Algorithms for Data Science

Lecture #1: Introduction

Mateusz Buczyński

University of Warsaw

18.02.2026

Thanks to dr Krzysztof Fleszar for the base material.

Algorithms for Data Science

Goal

"Data Science is not just about libraries. It is about feeding the processor efficiently."

- From silicon $\rightarrow$ Python: why hardware matters for algorithms
- Efficiency as a first-class design constraint

Topics

- **Lectures 1–3:** Introduction
- **Lectures 4–6:** Algorithms
- **Lectures 7–8:** Data Structures
- **Lecture 9:** Extra

About the instructor

Who I am

- Mateusz Buczyński
- Focus: practical usage of data science in business; time series predictions

Contact

- Email: mp.buczynski2@uw.edu.pl
- Office hours (online): on request (MS Teams; link on Moodle)
- Best way to reach me: email with subject prefix `[AFDS]`

Logistics & Organization

The schedule

- Moodle link
- Lecture - Wednesdays 15:00, recording available, 15 lectures in total.
- Labs - Wednesdays 16:45 and 18:30, recording available.
- Labs - every other week, 7 labs in total.

Exercise schedule

#	Week type	Date
~~1~~	~~O~~	~~18.02.2026~~
2	E	25.02.2026
3	O	04.03.2026
4	E	11.03.2026
5	O	18.03.2026
6	E	25.03.2026
7	O	01.04.2026
~~7'~~	~~E~~	~~08.04.2026~~
8	O	15.04.2026
9	E	22.04.2026
10	O	29.04.2026
11	E	06.05.2026
12	O	13.05.2026
13	E	20.05.2026
14	O	27.05.2026
15	E	03.06.2026

Passing criteria

To pass the labs

- Pass **6 programming assignments** (**PT**) - 10 points each.
- Pass the **final presentation** (**FP**) - 60 points.

To pass the whole course

- Pass the labs, and
- Pass the **exam** (**EX**) - 180 points.

Points split in final grade

- 20% PT
- 20% FP
- 60% EX

You can obtain $+1$ point to your final grade for activity during the lecture and the labs.

How to learn (resources)

Books

- The "Bible": *Introduction to Algorithms* (Cormen, Leiserson, Rivest, Stein)
- Visual/intuitive: *Grokking Algorithms* (Aditya Bhargava) — great for beginners

Practice (mental sport)

- Competitive programming:
  - HackerRank (learn syntax)
  - Codeforces (algorithms)
- Tip: treat coding like a sport — build muscle memory

The puzzle: "Guess my number"

The puzzle: "Guess my number"

Scenario

I have a secret number x between 0 and 1000.

The puzzle: "Guess my number"

Scenario

I have a secret number x between 0 and 1000.

Constraint

You can only ask questions of the form: "Is $x \leq \ldots$?"

The puzzle: "Guess my number"

Scenario

I have a secret number x between 0 and 1000.

Constraint

You can only ask questions of the form: "Is $x \leq \ldots$?"

Strategy

- Don't guess randomly
- Ask "Is $x \leq 500$?" $\rightarrow$ eliminate half instantly
- Repeat

The puzzle: "Guess my number"

Scenario

I have a secret number x between 0 and 1000.

Constraint

You can only ask questions of the form: "Is $x \leq \ldots$?"

Strategy

- Don't guess randomly
- Ask "Is $x \leq 500$?" $\rightarrow$ eliminate half instantly
- Repeat

Result: about 10 questions since $2^{10} = 1024$.

Translating logic to code

Variables

l (left bound), r (right bound), m (middle)

Loop

While the range is valid ($l < r$):

- compute m
- if $x \leq m$: answer is in left half $\Rightarrow r = m$
- else: answer is in right half $\Rightarrow l = m + 1$

Crucial logic - loop invariant

Why $l = m + 1$? Otherwise the range might not shrink $\rightarrow$ infinite loop.

Formal definition of an algorithm

Definition

An algorithm is a finite, unambiguous sequence of steps that:

- takes an input from a specified set of valid instances,
- produces the correct output for each valid instance,
- terminates after a finite number of steps.

In this lecture

Binary search is an algorithm for the ordered-search problem.

Application: insertion point

Problem

Given a sorted list a, find the first index i such that $a[i] \geq v$.

Boundary handling

- Use $l = 0$ and $r = \text{len}(a)$
- Handles "all elements $< v$" correctly (answer should be n)

Python implementation

```
def binary_search(a, v):
    l, r = 0, len(a)
    while l < r:
        m = (l + r) // 2
        if a[m] >= v:
            r = m
        else:
            l = m + 1
    return l
```

Note: in Python, use // for integer division; / produces floats.

How do we know it works? (Invariants)

The problem

Bugs happen at the edges: infinite loops, off-by-one errors.

The tool: loop invariants

A *loop invariant* is a statement that is true:

- before the loop starts,
- after every iteration,
- when the loop ends.

Proving binary search

Invariant

$l \leq \text{result} \leq r$ (the answer is trapped in the current window).

- Initialization: true at start (answer in $0 \ldots n$)
- Maintenance: each step shrinks the window ($r - l$ decreases), answer stays inside
- Termination: loop ends at $l = r$; since answer is trapped, l is the answer

The “lie” of computer science

RAM model

We are taught that accessing any memory address takes the same time ($O(1)$).

Reality

This is false for high-performance data science.

New goal

Stop thinking about “steps” and start thinking about **data movement**.

The cost of latency (the hierarchy)

- Registers (brain): < 1 ns
- L1/L2 cache (pocket): $\sim$ 1–10 ns
- RAM (library): $\sim$ 100 ns
- Disk (moon): $\sim$ 10,000,000 ns

Takeaway

The CPU often waits for data; performance is the art of minimizing this wait.

Why is `sum(list)` slow? (boxed integers)

C/NumPy integer

- 4 bytes (raw binary)

Python integer

- $\sim$ 28 bytes (`PyObject`)
- contains: refcount + type + size + value

The cost

Add $\rightarrow$ unwrap two boxes, check types, add, then wrap the result in a new box.

Visual intuition: a warehouse of boxes vs. a stream of raw data.

Memory layout: lists vs. arrays

Python list

- A list of pointers
- Data scattered in RAM (pointer chasing)
- Cache misses $\rightarrow$ CPU waits ~ 100 ns per item

NumPy array

- Contiguous block of memory
- Data lined up $\rightarrow$ cache hits
- CPU can predict the next number

Key concept

Locality of reference.

Vectorization & SIMD

SIMD

Single Instruction, Multiple Data.

- Modern CPUs can add many pairs of numbers per cycle (e.g., 8)
- Python loop: adds 1 pair at a time
- NumPy: uses SIMD to process chunks efficiently

Summary

`import numpy` isn't magic;

Linear vs. binary search

Linear search

- Check one-by-one
- Complexity: N steps

Binary search

- Halve the problem
- Complexity: $\log_2 N$ steps

The "technological disadvantage" experiment

The race: processing $N = 1{,}000{,}000{,}000$ items

- Contestant A: supercomputer (10^9 ops/sec) using linear search
- Contestant B: 1980s BASIC machine (1000 ops/sec) using binary search

- A (linear): 10^9 steps $\rightarrow \approx 1$ second
- B (binary): ≈ 30 steps $\rightarrow 30/1000 = 0.03$ seconds

Conclusion

A smart algorithm on a "dinosaur" beats brute force on a supercomputer.

Summary

- Algorithmic thinking: divide and conquer (binary search)
- Correctness: use invariants to prove your logic
- Performance: respect the hardware (cache & locality)
- Next week: time complexity and Big-O notation