CSE 331

Software Design & Implementation

Topic: Reasoning Wrap-up

💬 **Discussion:** What is your favorite summer food?

Reminders

- HW1 grades coming out soon
- Remember to read the HW2 setup instructions very carefully!

Upcoming Deadlines

• Prep. Quiz: HW2	due today (6/26)
• HW2	due Thursday (6/29)

Last Time…

- Recap: Reasoning
  - assignment statements
  - conditionals
- Loop invariants
  - sum of array

Today's Agenda

- Writing Loops
  - max of array
  - Dutch National Flag
  - Binary Search
- Reasoning Summary

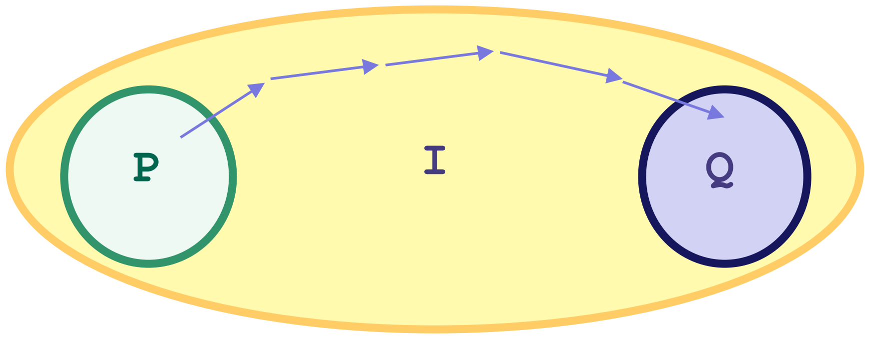
Previously on CSE 331...

{{ **`P`** }} `while (cond) S` {{ **`Q`** }}

Given an invariant, this triple is valid iff

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

- **`I`** holds initially
- **`I`** holds each time we execute `S`
- **`Q`** holds when **`I`** holds and `cond` is false



Previously on CSE 331…

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

- Essence of the algorithm idea is:
  - invariant
  - how you make progress on each step (e.g., `i = i + 1`)

- Code is *ideally* just details…

Termination

- Technically, this analysis does not check that the code **terminates**
  - it shows that the postcondition holds if the loop exits
  - but we never showed that the loop actually exits

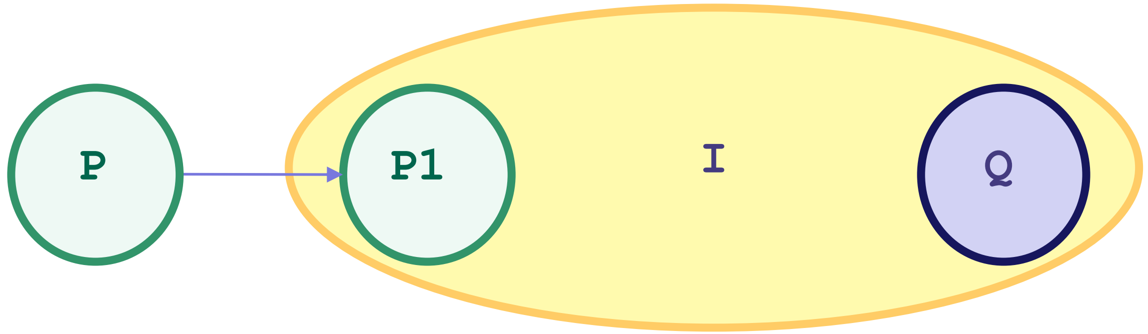
- However, that follows from an analysis of the running time
  - e.g., if the code runs in $O(n^2)$ time, then it terminates
  - an infinite loop would be O(infinity)
  - any finite bound on the running time proves it terminates

- It is normal to also analyze the running time of code we write, so we get termination already from that analysis.

