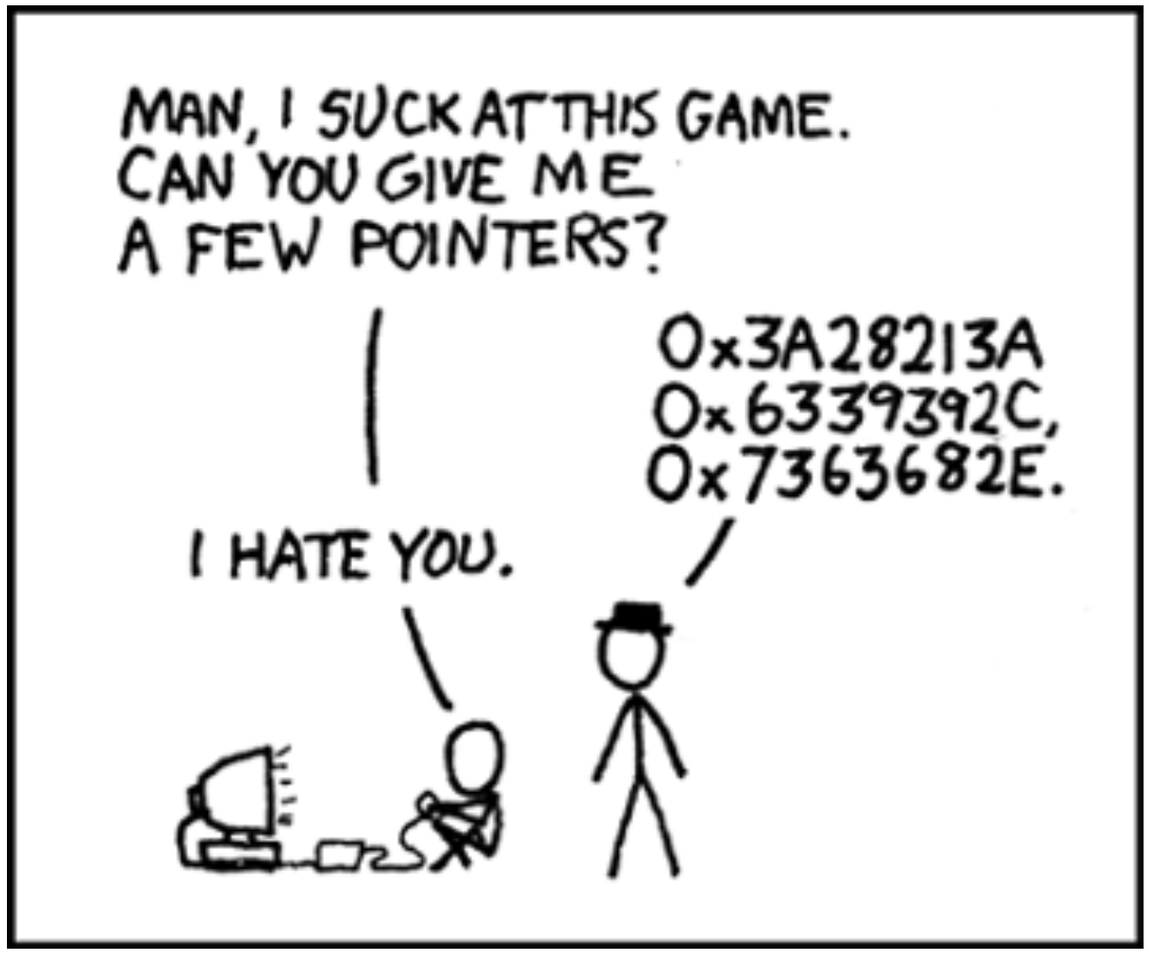
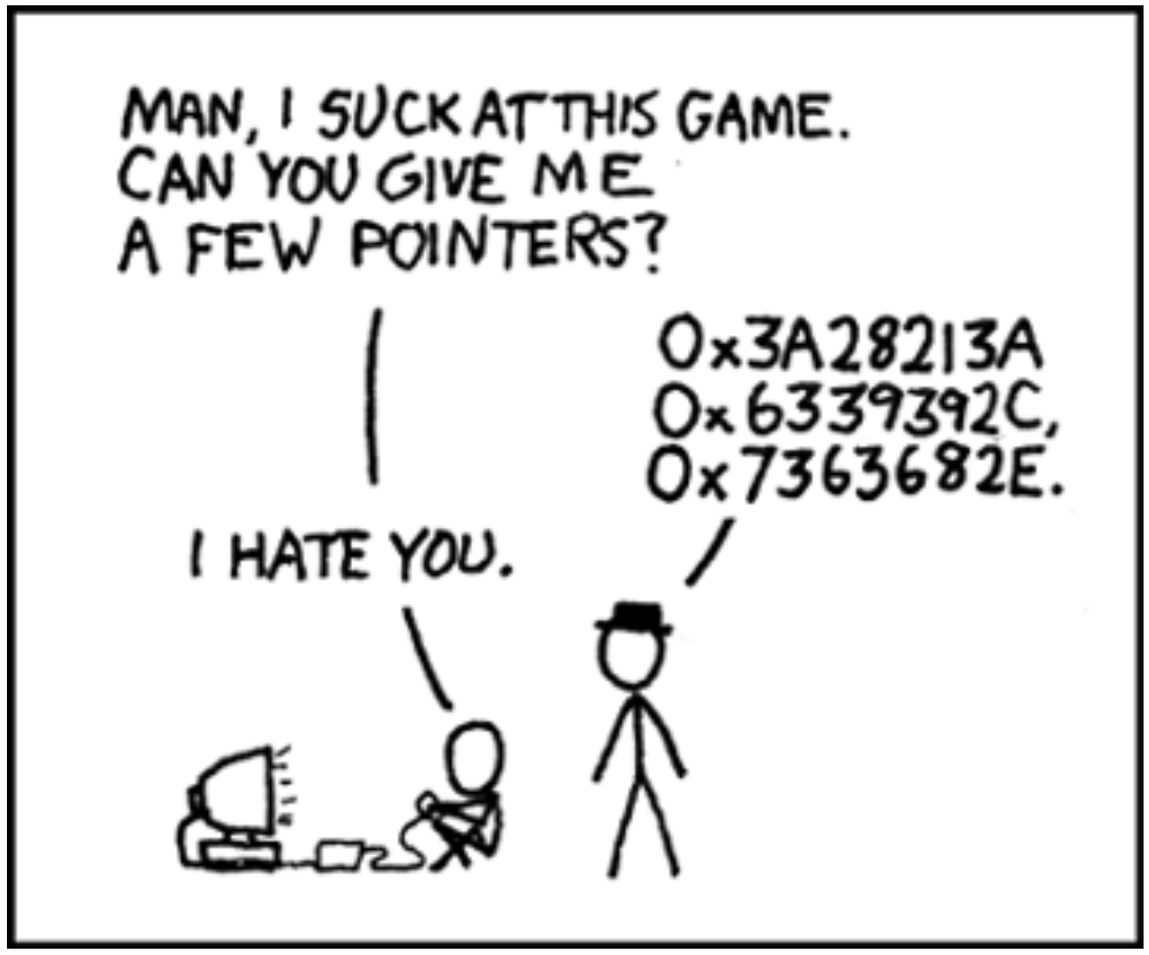
179 Range Algorithms in Less than an Hour



https://www.xkcd.com/138/

Dvir Yitzchaki
Core C++ 2019

~~179~~ 184 Range Algorithms in Less than an Hour



https://www.xkcd.com/138/

Dvir Yitzchaki
Core C++ 2019

Jonathan's Boccara's world of STL algorithms

About me

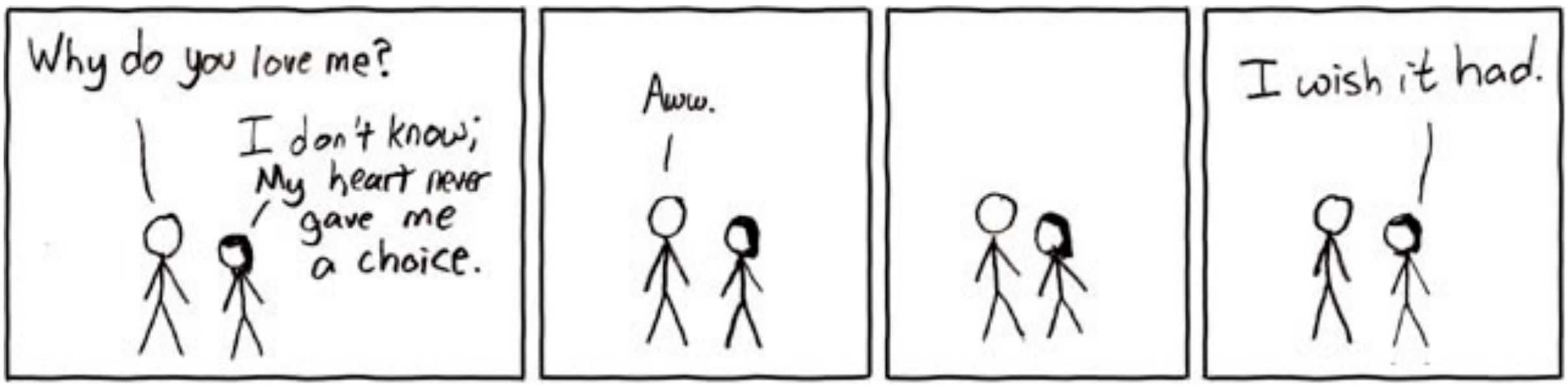
- Sr. data engineer at Verizon Media
- Casual speaker at Core C++ meetup
- An XCKD fan
- dvirtz at GitHub, slack and gmail
- @dvirtzwastaken on Twitter

https://xkcd.com/676/

Agenda

- Motivation
- A bit of history
- Concepts
- Range Algorithms
- Projections
- Views
- Composition
- Actions
- Performance

Motivation



https://www.xkcd.com/58/

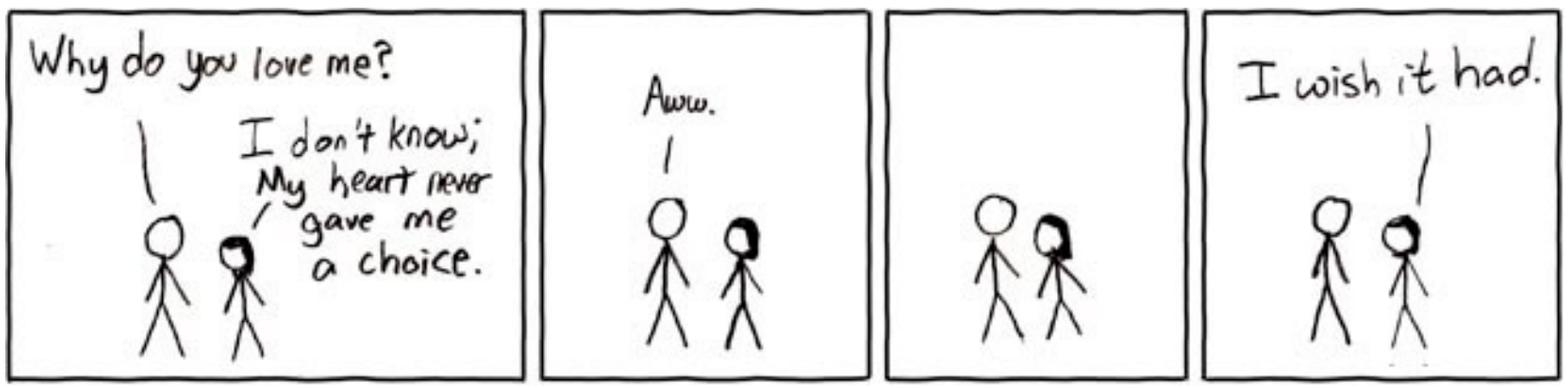
Motivation

- One prime motivation for ranges is to give users a simpler syntax for calling algorithms. Rather than this:

```
std::vector<int> v { /*...*/ };
std::sort(v.begin(), v.end());
```

write this:

```
std::ranges::sort(v);
```

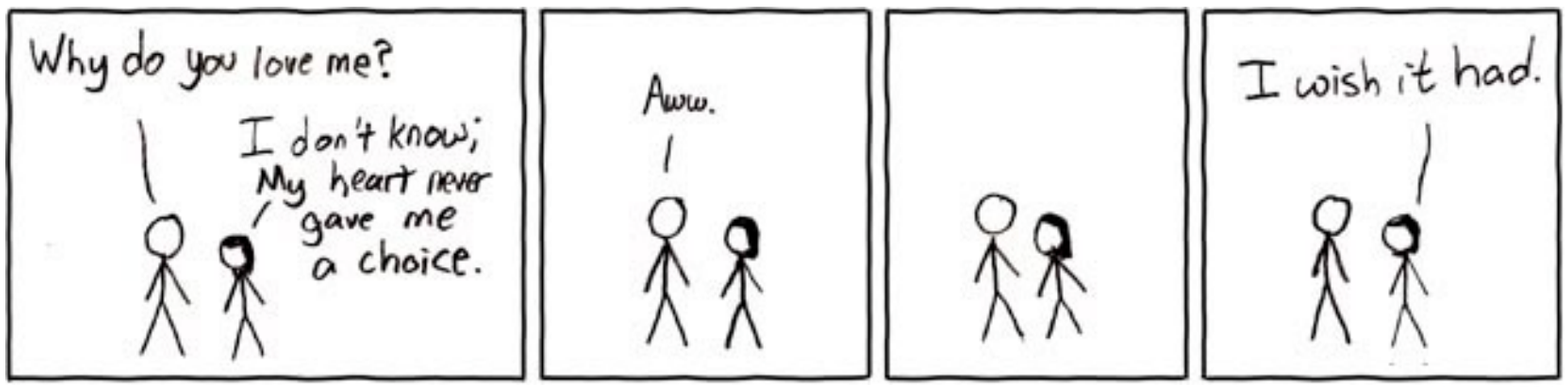


https://www.xkcd.com/58/

Motivation

- Allowing algorithms to take a single range object instead of separate begin and end iterators opens the door to range adaptors which lazily transform or filter their underlying sequence in interesting ways:

```
accumulate(view::iota(1)
  | view::transform([](int x) { return x * x; })
  | view::take(10), 0);
```



https://www.xkcd.com/58/

What is a range

- A sequence of elements between two locations i, k.
- Often denoted by [i, k)

What is a range

- From the perspective of the standard library, a range is a pair of iterators.

```
std::copy(v.begin(), v.end(), buf);
```

- But there are other interesting ways to denote a range of elements:
  - An iterator and a count of elements

```
std::copy_n(v.begin(), 20, buf);
```

  - An iterator and a (possibly stateful) predicate that indicates when the range is exhausted.

```
std::copy(std::istream_iterator<int>{std::cin},
          std::istream_iterator<int>{},
          buf);
```

- The ranges library design captures all of these forms in one concept

A bit of history

https://www.xkcd.com/1979/

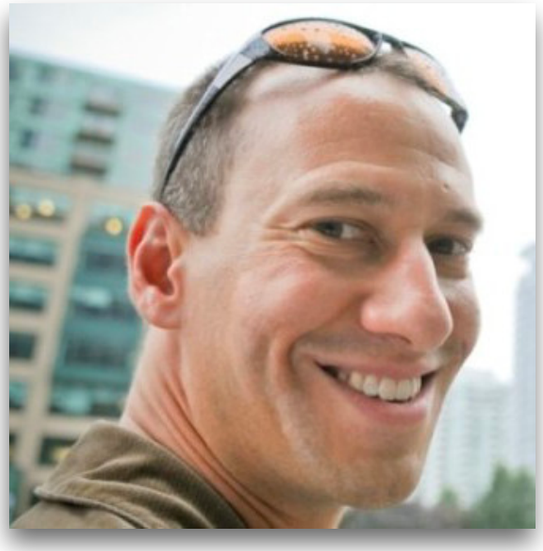
A bit of history

- November 2004 - Boost.Range by Niel Groves and Thorsten Ottosen
- May 2010 - Boost.Range 2.0
- October 2013 - Eric Niebler starts blogging about ranges
- November 2013 - First commit to range-v3
- October 2014 - Ranges for the Standard Library
- July 2017 - Ranges TS
- November 2018 - Merged to C++20

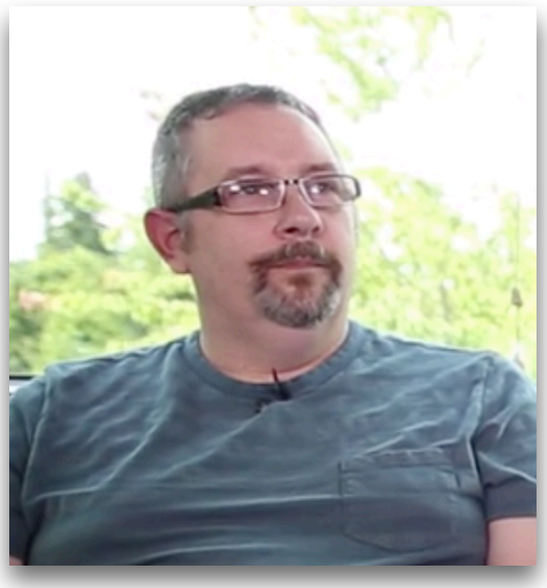
https://www.xkcd.com/1979/

Implementations

- range-v3 by Eric Niebler: https://github.com/ericniebler/range-v3
  - C++11 with optional features for C++14/17
  - Emulated concepts (std::enable_if)
  - Minimum compiler versions:
    - clang 3.6.2 (or later)
    - GCC 4.9.1 (or later)
    - MSVC VS2017 15.9, with `/std:c++17 /permissive-`



- cmcstl2 by Casey Carter: https://github.com/CaseyCarter/cmcstl2
  - Implementation of the Ranges TS
  - Requires GCC 7+ with `-std=c++1z -fconcepts`
- Both are available in Compiler Explorer



Availability

- In general, most algorithms are available in C++20. The ones that are not will be denoted by 

.
- Only a few views are available in C++20 and they will be denoted by 
.
- Stuff that is proposed to the standard but not voted in yet will be denoted with the proposal number as in P1243 .

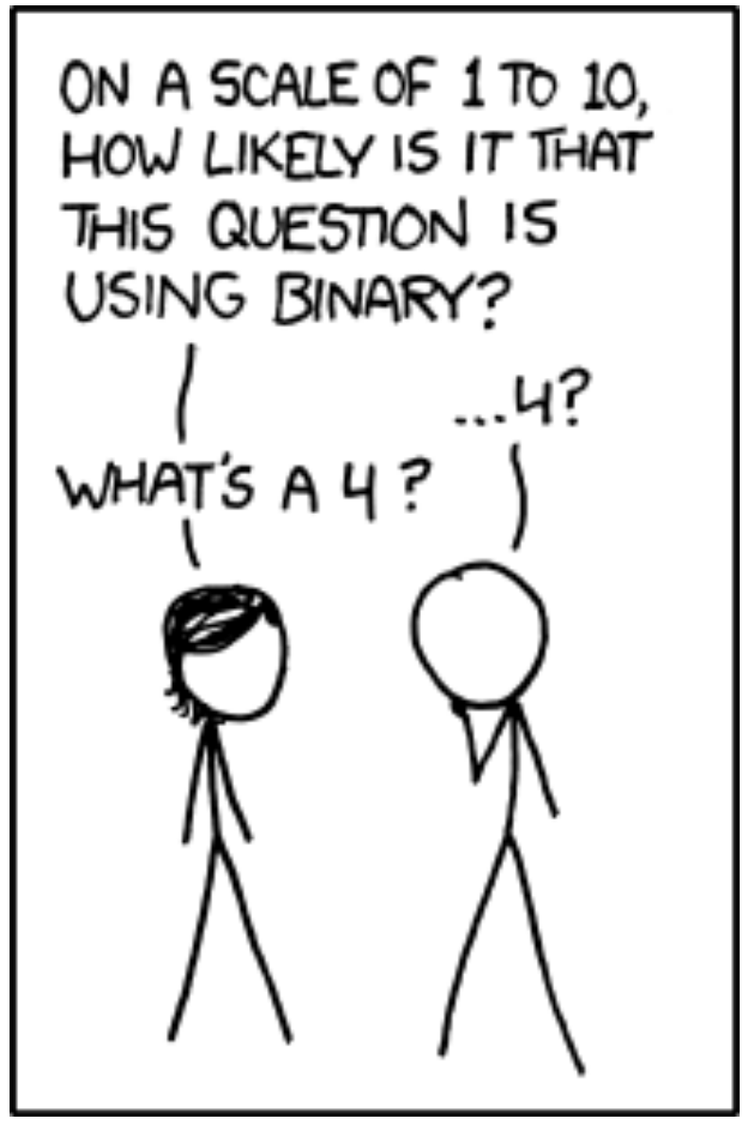
Conventions

- `copy(_n)` denotes the two overloads `copy` and `copy_n`.

- Code assumes

```
using namespace (std::)ranges;
```

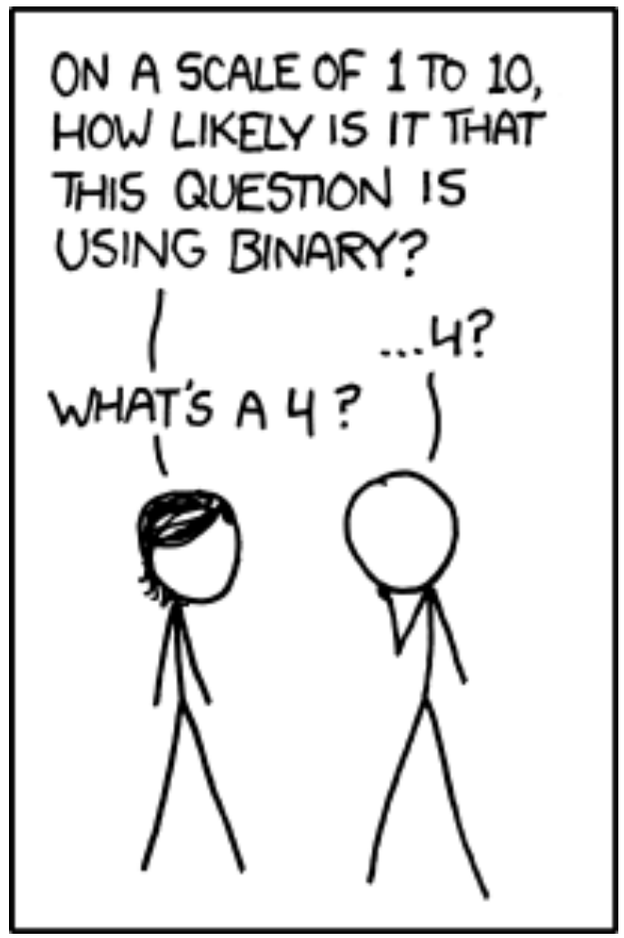
Concepts



https://www.xkcd.com/953/

std::for_each

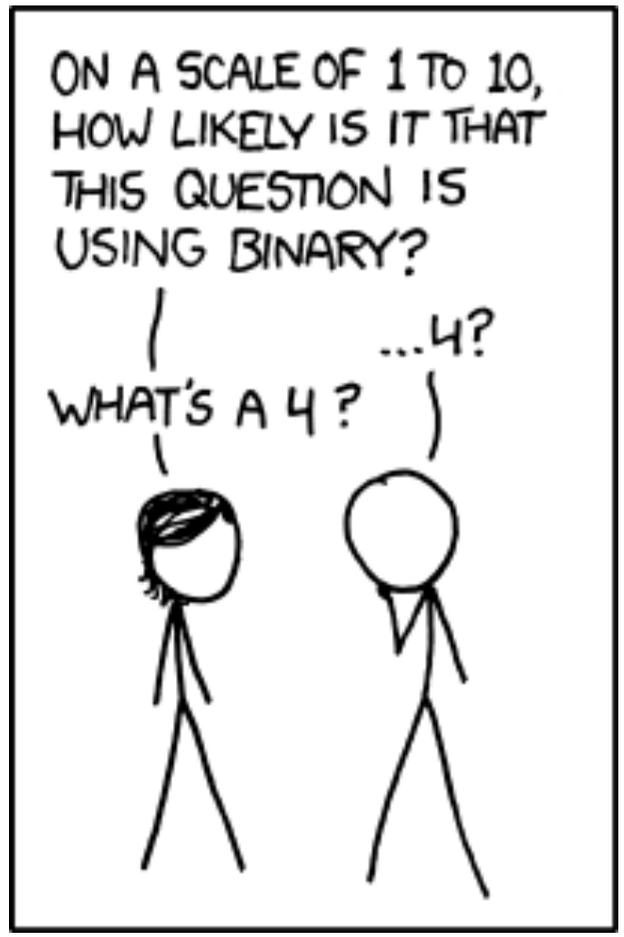
```
template<class InputIt, class UnaryFunction>
constexpr ? for_each(InputIt first, InputIt last, UnaryFunction f);
```



