Arrays as lists

- For abstract reasoning, arrays are often best thought of as lists.
- *Computational* methods for lists (e.g. cons, ++) of Haskell are not always best for arrays in Java.
- Nevertheless, we can still use the Haskell ideas in our *reasoning* about arrays
- Further illustrations of loop invariants.

ARRAYS REPRESENT LISTS

An array is characterised by
– its elements,
– and the order of its elements

In other words, abstractly, arrays are lists of elements.

Examples:
Given **double** [] a =new double[3] (i.e. a.length=3)

a represents the list [a[0], a[1], a[2]]

Given **double** [] a =new double[0] (i.e. a.length=0)

a represents the empty list []

(Java also has ArrayLists and Vectors, which are, in effect, variable length arrays. See later.)

MANIPULATING LISTS

If arrays are like "lists" then it could make sense to use
head, tail, last, concatenation (++) and sublists
when reasoning about them, even though it may not be simple to construct these things in Java.

Example:
Given **double** [] a =new double[3], **double** [] b =new double[4]

then a represents the list [a[0], a[1], a[2]],

head(a) = a[0], last(a) = a[2],

tail(a) = [a[1], a[2]],

[b[1], b[2]] is a sublist of b

a++b represents the list [a[0], a[1], a[2], b[0], b[1], b[2], b[3]]

Note for computing purposes, must also know how the elements are subscripted.

ARRAYS AS LISTS — AND SUBLISTS

Let a be any Java array, and let i, j be integers with 0≤i≤j≤a.length.
We write a(i **to** j) for the Haskell list
[a[i], a[i+1], …, a[j-1]] //usual convention on regions

Formally, a(i **to** j) is defined iff 0≤i≤j≤a.length
a(i **to** j) = [ ], if i = j
= a[i]: a(i+1 **to** j), if i < j

Properties (suppose someType [] a and a.length=n)

- If we view a as a list, that list would be a(0 **to** n).
  Let's define a-as-a-list = a(0 **to** n).
- If a(i **to** j) is defined then its length is j–i.
- If 0 ≤ i ≤ j ≤ k ≤ a.length then a(i **to** k) = a(i **to** j) ++ a(j **to** k)
- a(i **to** i+1) = [a[i]]
- for i<j, a(i **to** j) = a[i]:a(i+1 **to** j) = [a[i]]++a(i+1 **to** j)
  = a(i **to** j–1)++[a[j–1]], etc.

In this chapter we are concerned with transferring our reasoning about lists to reasoning about arrays, in order to make that reasoning simpler. We are not concerned (particularly) whether to use arrays, ArrayLists or Vectors to represent lists.

When reasoning about programs it can be convenient to talk about empty arrays. E.g., it is useful, when considering a portion of an array, to be able to state it is empty. This might be a condition for termination. You can define them in Java, e.g. as in int [] b = new int[0], even though they don't seem very useful in practice. On the other hand, an empty ArrayList in Java is a potentially useful object. You can add to or delete elements from an ArrayList, so increasing or diminishing its size; an empty ArrayList may indicate a special case to be considered differently from other cases.

The notation a(i to j) is chosen to represent the array elements a[i], a[i+1], upto a[j-1] (it does not include the element a[j]), as this conforms to the Java convention for array indices when i=0 and j=a.length. Note that a(i to j) is defined iff $0 \le i \le j \le$ a.length. It also follows the notation we use for segments of arrays in several of the algorithms considered in this part of the course, in which the end point is not included in the array. This choice also makes some things neater:

e.g. the length of the list (or number of array elements) is j - i; an empty list is a(i to i); joining two consecutive pieces of an array together is a(i to j) ++ a(j to k) = a(i to k).

Revision Exercise: On the next slide it is stated that there is just one function satisfying the given properties of `reverse`. Use list induction to show that if there were two "reverse" functions satisfying the properties, then they'd be equal.

USING WHAT WE KNOW ALREADY

You used lists in Haskell a lot, and so understand them quite well.

BUT the Haskell methods used cons and concatenation (++). Neither of these is used when constructing or manipulating arrays. So how does the Haskell understanding transfer to Java?

Example: *Haskell definition* of `reverse`

```
reverse [ ]   = [ ]
reverse (h:t) = (reverse t) ++ [h]
```

Properties:

```
reverse [ ]    = [ ]
reverse [h]    = [h]
reverse (s++t) = (reverse t) ++ (reverse s)
```

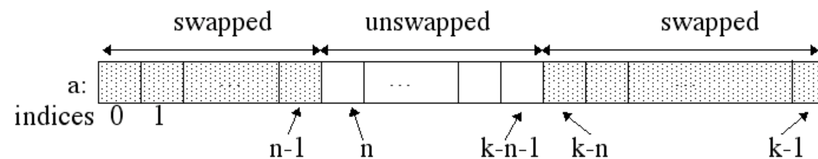
Actually, there is only one function with these properties, so we could use them as a specification for the many possible Haskell implementations of `reverse`.

JAVA REVERSE

```
void jReverse(double [] a) {
// elements of a could be any type;
// Pre:  none
// Post:a=reverse a0 (where reverse is the Haskell function)
//       i.e. a-as-a-list= reverse a0-as-a-list
  }
```

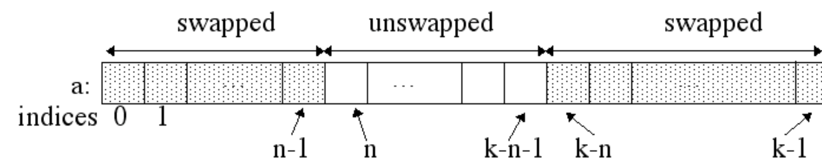
Idea: for efficiency, swap pairs of elements of a, starting at the two ends (swap a[0] with a[a.length-1]) and work towards the middle.

Do a diagram: n = number of pairs swapped, k = a.length.



Next move: swap a[n] with a[k–n–1]. (e.g. n=2:swap a[2] with a[k-3])

IDEA CONTINUED



From the diagram we see:

No. elements swapped = 2*n.

No. elements unswapped = k–(2*n).

Both must be non-negative.

Can stop when at most one element left unswapped, i.e. k–(2*n) ≤ 1.

This gives us a **loop variant**.

CODE

```
void jReverse(double [] a) {
// Pre: none
// Post: reverse a0 = a (both as lists)
final int K = a.length;
int n = 0;
    while  (K – 2*n > 1) {
        // Invariant:  ?? (Assume a.length =a0.length=K)
        // Variant: K-(2*n)
        swap(a, n, K–n–1);   n ++;
        // take care with swap; this one swaps a[n] with a[K-n-1]
        }
    }
```

(Note: Could instead declare k as in **int** k = a.length; would require (i) invariant to include k=a.length and (ii) to use k in place of K throughout in what follows.)

