

Module `std::vector`

A variable-sized container that can hold any type. Indexing is 0-based, and vectors are growable. This module has many native functions.

The index into the vector is out of bounds

Create an empty vector.

Return the length of the vector.

Acquire an immutable reference to the *i*th element of the vector *v*. Aborts if *i* is out of bounds.

Add element *e* to the end of the vector *v*.

Return a mutable reference to the *i*th element in the vector *v*. Aborts if *i* is out of bounds.

Pop an element from the end of vector *v*. Aborts if *v* is empty.

Destroy the vector *v*. Aborts if *v* is not empty.

Swaps the elements at the *i*th and *j*th indices in the vector *v*. Aborts if *i* or *j* is out of bounds.

Return an vector of size one containing element *e*.

Reverses the order of the elements in the vector *v* in place.

Pushes all of the elements of the other vector into the lhs vector.

Return true if the vector *v* has no elements and false otherwise.

Return true if *e* is in the vector *v*. Otherwise, returns false.

Return (true , *i*) if *e* is in the vector *v* at index *i*. Otherwise, returns (false , 0).

Remove the *i*th element of the vector *v*, shifting all subsequent elements. This is $O(n)$ and preserves ordering of elements in the vector. Aborts if *i* is out of bounds.

Insert *e* at position *i* in the vector *v*. If *i* is in bounds, this shifts the old *v*[*i*] and all subsequent elements to the right. If *i* == *v*. [length\(\)](#), this adds *e* to the end of the vector. This is $O(n)$ and preserves ordering of elements in the vector. Aborts if *i* > *v*. [length\(\)](#)

Swap the *i*th element of the vector *v* with the last element and then pop the vector. This is $O(1)$, but does not preserve ordering of elements in the vector. Aborts if *i* is out of bounds.

Create a vector of length *n* by calling the function *f* on each index.

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Preserves the order of elements in the vector.

Perform an action *f* on each element of the vector *v*. The vector is not modified.

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first

containing the elements for which `f` returns true , and the second containing the elements for which `f` returns false .

Finds the index of first element in the vector `v` that satisfies the predicate `f`. Returns `some(index)` if such an element is found, otherwise `none()`.

Count how many elements in the vector `v` satisfy the predicate `f`.

Reduce the vector `v` to a single value by applying the function `f` to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

Concatenate the vectors of `v` into a single vector, keeping the order of the elements.

Whether any element in the vector `v` satisfies the predicate `f`. If the vector is empty, returns false .

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns true .

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

Iterate through `v1` and `v2` and apply the function `f` to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

Destroys two vectors `v1` and `v2` by applying the function `f` to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns true if the vector is sorted, false otherwise.

Constants

The index into the vector is out of bounds

```
```bash
```

```
```
```

Create an empty vector.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Return the length of the vector.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Acquire an immutable reference to the *i*th element of the vector *v*. Aborts if *i* is out of bounds.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Add element *e* to the end of the vector *v*.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Return a mutable reference to the *i*th element in the vector *v*. Aborts if *i* is out of bounds.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Pop an element from the end of vector *v*. Aborts if *v* is empty.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroy the vector *v*. Aborts if *v* is not empty.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Swaps the elements at the *i*th and *j*th indices in the vector *v*. Aborts if *i* or *j* is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return an vector of size one containing element *e*.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Reverses the order of the elements in the vector *v* in place.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Pushes all of the elements of the other vector into the *lhs* vector.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return true if the vector *v* has no elements and false otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return true if *e* is in the vector *v*. Otherwise, returns false.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return (true , *i*) if *e* is in the vector *v* at index *i*. Otherwise, returns (false , 0).

```
```bash
```

