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

Some 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, its length is j–i; (eg a(2 to 4) =[a[2],a[3]])
- 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, say `reverse1` and `reverse2`, then they'd be equal – ie for all `ls`, `reverse1(ls) = reverse2(ls)`.

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: (One particular) *Haskell definition* of `reverse`

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

Properties:

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

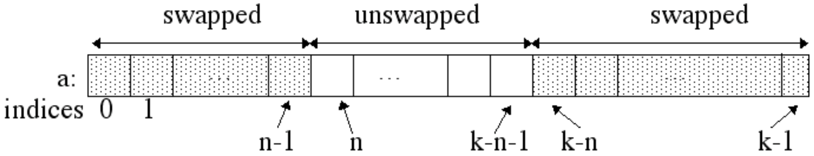
Actually, there is only one function with these properties, so we could use them as a specification for the many possible Haskell implementations of `reverse`.

JAVA REVERSE

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

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.

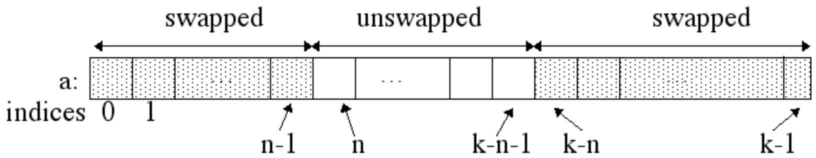
Do a diagram: n = number of pairs swapped, k = a.length.



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

IDEA FOR JAVA IMPLEMENTATION

We use the Haskell function in the specification of the Java method; can use any Haskell implementation of reverse; we chose the simplest



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

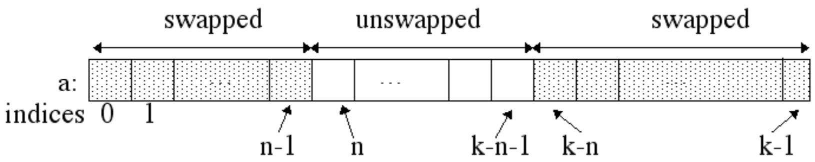
giving a **loop variant**.

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 **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: (look again at the diagram)

n ≥ 0 ∧ K–(2*n) ≥ 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)

Old style invariant might have been:

∀i:int(n≤i<k-n→ a0[i]=a[i]) ∧ n ≥ 0 ∧ K–(2*n) ≥ 0 ∧
∀i:int(0≤i<n→ (a0[i]=a[K-1-i] ∧ a0[K-1-i]=a[i]))

Need the first part as a particular rearrangement is required.

LOOP INVARIANT IS ESTABLISHED

Loop invariant:

n ≥ 0 ∧ K–(2*n) ≥ 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 ≥ 0 ∧ 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).

Write n2, a2 for the values of n, a at the end of this iteration.

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

= a2(0 **to** n1)
 ++[a2[n1]]++`reverse` a2(n1+1 **to** K–n1–1)++[a2[K–n1–1]]
 ++a2(K–n1 **to** K)

But then we increment n (i.e., n2 = n1+1, or n2-1=n1), so this

= a2(0 **to** n2–1)
 ++ [a2[n2–1]] ++`reverse` a2(n2 **to** K–n2) ++[a2[K–n2]]
 ++a2(K–n2+1 **to** K)

= a2(0 **to** n2)
 ++ `reverse` a2(n2 **to** K–n2)
 ++ a2(K–n2 **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 a2, n2. 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.

Write a3 and n3 for values of a and n after loop exit.

By Inv and false while condition 0 ≤ K–(2*n3) ≤ 1.

Then a3(n3 **to** K–n3) has length K–(2*n3) = either 1 or 0. Either way
 `reverse` a3(n3 **to** K–n3) = a3(n3 **to** K–n3).

Therefore, by the invariant,

`reverse` a0(0 **to** K))
 = a3(0 **to** n3) ++ `reverse` a3(n3 **to** K–n3) ++ a3(K–n3 **to** K)
 = a3(0 **to** n3) ++ a3(n3 **to** K–n3) ++ a3(K–n3 **to** K)
 = a3(0 **to** K) = a_r(0 to K).

