CSE 331

Reasoning About Array Indexing

James Wilcox and Kevin Zatloukal

Indexing

$$\text{at} : (\text{List}, \mathbb{N}) \to \mathbb{Z}$$

$$\begin{aligned} \text{at}(x :: L\,, 0) &:= x \\ \text{at}(x :: L\,, n{+}1) &:= \text{at}(L, n) \end{aligned}$$

- **Retrieve an element of the list by index**

- **For example:**

$$\begin{aligned} &\text{at}(1 :: 2 :: 3 :: 4 :: \text{nil}, 2) \\ &= \text{at}(2 :: 3 :: 4 :: \text{nil}, 1) && \textbf{def of } \text{at} \\ &= \text{at}(3 :: 4 :: \text{nil}, 0) && \textbf{def of } \text{at} \\ &= 3 && \textbf{def of } \text{at} \end{aligned}$$

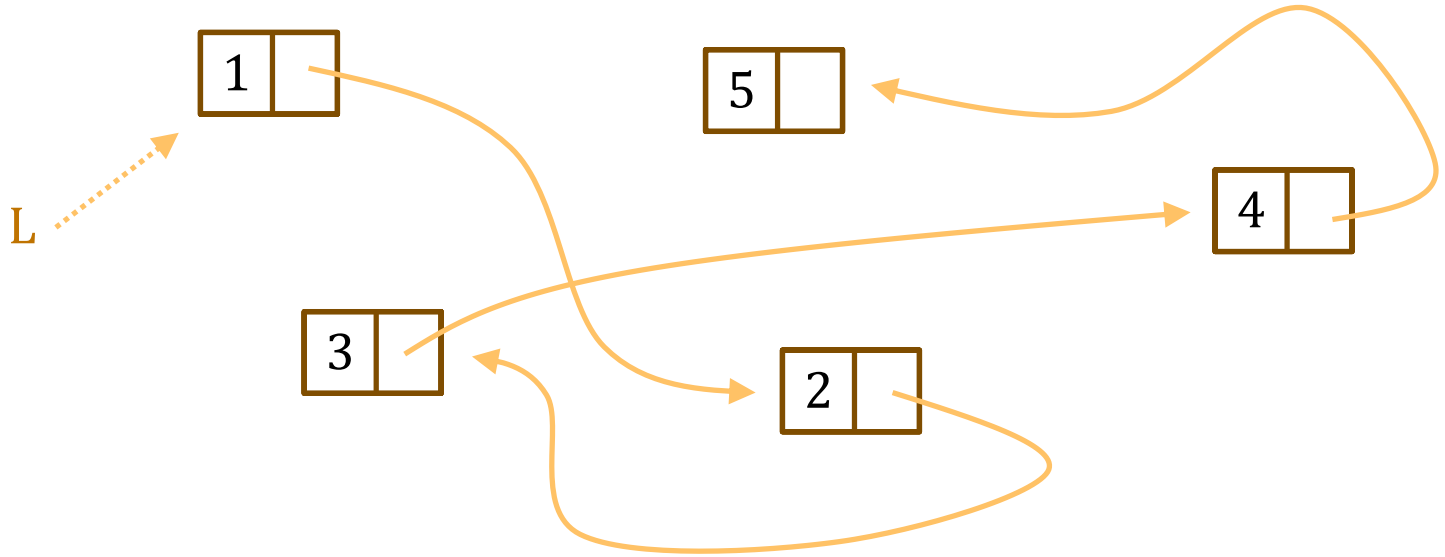
Indexing

$$\text{at} : (\text{List}, \mathbb{N}) \to \mathbb{Z}$$

$$\begin{aligned} \text{at}(x :: L\,, 0) &:= x \\ \text{at}(x :: L\,, n{+}1) &:= \text{at}(L, n) \end{aligned}$$

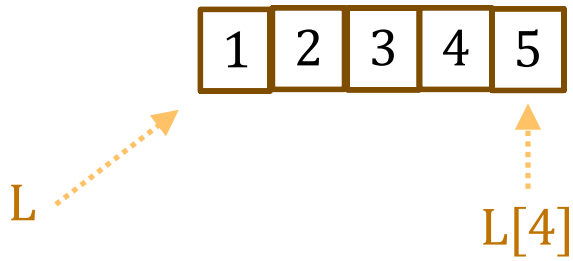
- **Use "$L[j]$" as an abbreviation for $\text{at}(j, L)$**
- **Not an efficient operation on lists…**

Linked Lists in Memory



- **Must follow the "next" pointers to find elements**
  - $\mathrm{at}(\mathrm{L}, \mathrm{n})$ **is an** $\mathrm{O}(\mathrm{n})$ **operation**
  - **no faster way to do this**

Faster Implementation of at

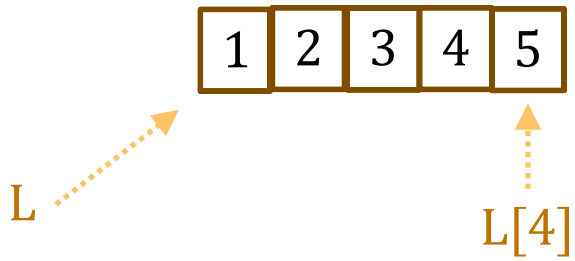


- Alternative: store the elements next to each other
  - can find the n-th entry by arithmetic:

$$\text{location of L[4]} = (\text{location of L}) + 4 * \text{sizeof(data)}$$

- Resulting data structure is an **array**

Faster Implementation of `at`



- Resulting data structure is an **array**
- **Efficient** to read `L[i]`
- **Inefficient** to...
  - insert elements anywhere but the end
  - write operations with an immutable ADT
  - trees can do <u>all of this</u> in $O(\log n)$ time

Access By Index

- **Easily access both $L[0]$ and $L[n\text{-}1]$, where $n = \text{len}(L)$**
  - **can process a list in either direction**

- **"With great power, comes great responsibility"**

  — the Peter Parker Principle

- **Whenever we write "$A[j]$", we must check $0 \leq j < n$**
  - **new bug just dropped!**

    with list, we only need to worry about nil and non-nil

    once we know L is non-nil, we know L.hd exists
  - **only checked at runtime in Java (no type checking)**

    TypeScript doesn't check it at all!!

