

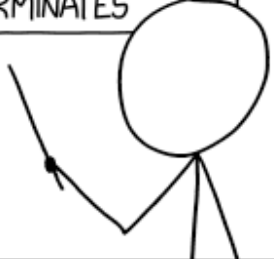
CSE 331

Summer 2025

Arrays I

RESULTS OF ALGORITHM COMPLEXITY ANALYSIS:

AVERAGE CASE	$O(N \log N)$
BEST CASE	ALGORITHM TURNS OUT TO BE UNNECESSARY AND IS HALTED, THEN CONGRESS ENACTS SURPRISE DAYLIGHT SAVING TIME AND WE GAIN AN HOUR
WORST CASE	TOWN IN WHICH HARDWARE IS LOCATED ENTERS A GROUNDHOG DAY SCENARIO, ALGORITHM NEVER TERMINATES



xkcd #2939, thanks Matt

Jaela Field

Administrivia (8/11)

- **Exam page is out and linked on resources page**
 - Includes all logistical details
 - Includes practice materials
 - 25su Practice final to come
- **Last section (8/21) will be final review**
 - With additional practice materials

List Indexing

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

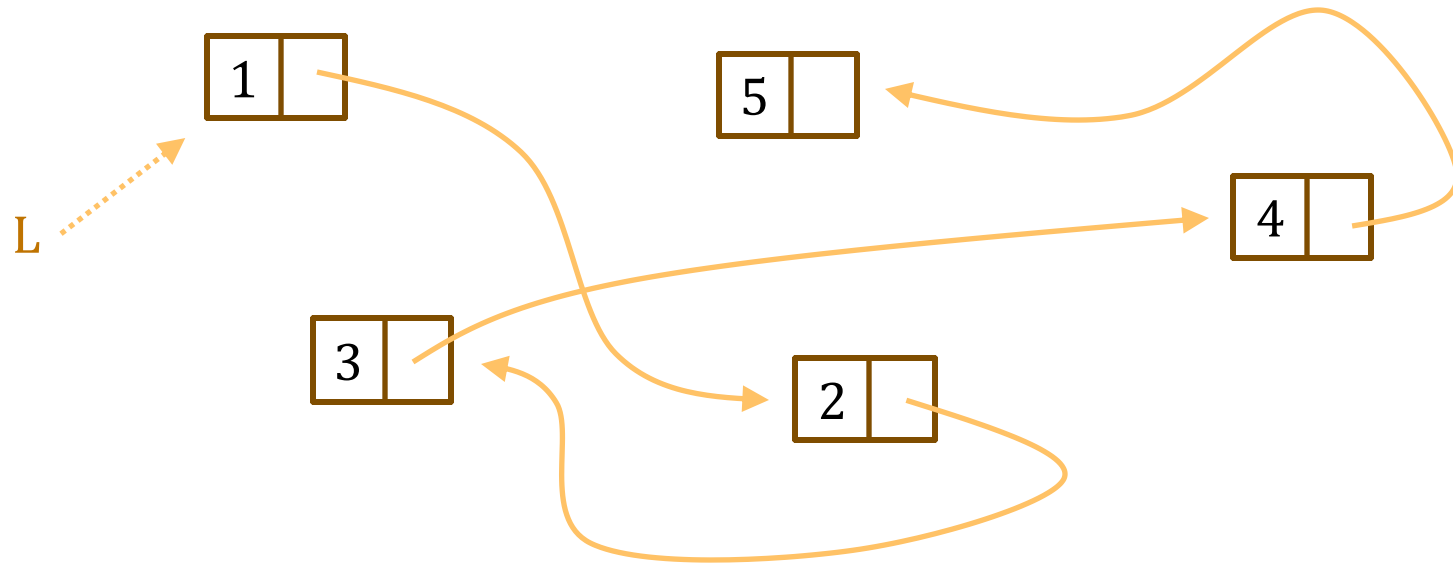
$\text{at}(\text{nil}, n) \quad \quad \quad := \text{undefined}$

$\text{at}(x :: L, 0) \quad \quad \quad := x$

$\text{at}(x :: L, n+1) \quad \quad := \text{at}(L, n)$

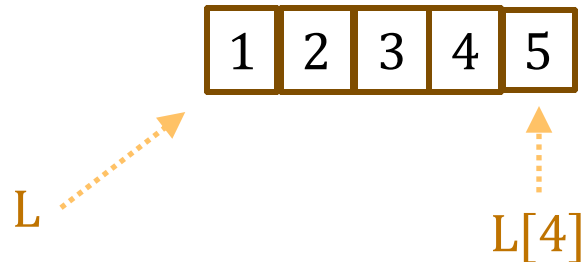
- **Retrieve an element of the list by index**
 - use " $L[j]$ " as an abbreviation for $\text{at}(j, L)$
- **Not an efficient operation on lists...**

Linked Lists in Memory



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

Faster Implementation of at

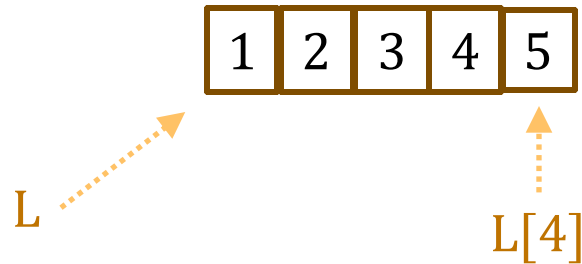


- **Alternative: store the elements next to each other**
 - can find the n -th entry by arithmetic:

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

- **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 all of this in $O(\log n)$ time

Access By Index

- **Easily access both $L[0]$ and $L[n-1]$, 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
 - **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

$\text{sum-acc} : (\text{List}, \mathbb{N}, \mathbb{Z}) \rightarrow \mathbb{Z}$

$\text{sum-acc}(S, j, r) \quad := r \quad \text{if } j = \text{len}(S)$

$\text{sum-acc}(S, j, r) \quad := \text{sum-acc}(S, j+1, S[j] + r) \quad \text{if } j \neq \text{len}(S)$

- 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"):

$\text{sublist} : (\text{List}\langle\mathbb{Z}\rangle, \mathbb{N}, \mathbb{Z}) \rightarrow \text{List}\langle\mathbb{Z}\rangle$

$\text{sublist}(L, i, j) \quad := \text{nil} \quad \text{if } j < i$

$\text{sublist}(L, i, j) \quad := L[i] :: \text{sublist}(L, i + 1, j) \quad \text{if } i \leq j$

- Useful for *reasoning* about lists and indexes
- This includes both $L[i]$ and $L[j]$

$\text{sublist}(L, 0, 2) = L[0] :: \text{sublist}(L, 1, 2)$

$= L[0] :: L[1] :: \text{sublist}(L, 2, 2)$

$= L[0] :: L[1] :: L[2] :: \text{sublist}(L, 3, 2)$

$= L[0] :: L[1] :: L[2] :: \text{nil}$

$= [L[0], L[1], L[2]]$

def of sublist (since $0 \leq 2$)

def of sublist (since $1 \leq 2$)

def of sublist (since $2 \leq 2$)

def of sublist (since $3 < 2$)

Sublists and Edge Cases

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

$$\text{sublist} : (\text{List}\langle\mathbb{Z}\rangle, \mathbb{N}, \mathbb{Z}) \rightarrow \text{List}\langle\mathbb{Z}\rangle$$
$$\text{sublist}(L, i, j) \quad := \text{nil} \quad \text{if } j < i$$
$$\text{sublist}(L, i, j) \quad := L[i] :: \text{sublist}(L, i + 1, j) \quad \text{if } i \leq j$$

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

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

- weird-looking example that comes up a lot:

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

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

Sublist Shorthands and Facts

$\text{sublist} : (\text{List}\langle\mathbb{Z}\rangle, \mathbb{N}, \mathbb{Z}) \rightarrow \text{List}\langle\mathbb{Z}\rangle$

$\text{sublist}(L, i, j) \quad := \text{nil} \quad \text{if } j < i$

$\text{sublist}(L, i, j) \quad := L[i] :: \text{sublist}(L, i + 1, j) \quad \text{if } i \leq j$

- **Will use " $L[i \dots j]$ " as shorthand for " $\text{sublist}(L, i, j)$ "**
 - again, using an operator for most common operations
- **Some useful facts about sublists:**

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

$L[i \dots j] = L[i \dots k] \# L[k+1 \dots j] \quad \text{for any } k \text{ with } i - 1 \leq k \leq j \text{ (and } 0 \leq i \leq j < n)$

