Advanced Algorithms

Master Données et Systèmes Connectés

Master Machine Learning and Data Mining

Amaury Habrard

amaury.habrard@univ-st-etienne.fr

Laboratoire Hubert Curien, UMR CNRS 5516
Université Jean Monnet Saint-Étienne
amaury.habrard@univ-st-etienne.fr

1st Semester - September to December

Advanced Algorithms

What we will see:

- *non trivial* algorithms for solving some (easy) problems,
- algorithms (accompanied with heuristics) for solving *difficult problems*.
- How to characterize that an algorithm is "smarter" ("better") than another one?
  $\Rightarrow$ the complexity of the algorithm that can be measured in different ways (worst case, average case, best case, expected behavior,...)

But we will not see how to characterize the difficulty of a problem but you will study this in the complexity/calculability class.
$\Rightarrow$ this is generally done with respect to different classes of complexity

Materials - Mostly available on Moodle

- Slides and materials available online.
- Practical aspects: project - Erick Gomez (TA)
- References:
  - Jon Kleinberg and Eva Tardos. Algorithm Design. *Pearson International Edition, Addison Wesley*.
  - Thomas Cormen, Charles Leiserson, Ronald Rivest and Clifford Stein. Introduction to algorithms. *Dunod 2nd edition*.
- Evaluation:
  - Exam October 25: you will be allowed to use a copy of the slides and your personal notes on the exercices, no books, no electronic device.
  - Project (by groups): starting late October, defenses mid-december
  - **Course mark**: 50% exam + 50% project

Introduction and "recaps"

NP-completeness, Polynomial aspects, Optimization problems

An Algorithm - Quick Definition

- **Ordered sequence of precise deterministic steps**, that may involve frequent repetitions of some operations, such that when applied faithfully yields as a result the achievement of the task for which it was designed in a finite time.
- An algorithm is a finite, define, effective procedure with some input and some output. [*D. Knuth, The Art of Computer Programming*]
- Algorithms should be expressed in a precise and clear way, independently of a specific programming language (*e.g.* **pseudo-code**)
- **Validation of Algorithms**: Formal proofs for proving that all the outputs computed are correct for every inputs
- **Analysis of Algorithms**: the definition of the resources (time, space) needed to solve a problem, allowing a quantitative comparison.
- **Empirical study**: Empirical evaluation of time and space usage for different instances of various sizes - practical implementation dependent

Examples of iterative algorithms (1)

Input:
Output:
begin
  print("hello!");
end

Input:
Output:
begin
  for $i \leftarrow 1$ *to* 10 **do**
    print("Hey this is "+i+" ");
end

Input:
Output:
begin
  for $i \leftarrow 1$ *to* 173 **do**
    print("HoHo");
end

- Dependence on input size?
- $\rightarrow$ Constant

Examples of iterative algorithms (2)

```
Input: Table T[1...n]
Output: Integer
begin
    sum← 0;
    for i ← 1 to n do
        sum += T[i];
    for i ← 1 to n do
        print(T[i]+" ");
    print(sum);
    return sum;
end
```

```
Input: Table T[1...n]
Output:
begin
    for i ← 1 to n do
        print(T[i]+" ");
end
```

- Dependence on input size?
- → Linear

Examples of iterative algorithms (3)

Input: Tables $T[1..m]$ and $A[1...n]$ of real numbers
Output: Real value
begin
  if *some condition (e.g.* $n \leq m$*)* **then**
    for $i \leftarrow 1$ *to* n **do** $T[i] \leftarrow 2 * T[i]$;
  for $i \leftarrow 1$ *to* m **do**
    for $j \leftarrow 1$ *to* n **do**
      if $T[i] < A[j]$ **then** $T[i] \leftarrow A[j]$;
      else $A[j] \leftarrow T[i] \times A[j]$;
  return $\sum_{j=1}^{n} A[j]$;
end

Algorithm 1: Iterative Algo

Complexity quadratic related to $m \times n$

An example of a recursive algorithm

Input: Data x
Output: Object y
begin
  if x *is small enough* **then**
    use specific algorithm and return the result;
  else
    Decompose x into l equivalent subproblems $x_1, \ldots, x_l$;
    for $i \leftarrow 1$ *to* l **do**
      $y_i \leftarrow$ apply divide-and-conquer recursively on x_i;
    $y \leftarrow$ combine solutions $(y_1, \ldots, y_l)$;
    return y;
end

Algorithm 2: Divide-and-conquer principle

Complexity ?

Measuring the complexity by Asymptotic Order of Growth

Let $T(n)$ be a **non negative** function measuring the (worst case) running time of a given algorithm on an input of size n.

