Problem Set 3 Solution

All parts are due Tuesday, November 1, 2016 at 11:59PM.

Name: Jisoo Min
Collaborators:

Part A

Problem 3-1. [20 points] The Future Campus of MIT

(a) [10 points] Solution:
Summary: In \( O(nd) \), we need to sort a size-\( n \) unsorted list of \( d \) digit building numbers.
Algorithm: Use radix sort. Start from the last digit and sort the list using stable counting sort, then by the next-last digit, and so on until we sort by the first digit.
Correctness: With stable counting sort, we do not mess up what we have already sorted. Repeating this stable sort for all digits will produce a completely sorted list.
Run Time: \( O(nd) \) is achieved with \( d \) iterations of \( O(n) \) stable counting sort.

![Figure 1: Example with \( d = 5, n = 10 \)](image)

(b) [10 points] Solution:
Summary: In \( O(d \log n) \), we need to extract existent building numbers for the specified \( i \)-artery from the sorted directory generated in part (a).
Algorithm: On the sorted directory, binary search, in increasing order, each of ten (or nine in some cases) numbers that can potentially be in the specified artery. Note that we are not looking to see if there is an appropriate place for our option to be inserted. We are looking for a match. If we find a match, we extract it and append it to our output array. But if we are down to looking at a single element and the number does
not match, the number we are looking for does not exist in the directory. Then we move on to finding the next number.

Correctness: Because the list of \( n \) building numbers is already sorted, we are able to binary search. The building numbers are all distinct, so there will be exactly one match for each potential match.

Run Time: With \( d - 1 \) coordinate values provided, \( \forall \), (size of any \( i \)-artery) \( \leq 10 \). So \( O(d \log n) \) is achieved with at most 10 iterations of \( O(d \log n) \) binary search where each check or comparison for a match between two \( d \) digit numbers costs \( O(d) \).

![Figure 2](image_url)

**Figure 2:** Example with \( d = 3, i = 2, n_0 = 4, n_1 = 7 \)

**Problem 3-2.** [25 points] **Finding a Place in the Search Engine Space!**

**(a) [10 points] Solution:**

Summary: In \( O(\log^2 n) \), we need to find the \( k \)th largest from two size-\( \frac{n}{2} \) sorted lists.

Algorithm: We only consider \( k \) largest elements from each list, or total of \( 2k \) elements, instead of \( n \). Starting with two lists with \( k \) elements, we use recursion. Look at
the middle element of each list and compare. Then reduce down the problem size to looking at two new lists: (1) the lower part of the list with the greater middle element and (2) the upper part of the list with the lesser middle element. (*middle index is computed by taking the floor of half the list size.) Repeat reducing down the problem size until we are down to considering length 1 lists. The larger number among two numbers is the \( k \)th largest from \( n \) numbers.

**Correctness:** Only considering \( 2k \) elements out of \( n \) elements is valid because \( k \)th largest element is guaranteed to be in one the top \( k \) elements of one of the two lists. All \( n \) elements are distinct, so the middle elements of the lists are never equal.

**Run Time:** We have a recurrence of \( T(k) = T(k/2) + O(1) \). By Master Theorem, \( T(k) = \Theta(\log k) \). We achieve \( O(\log n) \) run time with \( k = O(n) \).

---

**Figure 3:** Example with \( n = 20, k = 4 \)

**Solution:**

Regular AVL-INSERT algorithm inserts a new node at the appropriate location of the tree and walks up the tree from the node, restoring AVL property and updating heights. But we need to track the number of nodes in the subtree rooted at \( x \), at each node \( x \).
So we make the following modifications to the AVL-INSERT algorithm:

- Set the count of subtree nodes equal to one for the inserted node.
- While walking up the tree and updating the height, update the count of subtree nodes by incrementing one. When AVL property is restored with rotations, update the count of the subtree nodes accordingly.

Note that we only add constant runtime operations; we do not look at extra nodes in addition to the nodes we look at during the regular AVL-INSERT algorithm. Therefore the run time stays the same as the regular AVL-INSERT at $O(\log n)$. Updating counts during rotation may seem less obvious to be an addition of constant operation. If we left rotate a node, we update the counts for two nodes (Figure 4). Similarly, right rotation results in two updates.