```
...
```

```
```bash
```

```
'''
```

Remove the i th element of the vector v , shifting all subsequent elements. This is $O(n)$ and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert e at position i in the vector v . If i is in bounds, this shifts the old $v[i]$ and all subsequent elements to the right. If $i = v.length()$, this adds e to the end of the vector. This is $O(n)$ and preserves ordering of elements in the vector. Aborts if $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the i th element of the vector v with the last element and then pop the vector. This is $O(1)$, but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length n by calling the function f on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector v by calling f on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector v by calling f on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action f on each element of the vector v . The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector `v` satisfy the predicate `f`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector `v` to a single value by applying the function `f` to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of `v` into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector `v` satisfies the predicate `f`. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```



```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Create an empty vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return the length of the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Acquire an immutable reference to the `i`th element of the vector `v`. Aborts if `i` is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Add element *e* to the end of the vector *v*.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return a mutable reference to the *i*th element in the vector *v*. Aborts if *i* is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Pop an element from the end of vector *v*. Aborts if *v* is empty.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroy the vector *v*. Aborts if *v* is not empty.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Swaps the elements at the *i*th and *j*th indices in the vector *v*. Aborts if *i* or *j* is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return an vector of size one containing element *e*.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Reverses the order of the elements in the vector *v* in place.

```
```bash
```

```
...
```

```
```bash
```

```
'''
```

Pushes all of the elements of the other vector into the lhs vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return true if the vector v has no elements and false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return true if e is in the vector v. Otherwise, returns false.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return ( true , i) if e is in the vector v at index i. Otherwise, returns ( false , 0).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Remove the ith element of the vector v, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert e at position i in the vector v. If i is in bounds, this shifts the old v[i] and all subsequent elements to the right. If i == v. [length\(\)](#), this adds e to the end of the vector. This is O(n) and preserves ordering of elements in the vector. Aborts if i > v. [length\(\)](#)

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the ith element of the vector v with the last element and then pop the vector. This is O(1), but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length  $n$  by calling the function  $f$  on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action  $f$  on each element of the vector  $v$ . The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action  $f$  on each element of the vector  $v$ . The function  $f$  takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector  $v$  to a new vector by applying the function  $f$  to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold\_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of `v` into a single vector, keeping the order of the elements.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Whether any element in the vector `v` satisfies the predicate `f`. If the vector is empty, returns `false` .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector v is sorted in non-decreasing order according to the comparison function le (les). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Function

Return the length of the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Acquire an immutable reference to the *i*th element of the vector *v*. Aborts if *i* is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Add element *e* to the end of the vector *v*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return a mutable reference to the *i*th element in the vector *v*. Aborts if *i* is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Pop an element from the end of vector *v*. Aborts if *v* is empty.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector *v*. Aborts if *v* is not empty.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```


Swaps the elements at the *i*th and *j*th indices in the vector *v*. Aborts if *i* or *j* is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return an vector of size one containing element *e*.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Reverses the order of the elements in the vector *v* in place.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Pushes all of the elements of the other vector into the *lhs* vector.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return true if the vector *v* has no elements and false otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return true if *e* is in the vector *v*. Otherwise, returns false.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return (true , *i*) if *e* is in the vector *v* at index *i*. Otherwise, returns (false , 0).

```
```bash
```

```
...
```

```
```bash
```

```
'''
```

Remove the i th element of the vector v , shifting all subsequent elements. This is $O(n)$ and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert e at position i in the vector v . If i is in bounds, this shifts the old $v[i]$ and all subsequent elements to the right. If $i = v.length()$, this adds e to the end of the vector. This is $O(n)$ and preserves ordering of elements in the vector. Aborts if $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the i th element of the vector v with the last element and then pop the vector. This is $O(1)$, but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length n by calling the function f on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector v by calling f on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector v by calling f on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action f on each element of the vector v . The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector `v` satisfy the predicate `f`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector `v` to a single value by applying the function `f` to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of `v` into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector `v` satisfies the predicate `f`. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Acquire an immutable reference to the `i`th element of the vector `v`. Aborts if `i` is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Add element `e` to the end of the vector `v`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return a mutable reference to the `i`th element in the vector `v`. Aborts if `i` is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Pop an element from the end of vector v. Aborts if v is empty.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroy the vector v. Aborts if v is not empty.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Swaps the elements at the ith and jth indices in the vector v. Aborts if i or j is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return an vector of size one containing element e.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Reverses the order of the elements in the vector v in place.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Pushes all of the elements of the other vector into the lhs vector.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return true if the vector v has no elements and false otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
'''
```

