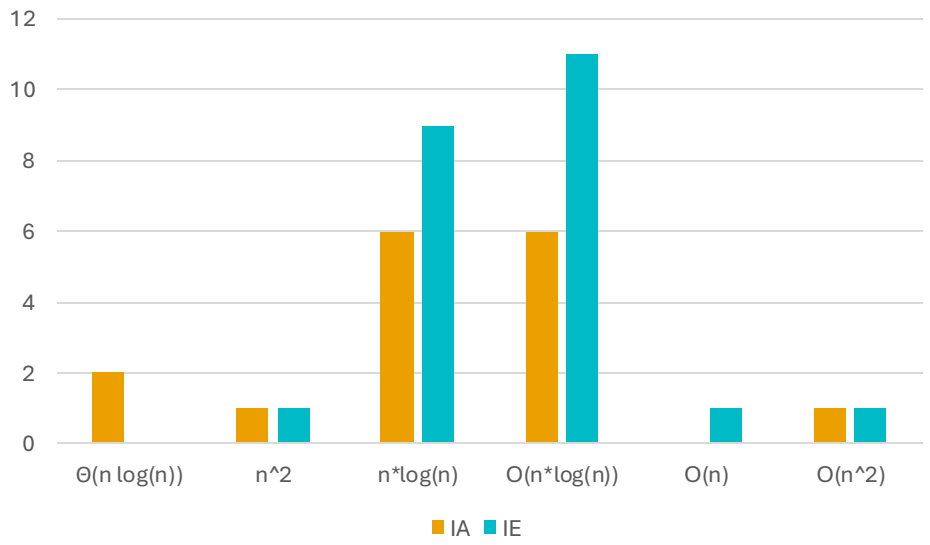
ALGORITHMS AND DATA STRUCTURES I

Course 4

Course 4. Quiz

The complexity order for the following sequence is?



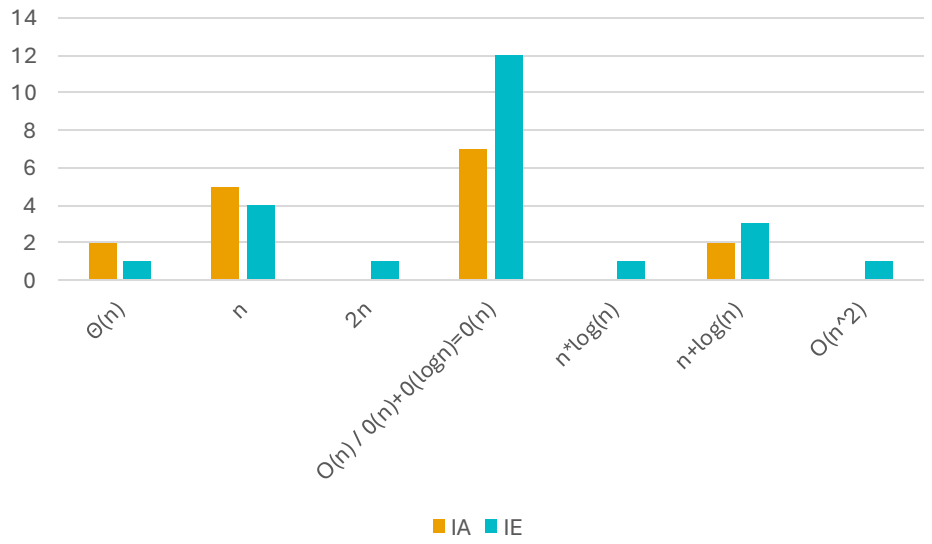

```
for i ← 1,n do          ← n times
  for j ← 1,n,j*2 do    ← log(n) times
    operation() // cost Θ(1)
  endfor
endfor
```

T(n) = n * log(n)

Θ(n*log(n))