Sum Array by Index: sum-acc, in math

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

- 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

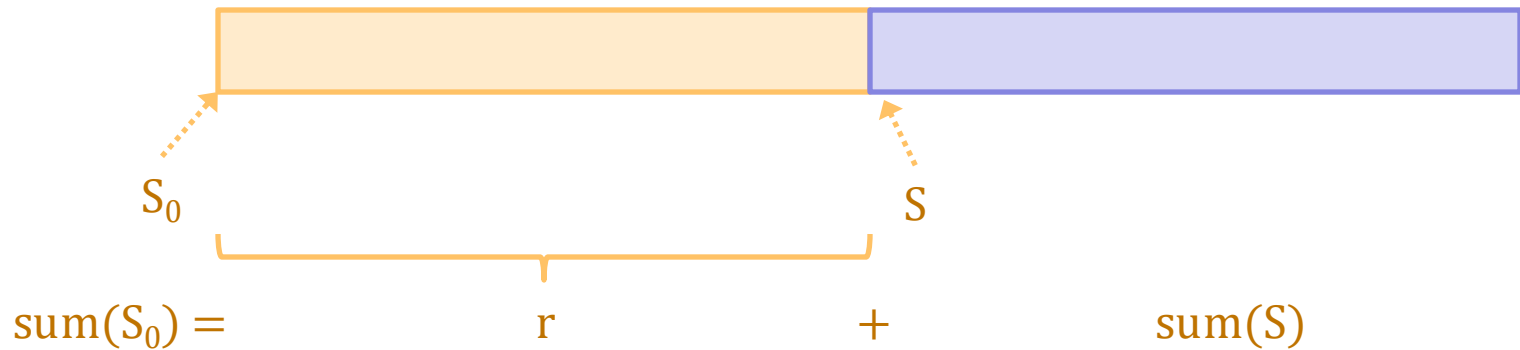
- Tail recursive version is a loop

```
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 $\text{sum}(S_0)$ is r plus sum of rest (S)

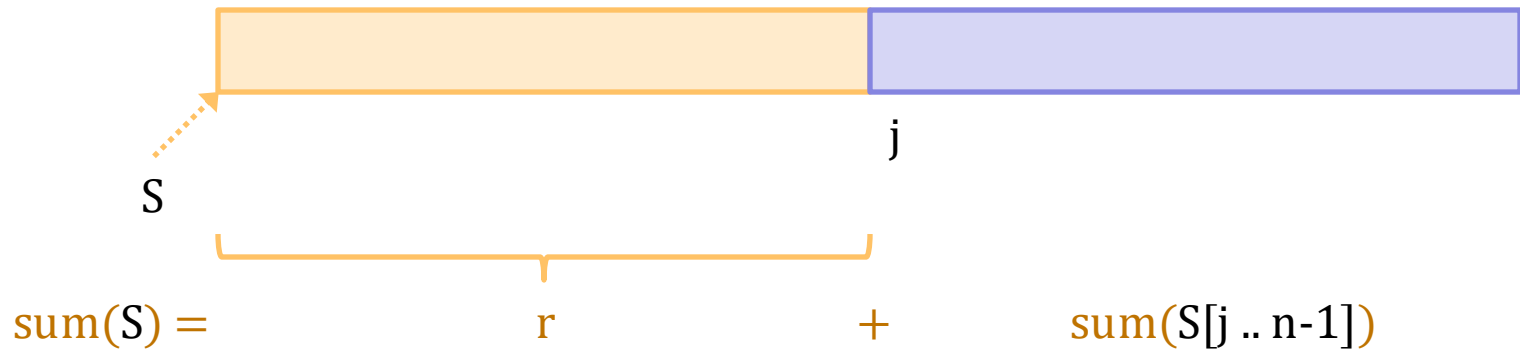
Not the most explicit way of explaining " r "...

Visual Intuition for Sum List Loop Invariant



- "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



- Sum is the part in "r" plus the part left in $S[j .. n-1]$
- What sum is in "r"?

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

- 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?

sum(S[0 .. j-1])
= sum(S[0 .. -1]) since j = 0
= sum([])
= 0 def of sum
= r

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?

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

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)

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

$S[j] + r$

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

$= \text{sum}(S[0 \dots j-1]) + S[j]$

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

$= \text{sum}(S[0 \dots j-1]) + \text{sum}(S[j \dots j])$

$= \dots$

$= \text{sum}(S[0 \dots j])$

Proving Loop Body “Preservation” (2/3)

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

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

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

Proving Loop Body “Preservation” (3/3)

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

$$\begin{aligned} & S[j] + r \\ &= S[j] + \text{sum}(S[0 \dots j-1]) && \text{since } r = \text{sum}(S[0 \dots j-1]) \\ &= \text{sum}(S[0 \dots j-1]) + S[j] \\ &= \text{sum}(S[0 \dots j-1]) + \text{sum}([S[j]]) && \text{def of sum} \\ &= \text{sum}(S[0 \dots j-1]) + \text{sum}(S[j \dots j]) \\ &= \text{sum}(S[0 \dots j-1] \# S[j \dots j]) && \text{by Lemma 3} \\ &= \text{sum}(S[0 \dots j]) \end{aligned}$$

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

Linear Search of a List

$\text{contains}(\text{nil}, y) \quad := \text{false}$
 $\text{contains}(x :: L, y) \quad := \text{true} \quad \text{if } x = y$
 $\text{contains}(x :: L, y) \quad := \text{contains}(L, y) \quad \text{if } x \neq y$

- 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

$\text{contains}(\text{nil}, y) \quad := \text{false}$
 $\text{contains}(x :: L, y) \quad := \text{true} \quad \text{if } x = y$
 $\text{contains}(x :: L, y) \quad := \text{contains}(L, y) \quad \text{if } x \neq y$

- 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;  
  };
```

$S.\text{hd}$ with S changing becomes
 $S[j]$ with j changing

What is the invariant now?

Linear Search of an Array: Loop Invariant

$\text{contains}(\text{nil}, y) \quad := \text{false}$
 $\text{contains}(x :: L, y) \quad := \text{true} \quad \text{if } x = y$
 $\text{contains}(x :: L, y) \quad := \text{contains}(L, y) \quad \text{if } x \neq y$

- 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



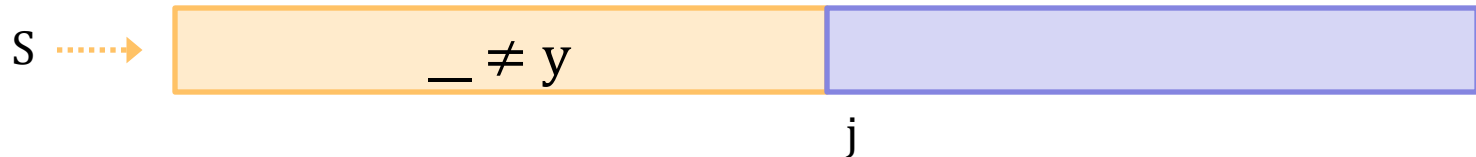
$\text{contains}(S, y) =$

$\text{contains}(S[j .. n-1], y)$

- 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 $L[j]$,
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 $0 \dots j-1$ are not y

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

- shorthand for j facts: $S[0] \neq y, \dots, S[j-1] \neq y$

Reasoning Toolkit

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

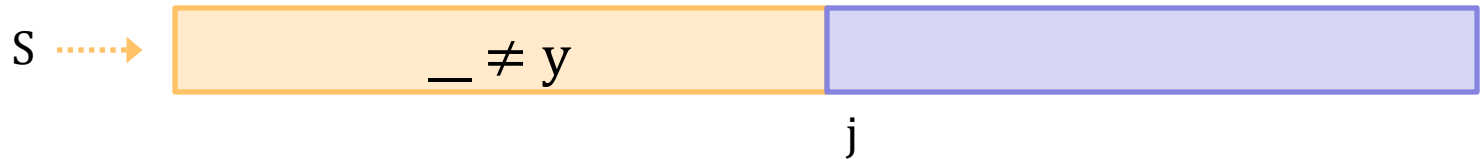
Sublist “For any” Facts & Pictures

- “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 $0 \dots j-1$ are not y

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

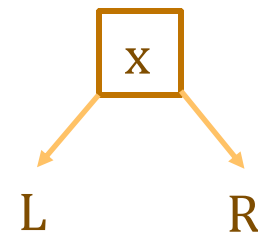
- These facts get hard to write down!
 - we will need to find ways to make this easier
 - a common trick is to **draw pictures** instead...

Visual Presentation of Facts



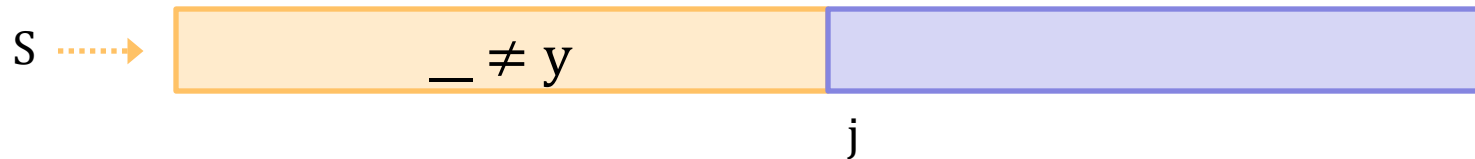
- Just saw this example
- But we have seen "for any" facts with BSTs...

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



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

Proving Linear Search of an Array: Initialization



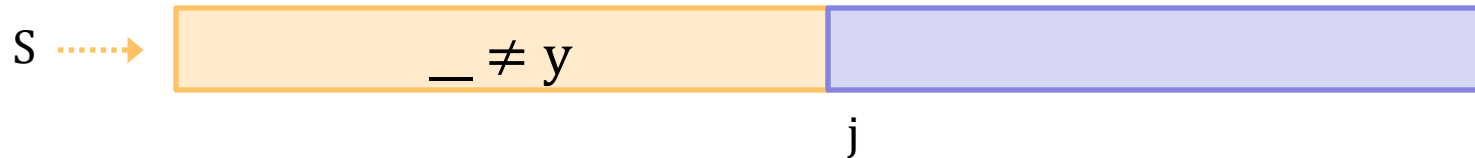
```
const contains =  
  (S: Array<bigint>, y: bigint): boolean => {  
    let j = 0;  
    {{ j = 0 }}  
    {{ Inv: S[i] ≠ y for any  $0 \leq i \leq j - 1$  }}  
    while (j !== S.length && S[j] !== y) {  
      j = j + 1;  
    }  
    return j !== S.length;  
  };
```

What is the picture when $j = 0$?

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

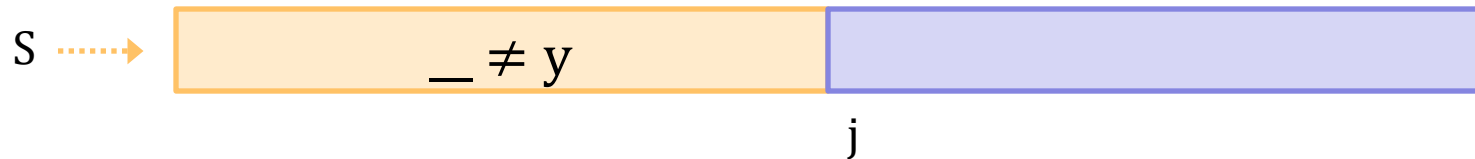


Linear Search of an Array: Preservation (1/4)



```
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  $0 \leq i \leq j - 1$  }}  
    while (j !== S.length && S[j] !== y) {  
      {{ (S[i] ≠ y for any  $0 \leq i \leq j - 1$ ) and  $j \neq \text{len}(S)$  and  $S[j] \neq y$  }}  
      ↑ {{ S[i] ≠ y for any  $0 \leq i \leq j$  }}  
      j = j + 1;  
      {{ S[i] ≠ y for any  $0 \leq i \leq j - 1$  }}  
    }  
    return j !== S.length;  
  };
```

Is this valid?

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



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

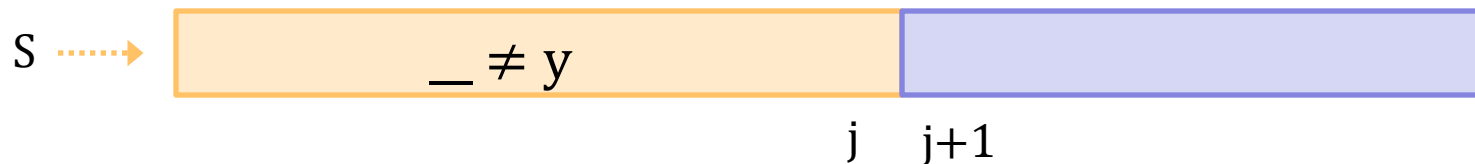
- What does the top assertion say about $S[j]$?
 - it is not y

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?



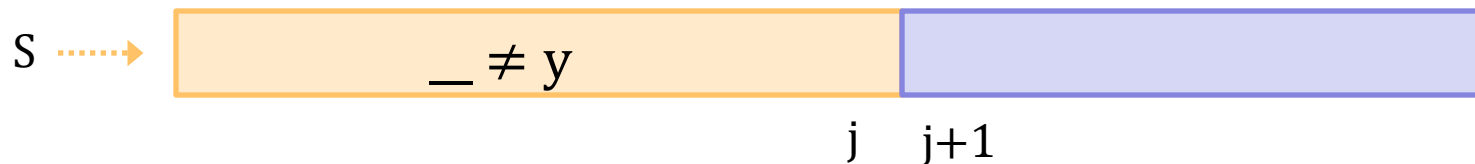
- Do the facts above imply this holds?
 - Yes! It's the same picture

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



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

- What is the picture for the bottom assertion?



