CSE 331
Software Design & Implementation

James Wilcox

Autumn 2021

Lecture 4 – Writing Loops

Updates

- We are nearing the end of Hoare logic part of the course
- HW2 will be out later this week, focusing on loops
- Already time to think about HW3, our first Java homework
- For section tomorrow:
  - complete first step of HW3
  - bring laptop

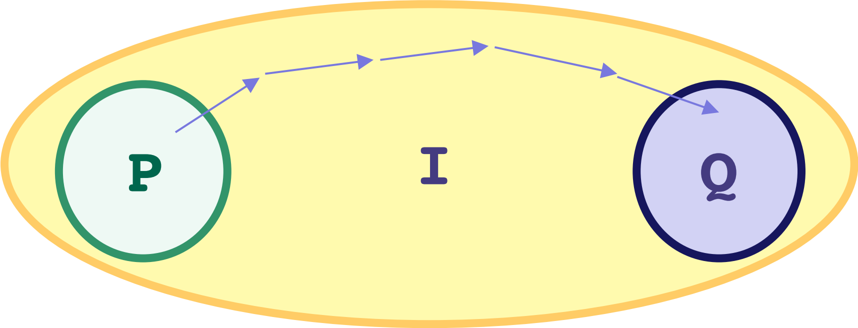
Previously on CSE 331...

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

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

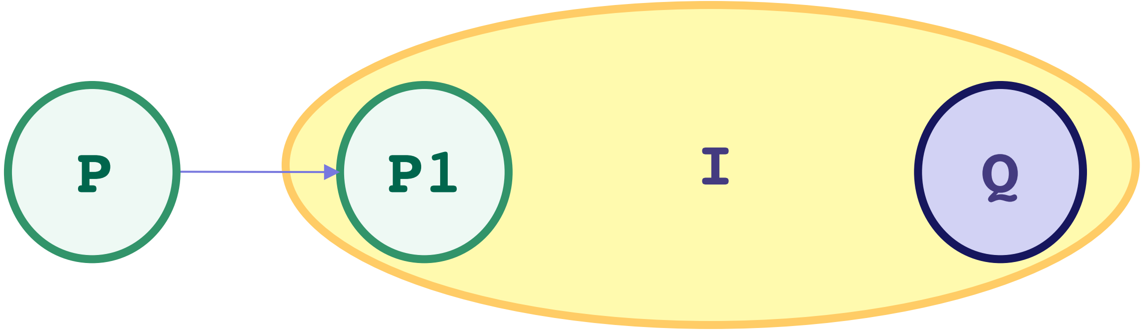
- Essence of the algorithm idea is:
  - invariant
  - how you make progress on each step (e.g., `i = i + 1`)

- Code is *ideally* just details...

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



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) {

    ??
                     ↑ {{ m = max(b[0], …, b[i]) }}
    i = i + 1;
                     | {{ m = max(b[0], …, b[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) {
                  {{ m = max(b[0], …, b[i-1]) }}

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

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

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

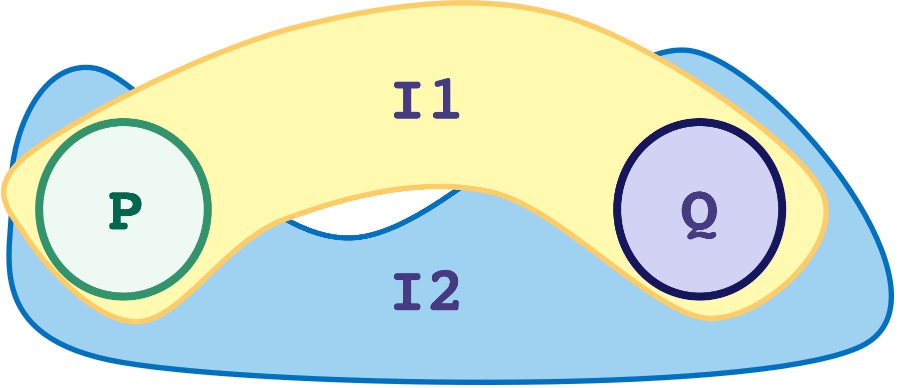
Invariant + progress step is the essence of the algorithm idea

