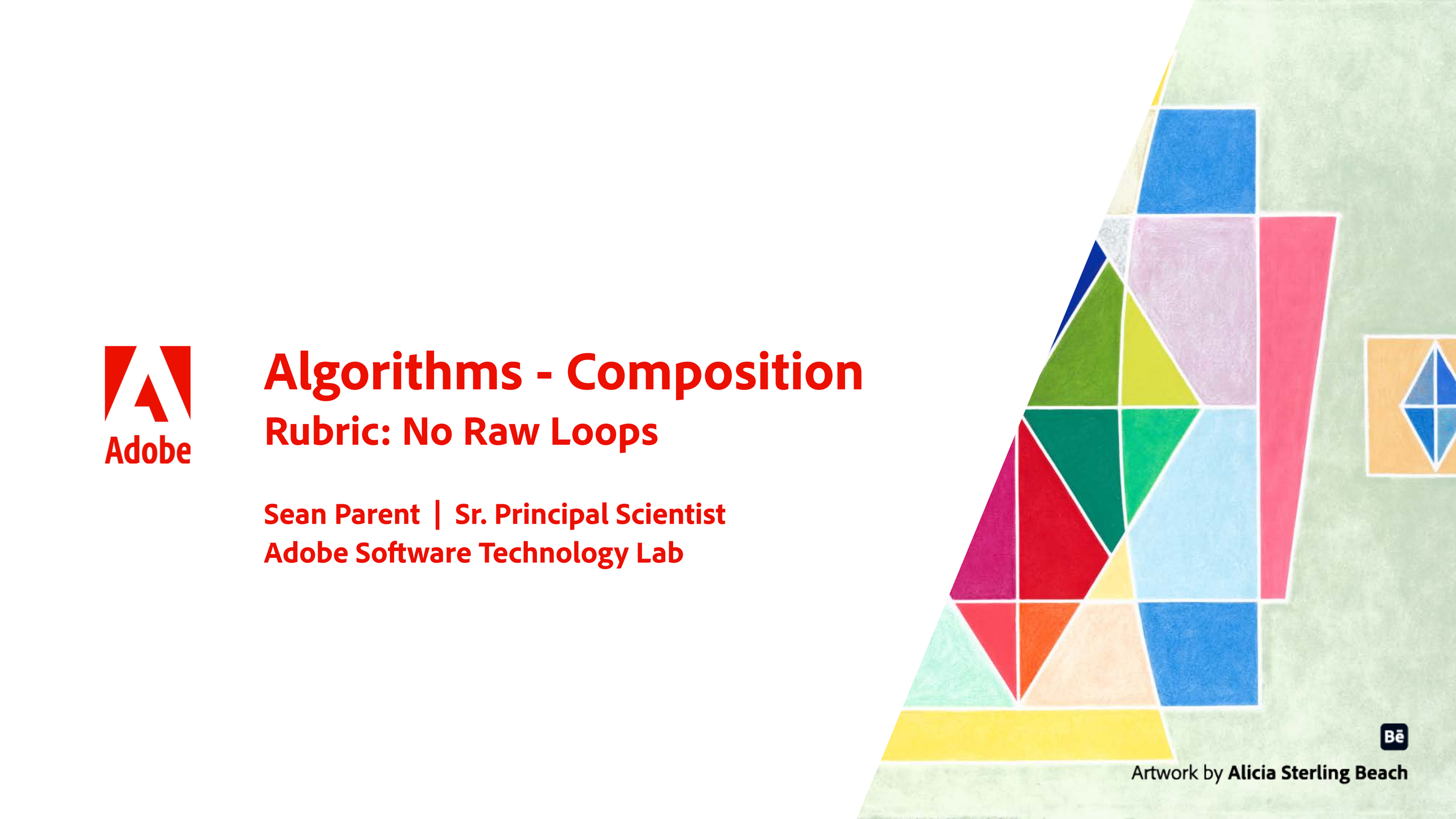


Algorithms - Composition

Rubric: No Raw Loops

Sean Parent | Sr. Principal Scientist
Adobe Software Technology Lab

Artwork by **Alicia Sterling Beach**

Preliminaries

Programming is the construction of algorithms

Factor algorithms into functions with meaningful names

- With a specification

Half-open intervals, `[f, l)`, simplify reasoning about sequences and avoid off by one errors

- Think of pointers and iterators as positions *between* elements

Not in this Talk

Combinators and Combinatory Logic

- See *To Mock a Mockingbird* by Raymond Smullyan
- Example: the identity combinator *(I x)* is equivalent to *((S K K) x)*

Category Theory

- See *Category Theory for Progammers* by Bartosz Milewski
- Example: A monoid

What is a Raw Loop?

A *raw* loop is any loop where the purpose of the loop is not clearly defined by the enclosing operation

What is a Raw Loop?

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
  CHECK(fixed_panel);

  // First, find the index of the fixed panel.
  int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
  CHECK_LT(fixed_index, expanded_panels_.size());

  // Next, check if the panel has moved to the other side of another panel.
  const int center_x = fixed_panel->cur_panel_center();
  for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (center_x <= panel->cur_panel_center() ||
        i == expanded_panels_.size() - 1) {
      if (panel != fixed_panel) {
        // If it has, then we reorder the panels.
        ref_ptr<Panel> ref = expanded_panels_[fixed_index];
        expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
        if (i < expanded_panels_.size()) {
          expanded_panels_.insert(expanded_panels_.begin() + i, ref);
        } else {
          expanded_panels_.push_back(ref);
        }
      }
      break;
    }
  }

  // Find the total width of the panels to the left of the fixed panel.
  int total_width = 0;
  fixed_index = -1;
```

What is a Raw Loop?

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
  CHECK(fixed_panel);

  // First, find the index of the fixed panel.
  int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
  CHECK_LT(fixed_index, expanded_panels_.size());

  // Next, check if the panel has moved to the other side of another panel.
  const int center_x = fixed_panel->cur_panel_center();
  for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (center_x <= panel->cur_panel_center() ||
        i == expanded_panels_.size() - 1) {
      if (panel != fixed_panel) {
        // If it has, then we reorder the panels.
        ref_ptr<Panel> ref = expanded_panels_[fixed_index];
        expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
        if (i < expanded_panels_.size()) {
          expanded_panels_.insert(expanded_panels_.begin() + i, ref);
        } else {
          expanded_panels_.push_back(ref);
        }
      }
      break;
    }
  }

  // Find the total width of the panels to the left of the fixed panel.
  int total_width = 0;
  fixed_index = -1;
```

What is a Raw Loop?

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
  CHECK(fixed_panel);

  // First, find the index of the fixed panel.
  int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
  CHECK_LT(fixed_index, expanded_panels_.size());

  // Next, check if the panel has moved to the other side of another panel.
  const int center_x = fixed_panel->cur_panel_center();
  for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (center_x <= panel->cur_panel_center() ||
        i == expanded_panels_.size() - 1) {
      if (panel != fixed_panel) {
        // If it has, then we reorder the panels.
        ref_ptr<Panel> ref = expanded_panels_[fixed_index];
        expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
        if (i < expanded_panels_.size()) {
          expanded_panels_.insert(expanded_panels_.begin() + i, ref);
        } else {
          expanded_panels_.push_back(ref);
        }
      }
      break;
    }
  }

  // Find the total width of the panels to the left of the fixed panel.
  int total_width = 0;
  fixed_index = -1;
```

What is a Raw Loop?

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
  CHECK(fixed_panel);

  // First, find the index of the fixed panel.
  int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
  CHECK_LT(fixed_index, expanded_panels_.size());

  // Next, check if the panel has moved to the other side of another panel.
  const int center_x = fixed_panel->cur_panel_center();
  for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (center_x <= panel->cur_panel_center() ||
        i == expanded_panels_.size() - 1) {
      if (panel != fixed_panel) {
        // If it has, then we reorder the panels.
        ref_ptr<Panel> ref = expanded_panels_[fixed_index];
        expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
        if (i < expanded_panels_.size()) {
          expanded_panels_.insert(expanded_panels_.begin() + i, ref);
        } else {
          expanded_panels_.push_back(ref);
        }
      }
      break;
    }
  }

  // Find the total width of the panels to the left of the fixed panel.
  int total_width = 0;
  fixed_index = -1;
```

What is a Raw Loop?

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
  CHECK(fixed_panel);

  // First, find the index of the fixed panel.
  int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
  CHECK_LT(fixed_index, expanded_panels_.size());

  // Next, check if the panel has moved to the other side of another panel.
  const int center_x = fixed_panel->cur_panel_center();
  for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (center_x <= panel->cur_panel_center() ||
        i == expanded_panels_.size() - 1) {
      if (panel != fixed_panel) {
        // If it has, then we reorder the panels.
        ref_ptr<Panel> ref = expanded_panels_[fixed_index];
        expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
        if (i < expanded_panels_.size()) {
          expanded_panels_.insert(expanded_panels_.begin() + i, ref);
        } else {
          expanded_panels_.push_back(ref);
        }
      }
      break;
    }
  }

  // Find the total width of the panels to the left of the fixed panel.
  int total_width = 0;
  fixed_index = -1;
```

What is a Raw Loop?

```
        expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
        if (i < expanded_panels_.size()) {
          expanded_panels_.insert(expanded_panels_.begin() + i, ref);
        } else {
          expanded_panels_.push_back(ref);
        }
      }
      break;
    }
  }

  // Find the total width of the panels to the left of the fixed panel.
  int total_width = 0;
  fixed_index = -1;
  for (int i = 0; i < static_cast<int>(expanded_panels_.size()); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (panel == fixed_panel) {
      fixed_index = i;
      break;
    }
    total_width += panel->panel_width();
  }
  CHECK_NE(fixed_index, -1);
  int new_fixed_index = fixed_index;

  // Move panels over to the right of the fixed panel until all of the ones
  // on the left will fit.
  int avail_width = max(fixed_panel->cur_panel_left() - kBarPadding, 0);
  while (total_width > avail_width) {
    new_fixed_index--;
    CHECK_GE(new_fixed_index, 0);
    total_width -= expanded_panels_[new_fixed_index]->panel_width();
  }
```

What is a Raw Loop?

```
  for (int i = 0; i < static_cast<int>(expanded_panels_.size()); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (panel == fixed_panel) {
      fixed_index = i;
      break;
    }
    total_width += panel->panel_width();
  }
  CHECK_NE(fixed_index, -1);
  int new_fixed_index = fixed_index;

  // Move panels over to the right of the fixed panel until all of the ones
  // on the left will fit.
  int avail_width = max(fixed_panel->cur_panel_left() - kBarPadding, 0);
  while (total_width > avail_width) {
    new_fixed_index--;
    CHECK_GE(new_fixed_index, 0);
    total_width -= expanded_panels_[new_fixed_index]->panel_width();
  }

  // Reorder the fixed panel if its index changed.
  if (new_fixed_index != fixed_index) {
    Panels::iterator it = expanded_panels_.begin() + fixed_index;
    ref_ptr<Panel> ref = *it;
    expanded_panels_.erase(it);
    expanded_panels_.insert(expanded_panels_.begin() + new_fixed_index, ref);
    fixed_index = new_fixed_index;
  }

  // Now find the width of the panels to the right, and move them to the
  // left as needed.
  total_width = 0;
  for (Panels::iterator it = expanded_panels_.begin() + fixed_index + 1;
       it != expanded_panels_.end(); ++it) {
```

What is a Raw Loop?

```
  total_width -= expanded_panels_[new_fixed_index]->panel_width();
}

// Reorder the fixed panel if its index changed.
if (new_fixed_index != fixed_index) {
  Panels::iterator it = expanded_panels_.begin() + fixed_index;
  ref_ptr<Panel> ref = *it;
  expanded_panels_.erase(it);
  expanded_panels_.insert(expanded_panels_.begin() + new_fixed_index, ref);
  fixed_index = new_fixed_index;
}

// Now find the width of the panels to the right, and move them to the
// left as needed.
total_width = 0;
for (Panels::iterator it = expanded_panels_.begin() + fixed_index + 1;
     it != expanded_panels_.end(); ++it) {
  total_width += (*it)->panel_width();
}

avail_width = max(wm_->width() - (fixed_panel->cur_right() + kBarPadding),
                  0);
while (total_width > avail_width) {
  new_fixed_index++;
  CHECK_LT(new_fixed_index, expanded_panels_.size());
  total_width -= expanded_panels_[new_fixed_index]->panel_width();
}

// Do the reordering again.
if (new_fixed_index != fixed_index) {
  Panels::iterator it = expanded_panels_.begin() + fixed_index;
  ref_ptr<Panel> ref = *it;
  expanded_panels_.erase(it);
  expanded_panels_.insert(expanded_panels_.begin() + new_fixed_index, ref);
```

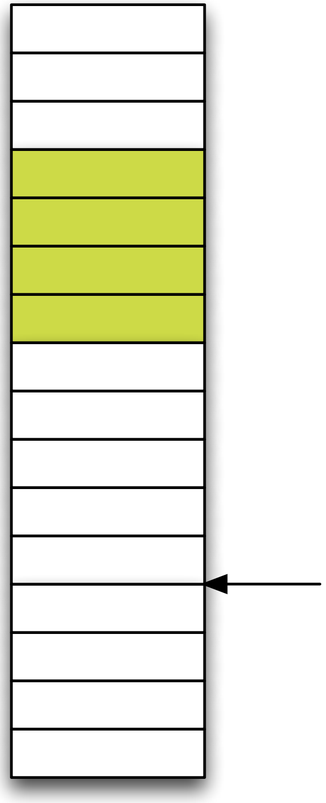
What is a Raw Loop?

