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

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

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

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

And for an upper bound, we have

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

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 comparison sort 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

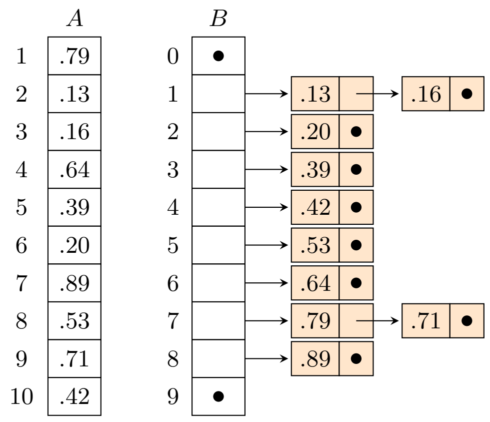
$$p^c.$$

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

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

Section 8.4 – Bucket sort

8.4-1 Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.



8.4-2 Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

The worst-case occurs when the input array is in decreasing order and every element falls into the same bucket. Since INSERTION-SORT takes $\Theta(n^2)$ to sort an array of size n that is in decreasing order, bucket sort will run in $\Theta(n^2)$.
The worst-case running time can be improved replacing INSERTION-SORT with HEAPSORT, which will make it run in $O(n \lg n)$.
As for the average-case, the expected running time of this variation of bucket sort is

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i \lg n_i)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i \lg n_i)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i \lg n_i]). \end{aligned}$$

Using the same logic adopted in the text book to compute $\mathrm{E}[n_i^2]$, we have

$$n_i = \sum_{j=1}^{n} X_{ij},$$

which implies

$$\begin{aligned} \mathrm{E}[n_i \lg n_i] &= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij} \lg \sum_{j=1}^{n} X_{ij}\right] \\ &\le \mathrm{E}\left[\sum_{j=1}^{n} X_{ij} \sum_{j=1}^{n} X_{ij}\right] \\ &= 2 - \frac{1}{n}. && \text{(from equation (8.2))} \end{aligned}$$

Thus, the average-case running time of this variation of bucket sort is

$$\mathrm{E}[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O\left(2 - \frac{1}{n}\right) = \Theta(n) + n \cdot O(1) = \Theta(n).$$

8.4-3 Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $\mathrm{E}[X^2]$? What is $\mathrm{E}^2[X]$?

Lets define the indicator random variable

$$X_i = \mathrm{I}\{\text{flip } i \text{ comes up heap}\}.$$

Thus, we have

$$X = X_1 + X_2,$$

and

$$X^2 = (X_1 + X_2)^2 = X_1^2 + 2X_1X_2 + X_2^2.$$

Note that

$$\mathrm{E}[X_i] = \Pr\{X_i = 1\} = \frac{1}{2},$$

and

$$\mathrm{E}[X_i^2] = 1^2 \cdot \frac{1}{2} + 0^2 \cdot \frac{1}{2} = \frac{1}{2}.$$

Using the above definitions and linearity of expectation, we have

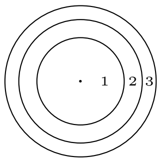
$$\begin{aligned}
\mathrm{E}[X^2] &= \mathrm{E}[X_1^2 + 2X_1X_2 + X_2^2] \\
&= \mathrm{E}[X_1^2] + 2\mathrm{E}[X_1X_2] + \mathrm{E}[X_2^2] \\
&= \frac{1}{2} + 2\left(\mathrm{E}[X_1]\mathrm{E}[X_2]\right) + \frac{1}{2} \qquad \text{(since } X_1 \text{ and } X_2 \text{ are independent)} \\
&= \frac{1}{2} + 2\left(\frac{1}{2} \cdot \frac{1}{2}\right) + \frac{1}{2} \\
&= \frac{3}{2},
\end{aligned}$$

and

$$\begin{aligned}
\mathrm{E}^2[X] &= \mathrm{E}^2[X_1 + X_2] \\
&= \mathrm{E}[X_1 + X_2]\mathrm{E}[X_1 + X_2] \\
&= (\mathrm{E}[X_1] + \mathrm{E}[X_2])(\mathrm{E}[X_1] + \mathrm{E}[X_2]) \\
&= \left(\frac{1}{2} + \frac{1}{2}\right)\left(\frac{1}{2} + \frac{1}{2}\right) \\
&= 1.
\end{aligned}$$

8.4-4 ($\star$) We are given n points in the unit circle, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \leq 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed; that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (*Hint:* Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit circle.)

