Problem Set 4

Name: Gabriel Chiong
Collaborators: None

Problem 4-1.

(a) Nodes 16 and 37 are non-height balanced nodes, with skews 2 and -3 respectively.

(b) In the diagrams below, only relevant nodes are included.

`T.insert(2)`
———47
—–16
—3
2

`T.delete(49)`
47
—–84
–64

`T.delete(35)`
———47
16
——-37
—28

`T.insert(85)`
47
—84
———86
——85—-88

`T.delete(84)`
47
—–85
–64—-86
————88

(c) For node 16, rotating right or left does not result in a height-balanced tree. For node 37, rotating left is not possible, while rotating right does result in a height-balanced tree.

Problem 4-2.

(a) Min-heap.

(b) Max-heap.

(c) Neither. To convert it into a min-heap, first swap the leaf node 0 and node 9. Then swap the new node 0 and the root, node 2. Lastly, swap the leaf node 2 and node 13. The result is a min-heap.

(d) Min-heap.

Problem 4-3.

(a) We use a max-heap data structure, keyed on the garden scores s_i. This can be done in $O(|A|)$ time. Each node will also store its corresponding identifier, r_i. Simply use the `delete_max()` operation k times to get the k highest scores, and return their registration numbers. Each `delete_max()` operation takes $O(\log |A|)$ time to maintain the max-heap property of the tree. Therefore, a worst-case time of $O(|A| + k \log |A|)$ time can be derived from a sum of all operations in this algorithm. The algorithm is correct since a max-heap will remove some maximum element from the heap at every step.

(b) Repeated `delete_max()` calls would result in a $O(n_x \log |A|)$ algorithm, which is more than what is specified for this problem. However, by the max-heap property, we can traverse through the tree structure, touching each node only once, and returning once the score at a node is less than or equal to x. Visiting each node at most once results in an algorithm with a worst-case run time of $O(n_x)$.

Recursively searching the left and right children of a node results in two cases. Firstly, if `node.s` is less than or equal to x, we can return an empty set - the subtree rooted at this node will be less than or equal to x as well by the max-heap property. Otherwise, if `node.s` is greater than x, recursively search its left and right children, returning a union of the sets of registrations returned in the recursive calls, and the node's s value itself. This algorithm is correct by induction.

This algorithm visits at most $3n_x$ nodes (counting for the base case children visited), therefore the worst-case time complexity is $O(n_x)$ as desired.

Problem 4-4. Supporting the specified operations of this database requires the use of three types of data structures. Firstly, a max-heap P, where each node contains a farm's address s_i, and available capacity c_i. Secondly, a set data structure B, which maps the building address b_j to a farm address that it is connected to s_i, and its demand d_j. Thirdly, we require a set data structure F, that maps each solar farm address s_i to its own set data structure B_i containing the buildings associated with that farm, and a pointer to the location of s_i in P.

To support the `initialize(S)` operation, we first build P and then S. This order is required to support linking the s_i pointers to P in F. Every other data structure is empty. If we build P and S in $O(n)$ time, this operation will be $O(n)$ since there are at most $O(n)$ empty data structures.

To support the `power_on(b, d)` operation, simply call `delete_max()` on P and check if the condition $d_j > c_i$ holds. If true, reinsert the farm back into P (re-linking the pointers from F), and return that no farm is available with sufficient capacity to meet the requested demand. Otherwise, subtract d_j from c_i and reinsert back into P (also required re-linking pointers). Add b_j to B, mapping to s_i in F, then find B_i in F associated with s_i and add b_j to B_i. This operation maintains the database invariant and the time complexity is $O(T_{\text{delete max in P}} + T_{\text{insert in P}} + T_{\text{find in F}} + T_{\text{insert in B}} + O(1)$ to perform pointer re-linking).

To support the `power_off(b)` operation, lookup s_i and d_j in B using b_j, lookup B_i in F using s_i, and remove b_j from B_i. Lastly, go to s_i's location in P and remove s_i from P, increase c_i by d_j, and reinsert back into P. This takes time $O(T_{\text{lookup in B}} + T_{\text{lookup in F}} + T_{\text{delete in } B_i} + T_{\text{remove in P}} + T_{\text{insert into P}} + O(1)$ additional constant work).

