CSE 331
Software Design & Implementation

James Wilcox & Kevin Zatloukal

Fall 2022

Lecture 4½ – Reasoning Wrap-up

Administrivia

- HW2 to be released tonight
  - includes coding part
  - (also has a written problem, independent of the rest)

- Section tomorrow will get you started on coding part

- Bring your **laptop** (if that is where you plan to work)
  - go through the pre-section setup **beforehand**

A Harder Example

Example: Dutch National Flag

Given an array of red, white, and blue pebbles, sort the array so the red pebbles are at the front, the white pebbles are in the middle, and the blue pebbles are at the end

Edsgar Dijkstra

Pre- and post-conditions

Precondition: Any mix of red, white, and blue

Postcondition:

- red then white then blue
- number of each color is unchanged

Pre- and post-conditions

Precondition: Any mix of red, white, and blue

Postcondition:

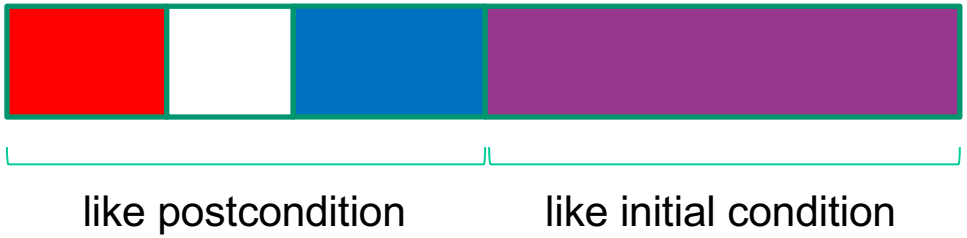
- red then white then blue
- number of each color is unchanged

Want an invariant with

- postcondition as a special case
- precondition as a special case (or easy to change to one)

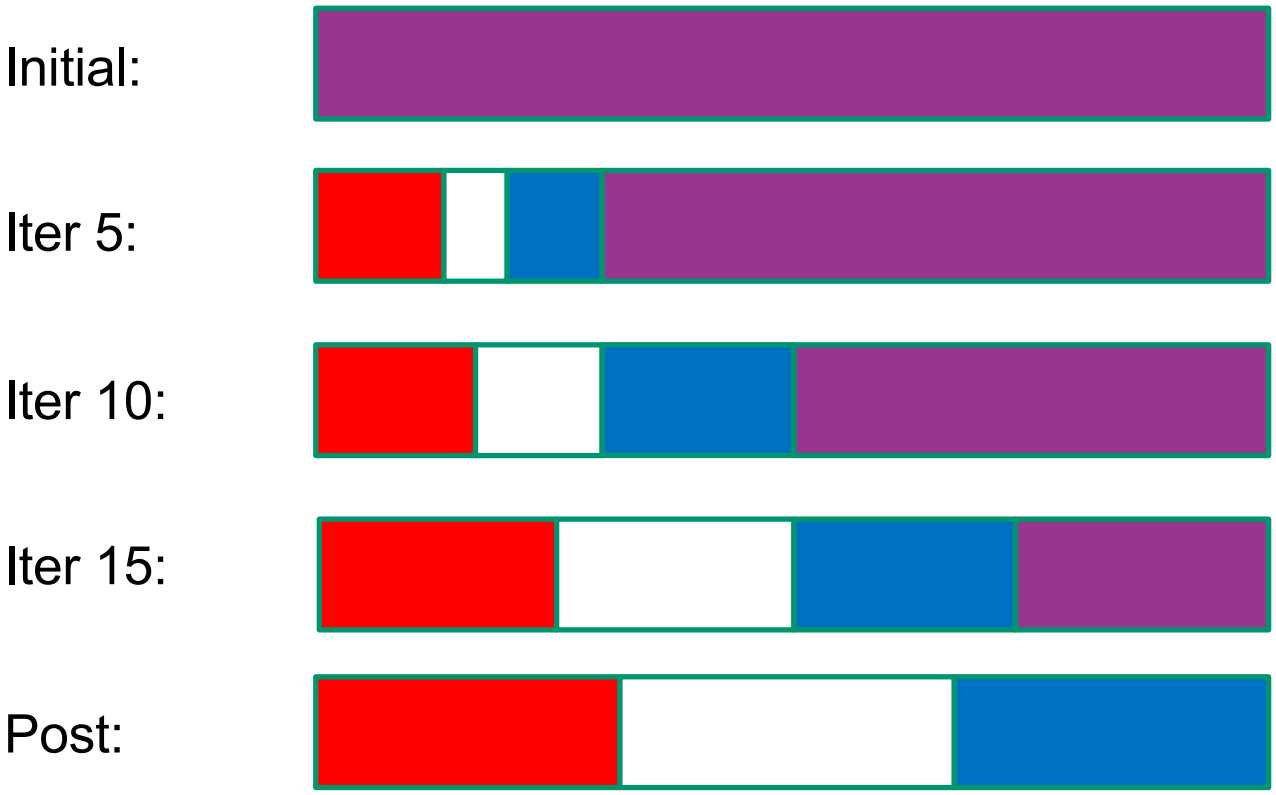
Example: Dutch National Flag

The first idea that comes to mind:



Example: Dutch National Flag

The first idea that comes to mind works.



