4 – Complexity

Luc Lapointe
luc.lapointe@ens-paris-saclay.fr
home.lmf.cnrs.fr/LucLapointe/

Abstract

The *theory of complexity* aims at classifying problems according to the resources needed to solve them, be it time or memory. It gives tools to prove that some problems always need a lot of computation time or memory, whatever the algorithm used to solve them.

1. What does “complexity” mean

In computer science, *complexity* is a word used to describe how much time or memory resources an algorithm or a program needs. In order not to make theory dependant on hardware, complexity tends to be explain as a function of size of the entry, rather than as a time or a number of octets.

Example All of those examples are admitted.

- If an input is n bits long, reading it takes a time linear in n.
- Reading each cell of a square matrix of size n takes a time linear in n^2.
- Sorting a list of n elements with merge sort takes a time in $n \log n$, and can be achieved by using an amount of memory independent of n.
- Finding the shortest path from one point to another on a map with n crossroads with the original Dijkstra algorithm takes at most a time linear in n^2.
- Solving a Sudoku grid with n empty cells by testing all possible solutions can be achieved in time at most linear in $n10^n$.

On all of those examples, it is the complexity *of an algorithm* that is given. We would like to define the complexity *of a problem* as something like

> “The amount of time or memory that is absolutely necessary to solve a problem.”

How to count? In order to have a clean count of the complexity of those algorithms, we must describe precisely what we are counting. For example, with sorting algorithm, the atomic operation (counting for 1 in the complexity) is *comparing two integers of the list*. The complexity of merge sorts (and other sorts) describes the number of comparisons.
Choosing the atomic operation depends on what we want to evaluate, and the atomic operation itself can, in some context, be a non-trivial operation. Comparing two binary integers, for example, has complexity 1 if the atomic operation is comparing the integers themselves, but complexity linear in their size if it rather is comparing bits.

Upper bound on a problem complexity With such definition of *problem complexity*, we can easily find an upper bound on the complexity of a problem $\mathcal{P}$, by **exhibiting an algorithm that solves it**. We know, for example, that sorting a list does not "absolutely necessarily needs" a time linear in n^2 to be achieved, as merge sort takes a time linear in $n \log n$.

Lower bound on a problem complexity Finding a lower bound on the time or space needed to solve a problem is, however, way harder. For a problem we currently find really hard to solve, how sure can we be that it is because the problem is indeed hard, or because we just do not have the correct tools yet?

A recent example of this dilemma can be found in the complexity of testing wether an integer is prime or not. For a really long time, only a method in time exponential in the number of digit of the input was known. It is only in 2002 that a method in time polynomial in the number of digit has been discovered [1].

On most problem, we simply do not have a precise lower complexity bound, such as n or n^2. Theory of complexity circumvents this issue by building huge families of somehow "equivalent" problem, in the sense that for a given family we have algorithms of the same complexity to solve them, and a major improve in the solving of one problem of the family results in an improve in each problem of the family.

The aim of this lecture is to present three of these families : P, NP and BPP.

2. A class of tractable problems: P

Counting complexity of families of algorithm with common aim, such as sorting algorithms, can be achieved by describing atomic operations, such as integers comparisons, and counting the number of atomic operations.

Counting the complexity of huge families of algorithms or problems can not be achieved this way, because such families can regroup algorithm of very, very different nature, such as:

- Problems on networks or maps,
- Mathematical programming,
- Text processing,
- Solving games and puzzle…

They however all have in common that they are described by Turing machines.

Definition: Turing machines complexity

Let $\mathcal{M}$ a Turing machine and $f : \mathbb{N} \to \mathbb{N}$ a function.

- The **time complexity** of $\mathcal{M}$ is *bounded by* f if, on all inputs of size n, it ends in at most $f(n)$ transitions.
- The **space complexity** of $\mathcal{M}$ is *bounded by* f if, on all inputs of size n, it never uses more than $f(n)$ different cells of the memory tape.

There are different ways of grouping problems around common characteristics. In this lecture we will only consider characteristic of computation time efficiency for different Turing machine models.