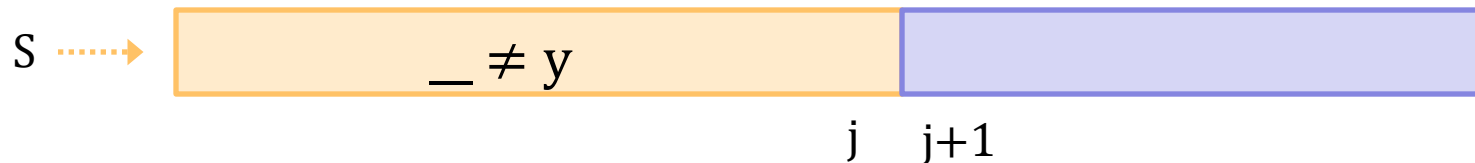
- Most likely bug is an off-by-one error
 - must check $S[j]$, not $S[j-1]$ or $S[j+1]$

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



```
while (j != S.length && S[j+1] != y) {  
    {{ (S[i] ≠ y for any  $0 \leq i \leq j-1$ ) and  $j \neq \text{len}(S)$  and  $S[j+1] \neq y$  }}  
    {{ S[i] ≠ 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  $0 \leq i \leq j - 1$  }}  
    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;  
  };
```

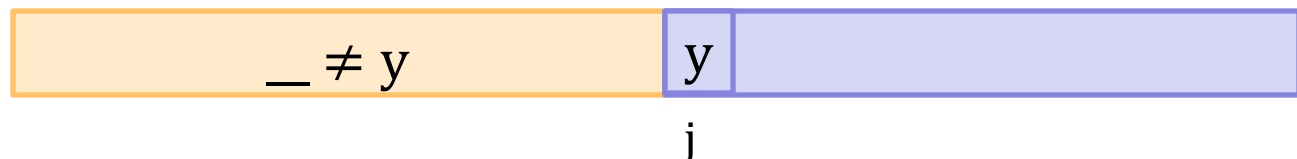
"or" means cases...

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;  
  };
```

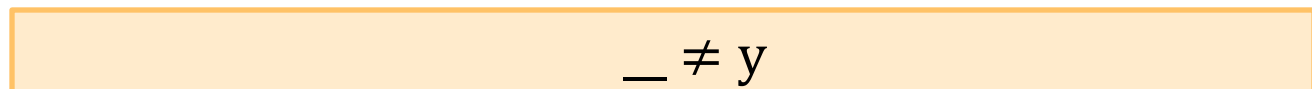
"or" means cases...

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

What does Inv say now?

Says y is not in the array!

Code should and does return **false**.



Finding an Element in an Array

- Can search for an element in an array as follows

$\text{contains}(\text{nil}, y) \quad := \text{false}$
 $\text{contains}(x :: L, y) \quad := \text{true} \quad \text{if } x = y$
 $\text{contains}(x :: L, y) \quad := \text{contains}(L, y) \quad \text{if } x \neq y$

- Searches through the array in linear time
 - did the same on lists
- Can be done more quickly if the list is sorted
 - binary search!

Finding an Element in a Sorted Array

- Can search more quickly if the list is sorted
 - precondition is $A[0] \leq A[1] \leq \dots \leq A[n-1]$ (informal)
 - write this formally as

$$A[j] \leq A[j+1] \text{ for any } 0 \leq j \leq n - 2$$

- Not easy to describe this visually...
 - how about a gradient?



Binary Search of an Array



```
const bsearch = (S: ..., y: ...) : boolean => {  
  let j = 0, k = S.length;  
  {{ Inv: (S[i] < y for any  $0 \leq i < j$ ) and ( $y \leq S[i]$  for any  $k \leq i < n$ ) }}  
  while (j != k) {  
    const m = (j + k) / 2n;  
    if (S[m] < y) {  
      j = m + 1;  
    } else {  
      k = m;  
    }  
  }  
  return (S[k] == y);  
};
```

Inv includes facts about two regions.
Let's check that this is right...

CSE 331 Summer 2025

Arrays II

Jaela Field

Recall: Binary Search of an Array



```
const bsearch = (S: ..., y: ...) : boolean => {  
  let j = 0, k = S.length;  
  {{ Inv: (S[i] < y for any  $0 \leq i < j$ ) and ( $y \leq S[i]$  for any  $k \leq i < n$ ) }}  
  while (j !== k) {  
    const m = (j + k) / 2n;  
    if (S[m] < y) {  
      j = m + 1;  
    } else {  
      k = m;  
    }  
  }  
  return (S[k] === y);  
};
```

Binary Search of an Array: Initialization



```
const bsearch = (S: ..., y: ...) : boolean => {  
  let j = 0, k = S.length;  
  {{ j = 0 and k = n }}  
  {{ Inv: (S[i] < y for any  $0 \leq i < j$ ) and ( $y \leq S[i]$  for any  $k \leq i < n$ ) }}
```

- What does the picture look like with $j = 0$ and $k = n$?



- Does this hold?
 - Yes! It's vacuously true

Binary Search of an Array: Exit Condition (1/3)



```
const bsearch = (S: ..., y: ...) : boolean => {  
  let j = 0, k = S.length;  
  {{ Inv: (S[i] < y for any  $0 \leq i < j$ ) and ( $y \leq S[i]$  for any  $k \leq i < n$ ) }}  
  while (j != k) {  
    ...  
  }  
  {{ Inv and ( $j = k$ ) }}  
  {{ contains(S, y) = ( $S[j] = y$ ) }}  
  return (S[k] == y);  
};
```

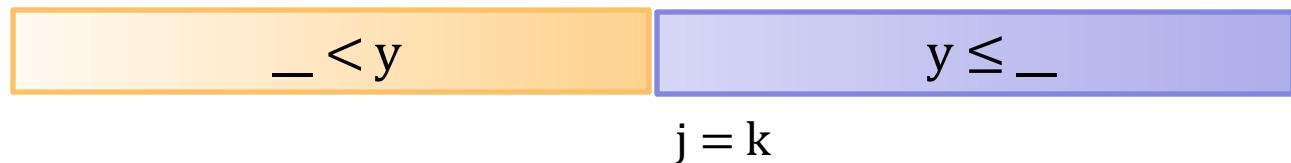
Binary Search of an Array: Exit Condition (2/3)



```
{{ Inv and (j = k) }}  
{{ contains(S, y) = (S[j] = y) }}  
return (S[k] == y);
```

```
};
```

- What does the picture look like with $j = k$?



- Does S contain y iff $S[k] = y$?
– If $S[k] = y$, then $\text{contains}(S, y) = \text{true}$
– If $S[k] \neq y$, then $S[k] < y$ and $S[i] < y$ for every $k < i$, so $\text{contains}(S, y) = \text{false}$

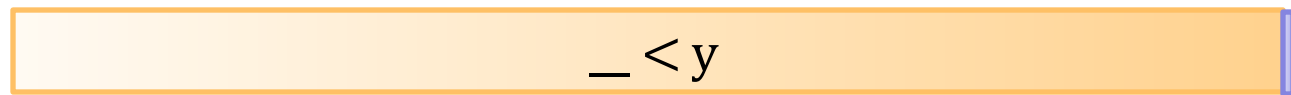
What case are we missing?

Binary Search of an Array: Exit Condition (3/3)



```
{{ Inv and (j = k) }}  
{{ contains(S, y) = (S[j] = y) }}  
return (S[k] == y) ;  
};
```

- What does the picture look like with $j = k = n$?



$j = k = n$

- In this case...
 - we see that $\text{contains}(S, y) = \text{false}$
 - and the code returns false because " $\text{undefined} == y$ " is false
(Okay, but yuck.)

Binary Search of an Array: Preservation (1/5)



```
{ { Inv: ( $S[i] < y$  for any  $0 \leq i < j$ ) and ( $y \leq S[i]$  for any  $k \leq i < n$ ) } }  
while ( $j \neq k$ ) {  
    { { Inv and ( $j < k$ ) } }  
    const  $m = (j + k) / 2$ ;  
    if ( $S[m] < y$ ) {  
         $j = m + 1$ ;  
    } else {  
         $k = m$ ;  
    }  
    { { ( $S[i] < y$  for any  $0 \leq i < j$ ) and ( $y \leq S[i]$  for any  $k \leq i < n$ ) } }  
}
```

Reason through both paths...

Binary Search of an Array: Preservation (2/5)



```
    {{ Inv and (j < k) }}  
    const m = (j + k) / 2n;  
    if (S[m] < y) {  
        → {{ Inv and (j < k) and (S[m] < y) }}  
        j = m + 1;  
    } else {  
        → {{ Inv and (j < k) and (S[m] ≥ y) }}  
        k = m;  
    }  
    {{ (S[i] < y for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}  
}
```


Binary Search of an Array: Preservation (3/5)



```
const m = (j + k) / 2n;
```

```
if (S[m] < y) {
```

```
  {{ Inv and (j < k) and (S[m] < y) }}
```

```
  {{ (S[i] < y for any  $0 \leq i < m+1$ ) and ( $y \leq S[i]$  for any  $k \leq i < n$ ) }}
```

```
  j = m + 1;
```

```
} else {
```

```
  {{ Inv and (j < k) and (S[m]  $\geq$  y) }}
```

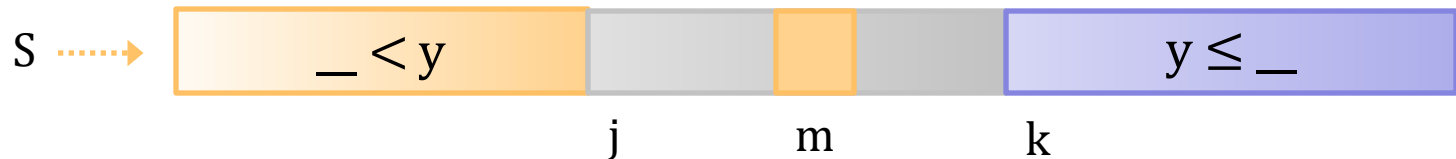
```
  {{ (S[i] < y for any  $0 \leq i < j$ ) and ( $y \leq S[i]$  for any  $m \leq i < n$ ) }}
```

