1. Each of the three functions is computable.
   - Having a counter for the number of 1s minus the number of 0s encountered thus far, and checking that the counter is never negative, works for implementing f. (Note that something like this is required for parsing programs in most programming languages, e.g. thinking of 1 as an opening parenthesis and 0 as a closing parenthesis.)
   - In an earlier lecture, we have given a (functional) program for computing $A(x, y)$, so g is computable. (As noted, although A is computable, it is not primitive recursive, i.e. to define it a definition scheme more powerful than primitive recursion is needed.)
   - Either the TM M writing a 1 on the tape and then halting, or the TM M' writing a 0 on the tape and then halting, computes h, depending on whether the answer to the open problem is yes or no, respectively. (Although currently we don't know which of M and M' is the right one, one of them surely is.)

2. - To show $MP \leq M_2P$. Define $f(M\#x) = M\#x\#x$. f is obviously computable (just duplicate the string the string after the #). Then $M\#x \in MP$ iff M accepts x iff (M accepts x and M accepts x) iff $M\#x\#x \in M_2P$ iff $f(M\#x) \in M_2P$. (Knowing/assuming MP is not recursive, it follows neither is M_2P.)
   - To show $M_2P \leq MP$. Define $g(M\#x\#y) = M'\#y$, where M' is a TM that first behaves as M on x and (only) if that results in acceptance it continues and subsequently behaves as M.[1] g is obviously computable. Then $M\#x\#y \in M_2P$ iff M accepts x and M accepts y iff M' accepts y iff $M'\#y \in$ MP iff $g(M\#x\#y) \in$ MP, where the second iff is proven by distinguishing cases on whether or not M accepts x: if it does then M' accepts y iff M accepts y, and if it does not then neither does M' accept y, so that indeed M' accepts y iff M accepts x and y. (Knowing/assuming M_2P is not recursive, it follows neither is MP.)

3. - Suppose $P \leq Q$ and $Q \leq R$. That is, there are computable functions f and g such that $x \in P$ iff $f(x) \in Q$ and $y \in Q$ iff $g(y) \in R$, say computed by TMs M and N, respectively. Define $h = f\,;g$, the composition of f and g, i.e. $h(x) = (f\,;g)(x) = g(f(x))$. Then $x \in P$ iff $f(x) \in Q$ iff $g(f(x)) \in R$ iff $h(x) \in R$. To see that h is computable, i.e. that there is a total TM K that for input x writes $g(f(x))$ on the tape, define the TM K to first run M and then run N. By assumption on M, running it on input x halts and leaves $f(x)$ on the tape, so subsequently running N halts by assumption on N and leaves $g(f(x))$ on the tape.
   - We take a DFA with four states q_0, q_1 q_2, q_f with the idea being that the DFA is in state q_i for $i \in \{0, \ldots, 2\}$ iff no three consecutive 1 have been encountered yet and i is the number of 1s the string read thus far ends in, and in (final) state q_f iff three consecutive 1s have been encountered already. The initial state is q_0 and the transition function δ is

[1] One can think of M' as being constructed 'by partial application': M' can be obtained from M by first constructing a machine M_2 that runs M on a first input and if that accepts runs M on a second input, and partially applying M_2 on x (yielding a TM taking one input only).

defined by the following table:

δ	q_0	q_1	q_2	q_f
0	q_0	q_0	q_0	q_f
1	q_1	q_2	q_f	q_f

4∗ To show $HP \leq UHP$. Define $h(M\#x) = M'$ where M' is a TM behaves as M does on x (that is, it ignores any input that is on the tape). h is easily seen to be computable (by a program that first overwrites whatever is on the tape with x and subsequently runs M). Then $M\#x \in HP$ iff M halts on x iff M' halts on all inputs iff $h(M\#x)$ halts on all inputs iff $h(M\#x) \in UHP$. (Knowing/assuming HP is not recursive, it follows neither is UHP.)

5∗ To show the diagonal language $d = \{x \in \{0,1\}^* \mid M_x \text{ accepts } x\}$ is recursively enumerable, is to give a TM M such that $L(M) = d$. A TM M can be constructed that first transforms x into $x\#x$ and then runs the universal TM U on that input, simulating running M_x (the TM corresponding to code x) on x. The coding of TMs should be simple enough that we indeed can decide whether x is the code of some TM and, if so find and execute its instructions, and if not, to replace it with some arbitrary, fixed, TM. The coding suggested on the slides of lecture 12 satisfies these conditions.