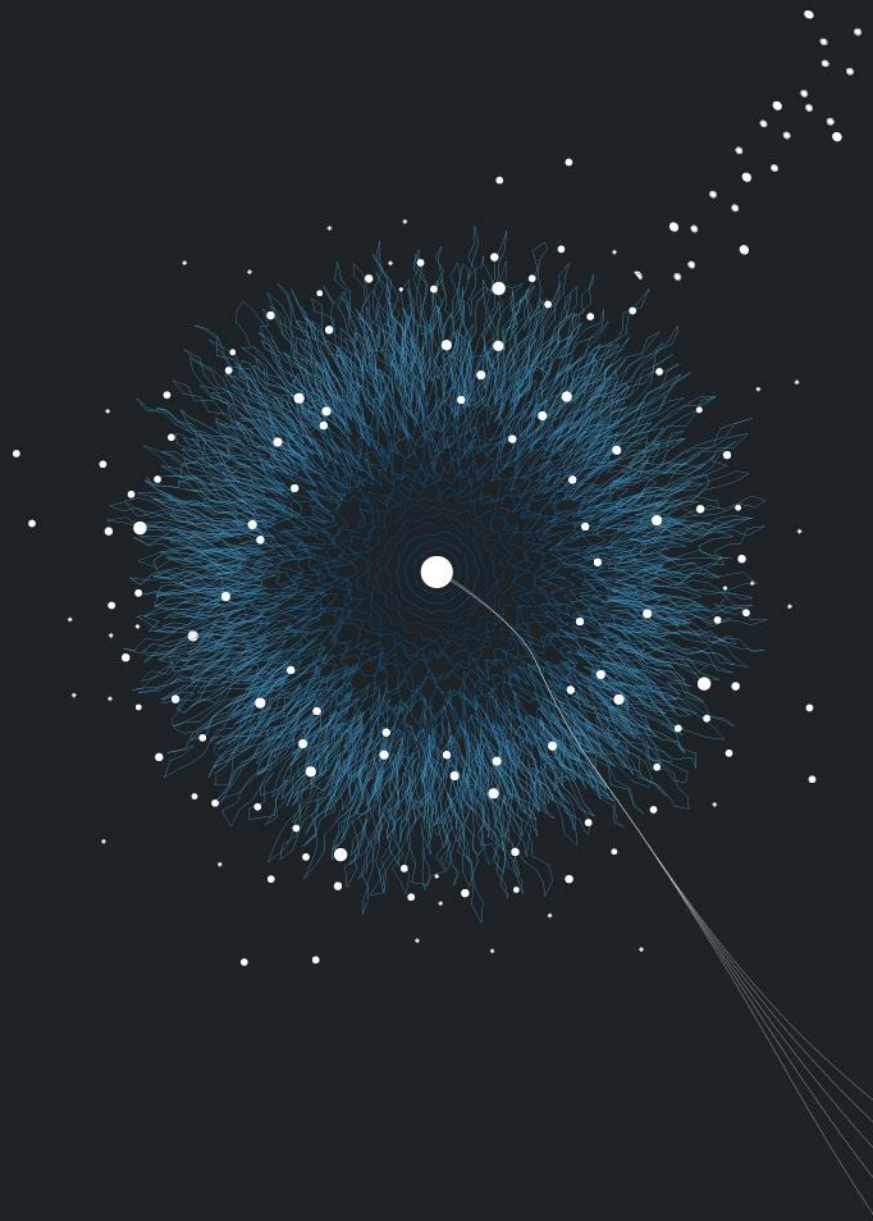
COMPUTATIONAL THINKING



M2 PROGRAMMING LECTURE 1.2

BY TOM VAN DIJK

UNIVERSITY
OF TWENTE.

SO FAR

Good programs are

- Functional
- Reliable, testable
- Understandable, documented
- Formally specified / verified
- Designed for **change**

SO FAR

- Variables and types **structure** memory
- Procedures **structure** code
- Types
  - **Whole numbers**: byte, short, int, long
  - **Floating-point numbers**: float, double
  - Other: **char**, **boolean**
  - Strings, Arrays
- Control statements
  - **conditionals**: if, else, switch
  - **loops**: for, while, do-while, continue, break

Q&A

Any questions about **procedural programming** so far?

COMPUTATIONAL THINKING

“the thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms” (Cuny, Snyder, Wing, Aho)

COMPUTATIONAL THINKING VS CODING

- **Coding** is **expressing** solutions in a language a machine understands
- **Computational thinking** is **finding** those solutions in the first place

- Memorizing Java syntax does not help with your skill in computational thinking
- You can get better at computational thinking without writing code

COMPUTATIONAL THINKING

Decomposition: Break down a problem into smaller, more manageable parts

Abstraction: Replace specifics by abstractions / placeholders to reason about all cases

Pattern recognition: Reuse solutions to similar problems that you've seen before

Precise / algorithmic thinking: Write exact, complete steps that a machine can follow

