Problem Set 7 Solutions

This problem set is due **at 11:59pm** on **Thursday, November 3, 2016.**

EXERCISES (NOT TO BE TURNED IN)

- CLRS 18.1-4, page 491
- CLRS 18.2-3, page 497
- CLRS 31.2-3, page 938
- CLRS 31.4-2, page 950

Problem 7-1. Range sum skip-lists [50 points]

Ben Bitdiddle really enjoys playing with skip lists. He makes a modification to his skip list so that each data point is a (key, value) pair. These pairs are inserted into the skip list using the key, as discussed in lecture.

He now wants to find the sum of values with keys in the range (k_1, k_2) for various values of k_1 and k_2. Help him modify his data structure to answer this query in expected $O(\log n)$ time, while still retaining expected $O(\log n)$ running times for INSERT and DELETE.

(a) [30 points]

For a node n, let $n.succ$ be the next node on the same level. Define the range "covered" by n as $[k_1, k_2)$ where k_1 and k_2 are the keys associated with n and $n.succ$ respectively. Augment your skip list such that each node n also stores the sum $n.rsum$ of values in the range covered by n in addition to its own key and value. Describe how to perform INSERT and DELETE operations on this augmented skip list in $O(\log n)$ expected run time. Provide a proof of why your operations achieve this runtime bound.

Solution: Insert:

Let's say we are inserting the (key,value) pair (k, v).

Use the same insert algorithm as before to add a node n_0 at the lowest level linked list, with $n_0.key = k$ and $n_0.value = v$. Now, we start flipping a coin to add nodes at higher levels. Let n_i denote the node added at level i. We need to find $n_i.rsum$ for each node. Notice that $n_i.rsum$ is the sum of $n_{i-1}.rsum$ and all the successors of n_{i-1} (on the same level) that don't have any nodes above them. Note that each node has a node above it with probability 1/2. Thus in expectation, we will see two sucessors of n_{i-1} before terminating our sum. This means that we can find $n_i.rsum$ in $O(1)$ time.

Let $n.pred$ be the predecessor of node n. Since $n_i.pred$ now has a new successor (n_i itself), it covers a smaller range. We can update $n_i.pred.rsum$ by subtracting $n_i.rsum - v$ (since v didn't exist earlier).

Since we can add at most one n_i for each level, we perform this update on $O(\log n)$ levels. So it takes expected $O(\log n)$ time for this part of the algorithm.

If n_i is the last node we add, there are still levels above level i that are affected by the insertion. We add v to the nearest node on each level $j > i$ that has key value less than k. We can do this by traversing in the following way :

- Start at n_i
- If it possible to go up one level, go up and add v to the newly visited node.
- Otherwise, go left
- Terminate when we reach the root

By the same analysis as before we go left expected $O(1)$ times on each level before going up. Since there are $O(\log n)$ levels, this part of the update takes expected $O(\log n)$ time.

Delete: We do the reverse of the insert algorithm. On deleting nodes n_0, ... , n_i, we add $n_j.rsum - v$ back to $n_j.pred$ for $0 < j \leq i$. We then subtract v from the nearest node on each level $j > i$ that has key value ¡ k. This takes expected $O(\log n)$ time by the same analysis as before.

(b) [15 points]

For a given key k_1, describe an expected $O(\log n)$ time algorithm to find the sum PREFIX-SUM(k_1) of all values with key less than k_1

Solution: We modify the algorithm for FIND(K). Recall that this algorithm proceeds by starting at the root and going right if possible, otherwise going down, as long as the key of the current node is $\leq k$.

We now keep track of a variable sum initialized to 0 at first. Every time we are at a node n and we go right instead of down, we add the value of $n.rsum$ to sum. Intuitively, we are just accumulating the pre-computed range sum for all the subtrees we are skipping over, so the final value of sum is the sum of all values with keys less than k.

Since we have added only $O(1)$ operations per level, this algorithm still takes expected $O(\log n)$ time.

(c) [5 points]

Use the previous part to describe how Ben Bitdiddle can answer his queries in expected $O(\log n)$ time.

Solution: Return PREFIX-SUM(k_2) − PREFIX-SUM($k_1 + 1$)

Problem 7-2. Range sum B-Trees [50 points]

Alyssa P. Hacker really enjoys playing with B-Trees. She makes a modification to her B-Tree so that each node is a (key, value) pair, and these pairs are inserted into the B-Tree using the key. Let $t - 1$ be the minimum number of nodes in an internal (non-root, non-leaf) block of the B-Tree, so that every block has between $t - 1$ and $2t - 1$ nodes.

Alyssa wants to augment her B-Tree to support a RANGE-SUM(k_1, k_2) query which returns the sum of values with keys in the range (k_1, k_2).