Return true if e is in the vector v. Otherwise, returns false.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return ( true , i) if e is in the vector v at index i. Otherwise, returns ( false , 0).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Remove the ith element of the vector v, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert e at position i in the vector v. If i is in bounds, this shifts the old v[i] and all subsequent elements to the right. If  $i = v.length()$ , this adds e to the end of the vector. This is O(n) and preserves ordering of elements in the vector. Aborts if  $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the ith element of the vector v with the last element and then pop the vector. This is O(1), but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length n by calling the function f on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector v by calling f on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).



```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, *first* is called *first*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, *first* is called *first*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns *true*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold\_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by applying the function `f` to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Add element `e` to the end of the vector `v`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return a mutable reference to the  $i$ th element in the vector  $v$ . Aborts if  $i$  is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Pop an element from the end of vector  $v$ . Aborts if  $v$  is empty.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroy the vector  $v$ . Aborts if  $v$  is not empty.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Swaps the elements at the  $i$ th and  $j$ th indices in the vector  $v$ . Aborts if  $i$  or  $j$  is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return an vector of size one containing element  $e$ .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Reverses the order of the elements in the vector  $v$  in place.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Pushes all of the elements of the other vector into the lhs vector.

```
```bash
```

```
...
```

```
```bash
```

```
'''
```

Return true if the vector v has no elements and false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return true if e is in the vector v. Otherwise, returns false.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return ( true , i) if e is in the vector v at index i. Otherwise, returns ( false , 0).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Remove the ith element of the vector v, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert e at position i in the vector v. If i is in bounds, this shifts the old v[i] and all subsequent elements to the right. If  $i = v.length()$ , this adds e to the end of the vector. This is O(n) and preserves ordering of elements in the vector. Aborts if  $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the ith element of the vector v with the last element and then pop the vector. This is O(1), but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length n by calling the function f on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold\_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```



Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns false .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns true .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through *v1* and *v2* and apply the function *f* to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through *v1* and *v2* and apply the function *f* to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors *v1* and *v2* by applying the function *f* to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Return a mutable reference to the *i*th element in the vector *v*. Aborts if *i* is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Pop an element from the end of vector *v*. Aborts if *v* is empty.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroy the vector *v*. Aborts if *v* is not empty.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Swaps the elements at the *i*th and *j*th indices in the vector *v*. Aborts if *i* or *j* is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return an vector of size one containing element *e*.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Reverses the order of the elements in the vector *v* in place.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Pushes all of the elements of the other vector into the *lhs* vector.

```
```bash
```

```
...
```

```
```bash
```

```
'''
```

Return true if the vector v has no elements and false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return true if e is in the vector v. Otherwise, returns false.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return ( true , i) if e is in the vector v at index i. Otherwise, returns ( false , 0).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Remove the ith element of the vector v, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert e at position i in the vector v. If i is in bounds, this shifts the old v[i] and all subsequent elements to the right. If  $i = v.length()$ , this adds e to the end of the vector. This is O(n) and preserves ordering of elements in the vector. Aborts if  $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the ith element of the vector v with the last element and then pop the vector. This is O(1), but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length n by calling the function f on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold\_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns false .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns true .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through *v1* and *v2* and apply the function *f* to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through *v1* and *v2* and apply the function *f* to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors *v1* and *v2* by applying the function *f* to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector v is sorted in non-decreasing order according to the comparison function le (les). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function



Pop an element from the end of vector v. Aborts if v is empty.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroy the vector v. Aborts if v is not empty.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Swaps the elements at the ith and jth indices in the vector v. Aborts if i or j is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return an vector of size one containing element e.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Reverses the order of the elements in the vector v in place.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Pushes all of the elements of the other vector into the lhs vector.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return true if the vector v has no elements and false otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
'''
```

