Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — March 1, 2016
Instructors: Nancy Lynch, Aleksander Madry, and Bruce Tidor — Problem Set 2

Problem Set 2

All parts are due March 3rd, 2016 at 11:59PM.

Name: Arturo Chavez-Gehrig

Collaborators: Sabrina Ibarra

Part A

Problem 2-1.

The general form

$T(n) = a * T(\frac{n}{b}) + f(n)$

Summary of the Master Theorem (4 cases)

Case 1: geometrically increasing: $T(n) = \Theta(L) = \Theta(n^{log_b a})$

Case 2: equal levels: $T(n) = \Theta(L * h) = \Theta(n^{log_b a} log n)$

Case 3: almost equal levels, log factors carry through: $T(n) = \Theta(L * log^{k+1} n)$

Case 4: geometrically decreasing: $T(n) = \Theta(f(n))$

(a) $T(n) = 3T(\frac{n}{3}) + 25n$

$n^{log_b a} = n^{log_3 3} = n$

same as level on each height O(n)

case 2 with equal levels takes form $\Theta(L * h) = \Theta(n^{log_b a} log n) = \Theta(n log n)$

(b) $T(n) = 3T(\frac{n}{4}) + 25n$

$log_b a = log_4 3 < 1$ geometrically decreasing on each level

case 4: $T(n) = \Theta(f(n)) = \Theta(n)$

(c) $T(n) = 4T(\frac{n}{3}) + 25n$

geometrically increasing

case 1: $\Theta(L) = \Theta(n^{log_b a} = \Theta(n^{log_3 4})$

(d) $T(n) = 9T(\frac{n}{3}) + 9n^2$

$9T(\frac{n}{3}) = n^{log_3 9} = n^2$ and $9n^2 = \Theta(n^2)$

This is equal levels case

case 2:

$\Theta(L * h) = \Theta(n^{log_b a} logn) = \Theta(n^2 logn)$

(e) $T(n) = 3T(\frac{n}{4}) + logn$

$log_4 3$ versus $logn$

This is case 2 where $T(n) = \Theta(L * log^{k+1} n)$

$T(n) = \Theta(n^{log_4 3} log^2 n)$

Problem 2-2.

(a) In augmented merge, we will be performing the traditional merge operation, but will add a variable called 'counter' which will track the number of inversions. We initialize our counter to the length of the left list and increase the counter each time a member of the right list is added. We also decrease the counter by one each time an element of the left list is added to the merged list. Note, this assumes left and right lists are already sorted. At the end, we will take the counter mod 2 and return the result as our parity fit.

AugmentedMerge(L,R):

counter= 0

i= 0

j= 0

k=len(L)

while $i < len(L)$ or $j < len(R)$

    if L[i] < R[j]:

        Find.append(L[i])

        i+ = 1

        k= $k - 1$

    else

        Final.append(R[j])

j+ = 1

count+ = k

parity = count mod 2

return final, parity

(b) The running time to execute augmented merge is bounded by the size of the two lists so within the while loop, it is an O(1) operation. Therefore, the overall running time is O(max(nRight,nLeft)).

(c) Augmented merge sort will behave much like merge sort in that it will employ a divide and conquer strategy. However, we will keep track of the parity bits through each merge step, recursively calling merge.

AugmentedMergeSort(n)

leftList = $n[1 : len/2]$

rightList = $n[len/2 : len]$

if length of leftList or rightList = 1

return final list, parity bit

sortedLeft, parityLeft = AugmentedMergeSort(leftList)

sortedRight, parityRight = AugmentedMergeSort(rightList)

final list, parityFull = AugmentedMerge(leftList,rightList)

parity = (parityLeft + parityRight + parityFull) mod 2

return final list, parity

(d) Augmented merge sort, like regular merge sort, divides the input into two and each operation level runs in linear time, as we proved in part B.

$T(n) = 2T(\frac{n}{2}) + O(n)$

Using Master Theorem, this is the same case as problem $2 - 1A$, so this function runs in O(nlogn).

Problem 2-3.

(a) To check whether it is a valid binary tree, we will take advantage of a property of all binary search trees. For all binary search trees, an inorder traversal will travel to each node in increasing order. In this case, we will perform an inorder traversal on the

arbitrary binary tree, taking O(n) time. Then, we will check if the returned order of the traversal is in increasing order, also of O(n) time.

```
def Checker(tree)
    inorder(tree.root)
    previous element = null
    for each element returned by the inorder traversal
        if prevelt < elt or prevelt == null:
            prevelt = elt
        else return false
    return true
def inorder(node)
    if node != null:
        inorder(node.left)
        print node
        inorder(node.right)
```

(b) Proof by contradiction

The definition of a binary search tree is that in each node of the tree, its left child (if that exists) is less than its value. Similarly, its right child is greater (if it exists). Because of this property, the inorder traversal returns a sorted list in increasing order. Suppose we have a binary search tree that is called false by the algorithm because it has an inorder traversal that is not in increasing order. CONTRADICTION, this violates the definition of a bst. Similarly, suppose we have a tree that is not a bst that is return true by the algorithm, but this tree's inorder traversal returns increasing list. CONTRADICTION, this tree must be a bst if it fulfills that property.

