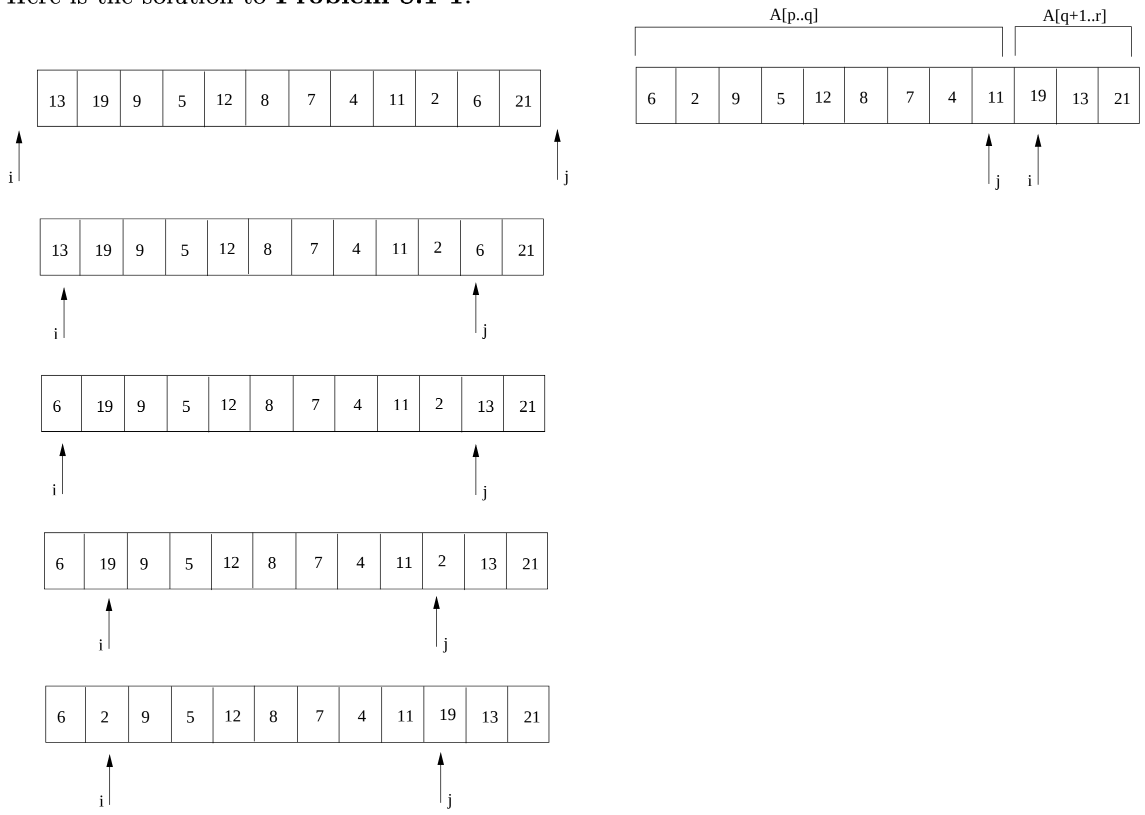
22C:44 Homework 4 Solution

1.
```
DELETE(A[1...n], i){
        SWAP(A[1...n], i, n)
        HEAPIFY(A[1...n-1], i)
}
```

This function has worst case running time $\Theta(\log n)$ because `SWAP` takes $\Theta(1)$ time and `HEAPIFY` takes $\Theta(\log n)$ in the worst case.

2. Here is the solution to **Problem 8.1-1**.



Here is the solution to **Problem 8.3-2**.

The worst case as well as the best case is $n-1$. To see this associate an execution tree with each execution of `RANDOMIZED-QUICKSORT` as follows. Associate to an array with 1 element a node containing that element. Associate to an array with 2 or more elements, a tree with a node whose left child is the root of execution tree associated with the first recursive call to `RANDOMIZED-QUICKSORT` and whose right child is the root of the execution tree associated with the second recursive call to `RANDOMIZED-QUICKSORT`. This tree has n leaves. Each internal node of this tree corresponds to a call to `RANDOMIZED-PARTITION` and therefore a call to `RANDOM`. Any binary tree with n leaves has $n-1$ internal nodes. Hence the result.

