Homework 3: Solutions by your TA

1 Problem 1

6.3-3 Need to show that the number of nodes of height $h$ in any $n$-element heap is at most $\lceil n/2^{h+1} \rceil$.

Number of nodes at depth $d$ is atmost $2^d$.

PROOF: By induction on number of nodes at height $h$.

Base case: Number of leaves at height $h = 0 = \text{number of leaves in the } n\text{-element heap}$.
Parent of the last node is the $\lceil n/2 \rceil$-node. This will be the last parent in the heap. All
the nodes after this one will be leaves. Therefore the number of leaves is $\lceil n/2 \rceil$. This
is also true for the formula, therefore base case is true.

We assume that number of nodes at height $h - 1$ is given by the formula, $\lceil n/2^h \rceil$.

Note that if we remove all the leaves from the heap, the nodes that were earlier at
height 1 now become leaves in the new heap, i.e. they have height 0. Similarly, all the
nodes in the new tree will have height that is one less than their old height. We shall
use this to prove the induction.

Induction: Consider a new tree that is obtained by removing the leaves. This tree is
a heap with $n - \lceil n/2 \rceil$ nodes in it. The number of nodes at height $h$ in the old heap
will be the same at the number of nodes at height $h - 1$ in the new heap, which is
$(n - \lceil n/2 \rceil)/2^h = \lceil n/2^{h+1} \rceil$.

2 Problem 2

6.5-8 There are $k$ sorted lists with $n/k$ number of elements each. We need to give an
$O(n \lg k)$ algorithm to merge the lists into a single sorted list.

Solution using min-heaps.

Properties of min-heaps:
• **min–heap** has the **min–heap property**, i.e. every node (other than the root) has its value at least as much as that of its parent node. Therefore, the root of a min–heap is the minimum of all the elements in the heap.

• **BUILD-MIN-HEAP**: builds a min–heap out of \( n \) elements in \( O(n) \) time.

• **HEAP-MINIMUM**: returns the minimum of the elements in heap in \( O(1) \) time.

• **MIN-HEAPIFY**(root): It assumes that subtrees rooted at the children of the root are already min–heaps and heapifies the tree (with \( n \) nodes) in \( O(\lg n) \) time.

(i) Pick out the first element from each of the \( k \) sorted lists and form a min–heap with them. The size of the heap is \( k \). **BUILD-MIN-HEAPIFY** takes \( O(k) \) time. Along with each element in the heap, we shall also store the list–id of the list from which it has been taken.

At this point the minimum is at the root of the min–heap.

(ii) Put the root into the final sorted list, and replace it with the next element of the same list. This takes \( O(1) \) time.

At this time, only the root may violate the min–heap property.

(iii) Run **MIN-HEAPIFY** over the root. This takes \( O(\lg k) \) time.

Then the root will contain the element next in the sorted list.

(iv) repeat step (ii). If there is an empty list, pick an element from the next non–empty list.

The running time of the algorithm is the time for step (i) + \( n \) times the time for steps (ii) & (iii).

i.e. \( O(k) + n\{O(1) + O(\lg k)\} = O(n \lg k) \).

**Solution using Merge Sort.**

Some students submitted an algorithm using Merge-Sort which works as follows.

There is a notion of level for the lists. The original \( k \) lists belong to level–0. The idea is to merge two lists at a lower level to get a list at the next level.
Initially there will be $k/2$ Merge-Sorts. Each of the Merge-Sorts will take two lists of size $n/k$ and produce a sorted list of size $2n/k$ of level-1. Therefore the cost of each of these Merge-Sorts is $O(2n/k)$. There will be almost $k/2$ such Merge-Sorts. Therefore the cost at this level will be $(k/2)O(2n/k) = O(n)$.

At the next level, two lists from the previous level are merged to get one list.