```
                0);
  while (total_width > avail_width) {
    new_fixed_index++;
    CHECK_LT(new_fixed_index, expanded_panels_.size());
    total_width -= expanded_panels_[new_fixed_index]->panel_width();
  }

  // Do the reordering again.
  if (new_fixed_index != fixed_index) {
    Panels::iterator it = expanded_panels_.begin() + fixed_index;
    ref_ptr<Panel> ref = *it;
    expanded_panels_.erase(it);
    expanded_panels_.insert(expanded_panels_.begin() + new_fixed_index, ref);
    fixed_index = new_fixed_index;
  }

  // Finally, push panels to the left and the right so they don't overlap.
  int boundary = expanded_panels_[fixed_index]->cur_panel_left() - kBarPadding;
  for (Panels::reverse_iterator it =
         // Start at the panel to the left of 'new_fixed_index'.
         expanded_panels_.rbegin() +
         (expanded_panels_.size() - new_fixed_index);
       it != expanded_panels_.rend(); ++it) {
    Panel* panel = it->get();
    if (panel->cur_right() > boundary) {
      panel->Move(boundary, kAnimMs);
    } else if (panel->cur_panel_left() < 0) {
      panel->Move(min(boundary, panel->panel_width() + kBarPadding), kAnimMs);
    }
    boundary = panel->cur_panel_left() - kBarPadding;
  }

  boundary = expanded_panels_[fixed_index]->cur_right() + kBarPadding;
  for (Panels::iterator it = expanded_panels_.begin() + new_fixed_index + 1;
```

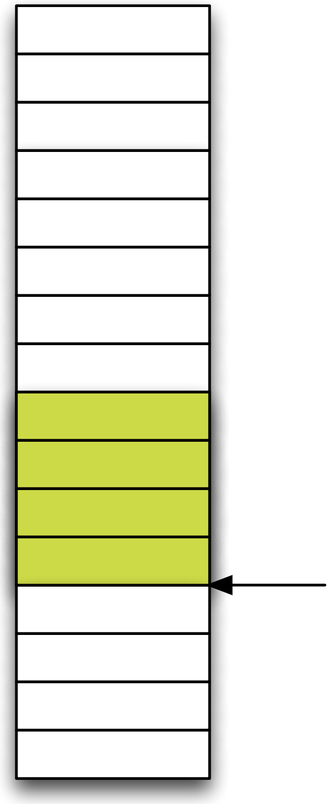
What is a Raw Loop?

```
  int boundary = expanded_panels_[fixed_index]->cur_panel_left() - kBarPadding;
  for (Panels::reverse_iterator it =
           // Start at the panel to the left of 'new_fixed_index'.
           expanded_panels_.rbegin() +
           (expanded_panels_.size() - new_fixed_index);
       it != expanded_panels_.rend(); ++it) {
    Panel* panel = it->get();
    if (panel->cur_right() > boundary) {
      panel->Move(boundary, kAnimMs);
    } else if (panel->cur_panel_left() < 0) {
      panel->Move(min(boundary, panel->panel_width() + kBarPadding), kAnimMs);
    }
    boundary = panel->cur_panel_left() - kBarPadding;
  }

  boundary = expanded_panels_[fixed_index]->cur_right() + kBarPadding;
  for (Panels::iterator it = expanded_panels_.begin() + new_fixed_index + 1;
       it != expanded_panels_.end(); ++it) {
    Panel* panel = it->get();
    if (panel->cur_panel_left() < boundary) {
      panel->Move(boundary + panel->panel_width(), kAnimMs);
    } else if (panel->cur_right() > wm_->width()) {
      panel->Move(max(boundary + panel->panel_width(),
                      wm_->width() - kBarPadding),
                  kAnimMs);
    }
    boundary = panel->cur_right() + kBarPadding;
  }
}
```

Slide

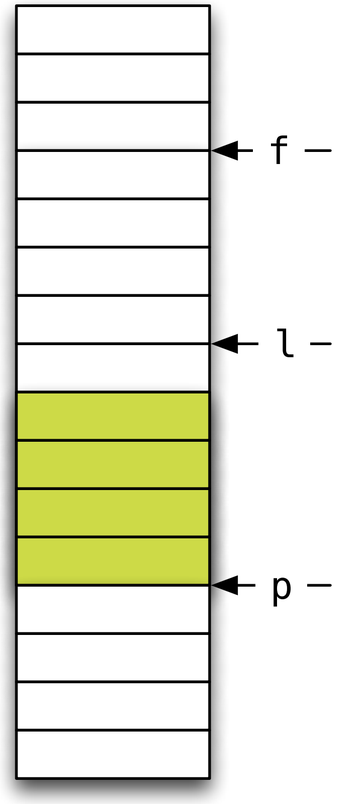


Slide

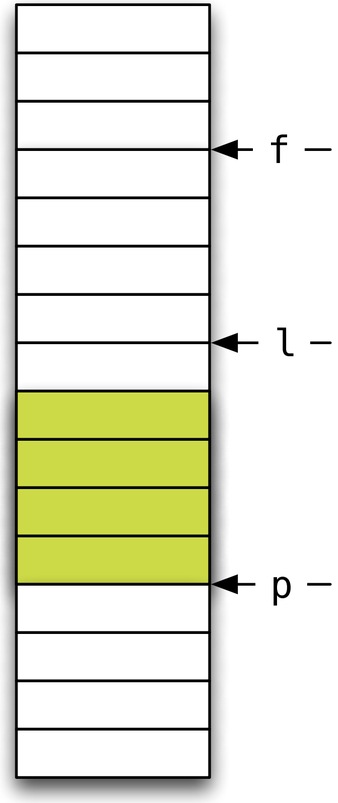


Slide

Slide

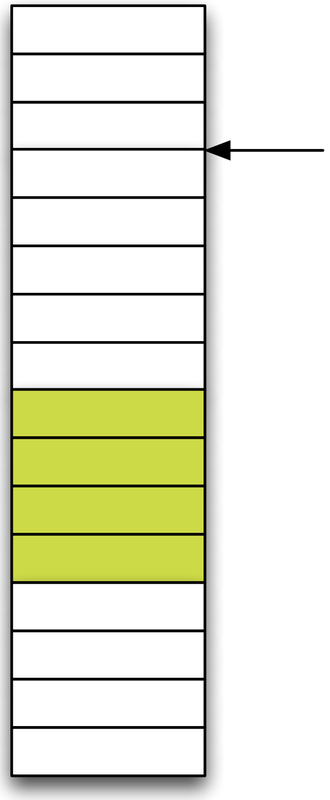


Slide



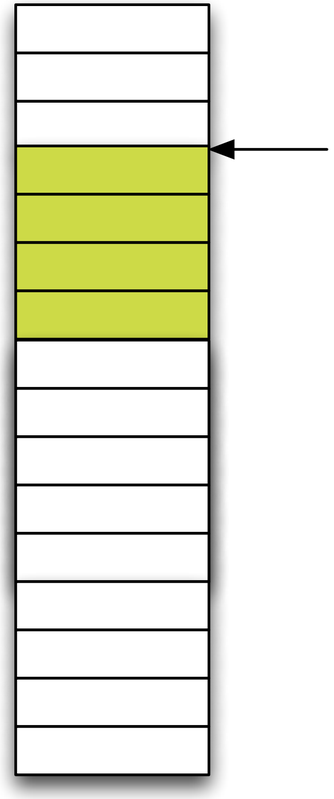

```
rotate(f, l, p);
```

Slide



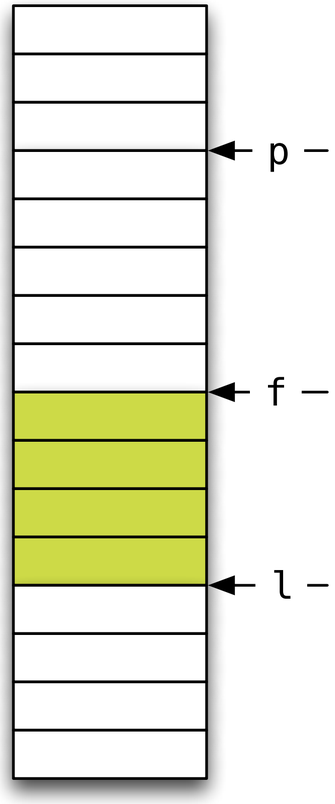
```
rotate(f, l, p);
```

Slide



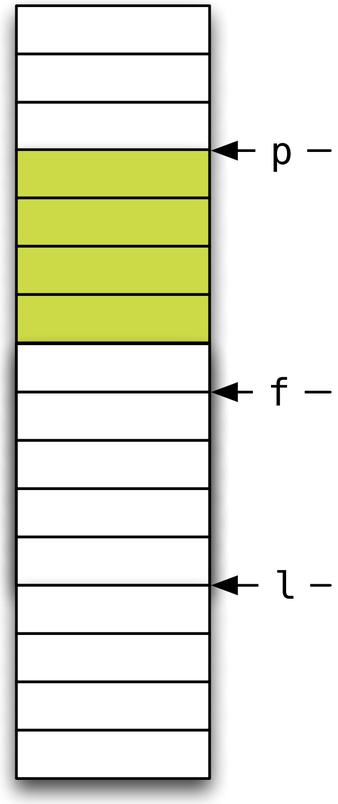
```
rotate(f, l, p);
```

Slide



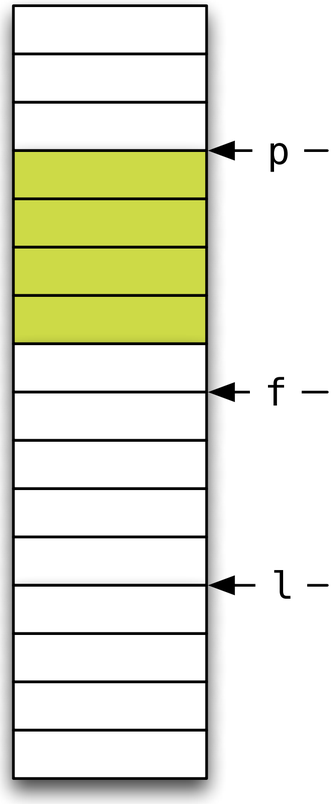

```
rotate(p, f, l);
rotate(f, l, p);
```

Slide



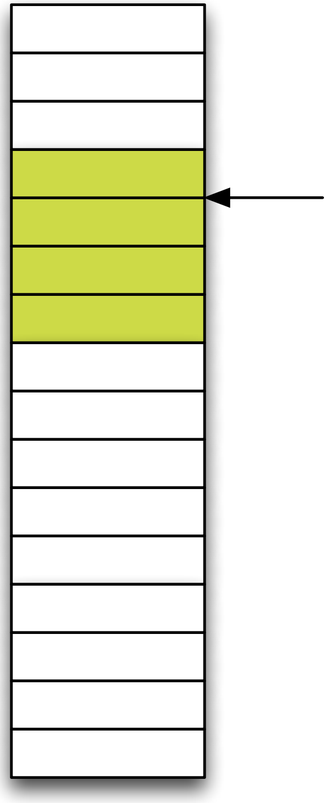

```
rotate(p, f, l);
rotate(f, l, p);
```

Slide



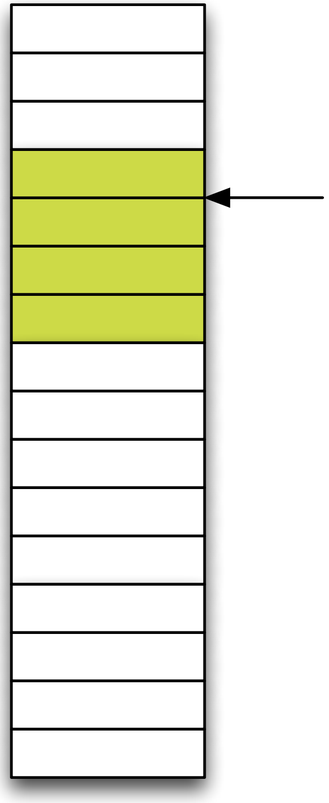

