Section 8.1 – Lower bounds for sorting

8.1-1 What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

The smallest possible depth of a leaf in a decision tree can be obtained by calculating the shortest simple path from the root to any of its reachable leaves. This smallest path occurs when the comparisons is made in the sorted order. For instance, if the input array is sorted, the following comparisons suffices

$$a_1 \leq a_2,$$
$$a_2 \leq a_3,$$
$$\vdots$$
$$a_{n-1} \leq an.$$

Thus, the smallest depth of a leaf is any decision tree is $n - 1 = \Theta(n)$.

8.1-2 Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.

Assume for convenience that n is even. For a lower bound, we have

$$\lg n! = \lg(n \cdot (n - 1) \cdot (n - 2) \cdots 1)$$
$$= \sum_{k=1}^{n} \lg k$$
$$= \sum_{k=1}^{n/2} \lg k + \sum_{k=n/2+1}^{n} \lg k$$
$$\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} \lg(n/2)$$
$$= \frac{n}{2} \lg \frac{n}{2}$$
$$= \frac{n}{2} \lg n - \frac{n}{2}$$
$$= \Omega(n \lg n).$$

And for an upper bound, we have

$$\lg n! = \lg(n \cdot (n - 1) \cdot (n - 2) \cdots 1)$$
$$= \sum_{k=1}^{n} \lg k$$
$$\leq \sum_{k=1}^{n} \lg n$$
$$= O(n \lg n).$$

Thus, $\lg n! = \Theta(n \lg n)$.

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. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

Such algorithm only exists if we can build a decision tree such that at least $n!/2$ of its $n!$ leaves has a depth of $\Theta(n)$. Suppose this decision tree exists. Let m be the depth of the leaf with the $(n!/2)$th smallest depth. Remove all nodes with depth greater than m. The result is a decision tree with height m and at least $n!/2$ leaves. Using the same reasoning as in the proof of Theorem 8.1, for every decision tree with at least $n!/2$ leaves, we have

$$\frac{n!}{2} \leq l \leq 2^m,$$

which implies

$$m \geq \lg \frac{n!}{2} = \lg n! - 1 = \Omega(n \lg n),$$

which proves that such a decision tree does not exists. The same reasoning can be applied to obtain the maximum depth of any fraction of the inputs. For a fraction of $1/n$, we have

$$m \geq \lg \frac{n!}{n} = \lg n! - \lg n = \Omega(n \lg n),$$

and for a fraction of $1/2^n$, we have

$$m \geq \lg \frac{n!}{2^n} = \lg n! - \lg 2^n = \lg n! - n = \Omega(n \lg n).$$

8.1-4 Suppose that you are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences. Show an $\Omega(n \lg k)$ lower bound on the number of comparisons needed to solve this variant of the sorting problem. (*Hint*: It is not rigorous to simply combine the lower bounds for the individual subsequences.)

All we know is the ordering of the elements of a given subsequence with respect to the elements of the previous/next subsequence. Thus, for each subsequence, we have $k!$ possible permutations. Since there are n/k input subsequences, the number of possible outcomes for this sorting problem is

$$\prod_{i=1}^{n/k} k! = k!^{(n/k)}.$$

We can use here the same argument used in the text book to prove a lower bound for any comparison sort algorithm. However, in this case, the number of possible permutations is $k!^{(n/k)}$, instead of $n!$. Thus, we need to show that the height of any decision tree with at least $k!^{(n/k)}$ leaves is $\Omega(n \lg k)$. We have

$$k!^{n/k} \leq l \leq 2^h,$$

which implies

$$
\begin{aligned}
h &\geq \lg \left(k!^{(n/k)} \right) \\
&= \frac{n}{k} \cdot \lg k! \\
&= \frac{n}{k} \cdot \sum_{i=1}^{k} \lg i \\
&= \frac{n}{k} \cdot \sum_{i=1}^{\lfloor k/2 \rfloor} \lg i + \frac{n}{k} \cdot \sum_{i=\lfloor k/2 \rfloor+1}^{k} \lg i \\
&\geq \frac{n}{k} \cdot \sum_{i=\lfloor k/2 \rfloor}^{k} \lg i \\
&\geq \frac{n}{k} \cdot \left(\frac{k}{2} \lg \frac{k}{2} \right) \\
&= \frac{n}{2} \lg \frac{k}{2} \\
&= \Omega(n \lg k).
\end{aligned}
$$

Section 8.2 – Counting sort

8.2-1 Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

	1	2	3	4	5	6	7	8	9	10	11			0	1	2	3	4	5	6
A	6	0	2	0	1	3	4	6	1	3	2		C	2	2	2	2	1	0	2
													C	2	4	6	8	9	9	11
B	–	–	–	–	–	2	–	–	–	–	–		C	2	4	5	8	9	9	11
B	–	–	–	–	–	2	–	3	–	–	–		C	2	4	5	7	9	9	11
B	–	–	–	1	–	2	–	3	–	–	–		C	2	3	5	7	9	9	11
B	–	–	–	1	–	2	–	3	4	–	6		C	2	3	5	7	8	9	10
B	–	–	–	1	–	2	3	3	4	–	6		C	2	3	5	6	8	9	10
B	–	–	1	1	–	2	3	3	4	–	6		C	2	2	5	6	8	9	10
B	–	0	1	1	–	2	3	3	4	–	6		C	1	2	5	6	8	9	10
B	–	0	1	1	2	2	3	3	4	–	6		C	1	2	4	6	8	9	10
B	0	0	1	1	2	2	3	3	4	–	6		C	0	2	4	6	8	9	10
B	0	0	1	1	2	2	3	3	4	6	6		C	0	2	4	6	8	9	9

8.2-2 Prove that COUNTING-SORT is stable.

> Suppose that the integer x appears k times in the output array. Since the **for** loop of lines 10-12 iterates over the input array backwards, the first integer x to be added to the output array on line 11 is the rightmost one. The decrement of the couting of x on line 12 ensures that the next integer x is added to the output array right before the previous one. This process repeats k times, until the leftmost integer x is added to the output array ($k-1$ positions before the rightmost one). This property ensures that elements with equal value in the input array appears in the same order in the output array. Thus, the algorithm is stable.

8.2-3 Suppose that we were to rewrite the **for** loop header in line 10 of the COUNTING-SORT as

     10   **for** $j = 1$ **to** $A.length$

Show that the algorithm still works properly. Is the modified algorithm stable?

> The only difference will be in the **for** loop of lines 10-12, in which elements with equal value in the input array will now be added to the output array in the same order as they appear in the input array. As observed on Question 8.2-2, each time an element with value x is added to the output array, the next element with value x is added right before the previous one. This implies that elements with equal value in the input array will appear in reverse order in the output array. Thus, this modified algorithm is not stable.

8.2-4 Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a \ldots b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

> For the preprocessing phase, build the array C in the same way it is built in the COUTING-SORT procedure (lines 1-8). This preprocessing will run in $\Theta(k) + \Theta(n) + \Theta(k) = \Theta(n + k)$. If $a > 0$, answer $C[b] - C[a - 1]$. Otherwise, answer $C[b]$.