3.
```
NEWPARTITION(A[p...r]){
        pivot $\leftarrow$ A[p];
        pivotIndex $\leftarrow$ p;
        j $\leftarrow$ r;
        **while** (j > pivotIndex) **do**{
                **if**(A[j] $\geq$ pivot) **then**
```

```
                    j ← j - 1;
                else{
                    A[pivotIndex] ← A[j];
                    A[j] ← A[pivotIndex+1];
                    A[pivotIndex+1] ← pivot;
                    pivotIndex ← pivotIndex + 1;
                }
            }
            return pivotIndex;
        }
```

The above function is not as elegant or subtle as the PARTITION function in the book. However, the main idea is the same and NEWPARTITION is probably easier to understand and modify. Before each execution of the while-loop the array A satisfies the following properties:

(a) The pivot is at a slot whose index is pivotIndex.

(b) Every element to the left of pivotIndex is strictly smaller than pivot. Initially, this condition is trivially true since there are no elements to the left of pivotIndex.

(c) j is strictly larger than pivotIndex.

In each execution of the while-loop, j moves to the left until it bumps into pivotIndex or it reaches an element smaller than the pivot. If the latter happens, the element smaller than the pivot is moved into the block to the left of the pivot and the pivot itself moves forward one slot. It is easy to see that conditions (a)-(f) in the problem are all met.

4. (a) The worst-case time complexity of BetterBubbleSort is $O(n)$.

We will call each execution of the while-loop in BetterBubbleSort a pass and show that in at most c passes BetterBubbleSort will sort A. BetterBubbleSort uses one extra pass to check that A is sorted. So BetterBubbleSort uses at most $(c+1)$ passes doing $\Theta(n)$ work in each pass. Since c is a fixed constant, the total amount of work BetterBubbleSort does is $\Theta((c+1)n) = \Theta(n)$.

Without loss of generality, let A contain the elements $1, 2, \ldots, n$. For any i, $1 \leq i \leq n$, let R_i be the number of elements that are smaller than i *and* to its right. Clearly, for any i, $1 \leq i \leq n$, $R_i \leq c$. Note that in each pass of BubbleSort, R_i does not increase and if it is non-zero, it decreases by 1. This implies that in c passes of BubbleSort, $R_i = 0$ for all i, $1 \leq i \leq n$.

$R_n = 0$ implies that n is in A[n]. The fact that $R_{n-1} = 0$ and n is in A[n] implies that $(n-1)$ is in A[$n-1$]. We can continue in this manner to show that when $R_i = 0$ for all i, $1 \leq i \leq n$, then the array is sorted. Hence c passes of BubbleSort suffice to sort the array and BetterBubbleSort uses pass $(c+1)$ to detect that A is sorted and stops.

(b)

```
NEWERPARTITION(A[$p \cdots r$]){
        $x$ ← A[$p$];
        SORT(A[$p \cdots p+c$]);
        for $i \leftarrow p$ to $p+c$ do
                if ($x$ == A[$i$]) then
                        return $i$;
}
```

It does not matter how the subarray A[$p \cdots p+c$] is sorted in the above function. What matters is that the size of the subarray is $\Theta(1)$ and therefore we can sort this subarray in $\Theta(1)$ time. Also note that the for-loop runs through $\Theta(1)$ times, doing $\Theta(1)$ work in each execution. Therefore, `NEWERPARTITION` has running time $\Theta(1)$.

We show that `QUICKSORT` now runs in $\Theta(n)$ time. Let $T(n)$ denote the running time of `QUICKSORT` on an array of n elements. Clearly, $T(n) = \Omega(n)$ since the function needs to at least look at every element. To show that $T(n) = O(n)$ we use the following recurrence:

$$T(n) \leq \max_{1 \leq q \leq n-1} [T(q) + T(n-q)] + \Theta(1).$$

Since any function $f(n)$ that is $\Theta(1)$ satisfies $f(n) \leq k'$ for some constant k', the above recurrence can be rewritten as:

$$T(n) \leq \max_{1 \leq q \leq n-1} [T(q) + T(n-q)] + k'.$$

We solve this recurrence using the substitution method and our guess is $T(n) \leq kn - k'$ for some constant k. The base case is when $n = 1$ and we can choose $k \geq k' + T(1)$ to make the guess true for the base case. Assume that the guess is true for all numbers less than n. Substituting our guess in the recurrence we get:

$$\begin{aligned} T(n) &\leq \max_{1 \leq q \leq n-1} [kq - k' + k(n-q) - k'] + k' \\ &= kn - k' \end{aligned}$$

This shows that $T(n)$ is $\Theta(n)$.