Other potential invariants

Any of these choices work, making the array more-and-more partitioned as you go:

Precise Invariant

Need indices to refer to the split points between colors

– call these i, j, k

Loop Invariant:

- 0 <= i <= j <= k <= n <= A.length
- A[0], …, A[i-1] are red
- A[i], …, A[j-1] are white
- A[k], …, A[n-1] are blue

No constraints on A[j], ..., A[k-1]

Dutch National Flag Code

Invariant:

Initialization?

Dutch National Flag Code

Invariant:

Initialization:

- i = j = 0 and k = n

Dutch National Flag Code

Invariant:

Initialization:

- i = j = 0 and k = n

Termination condition?

Dutch National Flag Code

Invariant:

Initialization:

- i = j = 0 and k = n

Termination condition:

- j = k

Dutch National Flag Code

```
int i = 0, j = 0;
int k = n;
```

{{ Inv: 0 <= i <= j <= k <= n and A[0], …, A[i-1] are red and ... }}

```
while (j != k) {

   ??

}
```

need to get `j` closer to `k`...
let's try increasing `j` by 1

Dutch National Flag Code

Three cases depending on the value of A[j]:

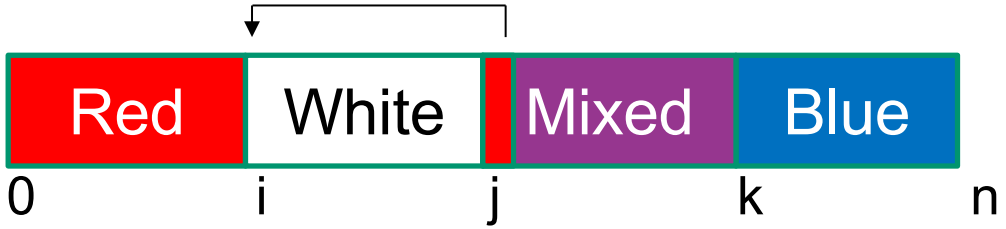
A[j] is either red, white, or blue

Dutch National Flag Code

Three cases depending on the value of A[j]:

white

red



blue

Dutch National Flag Code

```
int i = 0, j = 0;
int k = n;
```

{{ Inv: 0 <= i <= j <= k <= n and A[0], …, A[i-1] are red and ... }}

```
while (j != k) {
  if (A[j] is white) {
     j = j+1;
  } else if (A[j] is blue) {
     swap A[j], A[k-1];
     k = k - 1;
  } else { // A[j] is red
     swap A[i], A[j];
     i = i + 1;
     j = j + 1;
  }
}
```

Binary Search

Example: Binary Search

Problem: Given a sorted array A and a number x, find index of x (or where it would be inserted) in A.

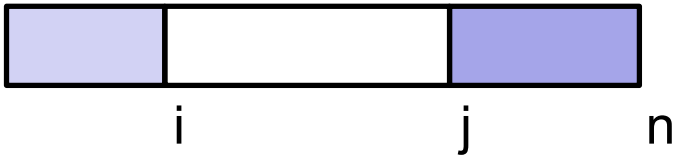
Idea: Look at A[n/2] to figure out if x is in A[0], A[1], ..., A[n/2] or in A[n/2+1], ..., A[n-1]. Narrow the search for x on each iteration.

(This is an algorithm where you probably still need to go line-by-line even as you get faster at reasoning...)

Example: Binary Search

Problem: Given a sorted array A and a number x, find index of x (or where it would be inserted) in A.

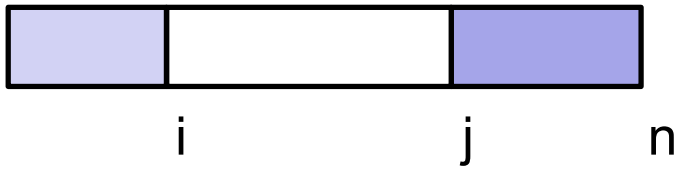
Idea: Look at A[n/2] to figure out if x is in A[0], A[1], ..., A[n/2] or in A[n/2+1], ..., A[n-1]. Narrow the search for x on each iteration.



Loop Invariant: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1]

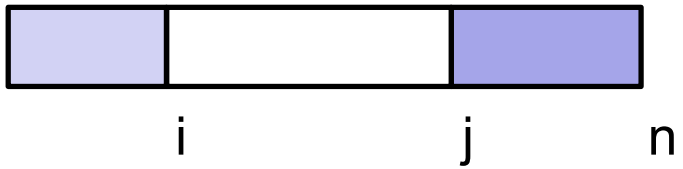
- A[i], ..., A[j-1] is the part where we don't know relation to x

Binary Search Code



Initialization?

