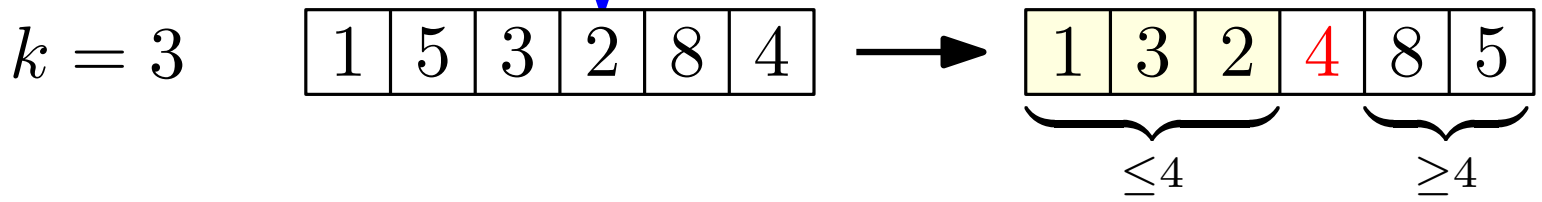
Sorting

- k-th smallest element $O(n)$ comparisons

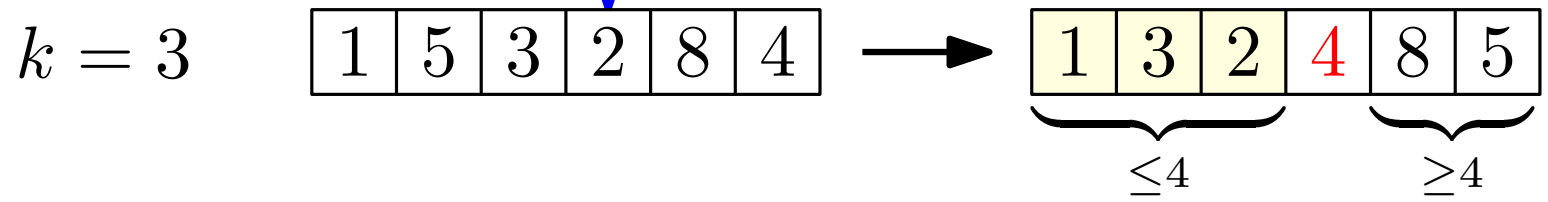
```
std::nth_element(vec.begin(), vec.begin()+k, vec.end());
```



Sorting

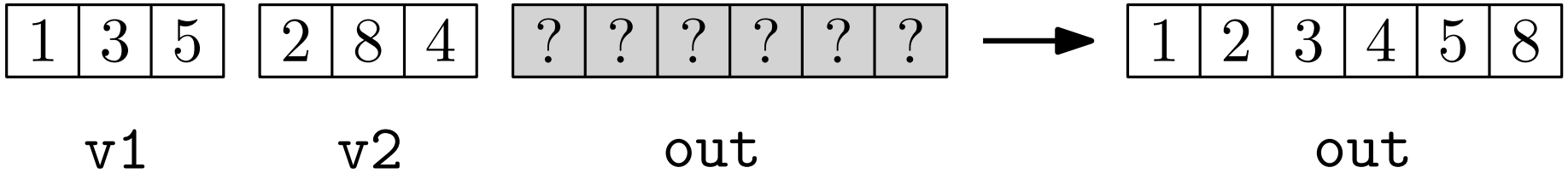
- k-th smallest element — $O(n)$ comparisons

```
std::nth_element(vec.begin(), vec.begin()+k, vec.end());
```



- Merge — $O(n_1 + n_2)$ comparisons

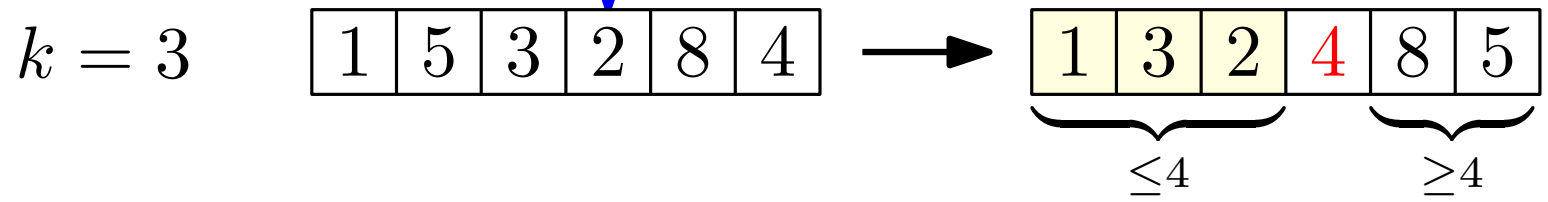
```
std :: merge(v1.begin(), v1.end(),
     v2.begin(), v2.end(),
     out.begin());
```



Sorting

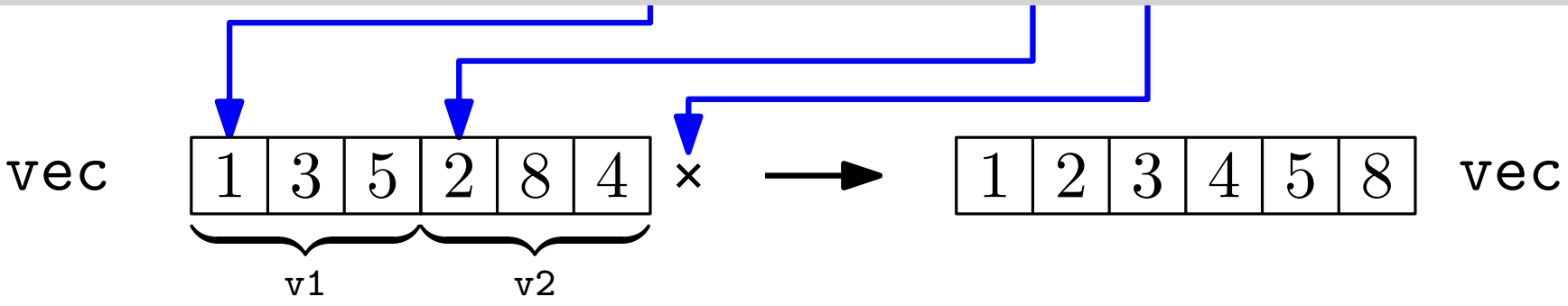
- k-th smallest element — $O(n)$ comparisons

```
std::nth_element(vec.begin(), vec.begin()+k, vec.end());
```



- Merge — $O(n_1 + n_2)$ comparisons

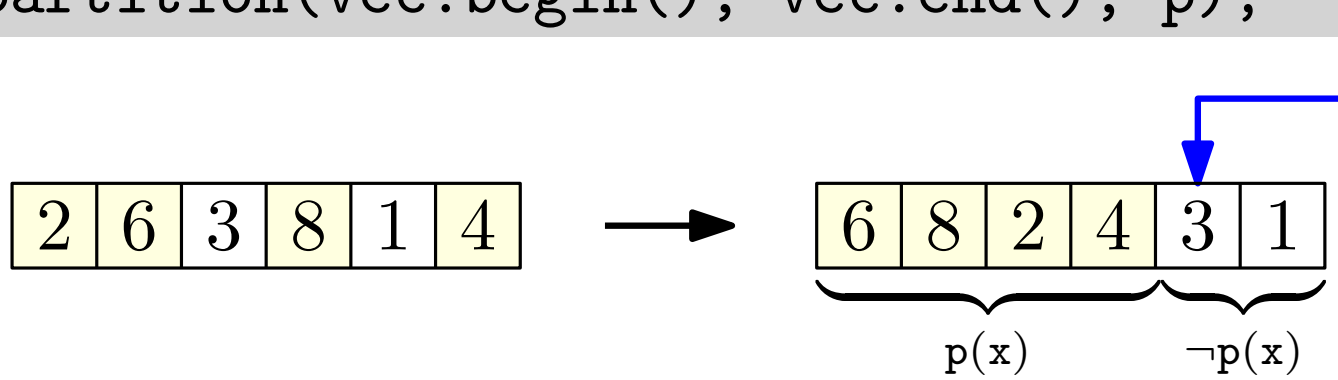
```
std::inplace_merge(vec.begin(), mid, vec.end());
```



Partitioning

$O(n)$ calls to p

```
std::partition(vec.begin(), vec.end(), p);
```




```
bool p(int i) { return i%2==0; }
```

```
std::partition(vec.begin(), vec.end(), [](int i){ return i%2==0; });
```

