(i) This problem is recognizable but undecidable.

**Proof of recognizability**: write a machine $V$ that, when given input $(M, x)$, simulates $M$ on input $x$ and reads the state of $M$ for every step, seeing what the value of $y$ is. If $y \neq 0$ in any of the steps, then return true.

**Proof of undecidability**: Let's design a program $M'$ that takes input $x$:

```
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 $M(x)$ halts, then $M'$ modifies $y$ in the end and halts; and if $M(x)$ doesn't halt, then $M'(x)$ also doesn't halt, and never gets to modify $y$.

Suppose for now that there exists a program to rewrite $(M, x)$ into $(M', x)$ according to the above pseudocode specification.
Assume there exists a program $modifiesY$ that decides the Modifies a Variable problem.
Then there exists a program $haltChecker$ 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 $modifiesY$ actually could decide the Modifies a Variable Problem. This contradicts the undecidability of the Halting Problem. Thus, our assumption that $modifiesY$ exists is wrong.

(ii) This problem is recognizable but undecidable.

**Proof of recognizability**: write a machine $V$ that, when given input $(M, x)$, simulates $M$ on input $x$. If it halts, then return true.

**Proof of undecidability**: Let's design a program $rewrite$ that takes another program $M$ 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 $modify(M)$ to rewrite $M$ as specified by the above pseudocode specification.
Assume there exists a program $limMemHalt$ that decides the Limited Memory Halting Problem.
Then there exists a program $haltChecker$ that decides the Halting Problem:

```
boolean haltChecker(string M, string x){
    M' = rewrite(M);
    return limMemHalt(M',x);
}
```

Observe that if $M(x)$ halts, then $M'(x)$ inside $haltChecker$ also halts, because they are the same program, just in different representations; and if $M(x)$ doesn't halt, then $M'(x)$ also doesn't halt for the same reason.

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

(iii) This problem is unrecognizable.

**Proof of unrecognizability**: Let's design two programs, $M_1$ and $M_2$, both of which take an input $x$:

```
M1(x){
    M(x);
    accept;
}

M2(x){
    M(x);
    reject;
}
```

Observe that if $M(x)$ doesn't halt, then neither $M_1(x)$ nor $M_2(x)$ halts. If $M(x)$ halts, then both $M_1(x)$ and $M_2(x)$ halt, but with different results.

Suppose for now there exists a program to rewrite $(M, x)$ into $(M_1, x)$ and $(M_2, x)$ according to the pseudocode specification above.
Assume there exists an $agreeChecker$ 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 $agreeChecker$ actually could decide the Program Agreement Problem. This contradicts the unrecognizability of the Halting Problem. Thus, our assumption that $agreeChecker$ exists is wrong.

(iv) This problem is decidable.

**Proof of decidability**: we devise an algorithm $winStrat1(allTrueConfigs)$ that decides if player 1 has a winning strategy, where $allTrueConfigs$ is the list of all variable assignments that would make $\phi$ true (e.g. if $\phi = (x_1 \vee x_2)$, then its $allTrueConfigs$ would be $[[T, F], [F, T], [T, T]]$, which corresponds to $[[x_1 = T, x_2 = F], [x_1 = F, x_2 = T], [x_1 = T, x_2 = T]]$).

```
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:

$$\exists x_1 \; \forall x_2 \; \exists x_3 \; \forall x_4 \ldots \exists x_{n-1} \forall x_n \; \phi$$

Our algorithm can be proved inductively.

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

**Inductive case**:

Assume the inductive hypothesis $h(i+2) = \exists x_{i+2} \; \forall x_{i+3} \ldots \exists x_{n-1} \forall x_n \; \phi$. We want to prove $h(i)$.

Say $i$ is odd (so it's player 1's turn), for all solutions where $x_i$ is True, if the $x_{i+1}$ 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 $x_{i+1}$; likewise, if $x_{i+1}$ 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 $x_i$ is False.

If there's only one option for $x_{i+1}$ in both $x_i = True$ and $x_i = False$ 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 $x_i = True$, then he sets $x_i = False$ and sees if the two branches of gameplay resulting from that will both work out (e.g. if both $(x_i = False, x_{i+1} = False, x_{i+2} = \cdots)$ and $(x_i = False, x_{i+1} = True, x_{i+2} = \cdots)$ 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 $x_{i+1}$.

If both options of $x_i$ 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.