Course 4. Quiz

The complexity order for the following sequence is?




```
for i ← 1,n do
 operation() // cost Θ(1)
endfor
```

n times

```
for j ← 1,n,j*2 do
   operation() // cost Θ(1)
endfor
```

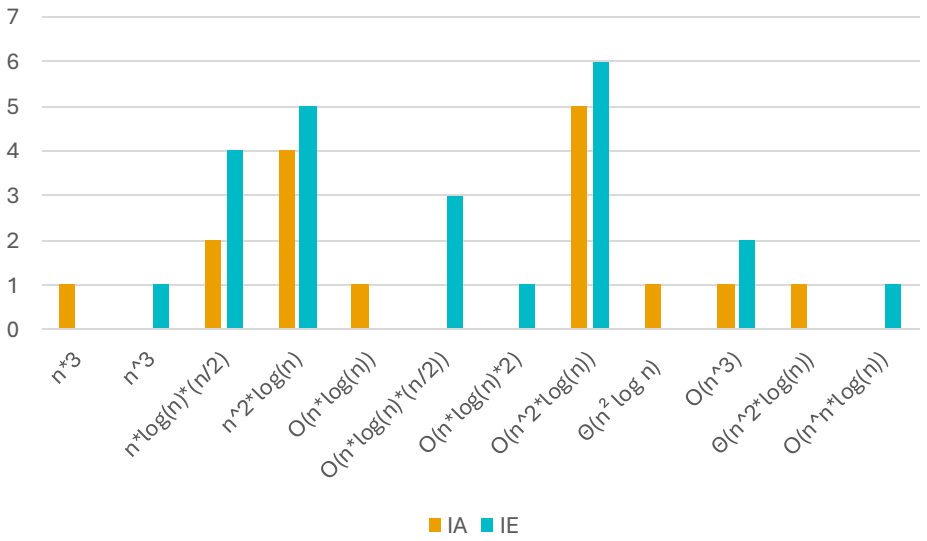
Log(n) times

T(n) = n +log(n)

Θ(n)

Course 4. Quiz

The complexity order for the following sequence is?




```
for i ← 1,n do          ← n times
  for j ← 1,n,j*2 do    ← log(n) times
    for k ← 1,n,k+2 do  ← n/2 times
      operation() // cost Θ(1)
    endfor
  endfor
endfor
```

$T(n) = n * \log(n) * (n/2) = (n^2 * \log(n))/2$

$\Theta(n^2 * \log(n))$

Previous course

… The main purpose of analyzing the efficiency of algorithms is to determine how the execution time of the algorithm grows with increasing the problem size

… In order to obtain this information, it is not necessary to know the detailed expression of the execution time, but it is sufficient to identify:

- Order of growth in execution time (relative to the size of the problem)
- The class of efficiency (complexity) to which the algorithm belongs

Today Course

- Analysis of algorithms
- Basic notations
- Steps in algorithm correctness checking
- Rules in algorithm correctness checking
- Examples

Analysis of algorithms

There are two main ways to verify the correctness of an algorithm:

- Experimental (by testing): the algorithm is executed for a set of input data instances
  - Most of the time it is impossible to analyze all possible cases

