Mid-Term Exam

(*reference answers*)

Prof. Manisha Kulkarni, Prof. G.Srinivasaraghavan

On: March 10, 2022	**Time**: 2 Hrs	**Max Marks**: 60

1 Theory

Q-1: Let p be an odd prime number. Prove using Fermat's Little Theorem that every prime divisor of $2^p - 1$ is greater than p. **Max Marks: 5**

Answer: Let $q \leq p$ be an odd prime (trivially 2 cannot be a factor of $2^p - 1$). By FLT we have $2^{q-1} = 1 \bmod q$. Note that 2 is co-prime to q. Therefore the multiplicative order k of 2 in Z_q is such that $1 < k \leq (q-1)$. Moreover we know that when $a^i = 1 \bmod n$ for some a with multiplicative order l and co-prime to n then $l|i$. Now if $q|(2^p - 1)$ then $2^p \equiv 1 \bmod q$. This implies that $k|p$ which contradicts the fact that p is a prime. What happens to this argument when $q > p$? ■

Q-2: If $a^2 + b^2 = c^2$ where a, b, c are positive integers then show that 3 divides ab. **Max Marks: 5**

Answer: Observe that for any integer a, $a^2 \bmod 3$ must be 0 or 1 (if $a = 3k+1$, $a^2 = 3l + 1 = 1$ mod 3; if $a = 3s + 2$, $a^2 = 3t + 4 = 1 \bmod 3$). The following table shows the valid values for a, b, $(a+b)^2$, a^2, b^2, $c^2 = (a^2 + b^2)$ all mod 3.

a	b	$(a+b)^2$	a^2	b^2	$c^2 = (a^2+b^2)$
0	0	0	0	0	0
0	1	1	0	1	1
0	2	1	0	1	1
1	0	1	1	0	1
2	0	1	1	0	1

Clearly in all these cases,

$$2ab = (a+b)^2 - c^2 = 0 \mod 3 \Rightarrow 3|ab$$

■

Q-3: Find a value of **n** in $3^{52} \equiv n \bmod 40$. **Max Marks: 5**

Answer:

$$3^{52} = 3^{4*13} = (3^4)^{13} = 81^{13} = 1^{13} = 1 \mod 40$$

Therefore $n = 1$. ■

Q-4: It follows from the Chinese Remainder Theorem that there is an isomorphism

$$\phi : \frac{Z}{20Z} \to \frac{Z}{4Z} \times \frac{Z}{5Z}$$

In this case what is $\phi^{-1}(1,3)$? **Max Marks: 5**
Answer: ϕ is the Chinese Remainder map and $\phi^{-1}(1,3)$ is the solution to the family of modular equations below:

$$a = 1 \mod 4$$
$$a = 3 \mod 5$$

Note that $4*5 = 20$. Let $n = 20, n_1 = 4, n_2 = 5, a_1 = 1, a_2 = 3$. Then $\left(\frac{n}{n_1}\right) = n_1^* = 5$ and therefore $n_1^* * 1 = 1 \bmod n_1$. So $1 = (n_1^*)^{-1} \bmod n_1$. Similarly $n_2^* = 4$ and $4 = (n_2^*)^{-1} \bmod n_2$. Therefore by the Chinese Remainder Theorem the solution to the modular equations is

$$\begin{aligned}\phi^{-1}(1,3) &= ((n_1^*)^{-1} \mod n_1) * n_1^* * a_1 + ((n_2^*)^{-1} \mod n_2) * n_2^* * a_2 \\ &= 1*5*1 + 4*4*3 \\ &= 53 = 13 \mod 40\end{aligned}$$

■

Q-5: Let p and q be distinct odd primes then $p^{q-1} + q^{p-1} \equiv 1 \bmod pq$. **Max Marks: 5**
Answer: From Fermat's little theorem

$$p^{q-1} \equiv 1 \mod q \Rightarrow p^q \equiv p \mod pq$$
$$q^{p-1} \equiv 1 \mod p \Rightarrow q^p \equiv q \mod pq$$

Adding the above two equations we get

$$\begin{aligned}p^q + q^p &\equiv (p+q) \mod pq \\ (p^{q-1} + q^{p-1})(p+q) - pq(p^{q-2} + q^{p-2}) &\equiv (p+q) \mod pq \\ (p^{q-1} + q^{p-1})(p+q) &\equiv (p+q) \mod pq \\ p^{q-1} + q^{p-1} &\equiv 1 \mod pq\end{aligned}$$

