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

val get : **'a array -> int -> 'a**
Array.get a n returns the element number n of array a. The first element has number 0. The last element has number Array.length a - 1. You can also write a.(n) instead of Array.get a n.
Raise Invalid_argument "index out of bounds" if n is outside the range 0 to (Array.length a - 1).

val set : ***'a array -> int -> 'a -> unit***
Array.set a n x modifies array a in place, replacing element number n with x. You can also write a.(n) <- x instead of Array.set a n x. Raise Invalid_argument "index out of bounds" if n is outside the range 0 to Array.length a - 1.

val make : int -> 'a -> 'a array
Array.make n x returns a fresh array of length n, initialized with x. All the elements of this new array are initially physically equal to x (in the sense of the == predicate). Consequently, if x is mutable, it is shared among all elements of the array, and modifying x through one of the array entries will modify all other entries at the same time.
Raise Invalid_argument if n < 0 or n > Sys.max_array_length. If the value of x is a floating-point number, then the maximum size is only Sys.max_array_length / 2.

val create : int -> 'a -> 'a array
Deprecated. Array.create is an alias for Array.make.

val create_float : int -> float array
Array.create_float n returns a fresh float array of length n, with uninitialized data.

- **Since** 4.03

val make_float : int -> float array

Deprecated. `Array.make_float` is an alias for `Array.create_float`.

val `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, `Array.init n f` tabulates the results of `f` applied to the integers `0` to `n-1`.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_array_length`. If the return type of `f` is `float`, then the maximum size is only `Sys.max_array_length / 2`.

val 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). Raise Invalid_argument if dimx or dimy is negative or greater than Sys.max_array_length. If the value of e is a floating-point number, then the maximum size is only Sys.max_array_length / 2.

val `create_matrix : int -> int -> 'a -> 'a array array`

Deprecated. `Array.create_matrix` is an alias for `Array.make_matrix`.

val append : 'a array -> 'a array -> 'a array

Array.append v1 v2 returns a fresh array containing the concatenation of the arrays v1 and v2.

val concat : 'a array list -> 'a array

Same as Array.append, but concatenates a list of arrays.

val `sub : 'a array -> int -> int -> 'a array`

`Array.sub a start len` returns a fresh array of length `len`, containing the elements number `start` to `start + len - 1` of array `a`.

Raise `Invalid_argument "Array.sub"` if `start` and `len` do not designate a valid subarray of `a`; that is, if `start < 0`, or `len < 0`, or `start + len > Array.length a`.

val copy : 'a array -> 'a array

Array.copy a returns a copy of a, that is, a fresh array containing the same elements as a.

val `fill : 'a array -> int -> int -> 'a -> unit`
`Array.fill a ofs len x` modifies the array `a` in place, storing `x` in elements number `ofs` to `ofs + len - 1`.
Raise `Invalid_argument "Array.fill"` if `ofs` and `len` do not designate a valid subarray of `a`.
val `blit : 'a array -> int -> 'a array -> int -> int -> unit`
`Array.blit v1 o1 v2 o2 len` copies `len` elements from array `v1`, starting at element number `o1`, to array `v2`, starting at element number `o2`. It works correctly even if `v1` and `v2` are the same array, and the source and destination chunks overlap.
Raise `Invalid_argument "Array.blit"` if `o1` and `len` do not designate a valid subarray of `v1`, or if `o2` and `len` do not designate a valid subarray of `v2`.

val to_list : 'a array -> 'a list
Array.to_list a returns the list of all the elements of a.
val of_list : 'a list -> 'a array
Array.of_list l returns a fresh array containing the elements of l.

Iterators

val 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.(Array.length a - 1); ().

val `iteri : (int -> 'a -> unit) -> 'a array -> unit`
Same as `Array.iter`, but the function is applied with the index of the element as first argument, and the element itself as second argument.

val 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.(Array.length a - 1) |].

val `mapi : (int -> 'a -> 'b) -> 'a array -> 'b array`
Same as `Array.map`, but the function is applied to the index of the element as first argument, and the element itself as second argument.

val fold_left *: ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a*
Array.fold_left f x a computes f (... (f (f x a.(0)) a.(1)) ...) a.(n-1), where n is the length of the array a.

val fold_right : ('b -> 'a -> 'a) -> 'b array -> 'a -> 'a
Array.fold_right f a x computes f a.(0) (f a.(1) (... (f a.(n-1) x) ...)), where n is the length of the array a.