```
  k = m;
```

```
}
```

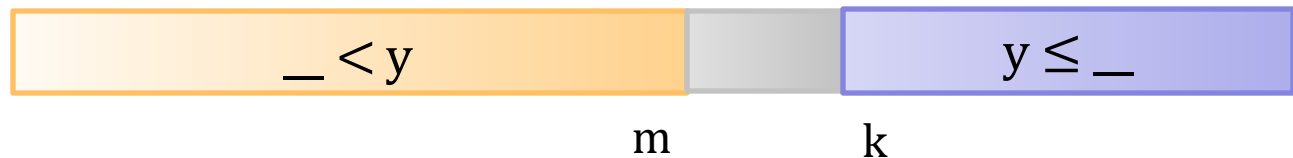
```
{{ (S[i] < y for any  $0 \leq i < j$ ) and ( $y \leq S[i]$  for any  $k \leq i < n$ ) }}
```

Binary Search of an Array: Preservation (4/5)



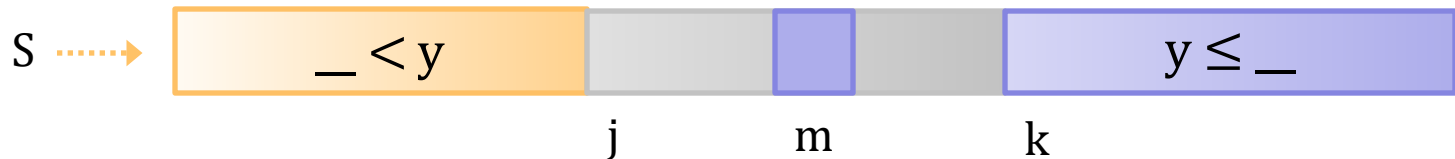
```
const m = (j + k) / 2n;  
if (S[m] < y) {  
    {{ Inv and (j < k) and (S[m] < y) }}  
    {{ (S[i] < y for any  $0 \leq i < m+1$ ) and ( $y \leq S[i]$  for any  $k \leq i < n$ ) }}  
    j = m + 1;  
} ...
```

- What does the picture look like in the bottom assertion?



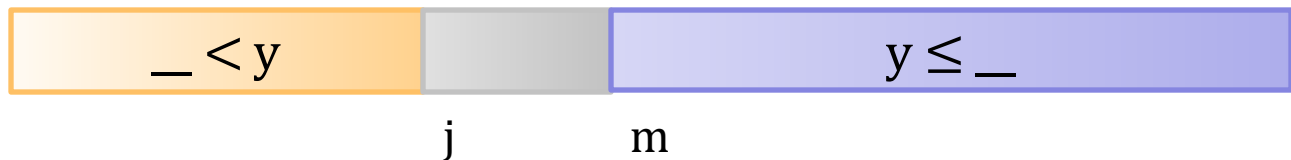
- Does this hold?
 - Yes! Because the array is sorted (everything **before** $S[m]$ is even **smaller**)

Binary Search of an Array: Preservation (5/5)



```
const m = (j + k) / 2n;  
... else {  
    {{ Inv and (j < k) and (S[m] ≥ y) }}  
    {{ (S[i] < y for any 0 ≤ i < j) and (y ≤ S[i] for any m ≤ i < n) }}  
    k = m;  
}
```

- What does the picture look like in the bottom assertion?



- Does this hold?
 - Yes! Because the array is sorted (everything **after** $S[m]$ is even **larger**)

Binary Search of an Array: Termination



```
const bsearch = (S: ..., y: ...) : boolean => {  
  let j = 0, k = S.length;  
  {{ Inv: (S[i] < y for any  $0 \leq i < j$ ) and ( $y \leq S[i]$  for any  $k \leq i < n$ ) }}  
  while (j != k) {  
    const m = (j + k) / 2n;  
    if (S[m] < y) {  
      j = m + 1;  
    } else {  
      k = m;  
    }  
  }  
  return (S[k] == y);  
};
```

Does this terminate?

Need to check that $k - j$ decreases

Can see that $j \leq m \leq k$, so
the "then" branch is fine.

Can see that $j < k$ implies $m < k$
(integer division rounds down), so
the "else" branch is also fine

Binary Search really *can* be tricky!

Extra, Extra – Read All About It: Nearly All Binary Searches and Mergesorts are Broken

June 2, 2006 · Posted by Joshua Bloch, Software Engineer

Google Research

I remember vividly Jon Bentley's first Algorithms lecture at CMU, where he asked all of us incoming Ph.D. students to write a binary search, and then dissected one of our implementations in front of the class. Of course it was broken, as were most of our implementations. This made a real impression on me, as did the treatment of this material in his wonderful *Programming Pearls* (Addison-Wesley, 1986; Second Edition, 2000). The key lesson was to carefully consider the invariants in your programs.

Fast forward to 2006. I was shocked to learn that the binary search program that Bentley proved correct and subsequently tested in Chapter 5 of *Programming Pearls* contains a bug. Once I tell you what it is, you will understand why it escaped detection for two decades. Lest you think I'm picking on Bentley, let me tell you how I discovered the bug: The version of binary search that I wrote for the JDK contained the same bug. It was reported to Sun recently when it broke someone's program, after lying in wait for nine years or so.

QUICK LINKS

 Share

A Sketch of a More Rigorous Approach

- **can convert visuals into rigorous proofs**
 - **requires some math machinery in 311 (and onwards)**
proving “for any” facts (related: “strong induction”)
more complicated reasoning for non-contiguous facts
 - **sketches for you to ponder:**
 $P(i) \text{ for any } j \leq i < k \text{ and } P(k) \rightarrow P(i) \text{ for any } j \leq i \leq k$
 $P(i) \text{ for any } j < i \leq k \text{ and } P(j) \rightarrow P(i) \text{ for any } j \leq i \leq k$
- **homework & exam:**
 - **won't require you to draw pictures or formally prove**
will ask you to write assertions & describe proof intuition (in English)
general conceptual questions are fair game!
 - **all array problems can be treated “as a list”**

Why Should You Care?

- **To justify practicing reasoning, we've said:**
 - professional programmers do formal reasoning for really tricky problems
 - it can help identify and prevent hard bugs
 - code review
- **Also, it can help you get a job!**
 - interviews are intentionally tricky
 - interviewers won't expect line-by-line reasoning or formal proofs
 - being able to precisely define loop invariants and pre and post conditions will go a long way
 - pictures count!

Creating Loop Invariants

Code Reviews vs 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

Loop Invariants with Arrays (1/3)

- Previous example:

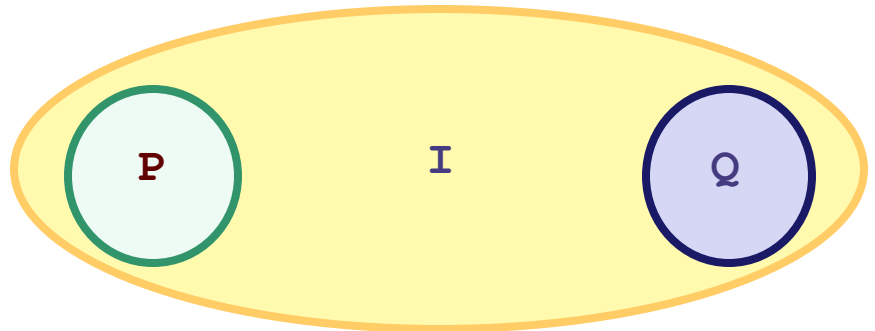
$\{\{ \text{Inv: } s = \text{sum}(S[0 \dots j - 1]) \dots \}\}$	sum of array
$\{\{ \text{Post: } s = \text{sum}(S[0 \dots n - 1]) \}\}$	

- in this case, Post is a special case of Inv (where $j = n$)
 - in other words, Inv is a **weakening** of Post
- Heuristic for loop invariants: weaken the postcondition
 - assertion that allows postcondition as a special case
 - must also allow states that are easy to prepare

Heuristic for Loop Invariants

- Loop Invariant allows both start and stop states
 - describing more states = weakening

```
{{ P }}  
{{ Inv: I }}  
while (cond) {  
    S  
}  
{{ Q }}
```



- usually are many ways to weaken it...

Loop Invariants with Arrays (2/3)

- Previous example

$\{\{ \text{Inv: } s = \text{sum}(S[0 \dots j - 1]) \dots \}\}$	sum of array
$\{\{ \text{Post: } s = \text{sum}(S[0 \dots n - 1]) \}\}$	

- Linear search also fits this pattern:

$\{\{ \text{Inv: } S[i] \neq y \text{ for any } 0 \leq i < j \}\}$	search an array
$\{\{ \text{Post: } (S[i] = y) \text{ or } (S[i] \neq y \text{ for any } 0 \leq i < n) \}\}$	

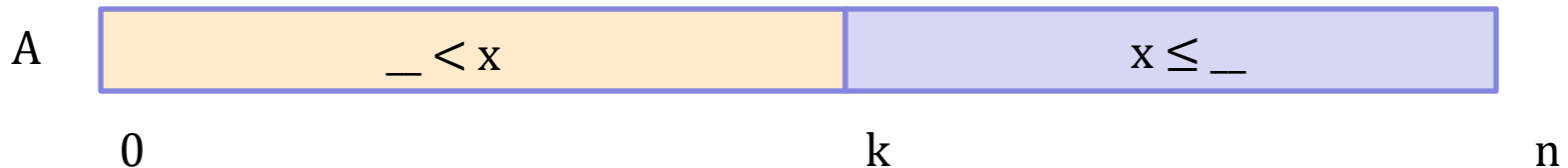
- a **weakening** of second part

Searching a Sorted Array: Starting Invariant

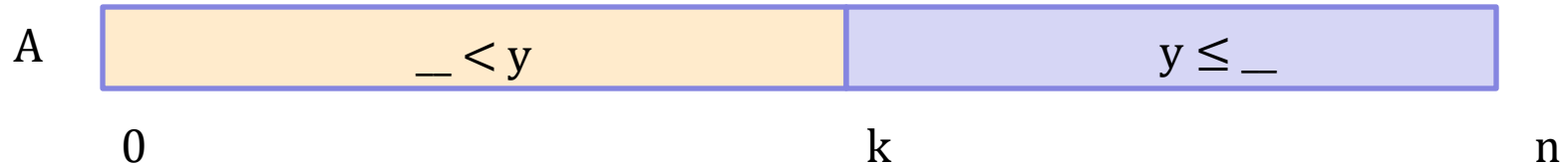
- Suppose we require A to be sorted:
 - precondition includes

$$A[j-1] \leq A[j] \text{ for any } 1 \leq j < n \quad (\text{where } n := A.\text{length})$$

- Want to find the index k where “ x ” would be...
 - picture would look like this:



Searching a Sorted Array: Weakened Invariant



- End with complete knowledge of $A[i]$ vs x
 - how can we describe *partial* knowledge?
 - know some elements are smaller and some larger



Loop Invariants with Arrays (3/3)

- **Previous example**

$\{\{ \text{Inv: } s = \text{sum}(S[0 .. j - 1]) \dots \}\}$ **sum of array**
 $\{\{ \text{Post: } s = \text{sum}(S[0 .. n - 1]) \}\}$

- **Linear search also fits this pattern:**

$\{\{ \text{Inv: } S[i] \neq y \text{ for any } 0 \leq i < j \}\}$ **search an array**
 $\{\{ \text{Post: } (S[i] = y) \text{ or } (S[i] \neq y \text{ for any } 0 \leq i < n) \}\}$

- **Binary search also still fits this pattern**

$\{\{ \text{Inv: } (S[i] < y \text{ for any } 0 \leq i < j) \text{ and } (y \leq S[i] \text{ for any } k \leq i < n) \}\}$
 $\{\{ \text{Post: } (S[i] < y \text{ for any } 0 \leq i < k) \text{ and } (y \leq S[i] \text{ for any } k \leq i < n) \}\}$

Loop Invariants in the Wild

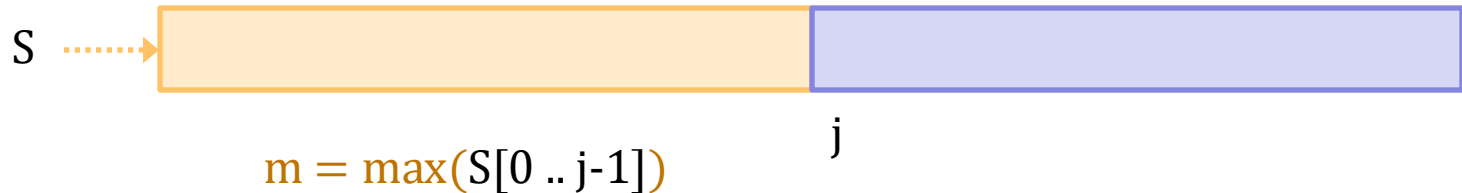
- **Heuristic for loop invariants: weaken the postcondition**
 - assertion that allows postcondition as a special case
 - must also allow states that are easy to prepare
- **421 covers complex heuristics for finding invariants...**
 - for 331, this heuristic is enough
 - (will give you the invariant for anything more complex)

Writing Loops

Code Reviews vs 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...

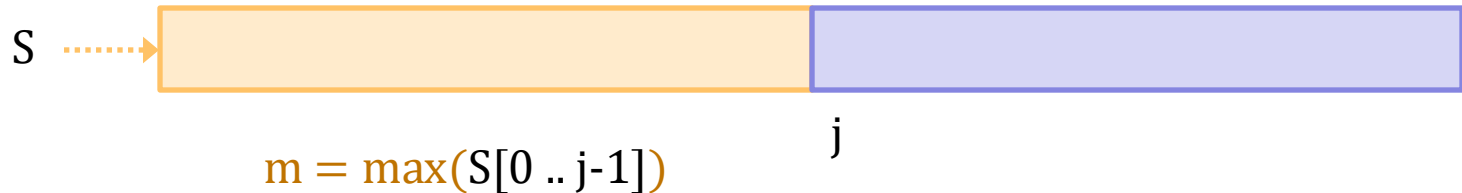
Max of an Array (1/7)