DECOMPOSITION

- Breaking down a complex task into smaller, more manageable components
  - Break down a problem into parts, then combine solutions of the parts to solve the larger problem
  - Subproblems can be solved simultaneously (parallel) or in a specific order (sequential)
  - Decompose until we could solve the subproblems
  - Does **not** actually solve the problem, it just organizes the space
- Strategies
  - Divide by functionality or responsibility: *"What are major things this system needs to do?"*
  - Separate input → processing → output
  - Identify repeating tasks
  - What parts could be isolated and tested separately?
  - Follow the data: decompose based on how data changes as it flows through the system
  - "Architectural": decompose a system into layers or modules, what parts talk to what

EXAMPLES FOR DECOMPOSITION

Making a **lasagna**

- Buying the ingredients — check what we have, plan quantities, go to the store, etc
- Making the ragú — cooking the soffrito, browning the meat, adding wine, milk, tomatoes, etc
- Making the lasagna sheets — making pasta dough, rolling out the dough, cutting into sheets
- Making the béchamel sauce — make roux (melt butter, add and cook the flour), whisk in milk
- Preheating the oven — must happen before baking
- Layering the lasagna — only when all parts are ready, alternate layers, end with cheese
- Baking and serving — place in oven, monitor progress, let it rest before cutting, etc

EXAMPLES FOR DECOMPOSITION

Decomposing a **messaging app**

- Handle input from user
- Display messages
- Pop up notifications
- Maintain connection with the server
- Store contacts
- Store message history

EXAMPLES FOR DECOMPOSITION

Other (non computing) examples

- Planning a party or a holiday
- Running Module 2
- Planning the Module 2 implementation project

Other (computing) examples

- Maze solving robot
- Video games
- File downloader

EXAMPLES FOR DECOMPOSITION

Any other examples to decompose?

PRECISE / ALGORITHMIC THINKING

- Thinking in clear, step-by-step instructions that a machine (or very literal human) can execute
- Every step must be **unambiguous**, **complete**, and **ordered**
- Anticipate edge cases, conditions, etc.
- **No skipping tricky details, vague descriptions, or open to interpretation** (the compiler isn't a large language model)

EXAMPLES FOR PRECISE / ALGORITHMIC THINKING

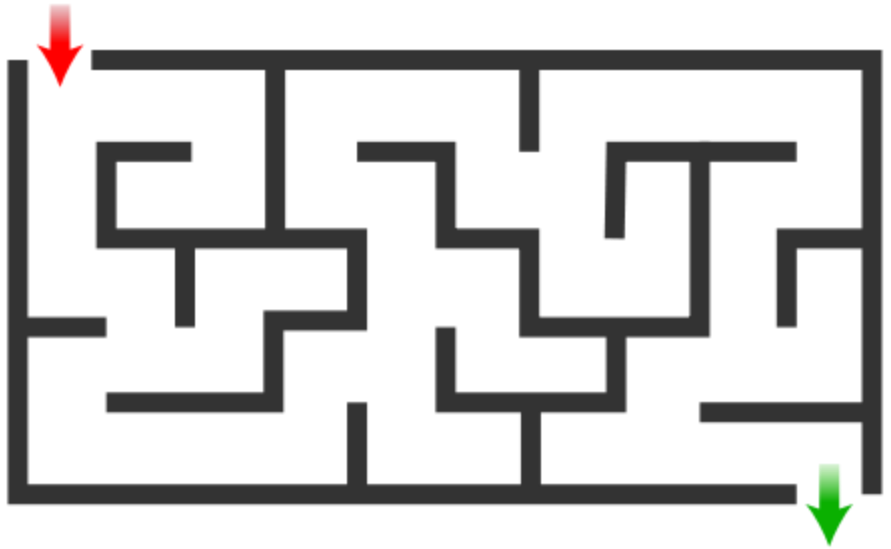
- A robot needs to find the exit of a maze.

- Capabilities:
  - MOVE — move one unit forward
  - RIGHT — turn 90 degrees clockwise
  - WALL_AHEAD? — **true** if a wall is directly in front, otherwise **false**
  - WALL_RIGHT? — **true** if a wall is directly to the right, otherwise **false**
  - AT_EXIT? — t**rue** if the exit is reached, otherwise **false**

- Write a program (pseudo-code) for the robot.

EXAMPLES FOR PRECISE / ALGORITHMIC THINKING

- MOVE — move one unit forward
- RIGHT — turn 90 degrees clockwise
- WALL_AHEAD? — **true** if a wall is directly in front, otherwise **false**
- WALL_RIGHT? — **true** if a wall is directly to the right, otherwise **false**
- AT_EXIT? — **true** if the exit is reached, otherwise **false**

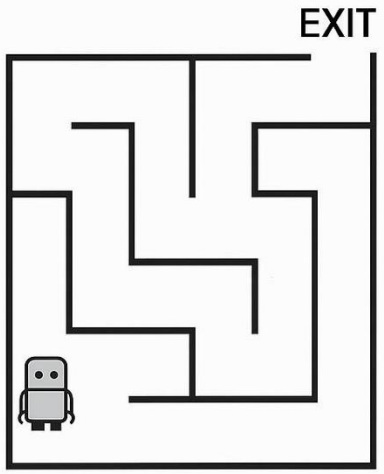


EXAMPLES FOR ALGORITHMIC THINKING

ChatGPT…

- **MOVE** – step forward (if no wall ahead)
- **RIGHT** – turn 90° to the right
- **WALL_AHEAD?** – is there a wall in front?
- **WALL_RIGHT?** – is there a wall to right?

