Practice Problems Set 4

Fall 25

1. Heaps (attributed to CLRS Exercise 6.1-5)

   At which levels in a max-heap might the kth largest element reside, for $2 \leq k \leq \lfloor n/2 \rfloor$, assuming that all elements are distinct?

Solution:

Minimum Level: For $2 \leq k \leq \lfloor n/2 \rfloor$, the kth largest element cannot be the root. Therefore it must be at a level greater than 1. So the minimum possible level for the kth largest element is 2. This level is also always achievable even for $k = \lfloor n/2 \rfloor$: consider a max-heap that is a full binary tree with the $\lfloor n/2 \rfloor - 2$ largest elements other than the root and the kth largest element in the left subtree of the root, and the other elements in the right subtree with the kth largest element as the root.

Maximum Level: A node at level L has $L-1$ ancestors (the nodes on the path from the root to the node, excluding the node itself). If the kth largest element is at level L, it must have $L-1$ ancestors that are all larger than it. This leads to the inequality $L - 1 \leq k - 1$, or $L \leq k$. Thus the maximum possible level for the kth largest element is k. We can show this is always achievable by constructing a valid heap. Let the k largest elements be $x_1, x_2, \ldots, x_k$ in decreasing order. Then consider the chain where x_1 is the root, x_2 is the child of x_1, x_3 is the child of x_2, and so on, until x_k is the child of x_{k-1}. In this configuration, x_k is at level k. The remaining $n - k$ smaller elements can be placed elsewhere in the heap without violating the max-heap property. The condition $k \leq \lfloor n/2 \rfloor$ ensures there are enough nodes to form the rest of the heap.

Therefore, for $2 \leq k \leq \lfloor n/2 \rfloor$, the kth largest element can be found at any level from 2 to k, inclusive.

2. Building a heap using insertion (attributed to CLRS Problem 6-1)

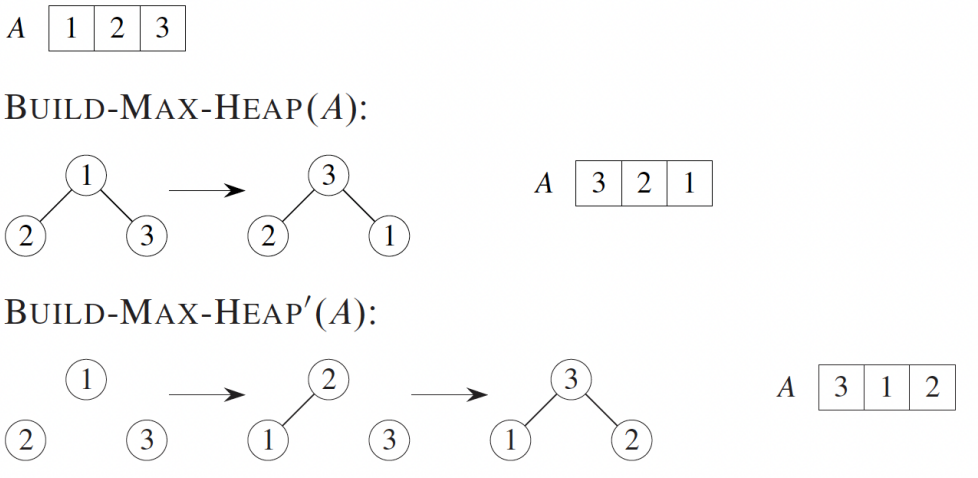
   One way to build a heap is by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the procedure BUILD-MAX-HEAP' below. It assumes that the objects being inserted are just the heap elements.

```
0   BUILD–MAX–HEAP'(A, n)
1       A.heap–size = 1
2       for i = 2 to n
3           MAX–HEAP–INSERT(A, A[i], n)
```

(a) Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP' always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

Solution:

The procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP' do not always create the same heap when run on the same input array. Consider the following counterexample of input array A:



(b) Show that in the worst case, BUILD-MAX-HEAP' requires $\Theta(n \lg n)$ time to build an n-element heap.

Solution:

An upper bound of $O(n \lg n)$ time follows immediately from there being $n-1$ calls to MAX-HEAP-INSERT, each taking $O(\lg n)$ time. For a lower bound of $\Omega(n \lg n)$, consider the case in which the input array is given in strictly increasing order. Each call to MAX-HEAP-INSERT causes HEAP-INCREASE-KEY to go all the way up to the root. Since the depth of node i is $\lfloor \lg i \rfloor$, the total time is

$$\begin{aligned}
\sum_{i=1}^{n} \Theta(\lfloor \lg i \rfloor) &\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg \lceil n/2 \rceil \rfloor) \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg(n/2) \rfloor) \\
&= \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg n - 1 \rfloor) \\
&\geq (n/2) \cdot \Theta(\lg n) \\
&= \Omega(n \lg n)
\end{aligned}$$

In the worst case, therefore, BUILD-MAX-HEAP' requires $\Theta(n \lg n)$ time to build an n-element heap.

