Problem Set 2

All parts are due March 3, 2016 at 11:59PM. Please download the .zip archive for this problem set. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

Problem 2-1. [15 points] **The Master Theorem**

Use the Master Theorem to give tight asymptotic bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Justify your answers by saying which case of the Master Theorem you used.

(a) [3 points] $T(n) = 3T(n/3) + 25n$

Solution: $T(n) = \Theta(n \log n)$, by case 2 of the Master Theorem.

(b) [3 points] $T(n) = 3T(n/4) + 25n$

Solution: $T(n) = \Theta(n)$, by case 3.

(c) [3 points] $T(n) = 4T(n/3) + 25n$

Solution: $T(n) = \Theta(n^{\log_3 4})$, by case 1

(d) [3 points] $T(n) = 9T(n/3) + 9n^2$

Solution: $T(n) = \Theta(n^2 \log n)$, by case 2.

(e) [3 points] $T(n) = 3T(n/4) + \log n$

Solution: $T(n) = \Theta(n^{\log_4 3})$, by case 1.

Problem 2-2. [20 points] **Augmenting Merge Sort**

Given an array of numbers $A = [x_1, x_2, \ldots, x_n]$, one way to measure how far the array is from being sorted is by counting the number of inversions, where an inversion is any pair of elements x_i and x_j in A such that x_i comes before x_j and $x_i > x_j$.

In this problem, we want to augment the Merge Sort algorithm so it also produces a *parity bit* which is 0 if the number of inversions in the original list is even and 1 if the number of inversions is odd.

For example, if the input array is $A = [6, 23, 5]$, then the output array should be $[5, 6, 23]$ and the value of the parity bit should be 0, because there are 2 inversions. As a larger example, if the input arrays is $A = [2, 6, 3, 9, 7, 4, 10, 5]$, the output should be $[2, 3, 4, 5, 6, 7, 9, 10]$ and the parity bit should be 1, because there are 9 inversions.

For this problem, assume that all the entries in the input array are distinct integers.

(a) [7 points] We start by defining AUGMENTEDMERGE which takes sorted arrays A_1 and A_2, of lengths n_1 and n_2 respectively, and outputs sorted array B which merges the elements from A_1 and A_2. In addition, AUGMENTEDMERGE should return a parity bit b describing the parity of the number of inversions in the joined list A_1 followed by A_2.

For example, if $A_1 = [1, 4, 6]$ and $A_2 = [2, 3, 5]$, then AUGMENTEDMERGE should produce $B = [1, 2, 3, 4, 5, 6]$ with a parity bit $b = 1$, because there are 5 inversions in the list $[1, 4, 6, 2, 3, 5]$.

Describe AUGMENTEDMERGE in plain English and in pseudocode.

Solution: We start out with our parity bit $b = 0$. Now, we do the normal merge process with A_1 and A_2, except whenever we add an element from A_2, if there is an odd number of remaining elements in A_1, we flip the parity bit.

```
AUGMENTEDMERGE(A_1, A_2)
 1  B, b = Array(), 0
 2  p_1 = 0, p_2 = 0
 3  while p_1 < |A_1| or p_2 < |A_2|
 4          if A_1[p_1] < A_2[p_2] or p_2 = |A_2|
 5                  B.append(A_1[p_1])
 6                  p_1 = p_1 + 1
 7  elseif A_1[p_1] > A_2[p_2] or p_1 = |A_1|
 8          B.append(A_2[p_2])
 9          p_2 = p_2 + 1
10          b = b + |A_1| - p_1( mod 2)
11  return B, b
```

(b) [3 points] Analyze the running time of AUGMENTEDMERGE.

Solution: Just like the ordinary merge algorithm, this takes time $O(n_1 + n_2)$ because each element is added to B exactly once and as the result of $O(1)$ comparisons. The only additional step is line 8, which is an $O(1)$ arithmetic operation (under our computational model).

