3.2 Dutch Flag Problem

Slide 317

We want to discuss a “larger” program. That is, one that uses both loops and method calls, so that we can use everything we have learnt so far.

Dutch Flag Problem - Motivation

The "Dutch National Flag Problem" is a famous Computer Science related programming problem originally proposed by Dijkstra:

> *The flag of the Netherlands consists of three colours: Red, White and Blue. Given balls of these three colours arranged randomly, rearrange them such that the balls of the same colour are together and their collective colour groups are in the correct order.*

Problem description adapted from `http://en.wikipedia.org/wiki/Dutch_national_flag_problem`

The Dutch National Flag Problem was first published as part of:

Dijkstra, E. W. "Letters to the editor: goto statement considered harmful" Communications of the ACM II (3).

From the 1970's Dijkstra's main interest was program derivation. That is, starting with a mathematical specification, apply mathematical transformations to the specification until it is turned into a program that can be executed.

Slide 319

Dutch Flag Problem - The Task

Preliminaries: We are provided with an enumeration type `Colour` with members `Red`, `White` and `Blue`.

Problem: Given an array of colours `Colour[] a`, rearrange `a` so that all of the `Red` entries occur before all of the `White` entries and these again occur before all of the `Blue` entries.

Idea: We limit modification of the array to swapping elements and use a helper method `swap(a,i,j)` for this.

Slide 320

Dutch Flag Problem - Pseudo Code

We can use the following rough algorithm:

1. Set the unprocessed range to be from `0` to `a.length`
2. Look at the first element of `a` in the unprocessed range.

3a. If it is `Red`, then it is already in the correct place and we can move on.

3b. If it is `White`, then we need to somehow swap this to the middle of the array and move on.

3c. If it is `Blue`, then we need to somehow swap this to the end of the array and move on.

4. Repeat this process from step 2 until we have processed all elements of the array.

In each of the cases 3a, 3b and 3c we reduce the unprocessed range by one element.

Slide 321

Dutch Flag Problem - Useful Predicates

For arrays a and b and natural numbers i and j:

$$
\begin{aligned}
\mathit{Swapped}(\mathtt{a},\mathtt{b},\mathtt{i},\mathtt{j}) \quad \longleftrightarrow \quad & \mathtt{a.length} = \mathtt{b.length} \\
& \land\ \mathtt{i},\mathtt{j} \in [0..\mathtt{a.length}) \\
& \land\ \mathtt{b[i]} = \mathtt{a[j]} \\
& \land\ \mathtt{b[j]} = \mathtt{a[i]} \\
& \land\ \forall k{\in}[0..\mathtt{a.length})\backslash\{\mathtt{i},\mathtt{j}\}.[\ \mathtt{a}[k] = \mathtt{b}[k]\ ]
\end{aligned}
$$

Slide 322

Dutch Flag Problem - Informal Specification

```
enum Colour = { Red, White, Blue }

void reorder(Colour[] a)
// PRE: a ≠ null
// POST: a ~ a₀  ∧  ∃k₁, k₂ ∈ ℕ.
//            [ a[0..k₁] = Red  ∧  a(k₁..k₂) = White
//                               ∧  a[k₂..a.length) = Blue ]
{ ... }
```

Is that enough?

- What if no Red/White/Blue entries in a?
- Does the postcondition imply that $k_1 < k_2$?
- Does the postcondition imply that $k_1, k_2 \in [0..\mathtt{a.length})$

Note that the `reorder` method does not actually return the values k_1 or k_2, we just need to know that they exist and hence we have sorted the array as desired.

Slide 323

Dutch Flag Problem - Formalising the Specification

```
void reorder(Colour[] a)
// PRE: a ≠ null
// POST: a ~ a₀  ∧  ∃k₁, k₂ ∈ ℕ.
//                 [ a[0..k₁] = Red  ∧  a(k₁..k₂) = White
//                                   ∧  a[k₂..a.length) = Blue ]
{ ... }
```

What if no Red/White/Blue entries in a?