Considering that the points are uniformly distributed over the area of the unit circle, we can divide the circle into n rings with equal area, such that the expected number of points in each ring is 1. The figure below illustrates a circle that is divided into three rings with equal area.



We them assign to the ith bucket the points that falls within the ith ring, sort each bucket individually with INSERTION-SORT, and combine the elements of each bucket sequentially. This is basically the BUCKET-SORT algorithm with a modification on the way we assign the elements to the buckets. Since the distribution of points over the buckets is uniform, the average-case running time of this algorithm is still $O(n)$.
We now need a function that maps a point to its bucket. Let r_i denote the (larger) radius of the ith ring. We claim that

$$r_i = \frac{\sqrt{i}}{\sqrt{n}},$$

which implies that a point p_j belongs to the ith ring if, and only if,

$$\sqrt{\frac{i-1}{n}} < d_j \leq \sqrt{\frac{i}{n}},$$

squaring both sides and multiplying by n, we have

$$i - 1 < d_j^2 \cdot n \leq i,$$

which implies

$$i = \lceil d_j^2 \cdot n \rceil.$$

Proof for the radius of the ith ring.
Since the area of the unit circle is π and each ring has equal area, for $i > 0$, we have

$$\pi r_i^2 - \pi r_{i-1}^2 = \frac{\pi}{n},$$

which implies

$$r_i = \sqrt{\frac{1}{n} + r_{i-1}^2}.$$

Note that

$$r_1 = \sqrt{\frac{1}{n} + 0^2} = \frac{\sqrt{1}}{\sqrt{n}},$$

$$r_2 = \sqrt{\frac{1}{n} + \left(\sqrt{\frac{1}{n}}\right)^2} = \frac{\sqrt{2}}{\sqrt{n}},$$

$$r_3 = \sqrt{\frac{1}{n} + \left(\sqrt{\frac{2}{n}}\right)^2} = \frac{\sqrt{3}}{\sqrt{n}},$$

which lead us to assume that

$$r_i = \frac{\sqrt{i}}{\sqrt{n}},$$

for $i = 0, 1, \ldots, n$. We shall prove it by induction. Note that it holds for $i = 0$, since

$$r_0 = \frac{\sqrt{0}}{\sqrt{n}} = 0.$$

To show that it holds for $i > 0$, we need to show that if it holds for i, it also holds for $i + 1$. We have

$$r_{i+1} = \sqrt{\frac{1}{n} + \left(\sqrt{\frac{i}{n}}\right)^2} = \sqrt{\frac{i+1}{n}} = \frac{\sqrt{i+1}}{\sqrt{n}},$$

which proves the inductive step.

8.4-5 ($\star$) A ***probability distribution function*** $P(x)$ for a random variable X is defined by $P(x) = \Pr\{X \leq x\}$. Suppose that we draw a list of n random variables $X_1, X_2, \ldots, X_n$ from a continuous probability distribution function P that is computable in $O(1)$ time. Give an algorithm that sorts these numbers in linear average-case time.

Skipped.

Problems

8-1 ***Probabilistic lower bounds on comparison sorting***
In this problem, we prove a probabilistic $\Omega(n \lg n)$ lower bound on the running time of any deterministic or randomized comparison sort on n distinct input elements. We begin by examining a deterministic comparison sort A with decision tree T_A. We assume that every permutation of A's inputs is equally likely.

a. Suppose that each leaf of T_A is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $1/n!$ and that the rest are labeled 0.

b. Let $D(T)$ denote the external path length of a decision tree T; that is, $D(T)$ is the sum of the depths of all the leaves of T. Let T be a decision tree with $k > 1$ leaves, and let LT and RT be the left and right subtrees of T. Show that $D(T) = D(LT) + D(RT) + k$.