Iterators on two arrays

val iter2 *: ('a -> 'b -> unit) -> 'a array -> 'b array -> unit*
Array.iter2 f a b applies function f to all the elements of a and b. Raise Invalid_argument if the arrays are not the same size.

- **Since** 4.03.0
-

val map2 *: ('a -> 'b -> 'c) -> 'a array -> 'b array -> 'c array*
Array.map2 f a b applies function f to all the elements of a and b, and builds an array with the results returned by f: [| f a.(0) b.(0); ...; f a.(Array.length a - 1) b.(Array.length b - 1)|]. Raise Invalid_argument if the arrays are not the same size.

- **Since** 4.03.0
-

Array scanning

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

- *Since 4.03.0*
-

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

- **Since** 4.03.0

val mem `: 'a -> 'a array -> bool`

`mem a l` is true if and only if `a` is equal to an element of `l`.

- **Since** 4.03.0

`val memq : 'a -> 'a array -> bool`

Same as `Array.mem`, but uses physical equality instead of structural equality to compare array elements.

- **Since** 4.03.0

Sorting

`val sort : ('a -> 'a -> int) -> 'a array -> unit`

Sort an array 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 below for a complete specification). For example, `compare` is a suitable comparison function, provided there are no floating-point NaN values in the data. After calling `Array.sort`, the array is sorted in place in increasing order. `Array.sort` is guaranteed to run in constant heap space and (at most) logarithmic stack space.

The current implementation uses Heap Sort. It runs in constant stack space.

Specification of the comparison function: Let `a` be the array and `cmp` the comparison function. The following must be true for all x, y, z in a :

- `cmp x y` > 0 if and only if `cmp y x` < 0
- if `cmp x y` >= 0 and `cmp y z` >= 0 then `cmp x z` >= 0

When `Array.sort` returns, `a` contains the same elements as before, reordered in such a way that for all i and j valid indices of `a` :

- `cmp a.(i) a.(j)` >= 0 if and only if i >= j

`val stable_sort : ('a -> 'a -> int) -> 'a array -> unit`

Same as `Array.sort`, but the sorting algorithm is stable (i.e. elements that compare equal are kept in their original order) and not guaranteed to run in constant heap space.

The current implementation uses Merge Sort. It uses `n/2` words of heap space, where `n` is the length of the array. It is usually faster than the current implementation of `Array.sort`.

`val fast_sort : ('a -> 'a -> int) -> 'a array -> unit`

Same as `Array.sort` or `Array.stable_sort`, whichever is faster on typical input.

LES LISTES

val length : 'a list -> int
Return the length (number of elements) of the given list.

val compare_lengths **:** 'a list -> 'b list -> int
Compare the lengths of two lists. `compare_lengths l1 l2` is equivalent to `compare (length l1) (length l2)`, except that the computation stops after itering on the shortest list.

- **Since** 4.05.0

val compare_length_with **:** 'a list -> int -> int
Compare the length of a list to an integer. `compare_length_with l n` is equivalent to `compare (length l) n`, except that the computation stops after at most `n` iterations on the list.

- **Since** 4.05.0

val cons **:** 'a -> 'a list -> 'a list
cons x xs is `x :: xs`

- **Since** 4.03.0

val hd **:** 'a list -> 'a
Return the first element of the given list. Raise `Failure "hd"` if the list is empty.

val tl **:** 'a list -> 'a list
Return the given list without its first element. Raise `Failure "tl"` if the list is empty.

val nth : 'a list -> int -> 'a
Return the n-th element of the given list. The first element (head of the list) is at position 0. Raise Failure "nth" if the list is too short. Raise Invalid_argument "List.nth" if n is negative.

val nth_opt **:** 'a list -> int -> 'a option
Return the `n`-th element of the given list. The first element (head of the list) is at position 0. Return `None` if the list is too short. Raise `Invalid_argument "List.nth"` if `n` is negative.