For $k_1, k_2 \in \mathbb{N}$ the range $[0..k_1]$ cannot be empty, but $(k_1..k_2)$ and $[k_2..\texttt{a.length})$ can.

We need to modify the ranges in our postcondition slightly to accommodate the no Red case.

Slide 324

Dutch Flag Problem - Formalising the Specification

```
void reorder(Colour[] a)
// PRE: a ≠ null
// POST: a ~ a₀  ∧  ∃k₁, k₂ ∈ ℕ.
//                 [ a[0..k₁) = Red  ∧  a[k₁..k₂) = White
//                                   ∧  a[k₂..a.length) = Blue ]
{ ... }
```

Does the postcondition imply that $k_1 < k_2$?

If there are no White elements in the array then we could have $k_1 = k_2$.

This is just something to be aware of, but requires no modifications to the specification.

Slide 325

Dutch Flag Problem - Formalising the Specification

```
void reorder(Colour[] a)
// PRE: a ≠ null
// POST: a ~ a₀ ∧ ∃k₁, k₂ ∈ ℕ.
//           [ a[0..k₁) = Red ∧ a[k₁..k₂) = White
//                            ∧ a[k₂..a.length) = Blue ]
{ ... }
```

Does the postcondition imply that $k_1, k_2 \in [0..\texttt{a.length})$

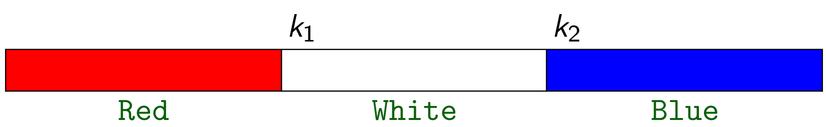
No. If there are no `Blue` elements in the array then k_2 could be any value $\geq$ `a.length`.

It would make sense to constrain these values in the postcondition, just to keep the meaning clear.

Slide 326

Dutch Flag Problem - Final Specification

```
enum Colour = { Red, White, Blue }

void reorder(Colour[] a)
// PRE: a ≠ null
// POST: a ~ a₀ ∧ ∃k₁, k₂ ∈ [0..a.length].
//           [ a[0..k₁) = Red ∧ a[k₁..k₂) = White
//                            ∧ a[k₂..a.length) = Blue ]
{ ... }
```



The final postcondition is simpler to establish and has consistent bounds on the sub ranges, making it easier to think about.

Slide 327

Swapping Elements in the Array - `swap`

```
void swap(Colour[] a, int i, int j)
// PRE: i, j ∈ [0..a.length)
// POST: Swapped(a, a0, i, j)
{ ... }
```

What happens if the array is empty?

Precondition becomes $\texttt{i}, \texttt{j} \in [0..0)$ which is unsatisfiable.

How might we implement `swap`?

An obvious implementation is:

```
Colour temp = a[i];
a[i] = a[j];
a[j] = temp;
```


Slide 328

Dutch Flag Problem - Code Skeleton

```
void reorder(Colour[] a)
// PRE: a ≠ null
// POST: a ∼ a0  ∧  ∃k1, k2 ∈ [0..a.length].
//                [ a[0..k1) = Red  ∧  a[k1..k2) = White  ∧  a[k2..a.length) = Blue ]
{

  ???

  // INV: ???
  //
  // VAR: ???
  while( ??? ) {


    ???


  }
  // MID: ???
  //
  ???
}
```

Slide 329

Dutch Flag Problem - Postcondition to Mindcondition

The postcondition states:

$$\begin{aligned}&\texttt{a} \sim \texttt{a}_0\\ &\quad \land\ \exists k_1, k_2 \in [0..\texttt{a.length}].[\ \texttt{a}[0..k_1) = \texttt{Red}\ \land\ \texttt{a}[k_1..k_2) = \texttt{White}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land\ \texttt{a}[k_2..\texttt{a.length}) = \texttt{Blue}\ ]\end{aligned}$$

We don't need to return anything, so we can choose to push all of the work into the loop.