- 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

m = max(b[0], ..., b[i-1])



m = max(b[i], ..., b[n-1])

Loop Invariant Design Pattern

Loop invariant is often a weakening of the postcondition

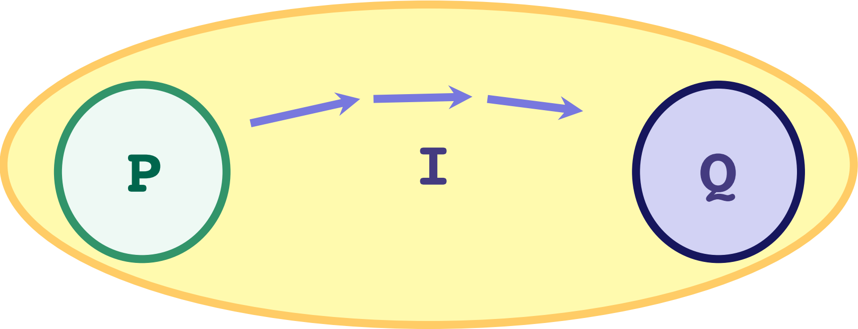
- partial progress with completion a special case
- small enough weakening that Inv + one condition gives Q

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 = 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 may ask you to inspect code with more complex invariants
- to learn about more ways of finding invariants: CSE 421

Another Example

Back to HW0

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

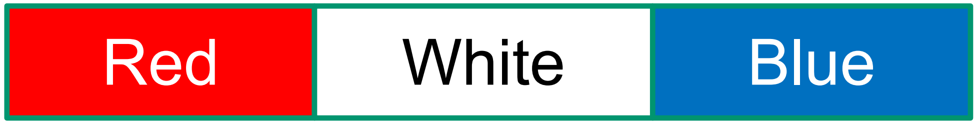
Pre- and post-conditions

Precondition: Any mix of red, white, and blue

Mixed colors: red, white, blue

Postcondition:

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



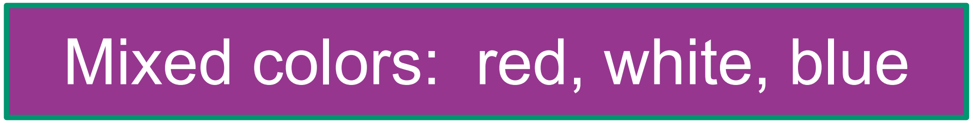
Red

White

Blue

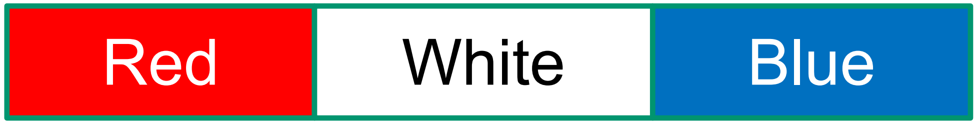
Pre- and post-conditions

Precondition: Any mix of red, white, and blue



Postcondition:

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

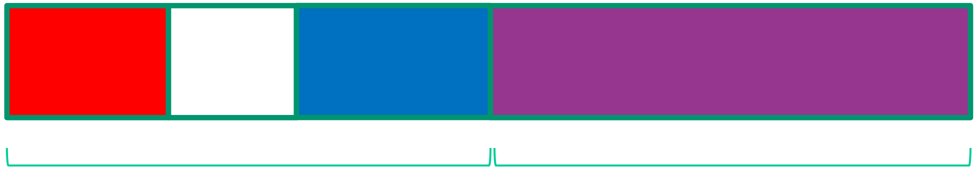


Want an invariant with

- 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:

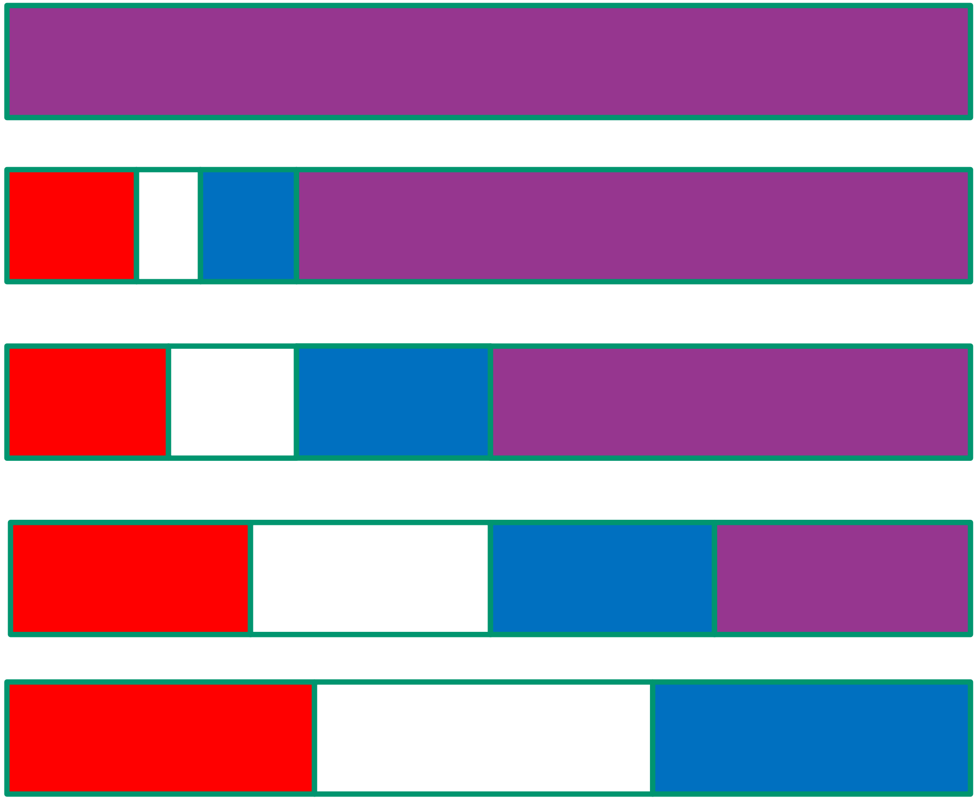


like postcondition

like initial condition

Example: Dutch National Flag

The first idea that comes to mind works.



Initial:

Iter 5:

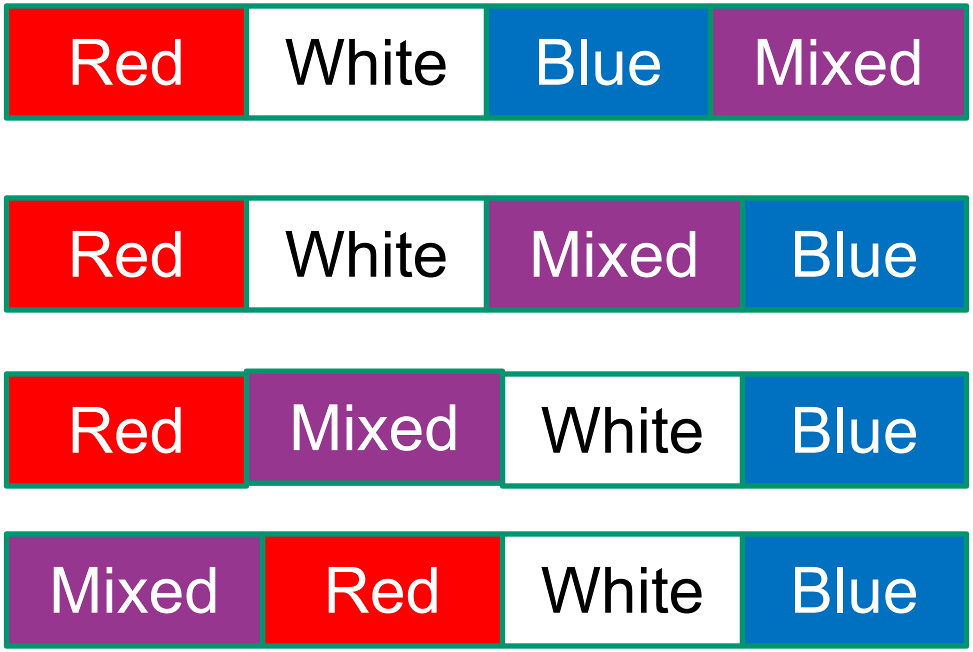
Iter 10:

Iter 15:

Post:

Other potential invariants

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



Red | White | Blue | Mixed

Red | White | Mixed | Blue

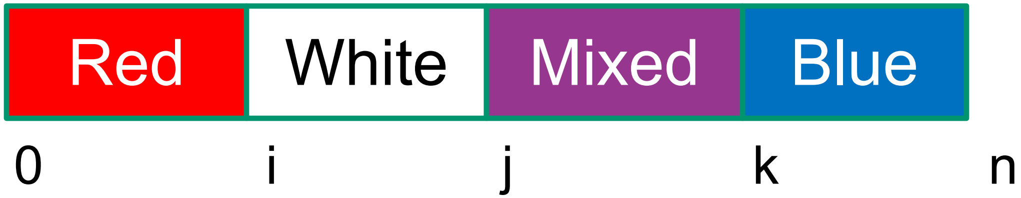
Red | Mixed | White | Blue

Mixed | Red | White | Blue

Precise Invariant

Need indices to refer to the split points between colors

- call these i, j, k



Red White Mixed Blue

0 i j k n

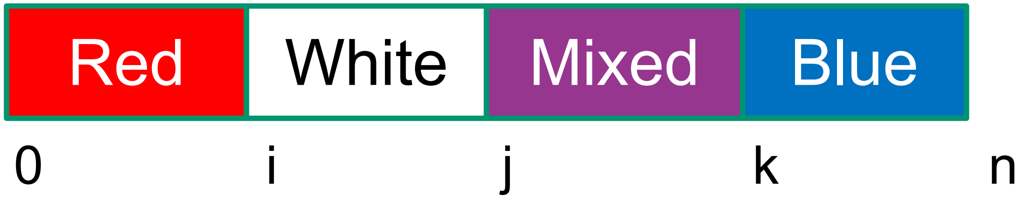
Loop Invariant:

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

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

Dutch National Flag Code

Invariant:



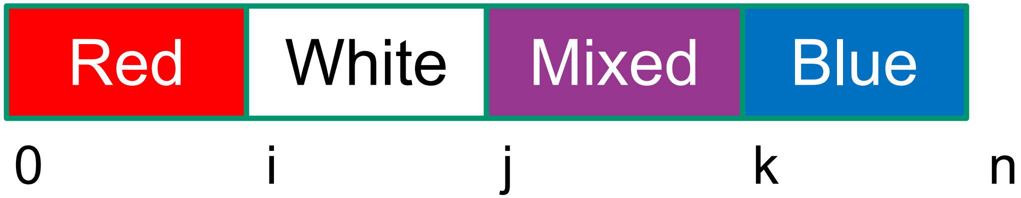
Red White Mixed Blue

0 i j k n

Initialization?