LOOP INVARIANT

Loop invariant:

$n \geq 0 \wedge K-(2*n) \geq 0$ //≥ 0 pairs swapped, and ≥ 0 unswapped

∧ `reverse` a0(0 **to** K) //the answer we want)

= a(0 **to** n) ++ `reverse` a(n **to** K–n) ++ a(K–n **to** K).

Initialisation (establishing invariant):

Initial code: n = 0 (no pairs swapped yet), K=a.length and a=a0

Then required to show –

$0 \geq 0 \wedge$ a.length ≥ 0 : true by arithmetic and property of length

and –

`reverse` a0(0 **to** a0.length)=

a0(0 to 0)++`reverse` a0(0 **to** a.length)++a0(a.length to a.length)

ie `reverse` a0=[ ]++`reverse` a0++ [ ]

This is true. So we established the invariant!

RE-ESTABLISHING INVARIANT

Assume the invariant holds at the start of an iteration and while condition is true (at least two elements left unswapped).

Write n1, a1 for the values of n, a at the start of this iteration.

- The invariant is true, so

`reverse` a0(0 **to** K)

= a1(0 **to** n1) ++ `reverse` a1(n1 **to** K–n1) ++ a1(K–n1 **to** K).

- The while cond is true: (K–n1)-n1≥2. So a1(n1 **to** K–n1) has length ≥2 and can be expanded. Then the expression for reverse a0

= a1(0 **to** n1)

++`reverse` ([a1[n1]]++a1(n1+1 **to** K–n1–1)++[a1[K–n1–1]])

++ a1(K–n1 **to** K).

By `reverse` properties (above), and associativity of ++, this

= a1(0 **to** n1)

++[a1[K–n1–1]]++`reverse` a1(n1+1 **to** K–n1–1)++[a1[n1]]

++a1(K–n1 **to** K).

In the iteration, we swap entries n1 and K–n1–1 of the array. So this

= a(0 **to** n1)

++ [a[n1]] ++`reverse` a(n1+1 **to** K–n1–1)++[a[K–n1–1]]

++a(K–n1 **to** K)

But then we increment n (i.e., n = n1+1, or n-1=n1), so this

= a(0 **to** n–1)

++ [a[n–1]] ++`reverse` a(n **to** K–n) ++[a[K–n]]

++a(K–n+1 **to** K)

But this is just

= a(0 **to** n)

++ `reverse` a(n **to** K–n)

++ a(K–n **to** K)

We've proved that this is equal to `reverse` a0(0 **to** K).

But this is just the (main bit of the) invariant for a, n after the iteration. So this part of the invariant is re-established.

Exercise: check other parts of Inv. (n≥0, etc.) are also re-established.

FINALISATION

Above argument worked when K–(2*n) ≥ 2. When the loop ends, the invariant is true and while condition is false, so 0 ≤ K–(2*n) ≤ 1. Then a(n **to** K–n) has length K–(2*n) = either 1 or 0. Either way

`reverse` a(n **to** K–n) = a(n **to** K–n).

Therefore, by the invariant,

`reverse` a0(0 **to** K))

= a(0 **to** n) ++ `reverse` a(n **to** K–n) ++ a(K–n **to** K)

= a(0 **to** n) ++ a(n **to** K–n) ++ a(K–n **to** K)

= a(0 to K).

That is, Postcondition holds (remember K=a.length):

`reverse` a0-as-a-list = a-as-a-list.

and can return.

Will it terminate?

Loop variant = number of elements left unswapped = K–(2*n) (which is ≥0 by Inv.). This decreases by 2 each iteration. Stop when it's ≤1.

ALTERNATIVE JAVA REVERSE USING A 'FOR' LOOP

The while loop in jReverse was **while** (K – 2*n >1){

K – 2*n > 1 <==> K – 2*n ≥ 2 <==> (2*n)/2 ≤(K–2)/2 <==> n ≤ (K–2)/2 (uses the fact that 2*n is even)

Use this as a limit for a for loop implementation.

```
void jReverse(double [] a) {
// Pre: none
// Post: a = reverse a0   (both as lists)
final int K = a.length;
    for (int n = 0; n <= (K–2)/2; n++) //could be done in parallel
        swap(a, n, K–n–1);
    }
```

Must still check termination – that only a finite number of n-values are considered; ie the list [0, 1, 2, .. , (K-2)/2] is finite.

Take care there are no effects on the loop variable n. (There aren't.)

REASONING ABOUT FOR LOOP IN JAVA REVERSE

```
for (int n = 0; n <= (K–2)/2; n++) //could be done in parallel
    swap(a, n, K–n–1);
```

Reasoning about this requires *Post* to be in terms of array elements:

∀i:int(0≤i<a.length → a[i]=a0[a0.length-i-1]) ∧ a.length=a0.length.

Must show for each i: 0≤i<a.length, there is some iteration I which makes a[i]=a0[a0.length-i-1]) true and this is not undone; i.e. no other iteration affects a[i].

In fact, iteration I makes a[I]= a0[K-I-1] and a[K-I-1]=a0[I] and no other iteration affects these two elements of a.

The condition a.length=a0.length is true as it is given by the postcondition of swap.

Exercise: Show by induction on the length of a that the new postcondition is equivalent to a = `reverse` a0.

EXAMPLE: STRING COMPARISON

Problem: given two strings, which comes first in lexicographic order? Lexicographic order is "Dictionary Order".
eg "cat"<"catch"; "cat"<"do"; "cat">"car"; "cart"<"cave"; "car"="car"

First attempt:

```
enum Ord{before, same, after}
int compare(char [] s, char [] t) {
//    Pre: none
//    Post: s=s0 and t=t0 and
//          ((r = Before and s is strictly before t in lex. order)
//          or (r= Same and s = t)
//           or (r= After and s is strictly after t))}
```

Track along s and t in parallel until either they disagree or one of them is exhausted. Then we can work out the result.

(Note: Before is short for Ord.before, etc.)

THE POST-CONDITION?

What does *lexicographic order* really mean? One explanation says – find the first place where s and t differ. So, using ^ for string concatenation, we write

s = u^a^s',   t = u^b^t',

where u, s', t' are strings and a, b are characters.

So:
- u = first part where s and t agree (could be empty)
- a and b ($a \neq b$) are the first differing characters,
- s' and t' are the remaining parts.

Then s is before (or after) t according as the unicode value of a is less than (or greater than) the unicode value of b, i.e. check a<b.