(c) The inorder traversal is a recursive function that splits the problem into two subproblems and each call runs in O(1). Inorder traversal takes O(n). Checking if a list is sorted in increasing order requires that each element is considered. The operation to determine if an element is smaller/larger than its neighbor takes O(1). Therefore, checking if the list is sorted takes O(n). Since the two components run in O(n), the algorithm as a whole runs in O(n).

(d) In order to create a balanced bst from an unbalanced one, we will take advantage of the algorithm we used in part A. We will employ inorder traversal to sort the elements of the bst in increasing order. Now, from this sorted list we can construct a balanced tree by defining a recursive algorithm where we take the middle element of the list, creating two sublists. Then taking the middle elements of each sublist and assigning these elements as the left and right children of that middle element. We can employ this recursively, transforming the array into a balanced tree from root to leaf.

def CreateBalancedBST(BST)

Sorted list = inorder(BST.Root)

tree = assign children(sorted list)

return tree

def assign children(Array)

size = len(Array)

root = Array[ceiling($n/2$)]

BST.Leftchild = Array[floor($n/4$)]

BST.Rightchild = Array[ceiling($3n/4$)]

Return BST

Part B

Problem 2-4.

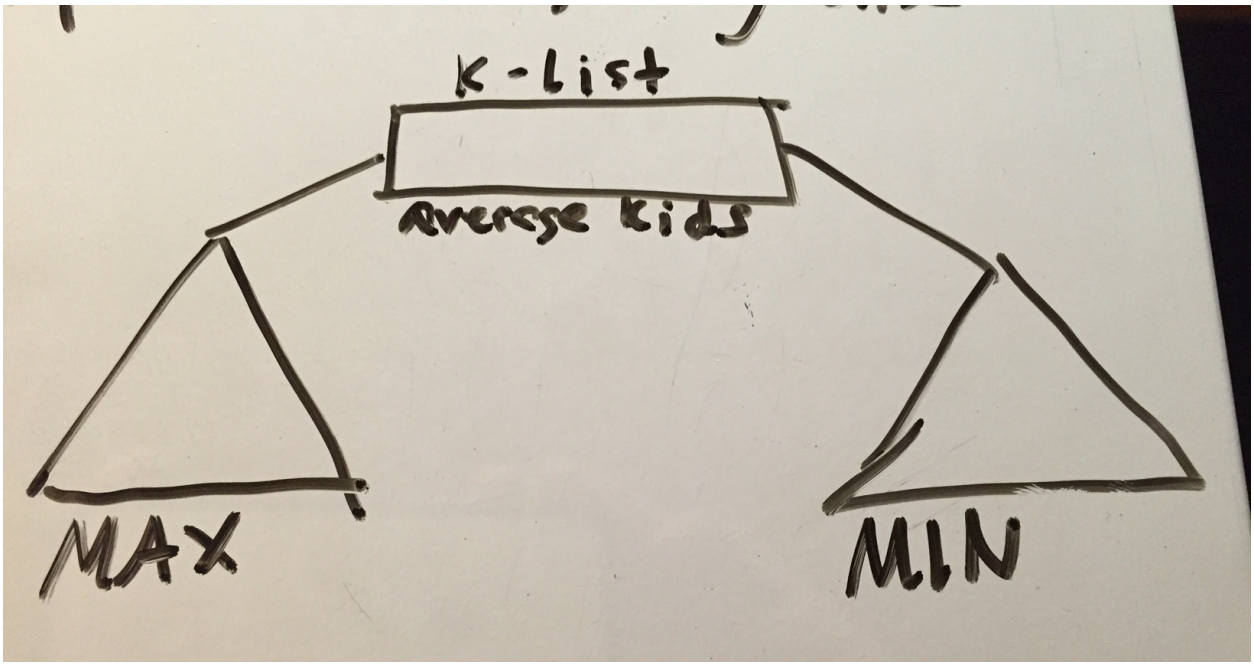
Submit your implemented python script.

(a)

(b)

(c) We could use names of students as keys and tuples(credits,GPA) as the associated value This way we can access GPA of a student through gradedict[student][1]. We can update a student's GPA by calculating $\frac{GPA*units+credits*grade}{credits+units}$ then updating our dictionary with the values gradedict[student]=($credit + units$,updated GPA).

(d) To do this, we can use two heaps and a list, along with a reference dictionary to hold the GPAs and credits for each student. We use a max heap to hold the underachievers and a min heap to hold the overachievers. The remaining k elements in between the two heaps are the most average students. The k element list is arranged in increasing order. We update grades by removing them from the heaps by updating their values to



really small or really large numbers and then extracting them from the root. Then, we recalculate the GPA and reenter it into the correct heap or into the k list. If we move it into the k list, we remove the largest/smallest element of the k list and tranfer it to one of the heaps in order to keep the structure balanced. It takes logn time to maintian the heap and k time to keep inserting elements into the k list preserving its increasing sort order. Figure on the next page.

(e)