Slide 349

Overview

We shall be using the example of

Quicksort

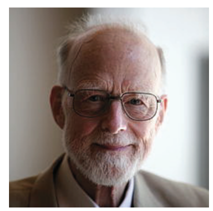
to demonstrate:

1. formal specifications
2. the use of midconditions in developing algorithms
3. the divide and conquer approach to problem solving
4. the use of auxiliary functions
5. the proof obligations involved in non-trivial recursive methods

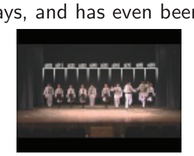
Slide 350

Background

- Tony Hoare proposed Quicksort in 1962.
- He proposed Reasoning about Imperative Programs in 1969.
- He received the ACM Turing award in 1980.
- He became a fellow of the Royal Society in 1982.
- He was knighted in 2000.



Quicksort has been implemented in thousands of languages, by millions of people, visualised in a multitude of ways, and has even been danced in Hungarian:



http://www.youtube.com/watch?v=ywWBy6J5gz8

After serving at many institutions (including the University of Belfast, Oxford University, Microsoft Research) Tony Hoare has now retired. However, he is still going to work every day and working on algebraic models of concurrency.

Slide 351

Sorting - the Task

Given:

- an array `a` of characters
- bounds `F` and `T`

We want to rearrange the array so that the elements between `F` and `T` are sorted in ascending (alphabetical) order.

The contents of `a` outside of the range `F`...`T` should not be modified.

Note: The sorting task can actually be defined for an array of any type that has a less-then-or-equal relation $\leq$.

Slide 352

Quicksort – The Idea

The basic Quicksort algorithm can be described as follows:

1. Choose an element in the list - this element serves as the pivot.
2. Partition the array of elements into two sets - those less than the pivot and those greater than the pivot
3. Repeat steps 1 and 2 on each of the two resulting partitions until each set has one or less elements.

Slide 353

Notation - Array Concatenation

We define the concatenation of two arrays $\mathtt{a} : \mathtt{b}$ such that:

$$\begin{aligned} \mathtt{a} \approx \mathtt{b} : \mathtt{c} \quad \longleftrightarrow \quad & \mathtt{a.length} = \mathtt{b.length} + \mathtt{c.length} \\ & \land \forall i \in [0..\mathtt{b.length})[\,\mathtt{a}[i] = \mathtt{b}[i]\,] \\ & \land \forall i \in [0..\mathtt{c.length})[\,\mathtt{a}[i + \mathtt{b.length}] = \mathtt{c}[i]\,] \end{aligned}$$

Using deep equality, array slices and concatenation we can describe many interesting properties of arrays. For example:

$$\mathtt{a} \approx \mathtt{a}[0..1) : \mathtt{b} : \mathtt{a}[\mathtt{b.length}+1..\mathtt{a.length})$$

says that the contents of the array `a` from index 1 to index `b.length` is the same as the array b. It does not say anything about the other elements of the array `a`.

We will see later that this notation can be very useful for describing changes to just part of an array, or even for specifying that parts of an array are unmodified.

Slide 354

Part III: Reasoning Case Studies — Quicksort

Predicates - reminder

We define some useful predicates over arrays. For arrays `a` and `b`, natural numbers i and j and arbitrary array element v:

- $\mathit{NrOccurs}(\mathtt{a}, v) \;\equiv\; |\{k \mid \mathtt{a}[k] = v\}|$
- $\mathtt{a} \sim \mathtt{b} \;\longleftrightarrow\; \forall v[\, \mathit{NrOccurs}(\mathtt{a}, v) = \mathit{NrOccurs}(\mathtt{b}, v)\,]$
- $\mathit{Swapped}(\mathtt{a}, \mathtt{b}, i, j) \;\longleftrightarrow\; \mathtt{a.length} = \mathtt{b.length}$
  $\qquad \wedge\, i, j \in [0..\mathtt{a.length})$
  $\qquad \wedge\, \mathtt{b}[i] = \mathtt{a}[j]$
  $\qquad \wedge\, \mathtt{b}[j] = \mathtt{a}[i]$
  $\qquad \wedge\, \forall k \in [0..\mathtt{a.length})\backslash\{i, j\}[\,\mathtt{a}[k] = \mathtt{b}[k]\,]$

Drossopoulou & Wheelhouse (DoC) — Reasoning about Programs — 354 / 395

Using the notation developed on the previous slide an alternative representation of the *Swapped* predicate can be given as:

$$\begin{aligned} \mathit{Swapped}(\mathtt{a}, \mathtt{b}, i, j) \;\longleftrightarrow\;& \mathtt{a.length} = \mathtt{b.length} \\ &\wedge\, i, j \in [0..\mathtt{a.length}) \\ &\wedge\, i = j \longrightarrow \mathtt{b} \approx \mathtt{a} \\ &\wedge\, i < j \longrightarrow \mathtt{b} \approx \mathtt{a}[0..i) : \mathtt{a}[j] : \mathtt{a}[i..j) : \mathtt{a}[i] : \mathtt{a}[j..\mathtt{a.length}) \\ &\wedge\, j < i \longrightarrow \mathtt{b} \approx \mathtt{a}[0..j) : \mathtt{a}[i] : \mathtt{a}[j..i) : \mathtt{a}[j] : \mathtt{a}[i..\mathtt{a.length}) \end{aligned}$$

(note that some of the array slices in these last two conjuncts could be empty.)