BUT this doesn't cover the cases where one string is an initial substring of the other (where u exhausts s or t) or when s=t:

- if t = s^t' & $t \neq s$ — then s is before t (u exhausts s), t' is non-empty
- if s = t — then they're the same
- if s = t^s' & $s \neq t$ — then s is after t (u exhausts t), s' is non-empty

PROGRAM IDEA

A possible Haskell solution:

```
data Order = Before | Same | After

listcomp [ ] [ ]     = Same
listcomp [ ] (y:t)   = Before
listcomp (x:s) [ ]   = After
listcomp (x:s) (y:t)
          | x < y    = Before
          | x==y     = listcomp s t
          | x>y      = After
```

Postcondition was:

// Post: s=s0 **and** t=t0 **and**

// ((**r** = Before **and** s is strictly before t in lex. order)

// **or** (**r**= Same **and** s = t)

// **or** (**r**= After **and** s is strictly after t))

STRING COMPARISON IN JAVA (1)

It is hard to give a neat characterisation of Post for jCompare.

Instead, relate the Java Post to Haskell function listcomp, (and later show listcomp is correct).

int jCompare(**char** [] s, **char** [] t) {

// Pre: none

// Post: **r** = `listcomp` s0 t0 (both s and t as lists) ∧ s=s0 ∧ t=t0

// Note that s and t won't change, so we could write s,t for s0,t0

// But we use s0, t0 as a reminder they are the initial values

STRING COMPARISON IN JAVA (2)

int jCompare(**char** [] s, **char** [] t) {

// Pre: none

// Post: **r** = `listcomp` s0 t0 (both s and t as lists) ∧ s=s0 ∧ t=t0

// In what follows we assume s and t don't change

// hence we can write s,t for s0,t0 throughout

Idea 1: Track along s and t in parallel (s[n] and t[n] are next characters to compare) until either they disagree or one of them is exhausted. Then we can work out the result.

Could use as Invariant

$0 \leq n \leq \min(s.length, t.length)$ (keep n within bounds)

$\wedge\ \forall i.\mathbf{int}\ (0 \leq i < n \ \rightarrow s[i] = t[i])$

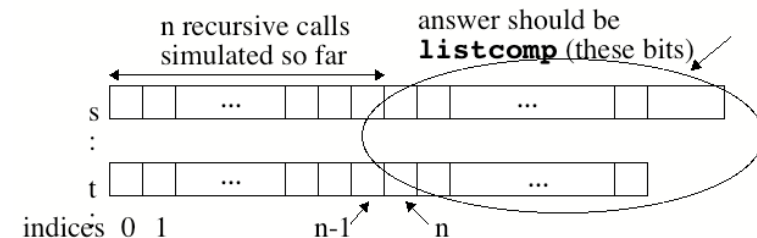
In English: 'No difference found yet'.

BUT: difficult to relate this invariant to Postcondition

Idea 2: Mimic the Haskell definition.

(We'll see a general form of this idea (called '*tail recursion*') soon.)

When `s` and `t` differ or one of them is exhausted we're in the Haskell base case and can work out the result.



Invariant imitates recursion of Haskell definition:

$0 \le n \le$ s.length $\wedge$ $0 \le n \le$ t.length $\wedge$

`listcomp` s0 t0 (the result we want using initial args)

//i.e. `listcomp` s0(0 **to** s.length) t0(0 **to** t.length)

= `listcomp` s(n **to** s.length) t(n **to** t.length)

(the result we'll get using current args)

METHOD

Having tracked along n elements, loop invariant tells us we can get the right answer simply by calculating

`listcomp` s(n **to** s.length) t(n **to** t.length)

To do this, look at parts of Haskell definition.

If s(n **to** s.length) or t(n **to** t.length) is [ ] (no more elements) then we can calculate the result immediately (use first 3 cases of definition.)

Otherwise (last 3 cases), compare s[n] and t[n]:
- if different, can calculate result;
- otherwise (recursive call) continue looping.

CODE

```
Ord jCompare(char [] s, char [] t) {
// Pre: none
// Post: r = listcomp s0 t0 (s and t as lists) ∧ s=s0 ∧ t=t0
  int n = 0
  // Invariant:    0 ≤ n ≤ s.length ∧ 0 ≤ n ≤ t.length ∧
  //               listcomp s0 t0 =
  //                  listcomp s(n to s.length) t(n to t.length)
  // Variant:      min(s.length, t.length)-n
  while          // Invariant true, s and t not exhausted
    ((n<s.length) &&  (n<t.length) && (s[n] == t[n])) {
                 // Neither s(n to s.length) nor t(n to t.length) is [ ]
                 // and s(n) = t(n); recursive case
      n ++;}     // s or t exhausted or s[n] < t[n] or s[n] > t[n]
                 // Finalisation - next slide
  }
```


FINALISATION

```
// Invariant true, and in base case of Haskell def. — copy it!
if (exhausted(s,n) && (exhausted(t,n))
      return Ord.same;
else if (exhausted(s,n))      // so s exhausted, t not exhausted
      return Ord.before;
else if (exhausted(t,n))      // so t is exhausted, s not exhausted
      return Ord.after;
else                          // neither exhausted
      if (s[n] < t[n]) return Ord.before;
else if (s[n] > t[n]  return Ord.after;
```

Loop variant: Use number of possible comparisons that may still be made: min(s.length, t.length) – n

```
boolean exhausted(char [] st, int k) {
  // Pre: 0<=k
  return (k >= st.length;}
```

For the record here are the proofs that compare is correct. We assume s=s0 and t=t0 throughout so

1) *Invariant is set up by initial code*. Required to show Invariant is true when n=0 and s=s0 and t=t0. i.e. show $0 \le 0 \le$ s.length $\wedge$ $0 \le 0 \le$ t.length $\wedge$ listcomp s0 t0 = listcomp s0 t0, which is true. (Note lengths are always ≥ 0.)

2) *Invariant + while condition false + final code implies postcondition*. False while condition means one or other (or both) of s or t is exhausted or neither is exhausted but they differ at position n. That is, one or more of the following conditions hold:
(i) n>=s.length, (ii) n>=t.length, or (iii) n<s.length $\wedge$ n<t.length $\wedge$ s[n]$\neq$ t[n].

There are 5 cases in all, which are:
(a) only s is exhausted, (b) only t is exhausted, (c) both s and t are exhausted, (d) neither s nor t is exhausted and s[n]<t[n], and (e) neither s nor t is exhausted and s[n]>t[n]. The code covers them all with the correct results. Below in (1) we show case (c), when both s and t are exhausted, and in (2) we give case (d). The other three cases are left as exercises.

