CSE 331
Software Design & Implementation

Kevin Zatloukal

Fall 2017

Lecture 4.5 – More Loops

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

Reminders

- Reading Quiz 1 is due **tonight**

- HW2 on loops due next Thursday
  - please start early
  - some problems will take thought
    - please ask for help if you get stuck

Previously on CSE 331...

- Reasoning on straight-line code
  – turn-the-crank process

- Loops are more difficult
  – checking correctness requires a **loop invariant**
  – checking correctness requires:
    1. Invariant is true initially
    2. Invariant remains true each time around the loop
    3. Invariant implies post condition upon loop exit

- Loop invariants are especially crucial for tricky loops

Previously on CSE 331...

Loop invariant contains the essence of the algorithm idea...

In fact, can usually deduce the code from the invariant:

- What is the easiest way to satisfy the loop invariant?
  - gives you the initialization code
- When does loop invariant satisfy the postcondition?
  - gives you the termination condition
- How will you make progress each iteration?
  - gives you the last line(s) of the loop body
- How does the invariant change as you make progress?
  - gives you the rest of the loop body

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
??
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
??
```

Easiest way to make this hold?

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
??
```

Easiest way to make this hold?
Take i = 1 and m = max(b[0])

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

When does Inv imply postcondition?

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

When does Inv imply postcondition?
Happens when i = n

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??

}
```

How do we progress toward termination?

{{ m = max(b[0], ..., b[n-1]) }}

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??

}
```

How do we progress toward termination?
We start at i = 1 and end at i = n, so…

{{ m = max(b[0], ..., b[n-1]) }}

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??
   i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

How do we progress toward termination?
We start at i = 1 and end at i = n, so
Try this.

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??
   i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

When i becomes i+1, Inv becomes:
m = max(b[0], …, b[i])

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??
   i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

How do we get
from m = max(b[0], …, b[i-1])
to m = max(b[0], …, b[i])?

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??
   i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

How do we get
from m = max(b[0], …, b[i-1])
to m = max(b[0], …, b[i])?

Set m = max(m, b[i])

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {
   if (b[i] > m)
     m = b[i];
   i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

How do we get
from m = max(b[0], …, b[i-1])
to m = max(b[0], …, b[i])?

Set m = max(m, b[i])

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {
  if (b[i] > m)
    m = b[i];
  i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

Finding the loop invariant

Not every loop invariant is simple weakening of postcondition, but…

- that is the easiest case
- it happens a lot

In this class (e.g., exams):

- if I ask you to find the invariant, it will *very likely* be of this type
- I may ask you to inspect code with more complex invariants
- to learn about more ways of finding invariants: CSE 421

Examples: finding loop invariants

1. sum of array
   – postcondition: s = b[0] + b[1] + … + b[n-1]

Examples: finding loop invariants

1. sum of array
   - postcondition: s = b[0] + b[1] + … + b[n-1]
   - loop invariant: s = b[0] + b[1] + … + b[i-1]
     - gives postcondition when i = n
     - gives s = 0 when i = 0

Examples: finding loop invariants

1. sum of array
   - postcondition: s = b[0] + b[1] + … + b[n-1]
   - loop invariant: s = b[0] + b[1] + … + b[i-1]
     - gives postcondition when i = n
     - gives s = 0 when i = 0

2. max of array
   - postcondition: m = max(b[0], b[1], …, b[n-1])

Examples: finding loop invariants

1. sum of array
   – postcondition: s = b[0] + b[1] + … + b[n-1]
   – loop invariant: s = b[0] + b[1] + … + b[i-1]
     • gives postcondition when i = n
     • gives s = 0 when i = 0

2. max of array
   – postcondition: m = max(b[0], b[1], …, b[n-1])
   – loop invariant: m = max(b[0], b[1], ..., b[i-1])
     • gives postcondition when i = n
     • gives m = b[0] when i = 1

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

- i.e., 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

- i.e., y doesn't go into x any more times

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

- 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 r (= q*y – x) >= 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

Mixed colors: red, white, blue

Postcondition:

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

Red	White	Blue

Pre- and post-conditions

Precondition: Any mix of red, white, and blue

Mixed colors: red, white, blue

Postcondition:

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

Red	White	Blue

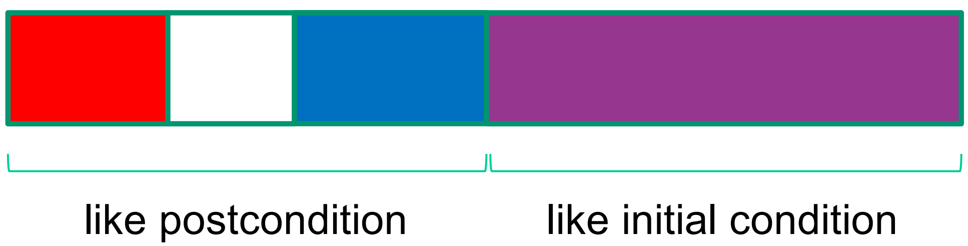
Loop invariant should (essentially) have

- postcondition as a special case
- initial condition as a special case

Loop invariant describes continuum of partial progress

Example: Dutch National Flag

The first idea that comes to mind:



Example: Dutch National Flag

The first idea that comes to mind works.

Initial:

Iter 5:

Iter 10:

Iter 15:

Post:

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[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 = 0, 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 try to increase j...

}
```

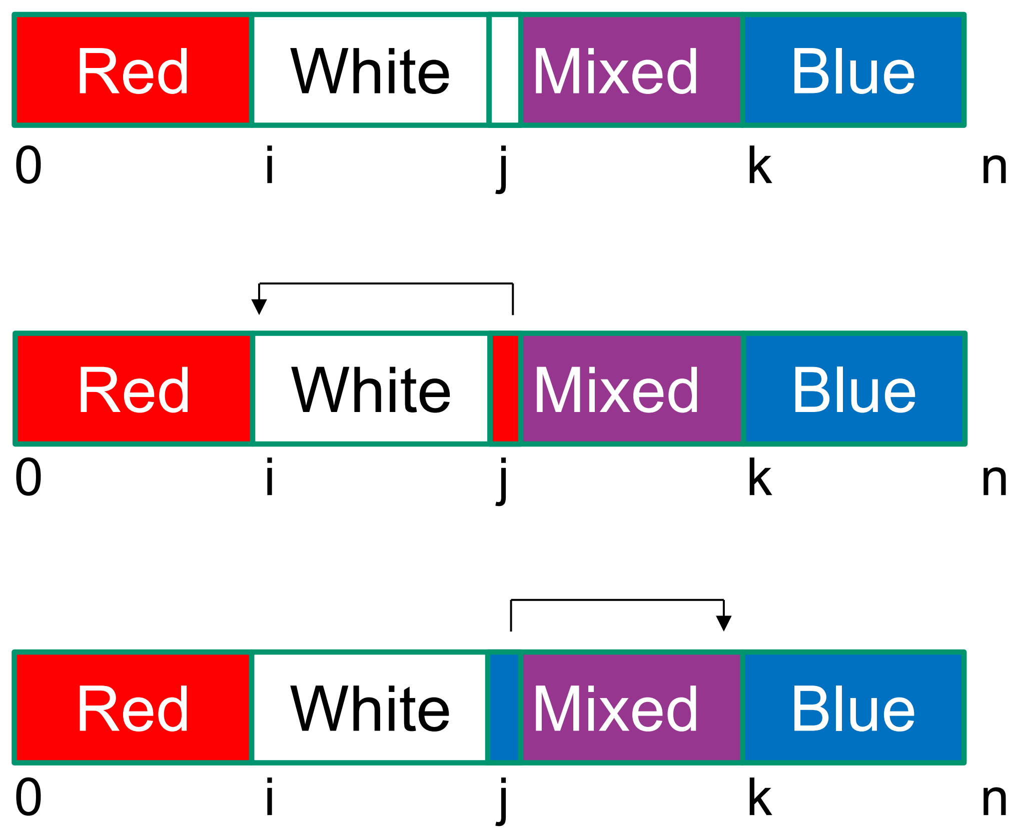
Dutch National Flag Code

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: Binary Search

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

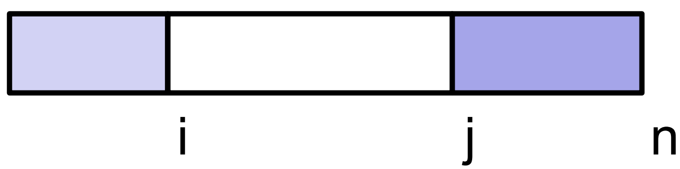
Idea: Look at A[n/2] to figure out if x is in A[0], A[1], ..., A[n/2] or in A[n/2+1], ..., A[n-1]. Narrow the search for x on each iteration.

(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.

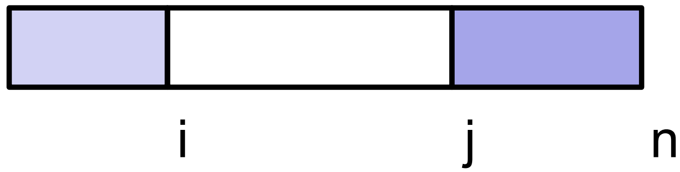
Idea: Look at A[n/2] to figure out if x is in A[0], A[1], ..., A[n/2] or in A[n/2+1], ..., A[n-1]. Narrow the search for x on each iteration.



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

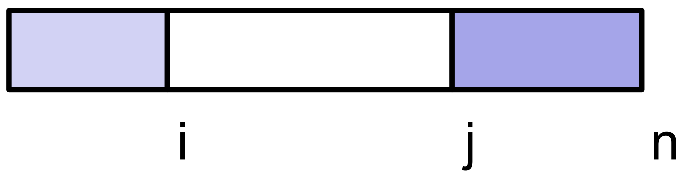
- A[i], ..., A[j-1] is the part where we don’t know relation to x

Binary Search Code



Initialization?

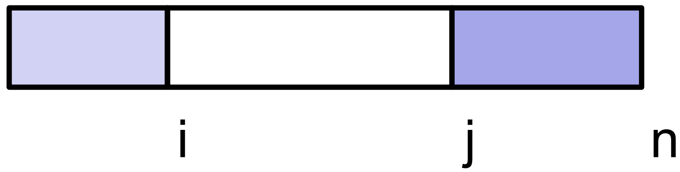
Binary Search Code



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) {
```

