4. LIMITS OF COMPUTATION: Tractable and Intractable Problems

Tractable problems: the class P

All the problems seen in the earlier part of the course (such as multiplying numbers and calculating a determinant) had algorithms whose time-demand was described by a polynomial function. Such problems are said to be **tractable** and in the class **PTIME** (**P**olynomial **TIME**).

Since we are not going to be dealing with issues of space-demand we will simplify the notation and refer to this (as is commonly done) as the **class P**.

A problem is in P if it admits an algorithm with worst-case time-demand in $O(n^k)$ for some integer k.

Note that to be in P a problem just has to have *some* algorithm which can solve it in polynomial time. It may also have algorithmic solutions whose time-demand grows unreasonably (as in the case of finding a determinant, where the naïve, definition-based algorithm took time in O(n!)) but this does not change the complexity class assignment (a determinant can also be evaluated in $O(n^3)$ using the Gaussian elimination method).

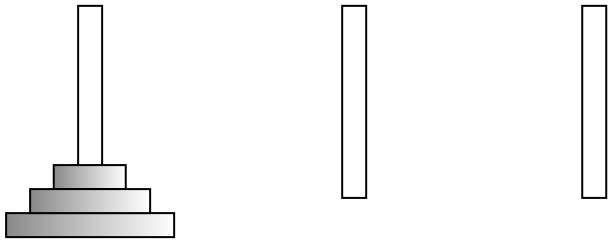
However there are some problems for which it is known that there are *no* algorithms which can solve them in polynomial time, these are referred to as **provably intractable** and as being in the class **EXPTIME** (**EXP**onential **TIME**) -- or worse. For these problems it has been shown that the lower bound on the time-demand of *any* possible algorithm to solve them is a function that grows 'unreasonably fast'.

Intractable problems: the class EXPTIME and beyond

A problem is in the class EXPTIME if all algorithms to solve it have a worst-case time demand which is in $O(2^{p(n)})$ for some polynomial p(n).

Example: the Towers of Hanoi

Consider the following problem:



"Move the three rings, which are piled up in descending order of magnitude, from peg A to peg C, possibly using peg B in the process, moving rings one at a time, and at no time allowing a larger ring to rest on top of a smaller one."

The puzzle above has the solution:

move topmost ring from A to C;
move topmost ring from A to B;
move topmost ring from C to B;
move topmost ring from A to C;
move topmost ring from B to A;
move topmost ring from B to C;
move topmost ring from A to C;

But what about the n-ring case?

The procedure Hanoi, below, solves the problem:

Hanoi(n, i, j) moves the n rings currently resting on peg i to peg j and is called initially with i=1, j=3. It uses the constant-time subroutine 'move(i, j)' which just takes the *topmost* ring from peg i and puts it onto peg j:

```
ALGORITHM Hanoi( n, i, j )
// Solves the Towers of Hanoi problem in the n-ring case
if n=1
        move( i, j )
else
        Hanoi( n-1, i, 6-i-j )
        move( i, j )
        Hanoi( n-1, 6-i-j, j )
```

For example if n=2 the algorithm executes as follows:

(start)

Hanoi(2, 1, 3)

↳ Hanoi(1, 1, 2)

↳ move(1, 2)

move(1, 3)

Hanoi(1, 2, 3)

↳ move(2, 3)

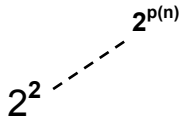
(finished)

It can be shown, using the simple analysis techniques for recursive procedures described in the earlier part of the course, that Hanoi takes $2^n - 1$ steps to solve the n-ring problem, so this algorithm is in $\mathbf{O(2^n)}$. However it can also be shown -- less easily -- that the lower bound on time-complexity for this problem is also in $O(2^n)$ and thus that the Towers of Hanoi puzzle is in EXPTIME.