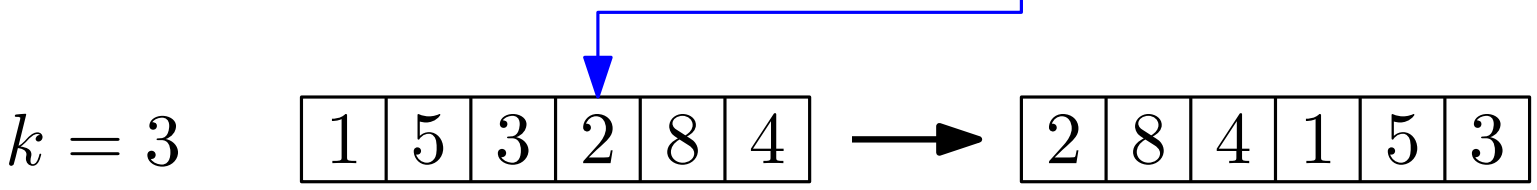
See also:

```
std::stable_partition(vec.begin(), vec.end(), p);
std::parition_point(vec.begin(), vec.end(), p);
bool std::is_partiotioned(vec.begin(), vec.end(), p);
```

Permutations

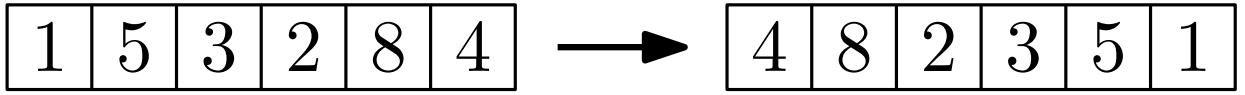
- Rotations $O(n)$

```
std::rotate(vec.begin(), vec.begin()+k, vec.end());
```



- Reversing a permutation $O(n)$

```
std::reverse(vec.begin(), vec.end());
```



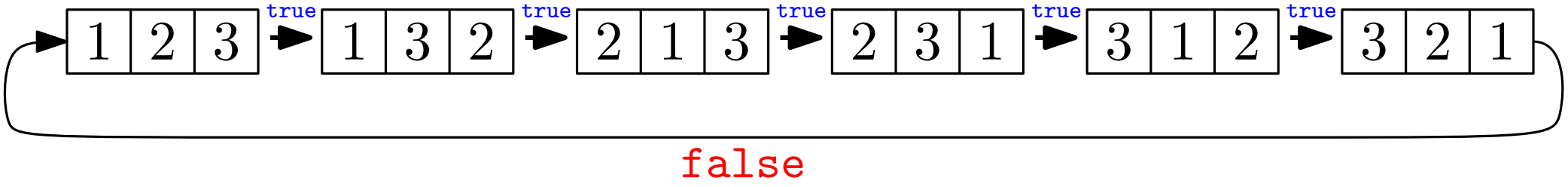
- Shuffling $O(n)$

```
std::shuffle(vec.begin(), vec.end(), rng);
```

Permutations

- Next/Previous permutation in lexicographic order $O(n)$

```
bool std::next_permutation(vec.begin(), vec.end());
```

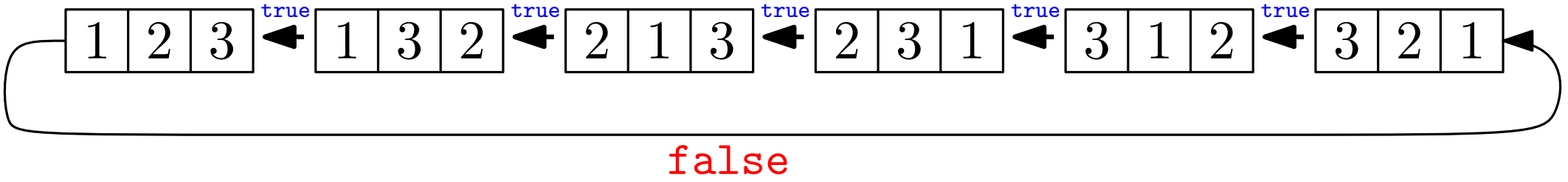


Permutations

- Next/Previous permutation in lexicographic order $O(n)$

```
bool std::next_permutation(vec.begin(), vec.end());
```

```
bool std::prev_permutation(vec.begin(), vec.end());
```



Permutations

- Can v2 be obtained as a permutation of $v1$?

```
bool std::is_permutation(v1.begin(), v1.end(), v2.begin(), v2.end())
```

- Does v1 precede v2 (w.r.t. the lexicographic order)?

```
bool std::lexicographical_compare(v1.begin(), v1.end(),
                                  v2.begin(), v2.end());
```

- Are v1 and v2 the same permutation?

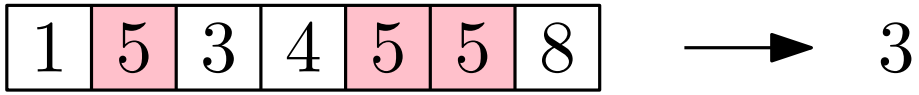
```
bool std::equal(v1.begin(), v1.end(), v2.begin(), v2.end());
```

$$\boxed{1}\boxed{2}\boxed{3} \prec \boxed{1}\boxed{3}\boxed{2} \prec \boxed{2}\boxed{1}\boxed{3} \prec \boxed{2}\boxed{3}\boxed{1} \prec \boxed{3}\boxed{1}\boxed{2} \prec \boxed{3}\boxed{2}\boxed{1}$$

Value Queries & Transformations

- Counting

```
std::count(vec.begin(), vec.end(), 5);
```

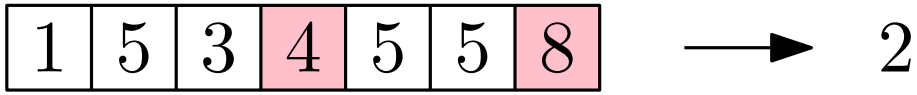


Value Queries & Transformations

- Counting

```
std::count(vec.begin(), vec.end(), 5);
```

```
std::count_if(vec.begin(), vec.end(), [](int i){ return i%2==0; });
```

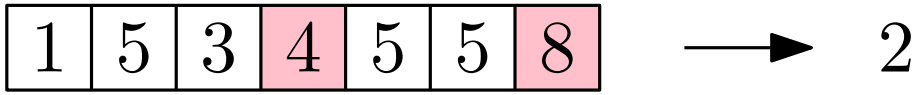


Value Queries & Transformations

- Counting

```
std::count(vec.begin(), vec.end(), 5);
```

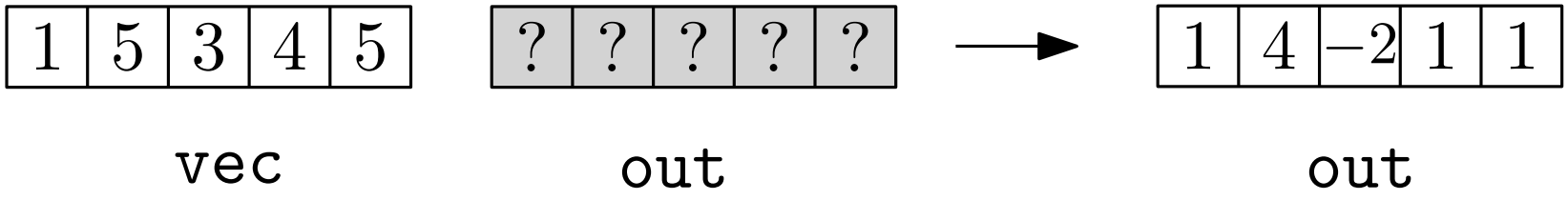
```
std::count_if(vec.begin(), vec.end(), [](int i){ return i%2==0; });
```



- Adjacent differences

```
std::adjacent_difference(vec.begin(), vec.end(), out.begin(), binary_op);
```

$$\texttt{out}[0] = \texttt{vec}[0] \qquad \texttt{out}[\texttt{i}] = \texttt{vec}[\texttt{i}] \ominus \texttt{vec}[\texttt{i}-1]$$



Value Queries & Transformations

- Reduce $\oplus$

```
std::reduce(vec.begin(), vec.end(), init, binary_op);
```

Note: `binary_op` needs to be commutative and associative.