```
const max = (S: Array<bigint>) : bigint => {  
  let m = ??  
  let j = ??  
  // Inv: m = max(S[0 .. j-1])  
  while (??) {  
    ??  
    j = j + 1;  
  }  
  return m;  
};
```

How do we initialize m & j ?
 $m = \max(S[0 \dots 0])$ is easiest
What case is missing?

Max of an Array (2/7)

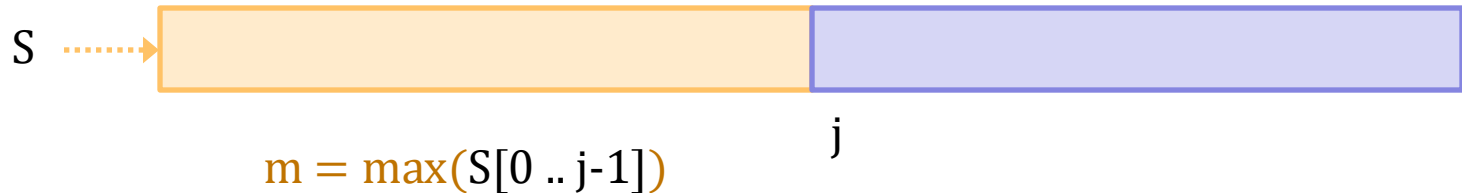


```
const max = (S: Array<bigint>) : bigint => {  
  if (S.length === 0) throw new Error('no elements');  
  let m = S[0];  
  let j = ??  
  // Inv: m = max(S[0 .. j-1])  
  while (??) {  
    ??  
    j = j + 1;  
  }  
  return m;  
};
```

How do we initialize j?

Want $m = \max(S[0 .. 0])$

Max of an Array (3/7)

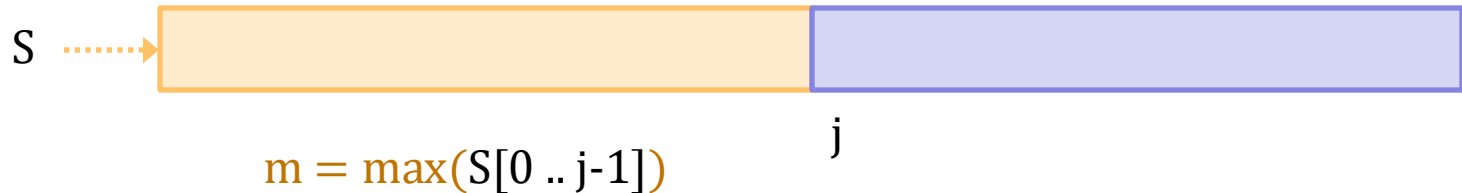


```
const max = (S: Array<bigint>) : bigint => {  
  if (S.length === 0) throw new Error('no elements');  
  let m = S[0];  
  let j = 1;  
  // Inv: m = max(S[0 .. j-1])  
  while (??) {  
    ??  
    j = j + 1;  
  }  
  return m;  
};
```

When do we exit?

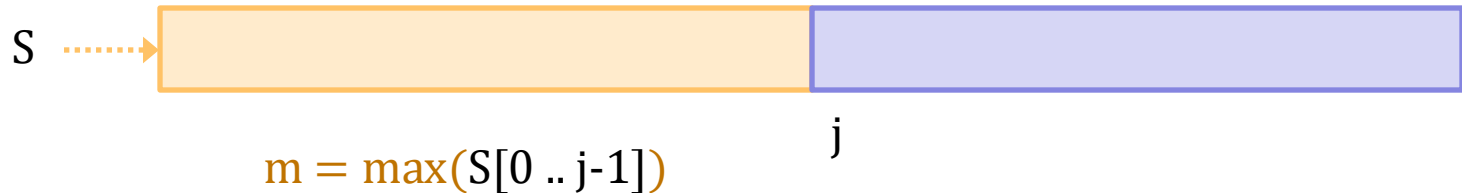
Want $m = \max(S[0 \dots n-1])$

Max of an Array (4/7)



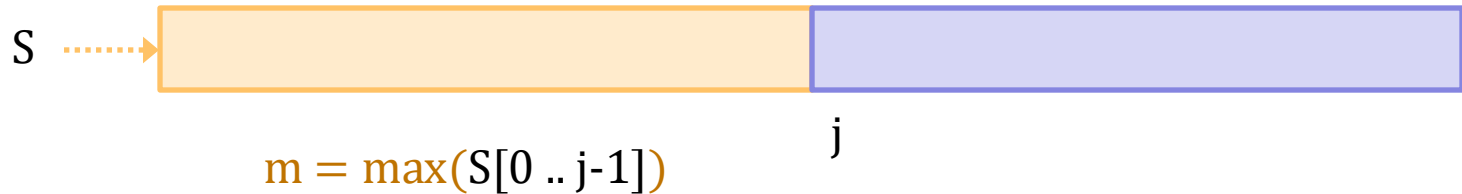
```
const max = (S: Array<bigint>) : bigint => {  
  if (S.length === 0) throw new Error('no elements');  
  let m = S[0];  
  let j = 1;  
  // Inv: m = max(S[0 .. j-1])  
  while (j !== S.length) {  
    ??  
    j = j + 1;  
  }  
  return m;  
};
```

Max of an Array (5/7)



```
const max = (S: Array<bigint>) : bigint => {  
  if (S.length === 0) throw new Error('no elements');  
  let m = S[0];  
  let j = 1;  
  // Inv: m = max(S[0 .. j-1])  
  while (j !== S.length) {  
    {{ m = max(S[0 .. j-1]) and j ≠ n }}  
    ??  
    {{ m = max(S[0 .. j]) }}  
    j = j + 1;  
  }
```

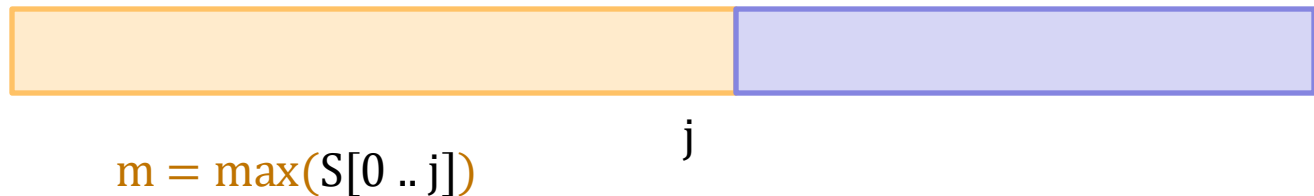
Max of an Array (6/7)



$\{ \{ m = \max(S[0 .. j-1]) \text{ and } j \neq n \} \}$

??

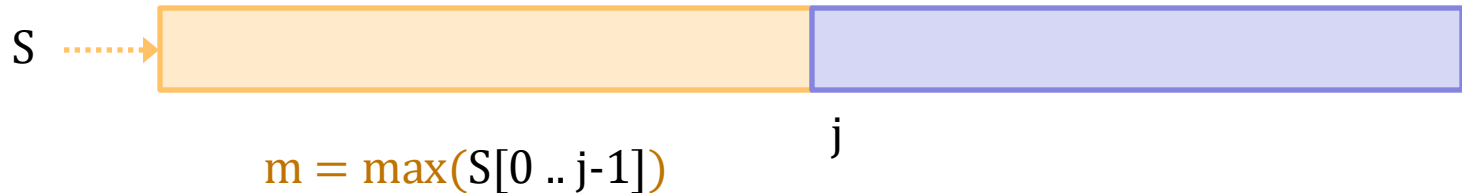
$\{ \{ m = \max(S[0 .. j]) \} \}$



How do we make the second one hold?

Set $m = S[j]$ iff $S[j] > m$

Max of an Array (7/7)



```
const max = (S: Array<bigint>) : bigint => {  
  if (S.length === 0) throw new Error('no elements');  
  let m = S[0];  
  let j = 1;  
  // Inv: m = max(S[0 .. j-1])  
  while (j !== S.length) {  
    if (S[j] > m)  
      m = S[j];  
    j = j + 1;  
  }  
  return m;  
};
```

Example: Sorting Negative, Zero, Positive

- Reorder an array so that
 - negative numbers come first, then zeros, then positives
(not necessarily fully sorted)

```
/**  
 * 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  
 */  
const sortPosNeg = (A: bigint[]): [bigint, bigint] =>
```

sortPosNeg on Wikipedia

Dutch national flag problem

🌐 4 languages ▾

Article [Talk](#)

Read [Edit](#) [View history](#) [Tools](#) ▾

From Wikipedia, the free encyclopedia

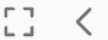
The **Dutch national flag problem**^[1] is a [computational problem](#) proposed by [Edsger Dijkstra](#).^[2] The [flag of the Netherlands](#) consists of three colors: red, white, and blue. Given balls of these three colors arranged randomly in a line (it does not matter how many balls there are), the task is to arrange them such that all balls of the same color are together and their collective color groups are in the correct order.

The solution to this problem is of interest for designing [sorting algorithms](#); in particular, variants of the [quicksort](#) algorithm that must be [robust to repeated elements](#) may use a three-way partitioning function that groups items less than a given key (red), equal to the key (white) and greater than the key (blue). Several solutions exist that have varying performance characteristics, tailored to sorting arrays with either small or large numbers of repeated elements.^[3]



sortPosNeg on LeetCode

[Description](#) | [Editorial](#) | [Solutions](#) | [Submissions](#)



75. Sort Colors

Medium

Topics

Companies

Hint

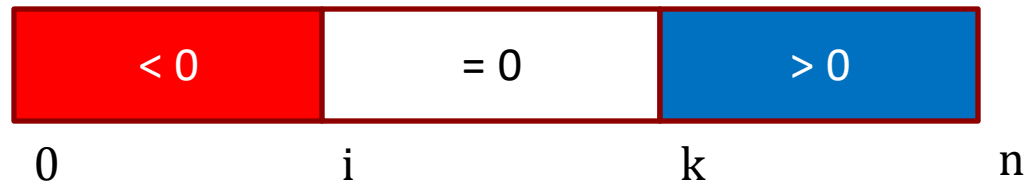
Given an array `nums` with `n` objects colored red, white, or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Drawing sortPosNeg (~ “Dutch flag problem”)

```
// @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 was our heuristic for guessing an invariant?
- weaken the postcondition

Potential Weaker sortPosNeg Invariants

How should we weaken this for the invariant?

