Algorithms & Complexity

September 21, 2020

CentraleSupélec / ESSEC Business School

Dimo Brockhoff

Inria Saclay – Ile-de-France

Algorithms & Complexity

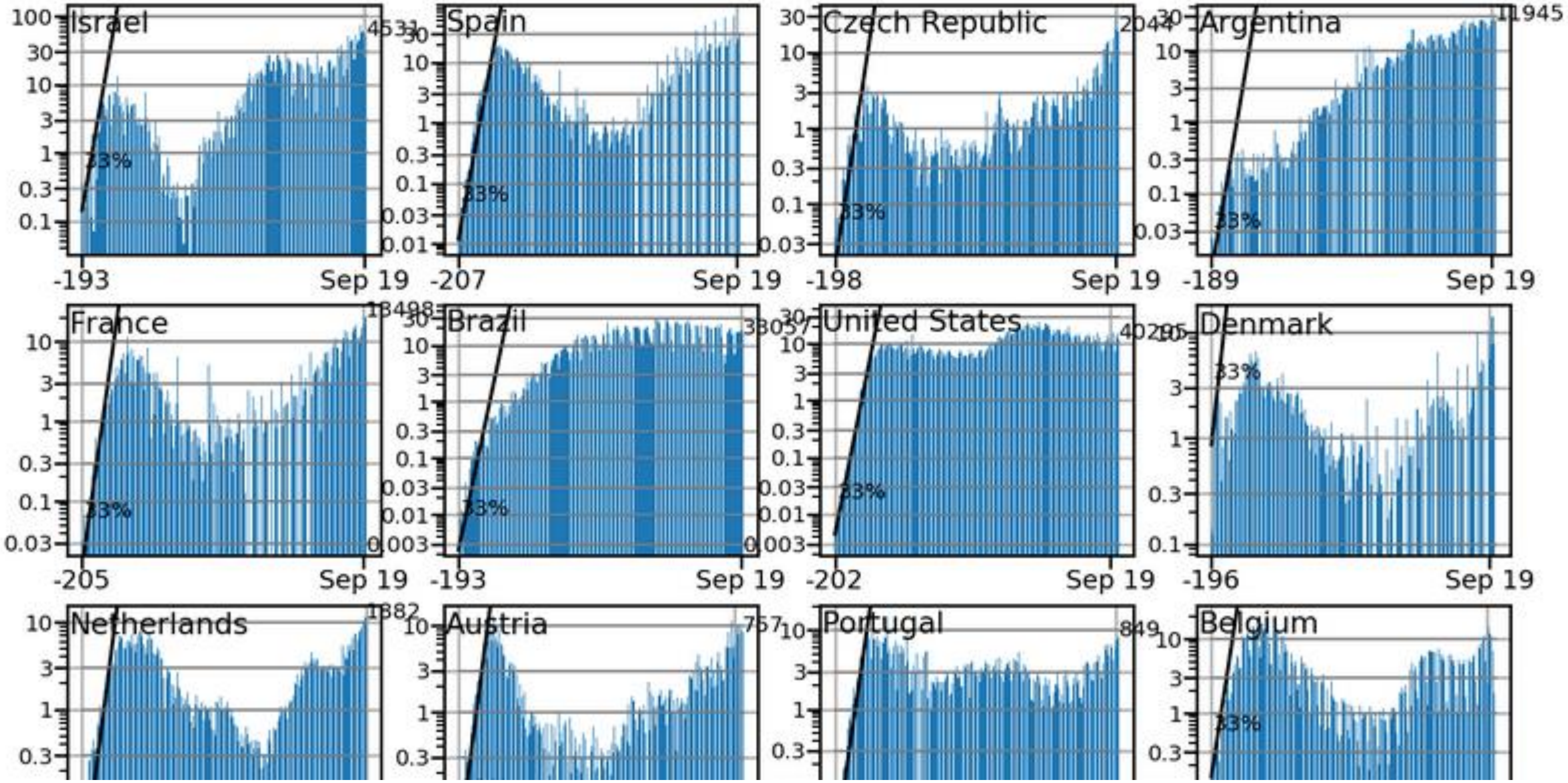
