**Homework #9 (Section 7.1 – 7.3) Solutions**

1) Let Prime = { x | x is represented in unary }. Prove that Prime $\in$ P.

**Solution**
Use the usual trial division method:

```
int isPrime(int n) {

    int i;
    for (i=2; i<n-1; i++)
        if (n%i == 0)
            return 0;
    return 1;
}
```

The number of times the loop runs is n-2, where n is the number being checked for primality. Since the input is in unary, the length of this input is ALSO n. Thus, the loop runs O(n) times, where n represents the input size. For each loop iteration, we do an integer division operation. When numbers are encoded in unary, this can easily run in O(n) time, where n is the input size, so long as we have an extra work tape. (We can simply cancel i numbers at a time and see if we have nothing left over.) Thus, this runs in $O(n^2)$ time at the very worst, **when the input is in unary.**

2) Consider the problem in #1, where the input x, is represented in binary. Now, consider the standard trial division algorithm to check for primality, where we try to divide x by each integer in between 2 and x – 1, inclusive. Why would this algorithm NOT run in polynomial time in the size of the input?

**Solution**
Doing the same analysis as above, the only difference now is that if we are checking n for primality, but n is represented in binary, the **size of n** is only $\log_2 n$. Let $x = \log_2 n$. Thus, x represents the **input size.** Now, the for loop shown in algorithm runs roughly n times, which is approximately $\mathbf{2}^x$ times. Since x is the input size, this algorithm now takes $\mathbf{x^2 2^x}$**,** assuming that division takes O(x) time. (This is a reasonable assumption since the standard grade school algorithm for division takes O(x) time for each iteration and runs at most x iterations, where x is the size of the number being divided into.) This is NOT polynomial time in size of the input.

3) Let CONNECTED = {G | G is a connected undirected graph}. Prove that CONNECTED $\in$ P.

**Solution**
Pick a node in the graph arbitrarily. Run DFS on this node, marking each visited node. Check if all nodes in the graph are visited after this DFS. A DFS runs in O(V + E) time, where V represents the number of vertices in the graph and E represents the number of edges in the graph. The input size must be at least this big. Thus, this algorithm runs in **linear** time of its input size, which easily shows the problem to belong to P.

4) Let MODEXP = { <a,b,c,p> | a, b, c, and p are binary integers such that $a^b \equiv c \pmod{p}$ }. Prove that MODEXP $\in$ P.

**Solution**

Use fast modular exponentiation:

```
// Fast Modular Exponentiation
int fastmodexp(int base, int exp, int mod) {

    if (exp == 0)
        return 1;

    // The key time savings is here! We can reuse the value
    // mysqrt without redoing the work to recalculate it.
    else if (exp%2 == 0) {
        int mysqrt = fastmodexp(base, exp/2, mod);
        return (mysqrt*mysqrt)%mod;
    }

    // This case runs at most every other time.
    else
        return (base*fastmodexp(base, exp-1, mod))%mod;
}
```

When we analyze the run-time, we see that we're dividing the exponent by 2 every other recursive call, in the worst case. (If the exponent is odd in one case, in the following recursive call it's even, and will get divided by 2.) Thus, the number of recursive calls made by this algorithm is simply no more than $2\log_2 exp$, where exp is the value of the exponent. This is linear in the **size of the input value** exp. Since the basic multiplications and mod required here are also polynomial in the input size, the overall run time is also a polynomial.

5) Let LCS = { <a, b, c> | a, b are strings and c (represented in binary) is a non-negative integer such that the longest common subsequence between a and b is of length c. } Prove that LCS $\in$ P.

**Solution**

Dynamic programming is necessary to solve this problem in polynomial time. Here is some Java code that solves the problem in its maximization form:

```
// Precondition: Both x and y are non-empty strings.
  public static int lcsdyn(String x, String y) {

    int[][] table = new int[x.length()+1][y.length()+1];

    for (int i = 1; i<= x.length(); i++) {
      for (int j = 1; j<= y.length(); j++) {

        // If last characters of prefixes match, add 1.
        if (x.charAt(i-1) == y.charAt(j-1))
          table[i][j] = 1+table[i-1][j-1];

        // Otherwise, take the maximum of 2 cases.
        else
          table[i][j] = max(table[i][j-1], table[i-1][j]);
      }
    }
    return table[lenx][leny];
  }
```

It is not difficult to turn the code above into a boolean function. (Pass in an integer as a third parameter, say value, and return value == table[lenx][leny].)

The run time here is simply |a| * |b|, which is polynomial in the input size because the actual sizes of the strings a and b are |a| and |b|.