(a) [15 points]

Alyssa decides to augment her data structure so that every node stores the sum of all values in its left subtree. Modify the INSERT operation to update these values in $O(t \log_t n)$ time. Think about how you would handle DELETE in $O(t \log_t n)$ time as well, but you don't have to include it in your solution.

Solution: For each node n, let $n.lsum$ denote the stored sum of values in its left subtree.

The INSERT operation proceeds in two stages - first, we find the leaf to add our current key to, and next we perform split operations going upwards until the B-tree property is satisfied. We will ensure that each of these stages maintains the invariant that each node n in our tree has the correct value of $n.lsum$.

For the first stage, we start at the root block and recurse by visiting one of the current block's subtrees. Observe that the (key,value) to be inserted affects $n.lsum$ for exactly one node in our current block - the node (if one exists) to the right of the subtree we are going to explore next. Thus, we increment $n.lsum$ for that node by value. We continue this until we get to the leaf where our new node is inserted. The recursion goes down $O(\log n)$ levels since it is bounded by the height of the tree. At each level, we perform $O(t)$ operations to find the next subtree to recurse on, and $O(1)$ operations to increment $n.lsum$ for the corresponding node. This is $O(t \log n)$ time overall.

Now, if the leaf block has more than $2t - 1$ nodes after the insert, we must split it by the algorithm described in lecture. Every time we split, we will ensure that the new configuration of the tree still has the correct values of $lsum$. Suppose we split a block by its median node m and promote m to a higher level. Then, the only values of $lsum$ that change are $m.lsum$ and $m.right.lsum$ where $m.right$ is the new node to the right of m. However, all of the nodes in the subtrees below these nodes still have the correct values of $lsum$. Thus we can update $m.lsum$ by simply adding the values of $lsum$ for all the nodes in the left child block of m. We can do the same thing for $m.right.lsum$. Since a block has $O(t)$ nodes, this takes $O(t)$ time per split operation. We perform at most $O(\log n)$ splits, and so running time for this step is $O(t \log n)$ as well.

Thus we take $O(t \log n)$ time overall to do the required update.

(b) [10 points]

Describe an $O(t \log_t n)$ algorithm to compute RANGE-SUM(k_1, k_2) using this augmented B-Tree.

Hint: You may find it useful to implement PREFIX-SUM(k), which computes the sum of values with keys in the range $(-\infty, k)$, as a subroutine.

Solution:

To implement PREFIX-SUM(k), we follow an algorithm similar to FIND(k), except that we also keep a variable sum that is initialized to 0 at first.

Every time we are at a node n within a block and we go to the successor of n (the node to the right) within the block, or recurse on the right subtree of n, we add the value $n.lsum + n.v$ to sum. Intuitively, we are just accumulating the pre-computed range sum for all the subtrees and nodes we are skipping over, so the final value of sum is the sum of all values with keys $\leq k$.

Since we have added only $O(1)$ operations per level, this algorithm still takes $O(\log n)$ time from the analysis of the B-tree find operation.

The final answer is then PREFIX-SUM($k2$) - PREFIX-SUM($k1 + 1$)

Alyssa realizes that her algorithm runs very slowly for large values of t. To mitigate this, she decides to alter the B-Tree SEARCH operation to perform binary search on the nodes of each block, instead of sequentially traversing through the nodes, to locate the appropriate child block.

(c) [25 points]

Describe a different augmentation of the B-Tree that allows us to implement RANGE-SUM (with Alyssa's binary search strategy) in $O(\log n)$ time (independent of t!). Briefly describe how to modify your INSERT from part (a) to still run in $O(t \log_t n)$ time.

Hint: $O(\log n) = O(\log t \log_t n)$

[In fact, you can perform $O(\log n)$ time INSERT and DELETE operations by storing a Fenwick tree at each node, but that is beyond the scope of this class]

Solution: Assume we have a node n in block b. We now store a value $n.presum$ which is the sum of

- $n.lsum$ for each node in block b to the left of (and including n)
- $n.v$ for each node in block b strictly to the left of (i.e. not including n)

If we have these values pre-computed at each node, we don't have to traverse through each block in $O(t)$ time during the PREFIX-SUM operation. We can binary search on the keys in the current block in $O(\log t)$ time to find the subtree we need to recurse on, and add $n.presum$ for the node n that this subtree is a right child of (if such n exists). We still need to traverse $O(\log_t n)$ levels so this algorithm takes $O(\log t \log_t n) = O(\log n)$ time.

