What is NP?

NP stands for Non-deterministic Polynomial time.

Is the set of decision problems solvable in polynomial time on a non-deterministic Turing machine.

Equivalently, it is the set of problems whose solutions can be "verified" by a deterministic Turing machine in polynomial time.

Turing machines are extremely basic abstract symbol-manipulating devices which, despite their simplicity, can be adapted to simulate the logic of any computer that could possibly be constructed.

A non-deterministic (Turing) machine can be viewed as one that "branches" into many copies, each of which follows one of the possible transitions. Whereas a DTM has a single "computation path" that it follows, a NTM has a "computation tree“. Or the machine that is the "luckiest possible guesser“.

Formal definition of NP-completeness

A decision problem *B* is NP-complete if it is complete for NP, meaning that:

1. it is in NP and

2. it is NP-hard, i.e. every other problem in NP is reducible to it.

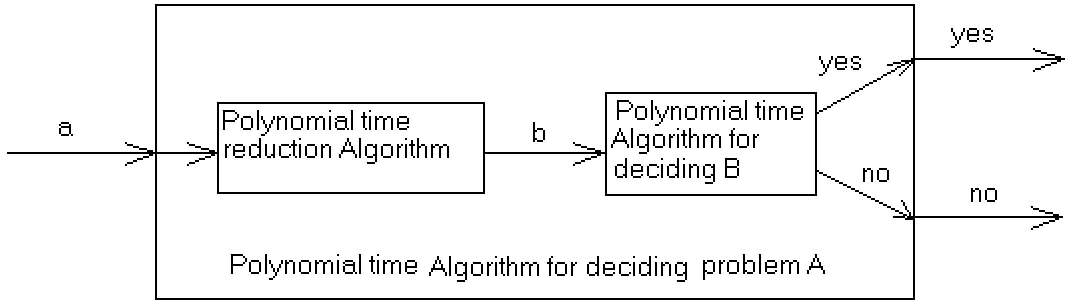
“Reducible" here means that for every problem A, there is a polynomial-time many-one reduction, a deterministic algorithm which transforms instances *a of A* into instances b *of B*, such that the answer to *b* is YES **if and only if** the answer to *a* is YES.

To prove that an NP problem *A* is in fact an NP-complete problem it is sufficient to show that an already known NP-complete problem reduces to *A*.

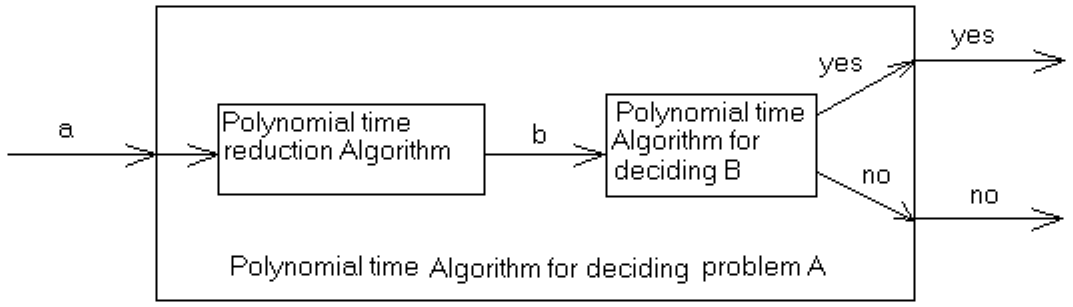
NP-completeness – Reduction

Suppose that we have a decision problem A that we would like solved in polynomial time, suppose also that we have a second decision problem B for which we already have a polynomial-time algorithm for returning a verdict of "yes" or "no“.