https://www.xkcd.com/953/

std::for_each

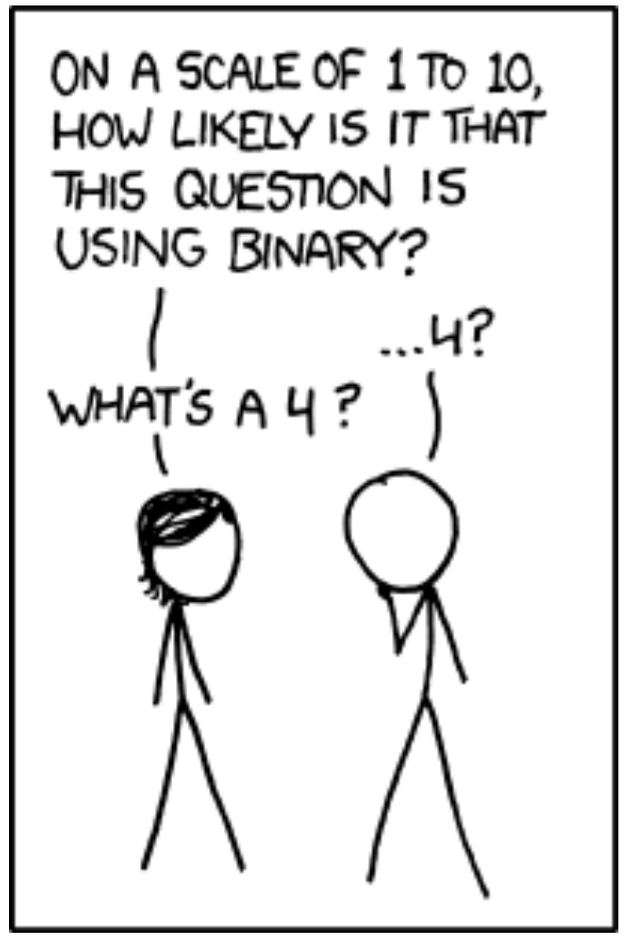
```
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```



https://www.xkcd.com/953/

std::for_each

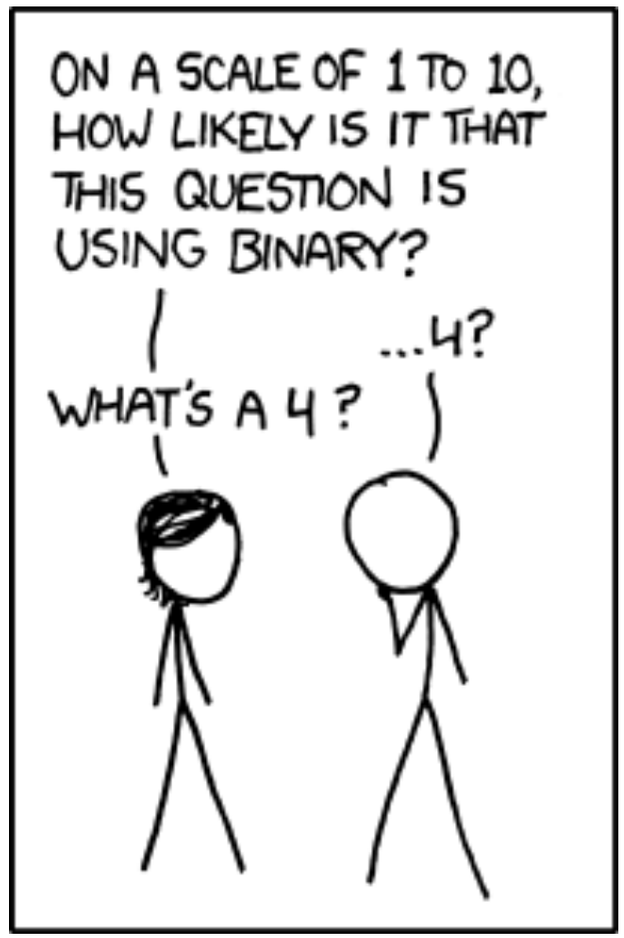
```
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```



https://www.xkcd.com/953/

std::for_each

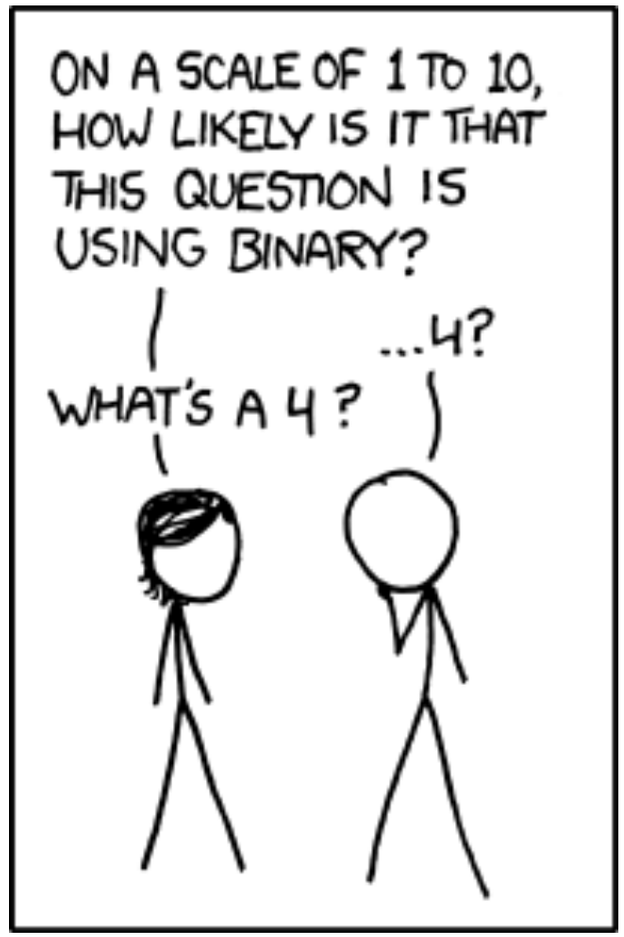
```cpp
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```



https://www.xkcd.com/953/

std::for_each

```
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```



https://www.xkcd.com/953/

Iterator Concepts

- **`Iterator`** - can be dereferenced (`*it`) and incremented (`++it`)
- **`InputIterator`** - referenced values can be read
  (`auto v = *it`)
- **`OutputIterator`** - referenced values can be written to
  (`*it = v`)

- **`ForwardIterator`** - `InputIterator` + comparable and multi-pass
- **`BidirectionalIterator`** - `ForwardIterator` + decrementable (`--it`)
- **`RandomAccessIterator`** - `BidirectionalIterator` + random access
  (`it += n`)
- **`ContiguousIterator`** - `RandomAccessIterator` + contiguous in memory

Sentinels - the new end iterator

- **`Sentinel`** - a relationship between an iterator `I` and a semi-regular `S`
  - Let `s` and `i` be values of type `S` and `I`, respectively, such that `[i, s)` denotes a range
    - `i == s` is well defined
    - If `i != s` then `i` is dereferenceable and `[++i, s)` denotes a range
- By abuse of terminology, such an `s` is also called a sentinel

Sentinels

- **Examples:**
  - Pair of iterators
  - Iterator and predicate: store predicate in the sentinel and let `i == s` return the result of calling `s.predicate(*i)`
  - Iterator and count: store distance to end in the iterator and let `i == s` return the result of `i.count_ == 0`

Range Concepts

- **`Range`** - a type that allows iteration over its elements by providing an iterator and a sentinel that denote the elements of the range
  - `ranges::begin` - returns an iterator
  - `ranges::end` - returns a sentinel
- **`Counted range`** - an `Iterator` and a count
  - used e.g. in `copy_n`

Range Concepts

- **`InputRange`** – e.g. Range over a `std::istream_iterator`
- **`OutputRange`** – e.g. Range over a `std::ostream_iterator`
- **`ForwardRange`** – e.g. `std::forward_list`
- **`BidirectionalRange`** – e.g. `std::list`
- **`RandomAccessRange`** – e.g. `std::deque`
- **`ContiguousRange`** – e.g. `std::vector`
- **`CommonRange`** - sentinel is same type as iterator
  - e.g. standard containers

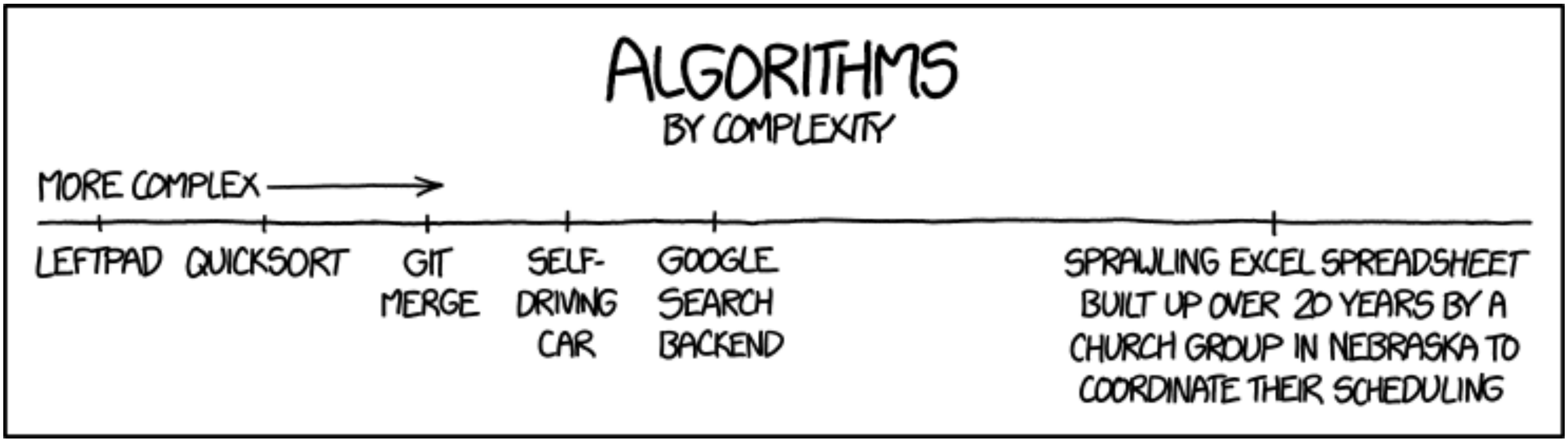
Sized ranges

- **`SizedSentinel`** - a `Sentinel` whose distance can be computed using the `-` `operator` in constant time.

- **`SizedRange`** - a `Range` type that knows its size in constant time with the `ranges::size` function
  - Not necessarily implies `SizeSentinel` (`std::list`)

- **`ranges::distance`** can compute the distant of any range but has linear complexity for non sized ranges/sentinels

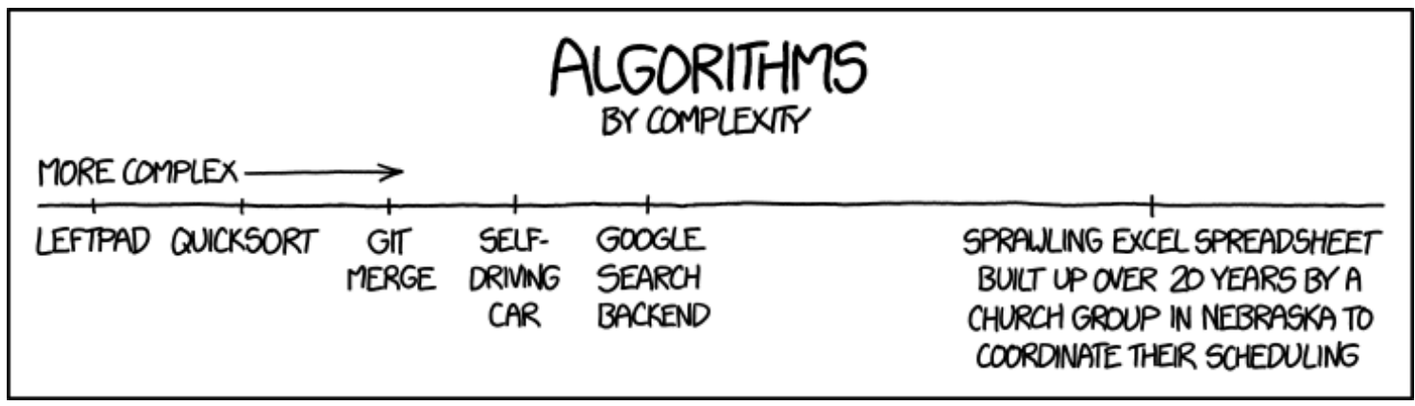
- **`ranges::data`** gives a pointer to the data of a **contiguous** range



https://www.xkcd.com/1667/

Main differences

- Accept either
  - a `Range` object
  - an `Iterator+Sentinel` pair
  - an `Iterator` and a count (`*_n`)
- Constrained
- Accept projections
- Some return type changes
- No Parallel overloads



https://www.xkcd.com/1667/

ranges::for_each

```
namespace ranges {

    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryInvocable<projected<I, Proj>> Fun>
      constexpr for_each_result<I, Fun>
        for_each(I first, S last, Fun f, Proj proj = {});

    template<InputRange R, class Proj = identity,
             IndirectUnaryInvocable<projected<iterator_t<R>, Proj>> Fun>
      constexpr for_each_result<safe_iterator_t<R>, Fun>
        for_each(R&& r, Fun f, Proj proj = {});
}
```

ranges::for_each

```
namespace ranges {

    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryInvocable<projected<I, Proj>> Fun>
      constexpr for_each_result<I, Fun>
        for_each(I first, S last, Fun f, Proj proj = {});

    template<InputRange R, class Proj = identity,
             IndirectUnaryInvocable<projected<iterator_t<R>, Proj>> Fun>
      constexpr for_each_result<safe_iterator_t<R>, Fun>
        for_each(R&& r, Fun f, Proj proj = {});
}
```

ranges::for_each

```
namespace ranges {

    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryInvocable<projected<I, Proj>> Fun>
      constexpr for_each_result<I, Fun>
        for_each(I first, S last, Fun f, Proj proj = {});

    template<InputRange R, class Proj = identity,
             IndirectUnaryInvocable<projected<iterator_t<R>, Proj>> Fun>
      constexpr for_each_result<safe_iterator_t<R>, Fun>
        for_each(R&& r, Fun f, Proj proj = {});
}
```

ranges::for_each

```
namespace ranges {

    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryInvocable<projected<I, Proj>> Fun>
      constexpr for_each_result<I, Fun>
        for_each(I first, S last, Fun f, Proj proj = {});

    template<InputRange R, class Proj = identity,
             IndirectUnaryInvocable<projected<iterator_t<R>, Proj>> Fun>
      constexpr for_each_result<safe_iterator_t<R>, Fun>
        for_each(R&& r, Fun f, Proj proj = {});
}
```

ranges::for_each

```
namespace ranges {

    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryInvocable<projected<I, Proj>> Fun>
      constexpr for_each_result<I, Fun>
        for_each(I first, S last, Fun f, Proj proj = {});

    template<InputRange R, class Proj = identity,
             IndirectUnaryInvocable<projected<iterator_t<R>, Proj>> Fun>
      constexpr for_each_result<safe_iterator_t<R>, Fun>
        for_each(R&& r, Fun f, Proj proj = {});
}
```

ranges::for_each

```
namespace ranges {
    template<class I, class F>
    struct for_each_result {
      [[no_unique_address]] I in;
      [[no_unique_address]] F fun;

      template<class I2, class F2>
        requires ConvertibleTo<const I&, I2> && ConvertibleTo<const F&, F2>
        operator for_each_result<I2, F2>() const & {
          return {in, fun};
        }

      template<class I2, class F2>
        requires ConvertibleTo<I, I2> && ConvertibleTo<F, F2>
        operator for_each_result<I2, F2>() && {
          return {std::move(in), std::move(fun)};
        }
    };
}
```

ranges::for_each

```
namespace ranges {
    template<class I, class F>
    struct for_each_result {
      [[no_unique_address]] I in;
      [[no_unique_address]] F fun;

      template<class I2, class F2>
        requires ConvertibleTo<const I&, I2> && ConvertibleTo<const F&, F2>
        operator for_each_result<I2, F2>() const & {
          return {in, fun};
        }

      template<class I2, class F2>
        requires ConvertibleTo<I, I2> && ConvertibleTo<F, F2>
        operator for_each_result<I2, F2>() && {
          return {std::move(in), std::move(fun)};
        }
    };
}
```

Example

```
1  int sum  = 0;
2  auto fun = [&](int i) { sum += i; };
3
4  std::vector<int> v1{0, 2, 4, 6};
5
6  auto res = for_each(v1, fun);
7
8  assert(res.in == v1.end());
9  assert(sum == 12);
10 res.fun(1);
11 assert(sum == 13);
```

Generators

`fill(_n)`

`iota`

`generate(_n)`

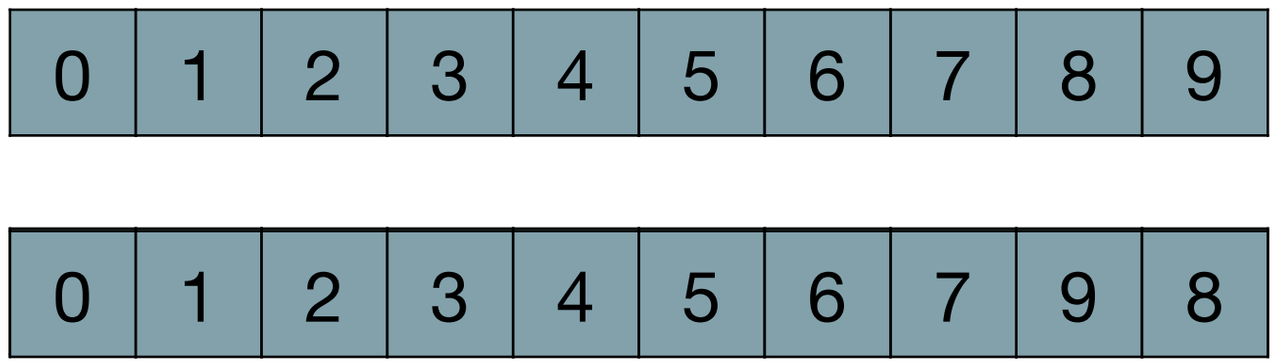
Permutations

0	1	2	3	4	5	6	7	8	9

Permutations

`next_permuation`

Permutations



Permutations

0	1	2	3	4	5	6	7	8	9

`next_permuation`

0	1	2	3	4	5	6	7	9	8

`prev_permuation`

Permutations

	0	1	2	3	4	5	6	7	8	9
`next_permuation`	0	1	2	3	4	5	6	7	9	8
`prev_permuation`	0	1	2	3	4	5	6	7	8	9

Permutations

	0	1	2	3	4	5	6	7	8	9
`next_permuation`	0	1	2	3	4	5	6	7	9	8
`prev_permuation`	0	1	2	3	4	5	6	7	8	9

`reverse(_copy)`

Permutations

	0	1	2	3	4	5	6	7	8	9
`next_permuation`	0	1	2	3	4	5	6	7	9	8
`prev_permuation`	0	1	2	3	4	5	6	7	8	9
`reverse(_copy)`	9	8	7	6	5	4	3	2	1	0

Permutations

	0	1	2	3	4	5	6	7	8	9
`next_permuation`	0	1	2	3	4	5	6	7	9	8
`prev_permuation`	0	1	2	3	4	5	6	7	8	9

`reverse(_copy)`

9	8	7	6	5	4	3	2	1	0

`rotate(_copy)`

Permutations

	0	1	2	3	4	5	6	7	8	9
`next_permuation`	0	1	2	3	4	5	6	7	9	8
`prev_permuation`	0	1	2	3	4	5	6	7	8	9
`reverse(_copy)`	9	8	7	6	5	4	3	2	1	0
`rotate(_copy)`	2	1	0	9	8	7	6	5	4	3

Permutations

	0	1	2	3	4	5	6	7	8	9
`next_permuation`	0	1	2	3	4	5	6	7	9	8
`prev_permuation`	0	1	2	3	4	5	6	7	8	9

`reverse(_copy)`

9	8	7	6	5	4	3	2	1	0

`rotate(_copy)`

2	1	0	9	8	7	6	5	4	3

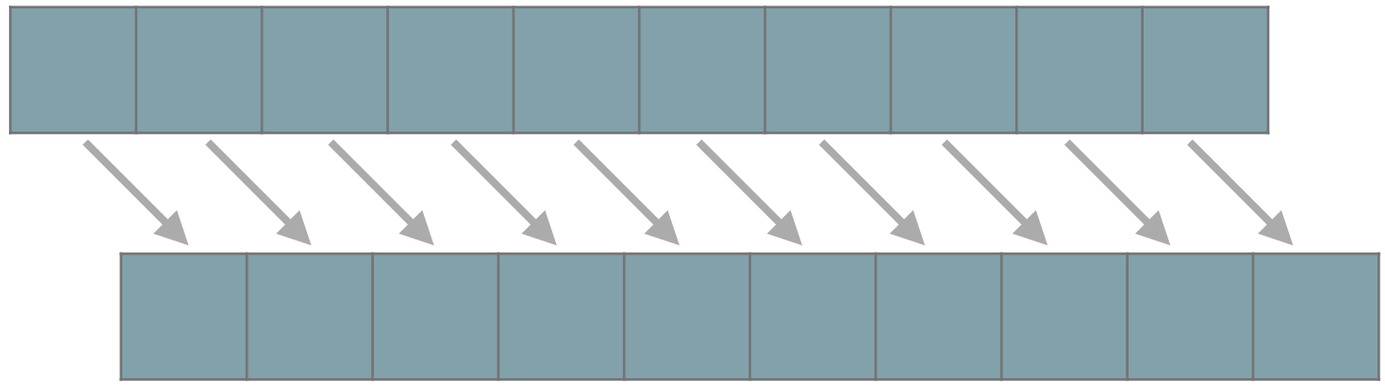
`shuffle`

Permutations

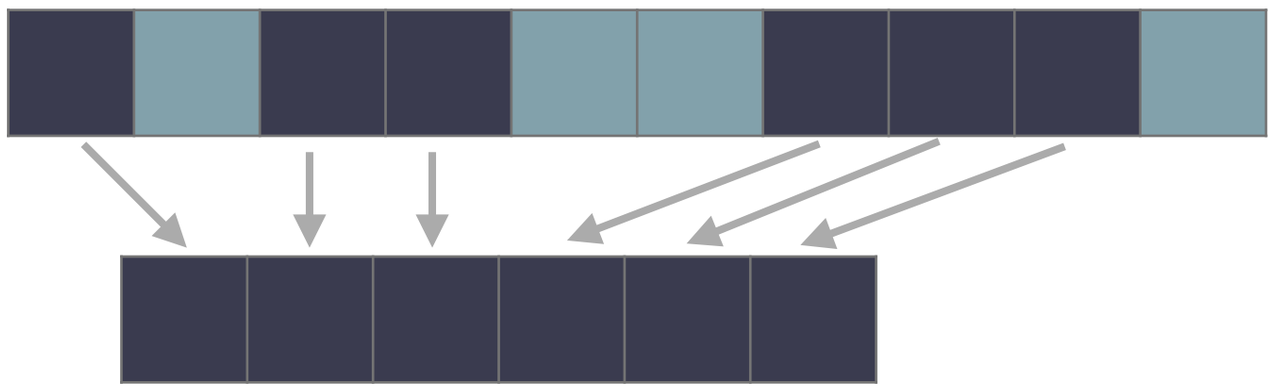
	0	1	2	3	4	5	6	7	8	9
`next_permuation`	0	1	2	3	4	5	6	7	9	8
`prev_permuation`	0	1	2	3	4	5	6	7	8	9
`reverse(_copy)`	9	8	7	6	5	4	3	2	1	0
`rotate(_copy)`	2	1	0	9	8	7	6	5	4	3
`shuffle`	5	9	6	0	1	2	3	8	4	7

Transformers

`copy(_n)`
`copy_backward`

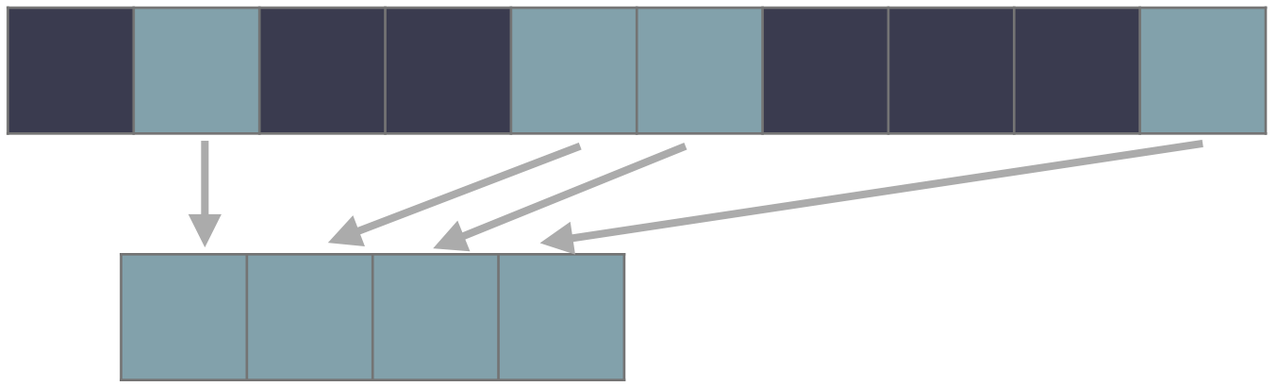


`copy_if`

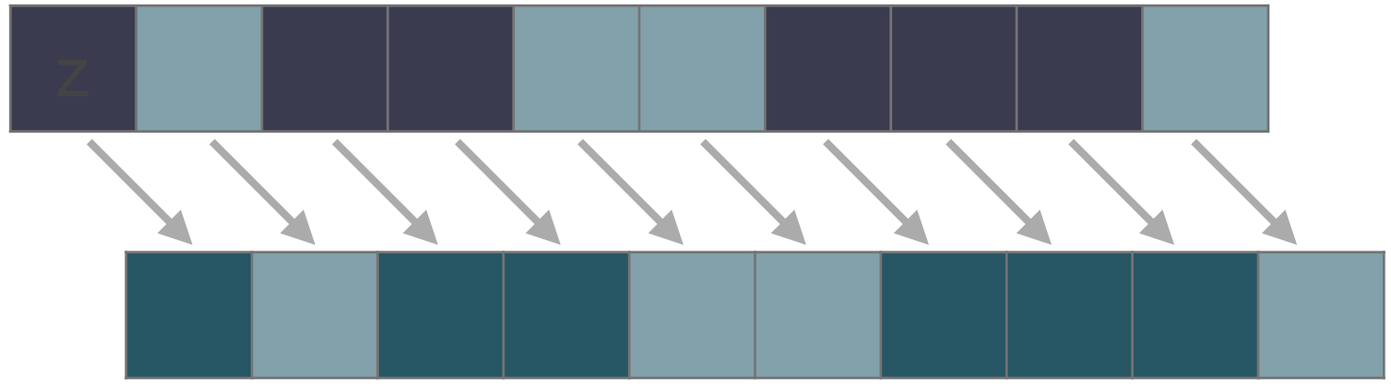


Transformers

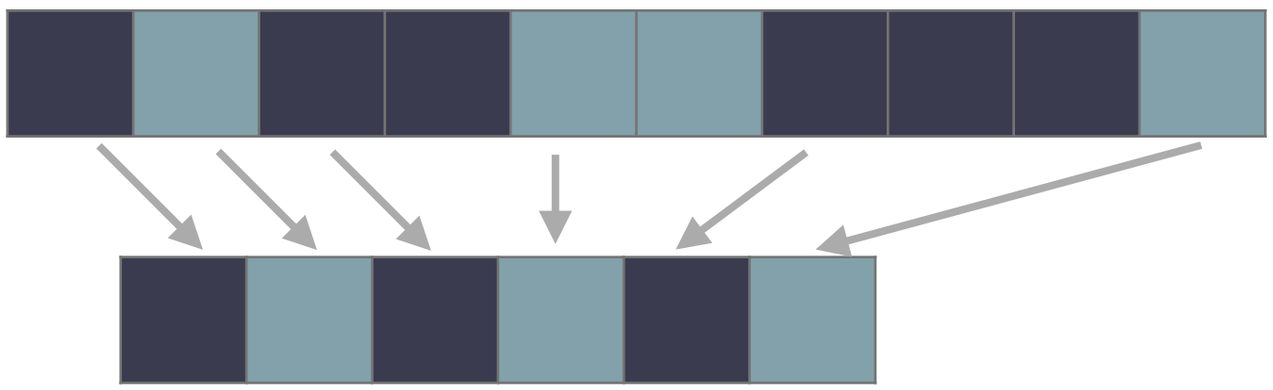
```
remove(_if)
remove_copy(_if)
```



```
replace(_if)
replace_copy(_if)
```