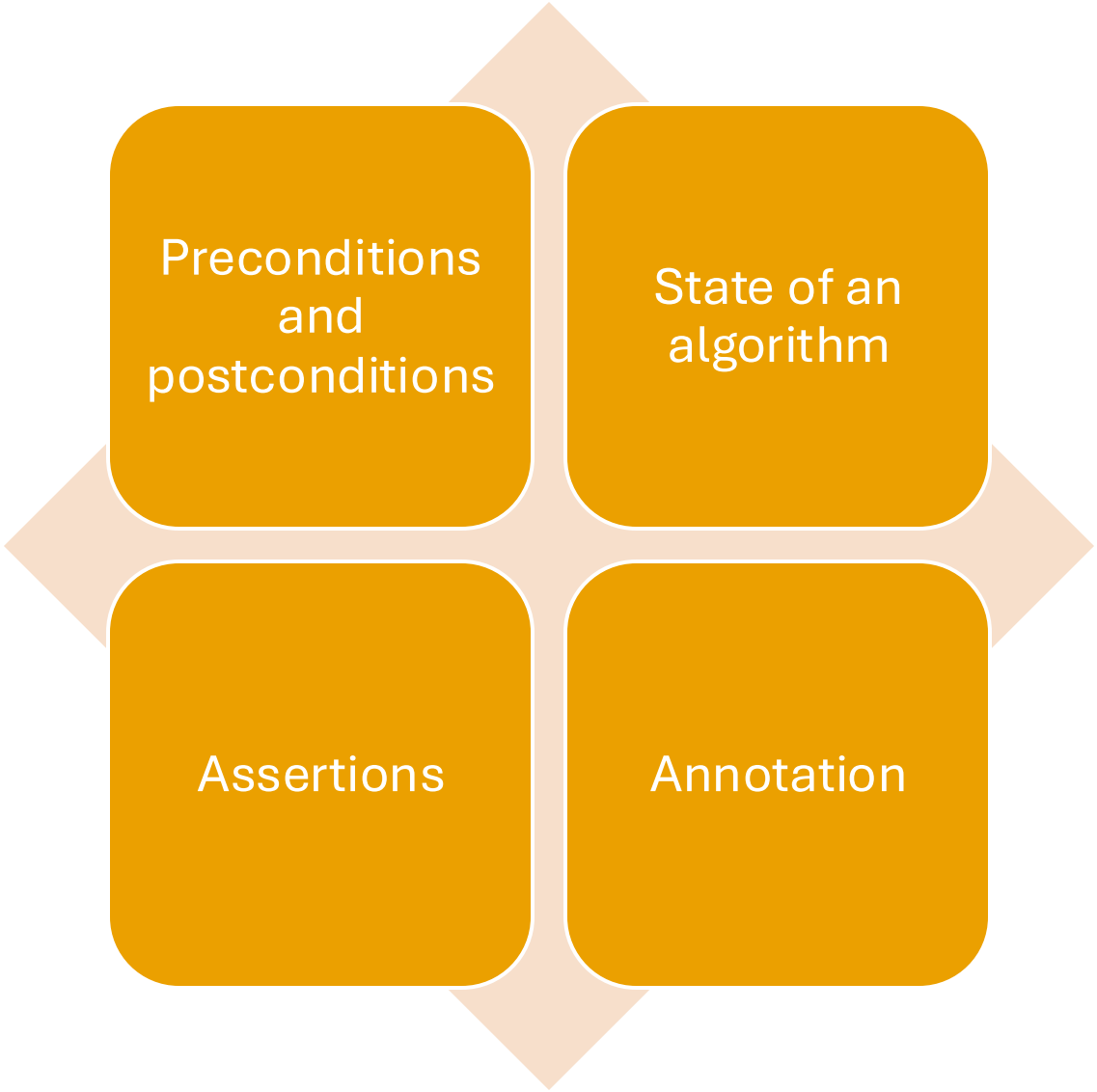
- Formal (by demonstration): it is demonstrated that the algorithm produces the correct result for any input data instance that satisfies the requirements of the problem

Analysis of algorithms
Advantages and Disadvantages

	Experimental	Formal
Advantages	• Simple • Relatively easy to apply	• Guarantee correctness
Disadvantages	• Does not guarantee correctness	• quite difficult • cannot be applied to complex algorithms

Remark: in practice a hybrid variant is used in which verification by testing can be guided by partial results obtained by formal analysis of small modules

Basic notations



Preconditions and postconditions

- Preconditions = properties satisfied by the input data
- Postconditions = properties satisfied by the output data (results)

- Example: Determine the minimum value, m, of a non-empty sequence (array), x[1..n]
  - Preconditions: n>=1 (the sequence is non-empty)
  - Postconditions: m=min{x[i]; 1<=i<=n} (or m <=x[i] for any i)

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

Partial correctness / Total correctness

Partial correctness = it is proven that if the algorithm terminates after a finite number of processings then it leads from preconditions to postconditions

Total correctness = partial correctness + finiteness

Intermediate steps in correctness verification:

– analysis of the algorithm state

– the effect that each processing step has on the algorithm state

Algorithm state

Algorithm state = set of values corresponding to the variables used within the algorithm

- Throughout the execution of the algorithm, its state changes as the variables change their values

- The algorithm can be considered correct if at the end of the execution its state implies the postconditions (i.e. the variables corresponding to the output data contain the correct values)

Algorithm state

Example: Compute the equation a*x=b, a≠0

Input data: a, b

Output data: x

Preconditions: a,b real values, a≠0

Postconditions: x satisfies a*x=b

Algorithm:

```
solve(real a,b)
  real x
   x←b/a
return x
```

Call:

solve(a_0,b_0)

Algorithm state

Example: Compute the equation a*x=b, a≠0

Input data: a, b

Output data: x

Preconditions: a,b real values, a≠0

Postconditions: x satisfies a*x=b

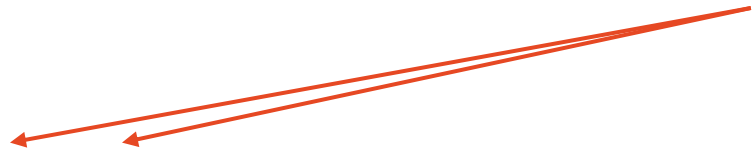
Algorithm:

and b

```
solve(real a,b)
 real x
  x←b/a
return x
```

Algorithm state

Current values for variables a



$a = a_0$ $b=b_0$ x undefined

$a = a_0$ $b=b_0$ $x = b_0/a_0$

Call:

solve(a_0,b_0)

