CSE 331
Software Design & Implementation

Kevin Zatloukal

Summer 2016

Lecture 4 – Writing Loops

(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins, Zach Tatlock)

Announcements

- HW0 is graded
  - talk to staff if you are unsure of why you missed points
- HW1 solutions handed out after class
- HW2 is posted
  - due by 11pm on Wed
  - last substantial assignment before the long break
  - (all assignments are regular speed after that)
- Don’t miss Thu section
  - HW3 should be very quick after section

Loop Invariants

- There is no general way to deduce the invariant from the code

- Why would we ever need to do this?
- It suggests coding like this:

$$\text{Idea} \Rightarrow \text{Code} \Rightarrow \text{Invariant} \Rightarrow \text{Proof}$$

Loop Invariants

- There is no general way to deduce the invariant from the code

- Why would we ever need to do this?
- Don’t do this:

~~Idea → Code → Invariant → Proof~~

Loop Invariants

- There is no general way to deduce the invariant from the code.

- Don’t do this:

~~Idea $\Rightarrow$ Code $\Rightarrow$ Invariant $\Rightarrow$ Proof~~

- Instead, do this:

Idea $\Rightarrow$ Invariant $\Rightarrow$ Code $\Rightarrow$ Proof

Loop Invariants Before Code

- Loop invariant comes out of the algorithm idea
  - describes partial progress toward the goal
    - how you will get from start to end
  - contains the essence of the algorithm idea

- A good invariant will make the code easier to write
  - a great invariant makes the code “write itself”
  - (we will see the same thing with invariants for ADTs etc.)

Loop Invariants in this Course

- We advocate writing invariants before the code
  - if the code is there, the invariant should be there too

- You will not be asked to find the invariant for the code
- Types of problems in HW2:
  - given invariant and code, prove it correct
  - given invariant, write code
  - write invariant and (then) code [for simple algorithms]

- When writing code, document your loop invariants
  - don’t make readers re-discover them
  - improves changeability and understandability

Loop Invariant Design Patterns

- Often loop invariant is a weakening of the postcondition
  - partial progress with completion a special case

- Example: finding the maximum value in an array
  - postcondition: m = max(A[0], …, A[n-1])
  - loop invariant: m = max(A[0], …, A[i-1]) for some i
    - postcondition is special case i = n

- Only *slightly* weakened postcondition: I and not `cond` implies Q
- Stronger is usually better
  - if it is strong enough, there is only one way to write body
  - (but if it's too strong, there may be no way to write the body!)

Example: quotient and remainder

Problem: Set q to be the quotient of x/y and r to be the remainder

Precondition: x >= 0 and y > 0

Postcondition: q*y + r = x and 0 <= r < y

- y doesn't go into x any more times

Example: quotient and remainder

Problem: Set q to be the quotient of x/y and r to be the remainder

Precondition: x >= 0 and y > 0

Postcondition: q*y + r = x and 0 <= r < y

- y doesn't go into x any more times

Loop invariant: q*y + r = x and r >= 0

- postcondition is special case when we also have r < y
- this suggests a loop condition…

Example: quotient and remainder

We want “r < y” when the conditions fails

- so the condition is r >= y
- can see immediately that the postcondition holds on loop exit

```
{{ Inv: q*y + r = x and 0 <= r }}
while (r >= y) {


}
{{ q*y + r = x and 0 <= r < y }}
```

Example: quotient and remainder

Need to make the invariant hold initially…

- search for the simplest way that works
- can only have q*y – x = r >= 0 for all y if we take q = 0

```
int q = 0;
int r = x;
{{ Inv: q*y + r = x and 0 <= r }}
while (r >= y) {


}
{{ q*y + r = x and 0 <= r < y }}
```

Example: quotient and remainder

We have r large initially.

Need to shrink r on each iteration in order to terminate…

- if r >= y, then y goes into x at least one more time

```
int q = 0;
int r = x;
{{ Inv: q*y + r = x and 0 <= r }}
while (r >= y) {
  q = q + 1;
  r = r - y;
}
{{ q*y + r = x and 0 <= r < y }}
```

Example: quotient and remainder

We have r large initially.

