To prove a problem is decidable or recognizable you must give an algorithm that decides or recognizes the problem and proves its correctness. **To prove a problem is undecidable or not recognizable** you can use a reduction from an undecidable or unrecognizable problem. You can use that the HALTING PROBLEM is not decidable (in fact that problem NEVERHALTS of program input pairs (M, x) where M runs forever on x is unrecognizable), or any problem we proved unrecognizable or undecidable in class on Friday November 15 or in the notes. We will prove the above claim about the HALTING PROBLEM problem in class on Monday, November 18th.

Proving computability. For each of the following problems, prove whether it is decidable, recognizable but not decidable, or not even recognizable.

(i) *(8 points)* MODIFIES A VARIABLE. Given a deterministic program M, that starts by setting an integer variable $y = 0$ as the first step of the code, and only uses a variable named y in the main function. Given such a program M and an input x for this program, we ask, does program M ever modify the value of y? (This is a relevant question for security: you might want to make sure someone can't write to a protected part of memory.)

Solution: The MODIFIES INPUT problem is recognizable but not decidable.

To prove recognizable: We can write a program `modifiesVariable(String` M`, String` x`)` that simulates the execution of M on x and checks at each timestep whether y changed. If y changes at any time, return true. If M finishes without changing y, return false. This recognizes the problem: if y is ever modified, it must happen at some finite time t, so the program will halt and accept at time t. If y is never modified, it will never output true.

To prove undecidability by reduction: We reduce to the Halting Problem to our problem. Suppose we have a decider `modifiesInput(String M, String x)`. We can write `Halting(program, input)` as follows: first, make a new program `wrappedProgram(String` M`, String` x`)` in which `wrappedProgram` first sets $y = 0$ and then simulates the execution of `program` on the input x, without using or modifying the variable y at all (using a different space for computation). After this, if the program accepts or rejects x the program sets $y = 1$ (and do not overwrite anything if the program doesn't halt).

- If `program` halts on input x, then `wrappedProgram` will reach the instruction where it modifies part of y to 1. This means `modifiesInput(wrappedProgram, x)` must return true.
- If `program` never halts on input x, or rejects this input, then `wrappedProgram` will never reach the instruction where it modifies $y = 1$, so `modifiesInput(wrappedProgram, x)` must return false.

If `modifiesInput` were a program that decided MODIFIES INPUT, this routine above would decide the Halting problem, which is a contradiction. So no such decider can exist, and thus MODIFIES INPUT is undecidable.

To prove undecidability by diagonalization: Suppose we have a program `modifiesInput(program, input)`, which decided for program with a integer variable y set to 0 to start with if it gets modified. Using this program we can create the following program

```
diagonalize(program)
int y=0;
if modifiesInput(program, program) return
else y=1
```

where the second `program` refers to using the raw text of the program as input. And note that `diagonalize` is the right kind of program, the first line indeed defines an integer variable and sets it to 0.

Now we need to wonder what happens if we ran `modifiesInput (diagonalize, diagonalize)` and we'll see that there is a contradiction here. There are two cases.

(a) assuming the program `diagonalize` on input `diagonalize` does not modify it's variable y, than our code above changes y to 1, so we should return `true`. So this cannot be true

(b) assuming the program `diagonalize` on input `diagonalize` does modify it's variable y, than our code above does not changes its y, so we should return `false`. So this cannot be true

A contradiction.

(ii) *(8 points)* Limited Memory Halting. Given a deterministic program M consisting has no function calls, just boolean variables, assignment statements, and control flow statements, with a total of n statements, that only defines and then modifies a Boolean array of length c (that is doesn't use any other variables). Given M and an input x, does M halt on input x?

Solution: Limited Memory Halting is decidable. Consider a program M with n lines of code. There are only 2^c possible configurations of the Boolean array the problem is using, and only n instructions that could be running for a particular memory state, so if M on input x hasn't halted after $n2^c + 1$ time-steps, it must have repeated a combination of what line of code it was running and what the state of memory was, which means it is in some infinite loop.

A program that decides this would simulate the execution of M on x for $n2^c + 1$ time-steps maximum, checking after each timestep that no more than c memory has been used.

Alternately, the more intensive route: take a program simulator and make it write out the up to c bits of memory used so far and what line of code is being executed (i.e. keep a list of the state of memory at each time-step). If the same line of code + state of memory is seen more than once or the memory limit is exceeded, return "false". This will happen in at most $2^c n + 1$ steps, as this is the total number of possible combinations of memory configurations and lines of code. If the simulated program returns before then without exceeding the memory limit, then return "true".

(iii) *(8 points)* Program Agreement. Given two deterministic programs M and M' and one input x, do M and M' agree on input x. In other words, this problem should accept if M and M both accept with input x, both reject, or both never terminate and reject otherwise.

Solution: Program Agreement is not recognizable.

Suppose by way of contradiction that we have a program `comparePrograms(String` M`, String` M'`, String` x`)` that *recognizes* whether M and M' have the same behavior on input x. Consider the program `neverHalts(input)`, which just loops forever on any input. And now consider running `comparePrograms(program, neverHalts, input)`. This program will decide the `Halting` problem: decide if `program` halts on `input`, which we know is not decidable. This contradiction shows that the `comparePrograms` is not recognizable.

