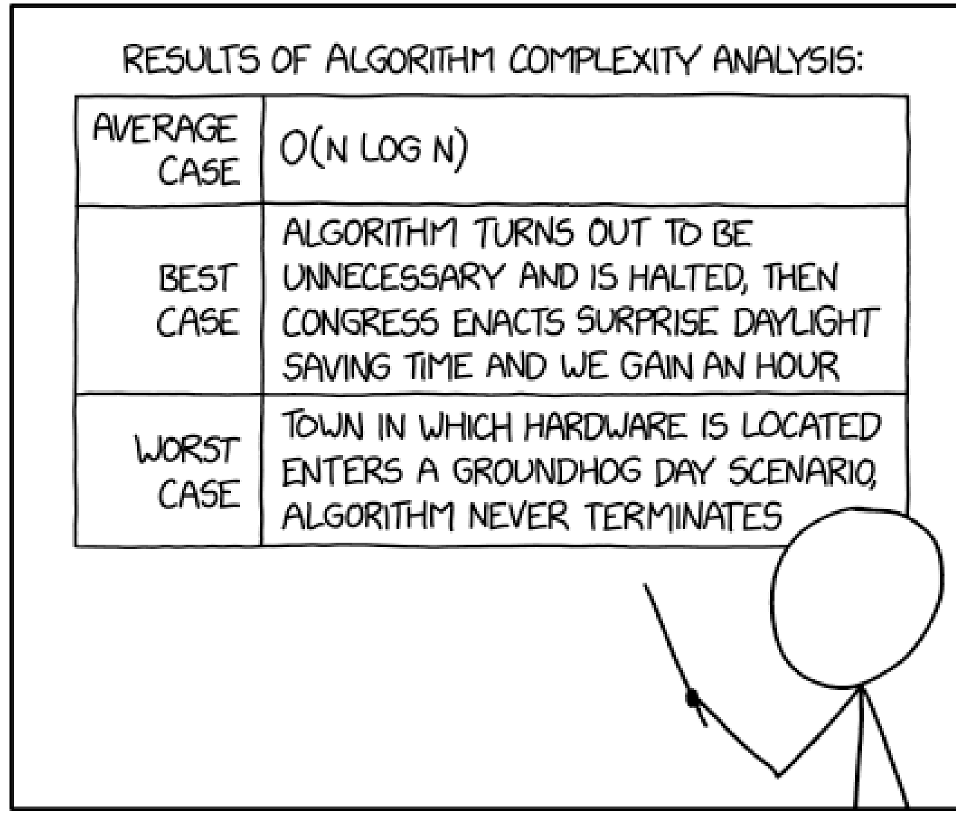
# CSE 331
# Spring 2025

## Arrays I




xkcd #2939


## Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

# Administrivia (05/23)

- HW8 is out!
  - beware: coding portion has a good chunk of math!
  - but also: very fun app :)
- Holiday on Monday…
  - **no class**
  - **no office hours**
  - **some (reduced) Ed activity**
- **Implication: please start HW8 early!!**
- **next Fri: a bit on the final exam**

