NP and NP-completeness


Ahto Buldas

Ahto.Buldas@ut.ee
There is often a big difference between:
• solving a problem from scratch, and
• verifying a given solution.

Crossword puzzles is one example.

We define the complexity class $\text{NP}$ that aims to capture the set of problems whose solutions can be efficiently verified.
The class $\textbf{NP}$

Class of problems having efficiently verifiable solutions.

A decision problem/language is in $\textbf{NP}$ if given an input $x$, we can easily verify that $x$ is a YES instance of the problem ($x$ is in the language) if we are given the polynomial-size solution for $x$, that certifies this fact.

**Def:** A language $L \subseteq \{0, 1\}^*$ is in $\textbf{NP}$ if there exists a polynomial $p$ and a polynomial-time Turing machine $M$ such that for every $x \in \{0, 1\}^n$:

$$x \in L \iff \exists u \in \{0, 1\}^{p(|x|)} : M(x, u) = 1.$$  

If $x \in L$ and $u \in \{0, 1\}^{p(|x|)}$ satisfy $M(x, u) = 1$ then we call $u$ a **certificate** (or a **witness**) for $x$ (with respect to the language $L$ and machine $M$).
Problems in \textbf{NP}

\textit{Independent set}: Given a graph $G$ and a number $k$, decide if there is a $k$-size independent subset of vertices in $G$. The certificate is the list of $k$ vertices forming an independent set.

\textit{Traveling salesman}: Given a set of $n$ nodes, $\binom{n}{2}$ numbers $d_{ij}$ denoting the distances between all pairs of nodes, and a number $k$, decide if there is a closed circuit (i.e., a “salesman tour”) that visits every node exactly once and has total length at most $k$. The certificate is the sequence of nodes in the tour.

\textit{Subset sum}: Given a list of $n$ numbers $A_1, \ldots, A_n$ and a number $T$, decide if there is a subset of the numbers that sums up to $T$. The certificate is the list of members in this subset.
Problems in \textbf{NP}

\textit{Linear programming}: Given a list of \( m \) linear inequalities with rational coefficients over \( n \) variables \( u_1, \ldots, u_n \) (in the form \( a_1 u_1 + a_2 u_2 + \ldots + a_n u_n \leq b \) for some coefficients \( a_1, \ldots, a_n, b \)), decide if there is an assignment of rational numbers to the variables \( u_1, \ldots, u_n \) that satisfies all the inequalities. The certificate is the assignment.

\textit{Integer programming}: Given a list of \( m \) linear inequalities with rational coefficients over \( n \) variables \( u_1, \ldots, u_m \), find out if there is an assignment of integer numbers to \( u_1, \ldots, u_n \) satisfying the inequalities. The certificate is the assignment.

\textit{Graph isomorphism}: Given two \( n \times n \) adjacency matrices \( M_1 \) and \( M_2 \), decide if \( M_1 \) and \( M_2 \) define the same graph, up to renaming of vertices. The certificate is the permutation \( \pi : [n] \to [n] \), such that \( M_2 \) is equal to \( M_1 \) after reordering \( M_1 \)'s indices according to \( \pi \).
Problems in \( \text{NP} \)

**Composite numbers**: Given a number \( N \) decide if \( N \) is a composite (i.e., non-prime) number. The certificate is the factorization of \( N \).

**Factoring**: Given three numbers \( N, L \) and \( U \) decide if \( N \) has a factor \( M \) in the interval \([L, U]\). The certificate is the factor \( M \).

**Connectivity**: Given a graph \( G \) and two vertices \( s, t \) in \( G \), decide if \( s \) is connected to \( t \) in \( G \). The certificate is the path from \( s \) to \( t \).
Relation between NP and P

We have the following trivial relationships between NP and the classes P and DTIME(T(n)):

Claim 2.3: $P \subseteq NP \subseteq \bigcup_{c > 1} \text{DTIME}(2^{n^c})$.

