Examples
Polish flag, (Dutch flag), quicksort.

- Examples for manipulating arrays (including Partition)
- Quicksort (uses Partition)
- A Challenge Problem

RESTORING FLAGS – FURTHER REQUIREMENTS
- Correct the flag in one pass, i.e. inspect each stripelet once only.
- Each stripe may be cut into a different number of stripelets.
- The only allowed way to rearrange stripelets is to swap two of them.

For simplicity we'll represent a flag as an array of colours:
- use **enum** Col(white, red);
  and Col.red, Col.white and Col[] a in code
- but for easy reading will use Red/White in comments
- Can compare two colours col1 and col2 by

```java
col1.compareTo(col2);
returns -1/0/1 if col1 before/same-as/after col2 in order
```

RESTORING FLAGS – SWAP
A polymorphic swap:

```java
<X> void swap(X[] a, int i, int j) {
    //Pre 0 ≤ i <a.length ∧ 0 ≤ j < a.length
    // Post a[j] = a0[i] ∧ a[i] = a0[j] ∧
    //      ∀k:int(0 ≤ k <a.length ∧ k ≠ i & k ≠ j → a[k] = a0[k])
    //      ∧ a.length=a0.length

    }
```

In swap, a is an object and so the postcondition must refer to two different values: a0 is the value on entry, a is the value on return.

Alternatively, could represent the colours by the integer constants WHITE and RED as in final int WHITE = 0; final int RED = 1; and then use normal integer comparisons. Also need extra Pre:
```
//Pre:∀k:int(0 ≤ k < a.length → a[k]=Red) va[k]=White)
```
RESTORING CORRECT ORDER (POLISH FLAG)

```java
int restore (Col [] a) {
    //Post: a is a rearrangement of a0
    //    ^ 0≤r≤a.length ∧ a[r]=Red
    //    ∧ the stripelets of a are in order (White before Red)
    //    i.e. ∀i,j: int (0≤ i < j < a.length → a[i] ≤ a[j])
    // or  ∀i:int (0≤ i < a.length-1 → a[i] ≤ a[i+1])
    // a[i]≤a[j] tested in code as a[i].compareTo(a[j])<=0
}
```

Exercise: Why is this specification not correct?

(The Rest of this page is deliberately left blank)

0≤r≤a.length ∧ a[r]=Red ∧ stripelets are in order
i.e. ∀i,j: int (0≤ i < j < a.length → a[i] ≤ a[j])

An Aside: Checking Postconditions

**Correct input/output makes Post true**
Doesn’t mean Post correct,
but may give confidence that it is

**Correct input/output makes Post False**
Post is definitely wrong
perhaps “it says too much”

**Wrong input/output makes Post True**
(Assume input meets Pre, so it’s output that’s wrong)
Post is definitely wrong
perhaps “it doesn’t say enough”

AN ASIDE: CHECKING POSTCONDITIONS

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Expected Outputs</th>
<th>Post</th>
<th>PostOK?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct input/output makes Post true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correct input/output makes Post False</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wrong input/output makes Post True</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Some Tips for Writing Correct Postconditions.**

It is very easy to make mistakes in writing postconditions as we showed on slide 5. (It's also easy to get preconditions wrong too, but usually they are corrected by sorting out the postcondition.) Once you've written a postcondition, a good idea is to take some typical input/output pairs, such that the input satisfies the preconditions and the output is what you expect from the method for that input.

For example, for `restore` (using `R` for Red and `W` for White), we could take as input an array \( a = \{ R, W, W, R \} \) and as output \( a = \{ W, W, R, R \} \).

Next, check that the output makes the postcondition true. Unfortunately, some obviously incorrect postconditions could be satisfied by the output. e.g. let the postcondition=\text{True}! So in addition to checking that the postcondition doesn't specify too much, you should also show that it doesn't specify too little (i.e. that too many outputs satisfy it, including incorrect ones). For this, you need to take some incorrect input/output pairs, again with the input satisfying the precondition, and check that the output does not satisfy the postcondition.

For instance, for `restore`, we could consider the pair \( a = \{ R, R, R \} \) as input (and the same as output). It's in this choice of input/output pairs where your ingenuity comes in. You are "testing" the specification. Another pair is \( a = \{ W, W, W \} \) and the same as output. For this one, we would see the first attempt at a postcondition was not true and be forced to amend it.

**Exercise:** Why is (wrong) Post on slide 5 not true for this pair?

---

**CORRECT SPECIFICATION OF RESTORE**

```java
int restore (Col [] a) {
    // Post: a is a rearrangement of a0
    //      \( 0 \leq r \leq a.\text{length} \)
    //      \( \forall i. \text{int} \)
    //      \( 0 \leq i \leq r \rightarrow a[i] = \text{White} \)
    //      \( r < a.\text{length} \rightarrow a[i] = \text{Red} \)
}
```

In other words:
- \( a[r] \) is the first red element in the restored flag if one exists, otherwise \( r = a.\text{length} \) (all elements are white)

---

**FORMALISING "A IS A REARRANGEMENT OF A0"**

We can express more formally that \( a \) is a rearrangement of \( a0 \) by

\[
\text{range}(a) = \text{range}(a0)
\]

where \( \text{range}(a) = "\text{bag}" \) of elements in \( a \).