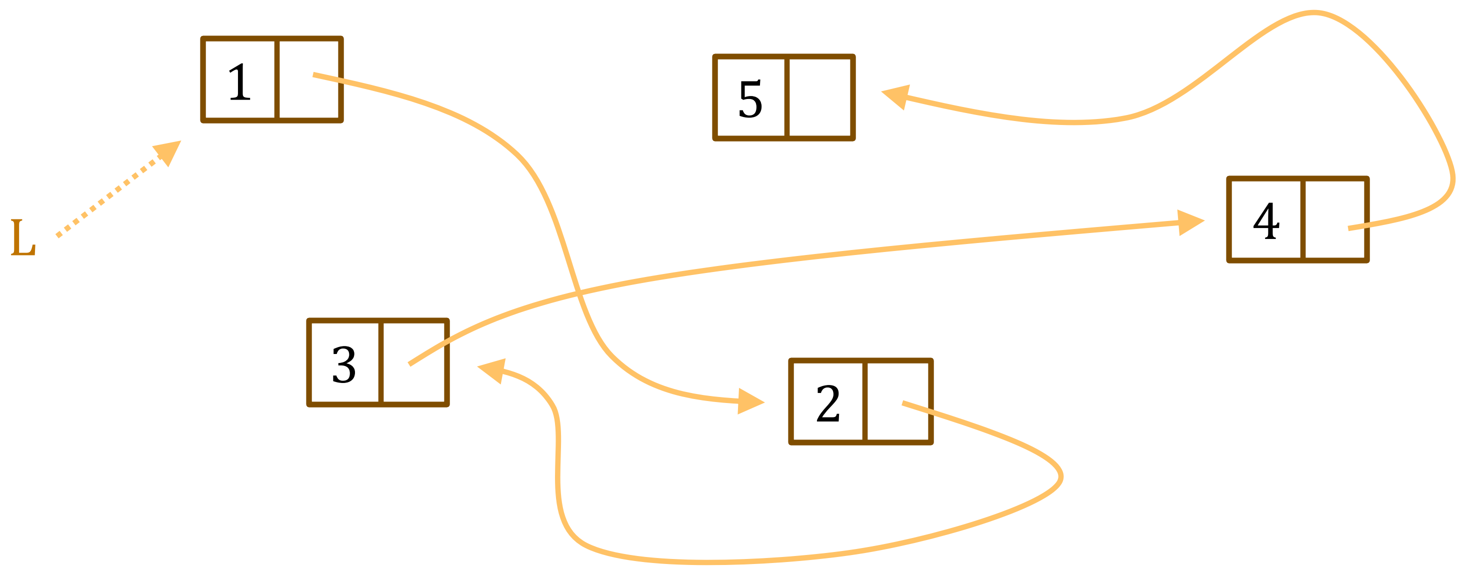
# List Indexing

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

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

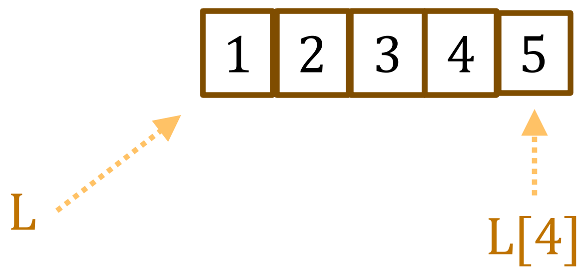
- **Retrieve an element of the list by <u>index</u>**
  - **use "$\mathrm{L}[\mathrm{j}]$" as an abbreviation for** $\mathrm{at}(\mathrm{j}, \mathrm{L})$
- **Not an efficient operation on lists…**

# Linked Lists in Memory




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

# Faster Implementation of at

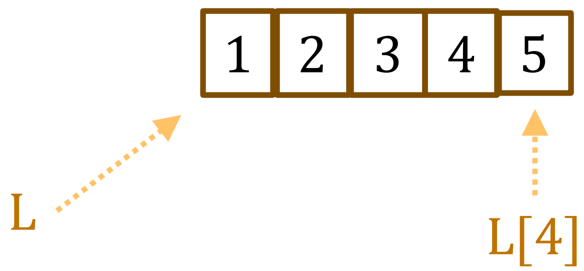



- **Alternative: store the elements next to each other**
  - **can find the** n**-th entry by arithmetic:**

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

- **Resulting data structure is an *array***
  - **consider: arrays can be an implementation of the List ADT**

# Array Efficiency




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

- "With great power, comes great responsibility"

  — the Peter Parker Principle

- Whenever we write "$\mathrm{A}[\mathrm{j}]$", we must check $0 \le \mathrm{j} < \mathrm{n}$
  - 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
  - TypeScript will not help us with this!

    type checker does catch "could be nil" bugs, but not this

# Recall: Sum List With a Loop

$$\text{sum-acc}(\text{nil}, r) \quad := r$$
$$\text{sum-acc}(x :: L, r) \quad := \text{sum-acc}(L, x + r)$$

- **Tail recursive version is a loop**

```
const sum = (S: List<bigint>): bigint => {
  let r = 0;
  // Inv: sum(S0) = r + sum(S)
  while (S.kind !== "nil") {
    r = S.hd + r;
    S = S.tl;
  }
  return r;
};
```

Change to a version that uses indexes…

# Sum Array by Index

- Change to using an array and accessing by index

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  // Inv: …
  while (j !== S.length) { // … S.kind !== "nil"
    r = S[j] + r;          // … r = S.hd + r
    j = j + 1;             // … S = S.tl
  }
  return r;
};
```

Note that **S** is no longer changing

# Sum Array by Index: compared to sum-acc

$$
\begin{aligned}
&\text{sum-acc} : (\text{List}, \mathbb{N}, \mathbb{Z}) \to \mathbb{Z} \\
&\text{sum-acc}(S, j, r) && := r && \textbf{if } j = \text{len}(S) \\
&\text{sum-acc}(S, j, r) && := \text{sum-acc}(S, j+1, S[j] + r) && \textbf{if } j \neq \text{len}(S)
\end{aligned}
$$

- **Change to using an array and accessing by index**

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  // Inv: …
  while (j !== S.length) {
    r = S[j] + r;
    j = j + 1;
  }
  return r;
};
```