Dutch National Flag Code

Invariant:

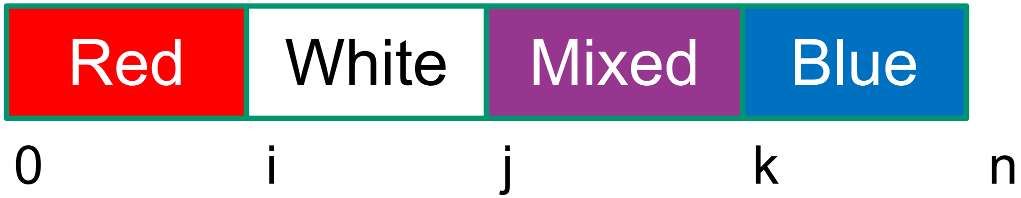


Initialization:

- i = j = 0 and k = n

Dutch National Flag Code

Invariant:



Red White Mixed Blue

0 i j k n

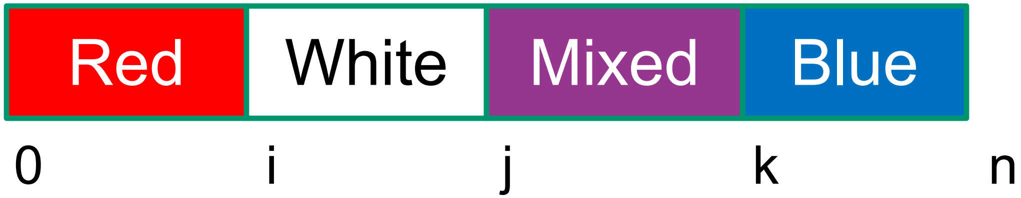
Initialization:

- i = j = 0 and k = n

Termination condition?

Dutch National Flag Code

Invariant:



Red White Mixed Blue

0 i j k n

Initialization:

- i = j = 0 and k = n

Termination condition:

- j = k

Dutch National Flag Code

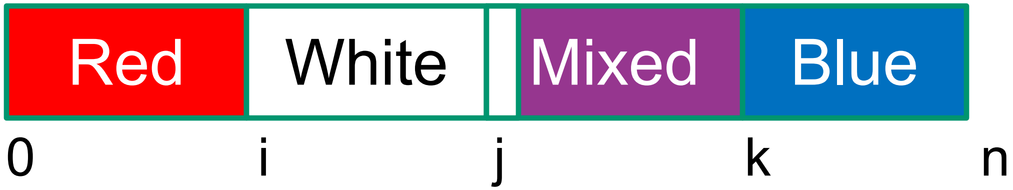
```
int i = 0, j = 0;
int k = n;
{{ Inv: 0 <= i <= j <= k <= n and A[0], …, A[i-1] are red and ... }}
while (j != k) {

   ??

}
```

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

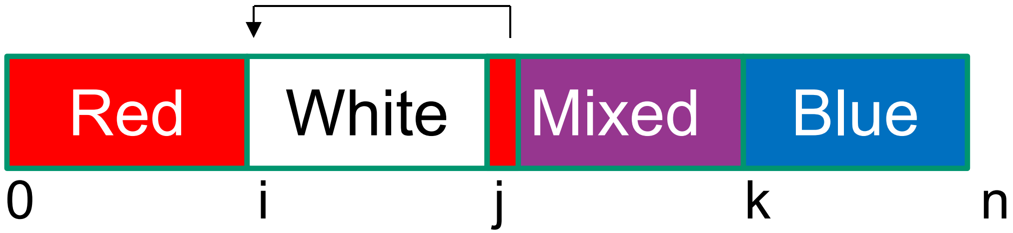
Dutch National Flag Code

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

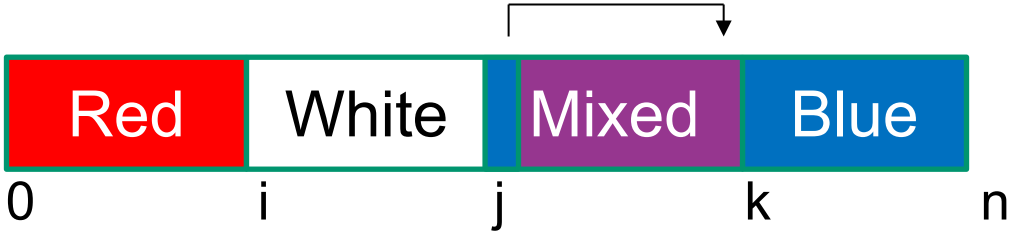
white



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;
  }
}
```