THE FINISH OFF SCHOOL

Complexity

Enseignant : Diviyan Kalainathan

Date : 01/06/2023

Summary

01 — Algorithmic Complexity

02 — Code Complexity: Design Patterns

03 — Project

Small notes before starting

- Let’s all code in Python !
- We will be using GitHub for sharing projects and slides: https://github.com/finish-off-school/course-complexity
- Projects are to be submitted via pull requests to this repo ! (we can have a discussion on it afterwards)
- Content is taken from Max Lai @ PyCon APAC 2022 and Dimo Brockhoff @ CMAP & Centrale-Supelec

Any questions: diviyan@fentech.ai

What is this course ?

Better code quality !

- An algorithm, or script can be coded in many different ways !
- There isn't one single “best” implementation, but lots of bad practices can be seen
- The rules of “good code” :
  - A efficient implementation (that executes rather quickly)
  - A maintainable code that allows for easy debugging
  - A modular code that can be easily upgraded

Example:

```
    ## Format supply plan et input_sp
    id cols = [out sc.sp date,
out sc.sp source, sc.warehouse, sc.product]
    supply plan =
supply plan.set_index(id_cols)
    input sp =
input sp.set_index(id_cols)[out_sc.sp_qty].to_
dict() #type: ignore

    ## Define initial_qty and is_old column
    supply plan["initial qty"] =
supply plan.index.map(input_sp).fillna(0)
    supply plan["is_old"] =
supply_plan["initial_qty"] > 0

    supply plan = supply_plan.reset_index()
    supply plan =
supply plan[~supply plan["is old"]]
    supply plan.drop(columns=["initial_qty",
"is_old"], inplace=True)
```

```
   id_cols = [out_sc.sp_date,
out_sc.sp_source, sc.warehouse, sc.product]
    supply_plan = pd.concat(
        [initial_sp, supply_plan],
ignore_index=True # type: ignore
    ).drop_duplicates(subset=id_cols,
keep="first")
```

01 — Algorithmic Complexity

Why Algorithms & Complexity?

Algorithm

(noun.)

Word used by programmers when they do not want to explain what they did.

Why Algorithms & Complexity?

[…] an algorithm is a set of instructions, typically to solve a class of problems or perform a computation.
[from wikipedia]

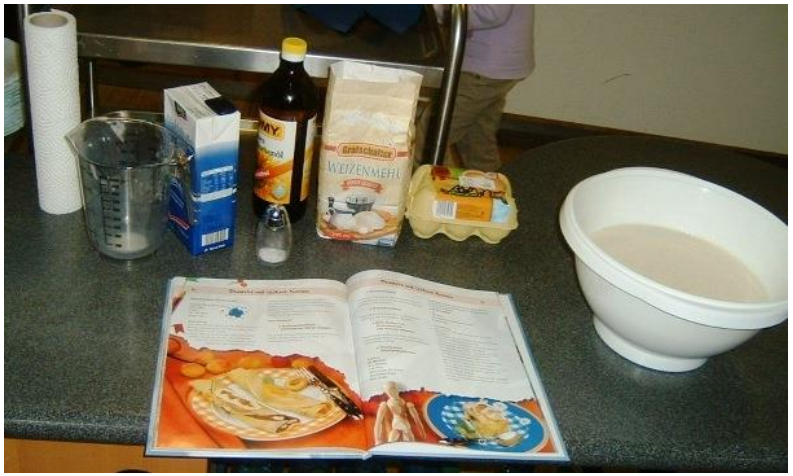
Algorithms widespread in almost every aspect of the “real-world”

- (automatic) problem solving
- sorting
- accessing data in data structures
- …

Mnemonic: Algorithm = Recipe

Recipe:

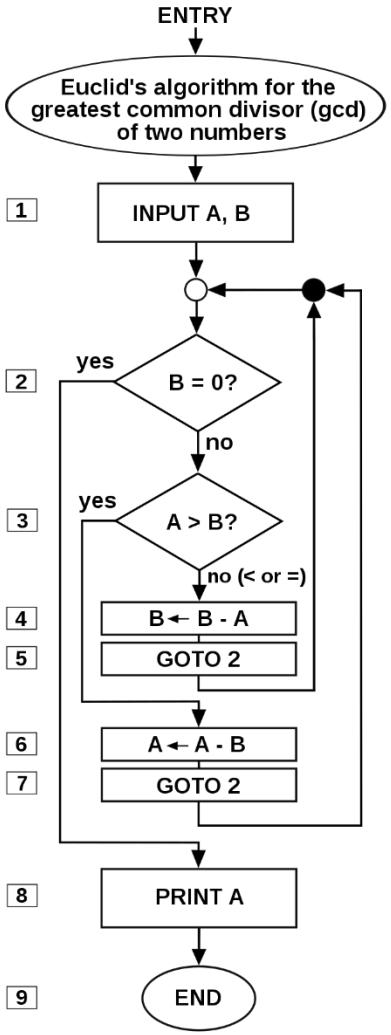
- Cook cooks a meal



Peng

Algorithm:

- A computer solves a problem



Somepics

Mnemonic: Algorithm = Recipe

Recipe:

- Cook cooks a meal
- Independent of cook, type of pan, type of stove/oven/…

Algorithm:

- A computer solves a problem
- Independent of programmer, computer, programming language, …
- Actually, a computer is running an *implementation* of an algorithm

Example: Sorting

Aim: Sort a set of cards/words/data

[Google, for example, has to sort all webpages according to the relevance of your search]

Re-formulation: minimize the “unsortedness”

E F C A D B

B A C F D E

A B C D E F

sortedness increases

Example: Sorting

Classical Questions:

- What is the underlying algorithm?

  (How do I solve a problem?)
- How long does it run to solve the problem?

  (How long does it take? Which guarantees can I give? How fast is the algorithm progressing?)
- Is there a better algorithm or did I find the optimal one?

related to the complexity part of the lecture

Basics II: The O-Notation

Excursion: The O-Notation

Motivation:

- we often want to characterize how quickly a function $f(x)$ grows asymptotically
- e.g. we might want to say that an algorithm takes quadratically many steps (in n) to find the optimum of a problem with n (binary) variables, it is never exactly n^2, but maybe $n^2 + 1$ or $(n+1)^2$

Big-O Notation

should be known, here mainly restating the definition:

Definition 1 We write $f(x) = O(g(x))$ iff there exists a constant $c > 0$ and an $x_0 > 0$ such that $|f(x)| \leq c \cdot g(x)$ holds for all $x > x_0$

we also view O(g(x)) as the set of all functions growing at most as quickly as g(x) and write $f(x) \in O(g(x))$

Big-O: Examples

- $f(x) + c = O(f(x))$ [as long as f(x) does not converge to zero]
- $c \cdot f(x) = O(f(x))$
- $f(x) \cdot g(x) = O(f(x) \cdot g(x))$
- $3n^4 + n^2 - 7 = O(n^4)$

Intuition of the Big-O:

- if $f(x) = O(g(x))$ then g(x) gives an upper bound (asymptotically) for f
- constants don't play a role
- with Big-O, you should have '≤' in mind

Excursion: The O-Notation

Further definitions to generalize from ‘≤’ to ‘≥‘ and ‘=‘:

- $f(x) = \Omega(g(x))$ if $g(x) = O(f(x))$
- $f(x) = \Theta(g(x))$ if $f(x) = O(g(x))$ and $g(x) = O(f(x))$

Note: Definitions equivalent to ‘<‘ and ‘>’ exist as well, but are not needed in this course

Exercise O-Notation

Please order the following functions in terms of their asymptotic behavior (from smallest to largest):

- $\exp(n^2)$
- $\log n$
- $\ln n / \ln \ln n$
- n
- $n \log n$
- $\exp(n)$
- $\ln(n!)$

Give for two of the relations a formal proof.

Exercise O-Notation (Solution)

Correct ordering:

$\frac{\ln(n)}{\ln(\ln(n))} = O(\log n)$ $\quad \log n = O(n)$ $\quad n = O(n \log n)$

$n \log n = \Theta(\ln(n!))$ $\quad \ln(n!) = O(e^n)$ $\quad e^n = O(e^{n^2})$

but for example $e^{n^2} \neq O(e^n)$

One exemplary proof:

$\frac{\ln(n)}{\ln(\ln(n))} = O(\log n)$:

$$\left|\frac{\ln(n)}{\ln(\ln(n))}\right| = \left|\frac{\log(n)}{\log(e)\ln(\ln(n))}\right| \leq \frac{3\log(n)}{\ln(\ln(n))} \leq 3\log(n)$$

for $n > 1$ for $n > 15$

Exercise O-Notation (Solution)

One more proof: ln n! = O(n log n)

- Stirling's approximation: $n! \sim \sqrt{2\pi n}\,(n/e)^n$ or even

$$\sqrt{2\pi}\, n^{n+1/2} e^{-n} \leq n! \leq e\, n^{n+1/2} e^{-n}$$

- $\ln n! \leq \ln(e n^{n+\frac{1}{2}} e^{-n}) = 1 + \left(n + \frac{1}{2}\right) \ln n - n$

$$\leq \left(n + \frac{1}{2}\right) \ln n \leq 2n \ln n = 2n \, \frac{\log n}{\log e} = c \cdot n \log n$$

okay for $c = 2/\log e$ and all $n \in \mathbb{N}$

- n ln n = O(ln n!) proven in a similar vein

If it’s not clear yet: Youtube

- https://www.youtube.com/watch?v=__vX2sjlpXU

basic data structures

Why Data Structures? What are those?

A data structure is a data organization, management, and storage format that enables efficient access and modification.