Classical measure: Asymptotic Upper Bounds

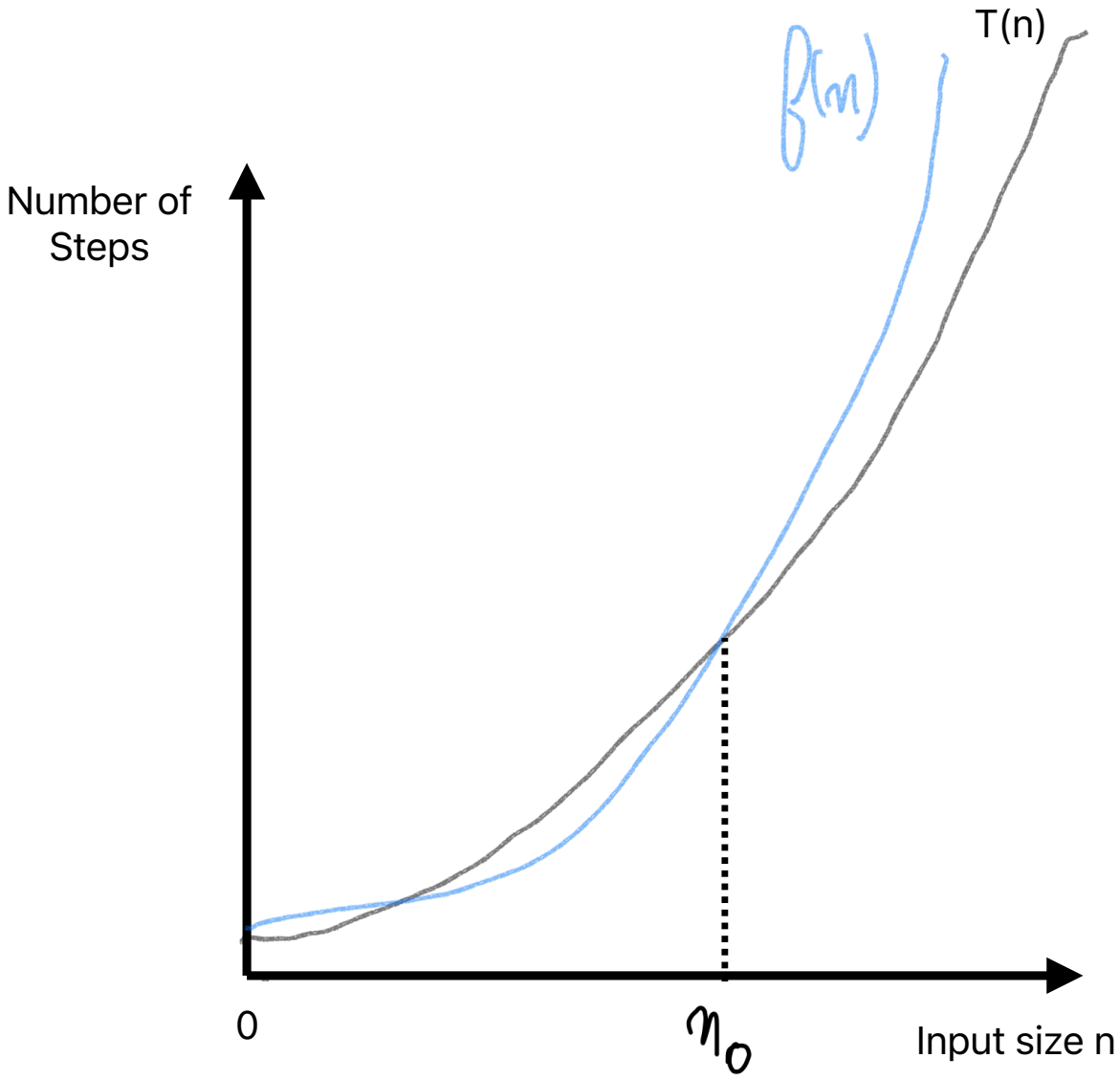
- Given a function $f(n)$, we say that $T(n)$ is $O(f(n))$ (*ie* $T(n)$ *has order of* $f(n)$ *in terms of time complexity*) if there exist **2 constants** $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$ we have
$$T(n) \leq c \times f(n).$$
In words: *for sufficiently large* n, $T(n)$ *is bounded above by* $f(n)$ *up to a constant factor*.

Note that $O(\cdot)$ expresses an **asymptotic** upper bound and not the exact growth rate of the function $T(n)$ encoding the complexity of the algorithm. The constant c must be fixed **independently** from n.