Returns: $\texttt{init} \oplus \left(\bigoplus_{i=0}^{n-1} \texttt{vec[i]} \right)$

Example:

```
std::reduce(vec.begin(), vec.end(), 2, [](int x, int y) { return x*y; });
```

Value Queries & Transformations

- Reduce $\bigoplus$

```
std::reduce(vec.begin(), vec.end(), init, binary_op);
```

Note: `binary_op` needs to be commutative and associative.

Returns: $\texttt{init} \oplus \left(\bigoplus_{i=0}^{n-1} \texttt{vec[i]} \right)$

Example:

```
std::reduce(vec.begin(), vec.end(), 2, [](int x, int y) { return x*y; });
```

$$\boxed{1}\boxed{5}\boxed{3} \longrightarrow 2 \prod_{i=0}^{2} \texttt{vec[i]} = 2 \cdot 1 \cdot 5 \cdot 3 = 30$$

Value Queries & Transformations

- Reduce $\oplus$

```
std::reduce(vec.begin(), vec.end(), init, binary_op);
```

Note: `binary_op` needs to be commutative and associative.

Returns: $\texttt{init} \oplus \left(\bigoplus_{i=0}^{n-1} \texttt{vec[i]} \right)$

Example:

```
std::reduce(vec.begin(), vec.end(), 2, [](int x, int y) { return x*y; });
```

1	5	3	$\longrightarrow$	$2 \prod_{i=0}^{2} \texttt{vec[i]} = 2 \cdot 1 \cdot 5 \cdot 3 = 30$

```
std::reduce(vec.begin(), vec.end());
```

1	5	3	$\longrightarrow$	$\sum_{i=0}^{2} \texttt{vec[i]} = 1 + 5 + 3 = 9$

Value Queries & Tranformations

- Transform (w. unary transform)

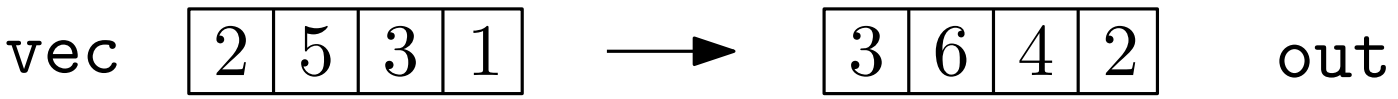
```
std::transform(vec.begin(), vec.end(), out.begin(), unary_op);
```

Applies `unary_op()` to every element of `vec`, stores the results in `out`.

Note: `unary_op` must not have side effect and must not modify the elements of `vec`.

Example:

```
int unary_op(const int x) { return x+1; }
```



Value Queries & Tranformations

- Transform (w. binary transform)

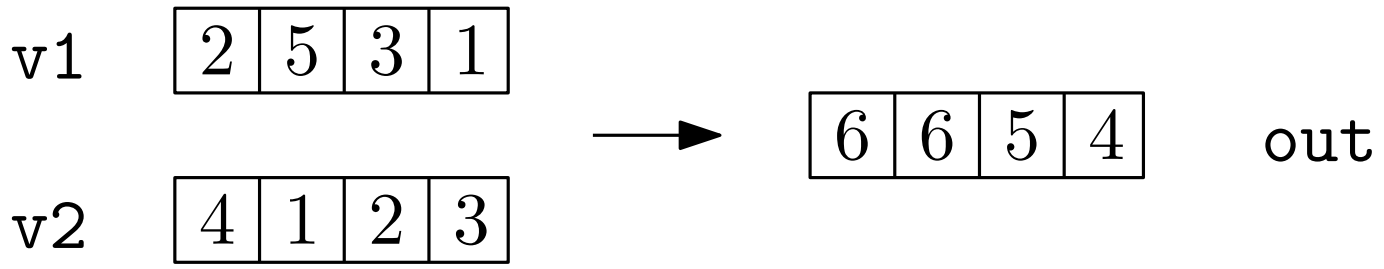
```
std::transform(v1.begin(), v1.end(), v2.begin(), out.begin(), binary_op);
```

Applies `binary_op()` to every pair of corresponding elements of `v1` and `v2`, stores the results in `out`.

Note: `binary_op` must not have side effect and must not modify the elements of `v1` or `v2`.

Example:

```
int binary_op(const int x, const int y) { return x+y; }
```



Value Queries & Transformations

- Transform reduce (w. unary transform)

$\bigoplus$

```
std::transform_reduce(vec.begin(), vec.end(), init, binary_op, unary_op);
```

Note: `binary_op` needs to be commutative and associative.

Returns: $\texttt{init} \oplus \left(\bigoplus_{i=0}^{n-1} \texttt{unary\_op(vec[i])} \right)$

Example:

```
std::transform_reduce(vec.begin(), vec.end(), 1, std::sum<int>,
                                                 [](int x) {return x*x});
```

Value Queries & Transformations

- Transform reduce (w. unary transform) $\bigoplus$

```
std::transform_reduce(vec.begin(), vec.end(), init, binary_op, unary_op);
```

Note: `binary_op` needs to be commutative and associative.

Returns: $\texttt{init} \oplus \left(\bigoplus_{i=0}^{n-1} \texttt{unary\_op(vec[i])} \right)$

Example:

```
std::transform_reduce(vec.begin(), vec.end(), 1, std::sum<int>,
                                                 [](int x) {return x*x});
```

$\boxed{2}\boxed{5}\boxed{3} \longrightarrow 1 + \sum_{i=0}^{2} (\texttt{vec[i]})^2 = 1 + 4 + 25 + 9 = 39$

Value Queries & Transformations

- Transform reduce (w. binary transform)

$\oplus$ $\otimes$

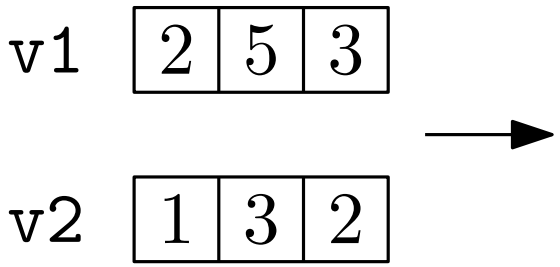
```
std::transform_reduce(v1.begin(), v1.end(), v2.begin(), init, op1, op2);
```

Note: op1 and op2 need to be commutative and associative.

Returns: $\texttt{init} \oplus \left(\bigoplus_{i=0}^{n-1} \texttt{v1}[\texttt{i}] \otimes \texttt{v2}[\texttt{i}] \right)$

Example:

```
std::transform_reduce(v1.begin(), v1.end(), v2.begin(), 1,
                                    std::plus<int>, std::multiplies<int>);
```



Value Queries & Transformations

- Transform reduce (w. binary transform)

$\oplus$ $\otimes$

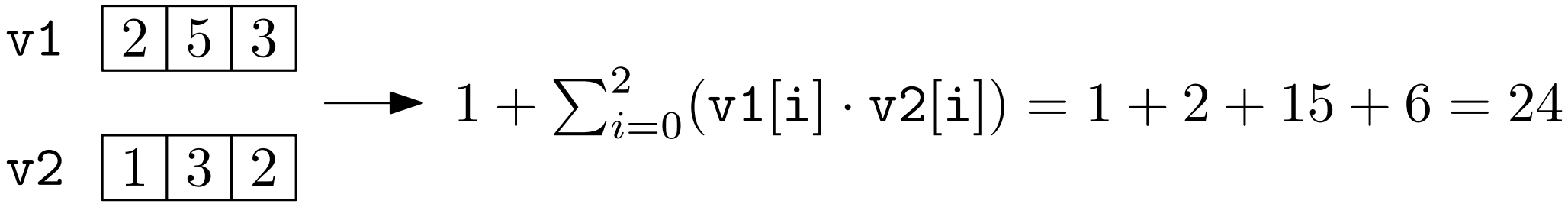
```
std::transform_reduce(v1.begin(), v1.end(), v2.begin(), init, op1, op2);
```

Note: op1 and op2 need to be commutative and associative.