Loop Invariant → Code

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

- When does loop invariant satisfy the postcondition?
  - gives you the termination condition
- What is the easiest way to satisfy the loop invariant?
  - gives you the initialization code
- How does the invariant change as you make progress?
  - gives you the rest of the loop body



Last time: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
i = 0;
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
{{ s = b[0] + ... + b[n-1] }}
```

Notice how the invariant is a weakening of postcondition

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 }}

 ??

{{ 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 }}

 ??

{{ 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 }}

 ??

{{ 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 }}

 ??

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

   ??

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

Easiest way to make this hold?

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 (i != n) {

   ??

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

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

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) {

    ??

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

How do we progress toward termination?
(comes from the algorithm idea)

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]) }}
```

$\uparrow$ {{ m = max(b[0], …, b[i]) }}
{{ m = max(b[0], …, b[i-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]) }}

$\boxed{\text{Set m = max(m, b[i])}}$

{{ m = max(b[0], …, b[i-1]) }}

{{ m = max(b[0], …, b[i]) }}

{{ m = max(b[0], …, b[i-1]) }}

How do we fill this in?

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)          OR m = Math.max(m, b[i]);
    m = b[i];
  i = i + 1;
}
{{ m = max(b[0], ..., b[n-1]) }}
```

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]) }}
```