c. Let $d(k)$ be the minimum value of $D(T)$ over all decision trees T with $k > 1$ leaves. Show that $d(k) = \min_{1 \le i \le k-1}\{d(i) + d(k-i) + k\}$. (Hint: Consider a decision tree T with k leaves that achieves the minimum. Let i_0 be the number of leaves in LT and $k - i_0$ the number of leaves in RT.)

d. Prove that for a given value of $k > 1$ and i in the range $1 \le i \le k-1$, the function $i \lg i + (k-i)\lg(k-i)$ is minimized at $i = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.

e. Prove that $D(T_A) = \Omega(n! \lg(n!))$, and conclude that the average-case time to sort n elements is $\Omega(n \lg n)$.

Now, consider a *randomized* comparison sort B. We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and "randomization" nodes. A randomization node models a random choice of the form RANDOM(l, r) made by algorithm B; the node has r children, each of which is equally likely to be chosen during an execution of the algorithm.

f. Show that for any randomized comparison sort B, there exists a deterministic comparison sort A whose expected number of comparisons is no more than those made by B.

(a) First note that there are $n!$ distinct permutations of an input of size n with distinct elements. To be a valid decision tree, T_A must have each of these permutations as a reachable leaf. Since every permutation is equally likely, each of these leaves must be reachable with probability $1/n!$. If T_A has any additional leaf, it is not reachable for any input and has probability 0.

(b) Note that, since $k > 1$, the root of T if not a leaf. Also, the only node that is present in T and is not present in the subtrees LT and RT together is the root of T. This implies that each leaf at depth d in T is either in LT or in RT, but at depth $d-1$. Since there are k leaves in T, we have $D(T) = D(LT) + D(RT) + k$.

(c) First, we will show that

$$d(k) \le \min_{1 \le i \le k-1}\{d(i) + d(k-i) + k\},$$

by showing that

$$d(k) \le d(i) + d(k-i) + k,$$

for $i = 1, \dots, k-1$. For every i such that $1 \le i \le k-1$, there exist decision trees LT and RT with i leaves and $k-i$ leaves, respectivelly, such that $D(LT) = d(i)$ and $D(RT) = d(k-i)$. Let T be a decision tree composed by a root node and LT and RT as its left and right subtrees, respectivelly. Note that T has k leaves, since it has all the leaves from LT and RT, and its root is not a leaf. Then, we have

$$\begin{aligned} d(k) &\le D(T) && \text{(from the definition of } d(\cdot)) \\ &= D(LT) + D(RT) + k && \text{(from item (b))} \\ &= d(i) + d(k-i) + k. \end{aligned}$$

Now, we will show that

$$d(k) \ge \min_{1 \le i \le k-1}\{d(i) + d(k-i) + k\},$$

by showing that

$$\exists\, i \in 1, \dots, k-1 \mid d(k) \ge d(i) + d(k-i) + k.$$

Let T be a decision tree with k leaves such that $D(T) = d(k)$. Let LT and RT be the left and right subtrees of T and let i and $k-i$ be the number of leaves of LT and RT, respectivelly. We have

$$\begin{aligned} d(k) &= D(T) && \text{(from the definition of } T) \\ &= D(LT) + D(RT) + k && \text{(from item (b))} \\ &\ge d(i) + d(k-i) + k. \end{aligned}$$

Then, we can conclude that

$$d(k) = \min_{1 \le i \le k-1}\{d(i) + d(k-i) + k\}.$$

(d) Let $f(i) = i \lg i + (k-i)\lg(k-i)$. We have

$$\begin{aligned} f'(i) &= \frac{d}{di}\left(i \lg i + (k-i)\lg(k-i)\right) \\ &= \frac{d}{di}\left(\frac{i \ln i + (k-i)\ln(k-i)}{\ln 2}\right) \\ &= \frac{d}{di}\left(\frac{i \ln i}{\ln 2} + \frac{(k-i)\ln(k-i)}{\ln 2}\right) \\ &= \frac{1+\ln i}{\ln 2} + \frac{-1-\ln(k-i)}{\ln 2} \\ &= \frac{\ln i - \ln(k-i)}{\ln 2} \\ &= \lg i - \lg(k-i), \end{aligned}$$

