Algebraic Cryptography
Homework 3

Due Monday, 25 September 2017

Note: this is graduate school, so due dates for homework are somewhat flexible (within reason). I trust you to make smart decisions regarding your understanding of the material. I encourage you to do these problems before the exam, but if you are having trouble completing the write-up while studying, that is okay.

Problem 1. Exercises 1,3,5 §3 of Chapter 2 (with proofs / arguments)

Solution to #1. (a) Computing 3^n is $O(n^2)$. (See the solution to part (c)).

(b) Computing n^n is $O(n^2 \ln^2 n)$(See the solution to part (c)).

(c) We will show that computing k^n is $O(n^2 \ln^2 k)$. Let us consider our algorithm:

- Compute n in binary ($O(\ln n)$ bit ops).
- Set $l_0 := k$, $res_0 := 1$.
- For $0 \le j \le m = \lfloor \log_2 n \rfloor$
  - Set $l_{j+1} := l_j^2$ ($O(\ln^2 l_j)$ bit ops).
  - If the j-th bit of n in binary is 1, set $res_{j+1} := res_j \cdot l_j$, otherwise $res_{j+1} := res_j$ ($O(\ln res_j \ln l_j)$ bit ops).
- Result: res_{m+1}.

Okay, so, this entire algorithm

$$O\left(\ln n + \sum_{j=0}^{m} \ln^2 l_j + \ln res_j \ln l_j\right)$$

bit operations. But notice that $l_j = k^{2^j}$ and

$$res_j \le \prod_{i=0}^{j-1} l_i = \prod_{i=0}^{j-1} k^{2^i} = k^{\sum_{i=0}^{j-1} 2^i} \le k^{2^j}.$$

Therefore, $\ln^2 l_j = (2^j)^2 \ln^2 k$ and $\ln res_j \ln l_j \leq (2^j)^2 \ln^2 k$ Hence the number of bit operations is

$$O\left(\ln n + 2\ln^2 k \sum_{j=0}^{m} (2^j)^2\right)$$

But the sum of those squares of powers of 2 is bounded by $(2^{m+1})^2 = \Theta(n^2)$. Thus, the $\ln n$ term is negligible in comparison, and we find that the number of bit operations is $O(n^2 \ln^2 k)$. Note that even in the ideal case when n is a power of 2, this still requires $n^2 \ln^2 k$ bit operations.

■

Solution to #3. (a) Let's consider our algorithm to compute $\sum_{j=1}^{n} j^2$.

- Set $sum_0 = 0$.
- For $1 \leq j \leq n$
  - Compute j^2 ($O(\ln^2 j)$ bit ops)
  - Set $sum_j := sum_{j-1} + j^2$ ($O(\ln sum_{j-1} + \ln j)$ bit ops)
- Result: sum_n.

Note that $sum_{j-1} = \Theta(j^3)$, the number of bit ops in the second step of the loop is $O(\ln j)$. So each step of the loop takes $O(\ln^2 j)$ bit ops. Thus the loop takes $O(\sum_{j=1}^{n} \ln^2 j)$ bit ops, but the sum here is $\Theta(n \ln^2 n)$ (too see the lower bound, just sum the largest half of the terms). So the algorithm is $O(n \ln^2 n)$ bit ops.

(b) The right-hand side is just two multiplications. The first takes $O(\ln n \ln(n+1)) = O(ln^2 n)$ bit ops, and the second takes $O(\ln(n(n+1)) \ln(2n+1)) = O(\ln^2 n)$ bit ops, for a total of $O(\ln^2 n)$ bit ops.

■

Solution to #5. (a) We will use the fact that $F_n \asymp \frac{\varphi^n}{\sqrt{5}}$. The algorithm is relatively simple, just set $sum_0 = 0$ and for $1 \leq j \leq n$, set $sum_j = sum_{j-1} + F_j$ which takes $O(\ln sum_j + \ln F_j)$ bit ops. But $sum_j \asymp \sum_{i=1}^{j} \frac{\varphi^j}{\sqrt{5}} = \Theta(\varphi^{j+1})$, and thus each round takes $O(\ln F_j) = O(j)$ bit ops. Finally, adding these up from $1 \leq j \leq n$ we find it takes $O(n^2)$ bit ops.