Returns: $\texttt{init} \oplus \left(\bigoplus_{i=0}^{n-1} \texttt{v1[i]} \otimes \texttt{v2[i]} \right)$

Example:

```
std::transform_reduce(v1.begin(), v1.end(), v2.begin(), 1,
                                    std::plus<int>, std::multiplies<int>);
```



Value Queries & Tranformations

- Inclusive scan $\oplus$

```
std::inclusive_scan(v.begin(), v.end(), out.begin(), binary_op, init);
```

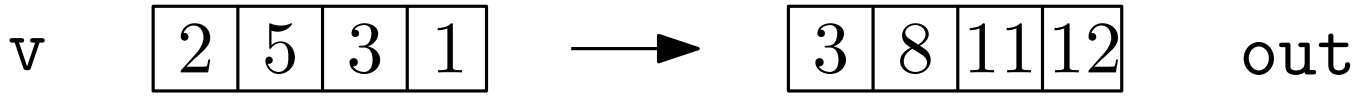
Note: `binary_op` needs to be commutative and associative.

For every $j = 0, 1, \ldots,$ `v.size()-1`, sets:

$$\texttt{out[j]} = \texttt{init} \oplus \left(\bigoplus_{i=0}^{j} \texttt{v[i]} \right)$$

Example:

```
std::inclusive_scan(v.begin(), v.end(), out.begin(), std::plus<>, 1);
```



Value Queries & Tranformations

- Exclusive scan $\oplus$

```
std::exclusive_scan(v.begin(), v.end(), out.begin(), binary_op, init);
```

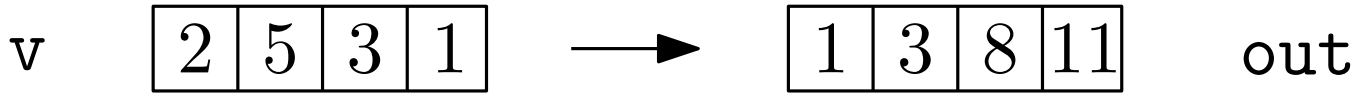
Note: `binary_op` needs to be commutative and associative.

For every $j = 0, 1, \ldots,$ `v.size()-1`, sets:

$$\texttt{out[j]} = \texttt{init} \oplus \left(\bigoplus_{i=0}^{j-1} \texttt{v[i]} \right)$$

Example:

```
std::exclusive_scan(v.begin(), v.end(), out.begin(), std::plus<>, 1);
```



Value Queries & Tranformations

- All of: $\forall x \in \texttt{vec},\ \texttt{p}(x)$?

```
bool std::all_of(vec.begin(), vec.end(), p);
```

- Any of: $\exists x \in \texttt{vec}\ :\ \texttt{p}(x)$?

```
bool std::any_of(vec.begin(), vec.end(), p);
```

- None of: $\nexists x \in \texttt{vec}\ :\ \texttt{p}(x)$? $\quad \forall x \in \texttt{vec}\ :\ \neg\texttt{p}(x)$?

```
bool std::none_of(vec.begin(), vec.end(), p);
```

Value Queries & Tranformations

- All of: $\forall x \in \texttt{vec},\ \texttt{p}(x)$?

```
bool std::all_of(vec.begin(), vec.end(), p);
```

- Any of: $\exists x \in \texttt{vec}\ :\ \texttt{p}(x)$?

```
bool std::any_of(vec.begin(), vec.end(), p);
```

- None of: $\nexists x \in \texttt{vec}\ :\ \texttt{p}(x)$? $\quad \forall x \in \texttt{vec}\ :\ \neg\texttt{p}(x)$?

```
bool std::none_of(vec.begin(), vec.end(), p);
```

Example:

```
bool p(int x) { return x%3 = 0 };
```

1	5	3	2	8	4

Value Queries & Tranformations

- All of: $\forall x \in \texttt{vec},\ \texttt{p}(x)$?

```
bool std::all_of(vec.begin(), vec.end(), p);
```

- Any of: $\exists x \in \texttt{vec}\ :\ \texttt{p}(x)$?

```
bool std::any_of(vec.begin(), vec.end(), p);
```

- None of: $\nexists x \in \texttt{vec}\ :\ \texttt{p}(x)$? $\quad \forall x \in \texttt{vec}\ :\ \neg\texttt{p}(x)$?

```
bool std::none_of(vec.begin(), vec.end(), p);
```

Example:

```
bool p(int x) { return x%3 = 0 };
```

1	5	3	2	8	4

```
all_of(..., p) = false
any_of(..., p) = true
none_of(..., p) = false
```

Value Queries & Tranformations

- All of: $\forall x \in \texttt{vec},\ \texttt{p}(x)$?

```
bool std::all_of(vec.begin(), vec.end(), p);
```

- Any of: $\exists x \in \texttt{vec}\ :\ \texttt{p}(x)$?

```
bool std::any_of(vec.begin(), vec.end(), p);
```

- None of: $\nexists x \in \texttt{vec}\ :\ \texttt{p}(x)$? $\quad \forall x \in \texttt{vec}\ :\ \neg\texttt{p}(x)$?

```
bool std::none_of(vec.begin(), vec.end(), p);
```

Example:

```
bool p(int x) { return x%3 = 0 };
```

```
std::vector<int> vec();
```

Value Queries & Tranformations

- All of: $\forall x \in \texttt{vec},\ \texttt{p}(x)$?

```
bool std::all_of(vec.begin(), vec.end(), p);
```

- Any of: $\exists x \in \texttt{vec}\ :\ \texttt{p}(x)$?

```
bool std::any_of(vec.begin(), vec.end(), p);
```

- None of: $\nexists x \in \texttt{vec}\ :\ \texttt{p}(x)$? $\quad \forall x \in \texttt{vec}\ :\ \neg\texttt{p}(x)$?

```
bool std::none_of(vec.begin(), vec.end(), p);
```

Example:

```
bool p(int x) { return x%3 = 0 };
```

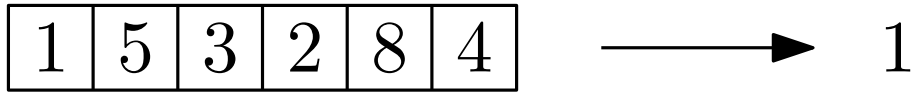
```
std::vector<int> vec();
```

```
all_of(..., p) = true
any_of(..., p) = false
none_of(..., p) = true
```

Linear Search

- Minimum/Maximum of a collection

```
std::min_element(vec.begin(), vec.end());
```



Linear Search

- Minimum/Maximum of a collection

```
std::min_element(vec.begin(), vec.end());
```

```
std::max_element(vec.begin(), vec.end());
```

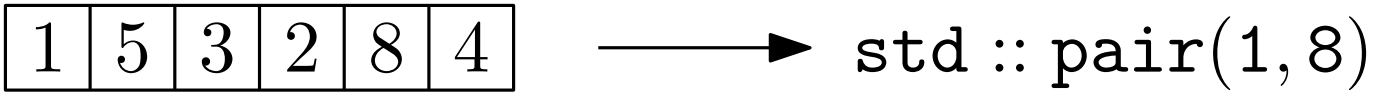
Linear Search

- Minimum/Maximum of a collection

```
std::min_element(vec.begin(), vec.end());
```

```
std::max_element(vec.begin(), vec.end());
```

```
std::minmax_element(vec.begin(), vec.end());
```



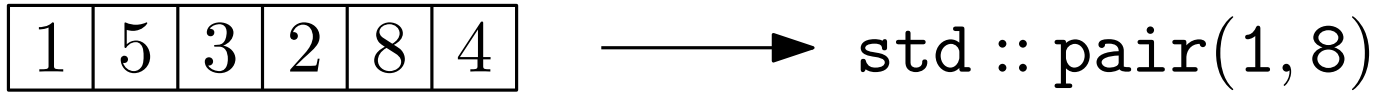
Linear Search

- Minimum/Maximum of a collection

```
std::min_element(vec.begin(), vec.end());
```