(1) The invariant gives listcomp s0 t0 = listcomp s(n to s.length) t(n to t.length)
= listcomp [] [] = Same = **r.**
(The invariant specifies n≤s.length and ≤t.length, hence n=s.length=t.length.)

(2) The invariant gives listcomp s0 t0 = listcomp s(n to s.length) t(n to t.length)
= listcomp (s[n]: s(n+1 to s.length)) (t[n]: t(n+1 to t.length))
(s[n] and [t[n] are defined since n<s.length and n<t.length by case of while test)
= Before = **r** (since s[n]<t[n])

3) *Variant decreases*. The variant is min(s.length, t.length) - n, which decreases at each iteration as n increases. Within the loop it is guaranteed by the invariant to be ≥ 0 and so the loop must terminate.

4) *Array accesses are ok*: In the array accesses within the while condition it is known that 0 ≤ n < s.length and 0 ≤ n < t.length, by the invariant and the conjuncts of the condition.

5) *Invariant is maintained*: There are 2 parts to check. Let n1 be the value of n at the start of the loop code. Since the while condition holds, then n1<min(s.length, t.length) and s[n1]=t[n1]. The new value of n = n1+1, which guarantees that n≤min(s.length, t.length). The second part of the invariant is re-established as follows, using the conjunct s[n1]=t[n1] of the while condition:

At start of loop listcomp s0 t0 = listcomp s(n1 **to** s.length) t(n1 **to** t.length) (by Invariant)

listcomp s(n1 **to** s.length) t(n1 **to** t.length)
= listcomp s(n1 to s.length) t(n1 to t.length)
= listcomp (s[n1]: s(n1+1 to s.length)) (t[n1]: t(n1+1 to t.length))
(since n1<s.length and <t.length)
= listcomp s(n1+1 to s.length) t(n1+1 to t.length) (since s[n1]=t[n1])
= listcomp s(n **to** s.length) t(n **to** t.length)

all using the Haskell code for listcomp.

Hence listcomp s0 t0 = listcomp s(n **to** s.length) t(n **to** t.length) which is the Invariant at the end of the loop and what we had to show.

Course 141 – Reasoning about Programs

Tail Recursion

- Tail Recursion as a technique for transferring Haskell reasoning into Java.
- Tail recursion used to transform recursion into loops, so gaining in efficiency.
- Transform recursion into Tail Recursion using accumulating parameters
- Further illustration of loop invariants and reasoning with them

TAIL RECURSION (REVISION)

A definition of a function f is *tail recursive* iff the results of any recursive calls of f are used immediately as the result of f, without any further calculation.

An example:

```
isin x [ ]     = False
isin x (h:t) | h==x      = True
             | otherwise = isin x t
```

A non-example:

```
concat [ ] u    = u
concat (h:t) u  = (h:(concat t u))
```

Not tail recursive: the result of the recursive call (concat t u) is used in a further calculation; it has h put on the front.

Therefore in a tail recursive definition, the recursion is used simply to call the same function but with *different arguments*.

TAIL RECURSION = LOOPS

- Think of the tail recursion as meaning
  "do the same computation again, but with new arguments".
- In Java, keep variables for the arguments, and then tail recursion means "update the variables, and repeat". This is just looping.
- Loop invariant says:
  "the answer you originally wanted is the same as if you had calculated it starting with the variables you've got now":
  `"function arg0 = function arg"`

e.g. `listcomp` was tail recursive

```
int jCompare(char [] s, char [] t){
//    (See earlier  )
//  Invariant ... ∧ listcomp s0 t0 =
//                    listcomp s(n to s.length) t(n to t.length) ....
```

The method of converting Haskell tail recursion to Java loops is the same whatever the argument types, as we'll see.

EXAMPLE – ISIN

```
isin : Eq a => a -> [a] -> Bool
isin x []      = false
isin x (h:t)
   | x==h      = true
   | otherwise = isin x t
```

ISIN IN JAVA

```
boolean jIsIn(int x, int [] t) {
  // Pre:  none
  // Post: r = isin x0 t0 ∧ t is unchanged
  int i=0;
  while  //Inv: x=x0 ∧ t=t0 ∧ isin x0 t0 = isin x t(i to t.length)
         // ie result we want = result from here ∧ 0≤i≤t.length
         // Variant: t.length-i
     ((i<t.length) && (x!=t[i]))  {i++;}
  return (i!=t.length); }
```


PROOF THAT JAVA ISIN IS CORRECT (1)

Invariant is initially established:

Note: x and t are unchanged throughout – x=x0 and t=t0 always.

Required to show invariant true just before entering loop first time:

Given:

  i=0 (initialisation by code)

To show:

  isin x0 t0 = isin x t(i to t.length) ∧ 0≤i≤t.length

First conjunct

<==> isin x0 t0 = isin x t(0 to t.length) (substitute for i)

<==> isin x0 t0 = isin x0 t0 (x and t are unchanged)

Second conjunct is true by arithmetic.

PROOF THAT JAVA ISIN IS CORRECT (2)

Postcondition is achieved:

At end of loop either i≥t.length or x=t[i]. Also Invariant is true:

isin x0 t0 = isin x t(i to t.length) ∧ 0≤i≤t.length ∧ t=t0

Case 1: i≥t.length.

i≤t.length (by Inv.) and i≥t.length (by Case) ==> i=t.length

Invariant ==> isin x0 t0 (result we want)

= isin x t(i to t.length) = isin x t(t.length to t.length)

= isin x [] = False (by Haskell code) = Java result **r**

Case 2: (i<t.length and x=t[i])

Since i<t.length , t(i to t.length)=t[i]:t(i+1 to t.length).

Hence isin x0 t0 = isin x t(i to t.length)

= isin x t[i]:t(i+1 to t.length)

= True (by case, Haskell) = Java result **r**

PROOF THAT JAVA ISIN IS CORRECT (3)

Invariant is re-established:
Let i1 be value of i just after the while test on an arbitrary iteration.
(No need for x0 and t0 as no changes made to either.)

Given: i1<t.length ∧ t[i]≠x ∧
`isin` x t = `isin` x t(i1 to t.length) ∧ 0≤i1≤t.length

Required to show invariant holds at end of loop (when i=i1+1):
`isin` x t = `isin` x t(i to t.length) ∧ 0≤i≤t.length

0≤i1<t.length (by Inv and loop test)
<==> 1≤i1+1≤t.length ==> 0≤i≤t.length.

`isin` x t = `isin` x t(i1 to t.length) (by Invariant)
= `isin` x t[i1]:t(i1+1 to t.length) (i1<t.length)
= `isin` x t(i1+1 to t.length) (by Haskell x ≠t[i1])
= `isin` x t(i to t.length), so the invariant is re-established.

