– Reasoning about Programs –

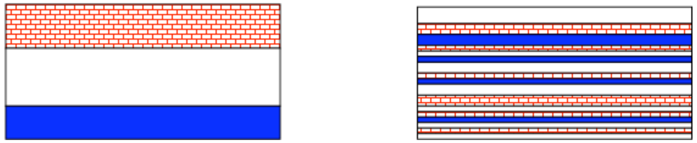
Examples
Polish flag,
(Dutch flag),
quicksort.

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

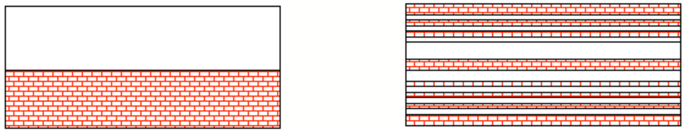
RESTORING FLAGS

In honour of several Dutch computer scientists and Polish logicians.

Dutch Flag (Red, White and Blue):



Polish Flag (White and Red):



Problem: given a scrambled computer representation of the flag with the stripes being cut up horizontally and rearranged, restore the flag.

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

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

RESTORING FLAGS – SWAP

A polymorphic swap:

```
<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∨a[k]=White)

RESTORING CORRECT ORDER (POLISH FLAG)

```
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)

$0 \le r \le a.length \wedge a[r]=Red \wedge$ stripelets are in order

i.e. $\forall i,j$: int $(0 \le i \le j < a.length \rightarrow a[i] \le a[j])$

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

Inputs Expected Outputs Post PostOK?

Correct input/output makes Post true

Correct input/output makes Post False

Wrong input/output makes Post True

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 precondition 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. eg, let the postcondition=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

```
int restore (Col [] a) {
//Post:  a is a rearrangement of a0
//          ∧ 0≤r≤a.length
//          ∧  ∀i:int
//              (0≤i<r → a[i]=White  ∧
//              r≤i<a.length → a[i]=Red)
}
```

In other words:

a[**r**] is the first red element in the restored flag if one exists, otherwise **r**=a.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

range(a)=range(a0)

where range(a) = "bag" of elements in a.

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

e.g. range({W, R, W, R, W})=(3 x W, 2 x R). As a set ={W, R}.

The postcondition of swap was

```
// 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
```

We can show Post ==> range(a)=range(a0)

(which also implies a.length=a0.length)

Postcondition of swap implies range(a)=range(a0)

```
// Post  a[j] = a0[i] ∧ a[i] = a0[j] ∧
//         ∀k:int(0 ≤ k <a.length ∧  k ≠ i & k ≠ j → a[k]= a0[k])
//          ∧ a is a rearrangement of a0
```

range(a)

$= \text{bag}(a[i], a[j]) \cup \text{bag}(a[k]), (0 \le k < a.\text{length}, k \ne i, k \ne j)$

$= \text{bag}(a0[j], a0[i]) \cup \text{bag}(a0[k]), (0 \le k < a0.\text{length}, k \ne i, k \ne j)$

(by postcondition of `swap`)

$= \text{range}(a0)$

Exercise:

(i) Where are the conjuncts of the Post of `swap` used in the proof?

(ii) The proof shows that every swap performs a rearrangement of `a`. Next show that any sequence of swaps performs a rearrangement and hence if `restore` only perform swaps then range(a)=range(a0).

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.

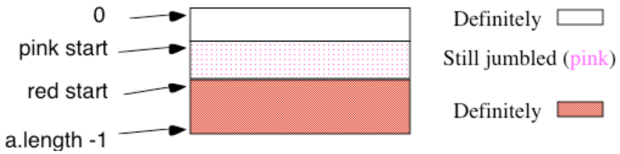
The second part can be formalised using a proof by induction on the number of swaps (n):
Base Case (n=0): a0 is a (trivial) rearrangement of itself; (no swaps)

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 an. This can be written as apply n swaps to a0 to get an', and then 1 swap to an' to get an.
Assuming swap(a, i, j) is used correctly, then the result an is a rearrangement of an'. By the IH, after the first n-1 swaps an' is a rearrangement of a0.
Therefore, by transitivity, an 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

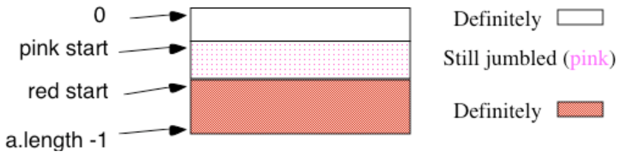
- Track through the stripelets and put each one in “the right place”.
- How does the flag look when it's “OK so far but not finished yet”?



