Recitation 11

Warm-up excercises

(a) Given array A of n integers, the Python function below appends all integers from set $\{A[x] \mid 0 \leq i \leq x < j \leq n \text{ and } A[x] < k\}$ to the end of dynamic array B.

```
def filter_below(A, k, i, j, B):
    if (j - i) > 1:
        c = (i + j) // 2
        filter_below(A, k, i, c, B)
        filter_below(A, k, c, j, B)
    elif (j - i) == 1 and A[i] < k:
        B.append(A[i])
```

Argue the **worst-case** running time of `filter_below(A, k, 0, len(A), [])` in terms of $n =$ `len(A)`. You may assume that n is a power of two.

Solution: This function has recurrence of form $T(n) = 2\,T(n/2) + f$, where f is constant amortized. The number of base cases $n^{\log_2 2}$ dominates, so since a linear number of constant amortized operations are performed, this function runs in worst-case $O(n)$ time.

(b) Let T be a binary search tree storing n integer keys in which the key k appears $m > 1$ times. Let p be the lowest common ancestor of all nodes in T which contain key k. Prove that p also contains key k.

Solution: Suppose for contradiction p contains some key $k^* \neq k$. Since p is the lowest common ancestor of at least two nodes containing key k, one such node exists in p's left subtree and one such node exists in p's right subtree. But then by the BST property, $k \leq k^* \leq k$, a contradiction.

Exercise: Sortid Casino

Jane Stock is secret agent 006. She is searching for criminal mastermind Dr. Yes who is known to frequent a fancy casino. Help Jane in each of the following scenarios. (In each case for this part, you may give a worst-case or average-case efficiency, but note which one you are giving.) Note that each scenario can be **solved independently**.

(a) A dealer in the casino has a deck of cards that is missing 3 cards. He will help Jane find Dr. Yes if she helps him determine which cards are missing from his deck. A full deck of cards contains kn cards, where each card has a value (an integer $i \in \{1, \ldots, n\}$) and a suit (one of k known English words), and no two cards have both the same value and the same suit. Describe

an efficient[1] algorithm to determine the value and suit of each of the 3 cards missing from the deck.

Solution: Construct an initially empty hash table array of size k in constant time. For each card in the deck, check whether its suit is in the hash table. If not, add the suit to the hash table mapping to an empty dynamic array. Then in either case, append the card's number to the end of the suit's dynamic array. At the end of this process, each suit array contains the card numbers of that suit from the deck. Processing each card in this way takes amortized average $O(1)$ time, so doing this with all $nk - 3$ cards takes $O(nk)$ time. After inserting all cards, check the length of each suit array in $O(k)$ time. Any suit array with length n has all its cards. For any suit array whose length is less than n, sort the numbers in the array using counting sort in worst-case $O(n)$ time since the numbers are positive and bounded by n. Then loop through the numbers in order to find any that are missing. At most three suit arrays have length less than n. In total, this algorithm runs in average-case $O(nk)$.

Alternate solution: Instead of mapping each spot in the hash table to a dynamic array, map it to a DAA of length n. Once you've inserted all $nk - 3$ cards, loop over all stored DAAs in $O(nk)$ time and any spot that is missing a card corresponds to one of the missing cards.

(b) After determining the locations of the p players with the most chips, Jane observes the game play of each of them. She watches each player play exactly $h < p$ game rounds. In any game round, a player will either win or lose chips. A player's **win ratio** is one plus the number of wins divided by one plus the number of losses during the h observed hands. Given the number of observed wins and losses from each of the p players, describe an efficient algorithm to sort the players by win ratio.

Solution: Observe that since wins plus losses equals h, one win ratio $(w_1 + 1)/(\ell_1 + 1)$ is larger than another win ratio $(w_2 + 1)/(\ell_2 + 1)$ if and only if $w_1 > w_2$, so it suffices to sort the players based on their wins. Since wins are positive and bounded by $h < p$, we can use counting sort to sort the players in worst-case $O(p)$ time.

Exercise: Range Pair

Given array $A = [a_0, a_1, \ldots, a_{n-1}]$ containing n **distinct** integers, and a pair of integers (b_1, b_2) with $b_1 \leq b_2$, a **range pair** is a pair of indices (i, j) with $i \neq j$ such that the sum $a_i + a_j$ is within range, i.e., $b_1 \leq a_i + a_j \leq b_2$. Note that parts (a) and (b) can be **solved independently**.

(a) Assuming $b_2 - b_1 < 6006$, describe an $O(n)$-time algorithm to return a range pair of A with respect to range (b_1, b_2) if one exists. State whether your algorithm's running time is average, worst-case, and/or amortized.