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

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

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: Using properties of `reverse`, show by induction on length of a that this 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}

Ord 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 or could be all of s or all of t)
- a and b (a ≠ 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≠ 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≠ 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 then show `listcomp` is correct).

Ord 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)

Ord 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: Use index n to 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

$0 \le n \le \min(s.length, t.length)$ (keep n within bounds)
∧ ∀i.**int** $(0 \le i < n \rightarrow s[i] = t[i])$

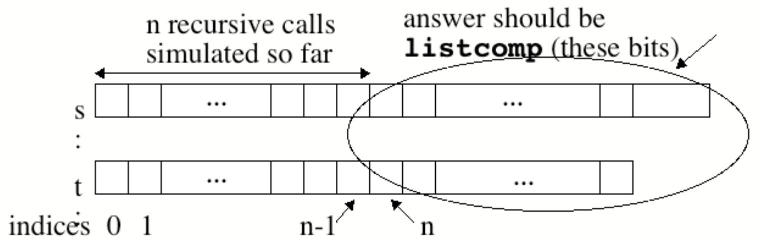
In English: 'No difference found yet'.

BUT: difficult to relate this invariant to Postcondition

(Better) 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) = [ ] (no more elements) then 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.

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≤0≤s.length ∧ 0≤0≤t.length ∧`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 ∧ n<t.length ∧ s[n]≠ 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 n≤t.length, hence n=s.length=t.length by Case: (i)+(ii).

(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], [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. In the finalisation it is known that ¬exhausted(s,n) and ¬exhausted(t,n), so by Post of exhausted n<s.length and n<t.length.

5) *Invariant is maintained*: There are 2 parts to check. Let n1 be the value of n at the start of the loop code and n2 the value at end. Since the while condition holds, then n1<min(s.length, t.length) and s[n1]=t[n1]. By the code n2 = n1+1, which guarantees that n2≤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)

RTS listcomp s0 t0 = listcomp s(n2 **to** s.length) t(n2 **to** t.length)

We show this by showing listcomp s(n1 **to** s.length) t(n1 **to** t.length)
= listcomp s(n2 **to** s.length) t(n2 **to** t.length)

LHS = 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(n2 **to** s.length) t(n2 **to** t.length) = RHS
all using the Haskell code for `listcomp`.

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.

We see that 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

```
Ord 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

<==> 0≤0≤t.length <==> true by arithmetic and length property.

PROOF THAT JAVA ISIN IS CORRECT (2)

***Postcondition is achieved*:**

At exit 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

`isin` x0 t0 (result we want) = `isin` x t(i to t.length) (by Inv)

= `isin` x t(t.length to t.length) (substitute for i)

= `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 and i2 be value at the iteration end.

(No need for x0 and t0 as no changes made to either.)

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

Required to show invariant holds at end of loop (i2=i1+1 by code):

`isin` x t = `isin` x t(i2 to t.length) ∧ 0≤i2≤t.length

0≤i1<t.length (by Inv and loop test)

<==> 1≤i1+1≤t.length <==> 1≤i2≤t.length ==> 0≤i2≤t.length

`isin` x t = `isin` x t(i1 to t.length) (by Invariant)
= `isin` x t[i1]:t(i1+1 to t.length) (since i1<t.length)
= `isin` x t(i1+1 to t.length) (by Haskell x ≠ t[i1])
= `isin` x t(i2 to t.length), so the invariant is re-established.

Proof that jIsIn is correct

Remaining proofs required to show that jIsIn is correct.

Postcondition set up: See slide 33. 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 would need to code using if-statements inside the while loop and employ an additional boolean variable, such as "finished" (or use some statement such as break if the language provides it).

Variant decreases:

as i increases, t.length-i decreases.

By Inv. i ≤ t.length, so the variant ≥0 and 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 more abstract list operations may sometimes enable the algorithm to be checked for correctness more easily (although in this case the proof doesn't seem too hard).

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: DATE (HASKELL)

This example is taken from an earlier tutorial on invariants

--pre: $1 \le d0 \wedge y0 \ge 1900$

--post: $(d0=\sum(i=y0,\mathbf{r_y}-1)(days(i))+\mathbf{r_d})$

-- $\wedge\ 1 \le \mathbf{r_d} \le days(\mathbf{r_y})$ where $(\mathbf{r_d},\mathbf{r_y})$=date d0 y0

