Chapter 1.5: Computation and Computational Thinking

Computer Fundamentals and Introduction to Programming

Computation | History of computing | The Turing machine | Knowledge and algorithms | Computational thinking

Sercan KÜLCÜ · Giresun University

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/computer/slides/Chapter_01_05_Computation_and_Computational_Thinking.html

Learning Outcomes

Computation

Explain computation as processing data according to rules.

Models of computation

Trace a Turing machine step by step.

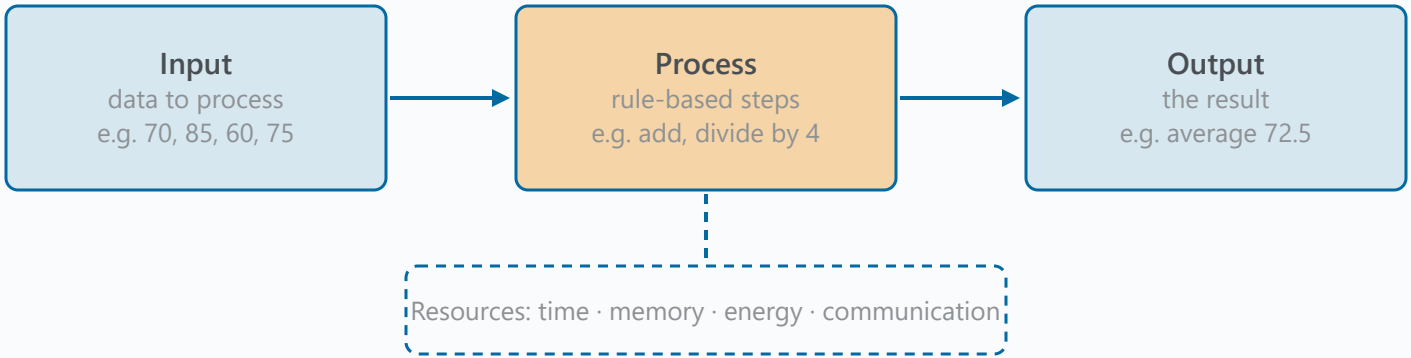
Knowledge and algorithms

Distinguish declarative from imperative knowledge; trace Euclid's algorithm.

Computational thinking

Apply decomposition, pattern recognition, abstraction and algorithm design.

Computation



Computation is more than arithmetic. Searching, sorting, deciding, transforming and communicating are computation too.

INTRODUCTION

The Building Blocks of Computation

Mathematics

Operations on quantities and relations.

Logic

Evaluating conditions as true or false.

Data

Encoded inputs and intermediate results of a problem.

Algorithm

A finite, ordered sequence of operations.

Hardware

The system that physically carries out the steps.

Programming language

A way to express an algorithm to a computer.

Computer Science

Computer science is not just about using tools; it studies how information is represented and how problems can be solved.

Abstraction

Hiding unnecessary detail to focus on the essential structure.

Algorithms

Designing and analyzing the steps of a solution.

Systems

Understanding how hardware and software work together.

People and society

Weighing technology's impact, security and ethics.

COMPUTER SCIENCE

Technology and Responsibility

Accessibility

Equal access to technology for different users.

Privacy

Using personal data securely and only for its stated purpose.

Fairness

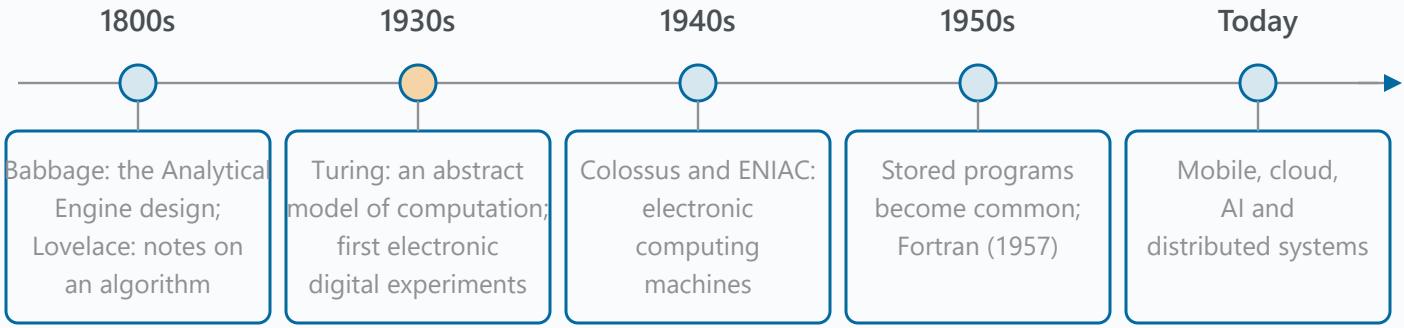
Data and algorithms should not produce discriminatory outcomes.

Sustainability

Accounting for energy, hardware lifetime and e-waste.

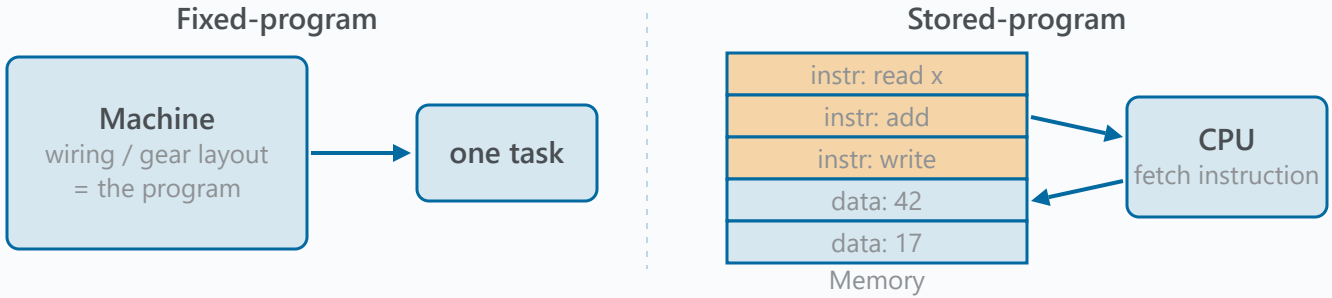
HISTORY

Milestones in the History of Computing



As hardware got faster and smaller, the same core ideas (algorithms, stored programs) found ever wider use.

