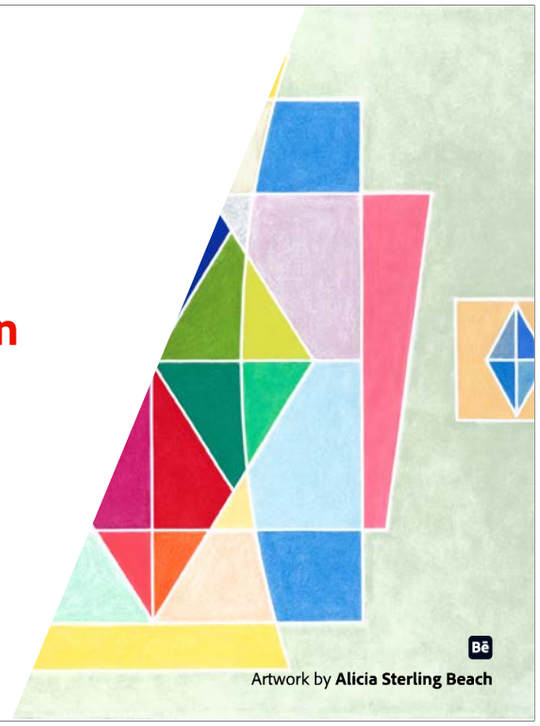
Algorithms - Composition

Rubric: No Raw Loops

Sean Parent | Sr. Principal Scientist
Adobe Software Technology Lab



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

A short recap of our last algorithms seminar.

A short apology - this talk is very C++-centric, but all ideas can be translated into any language. I also set out to use typescript, but the state of typescript/javascript algorithm libraries, to the best I could determine, is horrendous.

Rust is not much better.

The other options were Swift and APL.

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

Combinators are a way to represent functions without variables - purely through composition. Learning combinatory logic can be useful in seeing how to compose algorithms and reduce compositions. The S combinator is the "applies," and the K combinator is "constant". The S and K combinators are Turing complete. There is a $20,000 prize from Stephan Wolfram if you can prove his conjecture that just the S combinator is Turing complete.

Category theory is a general theory of mathematical structures and their relations. Category theory can provide insight into how types with specific operations can be composed. I won't even attempt the category theory definition of a monoid - see Wikipedia.

Both are about the abstract structure of computation - not about what the computation _does_.

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

Unencapsulate - tightly bound.

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;
```


Single function from Chromium OS
This is the first code I reviewed at Google - this code was included in the original Chrome OS
Not picking on Google (or anyone)- I've certainly written code like this
* What are the postconditions of this loop?
* What are the loop invariants?

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();
  }
```

```
  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?

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);
  fixed_index = new_fixed_index;
```

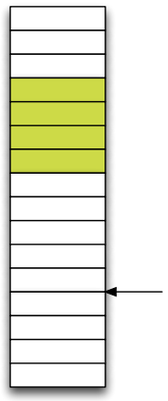
What is a Raw Loop?

```
  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;
```

```
  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;
  }
}
```

What is a Raw Loop?

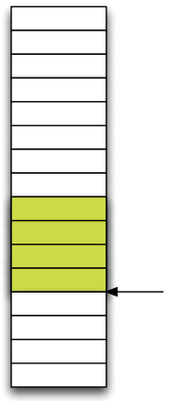
Slide



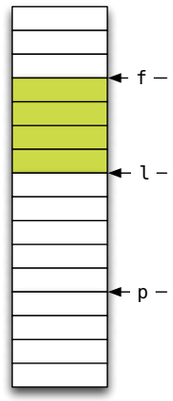
Setting that code aside - let's look at a couple of uses of algorithms to solve real problems. The first problem we will solve is moving a range of items in a sequence to a position. This operation would happen if a user selected a range and dragged them to a location in the sequence.

I want you to think briefly about how you would write this code.