Proof: Suppose $L \in P$ is decided in poly-time by $M$, i.e.

$$x \in L \iff M(x) = 1 \iff \exists u \in \{0, 1\}^0 M(x, u) = 1.$$ 

Hence, $L \in NP$.

If $L \in NP$ and $M$ and, $p(n)$ are as in the definition of NP, then we can decide $L$ in time $2^{O(p(n))}$ by enumerating all possible $u$ and using $M$ to check whether $u$ is a valid certificate for the input $x$. The machine accepts iff such a $u$ is ever found. Since $p(n) = O(n^c)$ for some $c > 1$, then this machine runs in $2^{O(n^c)}$ time.
Non-deterministic Turing machines

The class $\mathsf{NP}$ can also be defined using non-deterministic Turing machines (NDTMs). The only differences between an NDTM and a TM are:

- NDTM has two transition functions $\delta_0$ and $\delta_1$.
- NDTM has a special state we denote by $q_{\text{accept}}$.
- NDTM makes (at each step) an arbitrary choice as to which of its two transition functions to apply.

We say that a NDTM $N$ outputs 1 on a given input $x$ if there is some sequence of these non-deterministic choices that would make $N$ reach $q_{\text{accept}}$ on input $x$. Otherwise, if every sequence of choices makes $N$ halt without reaching $q_{\text{accept}}$, then we say that $N$ outputs 0.

We say that $N$ runs in $T(n)$ time if for every $x \in \{0, 1\}^n$ and every sequence of choices, $M(x)$ reaches either the halting state or $q_{\text{accept}}$ within $T(|x|)$ steps.
**Alternative definition of NP**

**Def**: For every function $T : \mathbb{N} \rightarrow \mathbb{N}$ and $L \subseteq \{0, 1\}^*$, we say that $L \in \text{NTIME}(T(n))$ if there is a constant $c > 0$ and a $cT(n)$-time NDTM $N$ such that for every $x \in \{0, 1\}^n$: $x \in L \iff N(x) = 1$.

**Theorem 2.6**: $\text{NP} = \bigcup_{c \in \mathbb{N}} \text{NTIME}(n^c)$.

**Proof idea**: If $L$ is decided by a $p(n)$-time NDTM $N$, then the sequence of choices that lead to $q_{\text{accept}}$ can be used as a certificate of size $p(n)$.

If $L \in \text{NP}$ (with machine $M$ and cert-size $p(n)$) then we can construct a NDTM $N$ that given $x \in \{0, 1\}^n$ as input first makes $p(n)$ non-deterministic choices to write down $u \in \{0, 1\}^{p(n)}$; after that, $N$ computes $M(x, u)$ and finishes in state $q_{\text{accept}}$ if $M(x, u) = 1$, otherwise $N$ just halts.
Def 2.7: A language $A \subseteq \{0, 1\}^*$ is polynomial-time Karp reducible to a language $B \subseteq \{0, 1\}^*$ denoted by $A \leq_p B$ if there is a poly-time computable function $f : \{0, 1\}^* \to \{0, 1\}^*$ such that for every $x \in \{0, 1\}^*$:

$x \in A \iff f(x) \in B$.

We say that $B$ is NP-hard if $A \leq_p B$ for every $A \in \text{NP}$. We say that $B$ is NP-complete if $B$ is NP-hard and $B \in \text{NP}$.
Properties of $\leq_p$

**Theorem 2.8:**

- $A \leq_p B, B \leq_p C \Rightarrow A \leq_p C$.
- If $A$ is $\text{NP}$-hard and $A \in \text{P}$ then $\text{P} = \text{NP}$.
- If $A$ is $\text{NP}$-complete, then $A \in \text{P} \iff \text{P} = \text{NP}$.

**Proof ideas:** The main observation is that if $p, q$ are two functions that have polynomial growth then their composition $p(q(n))$ also has polynomial growth.

If $f_1$ is a polynomial-time reduction from $A$ to $B$ and $f_2$ is a reduction from $B$ to $C$ then the mapping $x \mapsto f_2(f_1(x))$ is a polynomial-time reduction from $A$ to $C$ since $f_2(f_1(x))$ takes polynomial time to compute given $x$ and $f_2(f_1(x)) \in C \iff x \in A$. 

