Analysis of Algorithms: Solutions 3

```
                                     X
                                     X
number of                     X      X
homeworks                     X      X             X
                              X      X             X             X
                X             X      X             X             X      X
                X      X      X      X             X             X      X
  X      X      X      X      X      X      X      X      X      X      X
  ---------------------------------------------------------------------------
  0    0.5-1  1.5-2  2.5-3  3.5-4  4.5-5  5.5-6  6.5-7  7.5-8  8.5-9  9.5-10
                                   grades
```

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 $1 \leq j \leq d$):

$$\begin{aligned} \textsc{Parent}(i) &= \left\lfloor \frac{i+d-2}{d} \right\rfloor, \\ \textsc{Child}(i,j) &= (i-1)d + j + 1. \end{aligned}$$

The height h of a heap is *approximately* equal to $\log_d n$. The exact height is

$$h = \lceil \log_d(nd - n + 1) - 1 \rceil.$$

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

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

HEAPIFY(A, i, n, d)
largest $\leftarrow i$
for $j \leftarrow 1$ **to** d $\quad\triangleright$ loop through all children of i
    do **if** CHILD$(i,j) \leq n$ **and** $A[\textsc{Child}(i,j)] > A[largest]$
            then *largest* $\leftarrow$ CHILD(i,j)
if *largest* $\neq i$
    then exchange $A[i] \leftrightarrow A[largest]$
          HEAPIFY$(A, largest)$

(c) Give an efficient implementation of a HEAP-INCREASE-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** $A[\text{PARENT}(i)] < k$
    do $A[i] \leftarrow A[\text{PARENT}(i)]$
      $i \leftarrow \text{PARENT}(i)$
  $A[i] \leftarrow k$

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:

STOOGE-SORT(A, i, j)
1. **if** $A[i] > A[j]$
2.   **then** exchange $A[i] \leftrightarrow A[j]$
3. **if** $i + 1 \geq j$
4.   **then return**
5. $k \leftarrow \lfloor (j - i + 1)/3 \rfloor$
6. STOOGE-SORT$(A, i, j - k)$ ▷ First two-thirds.
7. STOOGE-SORT$(A, i + k, j)$ ▷ Last two-thirds.
8. STOOGE-SORT$(A, i, j - k)$ ▷ First two-thirds again.

(a) Argue that STOOGE-SORT$(A, 1, n)$ correctly sorts the input array $A[1..n]$.

We proof 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 $\mathit{length}(A[(i+k)..(j-k)]) = j - i - 2k + 1$ 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

(1) $A[(j-k+1)..j]$ is sorted, and

(2) $A[(j-k+1)..j] \geq A[(i+k)..(j-k)]$.

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 $\mathit{length}(A[(j-k+1)..j]) \leq j - i - 2k + 1$, we conclude that

(3) $A[(j-k+1)..j] \geq A[i..(i+k-1)]$.

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

(4) $A[(j-k+1)..j] \geq A[i..(j-k)]$.

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 STOOGE-SORT 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:

$$T(n) = 3T(\frac{2}{3}n) + \Theta(1).$$

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:

$$\begin{aligned}
T(n) &= 1 + 3T(\frac{2}{3}n) \\
&= 1 + 3 + 9T(\frac{4}{9}n) \\
&\quad \dots \\
&= 1 + 3 + 3^2 + \dots + 3^{\log_{3/2} n} \\
&= \frac{3^{\log_{3/2} n+1} - 1}{3 - 1} \\
&= \Theta(3^{\log_{3/2} n}) \\
&= \Theta(3^{(\log_3 n)/(\log_3 3/2)}) \\
&= \Theta(n^{1/(\log_3 3/2)}) \\
&= \Theta(n^{2.71}).
\end{aligned}$$

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

STOOGE-SORT 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 $1 \le p \le r \le n$, as follows:

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

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)
Local-Max ← 0
Global-Max ← 0
for i ← 1 to n
    do  Local-Max ← max(A[i], Local-Max+A[i])
                ▷ Local-Max is the maximum over the segments whose last element is A[i].
        Global-Max ← max(Local-Max, Global-Max)
                ▷ Global-Max is the maximum over all segments in A[1..i].
return Global-Max
```