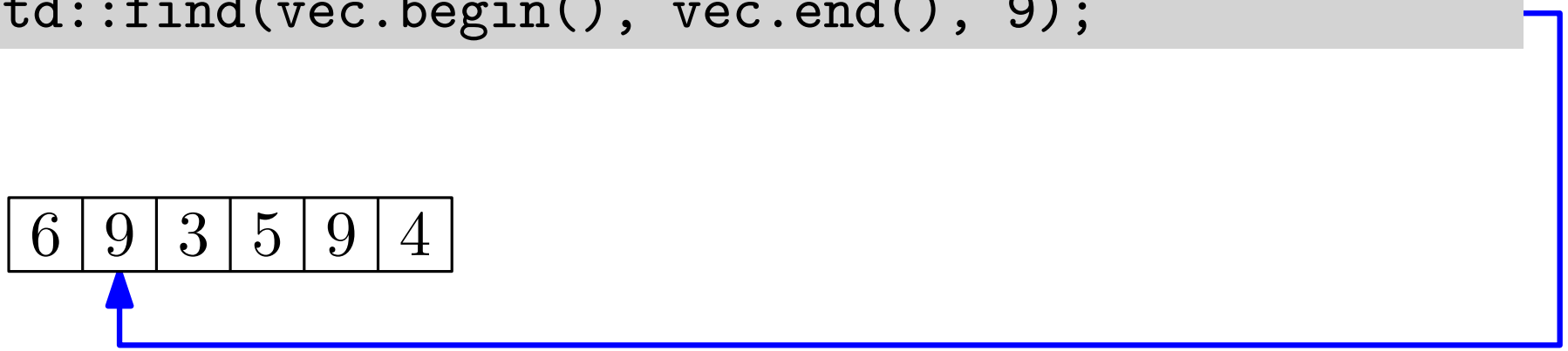
```
std::max_element(vec.begin(), vec.end());
```

```
std::minmax_element(vec.begin(), vec.end());
```



- Searching for an element

```
std::find(vec.begin(), vec.end(), 9);
```



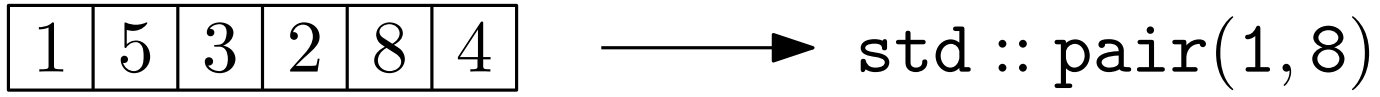
Linear Search

- Minimum/Maximum of a collection

```
std::min_element(vec.begin(), vec.end());
```

```
std::max_element(vec.begin(), vec.end());
```

```
std::minmax_element(vec.begin(), vec.end());
```



- Searching for an element

```
std::find(vec.begin(), vec.end(), 2);
```

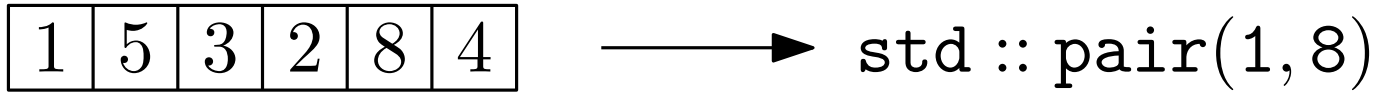
Linear Search

- Minimum/Maximum of a collection

```
std::min_element(vec.begin(), vec.end());
```

```
std::max_element(vec.begin(), vec.end());
```

```
std::minmax_element(vec.begin(), vec.end());
```



- Searching for an element

```
std::find(vec.begin(), vec.end(), 2);
```

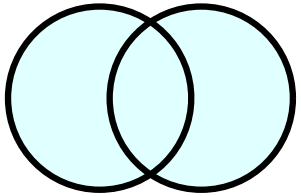
```
std::find_if(vec.begin(), vec.end(), [](int x) { return x%3; });
```

Set Operations

Note: For the purpose of the following operations, sets are just sorted containers (not necessarily `std::set`).

Set Operations

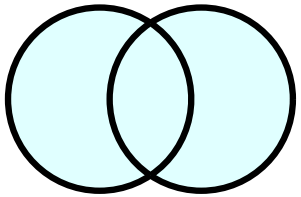
Note: For the purpose of the following operations, sets are just sorted containers (not necessarily `std::set`).



```
std::set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
```

Set Operations

Note: For the purpose of the following operations, sets are just sorted containers (not necessarily `std::set`).



```
std::set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
```

Example:

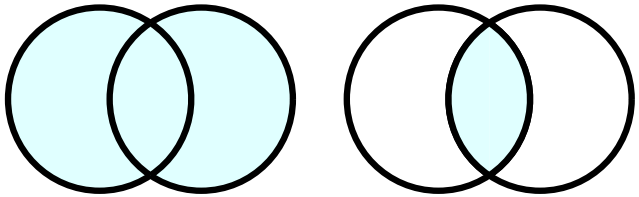
```
std::vector<int> v1 { 1, 2, 3 };
std::vector<int> v2 { 2, 4, 5};
std::vector<int> out();

std::set_union(v1.begin(), v1.end(), v2.begin(), v2.end(),
                                              std::back_inserter(out));

// out contains 1, 2, 3, 4, 5
```

Set Operations

Note: For the purpose of the following operations, sets are just sorted containers (not necessarily `std::set`).

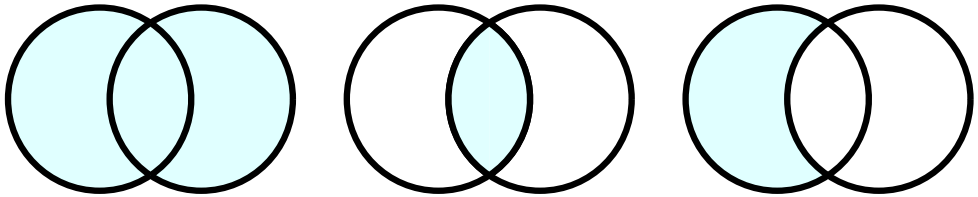


```
std::set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
```

```
std::set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
```

Set Operations

Note: For the purpose of the following operations, sets are just sorted containers (not necessarily `std::set`).



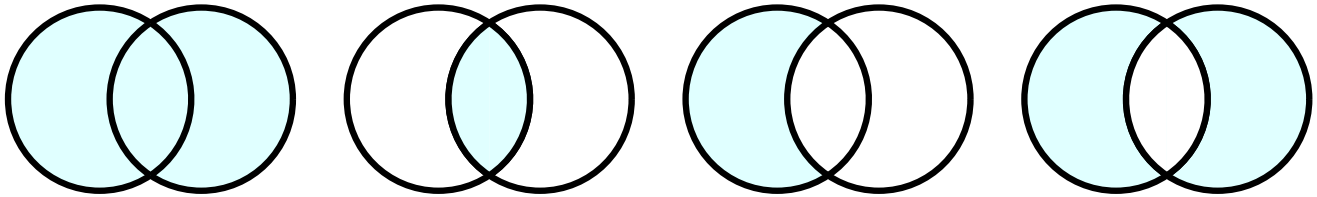
```
std::set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
```

```
std::set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
```

```
std::set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
```

Set Operations

Note: For the purpose of the following operations, sets are just sorted containers (not necessarily `std::set`).



```
std::set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
```

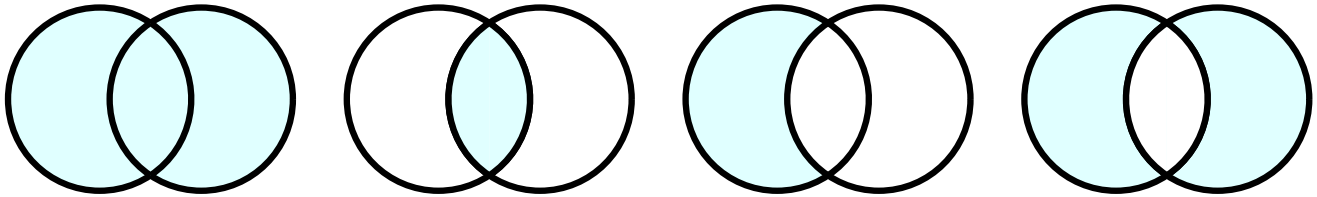
```
std::set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
```

```
std::set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
```

```
std::set_symmetric_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),
                                                                out_it);
```

Set Operations

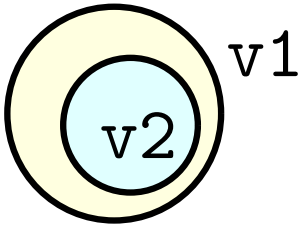
Note: For the purpose of the following operations, sets are just sorted containers (not necessarily `std::set`).



```
std::set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
std::set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
std::set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), out_it);
std::set_symmetric_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),
                              out_it);
```

```
bool std::includes(v1.begin(), v1.end(), v2.begin(), v2.end());
```

Returns: `true` iff `v2` is a subset of `v1`



Binary Search

- Is 5 in the collection?

```
bool std::binary_search(vec.begin(), vec.end(), 6);
```

Binary Search

- Is 5 in the collection?

```
bool std::binary_search(vec.begin(), vec.end(), 6);
```

- First index i s.t. $v[i] \geq 3$ (Where should 3 be inserted?)

```
std::lower_bound(vec.begin(), vec.end(), 3);
```

Binary Search

- Is 5 in the collection?

```
bool std::binary_search(vec.begin(), vec.end(), 6);
```

- First index i s.t. $v[i] \geq 3$ (Where should 3 be inserted?)

```
std::lower_bound(vec.begin(), vec.end(), 3);
```

- First index i s.t. $v[i] > 4$

```
std::upper_bound(vec.begin(), vec.end(), 4);
```

Binary Search

- Is 5 in the collection?

