LECTURE 2:

Algorithms

pseudocode; examples
Organizational:

**Webpage**: up and running.

**Newsgroup**: algouvt on yahoo groups. Please subscribe.

**First homework**: posted tomorrow on the webpage.

**DEADLINE (firm)**: Friday, October 19, 5pm.
Outline

• Continue with algorithms/pseudocode from last time.
• Describe some simple algorithms

• Decomposing problems in subproblems and algorithms in subalgorithms
Properties an algorithm should have

- Generality
- Finiteness
- Non-ambiguity
- Efficiency
Efficiency

An algorithm should use a reasonable amount of computing resources: memory and time.

Finiteness is not enough if we have to wait too much to obtain the result.

Example:
Consider a dictionary containing 50000 words.
Write an algorithm that takes a word as input and returns all anagrams of that word appearing in the dictionary.

Example of anagram: ship -> hips
Efficiency

First approach:

Step 1: generate all anagrams of the word
Step 2: for each anagram search for it in the dictionary (using binary search)

Let’s consider that:
- the dictionary contains $n$ words
- the analyzed word contains $m$ letters

Rough estimate of the number of basic operations:
- number of anagrams: $m!$
- words comparisons for each anagram: $\log_2 n$ (e.g. binary search)
- letters comparisons for each word: $m$

$m! \cdot m \log_2 n$
Efficiency

Second approach:

Step 1: sort the letters of the initial word
Step 2: for each word in the dictionary having m letters:
  • Sort the letters of this word
  • Compare the sorted version of the word with the sorted version of the original word

Rough estimate of the number of basic operations:
  – Sorting the initial word needs almost $m^2$ operations (e.g. insertion sort)
  – Sequentially searching the dictionary and sorting each word of length m needs at most $nm^2$ comparisons
  – Comparing the sorted words requires at most $nm$ comparisons

$$n \cdot m^2 + nm + m^2$$
Efficiency

Which approach is better?

First approach

\[ m! \cdot m \log_2 n \]

Second approach

\[ n \cdot m^2 + n \cdot m + m^2 \]

Example: \( m = 12 \) (e.g. word algorithmics)

\( n = 50000 \) (number of words in dictionary)

\[ 8 \cdot 10^{10} \]

\[ 8 \cdot 10^{6} \]

one basic operation (e.g. comparison) = \( 1 \text{ms} = 10^{-3} \text{s} \)

24000 hours

2 hours

Thus, important to analyze efficiency and choose more efficient algorithms.
Outline

• Problem solving
• What is an algorithm?
• Properties an algorithm should have
• Describing Algorithms
• Types of data to use
• Basic operations
How can we describe algorithms?

Solving problems can usually be described in *mathematical language*

Not always adequate to describe algorithms because:

- Operations which seem elementary when described in a mathematical language are not elementary when they have to be encoded in a programming language.

**Example:** computing a sum, computing the value of a polynomial

Mathematical description

\[
\sum_{i=1}^{n} i = 1 + 2 + \ldots + n
\]

Algorithmic description

(it should be a sequence of basic operations)
How can we describe algorithms?

Two basic instruments:

- **Flowcharts:**
  - graphical description of the flow of processing steps
  - not used very often, somewhat *old-fashioned*.
  - however, sometimes useful to describe the overall structure of an application

- **Pseudocode:**
  - artificial language based on
    - *vocabulary* (set of keywords)
    - *syntax* (set of rules used to construct the language’s “phrases”)
  - not as restrictive as a programming language
Why do we call it pseudocode?

Because …

- It is similar to a programming language (code)
- Not as rigorous as a programming language (pseudo)

In pseudocode the phrases are:

- Statements or instructions (used to describe processing steps)
- Declarations (used to specify the data)
Types of data

Data = container of information

Characteristics:
- name
- value
  - constant (same value during the entire algorithm)
  - variable (the value varies during the algorithm)
- type
  - primitive (numbers, characters, truth values …)
  - structured (arrays)
Types of data

Arrays - used to represent:

• **Sets** (e.g. \(\{3,7,4\} = \{3,4,7\}\))
  - the order of the elements doesn’t matter

• **Sequences** (e.g. \((3,7,4)\) is not \((3,4,7)\))
  - the order of the elements matters

• **Matrices**
  - bidimensional arrays
    \[
    \begin{pmatrix}
    1 & 0 \\
    0 & 1 \\
    \end{pmatrix}
    \]
How can we specify data?

