ALGORITHMS AND DATA STRUCTURES I

Course 1

Course content

- Short intro
- Problem-solving
- What is an algorithm?
- Properties of an algorithm
- How to describe algorithms
- Types of data used
- Specifying operations in an algorithm

About course

Teachers

Course: Flavia Micota

Seminary:

- Fabian Galiș
- Mario Reja

Information about course ... syllabus

Course subjects

Bibligraphy

Evaluation

Communication

Email: flavia.micota@e-uvt.ro

Office hours (consultații)

About yourself …

https://forms.gle/6xhvynhQzaPWGr6V6

Algorithms & Data Structures

Algorithm

- Algorithms are procedures that specify how to do a needed task or solve a problem.
- Algorithms are used to standardize processes and communicate them between different people.
- Algorithms are like recipes, tax codes, and sewing patterns.

Abstractisation --> Data Structures

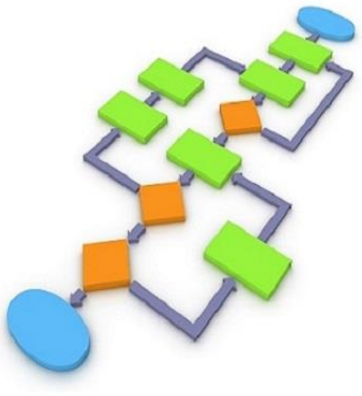
- Abstraction is the concept of representing ideas at different levels of detail by identifying what is essential.
- This can be done by making something more general, or by encoding information at multiple levels.
- For example, consider that a textbook is also a book, or how words can be represented as braille or written text.

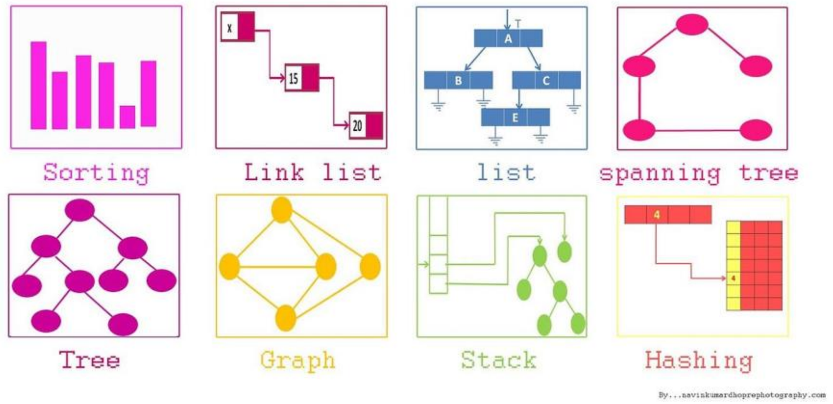
Algorithms & Data Structures

Good program → combination between Algorithms and Data Structures

- Algorithm
  - Step by step recipe for solving an instance of a problem
  - Operations

- Data Structure
  - Logical representation that exists between individual elements used to carry out a task
  - Organize data, identify relation, identify operation specific for that data