# Sublists

- **Use indexes to refer to a section of a list (a "sublist"):**

$$\mathrm{sublist} : (\mathrm{List}\langle\mathbb{Z}\rangle, \mathbb{N}, \mathbb{Z}) \to \mathrm{List}\langle\mathbb{Z}\rangle$$

$$\begin{aligned}
\mathrm{sublist}(L, i, j) &:= \mathrm{nil} && \textbf{if } j < i \\
\mathrm{sublist}(L, i, j) &:= L[i] :: \mathrm{sublist}(L, i+1, j) && \textbf{if } i \leq j
\end{aligned}$$

- **Useful for *reasoning* about lists and indexes**

- **This includes <u>both</u> $L[i]$ and $L[j]$**

$$\begin{aligned}
\mathrm{sublist}(L, 0, 2) &= L[0] :: \mathrm{sublist}(L, 1, 2) && \textbf{def of } \mathrm{sublist}\ (\textbf{since } 0 \leq 2) \\
&= L[0] :: L[1] :: \mathrm{sublist}(L, 2, 2) && \textbf{def of } \mathrm{sublist}\ (\textbf{since } 1 \leq 2) \\
&= L[0] :: L[1] :: L[2] :: \mathrm{sublist}(L, 3, 2) && \textbf{def of } \mathrm{sublist}\ (\textbf{since } 2 \leq 2) \\
&= L[0] :: L[1] :: L[2] :: \mathrm{nil} && \textbf{def of } \mathrm{sublist}\ (\textbf{since } 3 < 2) \\
&= [L[0], L[1], L[2]]
\end{aligned}$$

# Sublists and Edge Cases

- **Use indexes to refer to a section of a list (a "sublist"):**

$$\text{sublist} : (\text{List}<\mathbb{Z}>, \mathbb{N}, \mathbb{Z}) \to \text{List}<\mathbb{Z}>$$

$$\begin{aligned} \text{sublist}(L, i, j) &:= \text{nil} && \textbf{if } j < i \\ \text{sublist}(L, i, j) &:= L[i] :: \text{sublist}(L, i+1, j) && \textbf{if } i \le j \end{aligned}$$

- **The sublist is empty when the range is empty**

$$\text{sublist}(L, 3, 2) = \text{nil}$$

  - **weird-looking example that comes up a lot:**

$$\text{sublist}(L, 0, \text{-}1) = \text{nil}$$

  - **not an array out of bonds error! (this is math, not Java)**

# Sublist Shorthands and Facts

$$\text{sublist} : (\text{List}<\mathbb{Z}>, \mathbb{N}, \mathbb{Z}) \to \text{List}<\mathbb{Z}>$$

$$\begin{array}{lll} \text{sublist}(L, i, j) & := \text{nil} & \textbf{if } j < i \\ \text{sublist}(L, i, j) & := L[i] :: \text{sublist}(L, i+1, j) & \textbf{if } i \leq j \end{array}$$

- **Will use "$L[i\ ..\ j]$" as shorthand for "$\text{sublist}(L, i, j)$"**
  - again, using an operator for most common operations

- **Some useful facts about sublists:**

$$L = L[0\ ..\ \text{len}(L)\text{-}1]$$

$L[i\ ..\ j] = L[i\ ..\ k] +\!\!\!+ L[k+1\ ..\ j]$ **for any** $k$ **with** $i - 1 \leq k \leq j$ (and $0 \leq i \leq j < n$)

# Sum Array by Index: sum-acc, in math

$$\begin{aligned} &\text{sum-acc}(S, j, r) && := r && \textbf{if } j = \text{len}(S) \\ &\text{sum-acc}(S, j, r) && := \text{sum-acc}(S, j+1, S[j] + r) && \textbf{if } j \neq \text{len}(S) \end{aligned}$$

- Change to using an array and accessing by index

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  // Inv: … ?? …
  while (j != S.length) {
    r = S[j] + r;
    j = j + 1;
  }
  return r;
};
```

Still need to fill in Inv…

Need a version using indexes.

# Recall: Sum List With a Loop, with Invariant

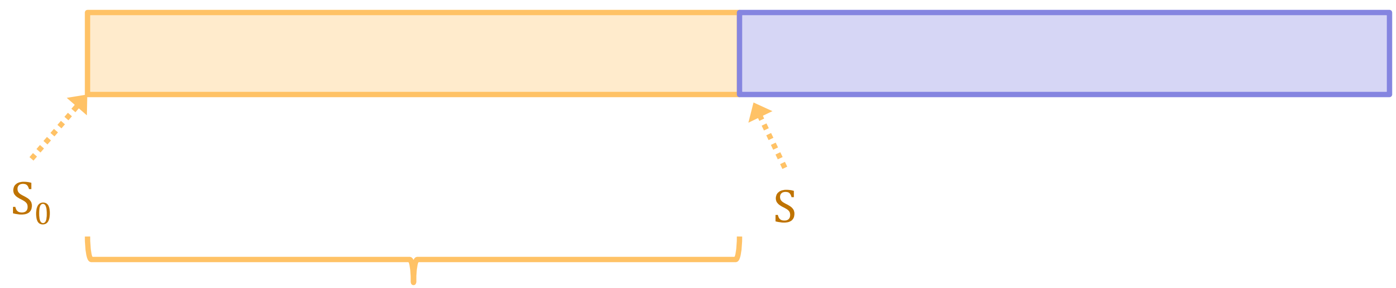
- Tail recursive version is a loop