Higher time-complexity classes

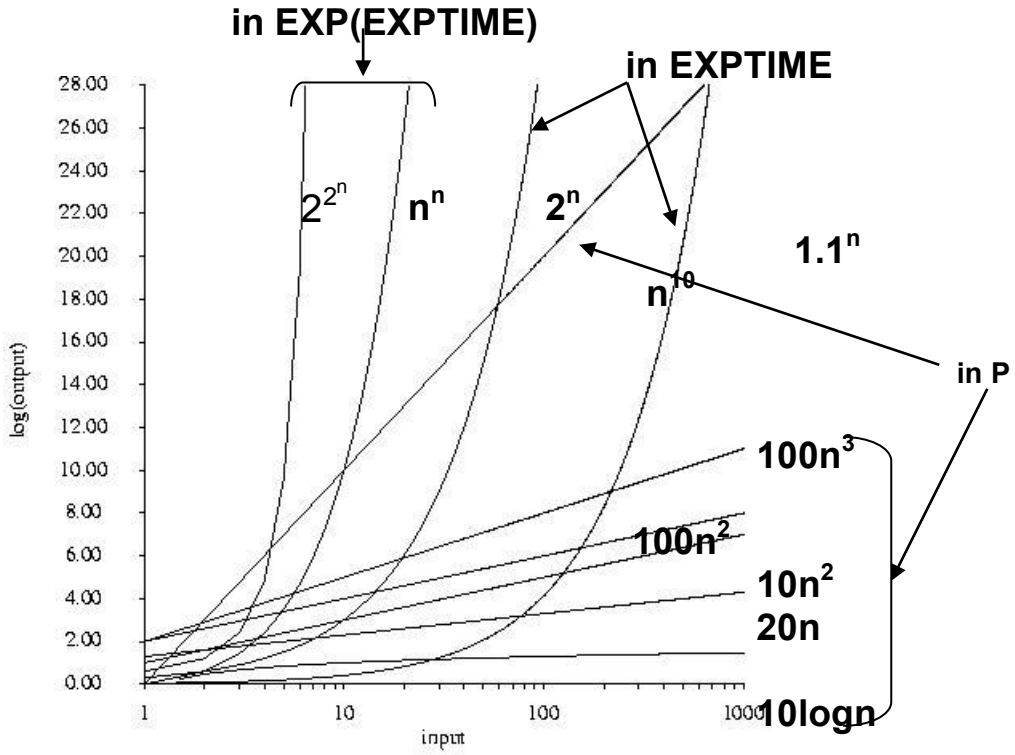
There are other classes of problems for which the time demand cannot be bounded above even by a function of the form $2^{p(n)}$. In fact there are is a hierarchy of these higher time-complexity classes such that a problem within a given class is considered 'more intractable' than all those within lower-ranked classes.

So beyond EXPTIME we can have EXP(EXPTIME), for which the time-demands of all known solutions are bounded above by a multiple of $2^{2^{p(n)}}$, EXP(EXP(EXPTIME)) problems which are in $O(2^{2^{2^{p(n)}}})$... and there are problems whose time-complexity is even worse, and cannot be bounded by *any*



(referred to as 'non-elementary' problems (!!!)).

All these classes of provably intractable problems, from EXPTIME upward, can be referred to as having a **super-polynomial** time demand. (In looser usage these are commonly said to require 'exponential time' but this should be taken to mean 'at least as bad as in EXPTIME'.) These problems are essentially insoluble for large instances; the Towers of Hanoi puzzle is in the lowest of these super-polynomial classes, EXPTIME, but yet moving one ring every second the 64-ring case would take more than 500 billion years to solve -- it's not surprising that the monks traditionally credited with formulating the puzzle (actually both the puzzle and the monks may have been invented in 1883 by a French mathematician) believed the world would end then.



However it turns out that the most interesting class of problems is a class which lies in some sense *between* the class of tractable problems P and those of the provably intractable, super-polynomial time problems.

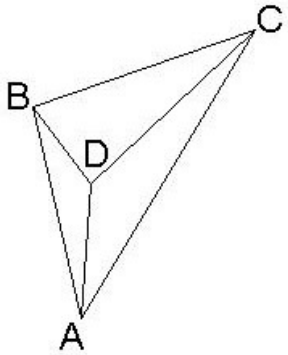
These are problems which are *probably* intractable -- but we're not quite sure.

The classes NP and NPC

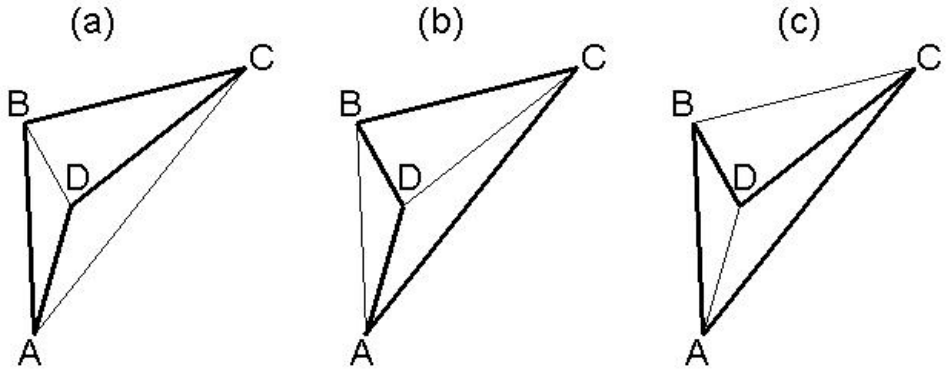
Example: the Hamiltonian Circuit Problem (HCP)

A connected, undirected, unweighted graph G has a Hamiltonian circuit if there is a way to link all of the nodes via a closed route that visits each node once, and only once.