A bag is like a set, except every element is counted, even duplicates.

\[
\text{e.g. range} \{ \{ W, R, W, W, R \} \} = \{ 3 \times W, 2 \times R \}
\]

As a set = \{ \{ W, R \} \}.

The postcondition of swap was

\[
\begin{align*}
// \text{Post} & \quad \text{a}[j] = a0[i] \land \text{a}[i] = a0[j] \land \\
// & \quad \forall k. \text{int} (0 \leq k < a.\text{length} \land \ k \neq i \land k \neq j \rightarrow a[k] = a0[k]) \\
// & \quad \land a.\text{length} = a0.\text{length}
\end{align*}
\]

We can show Post \( \Rightarrow \) range(a)=range(a0)

(which also implies a.\text{length}=a0.\text{length})
Formalising "a is a rearrangement of a0" continued

Once we have shown that a swap performs a rearrangement (previous slide) all we need to show then is that any sequence of swaps performs a rearrangement. Therefore, if we restrict ourselves to using the swap method, we'll only ever obtain rearrangements of a0.

The second part can be formalised using a proof by induction on the number of swaps (n):

**Base Case** (n=0): a0 is a (trivial) rearrangement of itself, (no swaps)

**Induction Step** (n>0): Assume as Induction Hypothesis that if the number of swaps made to elements of a is < n, the result is a rearrangement of a0.

Suppose that after n swaps applied to a0 we obtain a'n. This can be written as apply n swaps to a0 to get a'n, and then 1 swap to a'n to get a'n.

Assuming swap(a, i, j) is used correctly, then the result a'n is a rearrangement of a'n'. By the IH, after the first n-1 swaps a'n' is a rearrangement of a0.

Therefore, by transitivity, a'n is a rearrangement of a0 (after a total of n swaps).

Hence, by induction, for any n≥0, performing n Swaps on a0 rearranges it.

### PROOF IDEA FOR RESTORE

- Track through the stripelets and put each one in “the right place”.
- How does the flag look when it’s “OK so far but not finished yet”?

```
<table>
<thead>
<tr>
<th>0</th>
<th>pink start</th>
<th>red start</th>
<th>a.length -1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Definitely</td>
<td>Still jumbled (pink)</td>
<td>Definitely</td>
</tr>
</tbody>
</table>
```

- The **invariant** will then say that the flag can be divided like this and you know where the boundaries are, i.e. that the diagram is "correct".

\[
0 \leq \text{pinkStart} \leq \text{redStart} \leq \text{a.length} \land a \text{ is a rearrangement of a0} \land \\
\forall i: \text{int} (0 \leq i < \text{pinkStart} \rightarrow a[i] = \text{White}) \land \\
\forall i: \text{int} (\text{redStart} \leq i < \text{a.length} \rightarrow a[i] = \text{Red})
\]

### HELP IN FINDING THE INVARIANT?

```
• Invariant:
  \[ 0 \leq \text{pinkStart} \leq \text{redStart} \leq \text{a.length} \land a \text{ is a rearrangement of a0} \land \\
  \forall i: \text{int} (0 \leq i < \text{pinkStart} \rightarrow a[i] = \text{White}) \land \\
  \forall i: \text{int} (\text{redStart} \leq i < \text{a.length} \rightarrow a[i] = \text{Red}) \]

• Post:
  a is a rearrangement of a0 \land 0 \leq \text{pinkStart} \leq \text{redStart} \leq \text{a.length} \land \\
  \forall i: \text{int} (0 \leq i < \text{pinkStart} \rightarrow a[i] = \text{White}) \land \forall i: \text{int} (\text{redStart} \leq i < \text{a.length} \rightarrow a[i] = \text{Red})
```

**Exercise:**
What should the initial values of pinkStart and redStart be?

We'll assume a is a rearrangement of a0 as we only perform swaps

- **Variant** = redStart-pinkStart (ie the size of the still jumbled bit)
**REFINED PROOF IDEA FOR RESTORE (1)**

- Track through the stripelets, always inspecting the first pink.
- Update the boundary pointers `pinkStart` and `redStart` as you deal with each stripelet.
- If a stripelet is white, then it’s already in the right place; you can move `pinkStart` on to next stripelet.

![Diagram](image)

**THE CODE FOR RESTORE**

```c
int restore(Col [] a) {
    int pinkStart = 0;    // no whites yet
    int redStart = a.length;    // no reds yet
    // invariant true here (check 1)
    while (pinkStart < redStart)
        // invariant true and pinkStart < redStart (check 2)
        switch (a[pinkStart]) {
            case red: swap(a, pinkStart, redStart-1);
                      redStart--; break;
            case white: pinkStart++; }
    // invariant true and pinkStart ≥ redStart
    return redStart;    //post true (check 3)
}
```

**Exercise**: Show (i) the variant decreases (check 4); (ii) `a[pinkStart]` is a valid array access; (iii) precondition of swap is satisfied (check 5).

**REFINED PROOF IDEA FOR RESTORE (2)**

- If it’s red, swap it with last pink before red; don’t move `pinkStart` as you’ve fetched another pink to inspect. Do move `redStart`.

![Diagram](image)

The variant is the size of the jumbled (pink) area `redStart-pinkStart`.

Progress is made by reducing it. When there are no pinks left, i.e. `pinkStart=redStart`, then the stripelets are in the right order.