![Figure 4](image_url)

**Figure 4:** Diagram of updating counts during left rotation

(c) [10 points] **Solution:**

Summary: Given two size-augmented AVL trees of size $n/2$, we need to find the $k$th-largest data point across them in $O(\log^2 n)$.

Algorithm: Say the two AVL trees are $T_1$ and $T_2$. We set $T_1.root$ as current and $i$ as the number of nodes in $T_1$ with values greater than current.value. We go left or right on $T_1$ depending on $i + j + 1$ and $k$, where $j$ is the number of nodes in $T_2$ with values greater than current.value. As we go down $T_1$, we update $i$ and calculate $j$. If $i + j + 1 == k$, then $k$th-largest data point is found. If $k$th-largest data point is not found in $T_1$, complete the same steps on $T_2$. 
# find the kth largest data throughout T1 and T2
def find_kth(T1, T2, k):

    # initialize the variables
    current = T1.root
    i = T1.root.right.count

    # O(log n) iterations because of the AVL property
    while current.count != 1:  # current is not a leaf

        # number of nodes in T2 with values greater than current.value
        j = count_greater(T2, current.value)  # takes O(log n)

        # there is exactly k−1 data greater than current.value
        # found the kth−largest data
        if i+j+1 == k:
            return current.value

        # there is less than k−1 data greater than current.value
        # current was larger than kth−largest
        elif i+j+1 < k:
            current = current.left
            i += (current.right.count + 1)

        # there is more than k−1 data greater than current.value
        # current was smaller than kth−largest
        elif i+j+1 > k:
            current = current.right
            i -= (current.left.count + 1)

    # our current node is a leaf
    # found the kth−largest data
    if i+j+1 == k:
        return current.value

    # kth−largest is not found in T1; search in T2
    return find_kth(T2, T1, k)
# x is a value that does not exist in T
# assume leaf.left and leaf.right to be None and None.count to be 0

def count_greater(T, x):  # takes O(log n)

    # initialize the variables
    current = T.root
    j = T.root.right.count

    while current.count != 0:

        if current.value < x:
            current = current.right
            j -= (current.left.count + 1)

        elif current.value > x:
            current = current.left
            j += (current.right.count + 1)

    return j

Correctness: All data in two AVL trees are distinct. ∀k ≤ n, we are guaranteed to find the kth-largest data. By setting leaf.left and leaf.right to be None and None.count to be zero, we consider all edge cases.

Runtime: Walking down an AVL tree with O(log n) height until we find the kth-largest, we have O(log n) iterations. For each iteration, updating i, the number of nodes with values greater than the current node, takes constant time. Finding where current value would be inserted in the other AVL tree and calculating j, the number of nodes with values greater than the current value, takes O(log n), because we update j in constant time as we walk down the O(log n) height. Note that we do not actually insert the node into the other AVL tree. We achieve the total run time of O(log² n).

If we fail to find the kth-largest in the first tree, we have another O(log² n) operation, which does not change the overall run time.

Problem 3-3.

(a) [10 points] Solution:

Summary: Given a graph with fields V, dirt paths E, and m chickens, we need to construct a list of entries containing each chicken’s location and the shortest path length between field s and the chicken’s location in O(|V| + |E| + m).

Algorithm: Until all chickens are found, we use BFS (Breadth First Search) to find the shortest paths. Once a chicken is found, we store an entry (c, v, d) where c is the name of the chicken, v is the name of the field, and d is the shortest path length from
s to v. While visiting a field, we record an entry for all chickens in that field.
Correctness: Implementation of BFS ensures that we will find the shortest path to all
the fields that chickens are located in. We will be visiting all nodes if needed, so there
is no possibility that we will be leaving out any of our chickens from our catalog.
Run Time: BFS requires $O(|V| + |E|)$. No matter how chickens are distributed, we
have to catalog all chickens. If, for example, all chickens ran off to a single field, we
have to iterate through all chickens in $O(m)$. The final run time is $O(|V| + |E| + m)$.