The 4-node graph below



has three Hamiltonian circuits



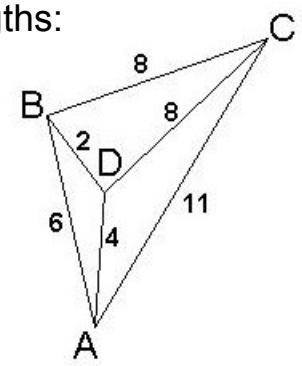
It is not difficult to find a Hamiltonian circuit in a small graph like this but as the size of the graph grows the time-demand appears to scale *very* badly and it is strongly believed that there are no polynomial time algorithms for this problem.

Example: the Travelling Salesman Problem (TSP)

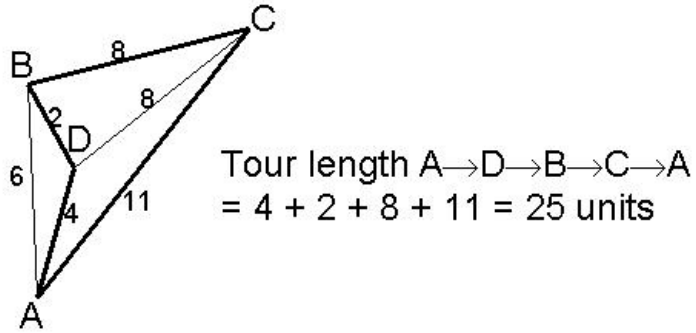
The TSP shares the extremely bad scaling behaviour of the HCP, and is one of the best-known examples of a problem in this 'probably intractable' class. This graph problem is similar to the HCP in that it looks for a route with the same properties as required by the HCP, but now of minimal length as well:

Given a connected, undirected, weighted graph (G, W), where W is the set of edge weights ('city distances'), the Travelling Salesman Problem (TSP) seeks to find the shortest valid tour (a circuit visiting each node ('city') once and only once).

Consider the example 4-node graph again, but now add some edge lengths:



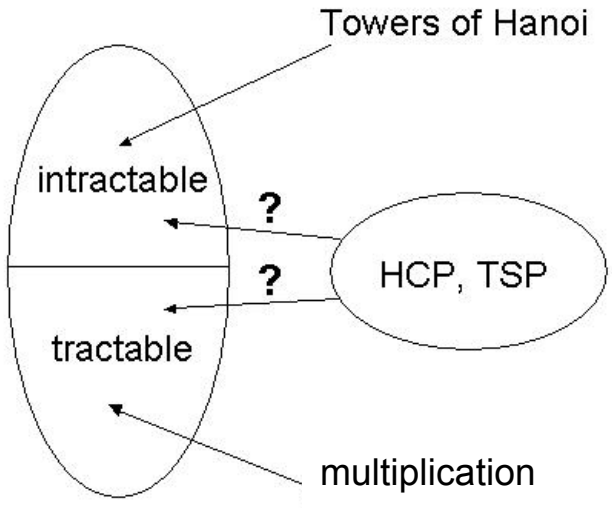
It can be seen that the three valid tours (a), (b), (c) marked earlier as Hamiltonian circuits have total lengths of 26, 25, 27 units, so the optimal tour is that of (b):



Again, there appear to be no algorithms which solve this problem in polynomial time.

However, the HCP and TSP differ from problems like the Towers of Hanoi because although no-one has yet found a polynomial time algorithm for them, no-one has *proved* that no such algorithm exists.

The HCP and TSP belong to the class **NPC**, which is a subset of the larger problem class **NP**. NPC is a class of problems whose time-complexity is presently unknown, though strongly believed to be super-polynomial, and can thus be thought of as being 'probably intractable'.



Thousands of problems are now known to have this probably-intractable character, including **optimisation problems** such as the TSP, **scheduling problems** (such as the timetabling of lectures and exams!), **decision problems** such as whether a map or graph can be coloured in a certain way, whether an area of a given size can be covered by a specified set of patterned tiles, or if a logical assertion can be satisfied. These problems can't be ignored since even when they don't have obvious practical consequences (as in the case of the timetabling problem) they are often abstract forms of problems that do have real-world relevance -- for example, variants of the TSP arise in communications networks planning and in optimising the layout of silicon chips.

But what, other than their probably-intractable character, sets apart problems in NP and NPC?

How does one know that a problem belongs in this class? (Simply failing to have found a good algorithm for it isn't a sufficient reason, it might just be that we hadn't tried hard enough.)

And what do the terms NP and NPC actually *mean*?

These questions are much easier to answer if the discussion is restricted to **decision problems**.

This type of problem is the most straightforward to reason about; most of the work in establishing the nature of complexity classes and the relationships between them has been done in the context of decision problems.

In a decision problem the output required is simply **yes** or **no**. The set of input instances is divided into **yes-instances** and **no-instances** -- for example if the problem was 'is this a prime number?' then 7, 17 and 23 would be yes-instances; 6, 15 and 21 would be no-instances.

