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

$$\begin{gathered} a_1 \le a_2, \\ a_2 \le a_3, \\ \vdots \\ a_{n-1} \le an. \end{gathered}$$

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

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

And for an upper bound, we have

$$\begin{aligned} \lg n! &= \lg(n \cdot (n-1) \cdot (n-2) \cdots 1) \\ &= \sum_{k=1}^{n} \lg k \\ &\le \sum_{k=1}^{n} \lg n \\ &= O(n \lg n). \end{aligned}$$

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

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

which implies

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

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

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

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

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

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

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

which implies

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

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 $C[b]$.

Section 8.3 – Radix sort

8.3-1 Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

COW		SE**A**		T**A**B		**B**AR
DOG		TE**A**		B**A**R		**B**IG
SEA		MO**B**		E**A**R		**B**OX
RUG		TA**B**		T**A**R		**C**OW
ROW		DO**G**		S**E**A		**D**IG
MOB		RU**G**		T**E**A		**D**OG
BOX		DI**G**		D**I**G		**E**AR
TAB	$\rightarrow$	BI**G**	$\rightarrow$	B**I**G	$\rightarrow$	**F**OX
BAR		BA**R**		M**O**B		**M**OB
EAR		EA**R**		D**O**G		**N**OW
TAR		TA**R**		C**O**W		**R**OW
DIG		CO**W**		R**O**W		**R**UG
BIG		RO**W**		N**O**W		**T**AB
TEA		NO**W**		B**O**X		**T**AR
NOW		BO**X**		F**O**X		**T**EA
FOX		FO**X**		R**U**G		**S**EA

8.3-2 Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any sorting algorithm stable. How much additional time and space does your scheme entail?

INSERTION-SORT and MERGE-SORT are stable. HEAPSORT and QUICKSORT are not. To make any sorting algorithm stable, we can store the original index of each element in the array and use this index to break ties. The additional space required is $\Theta(n)$. The asymptotic running time is the same, since the number of comparisons will be at most twice.

8.3-3 Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

Let d be the number of columns in the input array, where the dth column contains the highest-order digit. RADIX-SORT sorts one column at a time, from the column with the lowest-order digits to the column with the highest-order digits. The base case occurs when $d = 1$. Since in this case the elements on the array has a single digit, calling RADIX-SORT in this case is the same as calling its sorting subroutine directly with the input array as an argument. Thus, RADIX-SORT is correct when $d = 1$. Now assume RADIX-SORT works for $d-1$ columns. Note that sorting d columns is equivalent to sorting $d-1$ columns followed by calling the sorting subroutine on the dth column. The induction hypothesis ensures that RADIX-SORT sorts $d-1$ columns correctly. Since the sorting subroutine is stable, when sorting the dth column, digits that have the same value in the dth column will be kept in the same order as it was after sorting the $(d-1)$th column. This implies that RADIX-SORT breaks ties on higher-order digits by the lower-order digits, which is correct.

The sorting subroutine must be stable since a tie that occur while sorting the ith digit is determined by the previous sorts of the lower-order digits. Since lower-order digits are sorted before higher-order digits, this property is ensured with an stable sorting algorithm.

8.3-4 Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

An integer in the range 0 to $n^3 - 1$ is represented with at most $\lg n^3 = 3\lg n$ bits. We can view a $(3\lg n)$-bit integer as three $(\lg n)$-bit integers, so that $b = 3\lg n$ and $r = \lg n$. With these settings, RADIX-SORT correctly sorts these numbers in

$$\Theta\left(\frac{b}{r}\left(n + 2^r\right)\right) = \Theta\left(\frac{3\lg n}{\lg n}\left(n + 2^{\lg n}\right)\right) = \Theta(3(n+n)) = \Theta(n).$$

8.3-5 ($\star$) In the first card-sorting algorithm in this section, exactly how many sorting passes are needed to sort d-digit decimal numbers in the worst case? How many piles of cards would an operator need to keep track of in the worst case?

Suppose that the card-sorting machine represents numbers in base p, such that $2 \leq p \leq 10$. For a given value of p, we have

- Each card uses $c = \lceil \log_p 10^d \rceil$ columns;
- Each column uses up to p places.

Let cth-digit denote the most significant digit, the $(c-1)$th-digit denote the 2nd most significant digit, and so on.
The algorithm is recursive. It starts sorting on the cth-digit, which requires p piles to distribute the cards in the worst-case. In the next level, the algorithm sorts each of the p piles on the $(c-1)$th-digit, which requires p^2 piles to distribute the cards in the worst-case (each of the p piles is splitted into p piles). This process goes for c levels. Since the piles of the previous level can be reutilized in the current level, the number of piles required to distribute the cards in all levels is the number of piles required in the last level, which is

$$p^c.$$

Note that to sort the cards on the cth-digit, only one sorting pass is needed to distribute the cards into p piles. To sort on the next digit, these p piles that were sorted on the cth-digit must now be sorted on the $(c-1)$th-digit, which will require another p^1 sorting passes to distribute them into p^2 piles. In general, to sort on the ith digit, p^{c-i} sorting passes are required in the worst-case. Thus, the number of sorting passes in the worst-case is

$$\sum_{i=1}^{c} p^{c-i} = \sum_{i=0}^{c-1} p^i = \frac{p^c - 1}{p - 1}.$$