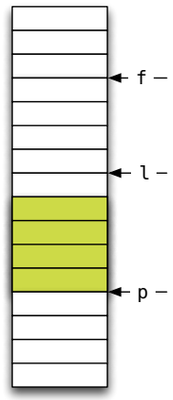
Slide



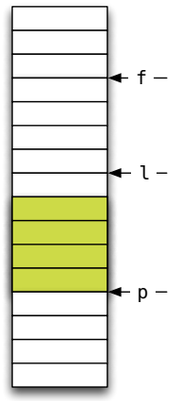
Slide



Slide

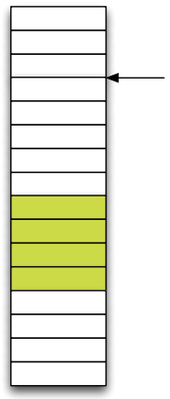


Slide



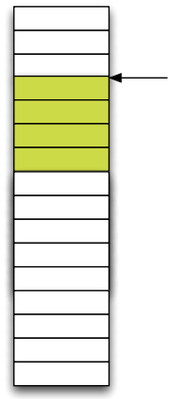

```
rotate(f, l, p);
```

Slide



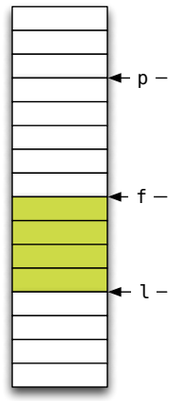
```
rotate(f, l, p);
```

Slide



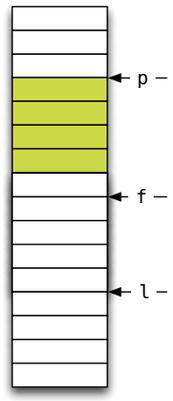
```
rotate(f, l, p);
```

Slide



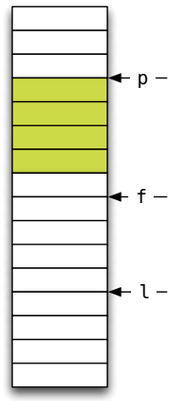

```
rotate(p, f, l);
rotate(f, l, p);
```

Slide



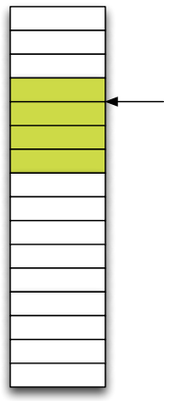

```
rotate(p, f, l);
rotate(f, l, p);
```

Slide




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

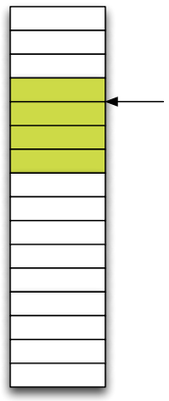
Slide



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

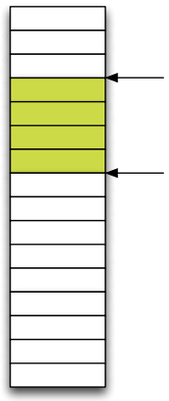

If the position we are moving the elements to is within the existing range of elements, we don't need to do anything. So, no code.

Slide



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

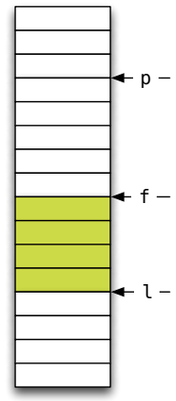
Slide



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


After this operation, we may want to know where all the selected items ended up.

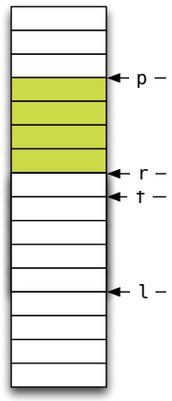
Slide




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

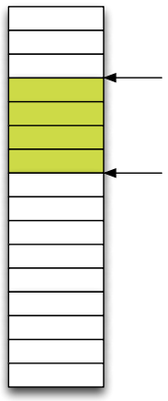
Rotate returns where the midpoint moved to.

Slide



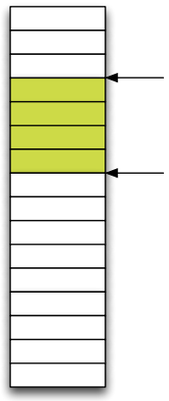

```
if (p < f) return { p, rotate(p, f, l) };
if (l < p) return { rotate(f, l, p), p };
```

Slide



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

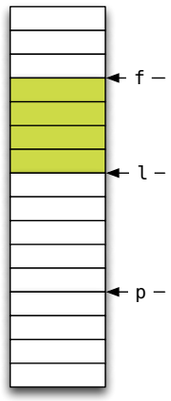
Slide



```
// 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 };
}
```


[Add a slide after this showing slide(f, l, p)...] [Add context in wording to build context. Maybe put a gif at the end... Consider adding elses... Maybe more vertical white space...]

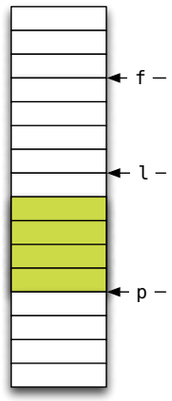
Slide




```
// 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 };
}
```

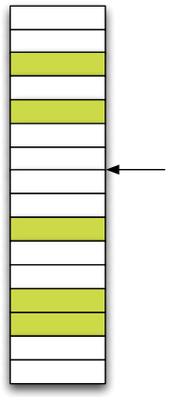

We've taken an existing algorithm, `rotate`, and composed it to create a new algorithm, `slide`.