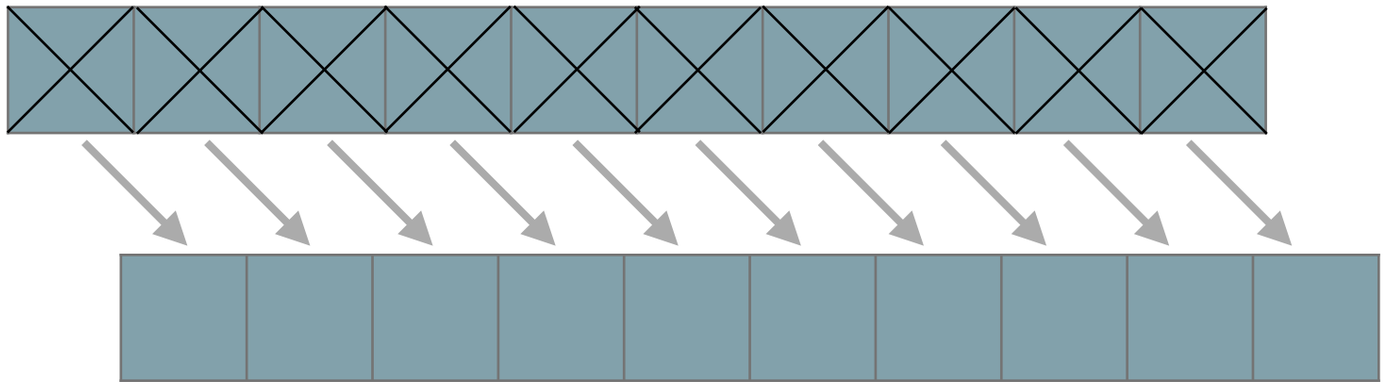



```
unique
unique_copy
```

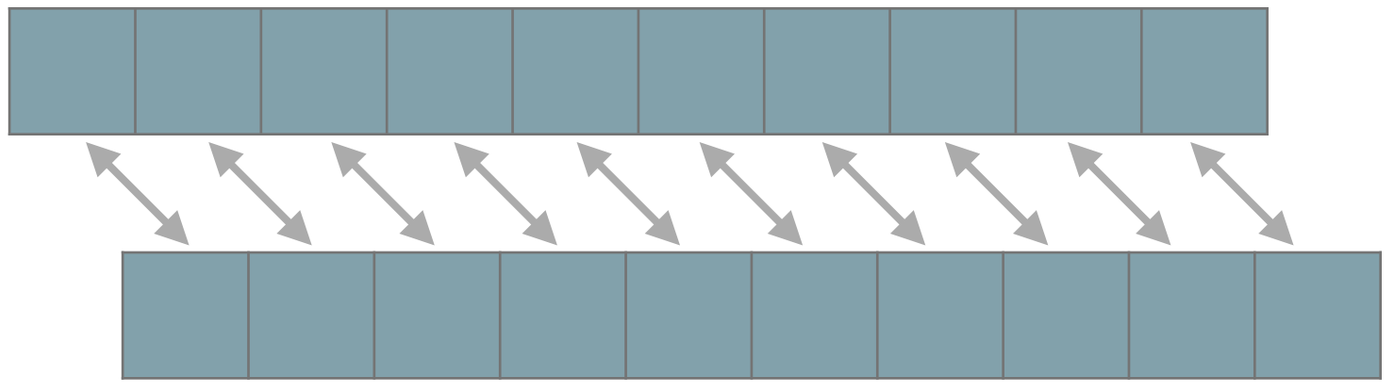


Transformers

`move`
`move_backward`



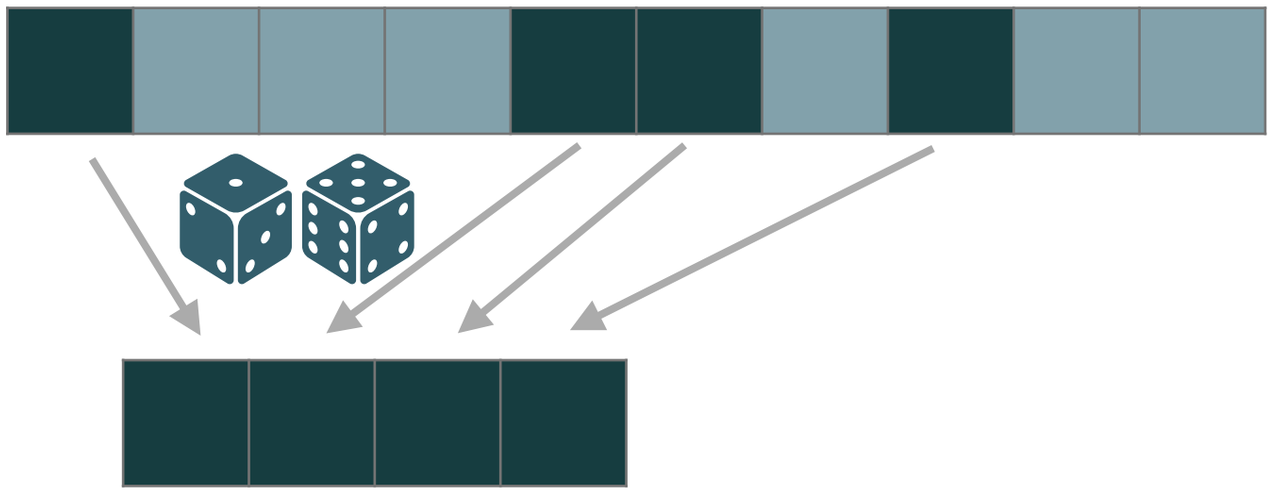
`swap_ranges`



Transformers

P1243

`sample`

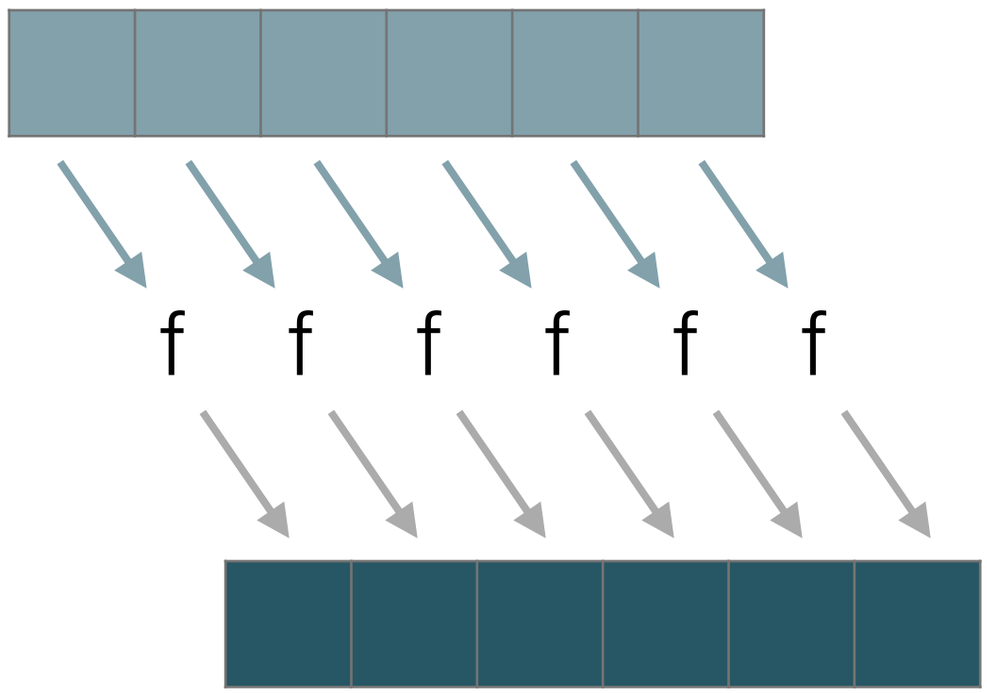


`shift_left`

`shift_right`

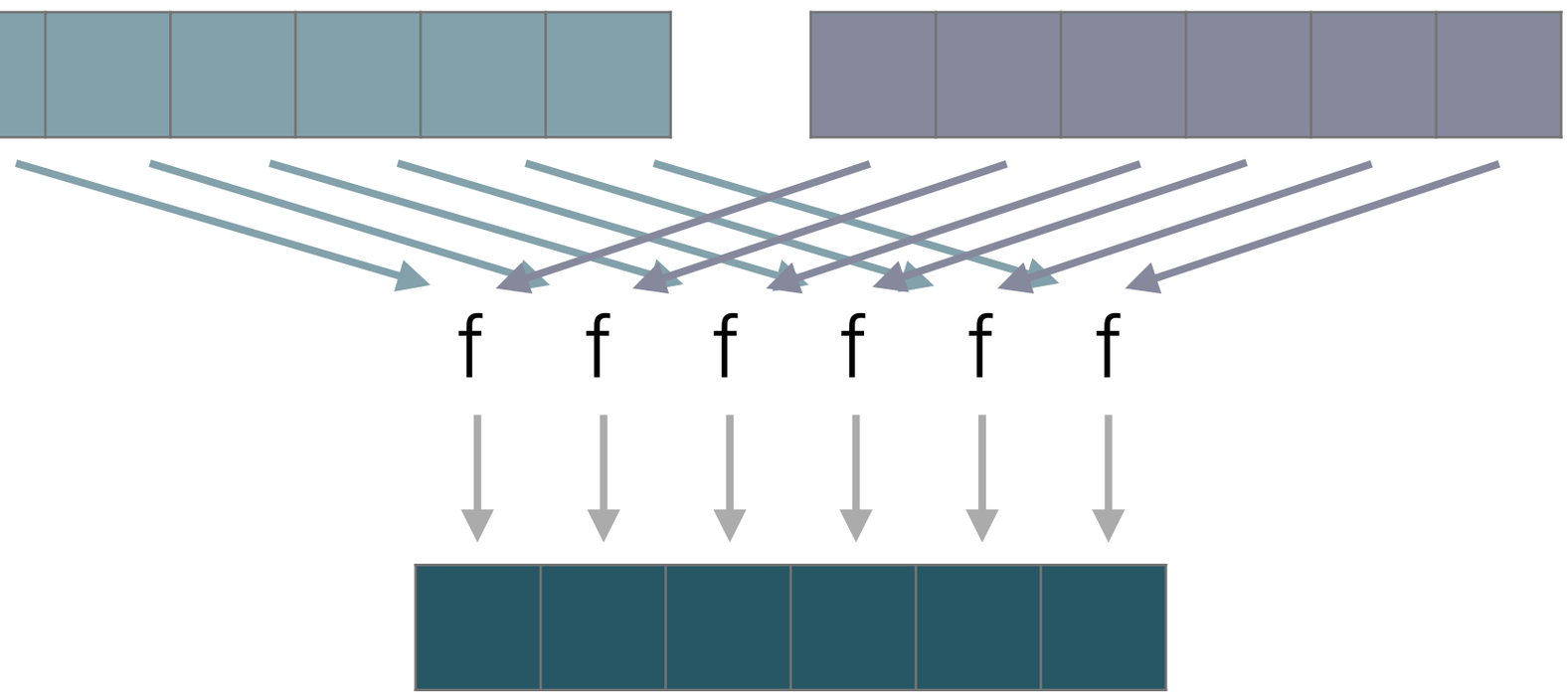
Transformers

`transform`



Transformers

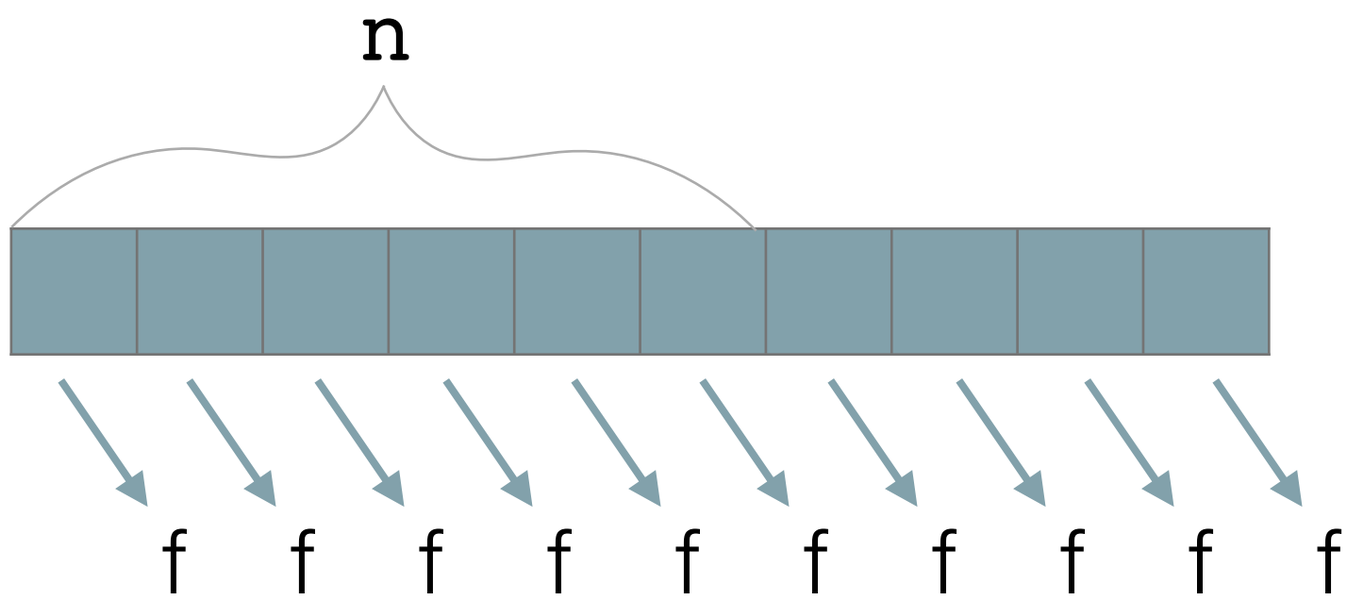
`transform`



for each

```
for_each
```

P1243 `for_each_n`



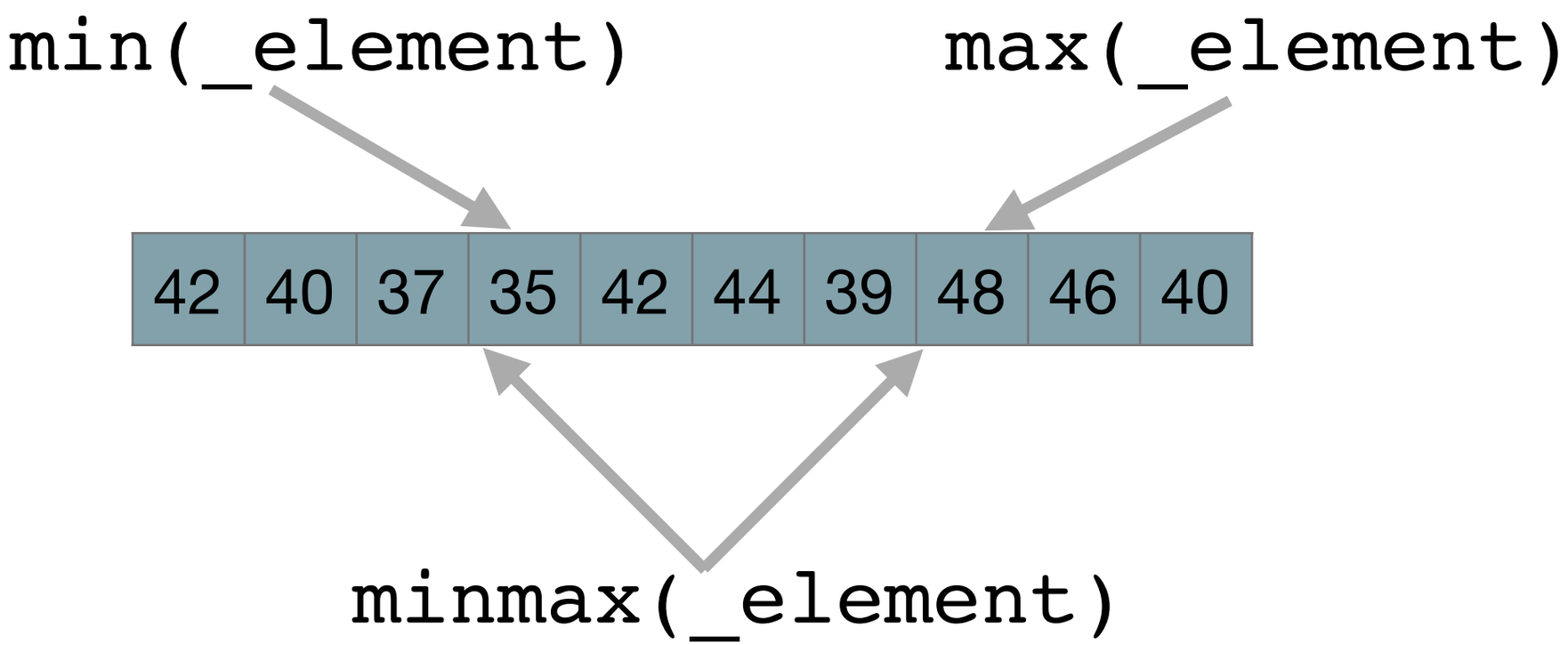
Property checks

`all_of`

`any_of`

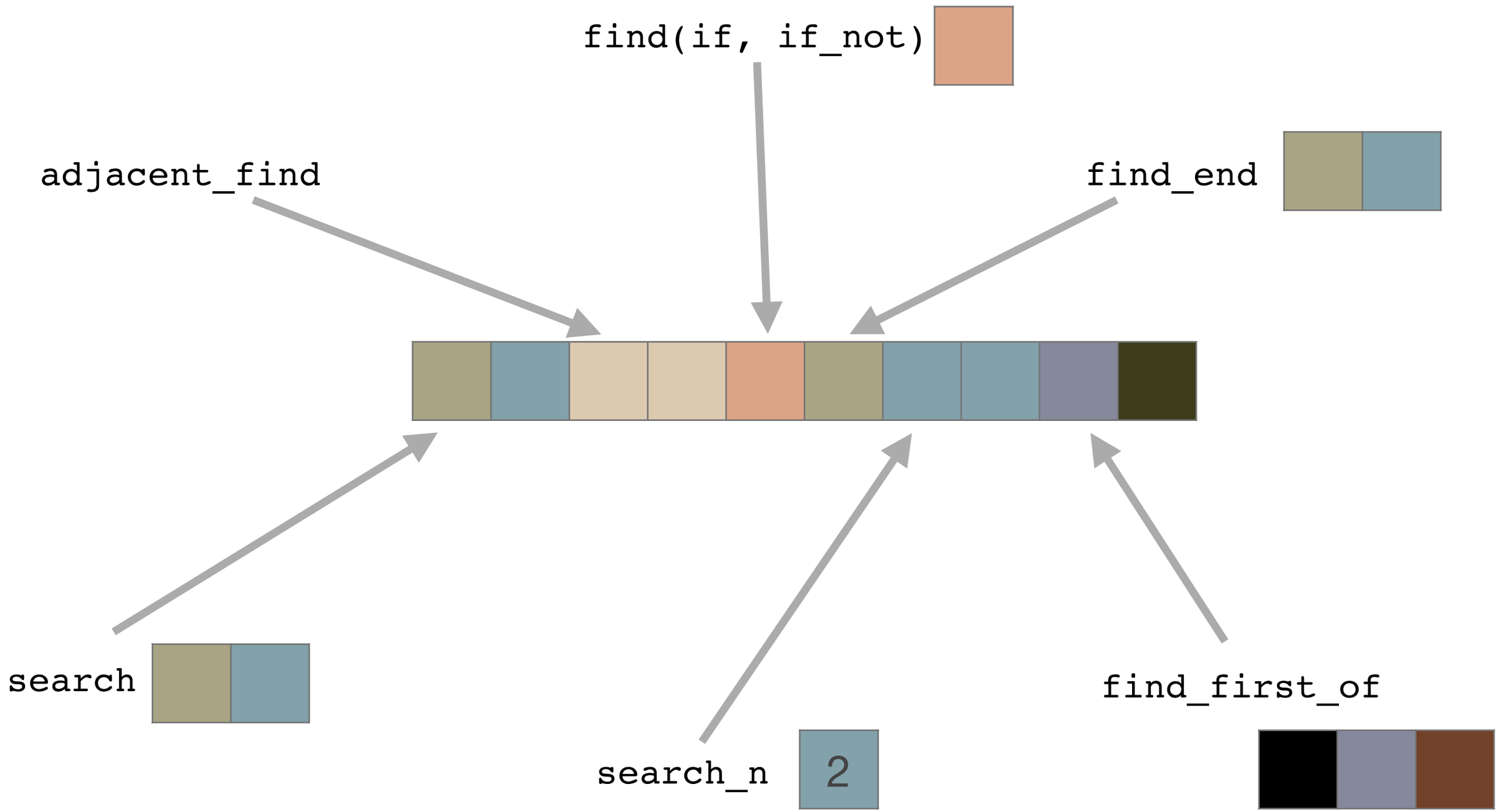
`none_of`

Value queries

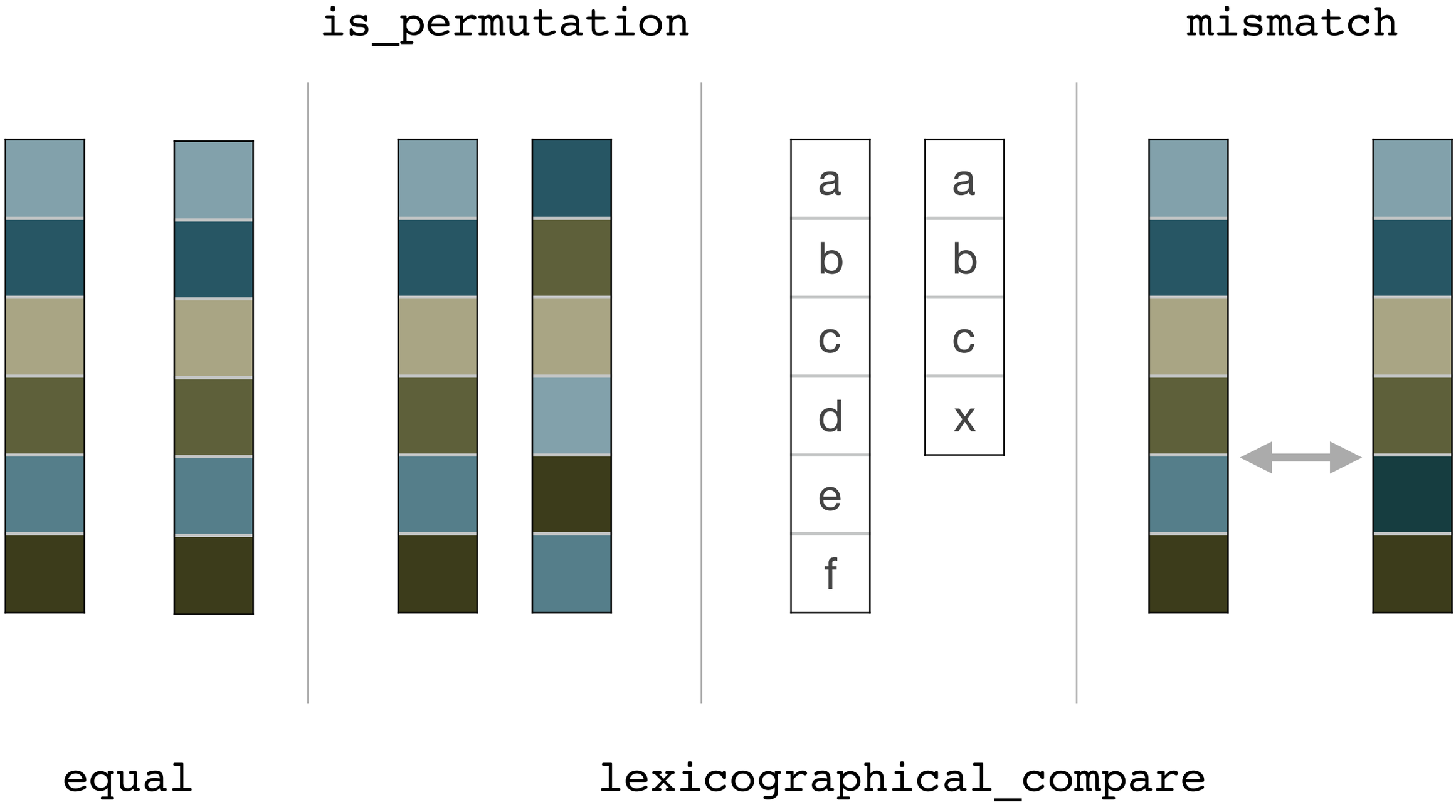



```
count(42) → 2

count_if(isOdd) → 3
```