// 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) {
    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] 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] }}

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 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 {
    j = m;
  }
}
```

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

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

Aside on Termination

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

- But also 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

- See 16su HW2 for a problem where correctness is trivial and the *only* difficult part is checking that it terminates

Example: Special Composites

Problem: Find the N-th largest number of the form $2^a 3^b 5^c$, for some exponents a, b, c >= 0.

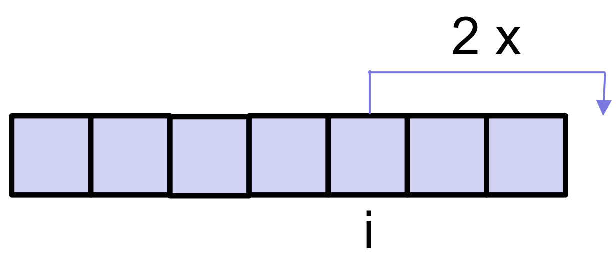
Idea: Generate these numbers in order ($1 = 2^0 3^0 5^0$, $2 = 2^1 3^0 5^0$, ...) until we get to the N-th.

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



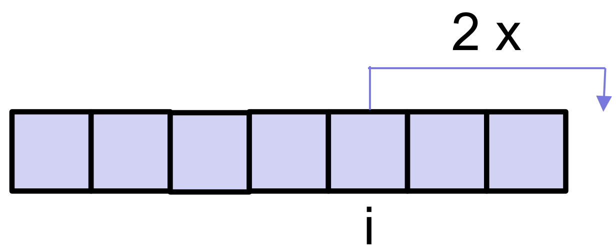
Optimization:

- 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:

- A is sorted
- 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)

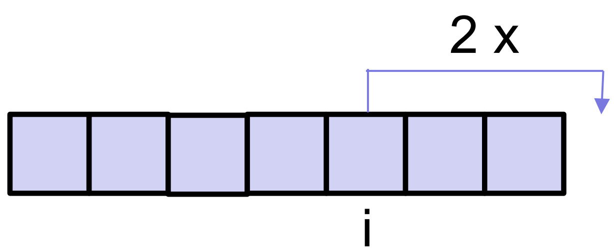
Special Composites Code



Initalization:

- Let A = [1] and m = 1
  - (note that array A also changes in this algorithm)
- Then i = j = k = 0 since 1 < 2, 3, 5

Special Composites Code



Termination:

- stop when m = N
- the N-th largest special composite 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];
```

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]

Special Composites Code

```
int[] A = new int[N]; A[0] = 1;
int i = 0, j = 0, k = 0, m = 1;
```