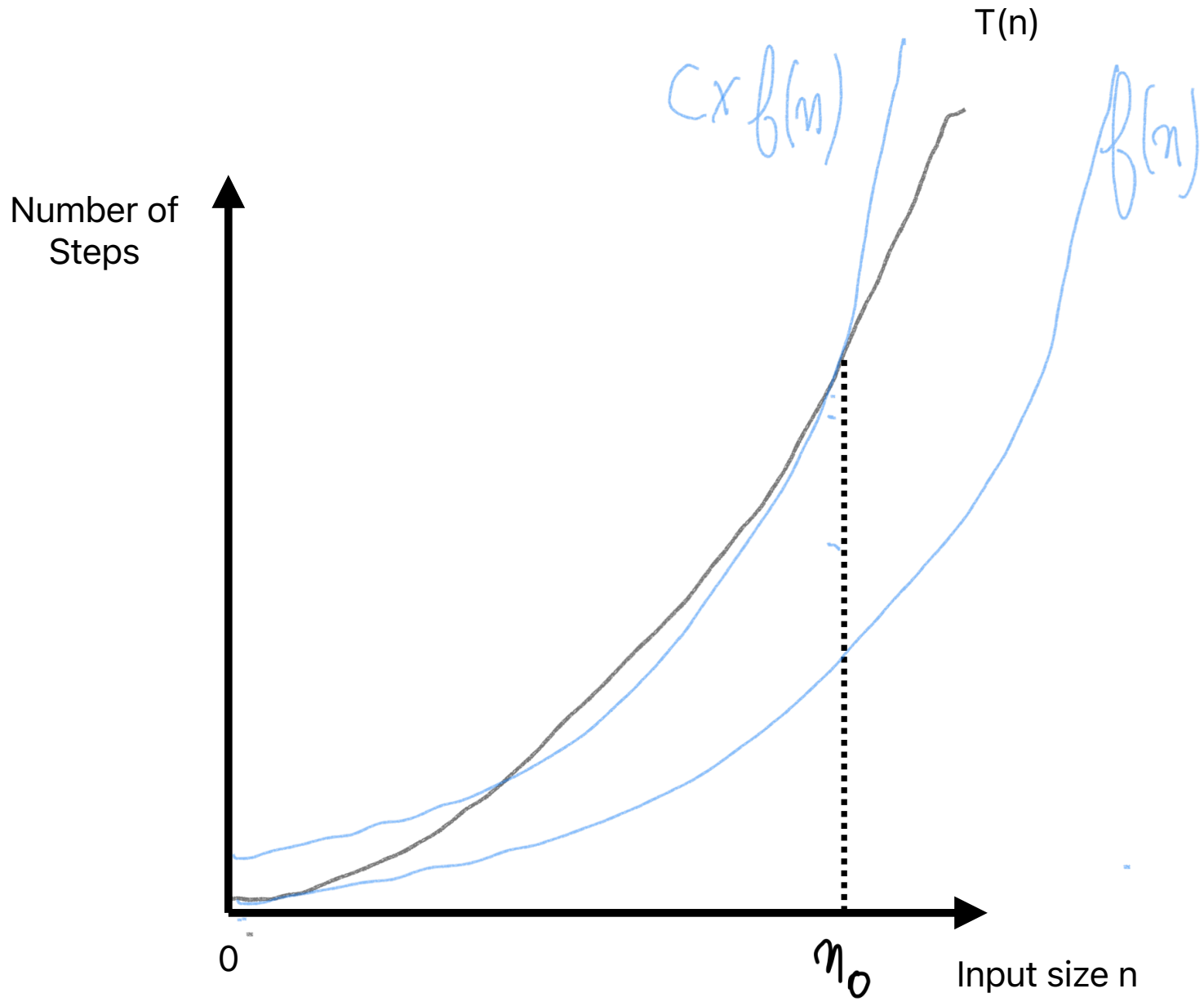
$\Rightarrow$ Try to draw some plots to illustrate the definition

$\Rightarrow$ What is the intuition behind the two constants?

An illustration



An illustration (2)



Example

- $T(n) = 3n^2 + 2n + 3$
- $f(n) = n^2$
- When $n \geq 1$, $T(n) \leq 3n^2 + 2n^2 + 3n^2 = 8n^2$
- then, we found $c = 8$ and $n_0 = 1$ that makes for all $n \geq n_0$: $T(n) \leq c \times f(n)$
- $T(n) \in O(f(n))$
- Note: we can improve c here !
- However, of having tight c and n_0 would be interesting, we are rather interesting in asymptotic behaviors and we only nefind c and n_0 that work !
- Note that all functions "above" f would work as well, but we are rather interesting in having a tight f which is key here for complexity estimation!

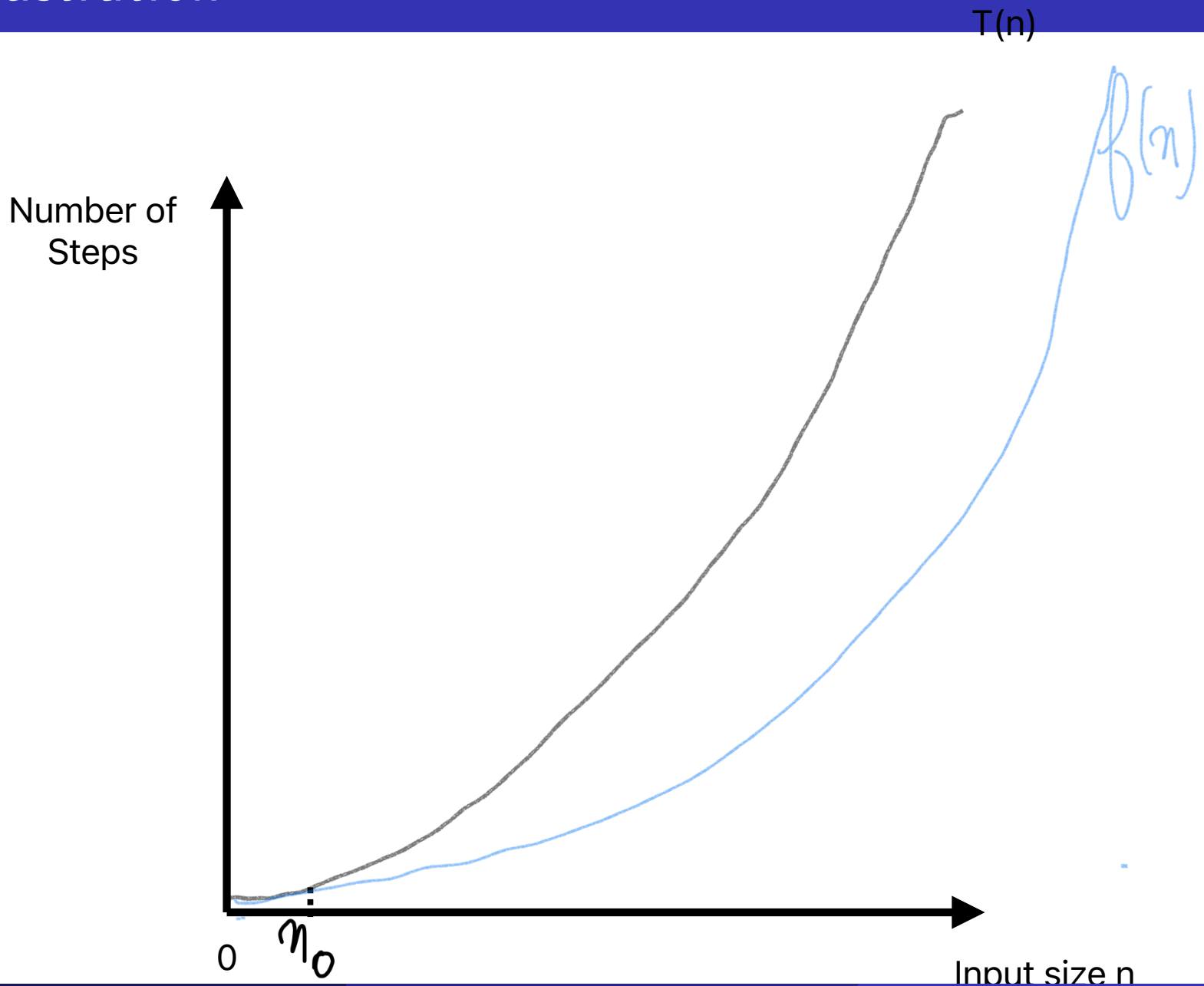
Measuring the complexity by Asymptotic Order of Growth (below)