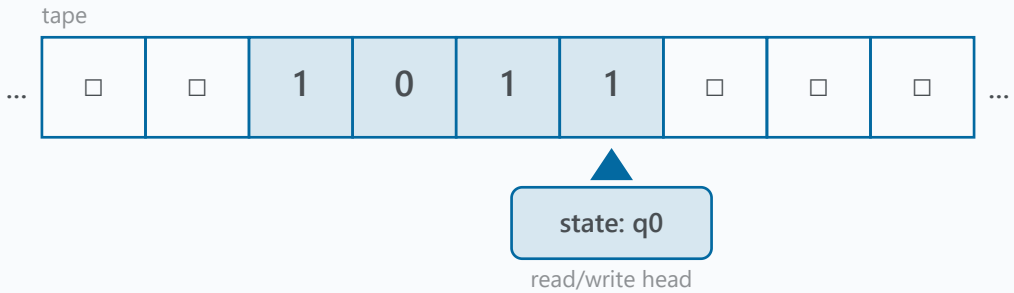
Fixed-Program vs. Stored-Program Machines



Approach	How is the program defined?	Flexibility
Fixed-program	By the machine's physical layout or a special mechanism	A specific task
Stored-program	Instructions are stored in memory like data	Different tasks by changing the program
General-purpose	An architecture that can run very different programs	Broad range of uses

The Universal Turing Machine

A Turing machine is an abstract model used to explain algorithmic computation.



- The head reads a symbol, may write a new one, and moves left or right.
- A transition rule picks the next step from the current state and symbol.
- A **universal** Turing machine reads the rules of another machine from its tape and simulates it: the theoretical basis of the stored-program idea.

A Turing Machine Step by Step: 111 + 1



Transition rules

1) q0, 1 → 0, left, q0

2) q0, 0 → 1, stop, HALT

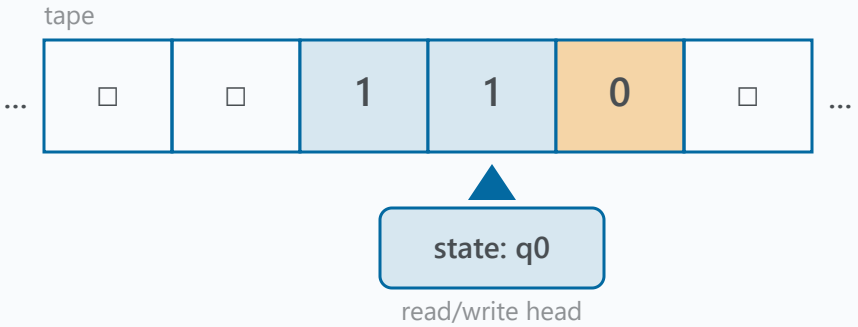
3) q0, □ → 1, stop, HALT

(state, read) → write, move, new state

On the tape: 111 (binary) = 7

Start: the tape holds **111** (binary 7), the head is on the rightmost digit, and the state is **q0**.
Goal: add 1.

A Turing Machine Step by Step: 111 + 1



Transition rules

1) q0, 1 → 0, left, q0

2) q0, 0 → 1, stop, HALT

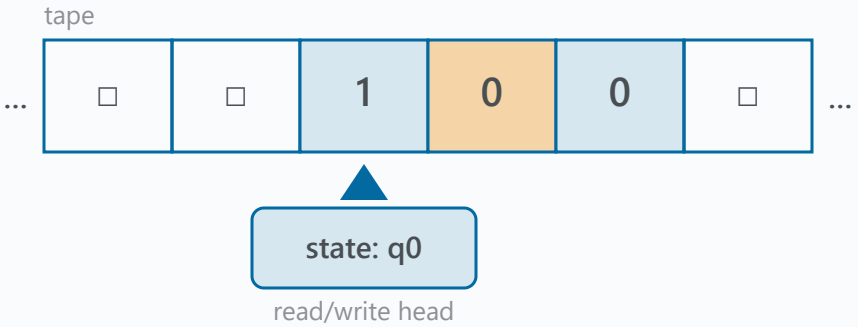
3) q0, □ → 1, stop, HALT

On the tape: 110 (binary) = 6

(state, read) → write, move, new state

Reads **1** in state q0 → rule 1: write **0**, move the head **left** (the carry of 1 continues).

A Turing Machine Step by Step: 111 + 1



Transition rules

1) q0, 1 → 0, left, q0

2) q0, 0 → 1, stop, HALT

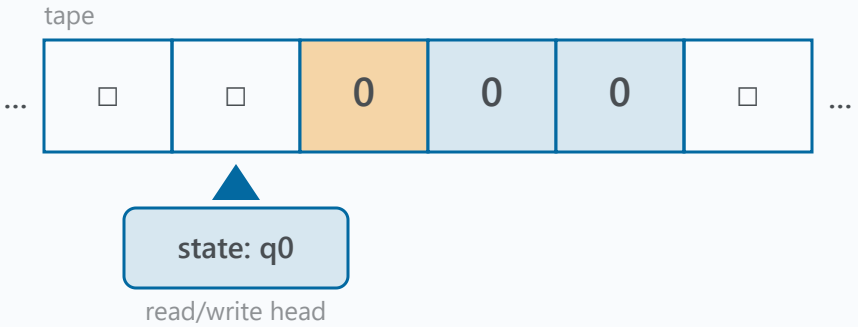
3) q0, □ → 1, stop, HALT

(state, read) → write, move, new state

On the tape: 100 (binary) = 4

Reads **1** in state q0 → rule 1: write **0**, move the head **left** (the carry of 1 continues).