Need to shrink r on each iteration in order to terminate…

- if r >= y, then y goes into x at least one more time

```
int q = 0;
int r = x;
{{ Inv: q*y + r = x and 0 <= r }}
while (r >= y) {
  q = q + 1;
  r = r - y;
}
{{ q*y + r = x and 0 <= r < y }}
```

(+y and -y cancel)

{{ (q+1)*y + r-y = x and y <= r }}
{{ q*y + r-y = x and 0 <= r-y }}
{{ q*y + r = x and 0 <= r }}

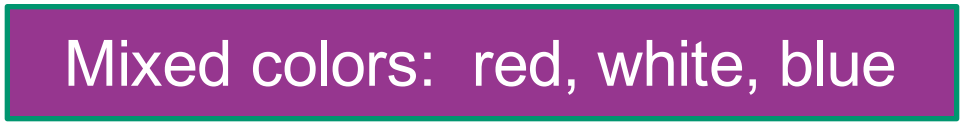
Example: Dutch National Flag

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

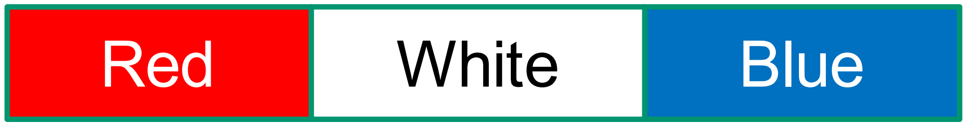
Pre- and post-conditions

Precondition: Any mix of red, white, and blue



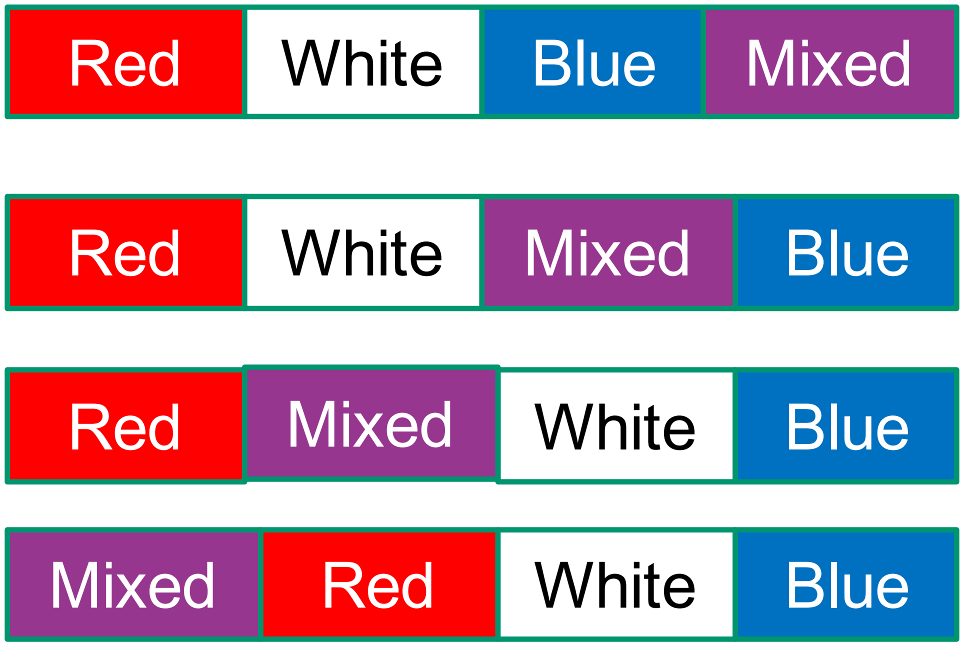
Postcondition:

