6.046/18.410 Problem Set 5

Yijun Jiang
Collaborator: Hengyun Zhou, Eric Lau

October 19, 2015

1 Fun Factoring

1.1 Part (a)

Remark:

In order to make the statement true, we need to assume $p > 2$.

Proof:

Let $x_1, x_2 \in (0, p)$ be two integer solutions to $a \equiv x^2 \pmod{p}$, where $a \in \mathbb{Z}_p^*$. Then $x_1^2 \equiv x_2^2 \pmod{p}$, which means $p|(x_1 - x_2)(x_1 + x_2)$. Since p is prime, either $p|(x_1 - x_2)$ or $p|(x_1 + x_2)$. Therefore, $x_1 \equiv x_2 \pmod{p}$ or $x_1 \equiv -x_2 \pmod{p}$. Since $x_1, x_2 \in (0, p)$, in the first case, $x_1 = x_2$, and in the second case, $x_2 = p - x_1$. Any two solutions can be classified into these two cases, so the total number of solutions is no more than 2.

Now we show that the number of solutions is exactly 2. Since a is a quadratic residue, it has at least one square root x. Then according to the previous argument, $p - x$ is also a solution. If $x = p - x$, p must be even, which contradicts with the assumption that $p > 2$ is prime. Therefore, $x \neq p - x$ and there are exactly 2 solutions.

In the proof above we only assumed $x \in \mathbb{Z}_p$, the group of integers under addition modulo p. We do not need to use the stronger assumption that $x \in \mathbb{Z}_p^*$.

1.2 Part (b)

Remark:

We must assume $p \neq q$ to make every quadratic residue mod N have exactly four distinct square roots. Otherwise, say $N = 9 = 3^2$, the quadratic residue 1 only has two square roots, namely 1 and 8.

Description:

We first pick a random x from all positive integers less than N (i.e. $\mathbb{Z}_N - \{0\}$). Then we check if $\gcd(x, N) = 1$. If so, $a = x^2$ is a quadratic residue. Notice: this coprimality check only serves to make sure that a is a valid first parameter of $\textsc{Sqrt}$. It becomes unnecessary if $\textsc{Sqrt}$ can take care of general inputs from $\mathbb{Z}_N$ rather than $\mathbb{Z}_N^*$.

Calculate $y = \textsc{Sqrt}(a, N)$. If $y \equiv \pm x \pmod{N}$, then discard x and start all over (pick another random integer). Otherwise, $p = \gcd(x - y, N)$ and $q = \gcd(x + y, N)$ are the two distinct nontrivial prime factors of N.

If the coprimality check is applied and $\gcd(x, N) \neq 1$, then we can either discard this x and restart, or store $p = \gcd(x, N)$ and jump out of the loop. p is one of the two nontrivial prime factors of N. The other one can be calculated by division (e.g. long integer division).

Correctness:

For a randomly chosen x, if $\gcd(x, N) = 1$, x is coprime with N and so is $a = x^2$. Thus $a \in \mathbb{Z}_p^*$ and we can call $\textsc{Sqrt}(a, N)$ to deterministically calculate one of its four square roots. By the theorem, a has 4 distinct square roots. No matter which one is chosen as x, $\textsc{Sqrt}(a, N)$ gives the same y. Therefore, there exists an x such that $y \not\equiv \pm x \pmod{N}$. Moreover, conditioned on x being coprime with N, the probability of $y \not\equiv \pm x \pmod{N}$ is $\frac{1}{2}$.

$y \not\equiv \pm x \pmod{N}$ implies that $x - y \not\equiv 0 \pmod{N}$ and $x + y \not\equiv 0 \pmod{N}$. Since $x^2 \equiv y^2 \pmod{N}$, $N|(x-y)(x+y)$. Yet both $x-y$ and $x+y$ are not multiples of $N = pq$. Since p, q are distinct prime numbers, WLOG, $p|(x-y)$ and $q|(x+y)$. So $p|\gcd(x-y, N)$. Moreover, $q\not|(x-y)$, otherwise it contradicts with the fact that $N\not|(x-y)$. Thus $\gcd(x-y, q) = 1$. This implies that $\gcd(x-y, N) = \gcd(x-y, \frac{N}{q}) = \gcd(x-y, p)$. Combining with $p|\gcd(x-y, N)$, we get $p = \gcd(x-y, N)$. Similarly, $q = \gcd(x+y, N)$. This completes the factorization of N.