11
Do NP-complete languages exist?

It may not be clear that \( \text{NP} \) should possess a language that is as hard as any other language in the class. However, this does turn out to be the case:

**Theorem 2.9**: The following language is \( \text{NP} \)-complete:

\[
\text{TMSAT} = \{ \langle \alpha, x, 1^n, 1^t \rangle : \exists u \in \{0, 1\}^n : 1 \leftarrow M_\alpha(x, u) \text{ within } t \text{ steps.} \}
\]

where \( M_\alpha \) denotes the TM represented by the string \( \alpha \).

The Cook-Levin Theorem shows that there are more natural examples of \( \text{NP} \)-complete problems.
Boolean formulae and the CNF form.

Several NP-complete problems come from propositional logic.

A *Boolean formula* over the variables $u_1, \ldots, u_n$ consists of the variables and the logical operators AND ($\land$), NOT ($\neg$) and OR ($\lor$). For example,

$$(a \land b) \lor (a \land c) \lor (b \land c)$$

is a Boolean formula that is True iff the majority of $a$, $b$, $c$ are True.

If $\varphi$ is a Boolean formula over variables $u_1, \ldots, u_n$, and $z \in \{0, 1\}^n$, then $\varphi(z)$ denotes the value of $\varphi$ when the variables of $\varphi$ are assigned the values $z$ ($1$ means True and $0$ means False).

A formula $\varphi$ is *satisfiable* if there there exists some assignment $z$ such that $\varphi(z)$ is True. Otherwise, we say that $\varphi$ is unsatisfiable.
A Boolean formula over variables $u_1, \ldots, u_n$ is in CNF form (Conjunctive Normal Form) if it is an AND of ORs of variables or their negations. For example, the following is a 3CNF formula:

$$(u_1 \lor \overline{u}_2 \lor u_3) \land (u_2 \lor \overline{u}_3 \lor u_4) \land (\overline{u}_1 \lor u_3 \lor \overline{u}_4).$$

where $\overline{u}$ denotes the negation of $u$.

More generally, a CNF formula has the form

$$\bigwedge_i \left( \bigvee_j v_{ij} \right),$$

where each $v_{ij}$ is either a variable $u_k$ or to its negation $\overline{u}_k$. The terms $v_{ij}$ are called the literals of the formula and the terms $(\bigvee_j v_{ij})$ are called its clauses. A $k$-CNF is a CNF formula in which all clauses contain at most $k$ literals.
The Cook-Levin Theorem

**Theorem 2.10:** Let SAT be the language of all satisfiable CNF formulae and 3SAT be the language of all satisfiable 3CNF formulae. Then,

- SAT is \textit{NP}-complete.
- 3SAT is \textit{NP}-complete.

Both languages are clearly in \textit{NP}. Thus we only need to prove that they are \textit{NP}-hard.

We do so by first proving that SAT is \textit{NP}-hard and then showing that SAT is polynomial-time Karp reducible to 3SAT. This implies that 3SAT is \textit{NP}-hard by the transitivity of polynomial-time reductions.
NP-hardness of SAT

Thus, the following lemma is the key to the proof.

Lemma 2.12: SAT is NP-hard.

To prove this we have to show how to reduce every NP language $L$ to SAT, in other words give a polynomial-time transformation that turns any $x \in \{0, 1\}^*$ into a CNF formula $\varphi_x$ such that:

$$x \in L \iff \varphi_x \text{ is satisfiable}.$$ 

Since we know nothing about the language $L$ except that it is in NP, this reduction has to rely just upon the definition of computation, and express it in some way using a Boolean formula.
Warmup: Expressiveness of boolean formulae

Example 2.13: The formula $(a \lor b) \land (\overline{a} \lor b)$ is in CNF form. It is satisfied by only those values of $a$, $b$ that are equal. Thus, the formula

$$(x_1 \lor \overline{y_1}) \land (\overline{x}_1 \lor y_1) \land \ldots \land (x_n \lor \overline{y}_n) \land (\overline{x}_n \lor y_n)$$

is True if and only if the strings $x, y \in \{0, 1\}^n$ are equal to one another. Thus, though $\equiv$ is not a standard boolean operator like $\lor$ or $\land$, we will use it as a convenient shorthand since the formula $\phi_1 \equiv \phi_2$ is equivalent to (in other words, has the same satisfying assignments as) $(\phi_1 \lor \overline{\phi}_2) \land (\overline{\phi}_1 \lor \phi_2)$.