- Red, then white, then blue
- Number of each color same as in original array



Some potential invariants

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

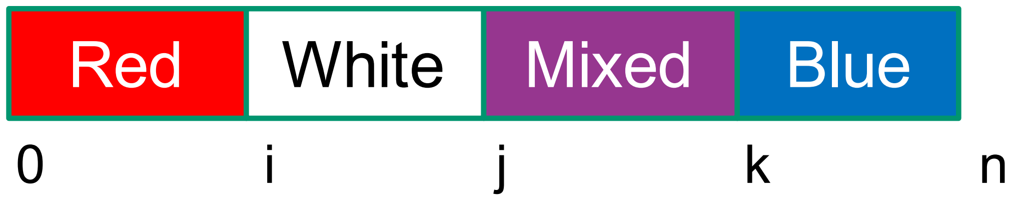


(Middle two have one less line of code… only matters on slides)

Precise Invariant

Need indices to refer to the split points between colors

- call these i, j, k



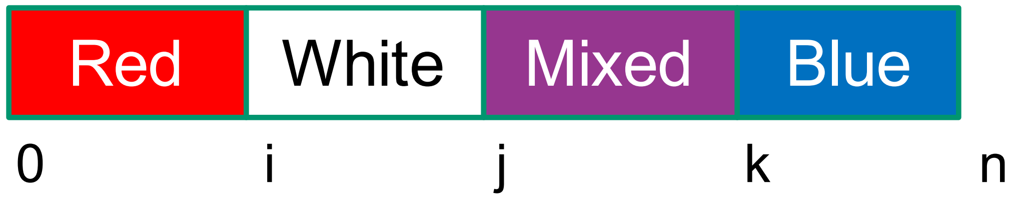
Loop Invariant:

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

No constraints on A[j], A[j+1], …, A[k-1]

Dutch National Flag Code

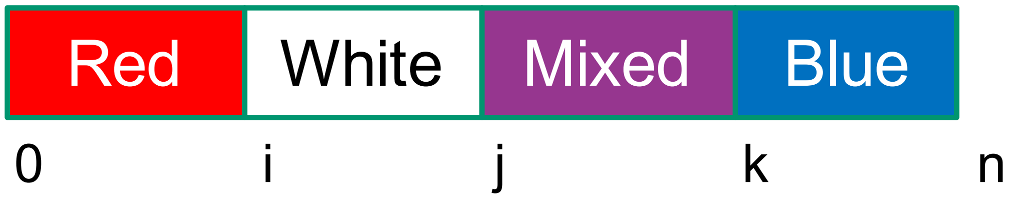
Invariant:



Initialization?

Dutch National Flag Code

Invariant:

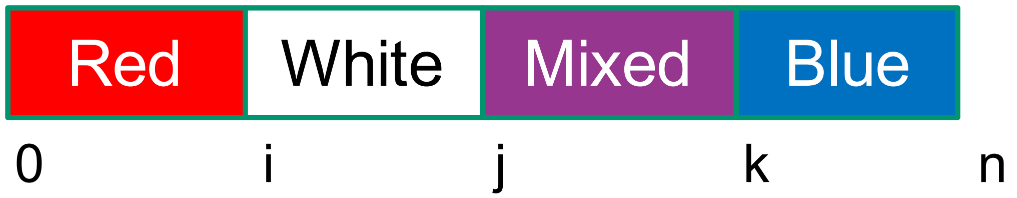


Initialization:

- i = j = 0 and k = n

Dutch National Flag Code

Invariant:



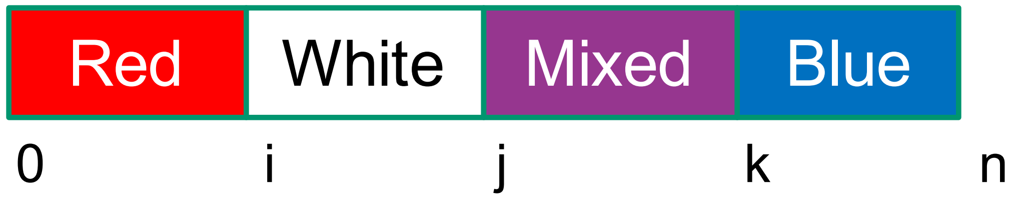
Initialization:

- i = j = 0 and k = n

Termination condition?

