1. Given a decision procedure for subset sum, we can create a procedure to identify an actual solution to an instance of subset sum as follows.
   - Maintain two variables N and L representing the list L of integers from which sums can be formed and the target sum N and another variable S to hold the solution.
   - As long as N is not 0:
     - set R equal to some element of L and delete R from L.
     - Apply the decision procedure to N and the reduced list L.
     - if the reduced list no longer contains a sublist that sums to N,
       * add R to S
       * set N = N - R
   - S contains a list of numbers that add up to the initial value of N.

2.

   (a) The top four states shown in the diagram in Figure 1 non-deterministically select a sublist of the numbers in the original list. State C decides whether to use a number or not. State K (for keep) leaves the digits of a number the machine wants to add to the sublist alone. State Z (for zero), crosses out all the digits a number that the machine decides should be left out of the solution set.

   The algorithm used to verify that the numbers selected by the initial steps of the Turing machines execution sum up to the correct value depends on the fact that the sum of a set of even numbers must be even.

   Suppose the input to the Turing machine specifies a target sum of N and that the list of numbers chosen by the first, non-deterministic phase of the machines execution is L. The validation process works by repeatedly scanning L looking for numbers in the list that are odd. It makes each odd value found even by subtracting one from its value. To ensure that the sum of the resulting values will still equal N, each time it subtracts a 1 from a member of L, it also subtracts 1 from N. When all of the elements of L are even, the validator checks that N is also even. It then divides N and all of the elements of L by 2 (by deleting their last digits which must be zeros).

   To do this, it starts scanning the members of L from the right to the left. If it finds a member of L that is even (ends in a 0), it divides it by 2 by deleting its last digit. If it finds a member of L that is odd, it marks its last digit, moves to the left to find the encoding of N and subtracts one from N, move back to the right to find the marked digit (an @), and deletes it (effectively subtracting one and dividing by 2). Once a pass from right to left is completed in this way, the machine has ensured all elements of L were made even and

