Chapter 9: "The Theory of *NP*" Categories of problems:

I. problems with know polynomial-time algorithm(s), e.g., sorting, searching, etc.

II. problems that have been proven to have **no** polynomial-time algorithm, called *intractable*
e.g., Halting problem - input: <an algorithm, algorithm's input>
output: Yes/No will the algorithm halt?

III. problems that have not been proven to be intractable, but have no known polynomial-time algorithm. e.g., TSP, 0-1 Knapsack, graph-coloring, etc.

1. The Theory of *NP*" The Phrase the 0-1 Knapsack problem in terms of a decision problem (Yes/No answer).

2. One of the biggest open-questions in Computer Science is whether $P = NP$.

a) What would need to be done to show that $P = NP$?

b) What would need to be done to show that $P \neq NP$?

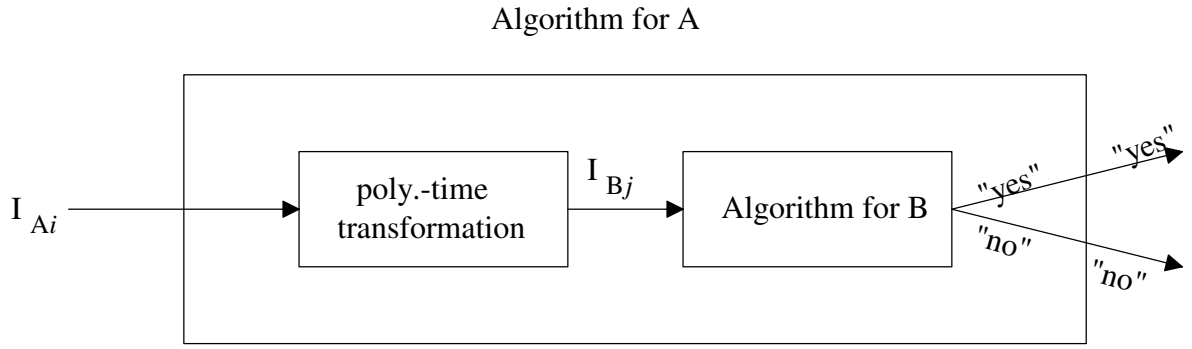
Definition of polynomial-time reducibility

Decision problem A reduces to decision problem B if any instance of problem A, say I_{Ai}, can be transformed into an instance of problem B, say I_{Bj}, such that

a) transformation time is a polynomial in the size of I_{Ai} (call it n_A),

b) the size of $I_{Bj} \leq$ polynomial w.r.t. to size of I_{Ai} , and

c) Algorithm B with I_{Bj} as input answers "yes" if and only if algorithm A with I_{Ai} as input answers "yes".



We say "A reduces to problem B," or "A$\propto$ B"

3. Assume there exists a polynomial-time transformation from decision problem A to decision problem B.

a) If we can solve B in poly. time, then how fast can we solve A?

b) If A is known to be "hard", say best worst-case algorithm of $\Theta(2^n)$, then what can we conclude about B?