i.e. choose:

$$\mathit{MID} \longleftrightarrow \mathit{POST}$$

Slide 330

Dutch Flag Problem - Code Skeleton

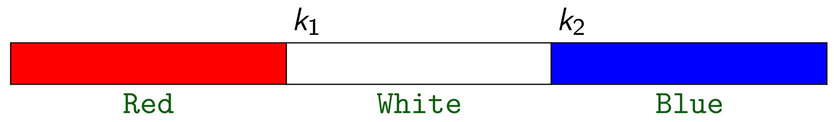
```
void reorder(Colour[] a)
// PRE: a ≠ null
// POST: a ∼ a₀  ∧  ∃k₁, k₂ ∈ [0..a.length].
//                 [ a[0..k₁) = Red  ∧  a[k₁..k₂) = White  ∧  a[k₂..a.length) = Blue ]
{

  ???

  // INV: ???
  //
  // VAR: ???
  while( ??? ) {


    ???


  }
  // MID: a ∼ a₀  ∧  ∃k₁, k₂ ∈ [0..a.length].
  //                 [ a[0..k₁) = Red  ∧  a[k₁..k₂) = White  ∧  a[k₂..a.length) = Blue ]
}
```

Slide 331

Dutch Flag Problem - Mindcondition to Invariant

Focusing just on the array, the postcondition states:

$$\exists k_1, k_2 \in [0..\texttt{a.length}].[\ \texttt{a}[0..k_1) = \texttt{Red}\ \wedge\ \texttt{a}[k_1..k_2) = \texttt{White}$$
$$\wedge\ \texttt{a}[k_2..\texttt{a.length}) = \texttt{Blue}\ ]$$

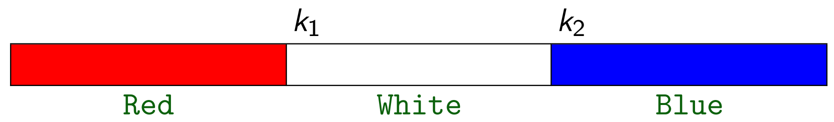


We need to generalise this property for some arbitrary mid-point after n loop iterations.

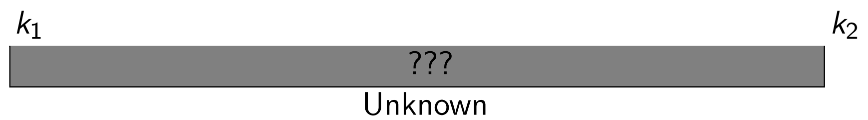
Slide 332

Dutch Flag Problem - Mindcondition to Invariant

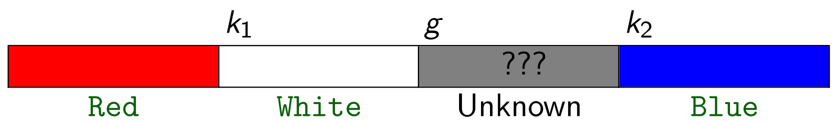
At the end of the method:



At the beginning of the method:



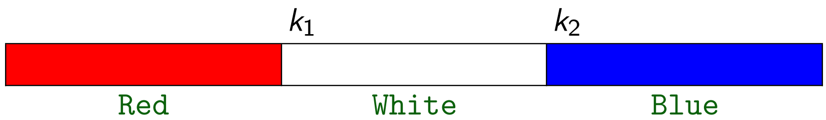
After some arbitrary number of iterations:



Slide 333

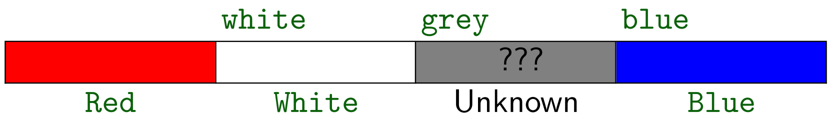
Dutch Flag Problem - Mindcondition to Invariant