```
if (p < f) rotate(p, f, l);
if (l < p) rotate(f, l, p);
```

Slide



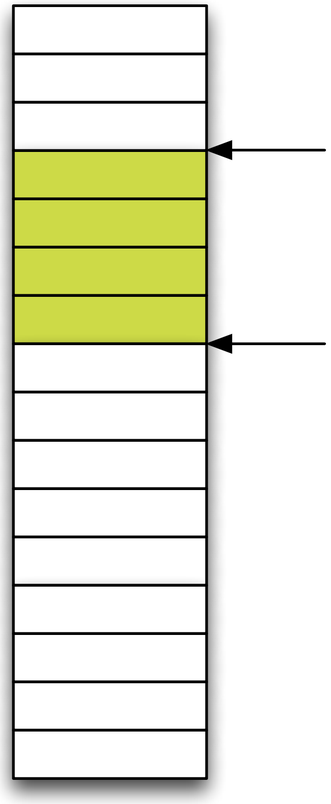
```
if (p < f) rotate(p, f, l);
if (l < p) rotate(f, l, p);
```

Slide



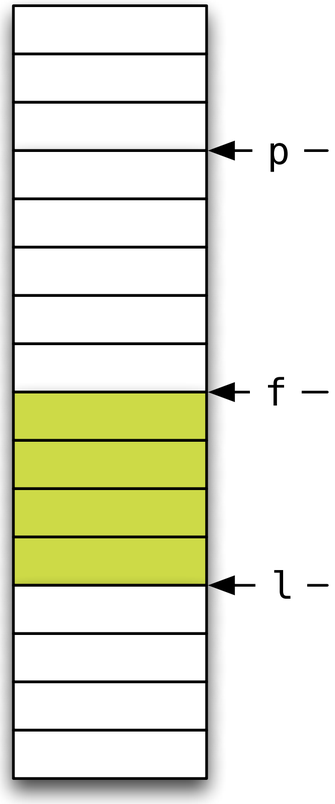
```
if (p < f) rotate(p, f, l);
if (l < p) rotate(f, l, p);
```

Slide



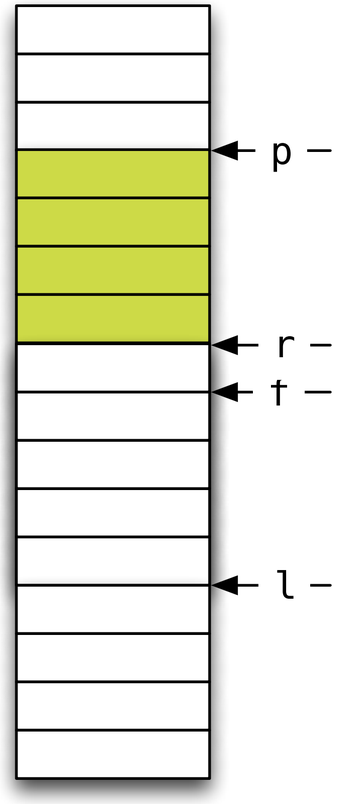
```
if (p < f) rotate(p, f, l);
if (l < p) rotate(f, l, p);
```

Slide



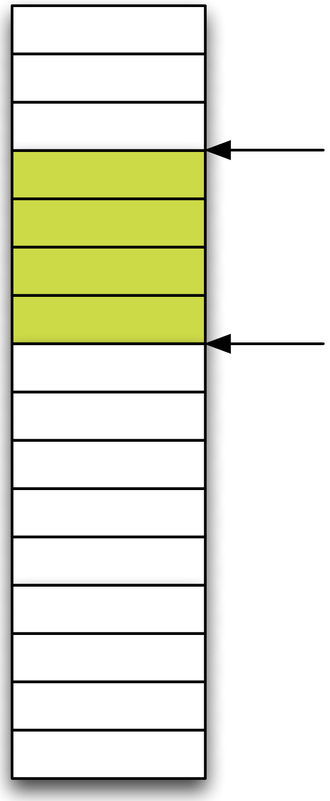

```
if (p < f) return { p, rotate(p, f, l) };
if (l < p) return { rotate(f, l, p), p };
```

Slide



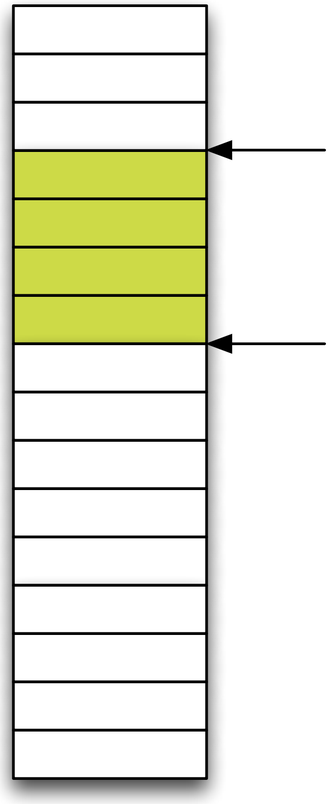

```
if (p < f) return { p, rotate(p, f, l) };
if (l < p) return { rotate(f, l, p), p };
```

Slide



```
if (p < f) return { p, rotate(p, f, l) };
if (l < p) return { rotate(f, l, p), p };
return { f, l };
```

Slide

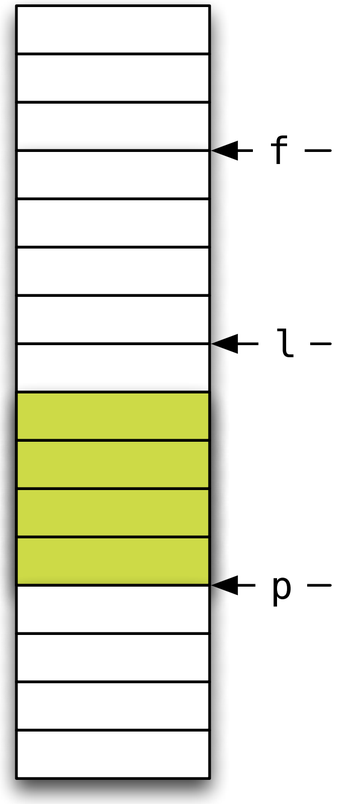


```
// Rotate the elements in [f, l) to p and return
// the new range for the elements
// - Requires: p is reachable from f

template <class I> // I models RandomAccessIterator
auto slide(I f, I l, I p) -> pair<I, I>
{
    if (p < f) return { p, rotate(p, f, l) };
    if (l < p) return { rotate(f, l, p), p };
    return { f, l };
}
```

Slide

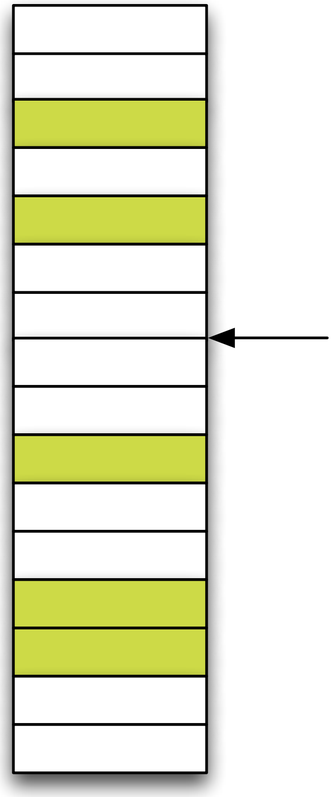

```
// Rotate the elements in [f, l) to p and return
// the new range for the elements
// - Requires: p is reachable from f

template <class I> // I models RandomAccessIterator
auto slide(I f, I l, I p) -> pair<I, I>
{
    if (p < f) return { p, rotate(p, f, l) };
    if (l < p) return { rotate(f, l, p), p };
    return { f, l };
}
```

Slide

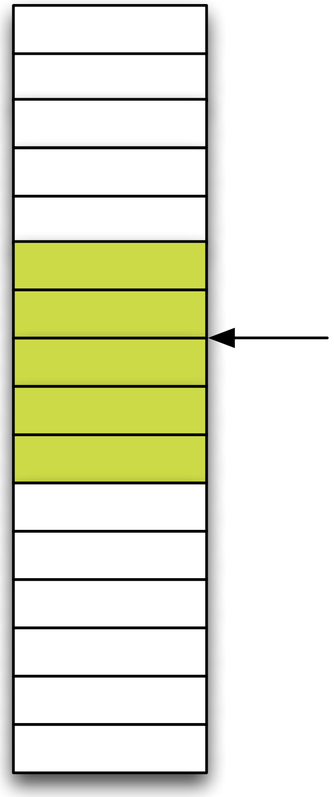



```
// Rotate the elements in [f, l) to p and return
// the new range for the elements
// - Requires: p is reachable from f

template <class I> // I models RandomAccessIterator
auto slide(I f, I l, I p) -> pair<I, I>
{
    if (p < f) return { p, rotate(p, f, l) };
    if (l < p) return { rotate(f, l, p), p };
    return { f, l };
}
```

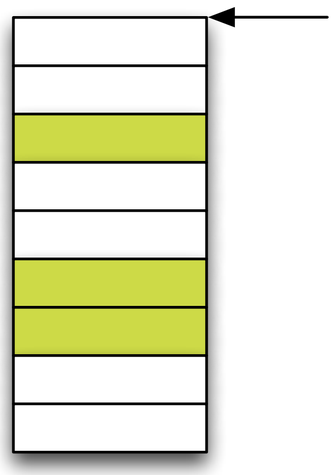
Gather



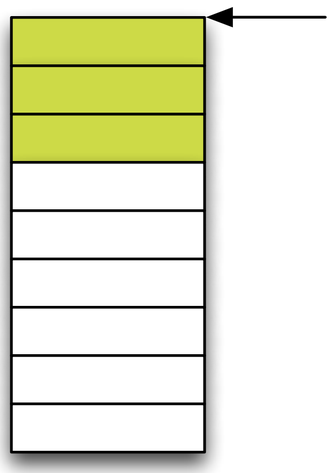
Gather



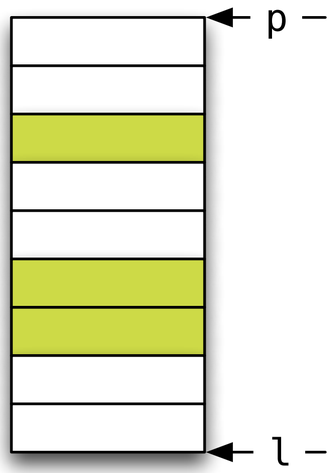
Composing Algorithms - Gather



Composing Algorithms - Gather

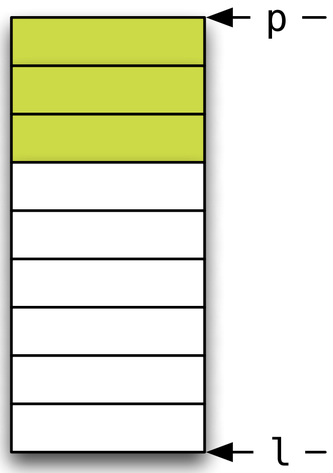


Composing Algorithms - Gather



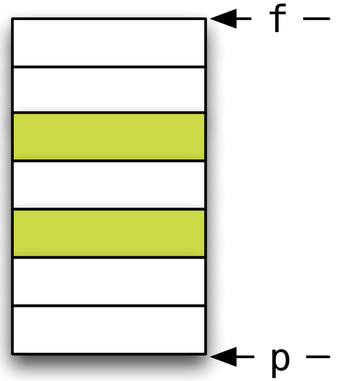

```
stable_partition(p, l, s)
```

Composing Algorithms - Gather



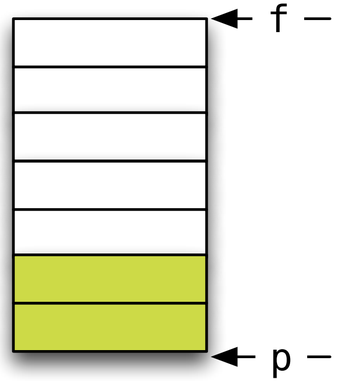

```
stable_partition(p, l, s)
```

Composing Algorithms - Gather



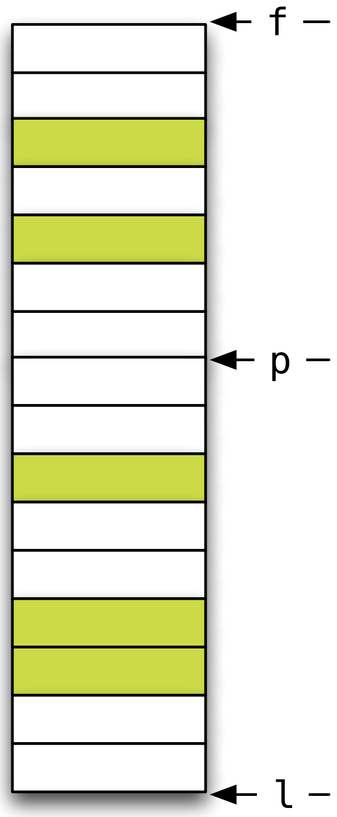

```
stable_partition(f, p, not_fn(s))
```

Composing Algorithms - Gather



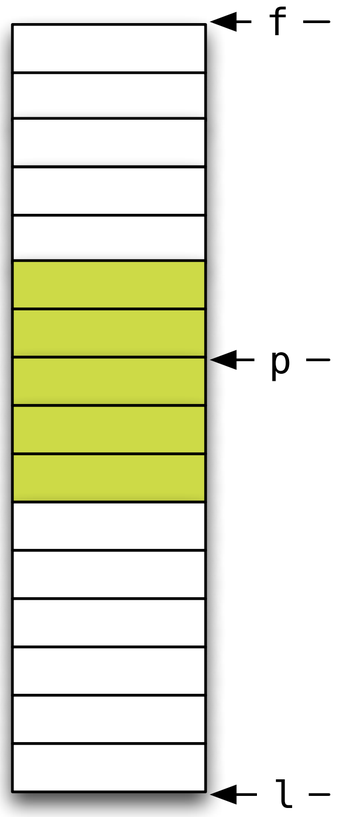

