overview

- practicalities
- introduction
- sorting
- insertion sort
- our model
- time complexity
- material

who

- lectures:
  Femke van Raamsdonk
  f.van.raamsdonk at vu.nl
  T446

- exercise classes:
  Ellen Maassen
  Petar Vukmirovic
when and where

in week 36–42:

- **lectures:**
  Mondays 13.30-15.15 in M143
  Thursdays 11.00-12.45 in different rooms in the main building

- **two groups for the exercise classes:**
  Tuesdays 15.30-17.15 in M655 (group 1) and in M639 (group 2)
  Fridays 09.00-10.45 in M655 (group 1) and M639 (group 2)

tests

- **written exam** (closed book) in week 8 of the course
  in case of double exams etcetera: contact the education office
  there is a **resit** in January

- **mid-term exam** in week 4 of the course
  recommended but not obligatory
  if mid-term exam better than exam, then the mid-term mark
  contributes for 30% to the exam-mark

- **practical work**
  two assignments, deadlines September 22 and October 13
  first counts for 40% and the second for 60% of the practical-mark

- **final mark:** 80% exam-mark 20% practical-mark
  both exam-mark and practical mark should be at least 5.5
  partial results are only valid in 2016-2017

material

*Introduction to Algorithms*
by Cormen, Leiserson, Rivest, Stein
data structures and algorithms: context

some problems cannot be solved
some problems cannot be solved efficiently
some problems can be solved efficiently
for some problems we do not know whether they can be (efficiently) solved

if $P \neq NP$ then the NP-complete problems cannot be efficiently solved

example algorithm: baking a cake

- software / data structures: tools
- input: ingredients
- program / algorithm: recipe
- hardware: oven
- output: cake

example algorithm: Euclid’s gcd

compute the greatest common divisor of two non-negative numbers $a \geq b$:
- if $b = 0$ then return $a$
- if $b \neq 0$, then compute the gcd of $b$ and $(a \mod b)$

the second line contains a recursive call

what this course is about

we will study basic data structures and algorithms
prerequisite: elementary programming
but this is not a programming course
prerequisite: elementary (discrete) mathematics and graph theory
but this is not a pure theory course
we study: algorithmic design, data structures, efficiency of algorithms
algorithm

an algorithm is een list of instructions, the essence van een program

what are important aspects?

- **correctness**
  does the algorithm meet the requirements?

- **termination**
  does the algorithm eventually produce an output?

- **efficiency or complexity**
  how much time and memory space does it use?

complexity

algorithms that 'do' the same may differ in performance

time complexity:
how much time does the algorithm use?
time as function of the input

space complexity:
how much space does the algorithm use?
space as function of the input

we care about time complexity

example: sort a finite sequence of length $n$ of numbers

assumption: our computer performs $10^9$ operations per second

insertion sort: uses say $2\cdot n^2$ steps

merge sort: uses say $50\cdot n\cdot \log n$ steps

then: sorting a sequence of length $n = 10^7$ takes

$2\cdot10^5$ seconds (55 hours) for insertion sort

$\sim 12$ seconds for merge sort

hence we care about steps

assumption: our computer performs $10^9$ operations per second

<table>
<thead>
<tr>
<th>steps</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^3$</td>
<td>0.0001s</td>
</tr>
<tr>
<td>$10^6$</td>
<td>0.001s</td>
</tr>
<tr>
<td>$10^7$</td>
<td>0.01s</td>
</tr>
<tr>
<td>$10^8$</td>
<td>0.1s</td>
</tr>
<tr>
<td>$10^{12}$</td>
<td>16.7min</td>
</tr>
<tr>
<td>$10^{14}$</td>
<td>27h47min</td>
</tr>
<tr>
<td>$10^{18}$</td>
<td>33yr</td>
</tr>
</tbody>
</table>

we also care about space
data structures

**data structure:**
a systematic way of storing and organizing data in a computer so that it can be used efficiently

different data structures for different applications

elementary operations performed on it

elementary operations performed on it

elementary operations performed on it

elementary operations performed on it

example of a data structure?