A Turing Machine Step by Step: 111 + 1



Transition rules

1) q0, 1 → 0, left, q0

2) q0, 0 → 1, stop, HALT

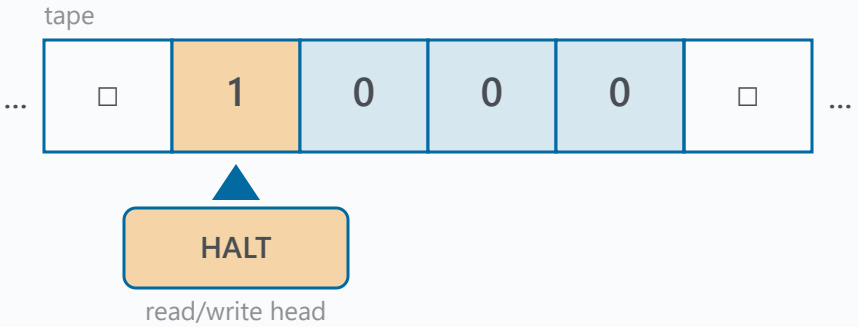
3) q0, □ → 1, stop, HALT

(state, read) → write, move, new state

On the tape: 000 (binary) = 0

Reads **1** in state q0 → rule 1: write **0**, move the head **left** (the carry of 1 continues).

A Turing Machine Step by Step: 111 + 1



Transition rules

1) q0, 1 → 0, left, q0

2) q0, 0 → 1, stop, HALT

3) q0, □ → 1, stop, HALT

(state, read) → write, move, new state

On the tape: 1000 (binary) = 8

Reads □ in state q0 → rule 3: write **1**, and the machine **halts**. Tape: **1000** = 8. Every step was determined only by (state, symbol read).

The Limits of Computers

Time

Large inputs can make some algorithms impractical.

Memory

Data and intermediate results use finite storage.

Energy

Every operation and data transfer consumes energy.

Computability

Some problems have no algorithm that always gives an answer (e.g. the halting problem).

Data quality

Missing or wrong input prevents a correct result.

Model limits

A program represents only selected features of the real world.

Kinds of Knowledge

Declarative knowledge

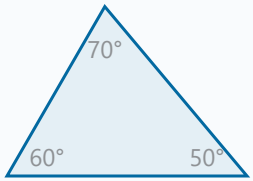
States what is true or what a concept is.

`The angles of a triangle add up to 180°.`

Imperative knowledge

Explains in steps how to reach a result.

`Approximate the square root of a number.`

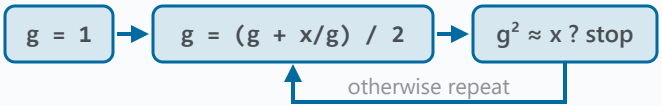


Declarative: what is true?

60° + 70° + 50° = 180°

states a fact; gives no method

Imperative: how to find it?



x = 25: 1 → 13 → 7.46 → 5.41 → 5.02 → 5.00

Algorithms turn imperative knowledge into clear, executable, finite steps.

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

We want the square root of **x = 25**.

Step 1 / 14

```
▶ x = 25
2 guess = 1
3 while |guess * guess - x| > 0.01
4     guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

x = **25**

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

A rough starting guess: **guess = 1**.

Step 2 / 14

```
1 x = 25
▶ guess = 1
3 while |guess * guess - x| > 0.01
4     guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

x = 25 | guess = 1

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Error $|1^2 - 25| = 24 \rightarrow$ **> 0.01: continue**.

Step 3 / 14

```
1 x = 25
2 guess = 1
▶ while |guess * guess - x| > 0.01
4     guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

`x = 25` `guess = 1`

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Pass 1: guess = (1 + 25 / 1) / 2 = **13**. Average the guess with x/guess.

Step 4 / 14

```
1 x = 25
2 guess = 1
3 while |guess * guess - x| > 0.01
      guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

`x = 25` `guess = 13`

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Error $|13^2 - 25| = 144 \rightarrow$ **> 0.01: continue**.

Step 5 / 14

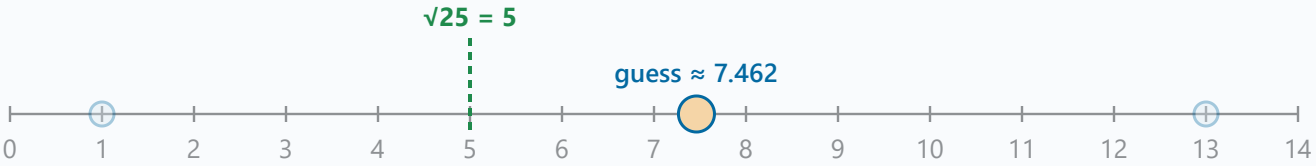
```
1 x = 25
2 guess = 1
▶ while |guess * guess - x| > 0.01
4     guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

`x = 25` `guess = 13`

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Pass 2: guess = (13 + 25 / 13) / 2 ≈ **7.462**. Average the guess with x/guess.

Step 6 / 14

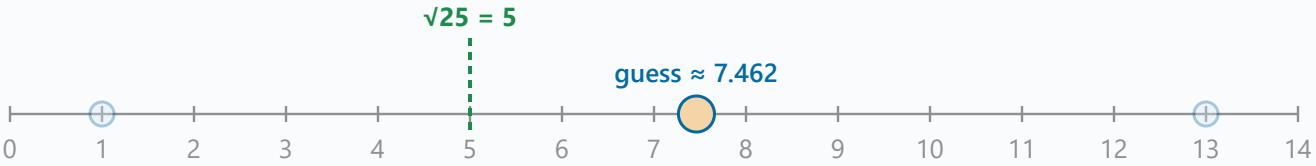
```
1 x = 25
2 guess = 1
3 while |guess * guess - x| > 0.01
      guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

`x = 25` `guess = 7.4615`

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Error $|7.462^2 - 25| \approx 30.67$ → **> 0.01: continue**.