Dutch National Flag Code

Invariant:



Initialization:

- i = j = 0 and k = n

Termination condition:

- j = k

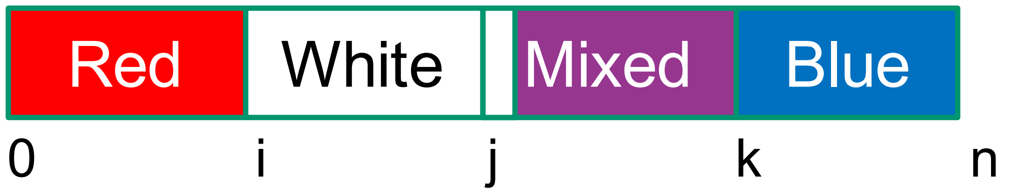
Dutch National Flag Code

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

    // need to get j closer to k
    // let's increase j…









}
```

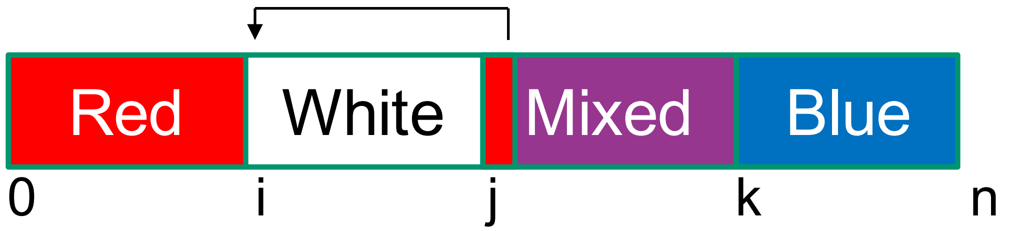
Dutch National Flag Code

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

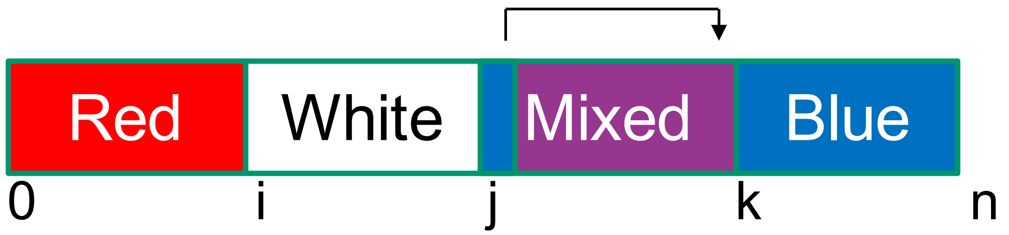
white



red



blue



Dutch National Flag Code

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

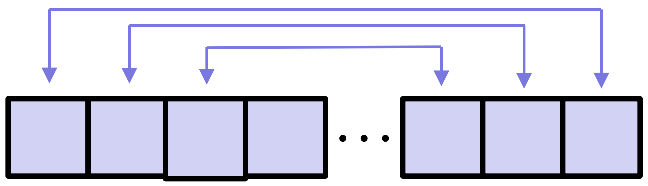
{{ Inv: 0 <= i <= j <= k <= n and A[0], …, A[i-1] is red and … }}

