Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Instructors: Piotr Indyk and Ron Rivest

October 1, 2015
Problem Set 2

Problem Set 2

All parts are due Thursday, October 15 at 11:59PM.

Name: David Walter

Collaborators: TAs in Office hours

Part A

Problem 2-1. Whimsical Tree Algorithm

Store the trees as they appear in an heap called $calc$. Step 1 ($O(log(nk))$: Every time you begin finding child whimsical trees, you remove the minimum value in $calc$, append it to an array called $output$, and set the variable $parent$ to its value. Step 2(run through k times): Then you iterate through S, indexing with i from 0 to k-1 and insert $(parent + s_{i+1})$ to $calc$ until you are finished inserting $(calc[k-1] + s_k)$ to calc. You find the minimum of $calc$ and then, Step 3(run through n times) repeat the process again, just like above. Once length of $output$ is equal to n you return $output$. Step 1 runs in $O(log(nk))$ time and you run that process nk times, so the run time is $O(nklog(nk))$.

Problem 2-2. Dynamic Product of Matrices

The collection is initially empty, so inserting the first matrix into the augmented AVL tree will run in constant time. When adding matrices into a tree with greater than zero matrices, you will have to check if a matrix with the same key already exists. Searching an AVL tree for a key takes $O(logn)$ time. If the same key does not already exist, you can insert the matrix into the tree in $O(logn)$ time. If the key already exists then you can delete the old matrix in the tree in $O(logn)$ time and then insert the new matrix into the tree in $O(logn)$ time. Each node will also contain the value of the matrix itself and the multiplication between their left and right sub trees, if they exist. Starting at the children nodes, the auxiliary value that will be stored is simply the matrix value itself. As you move up the tree, each node will have the product of the left and right subtrees products. The number of matrix multiplication calculations is at most the height of the tree, because, when you call $UPDATE$ and add a matrix into a tree, in the worst case, as the matrix is being sorted in the tree, it will have to move up the tree h times, where h is the height of the tree. The number of balancing operations needed to keep a tree balanced is proportional to h, and in this case, you would be adding one additional matrix multiplication calculation at each balancing operation.

Part B

Problem 2-3.

(a) *Python script submitted seperately.*

(b) *Python script submitted seperately.*

(c) *Justification.*

The $get\_smallest\_at\_least$ algorithm works in $O(logn)$ time because the algorithm checks the left node, the current node, and the right node and decides to either traverse left or right, or return the current node or none. If the algorithm traverses to the left or right node, it can only do so as many as h times, where h is the height of the tree. Since we can assume the tree is balanced, the algorithm runs in $O(logn)$ time. The main algorithm works in $O(nlogn)$ time because the algorithm iterates through all n segments in the worst case. At each step through the segments, the algorithm either runs calculations that run in constant time, or the algorithm could run $get\_smallest\_at\_least$, $insert$, or $delete$, and the latter three functions take $O(logn)$ time each. So the algorithm runs in $O(nlogn)$ time.