(a)

This problem is undecidable

Proof. Suppose we have some program ModifiesInput that decides this problem. We can write a ZeroChecker as follows: For any input program M to the ZeroChecker, we first create a slightly modified version of the program $M'(x)$:

```
def MPrime(x):
    res = M(0)
    return res
```

Then ZeroChecker can be constructed as follow:

```
def ZeroChecker(M):
    res = False
    return ModifiesInput(MPrime, res)
```

Basically when we put a program M in to ZeroChecker, we assign **False** to a variable res, then we execute ModifiesInput with the input M and res. If it returns **True**, ZeroChecker also returns **True**, and vice versa.

- If ModifiesInput returns **True**, it means that res has been modified by M'. As we dig into the source code of M', we noticed that the only code that can modify res is $res = M(0)$, which indicates that M(0) has halted and returned **True** (i.e., accepted). Hence, ZeroChecker should return **True**.
- If ModifiesInput returns **False**, it means that res has not been modified by M'. which means that either $M(0)$ is rejected (in this case res is still **False**, unchanged), or it runs into infinite loop (in this *case* res can never be modified). Hence, ZeroChecker should return **False**.

This program for the ZeroChecker decides the Zero Input Checker problem, so $\mathcal{L}_{\text{ZeroInput}} \leq \mathcal{L}_{\text{ModifiesInput}}$. This cannot exist, so this problem is undecidable.

(b)

This problem is decidable

Algorithm. Since we want to know whether $M(X)$ halts using less than c bits of memory, we just limit the available memory to $c-1$ bits. For those $c-1$ bits, there are 2^{c-1} possible states in total. So as we running program M using input x, we can actually keep track the state of the memory. Then the algorithm goes as follow: while running $M(x)$, either after finite time $M(X)$ halts (return **True**), or after finite time the memory revisits a state (return **False**), or we run out of memory (return **False** as well).

Proof of Correctness. We know that for Halting Problem, we assume that we are running the program on a computer with infinite memory (or, Turing Machine). However, for real world where the memory is limited, this is not the case. For the limit of $c-1$ bits of memory, there are only 2^{c-1} possible states in total. So when running the program, there will be only 3 possible scenarios:

- The program will halt. According to the definition of halting, it takes finite time to be decided whether the input is accepted (return **True**) or rejected (return **False**). Either way, it takes finite time, and will use less than c bits of memory since we have limited the available memory to $c-1$ bits.
- The program will run out of memory. No matter whether the program will halt using more than $c-1$ bits, as it ran out memory, we know it cannot halt using less than c bits. This also takes finite time since each single execution of an instruction is considered to take finite time (if it is calling into a function we also consider the number of instructions in that function).
- The program will run into infinite loop. For running deterministic program on a limited memory computer, this is equivalent as the memory revisits a state. Since we only have 2^{c-1} possible different states, we can go over all the states in finite time. Also, for a deterministic program, the next state is determined entirely by the previous state. Hence, if the machine revisits a state, we know that it will loop again and again and forever. This takes the time of updating 2^c states at most, so it is also finite time. Hence, this problem is decidable

(c)

This problem is undecidable

Proof. Suppose we have some program ProgramAgreement that decides this problem. We can write a HaltChecker as follows: For any input program M to the HaltChecker, we first create two simple functions $M_1(x)$ and $M_2(x)$:

```
def M1(x):
    return True

def M2(x):
    return False
```

Then HaltChecker can be constructed as follow:

```
def HaltChecker(M, x):
    if ProgramAgreement(M, M1, x) == False and
       ProgramAgreement(M, M2, x) == False:
        return False
    else:
        return True
```

Basically when we put a program M and its input x into HaltChecker, we then check the condition that whether "M, M1 disagree and M, M2 also disgree", if it is the case, we say $M(x)$ cannot halt (return **False**); if not, we say $M(x)$ halts (return **True**).

- If ProgramAgreement on $M, M1, x$ returns **False**, and on $M, M2, x$ also returns **False**, then we know that the output of $M(x)$ is neither the same as $M1(X)$, nor the same as $M2(X)$. As we constructed $M1$ and $M2$, we already know that $M1(x)$ will always be **True** and $M2(x)$ will always be **False**. Hence, $M(x)$ is neither **True** or **False**, which means it cannot halt. Hence, HaltChecker should return **False**.
- For all the remaining cases, there will be only 2 possibilities: $M, M1, x$ returns **True** while $M, M2, x$ returns **False** (in this case M accept x); $M, M1, x$ returns **False** while $M, M2, x$ returns **True** (in this case M reject x). Either way, $M(x)$ halts. Note that they cannot be both **True** since a program cannot both "accept" and "reject" an input (it is deterministic program). Hence, HaltChecker should return **True**.