(b) This algorithm is the same, with the sum replaced by the product. Each product takes $O(\ln prod_j \ln F_j)$ bit ops, and we know that $prod_j \approx \frac{\varphi^{j(j+1)/2}}{5^{j/2}} \approx \varphi^{j(j+1)/2}$, so $\ln prod_j \approx j(j+1)/2$. Thus $O(\ln prod_j \ln F_j) = O(j^3)$. Summing this over $1 \leq j \leq n$, we find $O(n^4)$ bit ops.

■

Problem 2. Exercises 6,7,9 from §4 of Chapter 2 (with proofs / arguments).

Solution to #6. We will show $\mathcal{P}_2$ reduces to $\mathcal{P}_1$. Suppose that we have two equations $ax + by = 0$ and $cx + dy = 0$ for some integers a, b, c, d. Note that these equations correspond to lines in the plane passing through the origin. They have multiple solutions if and only if the system of equations is linearly dependent, which in this case means the lines are the same. This occurs if and only if the vectors $\langle a, b\rangle, \langle c, d\rangle$ are scalar multiples of each other. Without loss of generality, we may assume $c = ak$ and $d = bk$ for some integer $k \in \mathbb{Q}$. Then $bc = b(ak) = a(bk) = ad$. This tells us what instance of $\mathcal{P}_1$ we should create. Consider the integers bc, ad, are they equal?

Note: if the answer is yes, then there are two possibilities: $bc = ad \neq 0$ in which case we set $k = \frac{c}{a}$ and we have our scalar multiples. If, on the other hand $bc = ad = 0$, then either b or c is zero, and either a or d is zero. By symmetry, we may assume without loss of generality that $b = 0$. Then, if $a = 0$ the first equation is easily a multiple of the second equation (namely, the second equation times zero). If $d = 0$, then both equations are just multiples of the variable x. Thus, if $bc = ad$, then one of the equations is a scalar multiple of the other and so they share multiple solutions. Moreover, our previous paragraph showed the converse. ■

Solution to #7. Consider two pairs v_1, v_2 and w_1, w_2 of non-proportional vectors in $\mathbb{R}^3$. The question is: do they span the same plane? Note that they span the same plane if and only if the orthogonal complement of their spans are equal. This orthogonal complement is 1-dimensional and spanned by any (nonzero) vector orthogonal to both vectors in the pair. Since each of our pairs are non-proportional, their cross-product vectors are nonzero and orthogonal to each element of the pair. Thus, the orthogonal complement of $\text{span}\{v_1, v_2\} = \text{span}\{v_1 \times v_2\}$ and similarly, for w_1, w_2. Thus, $\text{span}\{v_1, v_2\} = \text{span}\{w_1, w_2\}$ if and only if $\text{span}\{v_1 \times v_2\} = \text{span}\{w_1 \times w_2\}$ if and only if $v_1 \times v_2$ is proportional to $w_1 \times w_2$, which is an instance of $\mathcal{P}_1$ as desired. ■

Solution to #9. Consider the Integer Factorization search problem (IFS) and the RSA problem. We wish to show RSA reduces to IFS. So, consider an instance of RSA, namely, integers e, N with $N > 1$ being odd. we want an integer d so that $x \mapsto x^d \pmod{N}$ is the inverse of $x \mapsto x^e \pmod{N}$ for x such that $\gcd(x, N) = 1$, or the statement that no such d exists. So, we are really working in the group $\mathbb{Z}_N^*$ of units of the ring $\mathbb{Z}/N\mathbb{Z}$. Now, the order of $\mathbb{Z}_N^*$ is $\varphi(N)$ where φ is Euler's totient function. Recall the that totient function is multiplicative, so that if m, n are relatively prime, then $\varphi(mn) = \varphi(m)\varphi(n)$. Moreover, if p is prime, then $\varphi(p^k) = p^k - p^{k-1}$.