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]) }}
```

the algorithm idea

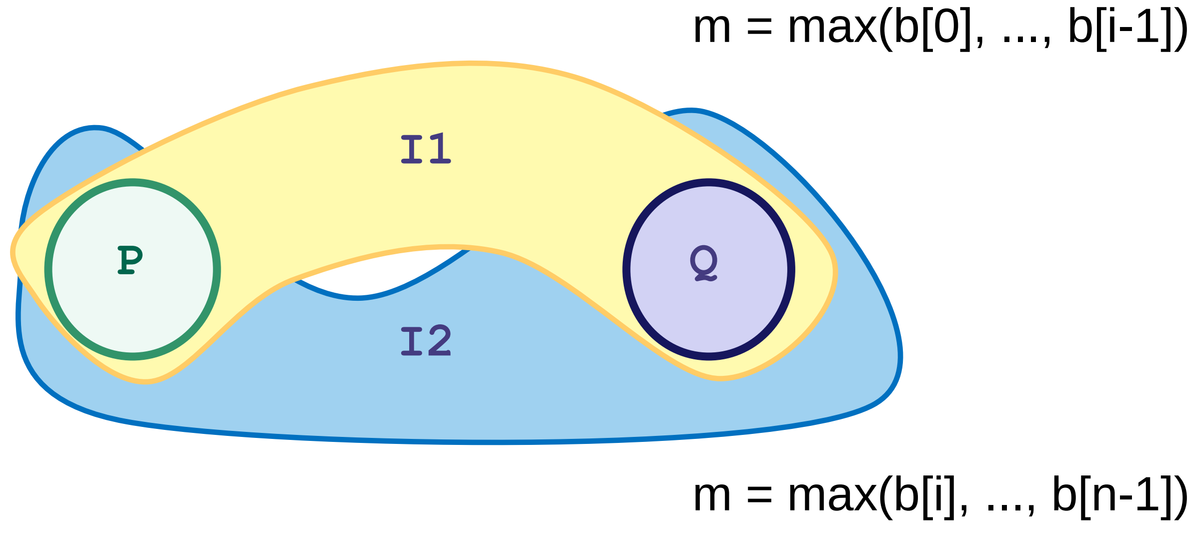
Invariants are Essential

Invariant + progress step is the essence of the algorithm idea

- rest is hopefully just details that follow from the invariant

Work toward thinking at the level of invariants not code

- gain confidence that you can do the rest without difficulty



Loop Invariant Design Pattern

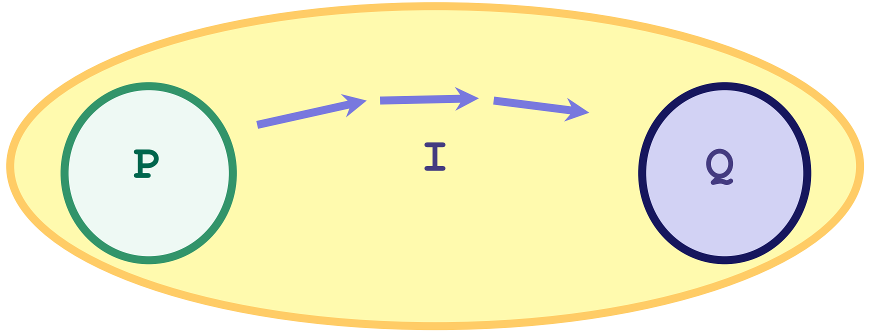
Loop invariant is often a weakening of the postcondition

- partial progress with completion a special case
- small enough weakening that Inv + one condition gives Q

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
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

Loop Invariant Design Patterns

Algorithm Idea formalized in: Invariant + *progress step*



- how do you make progress toward termination?
  - if condition is i != n (and i <= n)
    try i = i + 1
