Plan

1. *Machine model. Objects. Values. Assignment, swap, move.*
2. Introductory algorithms: advance, distance, find, copy. *Iterators: operations, properties, classification. Ranges and their validity. Linear recursion and iteration.*
3. Permutations. Cyclical structure, upper and lower bound. *Complexity analysis.*
4. Position-based permutations: random shuffle, reverse, rotate. *Loop invariants and correctness proofs.*
5. Predicate-based permutations: partition, stable partition. *Inductive properties and algorithm construction.*
6. *Reduction and bottom-up tree recursion.*
7. *Ordering, total ordering, weak ordering, strict ordering.* Lower and upper bounds. Merging and set operations. Insertion.
8. Stable merge and *top-down tree recursion.*
9. Merge-sort and *memory adaptive algorithms*.
10. *Linear representations of trees.* Heaps, d-heaps(?). Partial sort.
11. *Assembling a large function.* Sort. *Introspection.*
12. Node based algorithms. *Order changing iterators. Pointer manipulations.*
13. Simple linear data structures: vectors and lists. *Building data structures*.
14. *Axioms of programming: regular types, iterators, insertable containers. Semantics of overloading operations.*
15. *Concepts as a foundational mechanism. Type constraints.*

Appendices

1. GCD
2. C++ bindings
3. Optimizations. Inlining, common sub-expression, etc.
4. Measurements.
5. Testing and validation.
6. Visualization

All of the algorithms in the book presuppose that they are applied to a valid range.

Chapter 1. Introductory algorithms and programming language.

```
Type[3] bind1st(BinaryFunction function, Type[1] value)
     (Type[2] x)
{
     return function(value, x);
```

```
}


difference_type(InputIterator) distance( InputIterator first,
                                         InputIterator last )
{
     difference_type(InputIterator) n = 0;

     while (first != last) {
            ++first;
            ++n;
     }

     return n;
}

difference_type(RandomAccessIterator)
distance( RandomAccessIterator first,
          RandomAccessIterator last )
{
     return last - first;
}

void advance( InputIterator& first,
              Integer        n )
{
     assert(!(n < 0));

     while (n != 0) {
          ++first;
          --n;
     }
}

void advance( BidirectionalIterator & first,
              SignedInteger           n )
{
     while (n > 0) {
          ++first;
          --n;
     }
     while (n < 0) {
          --first;
          ++n;
     }
}

void advance( RandomAccessIterator & first,
              SignedInteger          n)
{
    first += n;
}

InputIterator successor( InputIterator x )
{
     ++x;
     return x;
```

```
}

InputIterator successor( InputIterator x,
                         Integer       n )
{
     advance(x, n);
     return x;
}

BidirectionalIterator predecessor (BidirectionalIterator x )
{
     --x;
     return x;
}

BidirectionalIterator predecessor (BidirectionalIterator x,
                                   Integer               n )
{
     advance(x, -n);
     return x;
}


ForwardIterator back(ForwardIterator first,
                     ForwardIterator last) {
     assert(first != last);

     ForwardIterator previous;

     do {
          previous = first;
          ++first;
     } while (first != last);

     return previous;
}

BidirectionalIterator back( BidirectionalIterator first,
                            BidirectionalIterator last ) {
     assert(first != last);

     return predecessor(last);

}


InputIterator find_if( InputIterator  first,
                       InputIterator  last,
                       UnaryPredicate predicate )
where (value_type(InputIterator) == argument_type(UnaryPredicate))
{
      precondition(valid_range(first, last));

      while (first != last && !predicate(*first)) {
            ++first;
      }
```

```
      postcondition(first == last || predicate(*first));

      return first;
}


InputIterator
find( InputIterator                              first,
      InputIterator                              last,
      const argument_type(BinaryPredicate, 2)& value,
      BinaryPredicate                            binary_predicate
                                                     = equal_to)
{
      return find_if(first, last, bind2nd(binary_predicate, value));
}


InputIterator find_if_unguarded( InputIterator  first,
                                 UnaryPredicate predicate )
{
      while (!predicate(*first)) {
            ++first;
      }

      postcondition(predicate(*first));

      return first;
}


InputIterator find_unguarded( InputIterator   first,
                              const T&        value,
                              BinaryPredicate binary_predicate =
                                 equal_to)
{
      return find_if_unguarded(first, bind2nd(binary_predicate, value));
}



pair(InputIterator, Integer) find_if_n( InputIterator   first,
                                        Integer         n,
                                        UnaryPredicate  predicate )
{
      precondition(valid_range_n(first, n));

      while(n > 0 && !predicate(*first)) {
            ++first;
            --n;
      }

      postcondition(n == 0 || predicate(*first));

      return pair(first, n);
}
```

```
pair(InputIterator, Integer) find_if_n( InputIterator   first,
                                        Integer         n,
                                        UnaryPredicate  predicate ) optimized
{
     precondition(valid_range_n(first, n));

     if (n <= 0) goto exit;

     const int unroll_factor = 4;

     Integer number_of_iterations =
          (n + unroll_factor – 1) /unroll_factor;

     switch (n % unroll_factor) {
     case 0: do { if (predicate(*first)) goto exit;
                  ++first; --n;
     case 3:      if (predicate(*first)) goto exit;
                  ++first; --n;
     case 2:      if (predicate(*first)) goto exit;
                  ++first; --n;
     case 1:      if (predicate(*first)) goto exit;
                  ++first; --n;
                  --number_of_iterations;
            } while (number_of_iterations != 0);
      }

exit:
      postcondition(n == 0 || predicate(*first));

      return pair(first, n);

}

pair(RandomAccessIterator, Integer) find_if_n(RandomAccessIterator first,
                                              Integer              n,
                                              UnaryPredicate       predicate )
optimized
{
     precondition(valid_range_n(first, n));

     if (n <= 0) goto exit;

     const int unroll_factor = 4;
     const RandomAccessIterator original_first = first;


     Integer number_of_iterations =
          (n + unroll_factor – 1) /unroll_factor;

     switch (n % unroll_factor) {

     case 0: do { if (predicate(*first)) goto exit;
                  ++first;
     case 3:      if (predicate(*first)) goto exit;
                  ++first;
     case 2:      if (predicate(*first)) goto exit;
```

```
                ++first;
     case 1:    if (predicate(*first)) goto exit;
                ++first;
                --number_of_iterations;
              } while (number_of_iterations != 0);
      }

exit:
     postcondition(n == 0 || predicate(*first));

     return pair(first, n – (first - original_first));

}


RandomAccessIterator find_if( RandomAccessIterator first,
                              RandomAccessIterator last,
                              UnaryPredicate       predicate )
{
      return first(find_if_n(first, last - first, predicate));
}



BidirectionalIterator find_with_sentinel(
                          BidirectionlIterator              first,
                          BidirectionlIterator              last,
                          value_type(BidirectionalIterator) value )
{
      if (first == last) return first;

      --last;
      swap(value, *last);
      first = find_unguarded(first, *last);
      swap(value, *last);

      if (first == last && *first != value) ++first;

      return first;
}

bool exist( InputIterator  first,
            InputIterator  last,
            UnaryPredicate predicate ) {
     return last != find_if(first, last, predicate);
}

bool all( InputIterator  first,
          InputIterator  last,
          UnaryPredicate predicate ) {
     return last == find_if(first, last, negate(predicate));
}

OutputIterator copy( InputIterator first,
                     InputIterator last,
```

```
                    OutputIterator result ) {
     while (first != last) {
            *result = *first;
            ++first;
            ++result;
     }
     return result;
}


pair(InputIterator, OutputIterator) copy_n( InputIterator  first,
                                            Integer        n,
                                            OutputIterator result ) {
     while (n != 0)  {
            *result = *first;
            ++first;
            ++result;
            --n;
     }
     return pair(first, result);
}
```