Assertion

Assertion = (true) statement about the state of the algorithm

- Assertions are used to annotate algorithms

- Annotation is useful both in
  - Checking the correctness of algorithms
  - A tool for documenting and debugging programs

Remark: programming languages allow the specification of assertions and the generation of exceptions if these assertions are not satisfied. In Python, assertions are specified by assert

Annotate algorithms

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

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

Annotate algorithms

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

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

Today Course

- Analysis of algorithms
- Basic notations
- Steps in algorithm correctness checking
- Rules in algorithm correctness checking
- Examples

Steps in algorithm correctness checking

- Identifying preconditions and postconditions

- Annotate the algorithm with assertions such that
  - Preconditions are satisfied
  - Final assertion implies postconditions

- Demonstrate that each processing step ensures that the state of the algorithm is modified so that the next assertion is true

Some notations

Lets consider the following notations

P – preconditions
Q – postconditions
A – algorithm

The triplet (P,A,Q) represents a correct algorithm if for input data satisfying preconditions P, the algorithm:

- Leads to postconditions Q
- Stops after a finite number of processings

Notation:

Rules for algorithm correctness checking

To prove that an algorithm is correct, some rules specific to the main types of processing can be useful:

- Sequential processing
- Decision processing (conditional or branching)
- Repetitive processing

Rule for sequential processing

Example: Let x and y be two variables with values a and b. Interchange the values of the two variables.

- P: {x=a, y=b}
- Q: {x=b, y=a}

What can you say about the following algorithm?

Swap (integer x, y)	{x=a, y=b}
x ← y	{x=b, y=b}
y ← x	{x=b, y=b} => Q

- This variant does not satisfy the problem specifications!

Rule for sequential processing

Structure

A:

$\{P_0\}$

A1

$\{P_1\}$

…

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

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

$P_n \Rightarrow Q$

THEN

$P \xrightarrow{A} Q$

How do we interpret ?

IF

- The preconditions imply the initial assertion P_0
- Each action (A_i) implies the next assertion (P_i)
- The final assertion implies the postconditions

THEN

- the sequence of processing is correct

Rule for conditional processing

Example: Let x and y be two variables with values a and b. Find the minimum value of the two variables.

- Preconditions (P): a ≠b
- Postconditions (Q): m=min{a,b}

{a ≠ b}

IF a<b

THEN

{a<b, m undefined}

m ← a

{a<b, m=a}

ELSE

{b<a, m undefined}

m ← b

{b<a, m=b}

IF

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

AND

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

Then the algorithm satisfies the specification

Rule for sequential processing

Structure

A:

$\{P_0\}$

IF c

THEN

$\{c, P_o\}$

A_1

$\{P_1\}$

ELSE

$\{NOT\ c, P_o\}$

A_2

$\{P_2\}$

Rule

IF

c is well defined

c AND $P_0 \xrightarrow{A_1} P_1$

P_1 => Q

AND

NOT c AND $P_0 \xrightarrow{A_2} P_2$

P_2 => Q

THEN

$P \xrightarrow{A} Q$

How do we interpret ?

The condition c (logical expression) is considered well defined if it can be evaluated.

Both branches of the structure lead to postconditions.

Rule for repetitive processing

What values are stored in variables x and y at the end of the algorithm ?

```
x = a
y = b
while y>0 do
    x = x+1
    y = y-1
endwhile
```

x	y	x+y
a	b	a+b
a+1	b-1	a+b
a+2	b-2	a+b
…	…	…
a+b	0	a+b

x+y=a+b after each execution of the loop body

when exiting the loop: y=0 => x=a+b

x+y=a+b is called an invariant property associated with repetitive processing

Invariante property

Lets consider the following WHILE structure

Preconditie P

WHILE c DO

A

ENDWHILE

Postconditie Q

Invariant property

Lets consider the following WHILE structure

P => {I}

WHILE c DO

{c, I}

A

{I}

ENDWHILE

{NOT c, I} => Q

Definition:

An invariant property is a statement about the algorithm state (assertion) that satisfies the following conditions:

1. It is true upon entry into the loop
2. It remains true through the execution of the body of the loop
3. When c becomes false (upon exit from the loop) the property implies the postconditions

If an invariant property can be identified for a cycle then the cycle is considered partially correct

Invariante property

- Precondition: x[1..n] on empty array (n>=1)
- Postcondition : m=min{x[i] | 1<=i<=n}

```
i ← 1
m ← x[i]

WHILE i<n DO

  i ← i+1

  IF x[i]<m THEN  m ← x[i]

ENDIF
ENDWHILE
```

How to identify the invariant property?