Classical measure: Asymptotic Lower Bounds

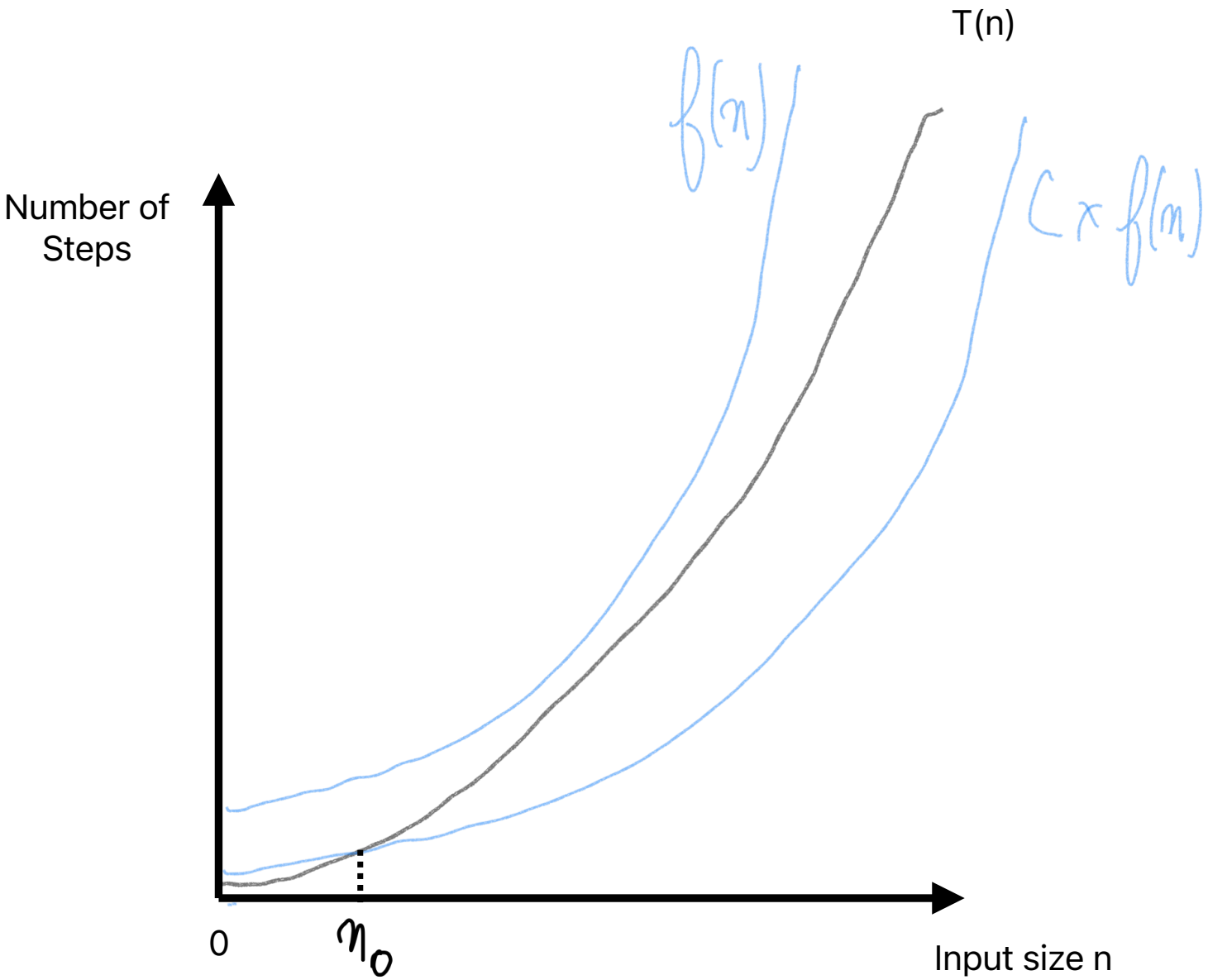
- Given a function $f(n)$, we say that $T(n)$ is $\Omega(f(n))$ (*i.e.* $f(n)$ *is order of* $T(n)$ *in terms of time complexity*), also written $T(n) = \Omega(f(n))$ or $T(n) \in \Omega(f(n))$, if there exist 2 constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have
$$c \times f(n) \leq T(n).$$
In words: *for sufficiently large* n, $T(n)$ *is asymptotically lower bounded by (a constant factor of)* $f(n)$.

