6.046/18.410 Problem Set 5

Yijun Jiang
Collaborator: Hengyun Zhou, Eric Lau

October 22, 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, $p = \gcd(x - y, N)$ is a nontrivial prime factor 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 a nontrivial prime factor of N.

The other nontrivial prime factor q can be calculated by division, $q = N/p$ (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 1/2.

Suppose a randomly generated x satisfies $y \not\equiv \pm x \pmod{N}$. Since $x^2 \equiv y^2 \pmod{N}$, $N|(x-y)(x+y)$. If $\gcd(x-y,N)=1$, $N|(x+y)$, which contradicts with $y \not\equiv -x \pmod{N}$. And $\gcd(x-y,N)=N$ contradicts with $y \not\equiv x \pmod{N}$. Therefore, $\gcd(x-y,N)$ must be a nontrivial factor of N. WLOG, $p=\gcd(x-y,N)$.

On the other hand, if the coprimality check gives $\gcd(x,N) \neq 1$, we can discard x and restart until we pass the coprimality check. A more efficient way is to notice that, since $0<x<N$ and x is not coprime with N, $\gcd(x,N) \neq 1, N$ and is thus a nontrivial factor of N. WLOG, $p=\gcd(x,N)$.

Once we get a nontrivial factor p, it is straightforward to calculate q by division. Since N is the product of exactly two primes, we complete 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. Therefore, the correctness of this algorithm is proved.

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 $\textsc{Sqrt}$ and ends the Las Vegas algorithm, while a bad choice of x calls $\textsc{Sqrt}$ and continues the loop. A failed x also ends the loop but does not call $\textsc{Sqrt}$. 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 $\textsc{Sqrt}$?
   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 first factor out powers of 2 from N until we get an odd N'. Check if N' is a prime. If so, the algorithm terminates.

Otherwise, check if N' is a prime power. This is done by the following: set the initial base b as (at least) $2\lfloor\sqrt{N'}/2\rfloor + 1$ and initial power i as 2. Iterate as follows: reduce b by 2 until b is prime. Calculate b^i. If $b^i > N'$, reduce b by 2. If $b^i < N'$, increase i by 1. If $b^i = N'$, N' is a prime power and break the loop. If the loop is not broken, it goes on until b becomes smaller than 3 (since 2 is factored out, base 3 is sufficient).

Another way to check prime power is, for all intergers i ranging from 2 to $\log_3 N$, calculate the i-th root of N', call it α_i. This can be done by the fast converging n-th root algorithm. Compare $\lfloor\alpha_i\rfloor^i$ with N'. But for now we stick to the first method.

We then use essentially the same algorithm as the previous part to factor N'. The difference is, this time the two factors may not be prime. So primality checks are performed. For every composite factor of N', we do a prime power check. If a composite factor is a prime power, this check automatically factors it. Otherwise, we recursively factor it with the previous algorithm. This process goes on until every output factor of the previous algorithm is either a prime or a prime power.

The pseudocode is shown below.

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 runtime 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:

Assume the trie is nonempty. First we try to find x in the trie by calling $\mathrm{H}_h(x)$, where $h = \log u$ is the height of the trie and length of a key. If x is in the trie, hashing returns a pointer to x and we have access to its successor (or if x is the largest key in the trie, we know x does not have a successor) in $O(1)$ time.

If x is not in the trie, we bisect its digits to find the internal node n who has the longest prefix matched with x. Specifically, we keep a "matched prefix" m which is initially set to empty. We also keep an "unchecked substring" uc which is a substring of x and is initially set as x (no need to store as an extra copy, just pin down the start and end on x). Each time we cut uc in half: $uc = uc_1 + uc_2$. If $m + uc_1$ is in the trie (we check this by hashing), append uc_1 to m and set uc_2 as the new uc. Otherwise, set uc_1 as the new uc. This bisection is done until the base case where uc has only one digit (in the base case if $m + uc$ is in the trie, append uc to m). Then we can locate n by hashing with m (or if m is empty, n is the root r).

n must have only one branch. So it also has a pointer connecting to a leaf node l. If the branch that n has is the left branch, the successor of l is the successor of x (or if l does not have a successor, neither does x). Otherwise, l itself is the successor of x.

Correctness:

If x is in the trie, hashing returns a pointer to x and we can locate its successor by the pointer from x to its successor.

