Ex. 6.3-1

Ex. 6.4-4. That is, for every value of \( n \), describe an input of length \( n \) that causes the time to be as bad as \( \Omega(n \log n) \).

When the input array is already sorted in increasing order, HEAPSORT takes \( \Omega(n \log n) \) time since each of the \( n-1 \) calls to MAX-HEAPIFY takes \( \Omega(\log n) \) time.

Prob. 6-2, parts (a) through (d)

(a)

The root of the heap is \( A[1] \), and given the index \( i \) of a node, the indices of its parent and \( j \)-th child (where \( j \) is between 1 and \( d \)) can be found as follows:
- Index of \( A[i] \)'s parent = floor \((i-2)/d + 1)\)
- Index of \( A[i] \)'s \( j \)-th child = \((i-1)d + 1 + j\)

(b)
The height is $\log_d n$

- (c) 
  EXTRACT-MAX(A) 
  \[
  \text{//identical to HEAP-EXTRACT-MAX(A) in p. 139}
  \]
  MAX-HEAPIFY (A, i) 
  \[
  \text{if } A[i] \text{ has children}
  \]
  \[
  \text{then find } A[i]'s \text{ largest child, } A[lc]
  \]
  \[
  \text{if } A[lc] \text{ is larger than } A[i]
  \]
  \[
  \text{then exchange } A[i] \text{ and } A[lc]
  \]
  \[
  \text{call MAX-HEAPIFY (A, lc)}
  \]

MAX-HEAPIFY takes $O(d \log_d n)$ time since the heap has height $O(\log_d n)$ and finding the largest child at each node takes $O(d)$ time. EXTRACT-MAX takes $O(d \log_d n) + O(1) = O(d \log_d n)$ time.

- (d) 
  INSERT (A, key) 
  \[
  \text{//identical to MAX-HEAP-INSERT (A, key) in p. 140.}
  \]

The running time of INSERT is $O(d \log_d n)$ since adding a new leaf node takes $O(1)$ time and updating the heap by calling HEAP-INCREASE-KEY takes $O(d \log_d n)$ time.

**Ex. 7.4-5**

- Argue that this sorting algorithm runs in $O(nk + n \log(n/k))$ expected time.

The expected running time of QUICKSORT for this sorting algorithm is $O(n \log (n/k))$ since the recursion tree has depth $\Theta(\lg (n/k))$ and PARTITION takes $\Theta(n)$ time at each level. Running insertion sort on the entire array afterwards takes $O(n(k-1)) = O(nk)$ time since for each element, at most $k-1$ shifts are required. Thus, the expected running time of this sorting algorithm is $O(nk + n \log(n/k))$.

- How should $k$ be picked, both in theory and in practice?

In theory, we want to pick $k$ so that the running time does not exceed $O(n \log n)$ expected running time of quicksort. In practice, we can pick $k$ between 1 and $\lg n$ by running experiments.

**Prob. 7-3**

- (a) 
  (i) When $n = 2$, STOOGESORT(A, 1, 2) correctly sorts the input array.
  
  (ii) Assume that STOOGESORT(A, 1, m) correctly sorts the input array for $m, 1 \leq m \leq n$. Consider STOOGESORT(A, 1, n+1).
    
    -- STOOGESORT in line 6 correctly sorts the first two-thirds of the array $A$ since $(2/3)^*(n+1) <= n$.
    
    -- Similarly, STOOGESORT in line 7 correctly sorts the last two-thirds of the array $A$.
    
    -- Before STOOGESORT in line 8 is called, the sorted elements in the last third of $A$ are larger than or equal to the elements in the first two thirds of $A$. STOOGESORT in line 8 correctly sorts the first two-thirds of the array, thereby sorting the whole array.

- (b) 
  $T(n) = 3T(2n/3) + O(1)$
  
  $T(n) = \Theta(n^{\log_{3/2} 3})$ by the Master theorem – case 1.

- (c)
\[ n^{\log_{1/3} 3} = n^{2.71}. \] The worst-case running time of STOOGE-SORT is worse than that of insertion sort, merge sort, heapsort, and quicksort.

**Ex. 8.1-3**
- Show that there is no comparison sort whose running time is linear for at least half of the \( n! \) inputs of length \( n \).

Consider a decision tree of height \( h \) with \( r \) reachable leaves corresponding to a comparison sort on \( n \) elements. From Theorem 8.1 (p. 167), we have \( n!/2 \leq r \leq 2^h \). By taking logarithms,

\[
\begin{align*}
\log (n!) - 1 &= \Theta (n \log n) - 1 \\
\Rightarrow h &\geq \Theta (n \log n).
\end{align*}
\]

Thus, there is no comparison sort whose running time is linear for at least half of the \( n! \) inputs of length \( n \).

**What about a fraction of \( 1/n \) of the inputs of length \( n \)?**

Consider a decision tree of height \( h \) with \( r \) reachable leaves corresponding to a comparison sort on \( n \) elements. From Theorem 8.1 (p. 167), we have \( (1/n)n! \leq r \leq 2^h \). By taking logarithms,

\[
\begin{align*}
\log (n!) - \log n &= \Theta (n \log n) - \log n \\
\Rightarrow h &\geq \Theta (n \log n).
\end{align*}
\]

Thus, there is no comparison sort whose running time is linear for a fraction of \( 1/n \) of the \( n! \) inputs of length \( n \).