Step 7 / 14

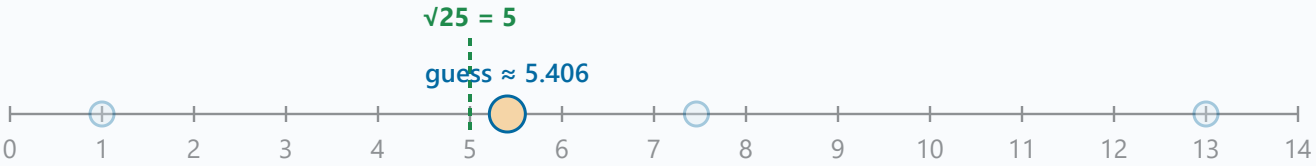
```
1 x = 25
2 guess = 1
▶ while |guess * guess - x| > 0.01
4     guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

`x = 25` `guess = 7.4615`

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Pass 3: guess = (7.462 + 25 / 7.462) / 2 ≈ **5.406**. Average the guess with x/guess.

Step 8 / 14

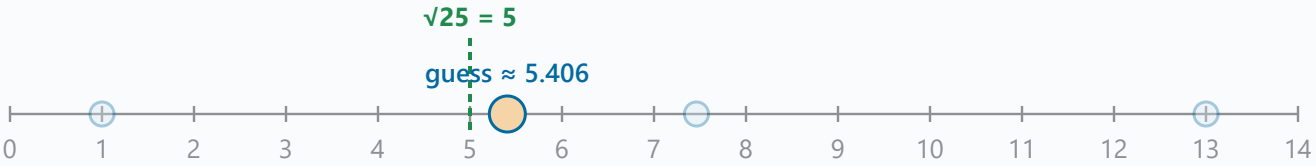
```
1 x = 25
2 guess = 1
3 while |guess * guess - x| > 0.01
      guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

`x = 25` `guess = 5.406`

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Error $|5.406^2 - 25| \approx 4.225$ → **> 0.01: continue**.

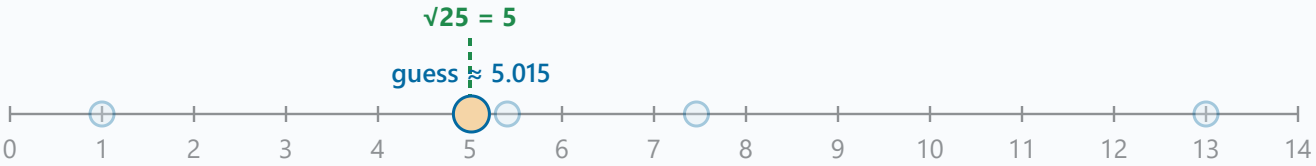

```
1 x = 25
2 guess = 1
▶ while |guess * guess - x| > 0.01
4     guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

`x = 25` `guess = 5.406`

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Pass 4: guess = (5.406 + 25 / 5.406) / 2 ≈ **5.015**. Average the guess with x/guess.

Step 10 / 14

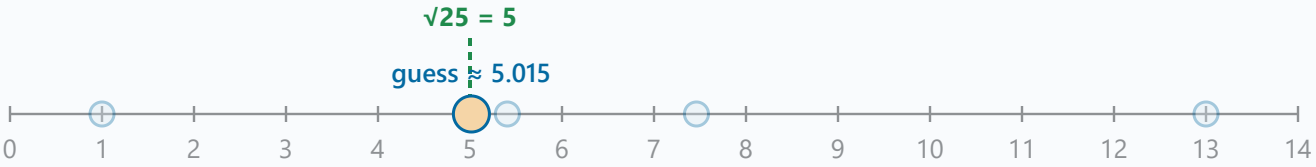
```
1 x = 25
2 guess = 1
3 while |guess * guess - x| > 0.01
      guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

x = 25 guess = 5.0152

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Error $|5.015^2 - 25| \approx 0.1527$ → **> 0.01: continue**.

Step 11 / 14

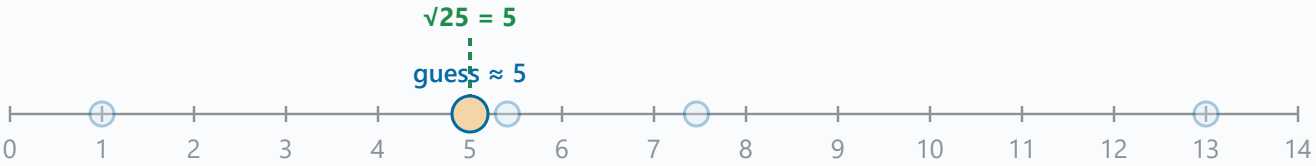
```
1 x = 25
2 guess = 1
▶ while |guess * guess - x| > 0.01
4     guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

`x = 25` `guess = 5.0152`

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Pass 5: guess = (5.015 + 25 / 5.015) / 2 ≈ **5.00002**. Average the guess with x/guess.

Step 12 / 14

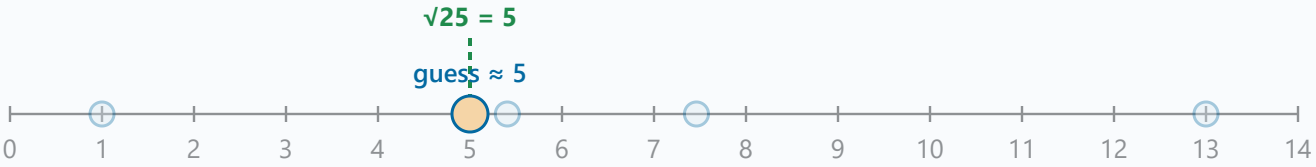
```
1 x = 25
2 guess = 1
3 while |guess * guess - x| > 0.01
      guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

x = 25 guess = 5

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Error $|5.00002^2 - 25| \approx 0.00023$ → **≤ 0.01: close enough, stop**.

Step 13 / 14

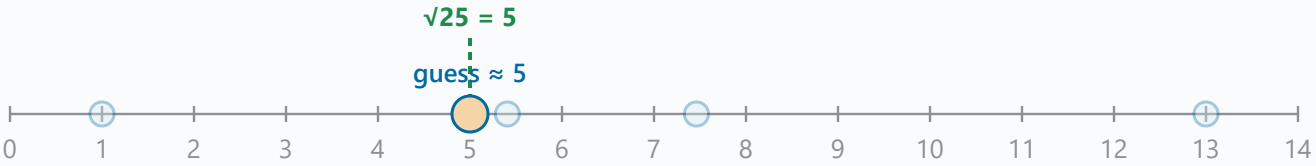
```
1 x = 25
2 guess = 1
▶ while |guess * guess - x| > 0.01
4     guess = (guess + x / guess) / 2
5 print(guess)
```

VARIABLES

`x = 25` `guess = 5`

OUTPUT

Imperative Knowledge: Approximating a Square Root (x = 25)



Faded dots: earlier guesses. Each round moves the guess closer to √x.