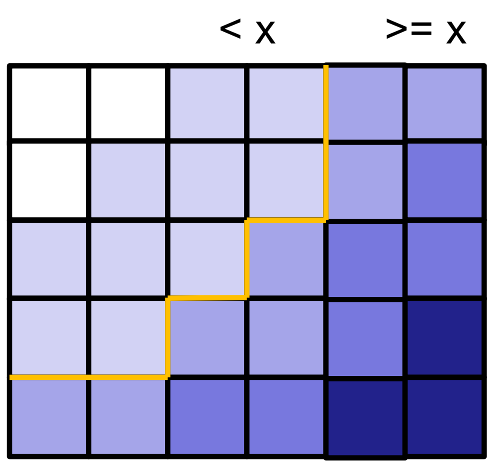
{{ Inv: A[m-1] < 2*A[i], 3*A[j], 5*A[k] (... abridged ...) }}

```
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” ?

Example: Sorted Matrix Search

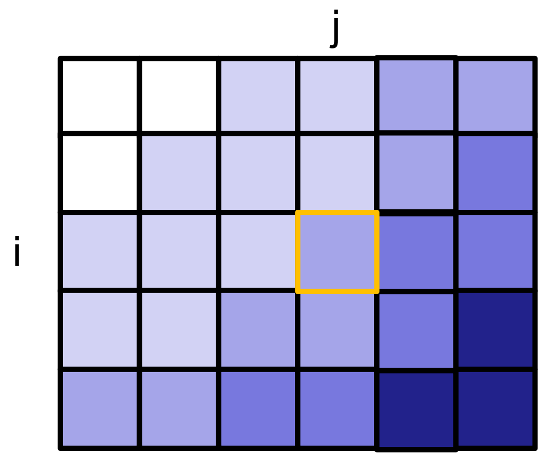
Problem: Given a sorted a matrix M (of size m x n), where every row and every column is sorted, find out whether a given number x is in the matrix.



(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

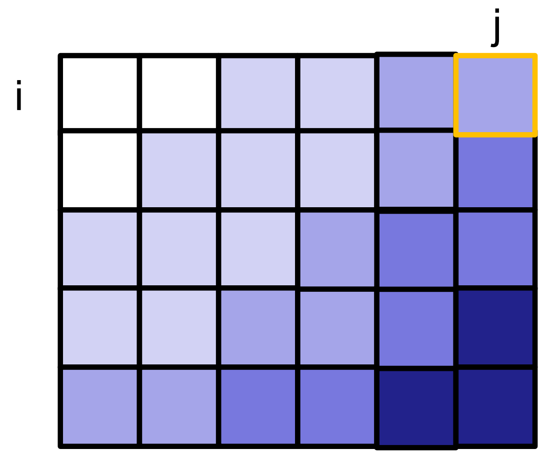


Loop Invariant: M[i,0], ..., M[i,j-1] < x <= M[i,j], ..., M[i,n-1]

- will increase i from 0 to m
- for each i, need to find the right j

Sorted Matrix Search Code

Initialization:

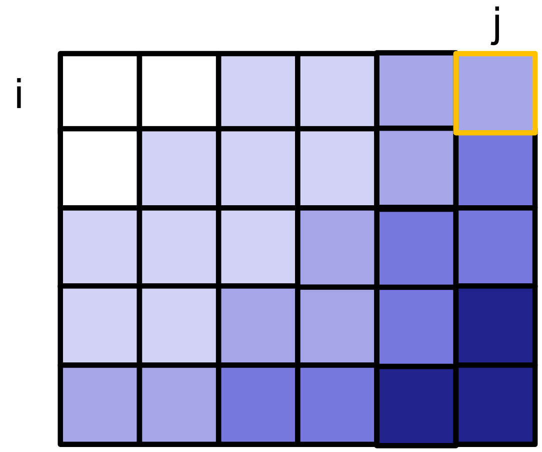


No obvious way to initialize so the invariant holds

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

Sorted Matrix Search Code

Initialization:



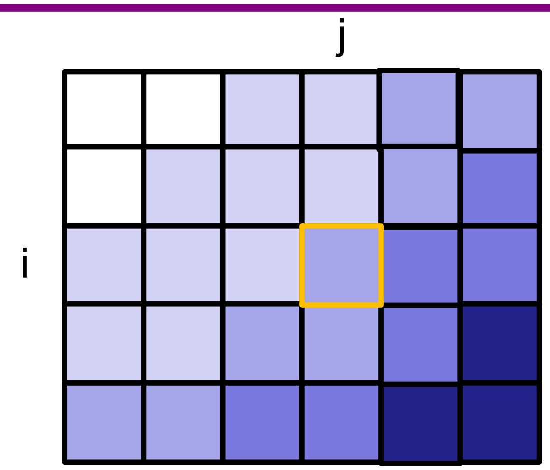

```
int i = 0;
int j = n;
```

{{ 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], so everything to right is still bigger
  - may need to decrease j to restore invariant for M[i,0], ..., M[i,j-1]
  - 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;
```