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:

$$\text{HUMBLE}(n):$$

- $\text{count} = 0, \text{prevOutput} = 0$
- $\text{HEAP.INSERT}(3)$
- $\text{HEAP.INSERT}(5)$
- while ($\text{count} < n$)
  - $\text{cur} = \text{HEAP.EXTRACTMIN}$
  - if $\text{cur} \neq \text{prevOutput}$ then
    - output $\text{cur}$
    - $\text{HEAP.INSERT}(3*\text{cur})$
    - $\text{HEAP.INSERT}(5*\text{cur})$
    - $\text{count} = \text{count} + 1$
    - $\text{prevOutput} = \text{cur}$

(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.

**Solution:** INSERT YOUR SOLUTION HERE

(b) (2 points) Derive an *exact* (i.e., no $O$-notation) bound on the number of times $\text{HEAP.INSERT}$ is called.

**Solution:** INSERT YOUR SOLUTION HERE

(c) (2 points) Bound *exactly* the number of times $\text{HEAP.EXTRACTMIN}$ is called.

(Hint: Use (b).)

**Solution:** INSERT YOUR SOLUTION HERE

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

**Solution:** INSERT YOUR SOLUTION HERE
In this task you will design a data structure supporting the following operations:

- **BUILD**(\(A[1 \ldots n]\)): 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 (incurring 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.)

**Solution:** INSERT YOUR SOLUTION HERE

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,\(^2\)

**Solution:** INSERT YOUR SOLUTION HERE

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

**Solution:** INSERT YOUR SOLUTION HERE

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

**Solution:** INSERT YOUR SOLUTION HERE

---

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

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

Solution: INSERT YOUR SOLUTION HERE

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

Solution: INSERT YOUR SOLUTION HERE

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

Solution: INSERT YOUR SOLUTION HERE
Recall that in the worst case the running time of (non-randomized) QuickSort and InsertionSort are $\Theta(n^2)$.

(a) (2 points) Give an example of array of length $n$ where (non-randomized) QuickSort and InsertionSort both take time $\Omega(n^2)$. Justify your answer.

**Solution:** INSERT YOUR SOLUTION HERE

(b) (1 points) Give an example of array of length $n$ where (non-randomized) QuickSort runs in time $\Omega(n^2)$ but InsertionSort runs in time $O(n)$. Justify your answer.

**Solution:** INSERT YOUR SOLUTION HERE

Let $A[1 \ldots n]$ be a random permutation of $\{1, 2, \ldots, n\}$. Recall that the pair $(i, j)$ is called an inversion if $i < j$ and $A[i] > A[j]$.

(c) (2 point) What is the expected running time of InsertionSort on $A$? (**Hint:** Recall that the running time of InsertionSort is $\Theta(n + I)$ where $I$ is the number of inversions in $A$. Thus, it suffices to compute the expected number of inversions in $A$.)

**Solution:** INSERT YOUR SOLUTION HERE

(d) (6 points) Assume we have a partition procedure that is stable – an algorithm is stable if it maintains the relative order of elements. Thus, a partition procedure is stable if it preserves the relative order of elements less than the pivot and the relative order of elements greater than pivot after the partition.

(i) Prove that a stable partition on $A$ where $A$ is a random permutation, results in subarrays that are random permutations themselves. (**Hint:** Consider the two distributions: (a) Running the stable partition on a random permutation $A$ and taking the left half. (b) a random permutation over $B$ where $B$ consists of all elements in $A$ less than the pivot.)

**Solution:** INSERT YOUR SOLUTION HERE

(ii) Use the above observation to write a recurrence for the expected running time of QuickSort on a random permutation $A$, and solve it. (**Hint:** Recall the recurrence for Randomized QuickSort from the lecture.)

**Solution:** INSERT YOUR SOLUTION HERE

(iii) Which of QuickSort and InsertionSort has better expected running time on a random permutation?
(e) (3 points) [Extra credit] Prove that the expected running time of QuickSort computed in part (d) does not change if we use the standard (non-stable) Partition procedure from the book.

Solution: INSERT YOUR SOLUTION HERE