```
stable_partition(f, p, not_fn(s))
```

Composing Algorithms - Gather



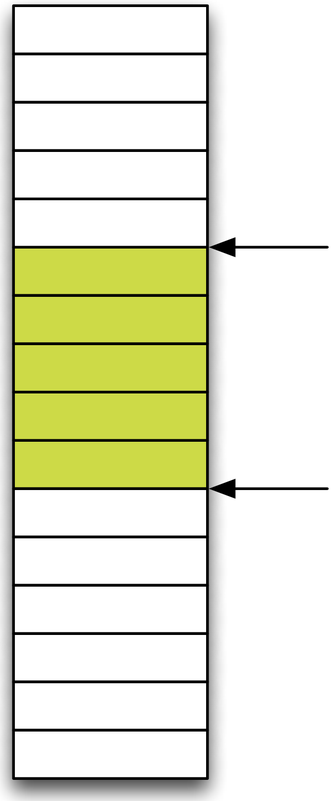

```
stable_partition(f, p, not_fn(s))
stable_partition(p, l, s)
```

Composing Algorithms - Gather



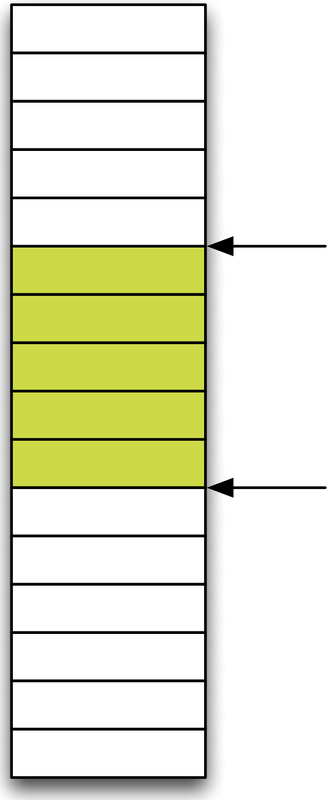

```
stable_partition(f, p, not_fn(s))
stable_partition(p, l, s)
```

Composing Algorithms - Gather



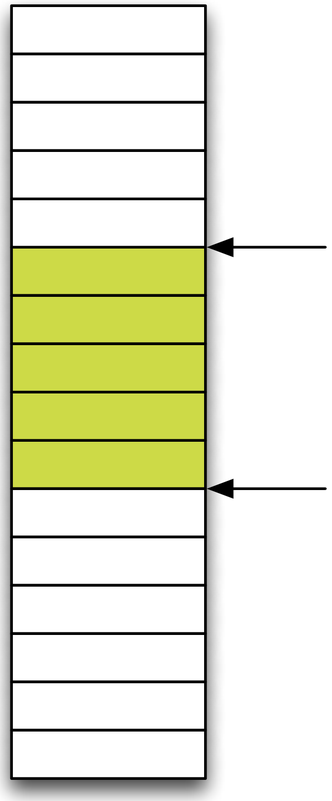
```
stable_partition(f, p, not_fn(s))
stable_partition(p, l, s)
```

Composing Algorithms - Gather



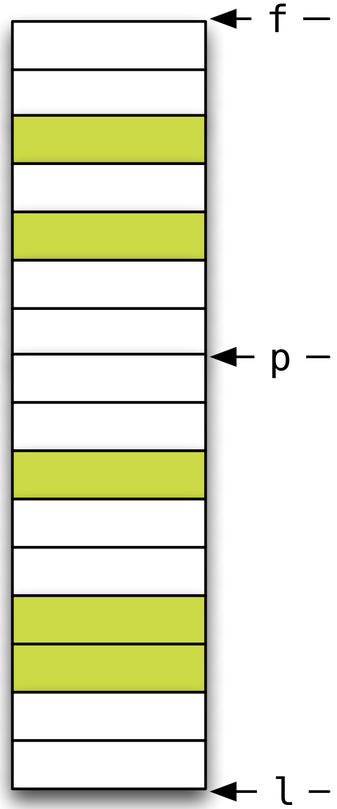
```
return { stable_partition(f, p, not_fn(s)),
         stable_partition(p, l, s) };
```

Composing Algorithms - Gather



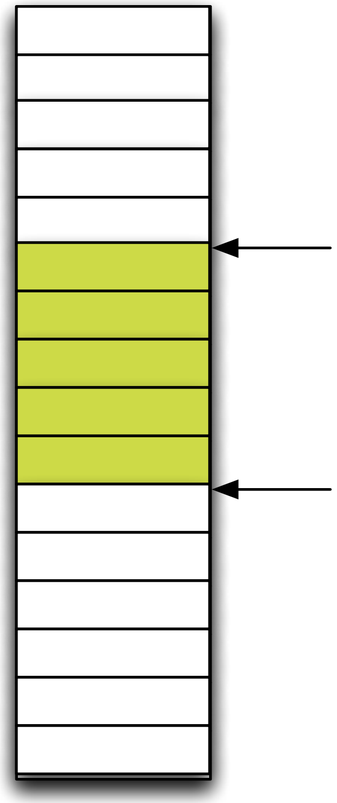
```
// Stably collects around p the elements in
// [f, l) satisfying s and returns the range
// satisfying s
// - Requires: p is within [f, l]

template <class I, // BidirectionalIterator
          class S> // UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not_fn(s)),
             stable_partition(p, l, s) };
}
```

Composing Algorithms - Gather




```
// Stably collects around p the elements in
// [f, l) satisfying s and returns the range
// satisfying s
// - Requires: p is within [f, l]

template <class I, // BidirectionalIterator
          class S> // UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not_fn(s)),
             stable_partition(p, l, s) };
}
```

Composing Algorithms - Gather



```
// Stably collects around p the elements in
// [f, l) satisfying s and returns the range
// satisfying s
// - Requires: p is within [f, l]

template <class I, // BidirectionalIterator
          class S> // UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not_fn(s)),
             stable_partition(p, l, s) };
}
```

What about that messy loop?

```
// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center() ||
      i == expanded_panels_.size() - 1) {
    if (panel != fixed_panel) {
      // If it has, then we reorder the panels.
      ref_ptr<Panel> ref = expanded_panels_[fixed_index];
      expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
      if (i < expanded_panels_.size()) {
        expanded_panels_.insert(expanded_panels_.begin() + i, ref);
      } else {
        expanded_panels_.push_back(ref);
      }
    }
    break;
  }
}
```

What about that messy loop?

```
// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center() ||
      i == expanded_panels_.size() - 1) {
    if (panel != fixed_panel) {
      // If it has, then we reorder the panels.
      ref_ptr<Panel> ref = expanded_panels_[fixed_index];
      expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
      if (i < expanded_panels_.size()) {
        expanded_panels_.insert(expanded_panels_.begin() + i, ref);
      } else {
        expanded_panels_.push_back(ref);
      }
    }
    break;
  }
}
```

What about that bad loop?

```
// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center() ||
      i == expanded_panels_.size() - 1) {
    if (panel != fixed_panel) {
      // If it has, then we reorder the panels.
      ref_ptr<Panel> ref = expanded_panels_[fixed_index];
      expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
      expanded_panels_.insert(expanded_panels_.begin() + i, ref);
    }
    break;
  }
}
```

What about that bad loop?

```
// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center() ||
      i == expanded_panels_.size() - 1) {
    if (panel != fixed_panel) {
      // If it has, then we reorder the panels.
      ref_ptr<Panel> ref = expanded_panels_[fixed_index];
      expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
      expanded_panels_.insert(expanded_panels_.begin() + i, ref);
    }
    break;
  }
}
```

What about that bad loop?

```
// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center() ||
      i == expanded_panels_.size() - 1) {
    break;
  }
}

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
   // If it has, then we reorder the panels.
   ref_ptr<Panel> ref = expanded_panels_[fixed_index];
   expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
   expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
```

What about that bad loop?

```
// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center() ||
      i == expanded_panels_.size() - 1) {
    break;
  }
}

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
   // If it has, then we reorder the panels.
   ref_ptr<Panel> ref = expanded_panels_[fixed_index];
   expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
   expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
```

What about that bad loop?

```
// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center()) break;
}

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
   // If it has, then we reorder the panels.
   ref_ptr<Panel> ref = expanded_panels_[fixed_index];
   expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
   expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
```

What about that bad loop?

```
// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center()) break;
}

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
   // If it has, then we reorder the panels.
   ref_ptr<Panel> ref = expanded_panels_[fixed_index];
   expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
   expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
```

What about that bad loop?

```
// Next, check if the panel has moved to the other side of another panel.

auto p = find_if(expanded_panels_,
  [&](const auto& e){ return center_x <= e->cur_panel_center(); });

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
   // If it has, then we reorder the panels.
   ref_ptr<Panel> ref = expanded_panels_[fixed_index];
   expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
   expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
```

What about that bad loop?

```
// Next, check if the panel has moved to the other side of another panel.

auto p = find_if(expanded_panels_,
  [&](const auto& e){ return center_x <= e->cur_panel_center(); });

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
   // If it has, then we reorder the panels.
   ref_ptr<Panel> ref = expanded_panels_[fixed_index];
   expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
   expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
```

What about that bad loop?

```
// Next, check if the panel has moved to the other side of another panel.

auto p = find_if(expanded_panels_,
  [&](const auto& e){ return center_x <= e->cur_panel_center(); });

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
  // If it has, then we reorder the panels.
  auto f = begin(expanded_panels_) + fixed_index;
  rotate(p, f, f + 1);
}
```

What about that bad loop?

```
// Next, check if the panel has moved to the other side of another panel.

auto p = find_if(expanded_panels_,
  [&](const auto& e){ return center_x <= e->cur_panel_center(); });

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
  // If it has, then we reorder the panels.
  auto f = begin(expanded_panels_) + fixed_index;
  rotate(p, f, f + 1);
}
```

What about that bad loop?

```
// Next, check if the panel has moved to the other side of another panel.

auto p = find_if(expanded_panels_,
  [&](const auto& e){ return center_x <= e->cur_panel_center(); });

// If it has, then we reorder the panels.
auto f = begin(expanded_panels_) + fixed_index;
rotate(p, f, f + 1);
```

What about that bad loop?

```
  // Next, check if the panel has moved to the other side of another panel.

  auto p = find_if(expanded_panels_,
    [&](const auto& e){ return center_x <= e->cur_panel_center(); });

  // If it has, then we reorder the panels.
  auto f = begin(expanded_panels_) + fixed_index;
  rotate(p, f, f + 1);
```

What about that bad loop?

```
// Next, check if the panel has moved to the left side of another panel.

auto p = find_if(expanded_panels_,
  [&](const auto& e){ return center_x <= e->cur_panel_center(); });