In the postcondition:



$$\exists k_1, k_2 \in [0..\texttt{a.length}].[\ \texttt{a}[0..k_1) = \texttt{Red} \ \wedge\ \texttt{a}[k_1..k_2) = \texttt{White} \ \wedge\ \texttt{a}[k_2..\texttt{a.length}) = \texttt{Blue}\ ]$$

So for the invariant:



$$\texttt{a}[0..\texttt{white}) = \texttt{Red} \ \wedge\ \texttt{a}[\texttt{white}..\texttt{grey}) = \texttt{White} \ \wedge\ \texttt{a}[\texttt{blue}..\texttt{a.length}) = \texttt{Blue} \ \wedge\ 0 \leq \texttt{white} \leq \texttt{grey} \leq \texttt{blue} \leq \texttt{a.length}$$

For the invariant, it is not enough just to know that some k_1, k_2 and g exist. We need to know these bounds exactly, so that we can modify them in the code and track them in our proofs.

Thus, we introduce the program variables `white`, `grey` and `blue` to track the start of their respective regions in the array.

Once we have an invariant, it is a good idea to check its sanity in some edge cases.

Interesting Question:

Can the invariant handle the case where there are no `Red`, `White` or `Blue` elements?

Answer:

Yes. All of the array ranges can be collapsed to empty ranges if required.

Interesting Question:

Does the invariant make sense when `white`, `grey` or `blue` overlap or are all equal to 0 or `a.length`?

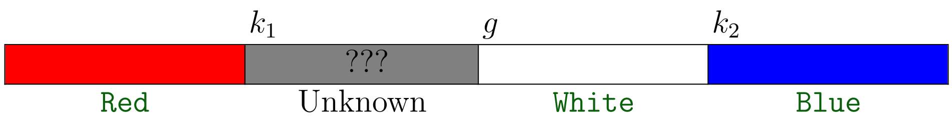
Answer:

Yes. The invariant can handle all of these cases.

- The `Red` and `White` regions cannot overlap as they are separated by the `white` element.
- The `White` and `Blue` regions cannot overlap due to the bounds on the variables `white`, `grey` and `blue`.
- If `grey` = `blue` then the unknown region must be empty.
- If `white` = `grey` = `blue` = 0 then every element of the array must be `Blue`.
- If `white` = `grey` = `blue` = `a.length` then every element of the array must be `Red`.
- We could also specify all elements of the array being `White` by setting `white` = 0 and `grey` = `blue` = `a.length`.

It is also worth noting that there are several other possible choices of the invariant based on where we choose to track the unknown region.

For example, you could have:



or even:

(although this last version would require some rather complex code to keep the `White` region the right place.

Slide 334

Dutch Flag Problem - Code Skeleton

```
void reorder(Colour[] a)
// PRE: a ≠ null
// POST: a ∼ a₀  ∧  ∃k1, k2 ∈ [0..a.length].
//                    [ a[0..k1) = Red  ∧  a[k1..k2) = White  ∧  a[k2..a.length) = Blue ]
{

  ???

  // INV: a ∼ a₀  ∧  0 ≤ white ≤ grey ≤ blue ≤ a.length
  //        ∧  a[0..white) = Red  ∧  a[white..grey) = White  ∧  a[blue..a.length) = Blue
  // VAR: ???
  while( ??? ) {


    ???


  }
  // MID: a ∼ a₀  ∧  ∃k1, k2 ∈ [0..a.length].
  //                  [ a[0..k1) = Red  ∧  a[k1..k2) = White  ∧  a[k2..a.length) = Blue ]
}
```


Slide 335

Dutch Flag Problem - Invariant to Initialisation Code

Invariant:

$$\texttt{a} \sim \texttt{a}_0 \ \wedge\ 0 \leq \texttt{white} \leq \texttt{grey} \leq \texttt{blue} \leq \texttt{a.length}$$
$$\wedge\ \texttt{a}[0..\texttt{white}) = \texttt{Red} \wedge \texttt{a}[\texttt{white}..\texttt{grey}) = \texttt{White} \wedge \texttt{a}[\texttt{blue}..\texttt{a.length}) = \texttt{Blue}$$

The invariant can be vacuously established if:

- $\texttt{white} = 0$
- $\texttt{grey} = 0$
- $\texttt{blue} = \texttt{a.length}$

So we should set this up with sensible initialisation code.