We focus on the case where x is not in the trie. Since m is initially empty and is extended by appending uc_1 to it, m is the substring of x that directly precedes uc. Since each time uc_2 is discarded only when $m+uc_1$ does not match x, the discarded substring (the substring that follows uc) cannot contribute to extend the prefix match. Therefore, in each bisection, the longest matched prefix of x is a substring of $m + uc$.

Since uc is bisected in each iteration, it takes finite number of iterations to cut uc down to the base case. After dealing with the base case, the longest matched prefix of x is a substring of m. But m itself is a matched prefix. So m is the longest matched prefix. So the node n acquired by hashing with m is the internal node with the longest prefix matched with x.

n does not have the branch corresponding to x, otherwise one of its children will match x with a longer prefix. Therefore, n has a pointer connecting either to the minimum key in its right subtree or the maximum key in its left subtree, depending on which subtree it has. If it has a left subtree, then x corresponds to its right subtree. This means that the maximum key in its left subtree is the largest key in the trie that is smaller than x. So its successor is the successor of x (if this node does not have a successor, neither does x). If it has a right subtree, then x corresponds to its left subtree. This means that the minimum key in its right subtree is the smallest key in the trie that is larger than x. So it is the successor of x. This proves the correctness of the algorithm.

Runtime:

When x is in the trie, the runtime is $O(1)$. When x is not in the trie, the runtime recursion is the same as bisection search.

$$T(h) = T(h/2) + O(1)$$

which implies that $T(h) = O(\log h)$. Since $h = \log u$, we have $T_{\textsc{Successor}}(u) = O(\log \log u)$.

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), which simply uses the hash table on the bottom level of 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 bottom level of the trie, which is also the top level of the skiplist (suppose TriePredecessor(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 to the next node 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 (in which case we can give a warning and return the predecessor of x on the bottom level of the skiplist).

Correctness:

If x is in the trie, its pointer can be obtained by hashing in constant time, which completes FIND. Otherwise, symmetric to TRIESUCCESSOR described above, we can use TRIEPREDECESSOR to obtain the predecessor of x in the trie, which is named y. The successor of y in the trie, as well as on the top level of the skiplist, must be larger than x. Therefore, y is the node from which we should go down a level in the skiplist. After that, we go back to the usual searching strategy in a skiplist.

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 $E[T_{\text{FIND}}] = \frac{1}{\log u}O(1) + (1-\frac{1}{\log u})O(\log\log u) = O(\log\log u)$.

2.3.3 Subpart iii

Description:

For simplicity assume the structure is doubly linked.

First call FIND(x). If x already exists in the structure, there is no need to insert it. Otherwise we can get the predecessor of x on the bottom level of the skiplist. We do the usual skiplist inserting procedure. Specifically, insert x on the bottom level (create a new node and set the pointers correctly). Decide by a random value if x should be promoted to the level above. If so, trace back to the first existing node that is promoted to the level above and insert x right after it on the level above. Keep doing this until either in a random value generation x is not promoted further up, or x has reached the top level of the skiplist. In the first case, the algorithm terminates. In the second case, we need to add x into the trie.

When x is inserted into the top level of the skiplist, it automatically appears on the bottom level of the trie. Add x to the bottom-level hash table. Keep tract of the newly created subtree, which for now is just the leaf x. Let the root of this subtree be y. Do the following procedure to enlarge this subtree. Take off the last digit in the prefix and use the hash table to detect if this new prefix node exists on the level above. If so, call it n. Delete the augmented node-leaf link that connects n to either the minimum key of its rightmost subtree or the maximum key of its leftmost subtree, and link n to y. Otherwise, create a new node z with the shortened prefix and add it to the hash table on the level above. Link z with y and create an augmented pointer from z to x. Then update y to be z. Repeat this procedure until the prefix node already exists.

Once we get the pre-existing node n, we need to update the augmented pointers on the path from the root r to n. Start from n and go upwards. Keep tract of whether x is the minimum key or the maximum key of the current node by two flags $isMin$ and $isMax$. For n, if x is in its left subtree, $isMin$ is set True and $isMax$ is set False. Otherwise $isMin$ is set False and $isMax$ is set True. Say we go up to node m along this path. If m has two children, $isMin$ and $isMax$ are updated in the following way: if x is in the left subtree and $isMax$ is TRUE, update $isMax$ to FALSE; if x is in the right subtree and $isMin$ is TRUE, update $isMin$ to FALSE. If m has only one child, $isMin$ and $isMax$ do not need to be updated, but the augmented pointer is updated in the following way: if m has a left child and $isMax$ is TRUE, or if m has a right child and $isMin$ is TRUE, update the augmented pointer of m to x.