**Exercise** – show more formally how, when the variant is 0, that the invariant implies the Postcondition.

**MAKING THE CHECKS**

In what follows we’ll use `ps` for `pinkStart` and `rs` for `redStart` and assume throughout that `a` is a rearrangement of `a0`.

*(Check 1)* Inv. true initially: \( (ps=0, rs=a.length=a0.length) \)

Note that `a0` as no changes have yet been made to `a`.

**Show** \( 0 \leq ps \leq rs \leq a.length \)

\( \forall i:0 \leq i < a.length \rightarrow a[i]=White \land \forall i:a.length \leq i < a.length \rightarrow a[i]=Red \) (ie substitute new values for variables in invariant.)

All easily true (but you must *explain* why - see formal proof in notes)

*(Check 2)* Inv. re-established by loop:

Let the values of `a`, `ps`, `rs` just after entering an arbitrary iteration of the loop be `a1`, `r1`, `p1` and at the end of the loop be `a2`, `r2`, `p2`.

**For Case 1 (the Red case):**

\( p2=p1, r2=r1-1, a1[p1] \) and `a1[r1-1]` (only) have been swapped; `a2[r1-1]=Red` since `a1[p1]=Red`. We know nothing about `a2[p1]`.
MAKING THE CHECKS (CHECK 2 CONTINUED)

We know, just inside the loop:

- **By true while condition** p1<r1 ⇒ p1≤r1-1.
- **By Inv.** (I1) 0≤p1<r1≤a.length ∧ (I2) ∀i:int(0≤i<p1→a[i]=White) ∧ (I3) ∀i:int(r1≤i<a.length→a[i]=Red)

**Case 1:** r2=r1-1, p2=p1, a2[r1-1] = a1[p1] = Red, a2[p1]=a1[r1-1] and other elements of a are unchanged.

**RTS:** 0≤p2≤r2≤a.length ∧ ∀i:int(0≤i<p2→a2[i]=White ∧ ∀i:int(r2≤i<a.length→a2[i]=Red))

Why are the 3 conjuncts true?

---

**MAKING THE CHECKS (CHECK 2 DETAILS)**

- **By true while condition** p1<r1 ⇒ p1≤r1-1 (*)
- **By Inv.** (I1) 0≤p1<r1≤a.length ∧ (I2) ∀i:int(0≤i<p1→a[i]=White ∧ (I3) ∀i:int(r1≤i<a.length→a[i]=Red)

**Case 1:** r2=r1-1, p2=p1, a2[r1-1] = a1[p1] = Red, a2[p1]=a1[r1-1]

**RTS:** 0≤p1<r1-1≤a.length ∧ ∀i:int(0≤i<p1→a2[i]=White ∧ ∀i:int(r1-1+1≤i<a.length→a2[i]=Red)

0≤p1≤r1-1+1≤a.length ∧ ∀i:int(0≤i<p1→a2[i]=White ∧ ∀i:int(r1-1+1≤i<a.length→a2[i]=Red)

Why are the 3 conjuncts true?

Full Details of Checks for restore

**check 3 Post achieved:** When the loop has finished, let the values of ps, rs be ps3 and rs3. Then ps3 ≤ rs3 by the invariant and ps3 ≥ rs3 by the false condition, so ps3=rs3.

Since r=rs3=ps3, the invariant becomes 0≤r≤a.length ∧ ∀i:int(0≤r→a[i]=White) ∧ ∀i:int(r≤i<a.length→a[i]=Red), which is equivalent to the postcondition.

**check1 Inv. established at start of loop:** InitCode sets ps=0 and rs=a.length, which are their values just before the loop. Substitute into the invariant and then RTS

0≤0≤a.length ∧ ∀i:int(0≤i→a[i]=White) ∧ ∀i:int(0≤i<a.length→a[i]=Red).

First conjunct is true (arith. and lengthts≥0), and other two conjuncts are true since their conditions are false for every i.

**check2 invariant re-established by loop:** Let a1, p1 and r1 be the values of a, ps and rs at the start of a loop and a2, p2 and r2 the values at the end:

Given: the while condition is true, so p1<r1 (1). The invariant holds for p1 and r1:
(I1) 0≤p1<r1≤a.length (I2) ∀i:int(0≤i<p1→a[i]=White) (I3) ∀i:int(r1≤i<a.length→a[i]=Red). Then there are 2 cases:

---

Full Details of Checks for restore (continued)

**Case 1:** a1[p1]=Red, r2=r1-1, p2=p1 and a is unchanged except for a2[p1] and a2[r1-1]. These were Red and unknown and are now unknown and Red, respectively. We require to show that 0≤p2≤2≤a.length, ∀i:int(0≤i→a2[i]=White) and ∀i:int(r2≤i<a.length→a2[i]=Red).

Substitute new values for variables r2, p2 in the invariant, then RTS

(14): 0≤p1≤r1-1≤a.length

(15): ∀i:int(0≤i→a2[i]=White)

(16): ∀i:int(r1-1≤i<a.length→a2[i]=Red)

(14) follows from (1) and (11);
(15) is true by (I2) since a is unchanged before element p1, and (16) follows from (14) since a is changed from r1.