Proof that jIsIn is correct

Remaining proofs required to show that jIsIn is correct.

Invariant is set up: Substitute i=0 in the invariant, which becomes 0≤0≤t.length ∧
`isin x t(0 to t.length)=isin x0 t0.`
Since `isin x t(0 to t.length)=isin x t` and x and t are the same as x0 and t0, and lengths are always ≥0, the invariant is true.

Postcondition set up: See slide 6. Note that in Case 2 we rely on the fact that in Java a test such as i<t.length && x ≠ t[i] evaluates the first condition and if it is false does not evaluate the second condition. In a language where this was not the case, you need to code using if-statements inside the while loop and employ an additional boolean variable, such as "finished",

Variant decreases:
as i increases, t.length-i decreases. By Inv. i ≤t.length, so the loop must stop.

Array accesses ok: Note that 0≤i<t.length in the loop.

In fact, jIsIn could also have been implemented directly in terms of linked lists or ArrayLists. (See later.) Using the more abstract list operations perhaps the algorithm can be checked for correctness more easily (although the proof doesn't seem too hard in this case).

Of course, at some point it must be proved that the abstract operations provided by the implementation (eg by ArrayLists) are correct.

TAIL RECURSION – GENERAL SCHEME

Haskell definition (assuming `f` has one parameter):

```
f x
   |c1   = a1
   |c2   = a2
   |…    = …
   |d1   = f x1
   |d2   = f x2
   |…    = …
```

a1, a2, … are expressions giving results in the non-recursive cases.
x1, x2, … are the new parameters used in the tail recursive cases.

a1, a2, …,x1, x2, …, as well as the guards c1, c2, …, d1, d2, …, are all calculated simply, without recursion.

No difficulty in making this work if f has more than one parameter.

Exercise:
Use the general translation given on next slide to obtain jIsIn.

LOOP TRANSLATION IN GENERAL

```
resultType jf(someType x, …) {
// Pre:  any preconditions needed for f
// Post: r =  f x0
  while  (!c1 && !c2 && …) {
  // Inv: f x0  = f x    <<--NOTE!!
  //  ∧ any preconditions for f in terms of x that are not implied
  //   by true while condition ∧ conditions for ok array bounds
  // Variant: same as value used to show Haskell terminates
      if       (d1)  {x = x1;}
      else if (d2)  {x = x2;}
      else if …  { }
   }
  if       (c1) return a1;
  else if (c2) return a2;
  else if …;
  }
```

ANOTHER EXAMPLE: GCD

```
--pre: x>0 ∧ y≥0
gcd x y
    | y==0       = x
    | otherwise = gcd y (mod x y)
```

```
int jGcd(int x, int y) {
  // Pre:   x>0 ∧ y ≥ 0
  // Post: r = (gcd x0 y0)
  while  (y != 0) {
        // Inv.: gcd x0 y0 = gcd x y ∧ x>0 ∧ y≥0
        // Variant = y
        int oldx = x;  x = y;
        y = oldx % y }  // variant decreased
return x; }
```

See slide 12 for verification of jGcd. Note the Inv. includes the precondition for x, y. (The harder bit is to show Haskell `gcd` works.)

Proof that jGcd is correct.

We show that for jGcd the postcondition is set up at the end of loop and the invariant is maintained within the loop.

Postcondition is set up by jGcd.
At the end y!=0 is false, so y=0. Required to show **r** = `gcd` x0 y0.
`gcd` x0 y0 = `gcd` x y (by the Inv.) = `gcd` x 0 (since y = 0) = x (by Haskell `gcd`)
= result **r** (by Java jGcd).

Invariant is established at start of loop.
Required to show `gcd` x0 y0 = `gcd` x y ∧ x>0 ∧ y≥0. First conjunct is true since x=x0 and y=y0 at the beginning and second and third conjuncts are true by Precondition.

Invariant is re-established by loop code.
Let x1 and y1 be the values of x and y at start of loop code.
Know (1) x1>0 (by Inv) and (2) y1>0 (y1≠0 by loop test and y1≥0 by Inv).
Also know: `gcd` x0 y0 = `gcd` x1 y1.
Required to show `gcd` x0 y0 = `gcd` x y. At the end of the loop code x = y1 and y = x1%y1. According to the Haskell code `gcd` x1 y1 = `gcd` y1 x1%y1 = `gcd` x y – note the precondition of `gcd` is ok by (1) and (2). Hence `gcd` x0 y0 = `gcd` x y. The other part of Invariant (x > 0 ∧ y ≥ 0 – i.e. y1 > 0 ∧ x1%y1 ≥ 0) is true by (2) and property of % .

Loop terminates.
Variant y decreases on each iteration since y = x1%y1 < y1 by definition of %.
Since y is always ≥0 within the loop by the Invariant, the looping cannot continue forever.

NOT ALL FUNCTIONS ARE TAIL RECURSIVE

```
--pre: n≥0    post: r = !n where r = fact n
fact n
    | n==0       = 1
    | otherwise = n*(fact (n–1))
```

BUT residual computations (n*…) can be "accumulated" into a single variable (you saw this many times in Haskell lectures):

```
--pre: n≥0
factTR m n
    | n==0       = m
    | otherwise = factTR (m*n) (n–1)
```

m is the *accumulator parameter* in `factTR`.

Postcondition of `factTR`?

```
--Post:r = m * fact n where r = factTR m n
```

Can then calculate `fact n` by `factTR 1 n`

FROM NON-TAIL RECURSIVE FUNCTIONS TO LOOPS VIA TAIL RECURSIVE FUNCTIONS

(1) Given correct non-tail recursive function `f` and the corresponding tail recursive function `fTR` must show that `f` and `fTR` compute the same result.

(2) The function `fTR` with the accumulating parameter *is* tail recursive, so can convert it into a loop.

(3) In fact, in Java, often don't need to implement `fTR` (called jFTR) separately from `f` (called jF) as can often amend the loop in the computation of jF to obtain the loop of jFTR, *by adding an extra local variable for the accumulator parameter* (initialised at the start, before the loop).

(4) Prove jFTR correctly implements `fTR`.

(5) Check `f` is correct!

