LECTURE 3:

Verifying correctness of algorithms

Organizational

- First homework deadline coming soon. **Next will come soon too.**

- Solutions will be posted next week.

- HONESTY !

- Last week's slides: additional example not covered in class. This week's probably as well. Take a look at the slides !
- College is about your work more than teaching.

Outline

- Analysis of algorithms
- Basic notions
- Basic steps in correctness verification
- Rules for correctness verification
- Isn’t this too theoretical ?

Analysis of algorithms

When we design an algorithm there are two main aspects which should be analyzed:

- correctness:
  - analyze if the algorithm produces the desired output after a finite number of operations
- efficiency:
  - estimate the amount of resources needed to execute the algorithm on a machine

Correctness

There are two main ways to verify if an algorithm solves a given problem:

- Experimental (by testing): the algorithm is executed for a several instances of the input data

- Formal (by proving): it is proved that the algorithm produces the right answer for any input data

- In practice: testing, informed by formal methods.

Advantages and Disadvantages

	Experimental	Formal
Advantages	• simple • easy to apply	• guarantees the correctness
Disadvantages	• doesn’t guarantee the correctness	• rather difficult • cannot be applied for complex algorithms

Outline

- Algorithm analysis
- Basic notions
- Basic steps in correctness verification
- Rules for correctness verification

Basic notions

- Preconditions and postconditions

- Algorithm state

- Assertions

- Annotation

Preconditions and postconditions

- Precondititions = properties satisfied by the input data
- Postconditions= properties satisfied by the result

Example: *Find the minimum, m, of a non-empty array, x[1..n]*

Preconditions: n>=1 (the array is non-empty)

Postconditions: m=min{x[i]| 1<=i<=n}

(the variable m contains the smallest value in x[1..n])

Preconditions and postconditions

(Partial) Correctness verification =

prove that **if** the algorithm terminates **then** it leads to postconditions starting from preconditions

Total correctness verification = prove partial correctness + finiteness

Intermediate steps in correctness verification:

- analyze the algorithm state and
- the effect of each processing step on the algorithm state

Basic notions

- Preconditions and postconditions
- Algorithm state
- Assertions
- Annotation

Algorithm state

- Algorithm state = set of values corresponding to all variables used in the algorithm
- During the execution of an algorithm its state changes (since the variables change their values)
- The algorithm is correct if at the end of the algorithm its state implies the postconditions

Algorithm state

Example: Solving the equation ax=b, a<>0

Input data: a

Preconditions: a<>0

Output data: x

Postconditions: x satisfies ax=b

Algorithm:

Solve (real a,b)
  real x
  x $\leftarrow$ b/a
  return x

Algorithm state

Current values of a and b

a=a0, b=b0, x undefined

a=a0, b=b0, x=b0/a0

ax=b

Basic notions

- Preconditions and postconditions
- **State** of the algorithm
- Assertions
- Annotation

Assertions

- Assertion = statement (asserted to be true) about the algorithm' state

- Assertions are used to annotate the algorithms

- Annotation is useful both in
  - correctness verification

and as
  - documentation tool

Basic notions

- Preconditions and postconditions
- Algorithm’s state
- Assertions
- Annotation

Annotation

Preconditions: a,b,c are distinct real numbers
Postconditions: m=min(a,b,c)

```
min (real a,b,c)               //{a<>b, b<>c, c<>a}

IF a<b THEN                    //{a<b}
   IF a<c THEN m ← a           //{a<b, a<c, m=a}   ⇒   m=min(a,b,c)
          ELSE m ← c           //{a<b, c<a, m=c}   ⇒   m=min(a,b,c)
   ENDIF
ELSE                           //{b<a}
   IF b<c THEN m ← b           //{b<a, b<c, m=b}   ⇒   m=min(a,b,c)
          ELSE m ← c           //{b<a, c<b, m=c}   ⇒   m=min(a,b,c)
   ENDIF
ENDIF
RETURN m
```

Annotation

Preconditions: a,b,c are distinct real numbers
Postconditions: m=min(a,b,c)

Another variant to find the minimum of three values

```
min (real a,b,c)                        //{a<>b, b<>c, c<>a}

m ← a                                    // m=a
IF m>b THEN m ← b ENDIF                  // m<=a, m<=b
IF m>c THEN  m ← c  ENDIF                // m<=a, m<=b, m<=c
RETURN m
```

m=min(a,b,c)

Outline

- Algorithms analysis
- Basic notions
- Basic steps in correctness verification
- Rules for correctness verification

Basic steps in correctness verification

- Identify the preconditions and postconditions

- Annotate the algorithm with assertions concerning its state such that

  - the preconditions are satisfied
  - the final assertion implies the postconditions

- Prove that by each processing step one arrives from the previous assertion to the next assertion