- 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".

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

HELP IN FINDING THE INVARIANT?



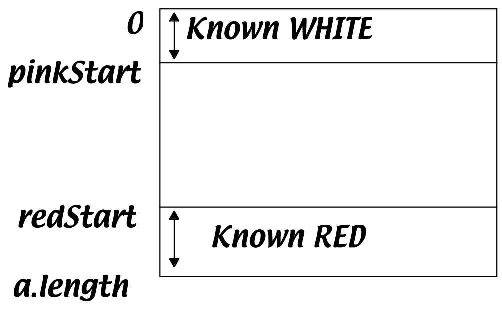
- **Invariant:**

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

Compare with Post:

- **Post:**

a is a rearrangement of a0 $\wedge$ $0 \le \mathbf{r} \le \text{a.length}$ $\wedge$
$\forall i{:}\textbf{int}\ (0 \le i < \mathbf{r} \rightarrow a[i]=\text{White}) \wedge \forall i{:}\ \textbf{int}\ (\mathbf{r} \le i < \text{a.length} \rightarrow a[i]=\text{Red})$



Initially

pinkStart=

redStart=

Exercise:

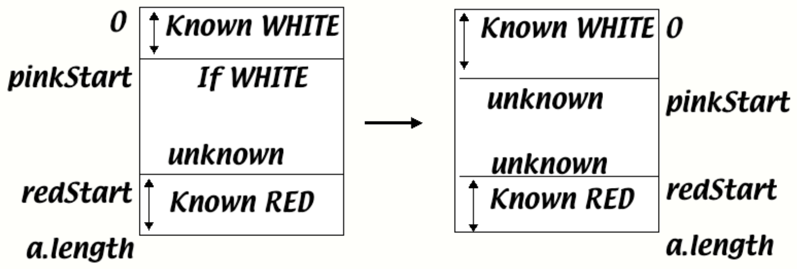
What should the initial values of pinkStart and redStart be?

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)

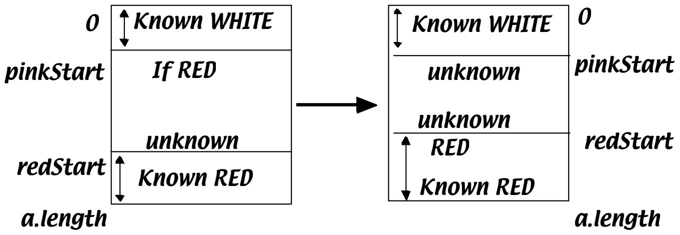
REFINED PROOF IDEA FOR RESTORE (1)

- 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.



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.



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.