Searchers


find(if, if_not)
adjacent_find
find_end
search
search_n
2
find_first_of

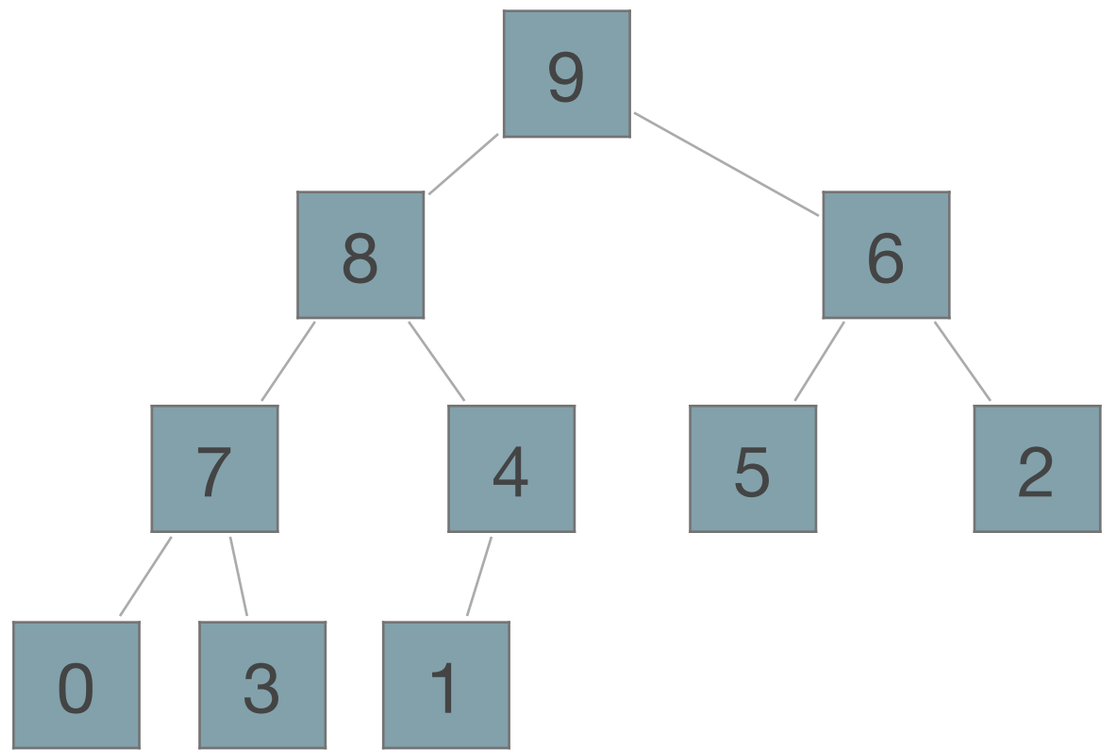
Comparison operations



Heap algorithms

Heap algorithms

`make_heap`



Heap algorithms

`make_heap`

Heap algorithms

`make_heap`

`push_heap`

Heap algorithms

`make_heap`

`push_heap`

Heap algorithms

`make_heap`

`push_heap`

`pop_heap`

Heap algorithms

`make_heap`

`push_heap`

`pop_heap`

Heap algorithms

`make_heap`

`push_heap`

`pop_heap`

`is_heap` `bool`

Heap algorithms

`make_heap`

`push_heap`

`pop_heap`

`is_heap` `bool`

`is_heap_until`

Sort algorithms

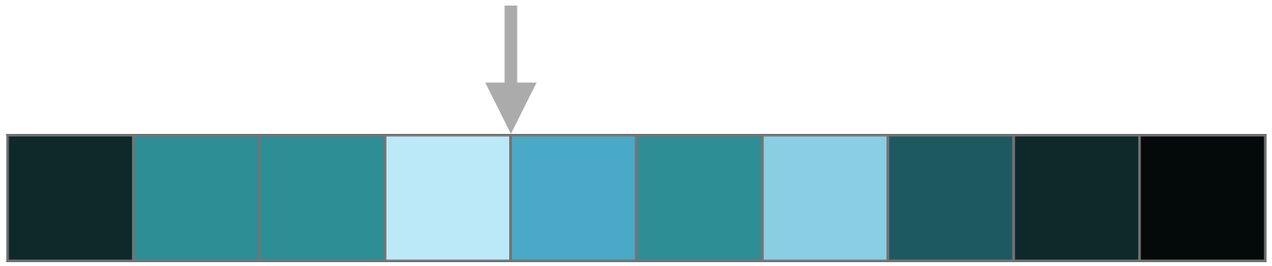
```
sort

sort_heap

stable_sort
```

Sort algorithms

```
sort

sort_heap

stable_sort
```

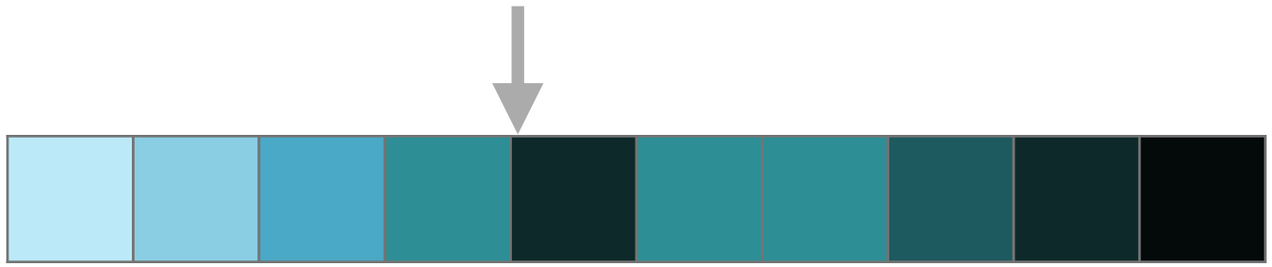
Sort algorithms

```
sort

sort_heap

stable_sort

partial_sort(_copy)
```



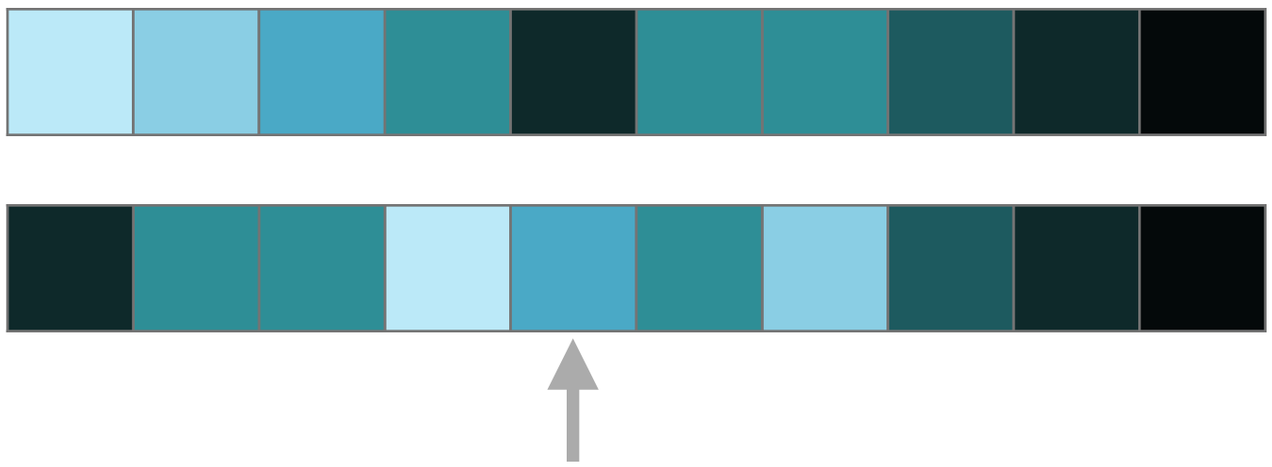
Sort algorithms

```
sort

sort_heap

stable_sort

partial_sort(_copy)
```



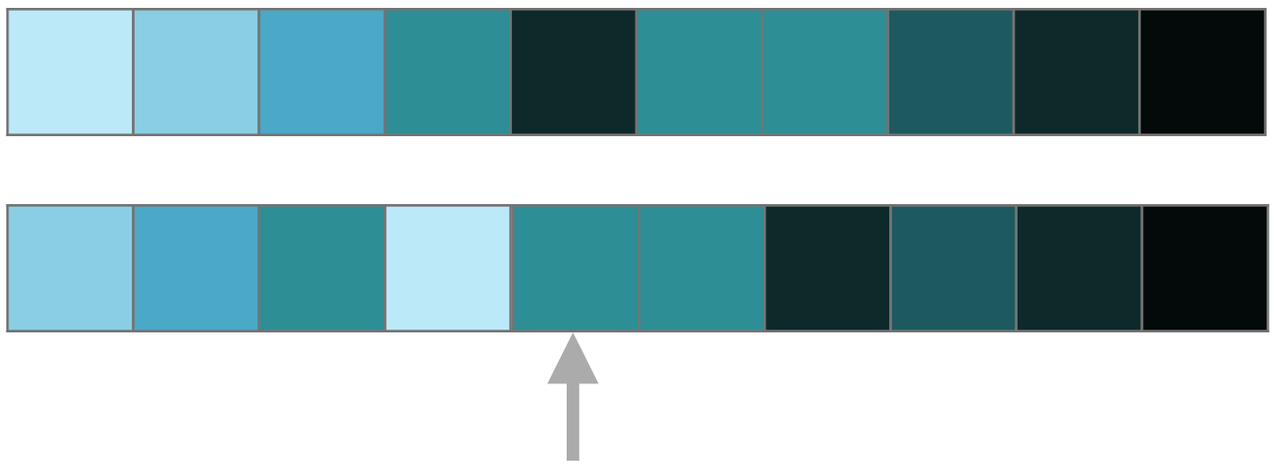
Sort algorithms

```
sort

sort_heap

stable_sort

partial_sort(_copy)

nth_element
```



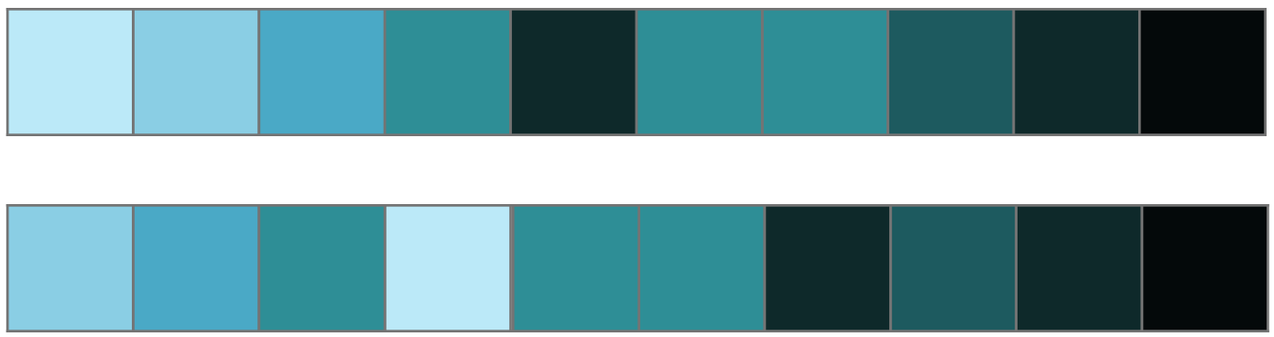
Sort algorithms

```
sort

sort_heap

stable_sort

partial_sort(_copy)

nth_element
```



Sort algorithms

```
sort

sort_heap

stable_sort

partial_sort(_copy)

nth_element

is_sorted                bool

is_sorted_until
```



Merge sort

`merge`

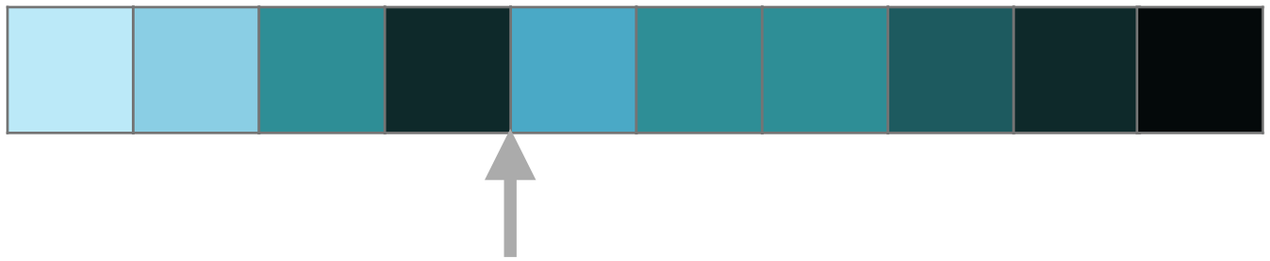
Merge sort

`merge`

Merge sort

`merge`

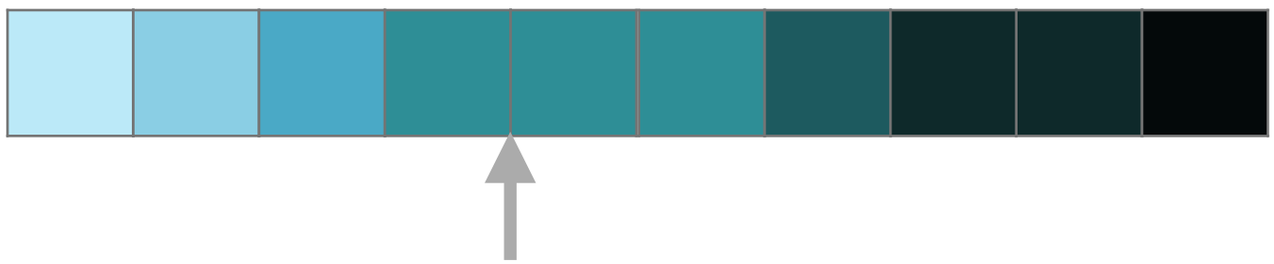
`inplace_merge`



Merge sort

`merge`

`inplace_merge`



Partition algorithms

```
partition(_copy)

stable_partition
```

Partition algorithms

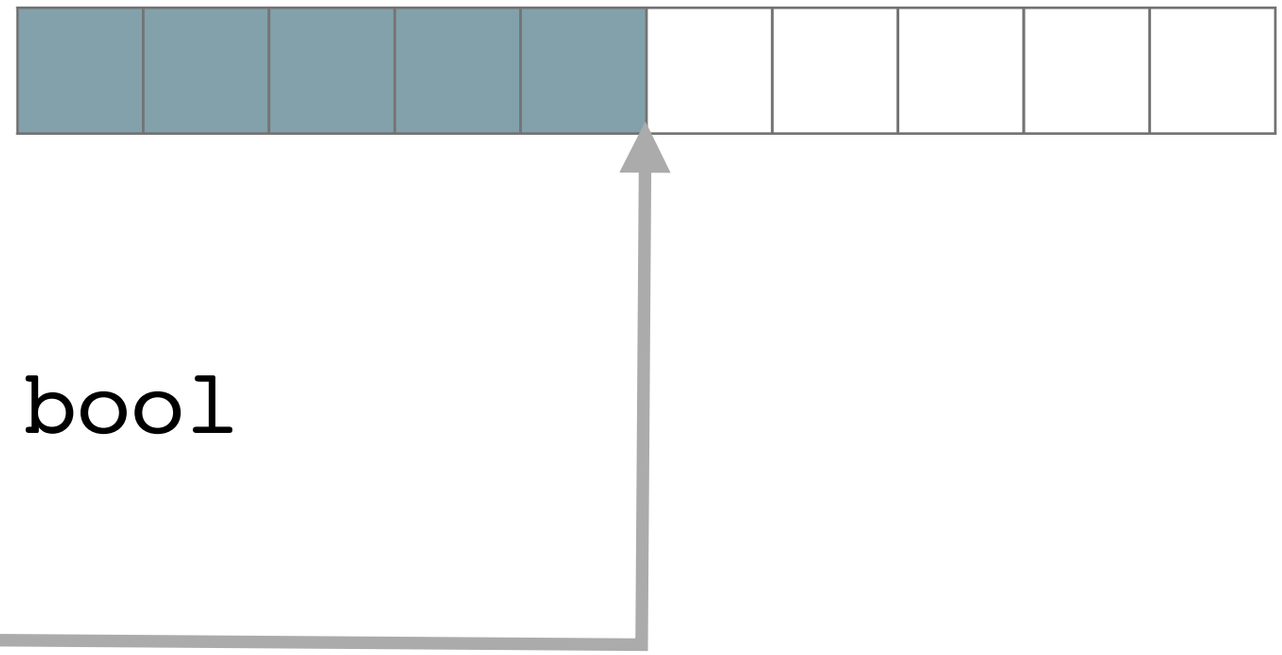
```
partition(_copy)

stable_partition
```

Partition algorithms

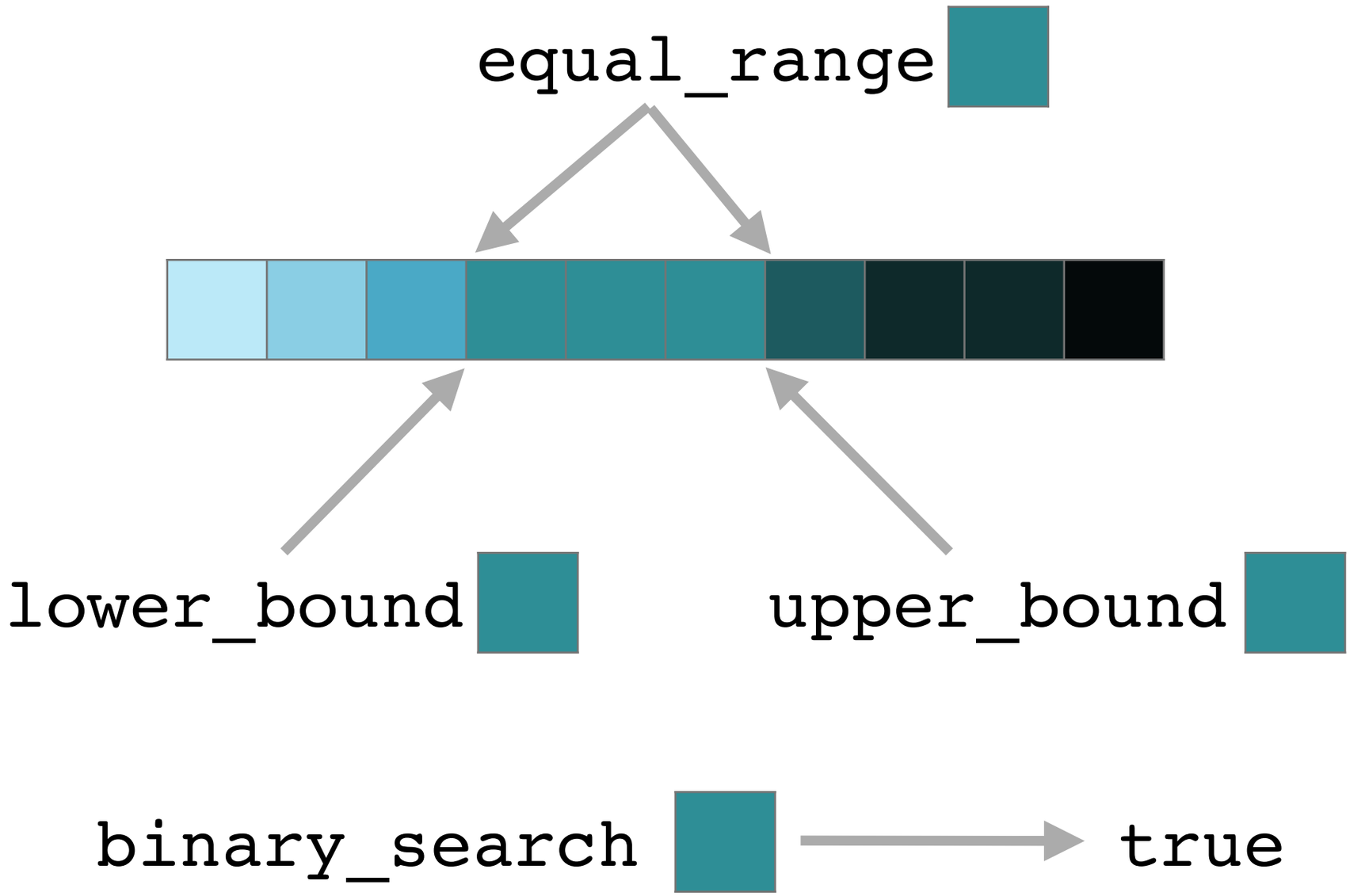
```
partition(_copy)

stable_partition

is_partitioned          bool
```

Partition algorithms

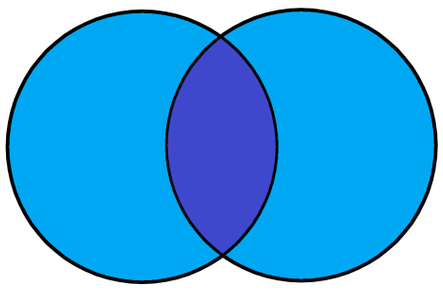
```
partition(_copy)

stable_partition


is_partitioned      bool

partition_point
```