Problem: Design an unrolled version of copy_n (easy).

Problem: Design a version of copy for random access iterator that uses copy_n (very easy).

```
BidirectionalIterator[2] copy_backward( BidirectionalIterator[1] first,
                                        BidirectionalIterator[1] last,
                                        BidirectionalIterator[2] result ) {
     while (first != last) {
            --last;
            --result;
            *result = *last;
     }
     return result;
}
```

Problem: Design an unrolled version of copy_backward_n (easy).

Problem: Design a version of copy_backward for random access iterator that uses copy_backward_n (very easy).

```
ForwardIterator[1] swap_ranges( ForwardIterator[1] first1,
                                ForwardIterator[1] last1,
                                ForwardIterator[2] first2 )
{
     while (first1 != last1) {
            swap(*first1, *first2);
            ++first1;
            ++first2;
     }

     return first2;
}
```

```
pair(ForwardIterator[1], ForwardIterator[2])
swap_ranges_n( ForwardIterator[1] first1,
               ForwardIterator[2] first2,
               Integer            n )
{
     while (n != 0) {
            swap(*first1, *first2);
            ++first1;
            ++first2;
            --n;
     }

     return pair(first1, first2);
}
```

```
pair(ForwardIterator[1], ForwardIterator[2])
swap_ranges( ForwardIterator[1] first1,
             ForwardIterator[1] last1,
             ForwardIterator[2] first2,
             ForwardIterator[2] last2 )
{
     while (first1 != last1 && first2 != last2) {
            swap(*first1, *first2);
            ++first1;
            ++first2;
     }

     return pair(first1, first2);
}
```

Permutations

Theorem. Every permutation is decomposable into a product of cycles.