Invariant property

P: n>=1

Q: m=min{x[i] | 1<=i<=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

Invariant:

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

Why? Because …

- When i=1 and m=x[1] the property considered invariant is true
- After the execution of the loop body the property m=min{x[j]; j=1..i} remains true
- Upon exiting the loop (when i=n) the invariant property becomes m=min{x[j]; j=1..n} which is exactly the postcondition

Invariant property

Problem: let n be a non zero natural number. Compute the sum of the digits of the number

Example: for n=5482 the result is 2+8+4+5=19

P: $n>=1$, $n=c_kc_{k-1}...c_1c_0$

Q: $s=c_k+c_{k-1}+...+c_1+c_0$

```
s ← 0
WHILE n != 0 DO
  d ← n MOD 10 //extract the last digit from the number
  s ← s+d //add the extracted digit to the sum
  n ← n DIV 10 // remove the digit from number n
ENDWHILE
```

Algorithm status analysis:

- Before entering the loop (p=0): {d=?, s = 0, $n=c_kc_{k-1}...c_1c_0$ }
- After the first execution of the loop body (p=1): {$d=c_0$, $s=c_0$, $n=c_kc_{k-1}...c_1$}
- After the second execution of the loop body (p=2): {$d=c_1$, $s=c_0+c_1$, $n=c_kc_{k-1}...c_2$}

...

What is the invariant property?

Invariant property

Problem: let n be a non zero natural number. Compute the sum of the digits of the number

Example: for n=5482 the result is 2+8+4+5=19

P: $n>=1$, $n=c_kc_{k-1}...c_1c_0$

Q: $s=c_k+c_{k-1}+...+c_1+c_0$

```
s ← 0
WHILE n != 0 DO
  d ← n MOD 10 //extract the last digit from the number
  s ← s+d //add the extracted digit to the sum
  n ← n DIV 10 // remove the digit from number n
ENDWHILE
```

Algorithm status analysis:

- (p=0): {d=?, s = 0, $n=c_kc_{k-1}...c_1c_0$ }
- (p=1): {$d=c_0$, $s=c_0$, $n=c_kc_{k-1}...c_1$}
- (p=2): {$d=c_1$, $s=c_0+c_1$, $n=c_kc_{k-1}...c_2$}

...

What is the invariant property?

I: {$s=c_0+c_1+...+ c_{p-1}$, $n=c_kc_{k-1}...c_p$}

Remark: p is a variable (implicit) that counts the number of executions of the loop (loop counter)

Invariant property

Problem: let n be a non zero natural number. Compute the sum of the digits of the number

P: $n>=1$, $n=c_k c_{k-1}...c_1 c_0$ Q: $s=c_k+c_{k-1}+...+c_1+c_0$

$s \leftarrow 0$

WHILE n != 0 DO

$d \leftarrow n$ MOD 10

$s \leftarrow s+d$

$n \leftarrow n$ DIV 10 ENDWHILE

Algorithm status analysis:

- (p=0): $\{d=?, s = 0, n=c_k c_{k-1}...c_1 c_0\}$
- (p=1): $\{d=c_0, s=c_0, n=c_k c_{k-1}...c_1\}$
- (p=2): $\{d=c_1, s=c_0+c_1, n=c_k c_{k-1}...c_2\}$

Invariant property

I: $\{s=c_0+c_1+...+c_{p-1}, n=c_k c_{k-1}...c_p\}$

Verify:

P⟶I: $p=0, s=0, n=c_k c_{k-1}...c_1 c_0 \rightarrow$ I

I remains true after loop execution (step p+1):

I (step p)⟶ $\{s=c_0+c_1+...+c_{p-1}, n=c_k c_{k-1}...c_p\}$

$\rightarrow \{s=c_0+c_1+...+c_{p-1}+c_p, n=c_k c_{k-1}...c_{p+1}\}$

$\rightarrow$ I (step p+1)