Note that $\Omega(\cdot)$ expresses an **asymptotic** lower bound that does not follow necessarily the exact growth rate of the function $T(n)$.

An illustration



An illustration (2)



Example

- $T(n) = 3n^2 + 2n + 3$
- $f(n) = n^2$
- When $n \geq 0$, $T(n) \geq f(n)$
- then, we found $c = 1$ and $n_0 = 0$ that makes for all $n \geq n_0$: $T(n) \geq c \times f(n)$
- $T(n) \in \Omega(f(n))$
- Similarly, as for the upper bound, we are rather interesting by having a tight f characterizing the complexity rather than having precise c and n_0 values (asymptotic behavior)

Measuring the complexity by Asymptotic Order of Growth (average)

Classical measure: Asymptotic Tight Bounds

- Given a function $f(n)$, if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$ then $T(n)$ is $\theta(f(n))$, also written $T(n) = \theta(f(n))$.
  In words: *for sufficiently large n, $T(n)$ grows exactly like $f(n)$ to within a constant factor.*

Note that $\theta(\cdot)$ expresses an **asymptotic** tight bound on the order of growth of $T(n)$.

For the three measures, $f(n)$ is assumed to be a positive function, which is natural for modeling time complexity.

Example

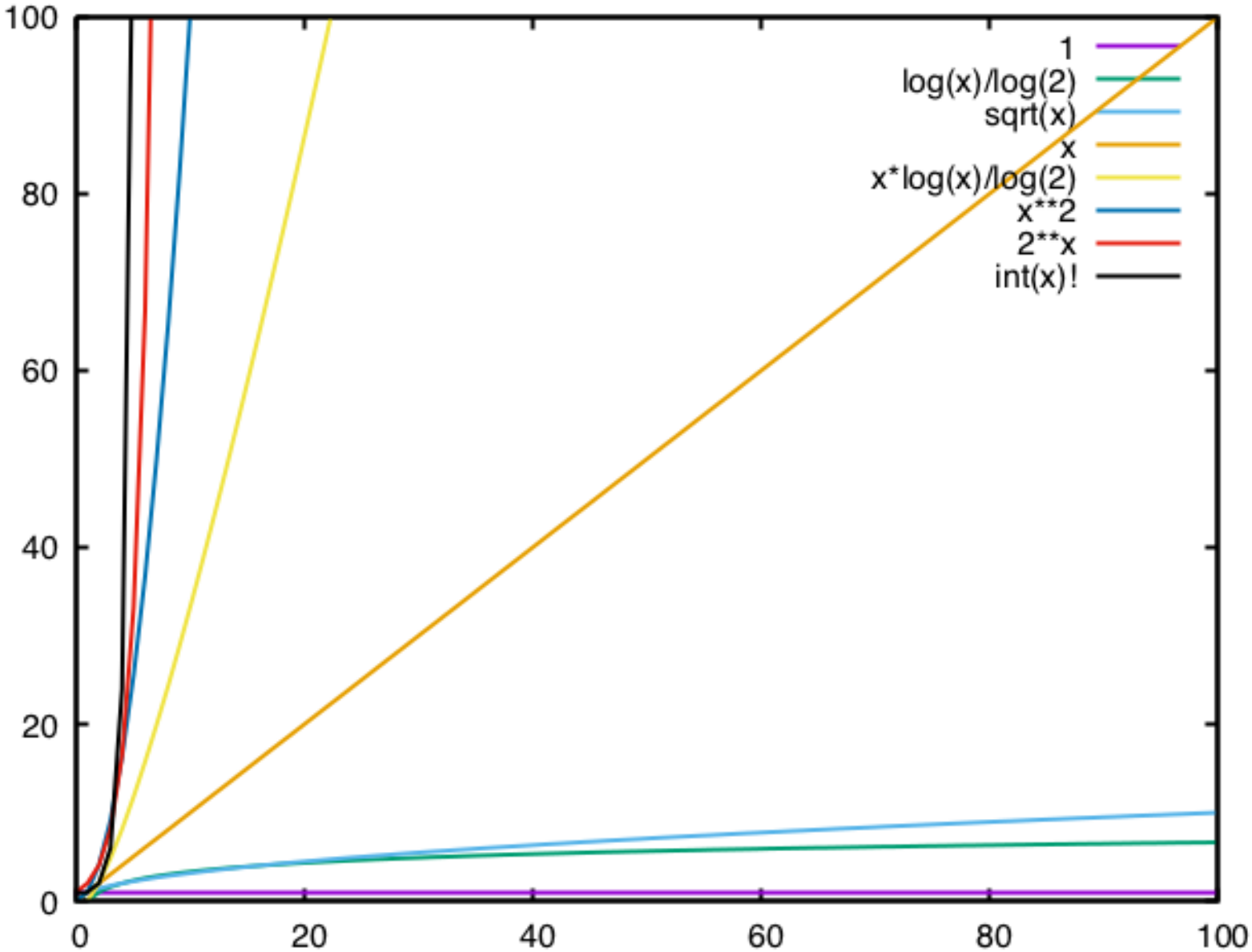
- $T(n) = 3n^2 + 2n + 3$
- $f(n) = n^2$
- Since we have $T(n) \in \Omega(f(n))$ and $T(n) \in O(f(n))$, then $T(n)$ is $\theta(f(n))$

Survey of classic running time

- Constant time $O(1)$: atomic operations
- Linear time $O(n)$: search in an unsorted array
- $O(n \log n)$: sorting, some divide and conquer approaches, sometimes called quasi linear
- Quadratic time $O(n^2)$:
- Polynomial time $O(n^k)$ with $k \geq 1$ a constant
- Exponential time: $O(2^n)$ and more generally $O(k^n)$ with $k > 1$ a constant (or $O\left(k^{\text{poly(n)}}\right)$)
- Factorial time $O(n!)$
- Doubly exponentiel $O(2^{2^n})$, or more...