Theorem. An in-place permutation requires at least `n + ntc – tc` assignments where n is the number of elements in the permutation, ntc is the number of non-trivial cycles, and tc is the number of trivial cycles in the permutation.

```
void reverse( BidirectionalIterator first,
              BidirectionalIterator last )
{
     while (first != last && first != --last) {
          swap(*first, *last);
          ++first;
     }
}

void reverse_n( BidirectionalIterator first,
                BidirectionalIterator last,
                difference_type n )
{
```

```
     n /= 2;

     while (n != 0) {
          --last;
          swap(*first, *last);
          ++first;
          --n;
     }
}

void reverse( RandomAccessIterator first,
              RandomAccessIterator last )
{
    reverse_n(first, last, last – first);
}
```

Problem: Design `reverse` that works on forward iterators and uses not more than O(log(n)) of additional storage and not more than O(nlog(n)) steps, where n is the size of the range (hard).

Solution:

```
ForwardIterator reverse_n( ForwardIterator first,
                           Integer         n )
{
     if (n == 0) return first;
     if (n == 1) return ++first;

     ForwardIterator middle = reverse_n(first, n/2);
     if (n % 2 == 1) ++middle;
     ForwardIterator result = reverse_n(middle, n/2);
     swap_ranges_n(first, middle, n/2);
     return result;
}
```

to see how it works:

1 2 3 4 5
2 1 3 4 5
2 1 3 5 4
5 4 3 2 1

```
void reverse( ForwardIterator first,
              ForwardIterator last )
{
          reverse_n(first, distance(first, last));
}
```

Problem: Prove that there is no linear time and polylog space algorithm for reverse on forward iterators (very hard: we are not aware of a solution).

pair(BidirectionalIterator, BidirectionalIterator)

```
reverse_until( BidirectionalIterator first,
               BidirectionalIterator middle,
               BidirectionalIterator last )
{
       while (first != middle && middle != last) {
              --last;
              swap(*first, *last);
              ++first;
       }
       return pair(first, last);
}

OutputIterator reverse_copy( BidirectionalIterator first,
                             BidirectionalIterator last,
                             OutputIterator result ) {
     while (first != last) {
           --last;
           *result = *last;
           ++result;
     }

     return result;
}

void rotate_non_empty( ForwardIterator first,
                       ForwardIterator middle,
                       ForwardIterator last )
{
     ForwardIterator current = middle;

     while (true) {
            swap(*first, *current);
            ++first;
            ++current;
            if (current == last) {
                 if (first == middle) return;
                 current = middle;
            } else if (first == middle) {
                 middle = current;
            }
    }
}

ForwardIterator rotate( ForwardIterator first,
                        ForwardIterator middle,
                        ForwardIterator last )
{
     if (first == middle) return last;
     if (middle == last)  return first;

     ForwardIterator current = middle;

     while (true) {
            swap(*first, *current);
            ++first;
            ++current;
```

```
            if (current == last) {
                 if (first != middle) {
                     rotate_non_empty(first, middle, last);
                 }
                 return first;
            } else if (first == middle) {
                 middle = current;
            }
       }
}

ForwardIterator rotate( ForwardIterator first,
                        ForwardIterator middle,
                        ForwardIterator last )
{
       if (first == middle) return last;
       if (middle == last)  return first;

       ForwardIterator new_middle = last;
       ForwardIterator current = middle;

       while (true) {
              swap(*first, *current);
              ++first;
              ++current;
              if (current == last) {
                   if (new_middle == last) {
                        new_middle = first;
                   }
                   if (first == middle) {
                        return new_middle;
                   }
                   current = middle;
              } else if (first == middle) {
                   middle = current;
              }
       }
}

BidirectionalIterator rotate( BidirectionalIterator first,
                              BidirectionalIterator middle,
                              BidirectionalIterator last )
{
       if (first == middle) return last;
       if (middle == last)  return first;

       reverse(first, middle);
       reverse(middle, last);

       pair(BidirectionalIterator, BidirectionalIterator) new_middle =
              reverse_until(first, middle, last);
       reverse(new_middle.first, new_middle.second);

       if (middle != new_middle.first) {
              return new_middle.first;
       } else {
              return new_middle.second;
```

```
     }
}
```