THE CODE FOR RESTORE

```
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*).

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 a=a0 as no changes have yet been made to a.

Show $0 \le 0 \le a.length \le a.length \wedge$
$\forall i{:}\mathbf{int}(0 \le i < 0 \rightarrow a0[i]=White) \wedge \forall i{:}\mathbf{int}(a.length \le i < a.length \rightarrow a0[i]=Red)$
(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, rs, ps 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 $p1<r1 \rightarrow p1 \le r1-1$.

By Inv. (I1) $0 \le p1 \le r1 \le a.length \wedge$

(I2) $\forall i{:}\mathbf{int}(0 \le i<p1 \rightarrow a1[i]=White \wedge$

(I3) $\forall i{:}\mathbf{int}(r1 \le i<a.length \rightarrow a1[i]=Red)$

Case1: $r2=r1-1$, $p2=p1$, $a2[r1-1] = a1[p1] = Red$, $a2[p1]=a1[r1-1]$ and other elements of a are unchanged.

RTS: $0 \le p2 \le r2 \le a.length \wedge$
$\forall i{:}\mathbf{int}(0 \le i<p2 \rightarrow a2[i]=White \wedge \forall i{:}\mathbf{int}(r2 \le i<a.length \rightarrow a2[i]=Red)$
<==> (after substitution by values of p2 etc.)

$0 \le p1 \le r1-1 \le a.length \wedge$
$\forall i{:}\mathbf{int}(0 \le i<p1 \rightarrow a2[i]=White \wedge \forall i{:}\mathbf{int}(r1-1 \le i<a.length \rightarrow a2[i]=Red)$

Why are the 3 conjuncts true?

MAKING THE CHECKS (CHECK 2 DETAILS)

By true while condition $p1<r1 \rightarrow p1 \le r1-1$ (*)

By Inv. (I1) $0 \le p1 \le r1 \le a.length \wedge$

(I2) $\forall i{:}\mathbf{int}(0 \le i<p1 \rightarrow a1[i]=White \wedge$

(I3) $\forall i{:}\mathbf{int}(r1 \le i<a.length \rightarrow a1[i]=Red)$

Case1: $r2=r1-1$, $p2=p1$, $a2[r1-1] = a1[p1] = Red$, $a2[p1]=a1[r1-1]$

RTS: $0 \le p1 \le r1-1 \le a.length \wedge$
$\forall i{:}int(0 \le i<p1 \rightarrow a2[i]=White \wedge \forall i{:}int(r1-1 \le i<a.length \rightarrow a2[i]=Red)$

$0 \le p1$(I1); $p1 \le r1-1$ (*); $r1-1 \le a.length$ (I1);
$\forall i{:}\mathbf{int}(0 \le i<p1 \rightarrow a2[i]=White) \Longleftrightarrow \forall i{:}\mathbf{int}(0 \le i<p1 \rightarrow a1[i]=White$
(elements of a before p1 are unchanged). True by (I2).
$\forall i{:}\mathbf{int}(r1-1 \le i<a.length \rightarrow a2[i]=Red) \Longleftrightarrow$
$a2[r1-1]=Red \wedge \forall i{:}\mathbf{int}(r1 \le i<a.length \rightarrow a2[i]=Red)$ **//Very useful step**
True by Case, and (I3) (a is unchanged from r1 onwards)

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 $ps3 \le rs3$ by the invariant and $ps3 \ge rs3$ by the false while condition, so $ps3=rs3$.

Since $\mathbf{r}=rs3=ps3$, the invariant becomes $0 \le \mathbf{r} \le a.length \wedge \forall i{:}int(0 \le i<\mathbf{r} \rightarrow a[i]=White) \wedge$
$\forall i{:}int(\mathbf{r} \le i<a.length \rightarrow a[i]=Red)$, 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
$0 \le 0 \le a.length \le a.length \wedge$
$\forall i{:}int(0 \le i<0 \rightarrow a[i]=White) \wedge$
$\forall i{:}int(a.length \le i<a.length \rightarrow a[i]=Red)$.

First conjunct is true (arith. and lengths≥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): $0 \le p1 \le r1 \le a.length$
(I2): $\forall i{:}int(0 \le i<p1 \rightarrow a1[i]=White)$
(I3): $\forall i{:}int(r1 \le i<a.length \rightarrow a1[i]=Red)$. 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 $0 \le p2 \le r2 \le a.length$, $\forall i{:}int(0 \le i<p2 \rightarrow a2[i]=White)$ and
$\forall i{:}int(r2 \le i<a.length \rightarrow a2[i]=Red)$.
Substitute new values for variables r2, p2 in the invariant), then RTS