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 };
}
```

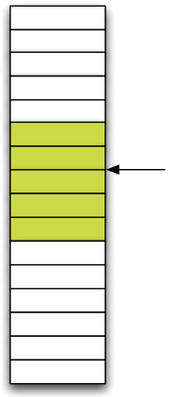
Gather



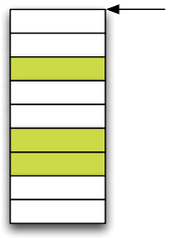
Let's make the problem harder. Now, the selected elements are disjoint.

We want to move the selected elements to a specific position. As if the user did a drag/drop of a disjoint selection.

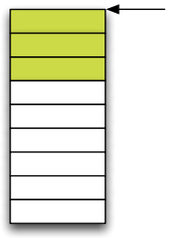
Gather



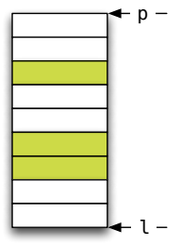
Composing Algorithms - Gather



Composing Algorithms - Gather

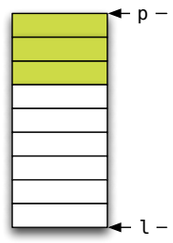


Composing Algorithms - Gather



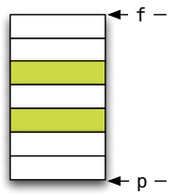

```
stable_partition(p, l, s)
```

Composing Algorithms - Gather




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

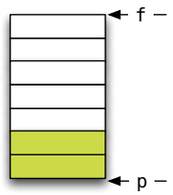
Composing Algorithms - Gather




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

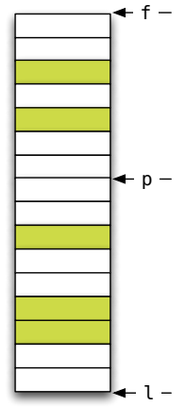
`not_fn` negates the predicate, so the selected elements are moved to the end.

Composing Algorithms - Gather



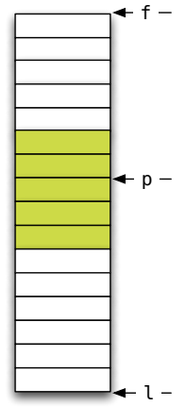

```
stable_partition(f, p, not_fn(s))
```

Composing Algorithms - Gather



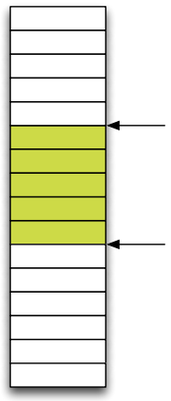

```
stable_partition(f, p, not_fn(s))
stable_partition(p, l, s)
```

Composing Algorithms - Gather



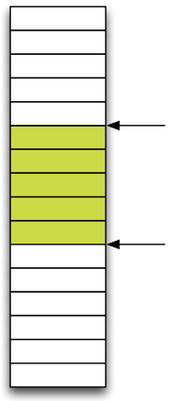

```
stable_partition(f, p, not_fn(s))
stable_partition(p, l, s)
```

Composing Algorithms - Gather



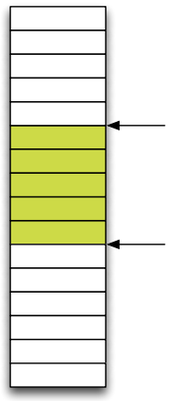
```
stable_partition(f, p, not_fn(s))
stable_partition(p, l, s)
```

Composing Algorithms - Gather



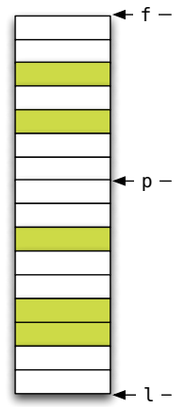
```
return { stable_partition(f, p, not_fn(s)),
         stable_partition(p, l, s) };
```

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

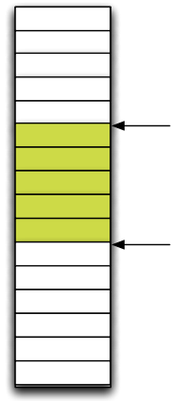



```
// Stably collects around p the elements in
// [f, l) satisfying s and returns the range
// satisfying s
// - Requires: p is within [f, l]

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) };
}
```


[Return range satisfy s. postcondition could be dropped.]

[note to improve] Stably collects around p the elements in [f, l) satisfying s, then returns the range satisfying 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);
}
```


This is redundant - we are looking for an element that we _know_ is in the list.