Return true if e is in the vector v. Otherwise, returns false.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return ( true , i) if e is in the vector v at index i. Otherwise, returns ( false , 0).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Remove the ith element of the vector v, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert e at position i in the vector v. If i is in bounds, this shifts the old v[i] and all subsequent elements to the right. If  $i == v.length()$ , this adds e to the end of the vector. This is O(n) and preserves ordering of elements in the vector. Aborts if  $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the ith element of the vector v with the last element and then pop the vector. This is O(1), but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length n by calling the function f on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector v by calling f on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, *first* is called *first*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, *first* is called *first*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns *true*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold\_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by applying the function `f` to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Destroy the vector `v`. Aborts if `v` is not empty.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swaps the elements at the *i*th and *j*th indices in the vector *v*. Aborts if *i* or *j* is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return an vector of size one containing element *e*.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Reverses the order of the elements in the vector *v* in place.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Pushes all of the elements of the other vector into the *lhs* vector.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return true if the vector *v* has no elements and false otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return true if *e* is in the vector *v*. Otherwise, returns false.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return ( true , *i*) if *e* is in the vector *v* at index *i*. Otherwise, returns ( false , 0).

```
```bash
```

```
...
```

```
```bash
```

```
'''
```

Remove the  $i$ th element of the vector  $v$ , shifting all subsequent elements. This is  $O(n)$  and preserves ordering of elements in the vector. Aborts if  $i$  is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert  $e$  at position  $i$  in the vector  $v$ . If  $i$  is in bounds, this shifts the old  $v[i]$  and all subsequent elements to the right. If  $i = v.length()$ , this adds  $e$  to the end of the vector. This is  $O(n)$  and preserves ordering of elements in the vector. Aborts if  $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the  $i$ th element of the vector  $v$  with the last element and then pop the vector. This is  $O(1)$ , but does not preserve ordering of elements in the vector. Aborts if  $i$  is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length  $n$  by calling the function  $f$  on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action  $f$  on each element of the vector  $v$ . The vector is not modified.



```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector `v` satisfy the predicate `f`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector `v` to a single value by applying the function `f` to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of `v` into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector `v` satisfies the predicate `f`. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Function

Swaps the elements at the `i`th and `j`th indices in the vector `v`. Aborts if `i` or `j` is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return an vector of size one containing element `e`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reverses the order of the elements in the vector `v` in place.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Pushes all of the elements of the other vector into the lhs vector.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return true if the vector v has no elements and false otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return true if e is in the vector v. Otherwise, returns false.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Return (true , i) if e is in the vector v at index i. Otherwise, returns (false , 0).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Remove the ith element of the vector v, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Insert e at position i in the vector v. If i is in bounds, this shifts the old v[i] and all subsequent elements to the right. If i == v. [length\(\)](#), this adds e to the end of the vector. This is O(n) and preserves ordering of elements in the vector. Aborts if i > v. [length\(\)](#)

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Swap the ith element of the vector v with the last element and then pop the vector. This is O(1), but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
```bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length  $n$  by calling the function  $f$  on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action  $f$  on each element of the vector  $v$ . The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action  $f$  on each element of the vector  $v$ . The function  $f$  takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector  $v$  to a new vector by applying the function  $f$  to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold\_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector v satisfies the predicate f. If the vector is empty, returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector v satisfy the predicate f. If the vector is empty, returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```



```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector v is sorted in non-decreasing order according to the comparison function le (les). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
```bash
```

```
```
```

## Function

Return an vector of size one containing element e.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Reverses the order of the elements in the vector v in place.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Pushes all of the elements of the other vector into the lhs vector.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Return true if the vector v has no elements and false otherwise.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Return true if e is in the vector v. Otherwise, returns false.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Return ( true , i) if e is in the vector v at index i. Otherwise, returns ( false , 0).