Slide 336

Dutch Flag Problem - Code Skeleton

```
void reorder(Colour[] a)
// PRE: a ≠ null
// POST: a ~ a₀  ∧  ∃k₁, k₂ ∈ [0..a.length].
//                 [ a[0..k₁) = Red  ∧  a[k₁..k₂) = White  ∧  a[k₂..a.length) = Blue ]
{
  int white = 0;
  int grey = 0;
  int blue = a.length;
  // INV: a ~ a₀  ∧  0 ≤ white ≤ grey ≤ blue ≤ a.length
  //       ∧  a[0..white) = Red  ∧  a[white..grey) = White  ∧  a[blue..a.length) = Blue
  // VAR: ???
  while( ??? ) {


    ???


  }
  // MID: a ~ a₀  ∧  ∃k₁, k₂ ∈ [0..a.length].
  //                 [ a[0..k₁) = Red  ∧  a[k₁..k₂) = White  ∧  a[k₂..a.length) = Blue ]
}
```


Slide 337

Dutch Flag Problem - Invariant to Loop Condition

Invariant:

$$\mathtt{a} \sim \mathtt{a}_0 \ \wedge\ 0 \leq \mathtt{white} \leq \mathtt{grey} \leq \mathtt{blue} \leq \mathtt{a.length}$$
$$\wedge\ \mathtt{a}[0..\mathtt{white}) = \mathtt{Red} \wedge \mathtt{a}[\mathtt{white}..\mathtt{grey}) = \mathtt{White} \wedge \mathtt{a}[\mathtt{blue}..\mathtt{a.length}) = \mathtt{Blue}$$

and Negation of Condition:

???

implies Midcondition:

$$\mathtt{a} \sim \mathtt{a}_0 \ \wedge\ \exists k_1, k_2 \in [0..\mathtt{a.length}].$$
$$[\ \mathtt{a}[0..k_1) = \mathtt{Red}\ \wedge\ \mathtt{a}[k_1..k_2) = \mathtt{White}\ \wedge\ \mathtt{a}[k_2..\mathtt{a.length}) = \mathtt{Blue}\ ]$$

Candidates for *negation* of condition:

- $\mathtt{grey} = \mathtt{blue}$

This really is the only possible option, as we want to have just two variables delimiting the ranges of the array.

Of course, this means that our actual loop condition must establish that `grey` $\neq$ `blue`. The simplest expression for this is `(grey != blue)`.

Slide 338

Part III: Reasoning Case Studies Dutch Flag Problem

Dutch Flag Problem - Code Skeleton

```
void reorder(Colour[] a)
// PRE: a ≠ null
// POST: a ~ a0  ∧  ∃k1, k2 ∈ [0..a.length].
//                  [ a[0..k1) = Red  ∧  a[k1..k2) = White  ∧  a[k2..a.length) = Blue ]
{
  int white = 0;
  int grey = 0;
  int blue = a.length;
  // INV: a ~ a0  ∧  0 ≤ white ≤ grey ≤ blue ≤ a.length
  //      ∧  a[0..white) = Red  ∧  a[white..grey) = White  ∧  a[blue..a.length) = Blue
  // VAR: ???
  while( grey != blue ) {


    ???


  }
  // MID: a ~ a0  ∧  ∃k1, k2 ∈ [0..a.length].
  //                [ a[0..k1) = Red  ∧  a[k1..k2) = White  ∧  a[k2..a.length) = Blue ]
}
```

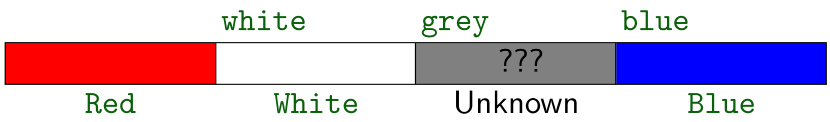
Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 338 / 400