Reasoning Toolkit

Description	Testing	Tools	Reasoning
no mutation	coverage	type checking	calculation induction
local variable mutation	“	“	Floyd logic
heap state mutation	“	“	rep invariants
array mutation	“	“	sublists for-any facts

Suffixes

$$\text{suffix} : (\text{List}, \mathbb{N}) \rightarrow \text{List}$$

$$\begin{aligned} \text{suffix}(L\ ,\ 0) &:= L \\ \text{suffix}(x :: L\ ,\ n{+}1) &:= \text{suffix}(L,\ n) \end{aligned}$$

everything from here on

not there yet

- **For example:**

$$\begin{aligned} &\text{suffix}(1 :: 2 :: 3 :: 4 :: \text{nil},\ 2) \\ &= \text{suffix}(2 :: 3 :: 4 :: \text{nil},\ 1) && \textbf{def of } \text{suffix} \\ &= \text{suffix}(3 :: 4 :: \text{nil},\ 0) && \textbf{def of } \text{suffix} \\ &= 3 :: 4 :: \text{nil} && \textbf{def of } \text{suffix} \end{aligned}$$

  - **will use "**$L[j\ ..]$**" as an abbreviation for** $\text{suffix}(L,\ j)$
  - $\text{suffix}(L,\ j)$ **is undefined if** $\text{len}(L) < j$

Prefixes

$$\text{prefix} : (\text{List}, \mathbb{N}) \to \text{List}$$

$$\begin{array}{lll} \text{prefix}(L\,, 0) & := \text{nil} & \text{nothing more} \\ \text{prefix}(x :: L\,, n{+}1) & := x :: \text{prefix}(L, n) & x \text{ is in the prefix} \end{array}$$

- **For example:**

$$\begin{array}{ll} \text{prefix}(1 :: 2 :: 3 :: 4 :: \text{nil}, 2) & \\ \quad = 1 :: \text{prefix}(2 :: 3 :: 4 :: \text{nil}, 1) & \textbf{def of } \text{prefix} \\ \quad = 1 :: 2 :: \text{prefix}(3 :: 4 :: \text{nil}, 0) & \textbf{def of } \text{prefix} \\ \quad = 1 :: 2 :: \text{nil} & \textbf{def of } \text{prefix} \end{array}$$

  - **will use "$L[..\,j]$" as an abbreviation for** $\text{prefix}(L, j{+}1)$
  - **note that** $L[..\,j]$ **includes** j **while** $\text{prefix}(L, j)$ **does <u>not</u>**

Prefixes & Suffixes

suffix(L , 0)	:= L	prefix(L , 0)	:= nil
suffix(x :: L , n+1)	:= suffix(L, n)	prefix(x :: L , n+1)	:= x :: prefix(L, n)

- **Can show that** $\text{prefix}(L, j) +\!\!+ \text{suffix}(L, j) = L$

Base Case: $\text{prefix}(L, 0) +\!\!+ \text{suffix}(L, 0)$

$= \text{nil} +\!\!+ \text{suffix}(L, 0)$ **def of** prefix

$= \text{nil} +\!\!+ L$ **def of** suffix

$= L$

Inductive Step: $\text{prefix}(x :: L, n+1) +\!\!+ \text{suffix}(x :: L, n+1)$

$= (x :: \text{prefix}(L, n)) +\!\!+ \text{suffix}(x :: L, n+1)$ **def of** prefix

$= (x :: \text{prefix}(L, n)) +\!\!+ \text{suffix}(L, n)$ **def of** suffix

$= x :: (\text{prefix}(L, n) +\!\!+ \text{suffix}(L, n))$ **def of** concat

$= x :: L$ Ind. Hyp.

Prefixes & Suffixes

suffix(L , 0)	:= L	prefix(L , 0)	:= nil
suffix(x :: L , n+1)	:= suffix(L, n)	prefix(x :: L , n+1)	:= x :: prefix(L, n)

- Can show that $\text{prefix}(L, j) +\!\!+ \text{suffix}(L, j) = L$

- In shorthand, we have $L[..\ j\text{-}1] +\!\!+ L[j\ ..] = L$
  - notation is inclusive of both indexes
  - hence, $L[..\ 0] = [L[0]]$ but $L[..\ \text{-}1] = \text{nil}$

Prefixes & Suffixes

suffix(L , 0)	:= L	prefix(L , 0)	:= nil
suffix(x :: L , n+1)	:= suffix(L, n)	prefix(x :: L , n+1)	:= x :: prefix(L, n)

- In shorthand, we have $L[..\ j\text{-}1] \mathbin{+\!\!+} L[j\ ..] = L$
  - notation is inclusive of both indexes

- More useful facts about sublists:

	Suffix	Prefix
Empty	$L[\text{len}(L)\ ..] = \text{nil}$	$L[..\ -1] = \text{nil}$
Add One	$[L[j]] \mathbin{+\!\!+} L[j+1\ ..] = L[j\ ..]$	$L[..\ j] \mathbin{+\!\!+} [L[j+1]] = L[..\ j+1]$

  - we will use these without explanation going forward…

Loop Invariants With Indexing

Recall: Loop Correctness Example 2 (Topic 4)

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- **This loop claims to calculate it as well:**

```
public int sum(List L) {
  int s = 0;
  {{ Inv: sum(L0) = s + sum(L) }}
  while (L != null) {
    s = s + L.hd;
    L = L.tl;
  }
  {{ s = sum(L0) }}
  return s;
}
```

Change to a version that uses indexes...

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: ... }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Note that A does not change, just i and s

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[.. i–1]) and i ≤ len(A) }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Important to know if A[i] is in the sum or not

Let's prove this correct!

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ ____________________ }}
  {{ Inv: s = sum(A[.. i-1]) and i ≤ len(A) }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ i = 0 and s = 0 }}
  {{ Inv: s = sum(A[.. i–1]) and i ≤ len(A) }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

$i = 0 \leq \text{len}(A)$

$\text{sum}(A[..\ i\text{-}1])$
$= \text{sum}(A[..\ 0\text{-}1])$ **since** $i = 0$
$= \text{sum}(A[..\ -1])$
$= \text{sum}(\text{prefix}(A, 0))$
$= \text{sum}(\text{nil})$ **def of** prefix
$= 0$
$= s$