September 21, 2020

CentraleSupélec / ESSEC Business School

Dimo Brockhoff

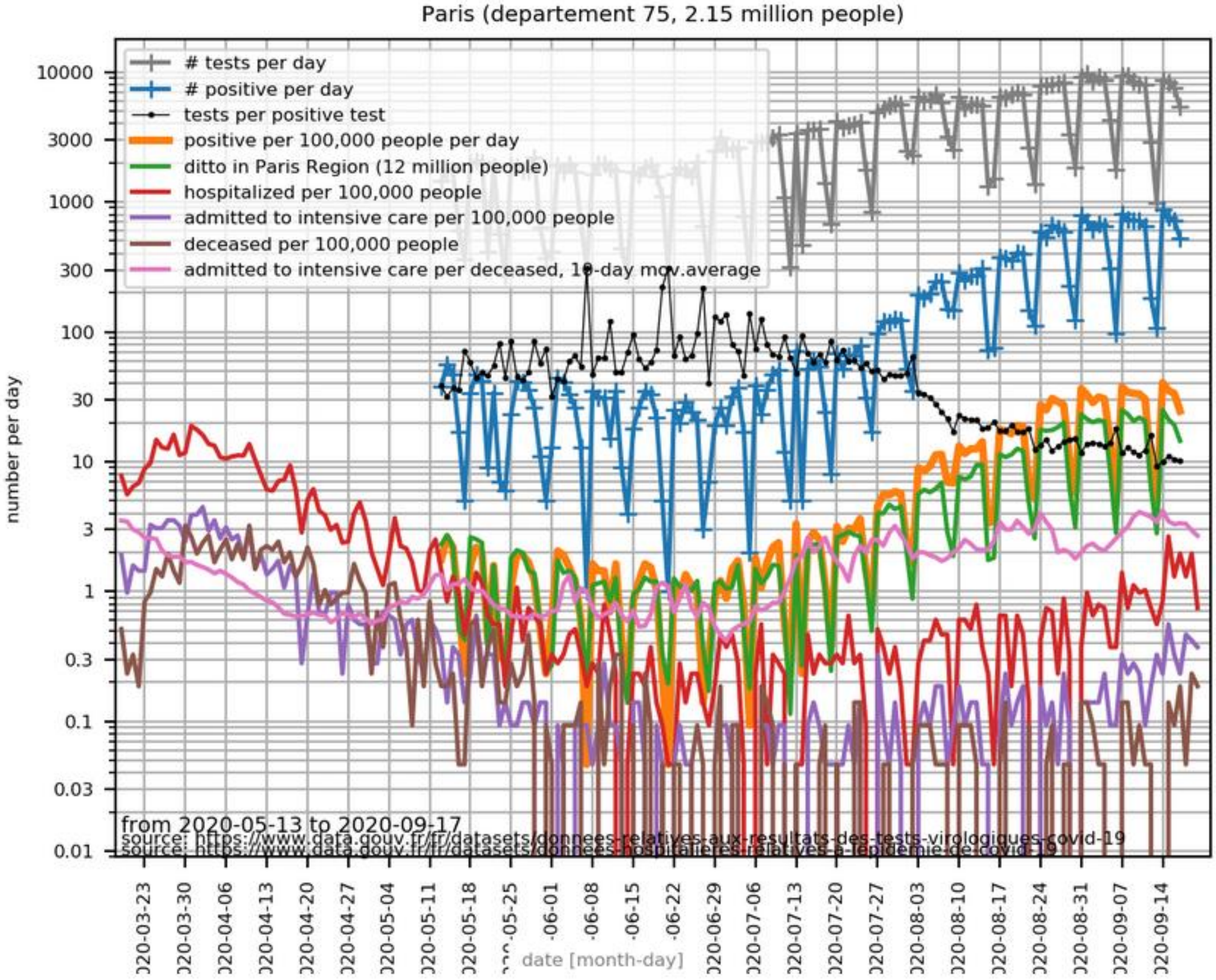
Inria Saclay – Ile-de-France