More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

from wikipedia

Why important to know?

- Only with knowledge of data structures can you program well
- Knowledge of them is important to design efficient algorithms

Data Structures and Algorithm Complexity

Depending on how data is stored, it is more or less efficient to

- Add data
- Remove data
- Search for data

Common Complexities

Complexity	Running Time	
constant	$O(1)$	independent of data size
logarithmic	$O(\log(n))$	often base 2, grows relatively slowly with data size
linear	$O(n)$	nearly same amount of steps than data points
	$O(n \log(n))$	Common, still efficient in practice if n not huge
quadratic	$O(n^2)$	Often not any more efficient with large data sets
…		
exponential	$O(2^n), O(n!), \ldots$	Should be avoided ☺

see also: https://introprogramming.info/english-intro-csharp-book/read-online/chapter-19-data-structures-and-algorithm-complexity

Best, Worst and Average Cases

Algorithm complexity can be given as best, worst or average cases:

Worst case:

- Assumes the worst possible scenario
- Algorithm can never perform worse
- Corresponds to an upper bound (on runtime, space requirements, …)
- Most common

Best case:

- Best possible scenario
- Algorithm is never quicker/better/more efficient/…

Average case:

- Complexity averaged over all possible scenarios
- Often difficult to analyze

Arrays

Array: a fixed chunk of memory of constant size that can contain a given number of n elements of a given type

- think of a vector or a table
- in python:
  - `import numpy as np`
  - `a = np.array([1, 2, 3])`
  - `a[1]` returns `2` [python counts from 0!]

Common operations and their complexity:

- Get(i) and Update(i) in constant time
- but Remove(i), Move j in between positions i and i+1, … are not possible in constant time, because necessary memory alterations not local
- To know whether a given item is in the array: linear time

Searching in Sorted Arrays

- Assume a sorted array $a[1] < a[2] < \ldots < a[n]$.
- How long will it take to find the smallest element $\geq k$?
  (Best case, worse case, average case)

Searching in Sorted Arrays

- Assume a sorted array $a[1] < a[2] < \ldots < a[n]$.
- How long will it take to find the smallest element $\geq k$?
  Or to decide whether a value a is in the array?
  (best case, worse case, average case)

Linear search

- go through array from $a[1]$ to $a[n]$ until entry found
- still $\Theta(n)$ in the worst case
- average case the same (if we assume that each item is queried with equal probability)

Searching in Sorted Arrays

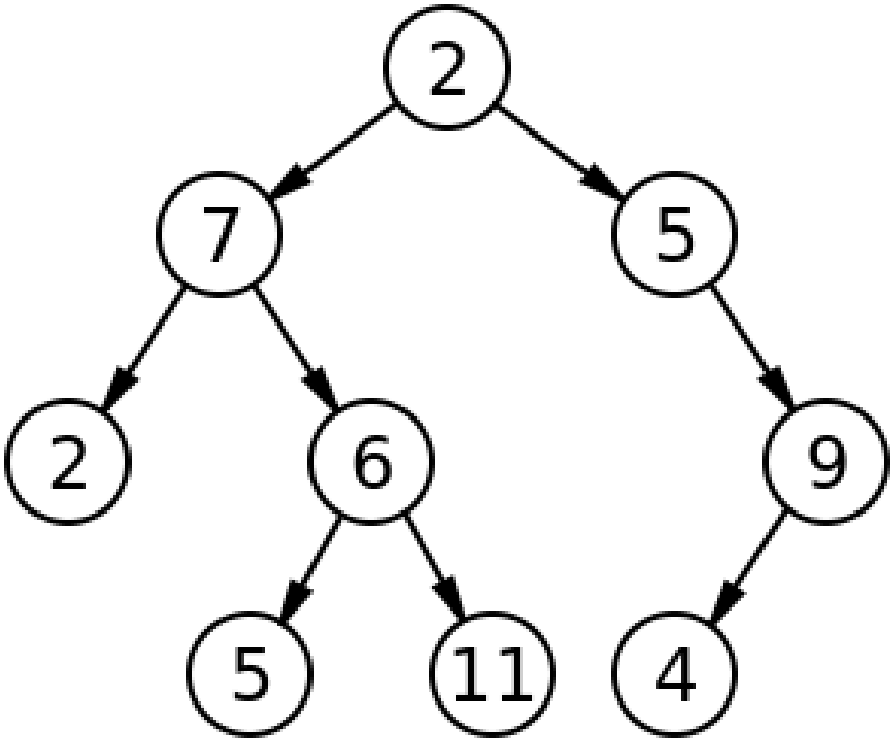
Binary search

- Look at position $\lceil n/2 \rceil$ first
- Is it the sought after entry? If yes, stop
- If not: search recursively in left or right interval, depending on whether the middle entry is larger or smaller than the sought after entry

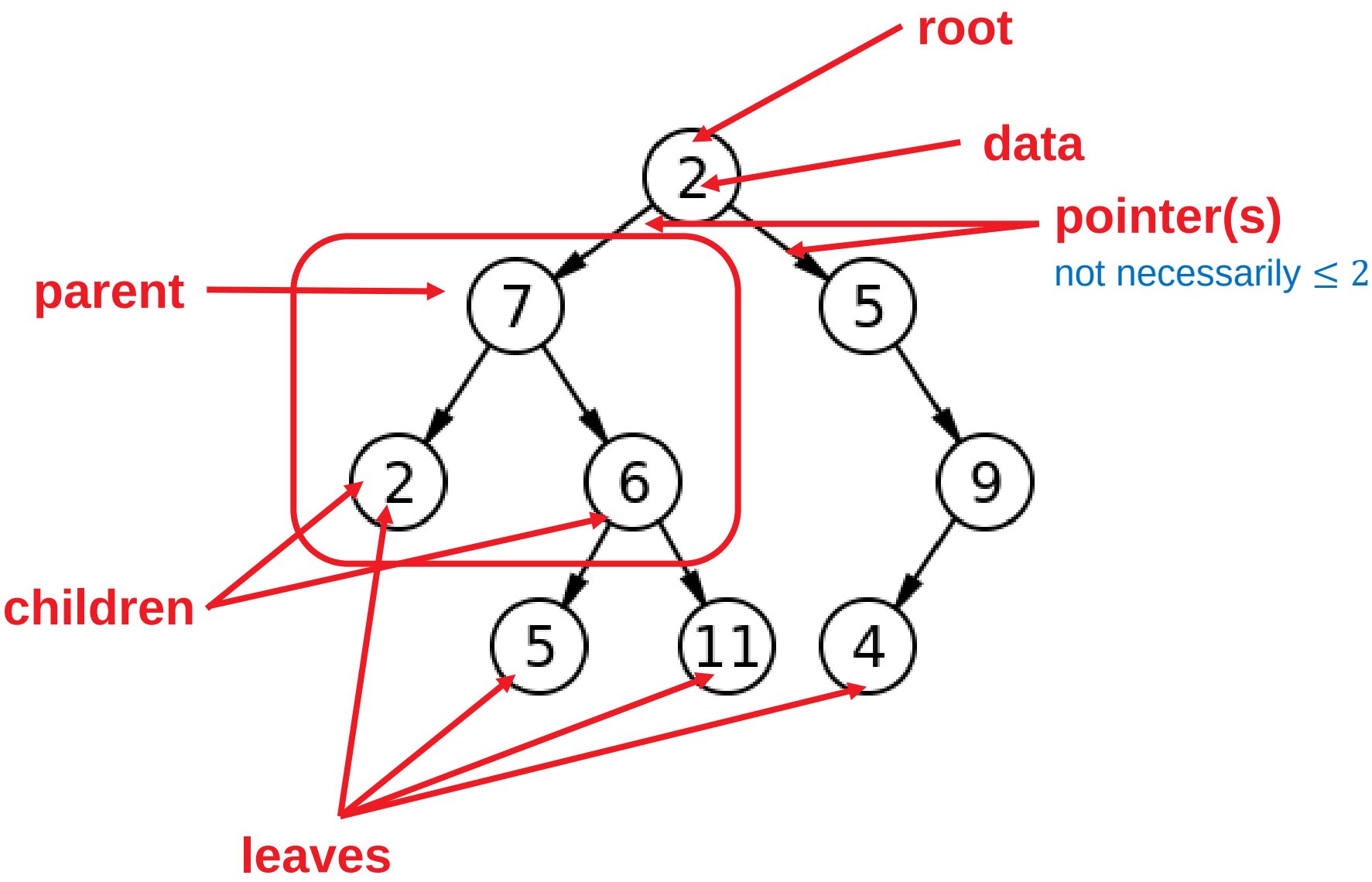
Runtimes

- Best case: 1
- Worst case:
  - sought after entry not in array
  - simple case: $n = 2^k - 1$ array elements
  - array-part where entry could be located is of length $2^{k-1} - 1$
  - by induction: maximally k comparisons needed
  - $k = \Theta(\log(n))$