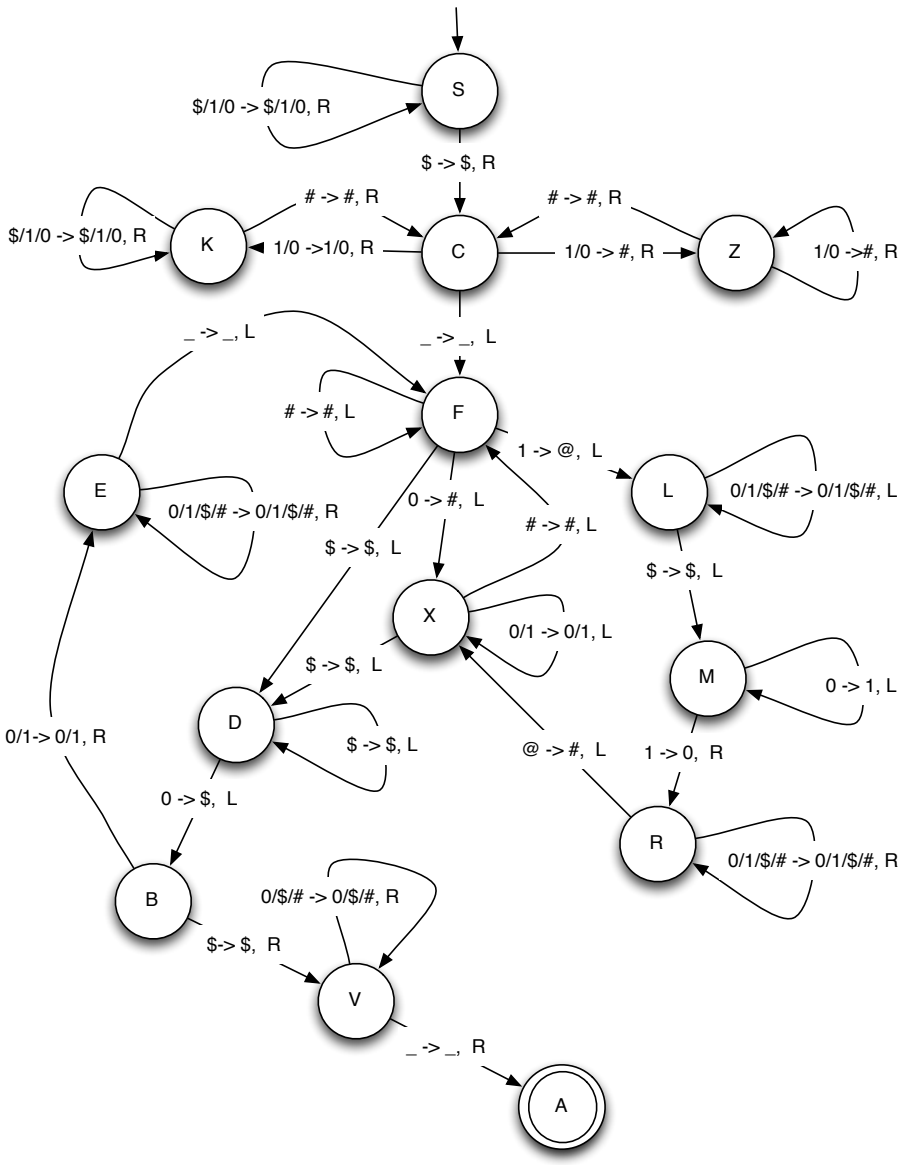

Figure 1: A NTM that decides Subset-sum

were divided by 2. It then verifies that N is even by checking that its last digit is 0 and deletes this digit if it is 0. If N still has more digits to process, it begins the next right to left scan of the members of L. If all the digits of N have been deleted, the machine scans to the right to check that all of the digits of the members of L have also been deleted and accepts if they have.

(b) The obvious bound on the running time of this algorithm is $O(n^3)$. Each time the machine scan through L looking for odd numbers, it gets to delete one digit of N. Since the number of digits in N must be less than the length of the input ($|w|$), there must be $|w|$ or fewer such passes. On each pass, the machine has to travel back and forth from somewhere in the representation of L to the representation of N for each odd number in L. Traveling back and forth can take no more than $2|w|$ steps and there can be no more than $|w|$ odd numbers in L. So, the total execution time must be less than $2|w|^3$.

A tighter bound of $O(n^2)$ can be obtained by looking at the process slightly differently. Each time the loop to subtract one from the target value (involving the states L, M and R) one digit with value 1 is eliminated from the collection of values in the subset the machine has guessed will add up to the target value. There can be at most $O(n)$ such 1 digits in this collection of values. Therefore, the total number of steps consumed in states L, M and R is bounded by $O(n^2)$. The loop that eliminates a digit from the target value and moves back to the right end to eliminate more digits from the source value also takes $O(n)$ steps and executes at most $O(n)$ times for a total of $O(n^2)$ steps.

(c) The fact that the machine in Fig. 1 run in $O(n^2)$ non-deterministic time satisfies the requirement that for any language to be NP-complete it must first be an element of NP. To demonstrate that a language is an element of NP we must show that there is a non-deterministic Turing machine that decides the language. A machine using the "obvious" algorithm would fail to do this. Such an algorithm would make a pass back and forth on the tape once each time it subtracted 1 from N. If N is represented by a binary string of length l its value can start as large as 2^l. That is, the number of passes required would be exponential in the length of N which is only limited by the length of the input w. Therefore, the obvious algorithm takes exponential rather than polynomial time.

3.

(a) Given an $\neq$-assignment for the variables of a 3-cnf formula ϕ, we know that each clause in ϕ contains at least one literals whose value is false under the assignment. If we negate the assignment, each

literal that had been false will become true. Therefore, each clause will contain a true literal and ϕ will still be satisfied. Similarly, since we know that each clause contained a true literal which will become a false literal if the assignment is negated, we can conclude that the negation will still be an $\neq$-assignment.

(b) Suppose we are given a 3-cnf formula ϕ and we produce a new formula ϕ' by replacing each clause $(y_1 \vee y_2 \vee y_3)$ with the two clauses

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

where z_i is only used in these two new clauses and b appears in all of the new clauses we introduce.

Suppose that we had some assignment that satisfied ϕ. It is then possible to extend this to an $\neq$-assignment for ϕ'. In the extended assignment, the values for all of the original variables in ϕ will be unchanged, b will be assigned false, and z_i will be assigned so that $z_i = \overline{(y_1 \vee y_2)}$. First, we can see that this is a satisfying assignment. The first clause of each pair we created must evaluate to true because if neither y_1 nor y_2 is true, z_i will be true. The second clause of each pair must be true because if y_3 is false, $y_1 \vee y_2$ must be true or the original assignment would not satisfy ϕ. In this case z_i will be false and $\overline{z_i}$ will be true. At the same time, neither class will have three true literals. b will always be false and z_i will be false unless both y_1 and y_2 are false.

On the other hand, suppose we have an $\neq$-assignment for ϕ'. This assignment will be a satisfying assignment for ϕ. Because both any $\neq$-assignment and its negation are $\neq$-assignments, we know that if there is an $\neq$-assignment for ϕ' then there is one in which b is false. As a result, for each pair of clauses introduced by the construction, either y_3 or z_i must be true. If y_3 is true, then the corresponding clause in ϕ will be satisfied. If y_3 is false, then $\overline{z_i}$ must be true so z_i must be false and either y_1 or y_2 must be true which would satisfy the original clause.

(c) The language of 3-cnf formulas for which there is a $\neq$-assignment is in NP since given an assignment, it is easy to verify that every clause has both true and false literals. This together with the reduction from 3-SAT to $\neq$-assignment implies that $\neq$-assignment is NP-complete.