```
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;
  }
}
```

Example: Reverse an Array

Problem: Given array A of length n, put elements in reverse order.

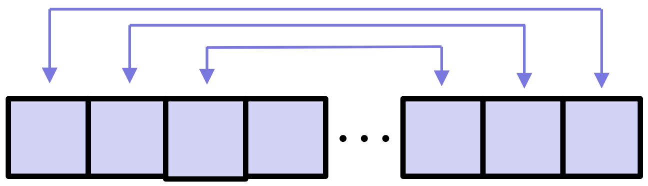
Idea: Swap A[0] and A[n-1] then A[1] and A[n-2] etc.



Example: Reverse an Array

Problem: Given array A of length n, put elements in reverse order.

Idea: Swap A[0] and A[n-1] then A[1] and A[n-2] etc.



Invariant: A[0], A[1], …, A[i-1] swapped with A[j], A[j+1], …, A[n-1]

- (here, "swapped" means swapped with reverse of)
- indices i and j keep track of what has been swapped

Reverse an Array Code

Invariant: A[0], A[1], …, A[i-1] swapped with A[j], A[j+1], …, A[n-1]

Initialization?

Reverse an Array Code

Invariant: A[0], A[1], …, A[i-1] swapped with A[j], A[j+1], …, A[n-1]

Initialization:

- i = 0 and j = n

Reverse an Array Code

Invariant: A[0], A[1], …, A[i-1] swapped with A[j], A[j+1], …, A[n-1]

Initialization:

- i = 0 and j = n

Termination condition:

- i != j and i != j - 1
- if i = j - 1, then A[i] = A[j-1] stays in place when A is reversed

Reverse an Array Code

Invariant: A[0], A[1], …, A[i-1] swapped with A[j], A[j+1], …, A[n-1]

Initialization:

- i = 0 and j = n

Termination condition:

- i != j and i != j - 1
- if i = j - 1, then A[i] = A[j-1] stays in place when A is reversed

Reverse an Array Code

```
int i = 0;
int j = n;
{{ Inv: A[0], A[1], …, A[i-1] swapped with A[j], A[j+1], …, A[n-1] }}
while (i != j and i != j – 1) {

  // perform another swap…

}
{{ A[0], ..., A[n/2] swapped with A[n-n/2], ..., A[n-1] }}
```

Reverse an Array Code

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

{{ Inv: A[0], A[1], …, A[i-1] swapped with A[j], A[j+1], …, A[n-1] }}

```
while (i != j and i != j - 1) {
  swap A[i], A[j-1];
  i = i + 1;
  j = j - 1;
}
```

{{ A[0], ..., A[n/2] swapped with A[n-n/2], ..., A[n-1] }}

Reverse an Array Code

```
int i = 0;
int j = n;
```
{{ Inv: A[0], A[1], …, A[i-1] swapped with A[j], A[j+1], …, A[n-1] }}
```
while (i != j and i != j – 1) {
   swap A[i], A[j-1];
   i = i + 1;
   j = j – 1;
}
```
$\downarrow$ {{ A[0], …, A[i] swapped with A[j-1], ..., A[n-1] }}

{{ A[0], ..., A[n/2] swapped with A[n-n/2], ..., A[n-1] }}

Example: Binary Search

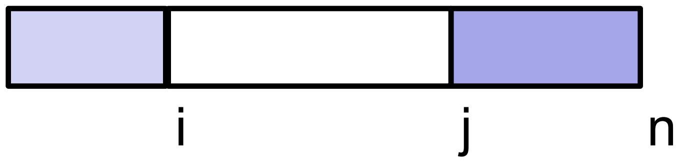
Problem: Given a sorted array A and a number x, find index of x (or where it would be inserted) in A.

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.

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.

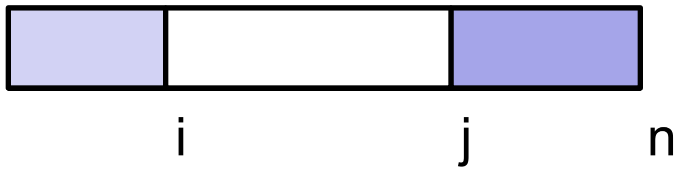
Idea: Look at A[n/2] to figure out if x is in A[0], A[1], …, A[n/2] or in A[n/2+1], …, A[n-1]. Narrow the search for x on each iteration.



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

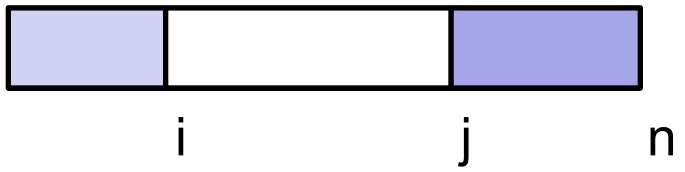
- so x must lie in A[i], …, A[j-1]
- A[i], …, A[j-1] is the part where we don't know relation to x

Binary Search Code



Initialization?

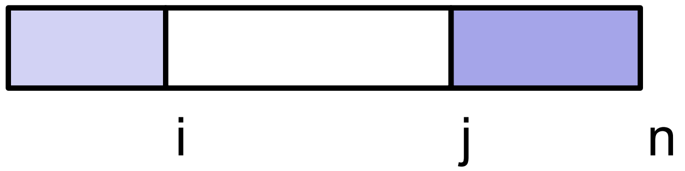
Binary Search Code



Initialization:

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

Binary Search Code



Initialization:

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

Termination condition:

- i = j
- white part 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] }}
while (i != j) {


  // 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] }}
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] }}
```