From this point on the discussion will be restricted to decision problems. This is not an unreasonable restriction since other problems can usually be reduced to sequences of decision problems (in which case the original problems must clearly be at least as hard as the component decision problems themselves).

For example the **Travelling Salesman Decision Problem (TSDP)** is a variant on the TSP defined as follows:

TSDP((G, W), d) = yes if the weighted graph (G, W) has a valid TSP tour of length ≤ d.

A solution to the TSDP could be used to give a solution to the problem TSP(G,W) for the n-node graph (G, W), provided that integer edge weights are used

```
for d <− min_tour_length to max_tour_length
        if TSDP( (G, W), d ) = yes then return d and halt
```

(where *min_tour_length* = n×*min_edge_length*, *max_tour_length* = n×*max_edge_length*).

This automatically retrieves the shortest route (and is guaranteed to halt given that n×*max_edge_length* is logically longest tour).

In the example previously given, where *min_tour_length* = 4×2 = 8, *max_tour_length* = 4×11 = 44, TSP(G, W) would execute as follows:

TSDP((G, W), 8) = no
TSDP((G, W), 9) = no
...
TSDP((G, W), 24) = no
TSDP((G, W), 25) = yes → ***halt***

Restricting the discussion to decision problems like the TSDP will allow a clearer statement of the defining properties of the classes NP and NPC.

However there is first just one more concept that needs to be introduced, that of the **polynomial time reduction** of one problem to another.

Polynomial time (p-time) reduction

Consider again the examples of the Hamiltonian Circuit and Travelling Salesman Decision problems (HCP and TSDP). Because the TSDP asks first for a valid tour (equivalent to a Hamiltonian circuit in an undirected graph) and *then* requires that its length should be less than some specified value it's therefore in some sense 'as least as hard as' the HCP. The idea of p-time reduction makes this intuition explicit by showing that a solution to the TSDP can be converted into a solution to the HCP in a negligible (in this context, polynomial) amount of time, so that in some sense the HCP is indeed contained within the TDSP.

To say in general that *a problem A reduces in p-time to another problem B*, written as

$$A \leq_p B$$

means that there is some procedure, taking no more than polynomial time as a function of the size of the input to A, which

- converts an input instance of A into an input instance of B
- allows a suitable algorithm for problem B to be executed
- provides a mechanism whereby the output obtained by this algorithm for problem B can be translated back into an output for problem A

The algorithm for problem B thus also provides a solution to problem A. Moreover A's solution will be obtained in a time which is in the same complexity class as the algorithm which solves B, since the extra work needed to 'translate' is just in p-time. Most importantly, if we know -- or in the case of NP and NPC, suspect -- that we have a lower bound on the time demand of all possible algorithms for B, we can say that in terms of its fundamental difficulty problem A is 'no worse than' problem B.

Example: to show that

$$\text{HCP} \leq_p \text{TSDP}$$

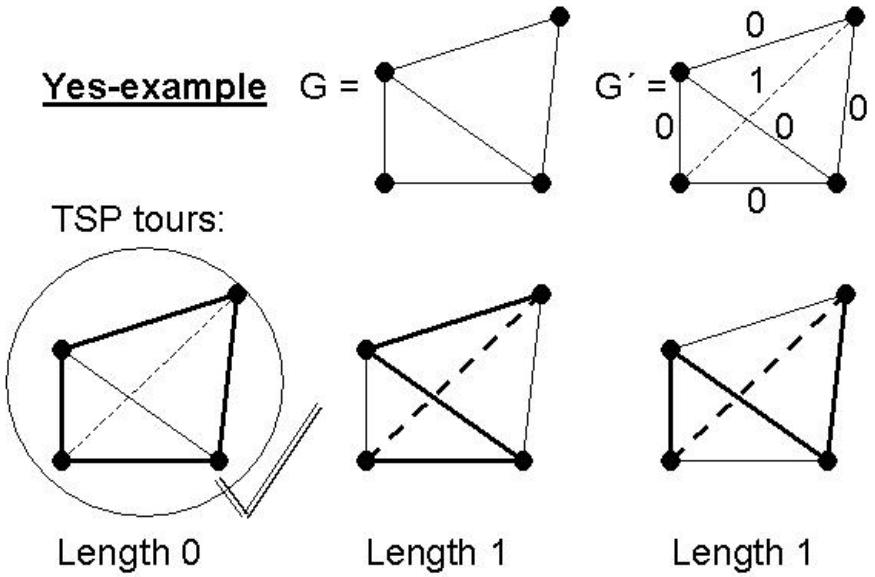
ie that HCP reduces in polynomial time to TSDP

- Take an instance of HCP, say G.
- Create a new weighted graph (G′,w) as follows:
  - Nodes of G′ are the same as nodes of G.
  - Add extra edges so that G′ is fully connected (so that it now has $n/2(n-1)$ edges).
  - Set the weights in the new graph G′ so that if an edge existed already in G it has weight 0, otherwise (a newly added edge) it has weight 1.
- Return TSDP((G′,w), 0) – ie ask if there is a valid city tour in this new graph of length not greater than zero.

