Sorting and Related Functions

Julia has an extensive, flexible API for sorting and interacting with already-sorted arrays of values. By default, Julia picks reasonable algorithms and sorts in standard ascending order:

```
julia> sort([2,3,1])
3-element Array{Int64,1}:
 1
 2
 3
```

You can easily sort in reverse order as well:

```
julia> sort([2,3,1], rev=true)
3-element Array{Int64,1}:
 3
 2
 1
```

To sort an array in-place, use the "bang" version of the sort function:

```
julia> a = [2,3,1];

julia> sort!(a);

julia> a
3-element Array{Int64,1}:
 1
 2
 3
```

Instead of directly sorting an array, you can compute a permutation of the array's indices that puts the array into sorted order:

```
julia> v = randn(5)
5-element Array{Float64,1}:
  0.297288
  0.382396
```

```
 -0.597634
 -0.0104452
 -0.839027

julia> p = sortperm(v)
5-element Array{Int64,1}:
 5
 3
 4
 1
 2

julia> v[p]
5-element Array{Float64,1}:
 -0.839027
 -0.597634
 -0.0104452
  0.297288
  0.382396
```

Arrays can easily be sorted according to an arbitrary transformation of their values:

```
julia> sort(v, by=abs)
5-element Array{Float64,1}:
 -0.0104452
  0.297288
  0.382396
 -0.597634
 -0.839027
```

Or in reverse order by a transformation:

```
julia> sort(v, by=abs, rev=true)
5-element Array{Float64,1}:
 -0.839027
 -0.597634
  0.382396
  0.297288
 -0.0104452
```

If needed, the sorting algorithm can be chosen:

```
julia> sort(v, alg=InsertionSort)
```

```
5-element Array{Float64,1}:
 -0.839027
 -0.597634
 -0.0104452
  0.297288
  0.382396
```

All the sorting and order related functions rely on a "less than" relation defining a total order on the values to be manipulated. The `isless` function is invoked by default, but the relation can be specified via the `lt` keyword.

Sorting Functions

`Base.sort!` — Function

```
sort!(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev::Bool=false, ord
```

Sort the vector `v` in place. `QuickSort` is used by default for numeric arrays while `MergeSort` is used for other arrays. You can specify an algorithm to use via the `alg` keyword (see Sorting Algorithms for available algorithms). The `by` keyword lets you provide a function that will be applied to each element before comparison; the `lt` keyword allows providing a custom "less than" function; use `rev=true` to reverse the sorting order. These options are independent and can be used together in all possible combinations: if both `by` and `lt` are specified, the `lt` function is applied to the result of the `by` function; `rev=true` reverses whatever ordering specified via the `by` and `lt` keywords.

Examples

```
julia> v = [3, 1, 2]; sort!(v); v
3-element Array{Int64,1}:
 1
 2
 3

julia> v = [3, 1, 2]; sort!(v, rev = true); v
3-element Array{Int64,1}:
 3
 2
 1
```

```
julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[1]); v
3-element Array{Tuple{Int64,String},1}:
 (1, "c")
 (2, "b")
 (3, "a")

julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[2]); v
3-element Array{Tuple{Int64,String},1}:
 (3, "a")
 (2, "b")
 (1, "c")
```

```
sort!(A; dims::Integer, alg::Algorithm=defalg(A), lt=isless, by=identity, rev::E
```

Sort the multidimensional array `A` along dimension `dims`. See `sort!` for a description of possible keyword arguments.

To sort slices of an array, refer to `sortslices`.

> **Julia 1.1**
>
> This function requires at least Julia 1.1.

Examples

```
julia> A = [4 3; 1 2]
2×2 Array{Int64,2}:
 4  3
 1  2

julia> sort!(A, dims = 1); A
2×2 Array{Int64,2}:
 1  2
 4  3

julia> sort!(A, dims = 2); A
2×2 Array{Int64,2}:
 1  2
 3  4
```

`Base.sort` — Function

```
sort(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev::Bool=false, orde
```

Variant of `sort!` that returns a sorted copy of `v` leaving `v` itself unmodified.