• Simple data:
  
  – Integers  
    INTEGER <variable>
  
  – Reals  
    REAL <variable>
  
  – Boolean  
    BOOLEAN <variable>
  
  – Characters  
    CHAR <variable>
How can we specify data?

Arrays

One dimensional

<elements type> <name>[n1..n2]
(ex: REAL x[1..n])

Two-dimensional

<elements type> <name>[m1..m2, n1..n2]
(ex: INTEGER A[1..m,1..n])
How can we specify data?

Specifying elements:

- One dimensional
  \[ x[i] \quad \text{- i is the element's index} \]

- Two-dimensional
  \[ A[i,j] \quad \text{- i is the row's index, while j is the column's index} \]
How can we specify data?

Specifying subarrays:
- **Subarray** = contiguous portion of an array
  - One dimensional: \( x[i_1..i_2] \) \((1 \leq i_1 < i_2 \leq n)\)
  - Bi dimensional: \( A[i_1..i_2, j_1..j_2] \)
    \( (1 \leq i_1 < i_2 \leq m, 1 \leq j_1 < j_2 \leq n)\)
Outline

• Problem solving
• What is an algorithm?
• Properties an algorithm should have
• Describing Algorithms
• Types of data to use
• Basic instructions
What are the basic instructions?

Instruction (statement)
  = action to be executed by the algorithm

There are two main types of instructions:
  – Simple
    • Assignment (assigns a value to a variable)
    • Transfer (reads an input data; writes a result)
    • Control (specifies which is the next step to be executed)
  – Structured ....
Assignment

- **Aim**: give a value to a variable
- **Description**:

  \[ v \leftarrow \text{<expression>} \]

  **Rmk**: sometimes we use `:=` instead of `\leftarrow`

- **Expression** = syntactic construction used to describe a computation

  It consists of:
  - **Operands**: variables, constant values
  - **Operators**: arithmetical, relational, logical
Operators

• **Arithmetical:**
  
  + (addition), - (subtraction), *(multiplication),
  
  / (division), ^ (power),
  
  DIV (from divide) or / (integer quotient),
  
  MOD (from modulo) or % (remainder)

• **Relational:**
  
  = (equal), != (different),
  
  < (less than), <= (less than or equal),
  
  >(greater than) >= (greater than or equal)

• **Logical:**
  
  OR (disjunction), AND (conjunction), NOT (negation)
Input/Output

• **Aim:**
  – read input data
  – output the results

• **Description:**

  read v1, v2, …  
  write e1, e2, …

input v1, v2, …  
print e1, e2, …
Instructions

Structured:
– **Sequence** of instructions
– **Conditional** statement
– **Loop** statement
Conditional statement

- **Aim:** choosing between two or several alternatives depending on the value of some conditions

- **General variant:**
  
  ```
  if <condition> then <S1>
  else <S2>
  endif
  ```

- **Simplified variant:**
  
  ```
  if <condition> then <S>
  endif
  ```
Loop statements

• **Aim:** repeating a processing step
• **Example:** compute a sum
  \[ S = 1 + 2 + \ldots + i + \ldots + n \]
• **Characterized by:**
  – Processing step which have to be repeated
  – Stopping (or continuation) condition

• Depending on the moment of analyzing the stopping condition there are two main loop statements:
  – **Preconditioned loops** (WHILE loops)
  – **Postconditioned loops** (REPEAT loops)
WHILE loop

- First, the condition is analyzed
- If it is true then the statement is executed and the condition is analyzed again
- If the condition becomes false the control of execution passes to the next statement in the algorithm
- If condition never becomes false then the loop is infinite
- If the condition is false from the beginning then the statement inside the loop is never executed

```
while <condition> do
  <statement>
endwhile
```
WHILE loop

```
while <condition> do
    <statement>
endwhile
```

```
S:=0 // initialize the variable which will contain the result
i:=1 // index initialization
while i<=n do
    S:=S+i // add the current term to S
    i:=i+1 // prepare the next term
endwhile
```

\[
\sum_{i=1}^{n} i = 1+2+\ldots+n
\]
FOR loop

- Sometimes the number of repetitions of a processing step is known a priori
- Then we can use a counting variable which varies from an initial value to a final value using a step value
- Repetitions: \( v_2 - v_1 + 1 \) if \( \text{step} = 1 \)

\[
\text{for } v := v_1, v_2, \text{step} \text{ do}
\]
\[
\text{<statement>}
\]
\[
\text{endfor}
\]

\[
\text{while } v \leq v_2 \text{ do}
\]
\[
\text{<statement>}
\]
\[
\text{v := v + step}
\]
\[
\text{endwhile}
\]
FOR loop

\[ \sum_{i=1}^{n} i = 1 + 2 + \ldots + n \]

S:=0  // initialize the variable which will  
// contain the result

for i:=1,n do
  S:=S+i  // add the term to S
endfor

for v:=v1,v2,step do
  <statement>
endfor
### REPEAT loop

- **First**, the statement is executed. Thus it is executed at least once.

