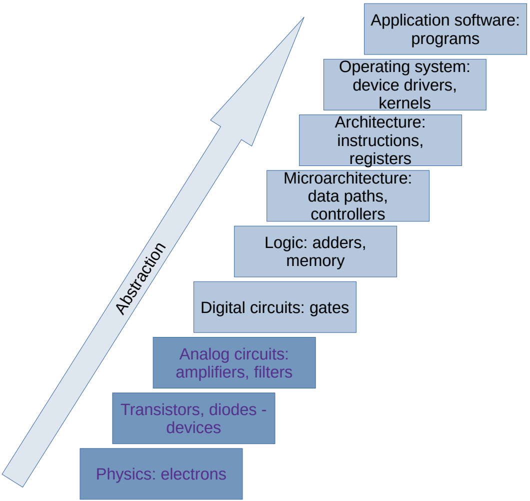
E2011: Theoretical fundamentals of computer science
Introduction to algorithms

Vlad Popovici, Ph.D.

Fac. of Science - RECETOX

Outline


Application software: programs
Operating system: device drivers, kernels
Architecture: instructions, registers
Microarchitecture: data paths, controllers
Logic: adders, memory
Digital circuits: gates
Analog circuits: amplifiers, filters
Transistors, diodes - devices
Physics: electrons
Abstraction

Algorithm

- a step-by-step procedure to solve a task/problem
- word *algorithm* originates from the Latin version of the name Muḥammad ibn Mūsā **al-Khwārizmī** (IX-th century) - a highly influential Arab mathematician

- an algorithm has an *input* and an *output*
- the procedure describes how the input is used to obtain the output
- attributes of an algorithm:
  - *correctness*
  - *efficiency*
  - *complexity*

Example - Greatest Common Divisor

(one of the oldest algorithms - Euclid (III-IV centuries BC))

Algorithm 1 Euclid's algorithm

Input: $a, b \in \mathbb{N}^*$
Output: GCD

```
r ← a mod b
while r ≠ 0 do                      ▷ We have the answer if r is 0
    a ← b
    b ← r
    r ← a mod b
end while
GCD ← b                             ▷ The gcd is b
```

while $r \neq 0$ **do**
    $a \leftarrow b$
    $b \leftarrow r$
    $r \leftarrow a \bmod b$
end while
GCD $\leftarrow b$

Example: GCD of $a = 72$ and $b = 120$

	r	a	b
before "while"	72	72	120
iteration 1	48	120	72
iteration 2	24	72	48
iteration 3	0	48	24

Pseudocode

- a means of describing an algorithm
- not directly interpretable by a computer; needs to be *implemented* in a *programming language*
- has a less strict syntax and vocabulary than a programming language
- it is independent on the programming language
- shortcuts are allowed if their meaning is clear (e.g. $\theta^* = \arg\min_\theta \Omega(\theta)$)
- describes the solution with enough granularity so it can be implemented

Main ingredients of the pseudocode

- *variables* - store some values (e.g. x, y); may refer to simple (e.g. scalar) values, or more complicated *data structures* (vectors, matrices, lists, etc.)
- *input* to specify the required values for the algorithm to compute the *output*
- variables are assigned values: $x \leftarrow 50$ or $x \leftarrow y$, but values are never assigned variables or other values: $50 \leftarrow x$ is a nonsense
- mathematical operators can be used as usual

Main ingredients of the pseudocode

If-then-else structure

- specifies the conditional execution of a part (branch) of the code
- the *else* part may be missing
- the $\langle condition \rangle$ is a Boolean predicate that is evaluated to either True or False (or, equivalently, to $\neq 0$ or 0.)

if $\langle condition \rangle$ **then**
    code for $\langle condition \rangle$ is True
else
    code for $\langle condition \rangle$ is False
end if

Main ingredients of the pseudocode

While-do loop

- specifies a repeated execution of a set of operations (*instruction block*)
- the block is executed as long as the ⟨*condition*⟩ is True and no forced exit is encountered
- if the condition is False at the very beginning, the block is not executed at all

while ⟨*condition*⟩ **do**
    instruction
    ...
end while

Main ingredients of the pseudocode

Repeat-until loop

- specifies a repeated execution of an instruction block
- the block is executed as long as the ⟨*condition*⟩ is False and no forced exit is encountered
- the block is executed at least once

```
repeat
    instruction
    ...
until ⟨condition⟩
```

Main ingredients of the pseudocode

For loop

- executes a block for a given number of steps (unless forced exit is encountered)
- typically used with vectors, lists, etc

for ⟨*iterator*⟩ **do**
    instructions
end for
for all ⟨*iterator*⟩ **do**
    instructions
end for

Examples:

$sum \leftarrow 0$
for $i = 1, \ldots, n$ **do**
    $sum \leftarrow sum + x_i$
end for
for all $a \in A$ **do**
    print(a)
end for

Main ingredients of the pseudocode

Procedures

- groups a set of instructions into a construct that can be invoked (*called*) with or without parameters
- the parameters may function as input/output or in/out parameters

procedure ⟨*name*⟩(⟨*params*⟩)
    block
end procedure

Functions

- special procedures with only input parameters which *returns* a value
- much like the mathematical equivalent (e.g. $\sin(x)$)

function ⟨*name*⟩(⟨*params*⟩)
    body
    return value
end function

Main ingredients of the pseudocode

- **continue**: used in loops; indicates a jump to the test condition, any instructions after it are not executed
- **break**: used in loops; indicates an exit from the loop, continuing execution with the instruction after the loop