// If it has, then we reorder the panels.
auto f = begin(expanded_panels_) + fixed_index;
rotate(p, f, f + 1);
```

None of the special cases were necessary

This code is considerably more efficient

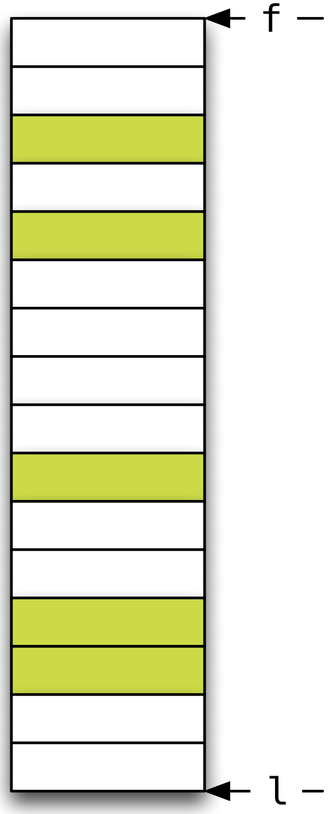
If you read the remaining code, this `rotate` is half of a `slide`

Now, we can have the conversation about supporting multiple selections and disjoint selections!

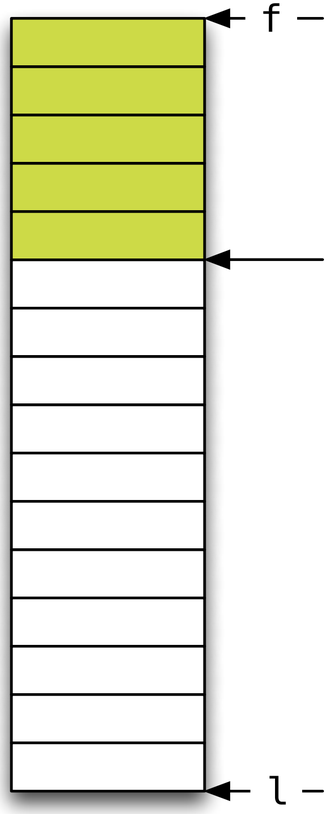
Key Point

Problem decomposition is the essence of algorithm composition.

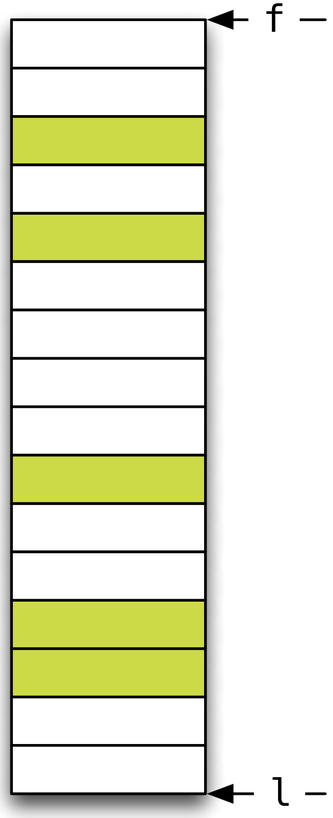
Stable Partition



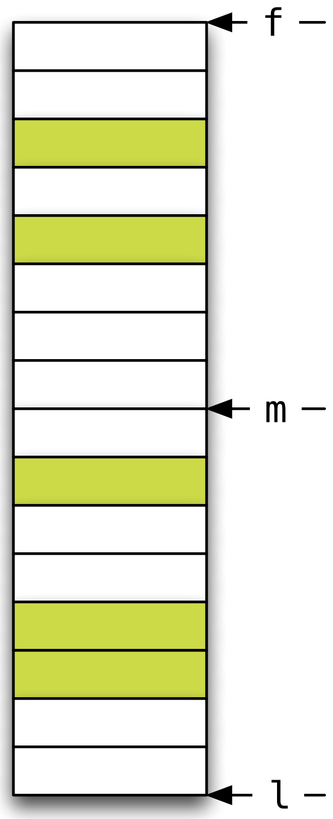
Stable Partition



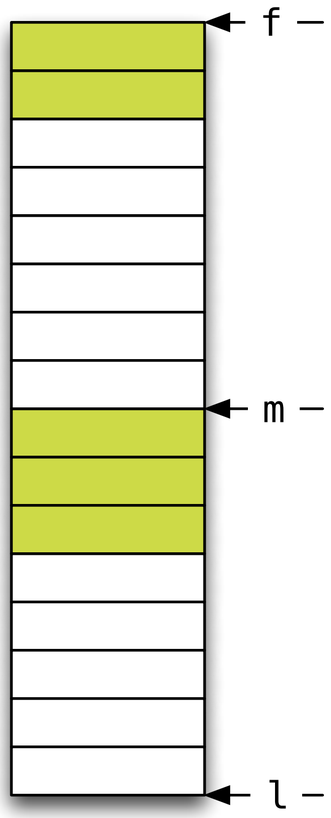
Stable Partition



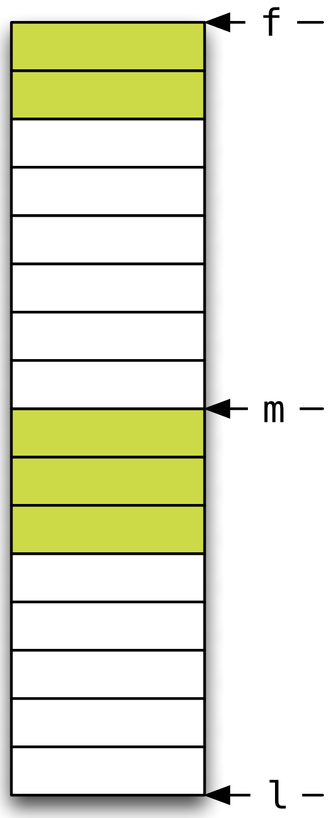
Stable Partition



Stable Partition

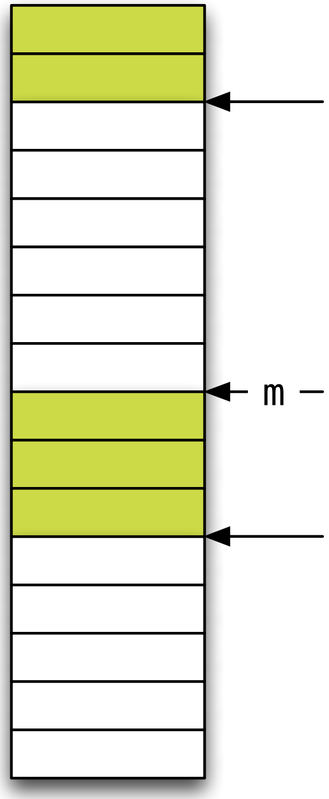


Stable Partition



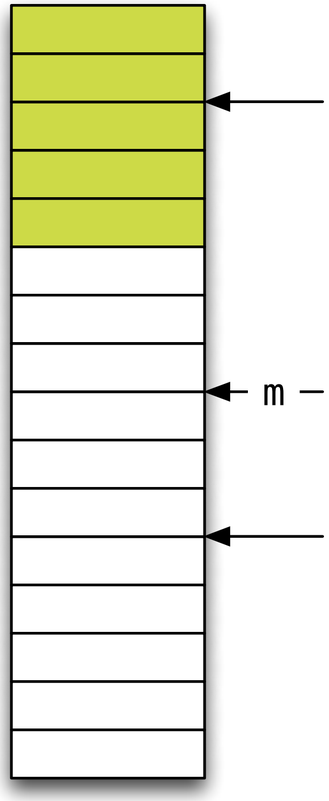

```
stable_partition(f, m, s)

stable_partition(m, l, s)
```

Stable Partition



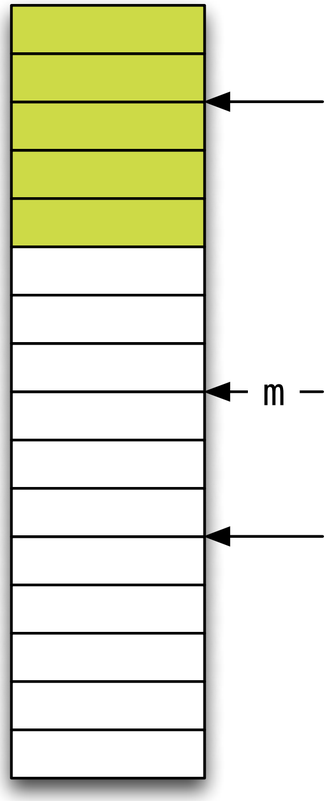

```
stable_partition(f, m, s)

stable_partition(m, l, s)
```

Stable Partition



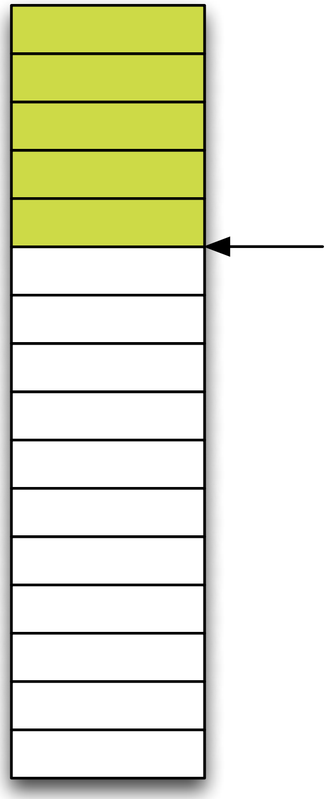

```
stable_partition(f, m, s)

stable_partition(m, l, s)
```

Stable Partition



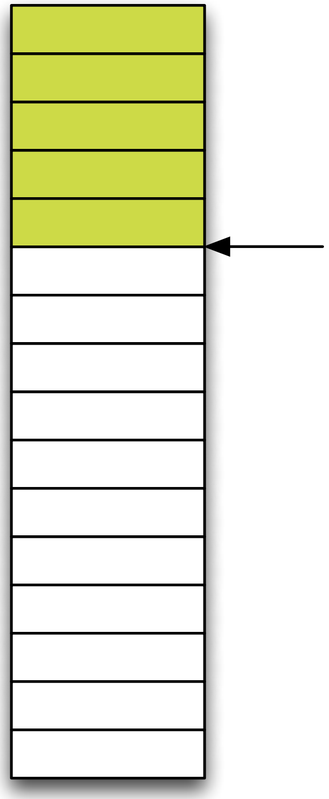

```
rotate(stable_partition(f, m, s),
       m,
       stable_partition(m, l, s));
```

Stable Partition



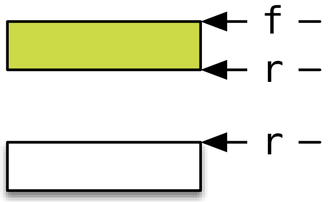
```
rotate(stable_partition(f, m, s),
       m,
       stable_partition(m, l, s));
```

Stable Partition



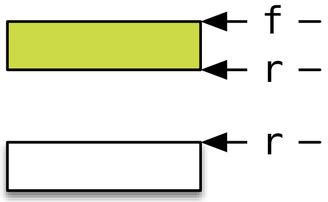
```
return rotate(stable_partition(f, m, s),
              m,
              stable_partition(m, l, s));
```

Stable Partition




```
return rotate(stable_partition(f, m, s),
              m,
              stable_partition(m, l, s));
```

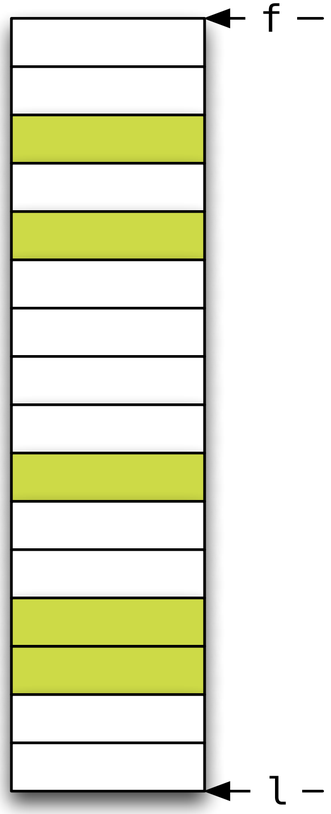
Stable Partition




```
if (n == 1) return next(f, s(*f));
```

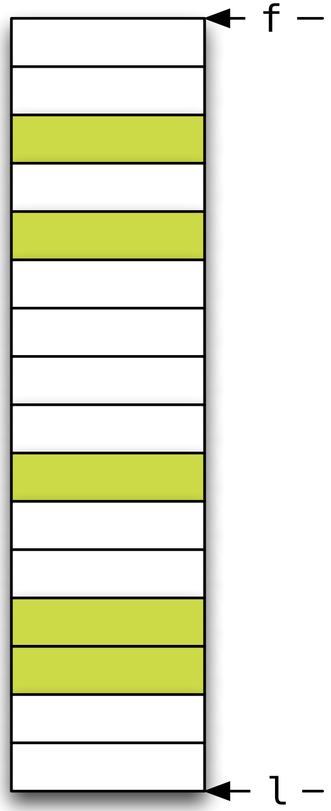
```
return rotate(stable_partition(f, m, s),
              m,
              stable_partition(m, l, s));
```

Stable Partition



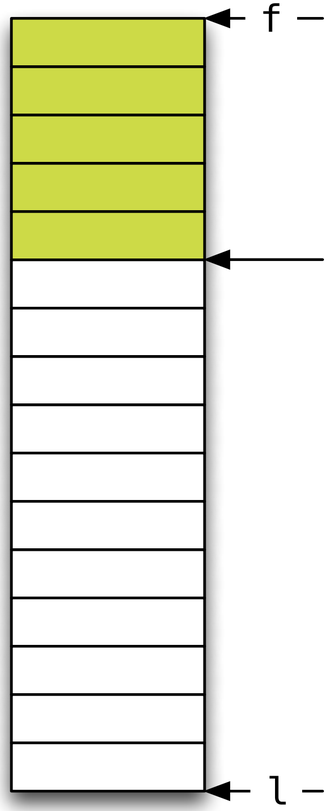

```
if (n == 1) return next(f, s(*f));


return rotate(stable_partition(f, m, s),
                        m,
                        stable_partition(m, l, s));
```

Stable Partition




```
template <class I, // ForwardIterator
          class S> // UnaryPredicate
auto stable_partition(I f, I l, S s) -> I
{
    auto n = distance(f, l);
    if (n == 0) return f;
    if (n == 1) return next(f, s(*f));

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, s),
                  m,
                  stable_partition(m, l, s));
}
```

Stable Partition




```
template <class I, // ForwardIterator
          class S> // UnaryPredicate
auto stable_partition(I f, I l, S s) -> I
{
    auto n = distance(f, l);
    if (n == 0) return f;
    if (n == 1) return next(f, s(*f));

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, s),
                  m,
                  stable_partition(m, l, s));
}
```

Algorithmic Forms

In Situ (in place)

Functional (non-mutating)

- Greedy
- Lazy

Lazy Stable Partition

Lazy Stable Partition

