1. Consider the SUBSET SUM problem: given a set of positive integers $X = \{x_1, \ldots, x_n\}$ and a positive integer T, we ask if there exists a subset $X' \subseteq X$ such that

$$\sum_{x \in X'} x = T.$$

SUBSET SUM is NP-complete. Note that as stated this is a decision problem. The answer is yes or no rather than a list of numbers.

Show that if P=NP then you can use the polynomial time solution to the decision problem to give a polynomial time solution to the problem of actually finding the subset. Present pseudo-code describing the algorithm and justify that it will have polynomial running time by giving an upper bound for its execution time under the assumption that the decision problem can be solved in $O(p_d(n))$ time.

- Let S be an empty set
- While $T \neq 0$ and $|X| \neq 0$
  - remove an x_i from X where $0 \leq i \leq |X|$
  - solve the decision problem using X and $T - x_i$
  - if the answer is yes
    - add x_i to S
    - set $T = T - x_i$
- The set S is a valid subset if one exists, and empty if no sum exists

This algorithm runs in polynomial time because the while loop will only run at most $|X|$ times. For every run, it will solve the decision problem. Thus, the runtime of this algorithm is $O(|X|p_d(n)) \in O(p_d(n))$

2. The diagram shown in Figure 2 is a non-deterministic Turing machine that solves the subset sum problem in non-deterministic polynomial time. I will use this machine as a motivating example when presenting the proof of the Cook-Levin Theorem showing that the 3-CNF satisfaction problem is NP-complete. Accordingly, I thought it might be useful for you to explore this Turing machine a bit before we discuss the proof.

   The input this machine expects is a value N encoded in binary and surrounded by a pair of dollar signs ($) followed by a list of integer values also encoded in binary but separated from one another and terminated by number signs (#). Thus, the input $1001$101#100#11#10# would represent the subset-sum question "Is there a sublist of the numbers (5, 4, 3, 2) that adds up to 9?"

   In this figure, a notation of the form $0/1/\# \rightarrow 0/1/\#, L$ on a transition is meant as a shorthand meaning that the transition can be taken on any of the inputs 0, 1 or # and that the symbols written if the transition is taken are 0, 1, and # respectively. That is, this particular example says that on any of the symbols listed, the machine can move one square to the left while leaving the previous tape cell unchanged.

   This machine is fairly simple, but it does not employ the most obvious algorithm to solve the problem. Unfortunately, like all too many programmers before me, I failed to include good comments explaining the algorithm the Turing machine shown in the figure uses.

   The "obvious" algorithm is to first randomly cross out some list of the numbers provided by replacing their digits with number signs (my non-obvious algorithm starts by performing exactly this process). Then, the obvious way to verify that the random guesses made generated a correct solution would be to repeatedly subtract one from the number between the dollar signs and from one of the numbers that were not crossed out initially to verify that this process leads to a situation in which all of the non-crossed out numbers get reduced to zeros at the same point that the number between the dollar signs reaches zero.

   (a) Help me out! Generate the missing documentation. That is, please give a brief, informal description of the algorithm implemented by the Turing machine in Fig. 2 and justify its correctness.

   (b) Analyze the running time of my Turing machine and give a bound on its worst-case running time as a function of the size of its input string.

   (c) Explain why I used this Turing machine as my lecture example rather than the "obvious" one. It turns out that although the obvious one might have been easier to implement, it would not have been an appropriate example.

(a) The top four states takes care of guessing the subset of numbers that are part of the sum that add up to the target. It does this by skipping the target number and guessing which numbers to keep denoted by the K and which to get rid of denoted by the Z. It keeps numbers by just moving to the pound sign to the right of the number, and it removes a number by replacing every binary digit with a #.
This machine scans for odd numbers in the guessed sublists and makes them even by subtracting one. It also subtracts one from the target to make sure that the sum of the sublist still equals the target. Once all numbers are even, it divides them all by 2.
Once the machine reaches the end of the input it integer divides all numbers by 2. An odd number is divided by 2 by placing an @ on the right-most bit, then subtracting one from the target, and replacing the @ with a #. Even numbers are divided by replacing the right-most zero with a #. It does this until all the numbers in the sublist are even. Once this happens, it checks that the target is even and divides it by 2. If the target still has more digits to process, the machine begins the next right to left scan. On the other hand, it scans to the right to make sure that all other digits have also been deleted. If so, the input is accepted.

(b) Run Time: $O(n^2)$.
The steps taken by states $L, M,$ and R take $O(2n)$ steps. Subtracting 1 from both the target and a value in the sublist requires a scan from the first 1 to the right of the tape to the left-most digit from the target. This is at most $|X|$ steps. The machine must then travel all the way back to the @ which is another $|X|$ steps away at most. This is done for all odd numbers which is at most $|X|$ to find, hence the $O(n^2)$ runtime.
The same runtime applies to dividing the target value by 2, as it has to move to the right of the tape which takes $O(n)$ time. This occurs $O(n)$ times for a total of $O(n^2)$.

3. Complete problem 7.26 in Sipser:

   Let ϕ be a 3CNF-fomula. An $\neq$-**assignment** to the variable of ϕ is one where each clause contains two literals with unequal truth values. In other words, an $\neq$-assignment satisfies ϕ without assigning three true literals in any clause.

   (a) Show that the negation of any $\neq$-assignment to ϕ is also an $\neq$-assignment.

   (b) Let $\neq SAT$ be the collection of 3CNF-formulas that have $\neq$-assignments. Show that we obtain a polynomial time reduction from $3SAT$ to $\neq SAT$ by replacing each clause c_i

   $$(y_1 \vee y_2 \vee y_3)$$

   with the two clauses

   $$(y_1 \vee y_2 \vee z_i) \text{ and } (\overline{z_i} \vee y_3 \vee v)$$

   where z_i is a new variable for each clause c_i and v is a single additional new variable.

   (c) Conclude that $\neq SAT$ is NP-complete. (Hint: Given parts (a) and (b) this step should be trivial.)

   Note: This problem depends on material we will not cover in class until Monday 11/27.

   (a) Let $\bar{\phi}$ be the negation of an $\neq$-assignment to the variable ϕ. We know that ϕ has at least one false literal for every clause. Thus, $\bar{\phi}$ must have at least one true literal for every clause. If this literal is not negated, then the clause is true. Also, we know that ϕ has at least one true literal for every clause, so $\bar{\phi}$ must have at least one false literal for every clause. Therefore, if this literal is negated then the clause must be true. $\bar{\phi}$ is also a valid $\neq$-assignment since its literals are just a negation of another $\neq$-assignment. We can conclude that every clause in $\bar{\phi}$ is true, so $\bar{\phi}$ is an $\neq$-assignment.

   (b) Suppose we have an assignment that satisfies a $3SAT$ formula ϕ. We can construct a $\neq SAT$ assignment ϕ' in the following way. Let every literal of ϕ be unchanged. Then, let every $z_i = \overline{(y_1 \vee y_2)}$ where y_1 and y_2 are the final value of the literal after negation or not. We know that these clauses satisfy an $\neq SAT$ assignment. We also know these clauses are true, because if both y_1 and y_2 are false, then z_i will be true. Also, let v be false all throughout the formula. If y_3 is false then either y_1 or y_2 must be true, so z_i must be false. Therefore, $\bar{z_i}$ must be true. This second clause

also satisfies an $\neq SAT$ assignment because even if both $\bar{z}_i$ and y_3 are true, v will always be false. All in all, neither clause will have all true literals and will have an $\neq SAT$ assignment, so such construction exists.