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] $T(n) = 3T(n/3) + 25n$

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

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

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

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

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.

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

(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.

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

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 $O(n)$.

(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.

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

(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 $O(n)$.

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.)

(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.

(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.