Trees



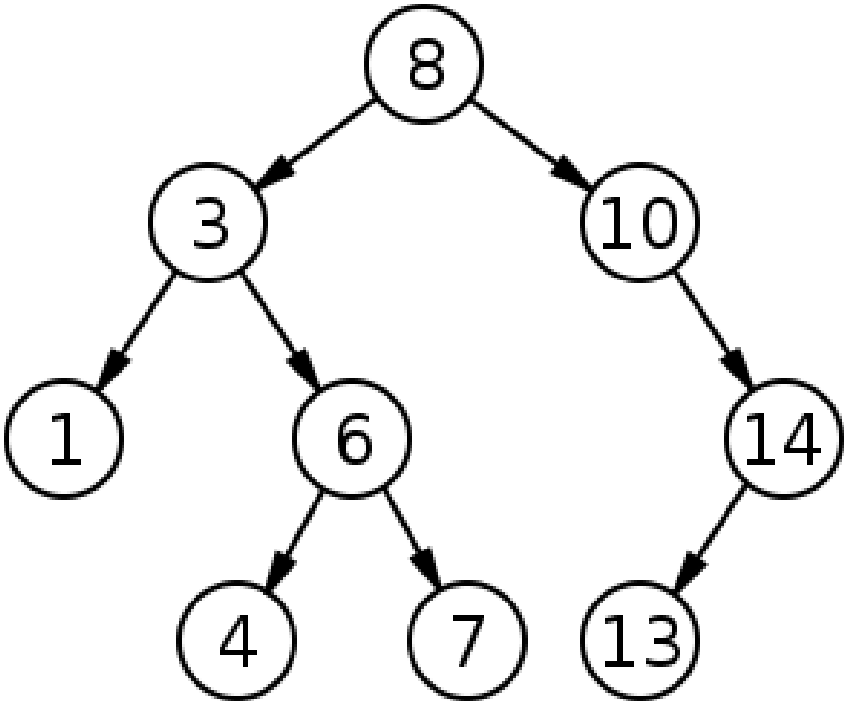
Trees



Back to Trees as Data Structure

Binary Search Tree

- a tree with degree ≤ 2
- children sorted such that the left subtree always contains values smaller than the corresponding root and the right subtree only values larger



Class Exercise: Filling a Binary Search Tree

Round 1:

Each online student: give an integer to be filled into tree

Round 2:

In class: tell where the next integer inserts

Binary Search Tree: Complexities

Search

- similar to binary search in array (go left or right until found)
- $\mathcal{O}(\log(n))$ if tree is well balanced
- $\Theta(n)$ in worst case (linear list)

Insertion

- first like search to determine the parent of the new node
- then add in $\mathcal{O}(1)$ [we are always at a leaf or have an “empty child”]

Remove (more tricky)

- if node has no child, remove it
- if node has a single child, replace node by its child
- if node has two children: find left-most tree entry L larger than the to-be-removed node, copy its value to the to-be-removed node, and remove L according to the two above rules
- cost: $\mathcal{O}(\text{tree depth})$, in worst case: $\Theta(n)$

Binary Trees: Can We Do Better?

Binary Search Tree

average case (random inserts) | worst case

search	insert	delete	Search	Insert	Delete
$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Guarantee a balanced tree:

- AVL trees
- B trees
- Red-Black trees
- …

average case (random inserts) | worst case

search	insert	delete	search	insert	delete
$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$

Can We Do Even Better on Average?

Balanced Trees

average case (random inserts) | | | worst case | | |

search	insert	delete	search	insert	delete
$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$

?

average case (random inserts) | | | worst case | | |

search	insert	delete	search	insert	delete
$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Dictionaries

In python:

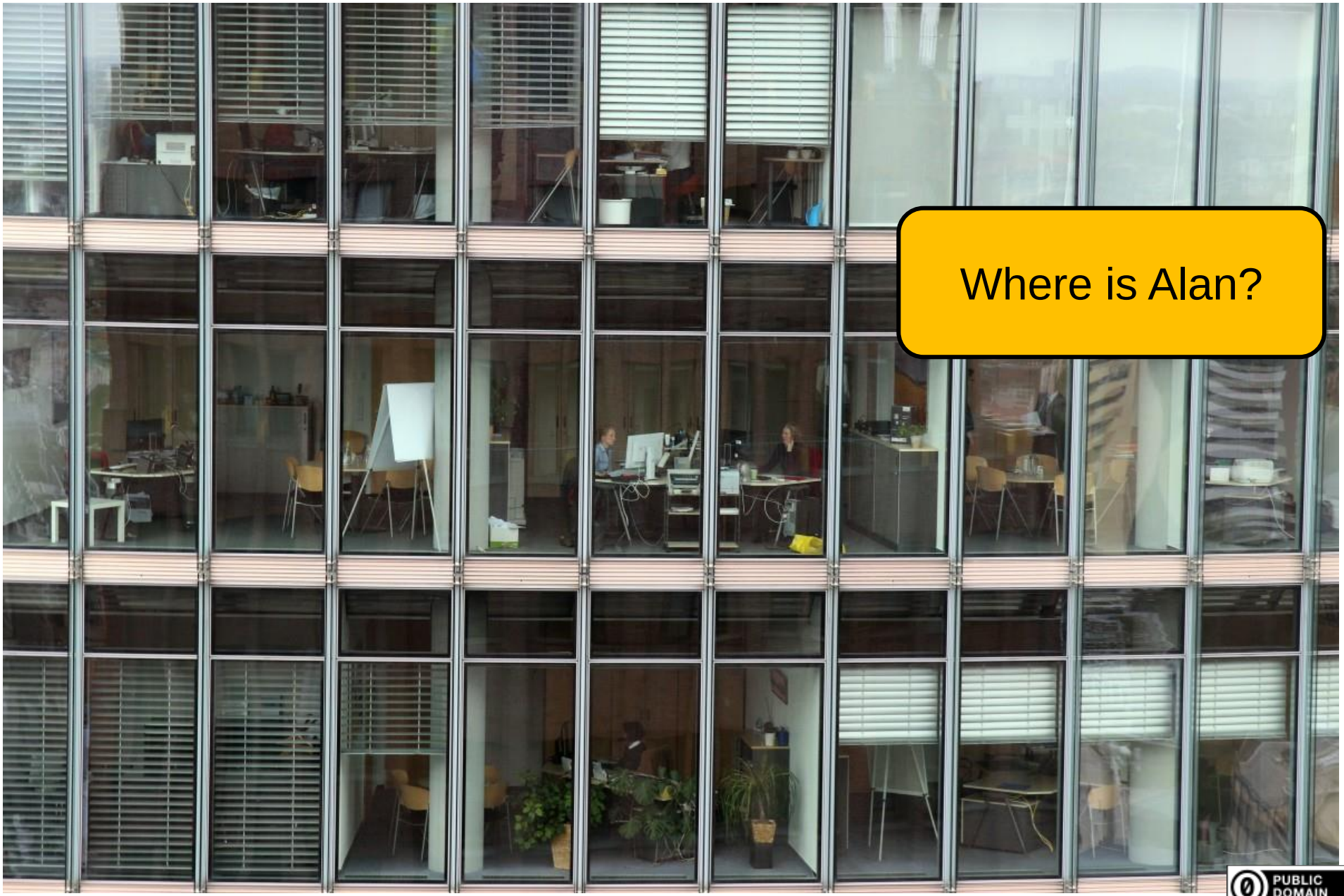
```
my_dict = {'Joe': 113, 'Pete': 7, 'Alan': '110'}
print("my_dict['Joe']: " + my_dict['Joe'])
```

gives `my_dict['Joe']: 113` as output

- the immutables `'Joe'`, `'Pete'`, and `'Alan'` are the keys
- **113**, **7**, and **110** are the values (or the stored data)

Next: Why dictionaries and how are they implemented?

Dictionaries

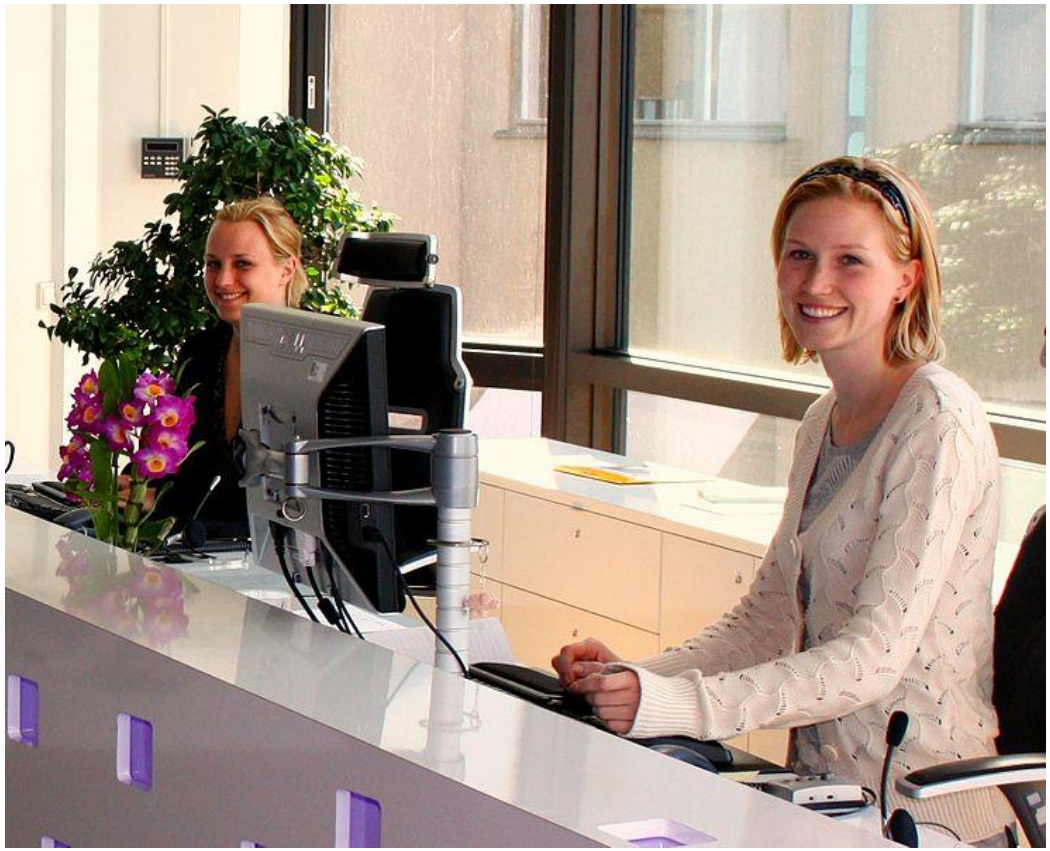


Where is Alan?

