Computability

1. Answer. Look for the end of the string (i.e., blank). Move back one. If there isn't a one there, reject. If there is a one there, accept.

State	Input		
	0	1	blank
Q0	(q0, 0, R)	(q0, 1,R)	(q1, b, L)
Q1	(q_r, 0, R)	(q_a, 1, R)	N/A

2.

A) We covered both these cases in lecture. L_a is not recognizable. How do you tell if it is looping or just taking a long time to process? Perhaps the machine will halt, but not for a long time. The complement is the halting problem, and is recognizable. If the machine loops on the input, that is fine because of the definition of recognizable.

B) This language is recognizable. All that needs to be done is to simulate the machine and check that it does not move past the 10^{th} input square.
The complement of this (M does move past the 10^{th} input square on input 10010) is also recognizable. The machine can be simulated just as before. If the machine moves past the 10^{th} input square with input 10010, then it accepts.

C) Both of these languages are recognizable. Since we are given the encoding of the machine, we can check that there is a path to reach every state. Thus L_c is recognizable. The complement is also recognizable. By checking the encoding, we can see if there is a state that can't be reached.

3.

A) The language I_{TM} is recognizable. Here is a recognizer for the language:

Feed input in a dovetailed manner into simulations of M_1 and M_2. If both machines accept the same input, we are done and the language has been accepted. Otherwise, the recognizer will keep running, feeding in longer and longer inputs into the two machines. This could loop forever if the languages of the two machines do not overlap, but that is acceptable according to the definition of the recognizability.

B) The complement of I_{TM} is such that the languages of the Turing machines do not overlap. In other words, they can never accept the same input string. This is unrecognizable. To accept this language, all possible input strings would have to be simulated on the two machines, which would never return. This does not meet the criteria of recognizability.

4. There would no longer be a one-to-one correspondence with the natural numbers if the string length is unbounded. In this case with an alphabet of length 2, we have the formulation as below:

Range of indices	Set
2^0	Empty set
2^1 to 2^2 -1	0,1
2^2 to 2^3-1	00,01,10,11
…	…
2^∞ to 2^∞-1	

At the "end" if this table, we have two indices that are the same number, infinity. Thus, there is not a one to one correspondence with the natural numbers.

Complexity

1. There are two ways to show that this problem is in NP. One is to show that a given solution can be checked whether or not it is correct in polynomial time. This is easy to show in this case. Given a solution (i.e., a Hamiltonian cycle), the costs can be found and summed in polynomial time. This sum is then compared to the maximum cost k.

   The other option is to use the formal definition of NP. Can a nondeterministic TM solve the TSP in polynomial time? In this case, yes. There will be a separate branch of computation for each possible path through the graph. All we need is for one of these paths to accept in polynomial time. This again amounts to summing all the costs for the edges travelled and comparing them to the maximum cost k. This can be done in polynomial time.

2. To reduce 3-CNF to 4-CNF, we need to show that every member within 3-CNF can map to some member within 4-CNF and that every member outside 3-CNF maps to a member outside of 4-CNF.

   I will show that every member within 3-CNF maps to a member within 4-CNF. This can be done by merely repeating the last literal in each clause so that each clause has 4-literals. For example,

   $$(a \vee \bar{b} \vee c) \wedge (b \vee c \vee \bar{d})$$

   becomes

   $$(a \vee \bar{b} \vee c \vee c) \wedge (b \vee c \vee \bar{d} \vee \bar{d})$$

   These two statements are logically equivalent since the literals are disjuncted, but now the second is in the form of 4-CNF. This reduction takes place in polynomial time. All that is required is to iterate over the clauses and append the last literal.

3. So we have a fully connected graph. This problem is similar to one for homework 4. We have an "oracle" that we can use to break the problem into smaller and smaller pieces to build a solution up.
   A) We know that a Hamiltonian cycle must exist since this is a fully connected graph. The question is what is the minimum cost. First, we must have an estimate for k. Find the greatest cost of travelling along an edge and multiply by the number length of the Hamiltonian cycle (number of vertices -1). Next, find the lower bound by finding the least cost for travelling between two edges and multiply by the length of the Hamiltonian cycle. Both of these operations will take polynomial time (breadth first search).

      Now that we have upper and lower bounds, we can use the oracle A and change the value of k, keeping c the same. I don't know what the algorithm is called, but I know that the minimum value of k can be found in polynomial time by splitting the difference in every iteration. For example, we have upper bound 100, lower bound 50. We start with a k value of 75. If the value is too high (A(c,k) returns true), our next guess for k will be 50 + ((75-

50)/2) = 63. If it is too low, the next guess for k would be 75 + 25/2 = 87. (I am rounding.) This continues until we settle on a value for k.

In total, the running time is the sum of two polynomials, which is still polynomial.

B) I will describe in words, then if I have time, try to write some pseudo code. I will assume that the minimum cost is known using the algorithm from part A. I am going to call the function below FindHamPath(matrix c, integer k).

- Run A(c,k). If it returns true, continue. Otherwise return null
- For each vertex v in the graph
    - For each edge E incident on vertex v
        - Run A on a matrix that does not include v and with the cost of travelling along edge E (A(c-v, k-cost(E)))
        - If A returns true, call FindHamPath(c-v, k-cost(E)) and add this to the path. Otherwise move to the next edge

Pseudo code here

I know there needs to be a base (stopping) case since this is recursive. That will be when the number of vertices is 2 (the number of edges is one).

This would still run in polynomial time. We would have at most n*(n-1)+n-1 iterations, where n is the number of vertices.

4. I am not really sure what help the hint offers. What I need to do is show that every instance of a subset sum problem maps to an instance of the partition problem. This relationship can be many to one. It seems to me that every instance of the subset sum problem inherently has a partition problem built in. Take a collection of subset sum problems that have the same input set but different k values. These will all map to the same partition problem because the desired sum is determined by the set itself not by another input parameter. This also works for different sets. Any subset sum problem by definition also is a partition problem. This reduction will take place in polynomial time because it is simply a matter of changing the definition. Nothing really needs to be computed.

   Here is me trying to make sense of the hint. Say I have a set A={1,2,3,4} and k = 7. So for the partition problem, we have T = 10 and T/2 = 5. If I were to add two very large numbers to the set A, say 1 million, I would then have T = 2,000,010 and T/2 = 1,000,005.

   At first I thought we would try to make k and T/2 equal, but I don't see why that would be necessary by the properties of reduction