```
date d y
    | d>days(y) = date (d-days(y)) (y+1)
    | otherwise = (d,y)
```

`days(y)` is a helper function and returns the number of days in y: eg

```
days y
    | (y `mod` 4)==0    = 366
    | otherwise = 365
```

See slide 40 for verification of `date` by induction.

ANOTHER EXAMPLE: DATE (JAVA)

```
ReturnPair jDate(int d, int y) {
  // Pre:   d0≥1 ∧ y0 ≥ 1900
  // Post: (r_day, r_year ) = (date d0 y0)
while  (d > jDays(y)) {
     // Inv.: (date d0 y0 = date d y) ∧ d≥1 ∧ y≥1900
     // Variant: d
     d = d-jDays(y); y++;  }
  return new ReturnPair(d, y);
  }
```

The Java helper function jDays and the class ReturnPair are assumed.

```
class ReturnPair{ public final int day, year;
    public ReturnPair(int d, int y){day=d; year=y;} }
```

See slide 41 for verification of jDate. Note the Inv. includes the precondition for d and y.

Proof that (Haskell) `date` is correct.

The proof is by induction on d. Let $1 \le d$ for arbitrary d.

Assume as IH: For all $d' < d$

$\forall y{:}int(1 \le d' \wedge y \ge 1900 \rightarrow (d' = \sum(i=y, \mathbf{r'}_y - 1)(days(i)) + \mathbf{r'}_d) \wedge 1 \le \mathbf{r'}_d \le days(\mathbf{r'}_y))$,

where $(\mathbf{r'}_d, \mathbf{r'}_y)$ = `date d' y`.

To show: $\forall y{:}int(y \ge 1900 \rightarrow (d = \sum(i=y, \mathbf{r}_y - 1)(days(i)) + \mathbf{r}_d) \wedge 1 \le \mathbf{r}_d \le days(\mathbf{r}_y))$,

where $(\mathbf{r}_d, \mathbf{r}_y)$ = `date d y`. Let y be arbitrary and satisfy $y \ge 1900$.

Case 1: $d \le days(y)$. Then $\mathbf{r}_d = d$ and $\mathbf{r}_y = y$. After substitution,
must show $(d = \sum(i=y, y-1)(days(i)) + d) \wedge 1 \le d \le days(y)$.
First conjunct reduces to $d = 0+d$, true by arithmetic.
Also $1 \le d$ by assumption and $d \le days(y)$ by Case.

Case 2: $d > days(y)$.
Let $(\mathbf{r''}_d, \mathbf{r''}_y)$ = `date (d-days(y)) y+1`. Then $(\mathbf{r}_d, \mathbf{r}_y) = (\mathbf{r''}_d, \mathbf{r''}_y)$.

Must show $(d = \sum(i=y, \mathbf{r}_y - 1)(days(i)) + \mathbf{r}_d) \wedge 1 \le \mathbf{r}_d \le days(\mathbf{r}_y))$ (*).

First note that the arguments for the call `date (d-days(y)) (y+1)` satisfy its precondition:

$d - days(y) > 0$ (by Case) so $d - days(y) \ge 1$, and $y+1 > y \ge 1900$ by assumption.

Also, $d - days(y) < d$, so IH is applicable giving:

$\forall y{:}int(1 \le d - days(y) \wedge y+1 \ge 1900 \rightarrow (d - days(y) = \sum(i=y+1, \mathbf{r''}_y - 1)(days(i)) + \mathbf{r''}_d)$
$\wedge 1 \le \mathbf{r''}_d \le days(\mathbf{r''}_y))$.

Hence can derive $(d - days(y) = \sum(i=y+1, \mathbf{r''}_y - 1)(days(i)) + \mathbf{r''}_d) \wedge 1 \le \mathbf{r''}_d \le days(\mathbf{r''}_y)$

$\Longleftrightarrow (d = \sum(i=y, \mathbf{r''}_y - 1)(days(i)) + \mathbf{r''}_d) \wedge 1 \le \mathbf{r''}_d \le days(\mathbf{r''}_y)$.

Substitute $\mathbf{r}_d = \mathbf{r''}_d$ and $\mathbf{r}_y = \mathbf{r''}_y$

giving $(d = \sum(i=y, \mathbf{r}_y - 1)(days(i)) + \mathbf{r}_d) \wedge 1 \le \mathbf{r}_d \le days(\mathbf{r}_y) \Longleftrightarrow$ (*).

Proof that jDate is correct.

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

Postcondition is set up by jDate
On exit from the while loop d>jDays(y) is false ==> d≤jDays(y).
By Invariant d≥1, so second part of Post is attained.
RTS also that $(\mathbf{r}_{day}, \mathbf{r}_{year})$=`date` d0 y0.
`date` d0 y0=`date` d y (by Inv.) = `(d,y)` (by Haskell `date`)
since d≤jDays(y)= $(\mathbf{r}_{day}, \mathbf{r}_{year})$ (by Java jDate).

Invariant is established at start of loop.
Required to show `date` d0 y0 = `date` d y ∧ d≥1 ∧ y≥1900. Know d=d0 and y=y0 by code ==> first conjunct true; second and third conjuncts are true by Precondition.

Invariant is re-established by loop code.
Let d1 and y1 be the values of d and y at start of loop code and d2 y2 be the values at end.

Know (1) d1≥1, y1≥1900 (by Inv) and (2) d1>jDays(y1) by loop test.

Also know: `date` d0 y0 = `date d1` y1.

(Assume that Haskell `days y` and Java jDays(y) compute the same answer (**))

Required to show `date` d0 y0 = `date` d2 y2.

At the end of the loop code d2 = d1-jDays(y1) and y2 = y1+1.

`date d1` y1 = `date (d1-days(y1)) (y1+1)` = `date d2` y2

(use (**) and Haskell code).

Note the precondition of `date d1 y1` holds by (1).

Hence `date d0` y0 = `date d2` y2.

The other parts of Invariant, d2≥1 ∧ y2 ≥ 1900, are true by (1) and (2) .

Loop terminates.
Variant d decreases each iteration since d1-jDays(y1) < d1 by definition of jDays.
Since d is always ≥1 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 (for appropriate initial values).

(2) The function `fTR` with the accumulating parameter *is* tail recursive, so can convert it into a loop (the Java program is jFTR). To obtain correct result from jFTR must initialise the accumulating parameter to the right value at the start.

(3) Prove jFTR correctly implements `fTR`.

(4) Check `f` is correct!

Proof that `fact = factTR 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 (STEP 2)