Slide 355

Lemmas

Throughout this section we will make use of a number of Lemmas that describe useful properties of arrays and their related predicates. The following hold for all arrays a and b and for all integers i, j, k, m and n:

Deep Equality Lemmas:

- $\mathtt{a} \approx \mathtt{b} \longrightarrow \mathtt{b} \approx \mathtt{a}$ (≈**Comm**)
- $\mathtt{a} \approx \mathtt{b} \wedge \mathtt{b} \approx \mathtt{c} \longrightarrow \mathtt{a} \approx \mathtt{c}$ (≈**Trans**)
- $\mathtt{a} \approx \mathtt{b} \longrightarrow \mathtt{a.length} = \mathtt{b.length}$ (≈**Length**)
- $\mathtt{a} \approx \mathtt{b} \longrightarrow \mathtt{a} \sim \mathtt{b}$ (≈**IsPrm**)

Permutation Lemmas:

- $\mathtt{a} \sim \mathtt{b} \longrightarrow \mathtt{b} \sim \mathtt{a}$ (∼**Comm**)
- $\mathtt{a} \sim \mathtt{b} \wedge \mathtt{b} \sim \mathtt{c} \longrightarrow \mathtt{a} \sim \mathtt{c}$ (∼**Trans**)
- $\mathtt{a} \sim \mathtt{b} \longrightarrow \mathtt{a.length} = \mathtt{b.length}$ (∼**Length**)

Slide 356

Lemmas

Swapped Lemmas:

- $\mathit{Swapped}(\mathtt{a}, \mathtt{b}, i, i) \longrightarrow \mathtt{a} \approx \mathtt{b}$ (≈ *Swapped*)
- $\mathit{Swapped}(\mathtt{a}, \mathtt{b}, i, j) \longrightarrow \mathtt{a} \sim \mathtt{b}$ (∼ *Swapped*)
- $\mathit{Swapped}(\mathtt{a}, \mathtt{b}, i, j) \longleftrightarrow \mathit{Swapped}(\mathtt{a}, \mathtt{b}, i, j)$ (*SwappedSymm*)

Array Range Lemmas:

- $\mathtt{b} \approx \mathtt{a}[0..i) : \mathtt{b}[i..j) : \mathtt{a}[j..\mathtt{a.length}) \wedge m \leq i \wedge j \leq n$
  $\longrightarrow \mathtt{b} \approx \mathtt{a}[0..m) : \mathtt{b}[m..n) : \mathtt{a}[n..\mathtt{a.length})$ (**RngWeak**)
- $\mathtt{b} \approx \mathtt{a}[0..i) : \mathtt{b}[i..j) : \mathtt{a}[j..\mathtt{a.length}) \wedge \mathtt{a}[i..j) \sim \mathtt{b}[i..j)$
  $\longrightarrow \mathtt{a} \sim \mathtt{b}$ (**RngPrm**)
- $\mathtt{a} \approx \mathtt{b} \wedge \mathtt{c} \approx \mathtt{c}[0..i) : \mathtt{a}[i..j) : \mathtt{c}[j..\mathtt{c.length})$
  $\longrightarrow \mathtt{c} \approx \mathtt{c}[0..i) : \mathtt{b}[i..j) : \mathtt{c}[j..\mathtt{c.length})$ (**RngSwap**)

In **RngWeak** the premise says a and b are identical except for the range $[i..j)$, and the conclusion says that a and b are identical except for the range $[m..n)$. The lemma holds, since the range $[m..n)$ includes the range $[i..j)$.

Here is a proof of **RngWeak**.

Given:
1. $\mathtt{b} \approx \mathtt{a}[0..i) : \mathtt{b}[i..j) : \mathtt{a}[j..\mathtt{a.length})$
2. $m \leq i \wedge j \leq n$

To Show:
α. $\mathtt{b} \approx \mathtt{a}[0..m) : \mathtt{b}[m..n) : \mathtt{a}[n..\mathtt{a.length})$

Proof:
From 1., and the definition of concatenation we have:
3. b.length=i+(j-i)+(a.length-j)
4. $\forall k \in [0..i).\ \mathtt{b}[k] = \mathtt{a}[k]$
5. $\forall k \in [j..\mathtt{a.length}).\ \mathtt{b}[k] = \mathtt{a}[k]$
From 3. we obtain by arithmetic:
6. b.length= a.length
From 4., and 2., (since $m \leq i \wedge k \leq m \quad \longrightarrow \quad k \leq i$) we obtain:
7. $\forall k \in [0..m).\ \mathtt{b}[k] = \mathtt{a}[k]$
Similarly, from 5., and 2., we obtain:
8. $\forall k \in [n..\mathtt{a.length}).\ \mathtt{b}[k] = \mathtt{a}[k]$
From 6., 7., and 8., and definition of concatenation (:), we obtain α

Slide 357

Quicksort – Formal Specification

```
void quicksort(char[] a, int F, int T)
//PRE: a ≠ null ∧ 0 ≤ F ≤ T ≤ a.length
//POST: a ≈ a0[0..F) : a[F..T) : a0[T..a.length)
        ∧ a ∼ a0 ∧ Sorted(a[F..T))
```

We do not need to distinguish between $\mathtt{F}$ and $\mathtt{F}_0$, nor between $\mathtt{T}$ and $\mathtt{T}_0$, because they are of primitive type and are passed by value - Java methods cannot modify the contents of value parameters. We do distinguish between $\mathtt{a}$ and $\mathtt{a}_0$ because arrays are passed by reference in Java.

