SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

FOUNDATIONS OF COMPUTER SCIENCE
LECTURE 14: P vs NP

Prof. Daniele Gorla

Time complexity

Even when a problem is decidable, it may not be solvable in practice if the solution requires a huge amount of time or memory

In this course, we shall mainly focus on time

- time is an upper bound for space

  → if our algorithm needs *k* steps, it can access AT MOST *k* memory locations
- All the techniques we shall see for time have an analogous for space

How do we measure time?

- Our perception of time is something that depends on the machine used

  → too concrete to build a solid and long-time theoretical study
- Time for an algorithm is how many basic steps it requires for terminating

  → also the notion of «basic step» depends on the model used, but here the variations are less relevant

Time complexity, formally

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

The number of steps that an algorithm uses on a particular input may depend on several parameters

For simplicity, we compute the running time of an algorithm purely as a function of the length of the string representing the input and don't consider any other parameter

We consider the ***worst-case analysis***, i.e. the longest running time of all inputs of a particular length

DEFINITION

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f\colon \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

O(-) and o(-) Notations

The exact running time of an algorithm often is a complex expression

Asymptotic analysis estimates the running time of the algorithm on large inputs

- We consider only the highest order term of the expression
- Hence, we disregard both the coefficient of that term and any lower order terms

→ the highest order term dominates the other terms on large inputs

<u>Def.:</u> Let $f,g : \mathbb{N} \longrightarrow \mathbb{R}^+$. We say that

- $f(n)$ is $O(g(n))$, or $f(n) \in O(g(n))$, if there exists $c > 0$ s.t. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$

Said it differently:

$f(n) \in O(g(n))$ if there exists $c > 0$ and n_0 s.t., for all $n \geq n_0$, we have $f(n) \leq c\, g(n)$

EXAMPLE: $f(n) = 6n^3 + 3n^2 - 5$ is $O(n^3)$ (but no $O(n^k)$, for any $k \neq 3$)

Properties of O(-) Notation

- $O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$

- $O(f(n))\, O(g(n)) = O(f(n)\, g(n))$
  - $k\, O(g(n)) = O(k\, g(n)) = O(g(n))$
  - If $f(n) \in O(g(n))$, then $O(f(n))\, O(g(n)) = O(g(n))\, O(g(n)) = O(g^2(n))$

- $b^{f(n)} \in 2^{O(f(n))}$

  Indeed, $b^{f(n)} = (2^{\log_2 b})^{f(n)} = 2^{(\log_2 b) f(n)} = 2^{O(f(n))}$, since $(\log_2 b) f(n) \in O(f(n))$

- If $f(n), g(n) \in 2^{O(h(n))}$, then $O(f(n))\, O(g(n)) \in 2^{O(h(n))}$

  Indeed, $f(n)\, g(n) \in 2^{O(h(n))}\, 2^{O(h(n))} = (2^{O(h(n))})^2 = 2^{2O(h(n))} = 2^{O(h(n))}$

Complexity relations among models (1)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Thm.: Let $t(n)$ be a function, where $t(n) \geq n$. Then, every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

Proof:

Let M be a k-tape TM that runs in $t(n)$ time. We show that the single-tape TM S that simulates M runs in $O(t^2(n))$ time.

Recall that S uses

- its single tape to represent the contents on all k of M's tapes, separated by a #
- The tapes are stored consecutively, with the positions of M's heads marked on the appropriate cell (by dotting the character in the cell).
- Initially, S puts its input in the first portion of the tape and blank anywhere else.
- To simulate one step:
  - S scans all the information stored on its tape to determine the symbols under M's heads.
  - S makes another pass over its tape to update the tape contents and head positions.
  - If one of M's heads moves rightward onto some #, S must increase the space of this tape

    → It does so by shifting a portion of its own tape one cell to the right

Complexity relations among models (2)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

We have to estimate:

1. The length of the active portion of S's tape:
   - we take the sum of the lengths of the active portions of M's k tapes.
   - Each of these active portions has length $O(t(n))$ because M uses $t(n)$ tape cells in $t(n)$ steps if the head moves rightward at every step, and fewer if a head ever moves leftward.
   - Hence, the active portion of S's tape is $k\,O(t(n)) = O(t(n))$.
2. The cost for right-shifting:
   - Each uses $O(t(n))$ time, since $O(t(n))$ characters have to be shifted

Hence, to simulate each of M's steps, S takes $O(t(n))$, since it performs

- two scans of the active portion of the tape, and
- possibly up to k rightward shifts.