- Then the condition is analyzed and if it is false the statement is executed again.

- When the condition becomes true the control passes to the next statement of the algorithm.

- If the condition doesn’t become true then the loop is infinite.
REPEAT loop

\[ \sum_{i=1}^{n} i = 1 + 2 + \ldots + n \]

\[
\begin{align*}
S &:= 0 \\
i &:= 1 \\
\text{repeat} \\
S &:= S + i \\
i &:= i + 1 \\
\text{until } i > n
\end{align*}
\]

\[
\begin{align*}
S &:= 0 \\
i &:= 0 \\
\text{repeat} \\
i &:= i + 1 \\
S &:= S + i \\
\text{until } i \geq n
\end{align*}
\]
Any REPEAT loop can be transformed in a WHILE loop:

```
<statement>
while NOT <condition> DO
    <statement>
endwhile
```

```
repeat <statement> until <condition>
```
Summary

• Algorithms are step-by-step procedures for problem solving

• They should have the following properties:
  • Generality
  • Finiteness
  • Non-ambiguity (rigorousness)
  • Efficiency

• Data processed by an algorithm can be
  • simple
  • structured (e.g. arrays)

• We describe algorithms by means of pseudocode
Summary

• Pseudocode:

Assignment :=

Data transfer  read (input), write (print)

Decisions if ... then ... else ... endif

Loops while ... do ... endwhile
for ... do ... endfor
repeat ... until
Example 1

Consider a table containing info about student results

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Marks</th>
<th>ECTS</th>
<th>Status</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>-</td>
<td>7</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>60</td>
</tr>
</tbody>
</table>

Task: fill in the **status** and **average** fields such that

- **status = 1** if **ECTS**=60
- **status = 2** if **ECTS** belongs to [30,60)
- **status = 3** if **ECTS**<30

the average is computed only if **ECTS**=60
Example 1

The filled table should look like this:

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Marks</th>
<th>ECTS</th>
<th>Status</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>-</td>
<td>7</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>60</td>
</tr>
</tbody>
</table>
Example 1

What kind of data should we process?

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Marks</th>
<th>ECTS</th>
<th>Status</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>-</td>
<td>7</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>60</td>
</tr>
</tbody>
</table>

Input data: marks and ECTS
marks[1..5,1..3] : two dimensional array (matrix) with 5 rows and 3 columns
Pseudocode specification: integer marks[1..5,1..3]
Example 1

What kind of data should we process?

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Marks</th>
<th>ECTS</th>
<th>Status</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>-</td>
<td>7</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>60</td>
</tr>
</tbody>
</table>

**Input data:** marks and ECTS

**ects[1..5]:** one-dimensional array with 5 elements

**Pseudocode specification:**

```plaintext
integer ects[1..5]
```
Example 1

What kind of data should we process?

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Marks</th>
<th>ECTS</th>
<th>Status</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>-</td>
<td>7</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>60</td>
</tr>
</tbody>
</table>

Output data: status and average
status[1..5], average[1..5]: one-dimensional arrays with 5 elements

Pseudocode specification:
integer status[1..5]
real average[1..5]
Example 1

Rule to fill in the status of a student

\[ \text{status} = 1 \text{ if ECTS}=60 \]
\[ \text{status} = 2 \text{ if ECTS belongs to } [30,60) \]
\[ \text{status} = 3 \text{ if ECTS}<30 \]

Pseudocode description:

\[
\begin{align*}
\text{if } \text{ects}=60 \text{ then } & \text{status} \leftarrow 1 \\
\text{else if } \text{ects} \geq 30 \text{ then } & \text{status} \leftarrow 2 \\
\text{else} \text{ status} \leftarrow 3
\end{align*}
\]

Python description

\[
\begin{align*}
\text{if } \text{ects}=60: \\
\quad \text{status}=1 \\
\text{elif } \text{ects} \geq 30: \\
\quad \text{status}=2 \\
\text{else:} \\
\quad \text{status}=3
\end{align*}
\]
Example 1