An alternative specification of the postcondition that *explicitly* states that F and T are unmodified by the code is:

```
void quicksort(char[] a, int F, int T)
//PRE: a ≠ null  ∧  0 ≤ F ≤ T ≤ a.length
//POST: F = F₀ ∧ T = T₀
          ∧ a ≈ a₀[0..F) : a[F..T) : a₀[T..a.length)
          ∧ a ∼ a₀ ∧ Sorted(a[F..T))
```

The alternative version presented above requires us to carry the assertion $\mathtt{F} = \mathtt{F}_0 \wedge \mathtt{T} = \mathtt{T}_0$ with us throughout the Midconditions of the program and also forbids us from modifying these variables (unless we also later restore them to their original values).

We prefer the version in the slides as this leads to less notational overheads.

Note that the condition $\mathtt{a} \sim \mathtt{a}_0$ implies that $\mathtt{a.length} = \mathtt{a}_0\mathtt{.length}$ by ($\sim$**Length**), so we can use the two lengths interchangeably in our assertions so long as this permutation property holds.

Slide 358

Examples of Running quicksort

Consider the following array a:

0	1	2	3	4	5	6	7	8	9
'h'	'f'	'i'	'd'	'e'	'f'	'f'	'b'	'a'	'h'

What is the result of the following calls to quicksort?

quicksort (a ,8,3)

0	1	2	3	4	5	6	7	8	9

quicksort (a ,3,8)

0	1	2	3	4	5	6	7	8	9

quicksort (a ,0,9)

0	1	2	3	4	5	6	7	8	9

`quicksort(a,8,3)`

0	1	2	3	4	5	6	7	8	9
'w'	'h'	'a'	't'	' '	'e'	'v'	'e'	'r'	'!'

In this case the precondition of the method is violated ($8 \not< 3$), so there is no guarantee that the postcondition will hold, i.e. anything can happen that satisfies the typing and call by value constraints of Java!

`quicksort(a,3,8)`

0	1	2	3	4	5	6	7	8	9
‘h’	‘f’	‘i’	‘b’	‘d’	‘e’	‘f’	‘f’	‘a’	‘h’

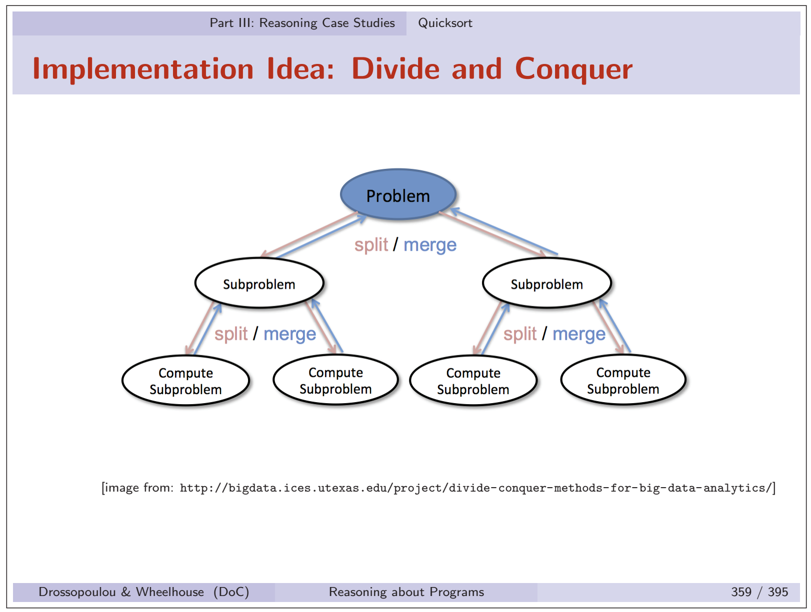
Notice that the range of the sorted slice of array `a` in the postcondition is closed-open, so while array element `F` is sorted, array element `T` is not.

`quicksort(a,0,9)`

0	1	2	3	4	5	6	7	8	9
‘a’	‘b’	‘d’	‘e’	‘f’	‘f’	‘f’	‘h’	‘i’	‘h’

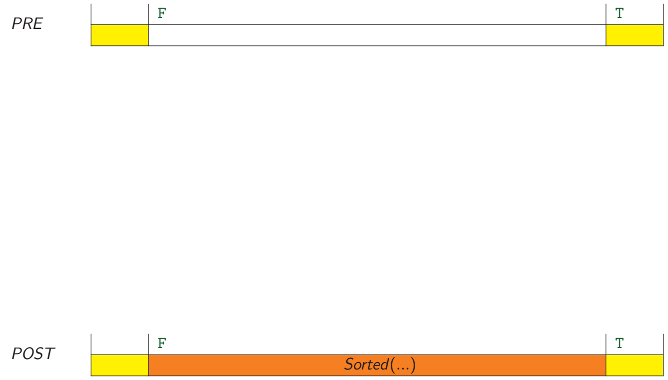
Recursive Quicksort

Slide 359



Slide 360

Quicksort PRE and POST - diagrammatically



Slide 361

Sketching the Midconditions - 1

```
void quicksort(char[] a, int F, int T)
//PRE: a ≠ null  ∧  0 ≤ F ≤ T ≤ a.length
//POST: a ≈ a0[0..F) : a[F..T) : a0[T..a.length)
        ∧ a ∼ a0 ∧ Sorted(a[F..T))
{
  if ( sometest ) {
      ...
      ...
      ...
      ... somecode ...
      ...
      ...
      ...
    //Mid: a ≈ a0[0..F) : a[F..T) : a0[T..a.length)
           ∧ a ∼ a0 ∧ Sorted(a[F..T))
  }
}
```

Slide 362

Knowing When to Stop - working out sometest

With any recursive algorithm, one of the first things you need to do is to work out when to stop recursing.

