1. This problem is recognizable but undecidable.

**Proof of recognizability**: write a machine that, when given input , simulates on input and reads the state of for every step, seeing what the value of is. If in any of the steps, then return true.

**Proof of undecidability**: Let’s design a program that takes input :

M'(x){

M\_m = M, but there's a a new line of code in the very

beginning that initiates y and sets it to 0,

everything inside main is copied to another function f,

rewrite main to only have two things:

call f and then set y to 1

M\_m(x) //run M\_m on x  
}

Observe see that if halts, then modifies in the end and halts; and if doesn’t halt, then also doesn’t halt, and never gets to modify .

Suppose for now that there exists a program to rewrite into according to the above pseudocode specification.

Assume there exists a program that decides the Modifies a Variable problem.

Then there exists a program that decides the halting problem:

boolean haltChecker(string M, string x){

M' = rewrite(M);

return modifiesY(M',x);

}

This would be a correct program that decides the Halting Problem if actually could decide the Modifies a Variable Problem. This contradicts the undecidability of the Halting Problem. Thus, our assumption that exists is wrong.

1. This problem is recognizable but undecidable.

**Proof of recognizability**: write a machine that, when given input , simulates on input . If it halts, then return true.

**Proof of undecidability**: Let’s design a program that takes another program as input:

rewrite(M){

M' = M, but rewritten so that

all recursion is turned into iteration;

turn all non-boolean variables into boolean array

representations //we can do this with some rules,

//like turning integers into their binary representations

//then turning 1's into trues and 0's into falses and

//prepending [T,T,F] onto all representations of integers

return M'  
}

Suppose for now that there exists a program to rewrite to rewrite as specified by the above pseudocode specification.  
Assume there exists a program that decides the Limited Memory Halting Problem.  
Then there exists a program that decides the Halting Problem:  
boolean haltChecker(string M, string x){

M' = rewrite(M);

return limMemHalt(M',x);

}

Observe that if halts, then inside also halts, because they are the same program, just in different representations; and if doesn’t halt, then also doesn’t halt for the same reason.

This would be a correct program that decides the Halting Problem if actually could decide the Limited Memory Halting Problem. This contradicts the undecidability of the Halting Problem. Thus, our assumption that exists is wrong.

1. This problem is unrecognizable.

**Proof of unrecognizability**: Let’s design two programs, and , both of which take an input :

M1(x){

M(x);

accept;

}

M2(x){

M(x);

reject;

}

Observe that if doesn’t halt, then neither nor halts. If halts, then both and halt, but with different results.

Suppose for now there exists a program to rewrite into and according to the pseudocode specification above.  
Assume there exists an that decides the Program Agreement Problem.

Then there exists a neverHaltChecker:

boolean neverHaltChecker(M,x){

M1 = rewrite1(M,x);

M2 = rewrite2(M,x);

return agreeChecker(M1,M2,x);

}

This would be a correct program that decides the Co-Halting Problem if actually could decide the Program Agreement Problem. This contradicts the unrecognizability of the Halting Problem. Thus, our assumption that exists is wrong.

1. This problem is decidable.

**Proof of decidability**: we devise an algorithm that decides if player 1 has a winning strategy, where is the list of all variable assignments that would make true (e.g. if , then its would be , which corresponds to ).

boolean winStrat1(allTrueConfigs){ // list of all variable assignments that makes phi true

// (e.g. [[T,F], [T,T]], which means means

// [[x1 = T, x2 = F], [x1 = T, x2 = T]])

length = allTrueConfigs.length

if (length == 0){ // base case; game over and player 1 won

return true;

}

trues = list of all A ∈ allTrueConfigs where A[0] == True;

falses = list of all A ∈ allTrueConfigs where A[0] == False;

if ((∀ x ∈ trues, x[1] is the same) &&

(∀ x ∈ falses, x[1] is the same)){

// e.g. trues = [[x1 = T, x2 = F, x3 = F], [x1 = T, x2 = F, x3 = T]]

// and falses = [[x1 = F, x2 = T, x3 = F], [x1 = F, x2 = T, x3 = T]]

// then when x1 == T, then x2 = F only, and

// when x1 == F, then x2 = T only, meaning player 1 can't fend against

// player two's two available options in either case

return false;

}

else{

falseTrues = list of all x ∈ falses where x[0]==F and x[1]==T;

falseFalses = list of all x ∈ falses where x[0]==F and x[1]==F;

trueFalses = list of all x ∈ trues where x[0]==T and x[1]==F;

trueTrues = list of all x ∈ trues where x[0]==T and x[1]==T;

if (∀ x ∈ trues, x[1] is the same){ // then player 1 can't set that x to True

return winStrat1(falseTrues.sublist(2,end)) &&

winStrat1(falseFalses.sublist(2,end))

}

else if (∀ x ∈ falses, x[1] is the same){ // then player 1 can't set that x to False

return winStrat1(trueFalses.sublist(2,end)) &&

winStrat1(trueTrues.sublist(2,end))

}

else{ // then player 1 can set that x to either True or False -- for now

return (winStrat1(falseTrues.sublist(2,end)) &&

winStrat1(falseFalses.sublist(2,end))) ||

(winStrat1(trueFalses.sublist(2,end)) &&

winStrat1(trueTrues.sublist(2,end)))

}

}

}

**Proof of Algorithm Correctness**:

In order for player one to have a winning strategy then the following propositional statement has to be true:

Our algorithm can be proved inductively.

**Base case**: if is empty, then player 1 does have a winning strategy because the game is over.

**Inductive case**:

Assume the inductive hypothesis . We want to prove .

Say is odd (so it’s player 1’s turn), for all solutions where is True, if the in those solutions is all True, then player 1 cannot choose True, because then he can’t fend against if player 2 chooses False for ; likewise, if in those solutions is all False, then player one still cannot choose True, because then he can’t fend against if player 2 chooses True. The same reasoning applies to when is False.

If there’s only one option for in both and cases, then player one does not have a winning strategy; return false. If one of those choices can potentially work out for him, then choose it; e.g. if he cannot set , then he sets and sees if the two branches of gameplay resulting from that will both work out (e.g. if both and ( have winning strategies for him, then he has a winning strategy). Note that both these branches must yield winning strategies in order for him to have an overall winning strategy because only then can he fend against both options available to player 2 in .

If both options of can potentially work out for player 1, then only one has to work, i.e. the disjunction of whether the two options can yield winning strategies.