RANDOM ACCESS VERSION OF ROTATE

```
void  rotate_cycle( RandomAccessIterator initial,
                    RandomAccessIterator new_middle,
                    Integer forward,
                    Integer backward)

{
     ValueType value  = *initial;

     RandomAccessIterator ptr1 = initial;
     RandomAccessIterator ptr2 = initial + forward;

     while (ptr2 != initial) {
          *ptr1 = *ptr2;
          ptr1 = ptr2;
          ptr2 += (new_middle > ptr2) ? forward : backward;
     }
     *ptr1 = value;
}

RandomAccessIterator rotate( RandomAccessIterator first,
                             RandomAccessIterator middle,
                             RandomAccessIterator last )
{
     if (first == middle) return last;
     if (middle == last)  return first;

     RandomAccessIterator new_middle = first + (last – middle);

     DifferenceType forward  = middle – first;
     DifferenceType backward = middle - last;

     RandomAccessIterator end =
          first + gcd(middle – first, last – middle);
     RandomAccessIterator start = first;

     while (start < end) {
          rotate_cycle(start, new_middle, forward, backward);
          ++start;
     }

     return new_middle;
}



OutputIterator rotate_copy( ForwardIterator first,
                            ForwardIterator middle,
```

```
                              ForwardIterator last,
                              OutputIterator  result ) {
     return copy(first, middle, copy(middle, last, result));
}

void random_shuffle( RandomAccessIterator first,
                     RandomAccessIterator last,
                     UnaryFunction random )
{
     RandomAccessIterator current = first;

     while (current != last) {
          swap(*current, *(first + random(current – first));
          ++current;
     }
}
```

Problem: Develop an algorithm that shuffles forward iterator range in place in nlogn operations. (Hard)

Solution:

```
bool random_unary_predicate( UnaryFunction random )
    (const ValueType&)
{
    return random(1) == 0;
}


void random_shuffle( ForwardIterator first,
                     ForwardIterator last,
                     UnaryFunction   random )
{
    if(first == last || successor(first) == last) return;

    ForwardIterator middle = partition(first,
                                       last,
                                       random_unary_predicate(random));

    random_shuffle(first, middle, random);
    random_shuffle(middle, last,  random);
}
```

Problem: Prove that there is no linear time, in-place random shuffle algorithm for forward and bidirectional iterators.

Predicate-based Permutations.

```
ForwardIterator partition( ForwardIterator first,
                           ForwardIterator last,
                           UnaryPredicate  predicate )
{
      ForwardIterator next = first;
```

```
      while (next != last) {
            if (predicate(*next)) {
                  swap(*first, *next);
                  ++first;
            }
            ++next;
      }

      return first;
}
```

Problem: If first k elements in the range satisfy the predicate, then our algorithm is going to do k unnecessary swaps. Modify the algorithm so that it does not do it.

Solution:

```
ForwardIterator partition( ForwardIterator first,
                           ForwardIterator last,
                           UnaryPredicate  predicate )
{
      first = find_if(first, last, negate(predicate));

      if (first == last) return first;

      ForwardIterator next = successor(first);

      while (next != last) {
            if (predicate(*next)) {
                  swap(*first, *next);
                  ++first;
            }
            ++next;
      }

      return first;
}

BidirectionalIterator partition( BidirectionalIterator first,
                                 BidirectionalIterator last,
                                 UnaryPredicate        predicate )
{
      while(true) {
          if (first == last) return first;
          while (predicate(*first)) {
              ++first;
              if (first == last) return first;
          }

          --last;

          if (first == last) return first;
          while (!predicate(*last)) {
              --last;
              if (first == last) return first;
```

```
            }

            swap(*first, *last);

            ++first;
        }
}
```

Problem: Use a sentinel technique to reduce the number of `first == last` operations for partition on bidirectional iterators. (Intermediate.)

Solution:

```
BidirectionalIterator
partition_unguarded( BidirectionalIterator first,
                     BidirectionalIterator last,
                     UnaryPredicate        predicate )
{
      assert(exists(first, last, predicate) &&
             exists(first, last, negate(predicate)));

      while(true) {
            --last;
            while (predicate(*first)) ++first;
            while (!predicate(*last)) --last;
            if (successor(last) == first) return first;
            swap(*first, *last);
            ++first;
      }
}
```

