Lists and Arrays

1. **compare :**

   `compare : 'a -> 'a -> int`
   `compare x y` returns `0` if `x` is equal to `y`, a negative integer if `x` is less than `y`, and a positive integer if `x` is greater than `y`. Uses lexicographic order on tuples.
   The compare function can be used as the comparison function required by the Set.Make and Map.Make functors, as well as the List.sort and Array.sort functions.

2. **Lists :**

   `List.mem: 'a -> 'a list -> bool`
   `List.mem a set` is `true` if and only if `a` is equal to an element of set.

   `List.exists: ('a -> bool) -> 'a list -> bool`
   `List.exists f [a1; ...; an]` checks if at least one element of the list satisfies the predicate `f`.
   That is, it returns `(f a1) || (f a2) || ... || (f an)` for a non-empty list and `false` if the list is empty.

   `List.for_all : ('a -> bool) -> 'a list -> bool`
   `List.for_all f [a1; ...; an]` checks if all elements of the list satisfy the predicate `f`.
   That is, it returns `(f a1) && (f a2) && ... && (f an)` for a non-empty list and `true` if the list is empty.

   `List.filter : ('a -> bool) -> 'a list -> 'a list`
   `List.filter f l` returns all the elements of the list `l` that satisfy the predicate `f`. The order of the elements in the input list is preserved.

   `List.map : ('a -> 'b) -> 'a list -> 'b list`
   `List.map f [a1; ...; an]` applies function `f` to `a1,...,an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.

   `List.iter : ('a -> unit) -> 'a list -> unit`
   `List.iter f [a1; ...; an]` applies function `f` in turn to `a1,...,an`.
   It is equivalent to `begin f a1; f a2; ...; f an; () end`.

   `List.compare : ('a -> 'a -> int) -> 'a list -> 'a list -> int`
   `List.compare cmp [a1; ...; an] [b1; ...; bm]` performs a lexicographic comparison of the two input lists, using the same `'a -> 'a -> int` interface as `compare` :
   — `a1 :: l1` is smaller than `a2 :: l2` (negative result) if `a1` is smaller than `a2`, or if they are equal (`0` result) and `l1` is smaller than `l2` ;
   — the empty list `[]` is strictly smaller than non-empty lists.
   Note : the cmp function will be called even if the lists have different lengths.

   `List.sort : ('a -> 'a -> int) -> 'a list -> 'a list`
   Sort a list in increasing order according to a comparison function. The comparison function must return `0` if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see `Array.sort` for a complete specification). For example, `compare` is a suitable comparison function. The resulting list is sorted in increasing order. `List.sort` is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.
   The current implementation uses Merge Sort. It runs in constant heap space and logarithmic stack space.

3. **Arrays :**

   Notation : `a = [|a0; a1; ...|]` with elements of same type

   Value of index `i` element : `a.(i)`

   Indices starting from `0` : `a.(0) = a0`, ...

   Modification : `a.(i) <- value`

   `Array.length : 'a array -> int`
   Return the length (number of elements) of the given array.

   `Array.make : int -> 'a -> 'a array`
   `Array.make n e` returns a fresh array of length `n`, initialized with `e`. All the elements of this new array are initially physically equal to `e` (in the sense of the == predicate). Consequently, if `e` is mutable, it is shared among all elements of the array, and modifying `e` through one of the array entries will modify all other entries at the same time.

   `Array.make_matrix : int -> int -> 'a -> 'a array array`
   `Array.make_matrix dimx dimy e` returns a two-dimensional array (an array of arrays) with first dimension `dimx` and

second dimension `dimy`. All the elements of this new matrix are initially physically equal to `e`. The element `(x,y)` of a matrix `m` is accessed with the notation `m.(x).(y)`.

`Array.init : int -> (int -> 'a) -> 'a array`
`Array.init n f` returns a fresh array of length `n`, with element number `i` initialized to the result of `f i`.
In other terms, `init n f` tabulates the results of `f` applied to the integers `0` to `n-1`.

`Array.copy : 'a array -> 'a array`
`Array.copy a` returns a copy of `a`, that is, a fresh array containing the same elements as `a`.

`Array.mem : 'a -> 'a array -> bool`
`Array.mem x a` is true if and only if `x` is structurally equal to an element of `a` (i.e. there is an `e` in `a` such that `compare x e = 0`.

`Array.exists : ('a -> bool) -> 'a array -> bool`
`Array.exists f [|a1; ...; an|]` checks if at least one element of the array satisfies the predicate `f`. That is, it returns `(f a1) || (f a2) || ... || (f an)`.

`Array.for_all : ('a -> bool) -> 'a array -> bool`
`Array.for_all f [|a1; ...; an|]` checks if all elements of the array satisfy the predicate `f`. That is, it returns `(f a1) && (f a2) && ... && (f an)`.

`Array.map : ('a -> 'b) -> 'a array -> 'b array`
`Array.map f a` applies function `f` to all the elements of `a`, and builds an array with the results returned by `f` : `[| f a.(0); f a.(1); ...; f a.(length a - 1) |]`.

`Array.iter : ('a -> unit) -> 'a array -> unit`
`Array.iter f a` applies function `f` in turn to all the elements of `a`. It is equivalent to `f a.(0); f a.(1); ...; f a.(length a - 1); ()`.