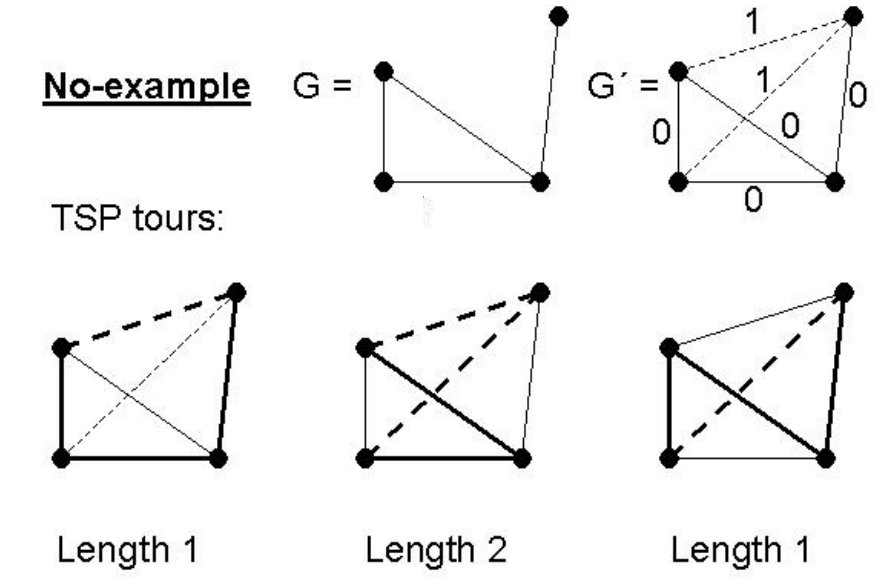
The reduction takes time $O(n^2)$ in the number of nodes since the maximum number of edges in any undirected graph is only $n/2(n-1)$, and thus the number of added edges must be bounded above by this.

The reduction works because:

If there *is* a Hamiltonian circuit in the graph G (yes-instance), there must also exist a circuit in G′ with length zero as all the circuit edges would have been given zero-weighting, and this zero-length tour would cause TSDP((G′,w), 0) to also be true.



If conversely there is *not* a Hamiltonian circuit in the graph G (no-instance), then any newly-created ‘circuit’ in G′ must have length > 0, as it must include at least one of the new edges with length 1. Hence TSDP((G′,w), 0) would in this case also be false.



There are three basic defining properties of problems in NP and NPC.

(i) Problems in NP and NPC are 'very hard to solve but easy to check'

The problems are hard because they appear to only admit algorithms whose time-demand behaviour is described by super-polynomial functions.

However if a solution to a yes-instance of the problem is asserted then it can be *checked* in polynomial time; this ability to check a solution for correctness in polynomial time is referred to as a **short certificate** for the problem.

This is equivalent to saying that the problem can be solved by a hypothetical algorithm that at each branch in its decision tree 'knows' whether that path will lead eventually to a solution. Such a hypothetical algorithm is referred to as **non-deterministic** (note, *not* the same as 'probabilistic') and since the execution path of the non-deterministic algorithm corresponds exactly to the steps needed to check in polynomial time the validity of a solution, the class of problems with this 'very hard to solve, but easy to check' property is known as **N**on-deterministic **P**olynomial (NP).

Example: Is there a Hamiltonian circuit in this graph?
Short certificate: Follow the route suggested, checking that every node is visited and that you end up back where you started -- can't take time greater than $O(n^2)$ since there are only a maximum of $O(n^2)$ edges in any connected graph.

(ii) Problems in NPC are 'the hardest problems in NP'

An **NP-hard problem** (which may not itself be in NP) is one to which any problem in NP can be reduced in polynomial time:

If A is NP-hard, for all B in NP it is true that $B \leq_p A$
(B reduces in p-time to A)

The class NPC is the class of problems within NP itself which have this property:

$$\textbf{NPC} = \textbf{NP} \cap \textbf{NP-hard}$$

One way to think of this is that NPC is the class of problems 'closest to being provably intractable' because it is a subset of a set of problems, NP-hard, which contains some that certainly are.