Output: ≈ **5.0000**. In 5 passes it got very close to the true value (5): knowing “how to find it” is **imperative knowledge**.

Step 14 / 14

```
1 x = 25
2 guess = 1
3 while |guess * guess - x| > 0.01
4     guess = (guess + x / guess) / 2
▶ print(guess)
```

VARIABLES

`x = 25` `guess = 5`

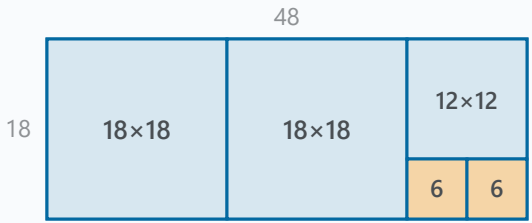
OUTPUT

```
5.0000
```

Euclid's Algorithm

An algorithm is a method that solves a problem in finite, unambiguous steps. One of the oldest known is Euclid's (around 300 BC): it finds the greatest common divisor of two positive integers by repeatedly taking remainders.

```
GCD(a, b)
while b ≠ 0
    r = a mod b
    a = b
    b = r
return a
```

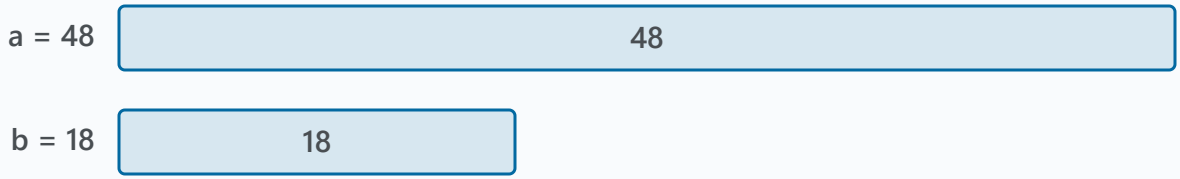


48 = 2×18 + 12 ; 18 = 1×12 + 6 ; 12 = 2×6

Largest square that tiles exactly: 6×6

GCD(48, 18): 48 mod 18 = 12 · 18 mod 12 = 6 · 12 mod 6 = 0 · result 6

Euclid Step by Step: GCD(48, 18)



GCD(48, 18) is called: a = 48, b = 18. The bars show the numbers as lengths.

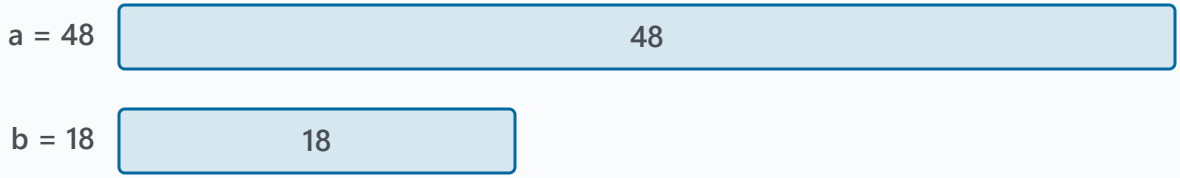
Step 1 / 15

```
▶ GCD(a, b)
2 while b ≠ 0
3     r = a mod b
4     a = b
5     b = r
6 return a
```

VARIABLES

a = 48 b = 18

Euclid Step by Step: GCD(48, 18)



b ≠ 0 → 18 ≠ 0 → **true**: pass 1 begins.

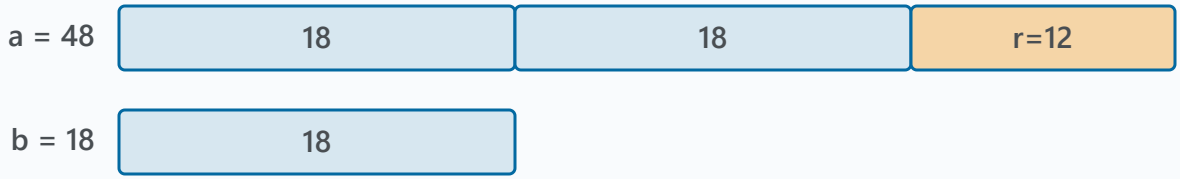
Step 2 / 15

```
1 GCD(a, b)
▶ while b ≠ 0
3     r = a mod b
4     a = b
5     b = r
6 return a
```

VARIABLES

a = **48** b = **18**

Euclid Step by Step: GCD(48, 18)



48 = 2 × 18 + 12 → remainder r = 12

r = a mod b = 48 mod 18 = **12**: 2 pieces of length b fit into a, leaving 12.

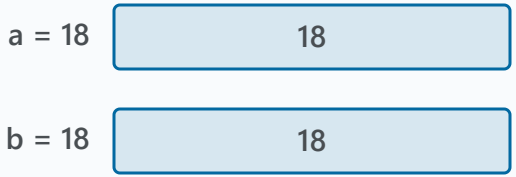
Step 3 / 15

```
1 GCD(a, b)
2 while b ≠ 0
▶     r = a mod b
4     a = b
5     b = r
6 return a
```

VARIABLES

a = 48 | b = 18 | r = 12

Euclid Step by Step: GCD(48, 18)



a = b → a = 18. The smaller number becomes the dividend.

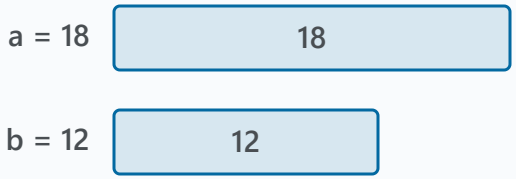
Step 4 / 15

```
1 GCD(a, b)
2 while b ≠ 0
3     r = a mod b
▶     a = b
5     b = r
6 return a
```

VARIABLES

a = 18 | b = 18 | r = 12

Euclid Step by Step: GCD(48, 18)



b = r → b = 12. A common divisor also divides the remainder; the problem got smaller.