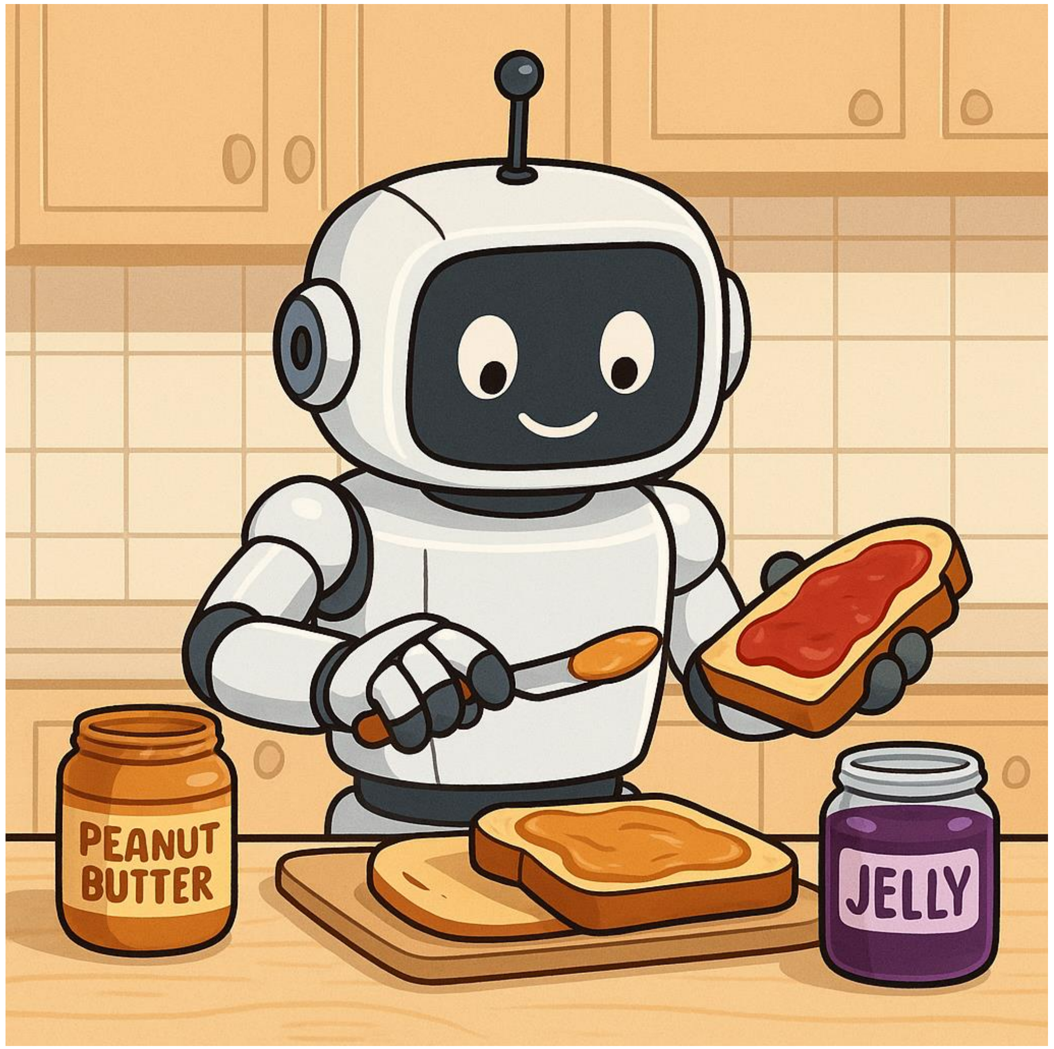
Which is the origin of algorithmic word

- The term comes from the name of the Persian mathematician Muḥammad ibn Mūsā al-Khwārizmī (c. 780–850).
- He lived in Baghdad and worked at the House of Wisdom, where Greek and Indian mathematical works were translated into Arabic.
- His book Al-Kitāb al-Mukhtaṣar fī ḥisāb al-jabr wa’l-muqābala (The Compendious Book on Calculation by Completion and Balancing) introduced systematic methods for solving linear and quadratic equations — this also gave us the word algebra (from al-jabr).
- His name, al-Khwārizmī was Latinized in the 12th century as Algoritmi or Algorismus.

Algorithms examples

- Day by day usage
  - Usage of an ATM, usage of a caffe machine, cooking a recipe …
- Mathematical algorithms
  - Euclid's Algorithm (considered the first algorithm)
    - Determining the greatest common divisor of two numbers
  - Eratosthenes' Algorithm
    - Generating prime numbers less than a given value
  - Horner's Algorithm
    - Calculating the value of a polynomial or the remainder of dividing by a binomial of the form (X-a)

Making a Peanut Butter and Jelly Sandwich



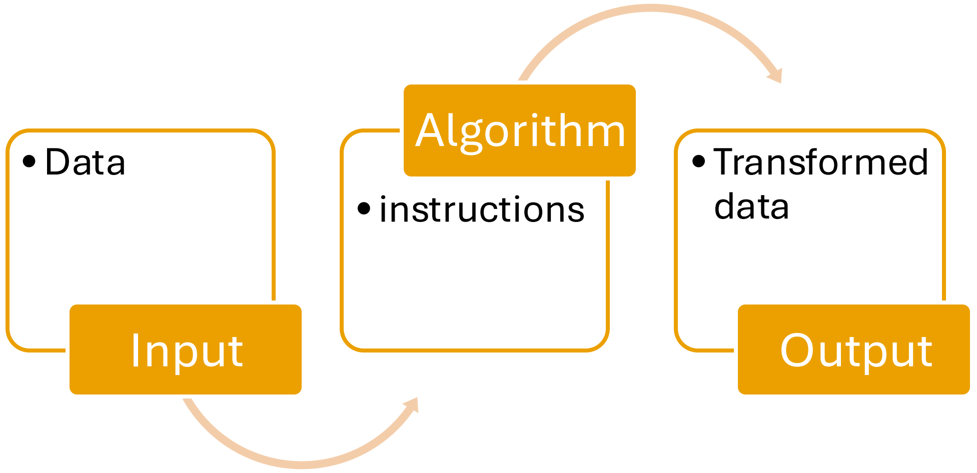
- Work with a partner to write a list of instructions (an algorithm!) on how to make a peanut butter and jelly sandwich.
- We'll test your instructions in a few minutes...

Making a Peanut Butter and Jelly Sandwich

1. Open bag of bread
2. Reach hand in and take out 2 slices of bread
3. Place each slice on a plate
4. Open jar of peanut butter
5. Pick up knife and stick sharp side of knife into open jar
6. Use knife to scoop out peanut butter
7. Wipe and spread peanut butter on one slice of bread
8. Repeat 5, 6, 7 until slice of bread is covered in peanut butter. Close jar
9. Open jar of jelly
10. Pick up knife and stick sharp side of knife into open jar
11. Use knife to scoop out jelly
12. Wipe and spread jelly on other (non-PB) slice of bread
13. Repeat 10, 11, 12 until the slice of bread is covered in jelly. Close jar.
14. Put the peanut butter side of one slice of bread on the jelly side of the other.
15. Pick up and eat sandwich

How to describe an algorithm?

- **Name & purpose**: one sentence on what problem it solves.
- **Inputs**: names, types, and constraints (e.g., “array A of n integers; n ≥ 1”).
- **Outputs**: exact result and type (e.g., “index of minimum element”).
- **Preconditions**: what must be true before running (e.g., “A is sorted”).
- **Postconditions**: what must be true after (e.g., “returned i s.t. A[i] ≤ all A[j]”).
- **High-level idea**: 2–4 lines explaining the strategy (intuition).
- **Pseudocode**: clear, language-agnostic steps.
- **Complexity**: time and space, with the dominant term and brief reasoning.
- **Correctness sketch**: invariants or proof idea (why it works).
- **Edge cases**: empty inputs, duplicates, ties, overflow, non-ASCII, etc.
- **Example**: a tiny, worked input→output trace.



How to describe an algorithm? Example: Euclid Algorithm

- **Name & purpose**: Euclid's Algorithm. Determining the greatest common divisor of two numbers
- **Inputs**: a, b – natural positive numbers
- **Outputs**: greatest common divisor of a and b
- **Preconditions**: a greater or equal to b
- **Postconditions**: -
- **High-level idea**:
  - divide a to b, store the remainder
  - Divide b to remainder and store the new remainder
  - Repeat the division while the remainder is different from zero
  - The last remainder different from zero is the result
- **Pseudocode**:
  - **Step1:** Assign the dividend the value of a and the divisor the value of b
  - **Step2:** Calculate the remainder of dividing the dividend by the divisor
  - **Step 3:** Assign the dividend the value of the divisor and the divisor the value of the remainder calculated in Step 2
  - **Step 4:** If the remainder is nonzero, resume from Step 2
- **Complexity**: O(log min(a,b))
- **Correctness sketch**: gcd(a,b)=gcd(a0,b0), where a0,b0 are the original inputs
- **Edge cases**:
  - Inputs are negative or zero
- **Example**: gcd(12, 9) = 3, : gcd(12, 17) = 1

When developing an algorithm …

Take in consideration

- It should be simple.
- It should be clear with no ambiguity.
- It should lead to a unique solution of the problem.
- It should involve a finite number of steps to arrive at a solution
- It should have the capability to handle some unexpected situations which may arise during the solution of a problem (for example, division by zero)

Be aware that

- It could be time consuming do develop
- Difficult to represent complex tasks
- Could appear issues with inflexibility, resource intensity, bias, security risks

Designing good algorithms

- Designing good algorithms is a large part of computer science. When we represent algorithms as program code, we are communicating with a computer to tell it how to do a specific task.