(iii) Problems in NPC 'stand or fall together'

Any problem in the class NPC can be shown to be **reducible in polynomial time** to any other problem in the class, meaning that there is a way in which any problem A can be mapped onto any other problem B using a number of steps taking no more than polynomial time such that a solution for B also provides a solution for A, and that the converse can also be done:

If A, B $\in$ **NPC then** $A \leq_p B$ ***(A reduces in p-time to B)*** **and**
$B \leq_p A$ ***(B reduces in p-time to A)***

The is the 'completeness' property of the class of **N**on-deterministic **P**olynomial **C**omplete (NPC) problems -- a solution to any one of them in this sense provides a solution to any other. It is the best-known property of problems in NPC because it means that should a p-time algorithm be found for just one problem in NPC, then *all* NPC problems would be soluble in p-time. Moreover if this were to happen all the problems in NP would be pulled in too; thousands of previously-intractable problems would then in principle become soluble in 'reasonable' amounts of time.

(It should be noted though that actually *finding* a p-time algorithm for the newly-reassigned problem of interest might be very hard, and that also a p-time algorithm that took time, say, in $O(n^{100})$, might not be very helpful, hardly better in practice than one in EXPTIME.)

Nevertheless, **'is P = NP?'** is the most famous problem in theoretical computer science -- there is even a prize of $1m for resolving it! And in summer 2010 it was believed someone would collect, when Vinay Deolalikar published a preprint paper "P ≠ NP". However there was later found to be a problem with the proof and the work is currently still in revision.

How (in generality) might the 'P = NP?' question be resolved?

To show P = NP…

Find a p-time algorithm for any of the problems in NPC. As discussed above, this also would provide, in principle, a p-time algorithm for *all* problems in the class NP.

To show P ≠ NP…

In spite of the problems with the Deolalikar proof this is still widely believed -- assuming that the problem is ever resolved -- to be the more likely outcome, and would require only that a single problem in NP (either inside or outside of the class of the 'hardest' problems NPC) be shown to have an EXPTIME *lower bound*. Just one counterexample to the assertion P=NP would be sufficient to show that the sets P and NP are not in fact equal.

However in this case less would be known about the subsequent destinations of individual problems. Certainly all problems in the class NPC, these being by definition 'at least as hard as' the one for which this new super-polynomial time lower bound had been established, would also go into EXPTIME or worse. However not all those in the former class NP \ NPC (where A\B in set theoretic terminology means 'what is left in set A after the members of set B have been taken out') would necessarily follow them -- some of this class might in fact end up in P.

It is important to emphasise that the stand-or-fall-together property applies only to problems in NPC, and not to those in NP which are not also NP-hard, those in the class NP \ NPC. In particular if a problem in this latter class has its complexity status downrated to p-time it may be an unexpected result but *only* that specific problem is affected. It would *not* prove P=NP.

Such a result was presented by Minandra Agrawal in 2002 in the paper "PRIMES is in P". Along with Deolalikar's attempted proof that P ≠ NP it was one of the few instances when theoretical computer science has hit the newspaper headlines.

PRIMES is the problem of determining whether an n-bit number greater than 1 has factors other than one and itself. Before 2002 most people believed that PRIMES should be in NPC, it was just that no-one had yet managed to construct a chain of p-time reductions that would show it.

However Agrawal, Kayal and Saxena showed conclusively that PRIMES was indeed in P, demonstrating it by constructing an algorithm that ran roughly in $O((\log n)^{7.5})$.

The reason this was a newsworthy result (other than the misunderstanding it had proved P=NP) was there was concern the new algorithm might make some cryptosystems insecure.

Primality testing is very important in cryptography; the most widely used **public-key cryptosystems** use the **RSA algorithm** of Rivest, Shamir and Adleman, which requires large prime numbers to be generated (the security of the system relies on the observation that is is much easier to multiply numbers than to determine the factors of a number).

So did the new algorithm make cryptosystems like RSA insecure? In fact no, because **probabilistic algorithms** which run faster than the Agrawal algorithm and have a very small probability of error already existed to generate prime numbers as keys.

The probabilistic algorithm used most often for generating prime numbers, the **Miller-Rabin algorithm**, can be proved to give a correct answer if a number is indeed prime and to be incorrect in the case of an actually-composite number with a probability described on the 'PRIMES is in P little FAQ' page as "smaller than, say, the probability that the computer hardware running the algorithm makes an error, while in the same minute you are struck by lightning and win the lottery." In other words, for all practical purposes it is free of error.

Probabilistic algorithms like this, which can often be made to have an arbitrarily small chance of error, can also be used to handle problems in NPC so as to allow workable solutions to be obtained to, say, timetabling and scheduling problems.

Others are:

- ***Restricting to small instances***, for which the super-polynomial time complexity might not be a serious problem.

- Hoping that the ***average case complexity*** of the problem might be in p-time even if its worse case isn't (though arguments about average cases are often very hard to construct in practice).

- Devising algorithms guaranteed to give ***near-optimal solutions*** (say, within 1% of the ideally-desired result) in all cases.

How in practice is a new problem assigned to NPC?

It would appear that for a new problem A to be classed as NPC it would need to be shown that

$A \leq_p B_1$ and $B_2 \leq_p A$, for some B_1, B_2 in NPC

(B_1 and B_2 could be the same problem, but wouldn't have to be).