Binary Search Code

```
int i = 0;
int j = n;
{{ Inv: A[0], …, A[i-1] <= x < A[j], …, A[n-1] }}
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] }}
```

Binary Search Code

```
int i = 0;
int j = n;
{{ Inv: A[0], …, A[i-1] <= x < A[j], …, A[n-1] }}
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 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] }}
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])

Aside on Termination

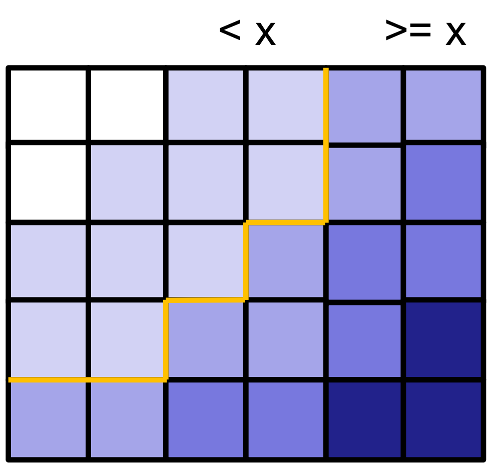
- Most often correctness is harder work than termination
  - the latter follows from running time bound

- We've seen examples where termination is more interesting
  - (cases with variable progress toward termination condition)
  - quotient and remainder (Inv: q*y + r == x and r >= 0)
  - binary search
- It's easy to make a mistake and have no progress
  - then the code may loop forever

- HW2 has a problem where correctness is trivial
- The only difficult part is checking that it terminates:
  - need to check that there is progress on every iteration

Example: Sorted Matrix Search

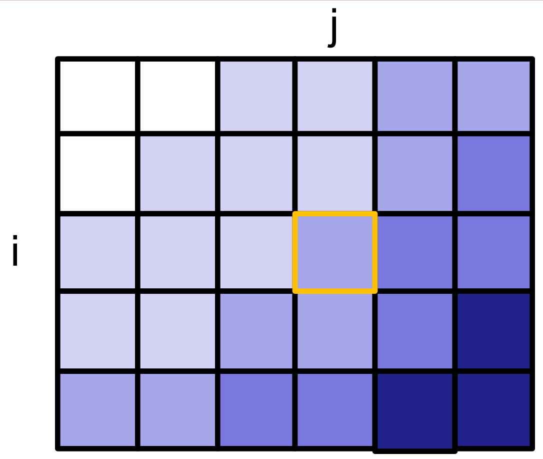
Problem: Given a sorted a matrix M (of size m x n), where every row and every column is sorted, find out whether a given number x is in the matrix.