In the last step we have used the fact that $\texttt{gcd}(p+q, pq) = 1$ when p, q are primes. ■

Q-6: Find all solutions of the congruence $57X \equiv 87 \bmod 105$. **Max Marks: 5**
Answer: $\texttt{gcd}(57, 105) = 3$. Therefore z is a solution to the equation $57X \equiv 87 \bmod 105$ iff it is a solution to $19X \equiv 29 \bmod 35$ (after dividing the original equation by 3). Now $\texttt{gcd}(19, 35) = 1$. Therefore $(19^{-1} \bmod 35).29$ will be the solutions to this equation. $19^{-1} \equiv 24 \bmod 35$. So $\forall z \in Z, (35z + 24) * 29 = 31 \bmod 35$ are all the solutions to the equation $57X \equiv 87 \bmod 105$. To see why $24 = 19^{-1} \bmod 35$, we need to run the EGCD algorithm on the pair $(35, 19)$ as shown below. Therefore $35*6 + 19*(-11) = 1 \bmod 35$ where 6 and -11 come from the last row on

r	**r'**	**s**	**s'**	**t**	**t'**	q
35	19	1	0	0	1	1
19	16	0	1	1	-1	1
16	3	1	-1	-1	2	5
3	1	-1	6	2	-11	3
1	0	**6**	-19	**-11**	35	0

the EGDC table (the highlighted items in the last row). Therefore $19^{-1} \equiv -11 \equiv 24 \bmod 35$.

■

2 Algorithms

Q-1: Show that the following algorithm (known as the *Repeated Squaring Algorithm*) correctly computes a^n for any two positive integers a, n. Assume that $n \equiv (b_{l-1}, b_{l-2}, ..., b_0)$ is the binary representation of n where b_0 is the least significant bit. Note that $l = \texttt{len}(n)$. What is its

1: Initialize $p \leftarrow 1, \quad m \leftarrow a, \quad i \leftarrow 0$
2: **repeat**
3:     **if** $b_i == 1$ **then**
4:         $p \leftarrow p * m$
5:     **end if**
6:     $m \leftarrow m^2, \quad i \leftarrow i + 1$
7: **until** $i == l$
8: **return** p

running time complexity in terms of l and $\texttt{len}(a)$, where $\texttt{len}(a)$ is the size of the representation (binary or otherwise) of a? **Max Marks: 10**

Answer: Correctness proof involves adding a couple of invariants as assertions in the algorithm, as shown below.

1: Initialize $p \leftarrow 1, \quad m \leftarrow a, \quad i \leftarrow 0$
2: **repeat**
$\rightarrow$`Assertion 1:` $m = a^{2^i}$
3:     **if** $b_i == 1$ **then**
4:         $p \leftarrow p * m$
5:     **end if**
$\rightarrow$`Assertion 2:` $p = a^{\sum_{j=0}^{i} \delta(b_j == 1) 2^j}$
$\triangleright$ $\delta(b_j == 1) = 1$ only when $b_j == 1$, 0 otherwise
6:     $m \leftarrow m^2, \quad i \leftarrow i + 1$
7: **until** $i == l$
8: **return** p

Assertion 2, for $i = (l - 1)$ implies that

$$p = a^{\sum_{j=0}^{l-1} \delta(b_j=1) 2^j}$$

and clearly $n = \sum_{j=0}^{l-1} \delta(b_j = 1) 2^j$. Therefore the algorithm ends with $p = a^n$ as required. It remains to be shown that both the assertions hold for all the iterations in the `repeat` loop. Both the assertions can be shown to be true easily by induction on i, with $i = 0$ as the base case. For Assertion 1, $m = a$ and for Assertion 2 $p = a$ if $b_0 == 1$ and 1 otherwise, both of which can be easily verified. For the induction step if $m = a^{2^i}$ in iteration i then in iteration $(i + 1)$, $m = \left(a^{2^i}\right)^2 = a^{2^i.2} = a^{2^{i+1}}$ (from Step 6). Therefore at step 4,

$$p_{i+1} = p_i * m = p_i * a^{2^{i+1}} = \left(a^{\sum_{j=0}^{i} \delta(b_j == 1) 2^j}\right).a^{2^{i+1}}$$

if $b_{i+1} == 1$ and $p_{i+1} = p_i$ otherwise (from Step 3). Hence $p = a^{\sum_{j=0}^{i+1} \delta(b_j == 1) 2^j}$ after step 5 in iteration $(i + 1)$.