Filling in the status of all students: for each student fill in the status field

Remark: Let us denote with n the number of students (in our example n=5)

Step 1: start from the first element (i:=1)
Step 2: check if there are still elements to process (i<=n); if not then STOP
Step 3: compute the status of element i
Step 4: prepare the index of the next element
Step 5: go to Step 2
Example 1

Filling in the status of all students: for each student fill in the status field

\[
i \leftarrow 1
\]

\[
i \leftarrow i + 1
\]

\[\text{i} \leq n\]

\[
\text{compute status}[i]
\]

\[
i \leftarrow i + 1
\]

Pseudocode:

\[
\text{integer ects}[1..n], \text{status}[1..n], i
\]

\[
i \leftarrow 1
\]

while \(i \leq n\) do

\[
\text{if ects}[i] = 60 \text{ then status}[i] \leftarrow 1
\]

\[
\text{else if ects}[i] \geq 30 \text{ then status}[i] \leftarrow 2
\]

\[
\text{else status}[i] \leftarrow 3
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
i \leftarrow i + 1
\]

endwhile
Example 1

Simplify the algorithm description by grouping some computation in “subalgorithms”

Pseudocode:
integer ects[1..n], status[1..n], i
i ← 1
while i<=n do
    status[i] ← compute(ects[i])
    i ← i+1
endwhile

Subalgorithm (function) description:
compute (integer ects)
integer s
if ects=60 then s ← 1
else if ects>=30 then s ← 2
    else s ← 3
endif
endif
return s

Remark: the subalgorithm describes a computation applied to generic data
Using subalgorithms

Basic ideas:

- Decompose the problem in subproblems
- Design for each subproblem an algorithm (called subalgorithm or module or function)
- The subalgorithm actions are applied to some generic data (called parameters) and to some additional data (called local variables)
- The execution of subalgorithm statements is ensured by calling the subalgorithm
- The effect of the subalgorithm consists of:
  • Returning some results
  • Modifying the values of some variables which are accessed by the algorithm (global variables)
Using subalgorithms

The communication mechanism between an algorithm and its subalgorithms:
- parameters and returned values

**Algorithm**
- Variables
- Local computations
- Call the subalgorithm
- Local computations

**Subalgorithm**
- Parameters:
  - input parameters
  - output parameters
- Local variables
- Computations on local variables and parameters
- Return results
Using subalgorithms

The communication mechanism between an algorithm and its subalgorithms:
- parameters and returned values

Algorithm

integer ects[1..n], status[1..n], i
i ← 1
while i<=n do
    status[i] ← compute(ects[i])
    i ← i+1
endwhile

Subalgorithm

compute (integer ects)
integer s
if ects=60 then s ← 1
else if ects>=30 then s ← 2
else s ← 3
endif
endif
return s
Using subalgorithms

The communication mechanism between an algorithm and its subalgorithms:
- parameters and returned values

Algorithm

```
integer ects[1..n], status[1..n], i
i ← 1
while i<=n do
    status[i] ← compute(ects[i])
    i ← i+1
endwhile
```

Subalgorithm

```
compute (integer ects)
    integer status
    if ects=60 then status ← 1
    else if ects>=30 then status ← 2
    else status ← 3
    endif
    endif
    return status
```
Using subalgorithms

The communication mechanism between an algorithm and its subalgorithms:
- parameters and returned values

Algorithm

int ects[1..n], status[1..n], i
i ← 1
while i <= n do
    status[i] ← compute(ects[i])
    i ← i + 1
endwhile

Subalgorithm

compute (int ects)
int status
if ects = 60 then status ← 1
else if ects >= 30 then status ← 2
else status ← 3
endif
endif
return status

Is it OK to use the variable status inside the subalgorithm? Yes, because it is a local variable.

