Discrete aths

2. Loop Invariants

- Objectives
  - to show the use of induction for proving properties of code involving loops
    - use induction to prove that functions work
  - introduce pre- and post- conditions,
    loop termination

Overview

1. What is a Loop Invariant?

2. Three simple examples
   – they involve while loops

3. Selection Sort

4. Further Information

1. What is a Loop Invariant?

- A *loop invariant* is an *inductive* statement which says something which is always true about a program loop.

- Loop invariants are useful for:
  - code specification
  - debugging

continued

- A loop invariant is typically written as an inductive statement S(n), where n is some changing element of the loop. For example:
  - the loop counter/index
  - a loop variable which changes on each iteration

2.1. Example 1

Problem: does this function work?

```
int square(int val)
{
  int result = 0;
  int counter = 0;
  while (counter < val) {
    result += val;
    counter++;
  }
  return result;
}
```

Pre- and Post-conditions

- We assume that `val` is a positive integer
  - the *precondition* for this function
- `square()` always returns $\mathtt{val}^2$
  - the *postcondition*
  - we will use the loop invariant to show that this is true

The Loop Invariant

- To make the proof clearer, let `counter`$_n$ and `result`$_n$ be the values of `counter` and `result` after passing round the loop `n` times.

- Loop invariant S(n):

$$result_n = val * counter_n$$

  – is this true in the loop for all n >= 0?

Basis

- S(0) is when the loop has not yet been executed.
  - $result_0 = counter_0 = 0$

- So:

$$result_0 = val * counter_0$$

  - which means that S(0) is true.

Induction

- We assume that S(k) is true for some k >= 0, which means that:

  $result_k = val * counter_k$ (1)

- After one more pass through the loop:

  $result_{k+1} = result_k + val$ **(2)**

  $counter_{k+1} = counter_k + 1$ (3)

continued

- Substitute the rhs of (1) for the 1st operand of the rhs of (2):

$$\begin{aligned} result_{k+1} &= (val * counter_k) + val \\ &= val * (counter_k + 1) \\ &= val * counter_{k+1} \qquad \text{(by using (3))} \end{aligned}$$

  – this is S(k+1), which is therefore true.

continued

- For the loop, we now know:
  - S(0) is true
  - S(k) --> S(k+1) is true

- That means S(n) is true for all n >= 0
  - for all n >= 0, the loop invariant is true:

$$result_n = val * counter_n$$

Termination

- The loop does actually terminate, since `counter` is increasing, and will reach `val`.

- At loop termination (and function return):

  $\texttt{counter}_n = \texttt{val}$

- So in S(n):

  $\texttt{result}_n = \texttt{val * val}$
  $= \texttt{val}^2$

- So the *postcondition is true*.

The function works!!

2.2. Example 2

Problem: does this function work?

```
int exp(int b, int m)
// return b^m
{
  int res = 1;
  while (m > 0) {
    res = res * b;
    m = m - 1;
  }
  return res;
}
```

Pre- and Post-conditions

- We assume `b` and `m` are non-negative integers
  - the *preconditions* for this function
- `exp()` always returns b^m
  - the *postcondition*
  - we will use the loop invariant to show that this is true

The Loop Invariant

- To clarify the proof, let $\texttt{res}_n$ and $\texttt{m}_n$ be the values of `res` and `m` after passing round the loop `n` times.

- Loop invariant S(n):

Inventing this is the hardest part.

$$res_n * b^{m_n} = b^m$$

  – is this true in the loop for all n >= 0?

Basis

- S(0) is when the loop has not yet been executed.
  - $res_0 = 1$; $m_0 = m$
- So:

$$1 * b^m = b^m$$

  - which means that S(0) is true.

Induction

- We assume that S(k) is true for some k >= 0, which means that:

$$res_k * b^{m_k} = b^m \qquad (1)$$

- After one more pass through the loop:

$$res_{k+1} = res_k * b \qquad (2)$$

$$m_{k+1} = m_k - 1 \qquad (3)$$

continued

- Rearrange the equations:

$res_k = res_{k+1} / b$ (2')

$m_k = m_{k+1} + 1$ (3')

- Substitute the right hand sides of (2') and (3') into (1):

$(res_{k+1} / b) * b^{(m_{k+1} + 1)} = b^m$

which is

$res_{k+1} * b^{(m_{k+1} + 1 - 1)} = b^m$

which is

$res_{k+1} * b^{m_{k+1}} = b^m$

continued

- S(k+1) is:

$$res_{k+1} * b^{m_{k+1}} = b^m \qquad (4)$$

- So we have shown S(k+1) is true by using S(k).

continued

- For the loop, we now know:
  - S(0) is true
  - S(k) --> S(k+1) is true

- That means S(n) is true for all n >= 0
  - for all n >= 0, the loop invariant is true:

$$res_n * b^{m_n} = b^m$$

Termination

- The loop does actually terminate, since `m` is decreasing, and will reach `0`.