**What about a fraction of \( 1/2^n \)?**

Consider a decision tree of height \( h \) with \( r \) reachable leaves corresponding to a comparison sort on \( n \) elements. From Theorem 8.1 (p. 167), we have \( (1/2^n)n! \leq r \leq 2^h \). By taking logarithms,

\[
\begin{align*}
\log (n!/2^n) &= \Theta (n \log n) - n \\
\Rightarrow h &\geq \Theta (n \log n).
\end{align*}
\]

Thus, there is no comparison sort whose running time is linear for a fraction of \( 1/2^n \) of the \( n! \) inputs of length \( n \).

**Ex. 8.2-2**

Consider two elements in the input array, \( A[s] \) and \( A[s+1] \), such that \( A[s] = A[s+1], 1 \leq s \leq n-1 \). After the execution of the final for loop in COUNTING-SORT (p. 168), \( B[p] = A[s+1] \) and \( B[p-1] = A[s], 2 \leq p \leq n \). \( A[s] \) and \( A[s+1] \) appear in the output array \( B \) in the same order as they appear in \( A \). Therefore, COUNTING-SORT is stable.

**Ex. 8.3-3**

(i) When \( d = 1 \), RADIX-SORT(A, 1) correctly sorts \( A \).

(ii) Assume that RADIX-SORT(A, d) works when \( d \leq n-1 \).

Consider RADIX-SORT(A, n). After \( (n-1) \)-th iteration of the for loop, the elements of \( A \) are sorted by their lower \( (n-1) \) digits. Sorting on digit \( n \) orders the elements by their \( n \)-th digit; since the sort is stable, the order of those elements whose \( n \)-th digits are equal do not change. Thus, RADIX-SORT works on \( n \) digits.

**Prob. 8-6 parts (a) and (b)**

- (a)
  
  Given \( 2n \) numbers, we can choose \( n \) numbers for the first list and put the other \( n \) numbers in the second list. Thus, there are \( \binom{2n}{n} \) possible ways to divide \( 2n \) numbers into two sorted lists, each with \( n \) numbers.

- (b)
  
  The number of comparisons needed for merging two sorted lists is at least \( \log (\binom{2n}{n}) \).

Using Stirling’s approximation (equations 3.19 and 3.20 in the textbook).
\[ \lg \left( \frac{2n}{n} \right) = \lg \left( \frac{(2n)!}{n!n!} \right) \]
\[ = \sqrt{2\pi \frac{2n}{e}} \left( \frac{2n}{e} \right)^{2n} e^{-2n} \]
\[ \leq \frac{1}{12n+1} < \alpha_n < \frac{1}{12n} \quad \text{and} \quad \frac{1}{12(2n)+1} < \alpha_{2n} < \frac{1}{12(2n)} \]
\[ = \lg(\sqrt{2\pi \frac{2n}{e}} \left( \frac{2n}{e} \right)^{2n} e^{-2n}) - 2\lg(\sqrt{2\pi \frac{n}{e}} \left( \frac{n}{e} \right)^n e^{-n}) \]
\[ = \left( \frac{1}{2} \lg 4 + \frac{1}{2} \lg \pi + \frac{1}{2} \lg n + 2n \lg 2 + 2n \lg n - 2n \lg e + \alpha_{2n} \lg e \right) - 2 \left( \frac{1}{2} \lg 2 + \frac{1}{2} \lg \pi + \frac{1}{2} \lg n + n \lg n - n \lg e + \alpha_n \lg e \right) \]
\[ = 2n - \frac{1}{2} \lg n + [(\alpha_{2n} - 2\alpha_n) \lg e - \frac{1}{2} \lg \pi] \]
\[ = 2n - \frac{1}{2} \lg n + d \quad \text{where } d = [(\alpha_{2n} - 2\alpha_n) \lg e - \frac{1}{2} \lg \pi] \]

Since \( \frac{1}{2} \lg n - d \) is clearly \( o(n) \), the number of comparisons needed is at least \( 2n - o(n). \)

- **Ex. 9.2-4**
  - 3, 2, 1, 0, 7, 5, 4, 8, 6  ||  9
  - 3, 2, 1, 0, 7, 5, 4, 6  ||  8
  - 3, 2, 1, 0, 6, 5, 4  ||  7
  - 3, 2, 1, 0, 4, 5  ||  6
  - 3, 2, 1, 0  ||  5
  - 0, 2, 1  ||  3
  - 0, 1  ||  2
  - 0  ||  1

- **Ex. 9.3-5**

  SELECTwithBBMedian (A, p, r, i)  //find the i-th smallest element in A
  1  if (p==r)
  2      return A[p]
  3  Use the “black-box” worst-case linear time median subroutine to find the median, x, of the elements in A.
  4  Partition A around the median, x.
  5  q = (p+r+1)/2
  6  k = q - p + 1
  7  if i == k
  8      return A[q]
  9  else if i < k
  10  SELECTwithBBMedian (A, p, q-1, i)
  11  else
  12  SELECTwithBBMedian (A, q+1, r, i-k)

The recurrence for the worst-case time is \( T(n) = T(n/2) + O(n) + O(1) = T(n/2) + O(n). \) \( T(n) = \Theta(n) \) by the Master theorem – case 3.