(I4): $0 \le p1 \le r1-1 \le a.length$
(I5): $\forall i{:}int(0 \le i<p1 \rightarrow a2[i]=White$
(I6): $\forall i{:}int(r1-1 \le i<a.length \rightarrow a2[i]=Red)$

(I4) follows from (1) and (I1);

(I5) is true by (I2) since a is unchanged before element p1, and

(I6) <==> $a2[r1-1]=Red \wedge \forall i{:}int(r1 \le i<a.length \rightarrow a2[i]=Red)$; follows since a[r1-1] is Red, and by (I3), since a is unchanged 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, $rs1-ps1 > 0$.
By the invariant at the end of the loop, $0 \leq ps2 \leq rs2 \implies 0 \leq rs2-ps2$.
By the code, rs decreases or ps increases, hence $0 \leq rs2-ps2 < rs1-ps1$.
The loop must terminate as rs-ps cannot continue to decrease and remain >0.

(check 5) array access is a[p1] – OK by Inv. By (I1) p1 and r1-1 satisfy the precondition of swap (ie they are valid indices for a). (Easy to forget this check.)

APPLICATIONS (1) – NATIONAL FLAGS OF –

Armenia
Belgium
Bulgaria
Chad
Colombia
Estonia
France
Gabon
Germany
Guinea
Hungary
Indonesia
Ireland
Italy
Lithuania
Luxembourg
Mali
Monaco
Netherlands
Romania
Russia
Sierra Leone
Ukraine
Yemen

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

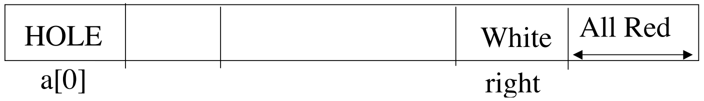
First STORE a[0]. We'll replace it at the end

This leaves a HOLE at a[0] which we'll mark by left

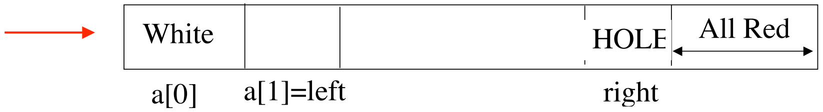
Inspect from upper end of a until find a White, which we'll mark by right

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];



After: STORE = a[0];



NEW IDEA (CONTINUED)

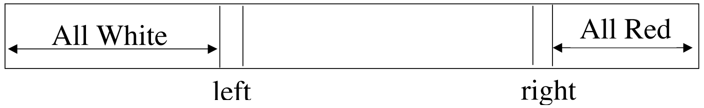
Remember STORE = a[0] and we have a HOLE at a[right]

Inspect from left until find a Red,

    which is moved to HOLE at a[right]

    leaving a HOLE at left

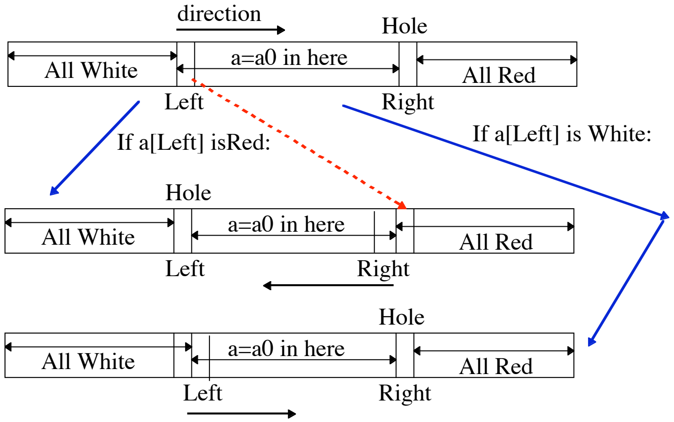
Decrement right and continue inspecting from right.



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 - PICTURE



We assume store=a0[0] doesn't change.
We are not using swaps here, so will have to reason carefully that the result a and a0 are rearrangements of one another.

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 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) $0 \le left \le right < a.length \wedge \forall i{:}int(left \le i \le right \rightarrow a[i]=a0[i]) \wedge$

(I3) $\forall i{:}int(0 \le i < left \rightarrow a[i]=White) \wedge$

(I4) $\forall i{:}int(right < i < a.length \rightarrow a[i]=Red) \wedge$

(I5) $((d=Down \wedge (bag(a)-a[left]=bag(a0)-a[store])) \vee$
$(d=Up \wedge (bag(a)-a[right]=bag(a0-a[store])))$

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=a.length-1, store=a0[0], d=Down, a=a0.

(I1)$\Longleftrightarrow 0\le 0\le a.length-1<a.length$. True by arithmetic and precondition (a is non-empty).

(I3) and (I4) are true since the conditions are false for every i.

(I2) is true as no changes yet to a. To make (I5) true note the first disjunct

$\Longleftrightarrow Down=Down \wedge (bag(a0)-a0[0]=bag(a0)-a0[0]) \Longleftrightarrow True$.

(*Post true at end*) Required to show the finalisation code sets up $0\le \mathbf{r}\le a.length \wedge \forall i{:}int(0\le i<\mathbf{r}\rightarrow a[i]=White) \wedge \forall i{:}int(\mathbf{r}\le i<a.length\rightarrow a[i]=Red) \wedge$ a 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):

(I1): $0\le left1\le right1<a.length$

(I2): $\forall i{:}int(left5\le i\le right5\rightarrow a\ [i]=a0[i])$

(I3):$\forall i{:}int(0\le i<left5\rightarrow a[i]=White)$