- What makes an algorithm 'good’?
  - It should specify what is needed at the beginning (input)
  - It should specify what is produced at the end (output)
  - It should produce the right output for all given inputs (correctness)
  - It shouldn't take too long to finish, or use large computing resources, like memory (efficiency)
  - Others should be able to understand and modify it as needed (clarity)

What properties has to have an algorithm?

- An algorithm must be:
  - General = applicable to a whole class of problems not just a few particular cases
  - Correct (including finite)
  - Efficient

What properties has to have an algorithm? General

- An algorithm provides the right answer in all cases

- Example:
  - Consider the problem of increasing sorting a sequence of numbers

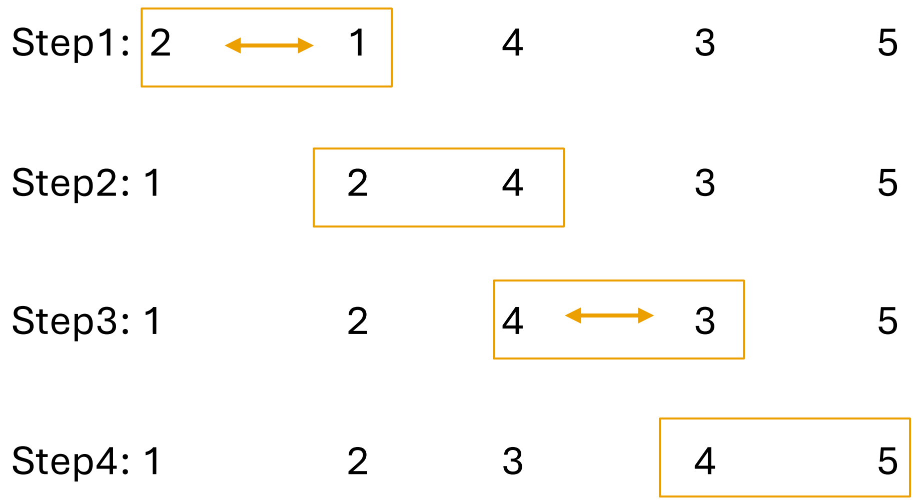
- For example
  - Input (2,1,4,3,5) expected output (1,2,3,4,5)

What properties has to have an algorithm?

General

An algorithm provides the right answer in all cases

Sequence



Method description

- Compare the first two elements; if they are not in the desired order, swap them
- Compare the second with the third and apply the same strategy
- …
- Continue the process until the last two elements in the sequence

For this example apply the algorithm -> correct solution

What properties has to have an algorithm?

General

An algorithm provides the right answer in all cases

Sequence

Step1:	2	3	5	4	1
Step2:	2	3	5	4	5
Step3:	2	3	5 ⟷	4	1
Step4:	2	3	4	5 ⟷	1
Step4:	2	3	4	1	5

Description

- Compare the first two elements; if they are not in the desired order, swap them
- Compare the second with the third and apply the same strategy
- …
- Continue the process until the last two elements in the sequence

For this example apply the algorithm -> wrong solution => THE ALGORITHM IS NOT GENERAL

What properties has to have an algorithm?

Correct

After a finite number of operations to the correct result for any input data values that belong to the problem domain

- Example
  - Step 1: Assign the value 1 to a and the value 2 to b ($a \leftarrow 1$, $b \leftarrow 2$)
  - Step 2: modify a: $a \leftarrow 3 - a$
  - Step 3: modify b: $b \leftarrow 3 - b$
  - Step 4: If $a + b > 1$ then resume from Step 2 otherwise STOP

- What is the value of a+b before executing step 4? The sequence of processing ends?

What properties has to have an algorithm?

Efficient

The correctness is not enough if the time needed to obtain the result is large

- Example: Consider a number consisting of 10 distinct digits.

- Determine the next element in the ascending sequence of natural numbers consisting of 10 distinct digits.
    - 0123456789, 0123456798, ..., 9876543210

- Example: x= 6309487521 → 6309512478

- What is the next number (in ascending order) that contains 10 distinct digits?

What properties has to have an algorithm?

Efficient

The correctness is not enough if the time needed to obtain the result is large

- First approach:
  - Step 1: generate all numbers in ascending order from 0 to 9876543210
  - Step 2: search the list for the first number that has distinct digits and is greater than x

- Second approach:
  - Step 1: generate only numbers consisting of 10 distinct digits
  - Step 2: sort the generated list in ascending order
  - Step 3: search the list for the first number that is greater than x

- Which variant is more efficient?

What properties has to have an algorithm?

Efficiet

The correctness is not enough if the time needed to obtain the result is large

- First approach:
  - Step 1: generate all numbers in ascending order from 0 to 9876543210
    - How many numbers are generated? 10 at power10
  - Step 2: search the list for the first number that has distinct digits and is greater than x
    - How many comparison are executed? cca 10 at power10
- Second approach:
  - Step 1: generate only numbers consisting of 10 distinct digits
    - How many numbers are generated?? 10!=3628800
  - Step 2: sort the generated list in ascending order
    - How many comparison are executed? 10! Log(10!) = 2*10^7
  - Step 3: search the list for the first number that is greater than x
    - Log(10! Log(10!))
- Which variant is more efficient?

Starting to write an algorihm
Algorithm Data

Data = entity carrying information

= container containing a value (values)

Features:

- name
- value
    - constant (the same value during the execution of the algorithm)
    - variable (the value changes during the execution of the algorithm)
- type
    - simple (ex: numbers, characters, truth values , etc.)
    - structured (ex: paintings , articles, etc.)

