Problem 4-1 (Humble Numbers) 10 points

A number $k > 1$ is called humble if the only prime factors of $k$ are 3 and 5. Consider the task of, on input $n$, outputting the $n$ smallest humble numbers and the following algorithm to do it:

```
Humble(n):
    count = 0, prevOutput = 0
    Heap.Insert(3)
    Heap.Insert(5)
    while (count < n)
        cur = Heap.ExtractMin
        if cur ≠ prevOutput then
            output cur
            Heap.Insert(3*cur)
            Heap.Insert(5*cur)
            count = count + 1
        prevOutput = cur
```

(a) (4 points) Argue that the algorithm above (1) outputs numbers in increasing order, (2) does not output any number twice, (3) only outputs humble numbers, and (4) outputs all of the first $n$ humble numbers.

(b) (2 points) Derive an exact (i.e., no $O$-notation) bound on the number of times Heap.Insert is called.

(c) (2 points) Bound exactly the number of times Heap.ExtractMin is called. (Hint: Use (b).)

(d) (2 points) Use the answers to (b) and (c) above to argue that Humble runs in $O(n \log n)$ time. Assume that arithmetic can be performed in $O(1)$ time.

Problem 4-2 (Running Median) 13 points

In this task you will design a data structure supporting the following operations:

- **Build($A[1..n]$)**: Initializes the data structure with the elements of the array $A$.
- **Insert(x)**: Inserts the element $x$ into the data structure.
• **MEDIAN**: Returns the median\(^1\) of the currently stored elements.

In the following, you are allowed to use the standard heap operations from CLRS Chapter 6 (incurred the corresponding running times).

(a) (4 points) Describe a data structure such that you can perform the operations BUILD, INSERT, and MEDIAN with running times as required below.

   (**Hint**: Use a min-heap and a max-heap.)

Describe in pseudo-code implementations of

(b) (3 points) BUILD running in time \(O(n)\) assuming that you can query an \(O(n)\) magic box for finding the median of an \(n\)-element array.\(^2\)

(c) (3 points) INSERT running in time \(O(\log n)\), where \(n\) is the number of elements in the data structure.

(d) (3 points) MEDIAN running in time \(O(1)\).

**Problem 4-3 (QuickSort vs Insertion Sort)** 14 Points

We say that an array \(A\) is \(c\)-nice for a constant \(c\) if for all \(1 \leq i < j \leq n\) such that \(j - i \geq c\), we have that \(A[i] \leq A[j]\). For example, a 1-nice array is completely sorted (in ascending order). In this problem we will sort such \(c\)-nice arrays \(A\) using **INSERTIONSORT** and **QUICKSORT** and compare the results.

(a) (4 Points) In asymptotic notation (remember that \(c\) is a constant) what is the worst-case running time of **INSERTIONSORT** on a \(c\)-nice array?

Consider now a run of **QUICKSORT** on a \(c\)-nice array (where the pivot element is chosen (deterministically) as the last element of the array).

(b) (2 points) Derive a lower bound on the rank \(q\) of the pivot.\(^3\)

(c) (3 points) Argue that after partitioning, the two subarrays \(A[1 \ldots q - 1]\) and \(A[q + 1 \ldots n]\) to the left and to the right of the pivot, respectively, are both \(c\)-nice.

(d) (4 points) From the lecture you already know that the running time of quicksort on sorted arrays is \(\Theta(n^2)\). Let \(B(n)\) denote the best-case running time of **QUICKSORT** on \(c\)-nice array with \(n\) elements. Using your results from (b) and (c), derive a recurrence for \(B(n)\) and solve it.

(e) (1 point) Asymptotically, which is faster on \(c\)-nice arrays: the worst-case running time of insertion sort or the best-case running time of quicksort?

---

\(^1\)The median of \(n\) elements is the \(\left\lceil \frac{n}{2} \right\rceil\)-largest element.

\(^2\)Such an algorithms will be discussed in class in a few weeks’ time.

\(^3\)Among \(n\) elements, the \(i\)th smallest element has rank \(i\).
Problem 4-4 (Three-way partitioning) 8 points

Recall that quicksort selects an element as pivot, partitions an array around the pivot, and recurses on the left and on the right of the pivot. Consider an array that contains many duplicates and observe that for such an array, quicksort recurses on all duplicates of the pivot element. In this task you are to develop a new partitioning procedure that works well on arrays with many duplicates. The idea is to partition the array into elements less than the pivot, equal to the pivot and greater than the pivot.

(a) (4 points) Develop this idea into a partitioning algorithm and provide pseudocode. Make sure your algorithm is in-place (i.e., do not use more than a constant amount of extra space).

(b) (2 points) Use your partitioning algorithm to come up with a sorting algorithm. Analyze the worst-case running time of your algorithm.

(c) (2 points) Find an array on which the original quicksort runs in time $\Theta(n^2)$ but your algorithm from (b) in $\Theta(n)$. 

PS 4, Page 3