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[.. i-1]) and i ≤ len(A) }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ ____________________ }}
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[.. i–1]) and i ≤ len(A) }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A[.. i-1]) and i = len(A)}}
  {{ s = sum(A) }}
  return s;
}
```

sum(A[.. i-1])
= sum(A[.. len(A)-1])
= sum(A)

since i = len(A)

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[.. i–1]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[.. i–1]) and i < len(A) }}
    s = s + A[i];
    i = i + 1;
    {{ s = sum(A[.. i–1]) and i ≤ len(A) }}
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[.. i–1]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[.. i–1]) and i < len(A) }}
    s = s + A[i];
    {{ _____ }}
    i = i + 1;
    {{ s = sum(A[.. i–1]) and i ≤ len(A) }}
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[.. i–1]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[.. i–1]) and i < len(A) }}
    s = s + A[i];
    {{ s = sum(A[.. i]) and i + 1 ≤ len(A) }}
    i = i + 1;
    {{ s = sum(A[.. i–1]) and i ≤ len(A) }}
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[.. i–1]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[.. i–1]) and i < len(A) }}
    {{ _____ }}
    s = s + A[i];
    {{ s = sum(A[.. i]) and i + 1 ≤ len(A) }}
    i = i + 1;
    {{ s = sum(A[.. i–1]) and i ≤ len(A) }}
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[.. i–1]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[.. i–1]) and i < len(A) }}
    {{ s + A[i] = sum(A[.. i]) and i + 1 ≤ len(A) }}
    s = s + A[i];
    {{ s = sum(A[.. i]) and i + 1 ≤ len(A) }}
    i = i + 1;
    {{ s = sum(A[.. i–1]) and i ≤ len(A) }}
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

{{ s = sum(A[.. i–1]) and i < len(A) }}

{{ s + A[i] = sum(A[.. i]) and i + 1 ≤ len(A) }}

i ≤ len(A) – 1 **or equivalently**

i + 1 ≤ len(A)

s + A[i] = sum(A[.. i-1]) + A[i] **since** s = sum(A[.. i-1])

= sum(A[.. i-1]) + sum([A[i]]) **def of** sum

Recall: Length of Concatenated Lists

```
concat(nil, R)      := R
concat(x :: L, R)   := x :: concat(L, R))
```

- **Proved that** $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$
  - **prove by induction on** S
  - **prove the claim for any choice of** R **(i.e.,** R **is a variable)**

- **I.e., that** $\text{len}(S +\!\!+ R) = \text{len}(S) + \text{len}(R)$

- **Can easily prove that** $\text{sum}(S +\!\!+ R) = \text{sum}(S) + \text{sum}(R)$
  - **follows by induction on** S

Example 1: Sum of an Array

{{ s = sum(A[.. i–1]) and i < len(A) }}

{{ s + A[i] = sum(A[.. i]) and i + 1 ≤ len(A) }}

i ≤ len(A) – 1 **or equivalently**
i + 1 ≤ len(A)

s + A[i]	= sum(A[.. i-1]) + A[i]	**since** s = sum(A[.. i-1])
	= sum(A[.. i-1]) + sum([A[i]])	**def of** sum
	= sum(A[.. i-1] ⧺ [A[i]])	**by Lemma 1**
	= sum(A[.. i])	

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[.. i–1]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[.. i–1]) and i < len(A) }}
    {{ s + A[i] = sum(A[.. i]) and i + 1 ≤ len(A) }}
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Done!

Example 2: Max of an Array

- Can define the maximum of a list as follows:

$$\max : (\text{List}) \to \mathbb{Z}$$

$$\begin{aligned}
&\max(x :: \text{nil}) && := x && \\
&\max(x :: y :: L) && := x && \text{if } x > \max(y :: L) \\
&\max(x :: y :: L) && := \max(y :: L) && \text{if } x \leq \max(y :: L)
\end{aligned}$$

  - compares x to the maximum of the rest $(y :: L)$

- Note that this is undefined on nil

Example 2: Max of an Array

- Can implement this with the following loop

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: ... }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Should this by i >= 0?

Should this by A[i]?

Clear that up with the invariant

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ _____ }}
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ len(A) > 0 and i = len(A) – 1 and m = A[i] }}
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Does this hold?

Yes, but only because $0 < \text{len}(A)$

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ len(A) > 0 and i = len(A) – 1 and m = A[i] }}
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

max(A[i ..])
= max(A[len(A) – 1 ..]) **since** i = len(A) – 1
= max([A[len(A) – 1]])
= A[len(A) – 1] **def of** max
= m

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ _____ }}
  {{ m = max(A) }}
  return m;
}
```

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A[i ..]) and i = 0 }}
  {{ m = max(A) }}
  return m;
}
```

m = max(A[i ..])
= max(A[0 ..]) since i = 0
= max(A)

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i ..]) and 0 < i < len(A) }}
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
    {{ m = max(A[i ..]) and 0 ≤ i < len(A) }}
  }
  {{ m = max(A) }}
  return m;
}
```

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i ..]) and 0 < i < len(A) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ ____________________________________ }}
    i = i - 1;
    {{ m = max(A[i ..]) and 0 ≤ i < len(A) }}
  }
  {{ m = max(A) }}
  return m;
}
```

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i ..]) and 0 < i < len(A) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ m = max(A[i-1 ..]) and 0 ≤ i-1 < len(A) }}
    i = i - 1;
    {{ m = max(A[i ..]) and 0 ≤ i < len(A) }}
  }
  {{ m = max(A) }}
  return m;
}
```

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i ..]) and 0 < i < len(A) }}
    {{ ...                    and 0 ≤ i-1 < len(A) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ m = max(A[i-1 ..]) and 0 ≤ i-1 < len(A) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

no changes to i

Does this hold?

Yes, but check both parts!

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i ..]) and 0 < i < len(A) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ m = max(A[i-1 ..]) and 0 ≤ i-1 < len(A) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Need to check other part
(Won't need facts about "i" anymore.)

Let's reason over paths...

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i ..]) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ _____ }}
    {{ m = max(A[i-1 ..]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Case "else" (skip if)

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i ..]) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ m = max(A[i ..]) and A[i-1] < m }}
    {{ m = max(A[i-1 ..]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Case "else" (skip if)

A[i-1] < m = max(A[i..])

max(A[i-1 ..])
= max(A[i-1] :: A[i ..])
= max(A[i ..]) **def of** max
(**since** A[i] < max(A[i..]))
= m

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i ..]) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ _____ }}
    {{ m = max(A[i-1 ..]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Case "then" (enter "if")

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i ..]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i ..]) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ m0 = max(A[i ..]) and A[i-1] ≥ m0 and m = A[i-1] }}
    {{ m = max(A[i-1 ..]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

$A[i-1] \geq m_0 = \max(A[i..])$

$\max(A[i-1 ..])$
$= \max(A[i-1] :: A[i ..])$
$= A[i-1]$ **def of** max
(**since** $A[i] \geq \max(A[i..])$)
$= m$

Takeaways

- Reasoning about arrays requires new tools
  - prefixes and suffixes now (more later)

- Trying move faster…
- But the tools still work!
  - can return to them when it gets tricky
  - always fine to slow down and check carefully

- The tools help you know what needs checking

From Invariant To Code

Writing Loops

- Examples so far have been code reviews
  - checking correctness of given code

- Steps to write a loop to solve a problem:
  1. Come up with an idea for the loop
  2. Formalize the idea in the invariant
  3. Write the code so that it is correct with that invariant

- Let's see some examples...