I⟶Q (when the loop finishes)

$n=0 \rightarrow p=k+1 \rightarrow s=c_0+c_1+...+c_{p-1}=c_0+c_1+...+c_k$

Invariant property

Problem: Let x[1..n] be an array containing the value x0. Determine the smallest value of i for which x[i]=x0 (the index of the first position where the value x0 is located)

P: n>=1 and there exists1<= 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
RETURN i
```

Invariant property:

{x[j] ≠ x0 for j=1..i-1}

Why? Because …

- if i=1 then the range of values for j (j=1..0) is empty so any statement regarding j in this range is true
- We assume that x[i]≠x0 and the invariant is true. Then x[j] ≠ x0 for j=1..i
- After i ← i+1 we obtain that x[j] ≠ x0 for j=1..i-1
- Finally, when x[i]=x0 we get postconditions

Invariant property

Invariant properties are not only useful for checking correctness but also for designing cycles

Ideally, when designing a cycle, you should:

- first identify the invariant property
- then construct the cycle

Example: calculate the sum of the first *n* natural values

Precondition: n>=1 Postcondition: S=1+2+...+n

What property should S satisfy after the i-th execution of the cycle body?

Invariant property

Invariant properties are not only useful for checking correctness but also for designing cycles

Ideally, when designing a cycle, you should:

- first identify the invariant property
- then construct the cycle

Example: compute the sum of the first *n* natural values

Precondition: n>=1 Postcondition: S=1+2+...+n

What property should S satisfy after the i-th execution of the cycle body?

Invariant: S=1+2+...+i

Idea for designing the cycle:

- First prepare the term to be added
- Then add the term to the sum

Invariant property

Problem: compute the sum of the first *n* natural values

Algoritm

```
i ← 1
S ← 1
WHILE i<n DO
  i ← i+1
  S ← S+i
ENDWHILE
```

Algoritm

```
S ← 0
i ← 1
WHILE i<=n DO
  S ← S+i
  i ← i+1
ENDWHILE
```

Invariant property

Algoritm:

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+2+..+n

Algoritm:

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+2+..+n

Termination functions

To prove the finiteness of a repetitive process of type

```
WHILE c DO
    process
ENDWHILE
```

it is sufficient to identify a termination function

Definition:

A function F:N→N (which depends on the implicit cycle counter) is a termination function if it satisfies the following properties:

i. F is strictly decreasing

ii. If the continuation condition c is true then F(p)>0

iii. If F(p)=0 then c is false

Termination functions

Remark:

- F depends on the (implicit) counter of the loop (denoted hereafter by p). At the first execution of the loop body p is 1, at the second execution of the loop body it is 2, etc.)

- F being strictly decreasing and taking natural values, will reach0 and when it becomes 0 the continuation condition (condition c) becomes false, so the loop ends.

Termination functions

Problem: S=1+2+..+n

```
i ← 1
S ← 1
WHILE i<n DO
  i ← i+1

  S ← S+i
ENDWHILE
```

i_p represents the value of the i variable at the pth execution of the loop body

$\{i_p=i_{p-1}+1\}$

```
S ← 0
i ← 1
WHILE i<=n DO
  S ← S+i
  i ← i+1

ENDWHILE
```

$\{i_p=i_{p-1}+1\}$

Termination functions

Problem: S=1+2+..+n

i_p represents the value of the i variable at the pth execution of the loop body

```
i ← 1
S ← 1
WHILE i<n DO
  i ← i+1
                    {i_p=i_p-1+1}
  S ← S+i
ENDWHILE
```

$F(p)=n-i_p$

$F(p)=n-i_{p-1}-1=F(p-1)-1<F(p-1)$

$i<n \Rightarrow F(p)>0$

$F(p)=0 \Rightarrow i_p=n$

```
S ← 0
i ← 1
WHILE i<=n DO
  S ← S+i
  i ← i+1
                                {i_p=i_p-1+1}
ENDWHILE
```

$F(p)=n+1-i_p$

$F(p)=n+1-i_{p-1}-1=F(p-1)-1<F(p-1)$

$i<=n \Rightarrow F(p)>0$

$F(p)=0 \Rightarrow i_p=n+1$

Termination functions

Example: Let x[1..n] be an array containing the value x0 in at least one position; determine the smallest index k for which x[k]=x0.

