The Classes NPI, co-NP, P^{NP}, and NP^{NP} of Problems and The Class #P of Enumeration Problems

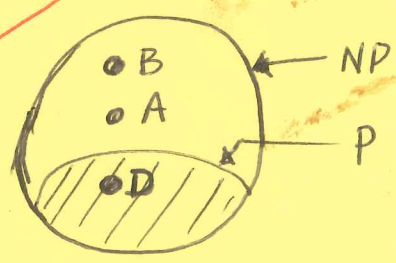
The Class NPI:

One could argue that it is not appropriate to call NPI a class, since as you will see, NPI is NOT an equivalence class in the same manner as P or NP-Complete

- 'I' in NPI stands for "Intermediate"
- NPI is what remains of the set NP after you have accounted for P and NP-Complete. That is, $\boxed{NPI = NP - (P \cup NP\text{-Complete})}$
- How do we know that NPI is not empty? The following theorem, which was proved by Ladner in 1975, establishes that NPI can not be empty:

THEOREM: There exist languages B and D (in class NP) with $B \notin P$ and $D \in P$ such that the language corresponding to their intersection, $\boxed{A = B \cap D}$, has the following properties: $\boxed{A \notin P}$, $\boxed{A \propto B}$, and $\boxed{B \not\propto A}$.

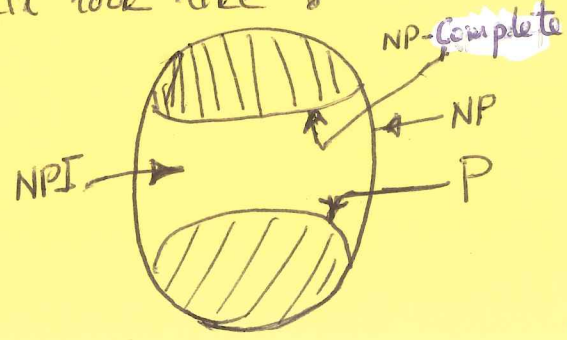
$B \not\propto A$ means B cannot be polynomially transformed to A



Assume for a moment that NPI is empty. In that case, everything outside P in NP is NP-Complete. Now, the theorem tells us we have a language A that is outside P and, therefore, that is NP-Complete. The theorem further states that while $A \propto B$ is true, $B \propto A$ is false. But that is a contradiction. We know that NP-Complete is an equivalence class and that we can construct a polynomial transformation between any two members of the NP-Complete class. Hence, the theorem asserts that NPI is not empty.

[As to why we can assume that A must also be in NP, since $A = B \cap D$, any algorithms that can be used to establish B's and D's membership in NP must also sanction A's membership in NP.]

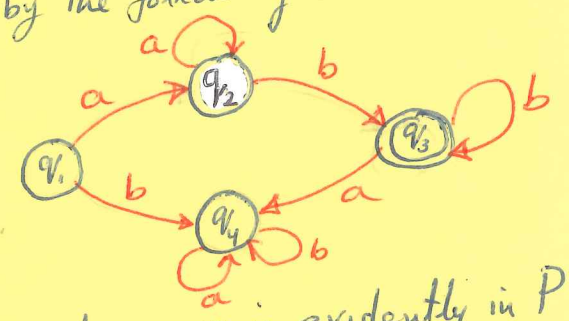
- So that establishes that NPI is not empty. Therefore, the class NP must look like:



- The theorem mentions two languages A and D, with $D \in P$, $A \subseteq D$, yet $A \notin P$. It seems counterintuitive, but only at first thought. Consider the following example:

$$L = \{a^n b^m \mid n, m > 0\}$$

where by a^n we mean n repetitions of a and b^m the m repetitions of b. This language is recognized by the following DFA:



This language is evidently in P because the length of time to accept a string is linearly proportional to the length of the string. Now consider a subset of L for which m and n are required to be the number of Hamiltonian

- By the way, Ladner also showed that NPI must contain a pair of languages C and D such that $C \propto D$ but $D \not\propto C$. What that implies is that NPI is NOT a single equivalence class, unlike the class P or the class NP-Complete.

circuits in the individual graphs in two different sets of graphs. Since the problem of ascertaining the number of Hamiltonian circuits in a graph is exponentially complex, the subset of L chosen in this manner is NOT in P.