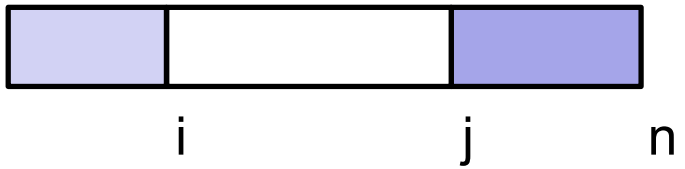
Binary Search Code



Initialization:

- i = 0 and j = n
- white region is the whole array

Binary Search Code



Initialization:

- i = 0 and j = n
- white region is the whole array

Termination condition:

- i = j
- white region is empty
- if x is in the array, it is A[i-1]
  - if there are multiple copies of x, this returns the *last*

Binary Search Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1] and A is sorted }}
```
while (i != j) {
```

Fall 2022

```
   // need to bring i and j closer together...
   // (e.g., increase i or decrease j)



}
```
{{ A[0], ..., A[i-1] <= x < A[i], ..., A[n-1] }}

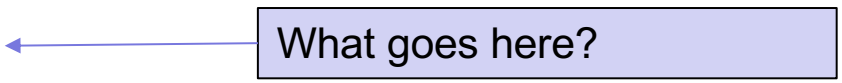
Binary Search Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1] and A is sorted }}
```
while (i != j) {
  int m = (i + j) / 2;
  if (A[m] <= x) {

  } else {


  }
}
```
{{ A[0], ..., A[i-1] <= x < A[i], ..., A[n-1] }}

Look at the element half way between i and j

Binary Search Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1] and A is sorted }}
```
while (i != j) {
  int m = (i + j) / 2;
  if (A[m] <= x) {
    ??
  } else {

  }
}
```
{{ A[0], ..., A[i-1] <= x < A[i], ..., A[n-1] }}



Binary Search Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1] and A is sorted }}
```
while (i != j) {
  int m = (i + j) / 2;
  if (A[m] <= x) {
    i = m + 1;
  } else {


  }
}
```
{{ A[0], ..., A[i-1] <= x < A[i], ..., A[n-1] }}

Since i-1 = m, we have A[i-1] = A[m] <= x
Why do we have A[0] <= … <= A[i-1]?

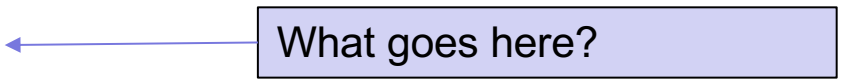
Binary Search Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1] and A is sorted }}
```
while (i != j) {
  int m = (i + j) / 2;
  if (A[m] <= x) {
    i = m + 1;
  } else {


  }
}
```
{{ A[0], ..., A[i-1] <= x < A[i], ..., A[n-1] }}

invariant satisfied since A[i-1] = A[m] <= x and A is sorted so A[0] <= … <= A[m]

Binary Search Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1] and A is sorted }}
```
while (i != j) {
  int m = (i + j) / 2;
  if (A[m] <= x) {
    i = m + 1;
  } else {
    ??
  }
}
```
{{ A[0], ..., A[i-1] <= x < A[i], ..., A[n-1] }}



Binary Search Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1] and A is sorted }}
```
while (i != j) {
  int m = (i + j) / 2;
  if (A[m] <= x) {
    i = m + 1;
  } else {
    j = m;
  }
}
```
{{ A[0], ..., A[i-1] <= x < A[i], ..., A[n-1] }}

invariant satisfied since x < A[m] = A[j]
(and A is sorted so A[m] <= ... <= A[n-1])

Binary Search Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1] and A is sorted }}
```
while (i != j) {
  int m = (i + j) / 2;
  if (A[m] <= x) {
    i = m + 1;
  } else {
    j = m;
  }
}
```
{{ A[0], ..., A[i-1] <= x < A[i], ..., A[n-1] }}

Does this always terminate?

Binary Search Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1] and A is sorted }}
```
while (i != j) {
  int m = (i + j) / 2;
  if (A[m] <= x) {
    i = m + 1;
  } else {
    j = m;
  }
}
```
{{ A[0], ..., A[i-1] <= x < A[i], ..., A[n-1] }}

Must satisfy i <= m < j (Why?)

Binary Search Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1] and A is sorted }}
```
while (i != j) {
  int m = (i + j) / 2;
  if (A[m] <= x) {
    i = m + 1;
  } else {
    j = m;
  }
}
```
{{ A[0], ..., A[i-1] <= x < A[i], ..., A[n-1] }}

Must satisfy i <= m < j
so i increases or j decreases
on every iteration

Binary Search Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], ..., A[i-1] <= x < A[j], ..., A[n-1] and A is sorted }}
```
while (i != j) {
  int m = (i + j) / 2;
  if (A[m] <= x) {
    i = m + 1;
  } else {
    j = m;
  }
}
```
{{ A[0], ..., A[i-1] <= x < A[i], ..., A[n-1] }}

Is that all we need to do?