- Go through all offices one by one?

like in list and array

- No, you would ask the receptionist for the office number



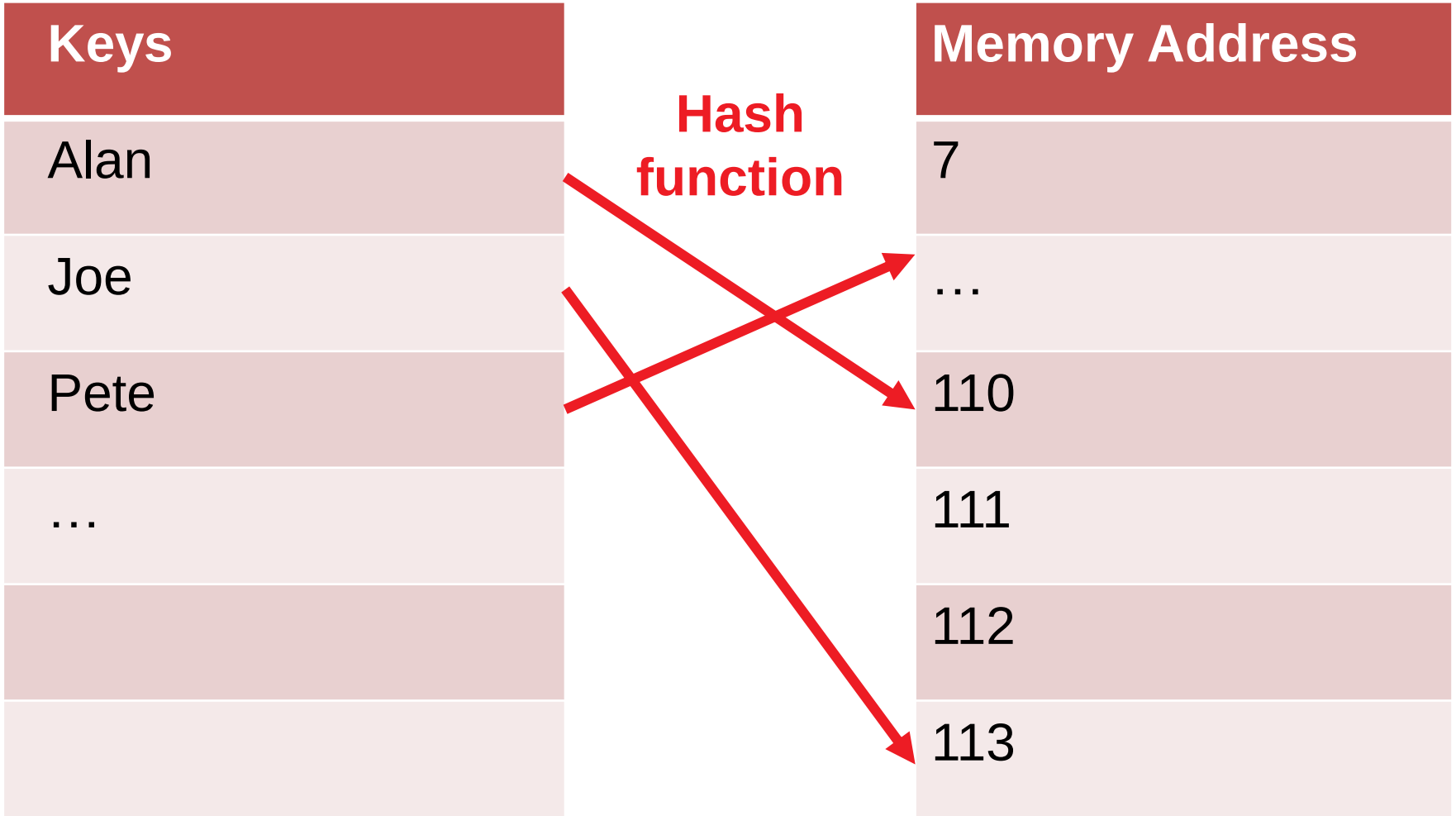
Evan Bench

Dictionaries Implemented as Hashtables

Names	Offices
Alan	7
Joe	…
Pete	110
…	111
	112
	113

Evan Bench

Dictionaries Implemented as Hashtables



Possible hash function: $h = z$ **mod** n

Hash Functions

…should be

- deterministic: find data again
- uniform: use allocated memory space well
  [more tricky with variable length keys such as strings]

Problems to address in practice:

- how to deal with collisions (e.g. via multiple hash functions)
- deleting needs to insert dummy keys when a collision appeared
- what if the hash table is full? → resizing

All this gives a constant average performance in practice
and a worst case of $\Theta(n)$ for insert/remove/search

Not more details here, but if you are interested:

For more details on python's dictionary:
https://www.youtube.com/watch?v=C4Kc8xzcA68

Quick Recap Data Structures

- Arrays: fast access, slow search, no insert
- (Linked) Lists: slow access, slow search, but insert/remove in constant time
  - Hence python lists are implemented as dynamic arrays (once array is full, a larger chunk of memory gets allocated) http://www.laurentluce.com/posts/python-list-implementation/
- Trees: log(n) access, log(n) add/remove
- Dictionaries: constant average performance in practice and a linear worst case for insert/remove/search

see also https://www.bigocheatsheet.com/

Exercise: Sorting

Aim: Sort a set of numbers

Questions:

- What is the underlying algorithm you used?
- How long did it take to sort?
  - What is a good measure?
- Is there a better algorithm or did you find the optimal one?

Overview of Today's Lecture

Sorting

- Insertion sort
- Insertion sort with binary search
- Mergesort
- Timsort idea

Exercise

- Comparison of sorting algorithms

Essential vs. Non-Essential Operations

In sorting, we distinguish

- comparison- and non-comparison-based sorting
- in the former, we distinguish further:
  - comparisons as essential operations
    - they are comparable over computer architectures, operating systems, implementations, (historic) time
    - they can take more time than other operations, e.g. when we compare trees w.r.t. their lexicographic DFS sorting
  - other non-essential operations: additions, multiplications, shifts/swaps in arrays, …

Insertion Sort

Idea:

for k from 1 to n-1:

- assume array a[1]…a[k] is already sorted
- insert a[k+1] correctly into a[1]…a[k+1]

swapping a[k+1] with all other numbers larger than a[k+1]

6 5 3 1 8 7 2 4

Swfung8

see also https://en.wikipedia.org/wiki/Insertion_sort

Insertion Sort: Analysis

Worst case:

- reverse ordering: insert always to the beginning
- then $1 + 2 + 3 + \cdots + (n-1) = \Theta(n^2)$ comparisons needed

Average Case:

- even here: $\Theta(n^2)$ comparisons needed (without proof)

Insertion Sort with Binary Search

Idea for an improved version:

use binary search for the right position of new entry in sorted subarray

- to insert array element $a[i]$, we need $\lceil \log(i-1) \rceil$ comparisons in worst case (= depth of the binary tree search)
- overall, therefore

$$\sum_{2 \le i \le n} \lceil \log(i-1) \rceil = \sum_{1 \le i \le n-1} \lceil \log(i) \rceil < \log(n!) + n$$

comparisons are needed

- from last time, we know that

$$\log(n!) \le \log\left(e n^{n+\frac{1}{2}} e^{-n}\right) = n\log(n) - n\log(e) + \frac{1}{2}(\log(n)) + \log(e)$$

in total, insertion sort with binary search needs

$$n\log(n) - 0.4426n + \mathcal{O}(\log(n))$$

comparisons in the worst case.

Mergesort

Another Possible Sorting Idea:

- sort first and second half of the array independently
- then merge the pre-sorted halves:
  - take the smaller of the smallest two values each time

$\text{Mergesort}(a_1, \dots, a_n)$

if $n = 1$ then stop

if $n > 1$ then:

- $(b_1, \dots, b_{\lceil n/2 \rceil}) = \text{Mergesort}(a_1, \dots, a_{\lceil n/2 \rceil})$
- $(c_1, \dots, c_{\lfloor n/2 \rfloor}) = \text{Mergesort}(a_{\lceil n/2 \rceil + 1}, \dots, a_n)$
- return $(d_1, \dots, d_n) = \text{Merge}(b_1, \dots, b_{\lceil n/2 \rceil}, c_1, \dots, c_{\lfloor n/2 \rfloor})$

Mergesort

Another Possible Sorting Idea:

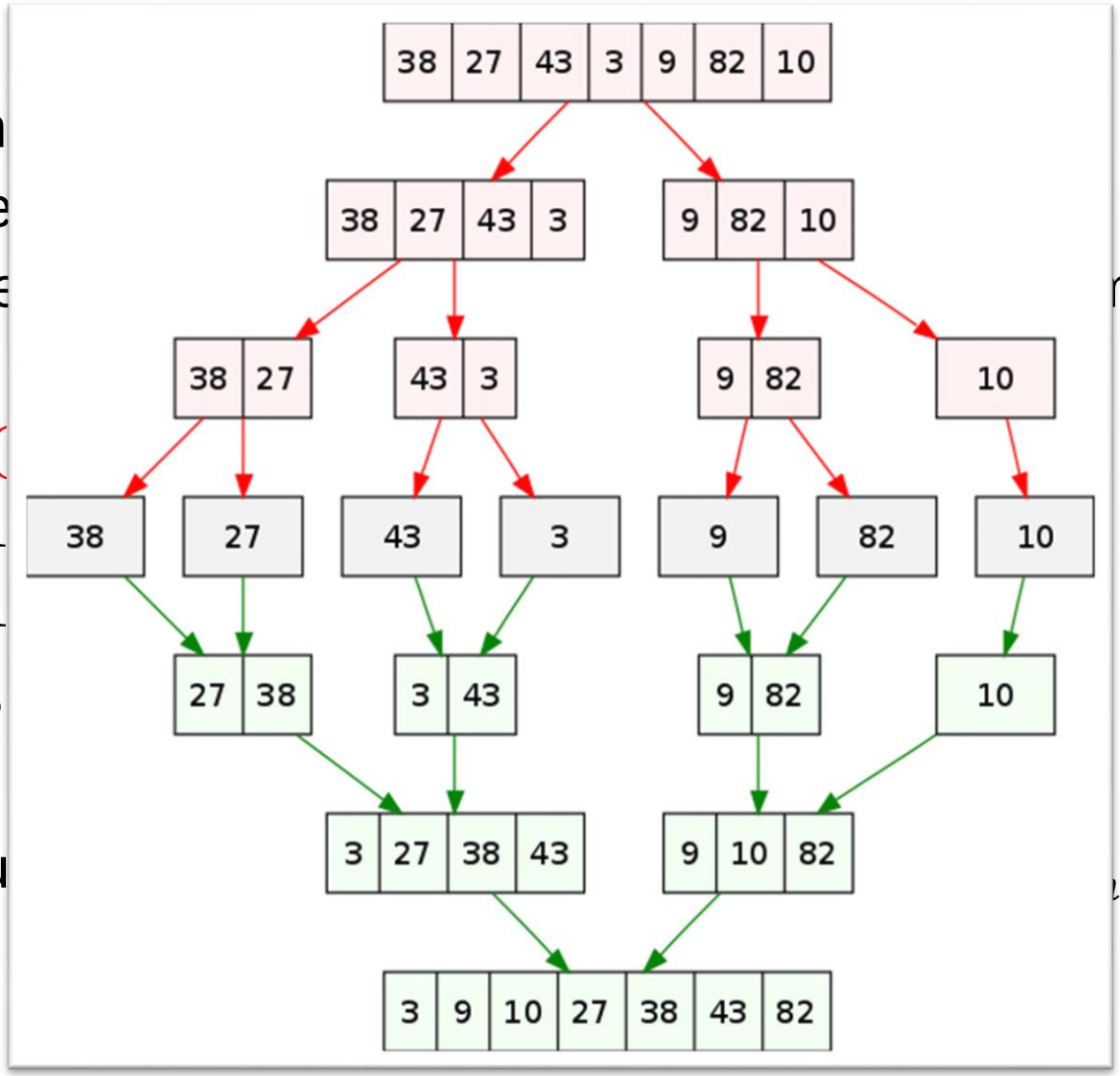
- sort first an
- then merge
  - take the ne

Mergesort(

if $n = 1$

if $n > 1$

- $(b_1,$
- $(c_1,$
- retu $\lfloor n/2 \rfloor)$



Mergesort: Runtime

- the number of essential comparisons C(n) when sorting n items with Mergesort is

$$C(1) = 0, \qquad C(n) = \underbrace{C\left(\left\lceil \frac{n}{2} \right\rceil\right)}_{\text{sorting left half}} + \underbrace{C\left(\left\lfloor \frac{n}{2} \right\rfloor\right)}_{\text{sorting right half}} + \underbrace{n-1}_{\text{merging}}$$

- without proof, $C(n) = n\log(n) + n - 1$ if $n = 2^k$

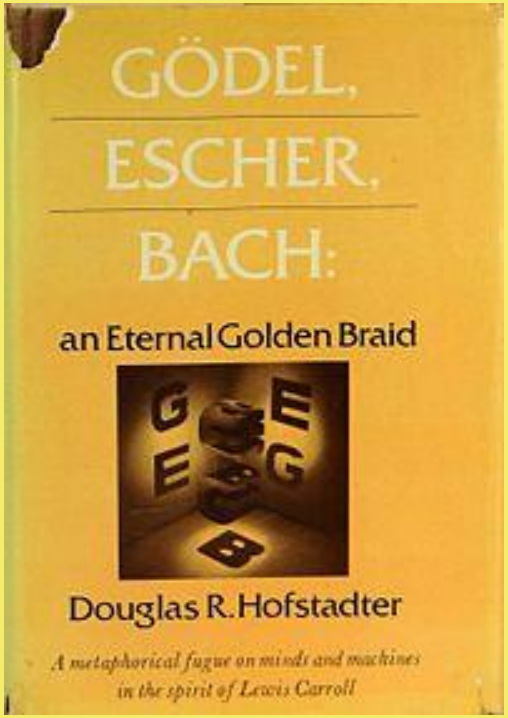
Remarks:

Mergesort is practical for huge data sets, that don't fit into memory

Mergesort is a recursive algorithm (= calls itself)

…solves a problem by solving smaller sub-problems first

Recommended Read



- “for leisure” – remark: it is quite hard to understand!
- https://en.wikipedia.org/wiki/G%C3%B6del,_Escher,_Bach

Python's Sorting: Timsort

- python uses a combination of Mergesort with insertion sort
  https://en.wikipedia.org/wiki/Timsort
- insertion sort for small arrays quicker than merging from n=1 (can be done in memory/cache)
- in addition, Timsort searches for subarrays which are already sorted (called "natural runs") and that are handled as blocks
- worst case runtime of $\mathcal{O}\big(n \log(n)\big)$, actually $\mathcal{O}\big(n \log(N)\big)$ with N being the number of natural runs
- best case: $\mathcal{O}(n)$

Lower Bound for Comparison-Based Sorting

- Insertion Sort, standard: $\Theta(n^2)$
- Insertion Sort with binary search: $n\log(n) - 0.4426n + \mathcal{O}(\log(n))$
- Mergesort: $n\log(n) + n - 1$ if $n = 2^k$

Can we do better than $n\log(n)$?

- No! [at least for comparison-based sorting]
- Lower bound for comparison-based sorting of $\Omega(n\log(n))$

without proof here

(Home-) Exercise in Python (question 3)

Comparing sorting algorithms in python

Goals:

- learn about Mergesort (and how to implement it)
- observe the differences in runtime between your own Mergesort and python's internal Timsort
- learn how to do a scientific (numerical) experiment and how to report the results

(Home-) Exercise in Python (question 3)

TODOs:

1. implement your own Mergesort e.g. based on lists
   http://www.cmap.polytechnique.fr/~dimo.brockhoff/algorithmsandcomplexity/2020/schedule.php
2. compare the differences in runtime between your own Mergesort and python's internal Timsort (`'sorted(…)'`) on randomly generated lists of integers
3. plot the times to sort 100 lists of equal length n with both algorithms for different values of $n \in \{10, 100, 1\,000, 10\,000\}$

Tip:

```
>>> import timeit
>>> timeit.timeit('your code', number=100)
```

Another (even more important) Tip:

use the "?" to get help on a module (and "??" to inspect the code)

Conclusions

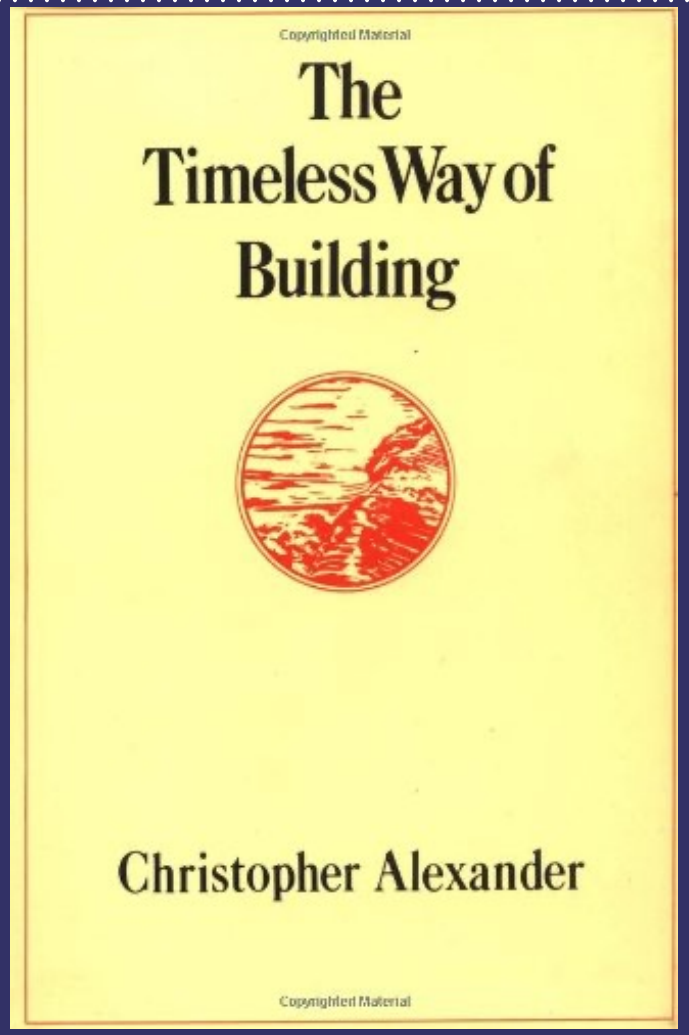
I hope it became clear...