Both B and F need to be built in $O(n)$ time and have $O(\log n)$ lookup, so we choose to use hash tables. This means our running times are expected bounds for their data structure operations, and amortized bounds on `power_on(b, d)` and `power_off(b)`. For each B_i, we need $O(\log n)$ lookup, insert, delete, so we can use a set AVL tree.

P requires $O(n)$ build and $O(\log n)$ insert, delete, and delete max operations, so we can use a max-heap. Although it should be noted that removing and item by index is not supported in a binary heap, but we can simply swap the item with the last leaf, remove it, and maintain the max-heap property by swapping in $O(\log n)$ time.

Problem 4-5. Since there is a requirement to build the data structure in $O(n)$ time, we choose to store the matrices in a sequence AVL tree, T. The sequence in T is determined based on the order of the matrices within the array $\mathcal{M}$. Each node `n` in T stores the matrix `n.M`, in addition to an augmentation `n.prod`, which contains the result of applying `matrix_multiply()` to its left child, `n.left.prod` and right child, `n.right.prod`. Since this augmentation can be computed in $O(1)$ time, this augmentation can be maintained.

To implement the `initialize()` operation, we can build T in $O(|\mathcal{M}|) = O(n)$ time in the worst-case. Since the augmentation containing the product of a node's left and right child can be computed in $O(1)$ time, the augmentation can be maintained within the $O(n)$ time bound.

To implement the `update_joint()` operation, locate the target node at k in T in at most $O(\log n)$ time. Replace the M_k stored at node k with the new value M in $O(1)$ time. Therefore, this operation runs in worst-case $O(\log n)$ time.

To implement the `full_transformation()` operation, simply return `T.root.prod` from the root node of T in $O(1)$ worst-case time.

Problem 4-6.

(a) We proceed using a proof by construction. We choose $x_i \leq x'$ and $y_j \leq y'$ such that x_i, y_i are maximized. Since we know that the slice (x', y') results in $t \neq 0$, there must be at least one topping and (x_i, y_i) must also exist. Since slice (x_i, y_i) contain the same toppings as slice (x', y'), they must have equal tastiness.

(b) We can augment a Set AVL Tree with the following three values to support the required specifications.

Firstly, store `node.sum` as the sum of all values stored in `node`'s subtrees. This is a $O(1)$ operation since all we need to do is compute the total of `node.left.sum` + `node.item.val` + `node.right.sum`.

Secondly, store `node.max_prefix`, which contains the maximum value of prefixes in all of `node`'s subtrees. Think of the in-order traversal array representation of the AVL Tree. This can be computed in $O(1)$ time by taking the maximum value from `node.left.max_prefix`, `node.left.sum` + `node.item.val`, and `node.left.sum` + `node.item.val` + `node.right.max_prefix`. Where `node` is a leaf node, the corresponding child values will be 0.

Thirdly, store `node.max_prefix_key`, which can be obtained by returning the key corresponding to the value chosen for `node.max_prefix` in $O(1)$ time using a maximum of 3 comparisons.

Since the three augmentations can all be maintained in $O(1)$ time, they do not affect the running times of the Set AVL Tree operations.

(c) First, sort the toppings by their x-coordinate in $O(n \log n)$ time using any optimal comparison-based sorting algorithm, and initialize an empty Set AVL Tree as described in PART (B). Let us call this data structure T.

Insert each topping into T using the y-coordinate as the key and t as the value in $O(\log n)$ time. Compute the augmentations, in particular `node.max_prefix`, (y^*, t^*) in $O(1)$ time along the way. The maximum prefix is by definition, the maximum tastiness of any slice with a coordinate (x_i, y^*).

By repeating this procedure at every topping sorted by x, the maximum tastiness of every slice at x_i can be computed in $O(n \log n)$ time. Looping through the toppings and returning the maximum tastiness in $O(n)$ time correctly returns the tastiest slice possible.

(d) Submit your implementation to `alg.mit.edu`.