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Remove the ith element of the vector v, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the

vector. Aborts if  $i$  is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert  $e$  at position  $i$  in the vector  $v$ . If  $i$  is in bounds, this shifts the old  $v[i]$  and all subsequent elements to the right. If  $i = v.length()$ , this adds  $e$  to the end of the vector. This is  $O(n)$  and preserves ordering of elements in the vector. Aborts if  $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the  $i$ th element of the vector  $v$  with the last element and then pop the vector. This is  $O(1)$ , but does not preserve ordering of elements in the vector. Aborts if  $i$  is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length  $n$  by calling the function  $f$  on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action  $f$  on each element of the vector  $v$ . The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector `v` satisfy the predicate `f`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector `v` to a single value by applying the function `f` to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of `v` into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector `v` satisfies the predicate `f`. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. Starts from the

end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Reverses the order of the elements in the vector `v` in place.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Pushes all of the elements of the other vector into the `lhs` vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return `true` if the vector `v` has no elements and `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return `true` if `e` is in the vector `v`. Otherwise, returns `false`.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Return ( true , i) if e is in the vector v at index i. Otherwise, returns ( false , 0).

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Remove the ith element of the vector v, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Insert e at position i in the vector v. If i is in bounds, this shifts the old v[i] and all subsequent elements to the right. If  $i = v.length()$ , this adds e to the end of the vector. This is O(n) and preserves ordering of elements in the vector. Aborts if  $i > v.length()$

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Swap the ith element of the vector v with the last element and then pop the vector. This is O(1), but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Create a vector of length n by calling the function f on each index.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroy the vector v by calling f on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
```bash
```

```
```
```



```
'''bash
```

```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector  $v$  into two vectors by applying the function  $f$  to each element. Return a tuple containing two vectors: the first containing the elements for which  $f$  returns true , and the second containing the elements for which  $f$  returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector  $v$  that satisfies the predicate  $f$ . Returns `some(index)` if such an element is found, otherwise `none()`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector  $v$  satisfy the predicate  $f$ .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector  $v$  to a single value by applying the function  $f$  to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of  $v$  into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector  $v$  satisfies the predicate  $f$ . If the vector is empty, returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector  $v$  satisfy the predicate  $f$ . If the vector is empty, returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Pushes all of the elements of the other vector into the `lhs` vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return `true` if the vector `v` has no elements and `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return true if e is in the vector v. Otherwise, returns false.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return ( true , i) if e is in the vector v at index i. Otherwise, returns ( false , 0).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Remove the ith element of the vector v, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert e at position i in the vector v. If i is in bounds, this shifts the old v[i] and all subsequent elements to the right. If  $i = v.length()$ , this adds e to the end of the vector. This is O(n) and preserves ordering of elements in the vector. Aborts if  $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the ith element of the vector v with the last element and then pop the vector. This is O(1), but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length n by calling the function f on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold\_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through *v1* and *v2* and apply the function *f* to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through *v1* and *v2* and apply the function *f* to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by applying the function *f* to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```



```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector v is sorted in non-decreasing order according to the comparison function le (les). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Return true if the vector v has no elements and false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return true if e is in the vector v. Otherwise, returns false.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Return ( true , i) if e is in the vector v at index i. Otherwise, returns ( false , 0).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Remove the ith element of the vector v, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert e at position i in the vector v. If i is in bounds, this shifts the old v[i] and all subsequent elements to the right. If  $i = v.length()$ , this adds e to the end of the vector. This is O(n) and preserves ordering of elements in the vector. Aborts if  $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the ith element of the vector v with the last element and then pop the vector. This is O(1), but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length n by calling the function f on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Preserves the order of elements in the vector.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Perform an action *f* on each element of the vector *v*. The vector is not modified.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
```bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector v satisfy the predicate f. If the vector is empty, returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Check if the vector v is sorted in non-decreasing order according to the comparison function le (les). Returns true if the vector is sorted, false otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Function