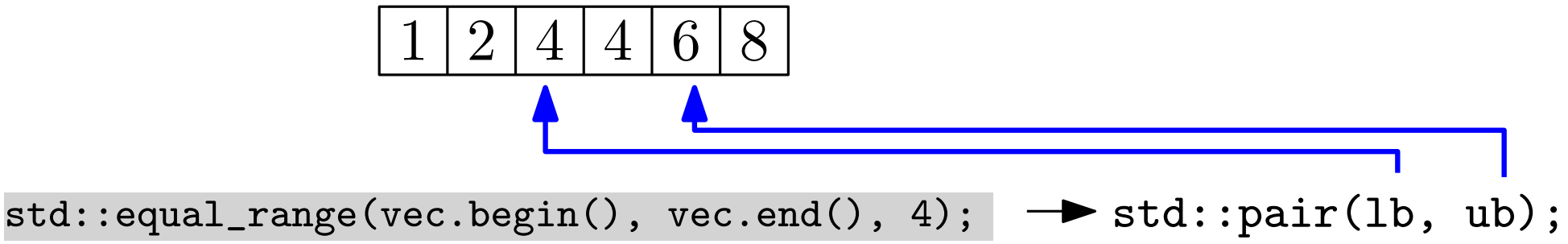
```
bool std::binary_search(vec.begin(), vec.end(), 6);
```

- First index i s.t. $v[i] \geq 3$ (Where should 3 be inserted?)

```
std::lower_bound(vec.begin(), vec.end(), 3);
```

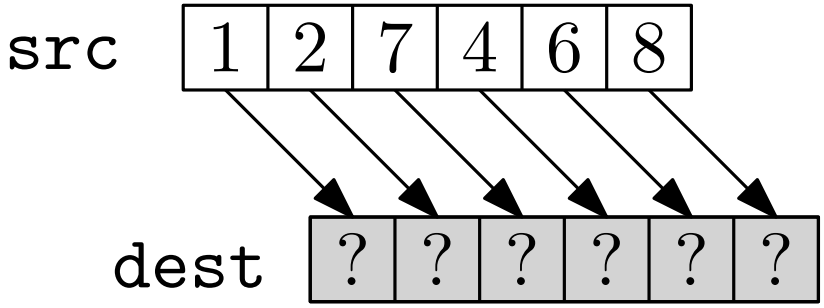
- First index i s.t. $v[i] > 4$

```
std::upper_bound(vec.begin(), vec.end(), 4);
```



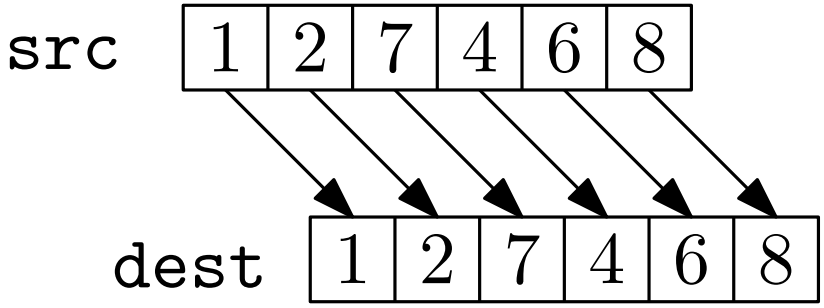
Copying collections

```
std::copy(src.begin(), src.end(), dest.begin());
```



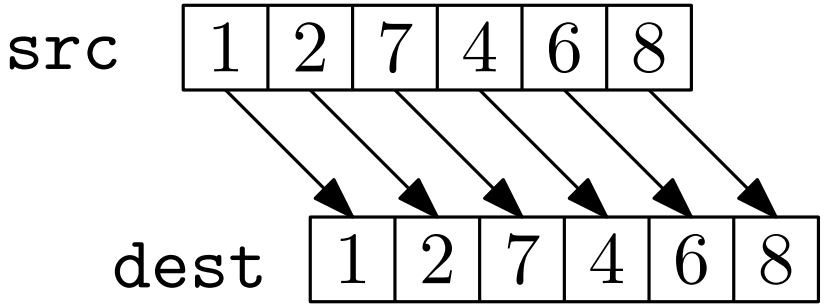
Copying collections

```
std::copy(src.begin(), src.end(), dest.begin());
```

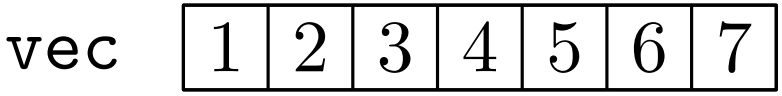


Copying collections

```
std::copy(src.begin(), src.end(), dest.begin());
```

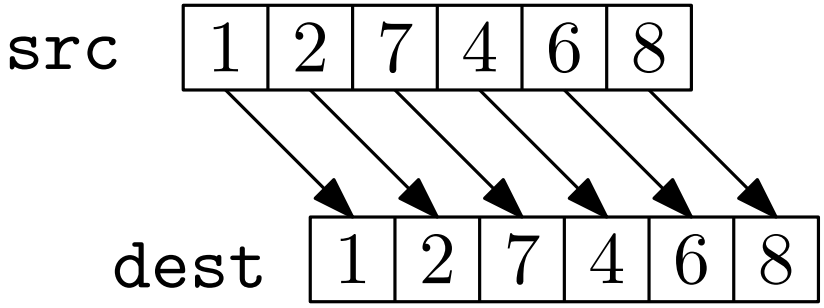



```
std::copy(vec.begin(), vec.begin()+4, vec.begin()+2);
```

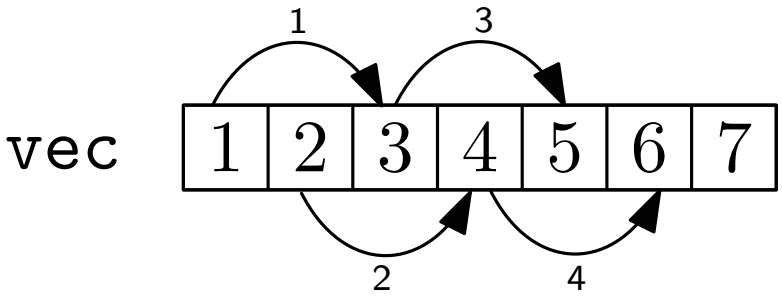


Copying collections

```
std::copy(src.begin(), src.end(), dest.begin());
```

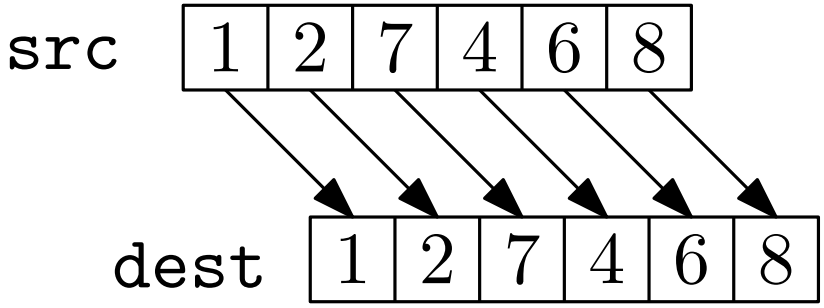



```
std::copy(vec.begin(), vec.begin()+4, vec.begin()+2);
```

