1)
        | _ -> invalid_arg "y negativo"
    in
    aux 0 y;;


let rec mcd x y =
    if x <= 0 or y <= 0
        then invalid_arg "x e y devono essere > 0"
    else
        match x with
        _ when x = y -> x
        | _ when x > y -> mcd (x - y) y
        | _ -> mcd x (y -x);;


(* funzioni composte                    f             g          x       *)
let compose f g x = f (g x);;    (* ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b *)

let succ n = n + 1 and doppio n = n * 2
in compose succ doppio 2;;

(* funzioni curry e uncurry *)
let curry f x y = f(x, y);; (* ('a * 'b -> 'c) -> 'a -> 'b -> 'c *)

let uncurry f(x, y) = f x y;; (* ('a -> 'b -> 'c) -> 'a * 'b -> 'c *)


(* conta le cifre pari o dispari di un numero intero *)
let pari n =
    if n mod 2 = 0  (* per dispari usare n mod 2 = 1 *)
        then 1
        else 0;;
let rec conta_pari n =
    if -10 < n & n < 10
        then pari n
        else
            pari (n mod 10) + conta_pari (n / 10);;

conta_pari 12345;;
conta_pari (-12345);;
```

```ocaml
(* tipi *)
(* enumerati: tipi arbitrari *)
type num = Pari | Dispari;;

let tipo_num x =
    if (x mod 2) = 0
        then Pari
        else Dispari;;

tipo_num 3;;



(* somma: definiti da altri tipi, si appartiene ad un solo sotto tipo *)
type misura = Metri of int | Centimetri of int;;

(* recupare i tipi somma *)
let m_cm x =
    match x with
    Centimetri x -> Centimetri x
    | Metri x -> Centimetri (x * 100);;

m_cm (Metri 3);;
m_cm (Centimetri 300);;


type coordinate = Coord of int * int;;
let ascissa (Coord (x, y)) = x;;


(* prodotto: definiti da altri tipi, si appartiene a tutti i sotto tipi *)
type studente = {nome: string; cognome: string; matricola: int};;

(* recupare i tipi prodotto *)
let torna_matr x = x.matricola;;

let pinco = {nome = "pinco"; cognome = "pallino"; matricola = 888888};;

torna_matr pinco;;



(* ricorsivi: definiti anche da loro stessi *)
(* numeri naturali *)
type nat = Zero | Succ of nat;;

(* esempio numero 2 scritto come nat *)
Succ (Succ Zero);;
```

```ocaml
(* definire la concatenazione @ attraverso il costruttore ::  *)
let rec concat l1 l2 =
    match l1 with
    [] -> l2
    | x :: xs -> x :: concat xs l2;;

concat [3; 2] [1; 4; 5];;



(* rev, inverti l'ordine degli elementi di una lista *)
let rec rev l =
    match l with
    [] -> []
    | x :: xs -> concat (rev xs) [x];; (* = rev xs @ [x] *)

rev [1; 2; 3; 4];;



(* data una lista tenere solo i pari *)
let rec pari l =
    match l with
    [] -> []
    | x :: xs ->
        if (x mod 2) = 0
            then x :: pari xs
            else pari xs;;

pari [2; 6; 3; 1; 4];;
(* per contare i pari porre caso base 0 e " 1 + " al posto di " x :: " *)
(* per dispari (x mod 2) = 1 *)



(* restituisci l'ennesimo elemento di una lista *)
let rec nth n l =
    match l with
    [] -> failwith "impossibile trovare l'elemento"
    | x :: xs ->
        if (n = 0)
            then x
            else nth (n - 1) xs;;

nth 2 [1; 2; 3];;
nth 3 [1; 2; 3];;



(* verifica se la lista è ordinata (true) o no (false) *)
let rec in_order l =
    match l with
    [] -> true
    | [x] -> true
    | x :: y :: xs -> x <= y & in_order (y :: xs);;

in_order [1; 2; 3; 4; 5];;
in_order [1; 2; 3; 5; 4];;