Problem: What are the assumptions on `predicate` that allow one to create sentinels. Give an example of a predicate that does not satisfy these assumptions.

```
ForwardIterator remove_if( ForwardIterator first,
                           ForwardIterator last,
                           UnaryPredicate  predicate )
{
      first = find_if(first, last, negate(predicate));

      if (first == last) return first;

      ForwardIterator next = successor(first);

      while (next != last) {
            if (predicate(*next)) {
                  *first = *next;
                  ++first;
            }
            ++next;
      }
```

```
      return first;
}

bool is_partitioned( InputIterator first,
                     InputIterator last,
                     UnaryPredicate  predicate )
{
     first = find_if(first, last, negate(predicate));

     return (first == last || find_if(first, last, predicate) == last);
}


ForwardIterator partition_point_n( ForwardIterator first,
                                   Integer         n,
                                   UnaryPredicate  predicate )
{
     while (n > 0) {
          ForwardIterator middle = first;
          advance(middle, n/2);
          if (predicate(*middle)) {
               first = successor(middle);
               n = n - n/2 - 1;
          } else {
               n = n/2;
          }
     }
     return first;
}

ForwardIterator partition_point( ForwardIterator first,
                                 ForwardIterator last,
                                 UnaryPredicate  predicate )
{
     return partition_point_n(first, distance(first, last), predicate);
}

pair(OutputIterator[1], OutputIterator[2])
partition_copy( InputIterator first,
                InputIterator last,
                OutputIterator[1] predicate_true,
                OutputIterator[2] predicate_false,
                UnaryPredicate predicate )
{
     while (first != last) {
            if (predicate(*first)) {
                *predicate_true = *first;
                ++predicate_true;
            } else {
                *predicate_false = *first;
                ++predicate_false;
            }
            ++first;
     }
     return pair(predicate_true, predicate_false);
}
```

```
pair(ForwardIterator, ForwardIterator)
partition_3way( ForwardIterator first,
                ForwardIterator last,
                UnaryFunction   function )
{
      ForwardIterator second = first;
      ForwardIterator third = second;

      while (third != last) {
            switch (function(*third)) {
            case -1:   swap(*first, *third);
                       ++first;
            case  0:   swap(*second, *third);
                       ++second;
            case  1:   ++third;
            }
      }

      return pair(first, second);
}

Pair(ForwardIterator, ForwardIterator)
partition_point_3way_n( ForwardIterator first,
                        Integer         n,
                        UnaryFunction   function )
{
     while (n > 0) {
          ForwardIterator middle = successor(first, n/2);
          switch (function(*middle)) {
          case -1:  first = successor(middle);
                    n = n – n/2 - 1;
                    break;
          case  0:  return pair(
                         partition_point_n(first,
                                           n – n/2 - 1,
                                           equal(function, -1)),
                         partition_point_n(successor(middle),
                                           n/2,
                                           equal(function, 0));
          case  1:  n = n/2;
                    break;
          }

          return make(pair, first, first);
}


Pair(ForwardIterator, ForwardIterator)
partition_point_3way( ForwardIterator first,
                      ForwardIterator last,
                      UnaryFunction   function )
{
     return partition_point_3way_n(first,
                                   distance(first, last),
```

```
                                    function);
}

pair stable_partition_n( ForwardIterator first,
                         Integer n)
{
     if (n == 0) return pair(first, first);
     if (n == 1) {
          ForwardIterator last = successor(first);
          return pair(predicate(*first) ? last : first,
                      last);
     }

     pair left_subproblem  =
          stable_partition_n(first, n/2);
     pair right_subproblem =
          stable_partition_n(second(left_subproblem), n - n/2);

     ForwardIterator cut = rotate(first(left_subproblem),
                                  second(left_subproblem),
                                  first(right_subproblem));

     return pair(cut, second(right_subproblem));
}


ForwardIterator stable_partition( ForwardIterator first,
                                  ForwardIterator last )
{
     return first(stable_partition_n(first,
                                     distance(first, last)));
}
```

Problem: Design an iterative version of forward iterator reverse using binary counter. (Intermidiate)

Problem: Design stable 3-way partition algorithm. (Intermediate)

```
value_type(InputIterator)
reduce( InputIterator             first,
        InputIterator             last,
        BinaryOperation           operation,
        value_type(InputIterator) identity =
             identity_element(operation))
{
     if (first == last) return identity;

     value_type(InputIterator) result = *first;

     while (++first != last) {
          result = operation(result, *first);
     }

     return result;
}
```

```
value_type(InputIterator)
reduce_optimized( InputIterator             first,
                  InputIterator             last,
                  BinaryOperation           operation,
                  value_type(InputIterator) identity =
                      identity_element(operation))
{
     first = find(first, last, identity, negate(equal_to));

     if (first == last) return identity;

     value_type(InputIterator) result = *first;

     while (++first != last) {
          if (*first != identity) {
               result = operation(result, *first);
          }
     }

     return result;
}

void add_bit_to_counter( BackInsertionSequence&            counter,
                         BinaryOperation                   operation,
                         value_type(BackInsertionSequence) carry,
                         value_type(BackInsertionSequence) identity =
                            identity_element(operation))
{
     iterator(BackInsertionSequence) first = begin(counter);
     iterator(BackInsertionSequence) last  = end(counter);
     if (carry == identity) return;
     while (first != last) {
          if (*first != identity) {
               carry = operation(*first, carry);
               *first = identity;
               ++first;
          } else {
               *first = carry;
               return;
          }
     }

     push_back(counter, carry);
}

NEED STATIC_VECTOR<a number> WITH PUSH_BACK TO KEEP COUNTER ON THE
STACK!

value_type(InputIterator)
reduce_with_counter ( InputIterator             first,
                      InputIterator             last,
                      BinaryOperation           operation,
                      value_type(InputIterator) identity =
                           identity_element(operation)))
{
```

```
     static_vector(value_type(InputIterator)) counter(64);
     while (first != last) {
          add_bit_to_counter(counter, operation, *first, identity);
          ++first;
     }
     return reduce_optimized(begin(counter),
                             end(counter),
                             transpose(operation),
                             identity);
}
```