Some notations

Let us denote by

P - the preconditions
Q - the postconditions
A - the algorithm

The triple (P,A,Q) denote a correct algorithm if for input data which satisfy the preconditions P the algorithm will:

- lead to postconditions Q
- stop after a finite number of processing steps

Notation:

$$P \xrightarrow{A} Q$$

Outline

- Algorithms analysis
- Basic notions
- Basic steps in correctness verification
- Rules for verifying correctness

Rules for verifying correctness

To prove that an algorithm is correct it can be useful to know rules corresponding to the usual statements:

- Sequential statement
- Conditional statement
- Loop statement

Sequential statement rule

Structure

A:

$\{P_0\}$

A_1

$\{P_1\}$

…

$\{P_{i-1}\}$

A_i

$\{P_i\}$

…

$\{P_{n-1}\}$

A_n

$\{P_n\}$

Rule:

If

$P \rightarrow P_0$

$P_{i-1} \xrightarrow{A_i} P_i$, $i=1..n$

$P_n \quad Q$

Then

$P \xrightarrow{A} Q$

What does this mean ?

If

- the precondition implies the initial assertion,
- each action implies the next assertion
- the final assertion implies the post-condition

then the sequence is correct

Sequential statement rule

Problem: Let x and y be two variables having the values a and b, respectively. Swap the values of the two variables.

P: {x=a, y=b}

Q: {x=b,y=a}

Variant 1:	Variant 2
aux:=x	x:=x+y
x:=y	y:=x-y
y:=aux	x:=x-y

Sequential statement rule

Problem: Let x and y be two variables having the values a and b, respectively. Swap the values of the two variables.

P: {x=a, y=b}

Q: {x=b,y=a}

Variant 1:

{x=a,y=b, aux undefined}

aux:=x

{x=a, y=b, aux=a}

x:=y

{x=b, y=b, aux=a}

y:=aux

{x=b, y=a, aux=a} Q

Variant 2 (a and b are numbers):

{x=a,y=b}

x:=x+y

{x=a+b, y=b}

y:=x-y

{x=a+b, y=a}

x:=x-y

{x=b, y=a} Q

Sequential statement rule

What about this variant ?

```
x:=y

y:=x
```

Sequential statement rule

What about this variant ?

$\{x=a,y=b\}$
x:=y
$\{x=b, y=b\}$
y:=x
$\{x=b, y=b\}$ $\cancel{Q}$

The code doesn't meet the specification !

Conditional statement rule

Structure

A:

$\{P_0\}$

IF c

THEN

$\{c,P_0\}$

A_1

$\{P_1\}$

ELSE

$\{NOTc,P_0\}$

A_2

$\{P_2\}$

Rule:

If

- c is well defined
- c AND $P_0 \xrightarrow{A_1} P_1$
- $P_1 \quad Q$
- NOT c AND $P_0 \xrightarrow{A_2} P_2$
- $P_2 \quad Q$

then A

$P \rightarrow Q$

What does it mean ?

The condition c can be evaluated

Both branches lead to the postconditions

Conditional statement rule

```
Problem:  compute the minimum of
          two distinct values
Preconditions:  a<>b
Postconditions: m=min{a,b}
   {a<>b}
IF a<b
    THEN
          {a<b}
        m:=a
          {a<b, m=a}
    ELSE
          {b<a}
        m:=b
          {b<a, m=b}
```

Since

{a<b, m=a} implies m=min{a,b}

and

{b<a, m=b} implies m=min{a,b}

the algorithm meets the specification

Loop statement rule

Verifying the correctness of sequential and conditional statements is easy…

Verifying loops is **not easy** …

Informally speaking, a loop is correct when:

- **If it finishes** it leads to postconditions
- It finishes after a finite number of steps

If only the first property is satisfied then the loop is partially correct

Partial correctness can be proved by using mathematical induction or by using loop invariants

Full correctness needs that the algorithm terminates

Loop invariants

Let us consider the WHILE loop:

P {I}
WHILE c DO
{c,I}
A
{I}
ENDWHILE
{NOT c, I} Q

Definition:

A **loop invariant** is an assertion that

1 is true at the beginning of the loop

2 As long as c is true **it remains true after each execution** of the loop body

3 When c is false it implies the postconditions

If we can find a loop invariant then that loop is partially correct

Loop invariants

Preconditions:

x[1..n] non-empty array (n>=1)

Postconditions:

m=min{x[i]|1<=i<=n}

```
m←x[1]
FOR i ← 2,n DO
 IF x[i]<m THEN m ← x[i]
ENDFOR
```

➡

```
m ← x[1]
i ← 2
WHILE i<=n DO
   IF x[i]<m THEN m ← x[i]
   ENDIF
    i ← i+1
ENDWHILE
```

Loop invariants

Preconditions:

x[1..n] non-empty array (n>=1)

