1 NP

A note on notation. I will use interchangeably ‘set’ and ‘language’, and, sometimes identify the Boolean function that is 1 exactly on the strings in the language with the language. We also talk of a language as a problem and strings as instances of the problem: with this notation the language is the set of YES instances. There is a small imprecision of the notation here similarly to the encoding of Turing machines when talking about computability – the complements of the YES instances should be the NO instances. However, in all interesting cases the problems are in effect questions (like ‘Is the input graph Hamiltonian?’) and we simply ignore the input strings that are syntactically incorrect. The corresponding notational problem was to toss aside strings that were not valid encodings of Turing machines.

1.1 Definition of NP

The class NP was defined as the collection of sets that could be defined by

\[ A = \{ x : \exists y \ P_A(x, y) \} \]

where the predicate \( P_A(x, y) \) is computable in time polynomial in \(|x|\).

It is easy to see that this is equivalent to the following

\[ A = \{ x : \exists \text{ polynomials } p(|x|), q(|x|) \text{ string } y \ P_A(x, y) \} \]
or
\[ \exists c_1, c_2 > 0 \text{ such that } A = \{ x : \exists y \ |y| \leq |x|^{c_1} \ P_A(x, y) \} \text{ with } P_A n^{c_2} \text{ time bounded.} \]

Note the similarity to the definition of c.e. sets. The crucial difference is that the length of the ‘proof’ \( y \) must be polynomially bounded: if we put no bounds on it, we get the c.e. sets.

### 1.2 Examples of sets in NP

1. All sets in P
2. Composite Numbers
3. SAT – satisfiable Boolean formulas in CNF form
4. Hamiltonian Graphs

Justifications (for the claim that the problems above are in NP):

1. The empty string serves as ‘proof’ in all cases where the language is in P.
2. If \( n \) is composite, \( n = a \times b \) for some natural numbers \( a \) and \( b \), neither equal to 1. The string \( y = a, b \) is the required proof: the algorithm \( P_{\text{composite}} \) simply multiplies \( a \) by \( b \) and checks that the result is \( n \). This is clearly a polynomial time algorithm.

3. For input \( x = G \), where \( G \) is a graph \( G = (V, E) \), the string \( y \) is a sequence of vertices of \( V \). The algorithm verifies that the first and the last vertex of the sequence is the same (it is a cycle), that each vertex in the sequence is connected to its successor by an edge of \( E \), that there are exactly \( |V| \) vertices in the sequence, and, except for the first being the same as the last, they are all distinct. Again, this is a polynomial time algorithm.

We should make explicit the arguments that if there is a proof of the form indicated above, the input is a YES instance, and if the input is a YES instance, there is a proof as outlined.

We saw before that the question \( P = NP? \) is very important, and, apparently, very difficult. One of the reasons for its importance is the number
of interesting problems in NP – for which we do not have good (polynomial
time) algorithms.

1.3 NP-hard and NP-complete Problems

**Definition 1.** A set $U$ is NP-hard if for every set $L$ in NP there is a poly-
nomial time computable function $f_L$ such that $\forall x \in L$ iff $f_L(x) \in U$

**Definition 2.** A set $H$ is NP-complete if it is in NP, and it is NP-hard.

We say that the function $f_L$ reduces $L$ to $H$ (or to $U$, as the case may
be).

Intuitively: ‘if we could have an answer to questions about $U$, we could
solve the problem $L$ in polynomial time’. Moreover, this is true for any
problem in NP.

We state this intuition formally.

**Theorem 3.** If $U \in P$, where $U$ is NP-hard, then $P=NP$.

**Proof.** If $U \in P$, the there is some algorithm, $D()$ that for any string $z$
decides whether $z \in U$ in time $|z|^c$ for some constant $c$. Since $D()$ runs in
polynomial time, this means that there is a constant $d > 0$ such that the
running time of $U$ on input $z$ is at most $|z|^d$.

Given a string $x$, the algorithm to decide whether $x \in L$ is the following:
compute $z = f(x)$
compute $D(z) = D(f(x))$
return the answer given by $D$.

The correctness follows from the definition of NP-hardness.

Let us bound the running time. Since $f()$ is a polynomial time algorithm,
there is a constant $e > 0$ such that its running time on $x$ is bounded by $|x|^e$.
Moreover, the length of the output $z$ is also bounded by $|x|^e$, as each step
of the computation can output at most one character. So the total running
time of the algorithm above is bounded by $|x|^e + (|x|^e)^d = O(|x|^c)$ where $c = de$. So the algorithm is polynomial time. \[\square\]

If the set $U$ is not only NP-hard but also NP-complete, we have a stronger
result.

**Theorem 4.** $P=NP$ iff for some NP-complete language $H$, $H \in P$
Proof. The previous theorem implies that if \( H \in P \) we have \( P = NP \).

The other direction is trivial: if \( H \not\in P \) then it is a witness that \( P \neq NP \).

\[ \square \]

**Remark 5.** An NP-complete language is a **hardest language in NP**. A ‘practical’ use of it is as an **excuse**. If we know that a problem is NP-complete (or, worse, NP-hard) we should not look for good algorithms. We may look at special cases, or approximation algorithms.