Discussion of why reduce with counter is faster than reduce

```
pair(ForwardIterator, ForwardIterator)
stable_partition_operation( pair(ForwardIterator, ForwardIterator) x,
                            pair(ForwardIterator, ForwardIterator) y )
{
     return pair(rotate(x.first, x.second, y.first), y.second);
}

pair(ForwardIterator, ForwardIterator)
stable_partition_transform ( UnaryPredicate predicate )
        ( ForwardIterator first,
          ForwardIterator last )
{
     last = successor(first);
     if (predicate(*first)) ++first;
     return pair(first, last);
}


pair(ForwardIterator, ForwardIterator)
counter_algorithm_driver( ForwardIterator first,
                          ForwardIterator last,
                          Generator       generator,
                          BinaryOperation operation) {
     static_vector(pair(ForwardIterator, ForwardIterator)) counter(64);
     while (first != last) {
          pair(ForwardIterator, ForwardIterator) step =
               generator(first, last);
          add_bit_to_counter(counter,
                             operation,
                             step,
                             pair(last, last));
          first = step.second;
     }
     return reduce(begin(counter),
                   end(counter),
                   transpose(operation),
                   pair(last, last));

}

ForwardIterator stable_partition( ForwardIterator first,
                                  ForwardIterator last,
```

```
                                   UnaryPredicate  predicate )
{

     return counter_algorithm_driver(
          first,
          last,
          stable_partition_operation,
          stable_partition_transform(predicate)).first;
}


pair(ForwardIterator[1], ForwardIterator[1])
step_partition_with_buffer( ForwardIterator[2] buffer_first,
                            ForwardIterator[2] buffer_last,
                            UnaryPredicate predicate )
          ( ForwardIterator[1] first,
            ForwardIterator[1] last )
{
     assert(buffer_first != buffer_last);
     ForwardIterator[1] next = first;
     ForwardIterator[1] buffer_next = buffer_first;
     while (next != last) {
            if (predicate(*next)) {
                *first = *next;
                ++first;
            } else {
                *buffer_next = *next;
                ++buffer_next;
                if (buffer_next == buffer_last) break;
            }
            ++next;
     }
     last = copy(buffer_first, buffer_next, first);

     return pair(first, last);
}

ForwardIterator stable_partition_with_buffer(
                  ForwardIterator[1] first,
                  ForwardIterator[1] last,
                  ForwardIterator[2] buffer_first,
                  ForwardIterator[2] buffer_last,
                  UnaryPredicate     predicate )
{

     return counter_algorithm_driver(
          first,
          last,
          stable_partition_operation,
          step_partition_with_buffer(predicate,
                                     buffer_first,
                                     buffer_last)).first;
}

ForwardIterator lower_bound( ForwardIterator first,
                             ForwardIterator last,
```

```
                            const T& value,
                            StrictWeakOrdering compare = less )

{
    return partition_point(first, last, bind2nd(compare, value));
}

ForwardIterator upper_bound( ForwardIterator    first,
                            ForwardIterator    last,
                            const T&           value,
                            StrictWeakOrdering compare = less )
{
    return partition_point(first,
                           last,
                           negate(bind1st(compare, value)));
}


int comparator( StrictWeakOrdering compare,
                const T&           value )
     (const T& x)
{
    if (compare(value, x)) return -1;
    if (compare(x, value)) return 1;
    else                   return 0;
}


pair(ForwardIterator, ForwardIterator)
equal_range( ForwardIterator    first,
             ForwardIterator    last,
             const T&           value,
             StrictWeakOrdering compare = less )
{
    return partition_point_3way(first,
                                last,
                                comparator(value, compare));
}


ForwardIterator min_element( ForwardIterator first,
                            ForwardIterator last )
{
    if (first == last) return first;

    ForwardIterator minimum = first;

    while (++first != last) {
        if (comp(*first, *result)) result = first;
    }

    return minimum;
}
```

```
BidirectionalIterator
unguarded_linear_insert( BidirectionalIterator current )
{
     BidirectionalIterator previous = predecessor(current);
     Value value = *current;
     while (compare(value, *current)) {
          *current = *previous;
          current = previous;
          --previous;
     }
     *current = value;
     return current;
}

void unguarded_insertion_sort( BidirectionalIterator first,
                               BidirectionalIterator last ) {
     assert(first != last && successor(first) != last);
     ++first;
     do {
          unguarded_linear_insert(first);
          ++first;
     } while (first != last);
}


void insertion_sort( BidirectionalIterator first,
                     BidirectionalIterator last ) {
     if (first == last && successor(first) == last) return;

     BidirectionalIterator minimum = min_element(first, last);
     Value value = *minimum;
     copy_backward(first, minimum, successor(minimum));
     *first = value;

     ++first;
     unguarded_insertion_sort(first, last);
}


OutputIterator merge_non_empty(ForwardIterator[1] first1,
                               ForwardIterator[1] last1,
                               ForwardIterator[2] first2,
                               ForwardIterator[2] last2,
                               OutputIterator     result,
                               StrictWeakOrdering compare)
{
     while (true) {
          if (compare(*first2, *first1)) {
               *result = *first2;
               ++first2;
               ++result;
               if (first2 == last2)
                    return copy(first1, last1, result);
          } else {
               *result = *first1;
               ++first1;
               ++result;
```

```
               if (first1 == last1)
                    return copy(first2, last2, result);
     }
}

OutputIterator merge(ForwardIterator[1] first1,
                     ForwardIterator[1] last1,
                     ForwardIterator[2] first2,
                     ForwardIterator[2] last2,
                     OutputIterator     result,
                     StrictWeakOrdering compare)
{
     if (first1 == last1) return copy(first2, last2, result);
     if (first2 == last2) return copy(first1, last1, result);

     return merge_non_empty(first1,
                            last1,
                            first2,
                            last2,
                            result,
                            compare);
}
```