For inputs of size n, you might think of designing families of problems that are solved in time at most linear in n, or in n^2, or in n^3, or in $n \log n$... This is a reasonable idea, but for technical reasons not detailed here, such classes are not large enough to allow powerful theorems such as

"A major improve in the solving of one problem of the family induces major improves for all of them."

This is however the case for the family of problems that can be solved in time at most polynomial in the size of the input. This class of problems is called PTIME, or more often just P.

Definition: PTIME

PTIME, often shortened to **P**, is the set of decision problems $\mathcal{P}$ that can be solved in **p**olynomial **time** by a deterministic Turing machine.

In other words, it is the class of decision problems $\mathcal{P}$ with a deterministic machine $\mathcal{M}$ solving them, and such that there exist a *polynomial* function $f_{\mathcal{M}}$ that bounds the *time* complexity of $\mathcal{M}$.

Note

- There is absolutely no consideration about space use of the machines in the formal definition of P. However, as writing 1 cell takes 1 transition, machines in P always use at most a polynomial amount of space.
- Problems in P are considered effectively computable by our real-world machines. That's why they are sometimes called *tractable problems*.
- Note that only *decision problems* are in P.

How to evaluate optimization problems? The three complexity classes presented in this lecture only contain decision problem. There are two important reasons for this choice:

1. Our current tools only allow "major improve transfer" theorems among a given family if it contains decision problems only.
2. Optimization problems can be transformed into decision problems by adding a threshold as follows. Consider a maximization or minimization problem of the form:

Input: Some data x.
Question: What is the maximal/minimal value of parameter f for data x?

Finding the shortest path between two coordinates on a given map is an example of such problem.

This can be transformed into a decision problem by introducing a constant λ which is **not** an input of the problem, and by considering the following decision problem:

Input: Some data x.

Question: Can parameter f grow bigger/go lower than λ for data x?

e.g. finding a path shorter than λ between two coordinates.

Some problems in P The following problems are representative P problems.

- Circuit Value: Finding the output of a boolean circuit.
- Grammar Membership: Checking wether a word is in the language of a grammar.
- Linear Programming: Is there a linear function subject to linear inequality constraints given as inputs.
- Maximum flow problem: Is there a way to reach a specific flow rate through a given network.
- Game of Life future: Given an initial configuration of Conway's Game of Life, a cell c, and an integer N, is c alive after N steps?

What next? P is currently considered as the hardest class of problems we can effectively solve with our real-world computers, on real data. There are a lot of problems we do not know for sure whether they are in P or not. In order to find the next level of difficulty for problem solving to classify them precisely, three interesting classes are possible:

1. The class of problems we can not easily solve yet, but whose solutions we can verify with our computers.
2. The class of problems that can be computed in polynomial time by a *nondeterministic* Turing machine, rather than a deterministic one.
3. The class of problems for which we still bound the computation time with a polynomial function, but we allow some mistakes in the answer.

The two first classes happen to be exactly the same one, whose name is NP. A class corresponding to the third (vague) description is BPP.

3. At the edge of real-world computability: NP

3.1. NP as a solution verifier

When trying to formalize the definition of NP as a solution verifier, we translate "we can verify with our computers" to "we can compute in polynomial time".

Definition: NP as a solution verifier

NP is the class of decision problems $\mathcal{P}$ such that, when given a solution attempt in addition to the input, it can be checked in polynomial time if the solution attempt answers the problem.

Example For a decision problems like

Input: A list of items.
Question: Can all items fit in your rucksack?

a solution attempt would be an explanation of how to place the items inside the rucksack. It can be easily check if they were all put inside, or if some are still missing.

3.2. NP with nondeterministic machines

Definition: NP with nondeterministic machines

NP is the class of decision problems $\mathcal{P}$ that can be solved by a **nondeterministic** Turing machine in **p**olynomial **time**.

A remainder about nondeterministic Turing machines Nondeterministic Turing machines, just as nondeterministic automata, can at some points of their computations have multiple possible choices. They accept an input if and only if there is *at least* one sequence of choices such that the word is accepted.

Some intuition about nondeterminism Nondeterministic machines can be described with pseudo-code, augmented with one specific operation: guessing. At the machine level, it is described with nondeterministic transition.