We will now use IFS to factor N completely. How? Recursively apply IFS to the factors of N until they are all prime. I claim that this requires at most $O(\ln N)$ calls to IFS. We prove by strong induction on N. For the base case, notice that we can factor $N = 2$ in one call to IFS, and $1 \leq 3\ln 2 - 1$. Now suppose $N > 2$ and for all $1 < m < N$ we can factor m completely in $3\ln m - 1$ calls to IFS. Then, apply IFS to N. If N is prime, we are done in one call, and $1 < 3\ln N - 1$. Otherwise $N = nk$ for some $1 < n, k < N$. Now, it takes at most $3\ln n - 1$ and $3\ln k - 1$ calls to factor completely n, k respectively. Thus, it takes $1+3\ln n-1+3\ln k-1 = 3\ln(nk)-1 = 3\ln N-1$ calls to IFS to factor N. By strong induction, we can factor N with $O(\ln N)$ calls to IFS, which is a polynomial in the size of N (i.e., length).

So, we have the prime factorization $N = p_1^{n_1}\cdots p_k^{n_k}$. We can now compute

$$\varphi(N) = \prod_{j=1}^{k} \varphi(p_j^{n_j}) = \prod_{j=1}^{k} (p_j^{n_j} - p_j^{n_j-1}).$$

The number of factors here is at most $\ln N$, and all the partial products are bounded by N, so computing the product takes at most $\ln^3 N$ operations (this is generally a gross overestimate). Finally, compute $g = \gcd(\varphi(N), e)$ which we can do in $\ln\varphi(N)\ln e$ bit operations. By the fundamental theorem of finite abelian groups, we are guaranteed an element of order q in $\mathbb{Z}_n^*$ for any prime q dividing $\varphi(N)$. If $g > 1$, then there is some prime q dividing g, and some element $y \in \mathbb{Z}_n^*$ of order q. Thus $y^e = 1$ since e is a multiple of g and so also of q. Thus the map $x \mapsto x^e \pmod N$ is not invertible. On the other hand, if $g = 1$, then by the Euclidean Algorithm (which takes $O(\ln e \ln\varphi(N))$, we can find d so that $de \equiv 1 \pmod{\varphi(N)}$. By an argument similar to a problem on the first homework assignment, we can conclude that $(x^e)^d \equiv x \pmod N$ using the Chinese Remainder Theorem.

Checking the above argument, we see that we have a polynomial time algorithm with a polynomial number of calls (in fact, $O(\ln N)$) to an IFS oracle. Thus RSA (polytime) reduces to IFS. ■

Problem 3. Using the work done both on previous homeworks, prove that the RSA encryption and decryption protocols have polynomial time algorithms (in terms of the lengths of the keys). Try to provide a bound on the degree of the polynomial. Note: you do *not* need to show at this time that there is a polynomial time algorithm for RSA key generation.

Proof. The RSA encryption and decryption protocols are just modular exponentiation. So, how many bit ops does modular exponentiation take? You should have some algorithm for the following $m^e \pmod n$ from the previous homework.

- First reduce $m \pmod n$ which takes $\ln m \ln n$ bit ops.
- Set $m_0 = m$, $res_0 = 1$
- For $1 \le j \le k = \lfloor \log_2 e \rfloor$,
  - If the j-th digit of e in binary (starting from the right) is 1, set $res_j := res_{j-1} \cdot m_{j-1} \pmod n$, otherwise, set $res_j = res_{j-1} \pmod n$ ($O(\ln res_{j-1} \ln m_{j-1} + (\ln(res_{j-1} m_{j-1}) \ln n))$ bit ops).
  - Set $m_j = m_{j-1}^2 \pmod n$. ($O(\ln^2 m_{j-1} + \ln(m_{j-1}^2)\ln n)$ bit ops)
- Result: res_k

Now, since we reduce mod n each time, all the values m_j, res_j are bounded by n. This means every step in the loop has complexity $O(\ln^2 n)$. There are $O(\ln e)$ steps in the loop, so the total complexity is $O(\ln m \ln n + \ln e \ln^2 n)$, which is obviously a polynomial in the lengths of m, n, e. So the RSA encryption and decryption protocols have polynomial time algorithms. ■