(* costruisci una lista dei valori compresi tra x e y inclusi, x <= y *)
let rec intervallo x y =
    if (x = y)
      then [x]
      else x :: intervallo (x + 1) y;;

intervallo 4 7;;
```

```ocaml
(* da coppia di liste della stessa lunghezza a lista di coppie *)
let rec combine (l1, l2) =
    match (l1, l2) with
    (x :: xs, y :: ys) -> (x, y) :: combine (xs, ys)
    | (_, _) -> [];; (* caso base liste vuote *)

combine ([1; 3; 5], [2; 4; 6]);;

(* da lista di coppie a coppia di liste *)
let rec split l =
        match l with
        [] -> ([], [])
        | (x, y) :: ls ->
            let (xs, ys) = split ls
            in
            (x :: xs, y :: ys);;

split [(1, 2); (3, 4); (5, 6)];;



(* take, prendi i primi n elementi della lista *)
let rec take n l =
    match l with
    | [] -> []
    | _ when n = 0 -> []
    | x :: xs -> x :: take (n-1) xs;;

take 2 [3; 7; 5];;
take 4 [3; 7; 5];;

(* drop, salta i primi n elementi della lista *)
let rec drop n l =
    match l with
    | [] -> []
    | _ when n = 0 -> l
    | x :: xs -> drop (n-1) xs;;

drop 2 [3; 7; 5];;
drop 4 [3; 7; 5];;

(* NB take è ricorsiva, drop è iterativa *)



(* swap, scambia un elemento della lista con il suo successore
   (se esiste altrimenti lasciala così com'è) *)
let rec swap n l =
    match l with
    [] -> l
    | [x] -> l
    | x :: y :: xs when n = 0 -> y :: x :: xs
    | x :: xs -> x :: swap (n - 1) xs;;

swap 2 [3; 4; 7; 1];;
swap 3 [3; 4; 7; 1];;
swap 4 [3; 4; 7; 1];;



(* interleave, mescola alternando due liste *)
let rec interleave l1 l2 =
    match l1 with
    [] -> l2
    | x :: xs -> x :: interleave l2 xs;;

interleave [1; 2; 3; 4] [5; 6; 7];;
```

```
(* numero più piccolo della lista *)
let rec min_l l =
    match l with
    [] -> invalid_arg "lista vuota"
    | [x] -> x
    | x :: y :: xs ->
        if x <= y
            then min_l (x :: xs)
            else min_l (y :: xs);;

(* numero più grande della lista *)
let rec max_l l =
    match l with
    [] -> invalid_arg "lista vuota"
    | [x] -> x
    | x :: y :: xs ->
        if x >= y
            then max_l (x :: xs)
            else max_l (y :: xs);;

(* minimo tra i massimi di più liste *)
let rec min_max l =
    match l with
    [] -> invalid_arg "lista vuota"
    | [x] -> max_l x
    | x :: xs -> min_l (max_l x :: [min_max xs]);;

min_l [2; 3; 1; 7; 5];;
max_l [2; 3; 1; 7; 5];;
min_max [[2; 4; 1]; [3; 5; 7]; [3; 1; 2]];;



(* minimo tra i numeri positivi *)
let rec minp l =
    match l with
    [] -> invalid_arg "lista vuota"
    | [x] ->
        if x < 0
            then failwith "la lista non contiene numeri positivi"
            else x
    | x :: y :: xs when x <= y ->
        if x >= 0
            then minp (x :: xs)
            else minp (y :: xs)
    | x :: y :: xs ->
        if y >= 0
            then minp (y :: xs)
            else minp (x :: xs);;

minp [5; -1; -5; 0; 4; -2];;
minp [-5; -1; -5; -3; -4; -1];;



(* sottosequenze *)
let seq l =
    let rec aux acc l =
        match l with
        [] -> [acc]
        | x :: xs -> acc :: aux (acc @ [x]) xs
    in
    aux [] l;;

seq [1; 2; 3; 4];; (* [[]; [1]; [1; 2]; [1; 2; 3]; [1; 2; 3; 4]] *)
```

```
(* cerca se c'è un elemento x in una lista (almeno uno) *)
let rec mem n l =
    match l with
    [] -> false
    | x :: xs ->
        x = n or mem n xs;;

mem 3 [5; 1; 3; 7; 3];;
mem 4 [5; 1; 3; 7; 3];;