Algorithmics - Lecture 2
Using subalgorithms

• Structure of a subalgorithm:

  \[ \text{<subalgorithm name> (<formal parameters>)} \]
  \[ \text{< declaration of local variables >} \]
  \[ \text{< statements}> \]
  \[ \text{RETURN <results> } \]

• Call of a subalgorithm:

  \[ \text{<subalgorithm name> (<actual parameters>)} \]
Back to Example 1

Pseudocode:

integer ects[1..n], status[1..n], i
i:=1
while i<=n do
    status[i] ← compute(ects[i])
i:=i+1
endwhile

Another variant

integer ects[1..n], status[1..n], i
for i:=1,n do
    status[i] ← compute(ects[i])
endfor

Subalgorithm (function) description:

compute (integer ects)
    integer status
    if ects=60 then status ← 1
    else if ects>=30 then status ← 2
    else status ← 3
    endif
    endif
    return status
Example 1: Python implementation

Python program:

```python
ects=[60,60,40,20,60]
status=[0]*5
n=5
i=0
while i<n:
    status[i]=compute(ects[i])
i=i+1
print status
```

Using a `for` statement instead of `while`:

```python
for i in range(5):
    status[i]=compute(ects[i])
```

Python function (module):

```python
def compute(ects):
    if ects==60:
        status=1
    elif ects>=30:
        status=2
    else:
        status=3
    return status
```

Remark: indentation is very important in Python
Example 1: computation of the average

Compute the averaged mark

<table>
<thead>
<tr>
<th>Integer marks[1..n,1..m], status[1..n]</th>
<th>ComputeAvg(integer values[1..m])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real avg[1..n]</td>
<td>Real sum</td>
</tr>
<tr>
<td></td>
<td>Integer i</td>
</tr>
<tr>
<td></td>
<td>Sum ← 0</td>
</tr>
<tr>
<td>...</td>
<td>For i ← 1,m do</td>
</tr>
<tr>
<td>For i ← 1,n do</td>
<td>Sum ← sum+values[i]</td>
</tr>
<tr>
<td>If status[i] = 1</td>
<td>Endfor</td>
</tr>
<tr>
<td>Avg[i] ← ComputeAvg(marks[i,1..m])</td>
<td>Sum ← sum/m</td>
</tr>
<tr>
<td>Endif</td>
<td>Return sum</td>
</tr>
</tbody>
</table>

Endfor
Example 1: computation of the average

Compute the averaged mark (Python example)

marks=[[8,6,7],[10,10,10],[0,7,5],[6,0,0],[8,7,9]]
status=[1,1,2,3,1]
avg=[0]*5

for i in range(5):
    if status[i]==1:
        avg[i]=computeAvg(marks[i])
print avg

Computation of an average (Python example)

def computeAvg(marks):
    m=len(marks)
    sum=0
    for i in range(m):
        sum = sum+marks[i]
    sum=sum/m
    return sum
Example 2 – greatest common divisor

Problem: Let a and b be two strictly positive integers. Find the greatest common divisor of a and b

Euclid’s method:

• compute \( r \), the remainder obtained by dividing \( a \) by \( b \)
• replace \( a \) with \( b \), \( b \) with \( r \), and start the process again
• the process continues until one obtains a remainder equal to zero
• then the previous remainder (which, obviously, is not zero) will be the \( \text{gcd}(a,b) \).
Example 2 - greatest common divisor

How does this method work?

1: \[ a = bq_1 + r_1, \quad 0 \leq r_1 < b \]
2: \[ b = r_1 q_2 + r_2, \quad 0 \leq r_2 < r_1 \]
3: \[ r_1 = r_2 q_3 + r_3, \quad 0 \leq r_3 < r_2 \]

... 

i: \[ r_{i-2} = r_{i-1} q_i + r_i, \quad 0 \leq r_i < r_{i-1} \]

... 

n-1: \[ r_{n-3} = r_{n-2} q_{n-1} + r_{n-1}, \quad 0 \leq r_{n-1} < r_{n-2} \]

n: \[ r_{n-2} = r_{n-1} q_n, \quad r_n = 0 \]

Remarks:

- at each step the dividend is the previous divisor and the new divisor is the old remainder
- the sequence of remainders is strictly decreasing, thus there exists a value \( n \) such that \( r_n = 0 \) (the method is finite)
- using these relations one can prove that \( r_{n-1} \) is indeed the gcd
Example 2 - greatest common divisor

The algorithm (WHILE variant):

```plaintext
integer a,b,dd,dr,r
read a,b
dd ← a
dr ← b
r ← dd MOD dr
while r<>0 do
    dd ← dr
dr ← r
    r ← dd MOD dr
endwhile
write dr
```

The algorithm: (REPEAT variant)

```plaintext
integer a,b,dd,dr,r
read a,b
dd ← a
dr ← b
repeat
    r ← dd MOD dr
dd ← dr
dr ← r
    r ← dd MOD dr
until r=0
write dd
```
Example 2 – gcd of a set of values