Overall:

- The initialization phase costs $O(t(n))$
- Each of the $t(n)$ steps of M is simulated with $O(t(n))$ steps by S; this costs $t(n)\ O(t(n)) = O(t^2(n))$
- Hence, the overall cost is $O(t(n)) + O(t^2(n)) = O(t^2(n))$, since $t(n) \geq n$. Q.E.D.

Complexity relations among models (3)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

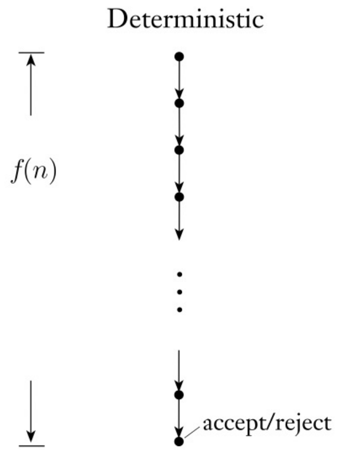
Hence, the time complexity between a multi-tape TM and its equivalent single-tape one has only a polynomial gap

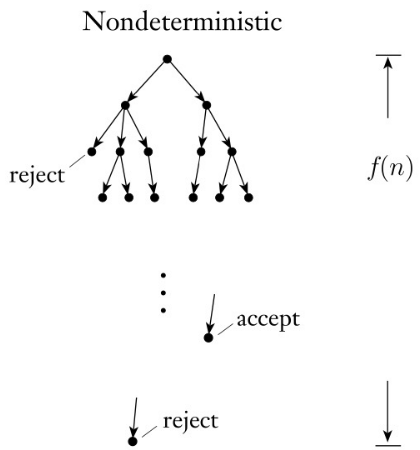
→ We will show that for non-determinism the gap is much higher, i.e. exponential

To this aim, let us first define the time complexity of a non-deterministic machine.

DEFINITION

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.





Complexity relations among models (4)

<u>***Thm.:***</u> Let $t(n)$ be a function, where $t(n) \geq n$. Then, every $t(n)$ time nondeterministic Turing machine has an equivalent $2^{O(t(n))}$ time deterministic Turing machine.

Proof:

Let N be a nondeterministic TM that runs in $t(n)$ time. We show that the there exists a (single-tape) deterministic TM D that simulates N and that runs in $2^{O(t(n))}$ time.

Recall that we need a 3-tape TM T that performs a breadth-first visit of N's computation tree:

- On an input of length n, every branch of such tree has a length of at most $t(n)$.
- Every node in the tree can have at most b children, where b is the maximum number of legal choices given by N's transition function (for a given state and input).
- Thus, the total number of leaves in the tree is at most $b^{t(n)}$.
- So, the total number of nodes in the tree is less than $2b^{t(n)} = O(b^{t(n)})$.
- To visit a node, we start from the root and follow the path to that node in time $O(t(n))$.
- Therefore, the running time of T is $O(t(n))\ O(b^{t(n)}) = 2^{O(t(n))}$.

Finally, converting T into a single-tape TM D costs $O((2^{O(t(n))})^2) = O(2^{2O(t(n))}) = 2^{O(t(n))}$.

<u>Q.E.D.</u>

Time Complexity Classes (1)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

- In computability theory, the Church–Turing thesis implies that all reasonable models of computation are equivalent
- In complexity theory, the choice of model affects the complexity of languages.
- Polynomial differences in running time are considered small, whereas exponential differences are considered large.

  → dramatic difference between the growth rate

  EX.: for n=1000, $n^3 = 10^9$ whereas $2^n >>$ the number of atoms in the universe

- We focus on aspects of time complexity theory that are unaffected by polynomial differences in running time.

  → Ignoring these differences allows us to develop a theory that doesn't depend on the selection of a particular model of computation.

- All reasonable ***deterministic*** computational models are ***polynomially equivalent***. That is, any one of them can simulate another with only a polynomial increase in running time.

Time Complexity Classes (2)

DEFINITION

$\textbf{TIME}(\boldsymbol{t(n)}) = \{L \mid L$ is a language decided by an $O(t(n))$ time deterministic Turing machine$\}$.

DEFINITION

$\textbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L$ is a language decided by an $O(t(n))$ time nondeterministic Turing machine$\}$.

P

DEFINITION

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

- **P** is invariant for all models of computation that are polynomially equivalent to deterministic single-tape Turing machines

  → it is a mathematically robust class (it isn't affected by the chosen model)
- **P** corresponds to the class of problems that are realistically solvable on a computer.