(* for_all, predicato vero per tutti gli elementi della lista *)
let rec for_all p l =
    match l with
    [] -> true
    | x :: xs -> p x & for_all p xs;;

for_all (fun x -> x > 0) [3; 5; 1];;
for_all (fun x -> x > 0) [3; -4; 1];;



(* exists, predicato vero per almeno un elemento della lista *)
let rec exists p l =
    match l with
    [] -> false
    | x :: xs -> p x or exists p xs;;

exists (fun x -> x > 0) [3; -5; -1];;
exists (fun x -> x > 0) [-3; -4; -1];;



(* map, applica una funzione a tutti gli elementi di una lista *)
let rec map f l =
    match l with
    [] -> []
    | x :: xs -> f x :: map f xs;;
(* ('a -> 'b) -> 'a list -> 'b list *)

map (fun x -> x + 1) [1; 2; 3];; (* funzione successore *)



(* filter, tieni gli elementi della lista che soddisfano un predicato *)
let rec filter p l =
    match l with
    [] -> []
    | x :: xs ->
        if p x
            then x :: filter p xs
            else filter p xs;;
(* ('a -> bool) -> 'a list -> 'a list *)

filter (fun x -> x > 2) [1; 2; 3];; (* 3 *)
filter ((<) 2) [1; 2; 3];; (* 2 < x, restituisce 3 > 2 *)
filter ((>) 2) [1; 2; 3];; (* 2 > x, restituisce 1 < 2 *)



(* togliere doppi *)
let rec togli_doppi l =
    match l with
    [] -> []
    | x :: xs -> x :: togli_doppi (filter ((<>) x) xs);;

togli_doppi [1; 2; 3; 3; 2; 1; 3];;
```

```ocaml
(* filtermap, tieni gli elementi della lista che soddisfano un predicato
              e applica a questi una funzione *)
let rec filtermap f p l =
    match l with
    [] -> []
    | x :: xs ->
        if p x
            then f x :: filtermap f p xs
            else filtermap f p xs;;
(* ('a -> 'b) -> ('a -> bool) -> 'a list -> 'b list *)

(* Data una lista di liste restituire i primi elementi delle liste non vuote *)
let firsts l =
    let hd l =
        match l with
        [] -> invalid_arg "lista vuota"
        | x :: xs -> x
    in
    filtermap hd ((<>) []) l;;

firsts [[1; 2]; []; [3; 4]; [5]; [6; 7; 8]; []];;


(* Ordinare una lista disordinata: ordinamento completo *)
let rec ordina l =
    let rec insord n l =
        match l with
        [] -> [n]
        | x :: xs ->
            if n <= x
                then n :: x :: xs
                else x :: insord n xs
    in
    match l with
    [] -> []
    | x :: xs -> insord x (ordina xs);;

ordina [2; 5; 3; 1; 7; 4; 6; 8];;


(* Ordinare una lista disordinata: ordinamento merge sort *)
let rec
oddpart l =
    match l with
    [] -> []
    | x :: xs -> x :: evenpart xs
and
evenpart l =
    match l with
    [] -> []
    | x :: xs -> oddpart xs;;

let rec merge l1 l2 =
    match (l1, l2) with
    ([], _) -> l2
    | (_, []) -> l1
    | (x :: xs, y :: ys) ->
        if (x <= y)
            then x :: merge xs l2
            else y :: merge l1 ys;;

let rec mergesort l =
    match l with
    [] -> []
    | [x] -> l
    | _ -> merge (mergesort (oddpart l)) (mergesort (evenpart l));;

mergesort [2; 5; 3; 1; 7; 4; 6; 8];;
```

```
(* calcola il numero di combinazioni (senza usare il fattoriale) *)
let combo l k =
    let rec aux n k =
        match k with
        0 -> 1
        | _ when n = k -> 1
        | _ -> aux (n - 1) (k - 1) + aux (n - 1) k
    in
    aux (List.length l) k;;

combo [1; 2; 3] 2;;



(* calcola le combinazioni di n elementi su k posti *)
let combo l k =
    if l = []
        then invalid_arg "lista_vuota"
    else
        let rec aux l acc =
            if List.length acc = k
                then [acc]
            else
                match l with
                [] -> []
                | x :: xs ->
                    aux xs (acc @ [x]) @ aux xs acc
        in
        aux l [];;

combo [1; 2; 3] 2;;