(c) [7 points] Using AUGMENTEDMERGE as a subroutine, describe a modified version of the Merge Sort algorithm called AUGMENTEDMERGESORT, which takes an unsorted list A of n distinct integers as its input and produces the sorted array B and parity bit b, which corresponds to the parity of the number of exchanges required to produce B from A.

Describe the AUGMENTEDMERGESORT algorithm in plain English and in pseudocode.

Solution: We run the normal merge-sort algorithm, except now on each recursive call, and each time we merge, we get the extra information from the parity bit. To calculate the final parity bit from the two recursive parity bits and merge parity bit, we simply add all the bits togther and take the result modulo 2.

AUGMENTEDMERGESORT(A)
1 **if** $|A| \leq 1$
2 **return** $(A, 0)$
3 $A_L = A[0 \ldots \frac{|A|}{2}]$
4 $A_R = A[\frac{|A|}{2} \ldots |A|)$
5 $(B_L, b_L) =$ AUGMENTEDMERGESORT(A_L)
6 $(B_R, b_R) =$ AUGMENTEDMERGESORT(A_R)
7 $(B, b) =$ AUGMENTEDMERGE(B_1, B_2)
8 $b = b + b_L + b_R \pmod 2$
9 **return** (B, b)

(d) [3 points] Analyze the running time of the AUGMENTEDMERGESORT algorithm.

Solution: The recursion here the same as Merge Sort, $T(n) = 2T(n/2) + O(n)$, so the runtime is $O(n \log n)$. The $2T(n/2)$ comes from the two recursive calls, and the $O(n)$ comes from the call to AUGMENTEDMERGE. Calculating the parity bit in the end is just another $O(1)$ operation, which doesn't change the recurrence.

Problem 2-3. [20 points] **Binary Search Trees**

In this problem, we will explore various aspects of Binary Search Trees.

(a) [7 points] First, we want to develop an algorithm to check whether an arbitrary binary tree with positive integer keys at the nodes is a binary search tree. Thus, describe in English and in pseudocode an algorithm that takes an arbitrary binary tree B with distinct positive integer keys at the nodes, and returns true if B is a binary search tree and false otherwise. Your algorithm should run in time $O(n)$.

Solution: First, we do an in-order traversal to get a list of numbers which will be sorted if and only if B is a binary search tree. Then, we simply check to see that the list is sorted.

```
INORDER(node, elements)
1  if node == NULL
2          return elements
3  INORDER(node.left, elements)
4  elements.append(node.value)
5  INORDER(node.right, elements)
```

```
CHECKBST(B)
1  elements = [ ]
2  INORDER(B.root, elements)
3  for i = 1 . . . |B|
4          if elements[i − 1] ≥ elements[i]
5                  return false
6  return true
```

Alternative solution:

We define a recursive procedure that determines whether a tree rooted at node n is a binary search tree whose values are restricted by lower and upper bounds. The base case is when n is a leaf, and we just return true when $n.value$ is within the bounds. If n is not a leaf, we still do this check, but we also recurse on the left and right subtrees. When we recurse on the left subtree, we replace the old upper bound with $n.value$ (everything in n's left subtree must be less than $n.value$, which is less than the old upper bound), and when we recurse on the right subtree, we replace the old lower bound with $n.value$ (everything in n's right subtree must be greater than $n.value$, which is greater than the old lower bound). To determine whether a tree is a BST, we simply call our recursive procedure on the root with bounds of $-\infty$ and ∞.

CHECKBST$(node, low = -\infty, high = \infty)$

```
1  if node.value < low or node.value > high return false
2  if node.left
3          if not CHECKBST(node.left, low, node.value) return false
4  if node.right
5          if not CHECKBST(node.right, node.value, high) return false
6  return true
```

(b) [3 points] Sketch a proof that your algorithm is correct. **Hint:** Show that the algorithm outputs $true$ if and only if the tree is a Binary Search Tree.

