Time complexity theory: the class NP
Meaning of NP

NP is an abbreviation for *Nondeterministic Polynomial* time. It refers to languages decided in polynomial time by *nondeterministic TMs*

Let’s recall that a NDTM accepts if at least one of its computing branches reaches the accept state.
Thus, NP expresses the fact that the (all the) accepting computing branch(es) reach(es) the accept state in polynomial time.

The time invested in guessing for an accepting compute branch is not taken into account. The analysis assumes that an accepting branch has been already identified and attempts to estimate the rate of growth of a computation on this particular branch.

Thus, ultimately, the analysis reduces to verifying the rate of growth of a method for verifying a solution
But, what does verifying mean?

**Definition:** A verifier of a language $L$ is a decider $V$ so that

$$L = \{ w \in A^* : (\exists c) \land V \_accepts \_ < w, c > \}$$

Thus, a verifier is a decider $V$ that allows for expressing membership in $L$ as the acceptance of a pair of strings, one representing a problem’s instance and the other, its “solution”
An illustration

Let $Pol[x]$ be the set of all polynomials of degree 2 on a real variable $x$. Let’s define

$L = \{ p \in Pol[x] : p \text{ has an integral root between 0 and 10} \}$

In this case, we may define for each pair $(p,c)$

$p \in Pol[x], 0 \leq c \leq 10$

$V=\text{“On input } <p,c>\text{:} $

1. Evaluate $p$ in $c$
2. If $p(c) = 0$ ACCEPT. Otherwise, REJECT$

And rewrite,

$L = \{ p \in Pol[x] : (\exists c \in \{0,1,...,10\}) V \text{ accepts } <p,c> \}$
The NP class of languages

**Definition:** The class NP is formed by all decidable languages that can be verified in polynomial time.

Clearly, $P \subseteq NP$ since the TM that decides a language in $P$ is its own polynomial verifier.

(Express:

$L(M) = \{w \in A^* : (\exists \lambda) M \_\text{accepts} \_ < w, \lambda >\}$)
A classical example of NP language

Any well-formed logical expression that combines Boolean variables and conjunctions, disjunctions or negations is said to be a Boolean formula.

A Boolean formula is said to be *satisfiable* if there exists a valuation (this is an assignment of 0’s or 1’s to the variables) making the formula true.
The satisfiability problem (SAT)

General Instance: A Boolean formula
Question: Is there a valuation making it true?

**Theorem 27:** SAT is decidable.

*Proof:* Consider, the following decider

\[ M=:\text{"On input } w:\text{"} \]

1. Generate a string \( c \) of 0’s, \(|c| = |w|\). Evaluate \( w \) on \( c \). If \( w=1 \), ACCEPT.
2. Else, while \( w = 0 \)
   
   Generate lexicographically, the next string \( c \) in \( \{0,1\}^* \), \(|c| = |w|\).
   
   Evaluate \( w \) on \( c \). If \( w=1 \), ACCEPT.
3. REJECT.”
Classifying SAT

The previous decider has time complexity $O(2^{|w|})$, since:
Step 1 is $O(|w|)$
Step 2 is $O(2^{|w|})$  However, there is still some hope:

**Theorem 28:** SAT is NP

**Proof:** Consider the following decider

V=“On input $<w,c>$, where $w$ is a string representing a Boolean formula and $c$ a string in $\{0,1\}^*$, $|w|=|c|$:
1. Evaluate $w$ in $c$
2. If $w = 1$, ACCEPT.
3. Else, REJECT.”

Clearly $V$ is $O(|w|)$, and therefore, polynomial. Now,

$SAT = \{w : (\exists c \in \{0,1\}^*) , |c| = |w| \land V \text{ accepts } <w,c>\}$
Showing that a language is NP

The previous analysis is logically founded proof that SAT is NP

In general, showing that a language $L$ is NP amounts to proving that:

1. $L$ is decidable
2. Each solution can be verified in polynomial time
How far is P from NP?

Is $P = NP$?, this is, is there a polynomial time solution for any problem in NP, or:

Does NP constitutes a completely separated complexity class?

We don’t know! and

This is a long standing open problem

There is a great deal of “practical evidence” suggesting that the answer is no. However, no logical proof has been produced so far.
Currently, if you are asked to design an efficient (i.e. polynomial) method for solving the SAT problem and you fail to find the algorithm, you may report either

1. “I cannot find an efficient algorithm, I guess I am just too dumb”
2. “I cannot find an efficient algorithm, because no such algorithm is possible”
3. “I cannot find an efficient algorithm, but neither can all these famous people”

(taken from Garey & Johnson-”Computers and Intractability”)
You’d better be professional!!!

A somewhat more formal but essentially equivalent report would be:

• “SAT is NP.” (Provide a formal proof)

• “SAT admits a polynomial time solution if P=NP”.

The idea of NP-completeness

Is one of the most significant advances in the search for an answer to the P=NP question.

It implies, that finding a polynomial time decider for anyone of the NP-complete languages renders a positive answer to the P=NP question.

The core concept behind it is the notion of polynomial time reducibility.
Polynomial time reducibility

A language $L$ is said to be polynomial time reducible to a language $U$ if $L$ is reducible to $U$ and the map reduction is computable in polynomial time.