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);
}
```


This is find_if()

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);
}
```


This is find_if()

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);
}
```


We don’t need this check - we can rotate an empty range. Put limit back on find_if range at this point.

Use next? p is panel.

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);
```


Now are comment makes sense!

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`

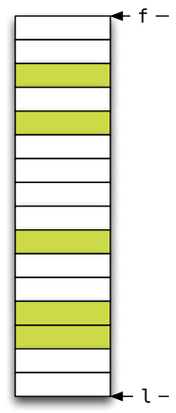
Now, we can have the conversation about supporting multiple selections and disjoint selections!

We don’t need this check - we can rotate an empty range!

Key Point

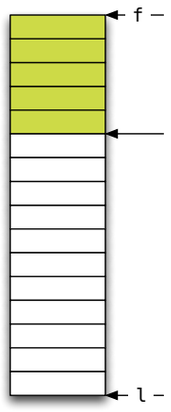
Problem decomposition is the essence of algorithm composition.

Stable Partition

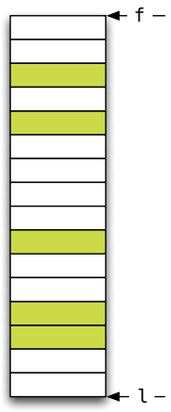


I presented the gather algorithm at a user group meeting. Jon Kalb commented after that, "it was pretty, but few algorithms compose like that." But this isn't true - most algorithms are simple compositions of other algorithms. Let's look at how to implement stable partition

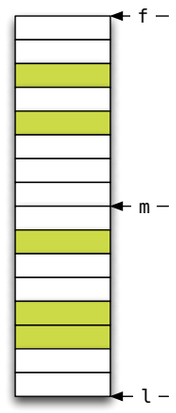
Stable Partition



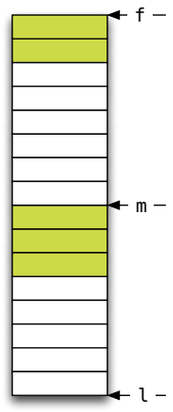
Stable Partition



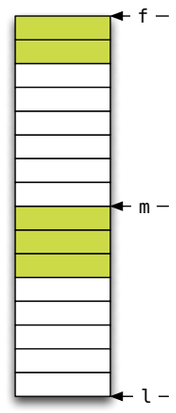
Stable Partition



Stable Partition

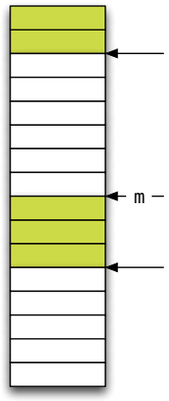


Stable Partition



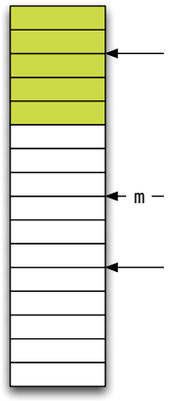

```
stable_partition(f, m, s)

stable_partition(m, l, s)
```

Stable Partition



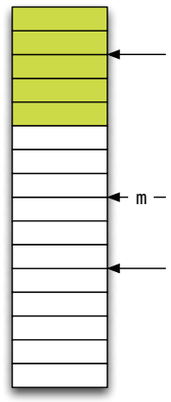

```
stable_partition(f, m, s)

stable_partition(m, l, s)
```

Stable Partition



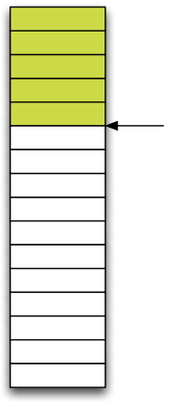

```
stable_partition(f, m, s)

stable_partition(m, l, s)
```

Stable Partition



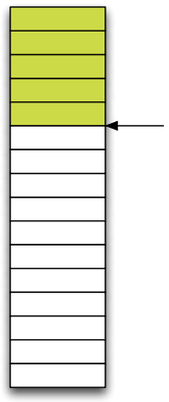

```
rotate(stable_partition(f, m, s),
       m,
       stable_partition(m, l, s));
```

Stable Partition



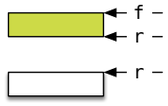
```
rotate(stable_partition(f, m, s),
       m,
       stable_partition(m, l, s));
```