Return true if e is in the vector v. Otherwise, returns false.

```
```bash
```

```
...
```

```
```bash
```

```
'''
```

Return (true , i) if e is in the vector v at index i. Otherwise, returns (false , 0).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Remove the ith element of the vector v, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert e at position i in the vector v. If i is in bounds, this shifts the old v[i] and all subsequent elements to the right. If $i == v.length()$, this adds e to the end of the vector. This is O(n) and preserves ordering of elements in the vector. Aborts if $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the ith element of the vector v with the last element and then pop the vector. This is O(1), but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length n by calling the function f on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector v by calling f on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector v by calling f on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```



```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns *false* .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns *true* .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of

elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Function

Return (`true` , `i`) if `e` is in the vector `v` at index `i`. Otherwise, returns (`false` , `0`).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Remove the `i`th element of the vector `v`, shifting all subsequent elements. This is $O(n)$ and preserves ordering of elements in the vector. Aborts if `i` is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Insert e at position i in the vector v . If i is in bounds, this shifts the old $v[i]$ and all subsequent elements to the right. If $i = v.length()$, this adds e to the end of the vector. This is $O(n)$ and preserves ordering of elements in the vector. Aborts if $i > v.length()$

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Swap the i th element of the vector v with the last element and then pop the vector. This is $O(1)$, but does not preserve ordering of elements in the vector. Aborts if i is out of bounds.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Create a vector of length n by calling the function f on each index.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroy the vector v by calling f on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroy the vector v by calling f on each element and then destroying the vector. Preserves the order of elements in the vector.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Perform an action f on each element of the vector v . The vector is not modified.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Perform an action f on each element of the vector v . The function f takes a mutable reference to the element.

```
```bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector  $v$  is sorted in non-decreasing order according to the comparison function  $le$  ( $les$ ). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Remove the  $i$ th element of the vector  $v$ , shifting all subsequent elements. This is  $O(n)$  and preserves ordering of elements in the vector. Aborts if  $i$  is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Insert  $e$  at position  $i$  in the vector  $v$ . If  $i$  is in bounds, this shifts the old  $v[i]$  and all subsequent elements to the right. If  $i = v.length()$ , this adds  $e$  to the end of the vector. This is  $O(n)$  and preserves ordering of elements in the vector. Aborts if  $i > v.length()$

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the  $i$ th element of the vector  $v$  with the last element and then pop the vector. This is  $O(1)$ , but does not preserve ordering of elements in the vector. Aborts if  $i$  is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length  $n$  by calling the function  $f$  on each index.

```
'''bash
```

```
'''
```

```
'''bash
```



```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold\_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through *v1* and *v2* and apply the function *f* to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through *v1* and *v2* and apply the function *f* to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by applying the function *f* to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector v is sorted in non-decreasing order according to the comparison function le (les). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Insert e at position i in the vector v. If i is in bounds, this shifts the old v[i] and all subsequent elements to the right. If i == v. [length\(\)](#), this adds e to the end of the vector. This is O(n) and preserves ordering of elements in the vector. Aborts if i > v. [length\(\)](#)

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Swap the  $i$ th element of the vector  $v$  with the last element and then pop the vector. This is  $O(1)$ , but does not preserve ordering of elements in the vector. Aborts if  $i$  is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length  $n$  by calling the function  $f$  on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action  $f$  on each element of the vector  $v$ . The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action  $f$  on each element of the vector  $v$ . The function  $f$  takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold\_left* in Rust and *reduce* in Python and JavaScript.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Concatenate the vectors of  $v$  into a single vector, keeping the order of the elements.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Whether any element in the vector  $v$  satisfies the predicate  $f$ . If the vector is empty, returns false .

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Whether all elements in the vector  $v$  satisfy the predicate  $f$ . If the vector is empty, returns true .

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroys two vectors  $v1$  and  $v2$  by calling  $f$  to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroys two vectors  $v1$  and  $v2$  by calling  $f$  to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through  $v1$  and  $v2$  and apply the function  $f$  to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```



```
'''bash
```

```
'''
```

Check if the vector  $v$  is sorted in non-decreasing order according to the comparison function  $le$  ( $les$ ). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Swap the  $i$ th element of the vector  $v$  with the last element and then pop the vector. This is  $O(1)$ , but does not preserve ordering of elements in the vector. Aborts if  $i$  is out of bounds.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Create a vector of length  $n$  by calling the function  $f$  on each index.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector  $v$  by calling  $f$  on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action  $f$  on each element of the vector  $v$ . The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector `v` satisfy the predicate `f`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector `v` to a single value by applying the function `f` to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of `v` into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector `v` satisfies the predicate `f`. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