Recall Example 1: Sum of an Array

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[.. i-1]) and i ≤ len(A) }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = _______
  int s = 0;
  {{ Inv: s = sum(A[.. i]) }}
  while (_________________________) {

    ___________
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Idea: add from front to back, but now i is the last index included in the sum

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = _______
  int s = 0;
  {{ Inv: s = sum(A[.. i]) }}
  while (_________________________) {

    ___________
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we initialize i so that $\text{sum}(A[..\,i]) = 0$**?**

Choose i **so that** $A[..\,i] = \text{nil}$ **since** $\text{sum}(\text{nil}) = 0$

We need $i = -1$ **since** $A[..\,-1] = \text{nil}$

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[.. i]) }}
  while (________________________) {

    ___________
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[.. i]) }}
  while (_____________________) {

    __________
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we exit so that $\text{sum}(A[..\,i]) = \text{sum}(A)$**?**

Choose exit so that $A[..\,i] = A$

We need $i = \text{len}(A) - 1$

Loop while $i < \text{len}(A) - 1$

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[.. i]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {

    __________
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Can now fill in the range of values for i.
(Important to avoid array out-of-bounds.)

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[.. i]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {

    _____

    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we update s?

Useful to do one step of Floyd logic...

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[.. i]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {
    {{ s = sum(A[.. i]) }}
    ------------------
    {{ s = sum(A[.. i+1]) }}
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we change s to go from sum(A[.. i]) to sum(A[.. i+1])?

Need to add in A[i+1] (Not A[i]!)

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[.. i]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {
    {{ s = sum(A[.. i]) }}
    s = s + A[i+1];
    {{ s = sum(A[.. i+1]) }}
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do know this is in bounds?

Let's fill in our information about i...

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[.. i]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {
    {{ s = sum(A[.. i]) and -1 ≤ i < len(A) - 1 }}
    s = s + A[i+1];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Why is that what we know?

Get $-1 \leq i$ from Inv and $i < len(A) - 1$ from the loop condition.

How do know $A[i+1]$ is in bounds?

$-1 \leq i < len(A) - 1$ implies $0 \leq i+1 < len(A)$

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[.. i]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {
    s = s + A[i+1];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Done!

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = _______
  int s = 0;
  {{ Inv: s = sum(A[i.. ]) }}
  while (_________________________) {

    ___________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Idea: add from back to front, with i included in the sum

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = _______
  int s = 0;
  {{ Inv: s = sum(A[i.. ]) }}
  while (_________________________) {

    ___________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we initialize i so that $\text{sum}(A[i..]) = 0$**?**

Choose i **so that** $A[i..] = \text{nil}$ **since** $\text{sum}(\text{nil}) = 0$

We need $i = \text{len}(A)$ **since** $A[\text{len}(A)..] = \text{nil}$

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i.. ]) }}
  while (_____________________) {

    __________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i.. ]) }}
  while (_____________________) {

    __________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we exit so that $\text{sum}(A[i..]) = \text{sum}(A)$**?**

Choose exit so that $A[i..] = A$

We need $i = 0$

Loop while $i > 0$

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i.. ]) }}
  while (i > 0) {

    ___________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i.. ]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {

    ___________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

This is the range of values for i.
(Note that 0 is included!)

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i.. ]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {

    ____________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we update s?

Let's do one step Floyd logic...

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i.. ]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    {{ s = sum(A[i.. ]) }}
    ____________
    {{ ______________ }}
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

What goes here?

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i.. ]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    {{ s = sum(A[i.. ]) }}
    ____________
    {{ s = sum(A[i-1.. ]) }}
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we change s to go from sum(A[i..]) to sum(A[i-1..])?

Need to add in A[i-1] (Not A[i]!)

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i.. ]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    {{ s = sum(A[i.. ]) }}
    s = s + A[i-1];
    {{ s = sum(A[i-1.. ]) }}
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Is this array access in bounds?

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i.. ]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    {{ s = sum(A[i.. ]) and 0 < i ≤ len(A) }}
    s = s + A[i-1];
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Why is this what we get from Floyd logic?

Why does this tell us the array access is within bounds?

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i..]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    s = s + A[i-1];
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Done!

Recall: Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = …;
  int m = …;
  {{ Inv: m = max(A[i ..]) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = _____________
  int m = _____________
  {{ Inv: m = max(A[i+1 ..]) }}
  while (_____________) {

    ___________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Idea: work from back to front, with i not included in value of m

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = _____________
  int m = _____________
  {{ Inv: m = max(A[i+1 ..]) }}
  while (_____________) {

    ___________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

What is the easiest way to make this hold initially?

Remember that max is undefined on nil

Easiest is $m = max([A[len(A)-1]])$

How do we initialize i so that $m = max(A[i+1..])$

We need $A[i+1..] = [A[len(A)-1]]$

Must take $i = len(A)-2$

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1 ..]) }}
  while (________________) {

    _____________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1 ..]) }}
  while (_____________) {

    ___________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

How do we exit so that $\max(A[i+1\ ..]) = \max(A)$**?**

Choose exit so that $A[i+1..\ ] = A$

We need $i = -1$

Loop while $i > -1$ **or in other words** $i \geq 0$

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1 ..]) }}
  while (i >= 0) {

    ____________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Can now fill in the range of i

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1 ..]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {

    ____________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1 ..]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {

    ____________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

How do we update s?

Let's do one step Floyd logic...

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1 ..]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    {{ m = max(A[i+1 ..]) }}
    ____________
    {{ _____________ }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

What goes here?

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1 ..]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    {{ m = max(A[i+1 ..]) }}
    ------------------------
    {{ m = max(A[i ..]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

How do we implement this?

How do we change s to go from max(A[i+1..]) to max(A[i..])?

Need to look at A[i]

If A[i] > m, then A[i] is new max

Otherwise, m is still the max

max(x :: nil) := x

max(x :: y :: L) := x if x > max(y :: L)

max(x :: y :: L) := max(y :: L) if x ≤ max(y :: L)

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1 ..]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    {{ m = max(A[i+1 ..]) }}
    if (A[i] > m)
      m = A[i];
    {{ m = max(A[i ..]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1 ..]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    {{ m = max(A[i+1 ..]) }}
    if (A[i] > m)
      m = A[i];
    {{ m = max(A[i ..]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Are we sure this is in bounds?

Let's reason forward to check...

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1 ..]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    if (A[i] > m)
      {{ m = max(A[i+1 ..]) and 0 ≤ i ≤ len(A)-2 }}
      m = A[i];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Why is that what we get?

Why does that tell us this is okay?