**overview**

- practicalities
- introduction
  - sorting
- insertion sort
- our model
- time complexity
- material

**sorting:** specification

- **input:**
a finite sequence of elements

- **output:**
an ordered permutation of the input-sequence
what do we sort?

- elements are usually integers or natural numbers
- an element may occur more than once
- the input-sequence is assumed to be an array
- often an element is actually the key of some item

(we do not bother about for instance the cost of moving big items)

sorted: definitions

- an ordering is a binary relation \( \leq \) such that
  - \( n \leq n \) (reflexivity)
  - if \( m \leq n \) and \( n \leq p \) then \( m \leq p \) (transitivity)
  - if \( m \leq n \) and \( n \leq m \) then \( m = n \) (anti-symmetry)

- an ordering is total if every pair of elements can be compared
- a sequence \( a_1 a_2 \ldots a_n \) is ordered if it is non-decreasing
  - that is, \( a_1 \leq a_2 \leq \ldots \leq a_n \)
- usually we consider natural numbers or integers with \( \leq \)

properties of sorting algorithms

- a sorting algorithm may or may not be
  - comparison-based
    - based on comparisons of pairs of elements
  - in-place
    - use the space for the input sequence plus a constant amount of space
  - stable
    - keep the order of equal elements
    - (only interesting if they are keys of some bigger item)

- usually we are interested in the worst-case time complexity

overview

- practicalities
- introduction
- sorting
- insertion sort
- our model
- time complexity
- material
insertion sort: idea and example

the sequence consists of a sorted part followed by a non-sorted part

initially: the sorted part consists only of the first element

loop: while the non-sorted part is non-empty
insert the first element of the non-sorted part in the correct position of the sorted part

idea (give more detail):

\[
\begin{array}{c}
5, 3, 4, 7, 1 \\
3, 5, 4, 7, 1 \\
3, 4, 5, 7, 1 \\
3, 4, 5, 7, 1 \\
1, 3, 4, 5, 7
\end{array}
\]

insertion sort: pseudo-code

Algorithm insertionSort(A, n):

for \( j := 2 \) to \( n \) do

\[ key := A[j] \]

\[ i := j - 1 \]

while \( i \geq 1 \) and \( A[i] > key \) do

\[ A[i + 1] := A[i] \]

\[ i := i - 1 \]

\[ A[i + 1] := key \]

insertion sort: correctness

loop invariant \( I \):

at the start of the for-loop,

the subarray \( A[1 \ldots j - 1] \) is a sorted permutation of the sub-array \( A[1 \ldots j - 1] \) of the input-array

init: \( I \) is initially (for \( j = 2 \)) true

loop: \( I \) remains true during the loop (!)

end: \( I \) gives correctness for \( j = n + 1 \)

overview

• practicalities
• introduction
• sorting
• our model
• time complexity
• material
we write algorithms in pseudo-code

pseudo-code resembles a programming language
but is independent of specific syntax

pseudo-code: globally

- input
- output using `return`
- block structure via indentation
- declare and call procedures
- declare and use data structures
- recursive calls
- objects with attributes, for example `A.length`

pseudo-code: calculating

- booleans: `true`, `false`
- calculating with booleans: `and`, `or` (short-circuiting)

- integers
- calculating with integers: addition, subtraction, multiplication, modulo
- elementary tests on integers: greater than, less than

pseudo-code: control

- declare and use variables
- assignment
- declare and update arrays and array elements
- `if` `then`, `while` `do`, `for` `do`, `repeat`
our model

Random Access Machine (RAM)

Central Processing Unit (CPU) with memory

unlimited number of memory cells (registers)

primitive operations take constant (little) time

overview

- practicalities
- introduction
- sorting
- insertion sort
- our model
- time complexity
- material

note that

we do not take into account hardware (processor, clock rate, caches, ...)

we do not take into account software (compiler, operating system, ...)

we do not perform experiments (no implementation needed)

theoretical analysis

independent of hardware,

independent of software,

and for all possible inputs

computer:
Random Access Machine (RAM)

algorithm:
description in pseudo-code

data structure:
specification as Abstract Data Type (ADT)

(worst-case) time complexity:
(upper bound on) running time as function of the input size