(* insieme delle parti di una lista *)
let rec powerset l =
    let rec aggiungi n l =
        match l with
        [] -> []
        | x :: xs -> (n :: x) :: aggiungi n xs
    in
    match l with
    | [] -> [[]]
    | x :: xs -> aggiungi x (powerset xs) @ powerset xs;;

powerset [1; 2; 3];;
```

```ocaml
(* alberi binari *)
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;

(* restituire albero sinistro *)
let left bt =
    match bt with
    Empty -> Empty
    | Node (x, lt, rt) -> lt;;
(* per albero destro usare rt *)


(* cercare elemento in un albero binario (almeno uno) *)
let rec search n bt =
    match bt with
    Empty -> false
    | Node (x, lt, rt) -> n = x or search n lt or search n rt;;

search 2 (Node (5, Node (2, Empty, Empty), Node (3, Empty, Empty)));;
search 1 (Node (5, Node (2, Empty, Empty), Node (3, Empty, Empty)));;


(* cercare elemento in un albero binario di ricerca (almeno uno) *)
let rec search n bt =
    match bt with
    | Empty -> false
    | Node (x, lt, rt) when n = x -> true
    | Node (x, lt, rt) when n < x -> search n lt
    | Node (x, lt, rt) -> search n rt;; (* n > x *)

search 2 (Node (5, Node (2, Empty, Empty), Node (6, Empty, Empty)));;
search 1 (Node (5, Node (2, Empty, Empty), Node (6, Empty, Empty)));;


(* contare i nodi di un albero *)
let rec count bt =
    match bt with
    Empty -> 0
    | Node (x, lt, rt) -> 1 + count lt + count rt;;

count (Node (5, Node (2, Empty, Empty), Node (3, Empty, Empty)));;


(* visita di un albero prefissa, infissa, postfissa *)
let rec preorder bt =
    match bt with
    Empty -> []
    | Node (x, lt, rt) -> x :: preorder lt @ preorder rt;;

let rec inorder bt =
    match bt with
    Empty -> []
    | Node (x, lt, rt) -> inorder lt @ x :: inorder rt;;

let rec postorder bt =
    match bt with
    Empty -> []
    | Node (x, lt, rt) -> postorder lt @ postorder rt @ [x];;

preorder (Node (1, Node (3, Node (2, Empty, Empty), Node (4, Empty,
         Node (5, Empty, Empty))), Node (6, Empty, Empty)));;

inorder (Node (1, Node (3, Node (2, Empty, Empty), Node (4, Empty,
         Node (5, Empty, Empty))), Node (6, Empty, Empty)));;

postorder (Node (1, Node (3, Node (2, Empty, Empty), Node (4,
           Empty, Node (5, Empty, Empty))), Node (6, Empty, Empty)));;
```

```ocaml
(* alberi binari *)
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;

(* altezza albero binario *)
let rec altezza bt =
    match bt with
    | Empty -> 0
    | Node (x, lt, rt) -> 1 + max (altezza lt) (altezza rt);;

altezza (Node(4, Node(2, Node(1, Empty, Empty), Empty), Node(5,
           Empty, Empty)));;

(* funzione predefinita max: let max x y = if x >= y then x else y *)
(* funzione predefinita min: let min x y = if x <= y then x else y *)



(* albero binario bilanciato: verifica *)
let rec bilanciato bt =
    match bt with
    | Empty -> true
    | Node (x, lt, rt) ->
        (-1) <= (altezza lt - altezza rt) & (altezza lt - altezza rt) <= 1
        & bilanciato lt & bilanciato rt;;



(* albero binario bilanciato completo: verifica *)
let rec completo bt =
    match bt with
    | Empty -> true
    | Node (x, lt, rt) ->
        (-1) <= (altezza lt - altezza rt) & (altezza lt - altezza rt) <= 1
        & completo lt & completo rt
        & ((lt = Empty & rt = Empty) or (lt <> Empty & rt <> Empty));;


(* Esempi: *)
(* albero bilanciato e completo: *)
bilanciato (Node(4, Node(2, Node(1, Empty, Empty), Node(3, Empty, Empty)),
           Node(5, Empty, Empty)));;

completo (Node(4, Node(2, Node(1, Empty, Empty), Node(3, Empty, Empty)),
          Node(5, Empty, Empty)));;