(I4): $\forall i{:}int(right5<i<a.length\rightarrow a[i]=Red)$.

In (I5), we'll write S for bag(a0)-bag(store).

(I5): $(d1=Down \wedge (bag(a)-a[left1]=S))\vee(d1=Up \wedge (bag(a)-a1[right]=S))$.

(T) $\Longrightarrow left5\ge right5$ and (I1)$\Longrightarrow left5\le right5$. Therefore left5=right5.

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

Apply ($\vee$E) to (I5); both disjuncts imply

$(bag(a)-a[left5] = bag(a0)-store)$

$\Longleftrightarrow(bag(a)-store=bag(a0)\ store)$ (since a[left5]=store)

$\Longleftrightarrow (bag(a)=bag(a0)) \Longleftrightarrow$a is a rearrangement of a0.

There are then two cases:

Case 1: $\mathbf{r}=left5+1,\ a[left5]=store=White$.

$0\le \mathbf{r}\le a.length \Longleftrightarrow 0\le left5+1\le a.length \Longleftrightarrow$ true (follows from (I1))

$\forall i{:}int(0\le i<\mathbf{r}\rightarrow a[i]=White)\wedge \forall i{:}int(\mathbf{r}\le i<a.length\rightarrow a[i]=Red)$

$\Longleftrightarrow\forall i{:}int(0\le i<left5+1\rightarrow a[i]=White) \wedge\forall i{:}int(left5+1\le i<a.length\rightarrow a[i]\ge x)$

$\Longleftrightarrow\forall i{:}int(0\le i<left5\rightarrow a[i]=White) \wedge a[left5]=White\wedge\forall i{:}int(right5<i<a.length\rightarrow a[i]=Red)$

$\Longleftrightarrow$True by (I3),(I4), left5=right5 and case.

Case 2: $\mathbf{r}=left5,\ a[left5]=store=Red$.

$0\le \mathbf{r}\le a.length \Longleftrightarrow 0\le left5\le a.length \Longleftrightarrow$ true (follows from (I1))

$\forall i{:}int(0\le i<\mathbf{r}\rightarrow a[i]=White)\wedge \forall i{:}int(\mathbf{r}\le i<a.length\rightarrow a[i]=Red)$

$\Longleftrightarrow\forall i{:}int(0\le i<left5\rightarrow a[i]=White) \wedge \forall i{:}int(left5\le i<a.length\rightarrow a[i]=Red)$

$\Longleftrightarrow\forall i{:}int(0\le i<left5 \rightarrow a[i]=White) \wedge\forall i{:}int(right5<i<a.length\rightarrow a[i]=Red)\wedge a[right5]=Red$

$\Longleftrightarrow$True by Inv(3), (4), left5=right5 and case.

(*Inv. re-established at end of loop*) Let a1, left1, right1 and d1 be values of left, right and d just after the while loop test and a2, left2, right2, d2 be the values at the end of the loop.

Given: successful loop test (*) left1<right1;

(I1): $0\le left1\le right1<a.length$

(I2): $\forall i{:}int(left1\le i\le right1\rightarrow a1[i]=a0[i])$

(I3):$\forall i{:}int(0\le i<left1\rightarrow a1[i]=White)$

(I4): $\forall i{:}int(right1<i<a.length\rightarrow a1[i]=Red)$.

In (I5), we'll write S for bag(a0)-bag(store), since it does not change in the loop.

(I5): $((d1=Down \wedge bag(a1-a1[left1])=S)\vee(d1=Up \wedge bag(a1-a1[right1])=S))$.

It is required to show all parts of the invariant hold again at the end of loop. That is:

(I6): $0\le left2\le right2<a.length$

(I7): $\forall i{:}int(left2\le i\le right2\rightarrow a2[i]=a0[i])$

(I8):$\forall i{:}int(0\le i<left2\rightarrow a1[i]=White)$

(I9): $\forall i{:}int(right2<i<a.length\rightarrow a2[i]=Red)$.

(I10): $((d2=Down \wedge bag(a2-a2[left2])=S)\vee(d2=Up \wedge bag(a2-a2[right2])=S))$.

There are two cases, each with their own two sub-cases. We give proofs for the first case. The second case (d1=Down) and its sub-cases is completely analogous and left for you to complete.

Case 1: d1 = Up.

Sub-case 1a: a1[left1]=a0[left1]=White, left2=left1+1, d2=Up, right2=right1, a2=a1.

(I6): $0\le left2\le right2<a.length \Longleftrightarrow 0\le left1+1\le right1<a.length\Longleftrightarrow$True by (I1) and (*).