Polynomial time reducibility is:

1. **Reflexive** (the identity is a polynomial time reduction)

2. **Transitive** (the composition of two polynomial time reductions is a polynomial time reduction)
An important theoretical result

**Theorem 29:** If $L$ is polynomial time reducible to $U$ and $U$ is in $P$, then $L$ is in $P$.

**Proof:** Let $M$ be a polynomial time decider for $U$ and $f$ a polynomial time reduction map from $L$ to $U$. Construct:

$N$ = “On input $w$:

1. Compute $f(w)$.
2. Run $M$ on $f(w)$ and output whatever $M$ outputs.”

Clearly, $N$ decides $L$ in polynomial time.
Definition of NP-complete languages

**Definition**: a language is said to be NP-complete if:

1. The language is NP
2. Every language in NP is polynomial time reducible to it

*Is there at least one such language?*

The answer is **yes** (Cook-Levin, 1970’s)

Indeed, several NP-complete languages have been discovered since the 70’s.
An interesting variant of the SAT problem

**Definition:** We define:

1. *Literal:* as any symbol representing a Boolean variable or its negation
2. *Clause:* as a finite set of literals connected with the logical OR operation
3. *Boolean formula in conjunctive normal form (cnf):* as a finite set of clauses connected with the logical AND operation.
4. *K-cnf:* as a Boolean formula in cnf where each of the clauses involve *at most* *K* literals
1SAT, 2SAT, 3SAT,…,kSAT, …

**Definition:** $k$SAT=$\{w: w$ is a $k$-cnf satisfiable Boolean formula$\}$

What is remarkable about the $k$SAT collection is the so-called “2SAT frontier”, this is:

1. $1$SAT and $2$SAT are P, but (surprisingly enough)…
2. For all $k>2$, $k$SAT is NP, meaning $k$SAT is P if P=NP

We’ll end this lecture by showing that $2$SAT is indeed in P
An instance of 2SAT

Consider the Boolean formula:

\[ \phi = (x_1 \lor x_2) \land (x_3 \lor \neg x_2) \land x_1 \land (\neg x_1 \lor \neg x_2) \land (x_3 \lor x_4) \]
\[ \land (\neg x_3 \lor x_5) \land (\neg x_4 \lor \neg x_5) \land (x_4 \lor \neg x_3) \]

This (rather small) formula has 5 literals, and eight 2-clauses.

Is \( \phi \in 2SAT \) ?. Here is a method for answering this kind of questions:
Trying valuations efficiently

Rational:
1. Single literals, this is, 1 clauses, must be set to 1 (i.e. true).
2. If a clause has a literal set to 1, the clause value is 1.
3. A literal with a 0 value does not contribute in making the clause to evaluate 1.

Thus, given:

\[ \phi = (x_1 \lor x_2) \land (x_3 \lor \neg x_2) \land x_1 \land (\neg x_1 \lor \neg x_2) \land (x_3 \lor x_4) \land (\neg x_3 \lor x_5) \land (\neg x_4 \lor \neg x_5) \land (x_4 \lor \neg x_3) \]
Trying valuations efficiently (cont.)

By setting \( x_1 = 1 \); we may reduce (purge) the original Boolean formula to:

\[
(x_3 \lor \neg x_2) \land (\neg x_2) \land (x_3 \lor x_4) \\
\land (\neg x_3 \lor x_5) \land (\neg x_4 \lor \neg x_5) \land (x_4 \lor \neg x_3)
\]

By setting the new single literal to 1, this is, \( \neg x_2 = 1 \lor x_2 = 0 \), we purge the remaining formula to get:

\[
(x_3 \lor x_4) \land (\neg x_3 \lor x_5) \land (\neg x_4 \lor \neg x_5) \land (x_4 \lor \neg x_3)
\]
Trying valuations efficiently (cont.)

No single literals remain at this point. The trial is now conducting by selecting a literal, evaluating it to 1, and purging, and if the purge fails, evaluate it to 0 and try again. Two failures means that $\phi \notin 2SAT$ while a success means $\phi \in 2SAT$

In our case,
Trying valuations efficiently (cont.)

A first trial: \[ x_3 = 1 \land \text{purge} \]
\[ x_5 \land (\neg x_4 \lor \neg x_5) \land x_4; \]
\[ x_4 = 1 \land \text{purge} \]
\[ x_5 \land \neg x_5 = 0 \quad \text{Failure!} \]

Second trial: \[ x_3 = 0 \land \text{purge} \]
\[ x_4 \land (\neg x_4 \lor \neg x_5); \]
\[ x_4 = 1 \land \text{purge} \]
\[ \neg x_5 = 1 \quad \text{Success!} \]
Generalizing the “purge” method

**Theorem 30:** 2SAT is P

**Proof:** Consider the following Turing machine:

P := “On input \(<\phi>\):

1. While there are single literals,
   1.1 Select a single literal and evaluate it to 1.
   1.2 If the negation of the selected literal is also a single literal, REJECT. Else, purge.
   1.3 If the purged string is null, ACCEPT.

2. If the string has no single literals,
   2.1 For each literal in the string
       Select a literal, evaluate to 1 and purge. If the purged string evaluates to 1, ACCEPT. Else, evaluate to 0, and purge. If the purged string evaluates to 1 ACCEPT.

3. REJECT”
Generalizing the “purge” method (cont.)

Time complexity analysis:
Step 1 $O(|w|)$
Step 2 $O(2|w|^2)$
Therefore, $TimeC(P) = O(n^2)$ and thus, 2SAT is a member of P.

QED