...what sorting is about and how fast we can do it

02 — Code Complexity: Design Patterns

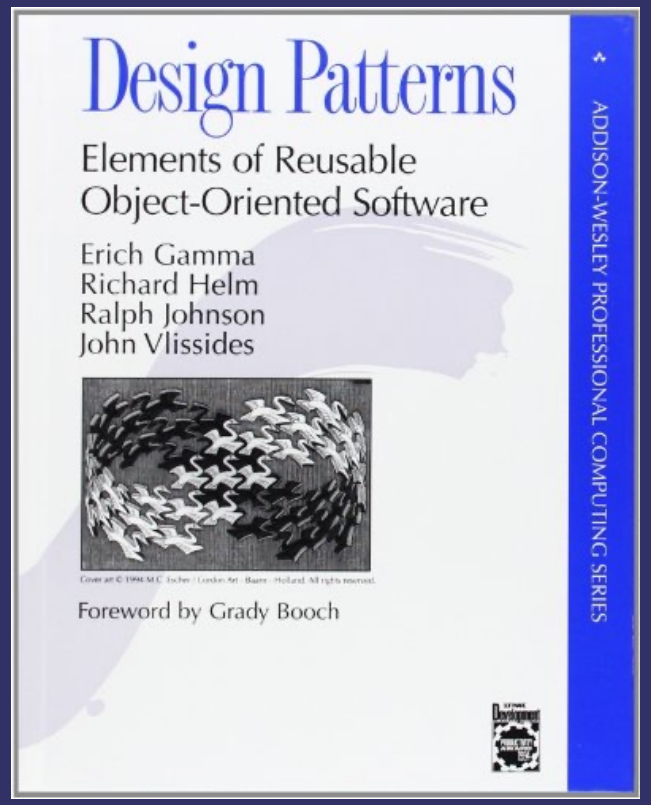
What Are Design Patterns

The Birth of Patterns



Each pattern is a three-part rule, which express a relation between a certain context, a problem, and a solution.

Software Design Pattern



A pattern is a proven solution

to a problem

in a context.

Source: https://wiki.c2.com/?PatternDefinitionThread

Why Design Patterns

- Design patterns help with reuse and communication
  - Reuse solutions : do not have to reinvent solutions for commonly recurring problems
  - Establish common terminology : provide a common point of reference during the analysis and design phase of a project
- Design patterns give a higher perspective on analysis and design
  - This frees you from the tyranny of dealing with the details too early

Source: Design Patterns Explained: A New Perspective on Object-Oriented Design, 2nd ed.

Design Pattern Descriptions

- Name
- Intent, problem and context
- Solution, implementation guidelines
- Participants and collaborators
- Positive and negative consequences

Pattern descriptions are often independent of programming language or implementation details

Design Patterns Catalog <<GOF>>

Creational patterns	Structural patterns	Behavioral patterns
• Factory Method • Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Interpreter • Iterator • Mediator • Memento • Observer • State • Strategy • Template Method • Visitor

Planning for Change

- The Open/Closed Principle
  - open for extension
  - closed for modification

Draw All Shape (POP)-1

```python
def draw_all_shapes(shapes: List):
    for shape in shapes:
        if isinstance(shape, Circle):
            draw_circle(shape)
        elif isinstance(shape, Rectangle):
            draw_rectangle(shape)
```

If we want to add **another shape class**?

```python
class Circle(object):
    def __init__(self, point, radius):
        self.center = point
        self.radius = radius

class Rectangle(object):
    def __init__(self, point, height, width):
        self.top_left = point
        self.height = height
        self.width = width

def draw_circle(circle):
    print('drawing circle:', circle)

def draw_rectangle(rectangle):
    print('drawing rectangle:', rectangle)
```

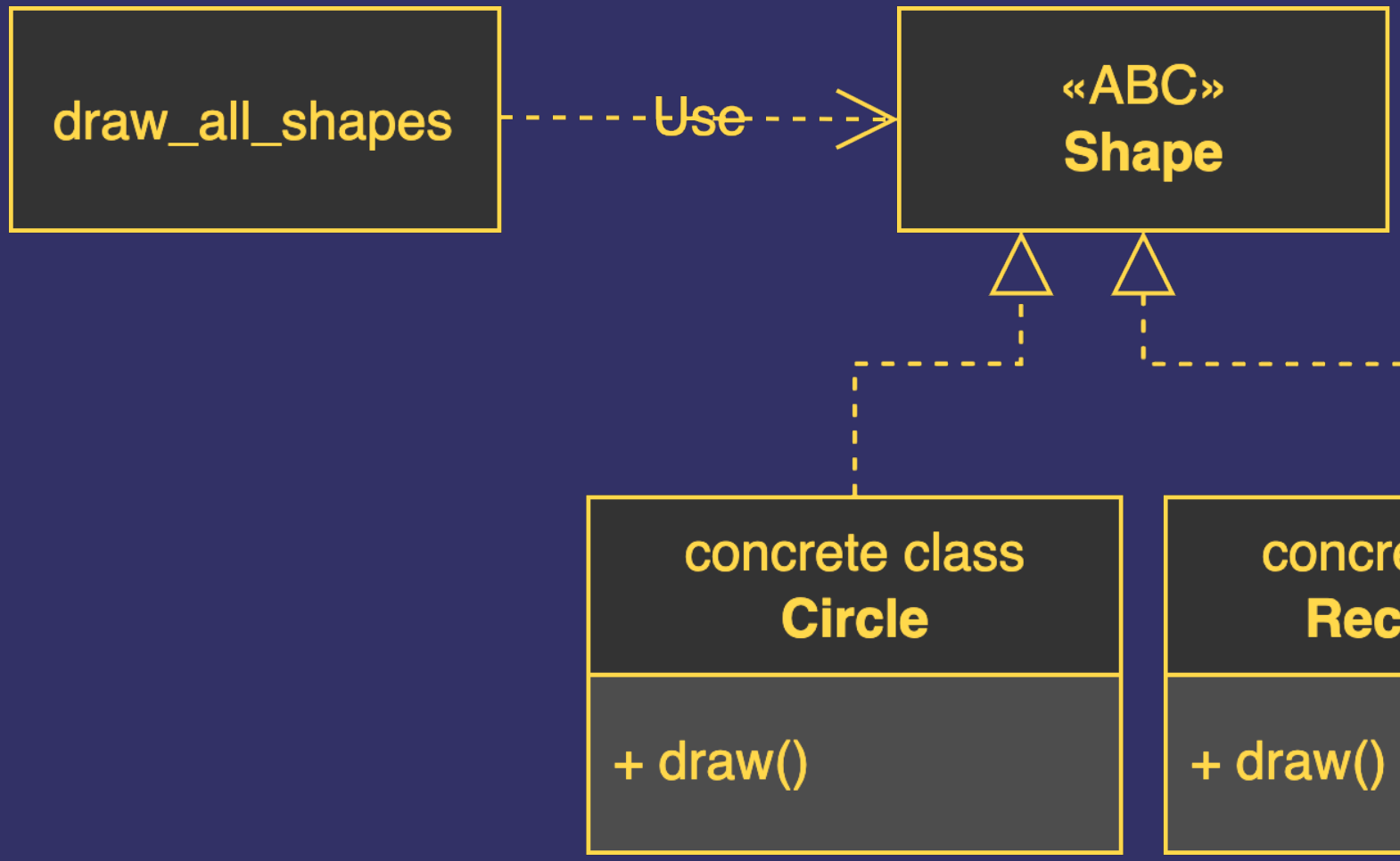
Draw All Shape (POP)-2

```python
def draw_all_shapes(shapes: List):
    for shape in shapes:
        if isinstance(shape, Circle):
            draw_circle(shape)
        elif isinstance(shape, Rectangle):
            draw_rectangle(shape)
        elif isinstance(shape, Square):
            draw_square(shape)
```

```python
class Square(object):
    def __init__(self, point, side):
        self.top_left = point
        self.side = side


def draw_square(square):
    print('drawing square:', square)
```