Problem: Implement algorithm for set_union. (Easy)

Problem: Implement algorithm for set_intersection. (Easy)

Problem: Implement algorithm for set_difference. (Easy)

Problem: Implement algorithm for set_symmetric_difference. (Easy)

```
void inplace_merge(RandomAccessIterator first,
                   RandomAccessIterator middle,
                   RandomAccessIterator last,
                   StrictWeakOrdering    compare)
{
     if (first == middle || middle == last) return;
     if (last – first == 2) {
          if (compare(*middle, *first)) swap(*first, *middle);
          return;
     }

     RandomAccessIterator first_cut  = first + (middle – first)/2;
     RandomAccessIterator second_cut = middle + (last – middle)/2;

     if (middle - first > last - middle) {
          second_cut = lower_bound(middle, last, *first_cut);
     } else {
          first_cut  = upper_bound(first, middle, *second_cut);
     }
```

```
     middle = rotate(first_cut, middle, second_cut);

     inplace_merge(first, first_cut, middle, compare);
     inplace_merge(middle, second_cut, last, compare);

     return;
}
```

Problem: Implement inplace_merge for forward iterators. (Intermediate)

```
homogeneous_pair(homogeneous_triple(RandomAccessIterator))
inplace_merge_divide_operation( StrictWeakOrdering compare )
     ( homogeneous_triple(RandomAccessIterator) merge_triple )
{
     RandomAccessIterator first  = first(merge_triple);
     RandomAccessIterator middle = second(merge_triple);
     RandomAccessIterator last   = third(merge_triple);

     RandomAccessIterator first_cut = first + (middle – first)/2;
     RandomAccessIterator second_cut = middle + (last – middle)/2;

     if (middle - first > last - middle) {
          second_cut = lower_bound(middle, last, *first_cut);
     } else {
          first_cut  = upper_bound(first, middle, *second_cut);
     }

     middle = rotate(first_cut, middle, second_cut);

     return pair(triple(first, first_cut, middle),
                 triple(middle, second_cut, last));
}

void top_down_divide_and_conquer(Value problem)
{
     static_vector(Value) stack(64);

     while (true) {
          if (size(problem) >= threshold) {
              pair(Value, Value) subproblems =
                    divide_operation(first, last);
              if (size(second(subproblems)) <
                  size(first(subproblems))) {
                   push_back(stack, first(subproblems));
                   problem = second(subproblems);
              } else {
                   push_back(stack, second(subproblems));
                   problem = first(subproblems);
              }
          } else {
              leaf_solution(problem);
              if (is_empty(stack)) break;
              problem = back(stack);
              pop_back(stack);
          }
     }
```

```
}

NodeIterator reverse_append( NodeIterator first,
                             NodeIterator last,
                             NodeIterator result )
{
     while (first != last) {
          NodeIterator tmp = first;
          ++first;
          set_next(tmp, result);
          result = tmp;
     }

     return result;
}

NodeIterator reverse( NodeIterator first,
                      NodeIterator last )
{
     return reverse_append(first, last, last);
}


NodeIterator merge_non_empty( NodeIterator first1,
                              NodeIterator last1,
                              NodeIterator first2,
                              NodeIterator last2 )
{
     NodeIterator result = first1;
     NodeIterator tail   = first2;

     if (compare(*first2, *first1)) {
          result = first2;
          ++first2;
     } else {
          tail = first1;
          ++first1;
     }

     while (first2 != last2 && first1 != last1) {
          if (compare(*first2, *first1)) {
              set_next(tail, first2);
              ++first2;
          } else {
              set_next(tail, first1);
              ++first1;
          }
          ++tail;
     }

     if (first1 == last1)
          set_next(tail, first1);
     else
          set_next(tail, first2);

     return result;
```

```
}


NodeIterator merge_non_empty( NodeIterator first1,
                              NodeIterator last1,
                              NodeIterator first2,
                              NodeIterator last2 )
{
     NodeIterator result = first1;
     NodeIterator tail   = first2;


     if (compare(*first2, *first1)) {
          result = first2;
          ++first2;
          if (first2 == last2) goto exit1;
     } else {
          tail = first1;
          ++first1;
          if (first1 == last1) goto exit2;
     }

     while (true) {
          if (compare(*first2, *first1)) {
              set_next(tail, first2);
              ++tail;
              ++first2;
              if (first2 == last2) goto exit1;
          } else {
              set_next(tail, first1);
              ++tail;
              ++first1;
