Reductions

A reduction is a hypothetical algorithm. It's a way of establishing that IF Algorithm A existed THEN Algorithm B would exist too. If you're able to do this, then you've shown that "Algorithm B reduces to Algorithm A". The way you show B reduces to A is by writing a hypothetical algorithm in pseudocode following this template.

```
AlgorithmB(bInstance):
    aInstance = preprocess(bInstance)
    aSoln = AlgorithmA(aInstance)
    bSoln = postprocess(aSoln)
    return bSoln
```

In this algorithm, the pre- and post-processing steps do whatever is needed (if anything) to setup AlgorithmA to be effective in helping solve AlgorithmB.

A hypothetical algorithm like this, that shows how Algorithm A can be used to solve Algorithm B, could be used to argue: if there exists AlgorithmA, then there exists AlgorithmB. It doesn't claim that AlgorithmA or AlgorithmB exists. It only establishes a link between their existences:

∃ AlgorithmA → ∃ AlgorithmB

Here's a simple example.

```
min(array):
    sortedArray = sort(array)
    soln = sortedArray[0]
    return soln
```

This hypothetical algorithm does nothing in the preprocessing phase and does something very simple in the post-processing step. It establishes

∃ sort algorithm → ∃ min finding algorithm

because it shows that if we had the ability to sort arrays, this algorithm shows how that algorithm could be used to find minimum elements of arrays. We all know that sorting and min finding are easy, but reductions are often used to link hard or impossible problems.

Contrapositive

In your discrete math class you probably learned that $p \rightarrow q$ is logically equivalent to its contrapositive statement $\neg q \rightarrow \neg p$

This means that a hypothetical algorithm like the one at the top of the page that establishes

∃ AlgorithmA → ∃ AlgorithmB

also establishes the contrapositive

¬∃ AlgorithmB → ¬∃ AlgorithmA

which says that if we know that Algorithm B cannot exist, then the reduction shows that Algorithm A cannot exist either. This is the more important result for problems that are intractable, like those examined for NP Completeness and for the Halting Problem. If the outer problem in a reduction is impossible to solve and it reduces to an inner problem, then this contrapositive says the inner problem is impossible to solve too.

Example: Halting Problem

Let's define the Halting problem as: Given algorithm M and input x, does M ever terminate when given x? (For historical reasons, we'll output "accept" for an affirmative answer and "reject" for a negative answer.)

Also, let's say that we know the following theorem.

Theorem: There is no algorithm Accept(M,x) capable of determining whether an arbitrary program M outputs "accept" an arbitrary input x.

Such an *Accept* algorithm would output "accept" if running M on input x runs to termination and outputs "accept" and otherwise (ie, M runs to termination and outputs "reject" or if M never terminates) outputs "reject".

We can prove using a reduction that an algorithm for the Halting Problem is also impossible since *Accept*(M,x) can't exist.

```
Accept(M,x):
    if (Halt(M,x) outputs "accept" && M(x) outputs "accept")
        output "accept"
    else
        output "reject"
```

This shows that if *Accept*'s job is to determine whether M eventually outputs "accept" when given x, it can get the job done if it has *Halt* to help. First it runs *Halt* to see if M(x) would go into an infinite loop or not. If it wouldn't, then it can safely run M to see if it accepts x. If M(x) would go into an infinite loop, then M(x) would not eventually output "accept" and so *Accept* can simply reject.

This reduction establishes that

$\exists\ Halt \rightarrow \exists\ Accept$

and also establishes the contrapositive

$\neg\exists\ Accept \rightarrow \neg\exists\ Halt$

Since we know *Accept* does not exist, this reduction establishes that *Halt* does not exist either.