Proof that `fact = fact1 1` (Revision) (Step 1)

Prove *by induction on n* that

`∀m:int ((factTR m n) = m*(fact n))`

Then `factTR 1 n = 1 * (fact n) = fact n`

Proof *Base case*: (n=0). Let M be an arbitrary int;
`factTR M 0=M` (by code) `= M*1=M*(fact 0)` (by code)
Induction step: assume as inductive hypothesis (note the `∀m`)
`∀m:int((factTR m n) = m*(fact n))`
Then, let M be an arbitrary int; `factTR M (n+1)`
`= factTR (M*(n+1)) n` (by code for `factTR`)
`= (M*(n+1))*(fact n)` (by Ind. Hyp. - here m is `M*(n+1)`)
`= M*(fact (n+1))` (by associativity of * and code for `fact`)

Very important note

Can't prove `fact n = factTR 1 n` directly by induction on n.
Must understand better how the accumulator parameter works, and prove a stronger statement. (`∀m((factTR m n)=m*(fact n))`.
You saw examples of this in the Induction part of the course.

IMPLEMENTATION USING A LOOP (STEPS 2 AND 3)

```
int jFactTR(int n) {
// Pre: n≥0
// Post: r = fact n0 (or factTr 0 n0)
  int m  = 1;  // m is the accumulator
  while       (n != 0) {
     // Inv:  fact n0 = factTR m n ∧ n≥0
     // Note n≥0 needed for pre of factTR in reasoning
     // Variant = n
     m = m*n;   n--;      //get new arguments m and n
   }
  return m;              //base case of factTR
  }
```

Could also code the recursive definition of `fact` directly into Java.
But this version with **while** is *much more efficient*.

Proof that jfactTR is correct (Step 4)

The proof that jFactTR is correct follows exactly the same pattern as the proof that jGcd is correct. Compare the two proofs to convince yourself this is so. (In fact, I just edited the proof for jGcd to get the proof for jFactTR!)

Postcondition is set up by jfactTR
At the end n != 0 is false ==>n=0 since n≥0 by Inv. Required to show **r**=`fact` n0.
`fact n0=factTR m n` (by the Inv.) = `factTR` m 0 = m (by Haskell `fact1`)
= result **r** (by Java jFactTR).

Invariant is established at start of loop
Required to show `fact` n0 = `factTR` m n ∧ n≥0. First conjunct is true since n=n0 and m=1 at the beginning (and `factTR` 1 n = `fact` n) and second conjunct is true by Precondition.

Invariant is re-established by loop code
At the start of the loop code of jFactTR the Inv. gives `fact` n0 = `factTR` m1 n1, where m1 and n1 are values of m and n at start of loop code. At the end of the loop code m = m1*n1 and n = n1-1. According to the Haskell code `factTR` m1 n1 = `factTR` m1*n1 n1-1 = `factTR` m n. (Note that the precondition of `factTR` m1 n1 is ok (n1 ≥ 0) by the Inv.). Hence `fact` n0 = `factTR` m n. The other part of Invariant (n ≥ 0 – i.e. n1-1 ≥ 0) is true by the true while condition and invariant (n1 != 0 & n1 ≥ 0 ==> n1 > 0 ==> n1-1 ≥ 0).

Loop terminates
Variant n decreases on each iteration since n1-1<n1. Since n is always ≥0 within the loop by the Invariant, the looping cannot continue forever.

OTHER DATA STRUCTURES

So far we've implemented Haskell list functions as methods using arrays, which represented our lists. However, there are other representations for lists eg ArrayLists, which implements the List interface using arrays. The reasoning can then use properties of lists explicitly, which may be easier.

We show the idea for jIsIn. Here's our previous version in Java:

```
boolean jIsIn(int x, int [] t) {
  // Pre:  none
  // Post: r =  isin x0 t0 ∧ t=t0
  int i=0;
  while  //Inv: x=x0 ∧ t=t0 ∧
         //       isin x0 t0=isin x0 t(i to t.length) ∧ 0≤i≤t.length
         // Variant: t.length-i
     (i<t.length && x!=t[i])   i++;
  return (i!=t.length);
  }
```

ISIN IN HASKELL

```
isin : Eq a => a -> [a] -> Bool
isin x []       = false
isin x (h:t)
    | x==h       = true
    | otherwise = isin x t
```

isin IN JAVA USING (Generic) ArrayLists

```
boolean jIsIn2(int x, List<Integer> t) {//See comment on 21
  // Pre:   none //t is declared as a List of Integers
  // Post:  r = isin x0 t0  ∧ t = t0
while  (! t.isEmpty() && x != (t.get(0)).intValue())
    //get value of head of t; Java knows it is an Integer
    // Invariant: isin x0 t0 = isin x t ∧ x = x0 ∧ t = t0
    // Variant: length of t
     {t=t.subList(1,t.size()); // set t to tail(t)}
  return !t.isEmpty();  }
```


Proof that jIsIn2 *is correct* — it should be quite easy. Assume x=x0 throughout.
1) *Invariant set up by initial code* (there isn't any!) `isin x0 t0 = isin x0 t0`.
2) (*Invariant + false while condition + final code*) *implies postcondition:*
Successful loop exit implies either t is empty or (t is not empty $\wedge$ x=t.head()).
The invariant says the required result **r** (= `isin x0 t0) =isin x t`.
<u>Case 1</u> <u>(t=[]):</u> `isin x []=False`, so the actual result given by the code (= !t.isEmpty() = false) is correct.
<u>Case 2</u> <u>(x=t.head()):</u> `isin head(t) t =True`, so the actual result given by the Java code (= !t.isEmpty() = true) is correct.
3) *Variant decreases*: length(t.tail())<length(t). The loop must stop as the length of a non-empty list cannot decrease forever.
4) *Invariant is maintained*: Let t1 be the value of t just after the while condition succeeds. x=x0 as it is unchanged by the code, so the invariant states `isin x0 t1=isin x0 t0`.
A true while-condition means t1 $\neq$ [] and x$\neq$t1.head().
We show `isin x t= isin x0 t0.`
`LHS = isin x0 t1.tail() =isin x0 t1` by the Haskell code taken from right to left (since x$\neq$t1.head()) `=isin x0 t0` by the invariant. So `isin x t=isin x0 t0.`
5) As the structure is an ArrayList must check access: element 0 is accessed twice and exists in the loop as the loop is non-empty by the while condition.

This kind of reasoning is just as we did before. To remind you:

- **The post-condition is in terms of a Haskell function `f`, of the form r = `f` initial-args.**
- **The invariant is also in terms of `f`, of the form `f` initial-args = `f` current-args.**
- **There may need to be additional requirements to enforce the preconditions of `f`.**

VARIATIONS

In the code on slide 19 the method jIsIn2 is defined for a List argument. The actual parameter can be an implementation of List, namely ArrayList, declared (eg) as in

```
ArrayList<Integer> t = new ArrayList<Integer>(10);
```

After adding some elements jIsIn2 can be called by jIsin2(6,t) for example.

