Exercises:

3.1

Since turing machines only map from non-negative to non-negative numbers, if any process in nature is found to map between different set of values, then that process cannot be run on a turing machine

3.2

Take the input of the Turing machine a1, …, ak, and then give the turing machine the value p1^a1 * p2^a2 * … * pk^ak, with p1, p2, …, pk being the first k prime numbers, and thus all turing machines with unique inputs will be given unique value identifiers, since all numbers only have one kind of prime factorization.

3.5

Define H2(M) = {0 if machine doesn't halt with blank input, 1 if machine does halt with blank input

We have the following algorithm:

```
Turing(M)
Y = h2(M)
If y == 0
        Halt
else
        Loop forever
End if
```

Here, if M is blank, then h2(M) = 1 only if y = h2(M) = 0, thus we form a contradiction, meaning this machine cannot read input M, which is blank, meaning there is no algorithm to solve for H2(M) for this particular machine.

3.6

Define Hp(x) = {0 if probability machine x halts in input x < ½, 1 if probability > ½}

We have the following algorithm:

```
Turing(x)
Y = hp(x)
Y2 = flip an unbiased coin
If y == 1 and y2 = heads
        Halt
Else
        Loop forever
End if
```

Here, assume hp(x) = 1 and thus corresponding probability p > ½. The probability program halts is p * ½ = at most ½(since p is at most 1). Thus, this contradicts our original

statement that hp(x) = 1, and thus there is no algorithm to correctly determine hp(x) for this machine

3.7

Yes. Before, the problem was no algorithm existed to compute HALT(x) for all x, but now that the blackbox exists that algorithm also exists, and since Turing machines can compute all algorithms, these Turing machines can compute this algorithm for all machines.

3.8

And - Input bit [1, x0, x1]. Apply NAND to x0 and x1, then a nand on [1, NAND(x0, x1)]

NOT - NAND input bit x0 and ancilla bit 1

XOR - input [1, x0, x1, 1, 1] NAND[1, NAND[NAND[x0, x1], NAND[NAND[1, x0] NAND[1, x1]]].

3.9

f(n) is O(g(n)) -> f(n) <= cg(n)
g(n) is OMEGA(f(n)) -> cf(n) <= g(n)
If g(n) is OMEGA(f(n)) then cf(n) <= g(n), f(n) <= g(n) / c
And thus f(n) <= cg(n) and f(n) is O(g(n))

Thus g(n) is THETA(f(n)) and f(n) is THETA(g(n))

3.10

g(n) is O(n^l) -> g(n) <= cn^l
g(n) = An^k + Bn^k-1 + … + d <= cn^l
THus, because An^k + Bn^k-1 + … + d <= n^k+1 <= n^k+2 … and we design c such that An^k + Bn^k-1 + … + d <= cn^k, thus g(n) is O(n^l)

3.11

Log n <= cn^k, n<=c10^n^k, so we design c^k such that n <= c*10^n^k for all k>0

3.12

From 3.10, we proved logn >= k for sufficiently large n, thus g(n) = n^k is in O(n^logn). However, log n is not <= k for large n, so g(n) = n^k is never in O(n^k)

3.13

c^n is OMEGA(n^logn) -> check graphically for sufficiently large n
Since c^n is OMEGA(n^logn) n^logn can’t be in OMEGA(c^n)

3.14

e(n) is O(f(n)) -> e(n) <= cf(n)
g(n) is O(h(n)) -> g(n) <= c2*h(n)
e(n)*g(n) <= c*c2*f(n)*h(n) = c3 * f(n) * h(n)

3.15

AFter 1 swapping, there is only 2^1 ordering such that tha swapping puts the whole thing in order. AFter 2 swapping, there are 2^2 orderings that were two swapping away from being sorted. After k swappings, it follows that 2^k initial orderings are now sorted.

2^nlogn = 2^log n^n = n^n >= n!

THus, the lower bound on sorting is nlogn. You can see that asymptotic notations has the important effect of allowing us to find how efficient a particular algorithm can be on a process, but it doesn't necessarily tell us HOW to make such an algorithm, but it gives us an idea of how efficient the most efficient algorithm can be.

3.17

If this problem is in P, then a Turing machine exists for identifying if a value is a factor of a number m and less than L, and thus setting L = m we have the factoring algorithm that can thus be done efficiently. If this problem isn't in P, the factoring obviously can't be done efficiently since if for any L it is inefficient than L=m is definitely inefficient.

3.18

P is a subset of CoNP so if CoNP does not equal NP, there are some problems in P and CoNP, and the rest of the problems in P and NP, and thus there are some problems in P that are in CoNP but not NP and thus P does not equal NP

3.19

Start at the first vertex, try to get to the 2nd. At most n vertices exist to visit, so its O(n)(algorithm would make sure not to visit a vertice more than once).

Then use Reachability algorithm to form the following O(n^2) algorithm:

```
For i through all vertices
    For j through all vertices
        Test Reachability(vertex(i), vertex(j)
    End j
End i
```

3.20

Euler's theorem is based on the following idea: If you visit a node, you can go on a new edge you haven't gone before, because the vertex has an even amount of incident edges so if a node is visited and then left, 2 edges have been traversed and used. Thus if you go through all nodes you will always have a new edge to move through for every node until there are no more edges left to visit, at which point you are back at the original node and have completed the cycle.

The procedure then would be to start at a node, go to all other nodes until you have reached back to the original node and there are no new edges to visit.

3.21

Since L1 -> L2 exists, there exists function R(x) that gives a string in language L2 iff x is in L1. Since L2 -> L3 exists, there exists a function R2(x2) that gives a string in L3 iff x2 is in L2. Thus, with R(x) + R2(R(x)) (polynomial time overhead), we reduce L1 to L3.

3.22

Since all other problems can be reduced to L, and L can be reduced to L', all other problems can be reduced to L'.

3.25

With lm^p(n) different states, all problems in PSPACE can be solved by going through all possible states in lm^p(n) time, or exponential time. Thus PSPACE is a subset of EXP

3.26

If you can solve a problem in L, then you can solve the problem going through around c*log(n) spaces, thus you can solve it in time clog(n), which is polynomial. Thus L is a subset of P.

3.27

In this algorithm, at worst the min vector span is made up of all the alpha's and none of the Beta's, in which case at worst this algorithm calculates the a vector span that is 2 * the space of the min vector span.