- Now that we have established NPI is <u>not</u> empty, we naturally wish to know as to what problems belong in NPI. As it turns out, no useful problems have been proven to belong to NPI. However, there do exist some important NP problems that cannot be proven to belong to either P or to NP-Complete. Such problems could potentially belong to NPI. Two of these problems are

  ① Graph Isomorphism
  ② Composite Numbers

It is now known that Composite Numbers is NOT in NPI. See the discussion on the next page

About the first of these, you already know that subgraph isomorphism is NP-Complete. So you might think that the closely related graph isomorphism might also be NP-Complete. At the same time, you might entertain the idea that graph isomorphism might be easier than subgraph isomorphism because the latter matching would have more degrees of freedom in general. The reality is that no one has been able to come up with a rigorous proof of graph isomorphism being NP-Complete. At the same time, no one has been able to present a P algorithm for graph isomorphism. So, possibly, graph isomorphism belongs to NPI.

▶ About the second potential candidate for NPI, <u>Composite Numbers</u>, we will take that up after we present the co-NP class.

The co-NP Class:

- The class co-NP is the set of all problems that are the complements of the problems in NP:

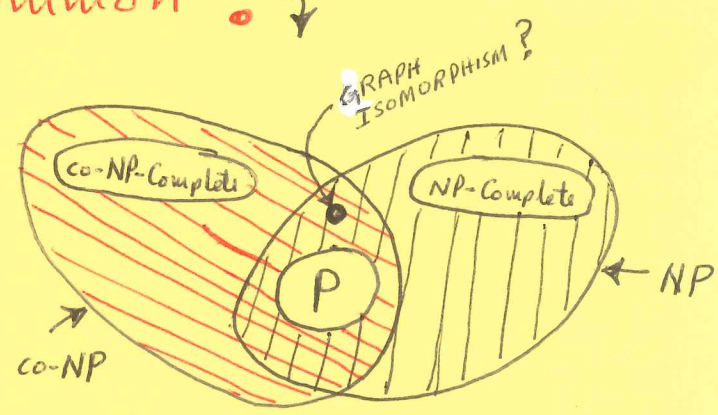
$$\text{co-NP} = \{ \Pi^c \mid \Pi \in NP \}$$

where Π^c is the complement of the problem Π. Given the usual meanings of D_Π and Y_Π with regard to $\Pi \in NP$, $\boxed{Y_{\Pi^c} = D_\Pi - Y_\Pi}$. Therefore, we can think of the complementary problem Π^c as the problem whose answer is reversed in relation to the answer for Π.

- You already know that the complement of an NP-Complete problem is <u>not</u> in NP. Therefore, it must be the case that $\boxed{NP \neq \text{co-NP}}$. However, since the complement of a P problem is also in P, NP and co-NP

must have at least the P subset in common ?

- The figure at right also shows the subset co-NP-Complete within the set co-NP. The problems in co-NP-Complete are the complements of the problems in the set NP-Complete. This is in agreement with our earlier assertion that the complement of an NP-Complete problem is NOT in NP.



Because of Turing reduction from every problem $\Pi \in$ NP-Complete to its counterpart $\Pi^c \in$ co-NP-Complete, the problems in co-NP-Complete are at least as difficult as the NP-Complete problems in NP. One can extend this argument to assert that co-NP-Complete is the set of the most difficult problems in co-NP.

- We will now get back to the problem of Composite Numbers mentioned in our discussion on the set NPI of problems :

Composite Numbers (CN)
INSTANCE : Positive integer K
QUESTION : Are there integers m and n, both greater than 1, such that K=mn?