We still need to show that NP-hard languages are not unicorns—they exist.

### 1.4 NP-complete Languages

Our first attempt is to use the analogy of NP-languages with c.e. languages. \( U(M, x) \), the language of the universal Turing machine that was complete for c.e. languages (I used \( U \) for the name of an NP-hard language on purpose...). So we could try to enumerate all polynomial time predicates \( P_i \), and consider the language

\[ U = \{ (P_i, x) : \text{machine } P_i \text{ accepts } x \text{ when given a proof string } y |y| \leq |x|^c \} \]

This is obviously NP-hard: for any language \( L \in NP \) there is an \( i \) such that \( P_i \) accepts the set \( \{ x \in A \} \) given some proof \( y \) (the length of \( y \) is automatically polynomial in the length of \( x \), since \( P_i \) is polynomially bounded in the length of \( x \).) Now the reduction \( f_L \) is simply \( f_L(x) = (P_i, x) \).

Unfortunately, \( U \) is not in NP: the runtime of each \( P_i \) is bounded by a polynomial, but it is a different polynomial for each algorithm \( P_i \).

There is a simple technical fix: change the definition of \( U \): if the runtime of the algorithm \( P_i \) is \( n^c \), consider the language

\[ H = \{ (P_i, z b^{[|x|^c]} ) : \text{machine } P_i \text{ accepts } x \} \]

where the symbol \( b \) is not in the input alphabet of the language of \( P_i \). In other words, we add a ‘filler’ to the input, so that the simulation will take time linear in the new input length.

It should be clear that the language is still NP-hard, but now it is in NP.

So we have the

**Theorem 6.** NP-complete languages exist.
Proof. Consider the language $H$ above.

Actually, there is a much better result

**Theorem 7.** There are interesting NP-complete languages.

In particular 3-SAT is NP-complete.

*Proof.* We only provide the outline of the proof. The details are not difficult, but there are many...

We need the following lemma about Boolean circuits:

**Lemma 8.** Consider a Boolean circuit with $k$ gates, that computes a relation $R(x_1, x_2, \cdots x_a, y_1, \cdots y_b)$ between its inputs $x_1, x_2, \cdots x_a$ and outputs $y_1, \cdots y_b$. Then there is a CNF formula $F()$ with a number of inputs polynomial in $a, b$ and $k$, such that it is satisfiable exactly when the variables satisfy $R(x_1, x_2, \cdots x_a, y_1, \cdots y_b)$.

*Proof.* For each wire of the circuit, get a Boolean variable (the inputs of the inputs gates will be the $x$s, the outputs, the $y$s.) For each of the $k$ gates write a formula that is satisfiable if the inputs and outputs satisfy the definition of the gate. For example if there is an AND gate with inputs $r, s$, and output $t$, we write the subformula $(\overline{t} \lor r) \land (\overline{t} \lor s) \land (\overline{s} \lor \overline{r} \lor t)$. We do this for every gate, and AND all subformulas.

Now consider the sequence $s_0, s_1, \cdots s_T$ of snapshots of the accepting computation of a Turing machine $M$ on input $x$ and proof $y$, as we studied before. We saw that the relationship between two successive string $s_i$ and $s_{i+1}$ is very simple. In fact each square of $s_{i+1}$ depends only on the values of the corresponding square of $s_i$ and its two neighbor squares. This relation can be verified by a simple circuit, as in the lemma above. The AND of all these formulas will yield a large CNF formula, where the bits $y$ representing the 'proof' determine all other bits (we know what the bits of the input $x$ are, and what the output bits are in $s_T$)

Putting all these pieces together, we find a polytime (in the length of $x$) CNF formula that is satisfiable iff $M$ accepts $x$.

Once we have a nice combinatorial problem that is NP-complete, we can prove many others to have the property. This is done by *reducing* an NP-complete language $H$ to the candidate language $C$.

The strategy to prove $C \in NP$ to be NP-complete is find an NP-complete
set $H$, and a polynomial time reduction $f$ such that
$x \in H$ iff $f(x) \in C$.

Now, for any set $L \in NP$, in order to decide $x \in L$ compute $z = f_L(x)$
where $f_L()$ is the reduction, guaranteed to exist because $H$ is NP-complete.
Then compute $w = f(z)$ the reduction given by our hypothesis.

It is not hard to verify that $x \in L$ iff $f(f_L(x)) \in C$ and that the process
is polynomial time. This shows that $C$ is NP-hard, and since $C \in NP$, $C$ is
NP-complete.

We have implicitly used (and outlined a proof of) the

**Lemma 9.** Reductions are transitive.

We mean that if $U$ reduces to $V$ and $V$ reduces to $W$, $U$ reduces to $W$.

**Remark 10.** There are many interesting NP-complete and NP-hard problems. Many of these have been studied for a long time, but no algorithms are known. The list includes Hamiltonian Cycle, Vertex Cover, Independent Set, Clique, 0-1 Integer Programming, 3D matching, and thousands of others. A (somewhat dated, but still excellent) survey and list of NP-complete problems is M.R. Garey and D.S. Johnson: Computers and Intractability: A Guide to the Theory of NP-Completeness.