Here is an example of nondeterministic pseudo-code describing how to find a password of size at most N, along with the corresponding nondeterministic Turing machine.

```
password():
  answer = []
  for i in [1, N]:
    answer[i] := guess a letter
  return answer
```

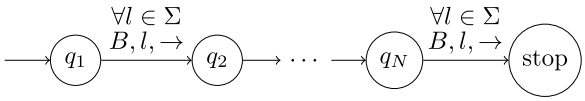

Figure 1: Non-determinism adds a "guess" basic operator.

Exercise: I see no difference between the two NP

- Try to find how to simulate guessing with a solution attempt input.
- Try to find how to simulate a solution attempt input with guessing.

3.3. Some problems in NP

The following problems are representative NP problems.

- Dominating Set: On a city map, is it possible to put less than λ street lamp at crossroads, so that each road has a street lamp at one of its end.
- Traveling Salesman: On a country map, is it possible for a traveling salesman to go to each city once by traveling less than λ kilometers.

- Minimum cut: On a network, is it possible to split the network in two by blocking less than λ connections.
- For a lot of boardgames[1]: how to win.
- Circuit Satisfiability: For a given boolean circuit, are there inputs so that the circuit answers *True*.
- Optimal Job Scheduling: For a list of tasks to achieve with n processing units, is it possible to achieve them with a time of less than λ.

3.4. One solution to solve them all

Representative NP problems are hard to solve efficiently, but we currently have no proof, for any of them, that there exist no efficient solution. It is an open problem in computer science, stated by the famous Millennium Prize Problem P versus NP, whose question is very short:

Does P equal NP?

In term of intuition of what P = NP means, it is very similar to something like "If I can recognize good music, then I can compose good music". It seems incredibly powerful, obviously false. But in the theoretical world, no one has managed to prove yet that this equality is false (nor true).

Meaning of the equality This equality is a set equality, i.e. a double set inclusion. We already know that

$$\mathrm{P} \subseteq \mathrm{NP}$$

because if we can solve a problem with a deterministic machine, we can also solve it with the same machine seen as a nondeterministic one. We do not know yet if the converse is true or not.

One step toward the solving of the problem In the late 1960s and early 1970s, a very powerful tool has been developed both by American and Soviet researchers. This tool is called *NP-hardness*[2]. Problems that are *NP-hard* are problems designed to be "As hard as any other NP problem".

How to achieve NP-hardness? Let $\mathcal{P}$ any problem in NP. The idea is: if we know how to solve a NP-hard problem $\mathcal{P}_{\text{hard}}$, we can solve $\mathcal{P}$ by encoding its input in $\mathcal{P}_{\text{hard}}$, and then solving $\mathcal{P}_{\text{hard}}$. The tool achieving this is called *polynomial reductions*.

[1] See https://en.wikipedia.org/wiki/List_of_NP-complete_problems#Games_and_puzzles

[2] For the American side, see [2] and [3]. For the Soviet side, see [4].

Definition: Reduction

Let $\mathcal{P}_1$ and $\mathcal{P}_2$ two decision problems. A **reduction from $\mathcal{P}_1$ to $\mathcal{P}_2$** is a **computable** function r that maps inputs of $\mathcal{P}_1$ into inputs of $\mathcal{P}_2$, and such that

$$x \text{ is a positive instance of } \mathcal{P}_1$$
$$\Leftrightarrow r(x) \text{ is a positive instance of } \mathcal{P}_2$$

Notation $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$ means that there exist a reduction from $\mathcal{P}_1$ to $\mathcal{P}_2$. We also say that $\mathcal{P}_1$ *reduces to* $\mathcal{P}_2$, or that $\mathcal{P}_2$ *is harder than* $\mathcal{P}_1$.

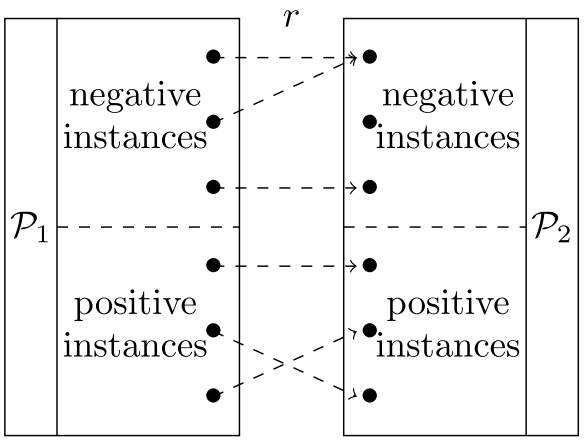

Figure 2: A reduction.