Reasoning Summary

Reasoning Summary

- Checking correctness can be a mechanical process
  - using forward or backward reasoning
- This requires that loop invariants are provided
  - those cannot be produced automatically
- Provided you document your loop invariants,
  it should not be too hard for someone else to review your code

Documenting Loop Invariants

- Write down loop invariants for all non-trivial code
- They are often best avoided for “for each” loops:

```
{{ Inv: printed all the strings seen so far }}
for (String s : L)
  System.out.println(s);
```

Documenting Loop Invariants

- Write down loop invariants for all non-trivial code

- They are often best avoided for “for each” loops:

```
// Print the strings in L, one per line.
for (String s : L)
  System.out.println(s);
```

Documenting Loop Invariants

- Write down loop invariants for all non-trivial code
- They are often best avoided for “for each” loops.
- Invariants are more helpful when a variable incorporates information from multiple iterations
  - e.g., {{ s = A[0] + … + A[i-1] }}
- *Use your best judgement!*

Reasoning Summary

- Correctness: tools, inspection, testing
  - need all three to ensure high quality
  - especially cannot leave out inspection

- Inspection (by reasoning) means
  - reasoning through your own code
  - do code reviews

- Practice!
  - essential skill for professional programmers

Reasoning Summary

- You will eventually do this in your head for most code

- Formalism remains useful
  - especially tricky problems
  - interview questions (often tricky)
    - see last example…

Next Topic…

A Problem

"Complete this method such that it returns the location of the largest value in the first **`n`** elements of the array **`arr`**."

```
int maxLoc(int[] arr, int n) {
    ...
}
```

One Solution

```
int maxLoc(int[] arr, int n) {
  int maxIndex = 0;
  int maxValue = arr[0];
  // Inv: maxValue = max of arr[0] .. arr[i-1] and
  //      maxValue = arr[maxIndex]
  for (int i = 1; i < n; i++) {
    if (arr[i] > maxValue) {
      maxIndex = i;
      maxValue = arr[i];
    }
  }
  return maxIndex;
}
```

Is this code correct?

What if n = 0?

What if n > arr.length?

What if there are two maximums?

A Problem

"Complete this method such that it returns the location of the largest value in the first **`n`** elements of the array **`arr`**."

```
int maxLoc(int[] arr, int n) {
    ...
}
```

Could we write a specification so that this is a **correct** solution?

- precondition that n > 0
- throw `ArrayOutOfBoundsException` if n > arr.length
- return smallest index achieving maximum

Morals

- You can all write the code correctly

- Writing the specification was harder than the code
  - multiple choices for the "right" specification
    - must carefully think through corner cases
  - once the specification is chosen, code is straightforward
  - (both of those will be recurrent themes)

- Some math (e.g. "if n <= 0") often shows up in specifications
  - English ("if n is less or equal to than 0") is often worse

How to Check Correctness

- Step 1: need a **specification** for the function
  - can't argue correctness if we don't know what it should do
  - surprisingly difficult to write!

- Step 2: determine whether the code meets the specification
  - apply **reasoning**
  - usually easy with the tools we learned

Interview Question

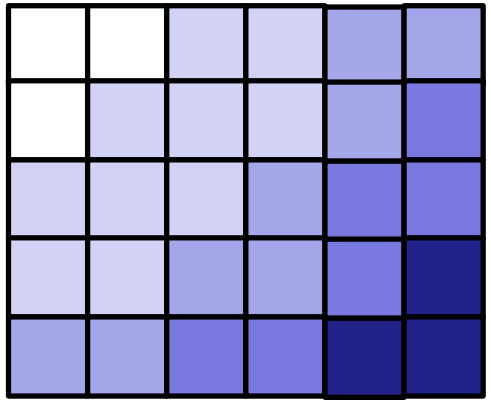
Sorted Matrix Search

Problem Description

Given a matrix M (of size m x n), where every row and every column is sorted, find out whether a given number x is in the matrix.

Sorted Matrix Search

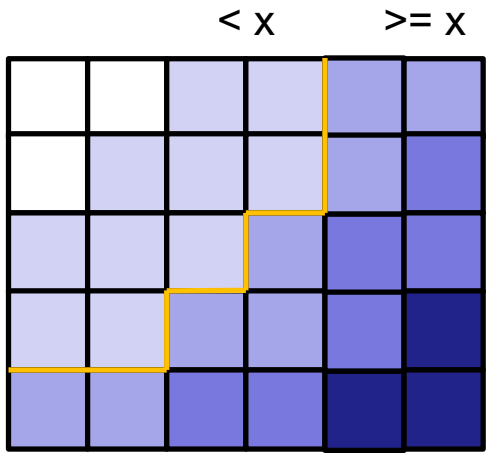
Given a sorted matrix M (of size m x n), where every row and every column is sorted, find out whether a given number x is in the matrix.



(darker color means larger)

Sorted Matrix Search

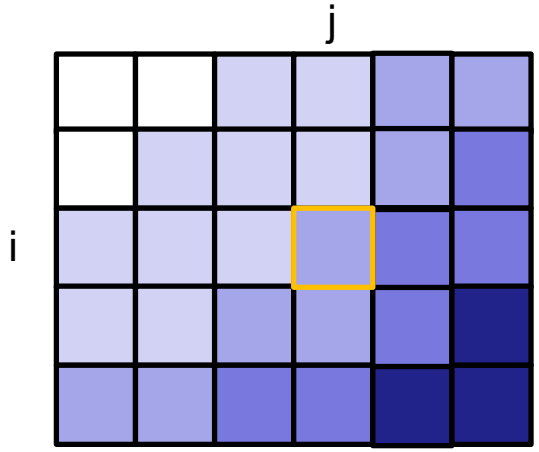
Given a sorted matrix M (of size m x n), where every row and every column is sorted, find out whether a given number x is in the matrix.



(darker color means larger)

(One) **Idea**: Trace the contour between the numbers ≤ x and > x in each row to see if x appears.