Stable Partition



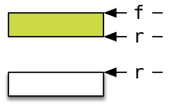
```
return rotate(stable_partition(f, m, s),
              m,
              stable_partition(m, l, s));
```

Stable Partition



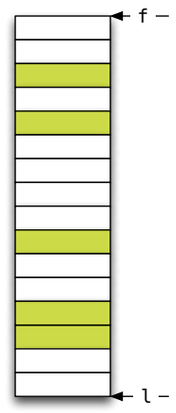

```
return rotate(stable_partition(f, m, s),
              m,
              stable_partition(m, l, s));
```

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



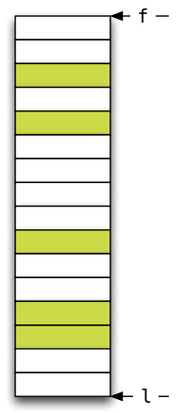

```
if (n == 1) return next(f, s(*f));


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


I did not include a contract here because stable_partition is a standard algorithm, and there wasn't space on the slide.

Interestingly, the predicate is only evaluated once on each element before the element is moved. Then everything is rotated into position.
A stable partition is implemented with *rotate()*. Rotate is a fascinating algorithm that could fill an hour, but one implementation is three reverses.
Reverse is iterative calls to swap. Many STL algorithms, including stable partition, exist to implement in-place stable sort.

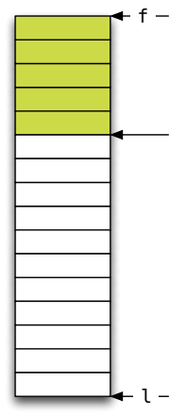
[Fix forward iterator requirement. Maybe take out genericity?]

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

All forms are Turing complete. Any algorithm can be expressed in all three ways
For any algorithm, one form or another may be more efficient in time, complexity, or space. But there may also be other tradeoffs.

One key advantage to functional forms is that they are simpler to compose.

Lazy Stable Partition

A lazy stable partition will just make two passes on the elements. The first pass filters by the predicate and the second pass filters by the negation of the predicate. Concat will just present these two views as a single, contiguous view. Lazy composition is a form of function composition. We aren't actually doing the stable partition, we are creating a function that will do the stable partition.

Our prior implementation will do n applications of the predicate and roughly n log n move operations. This code will do 2n applications of the predicate but no moves or copies.

This form doesn't return the partition point. Figuring out how to return the partition point is tricky. And at first glance, the question is "why".

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

We can simply "pipe" our lazy partition into a variable-sized container (we can't use an array anymore). Now we can see why having the partition point communicated would be useful. If we want the greedy form to also return the partition point, we would have to make the filters explicit.

This does 2n applications of the predicate and 2n copies and log n heap allocations because the vector will dynamically grow. Correct and simple, but we would need to do more work to be efficient.

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 << " ";
    }
}
```

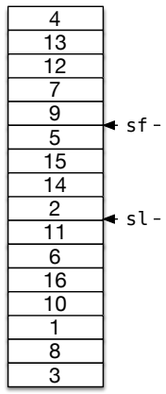
1 3 5 7 9 0 2 4 6 8

Partial Algorithms

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

To understand, lets use partial algorithms to solve a problem efficiently.

Minimize Work



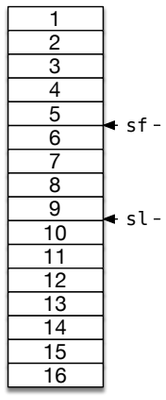
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work



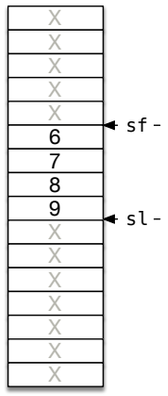
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work



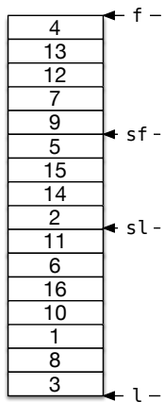
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work



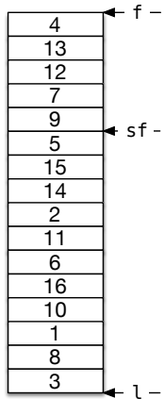
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work



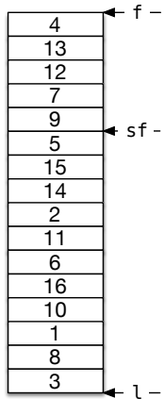
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

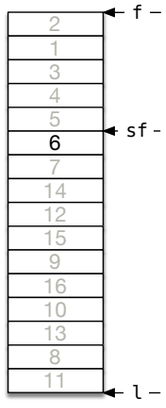
This algorithm is often 2-4x faster or more