Let's take a closer look at the postcondition of quicksort:

$$\mathtt{a} \approx \mathtt{a}_0[0..\mathtt{F}) : \mathtt{a}[\mathtt{F}..\mathtt{T}) : \mathtt{a}_0[\mathtt{T}..\mathtt{a.length}) \\ \land\ \mathtt{a} \sim \mathtt{a}_0 \land \mathit{Sorted}(\mathtt{a}[\mathtt{F}..\mathtt{T}))$$

Observe that if the array slice $\mathtt{a}[\mathtt{F}..\mathtt{T})$ is empty (i.e. $\mathtt{F} = \mathtt{T}$), then there is nothing to do. The empty array is vacuously sorted.

This leads to an initial suggestion of an if test of the form:

if (F < T)

Slide 363

Knowing When to Stop - working out sometest - 2

However, we can do better than this.

Let's look again at the postcondition of quicksort:

$$\mathtt{a} \approx \mathtt{a}_0[0..\mathtt{F}) : \mathtt{a}[\mathtt{F}..\mathtt{T}) : \mathtt{a}_0[\mathtt{T}..\mathtt{a.length}) \\ \land\ \mathtt{a} \sim \mathtt{a}_0 \land \mathit{Sorted}(\mathtt{a}[\mathtt{F}..\mathtt{T}))$$

We have already noted that the empty array is vacuously sorted, but an array of just one element is also trivially sorted.

So an improved suggestion for the if-test is:

if (F + 1 < T)

Slide 364

Quicksort - when to stop

```
void quicksort(char[] a, int F, int T)
//PRE: a ≠ null  ∧  0 ≤ F ≤ T ≤ a.length
//POST: a ≈ a_0[0..F) : a[F..T) : a_0[T..a.length)
          ∧ a ∼ a_0 ∧ Sorted(a[F..T))
{
  if ( F + 1 < T ) {
      ...
      ...
      ...
      ...  somecode ...
      ...
      ...
      ...
      //Mid: a ≈ a_0[0..F) : a[F..T) : a_0[T..a.length)
              ∧ a ∼ a_0 ∧ Sorted(a[F..T))
  }
}
```


Slide 365

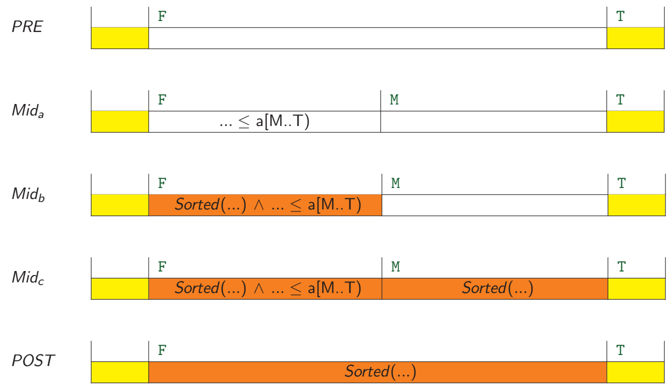
Quicksort 1st Attempt

Rough idea:

- Permute the array slice `a[F..T)`, in such a way that there exists an index `M`∈[`F`..`T`) such that the elements a `a` from `M` onwards are larger or equal to the elements in the array before `M`.
- Then sort the subarrays `a[F..M)` and `a[M..T)`, separately.

Slide 366

Quicksort 1st Attempt - Midconditions diagrammatically



Slide 367

Quicksort 1st Attempt - the problem

What happens if `M`=`F` or `M`=`T`? Then we would not be making progress.

It is possible, but not trivial to ensure that `M`≠`F` and `M`≠`T`.

We will chose another alternative. We will require that there is some midle element, which is larger than all preceding elements and greater or equal than all subsequent elements. Then the recursive calls are guaranteed to be called on strictly smaller ranges.

Slide 368

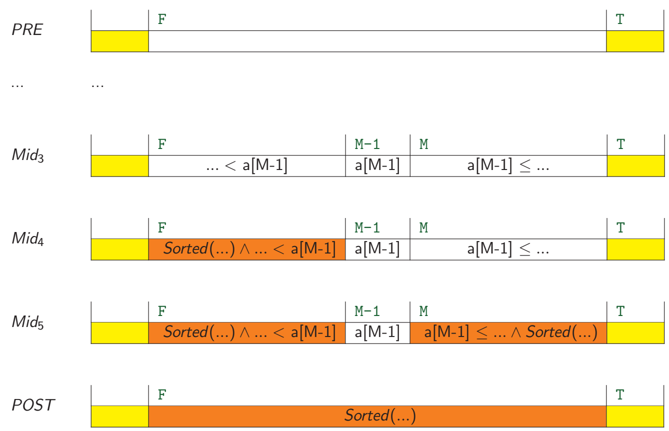
Quicksort 2nd Attempt

Rough idea:

- Permute the array slice `a[F..T)`, in such a way that there exists an index `M-1`∈[`F`..`T`), such that `a[M-1]` is strictly larger than all preceding elements, and smaller equal to all subsequent elements.
- Then sort the subarrays separately.