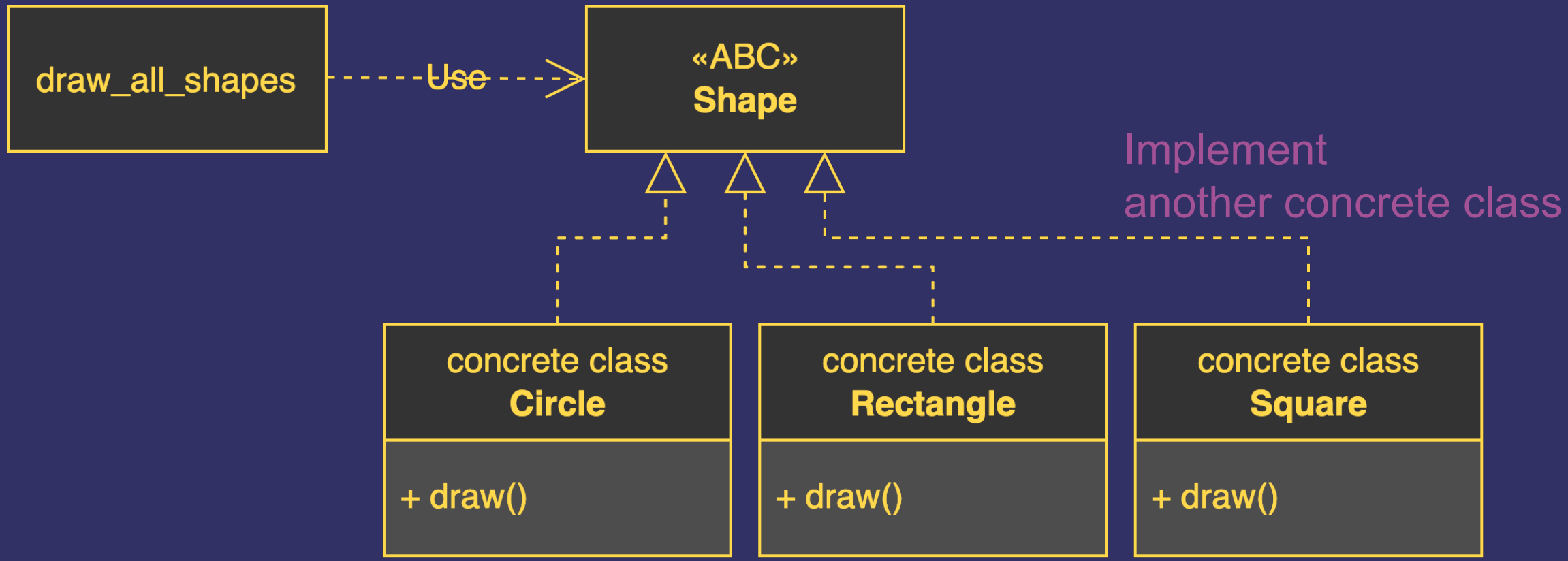
```python
class Circle(object):
    def __init__(self, point, radius):
        self.center = point
        self.radius = radius

class Rectangle(object):
    def __init__(self, point, height, width):
        self.top_left = point
        self.height = height
        self.width = width

def draw_circle(circle):
    print('drawing circle:', circle)
```

Draw All Shape (OOP)-1



Encapsulate
Interface – “draw()”
In an abstract in class

Draw All Shape (OOP)-3



Draw All Shape (OOP)-2

```python
def draw_all_shape(shapes: List[Shape]):
    for shape in shapes:
        shape.draw()

class Shape(ABC):
    @abstractmethod
    def draw(self):
        pass
```

```python
class Circle(Shape):
    def __init__(self, point, radius):
        self.center = point
        self.radius = radius

    def draw(self):
        print('drawing circle:', self)


class Rectangle(Shape):
    def __init__(self, point, height, width):
        self.top_left = point
        self.height = height
        self.width = width

    def draw(self):
        print('drawing rectangle:', self)
```

Draw All Shape (OOP)-2

```python
def draw_all_shape(shapes: List[Shape]):
    for shape in shapes:
        shape.draw()
```

```python
class Shape(ABC):
    @abstractmethod
    def draw(self):
        pass
```

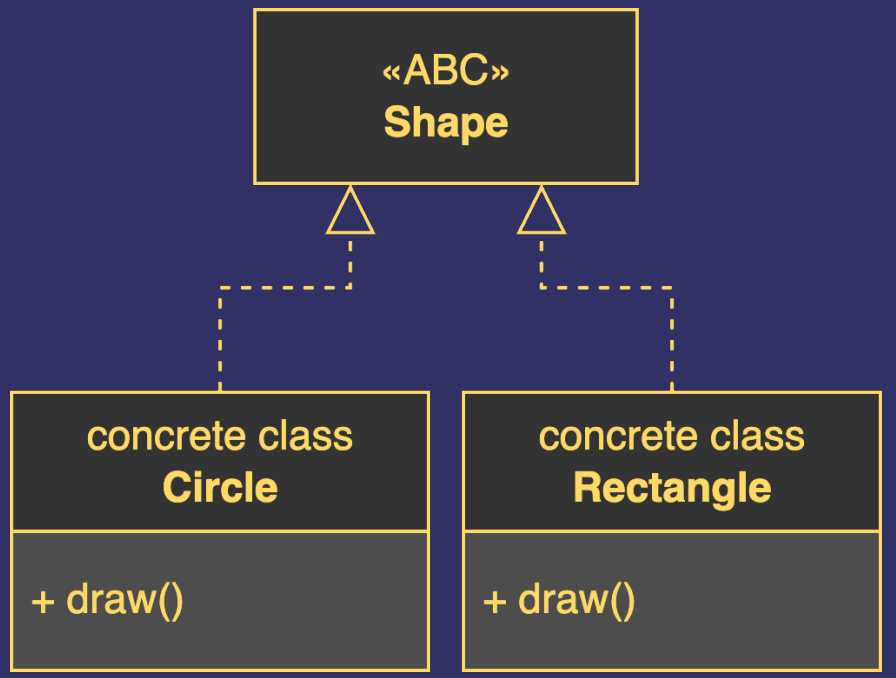
You don’t need to modify existing codes.

```python
class Square(Shape):
    def __init__(self, point, side):
        self.top_left = point
        self.side = side

    def draw(self):
        print('drawing square:', self)
```

```python
class Circle(Shape):
    def __init__(self, point, radius):
        self.center = point
        self.radius = radius

    def draw(self):
        print('drawing circle:', self)


class Rectangle(Shape):
    def __init__(self, point, height, width):
        self.top_left = point
        self.height = height
        self.width = width

    def draw(self):
        print('drawing rectangle:', self)
```

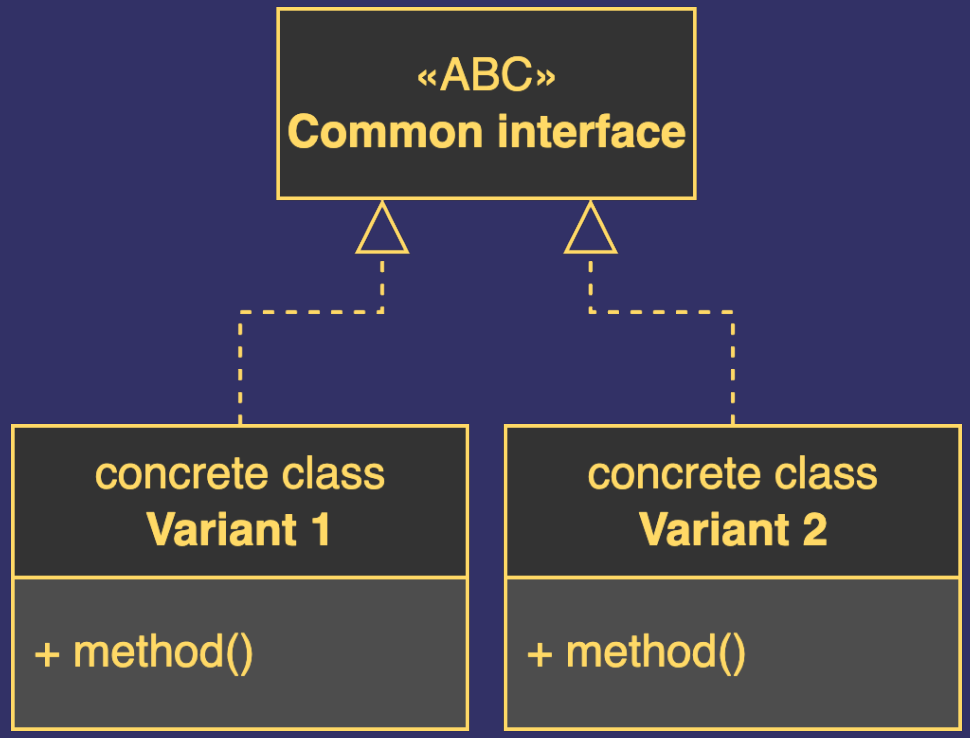
Principles for Creating Good OO Design

- Program to an interface, not an implementation
- Favor object composition over class inheritance



Thinking in Design Patterns

- Context first (design from context)
- Building by adding distinctions
- One pattern at a time
- Encapsulate variations in classes



Source: Design Patterns Explained: A New Perspective on Object-Oriented Design, 2nd ed.

Design Patterns in Python

Design Pattern in Dynamic Languages

- Peter Norvig states that 16 out of the 23 patterns in the original *Design Patterns* book become either “invisible or simpler” in a dynamic language (slide 9)
- “Design Patterns in Dynamic Languages” (1996)