  → it is practically relevant:
  - any problem in **P** can be solved in time n^k for some constant k.
  - Whether this running time is practical depends on k and on the application.
  - Nevertheless, it is useful to have polynomial time as the threshold of practical solvability

On representing the input (1)

Having polynomiality as reference time model, also the encoding of the input must be «polynomial»

→ The running time of an algorithm also depends on how we represent its input

EXAMPLE:

- suppose that a natural number k is the only input to an algorithm
- suppose that the running time of the algorithm is $O(k)$
- If k is provided in ***unary*** (a string of k 1's), then the running time of the algorithm is $O(k)$, which is polynomial time.
- If we use the (more natural) ***binary*** representation of k, then the size of the input is $n = \lfloor log_2 \ k \rfloor + 1$.

  → the running time of the algorithm is $O(2^n)$, which is exponential in the input size
- Hence, depending on the encoding, the algorithm runs in polynomial or exponential time!

- If we use an ***octal*** representation, this requires $n' = \lfloor log_8 \ k \rfloor + 1$ digits and

  $n/n' \sim log_2 k \ / \ log_8 k = log_2 k \ / \ (log_2 k \ / \ log_2 8\ ) = log_2 8 = 3$
- So, the binary and the octal only differ for a constant factor, whereas unary and all other bases differ for a non-constant factor (that tends to ∞ as k grows)

On representing the input (2)

- Let us consider a finite set Σ of cardinality at least 2
- An ***encoding*** of a set I (set of problem instances) is any (injective) mapping $e: I \longrightarrow \Sigma^*$
- Fixed an encoding e, an algorithm ***solves*** a problem in time $O(t(n))$ if, when it is provided a problem instance i with $|e(i)| = n$, it produces a solution in time $O(t(n))$.
- For some set I of problem instances, we say that two encodings e_1 and e_2 are ***polynomially related*** if there exist algorithms A_{12} and A_{21} such that there exist constants c and k s.t., for every $i \in I$:
  - A_{12} computes $e_2(i)$ from $e_1(i)$ in $O(|e_1(i)|^k)$, and
  - A_{21} computes $e_1(i)$ from $e_2(i)$ in $O(|e_2(i)|^c)$.

EXAMPLE:

- Binary and octal are polynomially related;
- Unary and binary are NOT polynomially related, since an n bit long binary number may need 2^n-1 ones to be represented in unary.

- An ***abstract decision problem*** Q is a mapping from an instance set I to $\{0,1\}$
- An encoding $e: I \longrightarrow \Sigma^*$ induces a related ***concrete decision problem***, denoted by $e(Q)$, s.t.:
  1. For every abstract problem instance $i \in I$, it holds that $Q(i) = e(Q)(e(i))$; and
  2. For every string $s \in \Sigma^* \setminus \text{range}(e)$ (e may not be surjective), we let $e(Q)(s) = 0$
- A decision problem is fully characterized by the language $\{e(i) \in \Sigma^* : i \in I \wedge Q(i) = 1\}$

On representing the input (3)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Prop.: Let Q be an abstract decision problem on an instance set I, and let e_1 and e_2 be polynomially related encodings on I. Then, $e_1(Q) \in \mathbf{P}$ if and only if $e_2(Q) \in \mathbf{P}$.

Proof (➔, the converse is the same)

- $e_1(Q) \in \mathbf{P}$ means that there exists A that, for all $i \in I$, $e_1(Q)(e_1(i)) = Q(i)$ in $O(n_1^k)$, for $n_1 = |e_1(i)|$
- e_1 and e_2 polynomially related implies existence of A_{21} that, for all $i \in I$, turns $e_2(i)$ into $e_1(i)$ in $O(n_2^c)$, for $n_2 = |e_2(i)|$
- Let us now consider the composition of A_{21} and A, that is $A(A_{21}(\text{-}))$. This algorithm:
  1. First turns the e_2-encoding of any instance into its corresponding e_1-encoding;
  2. Then, it calculates the solution of the concrete problem $e_1(Q)$ for the obtained e_1-encoded instance
  3. For any $i \in I$, this returns $Q(i)$ in $O(n_2^c) + O(\,(O(n_2^c))^k\,) = O(n_2^{ck}\,)$
  4. Since both c and k are constant, this entails that $e_2(Q) \in \mathbf{P}$.

Q.E.D.