Postconditions:

m=min{x[i]|1<=i<=n}

```
m ← x[1]
i ← 2
WHILE i<=n DO
    IF x[i]<m THEN m ← x[i]
    ENDIF
    i ← i+1
ENDWHILE
```

```
i ← 1
m ← x[i]
WHILE i<n DO
    i ← i+1
    IF x[i]<m THEN m ← x[i]
    ENDIF
ENDWHILE
```

Loop invariants

P: n>=1 Q: m=min{x[i]; i=1..n}

```
m ← x[1]
i ← 2
      {m=min{x[j]; j=1..i-1}}
WHILE i<=n DO {i<=n}
   IF x[i]<m THEN m ← x[i]
      {m=min{x[j]; j=1..i}}
   ENDIF
   i ← i+1
      {m=min{x[j]; j=1..i-1}}
ENDWHILE
```

Loop invariant:

m=min{x[j]; j=1..i-1}

Why ? Because …

- when i=2 and m=x[1] it holds
- while i<=n after the execution of the loop body it still holds
- finally, when i=n+1 it implies m=min{x[j]; j=1..n} which is exactly the postcondition

Loop invariants

P: $n>=1$ Q: $m=\min\{x[i];\ i=1..n\}$

```
i ← 1
m ← x[i]
     {m=min{x[j]; j=1..i}}
WHILE i<n DO {i<n}
   i ← i+1
     {m=min{x[j]; j=1..i-1}}
   IF x[i]<m THEN m ← x[i]
     {m=min{x[j]; j=1..i}}
   ENDIF
ENDWHILE
```

Loop invariant:

$m=\min\{x[j];\ j=1..i\}$

Why ? Because …

- when $i=1$ and $m=x[1]$ the invariant is true
- while $i<n$ after the execution of the loop body it still remains true
- finally, when $i=n$ it implies $m=\min\{x[j];\ j=1..n\}$ which is exactly the postcondition

Loop invariants

Problem: Let x[1..n] be an array which contains x0. Find the smallest index i for which x[i]=x0

P: n>=1 and there exists 1<= k <= n such that x[k]=x0

Q: x[i]=x0 and x[j]<>x0 for j=1..i-1

```
i ← 1
WHILE x[i]<>x0 DO
  i ← i+1
ENDWHILE
```

Loop invariants

Problem: Let x[1..n] be an array which contains x0. Find the smallest index i for which x[i]=x0

P: n>=1 and there exists 1<= k <= n such that x[k]=x0

Q: x[i]=x0 and x[j]<>x0 for j=1..i-1

```
i ← 1
    {x[j]<>x0 for j=1..0}
WHILE x[i]<>x0 DO
    {x[i]<>x0, x[j]<>x0 for j=1..i-1}
    i ← i+1
    {x[j]<>x0 for j=1..i-1}
ENDWHILE
```

Loop invariant:

x[j]<>x0 for j=1..i-1

Why ? Because …

- for i=1 the range j=1..0 is empty thus the assertion is satisfied
- Let us suppose that x[i]<>x0 and the invariant is true

  Then x[j]<>x0 for j=1..i
- After i:=i+1 we obtain again x[j]<>x0 for j=1..i-1
- Finally, when x[i]=x0 we obtain Q

Loop invariants

Loop invariants are useful not only for correctness proving but also for loop design

Ideally would be to
- find first the loop invariant
- then design the algorithm

Problem: compute the sum of the first n natural values
Precondition: n>=1 Postcondition: S=1+2+…+n

What is the property S should satisfy after the execution of the n-th loop ?
Invariant: S=1+2+…+i

Idea for loop design:
- first prepare the term
- then add the term to the sum

Loop invariants

Algorithm:

i ← 1
S ← 1
  {S=1+2+…+i}
WHILE i<n DO
    {S=1+2+…+i}
  i ← i+1
    {S=1+2+…+i-1}
  S ← S+i
    {S=1+2+…+i}
ENDWHILE

{i=n, S=1+2+…+i} S=1+…+n

Algorithm:

S ← 0
i ← 1
  {S=1+2+…+i-1}
WHILE i<=n DO
    {S=1+2+…+i-1}
  S ← S+i
    {S=1+2+…+i}
  i ← i+1
    {S=1+2+…+i-1}
ENDWHILE

{i=n+1, S=1+2+…+i-1} S=1+…+n

Isn't this too theoretical ?

- Seems too complicated to apply in practice.
- There is no algorithm to decide whether a given computer program will halt. Hard to find appropriate pre/post conditions.
- **Still useful.**
- **Scenario:** function f expects to be called with a natural number n as an argument. instead it receives an arbitrary integer.

  .....
- repeat {... n=n-1} until (n == 0) infinite loop !