  - if condition is i != j (and i <= j)
    try i = i + 1 or j = j – 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., homework):

- if I ask you to find the invariant, it will *very likely* be of this type
  - I will ask you to write more complex code when the invariant given
  - I will you to check correctness of even more complex code
  - HW2-4 will practice these
- to learn about more ways of finding invariants: CSE 421

A Harder Example

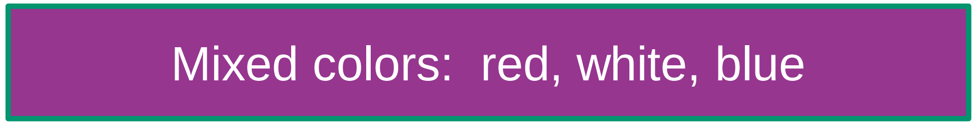
Example: Dutch National Flag

Problem: 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

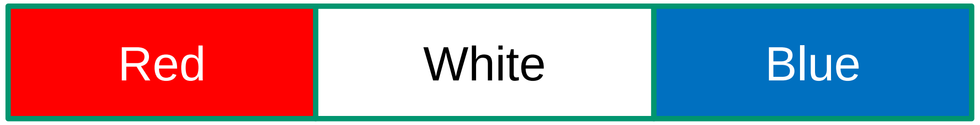
Pre- and post-conditions

Precondition: Any mix of red, white, and blue



Postcondition:

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

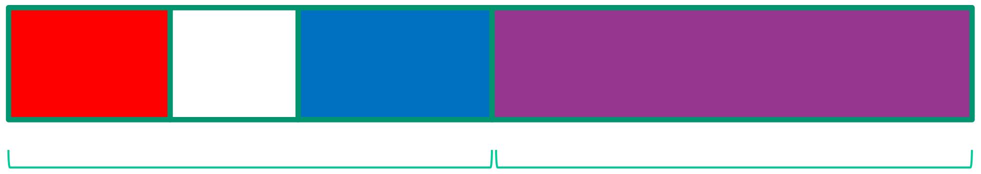


Want an invariant with

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

Example: Dutch National Flag

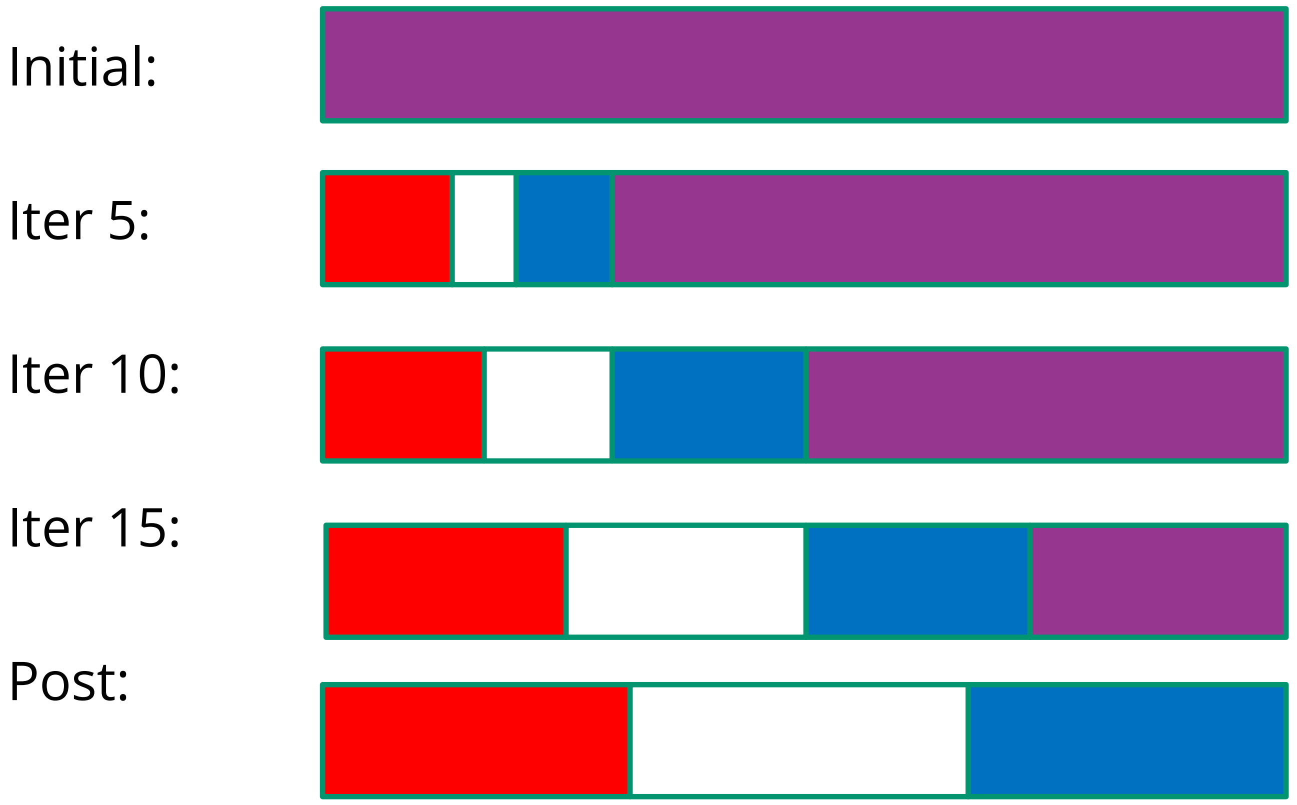
