7.3-2
In the worst case, when RANDOMIZED-PARTITION partition the array into n-1 elements and 0 elements. Number of call to RANDOM = Number of call to RANDOMIZED-PARTITION.

\[ T(n) = T(n-1) + 1 \]
So we know \( T(n) = \Theta(n) \)
Similarly, in the best case, \( T(n) = 2T(n/2) + 1, T(n) = \Theta(n) \)

9.3-1
If we divide the input into 7 groups instead of 5, after partitioning the input array with the median-of-median, say \( m^* \), as the pivot element, we can get a lower bound on the number of elements that are greater than \( m^* \) and a lower bound on the number of elements smaller than \( m^* \) as follows.

For elements greater than \( m^* \), half of sublists consisting of 7 elements has at least 4 elements that are greater than \( m^* \). We here are indeed discounting the sublist containing \( m^* \) and the final sublist which has a size at most 7. Thus number of elements greater than \( m^* \) is at least \( 4 \cdot (\lceil \frac{n}{7} \rceil - 2) \geq \frac{2n}{7} - 8 \).

Similarly, the number of elements that are smaller than \( m^* \) is also at least \( \frac{2n}{7} - 8 \).

So the algorithm in step 5, calls itself recursively on a problem of size at most \( n - \frac{2n}{7} - 8 = \frac{5n}{7} + 8 \). Thus the recurrence relation for the runtime becomes

\[ T(n) \leq T(\lceil \frac{n}{7} \rceil) + T(\frac{5n}{7} + 8) + O(n). \]

We want to verify if the above recurrence has the solution that \( T(n) = c \cdot n \) for some constant \( c > 0 \). To do so, we use the substitution method of solving recurrence relations. Assume \( T(n) \leq c \cdot n \) for some \( c > 0 \). We have that,

\[ T(n) \leq T(\lceil \frac{n}{7} \rceil) + T(\frac{5n}{7} + 8) + a \cdot n \]
\[ \leq c \cdot \frac{n}{7} + c(\frac{5n}{7} + 8) + a \cdot n \]
\[ \leq cn/7 + c(\frac{5n}{7} + 8) + a \cdot n \]

The above RHS is at most \( c \cdot n \) iff

\[ cn/7 + c(\frac{5n}{7} + 8) + a \cdot n \leq c \cdot n \]
\[ 8c + a \cdot n \leq c \cdot n \]
\[ a \cdot n \leq c \cdot (\frac{n}{n} - 8) \]
\[ c \geq \frac{7an}{2n-56} \]

So we should choose \( n > 56 \) and then a constant \( c \) exists such that \( c > \frac{7a}{1 - \frac{56}{n}} \).

So if we choose \( n > 2 \cdot 56 = 112 \), we have that \( 1 - \frac{56}{n} > 1/2 \) and \( c > 14a \). Since
the conditions for choosing \( c \) can be satisfied, the SELECT algorithm still runs in linear time.

For the case when we divide the input elements into groups of 3 elements each, we can still compute the number of elements that are greater than \( m^* \) and smaller than \( m^* \) similarly. We get that at least \( 2(\lceil \frac{1}{2} \frac{n}{3} \rceil - 2) \) elements are greater than \( m^* \) and a like number are smaller than \( m^* \). Thus, in step 5, the size of the subproblem is at most \( n - (\frac{n}{3} - 4) = \frac{2n}{3} + 4 \). The recurrence relation for the running time of SELECT becomes

\[
T(n) \leq T(\lceil n/3 \rceil) + T(\frac{2n}{3} + 4) + O(n).
\]

The solution for above recurrence does not satisfy \( T(n) = O(n) \). So with groups of 3 elements SELECT does not run in linear time. The reason for this is that during step 5, we are still left with a subproblems of total size \( n \). Thus, we did not manage to reduce the size of the problem effectively. This can also be seen by solving for \( T(n) \) using the iteration/recursion tree method where at each level of the tree we have a subproblem of size \( n \) and we are performing \( O(n) \) work at each level of the tree. So the overall runtime cannot be linear.

On the other hand, for the case where we divided the input into sublists of size 5 or 7, the total size of the problem reduces to less than \( n \).

For the case of dividing the input into sets of size 4, a similar calculation would show that the runtime of SELECT would be \( O(n \log n) \). It would be good exercise to put the size of the sublist as a parameter \( b \) and deduce conditions of the value of \( b \). (as in Q5).

9-1

\( a. \) Sort the numbers using merge-sort or heapsort, which takes \( \Theta(n \log n) \) worst-case time. (Don’t use quicksort or insertion sort, which can take \( \Theta(n^2) \) time.) Put the \( i \) largest elements into the output array, takes \( \Theta(i) \) time.

Total worst-case running time: \( \Theta(n \log n + i) = \Theta(n \log n) \)

\( b. \) Implement max-priority queue takes time of \( \Theta(n) \). Call Extract-Max \( i \) times takes time of \( \Theta(i \log n) \). So total worst-case running time is \( \Theta(n + i \log n) \).

\( c. \) Use the SELECT algorithm to find the \( i \)th largest number in \( \Theta(n) \) time. Sort the \( i \) largest takes \( \Theta(i \log i) \) worst-case time.

Total time: \( \Theta(n + i \log i) \).

17.1-3

For the first \( 2^n \) operations, the sum of costs are:

\[
1 + 2 + 1 + 4 + 1 + 1 + 1 + 8 + \ldots + 2^n = \sum_{i=1}^{n} 2^i + (2^n - n)
\]

So the cost per operation is \( \lim_{n \to \infty} \frac{\sum_{i=1}^{n} 2^i + (2^n - n)}{2^n} = \lim_{n \to \infty} \frac{2^{n+1} - 2 + 2^n - n}{2^n} = 3 \)

17.2-2

Let the amortized cost of each operation be: \( \hat{c} \)

Suppose \( \log n \) is integer, then:
\[\sum_{i=1}^{n} c_i = 2^{\lg n + 1} - 1 - 1 + 2^{\lg n} - \lg n\]
\[= 2n - 2 + n - \lg n\]
\[= 3n - \lg n - 2\]
\[\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i = 3n - (3n - \lg n - 2) = \lg n + 2 > 0\]