CNF formulae of sufficient size can express every Boolean condition ...
Warmup: Expressiveness of boolean formulae

Claim 2.14: For every Boolean function $f : \{0, 1\}^\ell \rightarrow \{0, 1\}$ there is an $\ell$-variable CNF formula $\varphi$ of size $\ell 2^\ell$ such that $\varphi(u) = f(u)$ for every $u \in \{0, 1\}^\ell$, where the size of a CNF formula is defined to be the number of $\land / \lor$ symbols it contains.

Proof Sketch: For every $v \in \{0, 1\}^\ell$, it is not hard to see that there exists a clause $C_v$ such that $C_v(v) = 0$ and $C_v(u) = 1$ for every $u \neq v$. For example, if $v = \langle 1, 1, 0, 1 \rangle$, the corresponding clause is $(\overline{u_1} \lor \overline{u_2} \lor u_3 \lor \overline{u_4})$.

We let $\varphi$ be the AND of all the clauses $C_v$ for $v$ such that $f(v) = 0$ (note that $\varphi$ is indeed of size at most $\ell 2^\ell$). Then for every $u$ such that $f(u) = 0$ it holds that $C_u(u) = 0$ and hence $\varphi(u)$ is also equal to 0. On the other hand, if $f(u) = 1$ then $C_v(u) = 1$ for every $v$ such that $f(v) = 0$ and hence $\varphi(u) = 1$. We get that for every $u$, $\varphi(u) = f(u)$. 

Proof of Cook-Levin: First idea

Let $L \in \text{NP}$ and let $M$ be the poly-time TM such that for every $x \in \{0, 1\}^*$:
\[ x \in L \iff \exists u \in \{0, 1\}^{p(|x|)} M(x, u) = 1, \]
where $p$ is a polynomial. We show that $L$ is reducible to SAT by transforming in poly-time every string $x \in \{0, 1\}^*$ into a CNF formula $\varphi_x$ such that $x \in L$ iff $\varphi_x$ is satisfiable.

How can we construct such a formula $\varphi_x$? By Claim 2.14, the function that maps $u \in \{0, 1\}^{p(|x|)}$ to $M(x, u)$ can be expressed as a CNF formula $\psi_x$ (i.e., $\psi_x(u) = M(x, u)$ for every $u \in \{0, 1\}^{p(|x|)}$). Thus a string $u$ such that $M(x, u) = 1$ exists iff $\psi_x$ is satisfiable.

But this is not useful! The size of $\psi_x$ is as large as $p(|x|)2^{p(|x|)}$.

To get a smaller formula we use the fact that $M$ runs in polynomial time, and that each basic step of a Turing machine is highly local (in the sense that it examines and changes only a few bits of the machines tapes).
Oblivious Turing Machines

We will make the following simplifying assumptions about $M$:

- $M$ has **two tapes**: an input tape and a work/output tape; and
- $M$ is **oblivious**, i.e. its head movement does not depend on the contents of its input tape. In particular, this means that $M$'s computation takes the same time for all inputs of size $n$ and for each time step $i$ the location of $M$'s heads at the $i$-th step depends only on $i$ and $M$'s input length. This is without loss of generality because:

**Exercise**: Prove that for every $T(n)$-time $M$ there exists a two-tape oblivious $\tilde{M}$ computing the same function in $O(T^2(n))$ time.