You can also implement jIsIn2 using an iterator for the ArrayList argument:

```
boolean jIsIn2(int x, ArrayList<Integer> t) {
  // Pre:  none         Post:  r = isin x0 t0
  Iterator<Integer> list = t.iterator();
  // Inv: isin x0 t0 = isin x t  x=x0 ∧ t = t0 and Variant is as before
  while  (list.hasNext() && (x != (list.next()).intValue())) {//Side effect}
  return !list.hasNext();  }
```

Note that hasNext has the side effect of moving one item along the list.
Variant still cannot go negative without causing exit from the method. Correctness is DIY!

Or even simply recursive:

```
if t.isEmpty() return false;
if (x == (t.get(0)).intValue()) return true;
t=t.subList(1,t.size()); return jIsIn3(x, t);
```


GENERAL METHOD

1) Find obvious solution in Haskell (usually easy)

2) Prove the function is correct by induction (often the hardest part).

3) Find less obvious tail recursive solution in Haskell and the relation between it and non-tail recursive function (sometimes not so easy).

4) Prove the two functions give the same answers by induction.

5) Translate the tail recursive version to Java with **while** loops (easy).

6) The loop invariant can be written down immediately in terms of the Haskell functions. If the base function *H* is tail recursive the invariant is trH args0 = trH currentArgs (eg isIn was like this). If not the invariant is H args0 = trH currentArgs (eg fact was like this).

7) Prove the Java method is correct (usually easy).

MAIN SUMMARY

Steps for developing an algorithm using a while loop:

(i) Decide on the precondition and postcondition
(ii) Work out the general idea
(iii) Draw a diagram to capture a typical state of the program "half-way through" (at the beginning of an arbitrary iteration of the loop)
(iv) Use the diagram to give the invariant and variant
(v) The property (variant>0) often gives the while condition
(vi) The intialisation code (before the while loop) should establish the invariant
(vii) The finalisation code (after the while loop) should establish the postcondition. This uses the false while condition and the invariant.
(viii) The loop code must be crafted (following the general idea of (ii)) in order to re-establish the invariant. May be useful to include additional midconditions at key places in the code.

WHERE DO THE INVARIANTS COME FROM?

Look at the post-condition. It may be in terms of a variable that could be replaced by a counter in the implementation. Then the invariant can mirror the post-condition using the current value of the counter instead of the final value. Usually there also need to be limits for the counter.

Draw a picture to illustrate the algorithm when it is part way through. Put as many features as you can on the diagram. Translate these to logic and they become the invariant.

These approaches usually work well for array methods and functions. But almost always the invariant comes from thinking about the algorithm. It is usually a conjunction of relationships between the variables of the program and between the variables and the various constants of the program.

SUMMARISING ...

1. In practice, the mid-conditions that are crucially necessary for a while loop are the pre- and post-conditions and the loop invariant. These are the ones that really explain your thinking. Write them down as comments in the program.
2. The rest of the reasoning counts as “exercise for the reader” – you should go through it privately, and so can anyone else who examines the code. But it’s relatively obvious, and including it as comments may obscure the overall argument.
3. Introducing logical variables (a1 and n1) as notation for values – this is common in loops because the same logical formula (the invariant) is considered at two different times.
4. **for** loops in Java are very flexible and can be used to operate as a **while** loop. Remember that we will only use them when each iteration of the body could be executed in parallel.

WHAT LOGIC CAN I PUT IN THE INVARIANT?

You can use whatever is easiest to reason with.

Sometimes it is convenient to *treat arrays as lists*.

Example:

in(x, a(0 to i)) instead of $(E)k{:}int(0 \leq k < i \rightarrow a[k]=x)$

Sometimes it is easiest not to.

Don't worry if the invariant turns out to be incomplete. As you try to reason (often when showing the postcondition), you will be able to work out the missing bits and add them in.

If you get stuck, *write the properties in English*. It is better than nothing.

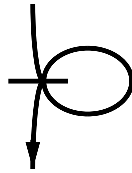
But remember, invariants are there to help you check your algorithm is correct. So do try to show they **are** invariants.

We developed algorithms for:

1. Binary chop
2. Restore (Polish national flag), leading to
3. Sorting (Quicksort)
4. Warshall's algorithm for transitive closure and Floyd's algorithm for shortest path

We looked at arrays as lists, and tail recursion.

For efficiency's sake, non-tail recursion can be turned into tail recursion, and tail recursion can be turned into a **while loop**.



APPENDIX – FOR LOOPS

```
public class Matrix{
  private int [] [] m, int size;
  //class invariant: size>0

public Matrix(int n){
    m = new int [n] [n]; size = n;
    }

//public String toString()  for printing out matrices

public void zeromatrix(){
    for (int i=0; i<size; i++)
      for (int j=0; j<size; j++)
        m[i][j] = 0;
    }
  }
```

What restriction would ensure the invariant is true initially?

FOR LOOP REASONING

for loops are typically used to do the same operation to all elements of an array. Different iterations of the loop do not interfere with each other and the fact they happen in some particular order is irrelevant.

(i) *Sometimes the operations could be executed in parallel.* Such **for** loops can be thought of as "do all these".

e.g. `for (int i = 0; i < a.length; i++) a[i]=0;`

(ii) *Sometimes this is not so, although the iterations could still be executed in any order.* Such **for** loops can be thought of as "do this, then this,...". e.g. `s = 0; for (int i = 1; i <= 5; i++) s+= i;`

Why can't the operations occur in parallel here?

It is safest to reserve **for** loops for independent operations as in (i); a **for** loop can be coded as a **while** loop and for reasoning purposes it is often simpler to reason about the corresponding **while** loop.

PROVING THE POSTCONDITION HOLDS

```
for (int i=0; i<size; i++)
    for (int j=0; j<size; j++)   m[i][j] = 0;
```

Let I and J be integers $0 \le I,J<size$. Show that, at the end, m[i][j] = 0.

Proof: there was an iteration of the **for** loops (namely with i=I and j=J) in which m[i][j] became 0; once that was done, none of the other iterations would ever undo it.

The pattern is quite general, and very easy. You reason that everything necessary was done, and then (because the iterations are independent) never undone. **for** loops should always terminate!

```
s = 0; for (int i = 1; i <= 5; i++)s+= i;
```

- We must show that s=(Sum(i=1 to 5)(i)). Let I be an arbitrary int.
- There is exactly one iteration which adds I to s. Since this step is not undone, s=(Sum(i=1 to 5)(i))+initial value =(Sum(i=1 to 5)(i)) +0 = (Sum(i=1 to 5)(i)).