(darker color means larger)

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

Sorted Matrix Search Code

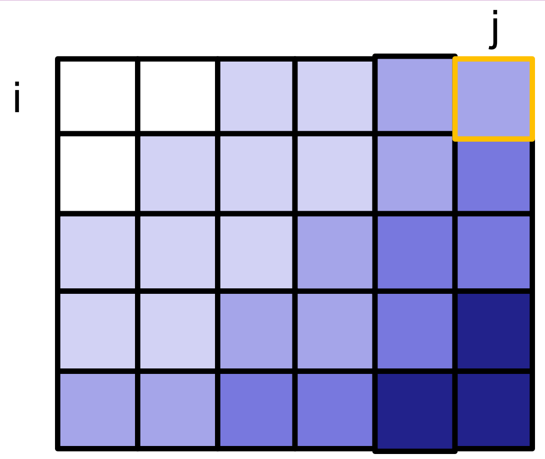


Loop Invariant: M[i,0], ..., M[i,j-1] < x <= M[i,j], ..., M[i,n-1]

- will increase i from 0 to m

Sorted Matrix Search Code

Initialization:

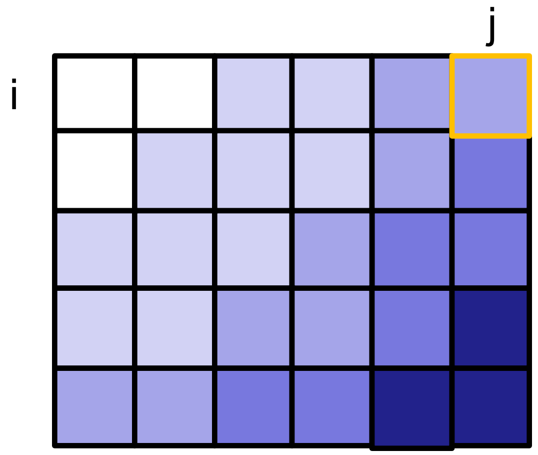


No obvious way to initialize so the invariant holds

To start in row 0 (i = 0), we need to search...

Sorted Matrix Search Code

Initialization:



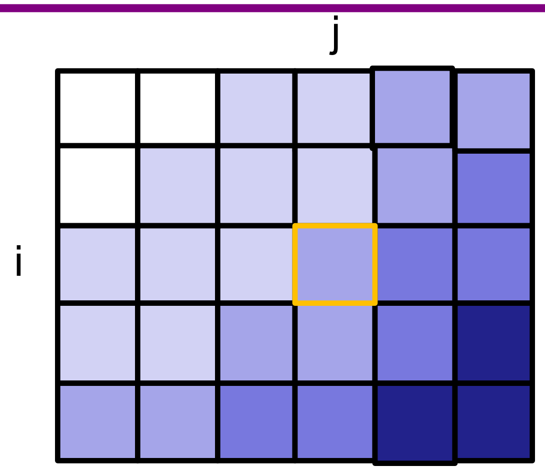
```
int i = 0;
int j = n;
```

{{ Inv: x <= M[i,j], ..., M[i,n-1] }}

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

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

Sorted Matrix Search Code

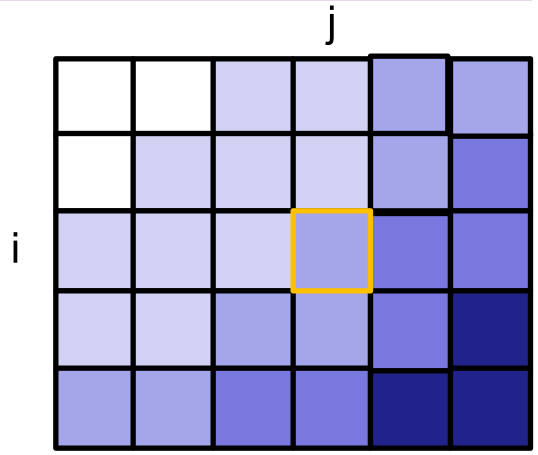


Loop body:

- when i increases, the invariant may be broken
  - we have M[i,j] <= M[i+1,j]
  - may need to decrease j to restore invariant
  - this is the same issue came up in initialization

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 and 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-1 and x == M[i,j])
        return true;
    i = i + 1;
}
return false;
```



Example: Special Composites

Problem: Find the N-th largest number of the form $2^a3^b5^c$, for some exponents a, b, c >= 0.