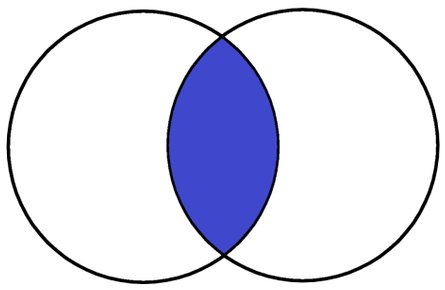
Binary searchers



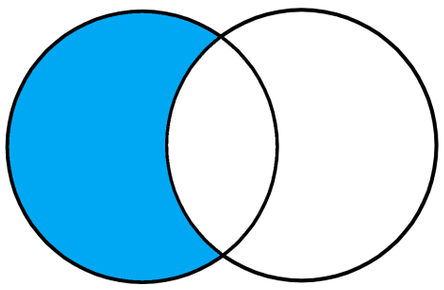
Set operations



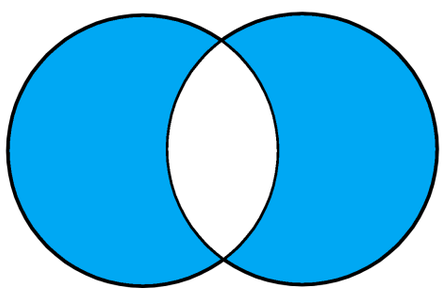
`set_union`



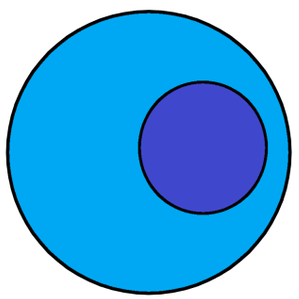
`set_intersection`



`set_difference`



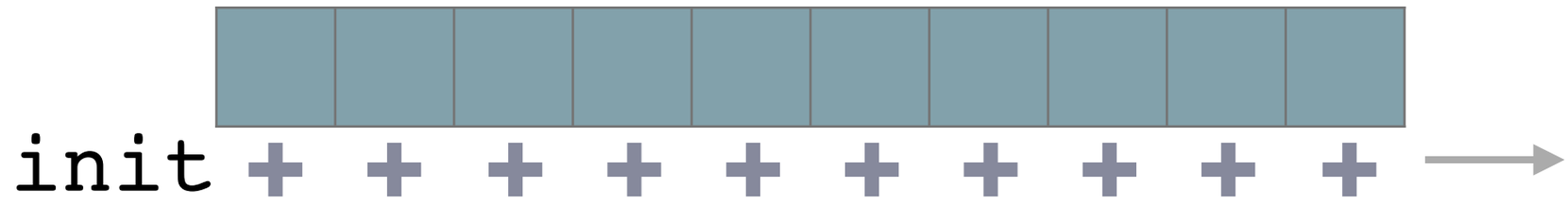
`set_symmetric_difference`



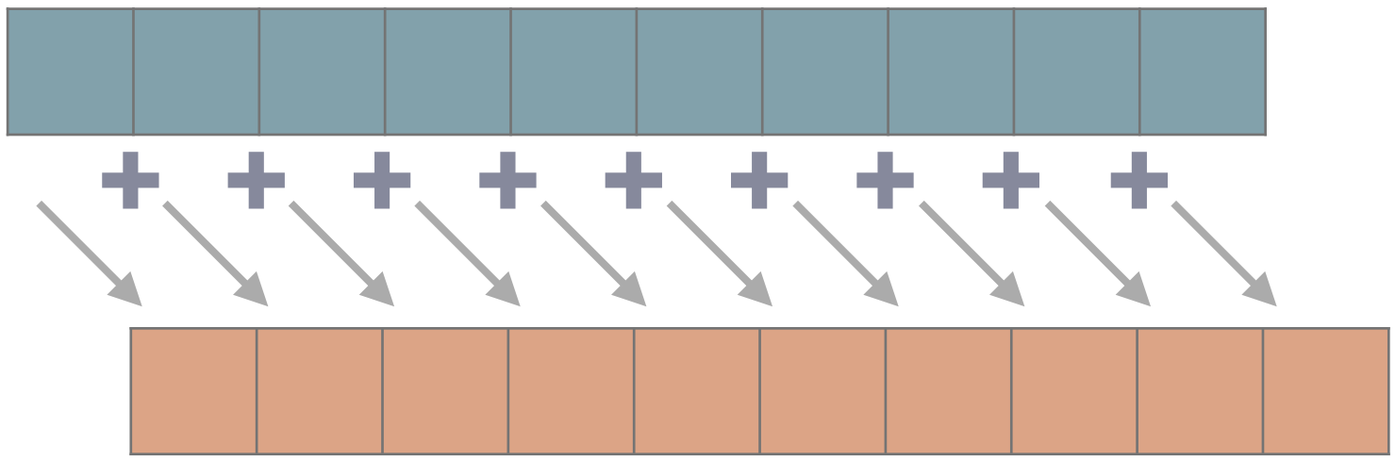
`includes`

Numerics

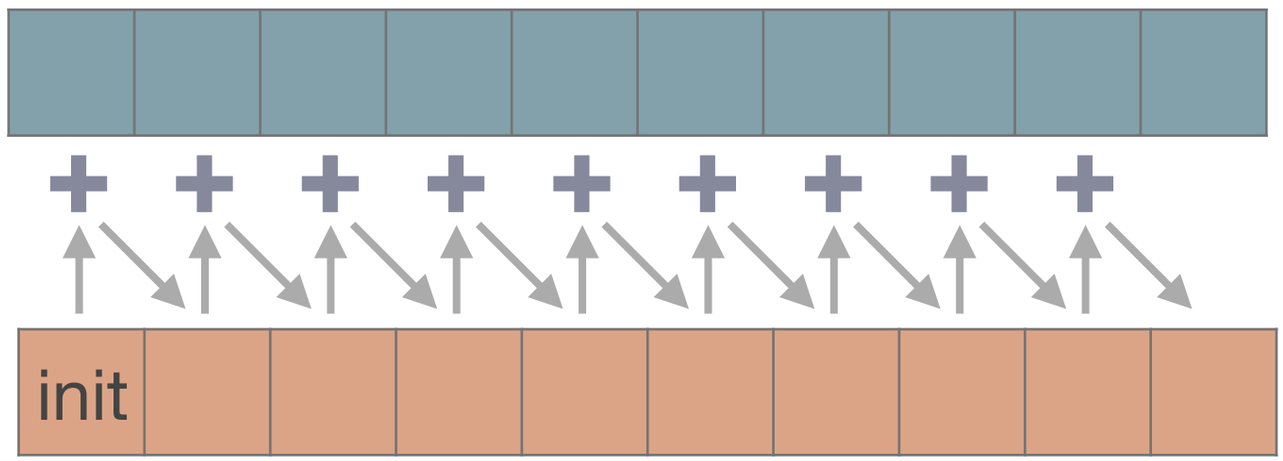
`accumulate`



`partial_sum`

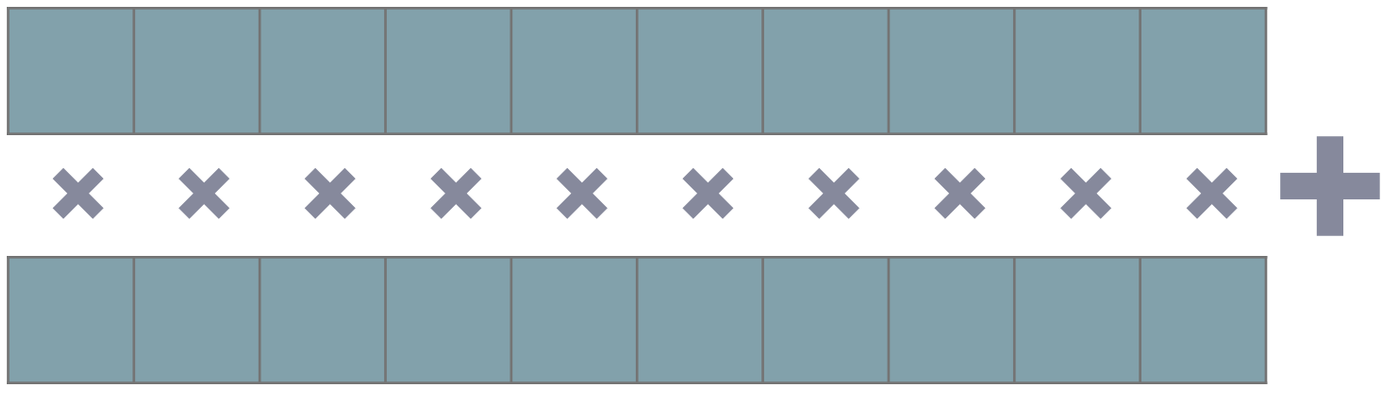


`exclusive_scan`

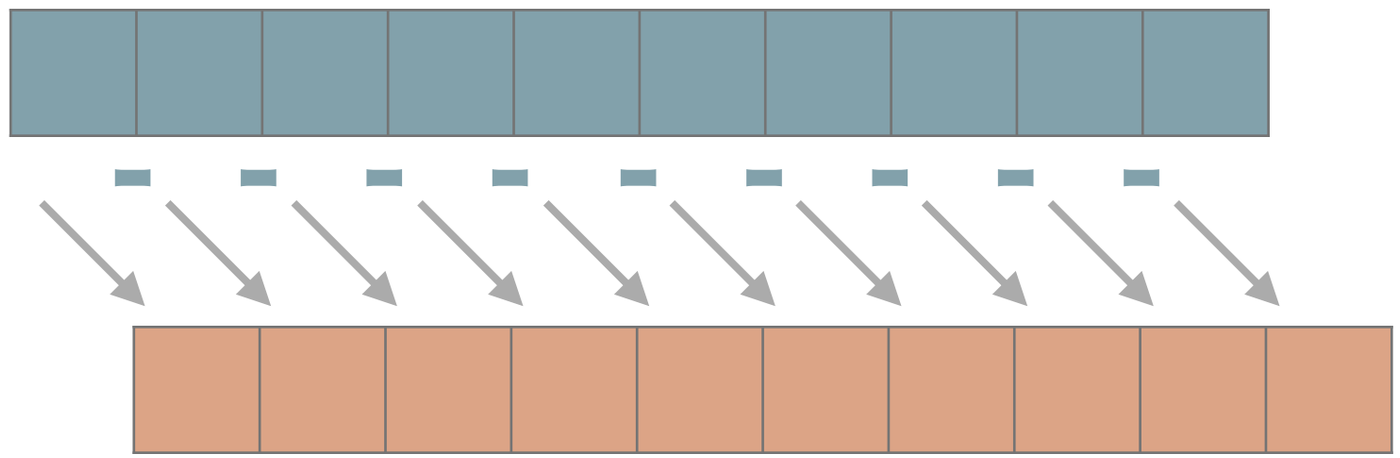


Numerics

`inner_product`



`adjacent_difference`



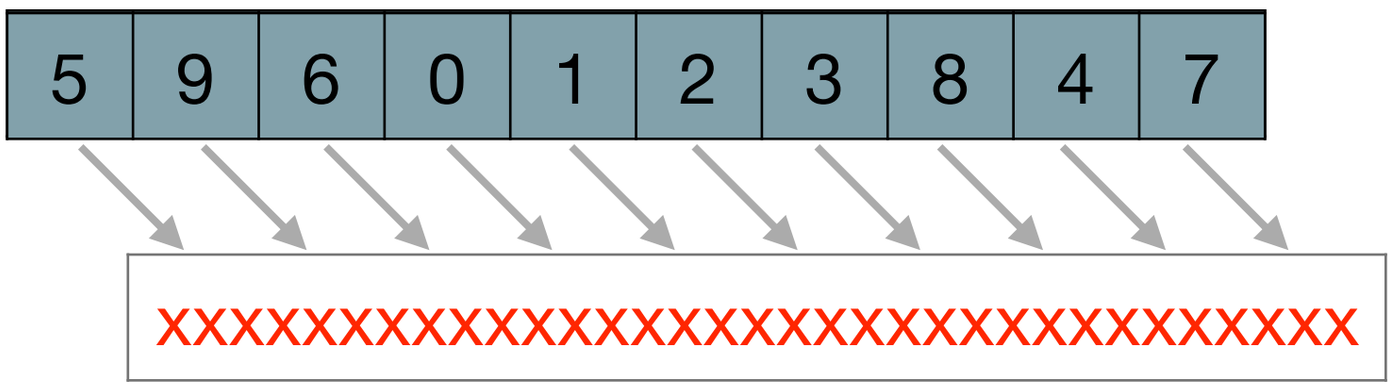
Uninitialized memory

P1033

```
uninitialized_default_construct(_n)
uninitialized_value_construct(_n)
```

```
uninitialized_fill
uninitialized_fill_n
```

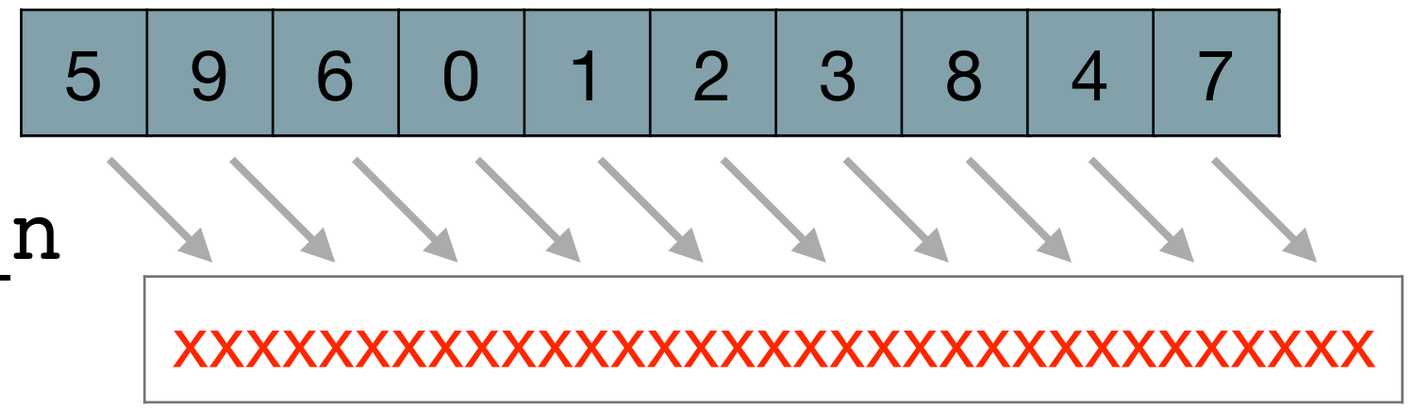

```
uninitialized_copy
uninitialized_copy_n
```



Uninitialized memory

P1033

`uninitialized_move`
`uninitialized_move_n`



`destroy(_n)`

Projections

https://www.xkcd.com/283/

Motivation

```
struct employee {
  std::string first;
  std::string last;
};

std::vector<employee> employees;

std::sort(employees.begin(), employees.end(),
      [](const employee &x, const employee &y) {
        return x.last < y.last;
});

auto p = std::lower_bound(
      employees.begin(), employees.end(), "Niebler",
      [](const employee &x, const std::string &y) {
 return x.last < y;
});
```

https://www.xkcd.com/283/

Project

- A projection is a transformation that an algorithm applies before inspecting the values of elements.

```
template <InputIterator I, Sentinel<I> S,
          class Proj = identity,
          IndirectUnaryInvocable<projected<I, Proj>> Fun>
constexpr for_each_result<I, Fun>
for_each(I first, S last, Fun f, Proj proj = {}) {
  for (; first != last; ++first) {
    invoke(fun, invoke(proj, *first));
  }
  return {std::move(first), std::move(fun)};
}
```

Motivation

```
struct employee {
  std::string first;
  std::string last;
};

std::vector<employee> employees;
```

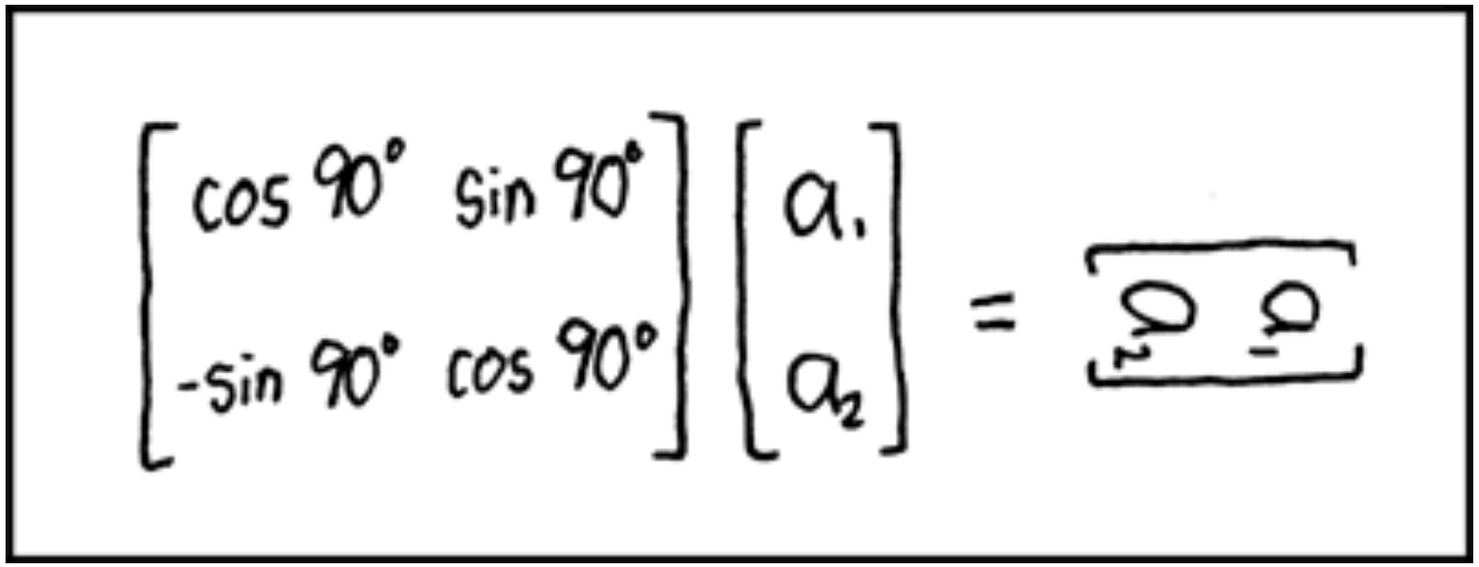
```
const auto getLast = [](const employee& e) {
  return e.last;
};

sort(employees, less{}, getLast);
auto p = lower_bound(employees, "Niebler", less{},
  getLast);
```

Motivation

```
struct employee {
  std::string first;
  std::string last;
};

std::vector<employee> employees;
```

```
sort(employees, {}, &employee::last);
auto p = lower_bound(employees, "Niebler", {},
  &employee::last);
```

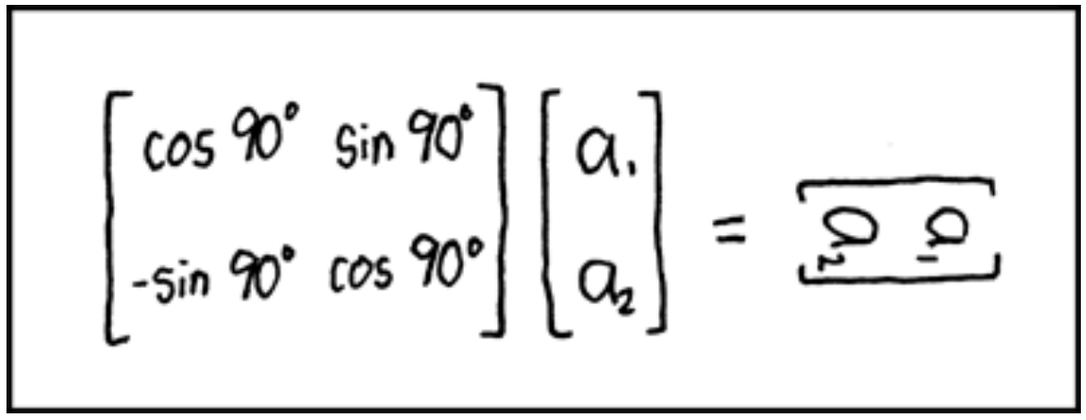
Views



https://www.xkcd.com/184/

Views

- **`View`** - a `Range` type that has constant time copy, move, and assignment operators.



https://www.xkcd.com/184/

- Examples of Views are:
  - A Range type that wraps a pair of iterators.
  - A Range type that holds its elements by `shared_ptr` and shares ownership with all its copies.
  - A Range type that generates its elements on demand
  - Note: most containers are NOT views.

Example

```
namespace std::ranges {
  template<class T>
    requires is_object_v<T>
  class empty_view : public view_interface<empty_view<T>> {
  public:
    static constexpr T* begin() noexcept { return nullptr; }
    static constexpr T* end() noexcept { return nullptr; }
    static constexpr T* data() noexcept { return nullptr; }
    static constexpr ptrdiff_t size() noexcept { return 0; }
    static constexpr bool empty() noexcept { return true; }

    friend constexpr T* begin(empty_view) noexcept { return nullptr; }
    friend constexpr T* end(empty_view) noexcept { return nullptr; }
  };
}
```

Factories & Adaptors

- For each view, the library defines a utility object called an adaptor (if it transforms an existing range) or a factory (otherwise), which creates such a view.

```
namespace std::ranges {
  namespace view {
    template<class T>
      inline constexpr empty_view<T> empty{};
  }
}
```

- Adapted views are **lazy**, which means they generate their elements only on demand, when the resulting adapted range is iterated.

- For this reason, there can be views with infinitely many elements.

Concepts

- **`ForwardingRange`** - a `Range` whose iterators' validity is not tied to the lifetime of the range.
  - Examples:
    - `View`
    - lvalue reference of a `Range`
- **`ViewableRange`** - a `Range` type that can be converted to a `View` safely.
  - Either a `View` or a `ForwardingRange`

Generators

`view::empty`

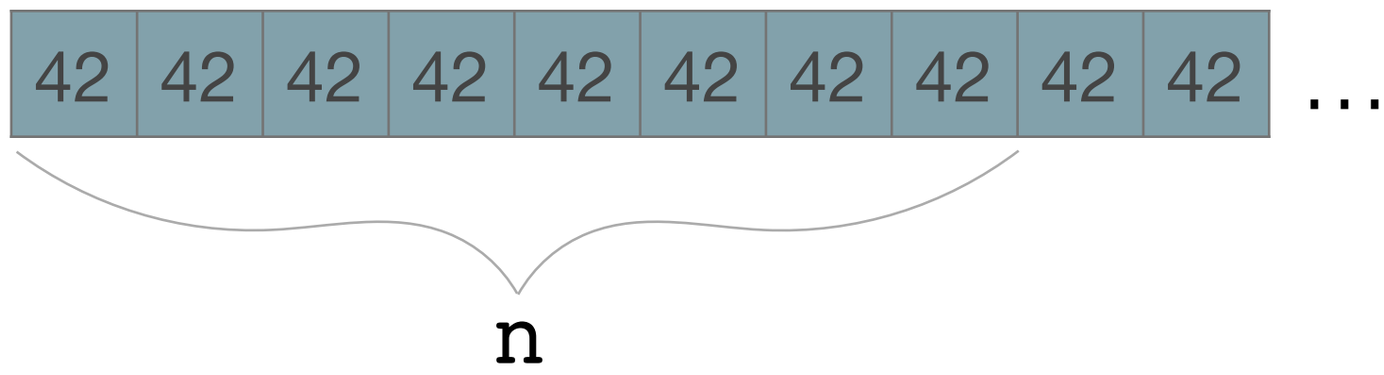
`view::single` 42

`view::all` Range → View

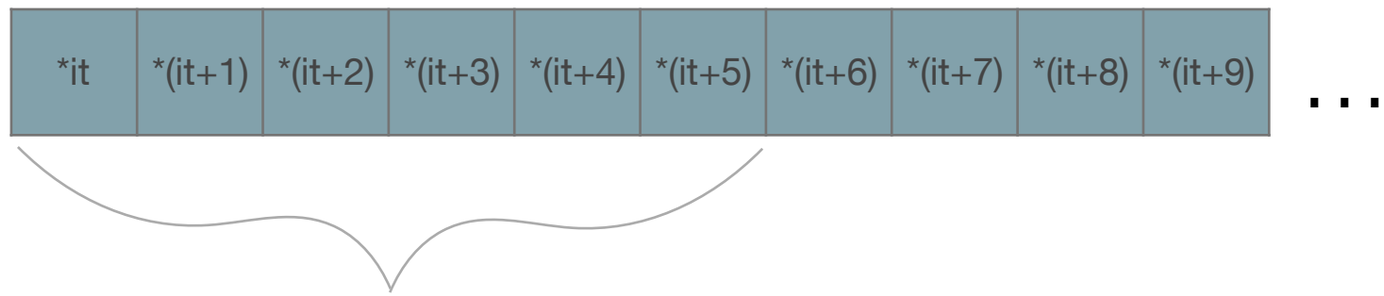
Generators

`view::repeat`

`view::repeat_n`



`view::unbounded`



`view::counted`

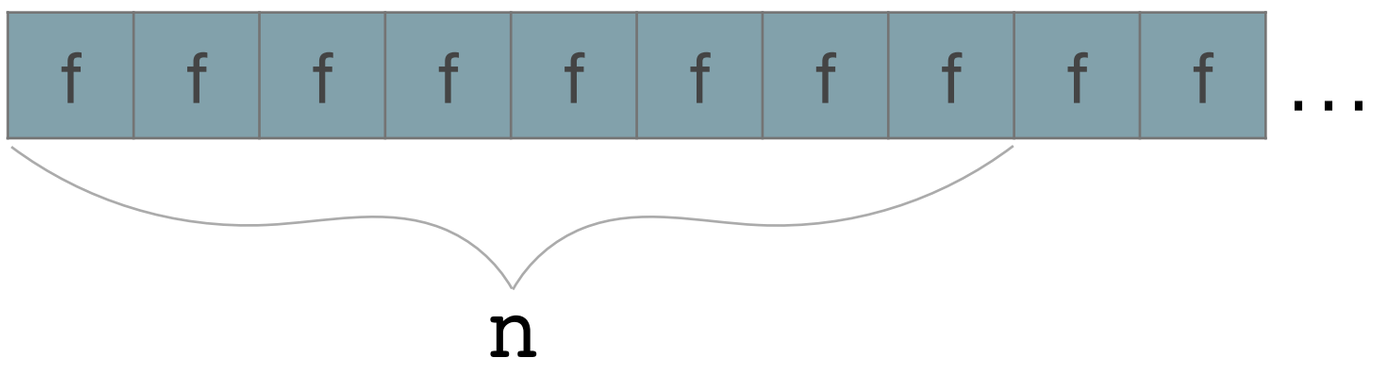
Generators

`view::iota`

`view::linear_distribute`

`view::generate`

`view::generate_n`



Iota

```
view::iota
```

0	1	2	3	4	5	6	7	8	9	...

```
view::iota(10)
```

10	11	12	13	14	15	16	17	18	19	...

```
view::iota(10, 18)
```

10	11	12	13	14	15	16	17

```
view::closed_iota(10, 18)
```

10	11	12	13	14	15	16	17	18

```
view::ints == view::iota<Integral>
```

```
view::closed_indices == view::closed_iota<Integral>
```

```
view::indices(10)
```

0	1	2	3	4	5	6	7	8	9

Strings

```
view::c_str
```

```
“Core C++”
```


```
pStr
```

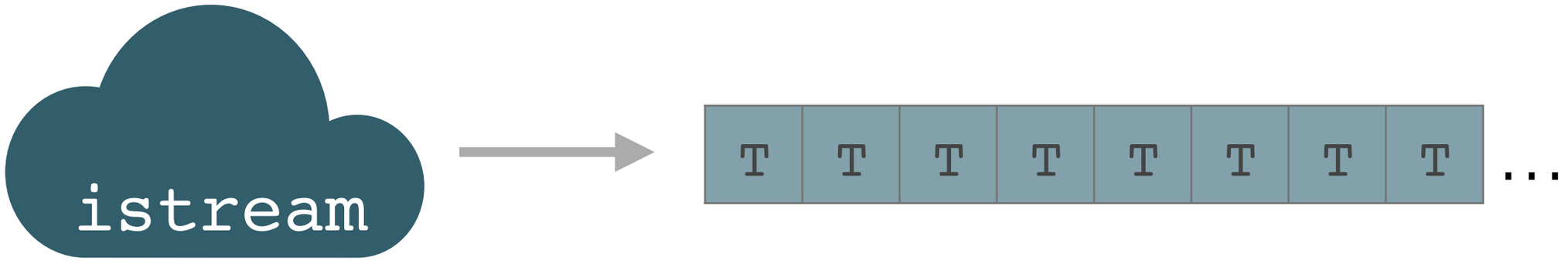
```
view::tokenize
```

```
view::tokenize(rng, std::regex{“([a-z]+)”}, 1);
```