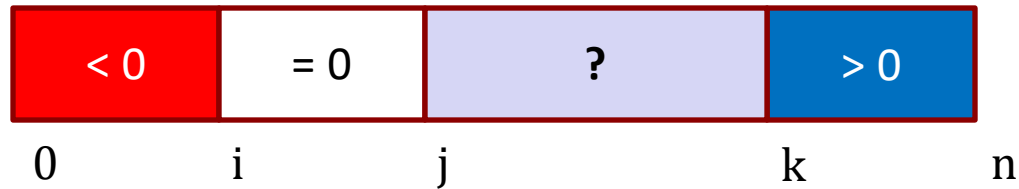
- needs allow elements with *unknown* values

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



Formalizing a “Visual Invariant”

Our Invariant:



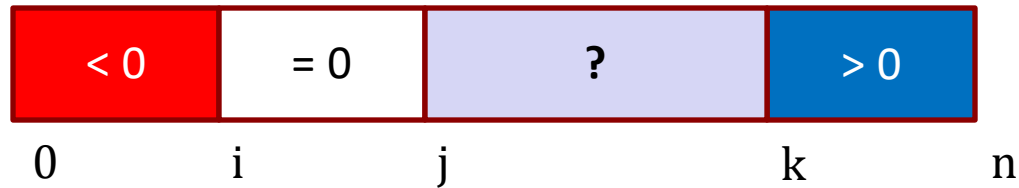
$A[\ell] < 0$ for any $0 \leq \ell < i$

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

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

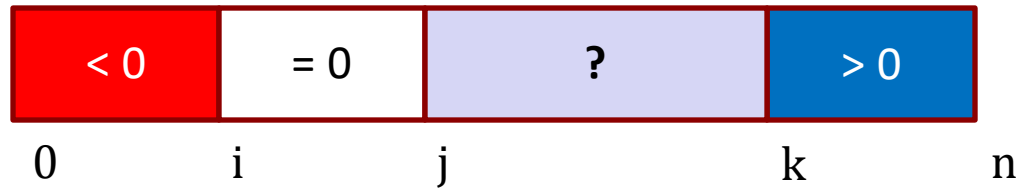
$A[\ell] > 0$ for any $k \leq \ell < n$

Sketching sortPosNeg: Loop Components

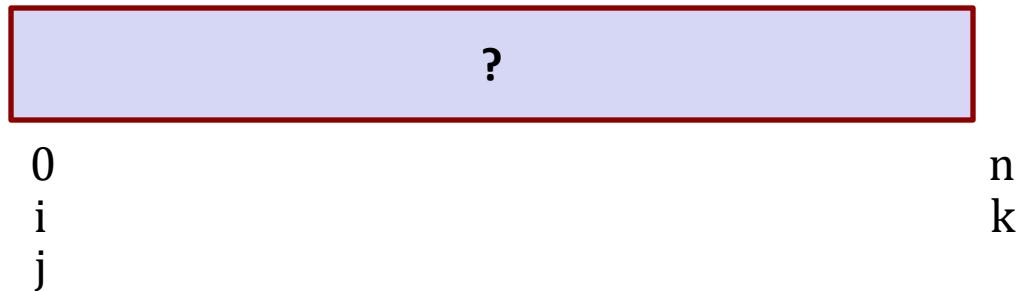


- Let's try figuring out the code to make it correct
- Figure out the code for
 - how to initialize
 - when to exit
 - loop body

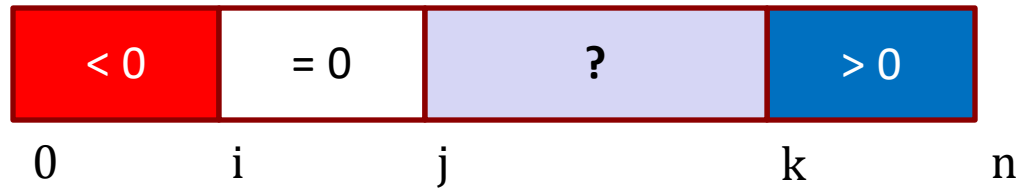
Sketching sortPosNeg: Initialization



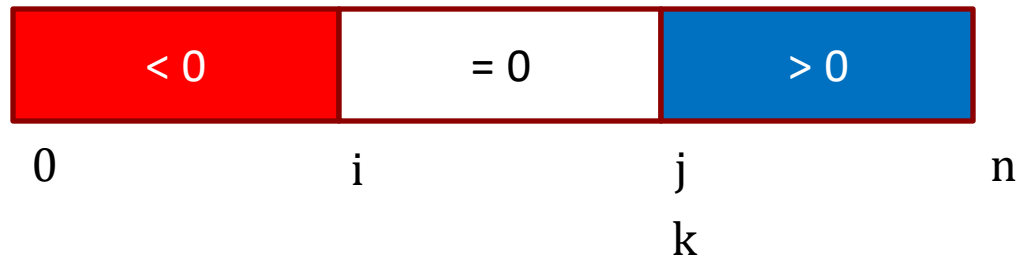
- Will have variables i , j , and k with $i \leq j \leq k$
- How do we set these to make it true initially?
 - we start out not knowing anything about the array values
 - set $i = j = 0$ and $k = n$



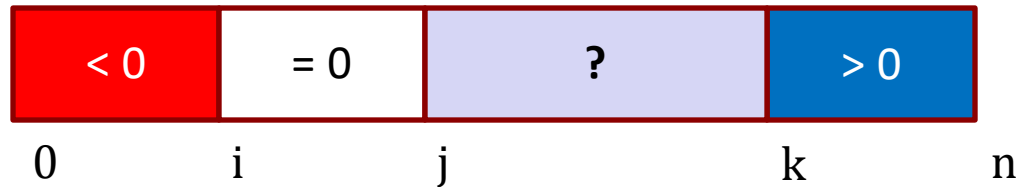
Sketching sortPosNeg: Exit Condition



- Set $i = j = 0$ and $k = n$ to make this hold initially
- When do we exit?
 - purple is empty if $j = k$

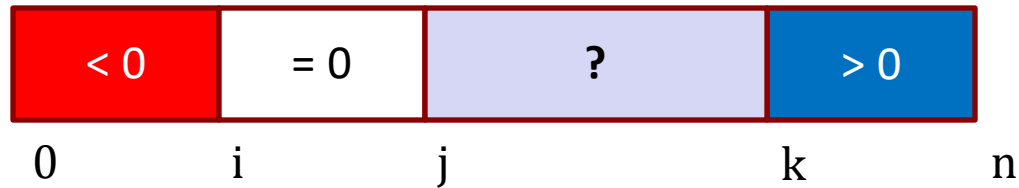


Filling In sortPosNeg's Implementation



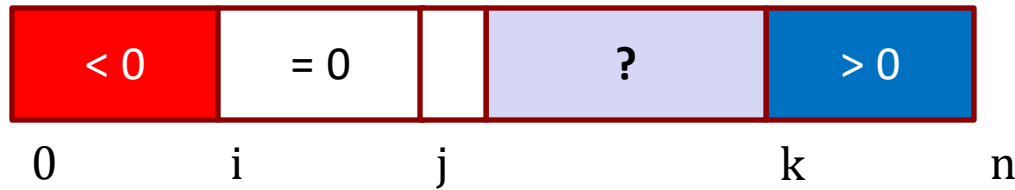
```
let i = 0;
let j = 0;
let k = A.length;
{{ Inv: A[ℓ] < 0 for any 0 ≤ ℓ < i and A[ℓ] = 0 for any i ≤ ℓ < j
      A[ℓ] > 0 for any k ≤ ℓ < n and 0 ≤ i ≤ j ≤ k ≤ n }}
while (j < k) {
    ...
}
{{ A[ℓ] < 0 for any 0 ≤ ℓ < i and A[ℓ] = 0 for any i ≤ ℓ < j
  A[ℓ] > 0 for any j ≤ ℓ < n }}
return [i, j];
```

Sketching sortPosNeg: Progress by Shrinking

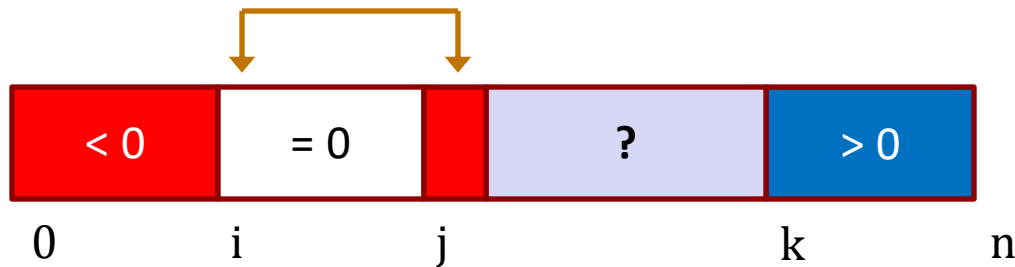


- How do we make progress?
 - try to increase j by 1 or decrease k by 1
- Look at $A[j]$ and figure out where it goes
- What to do depends on $A[j]$
 - could be < 0 , $= 0$, or > 0

Sketching sortPosNeg: Progress by Cases



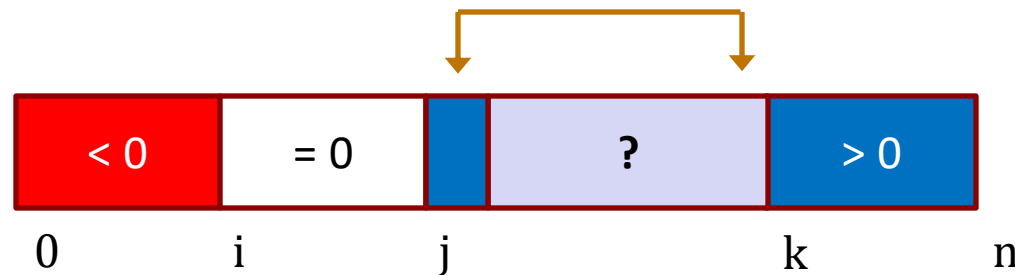
Set $j = j_0 + 1$



Swap $A[i]$ and $A[j]$

Set $i = i_0 + 1$

and $j = j_0 + 1$



Swap $A[j]$ and $A[k-1]$

Set $k = k_0 - 1$

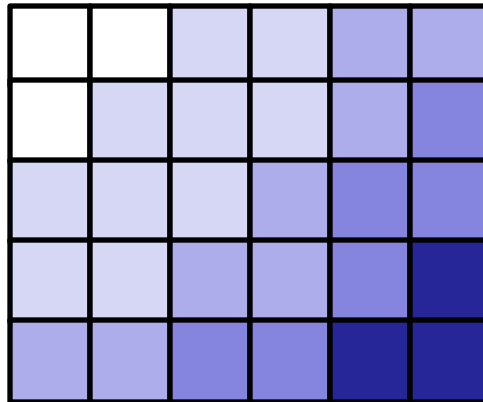
Full sortPosNeg Implementation

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

```
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;  
    }  
}
```