Slide 369

Quicksort 2nd Attempt - Midconditions graphically



Slide 370

Quicksort - 2nd attempt; Midconditions formally

```
void quicksort(char[] a, int F, int T)
//PRE: a ≠ null  ∧  0 ≤ F ≤ T ≤ a.length
//POST: a ≈ a₀[0..F) : a[F..T) : a₀[T..a.length)
        ∧ a ∼ a₀ ∧ Sorted(a[F..T))
{
  if ( F + 1 < T ) {
    ...some_code...
    //Mid3: a[F..M-1) < a[M-1] ≤ a[M..T)
            ∧ a ≈ a₀[0..F) : a[F..T) : a₀[T..a.length)
            ∧ a ∼ a₀
    ...other_code...
    //Mid4: a[F..M-1) < a[M-1] ≤ a[M..T)
            ∧ a ≈ a₀[0..F) : a[F..T) : a₀[T..a.length)
            ∧ a ∼ a₀  ∧ Sorted(a[F..M))
    ...more_code...
    //Mid5: a[F..M-1) < a[M-1] ≤ a[M..T)
            ∧ a ≈ a₀[0..F) : a[F..T) : a₀[T..a.length)
            ∧ a ∼ a₀ ∧ Sorted(a[F..M)) ∧ Sorted(a[M..T))
  }
}
```


A remaining question is whether such a value `M` can always be found. Consider for example the following array `b`, and values `F`=3, `T`=7.

`b`

0	1	2	3	4	5	6	7	8	9
'g'	'd'	'i'	'h'	'h'	'h'	'h'	'd'	'i'	'A'

We are looking for a `M` such that 3 ≤`M`≤7, such that

$$a[3..\texttt{M-1}) < a[\texttt{M-1}] \leq a[\texttt{M}..7)$$

Even though all of the elements of the array-slice are equal, we can still choose $\texttt{M} = 4$ to satisfy the above assertion.

Slide 371

Implementation: breaking down the problem

The question now is how do we reach the Midcondition on line 8?

Let's use a helper method to break the array into two parts.
We call this task

Partition

Slide 372

Implementing Quicksort - 2nd attempt

We will use the methods `partition` and `swap` to construct the quicksort method.

We can implement `partition` and `swap` later.

For now we will assume that these methods are *totally correct* with respect to their corresponding specifications.

This technique is known as *top-down* program development.

Slide 373

partition and swap – Formal Specification

```
int partition(char[] b, char pivot, int from, int to)
//PRE: b ≠ null  ∧  b.length > 0  ∧  0 ≤ from ≤ to ≤ b.length
//POST: from ≤ r ≤ to ∧ b ∼ b₀  ∧
        b ≈ b₀[0..from) : b[from..to) : b₀[to..b.length)  ∧
        b[from..r) < pivot ≤ b[r..to)


void swap(int[] c, int i, int j)
//PRE: i, j ∈ [0..c.length)
//POST: Swapped(c, c₀, i, j)
```


As for the specification of `quicksort`, we implicitly also assume that $\texttt{b} \neq \texttt{null}$

Similarly we do not distinguish between `from` and $\texttt{from}_0$, nor between `to` and $\texttt{to}_0$, because they are of primitive type, and the method cannot modify their values. But we do distinguish between `b` and $\texttt{b}_0$ because the array is passed by reference.

Slide 374

Examples of Running partition

Consider the following array a:

0	1	2	3	4
‘e’	‘p’	‘d’	‘q’	‘c’

What is a possible result of the following calls to `partition`?

`r=partition(a,’f’,0,4)` r = a’=

0	1	2	3	4

`r=partition(a,’b’,0,4)` r = a’=

0	1	2	3	4

`r=partition(a,’z’,0,4)` r = a’=

0	1	2	3	4

`r=partition(a,’f’,0,4)` $r = 2$ a’=

0	1	2	3	4
’d’	’e’	’q’	’p’	’c’

`r=partition(a,’b’,0,4)` $r = 0$ a’=

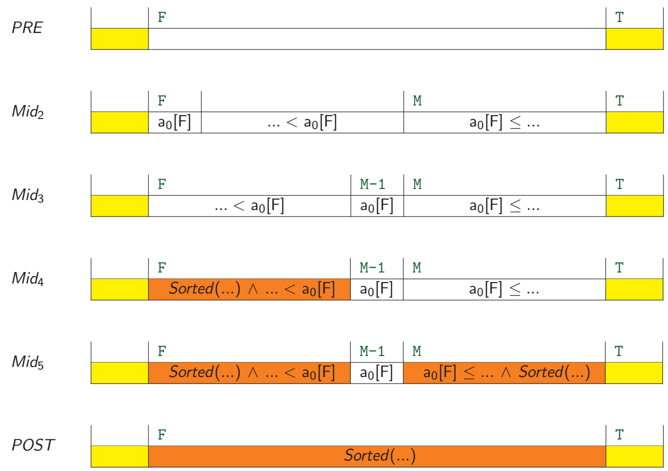
0	1	2	3	4
’e’	’q’	’p’	’d’	’c’

`r=partition(a,’z’,0,4)` $r = 5$ a’=

0	1	2	3	4
’e’	’q’	’p’	’d’	’c’