It is obviously the case that $\boxed{CN \in NP}$ since any guess for the integers m and n can be verified trivially by forming the product mn. What makes the story of CN interesting is that its complement is also in NP. The complement of CN is the problem of PRIMES :

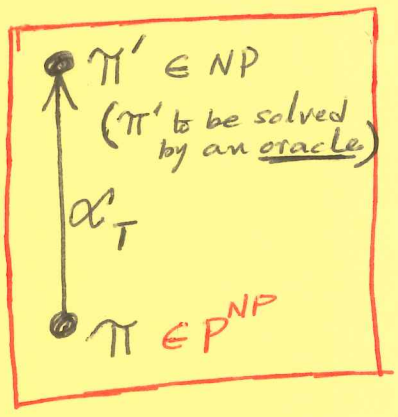
PRIMES
INSTANCE : Positive integer K
QUESTION : Is K a prime number ?

See Lecture 11 of my lecture notes on Computer and Network Security

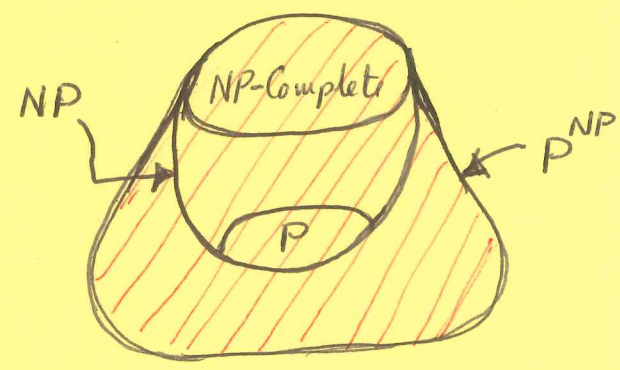
It was proved by Agrawal, Kayal, and Saxena in 2002 that $\boxed{PRIMES \in P}$. That implies that both CN and its complement are in NP. That precludes the possibility of CN being NP-Complete since the complement of an NP-Complete problem cannot be in NP. At the same time, it has not been possible to develop a polynomial time algorithm for CN. Despite the absence of a P solution for CN, it must obviously be the case that CN $\in$ P since its complement PRIMES $\in$ P. Before the discovery PRIMES $\in$ P, CN was considered to be a strong candidate for NPI.

The Classes P^{NP} and NP^{NP} :

- We will now define additional problem classes that help us categorize problems that are not necessarily in class NP. These additional classes, P^{NP} and NP^{NP}, will turn out to be superclasses of NP.



- The class P^{NP} is the set of all problems that can be Turing reduced to problems in NP. In other words, P^{NP} is the set of all π such that $\pi \propto_T \pi'$ with $\pi' \in NP$. (Since $\pi' \in NP$, $\pi \propto_T \pi'$ implies $\pi' \in$ NP-Complete, $\pi \propto_T \pi'$ we can say $P^{NP} \equiv$ NP-Easy)
- So if an answer to a problem π can be logically deduced from the answer to some problem π' in NP and if this logical inference can be made in polynomial time, then $\pi \in P^{NP}$.
- You may also think of P^{NP} as the set of all problems that are no more difficult than the problems in NP and that are in some way logically connected with the problems in NP.
- All problems in P^{NP} can be solved in polynomial time on an Oracle Turing Machine (OTM). Basically what that means is that the time it takes to transform π into a structure that looks like something in NP and the time it takes to transform the solution to that structure back into an answer to our original problem π should only take polynomial time.



- Since all problems in co-NP are Turing reducible to the problems in NP (that is a consequence of the definition of co-NP), it follows that all of co-NP is in P^{NP}.
- Note that P^{NP} gives us a computational device for solving problems that are in P^{NP}. (That computational device is OTM.) This was NOT the case with the problem class NP-Hard. That classification was just a means to talk about the relative complexity of problems vis-a-vis the NP-Complete problems. We said if $\pi \propto_T \pi'$ and $\pi \in$ NP-Complete, then π' is NP-Hard. In and of itself, that does not tell us how to solve π'. But when we say a problem is in P^{NP}, you know that you will be able to solve it in polynomial time on an OTM.
- We will now introduce a problem class that, in general, cannot be solved by an OTM. This new problem class will be designated NP^{NP}. The computational device for all problems in NP^{NP} is NOTM, which stands for Nondeterministic Oracle Turing Machine.
- An OTM has a DTM (Deterministic Turing Machine) consulting an oracle. An NOTM has an NDTM (Nondeterministic Turing Machine) consulting

an oracle:

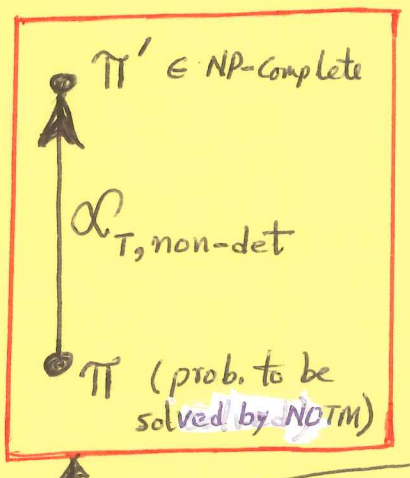
> NP^{NP} is the set of all problems that can be solved in polynomial time on a NOTM with the oracle supplying answers to some NP-Complete question.

- Note that for both P^{NP} and NP^{NP}, the oracle does the same thing — it provides an instantaneous answer to some NP-Complete problem. However, whereas the main computational engine for P^{NP} is a DTM, it is an NDTM for NP^{NP}. For that reason, NP^{NP} is a superset of P^{NP}. Note that every problem that can be solved by a DTM in polynomial time can also be solved by a NDTM in polynomial time, but not the other way around.

- Recall now the official definition of Turing Reduction as presented in Lecture 37. We say that Π Turing reduces to Π' if Π can be solved in polynomial time on a DTM that consults an oracle for a solution to Π'. Now if we replace the DTM with the more powerful NDTM, what we have is an extension of the concept of Turing reduction. This extension is represented by $(\text{Turing Reducibility})_{\text{non-det}}$, or more compactly by $\boxed{\propto_{T,\text{non-det}}}$.

- We can therefore provide the following alternative definition for class NP^{NP}:

> NP^{NP} is the set of all problems Π that can be $(\text{Turing reduced})_{\text{non-det}}$ to some NP-Complete Π', as shown figuratively at right.



- As is implied by the boxed comment at right, the problems in NP^{NP} can be more difficult than the problems in P^{NP}. (Recall, P^{NP} includes NP.) We will now present an example of a problem in NP^{NP}:

> Strictly speaking, displaying Π to Π' transformation as shown above does NOT make sense any longer. In the past, our upward pointing arrow was meant to indicate that the problem at the top would generally be harder than the problem at the bottom. But in a $\boxed{\Pi \propto_{T,\text{non-det}} \Pi'}$ transformation, Π will generally be much harder than Π'. Whereas, Π', being NP-Complete, could be solved in polynomial time on an NDTM, for Π to be solvable in polynomial time on an NDTM requires the help from an oracle.

MINIMUM EQUIVALENT EXPRESSION (MEE):

Instance: A well-formed Boolean expression E involving literals on a set V of variables and a non-negative integer K.

Question: Is there a well-formed Boolean expression E′ that contains K or fewer occurrences of literals such

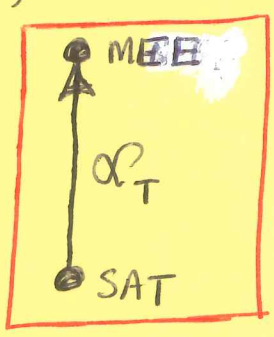
that E' is equivalent to E? We say that the Boolean expressions E and E' are equivalent when the truth values of E and E' agree for all possible truth assignments to the variables in V. (The expressions E and E' are allowed the usual Boolean connectives $\wedge, \vee, \neg$, and $\rightarrow$, in addition to the truth symbols T and F, and the variables in the set V.)