Sorted Matrix Search

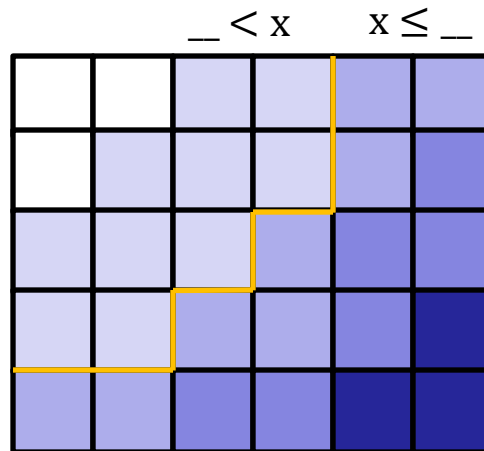
Given a sorted matrix M , with m rows and n cols, where every row and every column is sorted, find out whether a given number x is in the matrix



(darker color means larger)

Sorted Matrix Search: Solution Idea

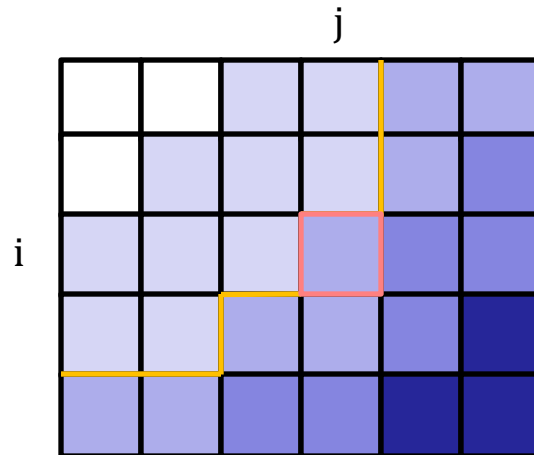
Given a sorted matrix M , with m rows and n cols, where every row and every column is sorted, find out whether a given number x is in the matrix



Idea: Trace the contour between the numbers $\leq x$ and $> x$ in each row to see if x appears.

Sorted Matrix Search: Idea as Invariant

Given a sorted matrix M , with m rows and n cols, where every row and every column is sorted, find out whether a given number x is in the matrix



Invariant: at the left-most entry with $x \leq _$ in the row
– for each row i , this holds for exactly one column j

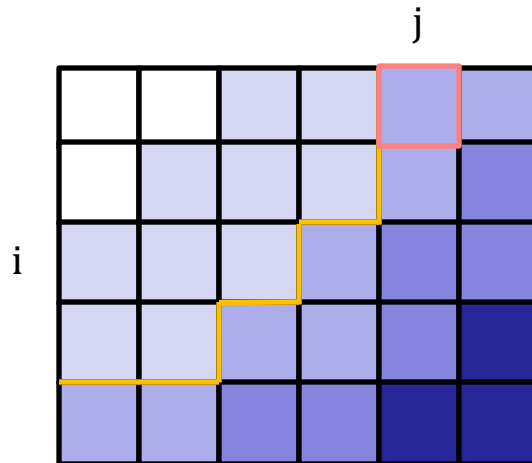
Sorted Matrix Search: Initialization

Invariant: at the left-most entry with $x \leq _$ in the row

- for each row i , this holds for exactly one column j

Initialization: how do we get this to hold for $i = 0$?

- could be anywhere in the first row

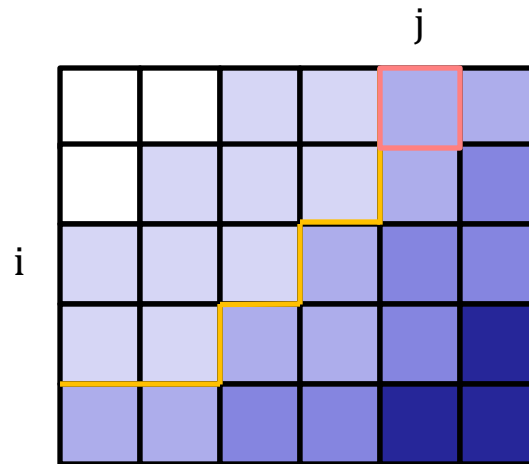


Need to *search* to find this location

Sorted Matrix Search: Row Subgoal

New Goal: find smallest j with $x \leq M[0, k]$ for any $j \leq k < n$

– will need a loop...



How do we find an invariant for that loop?

- try **weakening** this assertion (allow any j , not just smallest)
- decrease j until $x \leq M[0, j-1]$ does not hold

Sorted Matrix Search: Row Subgoal Initialization

New Goal: find smallest j with $x \leq M[0, k]$ for any $j \leq k < n$

```
let i = 0;
```

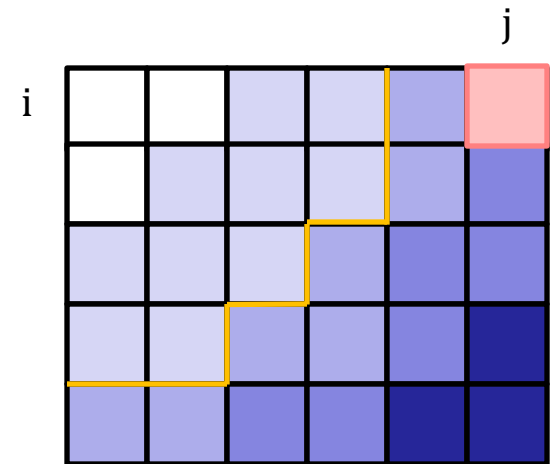
```
let j = ??
```

```
{{ Inv:  $x \leq M[0, k]$  for any  $j \leq k < n$  }}
```

```
while (??)
```

```
??
```

```
{{ Post:  $M[0, k] < x$  for any  $0 \leq k < j$  and  $x \leq M[0, k]$  for any  $j \leq k < n$  }}
```



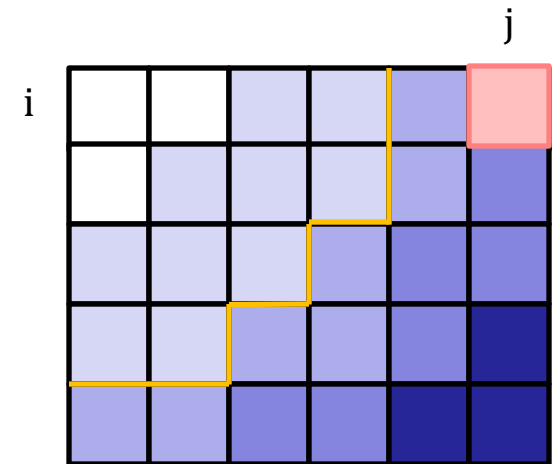
How do we set j to make Inv hold initially?

- range is empty when $j = n$

Sorted Matrix Search: Row Subgoal Exit

New Goal: find smallest j with $x \leq M[0, k]$ for any $j \leq k < n$

```
let i = 0;  
let j = n;  
{ { Inv:  $x \leq M[0, k]$  for any  $j \leq k < n$  } }  
while (??)  
    ??
```



```
{ { Post:  $M[0, k] < x$  for any  $0 \leq k < j$  and  $x \leq M[0, k]$  for any  $j \leq k < n$  } }
```

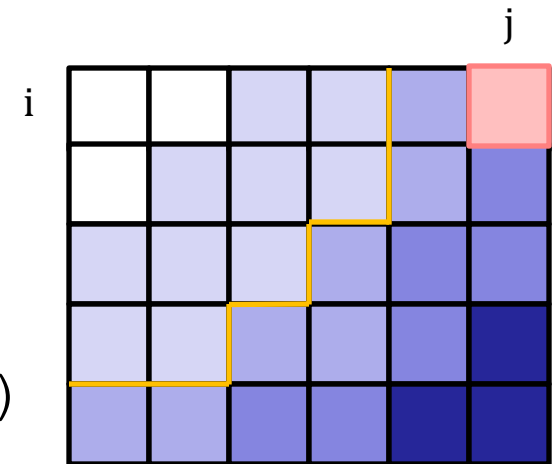
How do we exit so that the postcondition holds?

- can no longer decrease j when $j = 0$ or $M[0, j-1] < x$

Sorted Matrix Search: Row Subgoal Loop (1/4)

New Goal: find smallest j with $x \leq M[0, k]$ for any $j \leq k < n$

```
let i = 0;  
let j = n;  
{ { Inv:  $x \leq M[0, k]$  for any  $j \leq k < n$  } }  
while (j > 0 && x <= M[0][j-1])  
    ??
```



{ { Post: $M[0, k] < x$ for any $0 \leq k < j$ and $x \leq M[0, k]$ for any $j \leq k < n$ } }

Anything needed in the loop body?
(That is, other than $j = j - 1$?)

Sorted Matrix Search: Row Subgoal Loop (2/4)

New Goal: find smallest j with $x \leq M[0, k]$ for any $j \leq k < n$

{{ Inv: $x \leq M[0, k]$ for any $j \leq k < n$ }}

while ($j > 0 \ \&\& \ x \leq M[0][j-1]$) {

 {{ $x \leq M[0, k]$ for any $j \leq k < n$ and $j > 0$ and $x \leq M[0, j-1]$ }}

 ??

$j = j - 1;$

 {{ $x \leq M[0, k]$ for any $j \leq k < n$ }}

}

Sorted Matrix Search: Row Subgoal Loop (3/4)

New Goal: find smallest j with $x \leq M[0, k]$ for any $j \leq k < n$

$\{\{ \text{Inv: } x \leq M[0, k] \text{ for any } j \leq k < n \}\}$

while $(j > 0 \ \&\& \ x \leq M[0][j-1]) \ \{$

$\{\{ x \leq M[0, k] \text{ for any } j \leq k < n \text{ and } j > 0 \text{ and } x \leq M[0, j-1] \}\}$

??

 $\{\{ x \leq M[0, k] \text{ for any } j - 1 \leq k < n \}\}$

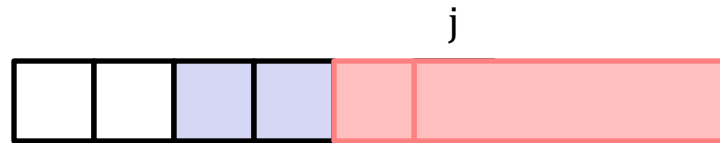
$j = j - 1;$

$\{\{ x \leq M[0, k] \text{ for any } j \leq k < n \}\}$

}

Sorted Matrix Search: Row Subgoal Loop (4/4)

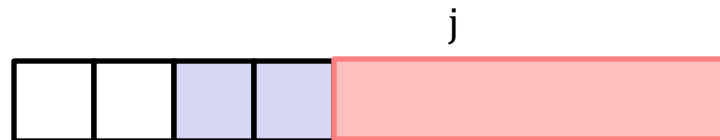
New Goal: find smallest j with $x \leq M[0, k]$ for any $j \leq k < n$



$\{ \{ x \leq M[0, k] \text{ for any } j \leq k < n \text{ and } j > 0 \text{ and } x \leq M[0, j-1] \} \}$

??