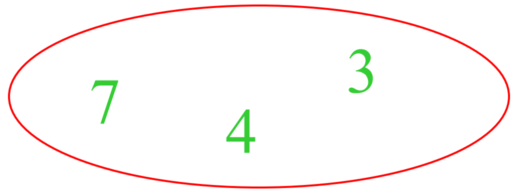
Data Types

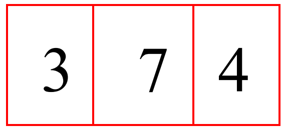
- Simple data types (default programming languages data types)
  - Integer   int < variable name >
  - Real real < variable name >
  - Logical bool < variable name >
  - Characters char < variable name >
  - Remark. There are programming languages (e.g. Python) that do not require explicit declaration of data

- Abstract data types
  - Implemented in libraries (includes data structured that will be discussed this and next semester)
  - Defined by user (mostly discussed in context of object oriented programming)

Data Types

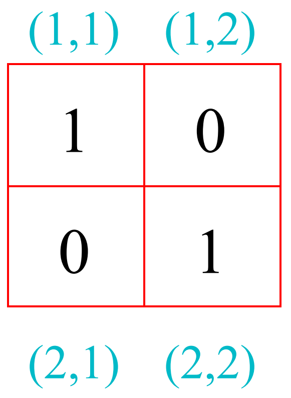
- Sets (remark: {3,7,4}={3,4,7})
  - The order of the elements does not matter
- Sequence (remark: (3,7,4) is different from (3,4,7))
  - The order of the elements is important
- Tables
  - Uni-dimensional table
    - <element type> <name>[n1..n2]
  - Bi dimensional tables (Matrices)
    - <element type> <name>[n1..n2][m1..m2]
  - Multidimensional





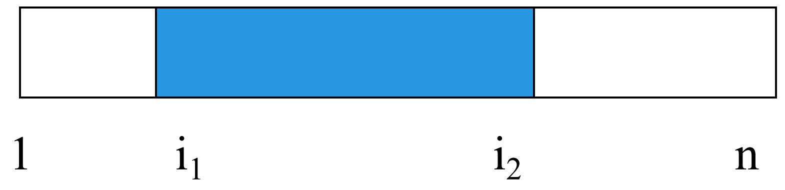
Index: 1 2 3

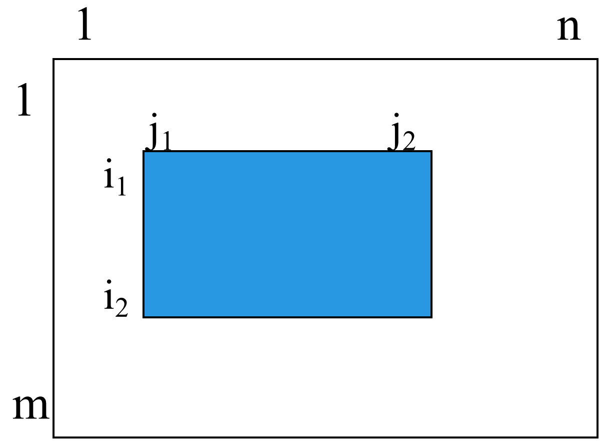
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$



Data Types

	Uni dimensional	Bi-dimensional
Declare	<element type> <name>[n1..n2] e.g. float x[1..n]	<element type> <name>[m1..m2, n1..n2] e.g. float x[1..n][1..m]
Specifying an element	x[i] where - i is the element index	A[i,j] where - i is row index - j is column index
Specifying subtables	$x[i_1..i_2]$ $1 \le i_1 < i_2 \le n$	$A[i_1..i_2, j_1..j_2]$ $1 \le i_1 < i_2 \le m, 1 \le j_1 < j_2 \le n$





Abstract data types

- An abstract data type (ADT) is a model of a data structure that specifies
  - the characteristics of the collection of data
  - the operations that can be performed on the collection
- It's abstract because it doesn't specify how the ADT will be implemented
  - does not commit to any low-level details
- A given ADT can have multiple implementations
- Example
  - Students from an academic year

Abstract data types

Students from an academic year

Characteristics

- Study program name
- Study program type
- Academic year
- Students

Operations

- add (student)
- remove(student)
- Get_number_of students()
- …

(Computer Science in English, BSc, 2024-2025, StudentList)
(Artificial Intelligence and Distributed Computing, 2023-2024, MSc, StudentList)
(Artificial Intelligence, BSc, 2024-2025, StudentList)
(Computer Science in English, BSc, 2024-2025, StudentList)
(Artificial Intelligence, BSc, 2025-2026, StudentList)

Order
- Does not meter
- Order by academic year
- Organize by study program type and name
- Relation between subjects

How can algorithms be described?

- Flowcharts
  - Graphical representation
  - Can be used to generate code
  - Advantages:
    - Communication: They are good visual aid to understand the program.
    - Proper program documentation
  - Limitations
    - Complex logic
    - No uniform practice
- Pseudocode (an artificial language)
  - It is somewhat similar to a programming language (code)
  - But it is not as rigorous as a (pseudo) programming language
- Programming languages (Python, C/C++, Java, …)
  - Much strict, used to implement algorithms on computers

Does algorithm depend on programming language?

How can processing in an algorithm be specified?

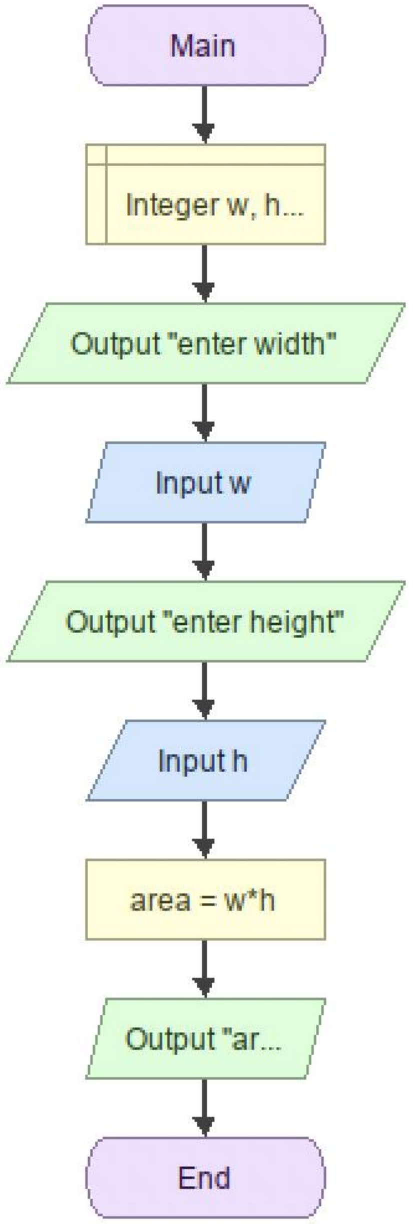
- Instructions
  - An action (operation/step) performed by an algorithm
  - Types of instructions
    - Simple
      - Assignment (assigns a value to a variable)
      - Transfer (takes input ; displays results)
      - Control (specifying what is the next step to be executed)
    - Structured
      - combine simpler instructions into more complex blocks