(I8): $\forall i{:}int(0\le i<left2 \rightarrow a2[i]=White)\Longleftrightarrow\forall i{:}int(0\le i<left1+1\rightarrow a1[i]=White)\Longleftrightarrow \forall i{:}int(0\le i<left1\rightarrow a1[i]=White) \wedge a1[left1]=White\Longleftrightarrow$True by (I3) and case.

(I9): $\forall i{:}int(right2<i<a.length\rightarrow a2[i]=Red)\Longleftrightarrow \forall i{:}int(right1<i<a.length\rightarrow a1[i]=Red) \Longleftrightarrow$True by (I4).

(I10): it is sufficient to show one disjunct – by ($\vee$I). The obvious one here is

$d2=Up \wedge bag(a2-a2[right])=S \Longleftrightarrow Up=Up \wedge bag(a1-a1[right1])=S$.

By the case, LH disjunct of (I5) is false (just inside the loop) since $d1\neq Down$

$\Longrightarrow$ RH disjunct must be true $\Longrightarrow bag(a1-a1[right1])=S$ is True as required.

(I7) is true by (I2) since no change is made to a.

NOTE: The trick used to show (I8) – i.e. to separate one case from the universal implication is very useful. It relies on the general structure:

$\forall i{:}int(0\le i\le v\rightarrow$condition involving a[i])$\Longleftrightarrow$

$\forall i{:}int(0\le i<v\rightarrow$ condition involving a[i]) $\wedge$ condition involving a[v]

Sub-case 1b: a1[left1]=Red, a2[right1]=a1[left1]=Red, right2=right1-1, left2=left1, d2=Down, elements of a other than a1[right1] unchanged.
(I6): 0≤left2≤right2<a.length<==>0≤left1≤right1-1<a.length<==>True by (I1) and (*).
(I8): ∀i:int(0≤i<left2→a2[i]=White)<==>∀i:int(0≤i<left1→a1[i]White)<==>True by (I3).
(I9): ∀i:int(right2<i<a.length→a2[i]=Red)<==> ∀i:int(right1-1<i<a.length→a2[i]=Red)
<==>∀i:int(right1<i<a.length→a1[i]=Red) ∧ a2[right1]=Red<==> True by (I4) and Case.
(I10): The obvious disjunct to make true here is d2=Down ∧ bag(a2-a2[left2])=S.
d2=Down is true by the case. bag(a2-a2[left2])=S <==> bag(a-a[left1])=S
<==> bag(a-a[right1])=S.
By the case LH disjunct of (I5) is false since d1≠Up ==> bag(a-a[right1])=S is True.
(I7): ∀i:int(left2≤i≤right2→a2[i]=a0[i])<==>
∀i:int(left1≤i≤right1-1→a1[i]=a0[i])<==>True by (I2).

(*Variant decreases*) The variant decreases each time through the loop since either left increases or right decreases. (I1) ==> right-left≥0. Hence the loop cannot go on forever.

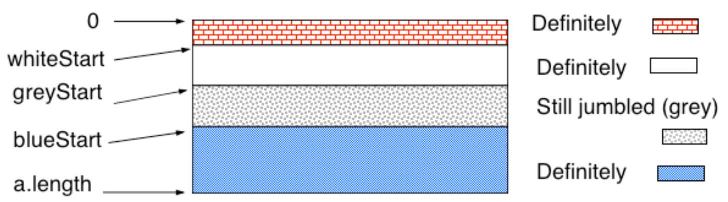
(*Valid array access*) The array accesses are a[0] - ok since a.length>0, a[left], a[right] within the loop - ok by (I1), a[left] after the loop - ok by (I1).

DUTCH FLAG PROBLEM GENERAL IDEA

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

enum Col{red, white, blue}

New Post: – restore will return wStart and bStart bundled as a pair.
a is a rearrangement of a0 ∧ 0≤wStart≤bStart≤a.length
∧ **r**=(wStart,bStart) ∧ ∀i:**int**(0≤i<wStart → a[i]=Red
∧ ∀i:**int**(bStart≤i<a.length → a[i]=Blue)
∧ ∀i:**int**(wStart≤i<bStart → a[i]=White)



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.

SPECIFICATION OF DUTCH FLAG

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

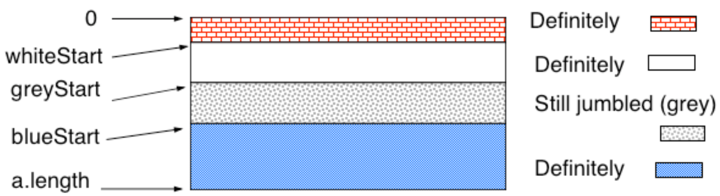
```
int restore (Col [] a) {
```

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

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

DutchReturnInfo dutchRestore (Col [] a) {   //Pre:   none
//Post:  ∀i: int (0 ≤ i < r.wStart → a[i] = Red) ∧
//          ∀i: int (r.wStart ≤ i < r.bStart → a[i] = White) ∧
//          ∀i: int (r.bStart ≤ i < a.length → a[i] = Blue)
//       ∧a is a rearrangement of a0∧0≤r.wStart≤r.bStart≤a.length}
```