Thus in particular, if $L$ is in NP then there exists a two-tape oblivious polynomial-time $M$ and a polynomial $p$ such that:

$$x \in L \iff \exists u \in \{0, 1\}^{p(|x|)} : M(x, u) = 1.$$
Snapshots

As $M$ is oblivious, we can define functions:

- $\text{inputpos}(i)$ – location of the input tape head at the $i$-th step
- $\text{prev}(i)$ – the last step before $i$ that $M$ visited the same location on its work tape. If there is no such step, then define $\text{prev}(i) = 0$

These functions can be computed in poly-time by simulating $M$ on, say, the all-zeros input.

**Def:** The **snapshot** $z_i$ of $M(y)$ at step $i$ is the triple $\langle a, b, q \rangle$, such that $a$ and $b$ are the symbols read by $M$'s heads from the two tapes and $q$ is the state $M$ is in at the $i$-th step.

$z_0 = \langle y_0, \square, q_{\text{start}} \rangle$.

$z_i = \langle y_{\text{inputpos}(i)}, \delta_w(z_{\text{prev}(i)}), \delta_q(z_{i-1}) \rangle$, if $i > 0$. 
Describing $M(y)$ with a Snapshot Sequence

The size of an encoded snapshot is $|z| = 1 + \log_2 |\Gamma| + \log_2 |Q|$. 

For any $M$, there is a Boolean function $F_M$ with $3 + 2 \log_2 |\Gamma| + 2 \log_2 |Q|$ input variables, such that for every $i$ and for every input $y$:

$$z_i = F_M(y_{\text{inputpos}}(i), z_{\text{prev}}(i), z_{i-1}) \quad (1)$$

There is a Boolean function $G_M$ with $4 + 3 \log_2 |\Gamma| + 3 \log_2 |Q|$ input variables that defines the condition (1):

$$G_M(y_{\text{inputpos}}(i), z_{\text{prev}}(i), z_{i-1}, z_i) = 1$$

$$\Leftrightarrow z_i = F(y_{\text{inputpos}}(i), z_{\text{prev}}(i), z_{i-1}) \quad .$$

$M$ is uniquely determined by $G_M$, because $F$ contains both components $\delta_w, \delta_q$ of $M$’s transition function.
Proof of Cook-Levin: SAT is NP-hard

For any oblivious $M$ with running time $t(n)$ and $x \in \{0, 1\}^n$, the condition $M(x, u) = 1$ (where $u \in \{0, 1\}^{p(|x|)}$) can be uniquely described with an input string $y \in \{0, 1\}^{n+p(n)}$ sequence of snapshots $z_0, z_1, z_2, \ldots, z_{t(n)}$, such that:

- First $n$ bits of $y$ encode $x$. Can be represented by a DNF with size $4n$.
- $z_0 = \langle y_0, \square, q_{\text{start}} \rangle$. Can be trivially represented by a DNF with size $c2^c$, where $c$ is the length of a snapshot.
- For every $i \in \{1, \ldots, t(n)\}$: $G_M \left( y_{\text{inputpos}}(i), z_{\text{prev}}(i), z_{i-1}, z_i \right) = 1$. Can be trivially represented by a DNF with size $t(n) \cdot (3c + 1) \cdot 2^{3c+1}$.
- $z_{t(n)} = \langle *, *, q_{\text{accept}} \rangle$. Can be represented by a DNF with size $c2^c$.

Hence, for every $x$, we have an efficiently computable Boolean formula $\varphi_x(y, z_0, \ldots, z_{t(n)})$, that is satisfiable if and only if there is $u \in \{0, 1\}^{|x| + p(|x|)}$, such that $M(x, u) = 1$. 


Proof of Cook-Levin: Reducing \textsc{SAT} to \textsc{3SAT}

We will map a CNF formula \( \varphi \) into a 3CNF formula \( \psi \), which is satisfiable iff \( \varphi \) is. We demonstrate first the case that \( \varphi \) is a 4CNF.