Slide 339

Dutch Flag Problem - Invariant to Loop Body

Invariant:

$$a \sim a_0 \ \wedge \ 0 \leq \texttt{white} \leq \texttt{grey} \leq \texttt{blue} \leq \texttt{a.length}$$
$$\wedge \ \texttt{a}[0..\texttt{white}) = \texttt{Red} \ \wedge \ \texttt{a}[\texttt{white}..\texttt{grey}) = \texttt{White} \ \wedge \ \texttt{a}[\texttt{blue}..\texttt{a.length}) = \texttt{Blue}$$



Loop Test: `grey != blue`

So there must be an element to examine at `a[grey]`

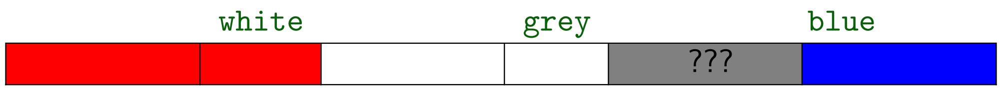
Three possible cases:

1. `a[grey]` = `Red`
2. `a[grey]` = `White`
3. `a[grey]` = `Blue`

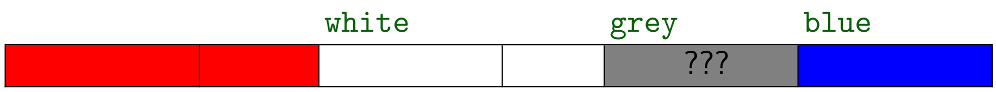
Slide 340

Dutch Flag Problem - Loop Body `a[grey]` = `Red`


```
swap(a, white, grey);
```

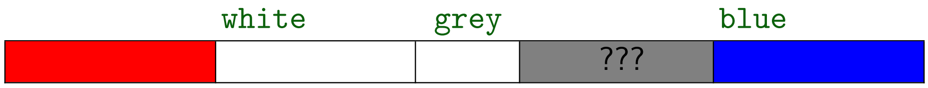



```
white++; grey++;
```

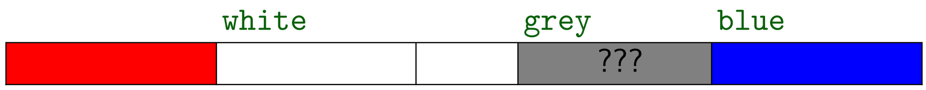


Slide 341

Dutch Flag Problem - Loop Body `a[grey] == White`

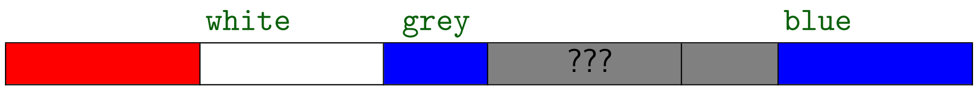



```
grey++;
```

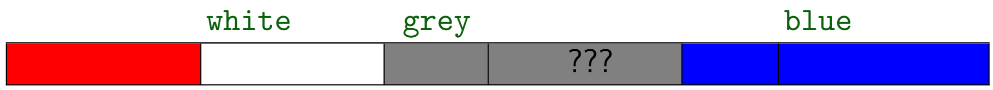


Slide 342

Dutch Flag Problem - Loop Body `a[grey] == Blue`




```
swap(a, blue-1, grey);
```




```
blue--;
```

Dutch Flag Problem - Loop Body Code

These three cases lead to the following natural code:

```
switch( a[grey] ) {
  case Red:   swap(a, white, grey);
              white++;
              grey++;
              break;

  case White: grey++;
              break;

  case Blue:  swap(a, blue-1, grey);
              blue--;
              break;
}
```


Of course, there are other possibilities. For example, we could traverse the "unknown" part of the array from right to left by inspecting the element at index `blue-1` on each iteration of the loop.