```
const sum = (S: List<bigint>): bigint => {
  let r = 0;
  // Inv: sum(S₀) = r + sum(S)
  while (S.kind !== "nil") {
    r = S.hd + r;
    S = S.tl;
  }
  return r;
};
```

Inv says $\mathrm{sum}(S_0)$ is $r$ plus sum of rest $(S)$

Not the most explicit way of explaining "$r$"...

# Visual Intuition for Sum List Loop Invariant

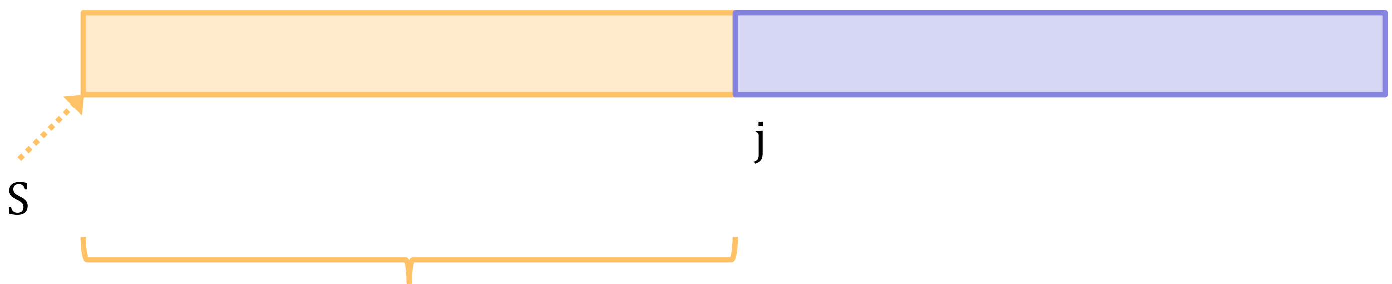



$$\mathrm{sum}(S_0) = r + \mathrm{sum}(S)$$

- "r" contains sum of the part of the list *seen so far*

- Can explain this more simply with indexes…
  - no longer need to move S

# Visual Intuition for Index & Sublist Loop Invariant




$\text{sum}(S) = \quad r \quad + \quad \text{sum}(S[j \mathrel{..} n-1])$

- **Sum is the part in "r" plus the part left in** $S[j \mathrel{..} n-1]$

- **What sum is in "r"?**

$r = \text{sum}(S[0 \mathrel{..} j-1])$

  - **we can use just this as our invariant! (it's all we need)**

# Sum of an Array: Loop Invariant

- Array version uses access by index

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  // Inv: r = sum(S[0 .. j-1])
  while (j != S.length) {
    r = S[j] + r;
    j = j + 1;
  }
  return r;
};
```

Are we sure this is right?

Let's think it through...

# Sum of an Array Floyd Logic: Initialization

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ r = 0 and j = 0 }}
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    r = S[j] + r;
    j = j + 1;
  }
  return r;
};
```

Does Inv hold initially?

$$
\begin{aligned}
& \mathrm{sum}(S[0 \,..\, j\text{-}1]) \\
& = \mathrm{sum}(S[0 \,..\, \text{-}1]) && \text{since } j = 0 \\
& = \mathrm{sum}([\,]) \\
& = 0 && \text{def of sum} \\
& = r
\end{aligned}
$$

# Sum of an Array Floyd Logic: Postcondition

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    r = S[j] + r;
    j = j + 1;
  }
  {{ r = sum(S[0 .. j-1]) and j = len(S) }}
  {{ r = sum(S) }}
  return r;
};
```

Does the postcondition hold?

