December 9

1. Show that if an algorithm makes at most a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then it runs in polynomial time. Also show that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

   Solution:

   We show the first half of this exercise by induction on the number of times that we call the polynomial time subroutine. If we only call it zero times, all we are doing is the polynomial amount of extra work, and therefore we have that the whole procedure only takes polynomial time.

   Now, suppose we want to show that if we only make $n+1$ calls to the polynomial time subroutine. Consider the execution of the program up until just before the last call. At this point, by the inductive hypothesis, we have only taken a polynomial amount of time. This means that all of the data that we have constructed so far fits in a polynomial amount of space. This means that whatever argument we pass into the last polynomial time subroutine will have a size bounded by some polynomial. The time that the last call takes is then the composition of two polynomials and is therefore a polynomial itself. So, since the time before the last call was polynomial and the time of the last call was polynomial, the total time taken is polynomial in the input. This proves the claim of the first half of the input.

   To see that it could take exponential time if we were to allow polynomially many calls to the subroutine, it suffices to provide a single example. In particular, let our polynomial time subroutine be the function that squares its input. Then our algorithm will take an integer x as input and then square it $\log x$ many times. Since the size of the input is $\log x$, this is only linearly many calls to the subroutine. However, the value of the end result will be $x^{2^{\log x}} = x^x = 2^{x \log x} = 2^{\log x \cdot 2^{\log x}} = \omega(2^{2^{\log x}})$. So, the output of the function will require exponentially many bits to represent, and so the whole program takes exponential time.

2. Define the optimization problem LONGEST-PATH-LENGTH as the relation that associates each instance of an undirected graph and two vertices with the number of edges in a longest simple path between the two vertices.

   Define the decision problem LONGEST-PATH $= \{(G,u,v,k) : G = (V,E)$ is an undirected graph, $u,v \in V, k \geq 0$ is an integer, and there exists a simple path from u to v in G consisting of at least k edges$\}$.

   Show that the optimization problem LONGEST-PATH-LENGTH can be solved in polynomial time if and only if LONGEST-PATH $\in P$. Here P denotes the class of problems having a polynomial time algorithm.

   Solution:

   Showing that LONGEST-PATH-LENGTH is polynomial implies that LONGEST-PATH is polynomial is trivial, because we can just compute the length of the longest path and reject the instance of LONGEST-PATH if and only if k is larger than the number we computed as the length of the longest path.

Since we know that the number of edges in the longest path length is between 0 and $|E|$, we can perform a binary search for its length. That is, we construct an instance of LONGEST-PATH with the given parameters along with $k = |E|/2$. If we hear yes, we know that the length of the longest path is somewhere above the halfway point. If we hear no, we know it is somewhere below. Since each time we are halving the possible range, we have that the procedure can require $O(\log |E|)$ many steps. However, running a polynomial time subroutine $\log n$ many times still gets us a polynomial time procedure, since we know that with this procedure we will never be feeding the output of one call of LONGEST-PATH into the next.