Examples

```
julia> v = [3, 1, 2];

julia> sort(v)
3-element Array{Int64,1}:
 1
 2
 3

julia> v
3-element Array{Int64,1}:
 3
 1
 2
```

```
sort(A; dims::Integer, alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity,
```

Sort a multidimensional array `A` along the given dimension. See `sort!` for a description of possible keyword arguments.

To sort slices of an array, refer to `sortslices`.

Examples

```
julia> A = [4 3; 1 2]
2×2 Array{Int64,2}:
 4  3
 1  2

julia> sort(A, dims = 1)
2×2 Array{Int64,2}:
 1  2
 4  3
```

```
julia> sort(A, dims = 2)
2×2 Array{Int64,2}:
 3  4
 1  2
```

`Base.sortperm` — Function

```
sortperm(v; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=
```

Return a permutation vector `I` that puts `v[I]` in sorted order. The order is specified using the same keywords as `sort!`. The permutation is guaranteed to be stable even if the sorting algorithm is unstable, meaning that indices of equal elements appear in ascending order.

See also `sortperm!`.

Examples

```
julia> v = [3, 1, 2];

julia> p = sortperm(v)
3-element Array{Int64,1}:
 2
 3
 1

julia> v[p]
3-element Array{Int64,1}:
 1
 2
 3
```

`Base.Sort.InsertionSort` — Constant

```
InsertionSort
```

Indicate that a sorting function should use the insertion sort algorithm. Insertion sort traverses the collection one element at a time, inserting each element into its correct, sorted position in the output list.

Characteristics:

- *stable*: preserves the ordering of elements which compare equal (e.g. "a" and "A" in a sort of letters which ignores case).
- *in-place* in memory.
- *quadratic performance* in the number of elements to be sorted: it is well-suited to small collections but should not be used for large ones.

`Base.Sort.MergeSort` — Constant

```
MergeSort
```

Indicate that a sorting function should use the merge sort algorithm. Merge sort divides the collection into subcollections and repeatedly merges them, sorting each subcollection at each step, until the entire collection has been recombined in sorted form.

Characteristics:

- *stable*: preserves the ordering of elements which compare equal (e.g. "a" and "A" in a sort of letters which ignores case).
- *not in-place* in memory.
- *divide-and-conquer* sort strategy.

`Base.Sort.QuickSort` — Constant

```
QuickSort
```

Indicate that a sorting function should use the quick sort algorithm, which is *not* stable.

Characteristics:

- *not stable*: does not preserve the ordering of elements which compare equal (e.g. "a" and "A" in a sort of letters which ignores case).
- *in-place* in memory.
- *divide-and-conquer*: sort strategy similar to `MergeSort`.
- *good performance* for large collections.

`Base.Sort.PartialQuickSort` — Type

```
PartialQuickSort{T <: Union{Integer,OrdinalRange}}
```

Indicate that a sorting function should use the partial quick sort algorithm. Partial quick sort returns the smallest `k` elements sorted from smallest to largest, finding them and sorting them using `QuickSort`.

Characteristics:

- *not stable*: does not preserve the ordering of elements which compare equal (e.g. "a" and "A" in a sort of letters which ignores case).
- *in-place* in memory.
- *divide-and-conquer*: sort strategy similar to `MergeSort`.

`Base.Sort.sortperm!` — Function

```
sortperm!(ix, v; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::
```

Like `sortperm`, but accepts a preallocated index vector `ix`. If `initialized` is `false` (the default), `ix` is initialized to contain the values `1:length(v)`.

Examples

```
julia> v = [3, 1, 2]; p = zeros(Int, 3);

julia> sortperm!(p, v); p
3-element Array{Int64,1}:
 2
 3
 1

julia> v[p]
3-element Array{Int64,1}:
 1
 2
 3
```

`Base.sortslices` — Function

```
sortslices(A; dims, alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, re
```

Sort slices of an array `A`. The required keyword argument `dims` must be either an integer or a tuple of integers. It specifies the dimension(s) over which the slices are sorted.