$r = \mathrm{sum}(S[0 .. j\text{-}1])$

$\quad = \mathrm{sum}(S[0 .. \mathrm{len}(S)\text{-}1])$ **since** $j = \mathrm{len}(S)$

$\quad = \mathrm{sum}(S)$

# Sum of an Array Floyd Logic: Loop Body (1/4)

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    {{ r = sum(S[0 .. j-1]) and j ≠ len(S) }}
    r = S[j] + r;
    j = j + 1;
    {{ r = sum(S[0 .. j-1]) }}
  }
  return r;
};
```

# Sum of an Array Floyd Logic: Loop Body (2/4)

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    {{ r = sum(S[0 .. j-1]) and j ≠ len(S) }}
    r = S[j] + r;
    {{ r = sum(S[0 .. j]) }}
    j = j + 1;
    {{ r = sum(S[0 .. j-1]) }}
  }
  return r;
};
```

# Sum of an Array Floyd Logic: Loop Body (3/4)

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    {{ r = sum(S[0 .. j-1]) and j ≠ len(S) }}
    {{ S[j] + r = sum(S[0 .. j]) }}
    r = S[j] + r;
    {{ r = sum(S[0 .. j]) }}
    j = j + 1;
    {{ r = sum(S[0 .. j-1]) }}
  }
  return r;
};
```

# Sum of an Array Floyd Logic: Loop Body (4/4)

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    {{ r = sum(S[0 .. j-1]) and j ≠ len(S) }}
    {{ S[j] + r = sum(S[0 .. j]) }}
    r = S[j] + r;
    {{ r = sum(S[0 .. j]) }}
    j = j + 1;
    {{ r = sum(S[0 .. j-1]) }}
  }
  return r;
};
```

Is this valid?

# Proving Loop Body "Preservation" (1/3)

{{ $r = \text{sum}(S[0 .. j\text{-}1])$ and $j \neq \text{len}(S)$ }}
{{ $S[j] + r = \text{sum}(S[0 .. j])$ }}

$S[j] + r$
$= S[j] + \text{sum}(S[0 .. j\text{-}1])$ **since** $r = \text{sum}(S[0 .. j\text{-}1])$
$= \text{sum}(S[0 .. j\text{-}1]) + S[j]$
$= \text{sum}(S[0 .. j\text{-}1]) + \text{sum}([S[j]])$ **def of** $\text{sum}$
$= \text{sum}(S[0 .. j\text{-}1]) + \text{sum}(S[j .. j])$
$= \ldots$
$= \text{sum}(S[0 .. j])$

# Proving Loop Body "Preservation" (2/3)

{{ $r = \mathrm{sum}(S[0 .. j\text{-}1])$ and $j \neq \mathrm{len}(S)$ }}
{{ $S[j] + r = \mathrm{sum}(S[0 .. j])$ }}

$$
\begin{aligned}
& S[j] + r \\
& \quad = S[j] + \mathrm{sum}(S[0 .. j\text{-}1]) && \text{since } r = \mathrm{sum}(S[0 .. j\text{-}1]) \\
& \quad = \mathrm{sum}(S[0 .. j\text{-}1]) + S[j] \\
& \quad = \mathrm{sum}(S[0 .. j\text{-}1]) + \mathrm{sum}([S[j]]) && \text{def of sum} \\
& \quad = \mathrm{sum}(S[0 .. j\text{-}1]) + \mathrm{sum}(S[j .. j]) \\
& \quad = \dots \\
& \quad = \mathrm{sum}(S[0 .. j\text{-}1] \mathbin{+\!\!+} S[j .. j]) \\
& \quad = \mathrm{sum}(S[0 .. j])
\end{aligned}
$$

- **We saw that** $\mathrm{len}(L \mathbin{+\!\!+} R) = \mathrm{len}(L) + \mathrm{len}(R)$
- **Does** $\mathrm{sum}(L \mathbin{+\!\!+} R) = \mathrm{sum}(L) + \mathrm{sum}(R)$?
  - Yes! Very similar proof by structural induction. (Call this Lemma 3)

# Proving Loop Body "Preservation" (3/3)

{{ $r = \text{sum}(S[0 .. j\text{-}1])$ and $j \neq \text{len}(S)$ }}
{{ $S[j] + r = \text{sum}(S[0 .. j])$ }}

$S[j] + r$
$= S[j] + \text{sum}(S[0 .. j\text{-}1])$     **since** $r = \text{sum}(S[0 .. j\text{-}1])$
$= \text{sum}(S[0 .. j\text{-}1]) + S[j]$
$= \text{sum}(S[0 .. j\text{-}1]) + \text{sum}([S[j]])$     **def of** sum
$= \text{sum}(S[0 .. j\text{-}1]) + \text{sum}(S[j .. j])$
$= \text{sum}(S[0 .. j\text{-}1] \mathbin{+\!\!+} S[j .. j])$     **by Lemma 3**
$= \text{sum}(S[0 .. j])$

**(The need to reason by induction comes up all the time.)**

# Linear Search of a List

$$\begin{aligned}
&\mathrm{contains}(\mathrm{nil}, y) &&:= \mathrm{false} \\
&\mathrm{contains}(x :: L, y) &&:= \mathrm{true} && \textbf{if } x = y \\
&\mathrm{contains}(x :: L, y) &&:= \mathrm{contains}(L, y) && \textbf{if } x \neq y
\end{aligned}$$