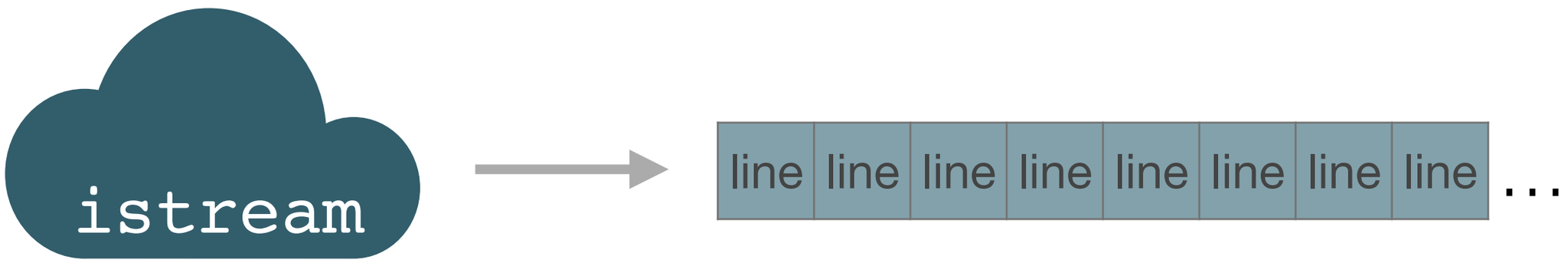
```
“abc\ndef\tghi klm”
```

Streams

P1035 `istream_view<T>/istream_range<T>`

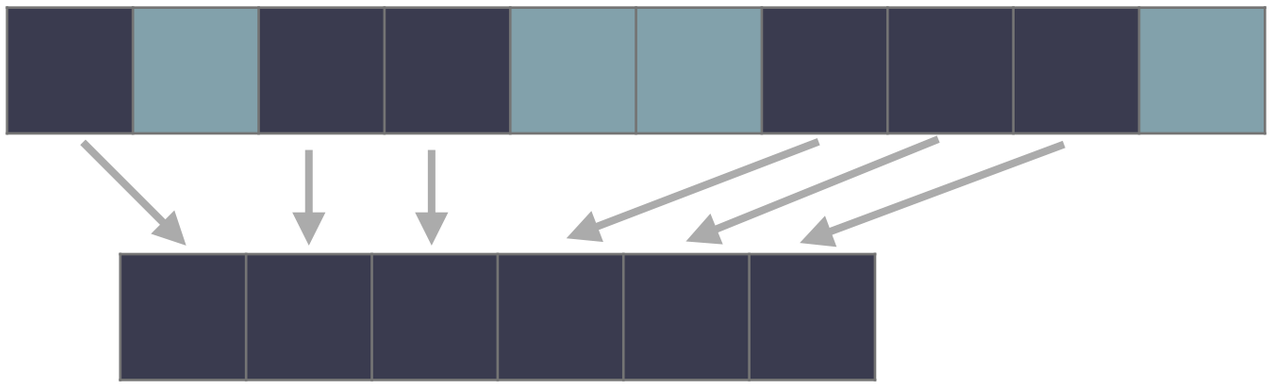


`getlines`

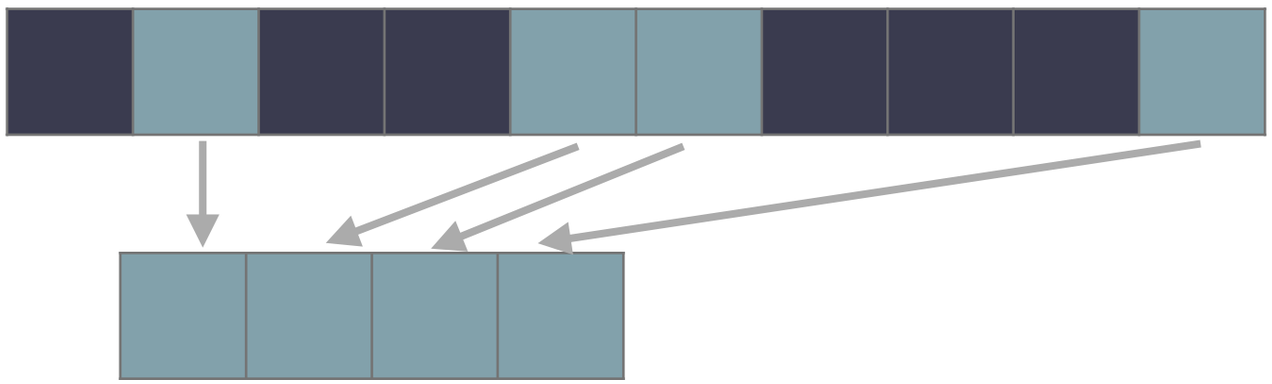


Filters

`view::filter`



`view::remove_if`



Filters

`view::take`

`view::take_exactly`

P1035 `view::take_while`

P1035 `view::drop`

`view::drop_exactly`

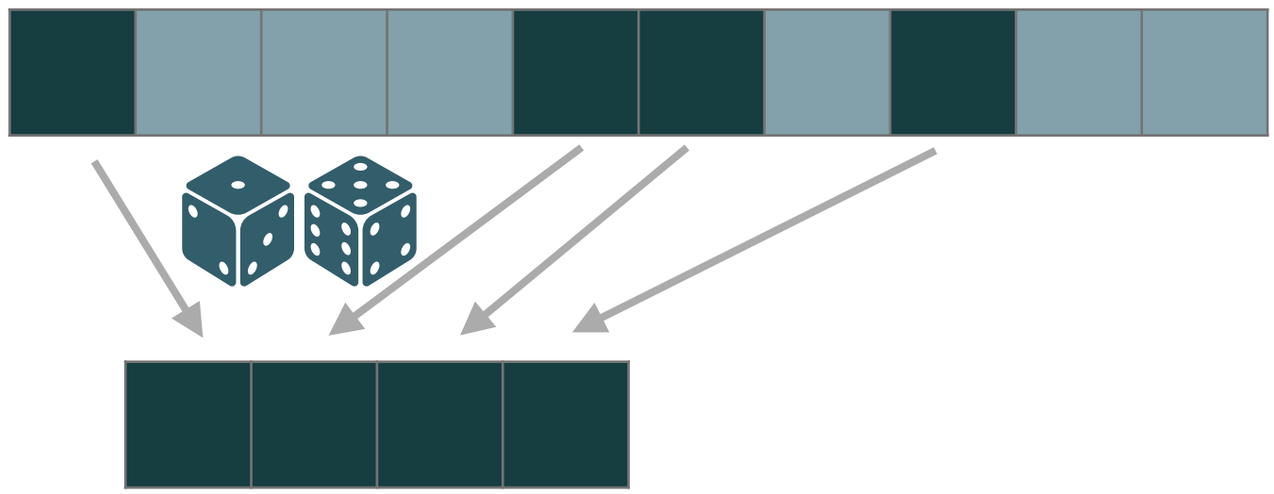
P1035 `view::drop_while`

`view::tail == view::drop(1)`

Filters

`view::delimit`

`view::sample`



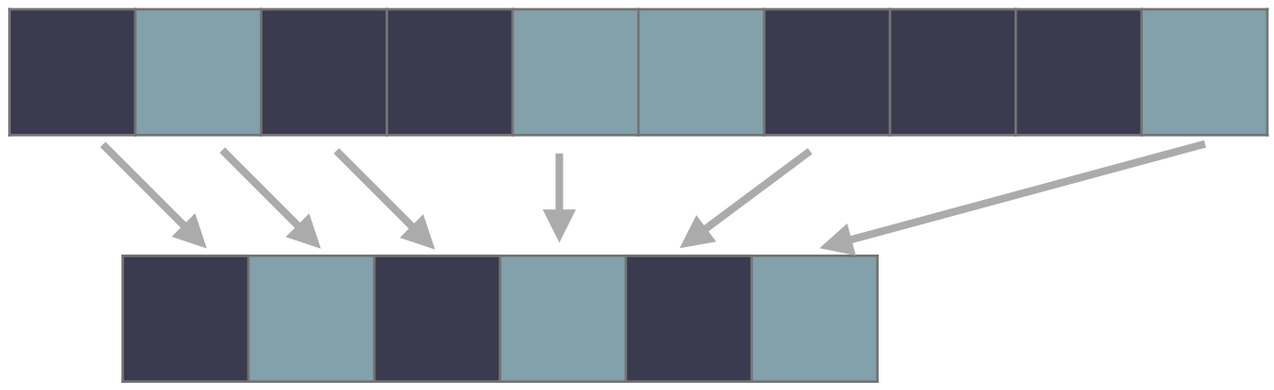
Filters

`view::slice`

`view::stride`

Filters

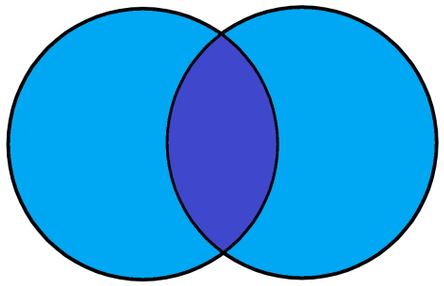
`view::unique`



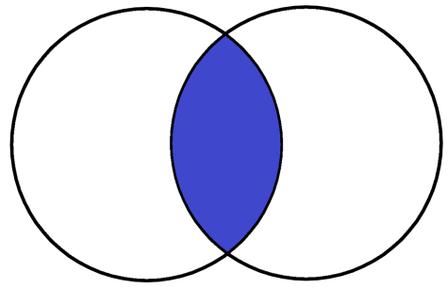
`view::adjacent_remove_if`

`view::adjacent_filter`

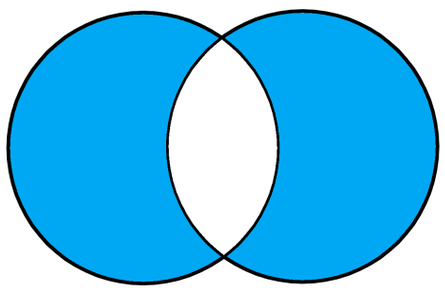
Set views



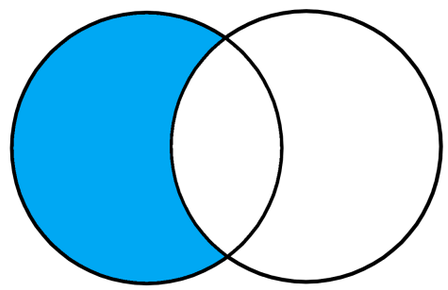
`view::set_union`



`view::set_intersection`



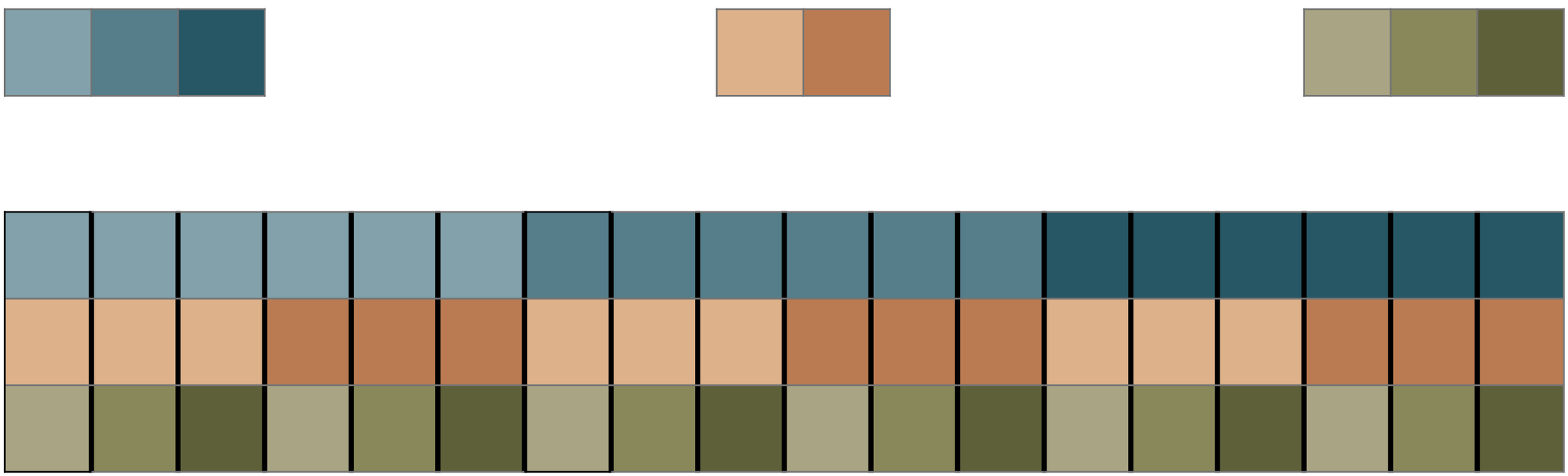
`view::set_symmetric_difference`



`view::set_difference`

Set views

```
view::cartesian_product
```

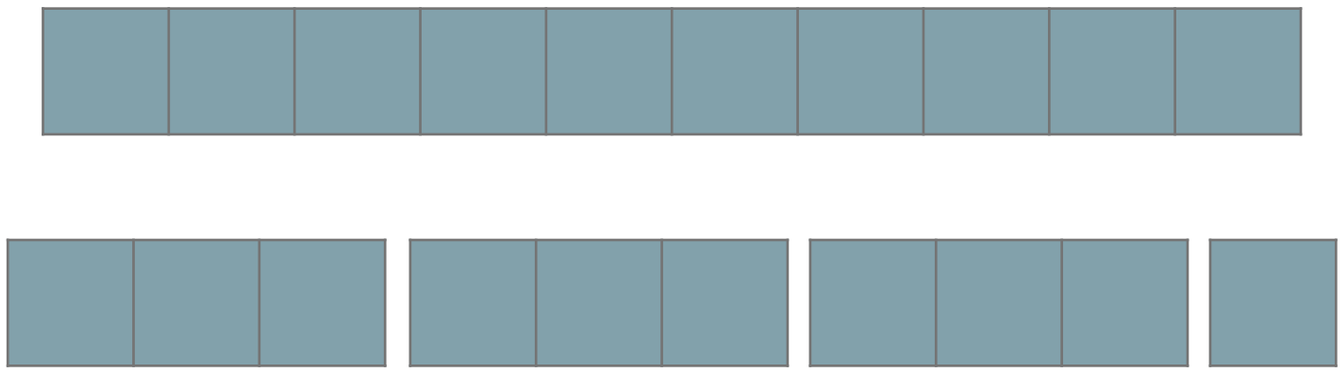


To multiple ranges

`view::chunk`

To multiple ranges

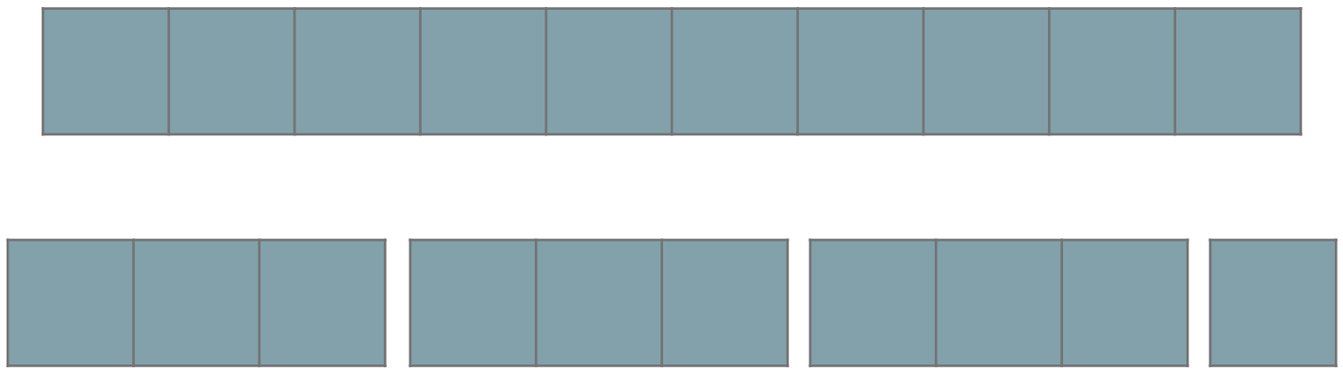
`view::chunk`



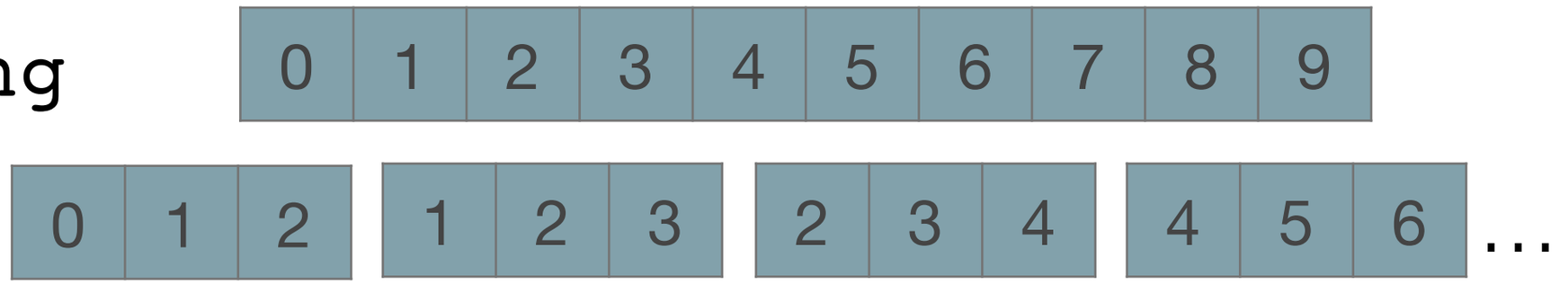
`view::sliding`

To multiple ranges

`view::chunk`



`view::sliding`

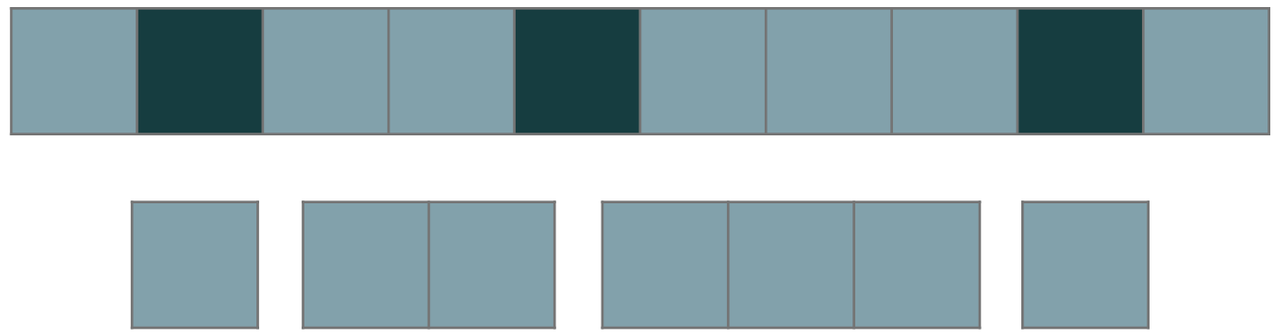


To multiple ranges

`view::split`

To multiple ranges

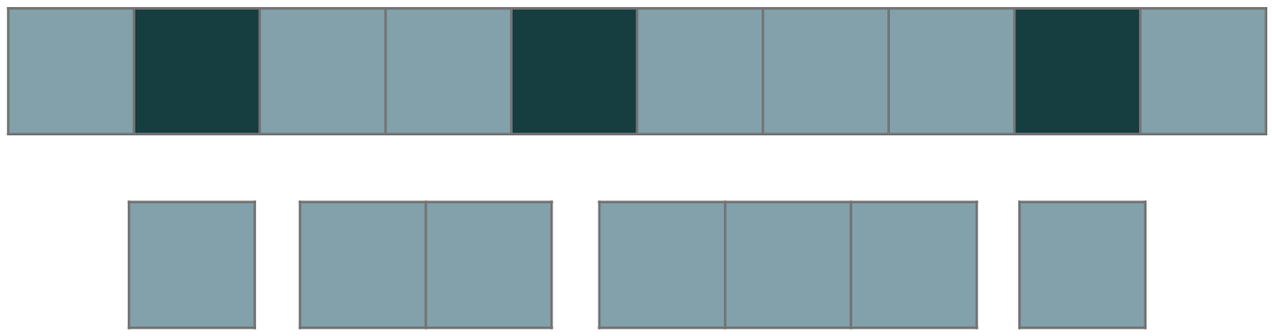
`view::split`



`view::group_by`

To multiple ranges

`view::split`



`view::group_by`

From multiple ranges

`view::concat`

From multiple ranges

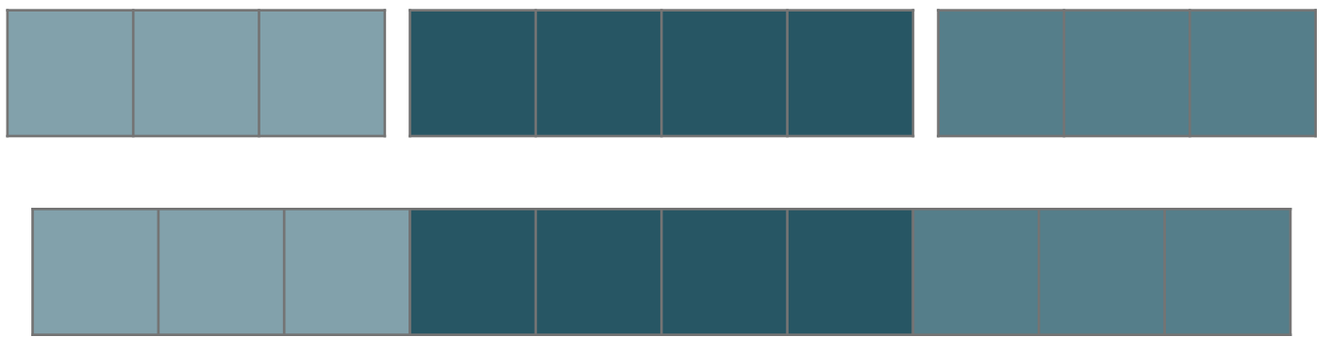
`view::concat`

`view::join`

From multiple ranges