**Case 2:** (a1[p1] is White) is similar, but easier and is left as an exercise.
Full Details of Checks for restore (continued)

(check 4 variant decreases at each iteration):

Just after the test of an arbitrary iteration of the loop, \( r_{1}-p_{1} > 0 \).
By the invariant at the end of the loop, \( 0 \leq p_{2} \leq r_{2} \implies 0 \leq r_{2}-p_{2} \).
By the code, \( rs \) decreases or \( ps \) increases, hence \( 0 \leq r_{2}-p_{2} < r_{1}-p_{1} \).
The loop must terminate as \( rs-ps \) cannot continue to decrease and remain >0.

(check 5) array access is \( a[p_{1}] \) – OK by Inv. By (11) \( p_{1} \) and 1-l satisfy the precondition of
swap (ie they are valid indices for \( a \)). (Easy to forget this check.)

APPLICATIONS (1) – NATIONAL FLAGS OF –

<table>
<thead>
<tr>
<th>Armenia</th>
<th>Germany</th>
<th>Mali</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belgium</td>
<td>Guinea</td>
<td>Monaco</td>
</tr>
<tr>
<td>Bulgaria</td>
<td>Hungary</td>
<td>Netherlands</td>
</tr>
<tr>
<td>Chad</td>
<td>Indonesia</td>
<td>Romania</td>
</tr>
<tr>
<td>Colombia</td>
<td>Ireland</td>
<td>Russia</td>
</tr>
<tr>
<td>Estonia</td>
<td>Italy</td>
<td>Sierra Leone</td>
</tr>
<tr>
<td>France</td>
<td>Lithuania</td>
<td>Ukraine</td>
</tr>
<tr>
<td>Gabon</td>
<td>Luxembourg</td>
<td>Yemen</td>
</tr>
</tbody>
</table>

Most of these flags have three colours. In the tutorial you'll adapt the
two colour algorithm developed here to cope with three colours. The
algorithm is due originally to Dijkstra.

Have we got the best algorithm?
Find a case when it behaves badly.

Better idea: alternatively track forwards and backwards from
pinkStart and redStart so that only wrongly placed colours are
swapped.
In fact, we can avoid swapping altogether, as we'll see.

NEW IDEA

First STORE \( a[0] \). We'll replace it at the end
This leaves a HOLE at \( a[0] \) which we'll mark by left
Inspect from upper end of \( a \) until find a White, which we'll mark by right
Move the White to the HOLE at \( a[left] \) (made by storing \( a[0] \)) leaving a
HOLE at \( a[right] \). Increment left.

Before: STORE = \( a[0] \);

\[
\begin{align*}
\text{HOLE} & & \text{White} & & \text{All Red} \\
 a[0] & & & & \\
\end{align*}
\]

After: STORE = \( a[0] \);

\[
\begin{align*}
\text{HOLE} & & \text{All Red} \\
 a[0] & & a[1]=\text{left} & & \text{right} \\
\end{align*}
\]
NEW IDEA (CONTINUED)

Remember STORE = a[0] and we have a HOLE at a[right]
Inspect from left until find a Red,
    which is moved to HOLE at a[right]
leaving a HOLE at left
Decrement right and continue inspecting from right.

At any stage will either have HOLE at left and be inspecting and moving
right down while looking for a White, or will have HOLE at right and be
inspecting and moving left up while looking for a Red.

At end replace STORE into the HOLE and work out start of Reds.

NEW IDEA - DETAILS

Declare directions Up and Down by enum Dir(up,down)
Inspect from right if direction (d) is Down and from left it's Up
Stop when left=right and replace store into the hole
Decide if it belongs with Whites (return left+1) or Reds (return left)

Invariant (5 conjuncts):
(I1\&I2) 0≤left≤right<a.length ∧ ∀i:int(left≤i<right→a[i]=a0[i]) ∧
(I3) ∀i:int(0≤i<left→a[i]=White) ∧
(I4) ∀i:int(right<i<length→a[i]=Red) ∧
(I5) ((d=Down ∧ (bag(a)-a[left]=bag(a0)-a[store])) ∨
   (d=Up ∧ (bag(a)-a[right]=bag(a0)-a[store])))

The last conjunct assumes either the hole is at left (d=Down) or it is at
right (d=Up). We assume store = a0[0] doesn't change.

What is the precondition this time if the method is to work?

NEW IDEA - CODE

int holeRestore(Col [] a) {
  Pre:a.length>0  Post: Same as restore
  int left= 0; Col store=a[0]; Dir d=Dir.down;
  int right= a.length-1  // no Reds or Whites known yet
  while  (left<right) {  // inv. true, test true
    switch (d) {
      case up:
        if  (a[left]==Col.white) left++;
        else  // {a[right]=a[left]; right--; d=Dir.down;} break;
      case down:
        if  (a[right]==Col.red) right--;
        else  // {a[left]=a[right]; left++; d=Dir.up;} break;
      // inv. true and left=right
    }
    a[left]=store; if (store==Col.white) return left+1;
    else return left; }  //post true (check)
For Interest: Some Details of Checks for holeRestore

(Invariant true at start of loop) Required to show the 5 conjuncts. The values of variables are left=0, right=0, length-1, store=[0], d=Down, a=0.  
(1) is true for a0=0, a=0. Length 0 is left justified by arithmetic and preconditions (a is non-empty).
(2) and (4) are true since the conditions are false for every i.
(2) is true as no changes yet to a. To make (15) true note the first disjunct
\[ \text{Down} = \text{Down} = (\text{bag}(a(0)-a(0)[0]) = \text{bag}(a(0)-a(0)[0])) \leftrightarrow \text{true}. \]

(Part true at end) Required to show the finalisation code sets up 0s=a.length' \& \& \forall i: \exists (s[i]=White) \& \& \forall i: (s[i]=\text{true} \leftrightarrow a[i]\text{Red} ) a \text{ is a rearrangement of a0.}
Let left5 and right5 be values of left and right just after exiting the loop. We know all parts of the invariant and also that the while test is false (T).