Example of MEE: Let $V = \{u_1, u_2, u_3\}$. Consider the Boolean expression $\boxed{E: \; u_1 \wedge (u_1 \vee \bar{u}_2) \rightarrow u_3}$. For each assignment of the truth values to the variables in V, E is either true or false. E involves 4 occurrences of the literals over V. Assume $K = 3$ in this instance of MEE. In this case, the <u>decision problem</u> MEE asks <u>whether there exists</u> another Boolean expression E' with 3 or fewer occurrences of the literals over V such that $\boxed{(E' \rightarrow E) \wedge (E \rightarrow E')}$ will be true for <u>every</u> assignment of truthvalues to V, implying that E and E' will be equivalent.

This problem is important to folks who design digital logic circuits

The word 'every' is important

- Although <u>not</u> important to demonstrating that MEE $\in NP^{NP}$, we will now establish that $\boxed{\text{MEE} \in \text{NP-Hard}}$. This we will do by proving a Turing reduction from SAT to MEE.

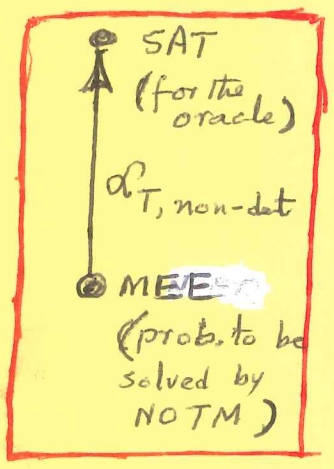


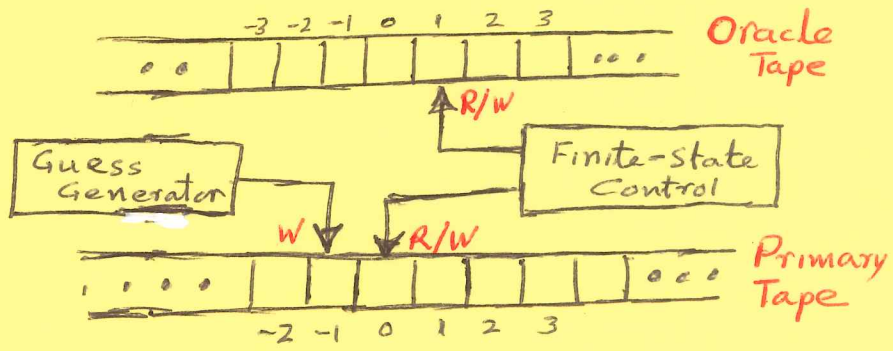
[Recall that SAT (Satisfiability) consists of a set of clauses, with each clause a disjunction of literals. SAT asks the question whether there exists a truth assignment for the variables in V so that all clauses will be simultaneously true.] For the Turing reduction $\boxed{\text{SAT} \propto_T \text{MEE}}$, we construct an expression E explicitly as a conjunction of the clauses in the SAT instance, set K equal to the number of occurrences of the literals in the SAT instance, ask the oracle whether exists an E' that is equivalent to E in the sense described earlier. Should it be the case that E is unsatisfiable, the oracle will return 'no' since E cannot be made to be true for any truth assignment to V. ▶ In order to establish equivalence between E and a hypothetical E', the oracle needs to know as to what truth assignments to V, E will be true for. So that proves MEE is NP-Hard.

- Also note that $\boxed{\text{MEE} \notin \text{NP}}$. This follows from the fact that to verify that E is equivalent to a $\boxed{\text{guess}}$ E', we have to compare the two for every truth assignment over V. That would require exponential time.

- We will now show that $\boxed{\text{MEE} \in NP^{NP}}$ by proving that a NOTM will solve

MEE in polynomial time. This we will do by proving $\boxed{\text{MEE} \propto_{T,\text{non-det}} \text{SAT}}$.
(Note how SAT plays opposite roles in the NP-Hardness proof and the NP^{NP} proofs for MEE.) We will use the NOTM shown below for solving the MEE problem:





We will start problem solving by laying down the expression E in the squares 1, 2, 3, ... of the primary tape. After that the Guess Generator will lay down a completely random guess E' in ~~squa~~ squares indexed $-1, -2, -3, \ldots$ of the primary tape. The only constraint that the Guess Generator will observe is that E' uses K or fewer occurrences of the literals. Subsequently, the Finite-State Control causes the following expression to be written out in the scratch-pad area of the primary tape: $\boxed{\neg((E' \rightarrow E) \wedge (E \rightarrow E'))}$. The Finite-State Control then transforms this expression into a $\boxed{\text{conjunctive normal form (CNF)}}$, which is a conjunction of clauses, with each clause a disjunction of literals. (Algorithms exist for doing this in polynomial time.) The NOTM then writes out the CNF to the oracle tape and, with a loudspeaker pointed toward the sky, asks the oracle whether or not the CNF is satisfiable.