From now on, we assume that notation $\langle \text{-} \rangle$ denotes:

- any encoding of an integer that is polynomially related to its binary representation; and
- any encoding of a finite set that is polynomially related to its encoding as a list of its encoded elements, enclosed in braces and separated by commas (with ASCII symbols used)

NP

DEFINITION

NP is the class of languages that are decidable in polynomial time on a nondeterministic Turing machine. In other words,

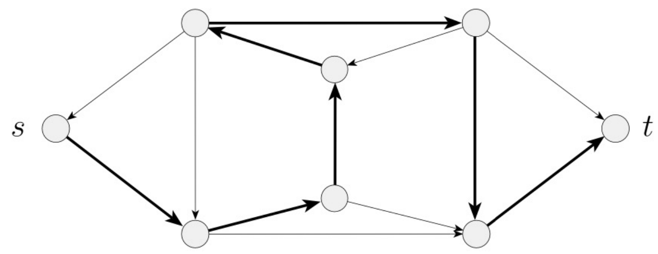
$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k).$$

- Like **P**, also **NP** is insensitive to the choice of the nondeterministic computational model because all such models are polynomially equivalent.
- When describing and analyzing **NP**-time algorithms, we follow the preceding conventions for **P**-time algorithms:
  - Each stage of a nondeterministic polynomial time algorithm must have an obvious implementation in nondeterministic polynomial time on a reasonable nondeterministic computational model.
  - We analyze the algorithm to show that every branch uses at most polynomially many stages.

An example of a NP problem

HAM-PATH: check whether a given directed graph G has an hamiltonian path (a path that touches its vertices exactly once) from s to t

For example:



On input $\langle G, s, t \rangle$, where G is a directed graph (V, E) with nodes s and t:

1. Write a list of m vertices $v_1,\ldots,v_m$, where $m = |V|$, and each v_i is nondeterministically chosen in V.
2. Check for repetitions in the list. If any are found, **reject**.
3. Check whether $s = v_1$ and $t = v_m$. If either fail, **reject**.
4. For each i between 1 and $m-1$, check whether (v_i, v_{i+1}) belongs to E. If any are not, **reject**; otherwise, **accept**.

Since each of these steps is polynomial in the size of G (the number of its vertices), we have that HAM-PATH is a **NP** problem.

NP and polynomial verifiability (1)

In the previous algorithm, the only step the requires nondeterminism is the first one

→ this «guesses» the path that should be hamiltonian

→ if turned to a deterministic algorithm, we should consider all possible guesses

→ these are all possible m-tuples of vertices from the a set with m elements

→ they are m^m (super-exponential!!)

By contrast, once provided with a guess, steps 2/3/4 (that are deterministic) verify whether this guess is a solution or not in polynomial time.

DEFINITION

A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

String c is called a ***certificate***, or ***proof***, of membership in A.

Observe that, for polynomial verifiers, the certificate has polynomial length (in the length of w) because that is all the verifier can access in its time bound.

NP and polynomial verifiability (2)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Thm.: A language is in **NP** if and only if it is polynomially verifiable.

Proof

(➔) Let L be in **NP** and show that it is polynomially verifiable via a verifier V.

Let N be the NTM for L; then, V:

On input $\langle w, c \rangle$, where w and c are strings:

Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step (as in the proof of Det vs NDetTM)

If this branch of N's computation accepts, **accept**; otherwise, **reject**.

(🡐) Let L be polynomially verifiable and show that it is decided by a polynomial time NTM N.

Let V be the polynomial time verifier for L, i.e. a TM that runs in time $O(n^k)$. Then, N:

On input w of length n:

Nondeterministically select a string c of length at most n^k

Run V on input $\langle w, c \rangle$

If V accepts, **accept**; otherwise, **reject**.

Q.E.D.

