Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — October 5th, 2010
Professors Konstantinos Daskalakis and Patrick Jaillet — Handout 3

Problem Set 3

This problem set is divided into two parts: Part A problems are theory questions, and Part B problems are programming tasks.

Part A questions are due **Tuesday, October 19th** at **11:59PM**.

Part B questions are due **Thursday, October 21st** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF form using LaTeX or scanned handwritten solutions. Your solution to Part B should be a valid Python file, together with one PDF file containing your solutions to the two theoretical questions in Part B.

Templates for writing up solutions in LaTeX are available on the course website.

Remember, your goal is to communicate. Full credit will be given only to the 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: Due Tuesday, October 19th

1. **(20 points)** Finding the Largest i Elements in Sorted Order

   Given an array of n numbers, we want to find the i largest elements in sorted order. That is, we want to produce a list containing, in order, the largest element of the array, followed by the 2nd largest, etc., until the ith largest. Assume that i is fixed beforehand, and all inputs have $n > i$.

   (a) **(5 points)** One idea is to mergesort the input array in descending order, and then list the first i elements of the array. Analyze the running time of this algorithm in terms of n and i.

   Solution: From the lecture we know that mergesort runs in $O(n \log n)$ time. Once the sorted array is obtained, looking at it first i elements takes $O(i)$ time. The resulting total running time is $O(i + n \log n)$, which is $O(n \log n)$ by our assumption that $i < n$.

   (b) **(10 points)** Describe an algorithm that achieves a faster asymptotic time bound than the one in Part (a). Analyze the running time of this algorithm in terms of n and i.

   Solution: We create first a max-heap out of the input array – from the lecture we know that this can be done in $O(n)$ time. Next, we extract the maximum element of this heap i times (each extraction can be done in $O(\log n)$ time, for a total of $O(i \log n)$ running time), and output the list of the i elements extracted in this way.

By definition of the max-heap, this algorithm is correct. Its total running time is $O(n + i \log n)$. Note that, for all $i < n$, this running time is asymptotically smaller than the above $O(n \log n)$ running time.

Observation: By employing the trick from the solution to problem 7 form Quiz 1, one can obtain an $O(n + i \log i)$ running time procedure. However, $O(n + i \log i)$ is asymptotically the same as $O(n + i \log n)$, so this improvement does not offer asymptotic speed-up. (Still, congratulations to a few of you that noticed this improvement!)

(c) **(5 points)** Now suppose that the elements of the array are drawn, without replacement, from the set $\{1, 2, ..., 2n\}$. Give an algorithm that solves the problem, with this additional information, and analyze its running time in terms of n and i.

Solution: We employ counting sort (as seen in the lecture) to sort in descending order the elements in $O(n + k) = O(n)$ time, where by our assumption k is equal to $2n$. Once we get the array sorted we just output first i elements. The total running time is $O(n + i) = O(n)$ time.

For parts (b) and (c), full credit will be awarded only for an asymptotically optimal solution, with partial credit given for solutions with slower asymptotic running times.

2. **(15 points)** Dynamic Medians

Marianne Midling needs a data structure "DM" for maintaining a set S of numbers, supporting the following operations:

- CREATE( ): Create an empty set S
- INSERT(x): Add a new given number x to S
- MEDIAN( ): Return a median of S. (Note that if S has even size, there are two medians; either may be returned. A median of a set of n distinct elements is larger than or equal to exactly $\lfloor (n + 1)/2 \rfloor$ or $\lceil (n + 1)/2 \rceil$ elements.)

(Assume no duplicates are added to S.)

Marianne proposes to implement this "dynamic median" data structure DM using a max-heap A and a min-heap B, such that the following two properties always hold:

1. Every element in A is less than every element in B, and
2. the size of A equals the size of B, or is one less.

To return a median of S, she proposes to return the minimum element of B.

(a) **(5 points)** Argue that this is correct (i.e., that a median is returned).

Solution: Let n be the size of S.

By property 1, we know that the minimum element e of B is larger than all the elements in A. Furthermore, by its minimality, e is smaller than all the other elements in B. By property 2, we know that A has to have exactly $\lceil (n+1)/2 \rceil - 1$ elements. So, e is larger or equal to exactly $\lceil (n+1)/2 \rceil$ elements. Thus, e is indeed the median of the set and Marianne's algorithm is correct.

(b) **(10 points)** Explain how to implement INSERT(x), while maintaining the relevant properties. Analyze the running time of your INSERT algorithm in terms of n, the number of elements in S.

Solution: Note that since initially property 2 holds, $|A| = \lceil (n+1)/2 \rceil - 1$, and $|B| = \lfloor (n+1)/2 \rfloor$, where n is the size of S.

Let x be the element to be inserted. We start by comparing x to the minimum element b of B (we can do it in $O(1)$ time since B is a min-heap). If $x > b$ we do a heap-insertion of x into B, otherwise, we do a heap-insertion of x into A. (Either of these steps will take $O(\log n)$ time.) This way of inserting x ensures that the property 1 still holds.

However, after this insertion, property 2 might be violated by either $|A|$ becoming equal to $|B| + 1$, or $|B|$ becoming equal to $|A| + 2$. In the first case we extract the maximum element of A and heap-insert it into B (since A is a max-heap and B is a heap, this can be done in $O(\log n)$ time). Similarly, in the second case we extract the minimum element of B and heap-insert it into A (once again, we can do it in $O(\log n)$ time). After this operation, property 2 holds again, and we did not violate property 1 while doing it.

3. **(15 points)** The Optimality of the Binary Search Tree Construction

Consider a list of n numbers and a task of building a binary search tree that contains them. In Problem 2.1 (from Problem set 2) we outlined an algorithm that performs this task by starting with an empty tree and inserting to it the elements of the list one by one. As we mentioned, by employing appropriate rebalancing procedures, e.g. as in AVL-trees, the total running time of this procedure is $O(n \log n)$.

Prove that in the comparison-based model of computation this running time is asymptotically optimal. Namely, show that there is no algorithm in the comparison-based model that given a list of n numbers constructs a binary search trees containing them in $o(n \log n)$ time.

Hint: As a first step, show that given a binary search tree one can output a sorted list of all the numbers it contains in $O(n)$ time.

Solution: First, recall from recitations (or CLRS) that given a binary search tree, we can use an in-order traversal of this tree to obtain a sorted list of all its elements in $O(n)$ time.

Now, assume for the sake of contradiction that there exists a comparison-based procedure that given a list of n elements can produce a binary search tree representing this list and that this procedure runs in $o(n \log n)$ time. By composing this procedure with in-order traversal of the resulting tree, we can output all the elements of the input list in sorted order in time $O(n) + o(n \log n)$ which is $o(n \log n)$.

Since in-order traversal does not require accessing the values of the elements that are kept in the tree, we would obtain in this way a comparison-based procedure for sorting an arbitrary list of n elements in $o(n \log n)$ time. However, this would contradict the fact proved in the lecture that any such procedure has to have $\Omega(n \log n)$ running time. Thus, we obtained a contradiction that yields our desired claim.