- **Tail-recursive definition**

```
const contains =
    (S: List<bigint>, y: bigint): bigint => {
  // Inv: contains(S0, y) = contains(S, y)
  while (S.kind !== "nil" && S.hd !== y) {
    S = S.tl;
  }
  return S.kind !== "nil";  // implies S.hd === y
};
```

Change to a version that uses indexes…

# Linear Search of an Array

$$
\begin{array}{lll}
\text{contains}(\text{nil}, y) & := \text{false} & \\
\text{contains}(x :: L, y) & := \text{true} & \textbf{if } x = y \\
\text{contains}(x :: L, y) & := \text{contains}(L, y) & \textbf{if } x \neq y
\end{array}
$$

- **Change to using an array and accessing by index**

```
const contains =
    (S: Array<bigint>, y: bigint): bigint => {
  let j = 0;
  // Inv: ...
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  return j !== S.length;
};
```

`S.hd` with `S` changing becomes `S[j]` with `j` changing

What is the invariant now?

# Linear Search of an Array: Loop Invariant

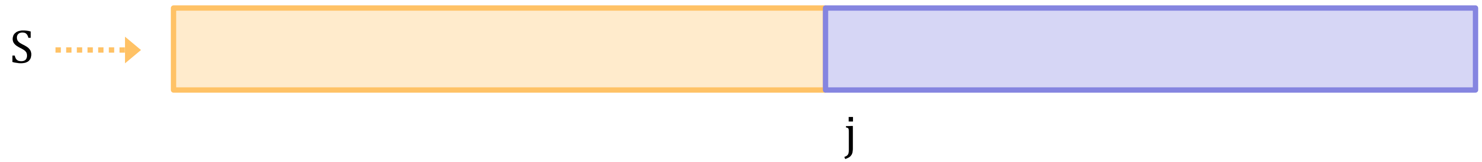
$$
\begin{array}{lll}
\text{contains}(\text{nil}, y) & := \text{false} & \\
\text{contains}(x :: L, y) & := \text{true} & \textbf{if } x = y \\
\text{contains}(x :: L, y) & := \text{contains}(L, y) & \textbf{if } x \neq y
\end{array}
$$

- **Change to using an array and accessing by index**

```
const contains =
    (S: Array<bigint>, y: bigint): bigint => {
  let j = 0;
  // Inv: contains(S, y) = contains(S[j .. n-1], y)
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  return j !== S.length;
};
```

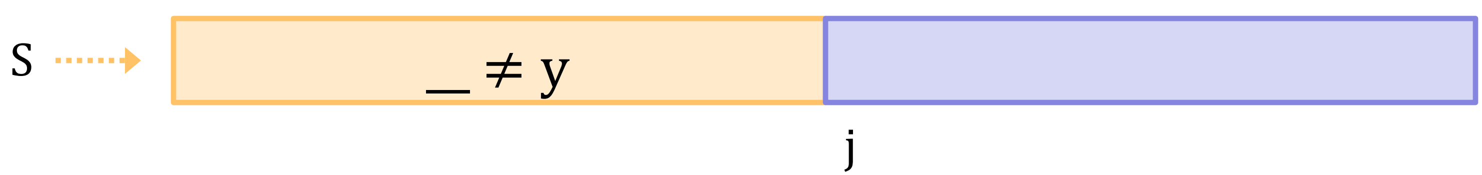
**Can we explain this better?**

# Linear Search of an Array: Visual Intuition




contains(S, y) = contains(S[j .. n-1], y)

- **What do we know about the left segment?**
  - it does not contain "y"
  - that's why we kept searching



# Linear Search of an Array: Refined Invariant


- **Update the invariant to be more informative**

```
const contains =
    (S: Array<bigint>, y: bigint): bigint => {
  let j = 0;
  // Inv: S[i] ≠ y for any i = 0 .. j-1
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  return j !== S.length;
};
```

# Sublist "For any" Facts

- "With great power, comes great responsibility"

- Since we can easily access any $\mathrm{L}[\mathrm{j}]$,
  may need to keep track of facts about it
  - may need facts about *every* element in the list

    applies to preconditions, postconditions, and intermediate assertions

- We can write facts about several elements at once:
  - this says that elements at indexes $0$ .. $\mathrm{j}$-$1$ are not $\mathrm{y}$

    $\mathrm{S}[\mathrm{i}] \neq \mathrm{y}$ for any $0 \leq \mathrm{i} < \mathrm{j}$

  - shorthand for $\mathrm{j}$ facts: $S[0] \neq \mathrm{y}, \ldots, S[\mathrm{j}\text{-}1] \neq \mathrm{y}$

# Reasoning Toolkit

| Description | Testing | Tools | Reasoning |
| --- | --- | --- | --- |
| no mutation | full coverage | type checker | calculation<br>induction |
| local variable mutation | “ | “ | Floyd logic |
| heap state | “ | “ | rep invariants |
| arrays | “ | “ | for-any facts |