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1 ..]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    if (A[i] > m)
      m = A[i];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Done!

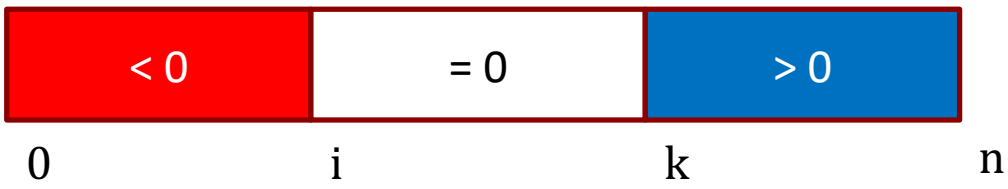
Example 7: Sorting Negative, Zero, Positive

- Reorder an array so that
  - negative numbers come first, then zeros, then positives
  - famous coding interview question (decades ago)

```
/**
 * Reorders A into negatives, then 0s, then positive
 * @modifies A
 * @effects leaves same integers in A but with
 *   A[j] < 0 for 0 <= j < i
 *   A[j] = 0 for i <= j < k
 *   A[j] > 0 for k <= j < n
 * @returns the indexes (i, k) above
 */
public static int[] sortPosNeg(int[] A)
```

Example 7: Sorting Negative, Zero, Positive

```
// @effects leaves same numbers in A but with
//   A[j] < 0 for 0 <= j < i
//   A[j] = 0 for i <= j < k
//   A[j] > 0 for k <= j < n
```



Let's implement this...

– what is the *idea* of our loop?

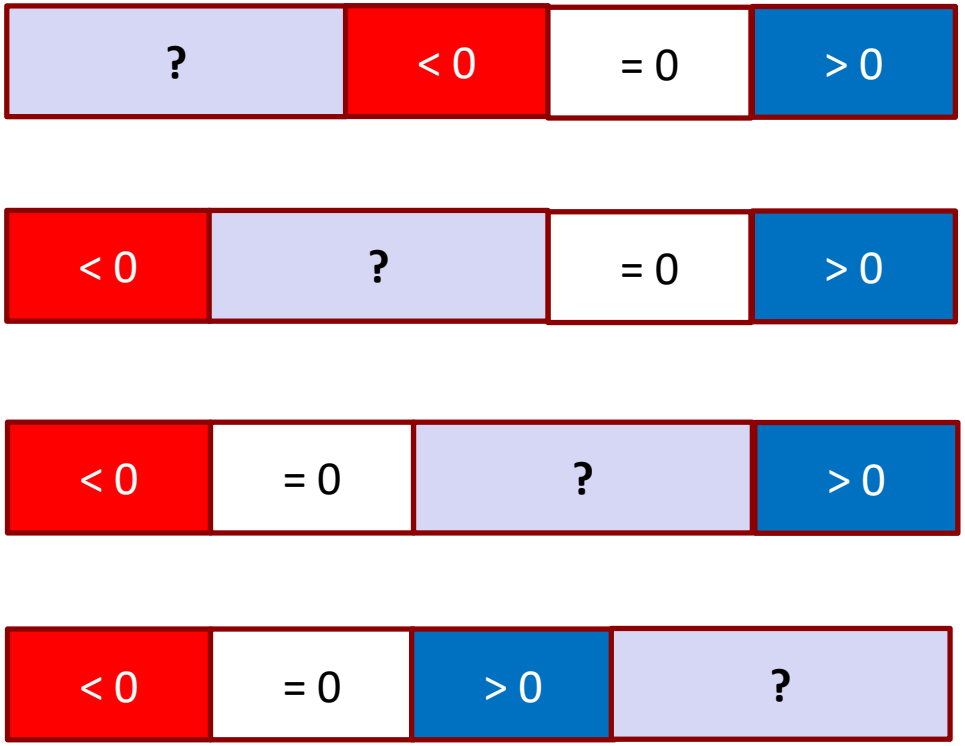
how do we represent partial progress in our invariant?

Example 7: Sorting Negative, Zero, Positive

How do we represent partial progress in our invariant?

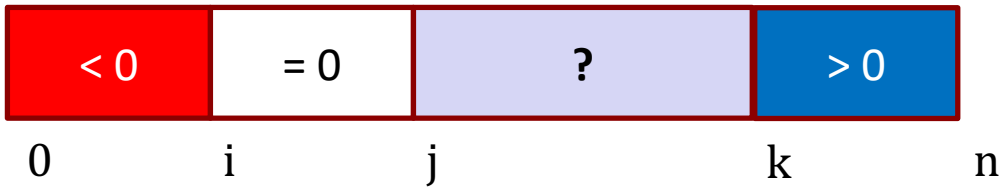
- **needs allow elements with *unknown* values**

  initially, we don't know anything about the array values



Example 7: Sorting Negative, Zero, Positive

Our Invariant:



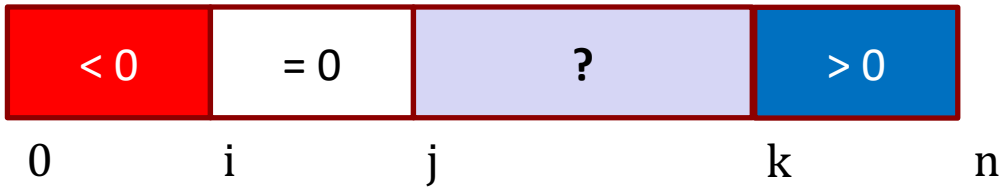
$A[\ell] < 0$ for any $0 \leq \ell < i$

$A[\ell] = 0$ for any $i \leq \ell < j$

(no constraints on $A[\ell]$ for $j \leq \ell < k$)

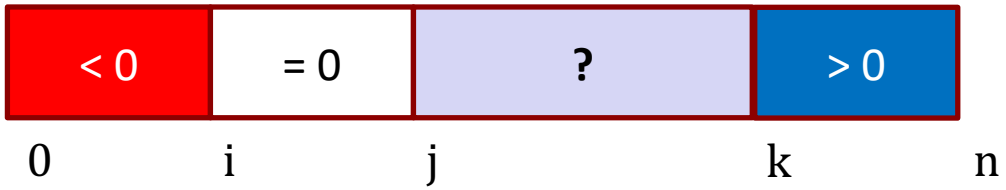
$A[\ell] > 0$ for any $k \leq \ell < n$

Example 7: Sorting Negative, Zero, Positive

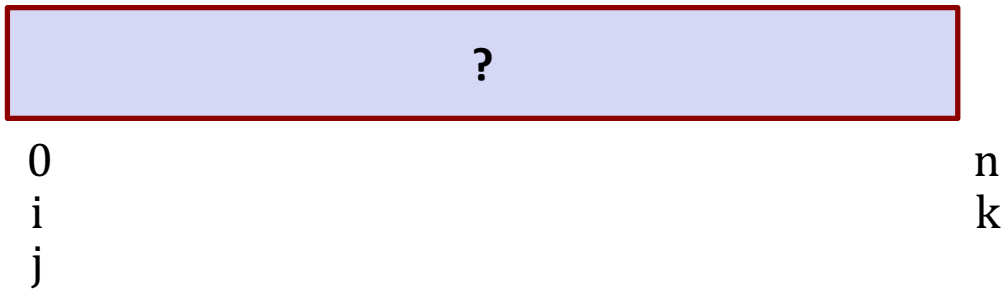


- Let's try figuring out the code to make it correct
- Figure out the code for
  - how to initialize
  - when to exit
  - loop body