Solution: Let $c = b_2 - b_1$, a constant. Then, for each $k \in \{1, \ldots, c\}$, we can find whether any pair (i, j) satisfies $a_i + a_j = b_1 + k$. For each $k \in \{1, \ldots, c\}$, build a hash table H mapping

[1] By "efficient", we mean that faster correct algorithms will receive more points than slower ones.

each integer a_i to i in average $O(n)$ time. Then for each a_i, check whether $b_1 + k - a_i$ is in the hash table. If it is, you have found a range pair, so return (i, j). Otherwise, there is no a_j for which $a_i + a_j = b_1 + k$, so we proceed to check a_{i+1} until $i = n$. Each of the n checks takes average constant time, so checking for range pairs for k runs in average $O(n)$ time. We repeat this procedure for all $k \in \{1, \ldots, c\}$, returning no range pair exists if the process terminates without finding a range pair. Since c is constant, this algorithm runs in expected $O(cn) = O(n)$ time.

(b) Assuming $\max A - \min A < n^{6006}$ (with no restriction on b_1 or b_2), describe an $O(n)$-time algorithm to return a range pair of A with respect to range (b_1, b_2) if one exists. State whether your algorithm's running time is average, worst-case, and/or amortized.

Solution: Observe that $n^k > \max A - \min A$ for a constant k. Loop through A to find $\min A$ in worst-case $O(n)$ time and subtract $\min A$ from each a_i. Now A contains integers in the range $\{0, \ldots, n^{k+1}\}$, and we can use radix sort to sort them in worst-case $O(n)$ time. Lastly, add $\min A$ to each value, also in worst-case $O(n)$ time, yielding array A', containing the elements of A in sorted order. Now we try to find a range pair. Initialize pointers $i = 0$ and $j = n - 1$. Repeat the following procedure, maintaining the invariant that at the start of each loop, we've either returned a range pair or confirmed that a_x cannot exist in a range pair for any $x < i$ and $x > j$, which is vacuously true at the start. To process loop (i, j), if $b_1 \leq a_i + a_j \leq b_2$, then return (i, j) as a range pair. If $i = j$, then by the invariant, there is no range pair, so return that none exists. Otherwise:

- If $a_i + a_j < b_1$, then a_i cannot be a part of a range pair with any a_x for $x \leq j$, so increase i by one, maintaining the invariant.
- If $a_i + a_j > b_2$, then a_j cannot be a part of a range pair with any a_x for $x \geq i$, so decrease j by one, maintaining the invariant.

Each loop takes worst-case $O(1)$ time to execute, and with each loop, either i increase or j decreases, so $j - i$ decreases from $n - 1$ to $j - i = 0$, at which point the algorithm terminates. Thus this algorithm runs in worst-case $O(n)$ time.

Exercise: Left Smaller Count

Given array $A = [a_0, a_1, \ldots, a_{n-1}]$ containing n **distinct** integers, the **left smaller count array** of A is an array $S = [s_0, s_1, \ldots, s_{n-1}]$ where s_i is the number of integers in A to the left of index i with value less than a_i, specifically:

$$s_i = |\{j \mid 0 \leq j < i \text{ and } a_j < a_i\}|.$$

For example, the left smaller count array of $A = [10, 5, 12, 1, 11]$ is $S = [0, 0, 2, 0, 3]$. Describe an $O(n \log n)$-time algorithm to compute the left smaller count array of an array of n distinct integers. State whether your algorithm's running time is worst-case, amortized, and/or average-case.

Solution: We compute values s_i increasing from $i = 0$ to $i = n - 1$ by maintaining at all times an AVL tree T_i on integer keys $a_0, \ldots, a_{i-1}$, where each AVL node p is augmented with the number of nodes $p.s$ in the subtree rooted at p. This augmentation can be maintained in constant time at a node p by adding 1 to the augmented subtree sizes of p's left and right children; thus this augmentation can be maintained without effecting the asymptotic running time of standard AVL tree operations.
To compute s_i, perform a one-sided range query on T. Specifically, let $\text{count}(p, k)$ be the number of nodes in the subtree rooted at p having key strictly less than k (or zero if p is not a node):

$$\text{count}(p, k) = \begin{cases} 1 + p.left.s + \text{count}(p.right, k) & \text{if } p.k < k \\ \text{count}(p.left, k) & \text{if } p.k \geq k \end{cases}.$$

Then s_i is simply $\text{count}(p, a_i)$ where p is the root of T_i, which can be computed using at most $O(\log n)$ recursive calls. So computing s_i takes worst-case $O(\log n)$ time. We then maintain the invariant by inserting a_i into T_i to form T_{i+1}. Repeating this procedure n times then computes all $s_i \in S$ in worst-case $O(n \log n)$ time.