	Based on input: [1, -6, -7, -2, 5, 4]
partition	Based on negative nrs: [-2, -6, -7, 1, 5, 4]
stable_partition	[-6, -7, -2, 1, 5, 4]
partition_copy	same as above but output in a new container
is_partitioned	check if it is already partitioned based on predicate
partition_point	The partition_point() algorithm returns an iterator to the first object for which a provided predicate returns false. If our range is partitioned, this means it returns an iterator to the first object in the second partition:

Partitioning vs sorting

Partitioning doesn't involve sorting, but the result from both proesses obeys to the same partitioning rule.
In sorting and partitioning all negative numbers are put at the beginning.
The key difference is that sorting a collection can require significantly more operations than partitioning it. As such, if our use case only requires partitioning our collection, we can get large performance benefits by not unnecessarily sorting everything.

count	Return nr of appearances of value in container
count_if	Count based on a predicate, example nr of neg numbers
any_of	Return true if any number in container is <predicate>
none_of	Return true if no number in container is <predicate>
all_of	All numbers in container are <predicate>
count_if	Count based on a predicate, example nr of neg numbers
any_of	Return true if any number in container is <predicate>
none_of	Return true if no number in container is <predicate>

Counting algorithms

find	find(input, 5); //returns an iterator to element with value 5
find_if	find based on predicate
find_first_of	find any value in a range of possibilities
adjacent_find	search for two equal consecutive elements
search_n	search_n(v, 2, 4); //search container for 2 sequential values of 4
search	serach a subrange in a range default_searcher, boyer_more_searcher
find_end	like "search" but in reverse direction
binary_search	search for a value in a sorted container
lower_bound	Ret earliest potential position of value if it were in the sorted container
upper_bound	Ret last potential position of value if it were in the sorted container
equal_range	Determines both the lower and upper bound in a single algorithm. It returns a std::pair of low/up iterators

```
lower_bound
        ↓ Lower
{1, 2, 3, 4, 5}
      Upper ↑
          upper_bound
```

Searching algorithms

rotate	rotate([1,2,3,4],begin()+1); // => [2,3,4,1] Rotate so the begin()+1 it is the first element in the range
reverse	Reverse order of elements in collection
shuffle	Reorder elements in a random order

Movement algorithms

copy	copy({1,2,3}, {0,0,0,0}); => {1,2,3,0}
copy_backward	copy({1,2,3}, {0,0,0,0}); => {0,1,2,3} Useful when "in" and "out" ranges are overlapping. We overwrite one of the input values so we lose it.
reverse_copy	reverse_copy({1,2,3}, {0,0,0,0}); => {3,2,1,0}
rotate_copy	rotate_copy({1,2,3}, {0,0,0,0}); => {3,1,2,0}
copy_n	copy_n({1,2,3,4}, 2, {0,0,0}); => {1,2,0}
copy_if	Copy only values that obeys to predicate
unique_copy	unique_copy({1,1,2,1,2,2,2},{0,0,0,0,0,0,0}); => {1,2,1,2}

Copy algorithms

clamp	clamp(-30, 0, 255); // returns 0
min	Take min value in a collection
max	Take the maximul element in a collection
minmax	Take smallest and biggest elements from collection
all_of	All numbers in container are <predicate>
count_if	Count based on a predicate, example nr of neg numbers
any_of	Return true if any number in container is <predicate>
none_of	Return true if no number in container is <predicate>

Counting algorithms

Remove algorithms don't remove anything, rather ensure the elements we want are placed at the beginning of the continuer

remove	[s,e] { remove({1,2,3,4}, 2); => {1,3,4,4} } //Use remove erase idiom container.erase(s, e)
remove_if	remove_if({1,2,3,4,5,6}, isEven); => {1,3,5,2,4,6}
remove_copy	remove_copy({1,2,3}, {0,0,0}, 2); => {1,3}
remove_copy_if	same as remove_copy but based on predicate

Removing algorithms

includes	includes({1,2,3}, {2,3}); // => return true
set_union	set_union({1,2},{2,3}); // => {1,2,3}
set_intersection	set_intersection({1,2,3,4}, {1,3,9}); // => {1,3}
set_difference	set_difference({1,2,3,4}, {1,3,9}); // => {1,3}
set_symetric_difference	set_sym_diff({1,2,3,4}, {1,3,9}); // => {2,4,9}

Set algorithms

replace	replace({1,2,3,3,4}, 3, 0}; // => {1,2,0,0,4}
replace_if	same as "replace" but based on predicate
replace_copy	remove_copy({1,2,3}, {0,0,0}, 2); => {1,3}
replace_copy_if	same as replace_copy but based on predicate

Replacing algorithms

reduce	reduce({2,2,3}, std::multiplier{}); // => returns 12
	// By default parallel execution, commutative and associative otherwise has non-deterministic result
	
	For example, if our operator function is func and our input is a collection comprising of a, b, and c, the std::reduce() algorithm might return the result of:
	• func(a, func(b, c))
	• func(b, func(a, c))
	• func(func(c, a), b)
	• or any other permutation

accumulate	Non-parallel but deterministic. Always from left to right. Can be used with non associative operations

accumulate with filter	`auto V{`
	`  std::views::filter(Numbers, [](int i){`
	`    return i % 2 == 1;`
	`  })};`
	
	`std::cout << "Result: "`
	`  << std::accumulate(V.begin(), V.end(), 0);`

lexicographical_compare	lexi_comp({1,2,3}, {1,2,4}); // A si less than B
starts_with	starts_with({1,2,3,4,5}, {1,2,3}); // Return true
ends_with	Same as starts_with but from the end of collection

reduce and accumulate algorithms

equal	equal({1,2,3}, {2,3,1}); //False, not the same order
is_permutation	is_permutation({1,2,3},{3,1,2}); // => True
mismatch	mismatch({1,2,3}, {1,3}); // => it to pos 2 // Where two input collections deviate
lexicographical_compare	lexi_comp({1,2,3}, {1,2,4}); // A si less than B
starts_with	starts_with({1,2,3,4,5}, {1,2,3}); // Return true
ends_with	Same as starts_with but from the end of collection

Comparison algorithms

for_each	Applies an operation on each element of the container for_each({1,2,3}, pow2); // => {1,4,9};

Individual algorithms