First, we will talk about a version of the SAT problem and prove it is NP-complete. In this version, there are two inputs: an SAT formula and an integer . The question is whether you can satisfy the formula by setting at most variables to true (and the rest of the variables to false). This problem is NP-complete because a given solution can be verified in polynomial time (just set the given variables to true and see if the whole formula is true) – this shows it is NP; and the regular SAT problem can be reduced to this version by setting to the number of total variables in the formula – thus it is NP-hard. Let’s call this version of SAT **#SAT**.

Now that we’ve shown #SAT is NP-complete, we will use it to show Monotone SAT is also NP-complete. Monotone SAT is NP because a given solution is easily verified in polynomial time (same reason as given above); and Monotone SAT is NP-hard because we can reduce #SAT to Monotone SAT in polynomial time:

# Reduction Algorithm

Given a #SAT problem with its two inputs of a formula and , turn all in the formula into a new variable in the new formula, which is not negated, and insert a clause . Run the imaginary Monotone SAT code on this new formula with the same . The answer from this is the answer for #SAT. For example:

Will be reduced to

This reduction algorithm is polynomial because you’re just going through the entire formula and creating a new, non-negated variable for each negated variable (linear, actually). Transforming the output is constant time.

# Proof of Correctness

We want to prove that the answer to #SAT(formula, k) is always the answer to MonotoneSAT(formula’, k), where is the transformed using our algorithm above.

Suppose MonotoneSAT(formula’, k) returns yes, then that means by setting at most k variables in to true. Each variable in has a corresponding variable in .