Sorted Matrix Search Code



Partial Invariant: M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1]

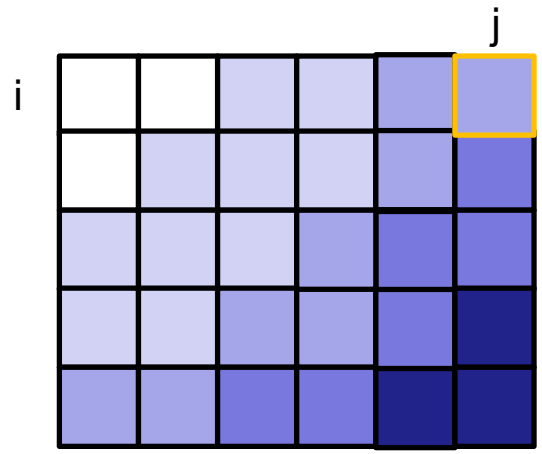
- for each i, holds for exactly one j
- holds when we are in the right spot in row i

"..." notation automatically handles special cases:

- if j = 0, nothing to the left ("<" constraint is vacuous)
- if j = n, nothing to the right ("≤" contraint is vacuous)

Sorted Matrix Search Code

Initialization:



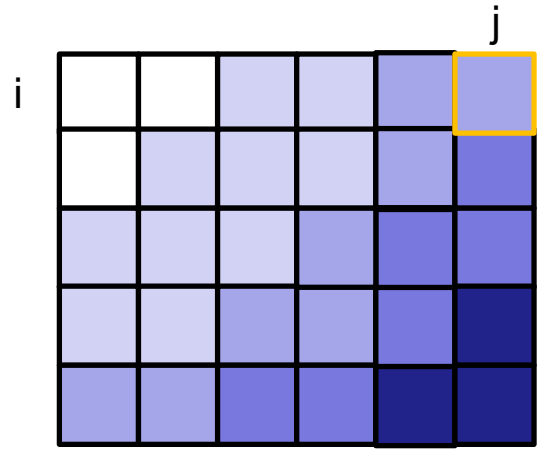
Partial Invariant: M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1]

How do we get the invariant to hold with i = 0?

- no easy way to initialize it so the invariant holds
- we need to search...

Sorted Matrix Search Code

Initialization:

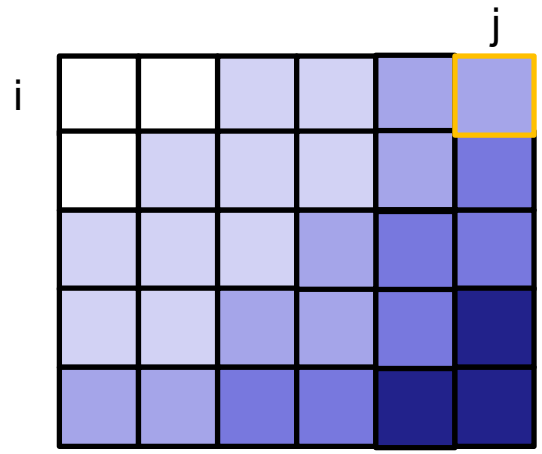


New goal: M[0,0], ..., M[0,j-1] < x ≤ M[0,j], ..., M[0,n-1]

- will need a loop to find j
- new loop invariant: x ≤ M[0,j], ..., M[0,n-1]
  - weakening of the new goal
  - decrease j until we get M[0,j-1] to also hold

Sorted Matrix Search Code

Initialization:




```
int i = 0;
int j = ?
```

{{ Inv: x ≤ M[i,j], ..., M[i,n-1] }}

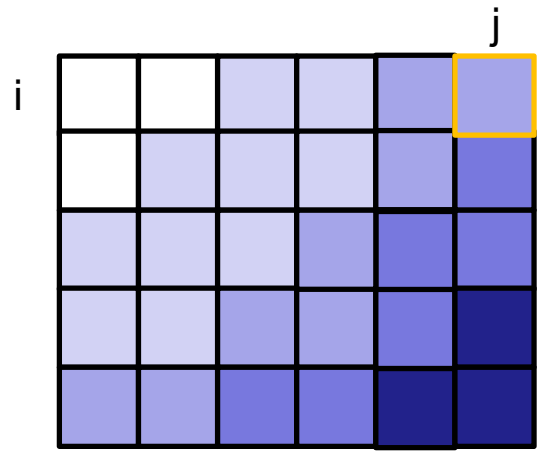
What is the easiest way to make this hold initially?

```
while ( ?? )
  ??
```

{{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}

Sorted Matrix Search Code

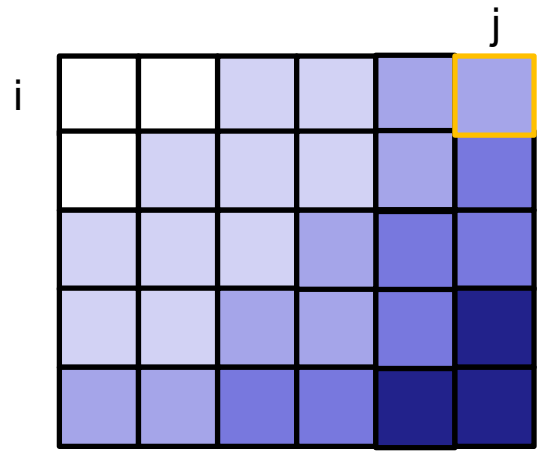
Initialization:




```
int i = 0;
int j = n;
```
{{ Inv: x ≤ M[i,j], ..., M[i,n-1] }}
```
while ( ?? )
  ??
```
{{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}