# Sublist "For any" Facts & Pictures

- "With great power, comes great responsibility"
  - since we can easily access any $L[j]$, may need facts about it

- We can write facts about several elements at once:
  - this says that elements at indexes $0$ .. $j\text{-}1$ are not $y$

$$S[i] \neq y \qquad \text{for any } 0 \leq i < j$$

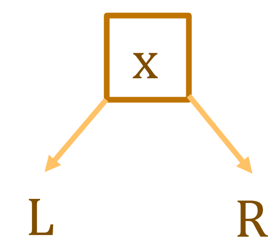
- These facts get hard to write down!
  - we will need to find ways to make this <u>easier</u>
  - a common trick is to draw pictures instead…

# Visual Presentation of Facts


- **Just saw this example**

- **But we have seen "for any" facts with BSTs...**

$$\text{contains-key}(y, L) \rightarrow (y < x)$$
$$\text{contains-key}(z, R) \rightarrow (x < z)$$



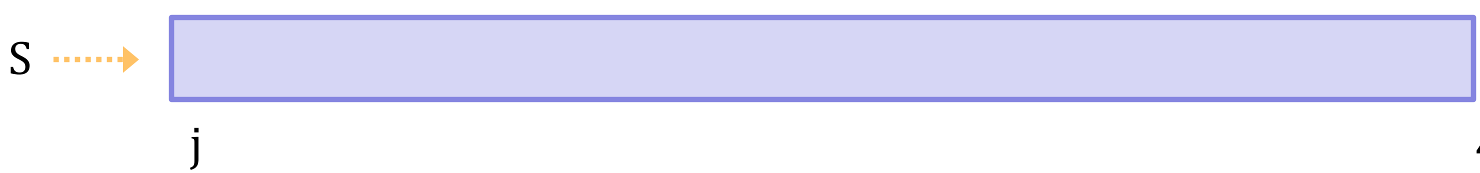

  - "for any" facts are common in more complex code
  - drawing pictures is a typical coping mechanism

# Proving Linear Search of an Array: Initialization


```
const contains =
    (S: Array<bigint>, y: bigint): boolean => {
  let j = 0;
  {{ j = 0 }}
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  return j !== S.length;
};
```

What is the picture when $j = 0$?

Inv holds because no $i$ is in [0, -1] (“vacuously true”)



# Linear Search of an Array: Preservation (1/4)


```
const contains =
    (S: Array<bigint>, y: bigint): boolean => {
  let j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j !== S.length && S[j] !== y) {
    {{ (S[i] ≠ y  for any 0 ≤ i ≤ j – 1) and j ≠ len(S) and S[j] ≠ y }}
    j = j + 1;
    {{ S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  }
  return j !== S.length;
};
```

# Linear Search of an Array: Preservation (2/4)


```
const contains =
    (S: Array<bigint>, y: bigint): boolean => {
  let j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j !== S.length && S[j] !== y) {
    {{ (S[i] ≠ y  for any 0 ≤ i ≤ j – 1) and j ≠ len(S) and S[j] ≠ y }}
    {{ S[i] ≠ y  for any 0 ≤ i ≤ j }}
    j = j + 1;
    {{ S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  }
  return j !== S.length;
};
```

Is this valid?

# Linear Search of an Array: Preservation (3/4)


{{ (S[i] $\neq$ y  for any $0 \leq i \leq j - 1$) and j $\neq$ len(S) and S[j] $\neq$ y }}
{{ S[i] $\neq$ y  for any $0 \leq i \leq j$ }}

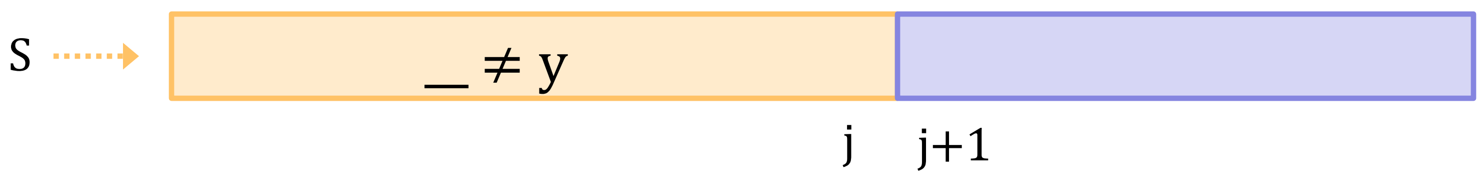
- **What does the top assertion say about S[j]?**
  - **it is not y**