3. Stooge Sort (attributed to CLRS Problem 7-4)

   Professors Howard, Fine, and Greene have proposed a deceptively simply sorting algorithm, named stooge sort in their honor:

```
0   Stooge–Sort(A, p, r)
1     if A[p] > A[r]
2       exchange A[p] with A[r]
3     if p + 1 < r
4       k = ⌊(r − p + 1)/3⌋ // round down
5       Stooge–Sort(A, p, r − k) // first two–thirds
6       Stooge–Sort(A, p + k, r) // last two–thirds
7       Stooge–Sort(A, p, k − k) // first two–thirds again
```

   (a) Argue that the call Stooge-Sort(A, 1, n) correctly sorts the array $A[1 : n]$.

Solution:

We first demonstrate that we always have $p \leq r$ in line 1, so that if line 2 executes, then the greater element moves to a higher index. In the initial call, $n \geq 1$, so that $p = 1 \leq n = r$. Now, consider any recursive call. Recursive calls occur only if line 3 finds that $p + 1 < r$, so that the subarray $A[p : r]$ has $r - p + 1 \geq 3$ elements. The value of k computed in line 4 is at least 1 and at most $(r-p+1)/3$ and so we have that $p < r - k$ in lines 5 and 7 and that $p + k < r$ in line 6. Therefore, all three recursive calls have the second parameter strictly less than the third parameter.

Note that by computing k as the floor of $(r - p + 1)/3$, rather than the ceiling, if the size of the subarray $A[p : r]$ is not an exact multiple of 3, then the subarray sizes in the recursive calls are for subarrays of size $\lceil 2(r - p + 1)/3 \rceil$, so that lines 5–7 are calls on subarrays at least 2/3 as large.

So now we need merely argue that sorting the first two-thirds, sorting the last two-thirds, and sorting the first two-thirds again suffices to sort the entire subarray. Denote the subarray size by n. Where can the $\lfloor n/3 \rfloor$ largest elements be after the first recursive call? They were either in the rightmost $\lfloor n/3 \rfloor$ positions, in which case they have not moved, or they were in the leftmost $\lceil 2n/3 \rceil$ positions, in which case they are in the rightmost positions within the leftmost $\lceil 2n/3 \rceil$. That is, they are in the middle $\lfloor n/3 \rfloor$ positions. The second recursive call guarantees that the largest $\lfloor n/3 \rfloor$ elements are in the rightmost $\lfloor n/3 \rfloor$ positions, and that they are sorted. All that remains is to sort the smallest $\lceil 2n/3 \rceil$ elements, which is taken care of by the third recursive call.

(b) Give a recurrence for the worst-case running time of Stooge-Sort and a tight asymptotic (Θ-notation) Bound on the worst-case running time.

Solution:

The recurrence is $T(n) = 3T(2n/3)+\Theta(1)$. We solve this recurrence by the master theorem with $a = 3$, $b = 3/2$, and $f(n) = \Theta(1) = \Theta(n^0)$. We need to determine $\log_{3/2} 3$, and it is approximately 2.71. Since $f(n) = O(n^{\log_{3/2} 3-\epsilon})$ for $\epsilon = 2.7$, this recurrence falls into case 1, with the solution $T(n) = \Theta(n^{\log_{3/2} 3})$.

(c) Compare the worst-case running time of Stooge-Sort with that of insertion sort, merge sort, heapsort and quicksort. Do the professors deserve tenure?

Solution:

The running time of Stooge-Sort is asymptotically greater than the worst-case running times of each of the four sorting methods in the question. No tenure for Professors Howard, Fine, and Greene. They should go back to teaching "Swingin' the Alphabet."

4. Trivially Sorted

An array with all elements having the same value is trivially sorted. Given a trivially-sorted array A of size n as input, what is the asymptotic running time of Heapsort?

Solution:

A trivially-sorted array is always a max-heap. In each iteration, the root is moved to the end of the array and the remaining portion is still a max-heap, so the Max-Heapify routine can be done in $\Theta(1)$ time, and the whole algorithm is $\Theta(n)$.