Weekly Covid-19 Update: It could be worse…



http://www.cmap.polytechnique.fr/~nikolaus.hansen/covid-19.html

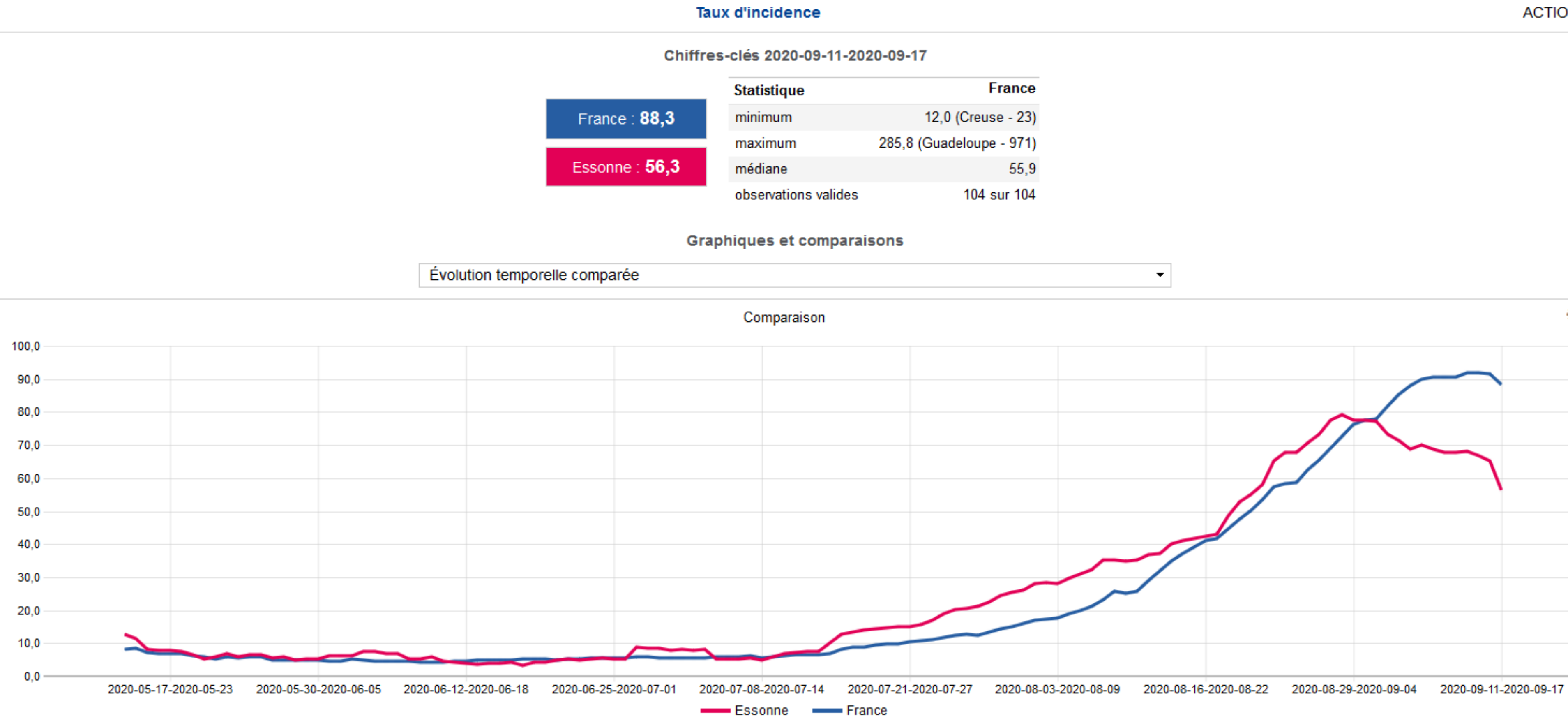
Weekly Covid-19 Update: It could be worse…