The algorithm goes through the `repeat` loop exactly l times and carries out at most one multiplication (step 5) in each iteration. Also in each iteration $p, m \leq a^n$. Therefore

$$T(a, n) = O\left(l * (\texttt{len}(a^n))^2\right)$$

If we know that $0 < a^n \leq M$ for some bound M then the complexity reduces to $O\left(l * (\texttt{len}(M))^2\right)$. ■

Q-2: **Application 1 of the Result of the Repeated Squaring Algorithm:** Pseudo-random numbers are often generated using an algorithm called a *linear congruential generator*. In this we choose a relatively large modulus M (with unknown factorization), a multiplier a, a constant c and seed value X_0. Successive pseudo-random numbers are generated using the recurrence

$$X_n = a.X_{n-1} + c \mod M$$

Give an algorithm to compute X_n in time polynomial in $\texttt{len}(n), \texttt{len}(M)$ assuming that $0 < a, c < M$. **Max Marks: 10**

Answer:

$$\begin{aligned}
X_n &= a.X_{n-1} + c \mod M \\
&= a\left(a.X_{n-2} + c\right) + c \mod M = a^2.X_{n-2} + c(1+a) \mod M \\
&= \ldots \\
&= a^k.X_{n-k} + c\sum_{i=0}^{k-1} a^i \mod M = a^k.X_{n-k} + c\frac{a^k - 1}{a-1} \mod M
\end{aligned}$$

For $k = n$ we get $X_n = a^n.X_0 + c\frac{a^n-1}{a-1} \bmod M$. Evaluating this expression involves 2 multiplications and a division all of which involve numbers that are bounded by M amounting to a running time of $O((\texttt{len}(M))^2)$. We know from Problem 1 that the time to compute a^n (with all intermediate values bounded by M) is $O(\texttt{len}(n)(\texttt{len}(M))^2)$. The total running time therefore is $O(\texttt{len}(n)(\texttt{len}(M))^2)$. ■

Q-3: Consider the following algorithm ($\texttt{len}(n)$ denotes the size of the representation – not necessarily binary – of n):

Initialize

$$k \leftarrow \left\lfloor \frac{\texttt{len}(n) - 1}{2} \right\rfloor, \quad m \leftarrow 2^k$$

for $i = (k-1)$ **downto** 0 **do**
  if $(m + 2^i)^2 \leq n$ **then**
    $m \leftarrow m + 2^i$
  end if
end for
return m

1. Show that this algorithm correctly computes $\lfloor \sqrt{n} \rfloor$. **Hint**: Think of the bit representation of m even though our internal representation may not be binary!!

2. Show how this algorithm can be implemented in time $O(\texttt{len}(n)^2)$. Can this be improved if we assume that we are working with a binary representation?
3. Extend this algorithm to compute $\lfloor n^{1/e} \rfloor$, assuming $n \geq 2^e$. What will its running time complexity be?

Max Marks: 4 + 4 + 2

Answer:

1. As we did for Problem 1, we again show the correctness of the algorithm by employing appropriate invariant assertions. We will in fact prove something more general covering both Parts 1 and 3 of the question. $e = 2$ in the version of the algorithm below will prove the correctness of the original algorithm. **Note**: There was a typo in the question paper — apologies for that. We will take care of this during correction of the answer scripts. In the initialization it should have been $\texttt{len}_2(n)$ and not $\texttt{len}(n)$ — $\texttt{len}_2(n)$ denotes the *binary length* of n though the representation is not necessarily binary.

Initialize

$$k \leftarrow \left\lfloor \frac{\texttt{len}_2(n) - 1}{e} \right\rfloor, \quad m \leftarrow 2^k$$

→`Assertion 1`: $m = 2^k \leq n^{1/e} < 2^{k+1} \Rightarrow \lfloor n^{1/e} \rfloor$ is a $(k+1)$-bit number with no leading 0's i.e., $n^{1/e} = 2^k + \delta$ for some $\delta < 2^k$

for $i = (k-1)$ **downto** 0 **do**
 if $(m + 2^i)^e \leq n$ **then**
  $m \leftarrow m + 2^i$
 end if
→`Assertion 2`: $m \leq n^{1/e} < (m + 2^i)$
end for
return m