LOOP INVARIANT



“The diagram is correct” –

$0 \le \text{whiteStart} \le \text{greyStart} \le \text{blueStart} \le \text{a.length} \wedge$
$\forall i\text{: } \textbf{int } (0 \le i < \text{whiteStart} \to a[i] = \text{Red}) \wedge$
$\forall i\text{: } \textbf{int } (\text{whiteStart} \le i < \text{greyStart} \to a[i] = \text{White}) \wedge$
$\forall i\text{: } \textbf{int } (\text{blueStart} \le i < \text{a.length} \to a[i] = \text{Blue}) \wedge$
a is a rearrangement of a0

Loop variant: number of greys left
= blueStart – greyStart.

CODE FOR DUTCH FLAG

```
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).

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.

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.

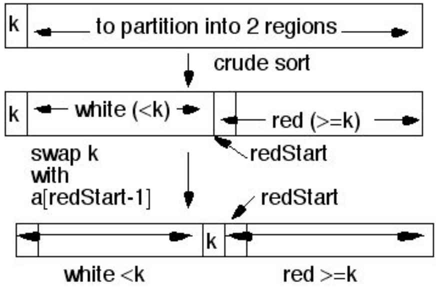
Idea: Sort an array of integers by a succession of crude sorts.

- First pick a "key" integer k and do a crude sort using the Polish National Flag method:

  white means "< k" *red* means ">= 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.

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.



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.

SPECIFICATION OF PARTITION

restore is called **partition** for this particular use (with key "k").

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

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

GENERAL SPECIFICATION OF SORT-REGION

```
void sortRegion(int [] a, int start, int rest) {
// Pre: 0  ≤ start ≤ rest ≤ a.length
// Post: a is a rearrangement of a0 ∧
//         haven't changed anything except between start and rest,
//         i.e. ∀i: int(0≤i< start or rest ≤i< a.length→a[i] = a0[i])
//          ∧ within the region, a is sorted,
//         i.e. ∀i,j: int(start ≤ i ≤ j < rest → a[i] ≤ a[j])
}
```

QUICKSORT CODE

```
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**$(start \le i \le j < rest \rightarrow a[i] \le a[j])$ and for $0 \le i < start$ and $rest \le i < a.length$ a is unchanged.

We'll use well-founded induction on the set

goodPairs(a) = {(s,r) | s:Nat, r:Nat ∧ 0≤s≤r≤a.length}

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

(s1, r1)<<(s2,r2) iff r1-s1<r2-s2

To prove the property for some given a, rest and start, 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 <==> rest-1=start

Case 2: rest-start=0 <==> rest=start

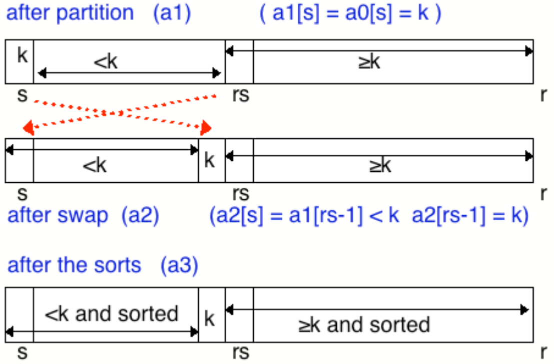
- In both cases quickSort obviously stops (with no change at all to a) and the postcondition is true. Exercise check it!
- Important part to check is $\forall i,j$: **int**$(start \le i \le j < rest \rightarrow a[i] \le a[j])$ especially for Case 1 (you need to check it for i=j=start=rest-1).