from 2020-05-13 to 2020-09-17
source: https://www.data.gouv.fr/fr/datasets/donnees-relatives-aux-resultats-des-tests-virologiques-covid-19
source: https://www.data.gouv.fr/fr/datasets/donnees-hospitalieres-relatives-a-lepidemie-de-covid-19

http://www.cmap.polytechnique.fr/~nikolaus.hansen/covid-19.html

Weekly Covid-19 Update: It could be worse…



Taux d'incidence

ACTIO

Chiffres-clés 2020-09-11-2020-09-17

France : **88,3**

Essonne : **56,3**

Statistique	France
minimum	12,0 (Creuse - 23)
maximum	285,8 (Guadeloupe - 971)
médiane	55,9
observations valides	104 sur 104

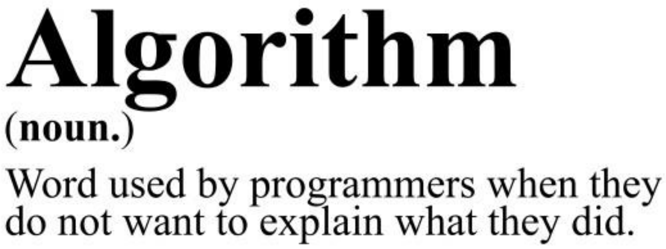
Graphiques et comparaisons

Évolution temporelle comparée ▾

Comparaison

https://geodes.santepubliquefrance.fr/#c=indicator&i=sp_ti_tp_7j.tx_pe_gliss&s=2020-09-11-2020-09-17&selcodgeo=91&t=a01&view=map2

Why Algorithms & Complexity?



Algorithm

(noun.)

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

Why Algorithms & Complexity?

~~**Algorithm**~~

~~**(noun.)**~~

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

[…] 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

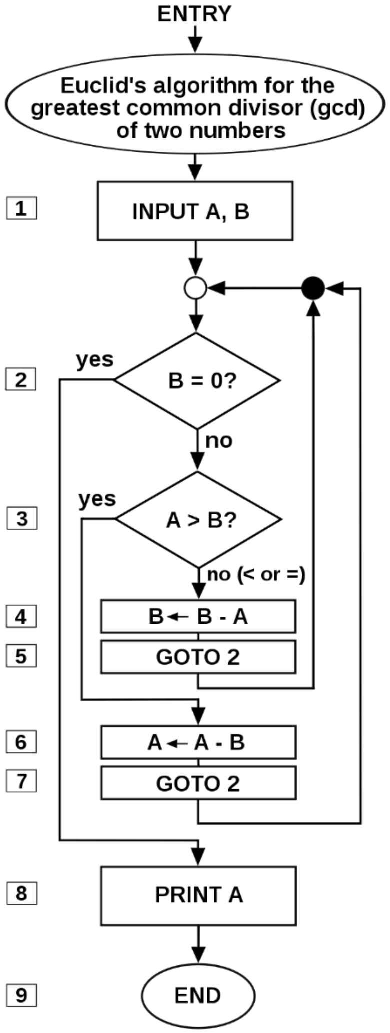
Recipe:

- 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:

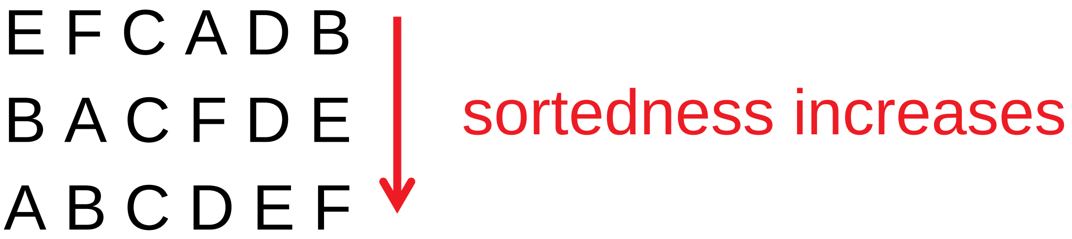
- A computer solves a problem
- Independent of programmer, computer, programming language, …
- Actually, a computer is running an *implementation* of an algorithm

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

Be Aware

Caution:

This is not an “algorithms for data scientists” lecture (!)

- we do not cover algorithms for regression, regularization, dimensionality reduction, clustering, deep learning, …
- …but cover much more basic things:
  - data structures
  - data sorting
  - fundamental algorithm design ideas
  - how to analyze an algorithm
  - how to prove lower runtime bounds for hard problems
  - …
- the actual data science related topics are taught in later lectures

What we plan to do in the A&C lecture

Learning Goals:

❶ know basic design principles behind good algorithms (*“building blocks to help solving “your own” problems”*)

❷ be able to analyze theoretically some algorithms

- give strong bounds on their “effectiveness”
- understand the ideas of (worst case) algo complexity (*"Am I too dumb to find a quick algorithm or can nobody do better?"*)

❸ be able to use and understand existing algorithms (*“practice, practice, practice!”*)

What we plan to do in the A&C lecture

How are we going to do that?

- look at a lot of examples of algorithms
- mixture of lectures and small exercises
- practice and theory
- additionally 1 home exercise per week

Please ask questions
if things are unclear throughout the course!

Course Overview

Thu		Topic
Mon, 21.09.2020	PM	Introduction, Combinatorics, O-notation, data structures
Mon, 28.09.2020	PM	Data structures II, Sorting algorithms I
Mon, 5.10.2020	PM	Sorting algorithms II, recursive algorithms
Mon, 12.10.2020	PM	Greedy algorithms
Mon, 19.10.2020	PM	Dynamic programming
Mon, 2.11.2020	PM	Randomized Algorithms and Blackbox Optimization
Mon, 16.11.2020	PM	Complexity theory I
Mon, 23.11.2020	PM	Complexity theory II
Mon, 14.12.2019	PM	Exam

Remarks on Exercises I

- included within the lecture (typically 1/3 of it)
- expected to be done on paper or in python [we'll see…]
- hence, please make sure you have python installed on your laptop until the second lecture
- Anaconda is the recommended way to get there:

  `https://www.anaconda.com/distribution/`
- (basic) example solutions will be made available afterwards
- I will try to also include some interactive formats for the students online
- not graded but please see it as training for the exam

Remarks on Exercises II

In addition:

- 7 home exercises with 20 points each
- Counts 1/3 to overall grade (exam is the other 2/3)

Remarks on Exercises II

In addition:

- 7 home exerc
- Counts 1/3 to
- Graded as:

Achieved points	grade	Difference
$136 \leq p \leq 140$	20	4
$132 \leq p < 136$	19	4
$128 \leq p < 132$	18	4
$124 \leq p < 128$	17	4
$118 \leq p < 124$	16	6
$112 \leq p < 118$	15	6
$106 \leq p < 112$	14	8
$98 \leq p < 106$	13	8
$90 \leq p < 98$	12	8
$80 \leq p < 90$	11	10
$70 \leq p < 80$	10	10
$60 \leq p < 70$	9	10
$50 \leq p < 60$	8	10
$40 \leq p < 50$	7	10
$34 \leq p \leq 40$	6	6
…	1..5	6, 6, 6, 6, 6
$0 \leq p < 4$	0	4

Remarks on Exercises II

In addition:

- 7 home exercises with 20 points each
- Counts 1/3 to overall grade (exam is the other 2/3)
- Graded as explained before
- Group submissions of 5 students allowed (and highly encouraged!)
- But: maximally 3 submissions with the same student pair
- Exercise available on Mondays
- Deadline for submission by email on Fridays
    - tight, but allows me to hopefully have them corrected by the next lecture
- Solutions will be discussed during the next lecture

The Exam

- Monday, 14th December 2020 in the afternoon (3 hours)
- (most likely) multiple-choice with 20-30 questions
- (most likely) on-site + online [details to be shared later]
- open book: use as much material as you want
- in previous year: no electronic devices allowed that connect to the internet [we'll also see for this one ☺]

All information available at

`http://www.cmap.polytechnique.fr/~dimo.brockhoff/algorithmsandcomplexity/2020/`

and also on EDUNAO
(exercise sheets, lecture slides, additional information, links, ...)

any questions?

Overview of Today’s Lecture

Basics

- Fundamental combinatorics
- notations such as the O-notation
- algorithms on basic data structures
    - arrays
    - lists
    - trees
    - …

Basics I: Combinatorics

For this and the next parts, a nice-to-read reference is https://www.math.upenn.edu/~wilf/AlgoComp.pdf

Combinatorics = Counting

counting combinations and counting permutations

Why combinatorics?

- In order to compute probabilities

$$P(event) = \frac{\#\text{favorable outcomes}}{\#\text{possible outcomes}}$$

- Related to graph theory (later)
- Related to combinatorial optimization (later)

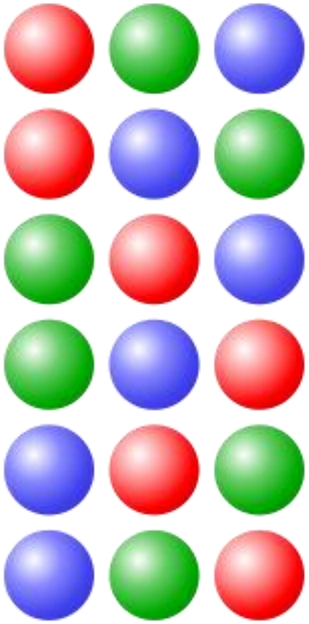
Number of Permutations

Permutation: a sequence/order of members of a set

How many different orders exist on $[n] := \ 1, \dots, n$?

- First integer: choice among n
- Second integer: choice among n-1
- Last integer: no choice among 1

- In total: $n \cdot (n-1) \cdot \dots \cdot 1 \ =: n!$



Watchduck (a.k.a. Tilman Piesk)

How to Generate a Random Permutation?

Idea: generate a random vector, sort it and use the generated sorting order as the permutation

```
import numpy as np
n = 4
random_array = np.random.rand(n)
random_perm = np.argsort(random_array)
```

More elegant way:

```
random_perm = np.random.permutation(n)
```
☺

Combinations Without Replacement (*k*-combination)

How many combinations of set members of a given size exist?

Example: number of different poker hands

- 52*51*50*49*48 = 311,875,200 ways to hand 5 cards out of 52
- but: order does not matter here!
- There are 5! = 120 orders of 5 cards
- Hence, there are 311,875,200/120 = 2,598,960 distinct pokers hands in total

In general, the number of k-combinations of n items (without replacements) is

$$\binom{n}{k} := \frac{n!}{k!\,(n-k)!}$$

Combinations with replacement

What if we want to allow duplicates?

- combinations with replacement
- also known as k-combination with repetitions or k-multicombination

Example:

Combinations with replacement

Wh

-
- ation

Exa

WestportWiki

Combinations with replacement

What if we want to allow duplicates?

- combinations with replacement
- also known as k-combination with repetitions or k-multicombination

Example:

eat 3 donuts from a choice of 4 different ones

Combinations with replacement

What if we want to allow duplicates?