              if (first1 == last1) goto exit2;
          }
     }
exit1:
     set_next(tail, first1);
     return result;
exit2:
     set_next(tail, first2);
     return result;
}

NodeIterator merge_non_empty( NodeIterator first1,
                              NodeIterator last1,
                              NodeIterator first2,
                              NodeIterator last2 )
{
     NodeIterator result = first1;
     NodeIterator tail   = first2;
     bool second_won;

     if (compare(*first2, *first1)) {
          result = first2;
          second_won = true;
          ++first2;
```

```
          if (first2 == last2) goto exit1;
     } else {
          tail = first1;
          second_won = false;
          ++first1;
          if (first1 == last1) goto exit2;
     }

     while (true) {
          if (compare(*first2, *first1)) {
              if (!second_won) {
                   set_next(tail, first2);
                   second_won = true;
              }
              ++tail;
              ++first2;
              if (first2 == last2) goto exit1;
          } else {
              if (second_won) {
                   set_next(tail, first1);
                   second_won = false;
              }
              ++tail;
              ++first1;
              if (first1 == last1) goto exit2;
          }
     }
exit1:
     set_next(tail, first1);
     return result;
exit2:
     set_next(tail, first2);
     return result;

}

NodeIterator merge_non_empty( NodeIterator first1,
                              NodeIterator last1,
                              NodeIterator first2,
                              NodeIterator last2 )
{
     NodeIterator result = first1;
     NodeIterator tail   = first2;

     if (!compare(*first2, *first1)) {
          tail = first1;
          goto first_won;
     }
     result = first2;
second_won:
     ++first2;
     if (first2 == last2) {
          set_next(tail, first1);
          return result;
     }
     if (compare(*first2, *first1)) {
          ++tail;
```

```
          goto second_won;
     }
     set_next(tail, first1);
     ++tail;
first_won:
     ++first1;
     if (first1 == last1) {
          set_next(tail, first2);
          return result;
     }
     if (compare(*first2, *first1)) {
          set_next(tail, first2);
          ++tail;
          goto second_won;
     }
     ++tail;
     goto first_won;
}

NodeIterator mergesort_operation( NodeIterator last )
     ( NodeIterator first1,
       NodeIterator first2 )
{
     return merge_non_empty(first1, last, first2, last);
}

NodeIterator merge_sort( NodeIterator first,
                         NodeIterator last )
{
     static_vector(NodeIterator) counter(64);
     while (first != last) {
          NodeIterator next = successor(first);
          set_next(first, last)
          add_bit_to_counter(counter,
                             mergesort_operation(last),
                             first,
                             last);
          first = next;
     }
     return reduce(begin(counter),
                   end(counter),
                   transpose(mergesort_operation(last)),
                   last);

}

pair reverse_partition( NodeIterator first,
                        NodeIterator last,
                        NodeIterator result1,
                        NodeIterator result2 )
{
     while (first != last) {
          NodeIterator next = successor(first);
          if (predicate(*first)) {
               set_next(first, result1);
               result1 = first;
          } else {
```

```
                set_next(first, result2);
                result2 = first;
           }
           first = next;
      }

      return pair(result1, result2);
}

NodeIterator remove_nodes( NodeIterator first,
                           NodeIterator last )
{
      NodeIterator result = find_if(first, last, negate(predicate));

      free_nodes(first, next);

      if (result == last) return result;

      first = result;

      NodeIterator next = successor(first);

      while (next != last) {
           NodeIterator after_next = successor(next);
           if (predicate(*next)) {
              set_next(first, after_next);
              free_node(next);
           } else {
              first = next;
           }
           next = after_next;
      }

      return result;
}


iterator(VectorStructure)
insert(VectorStructure vector_structure&,
       iterator(VectorStructure) position,
       const value_type(VectorStructure)& value) {
      DifferenceType index = position – first(vector_structure);
      if (last(vector_structure) == final(vector_structure)) {
           reallocate(vector_structure);
      }
      iterator(VectorStructure) my_last = last(vector_structure);
      position = first(vector_structure) + index;
           ++last(vector_structure);
      move_backward(first(vector_structure), my_last, my_last+1);
      construct(*my_last, value);
      return position+1;
}
```