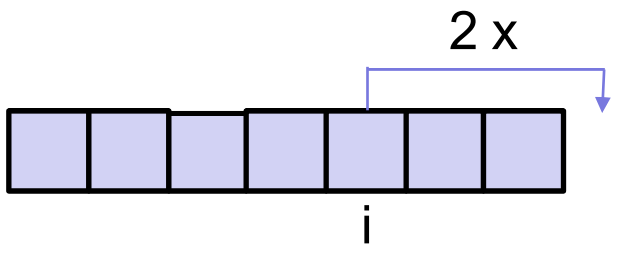
Idea: Generate these numbers in order ($1 = 2^03^05^0$, $2 = 2^13^05^0$, …) until we get to the N-th.

Subproblem: given the first m numbers of this form, find m+1st.

Idea: Multiply every number by 2, 3, 5. Take the smallest result that is larger than the m-th number.

- $O(n^2)$ if implemented naively
- O(n log n) if implemented using binary search for 2, 3, and 5
- O(n) if optimized

Example: Special Composites



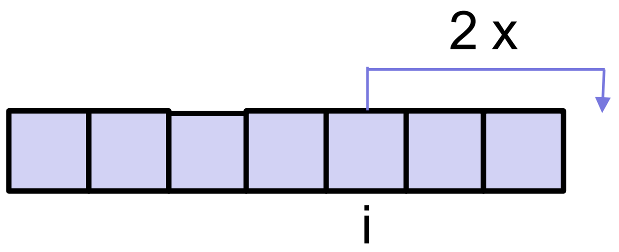
Optimization:

- Keep track of smallest index i such that 2 * A[i] > A[m-1]
- Do the same for 3 and 5. Call these indexes j and k
- Each iteration, we just need the smallest of these 3 numbers

Invariant:

- P2: 2*A[0], …, 2*A[i-1] <= A[m-1] < 2*A[i], …, 2*A[m-1]
- P3 (using j) and P5 (using k)

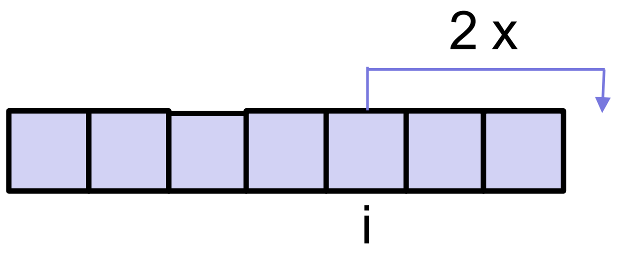
Special Composites Code



Initalization:

- Let A = {1} and m = 1
- Then i = j = k = 0 since 1 < 2, 3, 5

Special Composites Code



Termination:

- stop when m = N
- the N-th largest number is in A[m-1]

Special Composites Code

```
int[] A = new int[N]; A[0] = 1;
int i = 0, j = 0, k = 0, m = 1;
```
{{ Inv: A[m-1] < 2*A[i], 3*A[j], 5*A[k] (... abridged ...) }}
```
while (m < N) {
  A[m] = min(2*A[i], 3*A[j], 5*A[k]);
  if (2*A[i] == A[m])
    i = i + 1;
  if (3*A[j] == A[m])
    j = j + 1;
  if (5*A[k] == A[m])
    k = k + 1;
  m = m + 1;
}
return A[m-1];
```

Why not ”else if” ?

Special Composites Code

```
int[] A = new int[N]; A[0] = 1;
int i = 0, j = 0, k = 0, m = 1;
{{ Inv: A[m-1] < 2*A[i], 3*A[j], 5*A[k] (... abridged ...) }}
while (m < N) {
  A[m] = min(2*A[i], 3*A[j], 5*A[k]);
  if (2*A[i] == A[m])
    i = i + 1;
  if (3*A[j] == A[m])
    j = j + 1;
  if (5*A[k] == A[m])
    k = k + 1;
  m = m + 1;
}
return A[m-1];
```

Invariant says this is next

Preserves invariant:

- if 2*A[i] != A[m], then 2*A[i] > A[m]
- if 2*A[i] = A[m], then increasing i means we move to 2*A[i+1], which is > A[m]