(1): 0sleft5=sright5=1a.length
(2): \forall i: \exists (s[i]=0)
(3): \forall i: \exists (s[i]=White)
(4): \forall i: (s[i]=\text{true} \leftrightarrow a[i]\text{Red}).
In (5), we’ll write S for bag(a0)-bag(store).
(5): (d1=Down \& \& \forall i: \exists (a[i]=White) \& \& \forall i: (s[i]=\text{true} \leftrightarrow a[i]\text{Red})).
(T) \leftrightarrow \text{left5=right5} \& \& (1) \leftrightarrow \text{left5=right5}.
Therefore left5=right5.

In what follows, remember that a bag counts every element including duplicates.

February 10, 2011

flags and quicksort

Apply (vE) to (15), both disjunctions imply
\[ \text{bag}(a)-\text{left}[5] = \text{bag}(a)-\text{store} \]
\[ \leftrightarrow \text{bag}(a)-\text{store} = \text{bag}(a0)-\text{store} \text{ (since a[5]=store)} \]
\[ \leftrightarrow \text{bag}(a)-\text{bag}(a0) \leftrightarrow \text{a is a rearrangement of a0.} \]
There are then two cases:

Case 1: \text{r=left[5]+1, a[left]=White}.
\[ \forall i: \exists (s[i]=0) \& \& \forall i: \exists (s[i]=\text{true} \leftrightarrow a[i]\text{Red}) \]
\[ \leftrightarrow \text{bag}(a)-\text{left}[5] = \text{bag}(a)-\text{store} \]
\[ \leftrightarrow \text{true} \leftrightarrow \text{r =left[5]+1, a[right]=right5} \& \& \text{left5=right5} \& \& \text{case.} \]

Case 2: \text{r=left[5]+1, a[right]=red}.
\[ \forall i: \exists (s[i]=0) \& \& \forall i: \exists (s[i]=\text{true} \leftrightarrow a[i]\text{Red}) \]
\[ \leftrightarrow \text{true} \leftrightarrow \text{r =left[5]+1, a[right]=right5} \& \& \text{left5=right5} \& \& \text{case.} \]

February 10, 2011

flags and quicksort
Sub-case 1b: a[1][left1]=Red, a[2][right1]=a[1][left1]=Red, right2=right1-1, left2=left1, d2=Down, elements of a other than a[1][right1] unchanged.

- 0<left2<right2<ca.length <=0<left1<right1-1<ca.length => True by (11) and (*)
- \( \forall \)int(0<left2<right2<right2<left1)[White] \( \Rightarrow \) \( \forall \)int(0<left1<right1-1<ca.length)[White] \( \Rightarrow \) True by (13).
- \( \forall \)int(right2<ca.length<2[i]=Red) \( \Rightarrow \) \( \forall \)int(right1-1<ca.length<2[i]=Red) \( \Rightarrow \) True by (14) and Case. (I10): The obvious disjunct to make true here is d2=Down \( \land \) bag(a[2-a][left2])=S, d2=Down is true by the case. bag(a-a[2-a][left2])=S \( \Rightarrow \) bag(a-a[1][left1])=S
  \( \Rightarrow \) bag(a-a[1][right1])=S.

By the case LH disjunct of (I5) is false since d1\( \neq \)U[p] \( \Rightarrow \) bag(a-a[1][right1])=S is True. (I17): \( \forall \)[int(left2<right2<right2<left1)[left1]] \( \Rightarrow \) True by (12).

Variant decreases) The variant decreases each time through the loop since either left increases or right decreases. (I11) \( \Rightarrow \) right-left=0. Hence the loop cannot go on forever.

Valid array access) The array accesses are a[i] ok since a.length>0, a[left], a[right] within the loop - ok by (11), a[left] after the loop - ok by (11).

DUTCH FLAG – REFINED PROOF IDEA

- Track through the stripelets, always inspecting the first grey one. Keep pointers to the boundaries between the four areas, and update them as you deal with each stripelet.
- If a stripelet is white, then it’s in the right place and move on.
- If it’s red, then swap it with the first white and move on.
- If it’s blue, then swap with the last grey. Don’t move on, because you’ve now fetched another grey to inspect.
- When no greys are left (greyStart=blueStart), the stripelets are in the right order. Return whiteStart and blueStart.

DUTCH FLAG PROBLEM IDEA

Now suppose there are 3 colours: Red White and Blue and we want to arrange the stripelets so they are in that order.

```java
enum Col(red, white, blue)
```

New Post: – restore will return wStart and bStart bundled as a pair.