  ....
- Python: assertions.

def KelvinToFahrenheit(Temperature):

assert (Temperature >= 0),"Colder than absolute zero!"

*return ((Temperature-273)*1.8)+32*

Isn't this too theoretical ? (II)

- print KelvinToFahrenheit(273)
  print int(KelvinToFahrenheit(505.78))
  print KelvinToFahrenheit(-5)

  32.0
  451
  Traceback (most recent call last):
  File "test.py", line 9, in <module>
  print KelvinToFahrenheit(-5)
  File "test.py", line 4, in KelvinToFahrenheit
  assert (Temperature >= 0),"Colder than absolute zero!"
  AssertionError: Colder than absolute zero!

Isn't this too theoretical ? (III)

- Apply correctness-checking locally
- **Commenting always a good idea !**
- "Practical" version of testing: unit testing. Structured way to make assertions
- Test (on some examples) that function gives desired result.
- Python: pyunit. Most other programming languages (Junit, cppunit, ...)
- (pre)conditions often "hidden": when we write x[i] we implicitly assume that **i is less than the largest index of an element of array x**.
- while (i<n):

  x[i]....

  i++
- while (i<=n):

  x[i]....

  i++
- **they differ by loop invariants !**

Isn't this too theoretical ? (IV)

- Requires thinking about pre/post conditions.
- Somettimes preconditions derived from correctness constraints.

  {a< b}
  X= a*c
  Y= a*c
  {X<Y} requires c>0 !

- Not a method to apply mechanically. Think !
- Test-driven design: write functions/classes after you've written unit tests for them.
- **Good idea:** you often modify code. Unit tests make sure that the function remains correct.

Summary

Proving the correctness of an algorithm means:

- To prove that it leads from the preconditions to postconditions (partial correctness)
- To prove that it is finite

A loop invariant is a property which

- Is true before the loop
- Remains true by the execution of the loop body
- At the end of the loop it implies the postconditions

Work for you (informal, not assigned)

- Take a look at the extra example below.
- As you learn python
- read documentation on assert
- write programs using assert
- Read more on unit testing, documentation on pyunit
- Try to unit test a simple program using pyunit

Example: successor problem (last slides of Lecture 2)

Reminder: find the successor of an element in the strictly increasing sequence of natural values containing n distinct digits

```
Successor(integer x[1..n])
integer  i, k
i ← Identify(x[1..n])
IF i=1
THEN write "There is no successor !"
ELSE
   k ← Minimum(x[i-1..n])
   x[i-1]↔x[k]
   x[i..n] ← Reverse(x[i..n])
   write x[1..n]
ENDIF
```

Subalgorithms to be verified:

Identify (x[1..n])

P: n>1, there exists i such that x[i-1]<x[i]

Q: x[i-1]<x[i] and x[j-1]>x[j], j=i+1..n

Minimum (x[i-1..n])

P: x[i-1]<x[i]

Q: x[k]<=x[j], j=1..n, x[k]>x[i-1]

Reverse(x[i..n])

P: x[j]=x0[j], j=1..n

Q: x[j]=x0[n+i-j], j=1..n

Example: successor problem

Identify the rightmost element, x[i], which is larger than its left neighbour (x[i-1])

Identify(integer x[1..n])
Integer i
i ← n
WHILE (i>1) and (x[i]<x[i-1])
do
  i ← i-1
ENDWHILE
RETURN i

P: n>1, there exists i such that x[i-1]<x[i]

Q: x[i-1]<x[i] and x[j-1]>x[j], j=i+1..n

Loop invariant:

x[j-1]>x[j], j=i+1..n

Example: successor problem

Find the index of the smallest value in the subarray x[i..n] which is larger than x[i-1]

Minimum(integer x[i..n])

Integer j

k ← i

j ← i+1

WHILE j<=n do

  IFx[j]<x[k] and x[j]>x[i-1]

THEN k ← j

  ENDIF

  j ← j+1

RETURN k

P: x[i-1]<x[i]

Q: x[k]<=x[j], j=1..n, x[k]>x[i-1]

Loop invariant:

x[k]<=x[r], r=i..j-1

x[k]>x[i-1]

Example: successor problem (back to Lecture 2)

Reverse the order of elements of of x[left..right]

P: x[j]=x0[j], j=left..right

Q: x[j]=x0[left+right-j], j=left..right

```
reverse (INTEGER x[left..right])
  INTEGER j1,j2
  j1 ← left
  j2 ← right
  WHILE j1<j2 DO
      x[j1]↔x[j2]
      j1 ← j1+1
      j2 ← j2-1
  ENDWHILE
  RETURN x[left..right]
```

Loop invariant:

x[j]=x0[left+right-j], j=left..j1-1

x[j]>x0[j], j=left..right

x[j]=x0[left+right-j], j=j2+1..right