```
i ← 1
WHILE x[i] != x0 DO
    i ← i+1
ENDWHILE
```

$\{i_p=i_{p-1}+1\}$

Let k be the first occurrence of x0 in x[1..n]

Termination functions:

$F(p)=k-i_p$

Property verification:

(i) $F(p)=k-i_{p-1}-1=F(p-1)-1<F(p-1)$

(ii) $x[i]\neq x0 \Rightarrow i_p<k \Rightarrow F(p)>0$

$F(p)=0 \Rightarrow i_p=k \Rightarrow x[i]=x0$

Termination functions. Example

Problem: Analysis of the correctness of Euclid's algorithm (variant 1)

P: $a=a_0$, $b=b_0$, a_0, b_0 are natural numbers

Q: i=gcd(a0,b0)

```
gcd(a,b)
 d ← a
 i ← b
 r ← d MOD i
 WHILE r != 0 DO
   d ← i
   i ← r
   r ← d MOD i
 ENDWHILE
 RETURN i
```

Invariant property: $gcd(d,i)=gcd(a_0,b_0)$

1. $d=a=a_0$, $i=b=b_0$ => $gcd(d,i)=gcd(a_0,b_0)$
2. $gcd(d_p,i_p)=gcd(i_p,d_p \text{ MOD } i_p)= gcd(d_{p+1},i_{p+1})$
3. r=0 => i divides d => gct(d,i)=I

Termination functions: $F(p)=r_p$

Termination functions. Example

Problem: Analysis of the correctness of Euclid's algorithm (variant 2)

P: $a=a_0$, $b=b_0$, a_0, b_0 are natural numbers

Q: i=gcd(a0,b0)

```
gcd(a,b)
 WHILE a != 0 AND b != 0 DO
     a←a MOD b
     IF a!=0 THEN
         b ← b MOD a
      ENDIF
 ENDWHILE
 IF a!=0 THEN RETURN a
 ELSE RETURN b
 ENDIF
```

Invariant property: $gcd(a,b)=gcd(a_0,b_0)$

1. $p=0 \Rightarrow a=a_0, b=b_0 \Rightarrow gcd(a,b)=gcd(a_0,b_0)$
2. $gcd(a_0,b_0)=gcd(a_{p-1},b_{p-1}) \Rightarrow gcd(a_{p-1},b_{p-1})=gcd(b_{p-1},a_p)=gcd(a_p,b_p)$
3. $a_p=0 \Rightarrow$ cmmdc $(a,b)=b_p$

   $b_p=0 \Rightarrow$ cmmdc $(a,b)=a_p$

Termination functions: $F(p)=\min\{a_p,b_p\}$

- $(b_0>a_1>b_1>a_2>b_2>\ldots. \Rightarrow F(p)$ decrese.)

Termination functions. Example

Problem: Analysis of the correctness of Euclid's algorithm (variant 3)

P: $a=a_0$, $b=b_0$, a_0, b_0 are natural numbers

Q: i=gcd(a0,b0)

```
gcd(a,b)
  WHILE a != b
    IF a>b THEN a ← a-b
    ELSE b ← b-a
    ENDIF
  ENDWHILE
  RETURN a
```

Invariant property: $gcd(a,b)=gcd(a_0,b_0)$

1. $p=0$, $a=a_0$, $b=b_0$ => $gcd(a,b)=gcd(a_p,b_p)$

2. $gcd(a,b)=gcd(a_p,b_p)$=>

IF $a_{p-1} > b_{p-1}$

$gcd(a_{p-1},b_{p-1})= gcd\ (a_{p-1}-b_{p-1},b_{p-1}) = gcd\ (a_p,b_p)$

ELSE

$gcd(a_{p-1},b_{p-1})= gcd\ (a_{p-1},b_{p-1}-a_{p-1}) = gcd\ (a_p,b_p)$

3. $a_p=b_p$ => $gcd\ (a,b)= gcd\ (a_p,b_p)=a_p$

Summary

Verifying the correctness of algorithms involves:

- Proving that by executing the instructions, one gets from the preconditions to the postconditions (partial correctness)
- Proving that the algorithm terminates after a finite number of steps

A cycle invariant is a property (relating to the state of the algorithm) that satisfies the following conditions:

- It is true before entering the cycle
- It remains true through the execution of the cycle body
- At the end of the cycle, it implies the postconditions

Next course

Correctness verification and efficiency analysis of sorting algorithms

- Insertion sort
- Selection sort
- Neighbor swap sort

Q&A