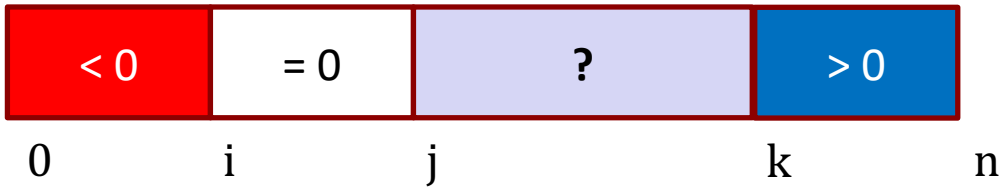
Example 7: Sorting Negative, Zero, Positive



- Will have variables i, j, and k with $i \leq j \leq k$
- How do we set these to make it true initially?
  - we start out not knowing anything about the array values
  - set $i = j = 0$ and $k = n$

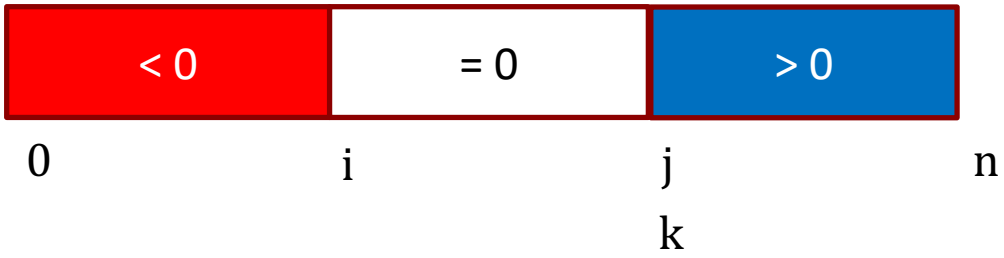


Example 7: Sorting Negative, Zero, Positive



- **Set $i = j = 0$ and $k = n$ to make this hold initially**

- **When do we exit?**
  - **purple is empty if $j = k$**



Sort Positive, Zero, Negative

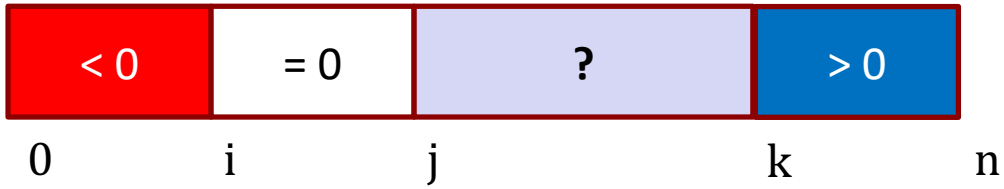
```
int i = 0;
int j = 0;
int k = A.length;
```

{{ **Inv**: $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $0 \le i \le j \le k \le n$}}

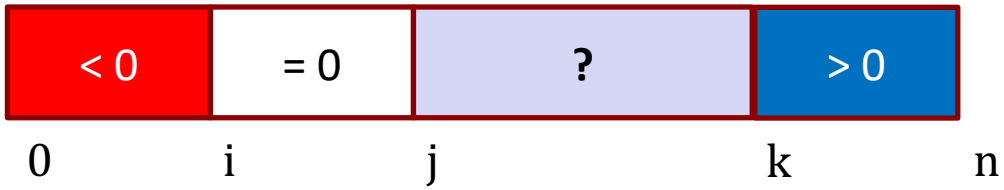
```
while (j < k) {
   ...
}
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < \mathbf{k}$
$A[\ell] > 0$ for any $k \le \ell < n$ }}

```
return new int[] {i, j};
```

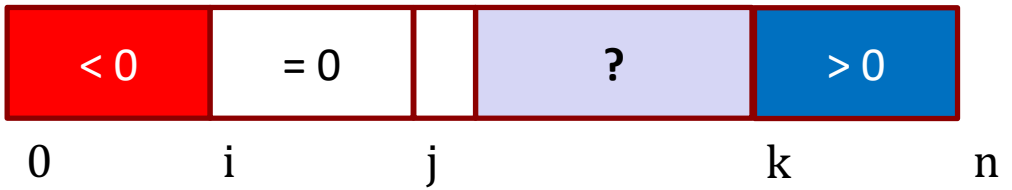


Example 7: Sorting Negative, Zero, Positive

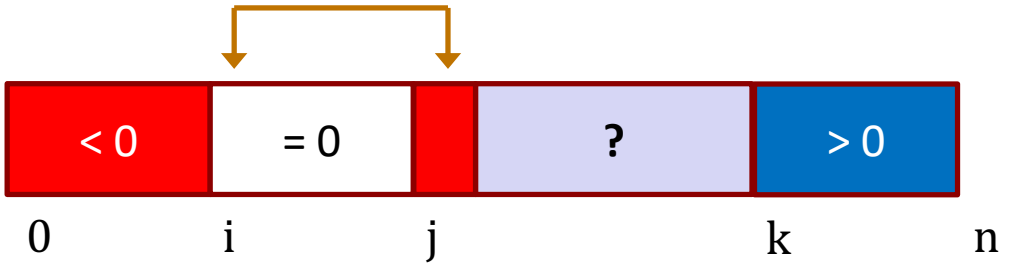


- How do we make progress?
  - try to increase j by 1 or decrease k by 1
- Look at $A[j]$ and figure out where it goes
- What to do depends on $A[j]$
  - could be < 0, $= 0$, or > 0

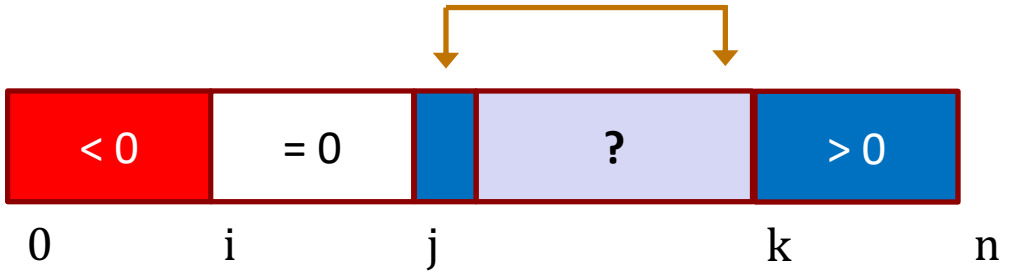
Example 7: Sorting Negative, Zero, Positive



Set $j = j_0 + 1$



Swap $A[i]$ **and** $A[j]$
Set $i = i_0 + 1$
and $j = j_0 + 1$



Swap $A[j]$ **and** $A[k-1]$
Set $k = k_0 - 1$

Example 7: Sort Positive, Zero, Negative

{{ **Inv**: $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $0 \le i \le j \le k \le n$ }}