Assignment

- **Assignment**
  - Aim
    - assign a value one variables
  - Description

    v ⟵< expression > or v : =< expression > or v=< expression >

- **Expression** = construction syntactic (= sequence of symbols that respect some known rules) used to describe a calculation
  - Consists of
    - Operands : variables , constant values
    - Operators : arithmetic, relational, logic



Operators

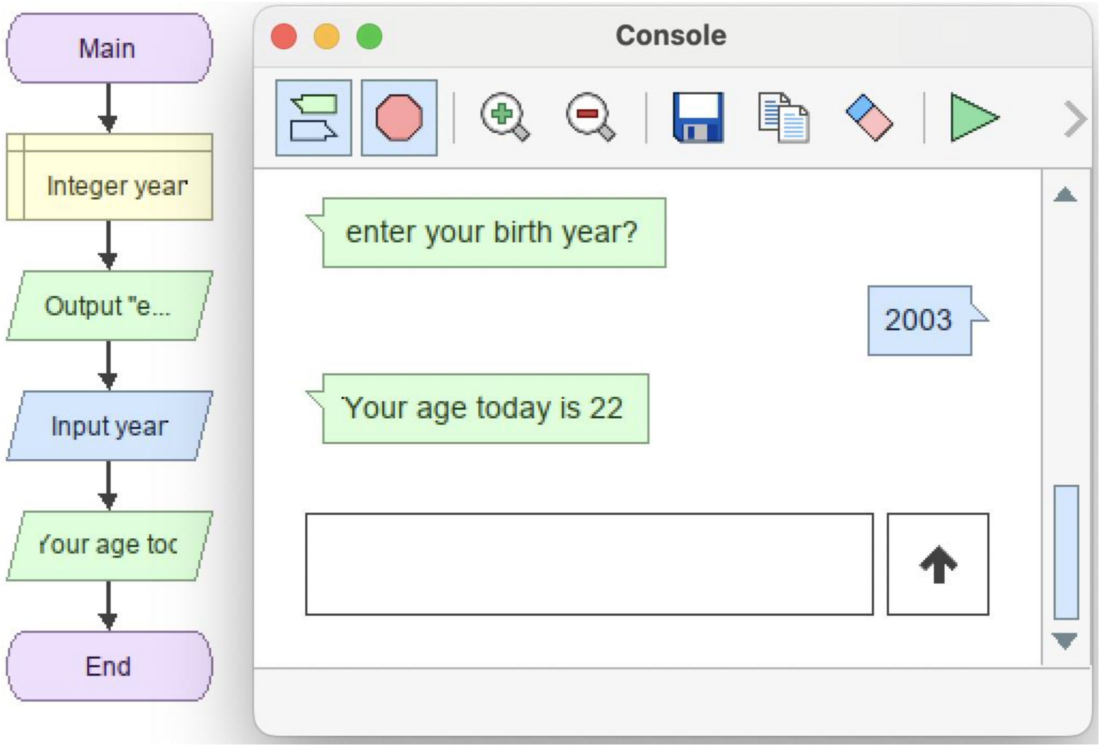
Characterized

- Execution order
    - Use () to group
- Operands numbers
- Operator priority

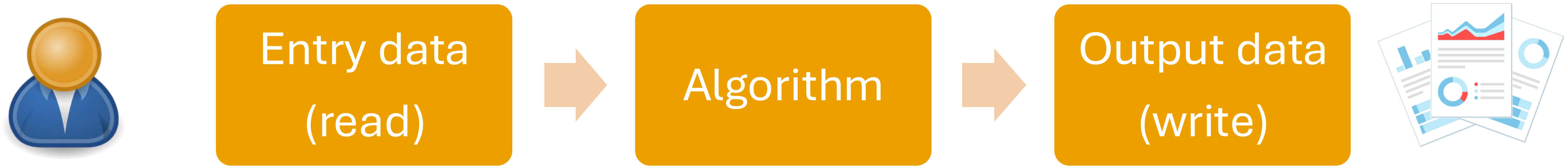
Type	Operator	Pseudocode	Python
Arithmetic	addition	+	+
	subtraction	-	-
	Multiplication	*	*
	ascension to power	^	**
	Real division	/	/
	Integer division quotient	DIV	//
	Integer division remainder	MOD	%
Relational	Equal	=	==
	Different	≠, <>	!=
	Strict smaller / smaller or equal	< ≤	< <=
	Strict greater / greater or equal	> ≥	> >=
Logical	Disjunction	or |	or
	Conjunction	and &	and
	Negation	not !	not

Input/output

- Scope:
  - Takes input data
  - It delivers results
- Description:
  - read v1,v2,...
  - write r1,r2,...



```
enter your birth year?
                                  2003
Your age today is 22
```



Examples in Python

Rectangle area

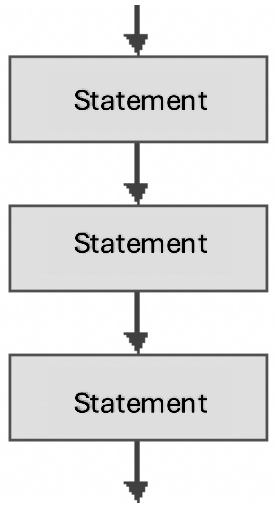
```
w=input("Enter width=")
h=input("Enter height=")

area=w*h

print("Area =", area)
```