The pseudocode is shown below.

Correctness:

Algorithm 1 Inserting into the combined structure

```
procedure Insert($x$)
    $y \leftarrow$ Find($x$)
    if $y = x$ then Report $x$ already exists
    else Insert $x$ into the skiplist bottom-up
    if $x$ is promoted to the top level then
        $pre \leftarrow$ the key stored in $x$, add $pre$ into the hash table of level Len($pre$)
        $y \leftarrow x$, $n \leftarrow$ NULL
        while $n =$ NULL do                                          ▷ Create a new branch containing $x$
            $last \leftarrow$ last digit of $pre$
            $pre \leftarrow$ first Len($pre$) $- 1$ digits of $pre$
            if Len($pre$) $= 0$ then $n \leftarrow r$
            else $n \leftarrow$ $\textsc{Hash}_{\textsc{Len}(pre)}(pre)$
            if $n =$ NULL then
                Add $pre$ into the hash table of level Len($pre$) and create a corresponding node $z$
                if $last = 0$ then $z.leftChild = y$
                else $z.rightChild = y$
                $z.augPointer \leftarrow x$
                $y \leftarrow z$
        $n.augPointer \leftarrow$ NULL
        if $last = 0$ then
            $n.leftChild \leftarrow y$, $isMin \leftarrow$ True, $isMax \leftarrow$ False
        else
            $n.leftChild \leftarrow y$, $isMin \leftarrow$ False, $isMax \leftarrow$ True
        while $n \neq r$ do                                          ▷ Update the augmented pointers
            $m \leftarrow n.father$
            if $m.leftChild \neq$ NULL and $m.rightChild \neq$ NULL then
                if $n = m.leftChild$ and $isMax =$ True then $isMax \leftarrow$ False
                else if $n = m.rightChild$ and $isMin =$ True then $isMin \leftarrow$ False
            else
                if $n = m.leftChild$ and $isMax =$ True then $m.augPointer \leftarrow x$
                else if $n = m.rightChild$ and $isMin =$ True then $m.augPointer \leftarrow x$
```

Suppose x does not exist in the structure. Then FIND(x) returns the predecessor of x on the bottom level of the skiplist. Inserting x into the skiplist is just as usual: a bottom-up insertion using randomness to decide if x is promoted to the level above. If x gets promoted into the topmost level of the skiplist, it must be further inserted into the trie. This is also done in a bottom-up fashion. Besides the leaf node x, all the internal nodes that previously do not exist have to be created. The algorithm guarantees that these nodes are linked in the correct order. Moreover, all such nodes have only path to go down to the leaf level. So their augmented pointers should all point to x.

Since we do this bottom-up, we eventually get to the pre-existing node n whose prefix has the longest match with x. The newly created subtree must be a branch of n. By adding this subtree, n has now two children and no longer needs an augmented pointer. So the algorithm deletes it.

Finally, all the nodes above n that contains this new subtree are updated in terms of their augmented pointers. This is necesssary when a node has exactly one child (so it needs such a pointer) and if either x becomes the new maximum key in the left subtree, or x becomes the new minimum key in the right subtree. This is the reason why we keep two flags $isMin$ and $isMax$. These two flags are updated only if there is another branch whose leaf takes over x as the minimum key or the maximum key.

After all these steps, it is guaranteed that (a) a path is established from r to x (b) all the nodes on this path with only one child has the correct augmented pointer. The nodes not on this path are not affected because their subtrees do not contain x. This proves the correctness of the algorithm.

Runtime:

If x is not promoted to the top level of the skiplist, the insertion only happens in the skiplist. The expected rumtime in this case is proportional to its height, $O(\log\log u)$. The probability for this case is $1 - \frac{1}{\log u}$

If x is promoted to the top level of the skiplist, then besides $O(\log\log u)$ time inserting inside the skiplist, additional time is spent inserting x into the trie. Since the algorithm does this bottom-up, spending only constant time in each level, the additional runtime is linear in the trie height, $O(\log u)$. The probability for this case is $\frac{1}{\log u}$.

Therefore, the overall expected runtime is

$$E[T_{\text{INSERT}}] = \left(1 - \frac{1}{\log u}\right) O(\log\log u) + \frac{1}{\log u}(O(\log\log u) + O(\log u)) = O(\log\log u)$$