```
// Returns a view of r stable partitioned by s

template <view R, class S> // S models UnaryPredicate
view auto stable_partition_lazy(R r, S s) {
    return concat(filter(r, s), filter(r, not_fn(s)));
}
```

Lazy Stable Partition

```
// Returns a view of r stable partitioned by s

template <view R, class S> // S models UnaryPredicate
view auto stable_partition_lazy(R r, S s) {
    return concat(filter(r, s), filter(r, not_fn(s)));
}

int main() {
    array a{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (const auto& e : stable_partition_lazy(all(a), is_odd)) {
        std::cout << e << " ";
    }
}
```

Lazy Stable Partition

```
// Returns a view of r stable partitioned by s

template <view R, class S> // S models UnaryPredicate
view auto stable_partition_lazy(R r, S s) {
    return concat(filter(r, s), filter(r, not_fn(s)));
}

int main() {
    array a{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (const auto& e : stable_partition_lazy(all(a), is_odd)) {
        std::cout << e << " ";
    }
}
```

1 3 5 7 9 0 2 4 6 8

Greedy Stable Partition

Greedy Stable Partition

```
// Returns an instance of R containing copies of the elements in r stable partitioned by s

template <range R, class S>
R stable_partition_greedy(const R& r, S s) {
    return stable_partition_lazy(all(r), s) | to<R>();
}
```

Greedy Stable Partition

```
// Returns an instance of R containing copies of the elements in r stable partitioned by s

template <range R, class S>
R stable_partition_greedy(const R& r, S s) {
    return stable_partition_lazy(all(r), s) | to<R>();
}

int main() {
    vector a{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector b{stable_partition_greedy(a, is_odd)};

    for (const auto& e : b) {
        std::cout << e << " ";
    }
}
```

Greedy Stable Partition

```
// Returns an instance of R containing copies of the elements in r stable partitioned by s

template <range R, class S>
R stable_partition_greedy(const R& r, S s) {
    return stable_partition_lazy(all(r), s) | to<R>();
}

int main() {
    vector a{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector b{stable_partition_greedy(a, is_odd)};

    for (const auto& e : b) {
        std::cout << e << " ";
    }
}
```

Greedy Stable Partition

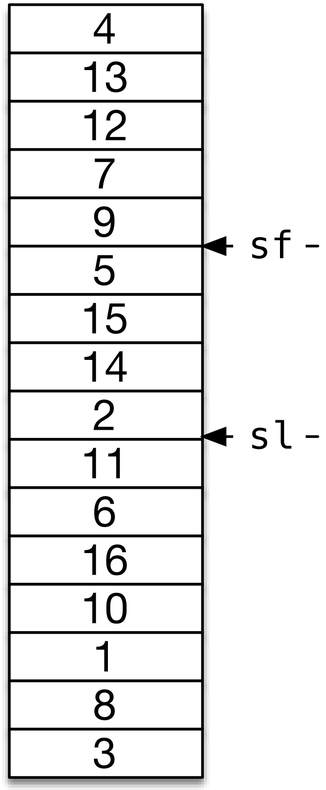
```
// Returns an instance of R containing copies of the elements in r stable partitioned by s

template <range R, class S>
R stable_partition_greedy(const R& r, S s) {
    return stable_partition_lazy(all(r), s) | to<R>();
}

int main() {
    vector a{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector b{stable_partition_greedy(a, is_odd)};

    for (const auto& e : b) {
        std::cout << e << " ";
    }
}
```

1 3 5 7 9 0 2 4 6 8

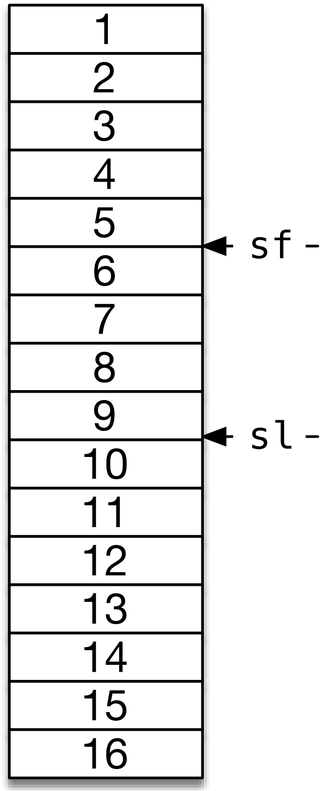
Partial Algorithms

Partial algorithms provide a *partial* result with better efficiency than a complete solution

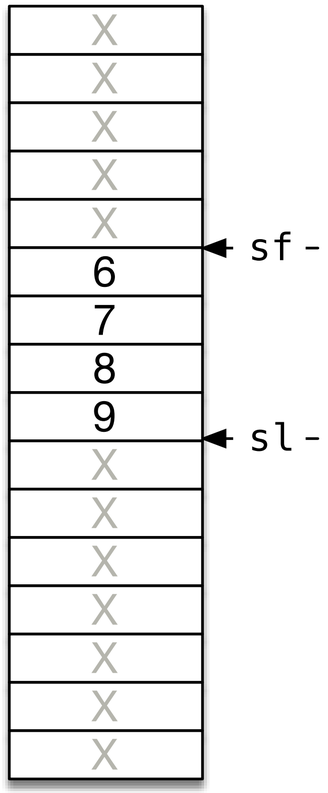
Minimize Work



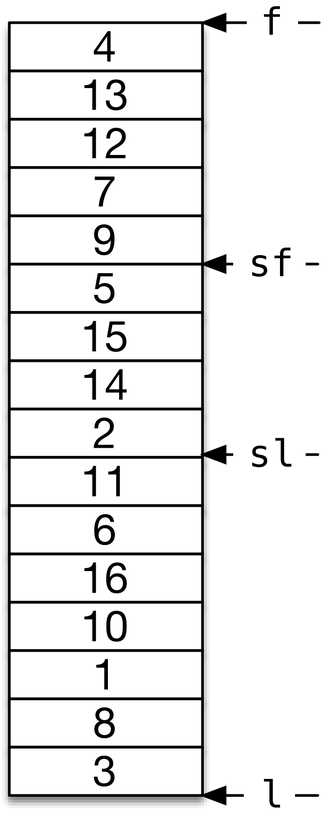
Minimize Work



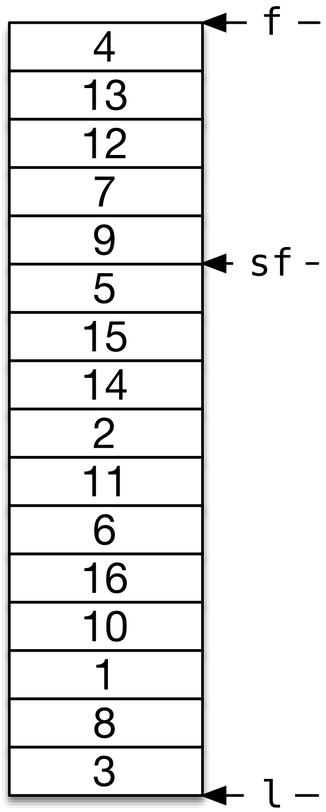
Minimize Work



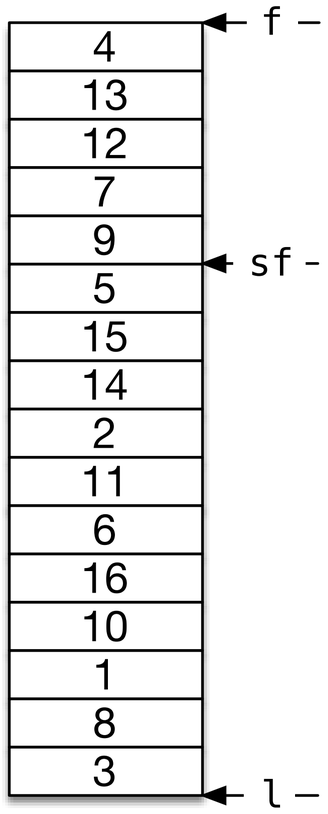
Minimize Work



Minimize Work

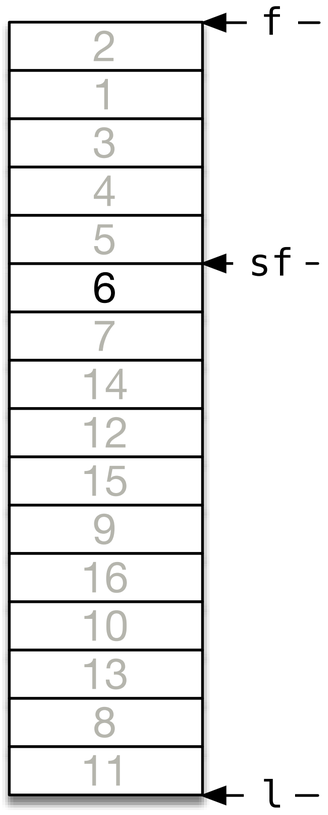


Minimize Work



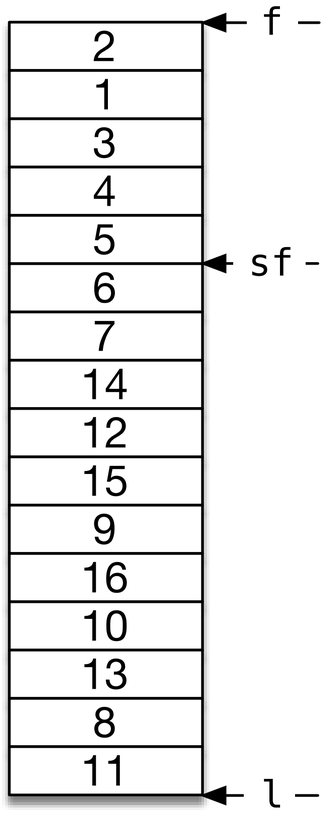

```
nth_element(f, sf, l);
```

Minimize Work



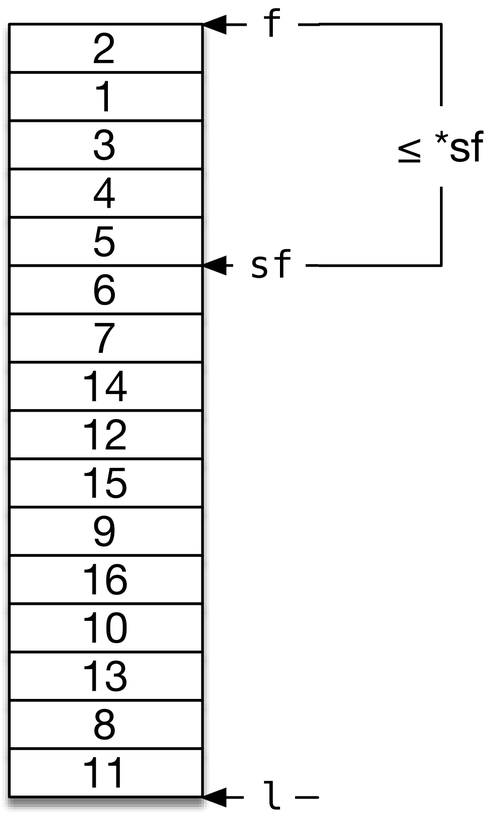

```
nth_element(f, sf, l);
```

Minimize Work



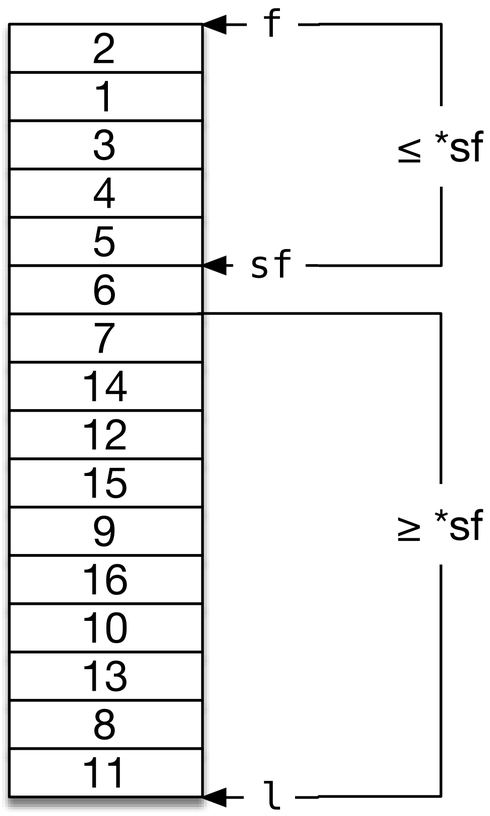

```
nth_element(f, sf, l);
```

Minimize Work



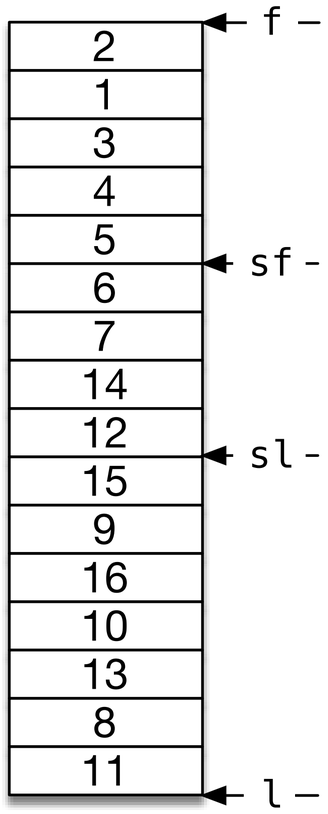