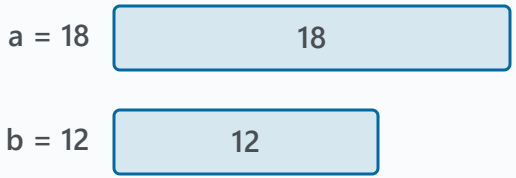
Step 5 / 15

```
1 GCD(a, b)
2 while b ≠ 0
3     r = a mod b
4     a = b
▶     b = r
6 return a
```

VARIABLES

a = 18 | b = 12 | r = 12

Euclid Step by Step: GCD(48, 18)



b ≠ 0 → 12 ≠ 0 → **true**: pass 2 begins.

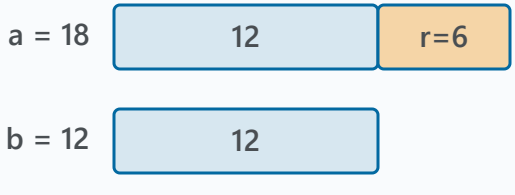
Step 6 / 15

```
1 GCD(a, b)
▶ while b ≠ 0
3     r = a mod b
4     a = b
5     b = r
6 return a
```

VARIABLES

a = **18** b = **12** r = **12**

Euclid Step by Step: GCD(48, 18)



18 = 1 × 12 + 6 → remainder r = 6

r = a mod b = 18 mod 12 = **6**: 1 piece of length b fits into a, leaving 6.

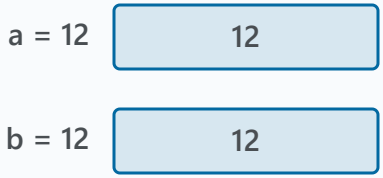
Step 7 / 15

```
1 GCD(a, b)
2 while b ≠ 0
▶     r = a mod b
4     a = b
5     b = r
6 return a
```

VARIABLES

a = **18** b = **12** r = **6**

Euclid Step by Step: GCD(48, 18)



a = b → a = 12. The smaller number becomes the dividend.

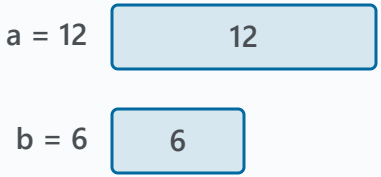
Step 8 / 15

```
1 GCD(a, b)
2 while b ≠ 0
3     r = a mod b
▶     a = b
5     b = r
6 return a
```

VARIABLES

a = 12 | b = 12 | r = 6

Euclid Step by Step: GCD(48, 18)



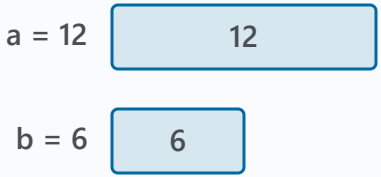
b = r → b = 6. A common divisor also divides the remainder; the problem got smaller.


```
1 GCD(a, b)
2 while b ≠ 0
3     r = a mod b
4     a = b
▶     b = r
6 return a
```

VARIABLES

a = 12 | b = 6 | r = 6

Euclid Step by Step: GCD(48, 18)



b ≠ 0 → 6 ≠ 0 → **true**: pass 3 begins.

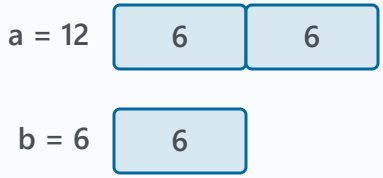
Step 10 / 15

```
1 GCD(a, b)
▶ while b ≠ 0
3     r = a mod b
4     a = b
5     b = r
6 return a
```

VARIABLES

a = **12** b = **6** r = **6**

Euclid Step by Step: GCD(48, 18)



12 = 2 × 6 + 0 → remainder r = 0

r = a mod b = 12 mod 6 = **0**: 2 pieces of length b fit into a, leaving 0.

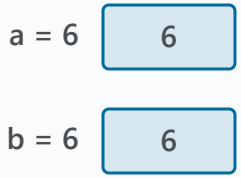
Step 11 / 15

```
1 GCD(a, b)
2 while b ≠ 0
▶     r = a mod b
4     a = b
5     b = r
6 return a
```

VARIABLES

a = 12 | b = 6 | r = 0

Euclid Step by Step: GCD(48, 18)



a = b → a = 6. The smaller number becomes the dividend.

Step 12 / 15

```
1 GCD(a, b)
2 while b ≠ 0
3     r = a mod b
▶     a = b
5     b = r
6 return a
```

VARIABLES

a = 6 | b = 6 | r = 0

Euclid Step by Step: GCD(48, 18)

a = 6 | 6

b = 0 (0: nothing left to divide)

b = r → b = 0. A common divisor also divides the remainder; the problem got smaller.

Step 13 / 15

```
1 GCD(a, b)
2 while b ≠ 0
3     r = a mod b
4     a = b
▶     b = r
6 return a
```

VARIABLES

a = 6 | b = 0 | r = 0

Euclid Step by Step: GCD(48, 18)

a = 6 | 6

b = 0 (0: nothing left to divide)

b ≠ 0 → 0 ≠ 0 → **false**: the loop ends.

Step 14 / 15

```
1 GCD(a, b)
▶ while b ≠ 0
3     r = a mod b
4     a = b
5     b = r
6 return a
```

VARIABLES

a = 6 | b = 0 | r = 0

Euclid Step by Step: GCD(48, 18)

a = 6 **6**

b = 0 (0: nothing left to divide)

return a → **6**. GCD(48, 18) = 6: both 48 and 18 are measured exactly by pieces of length 6. Done in 3 passes.