```
while (j != k) {
  if (A[j] == 0) {
    j = j + 1;
  } else if (A[j] < 0) {
    swap(A, i, j);
    i = i + 1;
    j = j + 1;
  } else {
    swap(A, j, k-1);
    k = k - 1;
  }
}
```

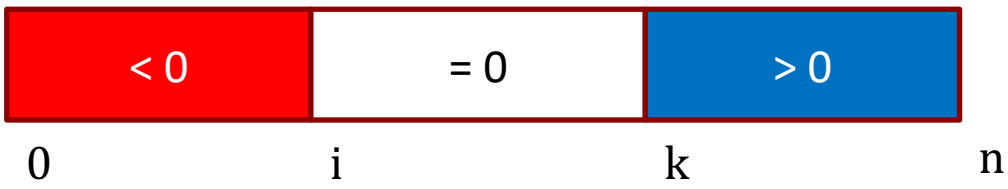
Recall: Reasoning Toolkit

Description	Testing	Tools	Reasoning
no mutation	coverage	type checking	calculation induction
local variable mutation	"	"	Floyd logic
heap state mutation	"	"	rep invariants
array mutation	"	"	sublists for-any facts

"For Any" Facts

Recall: Sorting Negative, Zero, Positive

```
// @effects leaves same numbers in A but with
//   A[j] < 0 for any 0 <= j < i
//   A[j] = 0 for any i <= j < k
//   A[j] > 0 for any k <= j < n
```



- could write first region as negative(A[.. i-1]) and last region as positive(A[k ..])
- but middle region is not a prefix or suffix

"For Any" Facts

- Necessary facts are not always prefix / suffix
  - may have facts about arbitrary parts of the array

- For example, what does "A is sorted" mean?
  - we have $A[0] < A[1] < \ldots < A[n\text{-}1]$ or more formally:

$$A[j] < A[j+1] \text{ for any } 0 \leq j \leq len(A) - 2$$

- We will call these "for any" facts
  - they are more cumbersome than simple equations
  - they will show up inside individual assertions now!

    "with great power comes great responsibility"…

Example 8: Binary Search

- Binary search looks for a number in a sorted array
  - return true if $A[i] = y$ for some i

```
// @requires A[i] < A[i+1] for any 0 <= i < len(A)-1
// @returns false if A[i] /= y for any 0 <= i < len(A)
//          true  otherwise
public boolean bsearch(int[] A, int y) { … }
```

Example 8: Binary Search

- Invariant of binary search has "for any" facts

```
public boolean bsearch(int[] A, int y) { // req A sorted
  int j = 0;
  int k = A.length;
  {{ Inv: (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  k ≤ i < len(A)) }}
  while (j < k) {
    int m = (j + k) / 2;
    if (y < A[m]) {
      k = m;
    } else if (A[m] < y) {
      j = m + 1;
    } else {
      return true;
    }
  }
  return false;
}
```

Why the asymmetry?

Example 8: Binary Search

- Invariant of binary search has "for any" facts

```
public boolean bsearch(int[] A, int y) { // req A sorted
  int j = 0;
  int k = A.length;
  {{ j = 0 and k = len(A) }}
  {{ Inv: (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  k ≤ i < len(A)) }}
  while (j < k) {
    int m = (j + k) / 2;
    if (y < A[m]) {
      k = m;
    } else if (A[m] < y) {
      j = m + 1;
    } else {
      return true;
    }
  }
  return false;
}
```

No integer i satisfies $0 \le i < 0$

No integer i satisfies $\text{len}(A) \le i < \text{len}(A)$

Example 8: Binary Search

- Invariant of binary search has "for any" facts

{{ **Inv**: (A[i] < y for any 0 ≤ i < j) and (y < A[i] for any k ≤ i < len(A)) }}

```
while (j < k) {
  int m = (j + k) / 2;
  if (y < A[m]) {
    k = m;
  } else if (A[m] < y) {
    j = m + 1;
  } else {
    return true;
  }
}
```

{{ Inv and j = k }}

{{ A[i] ≠ y for any 0 ≤ i < len(A) }}

```
return false;
}
```

Every index i < j is smaller than y
Every index k ≤ i is larger than y

Example 8: Binary Search

{{ **Inv**: (A[i] < y for any 0 ≤ i < j) and (y < A[i] for any k ≤ i < len(A)) }}

```
while (j < k) {
  int m = (j + k) / 2;
  if (y < A[m]) {
    k = m;
  } else if (A[m] < y) {
    j = m + 1;
  } else {
    return true;
  }
}
return false;
}
```

Work through cases...

Example 8: Binary Search

```
{{ Inv: (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  k ≤ i < len(A)) }}
while (j < k) {
  int m = (j + k) / 2;
  if (y < A[m]) {
    {{ Inv and y < A[m] }}
    k = m;
  } else if (A[m] < y) {
    {{ Inv and A[m] < y }}
    j = m + 1;
  } else {
    {{ Inv and A[m] = y }}
    return true;
  }
}
return false;
}
```

Example 8: Binary Search

```
{{ Inv: (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  k ≤ i < len(A)) }}
while (j < k) {
  int m = (j + k) / 2;
  if (y < A[m]) {
    {{ Inv and y < A[m] }}
    k = m;
  } else if (A[m] < y) {
    {{ Inv and A[m] < y }}
    j = m + 1;
  } else {
    {{ Inv and A[m] = y }}
    {{ A[i] = y for some i }}
    return true;
  }
}
return false;
}
```

True. Take i = m

Example 8: Binary Search