However the first of these reductions, $A \leq_p B_1$, is really asking us to show that 'A is no worse than any problem in NPC', which is another way of stating that is in NP. And to show a problem is in NP, one only needs to display a short certificate for it -- this is usually far easier than doing the reduction.

Showing one problem can be p-time reduced to another is frequently complex, going via a chain of reductions through intermediate problems.

The tricky nature of some of these arguments is reflected in the counter-intuitive conclusions that can be reached: it 'feels right' that $HCP \leq_p TDSP$ because the TDSP is similar to the HCP but with edge weights attached, but it certainly doesn't feel right that $TDSP \leq_p HCP$ (where do the edge lengths go?). Nevertheless it *is* true this way round as well: the TDSP and HCP are both in NPC and so it must be possible to map the TDSP into the HCP, by some -- perhaps very circuitous -- chain of p-time reductions.

There must however have been a first NPC problem whose complexity assignment wasn't achieved by p-time reduction to another NPC problem but in some other way. This was the **satisfiability problem for propositional logic (PSAT)**, shown in 1971 to be NP-complete using what has since become known as **Cook's theorem** or the **Cook-Levin theorem**.

Propositional logic and PSAT

Propositional logic is basically the same as the boolean logic used in digital circuit design. It differs from boolean logic only in some of the notations used

	Boolean (digital) logic	**Propositional logic**
TRUE	1	T
FALSE	0	$\perp$
NOT a	$\bar{a}$	$\neg a$
a OR b	a + b	$a \vee b$
a AND b	a.b	$a \wedge b$

and in the interpretation and use of the formulae.

There is one other operator used in propositional logic which is not normally used in digital logic, the **implication operator** $a \rightarrow b$. This is defined by the truth table (using the notation $\perp$ and T for 'false' and 'true', rather than digital logic's 0's and 1's):

a	**b**	**a → b**
$\perp$	$\perp$	T
$\perp$	T	T
T	$\perp$	$\perp$
T	T	T

However, occurences of $\rightarrow$ can be reduced to a more familar-looking form by noting that

$$a \rightarrow b \text{ is the same as } \underline{\neg a \vee b}$$

so there really isn't anything new here (though it is conventional to use the $\rightarrow$ symbol rather than decomposing the implication operator as above).

In digital logic the variables are the bit-values 0 and 1 (which conventionally are associated with 'false' and 'true' respectively).
In propositional logic the variables still have only these two possible values but they have a broader interpretation, for example they could be the propositions

a = 'Today is Tuesday'
b = 'The sun is shining'

A formula in propositional logic combines these basic true/false variables using the operators above to construct logical statements -- such as a $\wedge$b, interpreted in this case as 'Today is Tuesday **and** the sun is shining'.

The truth value of a formula can be established by knowing the truth values of the individual variables, and the ways that operators act to combine variables.

Propositional logic allows the use of boolean algebra to resolve logical puzzles. In this way it fulfils the original intentions of George Boole (1815-1864) when he published in 1854 the book "An Investigation of the Laws of Thought, on which are founded the Mathematical Theories of Logic and Probabilities".

PSAT, the **Propositional Satisfaction Problem**, is the problem of deciding whether there is any set of truth value assignments to n logical propositions $b_1 ... b_n$ which would allow a boolean formula $\Phi(b_1,..., b_n)$ constructed from them to itself be true.

It's not difficult to show that PSAT is in NP because its yes-instances have an easy short certificate, a set of n truth values for the $b_1 ... b_n$ that can be inserted into $\Phi(b_1,..., b_n)$ to check if Φ is then true. It's *much* harder to show that PSAT is also NP-hard; however this is what Cook's theorem effectively did (remember NPC = NP $\cap$ NP-hard).

QBF: an example of a logical decision problem in EXPTIME

Though it seems to be a very hard problem PSAT is still only in NPC, meaning that no-one has yet *proven* that all algorithms to solve it are in $O(2^n)$ or worse. A small modification of propositional logic called **quantified boolean formulas (QBF)** has however been shown to be in provably intractable, and like the Towers of Hanoi problem is in EXPTIME.

QBF is a variant of propositional logic where the quantifiers ∃ ('there exists'), ∀ ('for all') are applied to the boolean propositions, for example in the statement

∀b (b → b) or "for all b, b implies b"

Since (b → b) = (¬b ∨ b), this is true for b true *or* b false. So this QBF statement would be *true*.

Another example of a quantified formula which is true might be

∀b ∃c ((b ∨ c) = T) or "for all b there exists a c such that b or c is true"

This is solved by assigning the value to T to c (∃ means that the formula didn't have to be true for all c, just for some value of c). However

∀b ∃c ((b ∧ c) = T) or "for all b there exists a c such that b and c is true"

is *false*, because there's no way to solve it when b itself is false.