## Macro function

Create a vector of length `n` by calling the function `f` on each index.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroy the vector `v` by calling `f` on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroy the vector `v` by calling `f` on each element and then destroying the vector. Preserves the order of elements in the vector.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Perform an action `f` on each element of the vector `v`. The vector is not modified.

```
```bash
```

```
```
```

```
'''bash
```

```
'''
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
```
```

```
```bash
```



```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Macro function

Destroy the vector `v` by calling `f` on each element and then destroying the vector. Does not preserve the order of elements in the vector (starts from the end of the vector).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroy the vector `v` by calling `f` on each element and then destroying the vector. Preserves the order of elements in the vector.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action `f` on each element of the vector `v`. The vector is not modified.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Perform an action `f` on each element of the vector `v`. The function `f` takes a mutable reference to the element.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Check if the vector *v* is sorted in non-decreasing order according to the comparison function *le* (*les*). Returns true if the vector is sorted, false otherwise.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Macro function

Destroy the vector *v* by calling *f* on each element and then destroying the vector. Preserves the order of elements in the vector.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Perform an action *f* on each element of the vector *v*. The vector is not modified.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, *first* is called first.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through *v1* and *v2* and apply the function *f* to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through *v1* and *v2* and apply the function *f* to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector v is sorted in non-decreasing order according to the comparison function le (les). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```



```
```bash
```

```
```
```

Macro function

Perform an action *f* on each element of the vector *v*. The vector is not modified.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Perform an action *f* on each element of the vector *v*. The function *f* takes a mutable reference to the element.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
```bash
```

```
```
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns *false* .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns *true* .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of

elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Macro function

Perform an action `f` on each element of the vector `v`. The function `f` takes a mutable reference to the element.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Map the vector `v` to a new vector by applying the function `f` to each element. Preserves the order of elements in the vector, `first` is called first.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns false .

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns true .

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through *v1* and *v2* and apply the function *f* to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through *v1* and *v2* and apply the function *f* to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector v is sorted in non-decreasing order according to the comparison function le (les). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Macro function

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```



```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector *v* is sorted in non-decreasing order according to the comparison function *le* (*les*). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Macro function

Map the vector *v* to a new vector by applying the function *f* to each element. Preserves the order of elements in the vector, first is called first.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Filter the vector *v* by applying the function *f* to each element. Return a new vector containing only the elements for which *f* returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector *v* into two vectors by applying the function *f* to each element. Return a tuple containing two vectors: the first containing the elements for which *f* returns true , and the second containing the elements for which *f* returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Macro function

Filter the vector `v` by applying the function `f` to each element. Return a new vector containing only the elements for which `f` returns `true`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Split the vector `v` into two vectors by applying the function `f` to each element. Return a tuple containing two vectors: the first containing the elements for which `f` returns `true`, and the second containing the elements for which `f` returns `false`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector `v` that satisfies the predicate `f`. Returns `some(index)` if such an element is found, otherwise `none()`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector `v` satisfy the predicate `f`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector *v* satisfy the predicate *f*. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors *v1* and *v2* by calling *f* to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a

stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Macro function

Split the vector `v` into two vectors by applying the function `f` to each element. Return a tuple containing two vectors: the first containing the elements for which `f` returns `true`, and the second containing the elements for which `f` returns `false`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Finds the index of first element in the vector `v` that satisfies the predicate `f`. Returns `some(index)` if such an element is found, otherwise `none()`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector `v` satisfy the predicate `f`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector `v` to a single value by applying the function `f` to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of `v` into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector `v` satisfies the predicate `f`. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector *v* is sorted in non-decreasing order according to the comparison function *le* (*les*). Returns *true* if the vector is sorted, *false* otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Macro function

Finds the index of first element in the vector *v* that satisfies the predicate *f*. Returns *some(index)* if such an element is found, otherwise *none()*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Count how many elements in the vector *v* satisfy the predicate *f*.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector *v* to a single value by applying the function *f* to each element. Similar to *fold_left* in Rust and *reduce* in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of *v* into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector *v* satisfies the predicate *f*. If the vector is empty, returns *false* .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by applying the function `f` to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Macro function