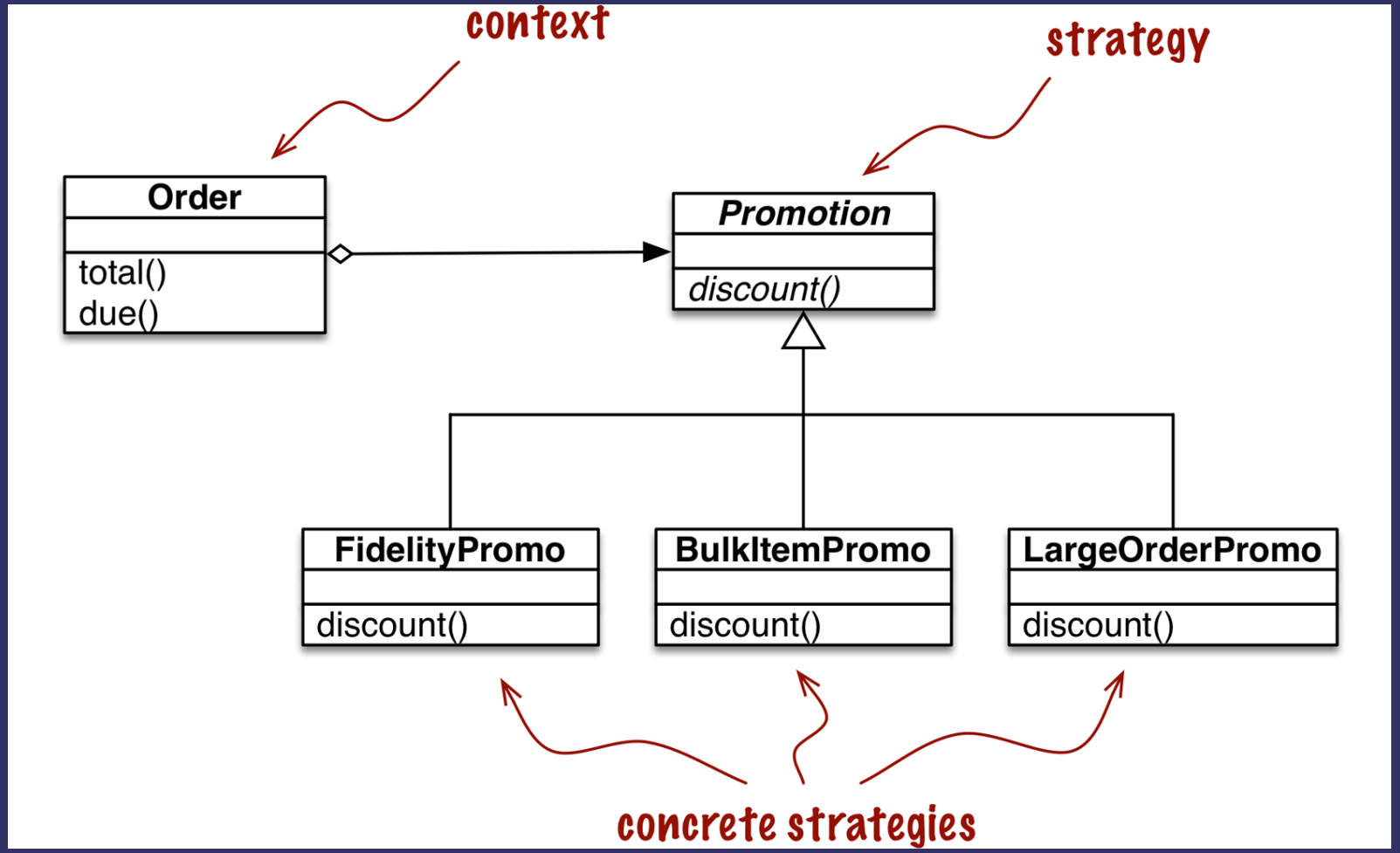
Source: http://norvig.com/design-patterns/design-patterns.pdf

First Class Object

- Functions in Python are first-class objects.
- A “first-class object” as a program entity that can be:
  - created at runtime;
  - assigned to a variable or element in a data structure;
  - passed as an argument to a function;
  - returned as the result of a function.

Source: Fluent Python, 2nd edition

Strategy pattern



Source: Fluent Python, 2nd ed., Ch.10

Function-Oriented Strategy

```python
def fidelity_promotion(order): """5% discount for customers with 1000 or more fidelity points"""
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0

def bulk_item_promotion(order): """10% discount for each LineItem with 20 or more units"""
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount

def large_order_promotion(order): """7% discount for orders with 10 or more distinct items"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0
```

Function-Oriented Strategy

function as an element in List “promos”

```
promos = [fidelity_promotion, bulk_item_promotion, large_order_promotion]

def best_promotion(order):
    return max(promo(order) for promo in promos)
```

Modules are “Singletons” in Python

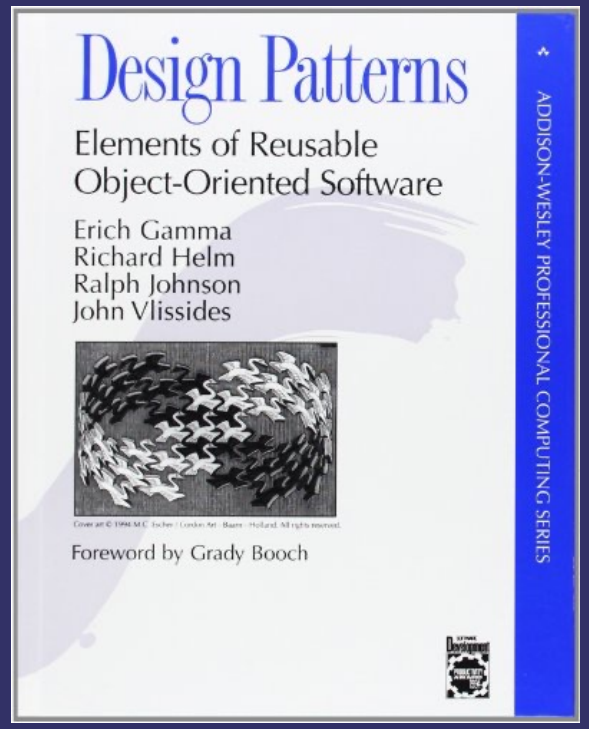
- import only creates a single copy of each module
- subsequent imports of the same name keep returning the same module object
- Official Python FAQ[1]:
  using a module is also the basis for implementing the Singleton design pattern, for the same reason.

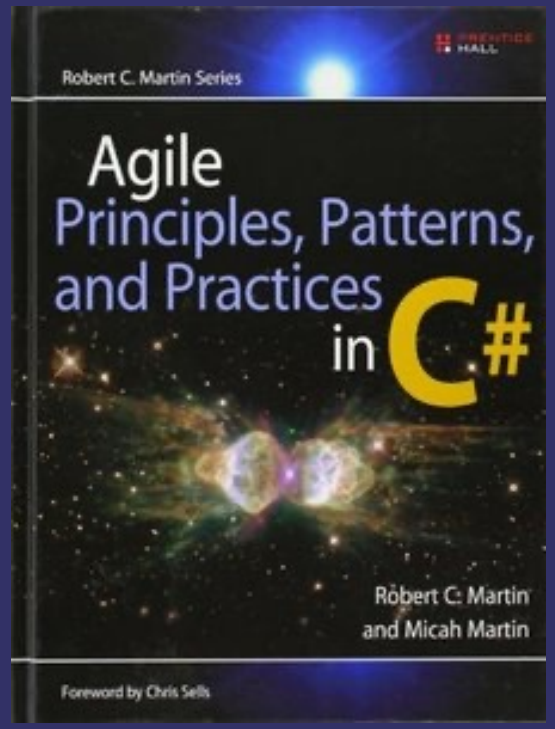
[1] https://docs.python.org/3/faq/programming.html#how-do-i-share-global-variables-across-modules

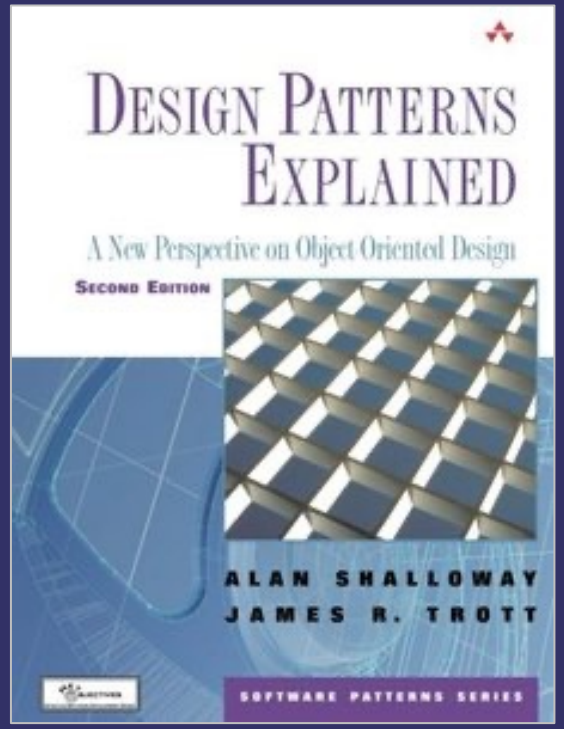
Key Takeaways

What	• Have a context • Solve a problem • Recur • Have a name
Why	• Reuse solutions • Establish common terminology • Give a higher perspective on design
How	• Context first • One pattern at a time • Encapsulate variations

References


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


Robert C. Martin Series
Agile Principles, Patterns, and Practices in C#
Robert C. Martin and Micah Martin
Foreword by Chris Sells


Design Patterns Explained
A New Perspective on Object Oriented Design
Second Edition
ALAN SHALLOWAY
JAMES R. TROTT
SOFTWARE PATTERNS SERIES


O'REILLY
Fluent Python
CLEAR, CONCISE, AND EFFECTIVE PROGRAMMING
Luciano Ramalho

03 - Project

Project: subject

Game of war (card game)

Bataille

- Create a game system of 52 Cards
- Rules at : https://fr.wikipedia.org/wiki/Bataille_(jeu) - Bataille ouverte (players know their cards and play 1-by-1)
- Define the game rules in a Class `Game`
- 2 Players, Opponent is a random player (Plays a random card in his hand)
- Your player is a algorithm that plays the best card in his hand and in case of draw the worst as token
- main.py:
    - Create a new game: shuffle and distribute cards to the two players
    - Play until the game is finished, print the played cards at each turn
      (turn=each player plays 1 card and the winner gets all, in case of draw: one card is put as token and they play again to win all)
    - Print the winning player and his number of cards

Project information

- Projects have to be submitted by 15/06/2023 23:59
- Create a pull request on github containing all your code and the main.py to execute it
- Everything has to be coded in python
- You will be evaluated on the quality of code (and if it works !)
- We have another course together on wednesday next week, so don’t hesitate to look at the project by then to ask me questions, otherwise send a mail to diviyan@fentech.ai

Good luck !

THE FINISH OFF SCHOOL

Questions ?