Analysis of Algorithms: Solutions 3

<table>
<thead>
<tr>
<th>number of homeworks</th>
<th>0</th>
<th>0.5-1</th>
<th>1.5-2</th>
<th>2.5-3</th>
<th>3.5-4</th>
<th>4.5-5</th>
<th>5.5-6</th>
<th>6.5-7</th>
<th>7.5-8</th>
<th>8.5-9</th>
<th>9.5-10</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>grades</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The histogram shows the distribution of grades for the homeworks submitted on time.

Problem 1
A d-ary heap is like a binary heap, but instead of 2 children, nodes have d children.

(a) How would you represent a d-ary heap in an array? What is the height of a d-ary heap of n elements in terms of n and d?

The following expressions determine the parent and j-th child of element i (where 1 ≤ j ≤ d):

\[
\text{PARENT}(i) = \left\lfloor \frac{i + d - 2}{d} \right\rfloor ;
\]

\[
\text{CHILD}(i, j) = (i - 1)d + j + 1.
\]

The height h of a heap is approximately equal to \(\log_d n\). The exact height is

\[h = \lfloor \log_d (nd - n + 1) - 1 \rfloor .\]

(b) Give an efficient implementation of HEA POP-EXTRACT-MAX for a d-ary heap.

The HEA POP-EXTRACT-MAX procedure for d-ary heaps is identical to that for binary heaps; however, we have to re-implement HEA POP, which is a subroutine of HEA POP-EXTRACT-MAX.

HEA POP(A, i, n, d)

\[\text{largest} \leftarrow i\]

\[\text{for } j \leftarrow 1 \text{ to } d \quad \triangleright \text{loop through all children of } i\]

\[\text{do if CHILD}(i, j) \leq n \text{ and } A[\text{CHILD}(i, j)] > A[\text{largest}]\]

\[\text{then } \text{largest} \leftarrow \text{CHILD}(i, j)\]

\[\text{if } \text{largest} \neq i\]

\[\text{then exchange } A[i] \leftrightarrow A[\text{largest}]\]

HEA POP(A, largest)

(c) Give an efficient implementation of a HEA POP-INC REASE-KEY(A, i, k) algorithm, which sets \(A[i] \leftarrow \max(A[i], k)\) and updates the heap structure appropriately. Give its time complexity, in terms of \(d\) and \(n\), and briefly explain your answer.
Heap-Increase-Key($A, i, k$)

if $k > A[i]$

then while $i > 1$ and $A[\text{Parent}(i)] < k$

\begin{align*}
&\text{do } A[i] \leftarrow A[\text{Parent}(i)] \\
&\quad i \leftarrow \text{Parent}(i)
\end{align*}

$A[i] \leftarrow k$

The worst-case running time is proportional to the height of the heap; hence, it is $O(\log_d n)$.

**Problem 2**

Consider the following sorting algorithm:

\begin{itemize}
  \item \textbf{Stooge-Sort($A, i, j$)}
  \item 2. \textbf{then return }
  \item 3. if $i + 1 \geq j$
  \item 4. \textbf{then return }
  \item 5. $k \leftarrow \lfloor (j - i + 1)/3 \rfloor$
  \item 6. \textbf{Stooge-Sort($A, i, j - k$)} \Comment{First two-thirds.}
  \item 7. \textbf{Stooge-Sort($A, i + k, j$)} \Comment{Last two-thirds.}
  \item 8. \textbf{Stooge-Sort($A, i, j - k$)} \Comment{First two-thirds again.}
\end{itemize}

(a) Argue that \textbf{Stooge-Sort($A, 1, n$)} correctly sorts the input array $A[1..n]$.

We prove the correctness of the algorithm by induction. Clearly, the algorithm works correctly for one- and two-element arrays, which provides the induction base. Now suppose that it works for all arrays shorter than $A[i..j]$ and let us show that it also works for $A[i..j]$.