E.g., if `A` is a matrix, `dims=1` will sort rows, `dims=2` will sort columns. Note that the default comparison function on one dimensional slices sorts lexicographically.

For the remaining keyword arguments, see the documentation of `sort!`.

Examples

```
julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1) # Sort rows
3×3 Array{Int64,2}:
 -1   6  4
  7   3  5
  9  -2  8

julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1, lt=(x,y)->isless(x[2],y[2]))
3×3 Array{Int64,2}:
  9  -2  8
  7   3  5
 -1   6  4

julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1, rev=true)
3×3 Array{Int64,2}:
  9  -2  8
  7   3  5
 -1   6  4

julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2) # Sort columns
3×3 Array{Int64,2}:
  3   5  7
 -1  -4  6
 -2   8  9

julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2, alg=InsertionSort, lt=(x,y)
3×3 Array{Int64,2}:
  5   3  7
 -4  -1  6
  8  -2  9
```

```
julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2, rev=true)
3×3 Array{Int64,2}:
 7   5   3
 6  -4  -1
 9   8  -2
```

Higher dimensions

`sortslices` extends naturally to higher dimensions. E.g., if `A` is a a 2x2x2 array, `sortslices(A, dims=3)` will sort slices within the 3rd dimension, passing the 2x2 slices `A[:, :, 1]` and `A[:, :, 2]` to the comparison function. Note that while there is no default order on higher-dimensional slices, you may use the `by` or `lt` keyword argument to specify such an order.

If `dims` is a tuple, the order of the dimensions in `dims` is relevant and specifies the linear order of the slices. E.g., if `A` is three dimensional and `dims` is `(1, 2)`, the orderings of the first two dimensions are re-arranged such such that the slices (of the remaining third dimension) are sorted. If `dims` is `(2, 1)` instead, the same slices will be taken, but the result order will be row-major instead.

Higher dimensional examples

```
julia> A = permutedims(reshape([4 3; 2 1; 'A' 'B'; 'C' 'D'], (2, 2, 2)), (1, 3,
2×2×2 Array{Any,3}:
[:, :, 1] =
 4  3
 2  1

[:, :, 2] =
 'A'  'B'
 'C'  'D'

julia> sortslices(A, dims=(1,2))
2×2×2 Array{Any,3}:
[:, :, 1] =
 1  3
 2  4

[:, :, 2] =
 'D'  'B'
 'C'  'A'

julia> sortslices(A, dims=(2,1))
2×2×2 Array{Any,3}:
```

```
[:, :, 1] =
 1  2
 3  4

[:, :, 2] =
 'D'  'C'
 'B'  'A'

julia> sortslices(reshape([5; 4; 3; 2; 1], (1,1,5)), dims=3, by=x->x[1,1])
1×1×5 Array{Int64,3}:
[:, :, 1] =
 1

[:, :, 2] =
 2

[:, :, 3] =
 3

[:, :, 4] =
 4

[:, :, 5] =
 5
```

Order-Related Functions

`Base.issorted` — Function

```
issorted(v, lt=isless, by=identity, rev:Bool=false, order::Ordering=Forward)
```

Test whether a vector is in sorted order. The `lt`, `by` and `rev` keywords modify what order is considered to be sorted just as they do for `sort`.

Examples

```
julia> issorted([1, 2, 3])
true

julia> issorted([(1, "b"), (2, "a")], by = x -> x[1])
true
```

```
julia> issorted([(1, "b"), (2, "a")], by = x -> x[2])
false

julia> issorted([(1, "b"), (2, "a")], by = x -> x[2], rev=true)
true
```

`Base.Sort.searchsorted` — Function

```
searchsorted(a, x; by=<transform>, lt=<comparison>, rev=false)
```

Return the range of indices of `a` which compare as equal to `x` (using binary search) according to the order specified by the `by`, `lt` and `rev` keywords, assuming that `a` is already sorted in that order. Return an empty range located at the insertion point if `a` does not contain values equal to `x`.