An algorithm for deciding problem A would exist if we had a polynomial-time algorithm for converting every instance of problem A into an instance of problem B and used the existing algorithm to decide that instance of B.


a
Polynomial time reduction Algorithm
b
Polynomial time Algorithm for deciding B
yes
no
yes
no
Polynomial time Algorithm for deciding problem A

NP-completeness – Reduction


a
Polynomial time reduction Algorithm
b
Polynomial time Algorithm for deciding B
yes
no
yes
no
Polynomial time Algorithm for deciding problem A

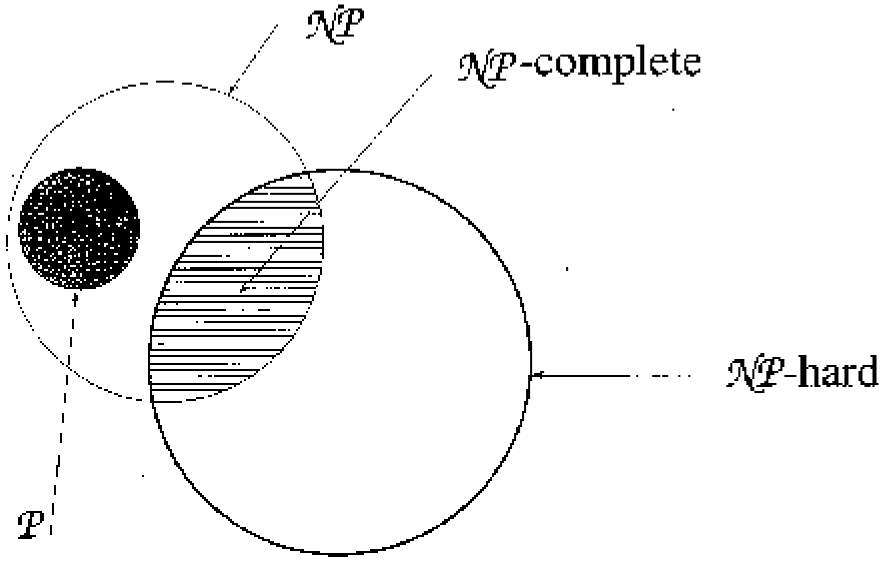
A reduction to prove a problem is NP-hard works by contradiction.

If a decision problem A is NP-hard and we assume the decision problem B is solvable in polynomial time, then the polynomial time reduction leads to the result that B is also polynomial time solvable, which is a contradiction.

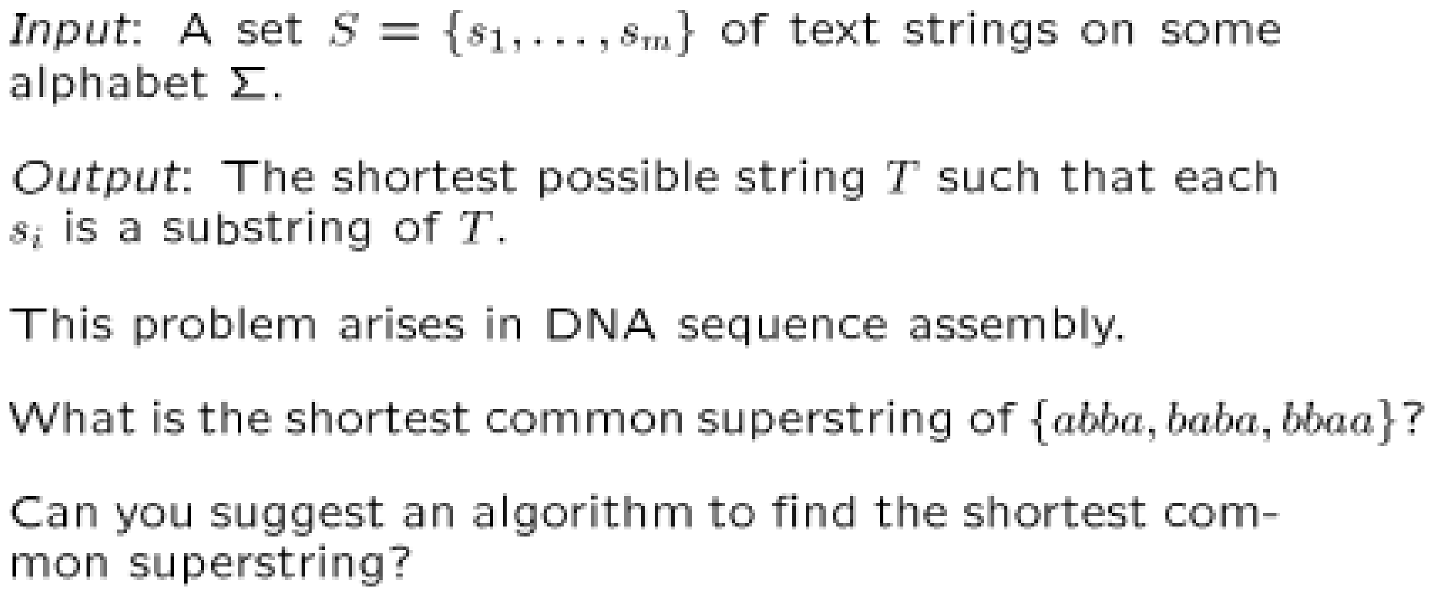
One has to be careful on the following two points:

1. The transformation takes polynomial-time
2. The answers are the same -- the answer to a is "yes" if and only if the answer to b is "yes".

Problem classes



Example: Shortest Common Superstring



Input: A set $S = \{s_1, \ldots, s_m\}$ of text strings on some alphabet Σ.

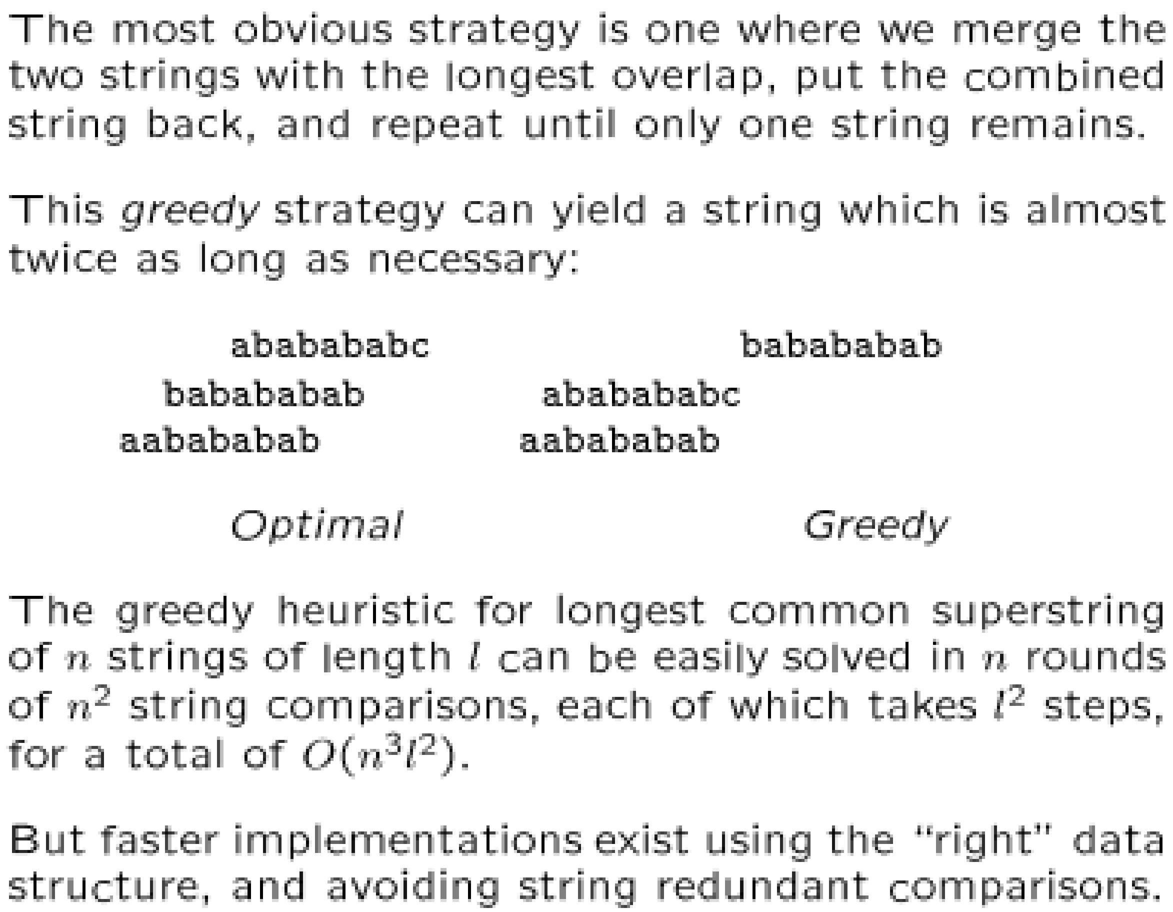
Output: The shortest possible string T such that each s_i is a substring of T.