```
nth_element(f, sf, l);
```

Minimize Work



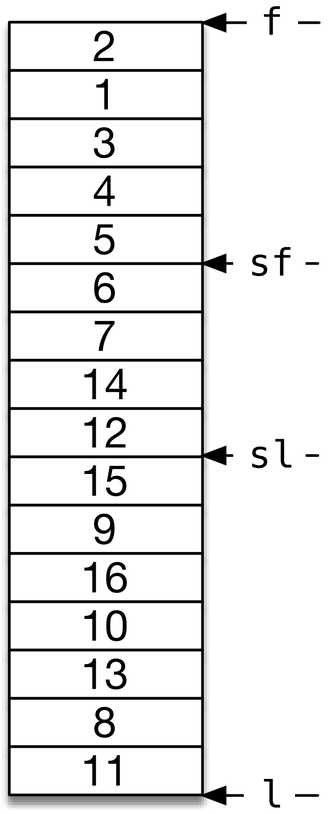

```
nth_element(f, sf, l);
```

Minimize Work



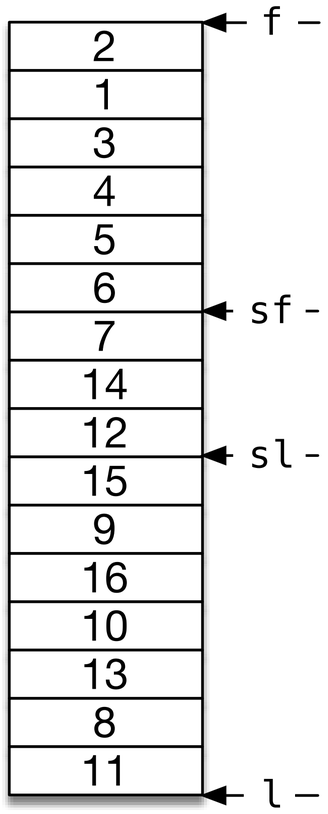

```
nth_element(f, sf, l);
```

Minimize Work



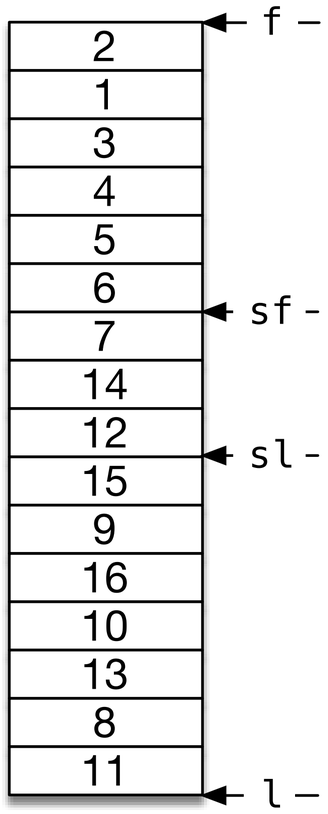

```
nth_element(f, sf, l);
++sf;
```

Minimize Work



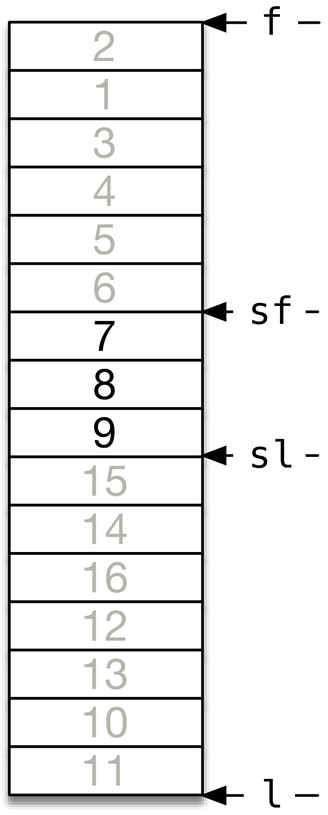

```
nth_element(f, sf, l);
++sf;
```

Minimize Work



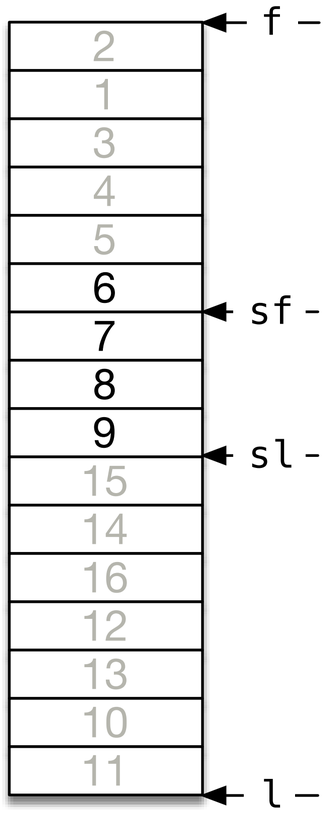

```
    nth_element(f, sf, l);
    ++sf;

partial_sort(sf, sl, l);
```

Minimize Work



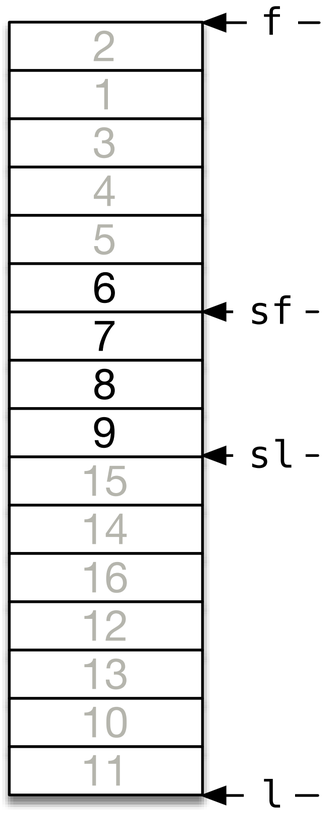

```
    nth_element(f, sf, l);
    ++sf;

partial_sort(sf, sl, l);
```

Minimize Work



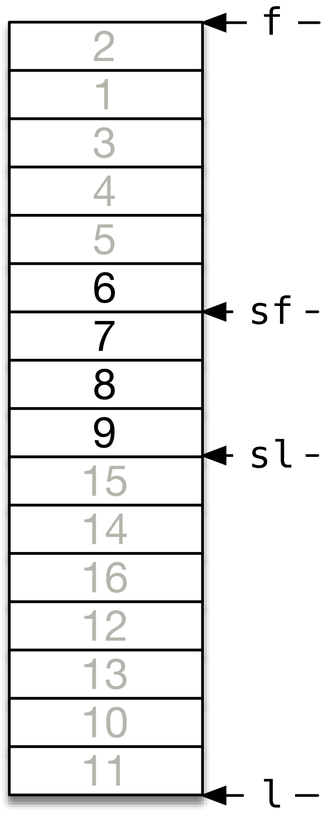

```
    nth_element(f, sf, l);
    ++sf;

partial_sort(sf, sl, l);
```

Minimize Work



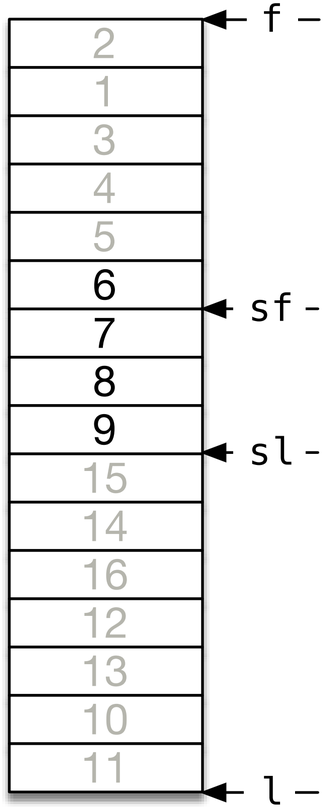

```
if (sf == sl) return;

    nth_element(f, sf, l);
    ++sf;

partial_sort(sf, sl, l);
```

Minimize Work



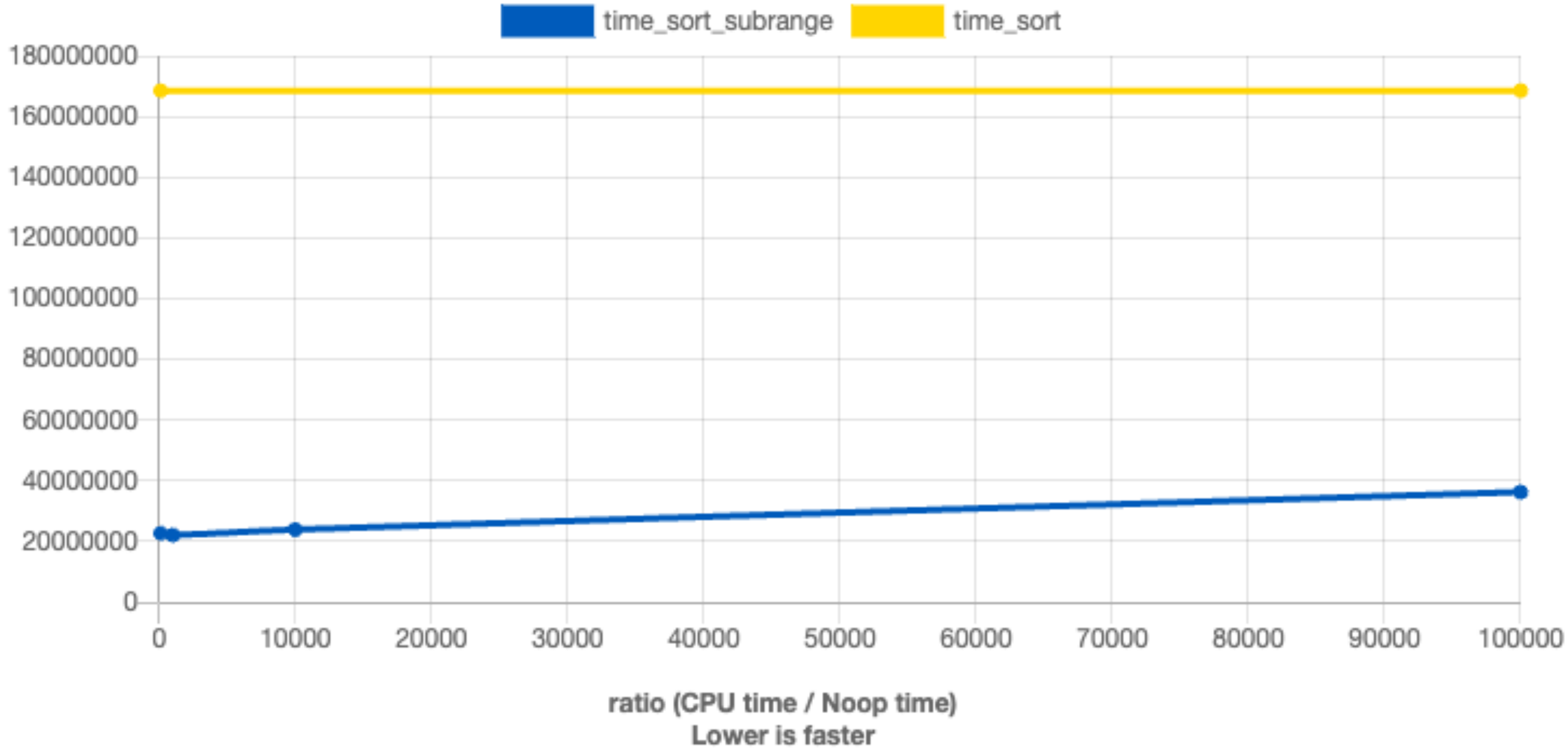

```
if (sf == sl) return;
if (sf != f) {
    nth_element(f, sf, l);
    ++sf;
}
partial_sort(sf, sl, l);
```

Minimize Work

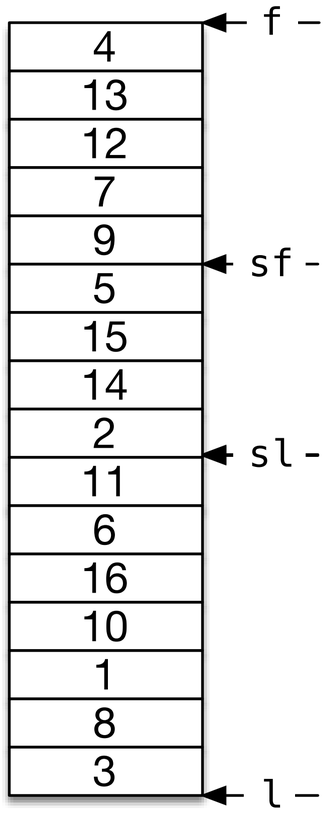