Solution:

For correctness, we must prove two directions:

- If the algorithm outputs $true$ then the tree is a BST.
- If the tree is a BST, then the algorithm outputs $true$

For the first statement, assume that the algorithm outputs $true$. Consider an arbitrary node u. In the in-order traversal, all of u's left descendants get output before u and all of u's right descendants get output after u. Because the algorithm returns $true$, the result of the in-order traversal must be sorted, which means that all of u's left descendants are strictly smaller than u and its right descendants are strictly larger. Since this is true for every node u, the BST property is satisfied.

For statement 2, assume that the tree is a BST. Consider an arbitrary node u. All of u's left descendants get output before u and all of u's right descendants are output after u. By the BST property, that means that u appears in the correct place in the list relative to all its descendants - it's greater than its left descendants, which come before it, and less than its right descendants, which come after it. Since this is true for every node, the list is sorted and the final check will succeed and return $true$.

Alternative solution:

Our algorithm takes as input a node n and numbers low and $high$ and returns true if and only if the tree rooted at n is a binary search tree where all nodes have a value between low and $high$. To prove correctness, we need to prove two directions:

- If the tree rooted at n is a BST with values between low and $high$, then the algorithm outputs $true$.
- If the algorithm outputs $true$ then the tree rooted at n is a BST with values between low and $high$.

We use induction on the height of the binary tree. If our height is 1, then we just have a root node. The tree is trivially a binary search tree, and the check in line 1 makes sure that $n.value$ is between low and $high$ if and only if we return true. Now, suppose our algorithm works for a tree of height $h-1$ and the tree rooted at n is of height h.

For the forward direction, suppose n is the root of a BST with values bounded by low and $high$. Then $low < n.value < high$, so line 1 passes. Our algorithm will recurse on the left and right subtrees. When we recurse on $n.left$ with our new value $high = n.value$, by induction our algorithm returns true - subtrees of BSTs are also BSTS and the nodes in the left subtree must have values less than $n.value$ by the BST property. The right subtree case is analagous, thus our algorithm will return true.

Now, suppose our algorithm returns true. This means that $low < n.value < high$, and by induction, that $n.left$ is a binary search tree with nodes bounded by low and $n.value$. For all x in the subtree rooted at $n.left$, we have $low < x < n.value < high$, so we are still within our original bounds of low and $high$. The case for the right is analagous, so n's left subtree is a BST with values less than $n.value$ and n's right subtree is a BST with values greater than $n.value$. Thus, n is a binary search tree, and all its values are bounded by low and $high$.

(c) [3 points] Analyze the time complexity of your algorithm.

Solution: The in-order traversal takes time $O(n)$ because each node is visited exactly once, and the work done at each node is $O(1)$ to add it to the list of elements. Checking that the resulting list of elements is sorted is also an $O(n)$ linear scan, so the overall runtime of the algorithm is $O(n) + O(n) = O(n)$.

Alternative solution: Our algorithm visits each tree node exactly once, and does $O(1)$ work at each node (checking that the value is within the bounds and reassigning the bounds on recursive calls). Thus, if there are n nodes in total, the overall runtime is $O(n)$.

(d) [7 points] Now we wish to take an arbitrary binary search tree and create a balanced binary search tree. Describe in English and in pseudocode an $O(n)$ algorithm which takes a binary search tree as input and produces a new binary search tree whose height is $O(\log n)$. Explain why your algorithm takes time $O(n)$.

Solution: The algorithm is in two parts. First, we acquire a sorted list of the elements by doing an in-order traversal, as in part (a). Then, we create a balanced binary search tree by picking the middle element of the list as the value for the root node, and then constructing the left and right subtrees by recursing on the left and right half of the list, respectively.

MAKEBSTFROMLIST($elements$)

1 **if** $elements$ is empty **return** $NULL$
2 $mid = |elements|/2$
3 $r = Node()$
4 $r.value = elements[mid]$
5 $r.left =$ MAKEBSTFROMLIST($elements[0..mid]$)
6 $r.right =$ MAKEBSTFROMLIST($elements[mid + 1..n]$)
7 **return** r

MAKEBST(B)

1 $elements = [\,]$
2 INORDER($B.root, elements$)
3 **return** MAKEBSTFROMLIST($elements$)

The proposed algorithm has a time complexity of $O(n)$ for the inorder traversal and $O(n)$ for the construction of the balanced BST- this follows from the recursive call $T(n) = 2T(n/2) + 1$.