Examples

```
julia> searchsorted([1, 2, 4, 5, 5, 7], 4) # single match
3:3

julia> searchsorted([1, 2, 4, 5, 5, 7], 5) # multiple matches
4:5

julia> searchsorted([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middle
3:2

julia> searchsorted([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
7:6

julia> searchsorted([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
1:0
```

`Base.Sort.searchsortedfirst` — Function

```
searchsortedfirst(a, x; by=<transform>, lt=<comparison>, rev=false)
```

Return the index of the first value in `a` greater than or equal to `x`, according to the specified order. Return `length(a) + 1` if `x` is greater than all values in `a`. `a` is assumed to be sorted.

Examples

```
julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 4) # single match
3

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 5) # multiple matches
4

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middl
3

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
7

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
1
```

`Base.Sort.searchsortedlast` — Function

```
searchsortedlast(a, x; by=<transform>, lt=<comparison>, rev=false)
```

Return the index of the last value in `a` less than or equal to `x`, according to the specified order. Return `0` if `x` is less than all values in `a`. `a` is assumed to be sorted.

Examples

```
julia> searchsortedlast([1, 2, 4, 5, 5, 7], 4) # single match
3

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 5) # multiple matches
5

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middle
2

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
6

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
0
```

`Base.Sort.partialsort!` — Function

```
partialsort!(v, k; by=<transform>, lt=<comparison>, rev=false)
```

Partially sort the vector `v` in place, according to the order specified by `by`, `lt` and `rev` so that the value at index `k` (or range of adjacent values if `k` is a range) occurs at the position where it would appear if the array were fully sorted via a non-stable algorithm. If `k` is a single index, that value is returned; if `k` is a range, an array of values at those indices is returned. Note that `partialsort!` does not fully sort the input array.

Examples

```
julia> a = [1, 2, 4, 3, 4]
5-element Array{Int64,1}:
 1
 2
 4
 3
 4

julia> partialsort!(a, 4)
4

julia> a
5-element Array{Int64,1}:
 1
 2
 3
 4
 4

julia> a = [1, 2, 4, 3, 4]
5-element Array{Int64,1}:
 1
 2
 4
 3
 4

julia> partialsort!(a, 4, rev=true)
2
```

```
julia> a
5-element Array{Int64,1}:
 4
 4
 3
 2
 1
```

`Base.Sort.partialsort` — Function

```
partialsort(v, k, by=<transform>, lt=<comparison>, rev=false)
```

Variant of `partialsort!` which copies `v` before partially sorting it, thereby returning the same thing as `partialsort!` but leaving `v` unmodified.

`Base.Sort.partialsortperm` — Function

```
partialsortperm(v, k; by=<transform>, lt=<comparison>, rev=false)
```

Return a partial permutation `I` of the vector `v`, so that `v[I]` returns values of a fully sorted version of `v` at index `k`. If `k` is a range, a vector of indices is returned; if `k` is an integer, a single index is returned. The order is specified using the same keywords as `sort!`. The permutation is stable, meaning that indices of equal elements appear in ascending order.

Note that this function is equivalent to, but more efficient than, calling `sortperm(...)[k]`.

Examples

```
julia> v = [3, 1, 2, 1];

julia> v[partialsortperm(v, 1)]
1

julia> p = partialsortperm(v, 1:3)
3-element view(::Array{Int64,1}, 1:3) with eltype Int64:
 2
 4
 3
```

```
julia> v[p]
3-element Array{Int64,1}:
 1
 1
 2
```

`Base.Sort.partialsortperm!` — Function

```
partialsortperm!(ix, v, k; by=<transform>, lt=<comparison>, rev=false, initiali
```

Like `partialsortperm`, but accepts a preallocated index vector `ix` the same size as `v`, which is used to store (a permutation of) the indices of `v`.