Note When there is a reduction from $\mathcal{P}_1$ to $\mathcal{P}_2$, then **all** possible instances of $\mathcal{P}_1$ must be encoded into instances of $\mathcal{P}_2$, but not all instances of $\mathcal{P}_2$ have to be encodings of an instance of $\mathcal{P}_1$.

Definition: Polynomial reduction

A **polynomial reduction** is a reduction whose underlying Turing machine time complexity is bounded by a polynomial function.

Definition: NP-hardness

Let $\mathcal{P}$ a decision problem. It is a **NP-hard problem** if, for all problem $\mathcal{P}'$ in NP, there exist a polynomial reduction $r_{\mathcal{P}'}$ from $\mathcal{P}'$ to $\mathcal{P}$.

Note

- A NP-hard problem do not have to be in NP. Indeed, there might be problems way harder than NP problems that are "as hard as any NP problems".
- A problem that is both NP-hard and in NP is called *NP-complete*. NP-complete problems are considered representative problems of the class NP.
- All "representative NP problems" in Section 3.3 are NP-complete problems.

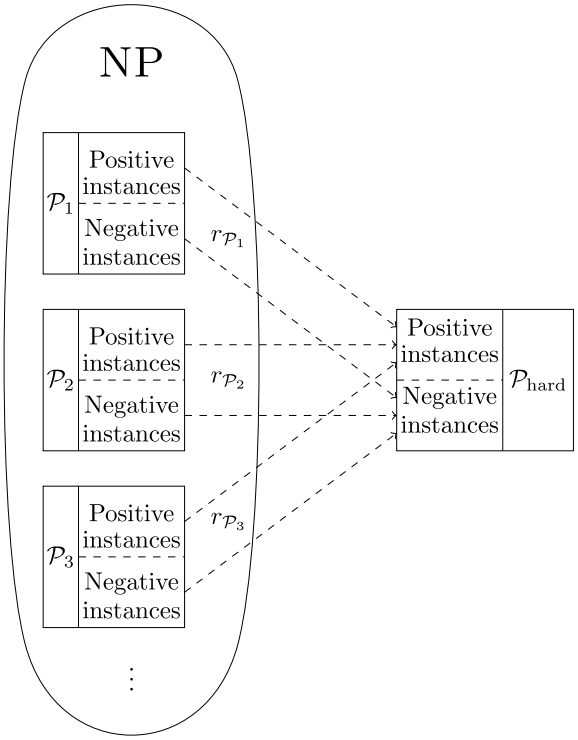

Figure 3: NP-hardness.

- A similar definition exists for other classes. All "representative P problems" in Section 2 are actually P-complete (and thus P-hard) problems.

Theorem: Use of polynomial reduction

Let $\mathcal{P}_1$ and $\mathcal{P}_2$ two decision problems. If both
- $\mathcal{P}_1$ is a NP-hard problem, and
- there exist a polynomial reduction from $\mathcal{P}_1$ to $\mathcal{P}_2$,

then $\mathcal{P}_2$ is also a NP-hard problem.

Theorem: Use of NP-hardness

Let $\mathcal{P}$ a NP-hard problem. If $\mathcal{P} \in \mathrm{P}$, then all problems in NP are also in P.

Exercise

Prove each of the two previous theorems.
Hint: do not hesitate to think with drawings.

Recap A NP-complete problem $\mathcal{P}$ is a problem that both is in NP, and is NP-hard. The consequence of being NP-hard is that if a polynomial algorithm is find for $\mathcal{P}$, then the whole NP class falls into P.