The first idea that comes to mind:



like postcondition like initial condition

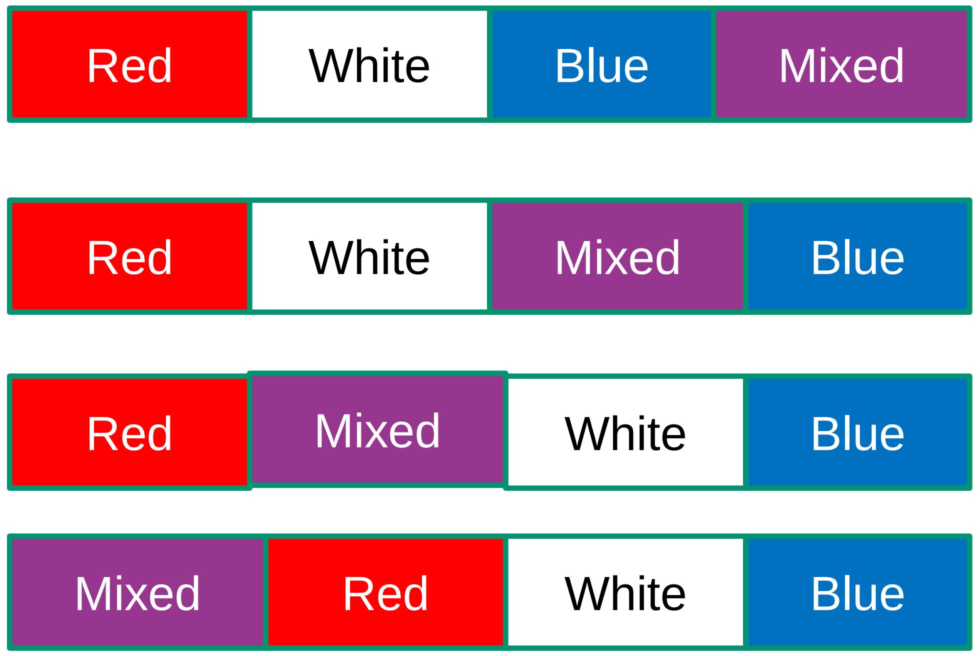
Example: Dutch National Flag

The first idea that comes to mind works.



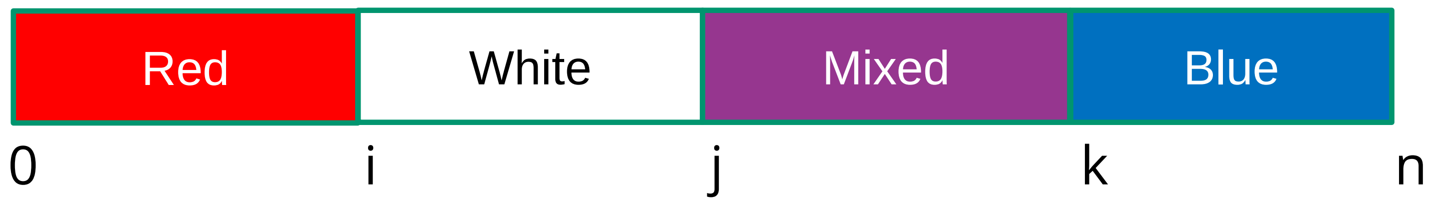
Other potential invariants

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



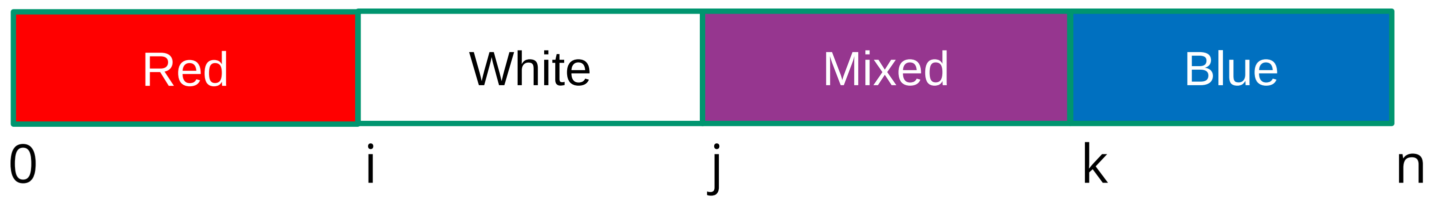
Loop Invariant:

- 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

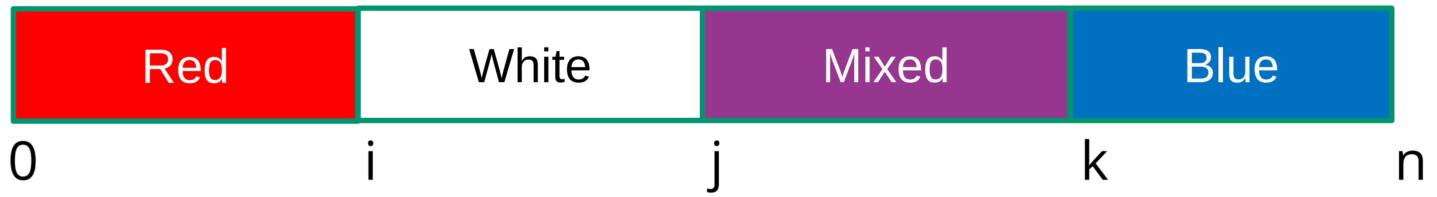
Invariant:



Initialization?

Dutch National Flag Code

Invariant:

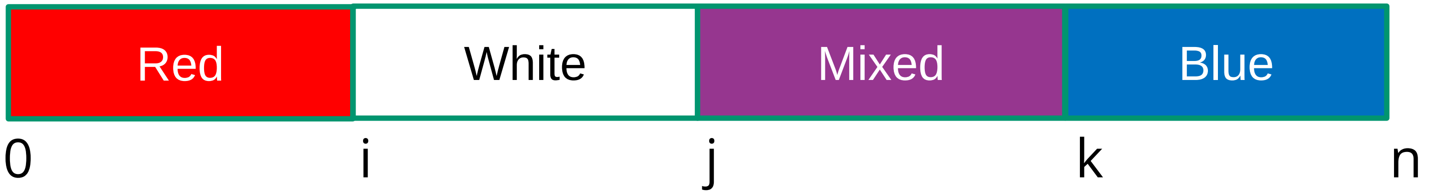


Initialization?

i = j = 0 and k = n

Dutch National Flag Code

Invariant:



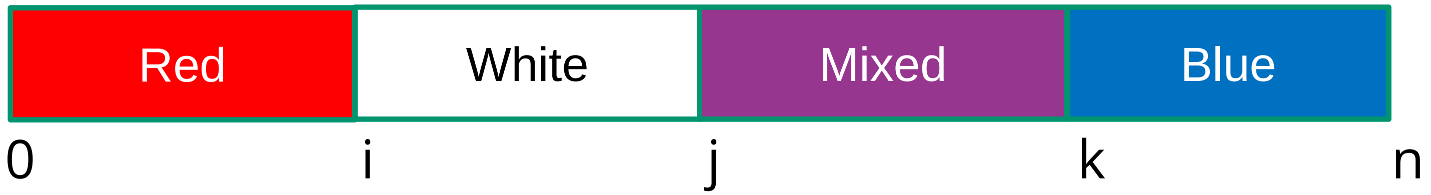
Initialization?

i = j = 0 and k = n

Termination Condition?

Dutch National Flag Code

Invariant:



Initialization?

i = j = 0 and k = n

Termination Condition?

j = k

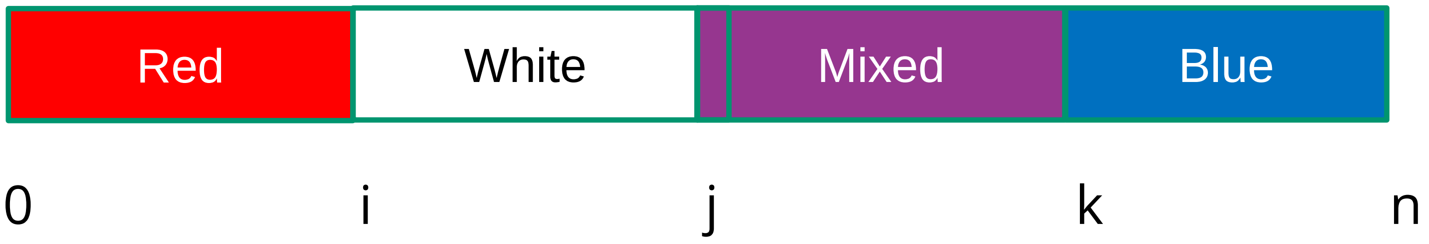
Dutch National Flag Code