Count how many elements in the vector `v` satisfy the predicate `f`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Reduce the vector `v` to a single value by applying the function `f` to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Concatenate the vectors of `v` into a single vector, keeping the order of the elements.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Whether any element in the vector `v` satisfies the predicate `f`. If the vector is empty, returns `false` .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```



```
'''
```

```
'''bash
```

```
'''
```

Check if the vector  $v$  is sorted in non-decreasing order according to the comparison function  $le$  ( $les$ ). Returns true if the vector is sorted, false otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Macro function

Reduce the vector  $v$  to a single value by applying the function  $f$  to each element. Similar to `fold_left` in Rust and `reduce` in Python and JavaScript.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Concatenate the vectors of  $v$  into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector  $v$  satisfies the predicate  $f$ . If the vector is empty, returns false .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector  $v$  satisfy the predicate  $f$ . If the vector is empty, returns true .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors  $v1$  and  $v2$  by calling  $f$  to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
```bash
```

```
```
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g.

when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Function

Concatenate the vectors of `v` into a single vector, keeping the order of the elements.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether any element in the vector `v` satisfies the predicate `f`. If the vector is empty, returns `false` .

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true` .

```
'''bash
```

```
'''
```

```
```bash
```

```
```
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns true if the vector is sorted, false otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Macro function

Whether any element in the vector `v` satisfies the predicate `f`. If the vector is empty, returns false .

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns true .

```
```bash
```

```
...
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Macro function

Whether all elements in the vector `v` satisfy the predicate `f`. If the vector is empty, returns `true`.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
```bash
```

```
```
```

Destroys two vectors v1 and v2 by calling f to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g.



when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Macro function

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

## Macro function

Destroys two vectors `v1` and `v2` by calling `f` to each pair of elements. Aborts if the vectors are not of the same length. Starts from the end of the vectors.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through `v1` and `v2` and apply the function `f` to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Destroys two vectors `v1` and `v2` by applying the function `f` to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
'''bash
```

```
'''
```

```
'''bash
```

```
'''
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns true if the vector is sorted, false otherwise.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

## Macro function

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The vectors are not modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
```
```

```
```bash
```

```
```
```

Iterate through v1 and v2 and apply the function f to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors v1 and v2 by applying the function f to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through v1 and v2 and apply the function f to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector v using the comparison function le. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function le expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector v using the comparison function le. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function le expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

## Macro function

Iterate through `v1` and `v2` and apply the function `f` to mutable references of each pair of elements. The vectors may be modified. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Destroys two vectors `v1` and `v2` by applying the function `f` to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns true if the vector is sorted, false otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

## Macro function

Destroys two vectors `v1` and `v2` by applying the function `f` to each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

## Macro function

Iterate through `v1` and `v2` and apply the function `f` to references of each pair of elements. The returned values are collected into a new vector. Aborts if the vectors are not of the same length. The order of elements in the vectors is preserved.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:



Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
```bash
```

```
...
```

```
```bash
```

```
...
```

## Macro function

Performs an in-place insertion sort on the vector `v` using the comparison function `le`. The sort is stable, meaning that equal elements will maintain their relative order.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Insertion sort is efficient for small vectors (~30 or less elements), and can be faster than merge sort for almost sorted vectors (e.g. when the vector is already sorted or nearly sorted).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

```
```bash
```

```
...
```

```
```bash
```

```
...
```

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns `true` if the vector is sorted, `false` otherwise.

```
```bash
```

```
...
```

```
```bash
```

'''

## Macro function

Performs an in-place merge sort on the vector `v` using the comparison function `le`. Merge sort is efficient for large vectors, and is a stable sort.

Please, note that the comparison function `le` expects less or equal, not less.

Example:

Merge sort performs better than insertion sort for large vectors (~30 elements or more).

'''bash

'''

'''bash

'''

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns true if the vector is sorted, false otherwise.

'''bash

'''

'''bash

'''

## Macro function

Check if the vector `v` is sorted in non-decreasing order according to the comparison function `le` (`les`). Returns true if the vector is sorted, false otherwise.

'''bash

'''

'''bash

'''