- combinations with replacement
- also known as k-combination with repetitions or k-multicombination

WestportWiki

Example:

eat 3 donuts from a choice of 4 different ones

Number of k-combinations with replacement:

$$\binom{n+k-1}{k} \left[= \binom{n+k-1}{n-1} \right]$$

Here with $n = 4$, $k = 3$: $\binom{4+3-1}{3} = \binom{6}{3} = 20$ combinations

Why That? The Stars and Bars Method

Stars and Bars: A useful counting method popularized by W. Feller*

How many combinations to put k objects into n bins?

- objects: stars
- bins: separated by bars

- Example of n=5 bins and k=7 objects: ★ ★ |★|| ★ ★ ★ | ★
- Donut example: n=4 bins/donut types, k=3 objects

Number of combinations to put k objects into n bins
= number of combinations to place k objects on n+k-1 places ⇨ $\binom{n+k-1}{k}$
= number of combinations to place n-1 bars on n+k-1 places ⇨ $\binom{n+k-1}{n-1}$

How to Generate a Random k-Combination?

Naïve way:

```
from itertools import combinations
import numpy as np

n = 4
k = 2
# all k-combinations of [0, 1, …, n-1]:
comb = list(combinations(np.arange(n), k))

# pick one at random
random_k_combination =
   comb[np.random.randint(len(comb))]
```

Works only for small enough n and k:

`len(comb)` is 15,890,700 for n=50 and k=6
and 99,884,400 for n=50 and k=7

How to Generate a Random k-Combination?

More efficient way:

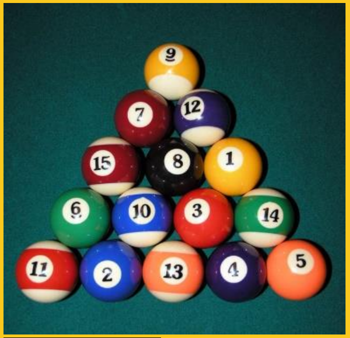
- iterate across each element of {1,…,n}
- pick each element with a dynamically changing probability of

$$\frac{k - \#samples\ chosen}{n - \#samples\ visited}$$

until k elements are picked.

Exercise

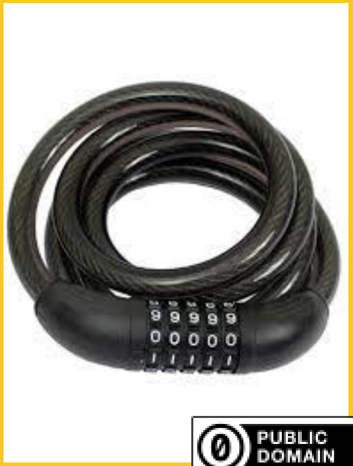
a) In how many different ways can the 15 balls of a pool billiard be placed (on a line)?



Jeanot

b) How many different combinations of five coins (Euros) can you have in your pocket?

c) How likely is it to get your bike stolen with the lock on the right?



Solutions

a) 15! (we look for the number of permutations of 15 distinct balls)

b) (8+5-1) choose 5 = 792 (8 different coins, choose 5 with repetition)

c) it’s pretty safe: the probability to find the right number is $\frac{1}{10^5} = 10^{-5}$, assuming that a random number out of all $10 \cdot 10 \cdot 10 \cdot 10 \cdot 10 = 10^5$ lock numbers is tried. It takes >10min to try out 1% of all 10^5 numbers if you try 2 lock combinations per second.

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:

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

n log n = Θ(ln(n!)) ln(n!)= O(e^n) e^n = O($e^{n\hat{}2}$)

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

One exemplary proof:

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

$$\left|\frac{\ln(n)}{\ln(\ln(n))}\right| = \left|\frac{\log(n)}{\log(e)\ln(\ln(n))}\right| \underset{\substack{\uparrow \\ \text{for } n>1}}{\leq} \frac{3\log(n)}{\ln(\ln(n))} \underset{\substack{\uparrow \\ \text{for } n>15}}{\leq} 3\log(n)$$

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

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

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

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

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	$O(2^n), O(n!)$, ...	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