```
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) {


   ??


}
```

$\Bigg\}$ 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]:

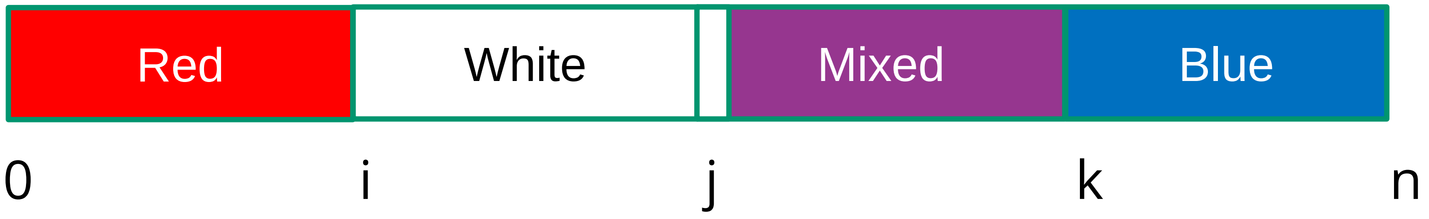


A[j] is either red, white, or blue

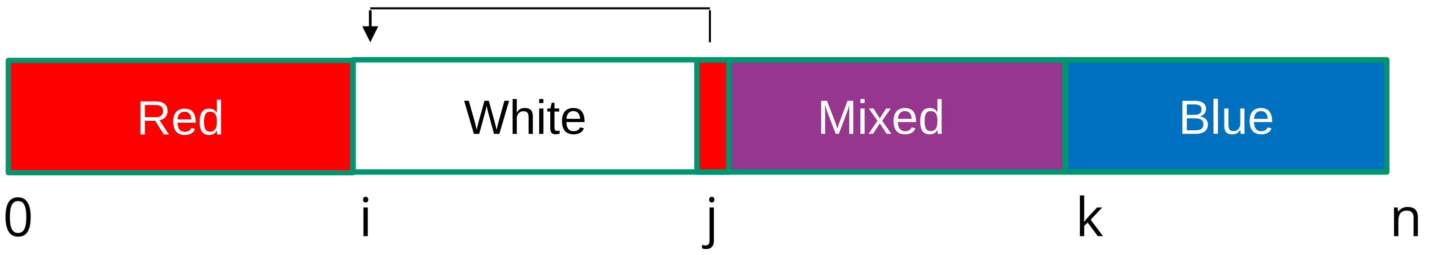
Dutch National Flag Code

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

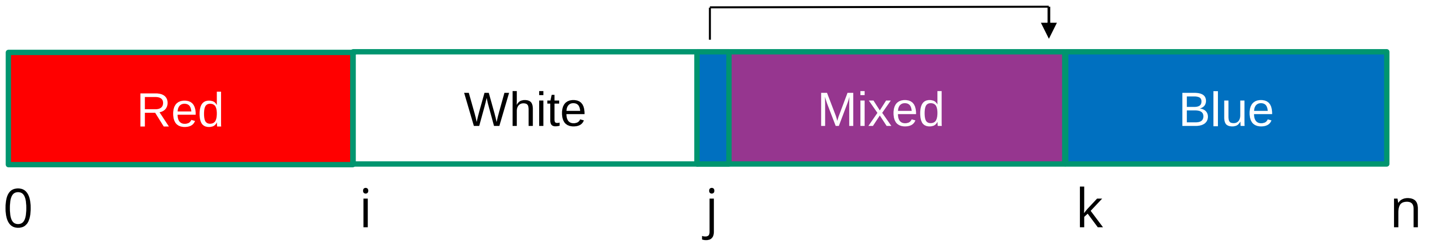
white



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;
    }
}
```