Note: for any $x > 0$, $\log_b(x) = \log_a(x)/\log_a(b)$

Illustration



From a practical side (1/2)

Complexity	$\log_{10}(n)$	n	n^2	n^3	2^n	$n!$
duration $= 1s$	*astronomic* : 10^{10^6}	10^6	10^3	10^2	19	10
$1min$	*astronomic*	6.10^7	8.10^3	4.10^2	25	11
$1h$	*astronomic*	4.10^8	2.10^4	1500	31	13
$1day$	*astronomic*	9.10^{10}	10^5	4400	36	16

Table 1: Approximative amount of a data an algorithm can process with respect to a duration in line and a complexity in column. For example, in 1 hour a program with an exact complexity of n^3 can process an input of size 1500. Astronomic refers to a size greater than the estimated number of atoms in universe (10^{80}). We assume that an elementary operation can be done in 1μs.

(From Bosc, Guyomard and Miclet, Conception d'algorithmes, Eyrolles).

From a practical side (2/2)

Complexity	$\log_2(n)$	n	$n\log(n)$	n^2	2^n
size = 10	$3\mu s$	$10\mu s$	$30\mu s$	$100\mu s$	$100\mu s$
100	$7\mu s$	$100\mu s$	$700\mu s$	$1/100s$	10^{14}centuries
1000	$10\mu s$	$1000\mu s$	$1/100s$	$1s$	*astronomic*
10000	$13\mu s$	$1/100s$	$1/7s$	$1.7min$	*astronomic*
100000	$17\mu s$	$1/10s$	$2s$	$2.8h$	*astronomic*

Table 2: Assuming that an elementary operation can be done in 1μs, this table gives the time an algorithm needs to process an amount of data of size mentioned in line given an exact complexity in column. For example, a program in n^2 can process an input of size 1000 in one second. An astronomic time is larger than a billion billion years.

(From Bosc, Guyomard and Miclet, Conception d'algorithmes, Eyrolles).

Polynomial time complexity: A class of interest

From a theoretical standpoint, it is generally admitted that problems that can be solved in polynomial time algorithm are tractable. However as we saw, there are some exceptions:

- If the constant k is too big *e.g.* $O(n^{100})$ but even $O(n^4)$
- If the input data are too large *e.g.* $n \sim 10^9$ (big data)

Polynomial time aspect is interesting in various aspects:

- In many computational models a problem that can be solved in polynomial time in a model can also be solved in polynomial time in another one. For example, the class of problems solved in polynomial time by a Turing machine can also be solved in poly time by a parallel computer.
- Closure properties:
  - If an algorithm makes calls to many polynomial algorithms a finite number of times, it remains polynomial.
  - Using the output from a polynomial time algo as input of another polynomial time algo remains polynomial.

Some classic properties

f, g, h functions taking non negative values

- If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$
- If $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$
- If $f \in \theta(g)$ and $g \in \theta(h)$, then $f \in \theta(h)$
- If $f \in O(h)$ and $g \in O(h)$ then $f + g \in O(h)$
- If $g \in O(f)$ then $f + g \in \theta(f)$
- If f is polynomial of degree d in which the coefficient a_d associated to the term of degree d is positive, then $f \in O(n^d)$
- For every $b > 1$ and every $x > 0$, we have $\log_b(n) \in O(n^x)$
- For every $r > 1$ and every $d > 0$, we have $n^d \in O(r^n)$

Abstract Problem, instance, solution

An abstract problem Q is defined as a binary relation between a set of instances of the problem and a set of solutions for this problem.

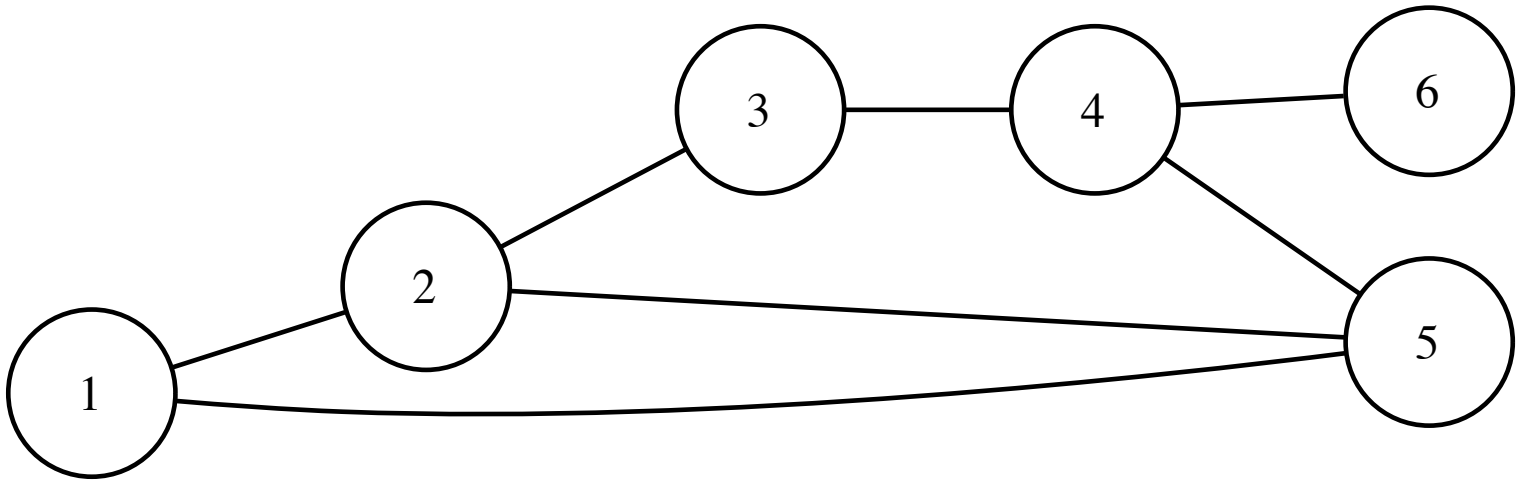
Example: Shortest Path Problem in a graph