Part B

Problem 2-4. [45 points] **Average Students**

A large private university wants to keep track of which students are "most typical," in the sense that they have the "most average" GPAs. (The university will ask these students to represent it in press interviews or reality TV shows). Accounting starts anew each academic year, on September 1, and finishes on August 31. Students take courses with variable numbers of credits (small positive integers), completing them at arbitrary times during the year, not just at the ends of normal semesters. There is no limit to how many courses, or credits, a student is allowed to take in one year. Grades are numbers in the range 0 to 5. A student's GPA can be calculated at any point during the year as $\frac{\Sigma_i c_i g_i}{\Sigma_i c_i}$, or 0 if no courses were taken. Here, each i represents a course, with c_i indicating its number of credits and g_i indicating its grade.

The university has hired you to develop a data structure and algorithms to keep track of the students with the k middle GPAs, for some fairly small number k that they will provide each year as a parameter. If n is the number of students, then the number of students higher than the "middle" ones should be $\lceil \frac{n-k}{2} \rceil$. You may assume you have a complete list of students available at the beginning of each year, and that the names of students are unique. Your program should process grades one at a time.

Design a structure to keep track of the k middle students for the current year so far. The data structure should take in a list of the students' names and the number k as parameters.

Your solution should use *heaps*. Your data structure must support two operations:

- UPDATE_GRADE$(self, student, credit, grade)$: Add grade information for a course completed by the student into the data structure. This should run in time $O(\log n + k)$.
- MIDDLE$(self)$: Return the middle k students with their GPAs for the year so far, in order from lowest to highest. The result should be a list of (student, grade) pairs. This should run in time $O(k)$.

(a) [10 points] In the file all_code.py, you will find a class called Max_Heap, which contains most of the routines needed for a max-heap data structure. Write one additional routine for this data structure, called MAX_HEAP_MODIFY$(self, key, data)$. It replaces the old data for the key with the new data and restores the heap invariant. See the code for a more thorough description. Your routine should work in time $O(\log m)$, where m is the number of elements in the heap.

If you find it helpful, you may call the method SHOW_TREE() to print out the heap as a tree.

(b) [5 points] Also in all_code.py, you will find a class called Min_Heap. Write the routine MIN_HEAP_MODIFY$(self, key, data)$; this should be similar to part (a).

(c) [5 points] Describe in words how one would use a Python dictionary to store all the names of all the students, plus, for each student, information about his/her credits and grades. Your data structure should support the operations:

- UPDATE_GRADE$(self, s, c, g)$ to enter a new grade for student s, and
- AVERAGE$(self, s)$ to retrieve the current average for student s.

You need not actually implement this part in Python. (An explanation of how to compute such a "running average" can be found in all_code.py, in the description of the class Gradebook.)

Solution: In order to keep a running average for each student, use a dictionary that maps student names ("keys") to an array containing information about their grade ("values"). The array should have two elements: the total number of credits the student has taken, and the credit-weighted sum of the student's grade.
So, each entry of the dictionary looks like:

$$\{student : [total\_credits, credit\_weighted\_grade]\}$$

If c_i and g_i represent the number of credits and the grade received in class i, then $total\_credits = \sum_i c_i$ and $credit\_weighted\_grade = \sum_i c_i g_i$.

Say our dictionary data structure is called "gradebook." Then, the two operations would be carried out in the following way:

- AVERAGE: return the credit-weighted sum of grades over the total number of credits

  AVERAGE$(self, student)$
  1 $total\_credits = self.gradebook[student][0]$
  2 $credit\_weighted\_grade = self.gradebook[student][1]$
  3 return $\frac{credit\_weighted\_grade}{total\_credits}$

- UPDATE_GRADE: update the total number of credits and the credit-weighted sum of grades with the new credit grade information.