- At loop termination (and function return):

  $m_n = 0$

- So in S(n):

  $res_n * b^0 = b^m$

  so $res_n = b^m$

- So the *postcondition is true*.

The function works!!

2.3. Example 3

Problem: does this function work?

```
int factorial(int num)
{
  int i = 2;
  int fact = 1;
  while (i <= num) {
    fact = fact * i;
    i++;
  }
  return fact;
}
```

Pre- and Post-conditions

- We assume `num` is a positive integer
  - the *precondition* for this function
- `factorial()` always returns `num!`
  - the *postcondition*
  - we will use the loop invariant to show that this is true

The Loop Invariant

- To clarify the proof, let $\texttt{fact}_n$ and $\texttt{i}_n$ be the values of `fact` and `i` after passing round the loop `n` times.

- Loop invariant S(n):

  $$fact_n = (i_n - 1)!$$

  – is this true in the loop for all n >= 0?

Basis

- S(0) is when the loop has not yet been executed.
  - $i_0 = 2$; $fact_0 = 1$
- So: $fact_0 = (i_0 - 1)!$

  $1 = (2 - 1)!$

  $1 = 1$

which means that S(0) is true.

Induction

- We assume that S(k) is true for some k >= 0, which means that:

$$fact_k = (i_k - 1)! \qquad (1)$$

- After one more pass through the loop:

$$fact_{k+1} = fact_k * i_k \qquad (2)$$

$$i_{k+1} = i_k + 1 \qquad (3)$$

continued

- Substitute the rhs of (1) for the 1st operand of the rhs of (2):

$$fact_{k+1} = (i_k - 1)! * i_k$$
$$= (i_k)!$$
$$= (i_{k+1} - 1)! \qquad \text{(by using (3))}$$

  – this is S(k+1), which is therefore true.

continued

- For the loop, we now know:
  - S(0) is true
  - S(k) --> S(k+1) is true

- That means S(n) is true for all n >= 0
  - for all n >= 0, the loop invariant is true:

$$fact_n = (i_n - 1)!$$

Termination

- The loop does actually terminate, since `i` is increasing, and will reach `num+1`.

- At loop termination (and function return):

  $\texttt{i}_n = \texttt{num+1}$

- So in S(n):
  - $fact_n = (i_n - 1)!$
    $= (num + 1 - 1)! \;=\; num!$
- So the *postcondition is true*.

The function works!!

3. Selection Sort

```
void selectionSort(int A[], int num)
{
  int i, j, small, temp;
  for (i=0; i < num-1; i++) {
    small = i;
    for( j= i+1; j < num; j++)
      if (A[j] < A[small])
        small = j;
    temp = A[small];
    A[small] = A[i];
    A[i] = temp;
 }
}
```

put index of smallest value in A[i .. num-1] into small

swap A[small] and A[i]

`selectionSort(A, 4);`

Execution Highlights

Start 1st iteration:

A	0	1	2	3
	25	30	20	10

Start 2nd iteration:

A	0	1	2	3
	10	30	20	25

sorted

Start 3rd iteration:

A	0	1	2	3
	10	20	30	25

sorted

Start 4th iteration:

A	0	1	2	3
	10	20	25	30

sorted

Pre- and Post-conditions

- We assume `num` is a positive integer, which contains the number of elements in the array
  - the *precondition* for this function
- `selectionSort()` sorts the array into ascending order
  - the *postcondition*
  - we will use the loop invariant to show that this is true

3.1. Consider the Inner Loop

```
small = i;
for( j= i+1; j < num; j++)
  if (A[j] < A[small])
    small = j;
```

Put index of smallest value
in A[i .. num-1] into small

Pre- and Post-conditions

- We assume `i` is a positive integer, which is an index into the array
  - the *precondition* for this part of the code
- This code finds the index of the smallest value in `A[i..num-1]`
  - the *postcondition*
  - we will use the loop invariant to show that this is true

The Inner Loop Invariant

- To clarify the proof, let $\texttt{small}_n$ and $\texttt{j}_n$ be the values of `small` and `j` after passing round the loop `n` times.

