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.

In this problem set, it suffices to solve three out of the four problems in Part A as well as the programming part B to get a full score of 100. However, you may choose to attempt all four problems in Part A for bonus points.

Part A

Problem 3-1. [20 points] **Factorization using Euler's Totient**

Recall that Euler's totient function $\phi(n)$ is defined as the number of positive integers less than or equal to n that are relatively prime to n.

Design an efficient algorithm that given

1) An integer n which is promised to be the product of two distinct odd primes, and

2) The value of $\phi(n)$

computes the integer factorization of n.

Problem 3-2. [20 points] **Mystery Function**

Consider the following algorithm for computing a function $g(n, m)$ of non-negative integers n and m.

```
g(n, m)
1  if n = 0 or m = 0
2     then return n + m
3  if n is even and m is even
4     then return 2 · g(n/2, m/2)
5  if n is even
6     then return g(n/2, m)
7  if m is even
8     then return g(n, m/2)
9  return g(min(n, m), |n − m|/2)
```

where the notation $|a|$ denotes the absolute value of the number a.

What does the function $g(n, m)$ do? Derive the asymptotic running time of the algorithm in terms of the bit-lengths of n and m.

Problem 3-3. [20 points] **Greatest common divisor**

Describe an algorithm that, given positive integers n and m where $n \geq m$, computes the GCD of $10^n - 1$ and $10^m - 1$ in time polynomial in $\log_2 n$.

Problem 3-4. [20 points] **Reducing factorization to square roots**

Let $BB(x, n)$ be a black box subroutine that, given positive integers x and n, computes a square root of x modulo n in a single computational step (or reports None if no square root exists).

Recall that in the lectures we saw how to use BB as a subroutine to factor integers that are products of two distinct odd primes. Generalize this idea to construct an algorithm that uses BB as a subroutine to factor any arbitrary positive integer n into its prime factors. Your procedure can be probabilistic, and should return the correct answer with probability $1/2$ or more.

Note that if x has more than one square root modulo n, the black box BB returns an arbitrary square root. In other words, you are not allowed to assume anything about which square root BB returns.

(*Hint:* As a first step, it may be useful to consider the case where n is a product of three distinct odd primes.)

Part B

Problem 3-5. [40 points] **Primality testing**

With your help in Problem Set Two, James Cameron was able to design a new world for his next movie, *Avatar 2*. Cameron needs to share the file containing his new world with his designers and animators, but he fears that reporters might be watching the network for information to leak. Cameron looks through the MIT 6.042 textbook for help and comes across RSA encryption. Not knowing that there are software libraries for this, he decides to do the encryption by hand. Recall from 6.042 that RSA encryption works by multiplying two very large prime integers and relies on the fact that integer factorization is a hard problem. Cameron knows that he needs very large prime numbers but he cant find a way to test whether or not a large number is prime so he comes back to MIT for help.

You are quick to think that you can simply try to divide the candidate prime, n, by all numbers from 1 to n. This works in a few small test cases but runs too long on Camerons very large prime numbers. Prove to yourself that this naive algorithm has an exponential runtime with respect to the length of the input. You decide to consult Professor Micali who helps you out by giving you an efficient primality testing algorithm, but insists that you write yourself the following functions: `perfect_power` and `sqrt` which are described below. To make things a bit easier, you only need to handle the special case where $n \equiv 3 \pmod 4$.

`silvio`(n)

if $n = 2$
  then return *true*
if n is even
  then return *false*
if `perfect_power`(n)
  then return *false*
Pick a random $x \in \{1, ..., n-1\}$
if `gcd`$(x, n) \neq 1$
  then return *false*
$y =$ `sqrt`$(x^2 \bmod n, n)$
if $y \neq x \bmod n$ and $y \neq -x \bmod n$
  then return *false*
return *true*

Note: This function will return *true* with probability $\leq \frac{1}{2}$ if n is composite. We leave it as an optional (but encouraged) exercise to prove why.

(a) Implement the function `perfect_power(`n`)`. A perfect power is defined as an integer that can be expressed as an integer power of another integer, i.e. $n = a^b$ for some $a, b > 1$. This function will take in a positive integer n and return the values that a and b if they exist.

(b) Implement the function `sqrt(`n, p`)`. This function should be a polynomial-time algorithm that, if p is a prime such that $p \equiv 3 \mod 4$, and n is a quadratic residue $\mod p$, outputs a square root of $n \bmod p$. Square roots modulo primes can in general be computed with the Tonelli–Shanks algorithm, which in this special case just reduces to the formula

$$\sqrt{n} = \pm n^{\frac{p+1}{4}} \bmod p.$$

(c) Implement the function `silvio(`n`)`. This function will take in a positive integer (which is congruent to 3 modulo 4) and return whether or not it is prime.

Your solution should be efficient and run in polynomial time with respect to the length of the input, that is, the number of bits in n. You are free to use any python standard libraries. You might find `fractions.gcd` and `pow` useful. We have included lines $1 - 4$ in `silvio` for completeness; however, your function only needs to support $n \equiv 3 \pmod 4$.