This problem arises in DNA sequence assembly.

What is the shortest common superstring of $\{abba, baba, bbaa\}$?

Can you suggest an algorithm to find the shortest common superstring?

The greedy heuristic



The most obvious strategy is one where we merge the two strings with the longest overlap, put the combined string back, and repeat until only one string remains.

This *greedy* strategy can yield a string which is almost twice as long as necessary:

```
     ababababc              babababab
  babababab        ababababc
aabababab         aabababab

     Optimal                  Greedy
```

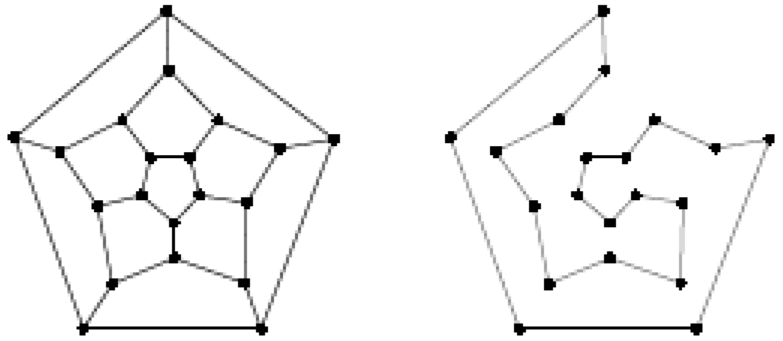
The greedy heuristic for longest common superstring of n strings of length l can be easily solved in n rounds of n^2 string comparisons, each of which takes l^2 steps, for a total of $O(n^3 l^2)$.

But faster implementations exist using the "right" data structure, and avoiding string redundant comparisons.

Hamiltonian path

The *Hamiltonian cycle* problem asks whether there is a tour using the edges of a given *graph* such that every vertex is visited exactly once.

When computer scientists talk about graphs, they mean networks of *nodes* or *vertices* where certain pairs are connected by *edges*.



The *Hamiltonian path* problem is well known to be NP-complete, even if (a) every edge is directed, (b) a particular node is designated as the start vertex, and (c) a particular node is designated as the stop vertex.

Parenthesis: How did it all start?

We prove problems to be NP-hard by reducing them to other known NP-complete problems.

But how did it all start?

The first NP-complete problem is *Boolean Satisfiability* problem.

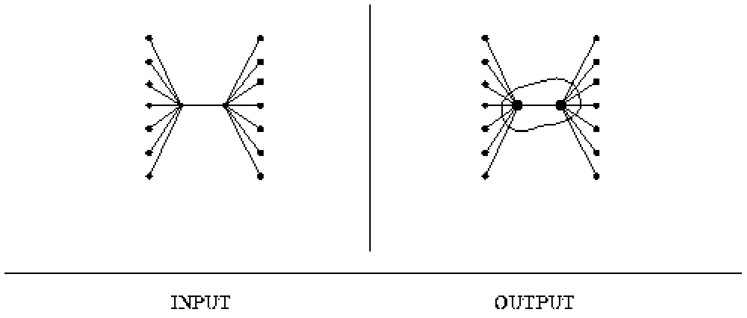
Proved by Cook in 1971 and obtained the Turing Award for this work. Also proved independently by Leonid Levin on the same year. It is known as the Cook-Levin Theorem.