► If the SAT oracle says "no" (usually, a severe thunderstorm or a tornado is an indication of that), then E' is the desired reduced expression for E. We have thus proved that $\boxed{\text{MEE} \in NP^{NP}}$.

- Just for contrast, we will now present a problem that is <u>not</u> in NP and that is in P^{NP}.

$\boxed{\text{MAXIMUM CLIQUE SIZE}}$ (MAX-CLIQ):

Instance: Graph G and positive integer K.
Question: Does the largest clique in G contain K vertices exactly?

- Note that MAX-CLIQ is NOT in NP. Even if we come up with a K-vertex clique in G, there is no way to verify in polynomial time that it is the largest clique in G.

- MAX-CLIQ can however be solved in polynomial time by an OTM, that is, by a DTM with access to an oracle. The DTM will make two successive calls to the oracle. The first call to the oracle will be

for solving the NP-Complete problem CLIQUE with K set to the same value as in the MAX-CLIQ instance. If the oracle answers 'yes', that means G contains a clique of size K or greater. The second call to the oracle, made only if the oracle's answer to the first call is 'yes', is for solving $CLIQUE^c$ with integer parameter set to K+1 ($CLIQUE^c$ means the complement of CLIQUE). If the oracle again says "yes", that means G does not contain cliques of size K+1 or greater. In other words, with the oracle saying "yes" to both questions, the largest clique in G must be size K exactly. Therefore, $\boxed{\text{MAX-CLIQ} \in P^{NP}}$.

The Class #P of Enumeration Problems:

- Examples of enumeration problems: ① How many Hamiltonian circuits are there in a graph? ② Given a set C of clauses over a set V of variables, how many truth assignments of V simultaneously satisfy all clauses in C?
- Formal Definition of an Enumeration Problem: For a given problem Π, associate a set $S_\Pi(I)$ of solutions with every instance $I \in D_\Pi$. (If no solutions exist for a given I, $S_\Pi(I)$ is empty.) An enumeration problem asks: What is the cardinality of $S_\Pi(I)$?
- Enumeration Problems That Can be Solved in Polynomial Time: ① How many different spanning trees exist in a graph? ② How many Eulerian walks exist in a graph? ; etc.
- The enumeration problems associated with NP-Complete decision problems are NP-Hard. Since the solution to an enumeration problem returns the cardinality of the solution set, if the cardinality is zero, the answer to the underlying decision problem must be 'no'. Otherwise, the answer to the decision problem must be 'yes'.
- Definition of the #P class of enumeration problems: An enumeration problem Π belongs to #P if there is a non-deterministic algorithm such that for each $I \in D_\Pi$ the number of distinct "guesses" that lead to the acceptance of I is exactly $|S_\Pi(I)|$ and such that the length of the longest ~~com~~ acceptance computation is bounded by a polynomial in length(I).
- The two examples of the enumeration problems at the beginning of this discussion on enumeration problems are both in #P.

#P is read as "number p"

- Just as the notion of "completeness" was used to capture the hardest problems in NP, one can define **#P-Complete** as the equivalent class of the most difficult problems in class #P:

> An enumeration problem Π is called #P-Complete if
> i) $\Pi \in \#P$
> ii) for all $\Pi' \in \#P$, $\Pi' \propto_T \Pi$

Note that whereas NP-Completeness is based on polynomial transformability, #P-Completeness is based on Turing reducibility.

- The problem of counting the number of satisfying truth assignments for an instance of SATISFIABILITY is #P-Complete.