(* albero né bilanciato né completo: *)
bilanciato (Node(4, Node(2, Node(1, Empty, Empty), Node(3, Empty, Empty)),
           Empty));;

completo (Node(4, Node(2, Node(1, Empty, Empty), Node(3, Empty, Empty)),
          Empty));;

(* albero bilanciato ma non completo: *)
bilanciato (Node(4, Node(2, Node(1, Empty, Empty), Empty), Node(5,
           Empty, Empty)));;

completo (Node(4, Node(2, Node(1, Empty, Empty), Empty), Node(5,
          Empty, Empty)));;
```

```
(* alberi binari *)
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;

(* max e min di un albero binario di ricerca, (elementi più a dx e più a sx) *)
let rec max_bt bt =
    match bt with
    Empty -> invalid_arg "albero vuoto!"
    | Node (x, _, Empty) -> x
    | Node (x, lt, rt) -> max_bt rt;;

let rec min_bt bt =
    match bt with
    Empty -> invalid_arg "albero vuoto!"
    | Node (x, Empty, _) -> x
    | Node (x, lt, rt) -> min_bt lt;;

max_bt (Node (5, Node (2, Node (1, Empty, Empty), Node (3, Empty,
        Node (4, Empty, Empty))), Node (6, Empty, Empty)));;

min_bt (Node (5, Node (2, Node (1, Empty, Empty), Node (3, Empty,
        Node (4, Empty, Empty))), Node (6, Empty, Empty)));;



(* max e min di un albero binario qualsiasi *)
let rec max_bt bt =
    match bt with
    Empty -> invalid_arg "albero vuoto!"
    | Node (x, Empty, Empty) -> x
    | Node (x, Empty, rt) -> max x (max_bt rt)
    | Node (x, lt, Empty) -> max x (max_bt lt)
    | Node (x, lt, rt) -> max x (max (max_bt lt) (max_bt rt));;

let rec min_bt bt =
    match bt with
    Empty -> invalid_arg "albero vuoto!"
    | Node (x, Empty, Empty) -> x
    | Node (x, Empty, rt) -> min x (min_bt rt)
    | Node (x, lt, Empty) -> min x (min_bt lt)
    | Node (x, lt, rt) -> min x (min (min_bt lt) (min_bt rt));;

max_bt (Node (5, Node (4, Node (1, Empty, Empty), Node (7, Node (6, Empty,
        Empty), Empty)), Node (3, Empty, Node (2, Empty, Empty))));;

min_bt (Node (5, Node (4, Node (1, Empty, Empty), Node (7, Node (6, Empty,
        Empty), Empty)), Node (3, Empty, Node (2, Empty, Empty))));;
```

```ocaml
(* alberi binari *)
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;

(* da lista disordinata ad albero binario di ricerca *)
let rec list_bt l =
    let rec insord n bt =
        match bt with
        Empty -> Node (n, Empty, Empty)
        | Node (x, lt, rt) ->
            if (n <= x)
                then Node (x, insord n lt, rt)
                else Node (x, lt, insord n rt)
    in
    match l with
    [] -> Empty
    | x :: xs -> insord x (list_bt xs);;
(* NB ricorsione di ricorsione come nella funzione di ordinamento lista *)

(* costruisce l'albero a partire dall'ultimo elemento della lista *)
list_bt [3; 1; 5; 6; 4; 7; 2];;



(* da lista ordinata ad albero di ricerca bilanciato, se possibile completo *)
let rec take l n =
    match l with
    | _ when (n <= 0) -> []
    | x :: xs -> x :: take xs (n - 1)
    | _ -> [];;

let rec drop l n =
    match l with
    | _ when (n <= 0) -> l
    | x :: xs -> drop xs (n - 1)
    | _ -> [];;

let rec list_bt l =
    match l with
    | [] -> Empty
    | _ ->
    let k = List.length l / 2
        in
        (* radice = numero a metà, sx < radice, dx > radice *)
        Node (List.hd (drop l k), list_bt (take l k),
              list_bt (List.tl (drop l k)));;

list_bt [1; 2; 3; 4; 5; 6; 7];;
```