- **Since** 4.05

val rev : 'a list -> 'a list
List reversal.

val init : int -> (int -> 'a) -> 'a list
List.init len f is f 0; f 1; ...; f (len-1), evaluated left to right.

- **Since** 4.06.0
- **Raises** `Invalid_argument` if len < 0.

val append : 'a list -> 'a list -> 'a list
Concatenate two lists. Same as the infix operator @. Not tail-recursive (length of the first argument).

val rev_append : 'a list -> 'a list -> 'a list
List.rev_append l1 l2 reverses l1 and concatenates it to l2. This is equivalent to List.rev l1 @ l2, but rev_append is tail-recursive and more efficient.

val concat : 'a list list -> 'a list
Concatenate a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result. Not tail-recursive (length of the argument + length of the longest sub-list).

val `flatten : 'a list list -> 'a list`
An alias for `concat`.

Iterators

val 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.

val `iteri : (int -> 'a -> unit) -> 'a list -> unit`
Same as `List.iter`, but the function is applied to the index of the element as first argument (counting from 0), and the element itself as second argument.

- **Since** 4.00.0
-

val 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.

`val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list`
Same as `List.map`, but the function is applied to the index of the element as first argument (counting from 0), and the element itself as second argument. Not tail-recursive.

- **Since** 4.00.0

`val rev_map : ('a -> 'b) -> 'a list -> 'b list`
`List.rev_map f l` gives the same result as `List.rev (List.map f l)`, but is tail-recursive and more efficient.

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_left f a [b1; ...; bn] is
f (... (f (f a b1) b2) ...) bn.

`val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
`List.fold_right f [a1; ...; an] b` is
`f a1 (f a2 (... (f an b) ...))`. Not tail-recursive.

Iterators on two lists

val iter2 : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit
List.iter2 f [a1; ...; an] [b1; ...; bn] calls in turn
f a1 b1; ...; f an bn. Raise Invalid_argument if the two lists are determined to have different lengths.

val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
List.map2 f [a1; ...; an] [b1; ...; bn] is
[f a1 b1; ...; f an bn]. Raise Invalid_argument if the two lists are determined to have different lengths. Not tail-recursive.

`val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`

`List.rev_map2 f l1 l2` gives the same result as `List.rev (List.map2 f l1 l2)`, but is tail-recursive and more efficient.

val `fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a`

`List.fold_left2 f a [b1; ...; bn] [c1; ...; cn]` is `f (... (f (f a b1 c1) b2 c2) ...) bn cn`. Raise `Invalid_argument` if the two lists are determined to have different lengths.

val `fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c`

`List.fold_right2 f [a1; ...; an] [b1; ...; bn] c` is `f a1 b1 (f a2 b2 (... (f an bn c) ...))`. Raise `Invalid_argument` if the two lists are determined to have different lengths. Not tail-recursive.

List scanning

val for_all : ('a -> bool) -> 'a list -> bool

for_all p [a1; ...; an] checks if all elements of the list satisfy the predicate p. That is, it returns (p a1) && (p a2) && ... && (p an).

val exists : ('a -> bool) -> 'a list -> bool

exists p [a1; ...; an] checks if at least one element of the list satisfies the predicate p. That is, it returns (p a1) || (p a2) || ... || (p an).

val `for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool`

Same as `List.for_all`, but for a two-argument predicate. Raise `Invalid_argument` if the two lists are determined to have different lengths.

val `exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool`

Same as `List.exists`, but for a two-argument predicate. Raise `Invalid_argument` if the two lists are determined to have different lengths.

val mem : 'a -> 'a list -> bool

mem a l is true if and only if a is equal to an element of l.

val `memq : 'a -> 'a list -> bool`

Same as `List.mem`, but uses physical equality instead of structural equality to compare list elements.