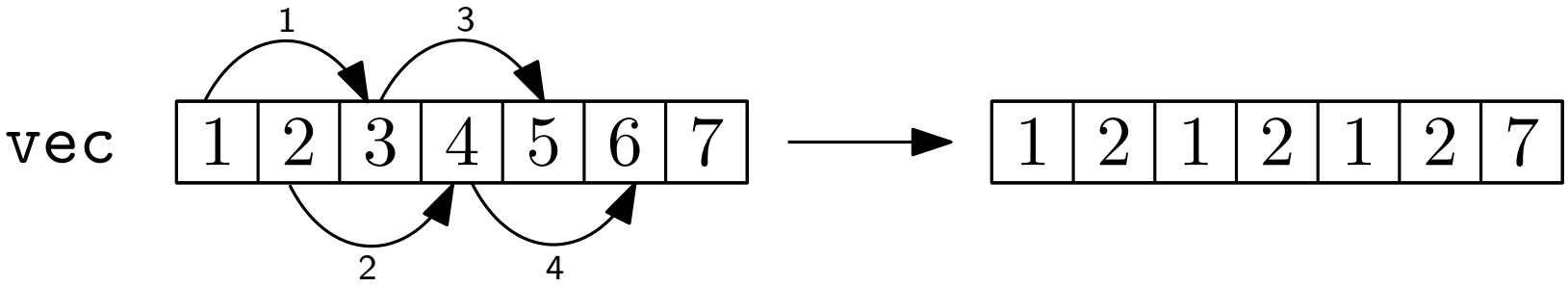


Copying collections

```
std::copy(src.begin(), src.end(), dest.begin());
```

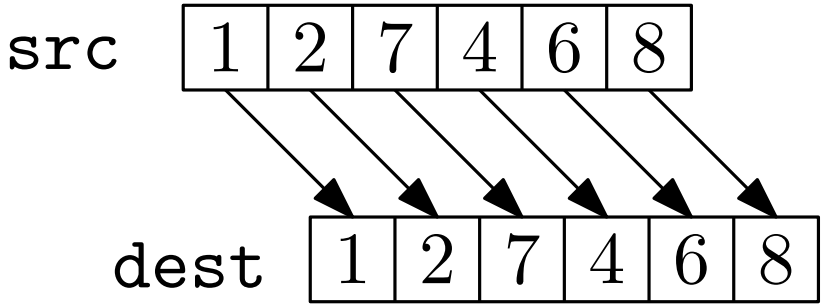



```
std::copy(vec.begin(), vec.begin()+4, vec.begin()+2);
```

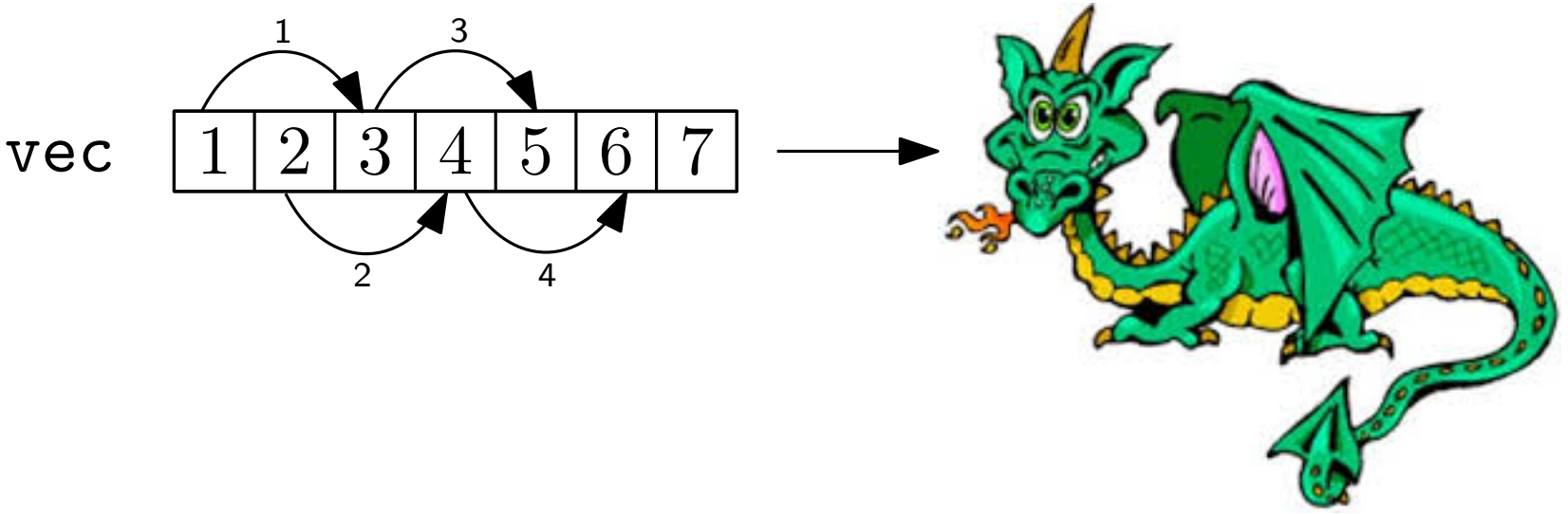


Copying collections

```
std::copy(src.begin(), src.end(), dest.begin());
```

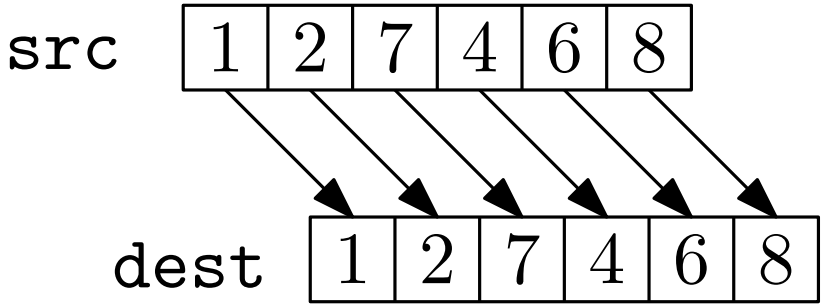



```
std::copy(vec.begin(), vec.begin()+4, vec.begin()+2);
```

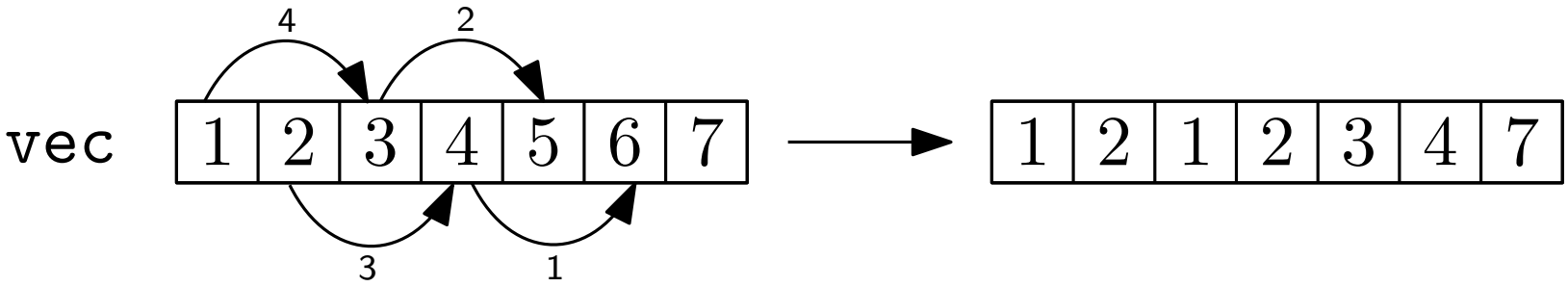


Copying collections

```
std::copy(src.begin(), src.end(), dest.begin());
```

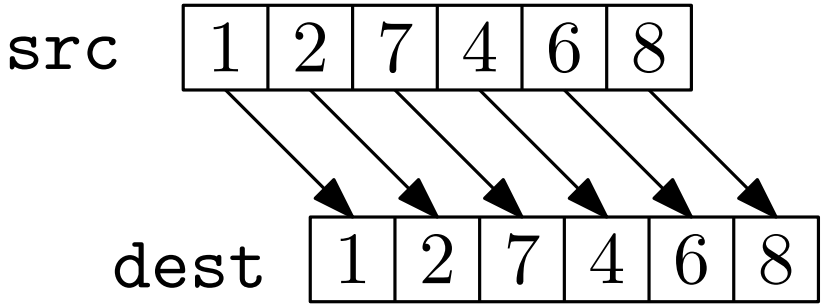



```
std::copy_backward(vec.begin(), vec.begin()+4, vec.begin()+5);
```