- a is a rearrangement of a0 \( \land \) 0\( \leq \)wStart\( \leq \)bStart\( \leq \)a.length
- r=(wStart, bStart) \( \land \) \( \forall \)int(0\( \leq \)wStart \( \rightarrow \) a[i]=Red
- \( \forall \)int(bStart\( \leq \)a.length \( \rightarrow \) a[i]=Blue)
- \( \forall \)int(wStart\( \leq \)bStart \( \rightarrow \) a[i]=White)

<table>
<thead>
<tr>
<th>whiteStart</th>
<th>definitely</th>
<th>greyStart</th>
<th>definitely</th>
<th>blueStart</th>
<th>definitely</th>
<th>a.length</th>
<th>still jumbled (grey)</th>
<th>definitely</th>
</tr>
</thead>
</table>

SPECIFICATION OF DUTCH FLAG

For the Polish flag we returned redStart and had the following header

```java
int restore (Col [] a) {
```

Now we have to return wStart and bStart (the Red region starts at 0)
So we’ll define a class DutchReturnInfo to hold wStart and bStart

```java
class DutchReturnInfo{
    public final int wStart, bStart;
    public DutchReturnInfo(int i, int j){wStart=i; bStart=j;}}
```

DutchReturnInfo dutchRestore (Col [] a) { //Pre: none
  //Post: \( \forall \)int (0 \( \leq \) i \( \leq \) r.wStart \( \rightarrow \) a[i]=Red) \( \land \)
  // \( \forall \)int (r.wStart \( \leq \) i \( \leq \) r.bStart \( \rightarrow \) a[i]=White) \( \land \)
  // \( \forall \)int (r.bStart \( \leq \) i \( \leq \) a.length \( \rightarrow \) a[i]=Blue)
  // \( \land \)a is a rearrangement of a0\( \land \)0\( \leq \)r.wStart\( \leq \)r.bStart\( \leq \)a.length}
“The diagram is correct” –

\[
0 \leq \text{whiteStart} \leq \text{greyStart} \leq \text{blueStart} \leq \text{a.length} \land \\
\forall i \in \text{int} \; (0 \leq i < \text{whiteStart} \rightarrow a[i] = \text{Red}) \land \\
\forall i \in \text{int} \; (\text{whiteStart} \leq i < \text{greyStart} \rightarrow a[i] = \text{White}) \land \\
\forall i \in \text{int} \; (\text{blueStart} \leq i < \text{a.length} \rightarrow a[i] = \text{Blue}) \land \\
\text{a is a rearrangement of} \; a0
\]

**Loop variant**: number of greys left

= blueStart – greyStart.

**Applications (2) – Sorting**

Donald Knuth (“Sorting and Searching”):

“Computer manufacturers estimate that over 25% of the running time on their computers is currently being spent on sorting, when all their customers are taken into account. There are many installations in which sorting uses more than half of the computing time. From these statistics we may conclude either that

(i) there are many important applications of sorting, or
(ii) many people sort when they shouldn’t, or
(iii) inefficient sorting algorithms are in common use.

The real truth probably involves some of all three. In any event we can see sorting is worthy of serious study as a practical matter.”

**Good Principle** – if a program’s used a lot, it’s worth making it fast. We’ll look at a sorting algorithm (Quicksort) that uses restore.

**Code for Dutch Flag**

```java
DutchReturnInfo dutchRestore (Col [] a) {
    int whiteStart=0, blueStart=a.length; // no whites or blues yet
    int greyStart = 0; // nothing checked yet

    while (greyStart < blueStart) // invariant true here (check 1)
        // invariant true and greyStart < blueStart
        (check 2)
    switch (a[greyStart]) {
        case red: swap with first white; move on one element
        case white: in right place; just move on one element
        case blue: swap with last grey; don't move on
    }
    // invariant true and greyStart ≥ blueStart,
    return new DutchReturnInfo(whiteStart,blueStart);
} //post true (check 3)
```

**Exercise**: complete the code and make the checks (see Problems).

**Sorting (Continued)**

Given an array of integers, sort its elements into ascending order.

- We choose integers for simplicity.
- But the order could be something you’ve implemented yourself, like alphabetical order on strings. You could even “implement” ≤ as ≥, to get descending order;
- We’ll sort not only entire arrays, but also regions within them.
- We describe a region by two parameters start and rest – the region goes from start up to, but not including, rest.
- We’ve sorted the stripelets purely by colour. Within the colours, there may be refined ways of sorting, eg by width of stripelet.

**Restore** takes no account of these: it is a crude sort by colour alone.

**Exercise** Think of examples where you might first do such a crude sort, then follow it by a more refined sort.
POLISH NATIONAL FLAG IS CRUDE SORTING

- If we want a more refined ordering, restore has helped; we can now sort the two colour regions separately—it's easier to sort small regions.
- More careful analysis—as regions get smaller, the complexity of sorting them goes down faster than the number of regions goes up.

**Idea:** Sort an array of integers by a succession of crude sorts.

- First pick a “key” integer \( k \) and do a crude sort using the Polish National Flag method:
  
  \[
  \text{white} \text{ means} \quad \langle k \quad \text{red} \text{ means} \quad \geq k
  \]

- Now the elements are in the right regions, but they still need sorting amongst themselves; do this by the same method (called recursively).
- How do we ensure progress is made? A bad \( k \) might give no red elements. Then the “simpler” problem of sorting the white elements amongst themselves is no simpler. The recursion might never stop.

SPECIFICATION OF PARTITION

**restore** is called **partition** for this particular use (with key “\( k \)”).

```c
int partition(int [] a, int start, int rest, int k) {
    // Pre: 0 \leq start \leq rest \leq a.length
    // Post: does a Polish flag (white/red) sort on the region of a
    // from start up to, but not including, rest;
    // "white" means \( < k \)
    // "red" means \( \geq k \)
    // returns r = redStart; start \leq r \leq rest
}
```

This is informal, but our work on the Polish flag tells us how to formalise and implement it.