Slide 375

Quicksort Midconditions



Slide 376

Quicksort - 2nd attempt; Midconditions and Code

```
void quicksort(char[] a, int F, int T)
//PRE: a ≠ null ∧ 0 ≤ F ≤ T ≤ a.length
//POST:
    a ≈ a₀[0..F) : a[F..T) : a₀[T..a.length) ∧  a ∼ a₀  ∧  Sorted(a[F..T))

{
  if ( F + 1 < T ) {
    int M = partition(a,a[F],F+1,T);
     //Mid₂: a[F+1..M) < a₀[F] ≤ a[M..T)  ∧  a ∼ a₀  ∧
              a ≈ a₀[0..F+1) : a[F+1..T) : a₀[T..a.length)
    swap(a,F,M-1);
    //Mid₃: a[F..M-1) < a[M-1] ≤ a[M..T)  ∧  a ∼ a₀  ∧
              a ≈ a₀[0..F) : a[F..T) : a₀[T..a.length)
    quicksort(a,F,M-1);
    //Mid₄: a[F..M-1) < a[M-1] ≤ a[M..T)  ∧  a ∼ a₀  ∧
              a ≈ a₀[0..F) : a[F..T) : a₀[T..a.length)  ∧
              Sorted(a[F..M-1))
    quicksort(a,M,T);
    //Mid₅: a[F..M-1) < a[M-1] ≤ a[M..T)  ∧  a ∼ a₀  ∧
              a ≈ a₀[0..F) : a[F..T) : a₀[T..a.length)  ∧
              Sorted(a[F..M-1))  ∧  Sorted(a[M..T))
    }
```

Quicksort - 2nd attempt; Midconditions Completed

We extend the midconditions

```
...
// Mid2: 0<F+1<T≤a.length ∧ a[F+1..M) < a0[F] ≤ a[M..T)
   ∧ a ∼ a0 ∧ a ≈ a0[0..F+1) : a[F+1..T) : a0[T..a.length)
   ∧ F+1 ≤ M ≤ T
...
//Mid3: 0<F+1<T≤a.length ∧ a[F..M-1) < a[M-1] ≤ a[M..T)
   ∧ a ∼ a0 ∧ a ≈ a0[0..F) : a[F..T) : a0[T..a.length)
   ∧ 0 ≤ F ≤ M-1 ≤ a.length
...
//Mid4: 0<F+1<T≤a.length ∧ a[F..M-1) < a[M-1] ≤ a[M..T)
   ∧ a ∼ a0 ∧ a ≈ a0[0..F) : a[F..T) : a0[T..a.length)
   ∧ Sorted(a[F..M-1)) ∧ 0 ≤ M ≤ T ≤ a.length
...
//Mid5: 0<F+1<T≤a.length ∧ a[F..M-1) < a[M-1] ≤ a[M..T)
   ∧ a ∼ a0 ∧ a ≈ a0[0..F) : a[F..T) : a0[T..a.length)
   ∧ Sorted(a[F..M-1)) ∧ Sorted(a[M..T))
```


We complete the mid-conditions so as to be able to satisfy the preconditions for the recursive calls of `quicksort` and `swap`. We are allowed to do this, because in the if-branch of the function `quicksort` we have that $\texttt{F+1} < \texttt{T}$. Also, the range of the value of `M` is determined from the call on `partition`

Slide 378

Proving Quicksort

Slide 379

Proof Obligations - partial correctness

1. `F+1`$\geq$`T` implies *POST*.
2. *PRE* and if-condition implies precondition of `partition(a,a[F],F+1,T)`.
3. Mid_2 holds at line 8.
4. Mid_2 implies precondition of `swap(a,F,M-1)`.
5. Mid_3 holds at line 11.
6. Mid_3 implies precondition of `quicksort(a,F,M-1)`.
7. Mid_4 holds at line 14.
8. Mid_4 implies precondition of `quicksort(a,M,T)`.
9. Mid_5 holds at line 18.
10. Mid_5 implies implies *POST*.

Slide 380

Proof Obligations - total correctness

1. All array accesses are legal.
2. `quicksort(a,F,T)` terminates.

Termination of `quicksort(a,F,T)` can be proven by induction on `T-F`.

Lemma 1 $\forall n, \mathtt{F}, \mathtt{T} \in \mathbb{N}.[\ \mathtt{T\text{-}F} = n \longrightarrow \mathtt{quicksort(a,F,T)}$ terminates $]$.

Lemma 1 can be proven by strong induction. That is, we shall prove

Lemma 2 $\forall n, \mathtt{F}, \mathtt{T} \in \mathbb{N}.[\ \mathtt{T\text{-}F} \leq n \longrightarrow \mathtt{quicksort(a,F,T)}$ terminates $]$.

Then, Lemma 1 is a direct corollary of Lemma 2.

Slide 381

Some proof obligations

We shall now sketch *some* of the proof obligations for partial correctness.

Slide 382

PO1: $\mathtt{F+1} \geq \mathtt{T}$ implies *POST*

Given

1. $0 \leq \mathtt{F} \leq \mathtt{T} \leq \mathtt{a.length}$
2. $\mathtt{F} + 1 \geq \mathtt{T}$
3. $\mathtt{a} \approx \mathtt{a}_0$

To Show

(α) $\mathtt{a} \approx \mathtt{a}_0[0..\mathtt{F}) : \mathtt{a}[\mathtt{F}..\mathtt{T}) : \mathtt{a}_0[\mathtt{T}..\mathtt{a.length})$
(β) $\mathtt{a} \sim \mathtt{a}_0$
(γ) $\mathit{Sorted}(\mathtt{a}[\mathtt{F}..\mathtt{T}))$

From the **Given**s, 1. comes from the precondition, 2. is the negation of the if-test, and

3. comes from the code (as we have not modified the array).