List searching

val find **: ('a -> bool) -> 'a list -> 'a**
find p l returns the first element of the list l that satisfies the predicate p. Raise Not_found if there is no value that satisfies p in the list l.

val find_opt **:** ('a -> bool) -> 'a list -> 'a option
find_opt p l returns the first element of the list l that satisfies the predicate p, or None if there is no value that satisfies p in the list l.

- **Since** 4.05

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

val find_all **:** ('a -> bool) -> 'a list -> 'a list
find_all is another name for List.filter.

val partition **:** ***('a -> bool) -> 'a list -> 'a list * 'a list***
partition p l returns a pair of lists (l1, l2), where l1 is the list of all the elements of l that satisfy the predicate p, and l2 is the list of all the elements of l that do not satisfy p. The order of the elements in the input list is preserved.

Association lists

val assoc **:** ***'a -> ('a * 'b) list -> 'b***
assoc a l returns the value associated with key a in the list of pairs l. That is, assoc a [...; (a,b); ...] = b if (a,b) is the leftmost binding of a in list l. Raise Not_found if there is no value associated with a in the list l.

`val assoc_opt : 'a -> ('a * 'b) list -> 'b option`
`assoc_opt a l` returns the value associated with key `a` in the list of pairs `l`. That is, `assoc_opt a [ ...; (a,b); ...] = b` if `(a,b)` is the leftmost binding of `a` in list `l`. Returns `None` if there is no value associated with `a` in the list `l`.

- **Since** 4.05

`val assq : 'a -> ('a * 'b) list -> 'b`
Same as `List.assoc`, but uses physical equality instead of structural equality to compare keys.

`val assq_opt : 'a -> ('a * 'b) list -> 'b option`
Same as `List.assoc_opt`, but uses physical equality instead of structural equality to compare keys.

- **Since** 4.05

`val mem_assoc : 'a -> ('a * 'b) list -> bool`
Same as `List.assoc`, but simply return true if a binding exists, and false if no bindings exist for the given key.

`val mem_assq : 'a -> ('a * 'b) list -> bool`
Same as `List.mem_assoc`, but uses physical equality instead of structural equality to compare keys.

`val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list`
`remove_assoc a l` returns the list of pairs `l` without the first pair with key `a`, if any. Not tail-recursive.

`val remove_assq : 'a -> ('a * 'b) list -> ('a * 'b) list`
Same as `List.remove_assoc`, but uses physical equality instead of structural equality to compare keys. Not tail-recursive.

Lists of pairs

`val split : ('a * 'b) list -> 'a list * 'b list`
Transform a list of pairs into a pair of lists:
`split [(a1,b1); ...; (an,bn)]` is
`([a1; ...; an], [b1; ...; bn])`. Not tail-recursive.

val combine : 'a list -> 'b list -> ('a * 'b) list
Transform a pair of lists into a list of pairs:
combine [a1; ...; an] [b1; ...; bn]* is *[(a1,b1); ...; (an,bn)].
Raise `Invalid_argument` if the two lists have different lengths. Not tail-recursive.

Sorting

`val 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.
`val stable_sort : ('a -> 'a -> int) -> 'a list -> 'a list`
Same as `List.sort`, but the sorting algorithm is guaranteed to be stable (i.e. elements that compare equal are kept in their original order) .
The current implementation uses Merge Sort. It runs in constant heap space and logarithmic stack space.
`val fast_sort : ('a -> 'a -> int) -> 'a list -> 'a list`
Same as `List.sort` or `List.stable_sort`, whichever is faster on typical input.
`val sort_uniq : ('a -> 'a -> int) -> 'a list -> 'a list`
Same as `List.sort`, but also remove duplicates.

- **Since** 4.02.0

`val merge : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list`
Merge two lists: Assuming that `l1` and `l2` are sorted according to the comparison function `cmp`, `merge cmp l1 l2` will return a sorted list containing all the elements of `l1` and `l2`. If several elements compare equal, the elements of `l1` will be before the elements of `l2`. Not tail-recursive (sum of the lengths of the arguments).