Proof of Assertion 1:

$$\begin{aligned}
\left\lfloor \frac{\texttt{len}_2(n) - 1}{e} \right\rfloor \leq \frac{\texttt{len}_2(n) - 1}{e} & \quad < \left\lfloor \frac{\texttt{len}_2(n) - 1}{e} \right\rfloor + 1 \quad \text{(from the definition of } \lfloor . \rfloor\text{)} \\
k \leq \frac{\texttt{len}_2(n) - 1}{e} & \quad < k + 1 \\
ek \leq (\texttt{len}_2(n) - 1) & \quad < e(k+1) \\
2^{ek} \leq 2^{\texttt{len}_2(n)-1}; \quad \texttt{len}_2(n) & \quad \leq e(k+1) \\
m^e = (2^k)^e \leq 2^{\texttt{len}_2(n)-1}; \quad 2^{\texttt{len}_2(n)} & \quad \leq (2^{k+1})^e \\
m^e = (2^k)^e \leq 2^{\texttt{len}_2(n)-1} \leq n < 2^{\texttt{len}_2(n)} & \quad \leq (2^{k+1})^e \\
m \leq n^{1/e} & \quad < 2^{k+1}
\end{aligned}$$

Note that when $i = 0$ (last iteration) Assertion 2 guarantees that $m \leq n^{1/e} < (m+1)$ which implies from the definition of $\lfloor . \rfloor$ that $m = \lfloor n^{1/e} \rfloor$.

Proof of Assertion 2: Let the binary representation of $\lfloor n^{1/e} \rfloor$ be $(1, b_{k-1}, b_{k-2}, \ldots, b_0)$. So $\lfloor n^{1/e} \rfloor = 2^k + \sum_{i=0}^{k-1} b_i 2^i$. The algorithm starts with $m = 2^k$ as in Assertion 1 and adds 2^i to m if $b_i == 1$, starting from b_{k-1} till b_0. It is convenient to subscript m with the iteration index i for the proof — let's denote the value of m at Assertion 2 in iteration i as m_i. So the assertion we need to prove is $m_i \leq n^{1/e} < (m_i + 2^i)$. The induction hypothesis implies $m_{i+1} \leq n^{1/e} < (m_{i+1} + 2^{i+1})$. Note that Assertion 1 is in fact the base case with $i = k$. There are two cases (step 3):

$\boldsymbol{(m_{i+1} + 2^i)^e \leq n}$: In this case $m_i = m_{i+1} + 2^i$ (step 4). Therefore trivially $m_i \leq n^{1/e}$ (the case condition). Also $m_i + 2^i = m_{i+1} + 2^i + 2^i = m_{i+1} + 2^{i+1} > n^{1/e}$ (induction).

$\boldsymbol{(m_{i+1} + 2^i)^e > n}$: Here $m_i = m_{i+1}$. Therefore $n^{1/e} < m_{i+1} + 2^i = m_i + 2^i$ (the case condition) and $m_i = m_{i+1} \leq n^{1/e}$ (induction).

2. The following is a version of the algorithm for square root that makes the implementation more explicit. The idea is to remember the value of m^2 from the earlier iteration and

---

1: Initialize

$$k \leftarrow \left\lfloor \frac{\texttt{len}_2(n) - 1}{2} \right\rfloor, \quad m \leftarrow 2^k, \quad \texttt{square} \leftarrow m^2$$

2: **for** $i = (k-1)$ **downto** 0 **do**
3:     $\texttt{tmp} \leftarrow \texttt{square} + 2^{i+1}.m + 2^{2i}$
4:     **if** $tmp \leq n$ **then**
5:         $m \leftarrow m + 2^i$
6:         $\texttt{square} \leftarrow \texttt{tmp}$
7:     **end if**
8: **end for**
9: **return** m

---

exploit the fact that $(m + 2^i)^2 = m^2 + 2^{i+1}m + 2^{2i}$ where for m^2 on the RHS we simply recall the value of m^2 stored from the previous iteration. The second term on the RHS can be implemented in time $O(\texttt{len}(n))$ using bit-shifts (it is a multiplication by a power of 2). The number of iterations that the loop in steps 2–8 will execute is also $O(\texttt{len}(n))$. The total running time is therefore $O((\texttt{len}(n))^2)$. This is no more than what it would be if the representation was base 2.

The squaring trick works only for the *square root* version of the algorithm. For the more general version for $n^{1/e}$ we need to compute $(m + 2^i)^e$ in every iteration. From Problem 1 the time taken to do this would be (since all the intermediate values are bounded by n) $O(\texttt{len}(e)(\texttt{len}(n))^2)$. Total running time therefore is $O(\texttt{len}(e)(\texttt{len}(n))^3)$.

∎