```
{{ Inv: (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  k ≤ i < len(A)) }}
while (j < k) {
  int m = (j + k) / 2;
  if (y < A[m]) {
    {{ Inv and y < A[m] }}
    {{ (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  m ≤ i < len(A)) }}
    k = m;
  } else if (A[m] < y) {
    {{ Inv and A[m] < y }}
    j = m + 1;
  } else {
    return true;
  }
}
return false;
}
```

Left part is known from Inv.

We know that $y < A[m]$ holds.

Why do we know $y < A[i]$ holds for $m < i$?

Holds because A is sorted.

Example 8: Binary Search

```
{{ Inv: (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  k ≤ i < len(A)) }}
while (j < k) {
  int m = (j + k) / 2;
  if (y < A[m]) {
    {{ Inv and y < A[m] }}
    k = m;
  } else if (A[m] < y) {
    {{ Inv and A[m] < y }}
    {{ (A[i] < y  for any  0 ≤ i < m+1) and (y < A[i]  for any  k ≤ i < len(A)) }}
    j = m + 1;
  } else {
    return true;
  }
}
return false;
}
```

Right part is known from Inv.

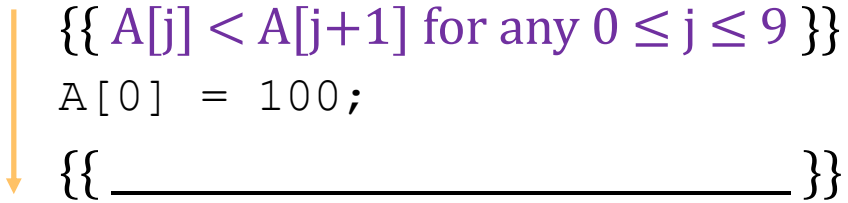
We know that $y < A[m]$ holds.

Have $y < A[i]$ holds for $i < m$ since A is sorted.

Mutation of Arrays

Mutation With For-Any Facts

- Mutation makes "for any" facts change
  - this can happen line by line!
  - working forward:



Mutation With For-Any Facts

- Mutation makes "for any" facts change
  - this can happen line by line!
  - working forward:

{{ $A[j] < A[j+1]$ for any $0 \leq j \leq 9$ }}

```
A[0] = 100;
```

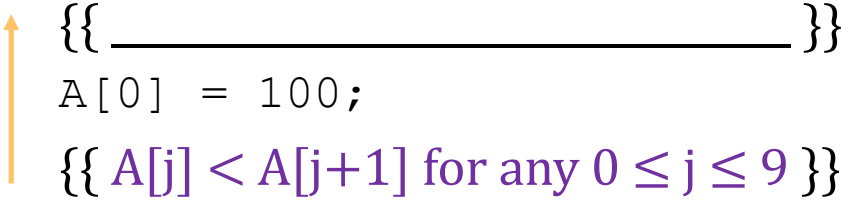
{{ $(A[j] < A[j+1]$ for any $1 \leq j \leq 9)$ and $A[0] = 100$ }}

  - old facts about A[0] are invalidated

    if we wanted to, we could note that $A_0[0] \leq A[1]$

Mutation With For-Any Facts

- Mutation makes "for any" facts change
  - this can happen line by line!
  - working backward:



Mutation With For-Any Facts

- Mutation makes "for any" facts change
  - this can happen line by line!
  - working backward:

{{ $100 < A[1]$ and $(A[j] < A[j+1]$ for any $1 \le j \le 9$ }}

```
A[0] = 100;
```

{{ $A[j] < A[j+1]$ for any $0 \le j \le 9$ }}

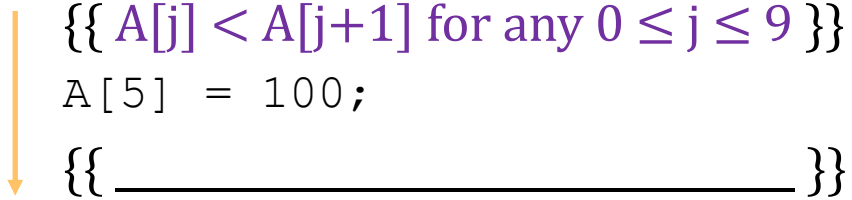
  - just substitution as usual

    but be careful about finding everywhere $A[0]$ appears

Mutation With For-Any Facts

- Mutation makes "for any" facts change
  - this can happen line by line!

- Easiest if we mutate the end value in the range
  - but other mutations are possible
  - working forward:



Mutation With For-Any Facts

- Mutation makes "for any" facts change
  - this can happen line by line!

- Easiest if we mutate the end value in the range
  - but other mutations are possible
  - working forward:

{{ $A[j] < A[j+1]$ for any $0 \le j \le 9$ }}

```
A[5] = 100;
```

{{ ($A[j] < A[j+1]$ for any $0 \le j \le 4$) and $A[5] = 100$ and
($A[j] < A[j+1]$ for any $6 \le j \le 9$) }}

For-any facts get out of hand quickly!

We will look at ways to cope in Topic 10

Adding & Removing Elements

- Can efficiently add & remove at end of array:

```
ArrayList<Integer> A = …;
A.add(100);
```

- How do we handle this in Floyd logic?
  - working forward:

{{ P(A) }}
A.add(100);
{{ $P(A_0)$ and $A = A_0 +\!\!+ [100]$ }}

```
// @modifies obj
// @effects obj = obj_0 ++ [n]
void add(int n)
```

  - value changes by appending $[100]$ to previous value

Adding & Removing Elements

- Can efficiently add & remove at end of array:

```
ArrayList<Integer> A = …;
A.add(100);
```

- How do we handle this in Floyd logic?
  - working backward:

{{ P(A ⧺ [100]) }}
A.add(100);
{{ P(A) }}

```
// @modifies obj
// @effects obj = obj_0 ++ [n]
void add(int n)
```

  - just substitution through assignment $A = A ⧺ [100]$

Adding & Removing Elements

- Can efficiently add & remove at end of array:

```
ArrayList<Integer> A = …;
A.remove(A.size() - 1);  // remove last element
```

- How do we handle this in Floyd logic?
  - working forward:

{{ $P(A)$ }}
`A.remove(A.size() - 1);`
{{ $P(A_0)$ and $A = A_0[.. \text{len}(A_0) - 2]$ }}

  - new value is a (long) prefix of old value

Adding & Removing Elements

- Can efficiently add & remove at end of array:

```
ArrayList<Integer> A = …;
A.remove(A.size() - 1);  // remove last element
```

- How do we handle this in Floyd logic?
  - working backward:

{{ $P(A[.. \text{len}(A_0) - 2])$ }}

```
A.remove(A.size() - 1);
```

{{ $P(A)$ }}

  - substitution through assignment $A = A_0[.. \text{len}(A_0) - 2]$