- Loop invariant S(n):

  $small_n$ is the index of the smallest of $A[i .. j_n\text{-}1]$

  – is this true in the loop for all n >= 0?

Execution Highlights

Start 1st pass: small 0

A	0	1	2	3
	25	30	20	10
		j		

Start 2nd pass: small 0

A	0	1	2	3
	25	30	20	10
			j	

Start 3rd pass: small 2

A	0	1	2	3
	25	30	20	10
				j

Start 4th pass: small 3

A	0	1	2	3	
	25	30	20	10	
					j

loop ends

Basis

- S(0) is when the loop has not yet been executed.
  - $small_0 = i$; $j_0 = i+1$
- So: $small_0$ is the index of the smallest of $A[i .. j_0-1]$

  which is A[i..(i+1-1)], or A[i..i], or A[i]
- That means that S(0) is true.

Induction

- We assume that S(k) is true for some k >= 0, which means that:

  $small_k$ is the index of the smallest of $A[i \,..\, j_k -1]$

- The pass through the loop involves two possible branches because of the if.

continued

- *Case 1*. If A[j_k] is *not* smaller than the smallest of A[i.. j_k -1]
  - then $\texttt{small}_{k+1}$ = $\texttt{small}_k$ (no change)

- *Case 2*. If A[j_k] is smaller than the smallest of A[i.. j_k -1]
  - then $\texttt{small}_{k+1}$ = j_k

continued

- At the end of the pass:
  - $small_{k+1}$ is the index of the smallest of $A[i..j_k]$
  - $j_{k+1} = j_k + 1$

- S(k+1) is:
  - $small_{k+1}$ is the index of the smallest of $A[i .. j_{k+1}-1]$
- So S(k+1) is true.

continued

- For the inner loop, we now know:
  - S(0) is true
  - S(k) --> S(k+1) is true

- That means S(n) is true for all n >= 0
  - for all n >= 0, the inner loop invariant is true:
    $small_n$ is the index of the smallest
    of A[i .. j_n -1]

Termination

- The loop does actually terminate, since `j` is increasing, and will reach `num`.

- At loop termination:

  $j_n = num$

- So in S(n):
  - $small_n$ is the index of the smallest of A[i .. j_n-1], or A[i..num-1]

- So the *postcondition is true*.

The inner loop is proved.

3.2 The Outer Loop

```
void selectionSort(int A[], int num)
{
  int i, j, small, temp;
  for (i=0; i < num-1; i++) {
    small = i;
     for( j= i+1; j < num; j++)
       if (A[j] < A[small])
          small = j;
     temp = A[small];
     A[small] = A[i];
     A[i] = temp;
 }
}
```