Step 15 / 15

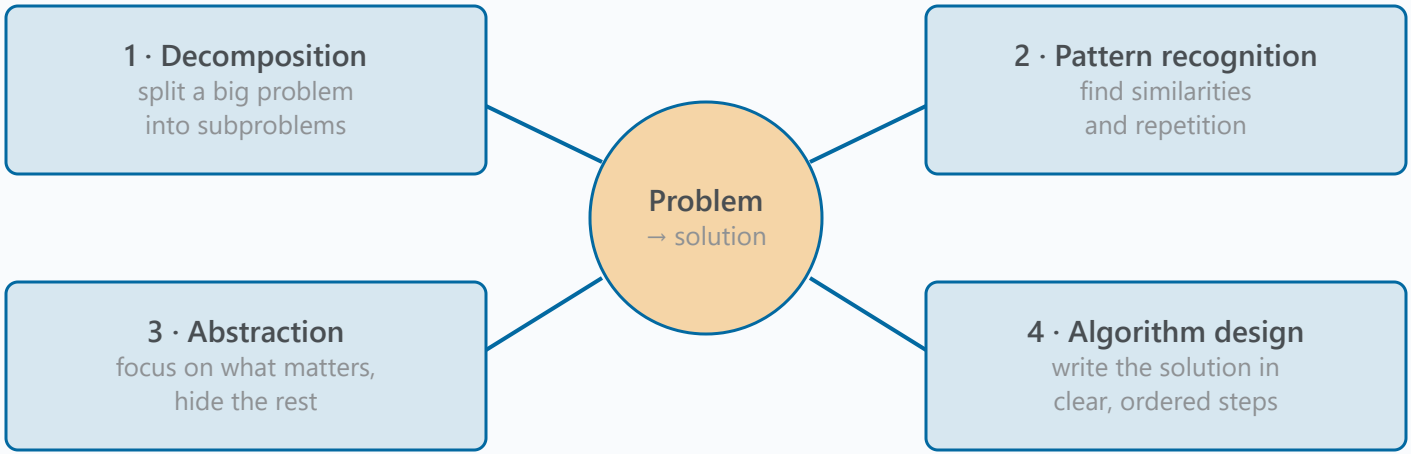
```
1 GCD(a, b)
2 while b ≠ 0
3     r = a mod b
4     a = b
5     b = r
▶ return a
```

VARIABLES

a = 6 b = 0 r = 0

Computational Thinking

Computational thinking is the practice of turning problems into forms that a computer or a person can solve systematically.



Decomposition

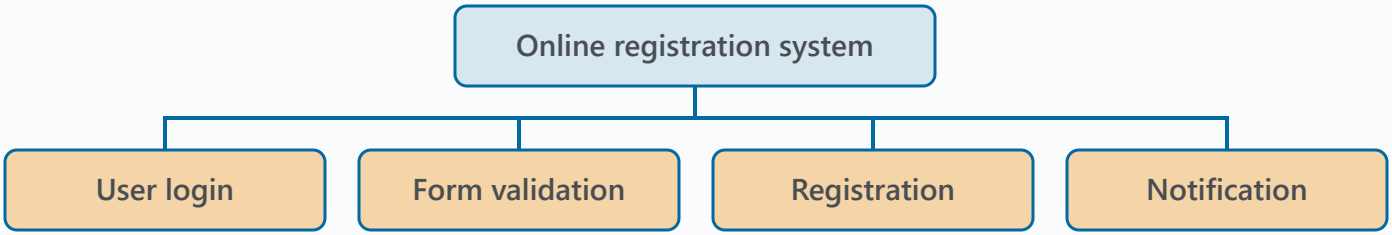


The big problem: an **online registration system**. Too hard to tackle in one piece.

Step 1 / 3

- Each subproblem can be developed and tested separately.
- The inputs and outputs of subproblems are clearly defined.
- Dependencies show up early.

Decomposition

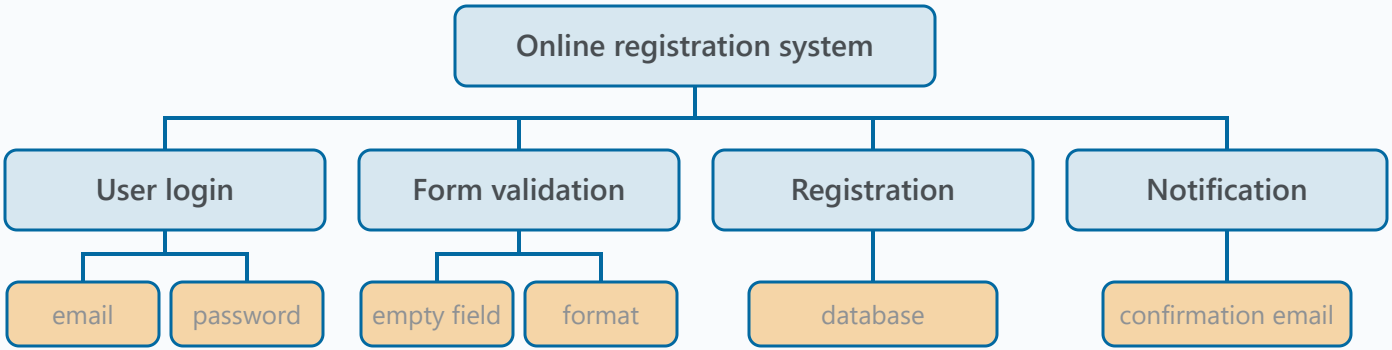


It is split into main subtasks, each with its own clearly defined inputs and outputs.

Step 2 / 3

- Each subproblem can be developed and tested separately.
- The inputs and outputs of subproblems are clearly defined.
- Dependencies show up early.

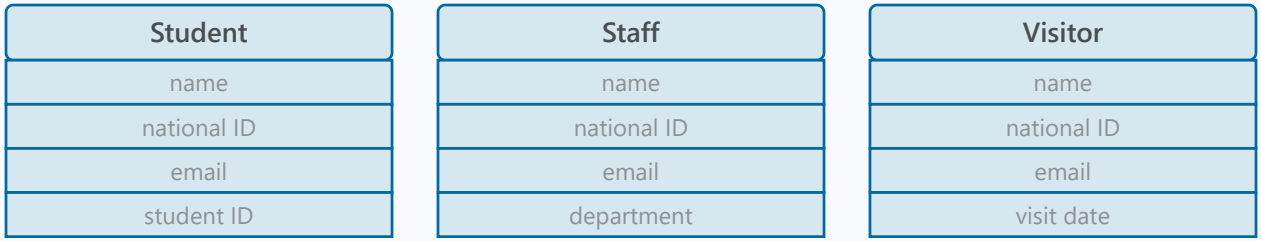
Decomposition



Split further if needed. Each leaf is a small job that can be written and tested on its own.