Slide 344

Dutch Flag Problem - Code Skeleton

```
void reorder(Colour[] a)
// PRE: a ≠ null
// POST: a ∼ a0  ∧  ∃k1, k2 ∈ [0..a.length].
//                    [ a[0..k1) = Red  ∧  a[k1..k2) = White  ∧  a[k2..a.length) = Blue ]
{
  int white = 0;
  int grey = 0;
  int blue = a.length;
  // INV: a ∼ a0  ∧  0 ≤ white ≤ grey ≤ blue ≤ a.length
  //       ∧  a[0..white) = Red  ∧  a[white..grey) = White  ∧  a[blue..a.length) = Blue
  // VAR: ???
  while( grey != blue ) {
    switch( a[grey] ) {
      case Red:   swap(a, white, grey); white++; grey++; break;
      case White: grey++; break;
      case Blue:  swap(a, blue-1, grey); blue--; break;
    }
  }
  // MID: a ∼ a0  ∧  ∃k1, k2 ∈ [0..a.length].
  //                  [ a[0..k1) = Red  ∧  a[k1..k2) = White  ∧  a[k2..a.length) = Blue ]
}
```


Slide 345

Dutch Flag Problem - Loop Body to Variant

Loop Body:

```
switch( a[grey] ) {
  case Red:   swap(a, white, grey);
              white++;
              grey++;
              break;

  case White: grey++;
              break;

  case Blue:  swap(a, blue-1, grey);
              blue--;
              break;
}
```

Notice that either `grey` increases or `blue` decreases on each iteration.
So the distance between them always decreases.
Possible Variant: `blue` − `grey`.

Slide 346

Dutch Flag Problem - Complete Code and Spec.

```
void reorder(Colour[] a)
// PRE: a ≠ null                                                              (P)
// POST: a ∼ a0  ∧  ∃k1, k2 ∈ [0..a.length].                                   (Q)
//            [ a[0..k1) = Red  ∧  a[k1..k2) = White  ∧  a[k2..a.length) = Blue ]
{
  int white = 0;
  int grey = 0;
  int blue = a.length;
  // INV: a ∼ a0  ∧  0 ≤ white ≤ grey ≤ blue ≤ a.length                        (I)
  //       ∧ a[0..white) = Red  ∧  a[white..grey) = White  ∧  a[blue..a.length) = Blue
  // VAR: blue − grey                                                          (V)
  while( grey != blue ) {
    switch( a[grey] ) {
      case Red:   swap(a, white, grey); white++; grey++; break;
      case White: grey++; break;
      case Blue:  swap(a, blue-1, grey); blue--; break;
    }
  }
  // MID: a ∼ a0  ∧  ∃k1, k2 ∈ [0..a.length].                                  (M)
  //            [ a[0..k1) = Red  ∧  a[k1..k2) = White  ∧  a[k2..a.length) = Blue ]
}
```


Slide 347

Dutch Flag Problem - Verifying the Code

The proof obligations are:

(a) The loop invariant holds before the loop is entered.

(b) Given the condition, the loop body re-establishes the loop invariant.

(c) Termination of the loop and the loop invariant imply the mid-condition immediately after loop.

(d) The variant is bounded.

(e) The variant decreases with each loop iteration.

(f) All array accesses within the method are valid.

By the code construction technique we have employed, these proofs ought to be pretty straightforward. However, we should still check them, just to be sure that we have not made any mistakes.

(a) Invariant holds before the loop is entered

$$P[\mathtt{a} \mapsto \mathtt{a}_0] \;\wedge\; \mathtt{white} = 0 \;\wedge\; \mathtt{grey} = 0 \;\wedge\; \mathtt{blue} = \mathtt{a.length} \;\wedge\; \mathtt{a} \approx \mathtt{a}_0$$
$$\longrightarrow$$
$$I$$