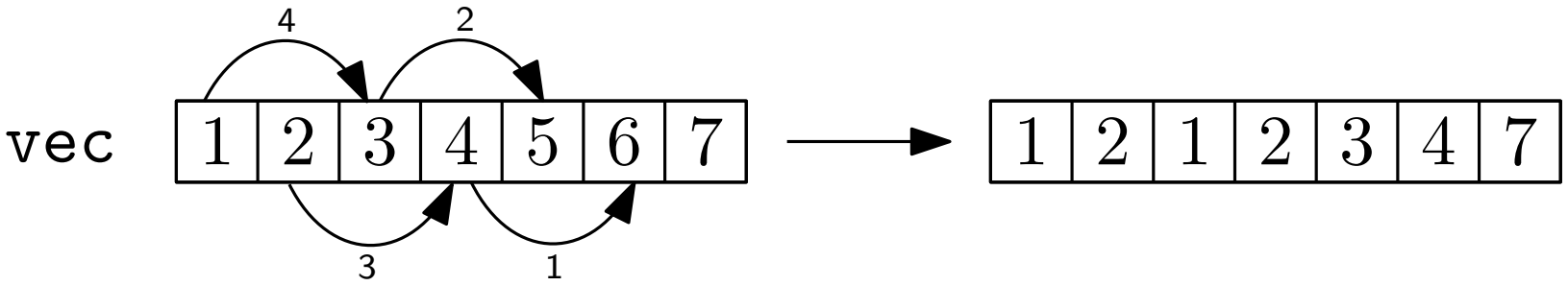


Copying collections

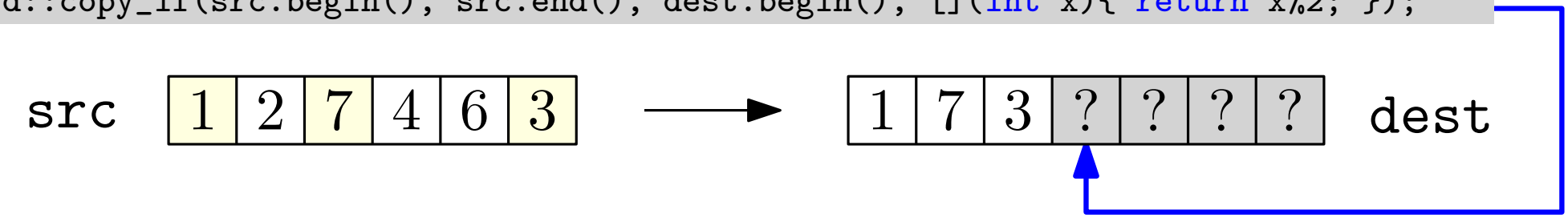
```
std::copy(src.begin(), src.end(), dest.begin());
```




```
std::copy_backward(vec.begin(), vec.begin()+4, vec.begin()+5);
```

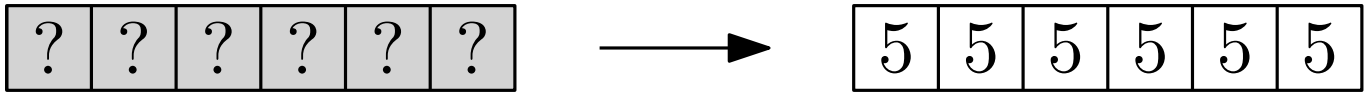



```
std::copy_if(src.begin(), src.end(), dest.begin(), [](int x){ return x%2; });
```

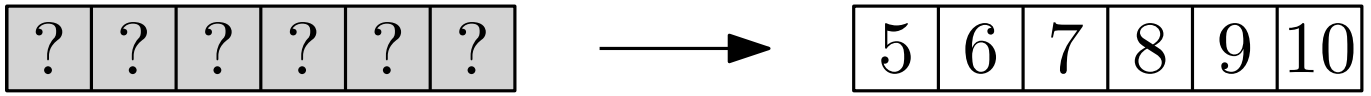


Initializing collections

```
std::fill(vec.begin(), vec.end(), 5);
```




```
std::iota(vec.begin(), vec.end(), 5);
```



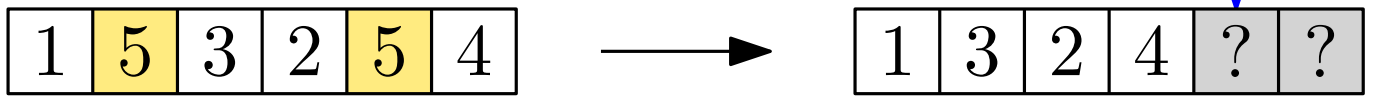

```
std::generate(vec.begin(), vec.end(), f);
```

Shorthand for:

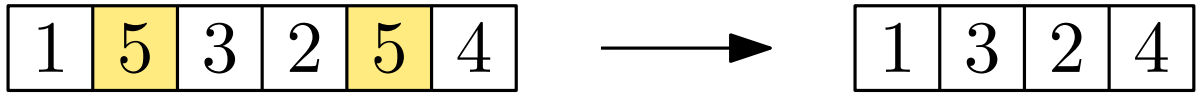
```
vec[0] = f();
vec[1] = f();
...
```

Deleting elements

```
std::remove(vec.begin(), vec.end(), 5);
```

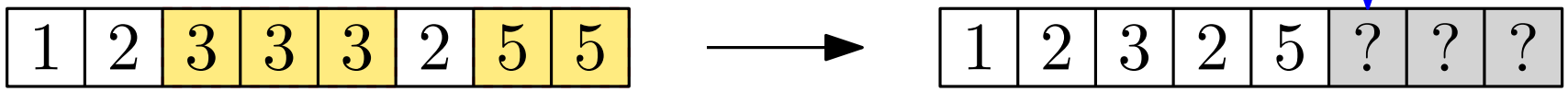



```
vec.erase(std::remove(vec.begin(), vec.end(), 5), vec.end());
```



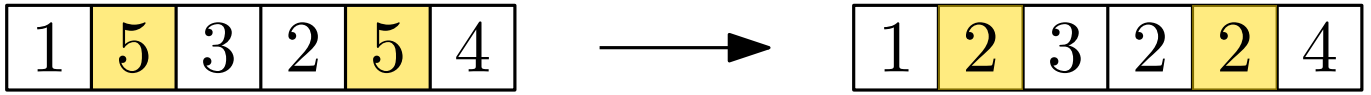
See also: `std::remove_if`.

```
std::unique(vec.begin(), vec.end());
```

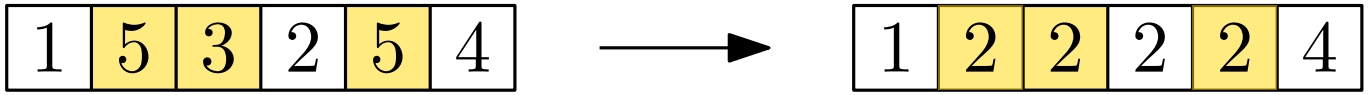


Replacing elements

```
std::replace(vec.begin(), vec.end(), 5, 2);
```




```
std::replace_if(vec.begin(), vec.end(), [](int x) { return x%2; }, 2);
```



Foreach

- Executes `f()` on every element between between `vec.begin()` (inclusive) and `vec.end()` (exclusive), in order.

```
std::foreach(vec.begin(), vec.end(), f);
```

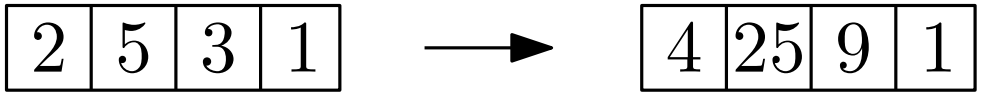
Foreach

- Executes `f()` on every element between between `vec.begin()` (inclusive) and `vec.end()` (exclusive), in order.

```
std::foreach(vec.begin(), vec.end(), f);
```

Example:

```
void f(int& x) { x = x*x; }
```



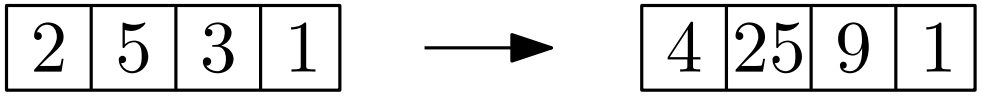
Foreach

- Executes `f()` on every element between between `vec.begin()` (inclusive) and `vec.end()` (exclusive), in order.

```
std::foreach(vec.begin(), vec.end(), f);
```

Example:

```
void f(int& x) { x = x*x; }
```



Consider using range-based for loops, when appropriate:

```
for(int& x : vec) { x = x*x; }
```