# Algorithms

Gökçe Aydos

# Lecture

# Goals

- understand the meaning of *algorithm*
- be able to formulate algorithms for easy problems
- be able to specify problems
- seeing programming languages as communication tools for algorithms
- understanding the concept of compilers

# Theme

- *algorithm* – a frightening word
- we use algorithms every day, e.g., for duplicating numbers
- *cook-book recipe* for computers

# History

- word originates from the name of mathematician Muhammad ibn Musa *al-Khwarizmi* (780-850)[1]
- he wrote an arabic book about Hindu-Arabic numbering system
- his books were widely used in Europe
- Latin translation of the book started with *Dixit Algorizmi*

[1] https://en.wikipedia.org/wiki/Algorithm#Etymology

# The first computer algorithm

- by *Ada Lovelace* in 1842
- she saw the potential of Charles Babbage's *Analytical Engine*
- published a program for calculating Bernoulli numbers
- Analytical Engine was never completed, so Ada's algorithm was never implemented

# An old algorithm for duplication by Adam Ries

Lehret wie du ein zahl zweyfaltigen solt.

Thu ihm also: Schreib die zahl vor dich
mach ein Linien darunter
heb an zu forderst
Duplir die erste Figur. Kompt ein zahl die du mit einer Fig
so seß die unden. Wo mit zweyen
schreib die erste
Die andere behalt im Sinn. Darnach duplir die ander
und gib darzu
das du behalten hast
und schreib abermals die erste Figur
wo zwo vorhanden
und duplir fort bis zur leßten
die schreibe gantz aus
als folgende Exempel aufweisen

# An old algorithm for duplication II

```
How to teach to duplicate numbers:

Say them: Write the number
draw a line below
start to calculate
Duplicate the first digit. If the result has a single digit
write it below. If two
write only the first digit
Memorize the second one. Afterwards duplicate the second di
add the number
that you memorized
Again write only the first digit
if you get two digits
and continue duplicating until the last digit
Write the last sum to the leftmost side
```

# An old algorithm for duplication III

```
 41232    98765   68704
 -----   ------ ------
 82464   197530 137408
```

# Definition

- *algorithm* is an abstract sequence of instructions to solve a problem
- a weak example: a recipe for a cake
- a good example: instructions to multiply numbers on paper

# Algorithms vs programs

algorithms:

- *general* description for a problem solution
- *not bound* to a *specific computer* or *programming language*
- same algorithm can generally be used *in many scenarios*

programs:

- sequence of *specific* instructions
- are written in a *specific* programming language
- a program solves *a specific problem*, and not others

# Generality of instructions

algorithms are *general* instructions, and their *generality* can differ:

basic:

1. swap the two numbers at the beginning of the list
2. if the first number is greater than the second, continue at step 19

complex:

1. sort the list
2. remove the maximum

# Sorting - demo

take three cards with numbers on them and demonstrate bubble sort

# Sorting - exercise

an example algorithm for sorting three cards:

1. if the left card is greater than the middle card, swap them
2. if the middle card is greater than the right one, swap them
3. if the left card is greater than the middle one, swap them

Write an algorithm for sorting four cards.

Optional: Can your algorithm sort the cards in minimum number of steps?

# Typical components of algorithms

1. *instruction sequences*, e.g., first this, then that
2. *conditionals*, e.g., if this then that else these
3. *loops*, e.g.,
   - while this is true repeat this
   - repeat this 100 times
4. jumps (considered harmful in programming), e.g., continue at this line. (generally no calculation)

# An old algorithm for duplication III

How to teach to duplicate numbers:

```
Say them: Write the number <-jmp
draw a line below          <-inst
start to calculate         <-inst
Duplicate the first digit.  <-loop start
If the result has a single digit  <-cond start
write it below. If two  <-cond
write only the first digit  <-cond
Memorize the second one.  <-cond end
Afterwards duplicate the second digit  <-inst
add the number             <-inst
that you memorized         <-inst
Again write only the first digit  <-inst
if you get two digits.  <- inst (repetition of cond above)
and  <- jmp
continue duplicating until the last digit <-loop end
```

# Jumps considered harmful

- originating from Dijkstra's[2] work *goto considered harmful*
- jumps lead to *spaghetti code* in long programs
- solution: *structured programming* w/o jumps
- programming languages are for humans (and for computers)
- machine code, e.g., assembly, still relies on jumps

[2] Dutch computer scientist (1930-2002)

# Specifications

- to solve problems, we need *clear problem descriptions*
- $=>$ need for *specifications*
- what are the conditions?
- which tools are allowed?
- when is a solution correct or acceptable?

# Specifications II

- creating exact specifications is hard
- typically the customers do not know what they *exactly* want
- specifications may change during development
- led to motivation for *agile development* vs *waterfall model* in software development

# Specifications - example

- calculate the sum of first n numbers.
- tools at your disposal: addition, comparison

# Specifications - problem

Long specifications tend to contain contradictions, e.g.:

```
page 50: if key 1 is pressed, the text should turn red
page 150: if key 1 is pressed, the text should turn green
```

Solution: specification languages, formal verification

# Programming languages

- programming languages turn our idea to an instruction sequence
- specification: you have a natural number. The number must be duplicated
- idea: duplicate the number digitwise from right to left and consider carry
- algorithm: Riese's algorithm or:

```
For every digit, beginning from the rightmost, do the f

1. duplicate the digit
2. if there is a carry from the last iteration, add it
3. write the right digit of the sum below the digit bei
4. memorize the left digit of the sum

If there is a carry left, write it leftmost.
```

# Algorithm - pseudocode

```
in: digits z_1 to z_n
c <- 0
for i <- n,...,1 do
  d <- 2z_i + c
  z'_i <- d mod 10
  c <- |_ d/10 _|
z'0 <- c
for i <- 0,...,n do
  output z'_i
```

# Algorithm - Python

```python
# expects a list of digits
def duplicate(digits):
    digits_duplicated = list(digits)
    carry = 0

    for i in reversed(range(len(digits))):
        print(digits[i], i)
        d = 2*int(digits[i]) + carry
        digits_duplicated[i] = str(d % 10)
        carry = d//10

 return str(carry) + ''.join(digits_duplicated)
```

# Programming languages III

variety:

- high-level, e.g., Python, Java, C++
- low-level, e.g., C, assembler

# Programming languages IV

- but the goal of a program does not change: programming languages are tools for communicating your algorithms to the computer
- programming languages *must be learned like a spoken language* by practising

# Compilers

- *compilers* are programs which translate programming code to another code that a CPU can understand

# Summary

- *algorithms* are general instructions to solve a problem
- a *programming language* defines an art how to describe algorithms
- *program* is a text written in a programming language, which implements an algorithm
- *specification* defines what a program should accomplish
- *compiler* is a program which translates between programming languages

# Summary II

Steps for solving a problem:

1. problem definition
2. what do you want to achieve?: create a *specification*
3. come up with a *solution idea*
4. refine it to an *algorithm*
5. implement it in a (high-level) programming language
6. compile your code to get the CPU instruction sequence