- Each subproblem can be developed and tested separately.
- The inputs and outputs of subproblems are clearly defined.
- Dependencies show up early.

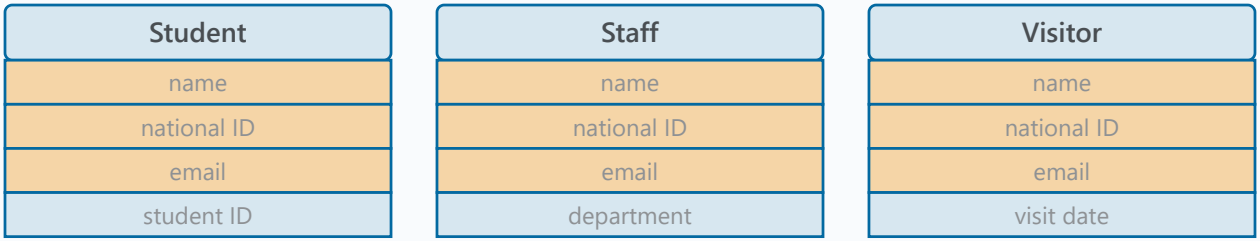
Patterns and Abstraction



Three different record types: student, staff and visitor.

Good abstraction factors out repeated structure without losing the important differences.

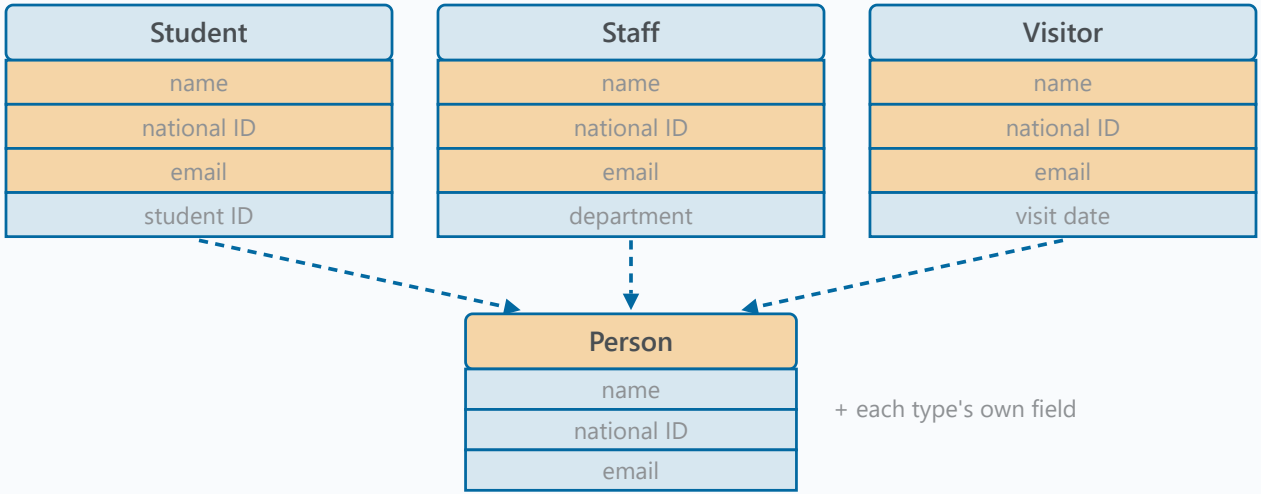
Patterns and Abstraction



Pattern recognition: the name, national ID and email fields repeat in all three.

Good abstraction factors out repeated structure without losing the important differences.

Patterns and Abstraction



Abstraction: the shared fields go into a single `Person` model; the differences stay only in the types that need them.

Good abstraction factors out repeated structure without losing the important differences.

Algorithm Design and Testing



- Consider normal, smallest, largest and invalid inputs.
- Check that each step produces the expected state.
- When you find a bug, fix the root cause, not just the symptom.

SUMMARY

Chapter Summary

- Computation is processing data according to rules; it is not limited to arithmetic.
- The stored-program idea, keeping instructions in memory like data, made the general-purpose computer possible.
- The Turing machine is an abstract model of algorithmic computation; each step is determined by the state and the symbol read.
- Imperative knowledge tells how to reach a result; Euclid's algorithm is a historical example.
- Computational thinking combines decomposition, pattern recognition, abstraction and algorithm design.
- Time, memory, energy and computability set the limits of computers.

Review Questions

1. Why is computation not just arithmetic?
2. Give an example each of declarative and imperative knowledge.
3. How does a fixed-program machine differ from a stored-program machine?
4. What determines the next step of a Turing machine?
5. Why does Euclid's algorithm always halt?
6. How would you approach a problem with the four components of computational thinking?

Exercises

1. Trace **GCD(84, 36)** with Euclid's algorithm; record a, b and r for each pass in a table.
2. Classify a calculator, a washing machine controller and a smartphone as **fixed-program** or **stored-program**, and justify your answer.
3. Which details of a real city does a subway map hide, and which does it keep? Why does it work as an abstraction?
4. Run the add-one Turing machine with **1011** on the tape (head on the rightmost digit); what value is on the tape when it halts?
5. Do three passes of the square-root method with x = 10 and guess = 1; what value is the guess approaching?
6. Decompose a **library lending system**: at least four subtasks with their inputs/outputs; which fields do member and staff records share?

References

- Wing, Computational Thinking, Communications of the ACM, 2006.
- Cormen et al., Introduction to Algorithms.
- Downey, Think Java.
- Sipser, Introduction to the Theory of Computation.

The End

Chapter 1.5: Computation and Computational Thinking

Previous: Chapter 1.4: The Internet · Next: Chapter 2.1: Software and Operating Systems