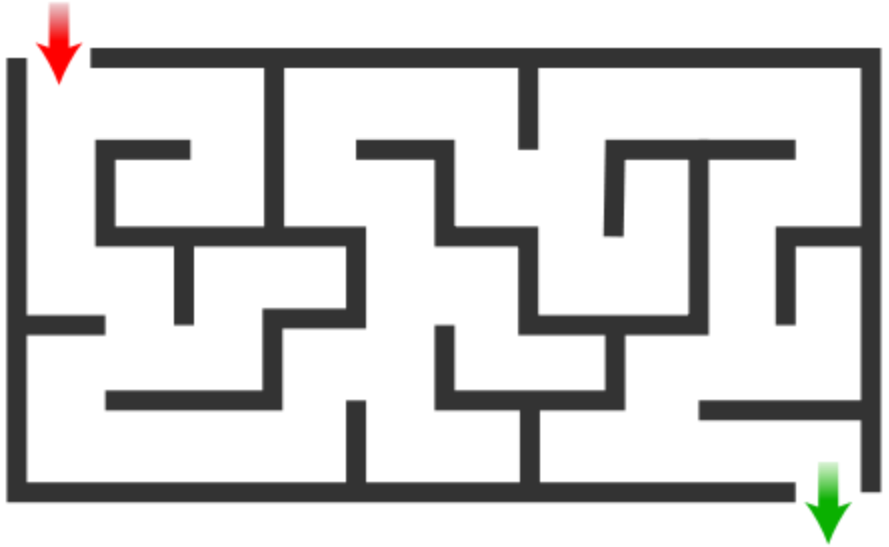
```
while not AT_EXIT?
  if not WALL_RIGHTT?
     RIGHT
  if not WALL_AHEADD?
     MOVE
```



EXAMPLES FOR ALGORITHMIC THINKING

- while not AT_EXIT?
  - if not WALL_RIGHT?
    - RIGHT
  - if not WALL_AHEAD?
    - MOVE
  - else
    - RIGHT, RIGHT

- while not AT_EXIT?
  - if not WALL_RIGHT?
    - RIGHT
  - **else** if not WALL_AHEAD?
    - MOVE
  - else
    - RIGHT, RIGHT



- while not AT_EXIT?
  - if not WALL_RIGHT?
    - RIGHT, MOVE
  - else if not WALL_AHEAD?
    - MOVE
  - else
    - RIGHT, RIGHT, RIGHT

EXAMPLES FOR PRECISE / ALGORITHMIC THINKING

How to make toast?

DECOMPOSITION VS ALGORITHMIC THINKING

- Decomposition is about **breaking down a problem** into ever smaller parts
  - Decomposition does not require knowing the solution

- Algorithmic thinking is about **formulating a solution** in precise, exact steps
  - Solving the problem suffers without decomposition

ABSTRACTION

Two different aspects:

- Focus on the **essential aspects** of a problem, and **ignore irrelevant details** (connected to **decomposition**)
- Replace **specific** values (names, numbers, etc) with **general** variables or types (connected to **algorithmic thinking**)

How it fits together:

1. Decomposition sets up boundaries
2. Selective abstraction (essential aspects) lets you work on each subproblem
3. First solve concrete cases, then generalize to handle all inputs precisely

PATTERN RECOGNITION

- Recognizing when a problem you face is similar to one you've solved or seen before
- It's not about patterns in data but about patterns in **problems and solutions**
- Choose the right **data structure**, **algorithm**, **strategy**, **design pattern**, etc.
- **Reuse of known solutions** (don't reinvent the wheel) and knowing which tool to use when
- For example https://refactoring.guru/ for design patterns, algorithms & data structures courses, etc.

EXAMPLES FOR PATTERN RECOGNITION

• Find the shortest route	Shortest-path graph problem, BFS, Dijkstra, Bellman-Ford
• Undo recent actions	Memento design pattern
• Avoid repeating calculations	Dynamic programming, memoization / caching
• Detect repeated entries	Hash set / Tree set
• Grouping things by a shared property	Hash map or counting array
• Process tasks in arrival order	Queue
• Backtracking puzzle solving	Recursive search, or use a SMT solver

SUMMARY

The four pillars of computational thinking

- **Decomposition**: Break down a problem into smaller, more manageable parts
- **Precise / algorithmic thinking**: Write exact, complete steps that a machine can follow
- **Abstraction**: Replace specifics by abstractions / placeholders to reason about all cases
- **Pattern recognition**: Reuse solutions to similar problems that you've seen before

These are not just skills for programming, they are tools for thinking clearly about *anything* complex
Computational thinking cannot be learned from a book, only from practice

Any questions about **computational thinking**?