On the other hand, if the coprimality check gives $\gcd(x, N) \neq 1$, we can obviously discard x and restart until we pass the coprimality check. A simpler way is to notice that, since $x < N$, $\gcd(x, N) < N$ and thus can only be 1 or p, q. If x is not coprime with N, WLOG, $\gcd(x, N) = p$, we can immediately calculate q by division, thus completing the factorization of N.

Due to the finite probability of getting $y \not\equiv \pm x \pmod{N}$, this algorithm terminates in finite time. Once it terminates, it always outputs the correct result, as we have discussed above.

Runtime:

The group $\mathbb{Z}_N^*$ has order $|\mathbb{Z}_N^*| = N - (p+q) + 1$ (removing multiples of p and q from all positive integers less than N). Randomly selecting an x in $\mathbb{Z}_N - \{0\}$, we have a probability of $\frac{N-(p+q)+1}{N-1}$ to get an $x \in \mathbb{Z}_N^*$, and thus an $a \in \mathbb{Z}_N^*$. However, as is discussed above, half of such x are equivalent to $\pm\textsc{Sqrt}(a, N)$ upto modulo N, which should be discarded. Therefore, the probability for a good x is $P[good] = \frac{N-(p+q)+1}{2(N-1)}$.

Let us assume that the coprimality check is applied, so $P[bad] = P[good]$. $P[bad]$ excludes the probability of failing the check, which is given by $P[fail] = \frac{p+q-2}{N-1}$. A good choice of x calls Sqrt and ends the Las Vegas algorithm, while a bad choice of x calls Sqrt and continues the loop. A failed x also ends the loop but does not call Sqrt. Therefore, the expected number of calls is given by

$$P[good] + P[bad] + P[bad]E[k] = E[k]$$
$$E[k] = \frac{2(N-(p+q)+1)}{(N-(p+q)+1) + 2(p+q-2)} \leqslant 2$$

For now, I will use 2 as an upper bound for $E[k]$. When $N \gg p, q \gg 1$, $E[k]$ approaches this bound. Let $n = \log N$ be the size of input. The algorithm also involves $O(n^2)$ work on coprimality check (constant amount of checks are expected, each costing $O(n^2)$ time due to gcd calculation), $O(n^2)$ work on possibly failed x (long division, for example, takes $O(n^2)$ time), and $O(n^2)$ work on factorization using gcd. Therefore, the overall runtime is $E[T] = 2T_{\textsc{Sqrt}}(n) + O(n^2)$. Again notice that n is the number of bits in N.

Now I can answer the questions raised in this part.

1. Q: How many calls of Sqrt?
   A: The expectation number is very close to, and bounded by, 2.

2. Q: Expected running time?
   A: $E[T] = 2T_{\textsc{Sqrt}}(n) + O(n^2)$, where $n = \log N$.

3. Q: Relative difficulty of computing square roots modulo N and factoring N?
   A: Once the square roots of quadratic residues are calculated, it becomes much easier to factor N. Since factoring is extremely difficult, calculating square roots modulo N must be equally difficult.

1.3 Part (c)

Description:

We use the same idea to factor N multiple times until we get all prime factors. In other words, in the i-th round of factorization, we take all the composite factors of N that are calculated in the $(i-1)$-th round as inputs and factor them. We check primality on the resulting smaller factors, separating prime ones and composite ones. The prime factors are eventually returned as outputs, while the composite ones are used as inputs of the next round. The algorithm terminates when all factors are prime.

In each round of factorization (say N' is being factored using a good x' and a y'), the only difference from the previous part is, although $y' \not\equiv \pm x' \pmod{N}$ and $N'|(x'-y')(x'+y')$ still holds, we cannot reach the conclusion that $\gcd(x'-y', N') \times \gcd(x'+y', N') = N'$. We can take $\gcd(x'-y', N')$ as the first nontrivial factor of N', and do a division to find the second nontrivial factor.