DRAW A PICTURE!

- Solution: \( k = \) first element, sort the rest. **restore** tells you where the red elements start, so swap the first element \( k \) up to just before it. Don’t include it in either of the recursively sorted regions.

```
\begin{tikzpicture}
  \node (k) at (0,0) {\text{to partition into 2 regions}};
  \node (white) at (1,0) {\text{white \( \langle k \)}};
  \node (red) at (1,1) {\text{red \( \geq k \)}};
  \node (swap k) at (2,0) {\text{swap \( k \)}};
  \node (redStart) at (3,0) {\text{redStart}};
  \node (white) at (4,0) {\text{white \( < k \)}};
  \node (red) at (4,1) {\text{red \( \geq k \)}};
  \draw [->] (k) -- (white);
  \draw [->] (white) -- (red);
  \draw [->] (red) -- (swap k);
  \draw [->] (swap k) -- (redStart);
  \draw [->] (redStart) -- (white);
  \draw [->] (white) -- (red);
\end{tikzpicture}
```

We again sort the white and red regions by the same method. Each is definitely smaller than the original region, because it doesn’t include the \( k \) element.

GENERAL SPECIFICATION OF SORT-REGION

```c
void sortRegion(int [] a, int start, int rest) {
    // Pre: 0 \leq start \leq rest \leq a.length
    // Post: a is a rearrangement of \( a[0 \wedge \) haven’t changed anything except between start and rest,
    // \( i.e. \forall i: \text{int}(0 \leq i < \text{start} \text{ or rest} \leq a.length \rightarrow a[i] = a[0]) \)
    // \( \wedge \) within the region, a is sorted,
    // \( i.e. \forall i,j: \text{int}(\text{start} \leq i < j < \text{rest} \rightarrow a[i] \leq a[j]) \)
}
```
QUICKSORT CODE

```c
void quickSort(int [ ] a, int start, int rest) {
    // Pre:  0 ≤ start ≤ rest ≤ a.length
    // Post: Does a sort as specified earlier (as sortRegion)
    int redStart;
    //for proof, do induction on rest-start
    if (start < rest-1) {
        // else, region has ≤1 element, nothing to do
        redStart=partition(a, start+1, rest, a[start]);
        //leave out a[start]=k and partition about k
        swap(a, start, redStart–1);    //a[redStart-1]=k
        quickSort(a, start, redStart–1);    //start upto redStart-1
        quickSort(a, redStart, rest);    //redStart upto rest
    }
}
```

CORRECTNESS OF QUICKSORT (1)

- We still have a variant, rest–start (size of region), but proof is different as we're using recursion instead of loops
- Show for all pairs (start, rest) such that 0 ≤ start ≤ rest ≤ a.length, that quickSort(a, start, rest) terminates satisfying the Postcondition:
  
  \[
  \forall i, j: \int (start ≤ i ≤ j < rest → a[i] ≤ a[j]) \quad \text{and for } 0 ≤ i ≤ start \text{ and } rest ≤ a.length \ a \text{ is unchanged.}
  \]

We'll use well-founded induction on the set

\[
\text{goodPairs}(a) = \{ (s, r) \mid s: \text{Nat}, r: \text{Nat} \land 0 ≤ s ≤ r ≤ a.length \}
\]

using the well-founded ordering on goodPairs(a) (gp(a) for short)

\[
(s_1, r_1) << (s_2, r_2) \iff r_1 - s_1 < r_2 - s_2
\]

To prove the property for some given a, start and rest, where (start, rest) ∈ gp(a), assume as IH:

for all (rest', start') ∈ gp(a), if (rest', start') << (rest, start) then quickSort(a, start', rest') terminates satisfying post

CORRECTNESS OF QUICKSORT (2)

Case 1: rest-start=1        \iff rest=1=start
Case 2: rest-start=0        \iff rest=start

- In both cases quickSort obviously stops (with no change at all to a) and the postcondition is true. Exercise check it!
- Important part to check is \( \forall i, j: \int (start ≤ i ≤ j < rest → a[i] ≤ a[j]) \) especially for Case 1 (you need to check it for \( i = j = \text{start} = \text{rest} - 1 \)).

Case 3: rest-start=n, where n>1

- The first two statements were
  
  redStart=partition(a, start+1, rest, a[start]);
  swap(a, start, redStart-1);

- The postcondition of partition gives \( \text{start+1} ≤ \text{redStart} ≤ \text{rest} \) and, after the swap we know \( a[\text{start}] \) upto \( a[\text{redStart}-2] \) are \( a[\text{redStart}] \) and also \( a[\text{redStart}-1] \) upto \( a[\text{rest}-1] \) are \( ≥ a[\text{redStart}] \) (all limits inclusive)

CORRECTNESS OF QUICKSORT (A PICTURE)

```
rs=redStart; s=start; r=rest; k=a[start]
```

after partition (a1)   (a1[s] = a0[s] = k )

```
\[
\begin{array}{c}
\text{k} \\
\text{<k}
\end{array}
\]
```

```
\[
\begin{array}{c}
\text{s} \\
\text{<k}
\end{array}
\]
```

after swap (a2)   (a2[s] = a1[rs-1] < k a2[rs-1] = k )

```
\[
\begin{array}{c}
\text{s} \\
\text{<k and sorted}
\end{array}
\]
```

```
\[
\begin{array}{c}
\text{rs} \\
\text{<k and sorted}
\end{array}
\]
```

after the sorts (a3)