proved

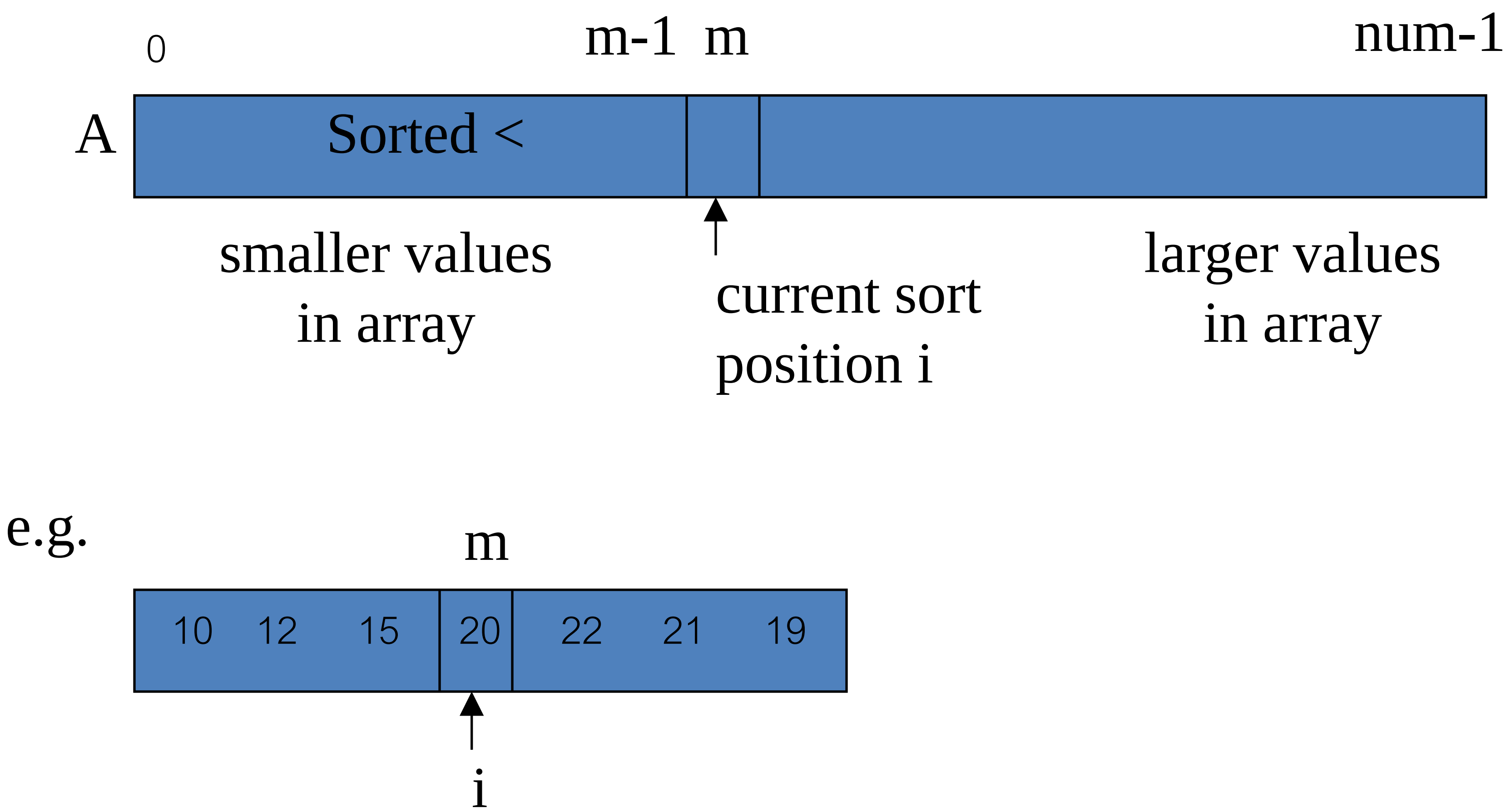
The Outer Loop Invariant

- To clarify the proof, let i_m be the value of `i` after passing round the loop `m` times.

- Informal loop invariant T(m):

  i_m indicates that we have selected m of the smallest elements and sorted them at the beginning of the array.

Graphically



More Formally

- Loop invariant T(m):
  - a) A[0..i_m-1] are in sorted order;
  - b) All of A[i_m..num-1] are >= all of A[0..i_m-1]

Basis

- T(0) is when the loop has not yet been executed. $i_0=0$
  - a) Range is A[0..-1], which means no elements.
  - b) Range is A[0..num-1] which means everything > A[0..-1]
- So T(0) is true.

Induction

- We assume that T(k) is true for some k >= 0, which means that:
  - a) $A[0..i_k-1]$ are in sorted order;
  - b) All of $A[i_k..num-1]$ are >= all of $A[0..i_k-1]$