Minimize Work




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

Minimize Work




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

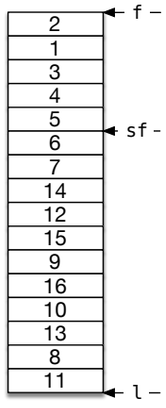
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work




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

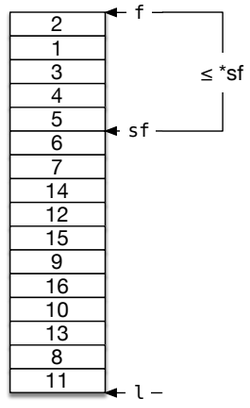
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

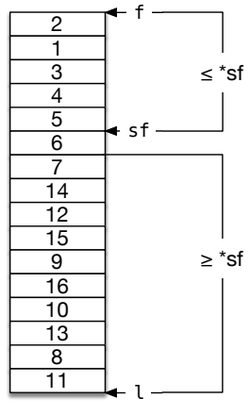
This algorithm is often 2-4x faster or more

Minimize Work



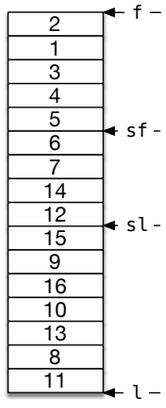

```
nth_element(f, sf, l);
```

Minimize Work




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

Minimize Work




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

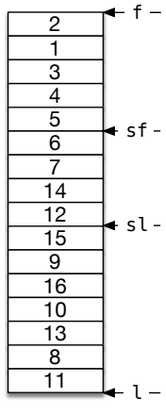
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work




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

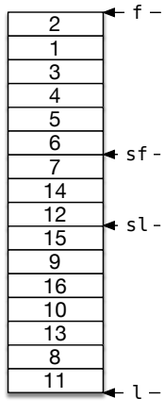
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

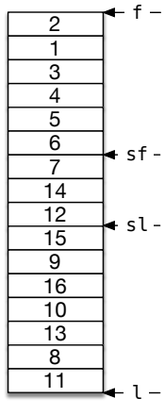
This algorithm is often 2-4x faster or more

Minimize Work




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

Minimize Work




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

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

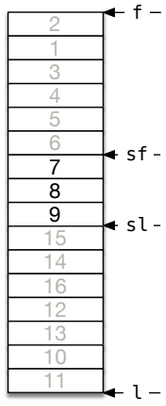
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work




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

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

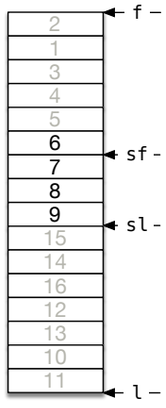
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work




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

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

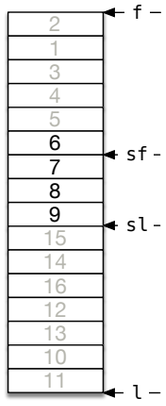
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work




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

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

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

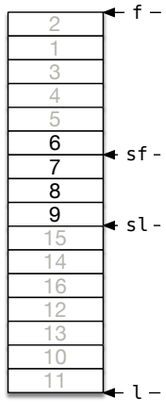
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work




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

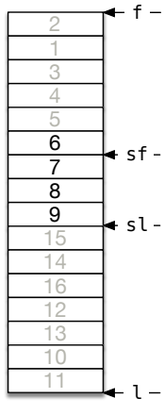
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