You may think that the second kind of for loop could be run in parallel. But consider again

```
s=0; for ( int i =1;  i<= 5;  i++){s+=i}
```

It does satisfy a kind of independence — it doesn't seem to matter in which order the steps are taken. To show the loop gives s=15, we could try to show that s=(Sum(i=1 to 5)(i))+0 (=1+2+3+4+5) = 15.

We could argue as follows: imagine s is a location (a 'bucket' if you wish), initially with value 0. For an arbitrary I, the Ith iteration adds I into the location s and nothing removes it. So at the end of the loop every value of i has been added in and the value of s is the total sum. However, it is only correct if we assume the additions are made at different times.

Imagine that we tried to make the additions for i =2 and i=3 at *exactly* the same time. We might argue as follows: "look in s" — the value is (say) 1 at the moment; "compute i+1 = 3" (i.e. 2+1) — make sure the value of s is now 3. But at the same time, the computation for i=3 would result in the conclusion "make sure the value of s is now 4". So what is the value of s? There are various calculi for reasoning about such parallel operations - see the second (and fourth) year courses in concurrency.

Of course, in practice, you will use **for** loops even when the operations are not independent. However, such loops are really masquerading as **while** loops and when reasoning about them you need to use the technique of invariants.

A TYPICAL FOR LOOP?

```
boolean neg1 (int [] a) {
//post: r ↔ a contains a negative integer ∧ a=a0
  boolean isNeg = false;
  for (int i=0; i<a.length; i++) isNeg=isNeg || (a[i]<0);
  return isNeg;}
```

If a[i]≥0 for every i then isNeg = **result** = false, which is correct.
If a[i]<0 for some i, say I, then isNeg = true after the Ith iteration and stays true, and **result** = true, which is correct.

```
boolean neg2 (int [] a) {
for (int i=0; i<a.length; i++) if a[i]<0 return true;
  return false;}
```

If a[i]≥0 for every i then neg2 returns from outside the for loop with **result** =false, which is correct. If a[i]<0 for some i, say I, then neg2 would return after the Ith iteration with **result**=true, which is correct.

BUT ...

• The result of executing neg1 and neg2 would be the same *even if* the iterations of the for loop were executed in a different order.

• The reason is that although the answer could have been determined by any of the iterations, that answer would be the same in all cases.

• This is ***not*** the case for neg3.

```
int neg3 (int [] a) {
//post "returns the first value of i: a[i]<0 (if any)" ∧ a=a0
//formally??
for (int i=0; i<a.length; i++) if a[i]<0 return i;
  return a.length;}
```

• The answer depends on which for loop iteration causes the return.
• Use a **while** loop for this kind of **for** loop.

CONVERTING FOR LOOPS TO WHILE LOOPS

Generally, a **for** loop of the form

```
for (<init> <test> <inc>) <code>
```

becomes the while loop

```
<init>
// inv true here
while <test> {    // inv true here and <test> true
      <code>
      <inc>       //variant decreased
      }
                  // inv true here and <test> false
```

It's up to you to find the right variant and invariant for the problem.
The variant is often 0 when the loop stops – so test is variant>0.
The invariant often includes the property variant≥0.

Let's apply the method to our earlier for loop:

```
s=0; for (int i= 1; i<=5; i++) s=s+i;
```

As a **while** loop it becomes

```
s=0;
int i = 1;
while i<=5 {            //inv true here and while condition true
    s = s+i;  i=i+1;   }   //inv true here and while condition false
```

In order to show this loop adds together the first five positive integers we must find and include the correct mid-condition as invariant and show it is maintained. We must also find a variant and show the loop stops at the right time.

The *variant* in this case is 6-i (the loop will stop when it reaches 0). $6-i>0 \iff 5-i\geq 0$, so the loop test is equivalent to variant>0.

The *invariant* should represent the state when we are making progress. It should tell us what we have added to s so far. It is $s=\text{Sum}(k)(k=1 \text{ to } i-1) \wedge 1\leq i\leq 6$. We make the convention that Sum(k)k=1 to 0 is 0. Now we show that the loop works and also that it stops. Note the second conjunct of the invariant – it's important! It is equivalent to variant≥0.

The loop stops: the variant decreases at each iteration since i increases. Within the loop (i.e. when the while condition is true) the variant is >0. Hence the loop must stop since the variant cannot continue decreasing and remain >0.

The invariant is set up initially: when i=1, s should be 0; it is. Also $1\leq 1\leq 6$.

The invariant is maintained: call the values of i and s at the start of the loop i1 and s1. The new requirement is $s= \text{Sum}(k)(k=1 \text{ to } (i1+1)-1) = s1+i1$. This is exactly what the loop code computes — first it adds i1 to s1, then it increments i1 to i1+1. Also $1\leq i1+1\leq 6$ as the true while condition gives $i1\leq 5$. More formally, you have to show that

$1\leq i1\leq 6 \wedge i1\leq 5 \wedge s1=\text{Sum}(k)(k=1 \text{ to } i1-1)$ (*before the code*)
$i=i1+1 \wedge s=s1+i1$ (*effect of code*)
$\Longrightarrow 1\leq i\leq 6 \wedge s=\text{Sum}(k)(k=1 \text{ to } i-1)$ (*after the code*)

First, $1\leq i1\leq 6 \wedge i1\leq 5 \iff 1\leq i1\leq 5 \iff 2\leq i\leq 6 \Longrightarrow 1\leq i\leq 6$
Then, $s1=\text{Sum}(k)(k=1 \text{ to } i1-1) \iff s1+i1=\text{Sum}(k)(k=1 \text{ to } i1-1)+i1$
$\iff s1+i1 = \text{Sum}(k)(k=1 \text{ to } i1) \iff s =\text{Sum}(k)(k=1 \text{ to } i-1)$.

The result is correct: at the end the false while condition (loop exited) tells us i>5 and the invariant that $i\leq 6 \wedge s=\text{Sum}(k)(k=1 \text{ to } i-1)$. So i=6 and s=Sum(k)(k=1 to 5). Done!

Generally, a for loop of the form **for (**<init> <test> <inc>) <code> becomes the while loop

```
<init>while <test> {                    //inv true here and <test> true
    <code><inc>     //variant decreased}     //inv true here and <test> false
```

It is up to you to find the right variant and invariant for the problem.
The variant is often 0 when the loop stops and the invariant often includes variant≥0.
E.g. in the above example the variant was 6-i; it is 0 when i=6, which is when the loop will terminate. In addition $6-i\geq 0$ is equivalent to $6\geq i$, which is included in the invariant.
Exercise: Formalise neg3 as a while loop with suitable invariant.