This program for the HaltChecker decides the Halting problem, so $\mathcal{L}_{\text{Halt}} \leq \mathcal{L}_{\text{ProgramAgreement}}$. This cannot exist, so this problem is undecidable.

(d)

This problem is undecidable

Proof. Suppose we have some program Accepts1Input that decides this problem, we can write a haltChecker as follows: For any input program M to the HaltChecker, we first create a slightly modified version of the program M' (Note that M' cannot be called without x being predefined):

```
def MPrime(y):
    M(x)
    return True
```

Then the HaltChecker can be constructed as follow (we copy the source code of M' here to make it more understandable):

```
def HaltChecker(x):

    def MPrime(y):
        M(x)
        return True

    return Accept1Input(MPrime)
```

This proof is very similar to the ZeroInput one. For M', we basically just ignore the input, run M using the input of HaltChecker, and return **True**. Then we warp M' and put it into Accept1Input as an input, and finally return the result of $Accept1Input(M')$.

- If Accept1Input returns **True**, $M(x)$ halts, since if $M(x)$ cannot halt, M' would not accept anything (which would not accept at least one input x that has a 1 in the input string as well). Hence, HaltChecker should return **True**.
- If Accept1Input returns **False**, $M(x)$ cannot halt, since if $M(x)$ halts, M' would accept anything ("it would accept at least one input x that has a 1 in the input string" also works since actually all of them are accepted), so Accept1Input should returns **True**, which is a contradiction. Hence, $M(x)$ halts, and HaltChecker should return **True**.

This program for the HaltChecker decides the Halting problem, so $\mathcal{L}_{\text{Halt}} \leq \mathcal{L}_{\text{Accept1Input}}$. This cannot exist, so this problem is undecidable.

(e)

This problem is decidable

Algorithm. Since n is a finite number, we can actually list out all the 2^n possibilities of these n values. Then for those 2^n possibilities, we list out which combinations make Φ **True** (Player 1 will win), and which combinations make Φ **False** (Player 2 will win). Then we start from x_1: when we assign x_1 to **True**, we check the set that has the all combinations of Player 2 will win (We name it $Player2WinSet$). If there is no combination that has x_1 ==**True**, we stop the algorithm, and say we have found the wining strategy for player 1. If it is not the case, we assign x_1 to **False**, and check if there is no combination that has x_1 == **False**. If this is the case, we also stop the program, saying we found such a wining strategy. The thing is no matter what x_1 is assigned, for that assignment, if we cannot find corresponding combinations exist in the $Play2WinSet$, we conclude that we found such an strategy.

What if there are combinations in $Player2WinSet$ that has both x_1 == **True** and x_1 == **False**? We then remove all x_1 and x_2 in $Player2WinSet$, and do the same thing for x_2 (now treat x_2 as x_1) in the set that has the all combinations of Player 2 will win (We name it $Player1WinSet$). However this time, if the condition is satisfied, we stop the algorithm and conclude that we cannot found the strategy. If not, we delete x_1 and x_2, and move to x_3 in $Player2WinSet$. (x_{odd} is checked in $Player2WinSet$ and x_{even} is checked in $Player1WinSet$) We do this all the way to x_{n-1}. This time if the condition is satisfied, we conclude that we found the strategy; If not, we conclude that we could not find such a strategy.

Proof of Correctness. We first prove this algorithm takes finite time: The worst case that this QUANTIFIEDSAT returns **False** (i.e., we cannot find such a strategy), which means we have check the variables from x_1 all the way to x_{n-1}. But this is not forever. First $Player2WinSet$ can be listed using $O(2^n)$. For each check, it takes $O(2^n \times 2) = O(2^{(n+1)})$. Since there are at most $n-1$ checks, we have time complexity of total checking $(n-1) \times O(2^(n+1)) = O(n2^n)$. Hence the total time complexity is $O(2^n) + O(n2^n) = O(n2^n)$. Though it is exponential, still finite (in fact it should be much less since we have kept deleting the variables in the set all the way).

Now we prove that this algorithm always works. Let's say if for some point we set x_t to be True and satisfy the condition as we stated in the algorithm. Before it comes to x_t since both winning set of 2 players have combinations that makes themselves win based on the current condition, they just choose it so the game is not over. However this time, when the rest of the wining set in the "enemy" has not the winning combinations based on the current condition (that's what we have defined in the algorithm), the player who assign x_t will be guaranteed to win. So if t is an odd number, player 1 will win, which means we found the strategy; if it is an even number, then player 2 will win, which means we could not find the strategy. If the algorithm did not stop in the middle, then we go all the way to $n-1$. Then it is basically the same case as $n = 2$, where we just check if player 1 will win if he assign x_{n-1} to **True** or **false** by checking $Player2WinSet$ which only has x_n left at this time.