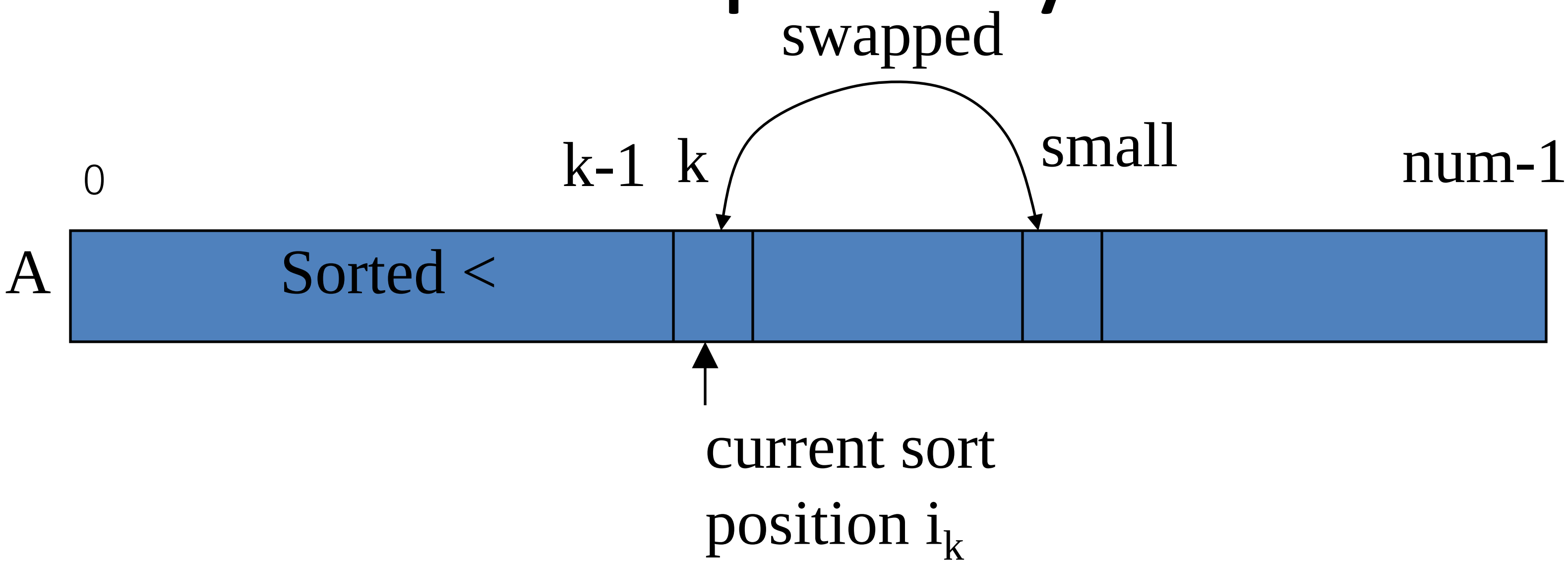
continued

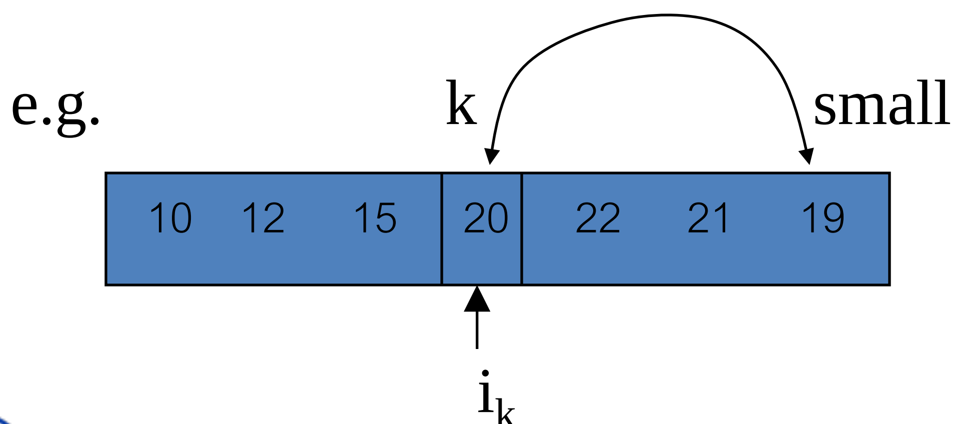
- By the end of the inner loop, we know that A[small] is the smallest element in A[i_k..num-1]
  - the *postcondition of the inner loop*

- The next three lines swap A[small] and A[i_k]
  - now A[i_k] contains the smallest element

continued

Graphically





continued

- By the end of the outer loop, we know:
  - a) $A[0..i_k]$ are in sorted order;
  - b) All of $A[i_k+1..num-1]$ are >= all of $A[0..i_k]$
  - $i_{k+1} = i_k + 1$
- This shows that $T(k+1)$ is true.

continued

- For the outer loop, we now know:
  - T(0) is true
  - T(k) --> T(k+1) is true

- That means T(m) is true for all m >= 0
  - a) A[0..i_m-1] are in sorted order;

  - b) All of A[i_m..num-1] are >= all of A[0..i_m-1]

Termination

- The outer loop does actually terminate, since `i` is increasing, and will reach `num-1`.

- At loop termination (and function return):

  $\texttt{i}_{\texttt{m}} = \texttt{num-1}$

continued

- So in T(m):
  - a) A[0..i_m-1] are in sorted order
  - so A[0..num-1-1] is sorted
  - so A[0..num-2] is sorted

  - all the array up to the last element is sorted

continued

– b) All of A[i_m..num-1] are >= all of A[0..i_m-1]

– so, A[num-1..num-1] >= all A[0..num-1-1]

– so, A[num-1] >= all A[0..num-2]

– so the last element is bigger than all the other array elements

- So the function does sort the array into ascending order: *the postcondition is true*

4. Further Information

- *Discrete Mathematics Structures in Computer Science*
  B. Kolman & R. C. Busby
  Prentice-Hall, 1987, 2nd ed.
  Section 1.6