Current age

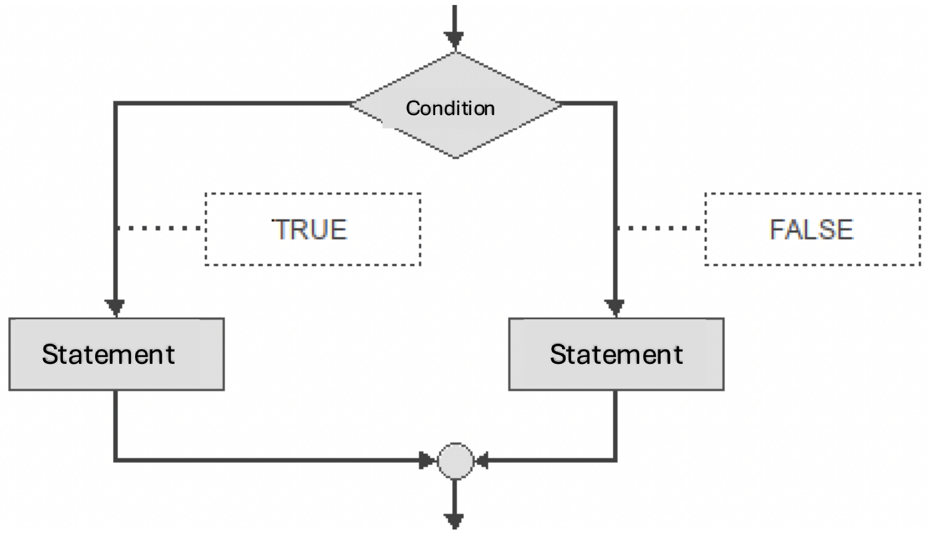
```
birth_year = input("Enter your birth year")

age = 2025 - birth_year

print("Your age today is ", age)
```

Processing structures

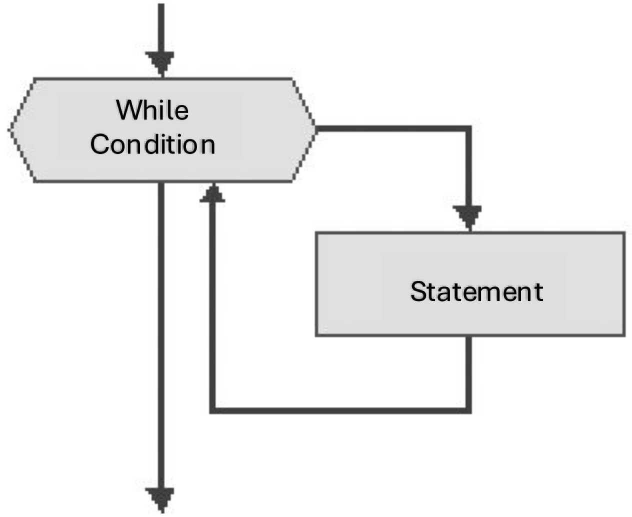
Sequence of instructions



Decision instructions (conditional)



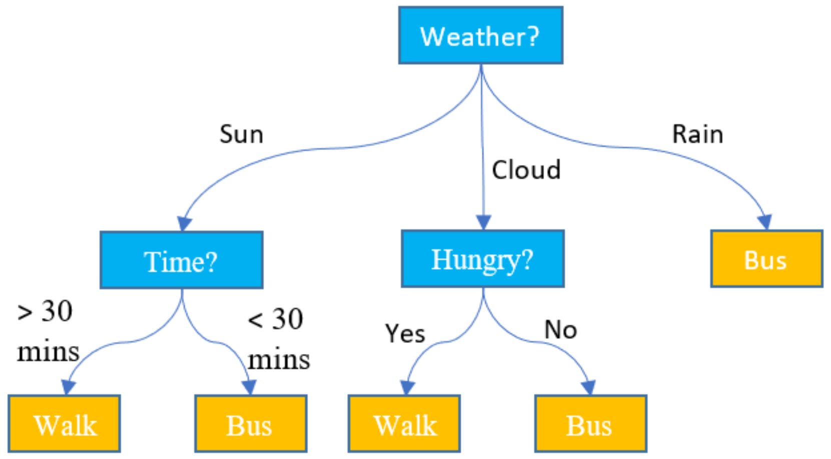
Looping instruction (repetitive)



Decisional instructions

- Purpose: allows the choice between two or more processing options depending on the fulfillment /non-fulfillment of one (some) condition

- Examples
  - Is a natural number odd or even?
  - Decision tree (Walk or take bus to home)