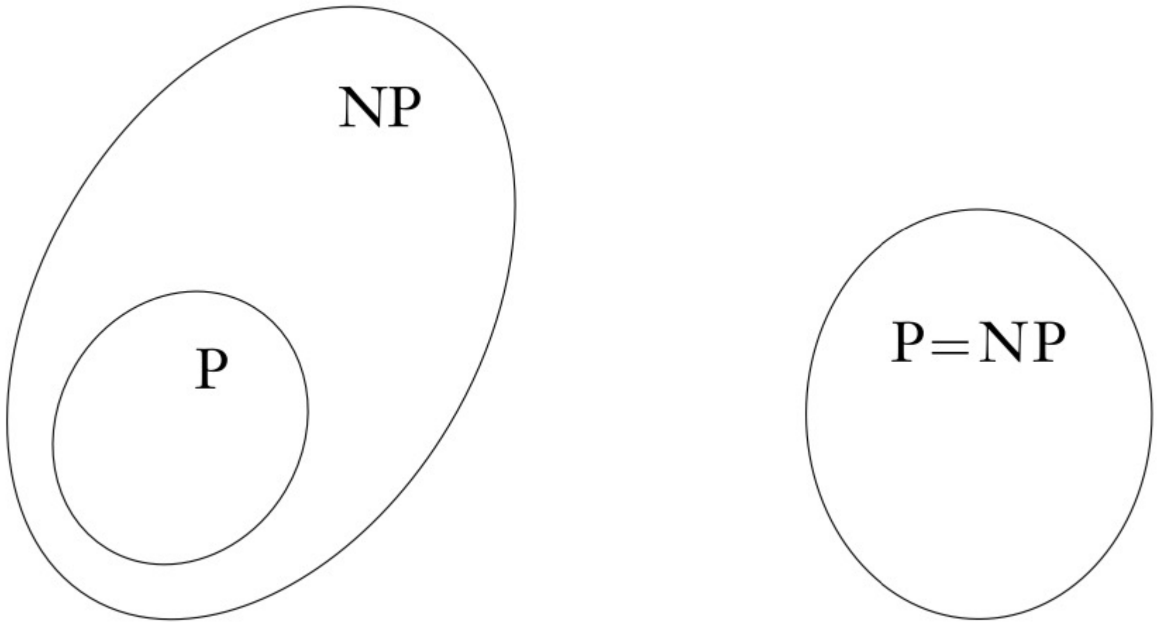
One million dollars problem: P vs NP

- **P** is the class of languages for which membership can be *decided* quickly.
- **NP** is the class of languages for which membership can be *verified* quickly.
- As HAM-PATH testifies, the power of polynomial verifiability seems to be much greater than that of polynomial decidability.
- However, **P** and **NP** could be equal:

  By today, we're unable to *prove* the existence of a single language in **NP** that is not in **P**
- If these classes were equal, any polynomially verifiable problem would be polynomially decidable
  - Most researchers believe that this is NOT the case because people have invested enormous effort to find polynomial time algorithms for many problems in **NP**, without success.
  - Researchers also have tried proving that the classes are unequal, but that would entail showing that the fastest algorithm for all these problems is brute-force search.
  - Indeed, all **NP** problems have an exponential number of certificates
  - So, by defining $\mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k})$, we trivially have that **NP** $\subseteq$ EXPTIME

One million dollars problem: P vs NP

Hence, there are two possibilties:



Today, researchers believe that the left-most inclusion holds

→ this belief is strenghtened by the existence of NP-complete problems, that we'll introduce in the next class

The coNP class

Def.: given a class **C**, we define **coC** = $\{L \mid \overline{L} = \Sigma^* \setminus L \in \mathbf{C}\}$

P = **coP** :

→ if a language L is decided by a poly-time DTM M, you can polynomially decide $\overline{L}$ by accepting when M rejects, and by rejecting when M accepts. problems in coP are still solved by a DTM polynomially

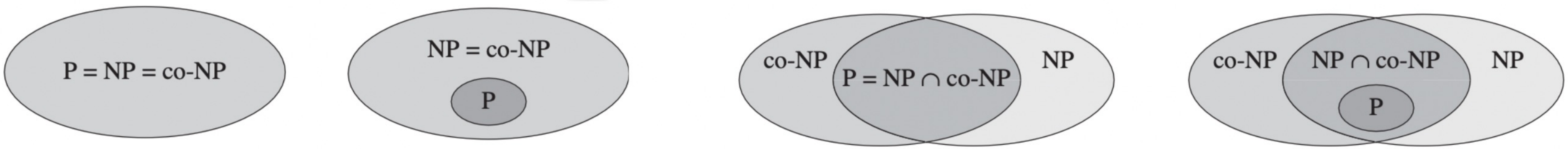
For **NP** the situation is more delicate:

→ to accept an input for $\overline{L}$, you have to check that *all* paths of the decider for L lead to rejection

→ e.g., verifying that a graph is NOT hamiltonian requires to check that all its (exponentially many) paths touch at least one vertex at least twice

→ indeed, we don't know whether **NP** = **coNP** or not

Possible scenarios:



The last one is the most widely believed by computer scientists today.