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 start+1≤redStart≤rest and, after the swap we know a[start] upto a[redStart-2] are <a[redStart] and also a[redStart-1] upto a[rest-1] are ≥a[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)

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

after the sorts (a3)

5 cases to check for sortedness:

i and j < rs-1; i and j ≥rs; i<rs-1 and j ≥ rs;

i < rs-1 and j=rs-1; i=rs-1 and j ≥ rs;

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≤a.length ==> (start, redStart-1) ∈ gp(a)
- Also redStart-1-start<rest-start 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
∀(s,r): if 0≤s≤r≤a.length then quickSort(a,s,r) stops & Post(a,s,r) is true, where Post(a,s,r) is ∀i(0≤i<s or r≤i<a.length→a[i] =a0[i]) ∧ ∀i(s≤i≤j<r→a[i]≤a[j]).

We use well-founded induction on the set
goodPair (a) ={(s,r): s:Nat, r:Nat, 0≤s≤r≤a.length}
with the ordering (s1,r1)<<(s2,r2) iff v(s1,r1)<v(s2,r2), where v(s,r) = r-s.

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

Suppose for some arbitrary values of s and r that 0≤s≤r≤a.length,
then (s,r) ∈ goodPair(a) and v(s,r)=n.

Assume as induction hypothesis (IH) that, for all (s',r') ∈ goodPair(a) such that v(s',r')<n, quickSort(a,s',r') stops and Post(a,s',r') is true.

Case 1: n=1. v(s,r)=1 means r=s+1.

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

Case 2: n=0. v(s,r)=0, i.e. r=s, is even simpler.

Case 3: v(s,r)>1.

From the Post of partition, s+1≤redStart≤r so v(s,redStart-1)<v(s,r) and v(redStart,r)<v(s,r).
In addition, the precondition of quickSort is satisfied in both cases.

Therefore, the IH can be applied to both recursive calls to show they terminate and result in a, from s upto redStart-1, and from redStart upto r, being sorted.

Let's label the various versions of a as on slide 52 – a1 after partition, a2 after swap and a3 after the recursive sorts. For short, we'll also use rs for redStart, s for start and r for rest.
In what follows, by "up to X" is meant including X-1 but not X.

By Post of partition, a1[s] = a0[s] and all elements in a1, from s+1 up to rs are <a1[s] (1) and those in a1 from rs up to r, are ≥ a1[s] (2).

The exchange moves a[s] to a[rs-1] (both valid array accesses by Post of partition),
so a2[s]=a1[rs-1] and a2[rs-1]=a1[s]. Hence a2[s]<a2[rs-1] = a1[s] (by (1)).

After the exchange and after the calls to quickSort, a, from s up to r, is sorted also, for the following reasons. Note that a3 from s up to rs-1 is an ordered permutation of a1 from s up to rs-1 (3), and a3 from rs up to r is an ordered permutation of a1 from rs up to r (4).

For any i and j :s≤i≤j<r, if i and j are both <redStart-1, then a3[i]≤a3[j] by (3) and if i and j are both ≥redStart, then a3[i]≤a3[j] by (4).
If i<rs-1 and j≥rs, then a3[i]<a1[s] (by (1) and (3)) and a1[s]≤a3[j] (by (2) and (4)).
If i=rs-1 and j≥rs-1, then a3[i]=a3[rs-1]=a1[s]≤a3[j] (by (2)).
If i<rs-1 and j=rs-1, then a3[i]< a1[s]=a3[j] (by (1)).

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, 21, 20, 3, 45, 19, 30} and n = 5.
  sorted(a)= {3, 8, 15, 19, 20, 21, 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:

```
void median(int [] a, int n) {
// Pre:     0 ≤ n < a.length
// Post:    a is a rearrangement of a0
//          ∧ a is "partially sorted" –
//          ∀j: int (0≤j<n → a[j]≤a[n] ∧ n≤j<a.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[**r**]=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!