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);
}
```

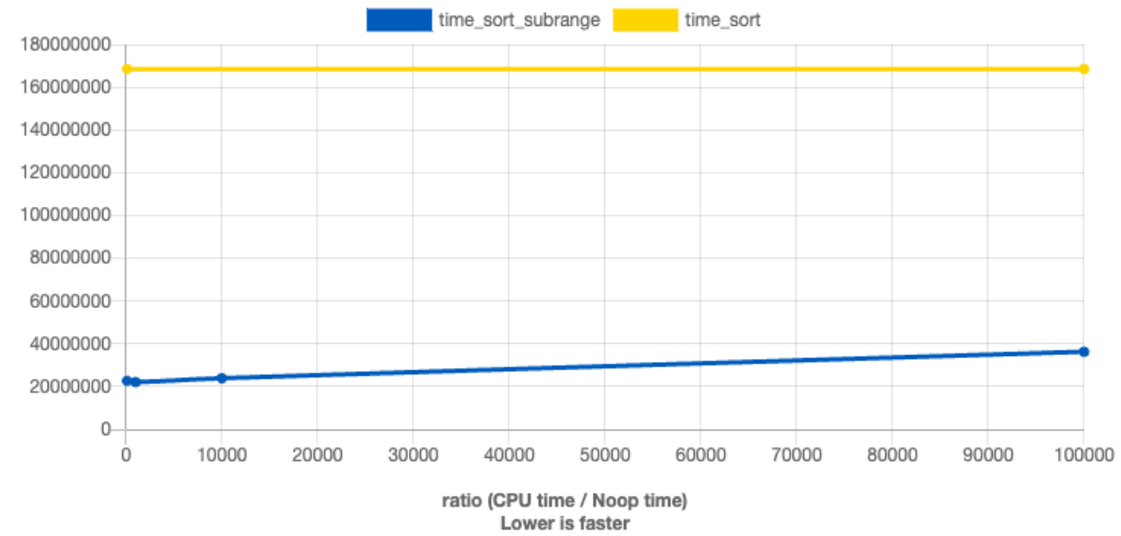
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

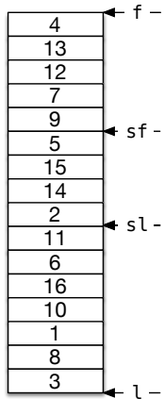
Sort Subrange



This is for 100, 1'000, 10'000, and 100'000 elements out of 1M elements. At 100'000 elements, sort_subrange is > 4x faster than a full sort.

https://quick-bench.com/q/QQYGDY-wpbRbpkvVspUYePa9nOQ

Minimize Work




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

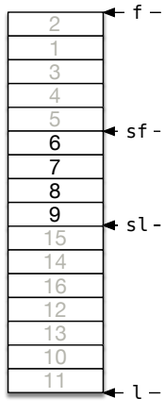
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work




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

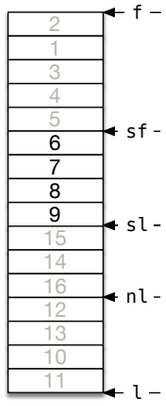
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

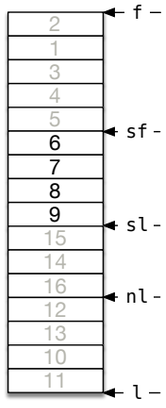
This algorithm is often 2-4x faster or more

Minimize Work




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

Minimize Work




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

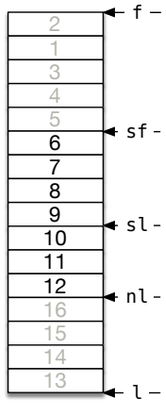
Only do what is needed

Works because half the work of divide and concur algorithms is done at the bottom of the pyramid

Trim layers off the bottom, get a 2x+ performance increase

This algorithm is often 2-4x faster or more

Minimize Work




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

Many, but not all algorithms have good locality (binary search algorithms like lower bound and heap operations are exceptions) -