Consider the problem where the objective is to find the shortest path between two nodes of an undirected and unweighted graph $G = (S, A)$

- Instance: a triple (G, u, v) where G is a graph and u and v are two nodes of the graph.
- Solution: a sequence a nodes defining a shortest path between u and v in G.

We associate to each instance (composed of a graph and two nodes u and v of the graph) a shortest path between u and v.

Example



The shortest path problem applied on this graph and the nodes 1 and 6 admits only one solution:

- 1, 5, 4, 6

Different versions of a problem

Decision problem

An abstract decision problem is a mapping from a set of instances to the set $\{0, 1\}$ of solutions.

Example: the *path* decision problem

Given a graph $G = (V, E)$, two nodes u and v, an integer $k \geq 0$, Is there a path between u and v of length lower or equal to k?

If $i = (G, u, v, k)$ is an instance of this problem, then $path(i) = 1$ if there exists a path between u and v of length lower or equal to k in G.

Concrete Problems and class of complexity P

Concrete problem

A concrete pb corresponds to the encoding of an abstract problem (use of data structures, use of binary strings, use of a formal language framework)

In this class we will not really pay attention to the encoding problems, even though it can be an important aspect, we will assume that every input/output is encoded as a string.

The class P

A concrete problem Q can be solved in polynomial time if there exists an algorithm solving it in $O(n^k)$ for any input of size n with $k \geq 0$ a constant. The set of problems that can be solved in polynomial time defines the class of complexity P.

Polynomial time certification - the class NP

Polynomial certifier

An abstract problem admits a polynomial certifier if for any instance $\mathtt{s}$ of the problem and any proposed solution $\mathtt{t}$ (a certificate), there exists a **polynomial** algorithm that can check if $\mathtt{t}$ is a solution for $\mathtt{s}$.
Note: $|\mathtt{t}|$ must be $\leq poly(|\mathtt{s}|)$, the polynomial constraint is only needed for positive solutions.

Non-deterministic Polynomial time (NP)

The class NP is defined as the set of problems for which there exists a polynomial certifier.

Example with the *path* problem

Given a sequence a nodes, it is easy to check that it forms a path of length lower or equal than k.

NP problem - other example

Problem: Hamiltonian Cycle (HC)

Given an undirected graph $G = (V, S)$, is there a simple cycle containing each vertex of V exactly once?

- To solve this problem, we could imagine to consider all the possible permutations of the nodes of G and check if each permutation defined an Hamiltonian path: $\Omega(n!)$, clearly not polynomial.
- If we only look for a certifier, given a sequence of nodes, we need to verify if it is an Hamiltonian path: we just to see if there exists an edge between two consecutive nodes and check that all the nodes appear exactly once $\rightarrow$ polynomial (linear).
- Thus $HC \in NP$

NP-Completeness (seen in the complexity Class)

Reductions and transitivity

A problem Q can be reduced to a problem Q' (denoted by $Q \leq_P Q'$) if there exists a function, computable in polynomial time, able to transform any instance of Q to an instance of Q'. Q is polynomial-time reducible by Q' or in other words any instance of Q can be solved by Q' by means a polynomial transformation.

The notation $P_1 \leq P_2$ means that P_2 is as difficult as P_1, with respect to a polynomial factor.

Another interpretation: (i) if P_2 can be solved in polynomial time, then P_1 can be solved in polynomial time, (ii) if P_1 cannot be solved in poly time, then P_2 cannot be solved in poly time.

NP-complete problems (seen in the complexity Class)

NP-Complete (NP-C) problems

A decision problem Q is NP-complete if:

1. $Q \in NP$
2. For every problem $Q' \in NP$, $Q' \leq_P Q$

The set of NP-complete decision problems defines the class of complexity NPC. If a problem verifies the property 2 (and not necessarily property 1), it is said *NP-hard*. The class of NP-hard problem is thus a superset of the class NPC, thus any optimization problem having NP-complete decision problem version is NP-hard.

$\Rightarrow$ NP-complete problems have an "interesting property": they are all equivalent with respect to a polynomial factor. As a consequence, if one of the problems can be solved in polynomial time then all the problems in NP can be solved in polynomial time !