which is 0 when $i = k/2$. Also note that

$$f'(1) = f'(k-1) = \ln(k-1) \geq \ln 1 = 0 = f(k/2),$$

which implies that the point $i = k/2$ is a minimum. We shall now prove that $d(k) = \Omega(k \lg k)$. Our guess is

$$d(i) \geq ci \lg i \;\; \forall\, 1 \leq i \leq k-1,$$

where c is a positive constant. Substituting into the recurrence, yields

$$\begin{aligned} d(k) &= \min_{1 \leq i \leq k-1} \{d(i) + d(k-i) + k\} \\ &\geq \min_{1 \leq 1 \leq k-1} \{c(i \lg i + (k-i)\lg(k-i)) + k\} \\ &= \min_{1 \leq 1 \leq k-1} \{cf(i) + k\} \\ &= cf\left(\frac{k}{2}\right) + k \\ &= c\left(\frac{k}{2}\lg\frac{k}{2} + \frac{k}{2}\lg\frac{k}{2}\right) + k \\ &= ck \lg\frac{k}{2} + k \\ &= ck \lg k - ck + k \\ &\geq ck \lg k, \end{aligned}$$

where the last step holds as long as $c \leq 1$.

(e) Since T_A has $n!$ leaves, we have

$$D(T_A) \geq d(n!) = \Omega(n! \lg n!).$$

Note that the external path length of a decision tree denotes the number of comparisons needed to sort every possible permutation of an input of size n. Since each of the $n!$ permutations have the same probability of happening (from item (a)), the expected number of comparisons for each input is

$$\begin{aligned} \frac{\Omega(n! \lg(n!))}{n!} &= \Omega(\lg(n!)) \\ &= \Omega(n \lg n). \quad \text{(from (3.19))} \end{aligned}$$

(f) Just replace each randomization node with one of its subtrees, particularly the one with the smaller external path. The result is a valid deterministic tree with no more comparisons than the radomized tree. Thus, we can conclude that the number of comparisons for any randomized comparison sort is also $\Omega(n \lg n)$.

8-2 ***Sorting in place in linear time***

Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

a. Give an algorithm that satisfies criteria 1 and 2 above.

b. Give an algorithm that satisfies criteria 1 and 3 above.

c. Give an algorithm that satisfies criteria 2 and 3 above.

d. Can you use any of your sorting algorithms form parts (a)−(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records with b-bit keys in $O(bn)$ time? Explain how or why not.

e. Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that it sorts the records in place in $O(n+k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable? (*Hint:* How would you do it for $k = 3$?)

(a) The following is a modified COUNTING-SORT for data records. If runs in $\Theta(n)$, is stable, but do not sort in place.

```
Counting-Sort-Binary-Records(A, B)
    one-pos = 1
    zero-pos = 1
    for i = 1 to A.length do
        key = A[i].key
        if key == 0 then
            one-pos = one-pos + 1
    for i = 1 to A.length do
        key = A[i].key
        if key == 0 then
            B[zero-pos] = A[i]
            zero-pos = zero-pos + 1
        else
            B[one-pos] = A[i]
            one-pos = one-pos + 1
```

(b) The following uses a technique similar to the one used for the HOARE-PARTITION, introduced on Question 7-1. It runs in $\Theta(n)$, sorts in place, but is not stable.

```
InPlace-Sort-Binary-Records(A)
    i = 1
    j = A.length
    while True do
        while j > i and A[j].key == 1 do
            j = j − 1
        while i < j and A[i].key == 0 do
            i = i + 1
        if i < j then
            swap A[i] with A[j]
        else
            break
```

(c) The following is a modified INSERTION-SORT for data records. It is stable, sorts in place, but is not linear.

```
Insertion-Sort-Binary-Records(A)
    for j = 2 to A.length do
        key = A[j].key
        i = j − 1
        while i > 0 and A[i].key > key do
            A[i + 1] = A[i]
            i = i − 1
        A[i + 1] = key
```

(d) To be used as a subroutine of Radix-Sort, the sorting algorithm must be stable. Also, in order to Radix-Sort run in $O(bn)$ time, the sorting subroutine must run in $O(n)$. The only algorithm that satisfies both of these constraints is the one on part (a).