```
template <typename I> // I models RandomAccessIterator
void sort_subrange(I f, I l, I sf, I sl)
{
    if (sf == sl) return;
    if (sf != f) {
        nth_element(f, sf, l);
        ++sf;
    }
    partial_sort(sf, sl, l);
}
```

Sort Subrange

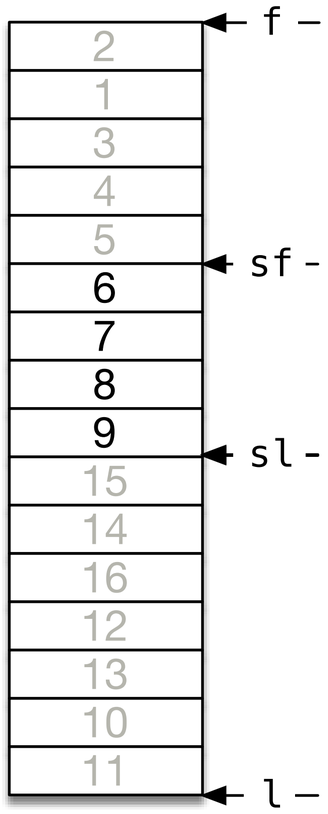


Minimize Work



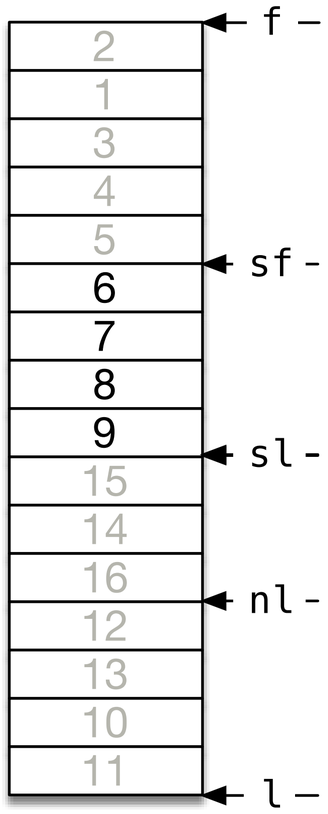

```
sort_subrange(f, l, sf, sl);
```

Minimize Work



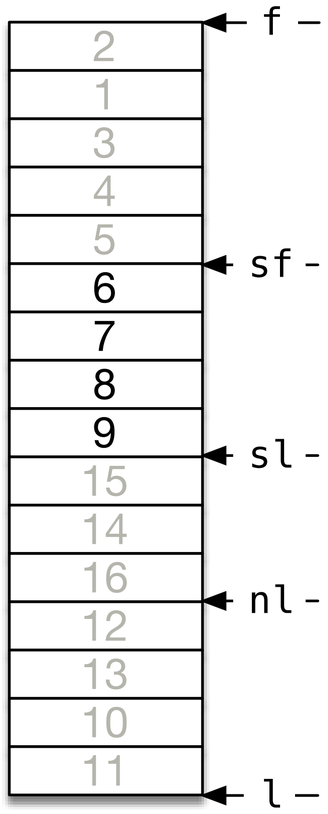

```
sort_subrange(f, l, sf, sl);
```

Minimize Work



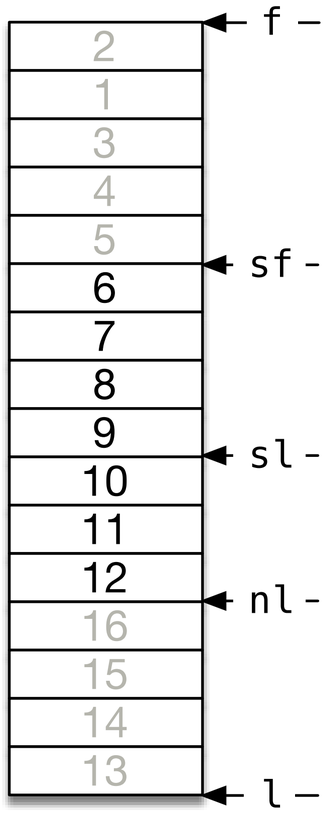

```
sort_subrange(f, l, sf, sl);
```

Minimize Work




```
sort_subrange(f, l, sf, sl);
partial_sort(sl, nl, l);
```

Minimize Work




```
sort_subrange(f, l, sf, sl);
partial_sort(sl, nl, l);
```

Structured Data

Structured data is data organized in memory such that a spatial relationship maps to a value relationship

Examples:

- A *sorted sequence* maps an ordering of values to an order in memory
- A *heap* maps an ordering of values to a tree structure, which is mapped to an order in memory

Operations on Sorted Sequences

Searching: `lower_bound`, `upper_bound`, `equal_range`

Set Operations: `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference`, `set_union`

Other: `merge`, `inplace_merge`

Operations on Heaps

Queue: `push_heap`, `pop_heap`

Sorting: `sort_heap`

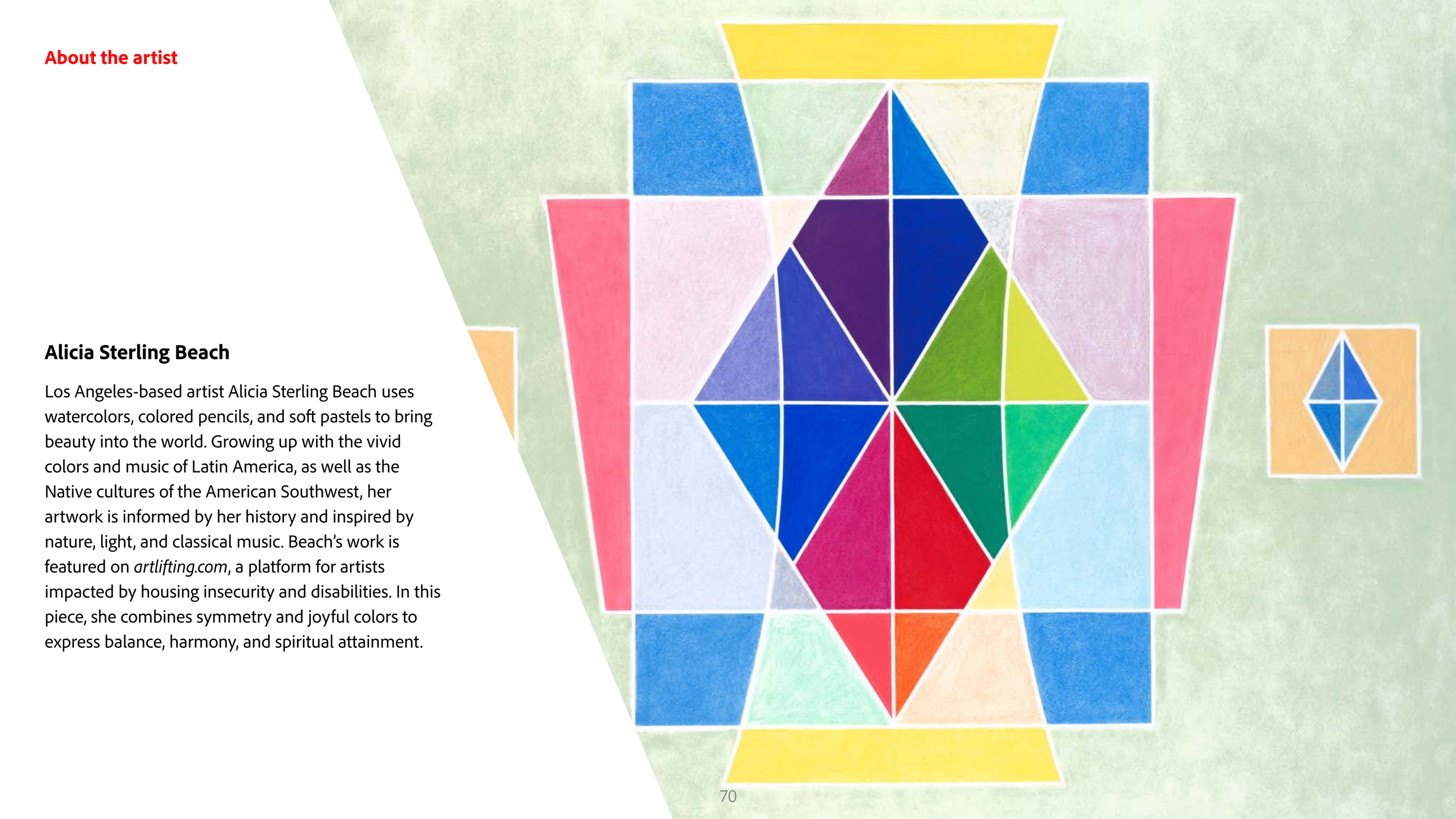
Closing Thoughts

"Science is about classification. Science is not about grand inspiration." - Alex Stepanov

Software engineering is an applied science

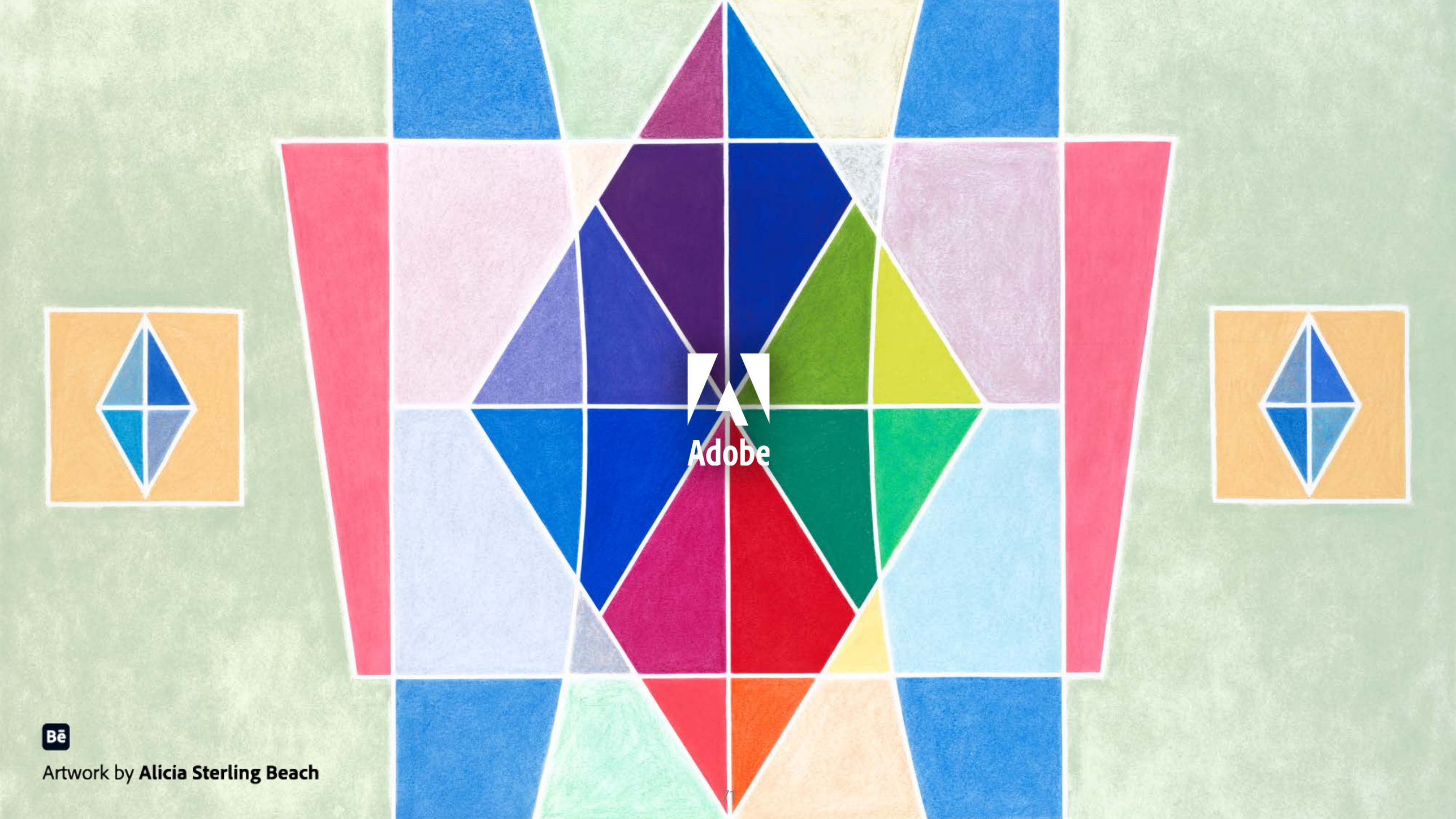
- Learn algorithms and their classifications
- Learn to recognize algorithms hidden in the problems you solve and the code you write

About the artist



Alicia Sterling Beach

Los Angeles-based artist Alicia Sterling Beach uses watercolors, colored pencils, and soft pastels to bring beauty into the world. Growing up with the vivid colors and music of Latin America, as well as the Native cultures of the American Southwest, her artwork is informed by her history and inspired by nature, light, and classical music. Beach's work is featured on *artlifting.com*, a platform for artists impacted by housing insecurity and disabilities. In this piece, she combines symmetry and joyful colors to express balance, harmony, and spiritual attainment.



Artwork by **Alicia Sterling Beach**