After the execution of Line 6, $A[i..(j - k)]$ is sorted, which means that every element of $A[(i + k)..(j - k)]$ is no smaller than every element of $A[i..(i + k - 1)]$ (we will write it as $A[(i + k)..(j - k)] \geq A[i..(i + k - 1)]$). Therefore, $A[(i + k)..j]$ contains at least $\text{length}(A[(i + k)..(j - k)]) = j - i - 2k + 1$ elements each of which is no smaller than each element of $A[i..(i + k - 1)]$.

After the execution of Line 7, $A[(i + k)..j]$ is sorted, which implies that

\begin{enumerate}
  \item $A[(j - k + 1)..j]$ is sorted, and
  \item $A[(j - k + 1)..j] \geq A[(i + k)..(j - k)]$.
\end{enumerate}

On the other hand, since $A[(i + k)..j]$ has at least $(j - i - 2k + 1)$ elements no smaller than each element of $A[i..(i + k - 1)]$ and $\text{length}(A[(j - k + 1)..j]) \leq j - i - 2k + 1$, we conclude that

\begin{enumerate}
  \item $A[(j - k + 1)..j] \geq A[i..(i + k - 1)]$.
\end{enumerate}

Putting together (2) and (3), we conclude that:

\begin{enumerate}
  \item $A[(j - k + 1)..j] \geq A[i..(j - k)]$.
\end{enumerate}

After the execution of Line 8, the array $A[i..(j - k)]$ is sorted. Putting this observation together with (1) and (4), we see that the whole array $A[i..j]$ is sorted.
(b) Give the recurrence for the worst-case running time of STOOGESORT and a tight asymptotic (Θ-notation) bound on the worst-case running time.

The algorithm first performs a constant-time computation (Lines 1–5), and then recursively calls itself three times (Lines 6–8), each time on an array whose size is 2/3 of the original array’s size. Thus, the recurrence is as follows:

\[ T(n) = 3T\left(\frac{2}{3}n\right) + \Theta(1). \]

This recurrence describes both the worst-case and best-case running time, since the algorithm’s behavior does not depend on the order of elements in the input array. We use the iteration method to solve it:

\[
T(n) = 1 + 3T\left(\frac{2}{3}n\right) \\
= 1 + 3 + 9T\left(\frac{4}{9}n\right) \\
\vdots \\
= 1 + 3 + 3^2 + \ldots + 3^{\log_3/2 n} \\
= \frac{3^{\log_3/2 n + 1} - 1}{3 - 1} \\
= \Theta(3^{\log_3/2 n}) \\
= \Theta\left(3^{\log_3 n}/\log_3 3/2\right) \\
= \Theta(n^{1/(\log_3 3/2)}) \\
= \Theta(n^{2.71}).
\]

(c) Compare the worst-case running time of STOOGESORT with that of insertion sort, merge-sort, heap-sort, and quick-sort. Is it a good algorithm?

STOOGESORT is slower than the other sorting algorithms. Even the insertion sort has the complexity \(O(n^2)\), which is much better than \(\Theta(n^{2.71})\).

**Problem 3**

We consider an integer array \(A[1..n]\) and define a segment sum from \(p\) to \(r\), where \(1 \leq p \leq r \leq n\), as follows:

\[ \text{sum}(p, r) = \sum_{i=p}^{r} A[i]. \]

That is, it is the sum of all array elements in the segment \(A[p..r]\). Note that the total number of distinct segments is \(\frac{n(n+1)}{2}\). Write a linear-time (that is, \(\Theta(n)\)) algorithm that determines the maximum over all segment sums.

**MAX-SEGMENT(A, n)**

\[ \text{Local-Max} \leftarrow 0 \]

\[ \text{Global-Max} \leftarrow 0 \]

**for** \(i \leftarrow 1 \text{ to } n\)

**do** \(\text{Local-Max} \leftarrow \max(A[i], \text{Local-Max} + A[i])\)

\(\triangleright \text{Local-Max} \) is the maximum over the segments whose last element is \(A[i]\).

\[ \text{Global-Max} \leftarrow \max(\text{Local-Max}, \text{Global-Max}) \]

\(\triangleright \text{Global-Max} \) is the maximum over all segments in \(A[1..i]\).

**return** \(\text{Global-Max}\)