Correctness:

Each round of factorization breaks the remaining composite factors of N down to smaller prime or composite factors. Once a factor is confirmed to be prime, it no longer goes through the next round of factorization. Since N is finite, its unique factorization contains finite number of prime factors. So this algorithm will terminate after finite rounds of factorization. The correctness of the entire factorization relies on the correctness of a single round.

In a certain factorization of a certain round, identical to the reasoning in the previous part, when x' is chosen such that $x'^2 \equiv y'^2 \pmod{p}$ but $y' \not\equiv \pm x' \pmod{N}$, we have $N'|(x'-y')(x'+y')$. $\gcd(x'-y', N')$ cannot be the trivial 1 or N' because both $x'-y'$ and $x'+y'$ are not multiples of N'. Then we factor N' into $\gcd(x'-y', N')$ and $N'/\gcd(x'-y', N')$. The division can be done in $O(n'^2)$ time by long division, where $n' = \log N'$.

Runtime:

Suppose N has k prime factors (including repeated powers), then in this algorithm $k-1$ factorizations are performed. Therefore, Sqrt is expected to be called $2(k-1) \approx 2k$ times, with the input size shrinking over iterations. We can thus give the following upper bound for the runtime, where $n = \log N$:

$$E[T] \leqslant 2kT_{\text{SQRT}}(n) + O(kn^2)$$

2 Fun with the Skiplist

2.1 Part (a)

1. Q: Probability of a node rising to the top level?
   A: $\frac{1}{\log u}$
2. Q: Expected number of keys between two top-level nodes?
   A: $\log u$
3. Q: Expected number of keys on the top level?
   A: $\frac{n}{\log u}$
4. Q: Expected running time of Find and Insert?
   A: $E[T_{\text{FIND}}] = O(\frac{n}{\log u} + \log\log u)$; $E[T_{\text{INSERT}}] = O(\frac{n}{\log u} + \log\log u)$

2.2 Part (b)

2.2.1 Subpart i

The running time is $T_{\text{SUCCESSOR}} = O(1)$. This is because each leaf node (except for the rightmost one who does not have a successor) is directly linked to its successor by a pointer.

2.2.2 Subpart iii

Description:

Correctness:

Runtime:

2.3 Part (c)

2.3.1 Subpart i

Since $\frac{n}{\log u}$ keys are expected to rise to the top level of the skiplist, $\frac{n}{\log u}$ keys will be stored in the top-level trie, on expectation.

2.3.2 Subpart ii

Description:

To deal with edge cases, suppose the skiplist has $-\infty$ nodes on its far left and ∞ nodes on its far right.

First call TrieFind(x) for the trie. If x is in the trie, this is an $O(1)$ operation. The algorithm returns a pointer to x and terminates. If TrieFind(x) reports that x does not exist in the trie, call TriePredecessor(x). This returns a node y on the top level of the skiplist (suppose Predecessor(x) returns the pointer to $-\infty$ if x does not have a predecessor in the trie).

We enter the second level of the skiplist from y and do a regular key search inside the skiplist. In other words, on each level we go right until we either find x or find a key larger than x. In the first case the algorithm returns a pointer to x and terminates. In the second case we go down a level from the rightmost node in the current level whose key is smaller than x. We keep doing this until we either find x or find that x does not exist down to the bottom level.

Correctness:

Runtime:

If x is in the trie, TrieFind(x) finishes the job in $O(1)$ time due to the Hash table structure. If x is not in the trie, TriePredecessor(x), just like TrieSuccessor(x), finds the predecessor of x in $O(\log\log u)$ time by bisecting the prefixes.

When searching inside the skiplist, since the entrance from the top level to the second level is already determined by TrieSuccessor(x), we save $O(\frac{n}{\log u})$ time finding this entrance. On each level constant number of nodes (2 nodes) are visited on expectation. And there are $O(\log\log u)$ levels. Therefore, the search time inside the skiplist is also $O(\log\log u)$.

The probability for an $O(1)$ runtime is $\frac{1}{\log u}$, while the probability for an $O(\log\log u)$ runtime is $1-\frac{1}{\log u}$. Therefore, the expected runtime is $E[T_{\text{Find}}] = \frac{1}{\log u}O(1) + (1-\frac{1}{\log u})O(\log\log u) = O(\log\log u)$.