Introduction to Algorithms: 6.006
Massachusetts Institute of Technology February 20, 2014
Professors Srini Devadas, Nancy Lynch and Vinod Vaikuntanathan Problem Set 2

Problem Set 2

All parts are due Thursday, March 6 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.md` file for instructions on preparing your solutions. 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. [20 points] **Augmented Merge Sort**

Simple algorithms like Merge Sort can sometimes be augmented with extra information and extra processing to solve other problems along the way. This problem provides a nice example.

First, recall ordinary Merge Sort: The input is an array A of any size n with keys chosen from some totally ordered set. To make things simple, we assume here that A does not contain any repeated keys and that n is a power of 2. The problem is to produce a new array B that contains the same elements as A, but in strictly increasing order. Merge Sort solves the problem recursively, by splitting A into two (approximately) equal parts, recursively sorting the two parts, and then merging the two sorted results with a simple linear-time Merge subroutine. This algorithm works in $\Theta(n \log n)$ time, as you can see by solving a recurrence or by expanding the recursion tree.

Now suppose that, while sorting, we want to keep track of how many pairs of elements get interchanged, that is, how many pairs are out of order in the original array A. For example, if A is the array 2,6,9,3,5,1,7,4, then the sorted version B is 1,2,3,4,5,6,7,9, and the number of pairs that are interchanged to get from A to its sorted version B is 14.

This could be interesting, for example, when we are comparing two ways of sorting the same list of elements. Suppose that A is a list of students in 6.006 in order of their SAT scores. Now we associate with each student a key representing the student's GPA. The number of interchanges needed to sort A according to GPA is some kind of measure of how well the SATs predict GPAs.

So let's define a new problem, the Interchange-Count Sorting Problem: The input is as before: an array A of any size n with keys in a totally ordered set, where A has no repeats and n is a power of 2. The problem is to sort A as before, and also to output the exact number of interchanges that are made in converting A into its sorted version B.

(a) [6 points] Describe (in English and pseudocode) an AugmentedMerge algorithm, which, given two sorted lists B_1 and B_2 of known lengths m_1 and m_2, produces the

merge B of the two lists, and at the same time tags each element v in the merged list B that came from B_2 with the exact number of elements from B_1 that follow v in B.

(b) [6 points] Use ideas from Part (a) to design an algorithm based on Merge Sort that solves the Interchange-Count Sorting Problem. Describe it in English and give pseudocode.

(c) [3 points] Analyze the time complexity of your algorithm, assuming the Python cost model.

(d) [5 points] Briefly outline a proof that the algorithm computes the correct interchange count.

Problem 2-2. [20 points] **DVL Trees**

In class, you learned about AVL trees. "AVL" abbreviates the names of the designers of the data structure, Adelson-Velskii and Landis, who presented the trees in a 1962 paper entitled "An Algorithm for the Organization of Information". An AVL tree maintains the invariant that for every node u in the tree, the heights of u's subtrees differ by at most 1. It does this by rebalancing whenever a node's subtrees would otherwise be greater than 1.

However, one might object that AVL trees do extra work by balancing every time the tree gets even a little out of balance. Perhaps it might be better to allow a little leeway, so we maintain the invariant that the heights of a node's subtrees differ by at most some larger constant c, rather than just 1. We call such a structure a DVL tree.

(a) [10 points] Design two algorithms to support the DVL Insert operation and the Search operation, respectively. Give pseudocode.

(b) [5 points] Give an example sequence of insertions, starting from an empty tree, for which an AVL tree rebalances at least once but a DVL tree does not.

(c) [5 points] Prove that DVL trees always have height $O(\log n)$.

Problem 2-3. [20 points] **Radix Sort**

Each day, your friend examines an on-line dictionary and produces a file consisting of a list of the distinct words in the dictionary (and their definitions), in an arbitrary order. Assume that the words are all over the usual 26-letter alphabet, without hyphens or other special characters. The spellings are not case-sensitive. The words may be of any length. Your job is to produce an alphabetized list of all the words (and their definitions).

(a) [3 points] Your first idea, based on misunderstanding radix sort, is to sort the list according to the first letter, then sort the resulting list according to the second letter, etc. Give a very small example showing that this idea does not work correctly.