From the assertions **to be shown**, α, β and γ come directly from the post-condition of `quicksort(a,F,T)`.

Proof

α and β follow from 3.
From 1, and 2 we obtain that `T=F`, or `T=F+1`. Therefore, the array `a[F..T)` has length at most 1. This gives us γ.

Slide 383

PRE and if-condition implies precondition `partition(a,a[F],F+1,T)`

Given
1. $0 \leq \mathtt{F} \leq \mathtt{T} \leq \mathtt{a.length}$
2. $\mathtt{F} + 1 < \mathtt{T}$
3. $\mathtt{a} \approx \mathtt{a}_0$

To Show
(α) $\mathtt{a.length} > 0$
(β) $0 \leq \mathtt{F+1} \leq \mathtt{T} \leq \mathtt{a.length}$

From the **Given**s, 1. comes from the precondition, 2. is the if-test, and 3. comes from the code (as we have not modified the array).

From the assertions **to be shown**, α and β are the precondition of `partition(a,a[F],F+1,T)`, with the following replacements: $\mathtt{b} \mapsto \mathtt{a}$, $\mathtt{pivot} \mapsto \mathtt{a[F]}$, $\mathtt{from} \mapsto \mathtt{F+1}$, and $\mathtt{to} \mapsto \mathtt{T}$.

Proof

From 1 and 2 we obtain that `T`$\geq$1, and with 2. we obtain that `a.length`$\geq$1, which gives α .
Moreover, 1. and 2. give β.

Slide 384

MID_2 holds at line 8.

Given

1. $0 \leq \texttt{F} \leq \texttt{T} \leq \texttt{a.length}$
2. $\texttt{F+1} < \texttt{T}$
3. $\texttt{a} \approx \texttt{a}_0$
4. $\texttt{F+1} \leq \texttt{M} \leq \texttt{T} \quad \wedge \quad \texttt{a}' \sim \texttt{a}$
5. $\texttt{a}'[\texttt{F+1}..\texttt{M}) < \texttt{a[F]} \leq \texttt{a}'[\texttt{M}..\texttt{T})$
6. $\texttt{a}' \approx \texttt{a}[0..\texttt{F+1}) : \texttt{a}'[\texttt{F+1}..\texttt{T}) : \texttt{a}[\texttt{T}..\texttt{a'.length})$

To Show

(α) $0 \leq \texttt{F+1} < \texttt{T} \leq \texttt{a.length}$
(β) $\texttt{a}'[\texttt{F+1}..\texttt{M}) < \texttt{a}_0[\texttt{F}] \leq \texttt{a}'[\texttt{M}..\texttt{T})$
(γ) $\texttt{a}' \sim \texttt{a}_0$
(δ) $\texttt{a}' \approx \texttt{a}_0[0..\texttt{F+1}) : \texttt{a}'[\texttt{F+1}..\texttt{T}) : \texttt{a}_0[\texttt{T}..\texttt{a}'.\texttt{length})$
(ϵ) $\texttt{F+1} \leq \texttt{M} \leq \texttt{T}$

From the **Given**s, 1. comes from the precondition, 2. is the if-test, and 3. comes from the code (as we have not modified the array before calling `partition`).
Assertions 4.-6. come from the postcondition of `partition(a,a[F],F+1,T)`, with the following replacements: $\texttt{b}_0 \mapsto \texttt{a}, \texttt{b} \mapsto \texttt{a}'$, $\texttt{pivot} \mapsto \texttt{a[F]}$, $\texttt{from} \mapsto \texttt{F+1}$, $\texttt{to} \mapsto \texttt{T}$, and $\mathbf{r} \mapsto \texttt{M}$. These lines describe the *effect* of the call `partition(a,a[F],F+1,T)` on the contents of the array `a`, hence we replace $\texttt{b}_0$ by `a`, ie the array before the call, and `b` by $\texttt{a}'$, ie the array after the call.

The assertions **to be shown**, α-ϵ are the midcondition Mid_2, with the replacement: $\texttt{a} \mapsto \texttt{a}'$. That is, we want to show that the assertion Mid_2 holds for the array *after* the call.

Proof

α follows from 1.
β follows from 5. and 3.
γ follows from 3. and the second conjunct in 4., and lemmas ($\approx$**Prm**) and ($\sim$**Trans**).
δ follows from 3. and 6.
ϵ follows from 4.

MID_2 guarantees the precondition of `swap(a,F,M-1)`.

Slide 385

Given
1. $0 \leq \texttt{F+1} < \texttt{T} \leq \texttt{a.length}$
2. $\texttt{a}[\texttt{F+1}..\texttt{M}) < \texttt{a}_0[\texttt{F}] \leq \texttt{a}[\texttt{M}..\texttt{T})$
3. $\texttt{a} \sim \texttt{a}_0$
4. $\texttt{a} \approx \texttt{a}_0[0..\texttt{F+1}) : \texttt{a}[\texttt{F+1}..\texttt{T}) : \texttt{a}_0[\texttt{T}..\texttt{a.length})$
5. $\texttt{F+1} \leq \texttt{M} \leq \texttt{T}$

To Show
(α) $\texttt{a} \neq \texttt{null}$
(β) $\texttt{F}, \texttt{M-1} \in [0..\texttt{a.length})$

From the **Given**s, 1.-4. come from the Mid_2.
Assertions $\alpha - \gamma$ come from the postcondition of `swap(b,pivot,from,to)`, with the following replacements: $\texttt{c} \mapsto \texttt{a}$, $\texttt{i} \mapsto \texttt{F}$, $\texttt{j} \mapsto \texttt{M-1}$.