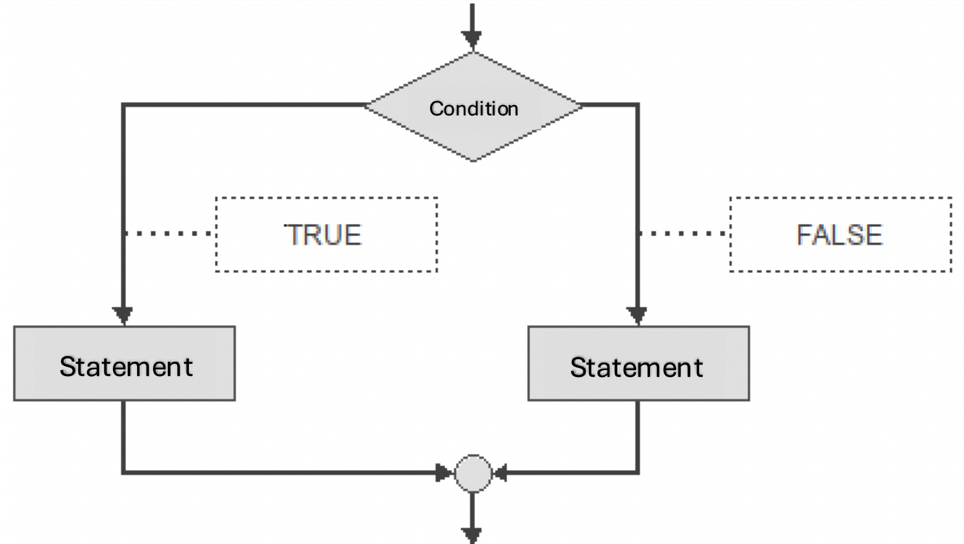


Conditional instructions

- General form

Pseudocode	Python

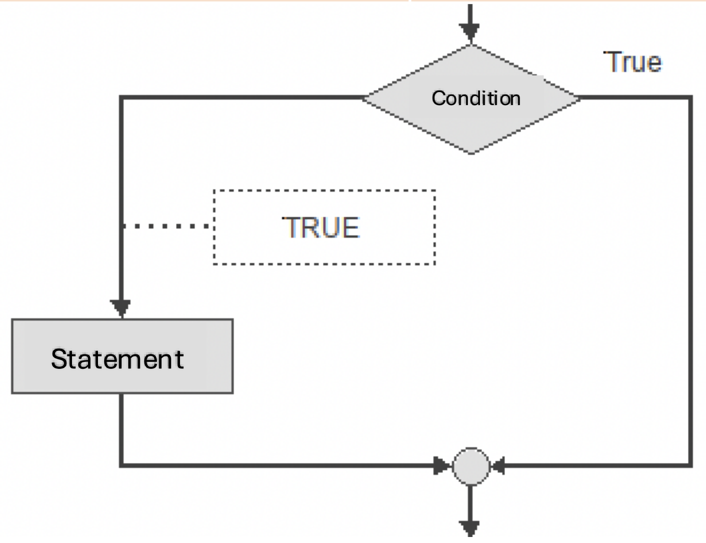
```
If <condition> then
  statement
Else
  statement
Endif
```
	```
	if <condition>:
	  statement
	else:
	  statement
	```



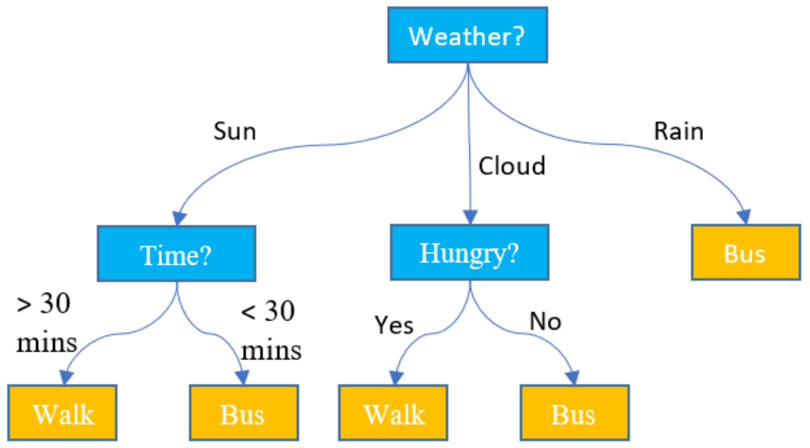
- Simplified form

Pseudocode	Python

```
If <condition> then
  statement
Endif
```
	```
	if <condition>:
	  statement
	```



Conditional instructions



Odd/Even number

Pseudocode

```
read n
if n MOD 2 = 0 then
  write "Even number"
else
  write "Odd number"
endif
```

Python

```
n=input("n=")
if n%2==0:
  print("Even number")
else:
  print("Odd number")
```

Decision tree

```
answer = input("weather?")
if answer == "sun":
    answer  = input("time less < 30 min?")
    if answer == "yes": print("Bus") else: print("Walk")
elif answer == "cloud" :
    answer  = input("Hungry?")
    if answer == "yes": print("Walk") else: print("Bus")
elseif answer=="rain":
    print("get bus")
else:
    print("unknown answer")
```

Loop instructions

- Purpose: allow the repetition of a processing
- Example: sum calculation

  S= 1+2+…+i+…+n
- A cycle is characterized by:
  - The processing step to be repeated

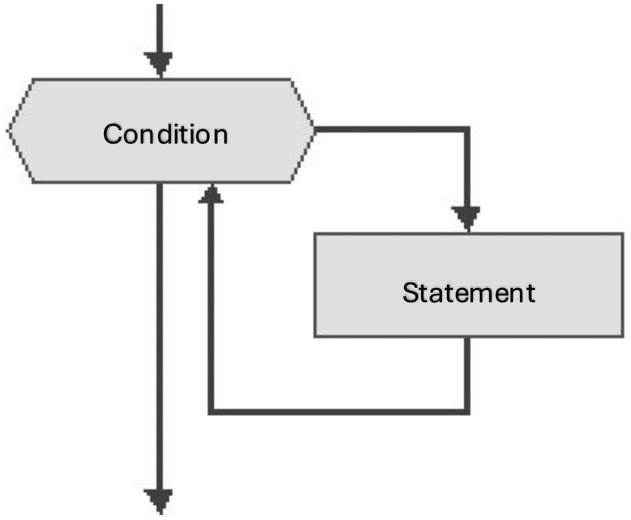
    (e.g. adding the next value to the current value of the amount)
  - A condition to stop (continue) repetitive

    (e.g. all values have already been collected)
- Depending on the moment when the continue/stop condition is analyzed there is :
  - Previously Conditioned Loops (WHILE)
  - Posterior conditioned Loops (REPEAT)

While instruction

Pseudocode	Python

`While <condition> do`
`   statement`
`Endwhile`
	`while <condition>:`
	`  statement`



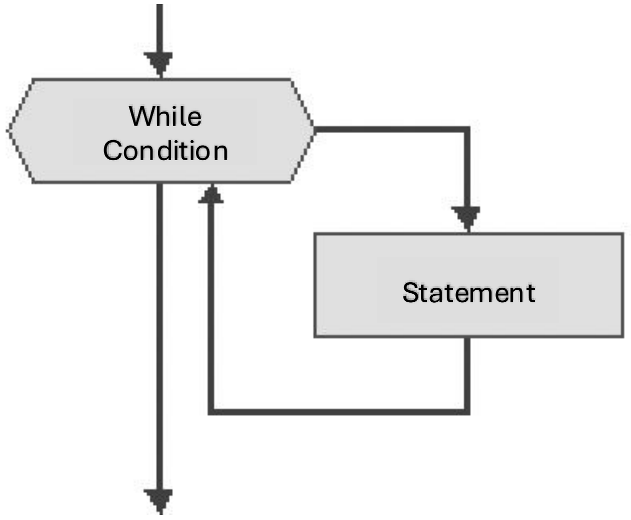
- The continuation condition is analyzed
- If the condition it is true, one of the instructions in the body of the cycle is executed, after which the condition is evaluated again
- When the condition becomes false, it goes to the next processing in the algorithm
- If the condition never becomes false the loop is infinite
- If the condition is false from the start then the loop body is never executed