Reminder: Nobody has yet been able to prove whether NP-complete problems are in fact solvable in polynomial time, making this one of the great unsolved problems of mathematics. The Clay Mathematics Institute in Cambridge, MA is offering a $1 million reward to anyone who has a formal proof that P=NP or that P≠NP.

Some basic NP-complete problems

- **3-Satisfiability :** Each clause contains at most three variables or their negations.
- **Vertex Cover:** Given a graph G=(V, E), find a subset V’ of V such that for each edge (u, v) in E, at least one of u and v is in V’ and the size of V’ is minimized.
- **Hamilton Path/Circuit:** (definition was given before)

- History: Satisfiability -> 3-Satisfiability -> vertex cover -> Hamilton circuit.
- Those proofs are very hard.
- Richard Karp proves the first few NPC problems and obtains Turing award.



TSP: Traveling salesman problem

The traveling salesman problem is probably the most notorious NP-complete problem. This is a function of its general usefulness, and because it is easy to explain to the public at large.

Problem description: Imagine a traveling salesman who has to visit each of a given set of cities by car. What is the shortest route that will enable him to do so and return home, thus minimizing his total driving?

In another way: Given a number of cities and the costs of traveling from any city to any other city, what is the cheapest round-trip route that visits each city exactly once and then returns to the starting city?

TSP is NP-complete

1. Verifying TSP is in NP is easy: Under the decision version of the problem, given a tour T and a distance D, we can verify in polynomial time that T is indeed a tour and that the total distance following T is less than D.

2. To prove TSP is NP-hard, we will reduce Hamiltonian Cycle to it.

Hamiltonian Cycle (G=(V,E))

Construct a complete weighted graph G'=(V',E') where V'=V.

```
$n = |V|$
for $i$ = 1 to $n$ do
   for $j$ = 1 to $n$ do if $(i,j)$ is an edge, then $w(i,j) = 1$ else $w(i,j) = 2$
```

Return the answer to Traveling-Salesman(G',n).

TSP is NP-complete

- The actual reduction is quite simple, with the translation from unweighted to weighted graph easily performed in linear time. Further, this translation is designed to ensure that the answers of the two problems will be identical.

- If the graph G has a Hamiltonian cycle $\{v_1,\ldots,v_n\}$, then this exact same tour will correspond to n edges in E', each with weight 1. Therefore, this gives a TSP tour of G' of weight exactly n.

- If G does not have a Hamiltonian cycle, then there can be no such TSP tour in G', because the only way to get a tour of cost n in G would be to use only edges of weight 1, which implies a Hamiltonian cycle in G.

- This reduction is both efficient and truth preserving. A fast algorithm for TSP would imply a fast algorithm for Hamiltonian cycle, while a hardness proof for Hamiltonian cycle would imply that TSP is hard. Since the latter is the case, this reduction shows that TSP is hard, at least as hard as Hamiltonian cycle.

NP-Complete problems – Approaches

You might look for efficient algorithms that solve various special cases of the general problem.

You might look for algorithms that, though not guaranteed to run quickly, seem likely to do so most of the time.

You might relax the problem, looking for a fast algorithm that merely finds designs that meet most of the component specifications.

You might want to find a “good enough” solution, instead of the best solution possible.

In short, the primary application of the theory of NP-completeness is to assist algorithm designers in directing their problem-solving efforts toward those approaches that have the greatest likelihood of leading to useful algorithms.

NP-Complete problems approaches: Examples

- Approximation: An algorithm that quickly finds a sub-optimal solution that is within a certain (known) range of the optimal one.

- Probabilistic: An algorithm that can be proven to yield good average runtime behavior for a given distribution of the problem instances—ideally, one that assigns low probability to "hard" inputs.

- Special cases: An algorithm that is provably fast if the problem instances belong to a certain special case.

- Heuristic: An algorithm that works "reasonably well" on many cases, but for which there is no proof that it is both always fast and always produces a good result.