Sorted Matrix Search Code

Initialization:




```
int i = 0;
int j = n;
```

{{ Inv: x ≤ M[i,j], ..., M[i,n-1] }}

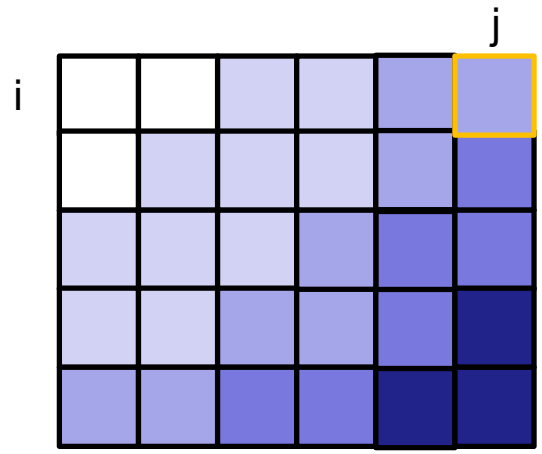
```
while ( ?? )
  ??
```

When does the postcondition hold? (Careful!)

{{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}

Sorted Matrix Search Code

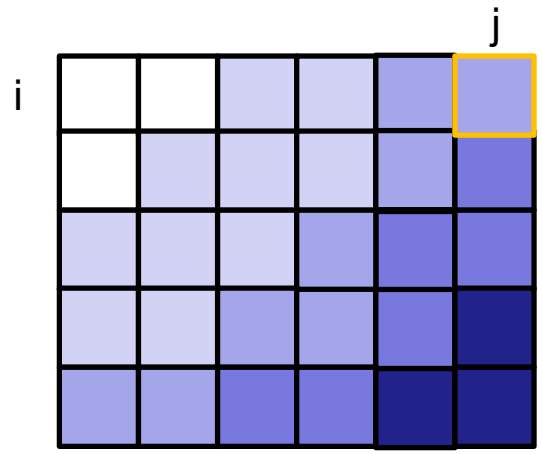
Initialization:




```
int i = 0;
int j = n;
```
{{ Inv: x ≤ M[i,j], ..., M[i,n-1] }}
```
while (j > 0 && x <= M[i,j-1])
  ??
```
{{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}

Sorted Matrix Search Code

Initialization:




```
int i = 0, j = n;
```

{{ Inv: x ≤ M[i,j], ..., M[i,n-1] }}

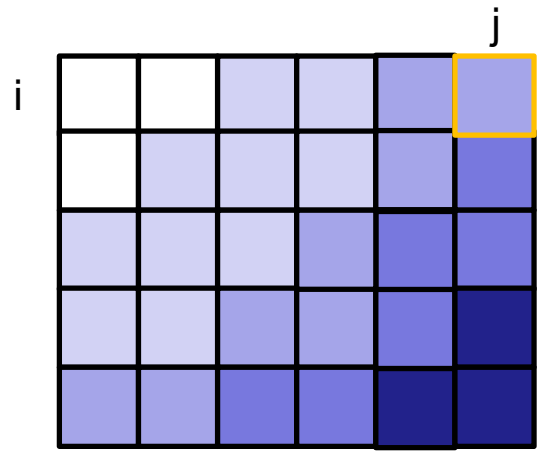
```
while (j > 0 && x <= M[i,j-1]) {
  ??
  j = j - 1;
}
```

What goes here?

{{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}

Sorted Matrix Search Code

Initialization:




```
int i = 0, j = n;
```

{{ Inv: x ≤ M[i,j], ..., M[i,n-1] }}

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

{{ x ≤ M[i,j], ..., M[i,n-1] and x ≤ M[i,j-1] }}

```
  ??
```

{{ x ≤ M[i,j-1], ..., M[i,n-1] }}

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

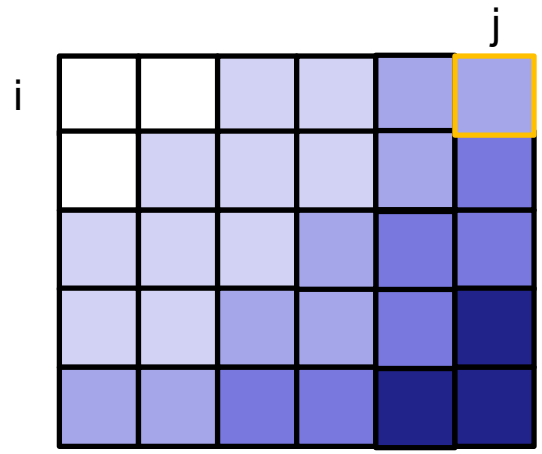
{{ x ≤ M[i,j], ..., M[i,n-1] }}

```
}
```