At level $i$, there will be $k/2^i$ Merge-Sorts, each of which will be over two lists of size $2^i n/k$. Therefore, the total running time at level $i$ is $(k/2^i)O(2^i n/k) = O(n)$. There will be $O(\log k)$ such levels. Therefore the running time of this algorithm is $O(\log k)O(n) = O(n \log k)$.

3 Problem 3

[7-3] Stooge Sort (page 161 from [CLRS]).

(a) Stooge Sort sorts the input list of elements.

Proof (by induction)

Base case: For a small list (list of sizes 1, 2, or 3), we can go over the algo and show that stooge sort actually works.

Inductive Hypothesis: Stooge sort works for lists of size less than or equal to $2n/3$.

Induction:

<table>
<thead>
<tr>
<th></th>
<th>First 1/3\textsuperscript{rd}</th>
<th>Second 1/3\textsuperscript{rd}</th>
<th>Third 1/3\textsuperscript{rd}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Sort</td>
<td>First 2/3\textsuperscript{rd} gets sorted</td>
<td>Unsorted</td>
<td></td>
</tr>
<tr>
<td>2nd Sort</td>
<td>Untouched</td>
<td>Last 2/3\textsuperscript{rd} gets sorted</td>
<td></td>
</tr>
<tr>
<td>3rd Sort</td>
<td>First 2/3\textsuperscript{rd} gets sorted</td>
<td>untouched (sorted)</td>
<td></td>
</tr>
</tbody>
</table>

At the end of the sorting for the third time (line 8 in the algo), we end up with two lists that are sorted for sure. The first 2/3\textsuperscript{rd} and the last 1/3\textsuperscript{rd} lists are sorted. However, we need to prove that the whole list is also sorted. For this we prove by contradiction.

Let $p$ be the last element from the first 2/3\textsuperscript{rd} of the list. Let $q$ be the first element from the last 1/3\textsuperscript{rd} of the list. Say $p > q$. 

3
If this is the case, then consider the list right after the second sort (line 7).  $p$ could not have been in the third $1/3^{rd}$ because that part is untouched in the third sort and $p$ does not belong there. (it is the last element in the other part of the list). It could not have been in the second $1/3^{rd}$ because, every element in that part is less than $q$, but $p > q$ according to our assumption.

So $p$ should have belonged to the first $1/3^{rd}$ part. But $p$ could have landed in this part only if it was smaller than half the elements sorted in the first sort. i.e. $p$ should have been smaller than atleast $1/3^{rd}$ of the elements in the whole list. This means that there should be atleast $1/3^{rd}$ elements following $p$ in the complete sorted list. But according to our assumption, $p > q$. This means that the number of elements following $p$ in the actual sorted list is less than $1/3^{rd}$, which is a contradiction.

Therefore, our assumption that $p > q$ is false and hence $p \leq q$. This means that the list at the end of the third sort (line 8) is sorted.

Note the importance of the fact that the size of the middle part needs to be atleast as large as the size of the other parts.

(b) Running time

Let $T(n)$ be the running time for input of size $n$. Lines 6, 7, and 8 of the algorithm take $T(2n/3)$ time each. The rest of the lines take $\Theta(1)$ time. Therefore we have the following recurrence,

$$T(n) = 3T(2n/3) + \Theta(1)$$

In this recurrence we have We have $a = 3, b = 3/2, f(n) = \Theta(1)$ and thus we have that $n^{\log_b a} = n^{\log_3 3/2} \approx n^{2.71}$. Since $f(n) = (n^{\log_3 3/2 - \epsilon})$, where $\epsilon \approx 2.71$, we can apply case 1 of the Master’s Theorem and get $T(n) = \Theta(n^{\log_3 3/2})$.

(c) Comparing with other sorting algorithms. The worst case running times of insert/quick sort are $O(n^2)$. The worst case running times of heap/merge sorts are $O(n \log n)$. The worst case running time of stooge sort is $O(n^{\log_3 3/2}) \approx O(n^{2.71})$ which is worse than all the others. ($O(n^2) = o(n^{2.71})$ and $O(n \log n) = o(n^{2.71})$).