(b) [5 points] Solution: No. Counterexample: $L = [(c_1, v_1, 1), (c_2, v_2, 1), (c_3, v_3, 1),
(c_4, v_4, 1), (c_5, v_5, 1), (c_6, v_7, 2), (c_7, v_7, 2), (c_8, v_7, 2), (c_9, v_7, 2), (c_{10}, v_7, 2)]$
I choose $c_1, c_2, c_3, c_4, c_5$ as my chickens to collect eggs from, because they have the
smallest distance $d$ of 1. Collecting eggs from these chickens requires traveling a
distance of 10. The shortest 5-collection-path (path length of 4), however, collects
eggs from different chickens: $c_6, c_7, c_8, c_9, c_{10}$.

Figure 5: Graph representation of the counterexample
(c) [10 points] Solution:

Summary: Given a DAG (Directed Acyclic Graph), we need to compute the sequence of fields in $O(|V| + |E|)$ so that chickens have places to run away until they reach $s$.

Algorithm: Call DFS (Depth First Search) to compute finishing times for each field. As each field is finished, insert it onto the front of a linked list. And return the final linked list of fields. In other words, we get the topological sort of the DAG. In topological sort order, we travel to fields on our helicopter.

Correctness: Chickens cannot run uphill, so they can only go from fields that appear earlier in the topologically sorted list to those that appear later. If we choose to travel with helicopters and visit the fields in topologically sorted order, we will be chasing down the chickens to field $s$.

Run Time: It takes $O(|V| + |E|)$ to obtain a topological sort using DFS.

Figure 6: Example of a topological sort with thirteen fields

(d) [1 points] Solution:

Yes. First visit field $s$ and fly over to any chicken. Then chase it. The chicken is forced to go to the unpleasant field $s$ where it detects the scent of human.
Part B

Problem 3-4.

(a) Submitted on alg.csail.mit.edu

(b) Submitted on alg.csail.mit.edu

(c) [15 points] Solution:

Summary: Given a corrupted document of length $n$, we want to search for a pattern of length $k$ in $O(2^k + n)$. The corrupted document contains '?' symbol(s) which can be thought of as any one of the alphabets to match the pattern.

Algorithm: Add '?' to our alphabet $A$ and assign a unique letter value to it. Using recursion, generate a hash table of size $m$ mapping all possible $2^k$ variations of pattern. There are $2^k$ combinations, because for each letter in the pattern, the letter itself or a '?' can be used in that location. Recursion works as follows. Set the starting value equal to zero. Keep track of two different values as we add in the letter value for the last letter. Then for each of the most recent values, add the letter value of the next-last letter taking into account the weight. And repeat for all letters. The number of values doubles everytime. Once we have completed this step for the first letter of the pattern string, hash with mod $m$, where $m = \Theta(n)$. We ideally set $m$ to be a large prime.

---

**Figure 7**: Example with pattern ‘abc’
Once we have our hash table ready, we roll through the document and check if the hash value of the current sliding window, or substring, match with any hash values we have in the table. When we roll through, use append and skip functions discussed in class. If hash values match, likely so do strings. Once the hash values match, we check the strings letter by letter for completeness. We terminate only when there is a true match or there is no match after searching the entire document.

Correctness: It is considered a match when pattern matches a string in the document either with the correct alphabets or with ‘?’s in place of the correct alphabet. We consider all possibilities of matches by computing the hash value of all $2^k$ options. Once the hash value matches, we check in $O(k)$ string by string to see if it is truly a match.

Run Time: Computing a hash value takes constant time. By using recursion, we avoid $O(k \times 2^k)$ and achieve $O(2^k)$ in building the hash table. Also $m = \Theta(n)$ helps us maintain the load factor to be $\Theta(1)$ and the search time to be $O(1)$. Until we find a first match and terminate or until we search through the end of the document for at most $n - k$ iterations, we roll forward in the document. Append and skip are constant operations. So rolling through the document is $O(n)$. Overall, we achieve $O(2^k + n)$.

Approximate conditions for better performance compared to naive approach:
When the pattern matches the length-$k$ substring almost perfectly but actually has a discrepancy towards the end of the string. We would have iterated through almost $k$ times only to find out that the strings do not match. In addition, if the situation of almost matching happens multiple times, algorithm discussed in 4(c) performs significantly better.