CSE 331
Software Design & Implementation

Kevin Zatloukal

Spring 2020

Lecture 4 – Writing Loops

Administrivia

- HW2 is split into two parts...
- HW2 (part 1) on loops, out now, due Wednesday
  - correctness of a loop
  - given invariant, fill in (simple) code
- HW2 (part 2) on loops, due next Monday
- Thursday section on preparing for HW3
  - should be quick if you attend section

Recap + Q&A

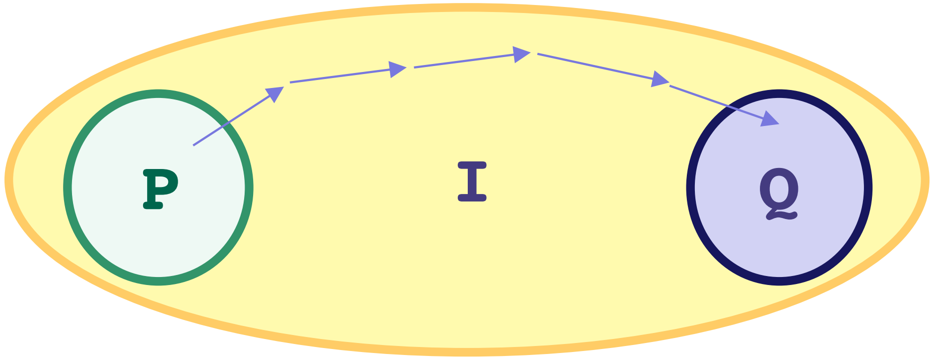
Previously on CSE 331...

```
{{ 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



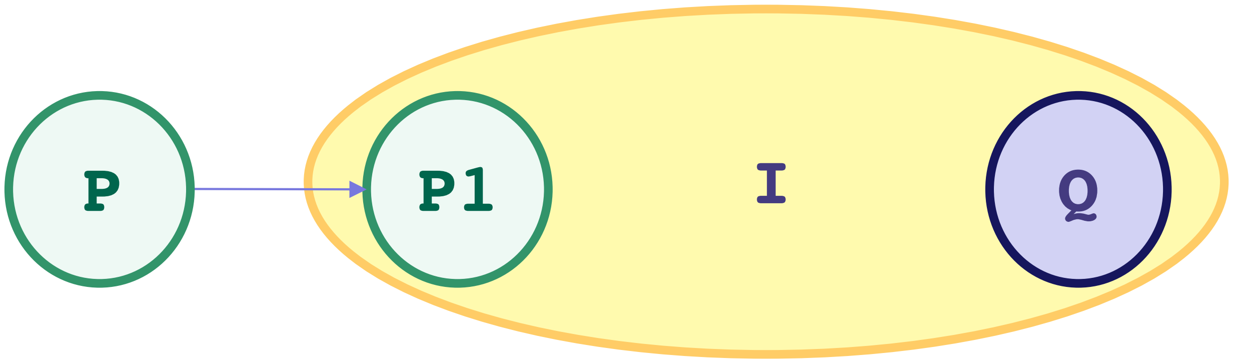
Loop Invariants

- 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 that follow from that idea...

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 (?) {

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

Algorithm idea

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) {

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

{{ 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];
```

Easiest way to make this hold?
Take i = 1 and m = max(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) {
    ??
    i = i + 1;
}
```

{{ m = max(b[0], …, b[i-1]) }}

{{ m = max(b[0], …, b[i]) }}

{{ m = max(b[0], …, b[i-1]) }}

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

How do we fill this in?
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]) }}
```

Loop Invariant → Code

- Benefits of improving at reasoning about code:
  1. find bugs
  2. thinking at the level of invariants

- To try out an invariant, think through:
  - can we easily check if the postcondition is satisfied?
  - can we easily get into a state that satisfies it?
  - can we efficiently make progress toward termination?

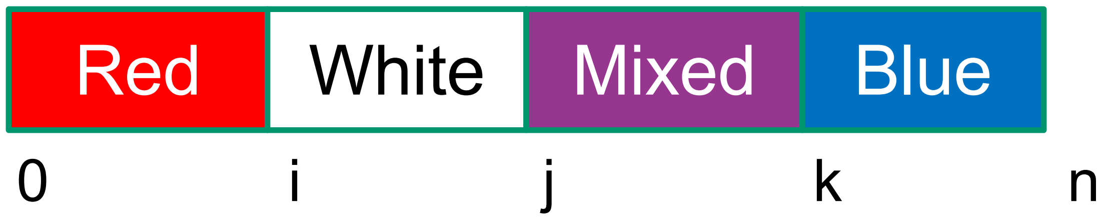
Example: Dutch National Flag

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

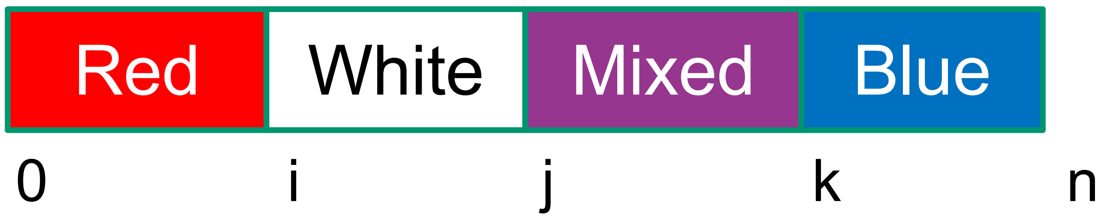
Dutch National Flag Code

Invariant:



Dutch National Flag Code

Invariant:

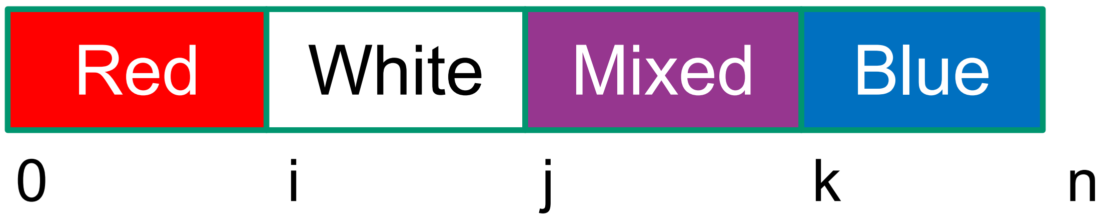


Termination condition:

- j = k

Dutch National Flag Code

Invariant:



Termination condition:

- j = k

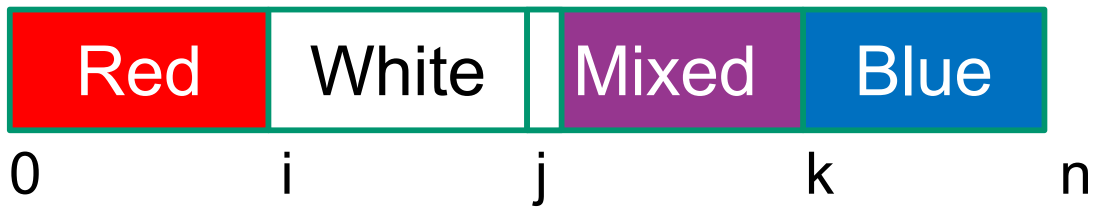
Initialization:

- i = 0, j = i, and k = n

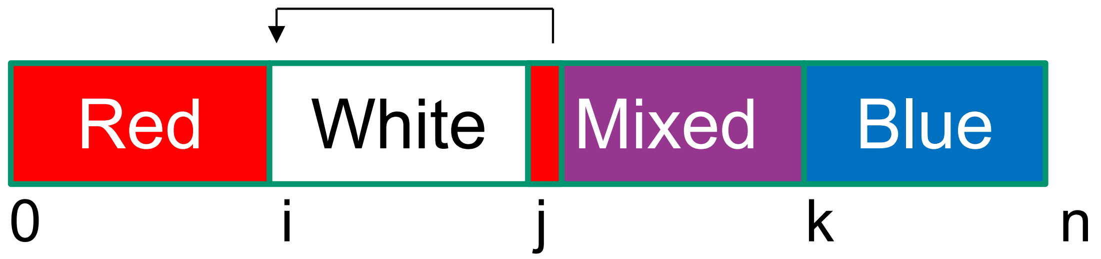
Dutch National Flag Code

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

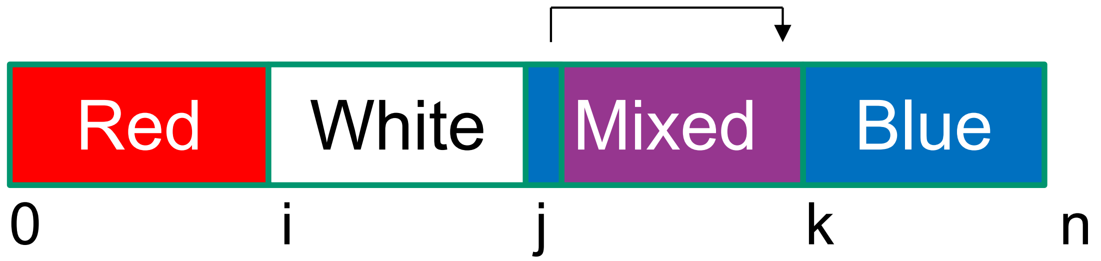
white



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;
  }
}
```

Loop Invariants

- Both of these invariants are **weakened** version of postcondition
  - describes partial vs complete solution
  - that is typical (esp. in this class)

- See CSE 421 for more on how to come up with invariants

Q & A

Another Example

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 an easy way to satisfy q*y + r = x and 0 <= r

```
{{ 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 an easy way to satisfy q*y + r = x and 0 <= r
- how about q = 0?
  - then we need r = x... and that is okay since 0 <= x

```
{{ 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

```
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) {

  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*y + r = x and 0 <= r and y <= r }}

                            ↑ {{ q*y + r-y = x and 0 <= r-y }}
  r = r - y;                  {{ q*y + r = x and 0 <= r }}
}
{{ 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) {

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

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

$\downarrow$ {{ q*y + r = x and 0 <= r and y <= r }}

$\uparrow$ {{ q*y + r-y = x and 0 <= r-y }}

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

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) {

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

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

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

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

↑ {{ q*y + r-y = x and 0 <= r-y }}

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

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 }}

Aside on Efficiency

- This is not an efficient agorithm
  - runs in O(x/y) time, which could be huge (e.g. $x/y = 2^{63}$)
  - but it is correct

- Grade school “long division“ is much more efficient
  - runs in $O((\log x)^2)$ time
  - makes progress in larger steps
    - (needs a more complex invariant)