(e) Roughly speaking, instead of copying each element to its correct position in a new array, we swap each element with the element that is in its correct position. The algorithm runs in $\Theta(n+k)$, sorts in place, but is not stable. The pseudocode is stated below.

```
Counting-Sort-InPlace($A$, $k$)
let $C[0,\dots,k]$ be a new array
let $T[0,\dots,k]$ be a new array
for $i = 0$ to $k$ do
    $C[i] = 0$
for $i = 1$ to $A.length$ do
    $key = A[i].key$
    $C[key] = C[key] + 1$
for $i = 1$ to $k$ do
    $C[i] = C[i] + C[i-1]$
for $i = 0$ to $k$ do
    $T[i] = C[i]$
$pos = 1$
while $pos < A.length$ do
    $key = A[pos].key$
    if $C[key] > pos$ then
        swap $A[pos]$ with $A[C[key]]$
        $C[key] = C[key] - 1$
    else
        $pos = pos + T[key]$
```

8-3 ***Sorting variable-length items***

a. You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over all the integers in the array is n. Show how to sort the array in $O(n)$ time.

b. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time. (Note that the desired order here is the standard alphabetical order; for example, `a` $<$ `ab` $<$ `b`.)

(a) Let m denote the number of integers in the input array. Start sorting the numbers by their sign, using an algorithm similar to COUNTING-SORT-BINARY-RECORDS, in which negative numbers are placed before positive numbers. This step runs in $O(m)$. Let m_{pos} and m_{neg} denote the number of positive and negative integers, respectivelly. For each group of numbers with the same sign, sort the elements in the group by their number of digits with COUNTING-SORT. This step will take $O(m_{pos} + n) + O(m_{neg} + n) = O(n)$. Let m_i denote the number of integers with i digits. For each group of numbers with i digits, sort the elements in the group with RADIX-SORT. Since each call to RADIX-SORT runs in $O(i \cdot (m_i + 10)) = O(i \cdot m_i)$, this step will take

$$\sum_{i=1}^{n} O(i \cdot m_i) = O\left(\sum_{i=1}^{n}(i \cdot m_i)\right) = O(n).$$

Thus, the running time of this algorithm is

$$O(m + n + n) = O(n).$$

(b) Note that, different from the problem of sorting integers, we can not use the length of the string to sort the elements – strings with longer length does not imply any order with respect to strings with smaller length. However, the leftmost character of each string does imply an order. When the first character of two strings are the same, the strings are untied by their following character, and so on until the last character. We can use this notion to derive a recursive algorithm that sorts and groups the strings by their first character, and recursively sorts each group by their following character. Note that strings that do not have the current character should be placed before the ones that does have the character. Since the range of characters is constant, we can sort each group using counting sort in linear time on the number of elements in the group. To avoid creating multiple subarrays in each recursion call, COUNTING-SORT-INPLACE is more suitable.

Let m denote the number of strings in the input array and let n_i denote the number of characters in the ith string. Since counting sort is linear on the number of elements being sorted and the ith string are considered in at most $n_i + 1$ calls of counting sort (one additional time for the special case when the string does not have the following character), the total cost of this algorithm is

$$O\left(\sum_{i=1}^{m}(n_i + 1)\right) = O\left(\sum_{i=1}^{m} n_i + m\right) = O(n + m) = O(n).$$

8-4 ***Water jugs***

Suppose that you are given n red and n blue water jugs, all of different shapes and sizes. All red jugs hold different amounts of water, as do the blue ones. Moreover, for every red jug, there is a blue jug that holds the same amount of water, and vice versa.

Your task is to find a grouping of the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation will tell you whether the red or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

a. Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.

b. Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons that an algorithm solving this problem must take.

c. Give a randomized algorithm whose expected number of comparisons is $O(n \lg n)$, and prove that this bound is correct. What is the worst-case number of comparisons for your algorithm?