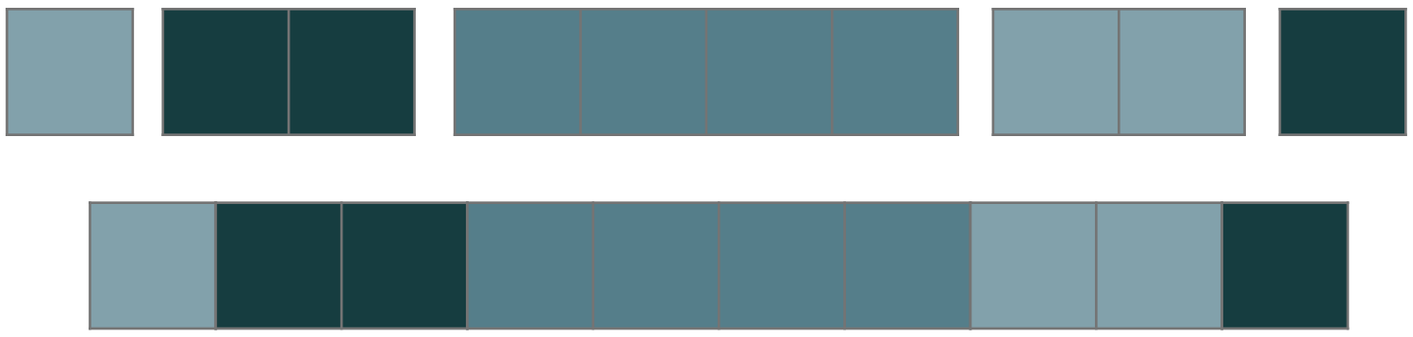
`view::concat`

`view::join`

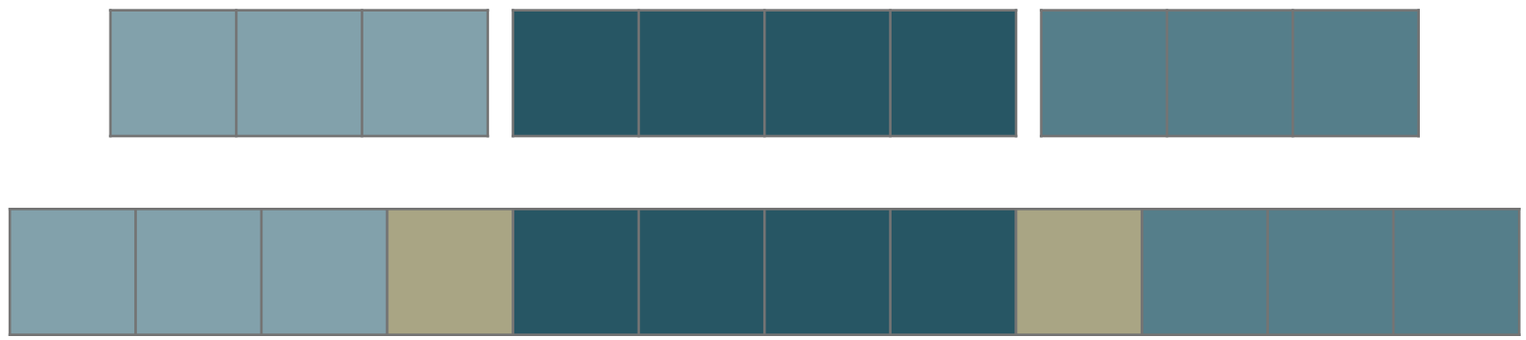


From multiple ranges

`view::concat`

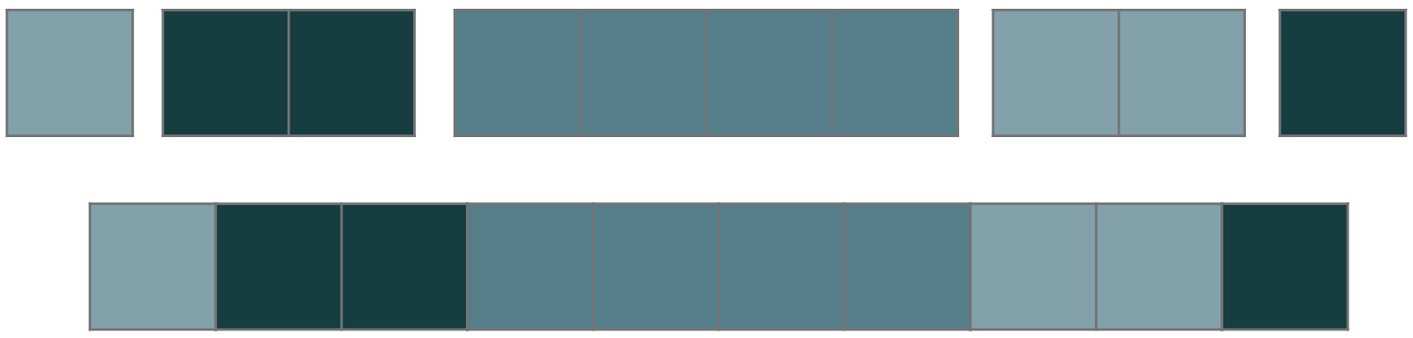


`view::join`

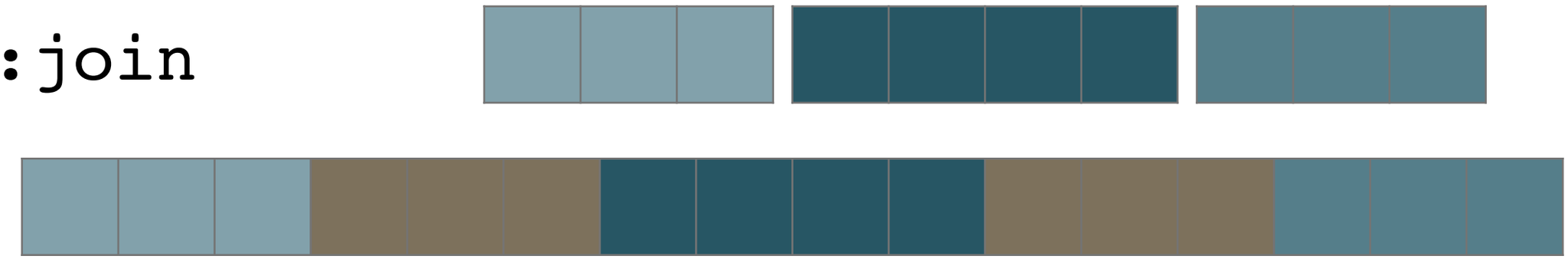


From multiple ranges

`view::concat`

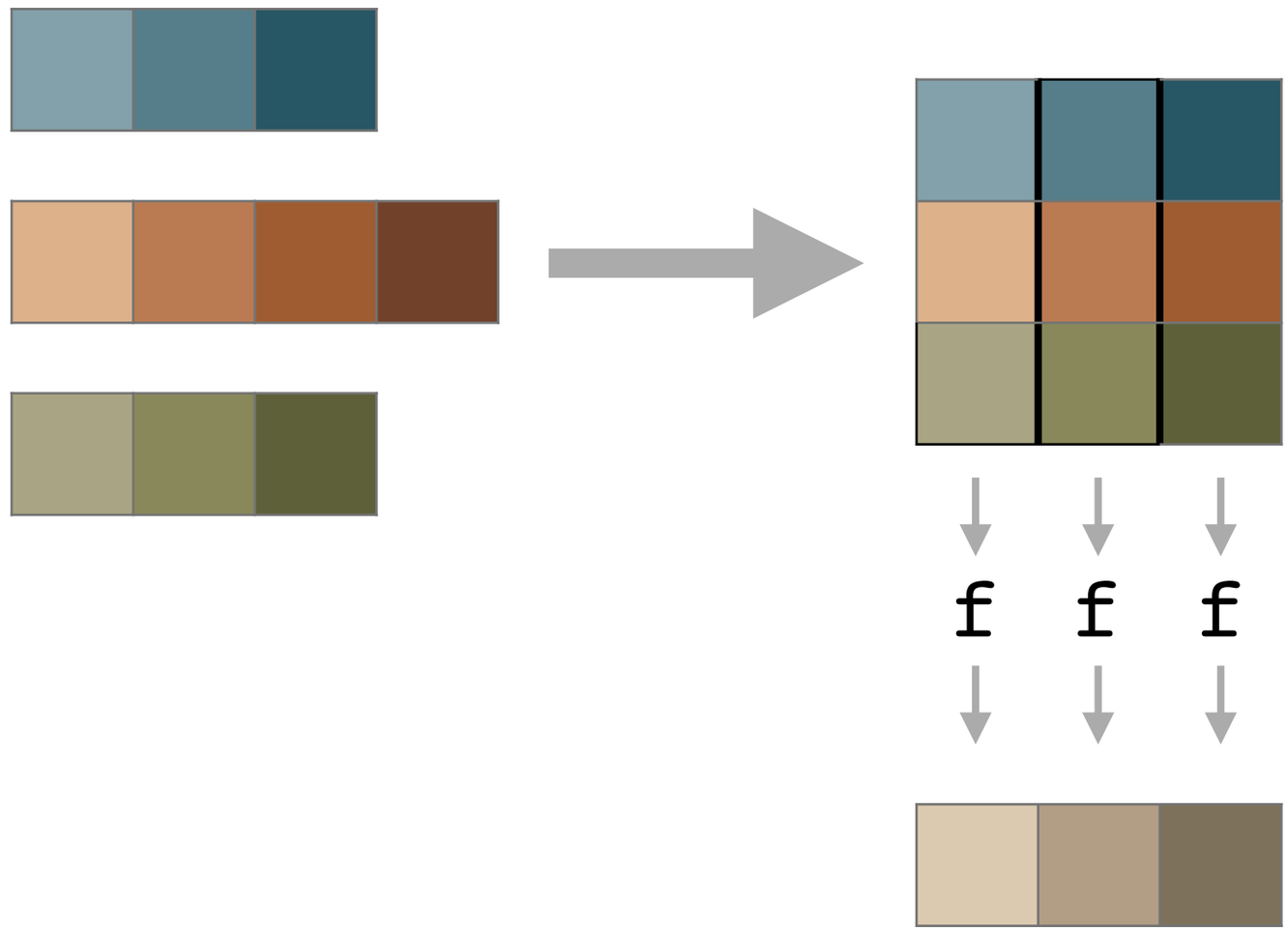


`view::join`



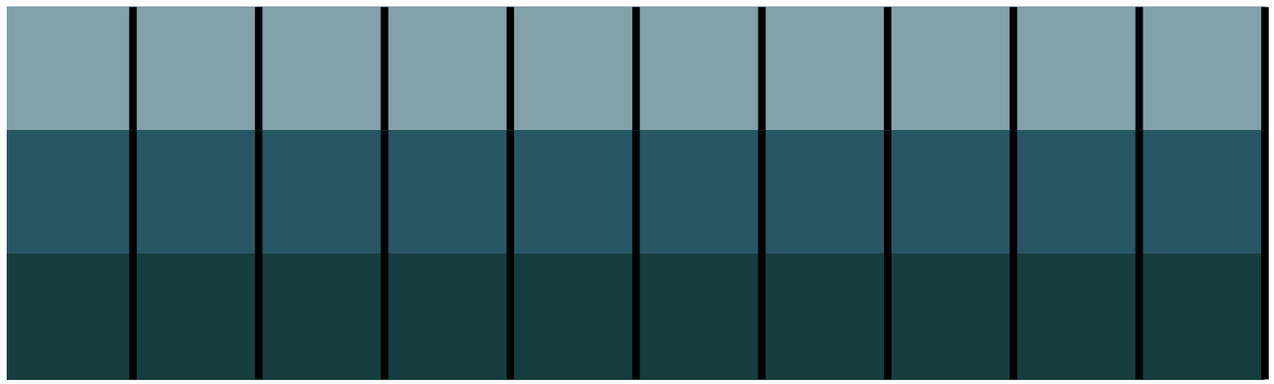
From multiple ranges

`view::zip`



`view::zip_with`

Tuple views



Tuple views

P1035

```
view::elements<0>
```

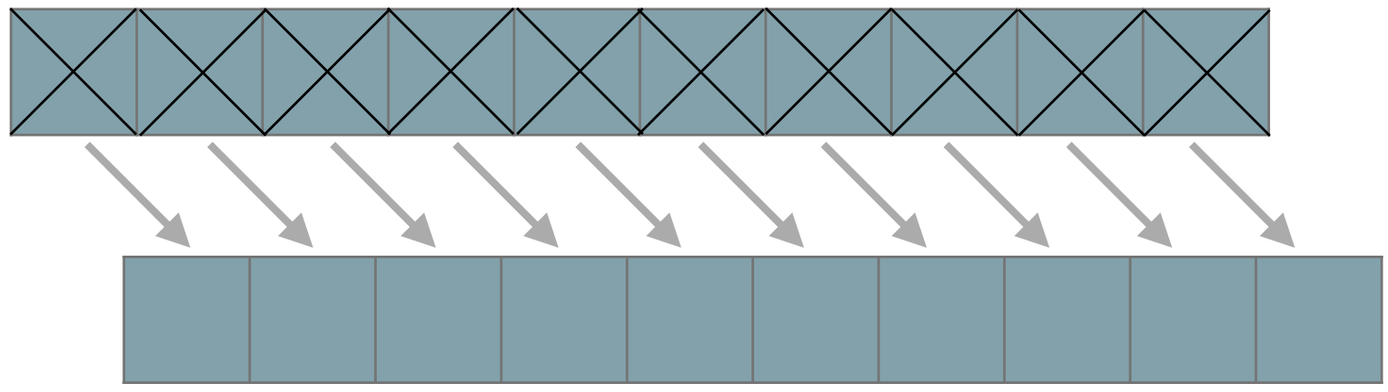
```
view::elements<1>
```

```
view::elements<2>
```

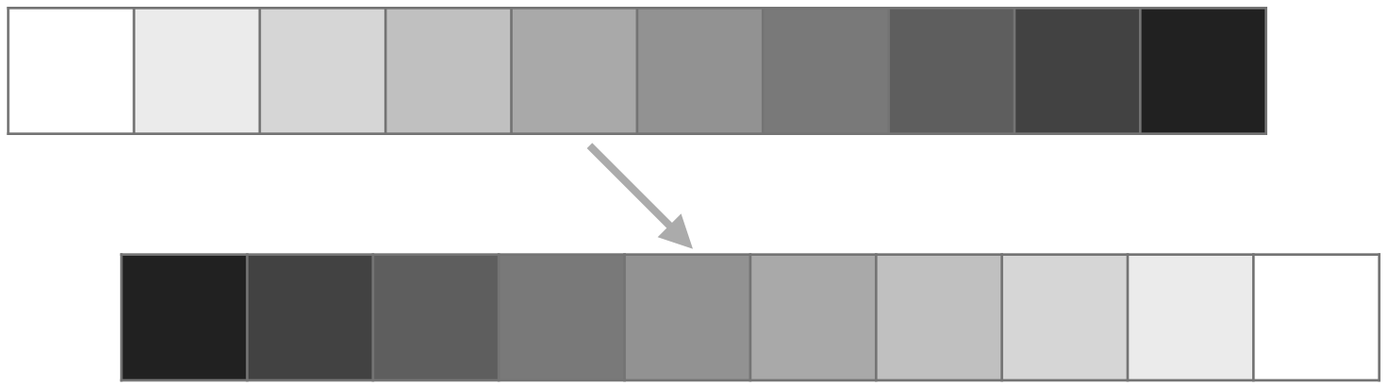
```
view::keys   == view::elements<0>
view::values == view::elements<1>
```

Transformers

`view::move`

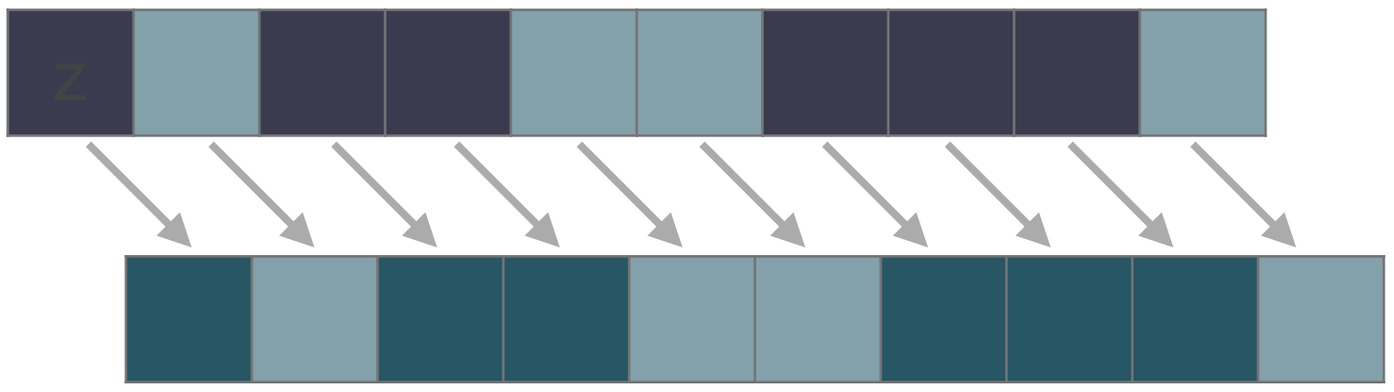


`view::reverse`

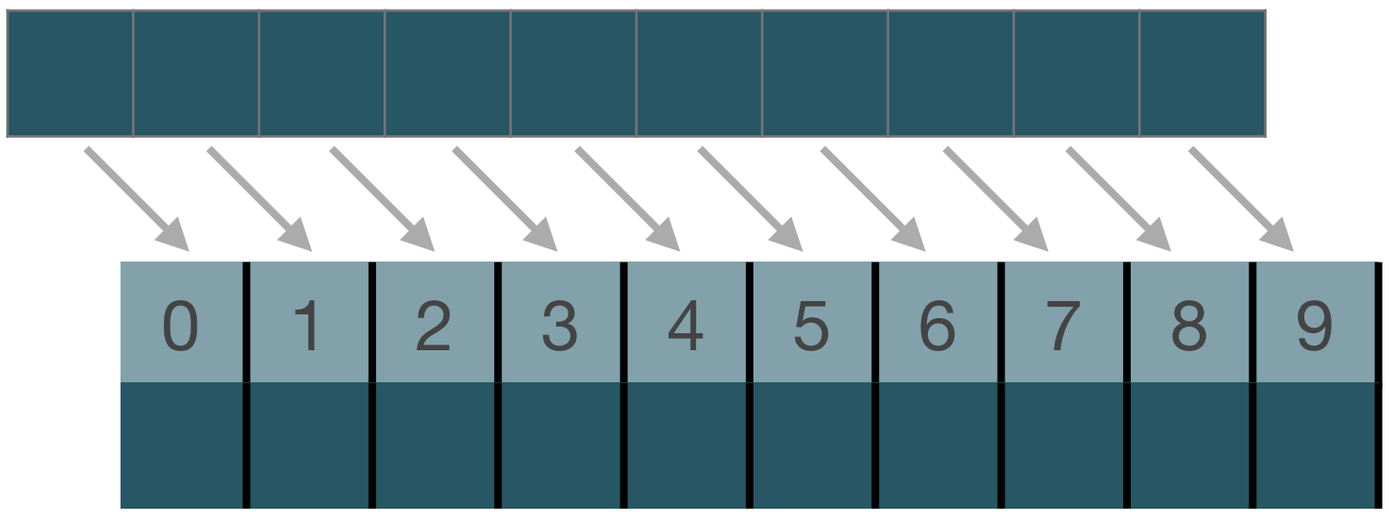


Transformers

`view::replace`
`view::replace_if`

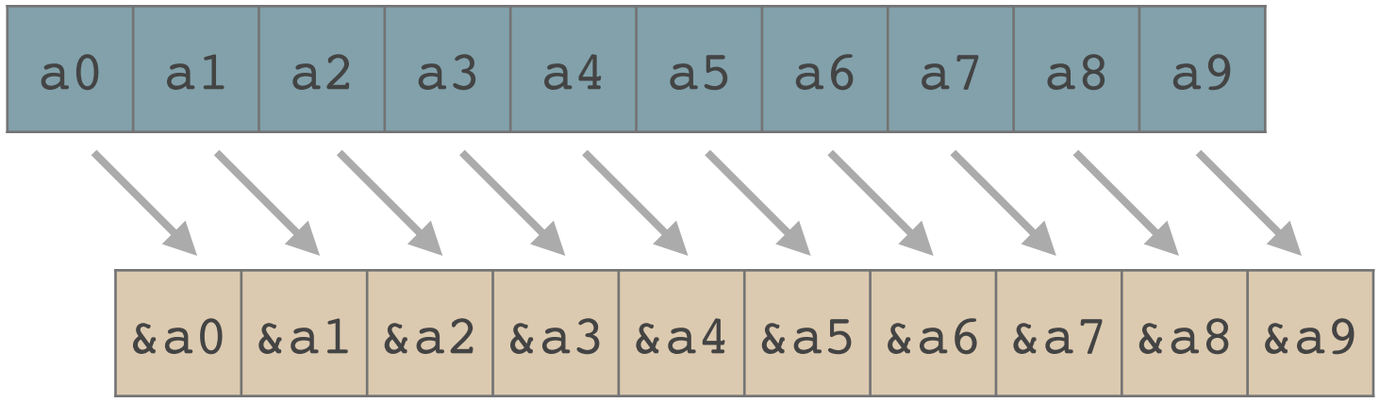


`view::enumerate`

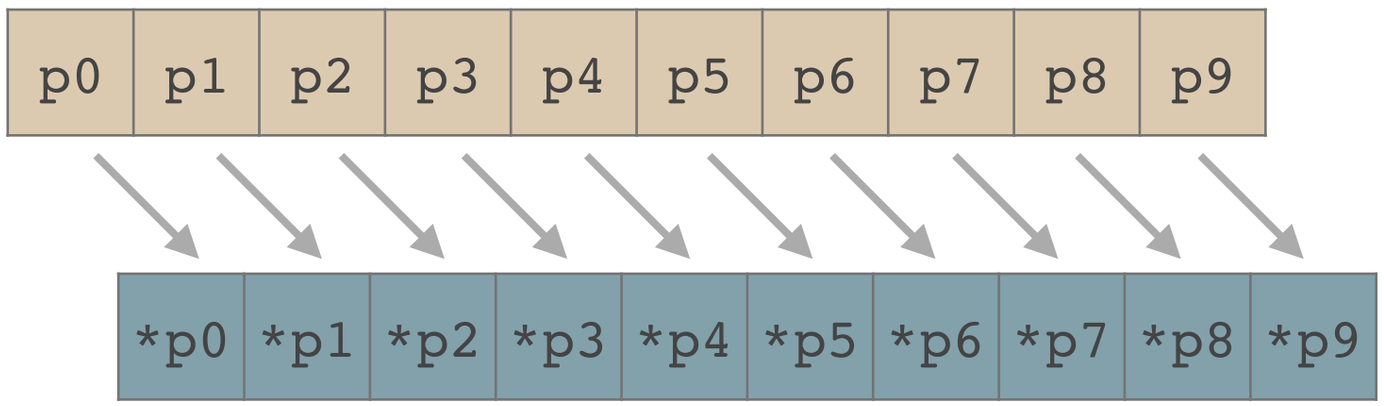


Transformers

`view::addressof`

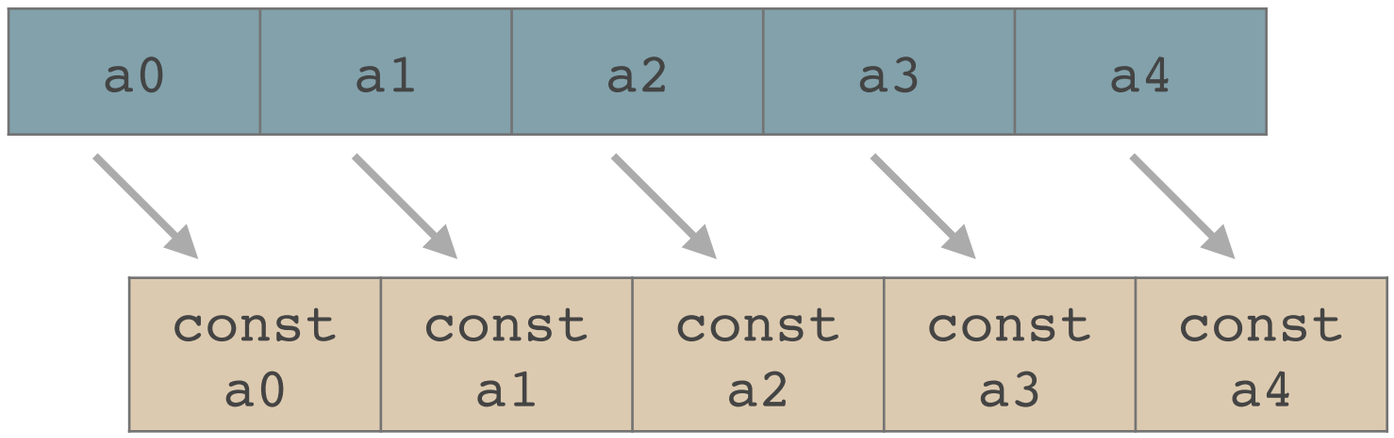


`view::indirect`

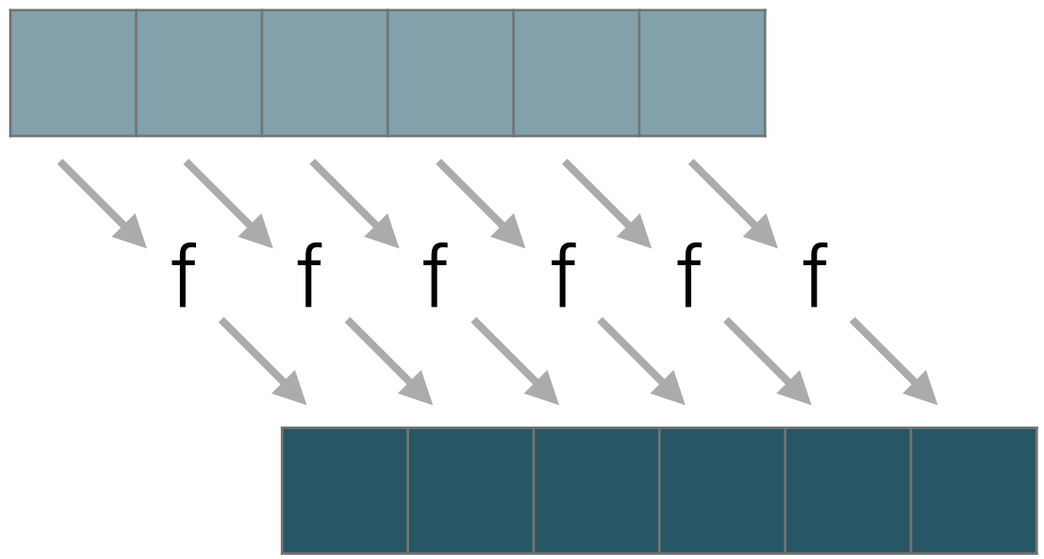


Transformers

`view::const_`



`view::transform`



Adding elements

`view::intersperse`

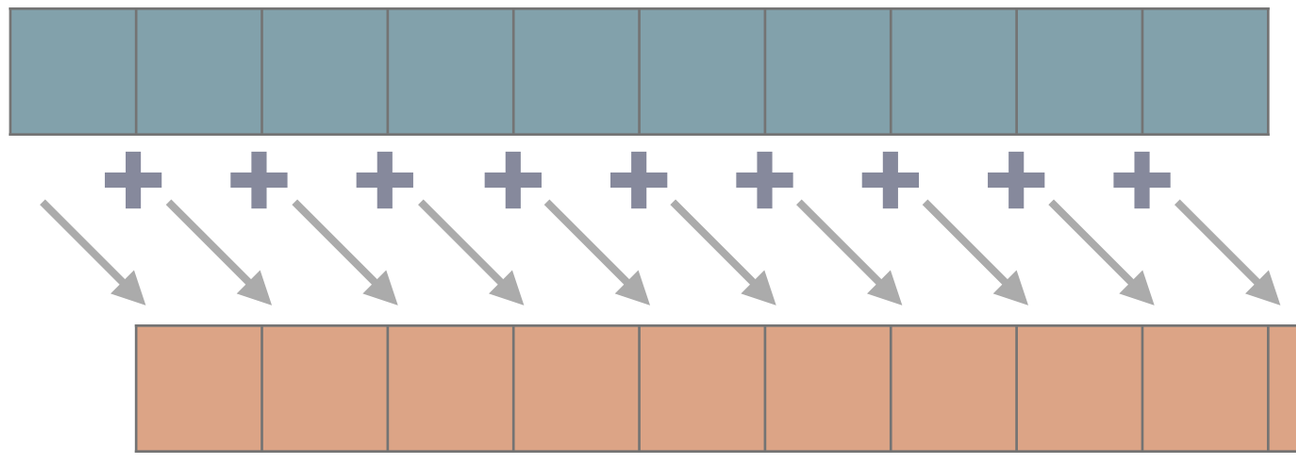
Adding elements

`view::intersperse`

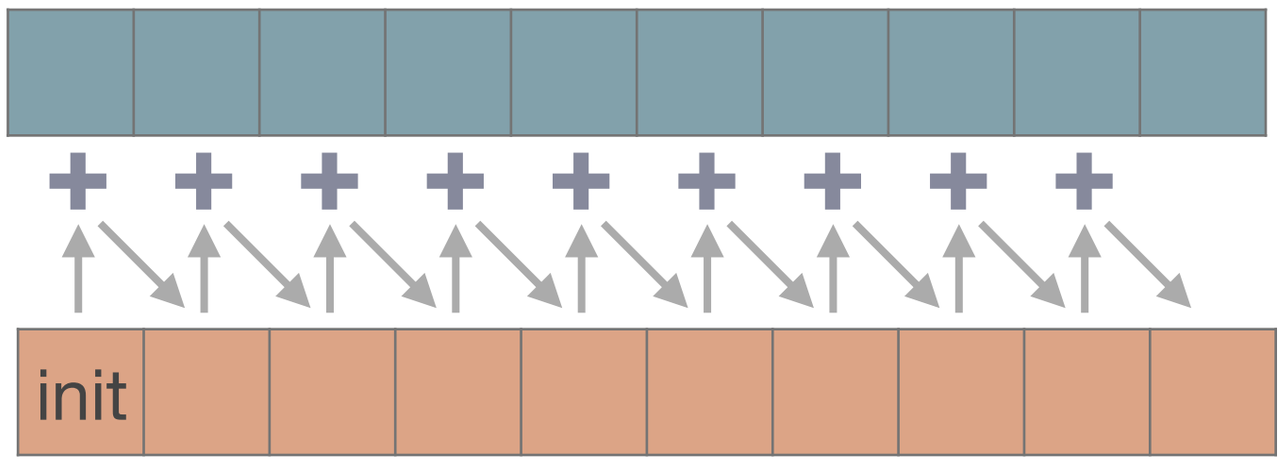
`view::cycle`

Numeric

`view::partial_sum`



`view::exclusive_scan`



Utility views

`view::common/view::bounded`

Range → CommonRange

`any_view<T>`

Can store any range with value type `T`

List comprehensions

`view::for_each`

Lazily applies to each element in the source range a unary function that returns another range (possibly empty), flattening the result.