$\{ \{ x \leq M[0, k] \text{ for any } j - 1 \leq k < n \} \}$



Nothing is missing!

Sorted Matrix Search: Row Subgoal Done!

New Goal: find smallest j with $x \leq M[0, k]$ for any $j \leq k < n$

```
let i = 0;
```

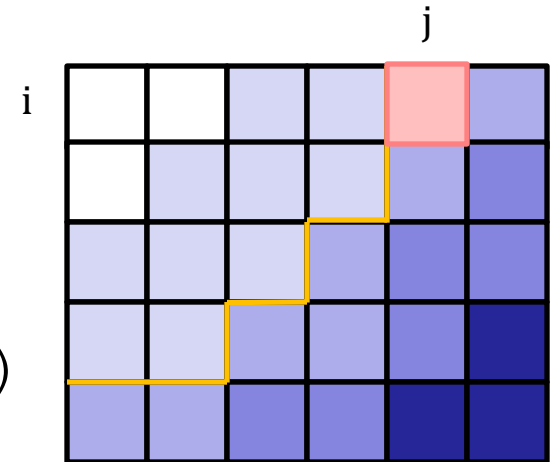
```
let j = n;
```

```
{{ Inv:  $x \leq M[0, k]$  for any  $j \leq k < n$  }}
```

```
while (j > 0 && x <= M[0][j-1])
```

```
    j = j - 1;
```

```
{{ Post:  $M[0, k] < x$  for any  $0 \leq k < j$  and  $x \leq M[0, k]$  for any  $j \leq k < n$  }}
```

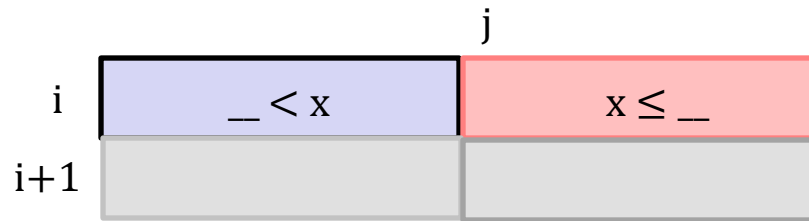


Can now check if $M[0, j] = x$

- if not, then it is not in the first row
- move on to the second row...

Sorted Matrix Search: Moving Rows (1/3)

Moving from row i to row $i+1$

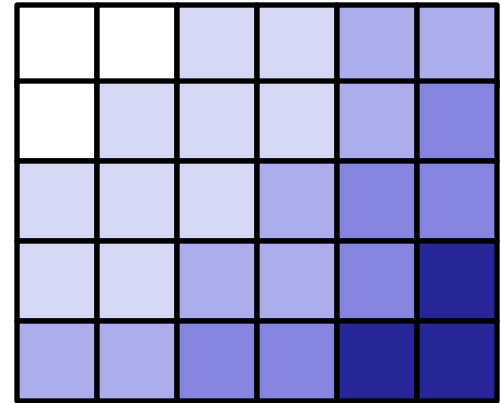
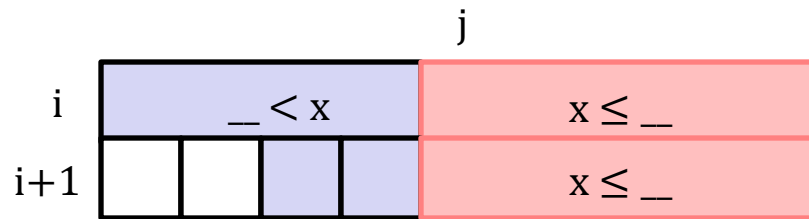


What does *vertical* sorting tell us about row $i+1$?

- right side is guaranteed to satisfy " $x \leq _$ "
- left side **not** guaranteed to satisfy " $_ < x$ "

Sorted Matrix Search: Moving Rows (2/3)

Moving from row i to row $i+1$

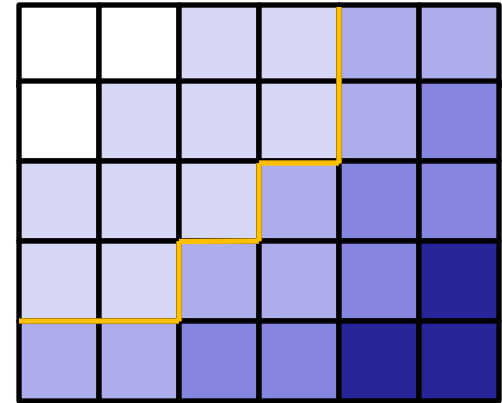
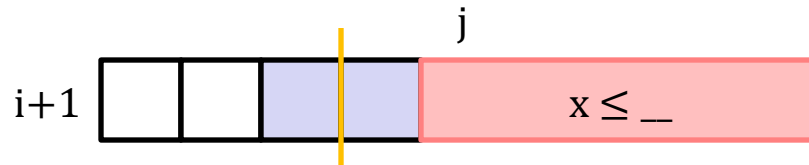


What does *vertical* sorting tell us about row $i+1$?

- right side is guaranteed to satisfy " $x \leq _$ "
- left side **not** guaranteed to satisfy " $_ < x$ "

Sorted Matrix Search: Moving Rows (3/3)

Moving from row i to row $i+1$



How do we restore the invariant?

- find the index j with $M[i+1, j-1] < x \leq M[i+1, j]$

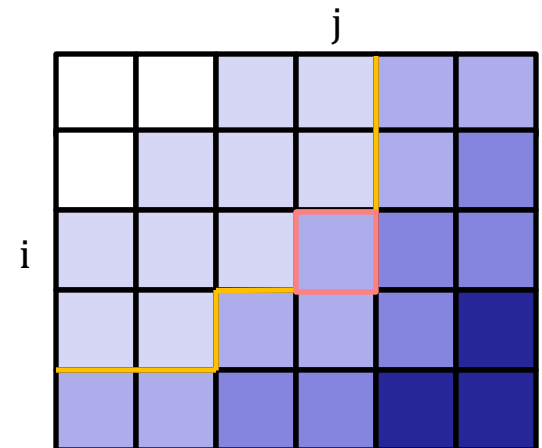
This is the same problem as before!

- move left until beginning or $M[i+1, j-1] < x$ holds

Sorted Matrix Search: Moving Columns (1/3)

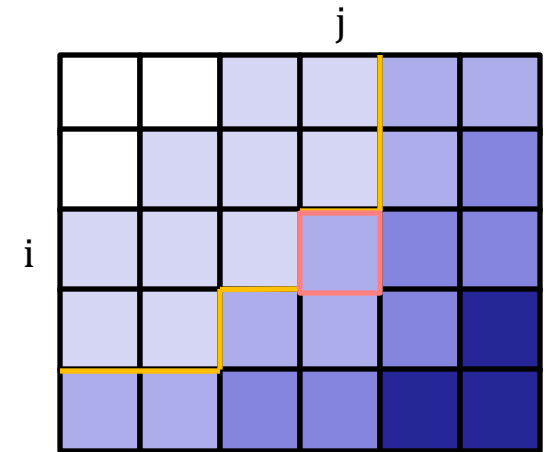
```
let i = 0;
let j = n;
... move j to left...
if (M[i][j] == x) return true;
{{ Inv: (x is not in row k for any  $0 \leq k \leq i$ ) and
      ( $M[i, k] < x$  for any  $0 \leq k < j$ ) and ( $x \leq M[i, k]$  for any  $j \leq k < n$ ) }}
while (i+1 != n) {
    ...
}
return false;
```

Inv says we ruled out rows $0 \dots i$
and col j is line between $_ < x$ and $x \leq _$



Sorted Matrix Search: Moving Columns (2/3)

```
let i = 0;
let j = n;
... move j to left...
if (M[i][j] == x) return true;
{{ Inv: (x is not in row k for any  $0 \leq k \leq i$ ) and
      ( $M[i, k] < x$  for any  $0 \leq k < j$ ) and ( $x \leq M[i, k]$  for any  $j \leq k < n$ ) }}
while (i+1 != n) {
    i = i + 1;
    ... move j to the left...
    if (M[i][j] == x) return true;
}
return false;
```



We can avoid writing this code twice
(without writing a separate function)...

Don't try this at home!

Sorted Matrix Search: Moving Columns (3/3)

```
let i = 0;
let j = n;
while (i != n) {
    ... move j to left...
    if (M[i][j] == x) return true;
    {{ Inv: (x is not in row k for any  $0 \leq k \leq i$ ) and
        ( $M[i, k] < x$  for any  $0 \leq k < j$ ) and ( $x \leq M[i, k]$  for any  $j \leq k < n$ ) }}
    i = i + 1;
}
return false;
```

Loop condition was also changed

Inv is now checked in the middle of the loop!

Sorted Matrix Search: Full Code & Assertions

```
let i = 0;
```

Final version is 9 lines of code.

```
let j = n;
```

Requires 6 lines of invariant assertions!

```
while (i != n) {
```

```
  {{ Inv:  $x \leq M[i, k]$  for any  $j \leq k < n$  }}
```

```
  while (j > 0 && x <= M[i][j-1])
```

```
    j = j - 1;
```

```
  if (M[i][j] == x)
```

```
    return true;
```

```
  {{ Inv: (x is not in row k for any  $0 \leq k \leq i$ ) and
```

```
    ( $M[i, k] < x$  for any  $0 \leq k < j$ ) and ( $x \leq M[i, k]$  for any  $j \leq k < n$ ) }}
```

```
  i = i + 1;
```

```
}
```

```
return false;
```

Wrapping Up Arrays

- We design invariants to tell us everything we need to know to prove our post condition
 - Precondition facts aren't (explicitly) copied in
 - Instead, we prove (P implies $Inv1$) and ($Inv1$ and $\neg cond$ implies Q)

```
    {{ P }}  
    {{ Inv1 }}  
    while (cond) {  
    → {{ P1: Inv1 and cond }}  
      S  
    → {{ Q1: Inv1 }}  
    }  
    {{ Q }}
```

Reasoning with Multiple Loops

- Applies with multiple loops also
 - Can treat them as distinct units, except in the middle

```
  {{ Inv1 }}  
  while (cond1) {  
    S  
  }  
  {{ Inv1 and !cond1 }}  
  {{ Inv2 }}  
  while (cond2) {  
    S  
  }  
  {{ Inv2 and cond2 }}
```

Must show this holds

Reasoning with Nested Loops

- and nested loops
 - still need to prove **Inv1** holds at the end of outer loop

```

{{ Inv1 }}
while (cond) {
    {{ Inv1 and cond1 }}
    {{ Inv2 }}
    while (cond2) {
        {{ Inv2 and cond2 }}
        S
    }
    {{ Inv2 and !cond2 }}
    {{ Q1: Inv1 }}
}

```

Must show this holds

Must show this holds