Even algorithms without good locality, however, can be more efficient than a linked structure applied to compact data structure.

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

We create structured data as a way to execute other algorithms efficiently. This is a form of composition by storing state.

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`

lower_bound is the most commonly used algorithm, returning the first position where an element _would go_ if inserted in order.

Operations on Heaps

Queue: `push_heap, pop_heap`

Sorting: `sort_heap`

Heaps are typically used to create priority queues. They can also be used as a form of partial sort. Be aware that the heap queues are not stable - the order in which equivalent elements are pushed has no relationship to the order in which they are popped. The order is determined by the heap structure and the time the element is pushed.

Closing Thoughts

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

Software engineering is an applied science

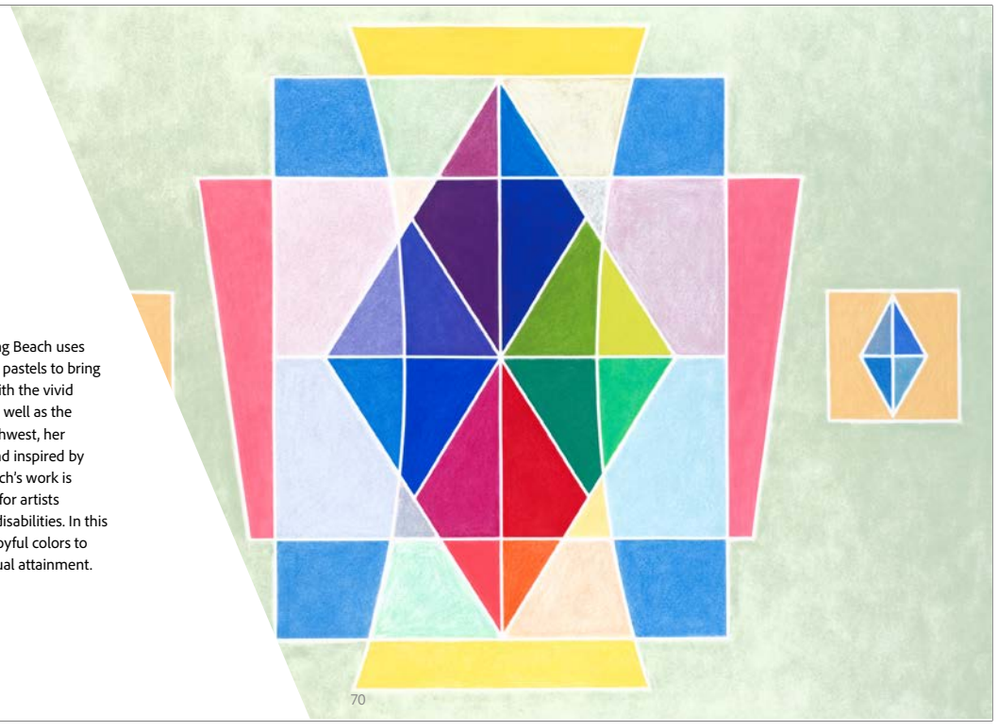
- Learn algorithms and their classifications
- Learn to recognize algorithms hidden in the problems you solve and the code you write

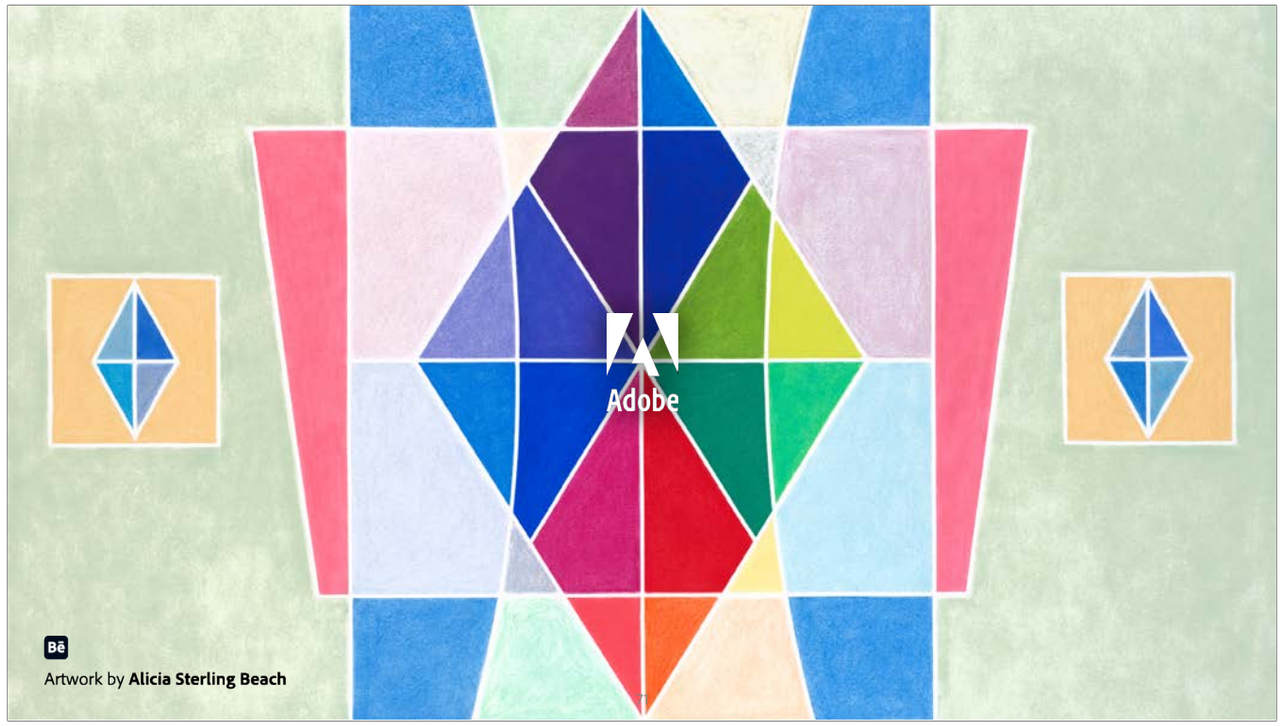
Alex Stepanov is the creator of the "STL" - the standard library of algorithms and data structures in C++, which was (and is) revolutionary.

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.




Adobe
Artwork by Alicia Sterling Beach