  UPDATE_GRADE$(self, student, credit, grade)$
  1 $self.gradebook[student][0] = self.gradebook[student][0] + credit$
  2 $self.gradebook[student][1] = self.gradebook[student][1] + credit * grade$

(d) [5 points] Describe in words a data structure based on heaps, a dictionary, and possibly other data structures, that keeps track of the k middle students. Your data structure should support the two operations listed at the start of this problem, with the indicated time bounds, that is, UPDATE_GRADE$(self, student, credit, grade)$ running in time $O(\log n + k)$, and MIDDLE$(self)$ running in time $O(k)$. Assume that you are provided with a list of student names, which you should load into your data structure initially.

Solution: We implement a data structure that uses a min-heap to store the top $\lceil \frac{n-k}{2} \rceil$ students, a list to store the middle k students, and a max-heap to store the bottom

$\lfloor \frac{n-k}{2} \rfloor$ students. In addition, we keep a dictionary that maps students names to the number of credits they have taken, the weighted sum of their grade so far ($\sum_i c_i g_i$), and their position in relation to the other students (top, middle, or bottom). From now on, we will refer to the heap that stores the top of the class as "top_ heap", the lists that stores the middle of the class as "middle_ list", and "bottom_ heap" for the remaining students.

Implementing UPDATE_ GRADE$(self, student, credit, grade)$:

The difficulty occurs when we want to update a student's grade, since this may affect where the student ends up (in the top heap, middle list, or bottom heap). We go through the cases below.

Case 1: The student was originally in the top heap. We update the top heap so that heap properties are maintained using min_ heap_ modify, and then, we compare the minimum value of the heap to the middle list and the bottom heap

If the minimum value is greater than or equal to the max of the middle list, no swaps are made.

If the minimum value is strictly less than the max of the middle list and greater than or equal to the max of the bottom heap, we insert the minimum value into the middle list in sorted order, extract the max of the middle list, and insert it into the top heap.

If the minimum value is strictly less than the max of the bottom heap, we insert the value into the bottom heap. Then, we extract_ max from the bottom heap, and insert it into the middle list. Finally, we take the max of the middle list, and insert it into the top heap.

Case 2: The student was originally in the middle list. We re-sort the middle list, and then we compare the min and max values.

If the max value is larger than the min value of the top heap, we extract the max value from the middle list and insert it into the top heap. Then, we extract the min value from the top heap, and insert it into the middle list.

If the min value of the middle list is smaller than the max value of the bottom heap, we swap those two elements in a similar fashion.

If neither of the two cases above hold, no swaps are made.

Case 3: The student was originally in the bottom heap. The possibilities in this case are symmetrical to the cases discussed in case 1.

Using this data structure, we always keep track of the k middle students in a list, so calling **MIDDLE(self)** to return them would be $O(k)$.

Finally, we maintain dictionaries that keep track of each student's grades and their whereabouts in the structure. The overall dictionary takes the students as keys, and keeps track of the number of credits the student has taken, the product-sum $\sum_i c_i * g_i$, and the position of the student (top heap, middle list, or bottom heap). With this structure, we can easily calculate the student's average GPA (as described in part c), and we can easily update the student's credit and grade information, in a way that wouldn't

be possible if we only stored the student's GPA. Within the structures comprising the overall data structure, there are dictionaries that keep track of the indices of the students. For instance, in the top heap, there is a dictionary that maps students to their index in the list representation of the heap. This ensures that all information pertaining to a student is easily and rapidly accessible.

(e) [20 points] Implement your algorithm from part (d) in Python in all_code.py under the class called "Gradebook". Write the three methods UPDATE_GRADE$(self, student, credit, grade)$, MIDDLE$(self)$, and AVERAGE$(self, student)$, and define whatever you need in your data structure under the __INIT__$(self, student_names, k)$ method.