$\sqrt{x}$ with precision $\epsilon > 0$ - binary search

Algorithm 2 $\sqrt{x}$ via binary search

```
function SQRT1(x ∈ ℝ+, ε > 0)
    low ← 0
    if x > 1 then
        high ← x
    else
        high ← 1
    end if
    while high − low > ε do
        middle = 0.5(high − low)
        if middle² > x then
            high ← middle
        else
            low ← middle
        end if
    end while
    return low
end function
```

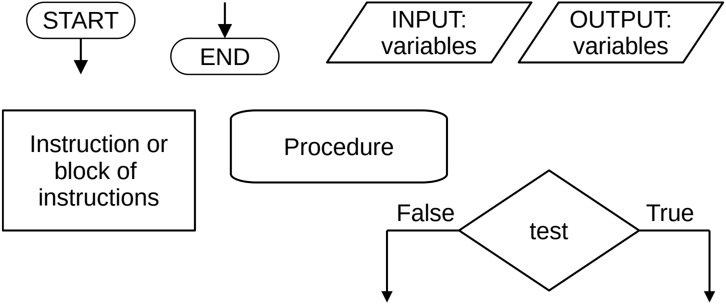
$\sqrt{x}$ with precision $\epsilon > 0$ - Babylonian method

Algorithm 3 $\sqrt{x}$ - Babylonian algorithm

```
function SQRT2(x ∈ ℝ+, ε > 0)
    r0 ← x/2                                  ▷ some initial guess
    r1 ← (r0 + x/r0)/2
    while |r1 − r0| > ε do
        r0 ← r1
        r1 ← (r0 + x/r0)/2
    end while
    return r1
end function
```

- there might be several algorithms for a given problem
- the choice of the "best" algorithm is not always obvious
- execution time, memory requirements, implementation options, etc are factors to keep in mind

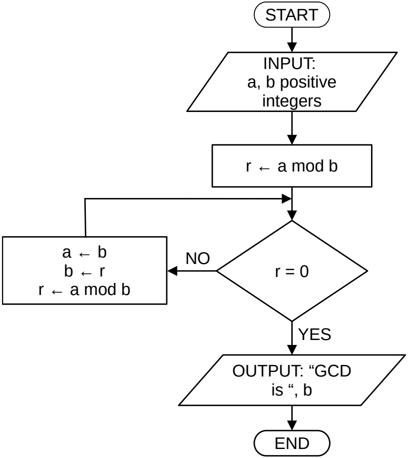
Flowcharts: alternative to pseudocode



GCD - with flowchart

Input: $a, b \in \mathbb{N}^*$
Output: GCD

$r \leftarrow a \bmod b$
while $r \neq 0$ **do**
    $a \leftarrow b$
    $b \leftarrow r$
    $r \leftarrow a \bmod b$
end while
GCD $\leftarrow b$



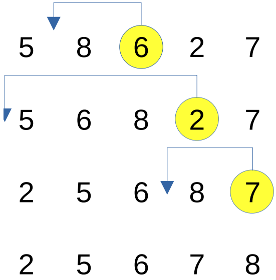
Analysis of algorithms

What's more important than performance?

- modularity
- correctness
- maintainability
- functionality
- robustness
- user-friendliness
- programmer time
- simplicity
- extensibility
- reliability

Insertion sort

Problem: sort a sequence of numbers $[a_i], i = 1, \ldots, N$ in increasing order.

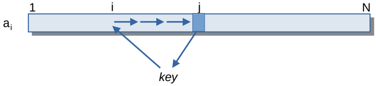


Insertion sort

Algorithm 4 Insertion sort

Input: $[a_i], i = 1, \ldots, N$
Output: sorted $[a_i]$

for $j = 2, \ldots, N$ **do**
  $key \leftarrow a_j$
  $i \leftarrow j - 1$
  while $i > 0$ AND $a_i > key$ **do**
    $a_{i+1} \leftarrow a_i$
    $i \leftarrow i - 1$
  end while
  $a_{i+1} \leftarrow key$
end for



Running time analysis

- depends on the ordering of the sequence: if it's ordered already, we just sweep once through it
- idea 1: find the dependency of the running time on the sequence size
- idea 2: find the upper limit of the running time
- idea 3: ignore machine-dependent part, concentrate on the intrinsic time

Running time analysis - two scenarios

$T(n) = ?$

- *worst case scenario*: gives the upper limit on the running time
- *average-case*: gives the expected running time

Running time analysis - asymptotic analysis

Main idea

Study

$$T(n) \text{ as } n \to \infty$$

"Big-Oh notation" - complexity function

- $\mathcal{O}(g(n)) = \{f(n) | \exists c_1, c_2 > 0, n_0 \in \mathbb{N} : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$
- e.g. for polynomial functions, consider just the dominating term:

$$an^3 + bn^2 + cn + d = \mathcal{O}(n^3), \forall a, b, c, d \in \mathbb{R}$$

Complexity of the insertion sort algorithm

- *worst case*: the sequence is reversed (decreasing order)

$$T(n) = \sum_{j=2}^{n} \mathcal{O}(j) = \mathcal{O}(n^2)$$

- *average case*: consider all permutations of n elements as equally probable

$$T(n) = \sum_{j=2}^{n} \mathcal{O}(j/2) = \mathcal{O}(n^2)$$

Questions?