1. Why do we need to study decidable and recursively enumerable (r.e.) sets?

   It is intuitive to use sets, and this can make it easier to think about or prove an algorithm in terms of the unions and intersections of sets.

   We can reformulate problems in terms of r.e. and decidable sets to do this.
2. Is the intersection of two r.e. sets always r.e.? If yes, prove it, if no, provide a counterexample.

Yes. We can recursively enumerate the set \( A \cap B \) with this algorithm:

1. run \( A - \text{alg} \) for 1 moment
2. run \( B - \text{alg} \) for 1 moment
3. print \( n \), generated by both \( A - \text{alg} \) and \( B - \text{alg} \)

Because \( A \) is r.e., all elements \( n \in A \) will be produced, and because \( B \) is r.e., all elements \( n \in B \) will be produced as well. By running the algorithms for \( A \) and \( B \) in parallel, all elements of both sets will be compared to produce \( n \in A \cap B \).
3. Is the difference $A - B = \{x: x \text{ is in } A \text{ and } x \text{ is not in } B\}$ between two r.e. sets always r.e.? If yes, prove it, if no, provide a counterexample.

No. To produce $A - B$ we need to know if an element $n \in A$ is an element of set $B$. While $B$ is r.e. it may not be decidable.

For example, if $B = H$, $B$ is not decidable, and therefore we cannot

case.
4. Prove that it is not possible, given a program that always halts, to check whether this program always computes $2 \times n$.

Theorem: No algorithm is possible that, given a program $p$ that always halts, checks whether $p$ always computes $2 \times n$.

\[ \forall n \left( p(n) = 2n \right) \]

Proof by contradiction. Let's assume that we have a $2n$ checker:

\[
2n\text{ checker}(p) = \begin{cases} 
1 & \text{if } \forall n(p(n) = 2n) \\
0 & \text{if } p\text{ always halts and } \exists n(p(n) \neq 2n) 
\end{cases}
\]

Let's build a zero checker program out of it:

```java
public static int p(int n) {
    return q(n) + 2 * n;
}
```

Now we apply $2n$ checker:

\[ \forall p(p(n) = 2n) \]

\[ \Rightarrow \forall n(q(n) + 2n = 2n) \Rightarrow \forall n(q(n) = 0) \]

Because zero checkers do not exist, this means that $2n$ checkers are not possible.
5. Design a Turing machine that computes $2 \times n$ in binary code. Trace this machine on the example of $n = 101_2$.

\[2 \times n \text{ in binary is like right shift once}\]

\[
\begin{array}{c}
\text{start,} - \rightarrow \text{set0, R} \\
\text{set0, 0} \rightarrow 0, \text{mov0, R} \\
\text{set0, 1} \rightarrow 0, \text{mov1, R} \\
\text{set0,} - \rightarrow \text{left, L} \\
\text{mov0, 0} \rightarrow 0, \text{mov0, R} \\
\text{mov0, 1} \rightarrow 0, \text{mov1, R} \\
\text{mov0,} - \rightarrow 0, \text{left, L} \\
\text{mov1, 0} \rightarrow 1, \text{mov0, R} \\
\text{mov1, 1} \rightarrow 1, \text{mov1, R} \\
\text{mov1,} - \rightarrow 1, \text{left, L} \\
\text{left, 0} \rightarrow 1 \\
\text{left, 1} \rightarrow 1 \\
\text{left,} - \rightarrow \text{halt}
\end{array}
\]
6. Use a general algorithm for a Turing machine that represents composition to transform your design from Problem 2 into a Turing machine for computing $f(f(n)) = 4 \times n$. 

```
start_0, _ → g_{00}, R
g_{00}, 0 → R
g_{00}, 1 → R
g_{00}, _ → 0, return_0, L
return_0, 0 → L
return_0, 1 → L
return_0, _ → start_1
```

```
start_1, _ → g_{11}, R
g_{11}, 0 → R
g_{11}, 1 → R
g_{11}, _ → 0, return_1, L
return_1, 0 → L
return_1, 1 → L
return_1, _ → halt
```
7. Give a formal definition of feasibility. Give two examples:

- an example when an algorithm is feasible in the sense of the formal definition but not practically feasible, and
- an example when an algorithm is practically feasible, but not feasible according to the formal definition.

These examples must be different from the examples that we had in class.

An algorithm $A$ is feasible if $T_A^w(n) \leq P(n)$ for some polynomial $P$, where $T_A^w(x)$ is the time that algorithm $A$ takes on data $x$ and $T_A^w(n) = \max_{x: \|x\|_w = n} T_A(x)$.

**Example**

Feasible by definition only: algorithm that takes time $10^{20}$.

It is constant, but requires more computational steps than can be performed ever.

**Example**

In practice feasible, but not by definition:

$\exp(10^{20} n)$, exponential, but the exponent is small for $n < 10^{20}$. 
8. What is P? NP? NP-hard? NP-complete? Brief definitions are OK. What do we gain and what do we lose when we prove that a problem is NP-complete? Explain one negative consequence (what we cannot do) and one positive one (what we can do).

- **P** is the class of problems that can be solved in polynomial time.
- **NP** is the class of problems for which we can check a possible solution in polynomial time.
- A problem is NP-hard if every problem from NP can be reduced to this problem (problem doesn't have to be NP).
- A problem from NP is called NP-complete if every problem from NP can be reduced to this problem.

**Positive Consequence**

If we come up with a good algorithm to find the solution of an NP-complete problem, we can use the algorithm to find the solution of any NP problem.

**Negative Consequence**

If a problem is NP-complete, it means it's in NP, so we do not know if we can find a solution to it in polynomial time as $P \neq NP$ is still an open problem.
9. What is propositional satisfiability? Give an example. Explain why this problem is important in software testing.

Propositional satisfiability is the problem of determining the values of the variables $v_1, \ldots, v_n$ for which a boolean expression $B(v_1, \ldots, v_n)$ is True and False.

It is important in software testing so that we can test the execution of all the branches of our code, and discover unreachable branches.

**Ex.** if $c = a \land b$ then {
    [...]
}

What are the values of $a$ and $b$, for which this code block will be executed?
10. Step-by-step, apply the general algorithm to translate the following formula into DNF and CNF:

\( a \leq b \).