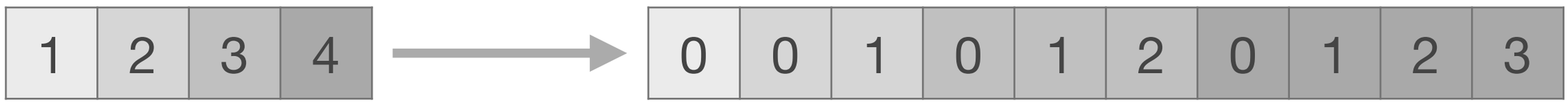
```
view::for_each(
  rng,
  [](int i){
    return yield(i * i);
  });
```

List comprehensions

`view::for_each`

Lazily applies to each element in the source range a unary function that returns another range (possibly empty), flattening the result.

```
view::for_each(
   rng,
   [](int i){
     return yield_from(view::indices(i));
   });
```



List comprehensions

`view::for_each`

Lazily applies to each element in the source range a unary function that returns another range (possibly empty), flattening the result.

```
view::for_each(
   rng,
   [](int i){
     return yield_if(i % 2 == 0, i / 2);
   });
```

List comprehensions

`view::for_each`

Lazily applies to each element in the source range a unary function that returns another range (possibly empty), flattening the result.

```
view::for_each(
   rng,
   [](int i){
      return lazy_yield_if(i % 2 == 0,
         [i] { return i / 2; });
   });
```

ranges::for_each

```
namespace ranges {

    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryInvocable<projected<I, Proj>> Fun>
      constexpr for_each_result<I, Fun>
        for_each(I first, S last, Fun f, Proj proj = {});

    template<InputRange R, class Proj = identity,
             IndirectUnaryInvocable<projected<iterator_t<R>, Proj>> Fun>
      constexpr for_each_result<safe_iterator_t<R>, Fun>
        for_each(R&& r, Fun f, Proj proj = {});
}
```

safe_iterator_t

```
template<Range R>
    using safe_iterator_t = conditional_t<forwarding-range<R>, iterator_t<R>,
dangling>;
```

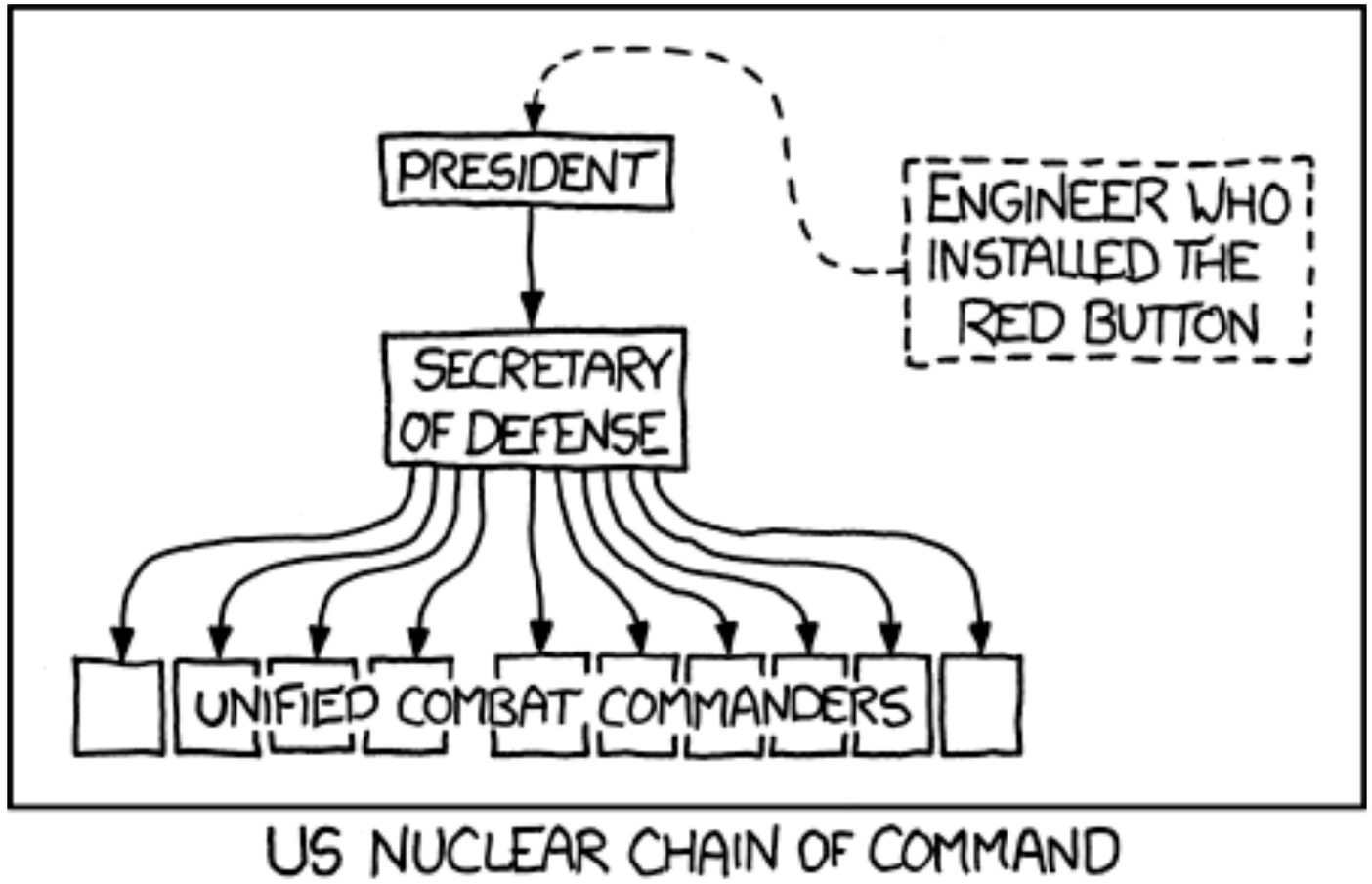
```
std::vector<int> f();

auto result1 = find(f(), 42);
static_assert(std::is_same_v<decltype(result1), dangling>);
// *result1 does not compile

auto vec     = f();
auto result2 = find(vec, 42);
static_assert(std::is_same_v<decltype(result2), std::vector<int>::iterator>);

auto result3 = find(view::all{vec}, 42);
static_assert(std::is_same_v<decltype(result3), std::vector<int>::iterator>);
```

Projection vs. `view::transform`

```
auto it = find(employees, "Sean", &employee::first);
static_assert(std::is_same_v<decltype(*it), employee&>);
```

```
auto v = employees | view::transform(&employee::first);
auto it2 = find(v, "Sean");

static_assert(std::is_same_v<decltype(*it2), std::string&>);
```

Composition



https://www.xkcd.com/898/

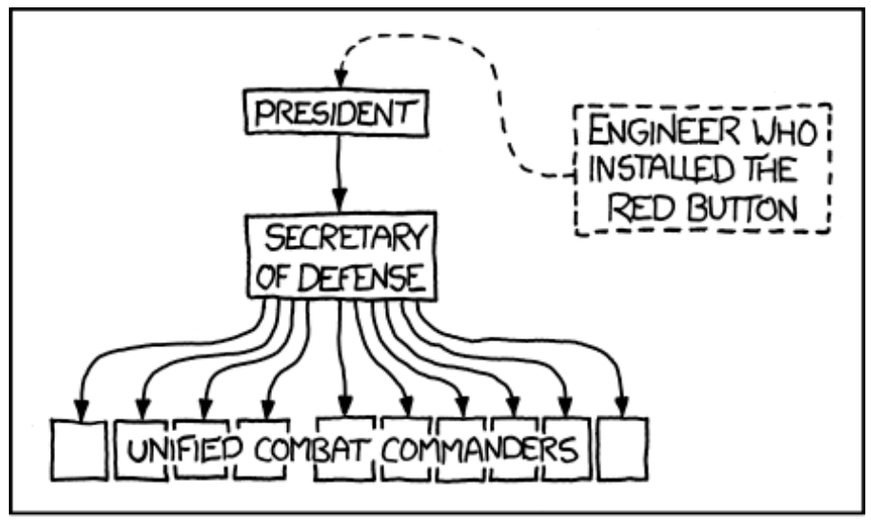
Composition

The bitwise OR operator is overloaded for the purpose of creating adaptor chain pipelines.

```
range | adaptor(args...)
```

The adaptors also support function call syntax with equivalent semantics.

```
adaptor(range, args...)
adaptor(args...)(range)
```



US NUCLEAR CHAIN OF COMMAND

https://www.xkcd.com/898/

Sum of first count squares:

```
int sum_of_squares(int count) {
  std::vector<int> numbers(static_cast<size_t>(count));
  std::iota(numbers.begin(), numbers.end(), 1);
  std::transform(numbers.begin(), numbers.end(), numbers.begin(),
    [](int x) { return x * x; });
  return std::accumulate(numbers.begin(), numbers.end(), 0);
}
```

Sum of first count squares:

```
int sum_of_squares(int count) {
  return accumulate(view::iota(1)
    | view::transform([](int x) { return x * x; })
    | view::take_exactly(count), 0);
}
```

Or equivalently:

```
return accumulate(
    view::take_exactly(
      view::transform(
        view::iota(1),
        [](int x) { return x * x; }
      ),
      count
    ),
    0
  );
```

View materialization

P1206

Save view elements to a container:

```
auto vec = view::ints
| view::transform([](int i) {return i + 42;})
| view::take(10)
| to<std::vector>();
```

```
const std::string names[] = {"john", "paul", "george", "richard"};
const int songs[]         = {72, 70, 22, 2};
auto map = view::zip(names, songs) | to<std::map>();
```

Actions

https://www.xkcd.com/311/

Actions

- We saw algorithms that are eager but that don't compose.

- We saw views that are lazy and do compose.

https://www.xkcd.com/311/

- What about composable AND eager?

- Actions are composable algorithms that operate eagerly on container-like things, mutating them in-place.

Adding elements

`action::push_back`

`action::push_front`

`action::insert`

Adding elements

`action::push_back`

`action::push_front`

`action::insert`

Adding elements

`action::push_back`

`action::push_front`

`action::insert`

Removing elements

`action::erase`

Removing elements

`action::erase`

Removing elements

`action::erase`

And more...

```
action::take                action::stride
action::take_while          action::reverse
action::drop                action::sort
action::drop_while          action::stable_sort
action::remove_if           action::shuffle
action::unique              action::transform
action::slice               action::split
```

Action pipelines

Read data into a vector, sort it, and make it unique:

before

```
auto vi = read_data();
std::sort(vi);
vi.erase(std::unique(vi),
vi.end());
```

after

```
auto vi = read_data()
  | action::sort
  | action::unique;
```

Action pipelines

- The container is moved through the pipeline, hence required to be an rvalue.

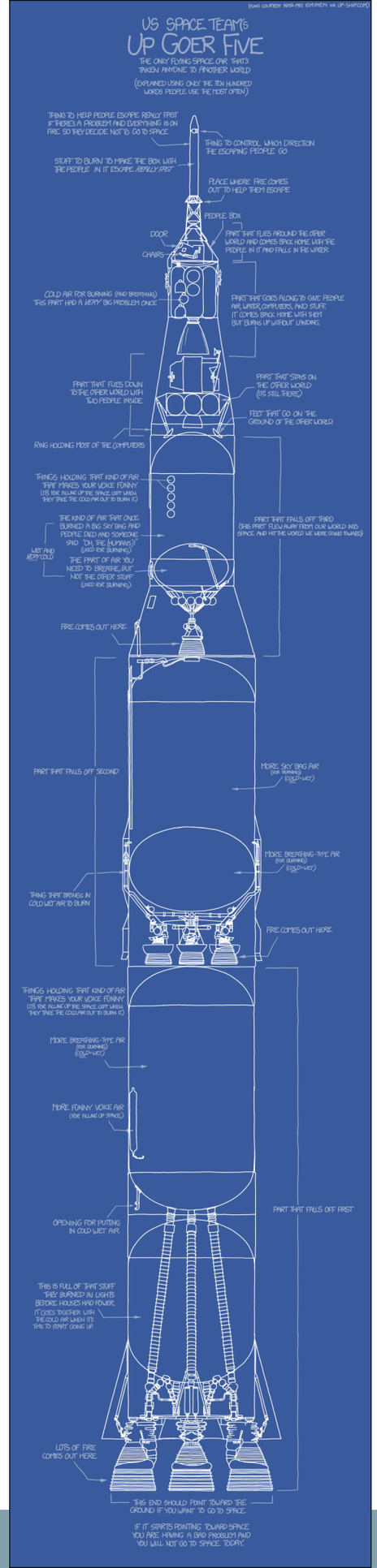
- To operate on an lvalue container, copy or move it:

```
auto v2 = v | copy | action::sort;
auto v3 = v | move | action::sort;
```

- There's an `operator|=` which enables mutating the container in place:

```
v |= action::sort;
```

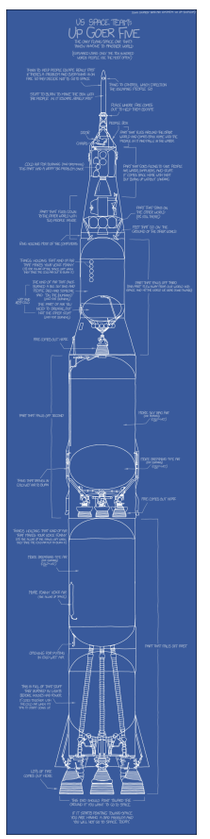
(A Touch of) Performance



https://xkcd.com/1133/

(A Touch of) Performance

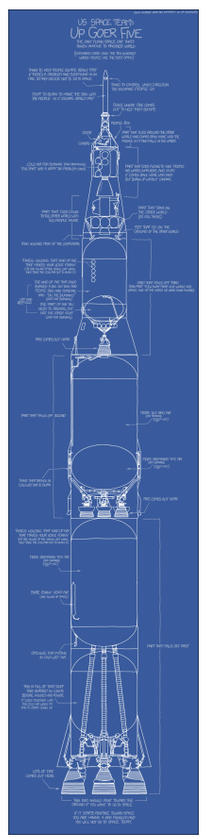
```
int sum_of_squares(int count) {
  std::vector<int> numbers(static_cast<size_t>(count));
  std::iota(numbers.begin(), numbers.end(), 1);
  std::transform(numbers.begin(), numbers.end(), numbers.begin(),
    [](int x) { return x * x; });
  return std::accumulate(numbers.begin(), numbers.end(), 0);
}
```



https://xkcd.com/1133/

(A Touch of) Performance

```
int sum_of_squares(int count) {
  return accumulate(view::iota(1)
    | view::transform([](int x) { return x * x; })
    | view::take_exactly(count), 0);
}
```



https://xkcd.com/1133/

QUESTIONS
FOUND IN GOOGLE AUTOCOMPLETE

WHY DO WHALES JUMP
WHY ARE WITCHES GREEN
WHY ARE THERE MIRRORS ABOVE BEDS
WHY DO I SAY UH
WHY IS SEA SALT BETTER
WHY ARE THERE TREES IN THE MIDDLE OF FIELDS
WHY IS THERE NOT A POKEMON MMO
WHY IS THERE LAUGHING IN TV SHOWS
WHY ARE THERE DOORS ON THE FREEWAY
WHY ARE THERE SO MANY SVCHOST.EXE RUNNING
WHY AREN'T THERE ANY COUNTRIES IN ANTARCTICA
WHY ARE THERE SCARY SOUNDS IN MINECRAFT
WHY IS THERE KICKING IN MY STOMACH
WHY ARE THERE TWO SLASHES AFTER HTTP
WHY ARE THERE CELEBRITIES
WHY DO SNAKES EXIST
WHY DO OYSTERS HAVE PEARLS
WHY ARE DUCKS CALLED DUCKS
WHY DO THEY CALL IT THE CLAP
WHY ARE KYLE AND CARTMAN FRIENDS
WHY IS THERE AN ARROW ON AANG'S HEAD
WHY ARE TEXT MESSAGES BLUE
WHY ARE THERE MUSTACHES ON CLOTHES
WHY ARE THERE MUSTACHES ON CARS
WHY ARE THERE MUSTACHES EVERYWHERE
WHY ARE THERE SO MANY BIRDS IN OHIO
WHY IS THERE SO MUCH RAIN IN OHIO
WHY IS OHIO WEATHER SO WEIRD
WHY ARE THERE MALE AND FEMALE BIKES
WHY ARE THERE BRIDESMAIDS
WHY DO DYING PEOPLE REACH UP
WHY AREN'T THERE VARICOSE ARTERIES
WHY ARE OLD KLINGONS DIFFERENT

WHY ARE THERE SQUIRRELS

WHY IS PROGRAMMING SO HARD
WHY IS THERE A 0 OHM RESISTOR
WHY DO AMERICANS HATE SOCCER
WHY DO RHYMES SOUND GOOD
WHY DO TREES DIE
WHY IS THERE NO SOUND ON CNN
WHY AREN'T POKEMON REAL
WHY AREN'T BULLETS SHARP
WHY DO DREAMS SEEM SO REAL

WHY AREN'T THERE DINOSAUR GHOSTS

WHY DO TESTICLES MOVE
WHY ARE THERE PSYCHICS
WHY ARE HATS SO EXPENSIVE
WHY IS THERE CAFFEINE IN MY SHAMPOO
WHY DO YOUR BOOBS HURT
WHY DO IGUANAS DIE
WHY ARE TREES TALL

WHY ARE THERE FEMALE MR MIMES

WHY ARE THERE SLAVES IN THE BIBLE
WHY DO TWINS HAVE DIFFERENT FINGERPRINTS
WHY ARE AMERICANS AFRAID OF DRAGONS
WHY IS HTTPS CROSSED OUT IN RED
WHY IS THERE A LINE THROUGH HTTPS
WHY IS THERE A RED LINE THROUGH HTTPS ON FACEBOOK
WHY IS HTTPS IMPORTANT

WHY IS YKK ON ALL ZIPPERS
WHY ARE THERE WEEKS
WHY DO I FEEL DIZZY

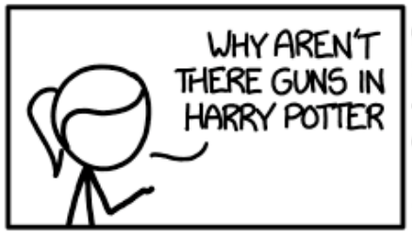
WHY AREN'T ECONOMISTS RICH
WHY DO AMERICANS CALL IT SOCCER
WHY ARE MY EARS RINGING
WHY ARE THERE SO MANY AVENGERS
WHY ARE THE AVENGERS FIGHTING THE X MEN
WHY IS WOLVERINE NOT IN THE AVENGERS
WHY ARE THERE GHOSTS
WHY IS EARTH TILTED
WHY IS SPACE BLACK
WHY IS OUTER SPACE SO COLD
WHY ARE THERE PYRAMIDS ON THE MOON
WHY IS NASA SHUTTING DOWN
WHY ARE THERE TINY SPIDERS IN MY HOUSE
WHY DO SPIDERS COME INSIDE
WHY ARE THERE HUGE SPIDERS IN MY HOUSE
WHY ARE THERE LOTS OF SPIDERS IN MY HOUSE
WHY ARE THERE SPIDERS IN MY ROOM
WHY ARE THERE SO MANY SPIDERS IN MY ROOM
WHY DO SPIDER BITES ITCH
WHY IS DYING SO SCARY
WHY IS THERE NO GPS IN LAPTOPS
WHY DO KNEES CLICK
WHY AREN'T THERE E GRADES
WHY IS ISOLATION BAD
WHY DO BOYS LIKE ME
WHY DON'T BOYS LIKE ME
WHY IS THERE ALWAYS A JAVA UPDATE
WHY ARE THERE RED DOTS ON MY THIGHS
WHY IS LYING GOOD

WHY IS THERE HELL IF GOD FORGIVES
WHY IS GPS FREE
WHY ARE THERE LAVA LAMPS

WHY ARE THERE SWARMS OF GNATS
WHY IS THERE PHLEGM
WHY ARE THERE SO MANY CROWS IN ROCHESTER, MN
WHY IS PSYCHIC WEAK TO BUG
WHY DO CHILDREN GET CANCER
WHY IS POSEIDON ANGRY WITH ODYSSEUS
WHY IS THERE ICE IN SPACE
WHY ARE THERE ANTS IN MY LAPTOP

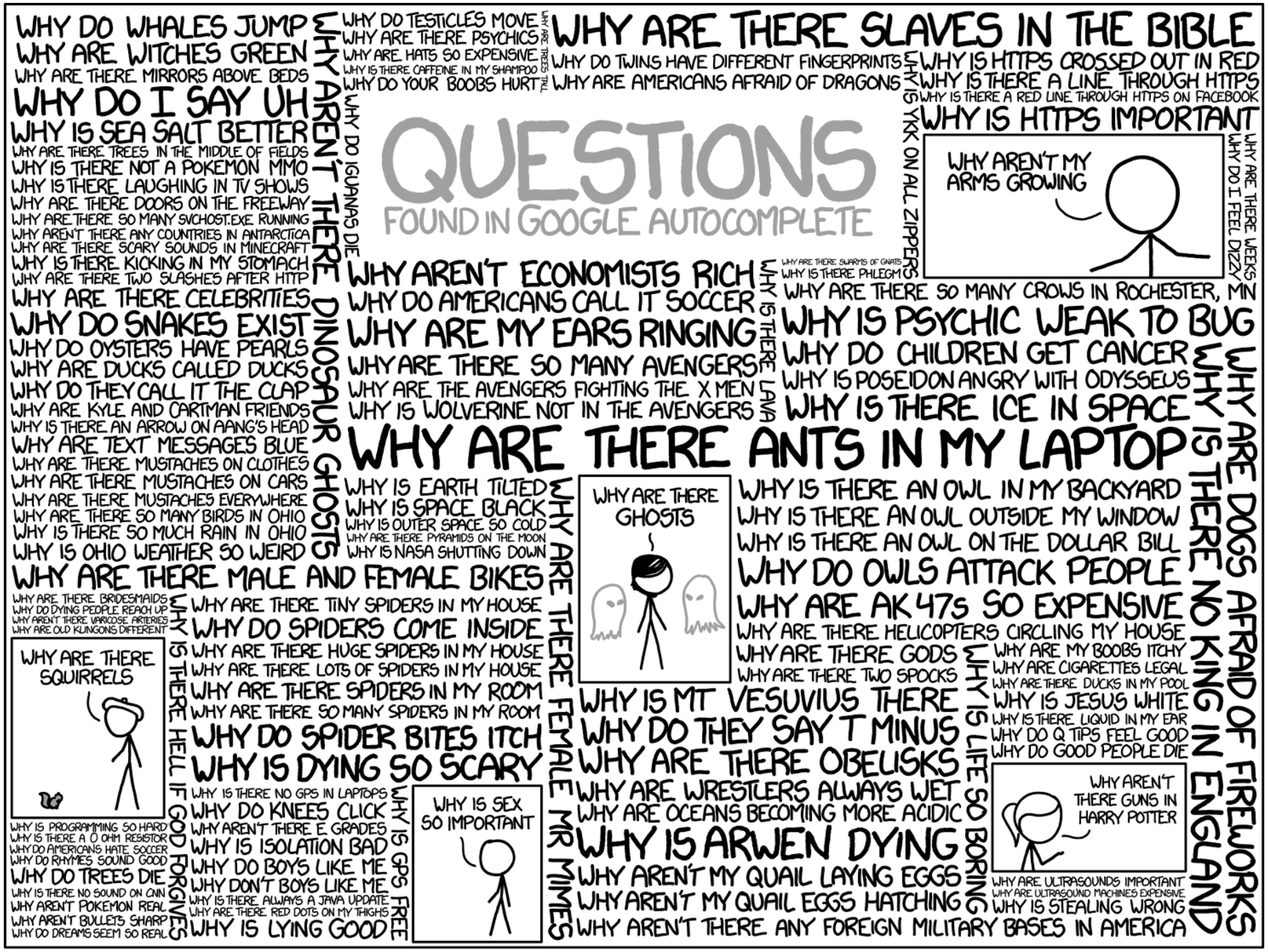
WHY IS THERE AN OWL IN MY BACKYARD
WHY IS THERE AN OWL OUTSIDE MY WINDOW
WHY IS THERE AN OWL ON THE DOLLAR BILL
WHY DO OWLS ATTACK PEOPLE
WHY ARE AK 47s SO EXPENSIVE
WHY ARE THERE HELICOPTERS CIRCLING MY HOUSE
WHY ARE THERE GODS
WHY ARE THERE TWO SPOCKS
WHY IS MT VESUVIUS THERE
WHY DO THEY SAY T MINUS
WHY ARE THERE OBELISKS
WHY ARE WRESTLERS ALWAYS WET
WHY ARE OCEANS BECOMING MORE ACIDIC
WHY IS ARWEN DYING
WHY AREN'T MY QUAIL LAYING EGGS
WHY AREN'T MY QUAIL EGGS HATCHING
WHY AREN'T THERE ANY FOREIGN MILITARY BASES IN AMERICA

WHY IS LIFE SO BORING
WHY ARE MY BOOBS ITCHY
WHY ARE CIGARETTES LEGAL
WHY ARE THERE DUCKS IN MY POOL
WHY IS JESUS WHITE
WHY IS THERE LIQUID IN MY EAR
WHY DO Q TIPS FEEL GOOD
WHY DO GOOD PEOPLE DIE



WHY ARE ULTRASOUNDS IMPORTANT
WHY ARE ULTRASOUND MACHINES EXPENSIVE
WHY IS STEALING WRONG

WHY ARE THERE DOGS AFRAID OF FIREWORKS
WHY IS THERE NO KING IN ENGLAND



https://www.xkcd.com/1256/

- dvirtz at GitHub, slack and gmail
- @dvirtzwastaken on Twitter