Insert proceeds in the same way as described earlier. We know that each change to $n.lsum$ for any n affects $n.predsum$ for every node succeeding n in the block. This can be up to $O(t)$ nodes. But at each level in our INSERT algorithm, we changed $n.lsum$ for only $O(1)$ nodes. So, we can go through and update $n.predsum$ for the nodes that were affected by these changes in $O(t)$ time per level. There are $O(\log_t n)$ levels and so we still take $O(t \log_t n)$ time.

Problem 7-3. Ternary GCD [50 points]

Tom Tritfiddle has a special ternary computer. By virtue of its architecture, dividing by 3 and finding the remainder of a number mod 3 are really efficient operations. To take advantage of these properties, he asks you to come up with an algorithm that will quickly compute the GCD (greatest common divisor) of two numbers.

(a) [5 points]
First, prove that if $a > b$, then $\text{GCD}(a, b) = \text{GCD}(a - b, b)$.

Solution: Let $g = \text{GCD}(a, b)$ and $g' = \text{GCD}(a - b, b)$. Since $g|a$ and $g|b$, $g|a - b$. Since $g|a - b$ and $g|b$, it divides their GCD, so $g|g'$.
Now, $g'|a - b$ and $g'|b$ and so it divides their sum $g'|(a - b) + b = a$. Since $g'|b$ and $g'|a$, it divides their GCD, so $g'|g$.
These two relationships imply that $g = g'$

(b) [5 points] Now prove that if $a \equiv b \bmod 3$, neither a nor b are divisible by 3 and $a > b$, then $\text{GCD}(a, b) = \text{GCD}(\frac{a-b}{3}, b)$.

Solution: $\text{GCD}(a, b) = \text{GCD}(a - b, b)$. But since $a - b$ is divisible by 3 and b is not, $\text{GCD}(a - b, b) = \text{GCD}(\frac{a-b}{3}, b)$

(c) [10 points] Prove that if $a \not\equiv b \bmod 3$, $a > b$, and none of them are divisible by 3, then

$$\text{GCD}(a, b) = \begin{cases} \text{GCD}\left(\frac{a-2b}{3}, b\right) & \text{if } a > 2b \\ \text{GCD}\left(\frac{2b-a}{3}, a - b\right) & \text{if } a \leq 2b \end{cases}$$

Solution: If $a > 2b$, then $\text{GCD}(a, b) = \text{GCD}\,(a - b, b) = \text{GCD}\,(a - 2b, b)$ using part (a). Now, notice that we have two cases: $a \equiv 1 \mod 3$ and $b \equiv 2 \mod 3$ or $a \equiv 2 \mod 3$ and $b \equiv 1 \mod 3$. In either case $a - 2b \equiv 0 \mod 3$ but b is still not divisible by 3. Thus, using the same argument as in the solution for part (b), $\text{GCD}\,(a - 2b, b) = \text{GCD}\left(\frac{a-2b}{3}, b\right)$.
If $a < 2b$, then $\text{GCD}(a, b) = \text{GCD}\,(a - b, b) = \text{GCD}\,(a - b, 2b - a)$ using part (a). We then proceed in the same way.

(d) [30 points]
Use your previous observations to devise an algorithm that computes the GCD of two numbers a and b in $O(\log(\max(a, b)))$ time. Prove its correctness and running time.

Solution: Note that if a and b are both divisible by 3, then $\text{GCD}(a, b) = 3{\cdot}\text{GCD}(\frac{a}{3}, \frac{b}{3})$. If a is divisible by 3 and b is not, $\text{GCD}(a, b) = \text{GCD}(\frac{a}{3}, b)$.
Thus, our algorithm proceeds with the following transformations (assuming $a \geq b$ - we swap them if not) -

$$\text{GCD}(a, b) = \begin{cases} 3\text{GCD}\left(\frac{a}{3}, \frac{b}{3}\right) & \text{if } a \equiv b \equiv 0 \mod 3 \\ \text{GCD}\left(\frac{a}{3}, b\right) & \text{if } a \equiv 0 \mod 3, b \not\equiv 0 \mod 3 \\ \text{GCD}\left(\frac{b}{3}, a\right) & \text{if } b \equiv 0 \mod 3, a \not\equiv 0 \mod 3 \\ \text{GCD}\left(\frac{a-2b}{3}, b\right) & \text{if } a > 2b, a, b \not\equiv 0 \mod 3 \\ \text{GCD}\left(\frac{2b-a}{3}, a - b\right) & \text{if } a < 2b, a, b \not\equiv 0 \mod 3 \end{cases}$$

To terminate, we return a if $a == b$ or $b = 0$.

Still assuming that $a > b$, when we transform the original pair (a, b) to (a', b'), we notice that $\max(a', b') \leq \max(a/3, b)$ in each case of our algorithm. This means that after two transformations, $\max(a'', b'') \leq \max(a/3, b/3)$. This means that our maximum number goes down by a factor of 3 in $O(1)$ steps.