While instruction. Exemple

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$

Pseudocode	Python

`While <condition> do`
`   statement`
`Endwhile`
	`while <condition>:`
	`  statement`



Prepare variable to collect the result

Initialize loop step

Modify loop step

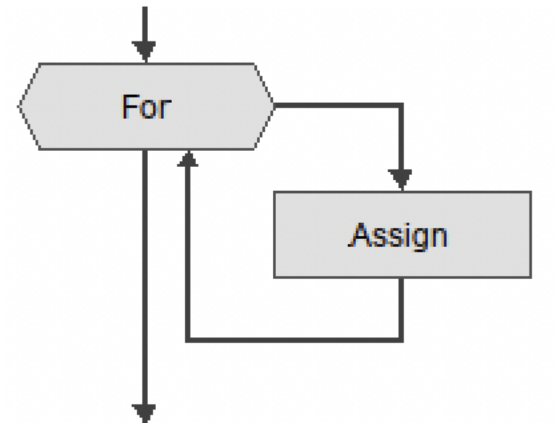
Pseudocode	Python

`s=0`
`i=0`
`While i≤n do`
`   s=s+i`
`   i=i+1`
`Endwhile`
`Write "s="+s`
	`n=input("n=")`
	`s=0`
	`i=1`
	`while i<=n:`
	`    s=s+i`
	`    i=i+1`
	`print("s=",s)`

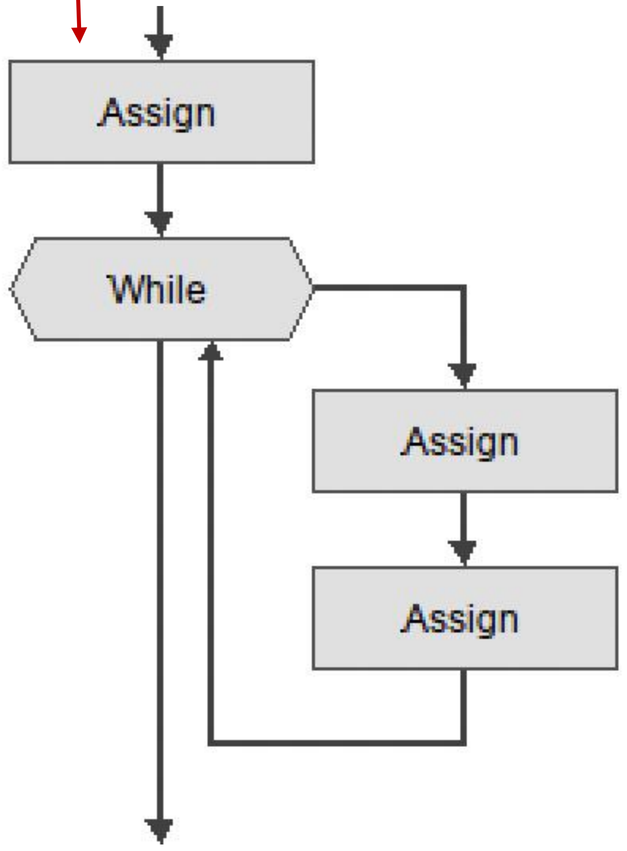
For instruction

Pseudocode	Python

`for iterator=start,stop,step do`
`  statement`
`Endfor`
	`for iterator in range(start, stop, step):`
	`    statement`



Expand for with while



For instruction. Example

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$

Pseudocode	Python

`Read n`
`s=0`
`For it<-1,n,1 do`
`  s=s+it`
`Write "s="+s`
	`n=input("n=")`
	`s=0`
	`for it in range(1,n+1,1):`
	`  s=s+it`
	`print("s=", s)`

range() function generates a list of numbers in interval [1,n+1)

Pseudocode	Python

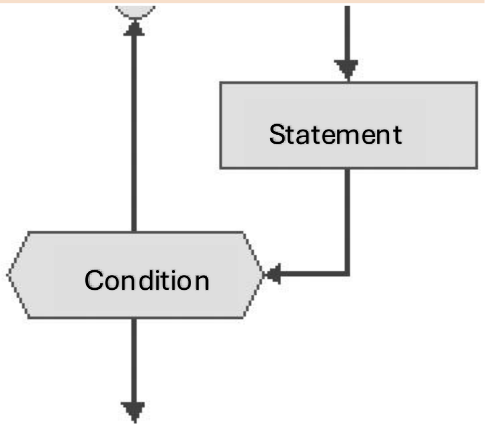
`Read n`
`s=0`
`For it<-n,1,-1 do`
`  s=s+it`
`Write "s="+s`
	`n=input("n=")`
	`s=0`
	`for it in range(n,0,-1):`
	`  s=s+it`
	`print("s=", s)`

Sometimes starting with a larger value and reaching a lower value is needed

Repeat-until instruction

Pseudocode	Python

```
Repeat
  statement
Until <condition>
```
	• No equivalent
	• Can be simulated with a while loop, negating the repeat-until condition, and executing the loop body ones



- At the beginning, the body of the cycle is executed. Therefore it will be executed at least ones
- The stop condition is analyzed and if it is false, the body of the loop is executed again
- When the stop condition becomes true, the next processing of the algorithm is carried out
- If the stop condition never becomes true then the loop is infinite

Repeat-until instruction. Example

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$

Pseudocode	Python

```
Read n
s=0
i=0
Repeat
  i=i+1
  s=s+i
Until i>n
Write "s="+s
```
	```
	n=input("n=")
	s=0
	i=0
	i=i+1
	s=s+i
	While i<=n:
	  i=i+1
	  s=s+i
	print("s=”, s)
	```

Execute repeat until body ones

Remark:
In some programming languages repeat-until is replaced with do-while.

Summary

- Algorithms are step-by-step procedures for solving problems
- It must have the following properties
  - Fairness and generality
  - Finitude
  - Rigor (unambiguity)
  - Efficiency
- Data processed by an algorithm can be
  - simple
  - structured (ex: tables)
- Algorithms can also be described in pseudocode or directly in a programming language