```
int jFactTR(int n) {
// Pre: n0≥0
// Post: r = fact n0 (or r = factTR 1 n0)
  int m = 1; // m is the accumulator
  while     (n != 0) {
  // Inv: factTR 1 n0 (= 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 3)

The proof that jFactTR is correct follows exactly the same pattern as used to show that jDate is correct. Compare the two proofs to convince yourself this is so.

Postcondition is set up by jfactTR
At loop exit (n != 0) is false ==> n=0.
Required to show **r** = `factTR` 1 n0.
`factTR` 1 n0 = `factTR m n` (by the Inv.) = `factTR` m 0 (substitute for n)
= m (by Haskell `factTR`) = result **r** (by Java jFactTR).

Invariant is established at start of loop: n=n0 and m=1
Required to show (`factTR` 1 n0 = `factTR` m n) ∧ n≥0 <==>
(`factTR` 1 n0 = `factTR` 1 n0) ∧ n0≥0 <==> True by precondition

Invariant is re-established by loop code
Let m1/m2 and n1/n2 be the values of m and n at start and end of an arbitrary loop iteration.

The Invariant gives n1≥0 and `factTR` 1 n0 = `factTR` m1 n1
and the loop test gives n1 !=0.

By the loop code m2 = m1*n1 and n2 = n1-1.