notice that we make progress at each step

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.

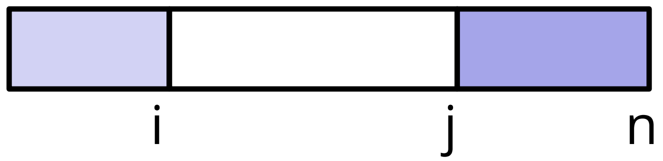
Idea: Look at A[n/2] to figure out if x is in A[0], A[1], ..., A[n/2] or in A[n/2+1], ..., A[n-1]. Narrow the search for x on each iteration.

(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.

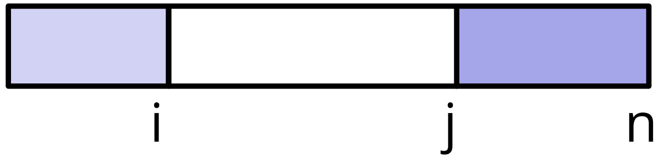
Idea: Look at A[n/2] to figure out if x is in A[0], A[1], ..., A[n/2] or in A[n/2+1], ..., A[n-1]. Narrow the search for x on each iteration.



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

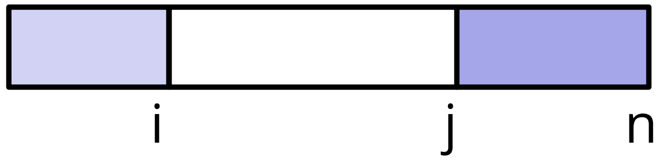
- A[i], ..., A[j-1] is the part where we don't know relation to x

Binary Search Code



Initialization?

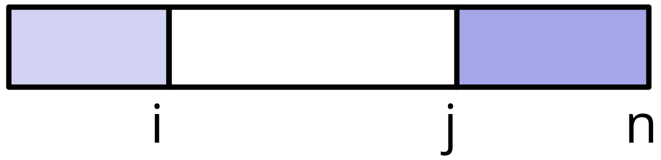
Binary Search Code



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) {



    // 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;





}
{{ 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] }}
```

What goes here?

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] }}
```

What goes here?

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)

Next Time…

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

Before next class...

1. If you haven't already done it, do Prep. Quiz: HW2 tonight!
   - Reasoning questions
   - Designed to take at most 15 minutes

2. Read the HW2 spec early!
   - Reasoning worksheet
   - Environment setup
   - Applying reasoning to code