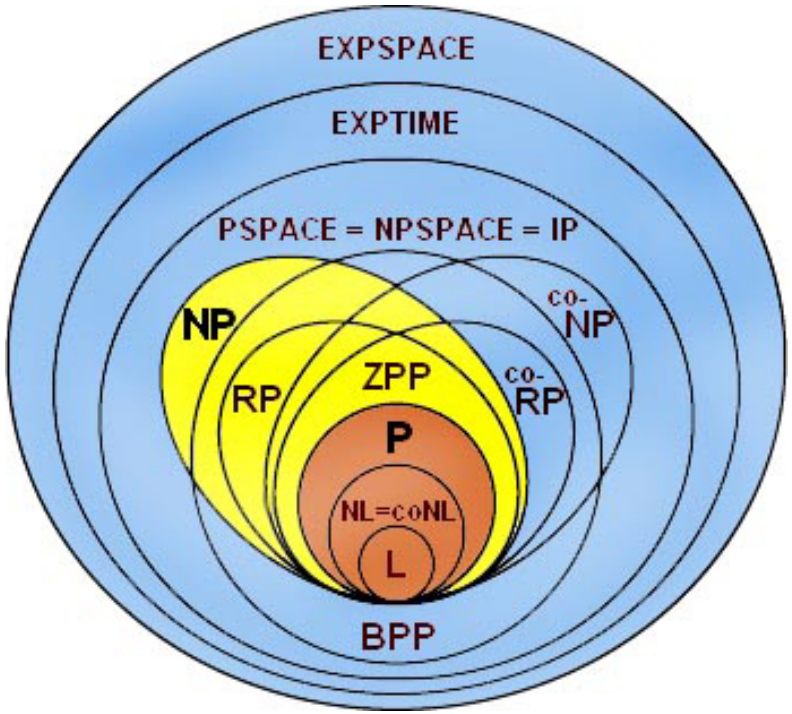
NP-complete problems

Important remarks

- we know that $P \subseteq NP$: easy since any problem P can be solved in polynomial time, you have directly a certifier,
- The question ?$NP = P$? is still an open problem today ...
- Note that If one can prove that one NP-complete problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time.
- The term *NP-complete* is used for decision problems, otherwise we use the term *NP-hard* for the associated general problem: they correspond to the problems that are at least as hard as the hardest problems in *NP*.

Note : in this class we will not directly work on the notion of NP-complete problems - you will study this in the complexity/calculability class.
But we will see some strategies to solve in practice these problems.

Note: there exist many classes of complexity



Problems solvable in EXP-TIME, EXP-SPACE, P-SPACE, LOG-SPACE, randomized algorithms. The figure above is not complete.

Illustration taken from

`https://jeremykun.com/2012/02/29/other-complexity-classes/`

Optimization problems

In this class, we will generally consider optimization problems. They have potentially different solutions, each of these having a specific value and we look for the solution with the best possible value (minimum or maximum): the optimal value. If there are more than one optimal values, we just take one.

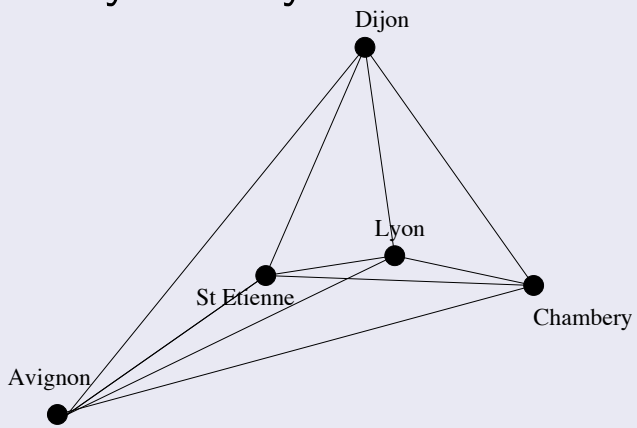
Shortest path

Given a triple (G, u, v) where $G = (S, A)$ is a graph and u and v are two nodes of the graph, find the minimum path between u and v in G.

Optimization problems

TSP: Traveling Salesman Problem

Given a list of cities and the distances between each pair of them, find the shortest possible route that visits each city exactly once and returns to the origin city?

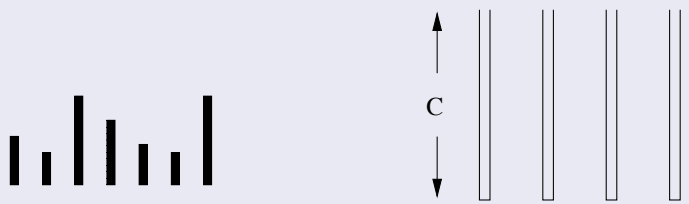


This can be modeled as finding an Hamiltonian cycle with the minimal weight in an undirected weighted graph.

Optimization problems

Bin packing

Given a finite set of bins of capacity C, and a set of objects $\{o_1, \ldots, o_n\}$ each of them with a capacity v_i, is it possible to pack all of them using less than k bins?



a set of 7 objects a set of 4 bins of capacity C

Knapsack problem

Input: the capacity C of the knapsack, n different types of objects, and for each object its weight $w(i)$ and its value $v(i)$

Output: Find a collection of objects such that the sum of the individual weights is lower or equal than C and that maximizes the sum of the individual values.

Needing heuristics

Optimization problems are generally NP-hard. Using naive algorithms to solve them is generally intractable (exponential time complexity). The idea is to use heuristics to find approximated reasonable solutions in reasonable time. This can be done by using various algorithms:

- Dynamic programming
- Greedy algorithm
- Stochastic/randomized approaches
- Branch and bound algorithms
- Linear programming
- Genetic algorithms
- ...

In this class, we propose to study some of these methods. We will sometimes see that for some problems that may appear difficult, there exits a polynomial solution using a non-naive analysis of the problem (dynamic programming and greedy approaches).

Next !

- We will see two strategies for solving problems that may appear (NP-)Hard in polynomial time : Dynamic Programming and Greedy Approaches.
- Then, we will some algorithm and strategies for solving or at least approximate the solution of NP-Hard problems. **Note that Greedy strategies can also be used to find approximated solutions.**