If the index vector `ix` is initialized with the indices of `v` (or a permutation thereof), `initialized` should be set to `true`.

If `initialized` is `false` (the default), then `ix` is initialized to contain the indices of `v`.

If `initialized` is `true`, but `ix` does not contain (a permutation of) the indices of `v`, the behavior of `partialsortperm!` is undefined.

(Typically, the indices of `v` will be `1:length(v)`, although if `v` has an alternative array type with non-one-based indices, such as an `OffsetArray`, `ix` must also be an `OffsetArray` with the same indices, and must contain as values (a permutation of) these same indices.)

Upon return, `ix` is guaranteed to have the indices `k` in their sorted positions, such that

```
partialsortperm!(ix, v, k);
v[ix[k]] == partialsort(v, k)
```

The return value is the `k`th element of `ix` if `k` is an integer, or view into `ix` if `k` is a range.

Examples

```
julia> v = [3, 1, 2, 1];

julia> ix = Vector{Int}(undef, 4);

julia> partialsortperm!(ix, v, 1)
2
```

```
julia> ix = [1:4;];

julia> partialsortperm!(ix, v, 2:3, initialized=true)
2-element view(::Array{Int64,1}, 2:3) with eltype Int64:
 4
 3
```

Sorting Algorithms

There are currently four sorting algorithms available in base Julia:

- `InsertionSort`
- `QuickSort`
- `PartialQuickSort(k)`
- `MergeSort`

`InsertionSort` is an O(n^2) stable sorting algorithm. It is efficient for very small `n`, and is used internally by `QuickSort`.

`QuickSort` is an O(n log n) sorting algorithm which is in-place, very fast, but not stable – i.e. elements which are considered equal will not remain in the same order in which they originally appeared in the array to be sorted. `QuickSort` is the default algorithm for numeric values, including integers and floats.

`PartialQuickSort(k)` is similar to `QuickSort`, but the output array is only sorted up to index `k` if `k` is an integer, or in the range of `k` if `k` is an `OrdinalRange`. For example:

```
x = rand(1:500, 100)
k = 50
k2 = 50:100
s = sort(x; alg=QuickSort)
ps = sort(x; alg=PartialQuickSort(k))
qs = sort(x; alg=PartialQuickSort(k2))
map(issorted, (s, ps, qs))             # => (true, false, false)
map(x->issorted(x[1:k]), (s, ps, qs))  # => (true, true, false)
map(x->issorted(x[k2]), (s, ps, qs))   # => (true, false, true)
s[1:k] == ps[1:k]                      # => true
s[k2] == qs[k2]                        # => true
```

`MergeSort` is an O(n log n) stable sorting algorithm but is not in-place – it requires a temporary array of half the size of the input array – and is typically not quite as fast as `QuickSort`. It is the default algorithm for non-numeric data.

The default sorting algorithms are chosen on the basis that they are fast and stable, or *appear* to be so. For numeric types indeed, `QuickSort` is selected as it is faster and indistinguishable in this case from a stable sort (unless the array records its mutations in some way). The stability property comes at a non-negligible cost, so if you don't need it, you may want to explicitly specify your preferred algorithm, e.g. `sort!(v, alg=QuickSort)`.

The mechanism by which Julia picks default sorting algorithms is implemented via the `Base.Sort.defalg` function. It allows a particular algorithm to be registered as the default in all sorting functions for specific arrays. For example, here are the two default methods from `sort.jl`:

```
defalg(v::AbstractArray) = MergeSort
defalg(v::AbstractArray{<:Number}) = QuickSort
```

As for numeric arrays, choosing a non-stable default algorithm for array types for which the notion of a stable sort is meaningless (i.e. when two values comparing equal can not be distinguished) may make sense.

« Punctuation Iteration utilities »