Random Access Machine (RAM)
**insertion sort: pseudo-code (again)**

**Algorithm** `insertionSort(A, n)`:  
\[
\text{for } j := 2 \text{ to } n \text{ do} \\
\quad \text{key} := A[j] \\
\quad i := j - 1 \\
\quad \text{while } i \geq 1 \text{ and } A[i] > key \text{ do} \\
\qquad A[i + 1] := A[i] \\
\qquad i := i - 1 \\
\quad A[i + 1] := key
\]

**pseudo-code for arrayMax**

**Algorithm** `arrayMax(A, n)`:  
\[
\text{Input: array } A \text{ storing } n \text{ integers} \\
\text{Output: the maximum element of } A \\
\quad \text{currentMax} := A[1] \\
\quad \text{for } i := 2 \text{ to } n \text{ do} \\
\qquad \text{if } \text{currentMax} < A[i] \text{ then} \\
\qquad \quad \text{currentMax} := A[i] \]
\[\text{return } \text{currentMax}\]

**arrayMax: counting primitive operations**

\begin{align*}
\text{init:} & \\
\text{read } A[1] & 1 \\
\text{assign value } A[1] \text{ to } \text{currentMax} & 1
\end{align*}

\[
\text{bookkeeping for loop:} \\
\text{assign value } 2 \text{ to } i & 1 \\
\text{check } i \leq n \text{ (} n-1 \text{ times true, } 1 \text{ time false) } & n \\
\text{compute } i + 1 & n - 1 \\
\text{assign value } i + 1 \text{ to } i & n - 1
\]

\[
\text{body of loop for } i = 2, \ldots, n \\
\text{read } \text{currentMax} & n - 1 \\
\text{read } A[i] & n - 1 \\
\text{check } \text{currentMax} < A[i] & n - 1 \\
\text{possibly assign } \text{currentMax} \text{ value } A[i] \text{ (worst-case!) } & n - 1
\]

\[
\text{return:} \\
\text{return } \text{currentMax} & 1
\]

**arrayMax: organize counting primitive operations**

count the number of primitive operations for **worst** or **best** or **average** case  
for the latter: probability distribution over possible executions needed  
for **arrayMax**:  
\[
\text{worst case: } 4 + n + 6(n - 1) = 7n - 2 \text{ operations} \\
\text{best case: } 4 + n + 5(n - 1) = 6n - 1 \text{ operations}
\]
arrayMax: order of growth

first,
we are not so interested in the constants \((7, -2, 6, -1)\)

second,
for \(c_1 \cdot n + c_0\) we are not so interested in the lower-order terms \((c_0)\)

then:
worst-case running time of arrayMax is \text{linear in input size } n

put differently:
worst-case complexity of arrayMax is \(\Theta(n)\)

insertion sort: worst-case time complexity

test for-loop: \(n\)

assignment \(key: n - 1\)

assignment \(i: n - 1\)

worst case: \(A[i] > key\) always succeeds

for fixed \(j\): we do \(j\) times the while-test
and \(\sum_{j=2}^{n} j = \frac{1}{2}n(n + 1) - 1\)

for fixed \(j\): we do \(j - 1\) times the assignment \(A[i + 1]\)
and \(\sum_{j=2}^{n} (j - 1) = \frac{1}{2}(n - 1)n\)

for fixed \(j\): we do \(j - 1\) times the assignment \(i\)
and \(\sum_{j=2}^{n} (j - 1) = \frac{1}{2}(n - 1)n\)

assignment \(A[i + 1]: n - 1\) times

we often use

\[
\sum_{i=0}^{n} a^i = \frac{1 - a^{n+1}}{1 - a}
\]

\[
\sum_{i=1}^{n} i = \frac{n(n + 1)}{2}
\]
we are actually mostly interested in:

- running time as a function of the size of the input
- growth of running time when input size increases
  - the running time of arrayMax is linear in $n$
  - the running time of insertion sort is quadratic in $n$
- asymptotic approximation
  - constant factor difference becomes irrelevant

$\Theta$ of $n$ gives us the order of growth

for example: we ignore constant factors of polynomials

for example: we restrict attention to the highest degree in a polynomial

for example: $10n^3 + 100n^2 + 1000n$ is in $\Theta(n^3)$