(b) [5 points] Give pseudocode for an algorithm that alphabetizes the dictionary using radix sort correctly.

(c) [3 points] Analyze the time cost of your algorithm from part (b). Suppose that the list consists of at most n words and the maximum word length is k.

(d) [5 points] Now suppose we try to reduce the number of passes made by radix sort, by letting each pass work with a string of m adjacent letters rather than just a single letter. Give pseudocode for such a modification.

(e) [4 points] Analyze the time cost for your algorithm from part (d). What are good choices for m?

Part B

Problem 2-4. [40 points] **Baseball Stats Database**

Major League Baseball keeps track of a *lot* of statistics, which are used to compare players along many dimensions. These statistics are used by teams to decide which players to hire, and which should play in certain game situations. They are used by fans for other purposes, such as gambling and playing Fantasy Baseball.

In this problem, we will consider a highly simplified database of baseball statistics: for about 12,000 players, the database will keep track of just a few things:

- The player's name.
- AB, his number of "at-bats", or turns to hit.
- H, his number of "hits", which are successful at-bats.
- AVG, his "batting average", which is simply the ratio H / AB.
- RBI, his number of "runs batted in"; each successful at-bat generates a number from 0 to 4 indicating how many runs are scored as a result; this gets added to his total RBI.

(a) [35 points] Implement in Python a data structure to keep track of the latest values of all these stats for all players, throughout an entire baseball season. *Your data structure should be based on heaps.* You are **NOT** allowed to use the python `heapq` module.

We will consider a total of approximately 1,200,000 at-bats, for all the players and many games. This number of at-bats represents several seasons. (Note: This data has been artificially produced.) Information about at-bats arrives one at a time. Your data structure should support the following operations:

- NewAtBat, which incorporates the result of a new at-bat into the data structure. This operation takes a player's name, a 1 or a 0 saying whether he got a hit or not, and a number from 0 to 4 indicating the number of runs he batted in.
- CurrentStats, which takes a player's name and returns all of his current stats.
- CurrentBest, which takes the name of a stat (AB, H, AVG, or R), and a number k of players. It returns the names of the k players with the highest values of the given stat, in order high-to-low. In case of ties, the order doesn't matter.

To get you started, we have provided a `solution_template.py` file. This file contains a class called `Problem`; please implement the NewAtBat and CurrentBest operations as methods of this class (the signatures of these methods have already been provided in the distributed template file). You can initialize all data structures that you need for this problem in the constructor of `Problem`.

The `solution_template.py` file already contains a class called `Heap` which contains methods for some heap manipulation operations. To get a complete working implementation, you have to implement the remaining heap operations: `heapify_down` and `extract_max`.

To be completely clear, we define the `heapify_up` and `heapify_down` operations below.

- `heapify_up(A, i)`: Trickles a value up the heap. Compares the value originally at index i of the heap with its parent, and if the heap property is violated, swaps the two. Continues up the heap until the heap property is finally satisfied.
- `heapify_down(A, i)`: Trickles a value down the heap. Same operation as the `MAX_HEAPIFY` operation described in lecture and in CLRS. Compares the value originally at index i of the heap with the larger of its two children, and if the heap property is violated, swaps the two. Continues down the heap until the heap property is satisfied.

In addition, to make our lives easier, we **require** that the CurrentStats method return a `PlayerRecord` object and the CurrentBest method return a list of `PlayerRecord` objects of the desired size.

We will be testing your code for correctness as well as performance. Please make sure your submitted code is correct; you will receive no partial credit for an incorrect solution but partial credit will be awarded for less efficient solutions. Once you are done, please upload the `solution_template.py` to `alg.csail.mit.edu`.

We *strongly* encourage you to test locally, tests are available in the `prompt/tests/` directory. To check for correctness, run `python solution_template.py --c <test_file_name>` and to test performance run `python solution_template.py <test_file_name>`. More information about local testing can be found in the `README.md` file.

(b) [5 points] Analyze the worst-case time complexity of the operations on your data structure.

NOTE: Problem 2.4(b) should be turned in along with **Part A**. Problem 2.4(a) should be turned in on `alg.csail.mit.edu`.