Problem Set 3

All parts are due Thursday, October 16 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Name: Rishad Rahman

Collaborators: None

Part A

Problem 3-1.

Solution: Suppose $n = p_1p_2$, then $\phi(n) = p_1p_2 - p_1 - p_2 + 1 = n - p_1 - p_2 + 1$ (via PIE or Totient Formula). So $p_1 + p_2 = n - \phi(n) + 1$. This determines p_1, p_2 so we can just iterate through the integers values of p_1 and calculate $p_2 = n - \phi(n) + 1 - p_1$ then p_1p_2 to check if it is equal to n. This is efficient since the $O(n)$ subtractions and multiplications are polynomial in the number of bits of n compared to the exponential time of finding the prime factorization.

Problem 3-2.

Solution: $g(m, n)$ computes the gcd of m and n. This is easy to see since the algorithm breaks the problem into cases involving the parity of m and n. When m and n are both even, a factor of 2 is thrown in while dividing each by 2. When one is even, and not the other, then we divide the even number by 2 without throwing any factors until we get two odd numbers since the two numbers don't share an even factor. Now suppose we have two odd numbers, we already extracted the highest common power of 2 from each number, so all that is left is to extract the highest odd factor. Well notice that $g(\min(n, m), \frac{|n-m|}{2})$ is actually the Eucliean Algorithm except with the added information that $|n - m|$ is even but we already extracted all the powers of 2 so we can divide it by 2.

Runtime: The algorithm can take as little as constant time i.e. when $m = n$ and they are odd. Otherwise we are halving both or one of the numbers, effectively at least one of the next inputs has 1 fewer bit, until we reach two odd numbers. At that point we are

decreasing the bit of one of the inputs into g in our recursion by at least 1 because of $\frac{|n-m|}{2}$. Therefore at each step of the recursion on g, we are decreasing the bit length of one of the inputs by at least 1. So the runtime is $\boxed{O(n_b + m_b) = O(\log n + \log m)}$ where n_b and m_b are the respective bit lengths of n and m. Note we can't derive a better bound since $g(2^a, 2^a p)$, p is a large prime, takes $a + \log p$ time since $g(1, p)$ will not terminate until p turns into a 1 as well which we do by constantly truncating its terminating bit.

Problem 3-3.

Solution: Note $10^n - 1$ has n digits with all 9's while $10^m - 1$ has m digits with all 9's. Let $k = \lfloor \frac{n}{m} \rfloor$. Note $10^n - 1 - (10^m - 1)\sum_{i=1}^{k} 10^{n-mi} = 9\left[\sum_{i=0}^{n-1} 10^i - \sum_{j=0}^{m-1} 10^j \sum_{l=1}^{k} 10^{n-ml}\right] = 9\left[\sum_{i=0}^{n-1} 10^i - \sum_{j=0}^{m-1}\sum_{l=1}^{k} 10^{j+n-ml}\right] = 9\left[\sum_{i=0}^{n-1} 10^i - 10^{n-mk}\sum_{j=0}^{m-1}\sum_{l=0}^{k-1} 10^{j+ml}\right] = 9\left[\sum_{i=0}^{n-1} 10^i - 10^{n-mk}\sum_{j=0}^{mk-1} 10^j\right] = 9\sum_{i=0}^{n-mk-1} 10^i = 10^{n-mk}$. Therefore $\gcd(10^n, 10^m) = \gcd(10^m, 10^{n-mk})$ but this is precisely the Euclidean Algorithm except on the exponents. Therefore we can just compute $\gcd(n, m)$ and our answer will be $10^{\gcd(n,m)}$.

Runtime: Since $\gcd$ takes time polynomial in the number of bits i.e. $\log_2 n$, or we could use the algorithm from Problem 3-2, we have that that the runtime is $\boxed{O(\log_2 n)}$.

Problem 3-4.

Solution: We pick a random $x \in [1, n]$ and compute $y = BB(x^2, n)$. Since $(y+x)(y-x) = 0 \mod n$ and with probability greater than half we will not have $y = \pm x \mod n$ because of BB so we can compute $\gcd(y + x, y - x)$ to find a factor of n. We can recurse this process on that factor and the other factor until we reduce to the prime factorization. If $y+x$ or $y-x$ are 0 then we check if the current number we are trying to prime factorize is prime. If not, the algorithm fails. If yes then we have found a prime factor. Doing this for all parts of the original factorization will give us all the prime factors, with degeneracy, so we have the prime factorization.

Part B

Submit your implemented python script on alg.csail.mit.edu.