(b) Loop body re-establishes the loop invariant

$$\begin{array}{c} I \;\wedge\; \mathtt{grey} \neq \mathtt{blue} \\ \wedge\; (\mathtt{a[grey]} = \mathtt{Red} \longrightarrow \mathit{Swapped}(\mathtt{a}', \mathtt{a}, \mathtt{white}, \mathtt{grey}) \wedge \mathtt{white}' = \mathtt{white} + 1 \\ \wedge\, \mathtt{grey}' = \mathtt{grey} + 1 \wedge \mathtt{blue}' = \mathtt{blue}) \\ \wedge\; (\mathtt{a[grey]} = \mathtt{White} \longrightarrow \mathit{Swapped}(\mathtt{a}', \mathtt{a}, \mathtt{grey}, \mathtt{grey}) \wedge \mathtt{white}' = \mathtt{white} \\ \wedge\, \mathtt{grey}' = \mathtt{grey} + 1 \wedge \mathtt{blue}' = \mathtt{blue}) \\ \wedge\; (\mathtt{a[grey]} = \mathtt{Blue} \longrightarrow \mathit{Swapped}(\mathtt{a}', \mathtt{a}, \mathtt{blue\text{-}1}, \mathtt{grey}) \wedge \mathtt{white}' = \mathtt{white} \\ \wedge\, \mathtt{grey}' = \mathtt{grey} \wedge \mathtt{blue}' = \mathtt{blue} - 1) \\ \longrightarrow \\ I[\mathtt{a} \mapsto \mathtt{a}', \mathtt{white} \mapsto \mathtt{white}', \mathtt{grey} \mapsto \mathtt{grey}', \mathtt{blue} \mapsto \mathtt{blue}'] \end{array}$$

We also need to be sure that we satisfy the precondition of `swap` at both of its call points in our code. For the sake of brevity, we compress both of these checks into a single proof obligation:

$$I \;\wedge\; \mathtt{grey} \neq \mathtt{blue}$$
$$\longrightarrow$$
$$\mathtt{white}, \mathtt{grey}, \mathtt{blue\text{-}1} \in [0..\mathtt{a.length})$$

(c) Midcondition holds straight after loop

$$I \;\wedge\; \mathtt{grey} = \mathtt{blue}$$
$$\longrightarrow$$
$$M$$

(d) + (e) The loop terminates

$$
\begin{array}{c}
I \ \wedge \ \texttt{grey} \neq \texttt{blue} \\
\wedge\ (\texttt{a[grey]} = \texttt{Red} \longrightarrow \mathit{Swapped}(\texttt{a}', \texttt{a}, \texttt{white}, \texttt{grey}) \wedge \texttt{white}' = \texttt{white} + 1 \\
\qquad \wedge\, \texttt{grey}' = \texttt{grey} + 1 \wedge \texttt{blue}' = \texttt{blue}) \\
\wedge\ (\texttt{a[grey]} = \texttt{White} \longrightarrow \mathit{Swapped}(\texttt{a}', \texttt{a}, \texttt{grey}, \texttt{grey}) \wedge \texttt{white}' = \texttt{white} \\
\qquad \wedge\, \texttt{grey}' = \texttt{grey} + 1 \wedge \texttt{blue}' = \texttt{blue}) \\
\wedge\ (\texttt{a[grey]} = \texttt{Blue} \longrightarrow \mathit{Swapped}(\texttt{a}', \texttt{a}, \texttt{blue-1}, \texttt{grey}) \wedge \texttt{white}' = \texttt{white} \\
\qquad \wedge\, \texttt{grey}' = \texttt{grey} \wedge \texttt{blue}' = \texttt{blue} - 1) \\
\longrightarrow \\
V \text{ bound } \wedge \ V[\texttt{a} \mapsto \texttt{a}', \texttt{white} \mapsto \texttt{white}', \texttt{grey} \mapsto \texttt{grey}', \texttt{blue} \mapsto \texttt{blue}']
\end{array}
$$

(f) Array access are legal

$$
\begin{array}{c}
I \ \wedge \ \texttt{grey} \neq \texttt{blue} \\
\longrightarrow \\
0 \leq \texttt{grey} < \texttt{a.length}
\end{array}
$$

These proofs are left as an exercise for the reader.

3.3 Quicksort

Quicksort

Slide 348