We know that we stop when before the maximum number becomes 0. Thus if x is the current maximum number in our iteration of GCD, our running time is $T(x) = T(x/3) + O(1) = O(\log x)$. This proves that $T(\text{GCD}(a, b)) = O(\log(\max(a, b)))$

Problem 7-4. Randomized Factoring [50 points] In this problem, your task will be to factor numbers of the form $N = pq$ where p and q are prime. This is generally considered to be a hard problem, so we will give you a magic box to assist you in the task.

(a) (15 parts) Let p be prime. Show that given integers a and b such that $(b - a^2) \not\equiv 0 \mod p$ the equation $x^2 + 2ax + b \equiv 0 \mod p$ has either exactly 0 or 2 solutions for $0 \leq x < p$.

Solution:

Assume z is a solution, then $z^2 + 2az + b = (z + a)^2 + (b - a^2) \equiv 0 \mod p$. This shows us how we could find a second solution : let $z' = -z - 2a$. This means

$$\begin{aligned} z'^2 + 2az' + b &= z^2 + 4az + 4a^2 - 2a(z + 2a) + b \\ &= z^2 + 2az + b \\ &\equiv 0 \mod p \end{aligned}$$

Assume $-z - 2a \equiv z \mod p$. Then, $2(z + a) \equiv 0 \mod p$. But this means (substituting $z = -a$ in the original equation) that $b - a^2 \equiv 0 \mod p$ which we know is false. Thus, if we have one solution, we have at least two distinct solutions. We will now show that there cannot be more.

For the sake of contradiction, consider a y such that $y \not\equiv z, (-z - a) \mod p$ but y is a solution to our equation. But this means that

$$\begin{aligned} y^2 + 2ay + b &\equiv z^2 + 2az + b \mod p \\ y^2 - z^2 + 2ay - 2az &\equiv 0 \mod p \\ (y - z)(y + z) + 2a(y - z) &\equiv 0 \mod p \\ (y - z)(y + z + 2a) &\equiv 0 \mod p \end{aligned}$$

As p is prime, either $p \mid y - z$ or $p \mid y + z + 2a$. In other words:

$y - z \equiv 0 \mod p$ or $y + z + 2a \equiv 0 \mod p$

Thus, $y \equiv z \mod p$ or $y \equiv -z - 2a \mod p$. This contradicts our assumption about y and so there can be at most 2 solutions.

(b) (35 parts) In fact, we can generalize the previous property to composite numbers as well. If $N = pq$, where p and q are prime, then the equation $x^2 + 2ax + b \equiv 0 \mod N$ has either 0 or 4 solutions if $(b - a^2) \not\equiv 0 \mod p$.

Assume that you are given a magic-box $M(b, N)$ algorithm, which is a deterministic polynomial time algorithm that outputs one of the solutions of $x^2 + 2ax + b \equiv 0 \mod N$, given that such a solution exists. It is illegal to call M with an b such that such a solution doesn't exist. Note that the value a is a property of our magic box, and cannot be changed.

Design a Las Vegas algorithm that computes the factorization of N when N is a product of two primes, which can make calls to the magic box algorithm M. How many calls does your algorithm make to M in expectation?

[Hint: Use the property that $x^2 + 2ax + b \equiv 0 \mod N$ has four solutions, if it has at least one]

Solution:

The factoring algorithm is as follows:

$Factor(N = pq)$:

1 choose random r such that $\gcd(r, N) = 1$

2 $b = -r^2 - 2ar$

3 $x = M(b, N)$

4 **if** $x \equiv r \mod N$ **OR** $x \equiv -r - 2a \mod N$

5 **go back** to line 1

6 **else**

7 $p = gcd(x - r, N)$

8 $q = N/p$

9 **return**(p, q)

The goal of Factor is to find an x such that $x \not\equiv r$ and $x \not\equiv (-r - 2a)$ but $x^2 + 2ax \equiv r^2 + 2ar \mod N$. If this is true then it means that $x^2 - r^2 + 2ax - 2ar \equiv (x - r)(x + r + 2a) \equiv 0 \mod N$. However, $x \not\equiv r \mod N$ and $x \not\equiv -r - 2a \mod N$. This means that together, $x - r$ times $x + r + 2a$ are divisible by $N = pq$ but not individually. This means that $q \mid x - r$ or $p \mid x - r$ but not both. Therefore, computing $gcd(x - r, N)$ yields either p or q, thus N is factored.

Runtime Analysis:

We will go back to step 1 with probability $1/2$. This is because there are four solutions to the equation and therefore, the probability that we get one of the two solutions we already knew is $1/2$. Therefore, we expect to do 2 runs before we succeed, leading to a $O(1)$ expected runtime.