{{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}

Sorted Matrix Search Code

Initialization:




```
int i = 0, j = n;
```

{{ Inv: x ≤ M[i,j], ..., M[i,n-1] }}

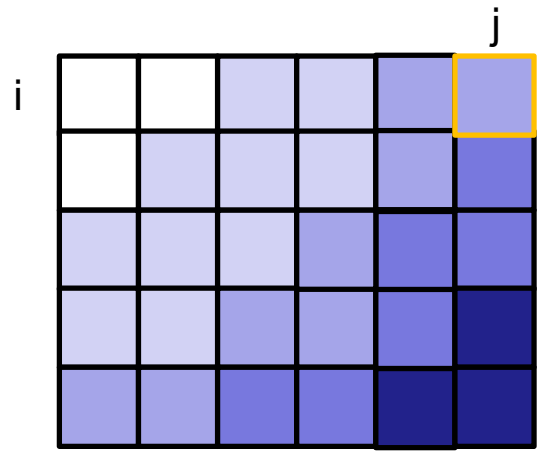
```
while (j > 0 && x <= M[i,j-1]) {

  j = j - 1;
}
```

What goes here?
Nothing!

{{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}

Sorted Matrix Search Code

Initialization:



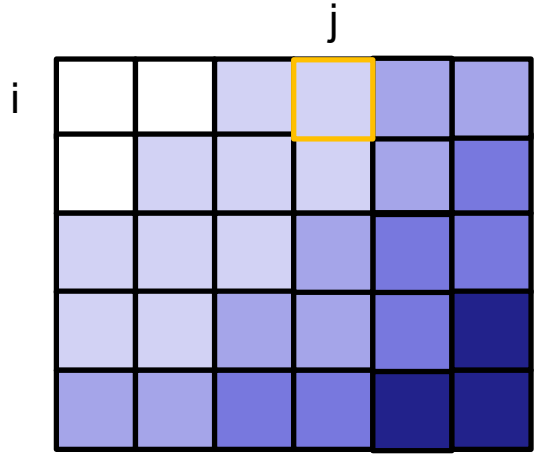

```
int i = 0;
int j = n;
```

{{ Inv: x ≤ M[i,j], ..., M[i,n-1] }}

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

{{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}

Sorted Matrix Search Code

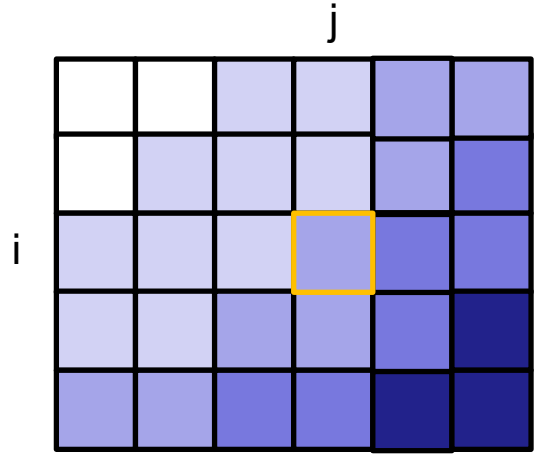


{{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}

That finds the right column in row 0

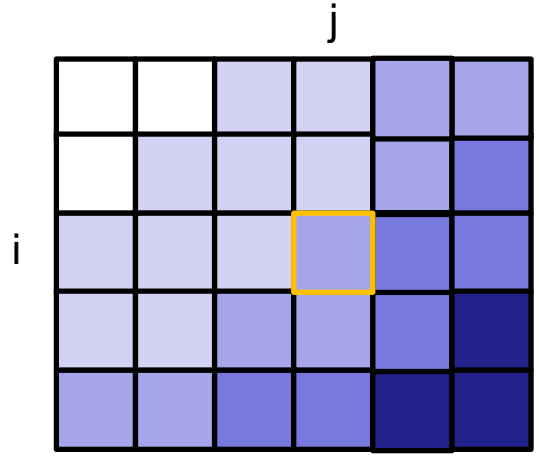
- can now check M[0,j] = x (if j < n)
- if not, we can move onto the next row
  - set `i = i + 1`
  - same idea on each row thereafter...

Sorted Matrix Search Code



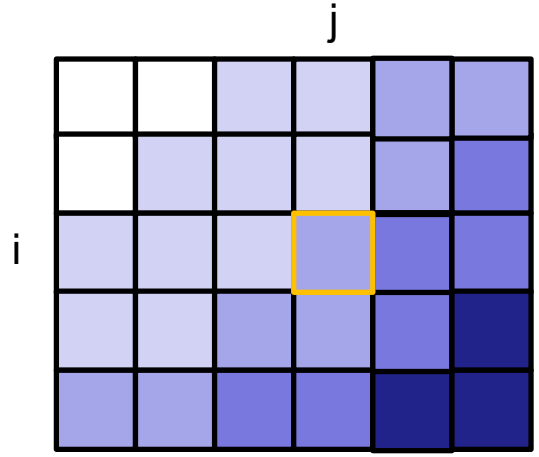
- Make progress by setting `i = i + 1`
- When i increases, the invariant may be broken
  - we have x ≤ M[i,j] ≤ M[i+1,j] since columns are sorted
  - and M[i+1,j] ≤ M[i +1,j+1], .., M[i +1,n-1] since rows are sorted
  - so we get x ≤ M[i +1,j], .., M[i +1,n-1]