***Our* failings** When we do not find an efficient (i.e. polynomial) algorithm to solve a problem $\mathcal{P}$, it might be because there are none, or because we have not find it yet, and we should look more carefully, or discover some new revolutionary technique. NP-hardness allows to link our failings in finding such techniques to those of decades of past researchers, who also failed to find them[3]. Not finding a polynomial algorithm for a NP-hard problem becomes expected, rather than a failing. This is why proving that a problem is NP-hard also amounts to proving that a nondeterministic polynomial algorithm to solve it is some form of lower bound to its complexity.

4. Allowing uncertainty: BPP

One other way to extend P is to allow mistakes in the answer. This can be achieved by adding probabilities to Turing machines solving the problems.

A *probabilistic Turing machine* is a machine that can, at some points of its computation, test a random variable, and act depending on the result. It allows to design *probabilistic algorithms*, that are *likely* to answer correctly, but can still sometimes fail if not lucky enough.

The class BPP is a class that allows both false positive and false negative results.

[3] Or, sometimes, indeed found some new revolutionary techniques but with unexpected applications.

Definition: BPP

A problem $\mathcal{P}$ is in **BPP** if there exist a probabilistic machine $\mathcal{M}$ such that:
- The time complexity of $\mathcal{M}$ is bounded by a polynomial function,
- For all positive instances, $\mathcal{M}$ answers *True* with probability at least $\frac{2}{3}$,
- For all negative instances, $\mathcal{M}$ answers *False* with probability at least $\frac{2}{3}$.

BPP stands for **B**ounded-error **P**robabilistic **P**olynomial time.

Note
- Decision problems in BPP do not *a priori* need to have probabilities in their statements.
- Iterating the algorithm and picking the majority answer allows to have more precise results.
- Choosing any constant between 0 and $\frac{1}{2}$ defines the same class[4].
- BPP has a quantum complexity class equivalent, named BQP. You will learn more about it in some next lecture.

How are P, NP and BPP linked? P is included in BPP, as solving a problem in polynomial time with no possible mistake is indeed solving the problem with mistake probability less than $\frac{1}{3}$. We however do not know if it is included in NP, nor if NP is included in BPP.

In other words, we do not know if adding randomness improves computation power. It is currently believed that it does not, as more and more problems in BPP are proven to be in P. One recent example is primality testing: efficient probabilistic algorithms have been known for a long time, and it is only recently that a deterministic polynomial one has been discovered [1].

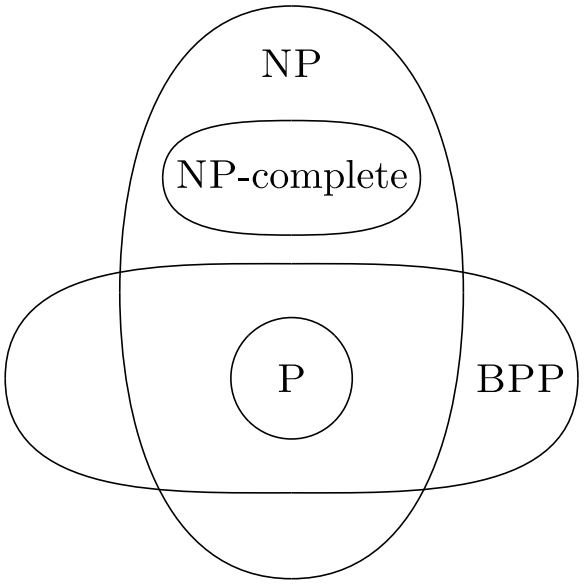

Figure 4: State of the art on complexity classes inclusions.

[4]This actually is a theorem.

Bibliography

[1] M. Agrawal, N. Kayal, and N. Saxena, “PRIMES is in P,” *Annals of mathematics*, pp. 781–793, 2004, doi: 10.4007/annals.2004.160.781.

[2] R. M. Karp, “Reducibility among Combinatorial Problems,” R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, Eds., Boston, MA: Springer US, 1972, pp. 85–103. doi: 10.1007/978-1-4684-2001-2_9.

[3] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, in STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. doi: 10.1145/800157.805047.

[4] Л. А. Левин, “Универсальные задачи перебора,” *Проблемы передачи информации*, vol. 9, no. 3, pp. 115–116, 1973.

I hope you liked this course! I would like to have your anonymous opinion about it.

https://framaforms.org/avis-cours-informatique-arteq-1-a-4-1728399026