Let \( C \) be a clause of \( \varphi \), say \( C = u_1 \lor \overline{u}_2 \lor \overline{u}_3 \lor u_4 \). We add a new variable \( z \) to the \( \varphi \) and replace \( C \) with the pair of clauses \( C_1 = u_1 \lor \overline{u}_2 \lor z \) and \( C_2 = \overline{u}_3 \lor u_4 \lor \overline{z} \). Clearly, if \( u_1 \lor \overline{u}_2 \lor \overline{u}_3 \lor u_4 \) is true then there is an assignment to \( z \) that satisfies both \( u_1 \lor \overline{u}_2 \lor z \) and \( \overline{u}_3 \lor u_4 \lor \overline{z} \) and vice versa: if \( C \) is false then no matter what value we assign to \( z \) either \( C_1 \) or \( C_2 \) will be false.

The same idea can be applied to a general clause of size 4, and in fact can be used to change every clause \( C \) of size \( k \) (for \( k > 3 \)) into an equivalent pair of clauses \( C_1 \) of size \( k - 1 \) and \( C_2 \) of size 3 that depend on the \( k \) variables of \( C \) and an additional auxiliary variable \( z \).
The web of reductions

Cook and Levin had to show how every NP-language can be reduced to SAT.

To prove the NP-completeness of any other language $L$, we do not need to work as hard: it suffices to reduce SAT or 3SAT to $L$. Once we know that $L$ is NP-complete we can show that an NP-language $L_0$ is in fact NP-complete by reducing $L$ to $L_0$.

This approach has been used to build a web of reductions and show that thousands of interesting languages are in fact NP-complete.
**INDSET is NP-complete**

**Proof**: Since INDSET is clearly in \( \text{NP} \), we only need to show that it is \( \text{NP} \)-hard, which we do by reducing 3SAT to INDSET. Let \( \varphi \) be a 3CNF formula on \( n \) variables with \( m \) clauses. We define a graph \( G \) of \( 7m \) vertices as follows: we associate a cluster of 7 vertices in \( G \) with each clause of \( \varphi \).

The vertices in cluster associated with a clause \( C \) correspond to the 7 possible partial assignments to the three variables \( C \) depends on (they are *partial assignments*, since they only give values for some of the variables).

If \( C \) is \( \overline{u}_2 \lor \overline{u}_5 \lor \overline{u}_7 \) then the 7 vertices in \( C \) correspond to all partial assignments of the form \( u_1 = a, u_2 = b, u_3 = c \) for a binary vector \( \langle a, b, c \rangle \neq \langle 1, 1, 1 \rangle \). (If \( C \) depends on less than three variables then we...
repeat one of the partial assignments and so some of the 7 vertices will correspond to the same assignment.)

We put an edge between two vertices of $G$ if they correspond to inconsistent partial assignments. Two partial assignments are consistent if they give the same value to all the variables they share. For example, the assignment $u_1 = 1, u_2 = 0, u_3 = 0$ is inconsistent with the assignment $u_3 = 1, u_5 = 0, u_7 = 1$ because they share a variable ($u_3$) to which they give a different value.

In addition, we put edges between every two vertices that are in the same cluster.

Clearly transforming $\varphi$ into $G$ can be done in poly-time.
We claim that $\varphi$ is satisfiable iff $G$ has an independent set of size $m$. Indeed, suppose that $\varphi$ has a satisfying assignment $u$. Define a set $S$ of $m$ vertices as follows: for every clause $C$ of $\varphi$ put in $S$ the vertex in the cluster associated with $C$ that corresponds to the restriction of $u$ to the variables $C$ depends on. Because we only choose vertices that correspond to restrictions of the assignment $u$, no two vertices of $S$ correspond to inconsistent assignments and hence $S$ is an independent set of size $m$.

On the other hand, if $G$ has an independent set $S$ of size $m$ then define a satisfying assignment $u$ as follows: for every $i \in \{1, \ldots, n\}$, if there is a vertex in $S$ whose partial assignment gives a value $a$ to $u_i$, then set $u_i = a$; otherwise set $u_i = 0$. This is well defined because $S$ is an independent set, and hence each variable $u_i$ can get at most a single value by assignments of the vertices in $S$. On the other hand, $S$ can contain at most one vertex in each cluster, and hence there is an element of $S$ in every one of the $m$ clusters. Thus, by the definition $u$, it satisfies all $\varphi$s clauses.
**coNP**

*Def*: $\text{coNP} = \{L \subseteq \{0, 1\}^*: \overline{L} \in \text{NP}\}$.