An alternate reduction from c-Halting is to modify the input program M to a new program M' by reversing the *reject* and *accept* decisions (that is, rejecting if M accepts, and accepting if M rejects). Then run `comparePrograms(M, M', input)`, which should accept if and only of M never halts on the input given.

(iv) *(8 points)* QUANTIFIED SAT. Deciding who has a winning strategy in games offer an interesting class of hard problems. Here we consider a very CS-ish game: two players play setting variables in a formula: Player 1 wants to make the formula true, player 2 wants to make it false, and they get to set variables in turn. Here is how the game is played: given a formula Φ with variables $x_1, \ldots, x_n$; player 1 sets variable x_1, then player 2 sets variable x_2, etc. In this game, player 1 has a winning strategy if he can set variable x_1 in a way, that no matter how player 2 sets variable x_2, he then can then set variable x_3, and so on ..., so that the formula becomes true. The problem of deciding if player 1 has a winning strategy is formalized by the following QUANTIFIED SAT problem.

Consider a SAT formula Φ (in conjunctive normal form) with variables $x_1, \ldots, x_n$ with an even n. The QUANTIFIED SAT problem is to decide if the following quantified version of the formula is true

$$\exists x_1 \, \forall x_2 \ldots \exists x_{n-1} \, \forall x_n \, \Phi$$

meaning that there is a setting fo x_1 such that for all settings of x_2, there is a setting of x_3, etc. that makes the formula true. For an example, for the formula

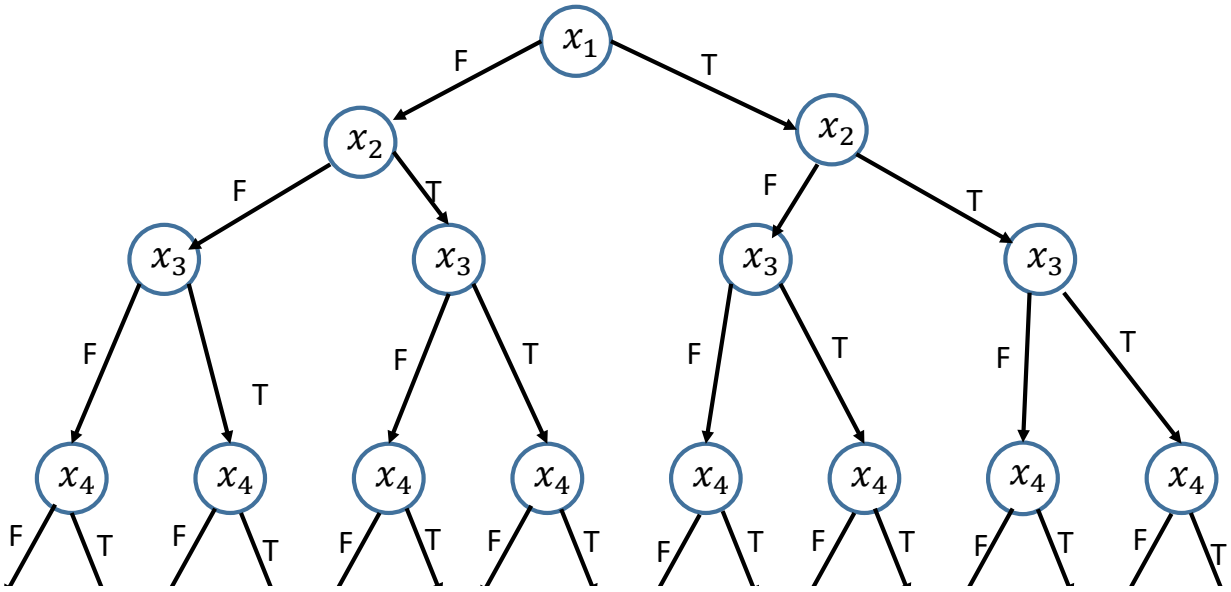
$$\Phi_1 = (x_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3) \wedge (x_4 \vee \bar{x}_4)$$

the quantified version is true: by setting $x_1 = T$, no matter what x_2 is set, there is a way to satisfy the formula. In contrast for the formula

$$\Phi_2 = x_1 \wedge (x_2 \vee x_3) \wedge (\bar{x}_3 \vee x_4)$$

is not true: we need to set $x_1 = T$, now if $x_2 = F$, then we must set $x_3 = T$, and now setting $x_4 = F$ gets the formula false.

Solution: This is decidable. There is an exponential size binary decision tree of all the options: the top level is the decision about variable x_1, at level i the decisions about level x_i, as shown on the figure below



At the bottom level of the tree all variables are set, and with each setting we can decide if the formula is true or false. We can then propagate the decision up the tree: at an even level the formula is true if both decisions at one level down lead to true formulas, while at odd levels (when

player 1 gets to choose), the formula is true if one of the two decisions at one level down lead to true formulas.