Exercise 6.1-1

Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $2^{h+1} - 1$ elements (if it is complete) and at least $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

Exercise 6.1-2

Given an n-element heap of height h, we know from Exercise 6.1-1 that

$$2^h \le n \le 2^{h+1} - 1 < 2^{h+1} .$$

Thus, $h \le \lg n < h + 1$. Since h is an integer, $h = \lfloor \lg n \rfloor$ (by definition of $\lfloor \rfloor$).

Exercise 6.1-6

No, the array is not a max-heap. 7 is contained in position 9 of the array, so its parent must be in position 4, which contains 6. This violates the max-heap property.

Exercise 6.2-1

27	17	3	16	13	10	1	5	7	12	4	8	9	0
27	17	10	16	13	3	1	5	7	12	4	8	9	0
27	17	10	16	13	9	1	5	7	12	4	8	3	0

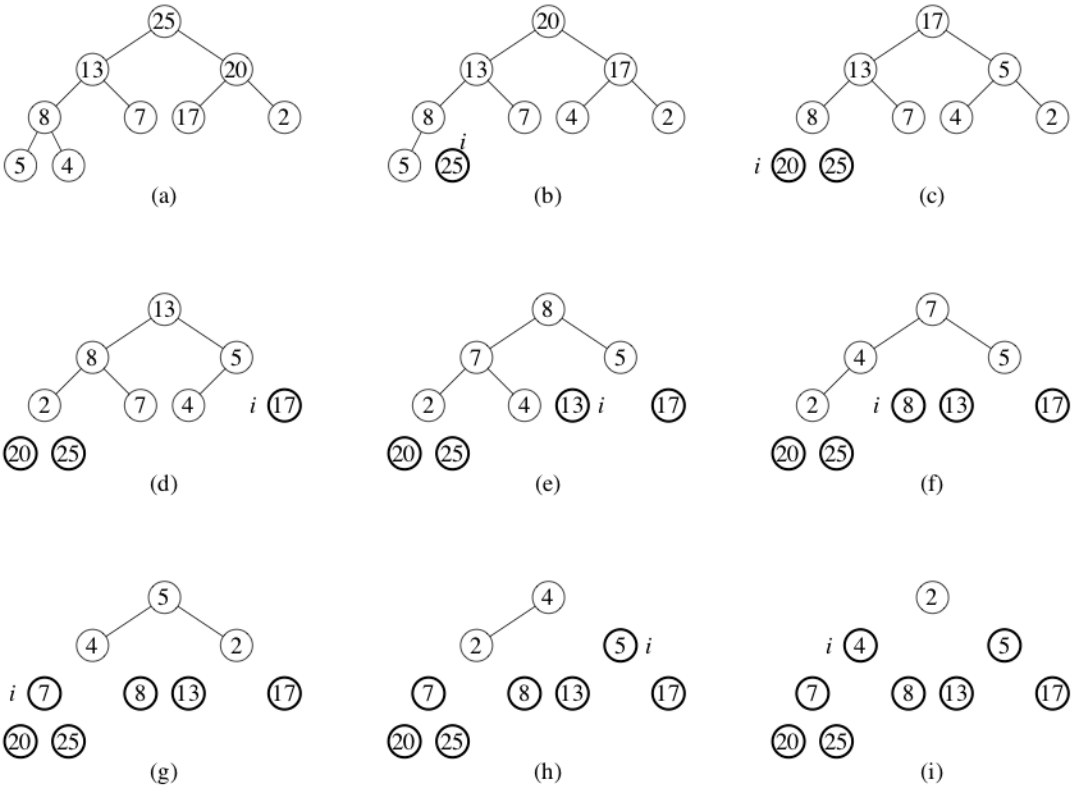
Exercise 6.2-3

The array remains unchanged since the if statement on line line 8 will be false.

Exercise 6.3-1

5	3	17	10	84	19	6	22	9
5	3	17	22	84	19	6	10	9
5	3	19	22	84	17	6	10	9
5	84	19	22	3	17	6	10	9
84	5	19	22	3	17	6	10	9
84	22	19	5	3	17	6	10	9
84	22	19	10	3	17	6	5	9

Exercise 6.4-1



A	2	4	5	7	8	13	17	20	25

Exercise 6.4-3

If it's already sorted in increasing order, doing the BUILD-MAX-HEAP call on line 1 will take $\Theta(n \lg(n))$ time. As we call MAX-HEAPIFY(A,i) in BUILD-HEAP, we know that $A[i]$ is larger than all elements to its right. This means that the time it takes will be $\lfloor \lg(n) - \lg(i) \rfloor$. So the total time to build the heap will be

$$\sum_{i=1}^{i=\lfloor n/2 \rfloor} \lg(n) - \lg(i) \in \Theta(n \lg(n))$$

There will be n iterations of the for loop in HEAPSORT, each taking $\Theta(\lg(n))$ time because the element that was at position i was the smallest and so will have $\lfloor \lg(n) \rfloor$ steps when doing MAX-HEAPIFY on line 5. So, it will be $\Theta(n \lg(n))$ time.

If it's already sorted in decreasing order, then the call on line one will only take $\Theta(n)$ time, since it was already a max heap to begin with. This is because elements in earlier positions correspond to begin further up in the heap, and since the array is descending, those have larger values. So, each of the MAX-HEAPIFY calls in the BUILD-MAX-HEAP procedure will only take constant time since the check on line 8 of MAX-HEAPIFY will always be false. It will still take $n \lg(n)$ peel off the elements from the heap and re-heapify. This is because each time we swap the head of the heap with the last element, we now have the smallest element at the head, so it will have to have as many recursive calls of MAX-HEAPIFY as the depth of the heap, which is about $\lg(n)$.

Exercise 7.1-1

13	19	9	5	12	8	7	4	21	2	6	11
13	19	9	5	12	8	7	4	21	2	6	11
13	19	9	5	12	8	7	4	21	2	6	11
9	19	13	5	12	8	7	4	21	2	6	11
9	5	13	19	12	8	7	4	21	2	6	11
9	5	13	19	12	8	7	4	21	2	6	11
9	5	8	19	12	13	7	4	21	2	6	11
9	5	8	7	12	13	19	4	21	2	6	11
9	5	8	7	4	13	19	12	21	2	6	11
9	5	8	7	4	13	19	12	21	2	6	11
9	5	8	7	4	2	19	12	21	13	6	11
9	5	8	7	4	2	6	12	21	13	19	11
9	5	8	7	4	2	6	11	21	13	19	12

Exercise 7.1-4

To modify QUICKSORT to run in non-increasing order we need only modify line 4 of PARTITION, changing $\leq$ to $\geq$.

7.2-3

PARTITION does a "worst-case partitioning" when the elements are in decreasing order. It reduces the size of the subarray under consideration by only 1 at each step, which we've seen has running time $\Theta(n^2)$.

In particular, PARTITION, given a subarray $A[p\,..\,r]$ of distinct elements in decreasing order, produces an empty partition in $A[p\,..\,q-1]$, puts the pivot (originally in $A[r]$) into $A[p]$, and produces a partition $A[p+1\,..\,r]$ with only one fewer element than $A[p\,..\,r]$. The recurrence for QUICKSORT becomes $T(n) = T(n-1) + \Theta(n)$, which has the solution $T(n) = \Theta(n^2)$.

Exercise 7.3-2

In either case, $\Theta(n)$ calls are made to RANDOM. PARTITION will run faster in the best case because the inputs will generally be smaller, but RANDOM is called every single time RANDOMIZED-PARTITION is called, which happens $\Theta(n)$ times.