Establishing the truth of expressions like this has been demonstrated to be in EXPTIME. It's the presence of 'for all' and 'there exists' that precludes a short certificate -- as for yes-instances of propositional logic -- and which gives an $O(2^n)$ lower bound for QBF. (The difficulties are especially clear in the case of QBF statements involving 'for all' which are asserted to be true as there's obviously in this case no way to avoid checking all possible truth value assignments for the variables in the statement.)

Decision problems that are harder still

Presburger arithmetic, which is in EXP(EXPTIME), is a more elaborate logic which like QBF has the quantifiers 'there exists' and 'for all'. However in this logic it is also possible to make statements about *positive integers*, using the additional, non-logical arithmetic operators '+' and '='.

For example, the following (true) statement in Presburger arithmetic

$$\forall x \, \exists y \, \exists z \, (x+z = y) \, \& \, \exists w \, (y = w+w))$$

states that for every x there is an even y (y = w+w, for some w) that is larger than x (x+z = y, for some z), and so therefore there are an infinite number of even numbers.

And it gets even worse! There is a logical/arithmetic system called **WS1S** that allows references not just to individual integers but to *sets* of integers.

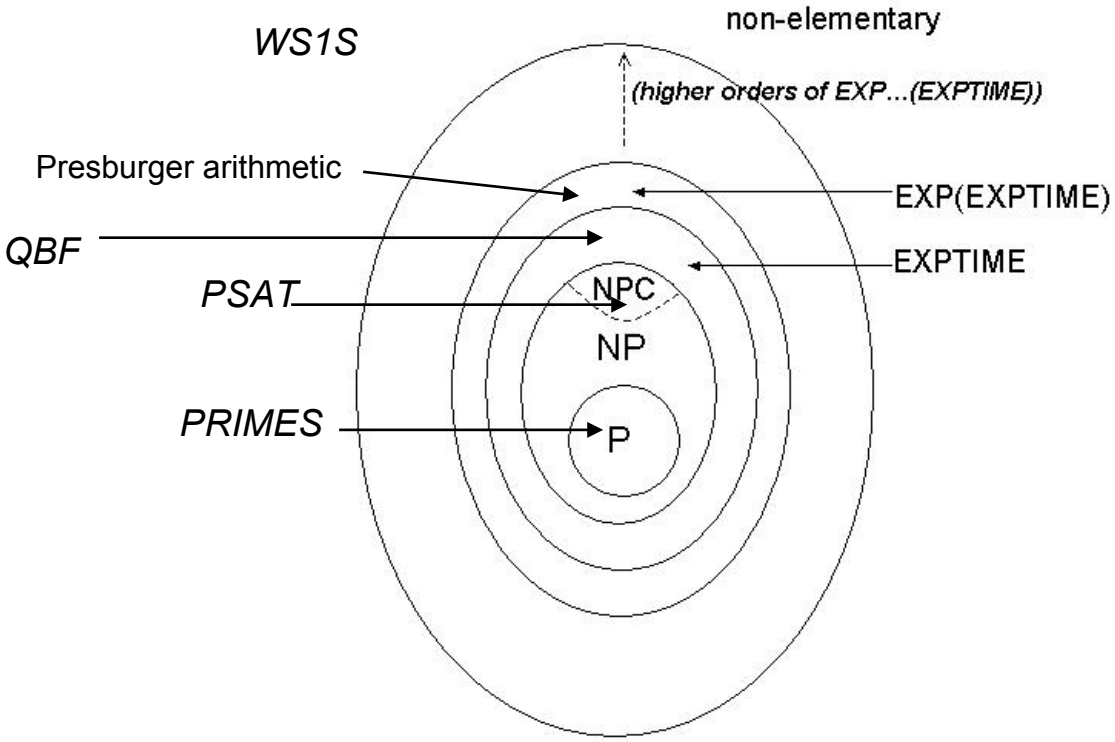
For example the following (true) statement in WS1S

$$\forall S \, (\, (0 \in S) \, \& \, \forall x \, (x \in S \rightarrow (x+2) \in S) \rightarrow \forall y \, (\exists w \, (y = w+w) \rightarrow y \in S))$$

says that 'any set S which contains 0, and which contains x+2 when it contains x, must contain all even numbers' (or 'every even number can be obtained by adding 2 to 0 some number of times').

The problem of deciding, in general, whether a WS1S statement like this is true is in the 'non-elementary' class that can't be bounded by *any* EXP(EXP(EXP...(EXPTIME))) function.

A hierarchy of complexity classes (with some decision problem examples)



What does all this really mean?

It means that as the framework within which a question can be asked becomes more sophisticated (for example, formalisms that include sets allow us to generalise the discussion to groups of things that share common properties) -- in other words, as the questions allowed become more interesting -- it becomes harder and harder to get answers we can rely on.

But the final section will show that for some questions it is impossible to guarantee any kind of answer at all.