5 cases to check for sortedness:

\[
\begin{align*}
& i \text{ and } j < rs-1; \\
& i \text{ and } j ≥ rs; \\
& i < rs-1 \text{ and } j ≥ rs; \\
& i < rs-1 \text{ and } j = rs; \\
& i = rs-1 \text{ and } j ≥ rs;
\end{align*}
\]

February 10, 2011
CORRECTNESS OF QUICKSORT (3)

- For the recursive call quickSort(a, start, redStart-1) must check that (IH) applies: ie (start, redStart-1) ∈ gp(a) and it is < (start, rest)) From Pre of quickSort and Post of partition:
  - 0<start<redStart-1<rest<length => (start, redStart-1) ∈ gp(a)
  - Also redStart-1<start<redStart since redStart-1<rest.
  - Similarly, for the subcall quickSort(a, redStart, rest) (see 53).
  - Therefore (IH) applies to the two regions and so both calls terminate having sorted their respective regions.
  - The first sorts the white region and the second sorts the red. After the calls to partition and swap all elements in the white region (up to a[redStart-1]) are < a[redStart-1]≤ every element in the red region.
  - After the sorts (not involving a[redStart-1]) the array will be sorted. Hence Postcondition is achieved. (See picture on 52 again.)

Details of Correctness of quickSort

The missing parts of the proof on the slides is here. The proposition to prove is that $\forall (s, t)$: if $0<s<\text{length}$ then quickSort(a, s, t) stops & Post(a, s, t) is true, where Post(a, s, t) is $\forall i (0<i<s \lor rs<s<\text{length} \rightarrow a[i]\neq a[t])$ \& $\forall i (s<i<rs \rightarrow a[i]=a[t])$.

We use well-founded induction on the set goodPair(a) = {(s, t): sNat, tNat, 0<s<\text{length}} with the ordering (s, t) < (rs, r) iff $\forall(i, r)(s, i) < \forall(r, t)$, where $\forall(s, t) = r-s$.

Note that the restriction on $s$ and $t$ gives the pre-condition for initial call to quickSort.

Suppose for some arbitrary values of $s$ and $t$ that $0<s<\text{length}$, then $(s, t) \in \text{goodPair}(a)$ and $\forall(s, t) = n$.

Assume as induction hypothesis (IH) that, for all $(s', t') \in \text{goodPair}(a)$ such that $\forall(s', t')<n$, quickSort(a, s', t') stops and Post(a, s', t') is true.

Case 1: n=1. $\forall(s, t) = 1$ means $r=s+1$.

According to the code quickSort does nothing in this case and so the parts of a outside $s$ to $t$ are unchanged. The part between $s$ and $t$ has just 1 element, $a[s]$, and of course it is sorted.

Case 2: n=0. $\forall(s, t) = 0$, i.e. $rs=s$, is even simpler.

SUMMARY

- Thinking about the invariant helps to guide the code.
- If a problem looks hard, try to formulate a simpler version and solve that first.
- A good algorithm (eg restore or partition) can often be used in different ways to solve other problems.
- A larger example, called median, is set as a challenge. See next slide.
CHALLENGE QUESTION

- Write and test a method (called median) that uses partition to do the following:
  Given an array of integers and integer n, find the element at index n of the sequence that would result if its elements were put into ascending order. (Since arrays are indexed from 0, median will actually find the n+1th element in the sorted array!)

- Example: a is \{50, 15, 8, 21, 20, 3, 45, 19, 30\} and n = 5.
  sorted(a)= \{3, 8, 15, 19, 20, 21, 30, 45, 50\} and sorted(a)[5]=21.
  if n=6, then sorted(a)[6]=21 also.
  Some hints appear on the next slide.
  It's not acceptable to sort a and pick the element at index n.
  Of course it will have pre/post conditions and a justification of correctness

PROGRAM — BASIC IDEA

The specification of median is given as:
```c
void median(int [] a, int n) {
// Pre: 0 ≤ n < a.length
// Post: a is a rearrangement of a0
//   \& a is "partially sorted" –
//   \forall: int (0≤j<n → a[j]=a[n] \& n≤j<ca.length → a[n]=a[j])
// i.e. a[n] would occur at position n if a0 were sorted 
```

Solution 1. Sort the array a and choose the element at index n. This is more than is needed and is not acceptable as an answer!

Solution 2: Notice: the element at index n in the sorted array has exactly n elements ≤ to it, with the rest of the elements ≥ to it.
  e.g. for the above examples, when n=5, there were 5 elements ≤21 and the rest were ≥21.
  When n=6, there were 6 elements ≤21 and the rest ≥21.
  Outline method: use partition to split a into two parts about some x in a, such that all elements in the first part are <x, all elements of the second part are ≥x and a[x]=x.
  You might be lucky – result of partition = n. Then a[n] = x is the required value. Most likely you’ll not be so lucky. If the first part is longer than n, then a[n]<x and n lies within the first part, otherwise a[n]≥x and n lies within the second part. Choose the part in which a[n] must lie and repeat the algorithm on this part. Continue until ..., well, you must work it out!