Sorted Matrix Search Code



- Make progress by setting `i = i + 1`
- When i increases, the invariant may be broken
  - we have x <= M[i +1,j], .., M[i +1,n-1]
  - may need to restore invariant for M[i,0], ..., M[i,j-1] < x
  - decrease j until it holds again...
    - when have we seen this before?
    - initialization

Sorted Matrix Search Code



- Make progress by setting `i = i + 1`
- When i increases, the invariant may be broken
  - we have x <= M[i +1,j], .., M[i +1,n-1]
  - may need to restore invariant for M[i,0], ..., M[i,j-1] < x
  - could copy and paste the same loop
    - or you can do it with one copy

Don't try this at home!

Sorted Matrix Search Code

instead of

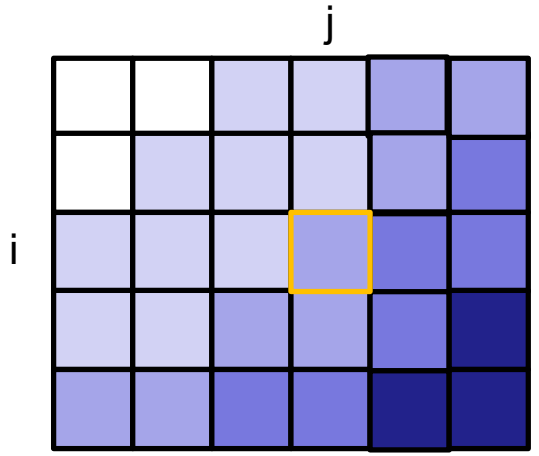
```
int i = 0, j = n;
[move j left]
{{ Inv: M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}
while (i != n) {
  i = i + 1;
  [move j left]
}
```

we can write

```
int i = 0, j = n;
while (i != n) {
  [move j left]
  {{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}
  i = i + 1;
}
```

Sorted Matrix Search Code

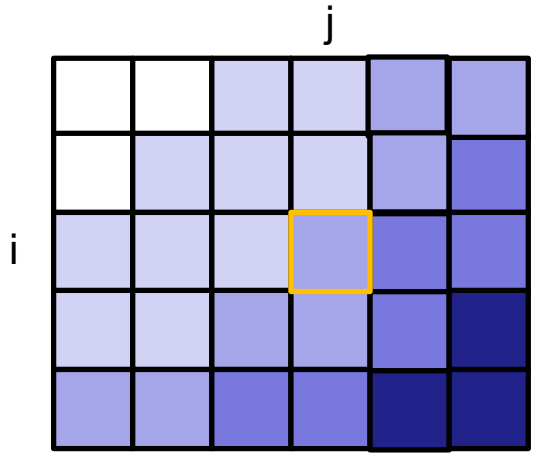
```
int i = 0;
int j = n;

while (i != n) {
  {{ Inv: x ≤ M[i,j], ..., M[i,n-1] }}
  while (j > 0 && x <= M[i][j-1])
    j = j - 1;

  {{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}
  if (j < n && x == M[i][j])
    return true;
  i = i + 1;
}
return false;
```



How do we know from Inv that this is correct?

Sorted Matrix Search Code

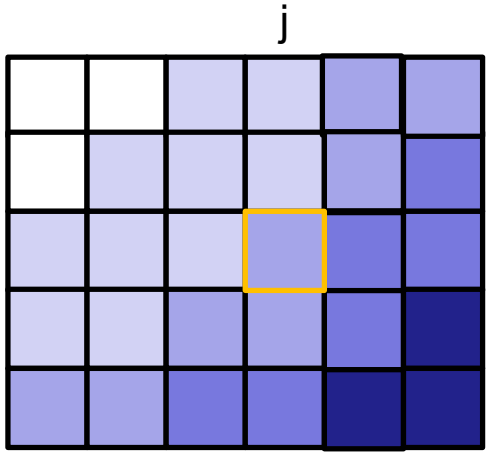
```
int i = 0;
int j = n;

while (i != n) {
  {{ Inv: x ≤ M[i,j], ..., M[i,n-1] }}
  while (j > 0 && x <= M[i][j-1])
    j = j - 1;

  {{ M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}
  if (j < n && x == M[i][j])
    return true;
  i = i + 1;
}
return false;
```



How do we know from Inv that this is correct?

We don't! Something is missing…

Sorted Matrix Search Code




```
int i = 0;
int j = n;
{{ Inv: x not in M[k,l] for k < i and x ≤ M[i,j], ..., M[i,n-1] }}
while (i != n) {
  {{ Inv: x not in M[k,l] for k < i and x ≤ M[i,j], ..., M[i,n-1] }}
  while (j > 0 && x <= M[i][j-1])
    j = j - 1;

  {{ x not in M[k,l] for k < i and M[i,0], ..., M[i,j-1] < x ≤ M[i,j], ..., M[i,n-1] }}
  if (j < n && x == M[i][j])
    return true;
  i = i + 1;
}
return false;
```

x not in M[k,l] for k < i+1