• **Problem:**
  Find the greatest common divisor of a sequence of non-zero natural numbers

• **Example:**
  \[
  \text{gcd}(12, 8, 10) = \text{gcd}(\text{gcd}(12, 8), 10) = \text{gcd}(4, 10) = 2
  \]

• **Basic idea:**
  compute the gcd of the first two elements, then compute the gcd between the previous gcd and the third element and so on …

  natural to use a (sub)algorithm for computing the gcd of two values
Example 2 – gcd of a set of values

- Structure of the algorithm:

```
gcd_sequence(INTEGER a[1..n])
INTEGER d,i
d ← gcd(a[1],a[2])
FOR i ← 3,n DO
  d ← gcd(d,a[i])
ENDFOR
RETURN d
```

```
gcd(integer a,b)
integer dd,dr,r
dd←a
dr ← b
r ← dd MOD dr
while r<>0 do
  dd ← dr
  dr ← r
  r ← dd MOD dr
endwhile
return dr
```
Example 3: The successor problem

Let us consider a natural number of 10 distinct digits. Compute the next number (in increasing order) in the sequence of all naturals consisting of 10 distinct digits.

Example: $x = 6309487521$

Next number consisting of different digits
6309512478
The successor problem

Step 1. Find the largest index $i$ having the property that $x[i-1] < x[i]$

Example: $x = 6309487521$ \quad i = 6$ (the pair of digits 4 and 8)

Step 2. Find the smallest element $x[k]$ in $x[i..n]$ which is larger than $x[i-1]$

Example: $x = 6309487521$ \quad k = 8$ (the digit 5 has this property)

Step 3. Interchange $x[k]$ with $x[i-1]$

Example: $x = 6309587421$ (this is a value larger than the first one)

Step 4. Sort $x[i..n]$ increasingly (in order to obtain the smaller number satisfying the requirements)

Example: $x = 6309512478$ (it is enough to reverse the order of elements in $x[i..n]$)
The successor problem

Subproblems / subalgorithms:

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

Input: $x[1..n]$
Output: $i$

Minimum: find the index of the smallest value in the subarray $x[i..n]$ which is larger than $x[i-1]$

Input: $x[i..n]$
Output: $k$

Sorting: reverse the order of elements of the subarray $x[i..n]$

Input: $x[i..n]$
Output: $x[i..n]$
The successor problem

The general structure of the algorithm:

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..n])
  x[i-1] ← x[k]
  x[i..n] ← Reverse(x[i..n])
  write x[1..n]
endif
The 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 $i$

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 ← $i$
for $j ← i+1,n$ do
    i ← $i-1$
    if $x[j]<x[k]$ and $x[j]>x[i-1]$ then
        k ← $j$
endfor
return $k$
The successor problem

Reverse the order of elements of a subarray of x

reverse (integer x[left..right])

integer i,j
i ← left
j ← right
while i<j DO
    x[i] ↔ x[j]
    i ← i+1
    j ← j-1
endwhile
return x[left..right]
The successor problem

Python implementation:

```python
def identify(x):
    n=len(x)
    i=n-1
    while (i>0) and (x[i-1]>x[i]):
        i=i-1
    return i

def minimum(x,i):
    n=len(x)
    k=i
    for j in range(i+1,n):
        if (x[j]<x[k]) and (x[j]>x[i-1]):
            k=j
    return k

def swap(a,b):
    aux=a
    a=b
    b=aux
    return a,b

def reverse(x,left,right):
    i=left
    j=right
    while i<j:
        x[i],x[j]=x[j],x[i]  # other type of swap
        i=i+1
        j=j-1
    return x
```
The successor problem

Python implementation:

```python
x=[6,3,0,9,4,8,7,5,2,1]
print "Digits of the initial number :",x
i=identify(x)
print "i=",i
k=minimum(x,i)
print "k=",k
x[i-1],x[k]=swap(x[i-1],x[k])
print "Sequence after swap:",x
x=reverse(x,i,len(x)-1)
print "Sequence after reverse:",x
```
Summary

• The problems are usually decomposed in smaller subproblems solved by subalgorithms

• A subalgorithm is characterized through:
  – A name
  – Parameters (input data)
  – Returned values (output data)
  – Local variables (additional data)
  – Processing steps

• Call of a subalgorithm:
  – The parameters values are set to the input data
  – The statements of the subalgorithm are executed
Next lecture will be on …

• how to verify the correctness of an algorithm

• some formal methods in correctness verification