# Linear Search of an Array: Preservation (4/4)


$\{\{\ (S[i] \neq y \text{ for any } 0 \leq i \leq j - 1) \text{ and } j \neq \text{len}(S) \text{ and } S[j] \neq y\ \}\}$

$\{\{\ S[i] \neq y \text{ for any } 0 \leq i \leq j\ \}\}$

- **What is the picture for the bottom assertion?**



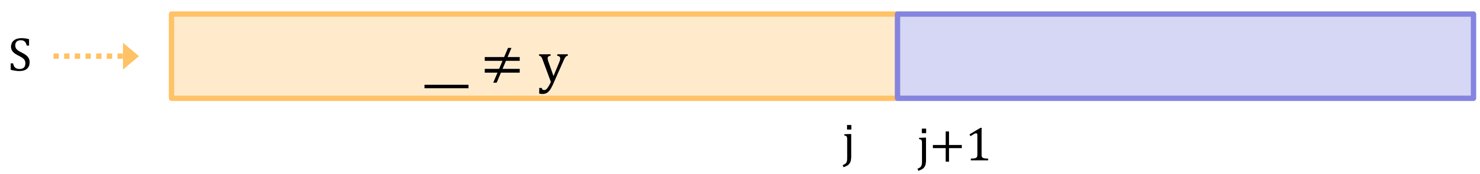

- **Do the facts above imply this holds?**
  - **Yes! It's the same picture**

# Array Indexing & Off-By-One Bugs (1/2)


{{ ($S[i] \neq y$ for any $0 \leq i \leq j - 1$) and $j \neq \text{len}(S)$ and $S[j] \neq y$ }}

{{ $S[i] \neq y$ for any $0 \leq i \leq j$ }}

- **What is the picture for the bottom assertion?**




- **Most likely bug is an off-by-one error**
  - **must check** S[j], **not** S[j-1] **or** S[j+1]

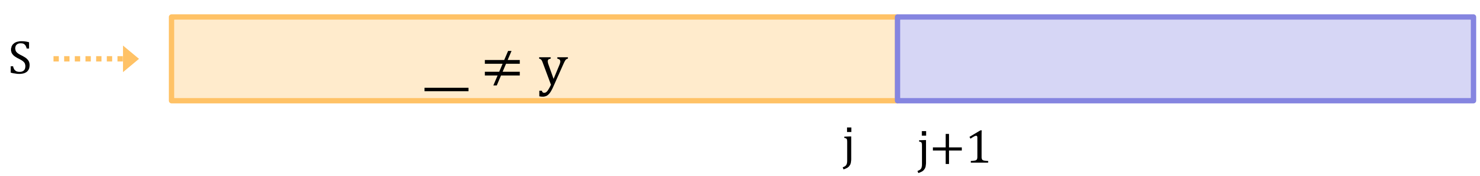
# Array Indexing & Off-By-One Bugs (2/2)


```
while (j !== S.length && S[j+1] !== y) {
```

{{ (S[i] $\neq$ y for any $0 \leq i \leq j - 1$) and j $\neq$ len(S) and S[j+1] $\neq$ y }}

{{ S[i] $\neq$ y for any $0 \leq i \leq j$ }}

- **What is the picture for the bottom assertion?**




- **Reasoning would verify that this is not correct**

# Proving Linear Search of an Array: Exit (1/2)


```
const contains =
    (S: Array<bigint>, y: bigint): boolean => {
  let j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  {{ Inv and (j = len(S) or S[j] = y) }}
  {{ contains(S, y) = (j ≠ len(S)) }}
  return j !== S.length;
};
```

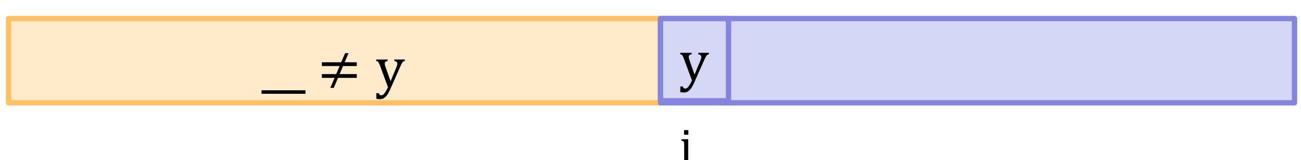
"or" means cases...

Case $j \neq \text{len}(S)$:

Must have $S[j] = y$.

What is the picture now?

Code should and does return true.



# Proving Linear Search of an Array: Exit (2/2)


```
const contains =
    (S: Array<bigint>, y: bigint): boolean => {
  let j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  {{ Inv and (j = len(S) or S[j] = y) }}
  {{ contains(S, y) = (j ≠ len(S)) }}
  return j !== S.length;
};
```

"or" means cases...

Case $j = \text{len}(S)$:

What does Inv say now?

Says y is not in the array!

Code should and does return false.