Hence, $\overline{\text{SAT}} \in \text{coNP}$.

*Alternative def*: $L \in \text{coNP}$ if there exists a polynomial $p$ and a polynomial-time Turing machine $M$ such that for every $x \in \{0, 1\}^*$:

$$x \in L \iff \forall u \in \{0, 1\}^p(|x|): M(x, u) = 1.$$

TAUTOLOGY is coNP-complete

In classical logic, tautologies are true statements. The following language is coNP-complete:

TAUTOLOGY = \{ \varphi : \varphi \text{ – Boolean formula that is satisfied by every assignment} \} .

It is clearly in coNP and so all we have to show is that for every $L \in \text{coNP}$, $L \leq_p \text{TAUTOLOGY}$. But this is easy: just modify the Cook-Levin reduction from $\widehat{L}$ (which is in NP) to SAT. For every input $x \in \{0, 1\}^*$ that reduction produces a formula $\varphi_x$ that is satisfiable iff $x \in \overline{L}$. Now consider the formula $\neg \varphi_x$. It is in TAUTOLOGY iff $x \in L$, and this completes the description of the reduction.
**EXP and NEXP**

The following two classes are exponential time analogues of $P$ and $NP$.

**Def:**
- $\text{EXP} = \bigcup_{c \geq 0} \text{DTIME}(2^{n^c})$.
- $\text{NEXP} = \bigcup_{c \geq 0} \text{NTIME}(2^{n^c})$.

Because every problem in $NP$ can be solved in exponential time by a brute force search for the certificate, $P \subseteq NP \subseteq \text{EXP} \subseteq \text{NEXP}$.

Is there any point to studying classes involving exponential running times?

The following simple result may be a partial answer.
If $\text{EXP} \neq \text{NEXP}$ then $P \neq \text{NP}$

We prove the contrapositive: $P = \text{NP}$ implies $\text{EXP} = \text{NEXP}$.

Suppose $L \in \text{NTIME}(2^{n^c})$ and NDTM $M$ decides it. We claim that then the language

$$L_{\text{pad}} = \{ \langle x, 1^{2|x|^c} \rangle : x \in L \}$$

is in $\text{NP}$. Here is an NDTM for $L_{\text{pad}}$:

- given $y$, first check if there is a string $z$ such that $y = \langle z, 1^{2|z|^c} \rangle$. If not, output REJECT.
- If $y$ is of this form, then run $M$ on $z$ for $2^{|z|^c}$ steps and output its answer.

Clearly, the running time is polynomial in $|y|$, and hence $L_{\text{pad}} \in \text{NP}$. Hence if $P = \text{NP}$ then $L_{\text{pad}}$ is in $P$. But if $L_{\text{pad}}$ is in $P$ then $L$ is in $\text{EXP}$: to determine whether an input $x$ is in $L$, we just pad the input and decide whether it is in $L_{\text{pad}}$ using the polynomial-time machine for $L_{\text{pad}}$. 
What have we learned?

• The class \( \text{NP} \) consists of all the languages for which membership can be certified to a polynomial-time algorithm. It contains many important problems not known to be in \( \text{P} \). \( \text{NP} \) can also be defined using non-deterministic Turing machines.

• \( \text{NP} \)-complete problems are the hardest problems in \( \text{NP} \), in the sense that they have a polynomial-time algorithm if and only if \( \text{P} = \text{NP} \). Many natural problems that seemingly have nothing to do with Turing machines turn out to be \( \text{NP} \)-complete. One such example is the language 3SAT of satisfiable Boolean formulae in 3CNF form.

• If \( \text{P} = \text{NP} \) then for every search problem for which one can efficiently verify a given solution, one can also efficiently find such a solution from scratch.