Proof α follows from the precondition of `quicksort`, and 3.
β follows from the precondition of `quicksort` ($0 \leq \texttt{F}$) and 5, and 1. ($\texttt{M} \leq \texttt{a.length}$).

Slide 386

MID_3 holds at line 10.

Given

1. $0 < \mathtt{F+1} < \mathtt{T} \leq \mathtt{a.length}$
1. $\mathtt{a}[\mathtt{F+1..M}) < \mathtt{a}_0[\mathtt{F}] \leq \mathtt{a}[\mathtt{M..T})$
2. $\mathtt{a} \sim \mathtt{a}_0$
3. $\mathtt{a} \approx \mathtt{a}_0[0..\mathtt{F+1}) : \mathtt{a}[\mathtt{F+1..T}) : \mathtt{a}_0[\mathtt{T..a.length})$
4. $\mathtt{F+1} \leq \mathtt{M} \leq \mathtt{T}$
5. $\mathit{Swapped}(\mathtt{a}, \mathtt{a}', \mathtt{F}, \mathtt{M-1})$

To Show

(α) $0 < \mathtt{F+1} < \mathtt{T} \leq \mathtt{a.length}$
(β) $\mathtt{a}'[\mathtt{F..M-1}) < \mathtt{a}'[\mathtt{M-1}] \leq \mathtt{a'}[\mathtt{M..T})$
(γ) $\mathtt{a}' \sim \mathtt{a}_0$
(δ) $\mathtt{a}' \approx \mathtt{a}_0[0..\mathtt{F}) : \mathtt{a}'[\mathtt{F..T}) : \mathtt{a}_0[\mathtt{T..a.length})$
(ϵ) $0 \leq \mathtt{F} \leq \mathtt{M-1} \leq \mathtt{a}'\mathtt{.length}$

From the **Given**s, 1.-4. come from Mid_2 And **Given**.5. comes from the postcondition of sap, where we apply the the following replacements:
$\mathtt{c} \mapsto \mathtt{a}, \mathtt{i} \mapsto \mathtt{F}, \mathtt{j} \mapsto \mathtt{M-1}$.

The assertions **to be shown**, α-ϵ are the midcondition Mid_3, with the replacement: $\mathtt{a} \mapsto \mathtt{a}'$. That is, we want to show that the assertion Mid_3 holds for the array after the call of `swap(a,F,M-1)`.

Proof α follows from 1.
β follows from 5. and 3.
γ follows from 3. and the second conjunct in 4., and lemmas ($\approx$**Prm**) and ($\sim$**Trans**).
δ follows from 3. and 6.
ϵ follows from 4.

Slide 387

Remaining proof obligations and other questions

We leave the remaining proof obligations and their proofs as exercise.

For fun: Can you turn the recursive `quicksort` to an iterative one? Can you then optimize the iterative version?

... read on in
Razvan Certezeanu, Sophia Drossopoulou, Benjamin Egelund-Mller, K. Rustan M. Leino, Sinduran Sivarajan, Mark J. Wheelhouse:
Quicksort Revisited - Verifying Alternative Versions of Quicksort.
Theory and Practice of Formal Methods 2016: 407-426
This is one of the outcomes of UROP placements in 2015.

Summary of Part III

Slide 388

Part III – Conclusion

- Midconditions can be used to develop a program
- Before a method call, we must establish its precondition
- We use the current midcondition to establish a method's precondition
- After a method call, we may assume its postcondition
- We use the postcondition to establish the next midcondition

The steps outlined above reflect the mental process that underlies program development

- In *top-down* development, functions call other functions whose specifications are known, but which have not yet been implemented

Slide 389

Summary of Part III

This concludes this lecture, and also our course.

Slide 390

Summary of course - 1

- Every inductively defined structure introduces an inductive principle.
- We can reason inductively over sets, in particular over the set $\mathbb{N}$, or Haskell data structures (structural induction).
- We can reason inductively over relations - you will see this in the second year course Models of Computation.
- We can reason inductively over functions.

Slide 391

Summary of course - 2

- The behaviour of imperative programs is characterized by pre- and post-conditions.
- Before a function call we have to prove *its* precondition.
- After a function call we may assume *its* postocondtion.
- A loop is characterized by an *invariant* and a *variant*.
- Before a loop we must establish the invariant.
- The loop body must re-establish the invariant.
- After the loop the invariant and negation of the condition hold.
- For termination, we need to show that the variant is bounded, and decreases with each iteration.

Slide 392

Summary of course - 3

- Be clear about what is given and what is to be shown.
- Justify each step.
- Vary size of steps according to task and your confidence.

Summary of course - 4

The exams will be ...

- in the same spirit as the tutorial sheets,
- but easier,
- and will contain a small, more demanding part.

One Final Remark

So, do we need to test our programs anymore...?

Dean, to the Computing Department:

> *"Why do I always have to give you guys so much money for laboratories and expensive equipment and stuff? Why couldn't you be more like the maths department - all they need is pencils, paper and waste-paper baskets!"*

Thank you

Slide 395

Thank you