RTS n2≥0 ∧ `factTR` 1 n0 = `factTR` m2 n2.

n1 !=0 ∧ n1≥0 ==> n1>0 (*) <==> n1-1≥0 <==> n2≥0.

By the Haskell code `factTR` m1 n1 = `factTR` (m1*n1) (n1-1) (since n1 > 0)
= `factTR` m2 n2.

The precondition of `factTR` m1 n1 holds by the Invariant.
Hence `factTR` 1 n0 = `factTR` m2 n2.

Loop terminates
Variant n decreases on each iteration since n1-1<n1. Since n is always ≥0 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 list properties 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
```

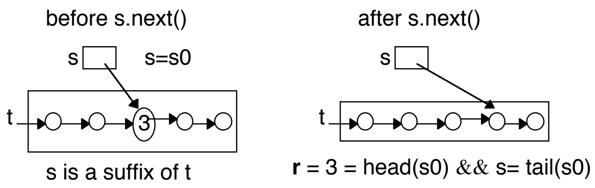
isin IN JAVA USING ArrayList

```
boolean jIsIn2(int x, ArrayList<Integer> t)   {
// Pre:   none          //t is declared as a List of Integers
// Post:  r = isin x0 t0  ∧ deep struct. t = deep struct. t0
Iterator<Integer> s = t.iterator();
while  (s.hasNext() && x != (s.next()).intValue())
//get value of head of t; Java knows it is an Integer; side effect
// Invariant: (isin x0 t0 = isin x s) ∧ "t = t0"
// Variant: no. remaining elements of t to examine
return s.hasNext();  }
```

Proof that jIsIn2 *is correct* —Assume x=x0 and deep structure of t = deep structure of t0 throughout. This might be represented by "t=t0". (See comment below.)

Reasoning about jIsIn2 is difficult. The invariant has to capture that the iterator s is moving along the list t, and that the deep structure of t, ie the elements in t, are not changed. Moreover, the s.next() method call has the side effect of moving s along one element of t to find the next element in t. I am not sure at the moment exactly how best to tie the position of s into the list t. Perhaps can capture it by "s=suffix of t", and the postcondition of s.next() as "s=tail(s0) && r=head(s0)".

Let's see. First though, here is a picture to illustrate these things.



Invariant: x=x0 ∧ "t=t0" ∧ t=t0 ∧
(I1): s is a suffix of t0 ∧
(I2): `isin x0 t0= isin x s.`
We'll assume x=x0, t=t0 and "t=t0" throughout and omit it from the reasoning to save clutter.

1) *Invariant set up by initial code*: s=t0 (impicit in iterator initialisation)

(I1) <==> t0 is a suffix of t0 <==> True

(I2) <==> `isin x0 t0=isin x t0` <==> `True`

2) (*Invariant + false while condition + final code*) *implies postcondition:*
Successful loop exit implies either s has completed iterating over t (case 1)
or (s has not completed iterating over t ∧ x0=s.head()) (case 2)
The invariant says the required result **r** (= `isin x0 t0) =isin x0 s.`

Let s3 be value of s when the call s.next().intValue() is made;
<u>Case 1</u> <u>(s has finished):</u> s3=[]; `isin x0 []=False,`
so the actual result given by the code (= s.hasNext()) = false is correct.
<u>Case 2</u> <u>(x0=s3.head()) and s4=tail(s3):</u> `isin x0 s3 =True,`
so the actual result given by the Java code (=s.hasNext()) = true is correct.

3) *Variant decreases*: The number of elements still to be iterated over reduces by 1 on each iteration by side effect of next() method of the iterator. Hence the loop must stop as the length of a non-empty list cannot decrease forever.

4) *Invariant is maintained*:
Let s1 be the value of s when the loop test is made, and s2 be the value of s after the call to s.next() – i.e. at the end of an arbitrary loop iteration.

A true loop test means s1 not finished and x≠value of head(s1). Given

(I1): s1 is a suffix of t

(I2): `isin x0 s1=isin x0 t0.`

RTS
(I1'): s2 is a suffix of t0

(I2'): `isin x0 s2= isin x0 t0.`

(I1') is true by definition of suffix and tail given (I1).
(I2'): `LHS = isin x0 tail(s1) = isin x0 s1` by the Haskell code taken from right to left (since x≠ value of head(s1)) = `isin x0 t0` by the invariant = RHS.

5) *Array accesses*.
As the structure is an ArrayList must check access: element s1.next() is accessed and exists in the loop as the loop is non-empty by the successful call to s1.hasNext() in the while condition.

Note: the deep structure of t doesn't change - the list iterator s (for t) simply moves over the list in order. The method next() has the side effect of doing this.

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 52 the method jIsIn2 is defined for an ArrayList argument.

A recursive version is the following:

```
boolean jIsIn3(int x, ArrayList<Integer> t) {
ArrayList<Integer> s=t;
if (s.isEmpty()) return false;
if (x == (s.get(0)).intValue()) return true;
s=s.subList(1,s.size()); return jIsIn3(x, s); }
```

You still have to reason that s remains a suffix of t0, etc.
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 generic Lists as in:

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

If you're interested in reasoning about linked structures, see Sophia, who is much more expert than me on that topic!

GENERAL METHOD USING HASKELL

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 Haskell function `H` is tail recursive the invariant is H args0 = H currentArgs (eg isIn was like this). If not the invariant is HTR initialArgs = HTR currentArgs (eg fact was like this). Initial Args are args0 and appropriate initial accumulator value.

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

SUMMARY

• A useful notation for arrays-as-lists was introduced

a-as-a-list = a(0 to a.length)

• Haskell functions can be used to specify Java methods

• Tail recursive Haskell functions can be systematically converted into while loop Java methods

APPENDIX – FOR LOOPS

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

public Matrix(int n){Pre??
    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 precondition on Matrix will ensure the invariant 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, although the iterations cannot be executed in parallel, they 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;`

(We'll see examples next week.)

PROVING THE POSTCONDITION HOLDS

For case i):

```
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 ≤ I,J<size. Show that, at the end, m[i][j] = 0.

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

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

PROVING THE POSTCONDITION HOLDS (2)

For Case ii)

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

Why can't the operations occur in parallel here?

It is <u>safest</u> 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.

For this example, it's possible to reason similar to Case 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. (eg see an example next week.)

A TYPICAL FOR LOOP?

```
boolean neg1 (int [] a) {
//post: r ↔ at least one negative integer is in a ∧ 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 I^{th} 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 I^{th} 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++;    }  //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 <==>5-i≥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=Sum(k)(k=1 to i-1) ∧ 1≤i≤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 implies 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≤1≤6.

The invariant is maintained: call the values of i and s at the start of the loop i1 and s1. The requirement is s= Sum(k)(k=1 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 $1 \le i1+1 \le 6$ as the true while condition gives $i1 \le 5$. More formally, let i2 and s2 be the values of i and s at the end of the loop. Given:

$1 \le i1 \le 6 \wedge i1 \le 5 \wedge$ s1=Sum(k)(k=1 to i1-1) (*invariant before the code* and *loop test*)
i2=i1+1 $\wedge$ s2=s1+i1 (*effect of code*)

Then RTS: $1 \le i2 \le 6 \wedge$ s2=Sum(k)(k=1 to i2-1) (*after the code*)

First, $1 \le i1 \le 6 \wedge i1 \le 5 \iff 1 \le i1 \le 5 \iff 2 \le i1+1 \le 6 \iff 2 \le i2 \le 6 \implies 1 \le i2 \le 6$
Next, (s1=Sum(k)(k=1 to i1-1)) <==> (s1+i1=Sum(k)(k=1 to i1-1)+i1) (add i1 to both sides) <==> (s1+i1 = Sum(k)(k=1 to i1)) <==> s2 =Sum(k)(k=1 to i2-1).

The result is correct: at the end the false while condition (loop exited) tells us i>5 and the invariant that $i \le 6 \wedge$ s=Sum(k)(k=1 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≥0 is equivalent to 6≥i, which is included in the invariant.

Exercise: Formalise neg3 as a while loop with suitable invariant.