(a) A brute-force algorithm runs in $\Theta(n^2)$. Compare the volume of each red jub to each blue jug until a match is found. The pseudocode is stated below.

```
Water-Jugs-Brute-Force($R$, $B$)
    $n = R.length$
    let $P$ be a new array of size $n$
    for $i = 1$ to $n$ do
        for $j = 1$ to $n$ do
            if Volume($R[i]$) == Volume($B[j]$) then
                $P[i] = (R[i], B[j])$
                break
```

(b) We shall use a decision tree to compute a lower bound on the worst-case number of comparisons needed to solve this problem. Let each node on the tree be a comparison between one red jug and one blue jug. Each comparison results in one of three results, which implies that each non-leaf node in the decision tree has three children. To compute the number of leaves, note that this problem is the same as sorting the array of blue jugs based on the current order of the array of red jugs. Since the elements of each array are distinct, there are $n!$ different permutations of the array of blue jugs array and only one of them correspond to the order of the array of red jugs, which then implies that the decision tree must have at least $n!$ leaves. Note that the height h of the decision tree is equal to the worst-case number of comparisons needed to solve the problem. Thus, we have

$$3^h \geq n!,$$

which implies

$$\begin{aligned} h &\geq \log_3(n!) \\ &= \frac{\lg n!}{\lg 3} \\ &= \frac{\Omega(n \lg n)}{\lg 3} \quad \text{(from (3.19))} \\ &= \Omega(n \lg n). \end{aligned}$$

(c) We can use an algorithm very similar to Randomized-Quicksort. The partition procedure also need to be updated in order to receive the pivot as an argument. The pseudocode is stated below.

```
Partition-Jugs($A$, $s$, $e$, $x$)
    $i = s - 1$
    for $j = s$ to $e - 1$ do
        $v_j$ = Volume($A[j]$)
        $v_x$ = Volume($x$)
        if $v_j == v_x$ then
            exchange $A[j]$ with $A[e]$
        if $v_j < v_x$ then
            $i = i + 1$
            exchange $A[i]$ with $A[j]$
    exchange $A[i + 1]$ with $A[e]$
    return $i + 1$
```

```
Group-Jugs($R$, $B$, $s$, $e$, $G$)
    if $e < s$ then
        return
    else if $e == s$ then
        Insert($G, (R[s], B[s])$)
    else
        $i$ = Random($s, e$)
        $b$ = Partition-Jugs($B, s, e, R[i]$)
        $r$ = Partition-Jugs($R, s, e, B[b]$)
        Insert($G, (R[r], B[b])$)
        Group-Jugs($R, B, s, i - 1$)
        Group-Jugs($R, B, i + 1, e$)
```

The analysis this algorithm is almost the same as the analysis of RANDOMIZED-QUICKSORT, which gives an expected running time of $O(n \lg n)$. Also just like in RANDOMIZED-QUICKSORT, the worst-case occurs when partition always make bad partitions, which gives a running time of $\Theta(n^2)$.

8-5 Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an n-element array A ***k*-sorted** if, for all $i = 1, 2, \ldots, n-k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \le \frac{\sum_{j=i+1}^{i+k} A[j]}{k}.$$

a. What does it mean for an array to be 1-sorted?

b. Give a permutation of the numbers $1, 2, \ldots, 10$ that is 2-sorted, but not sorted.

c. Prove that an n-element array is k-sorted if and only if $A[i] \le A[i+k]$ for all $i = 1, 2, \ldots, n-k$.

d. Give an algorithm that k-sorts an n-element array in $O(n \lg(n/k))$ time.

We can also show a lower bound on the time to produce a k-sorted array, when k is a constant.

e. Show that we can sort a k-sorted array of length n in $O(n \lg k)$ time. (*Hint:* Use the solution to Exercise 6.5-9.)

f. Show that when k is a constant, k-sorting an n-element array requires $\Omega(n \lg n)$ time. (*Hint:* Use the solution to the previous part along with the lower bound on comparison sorts.)

(a) It means that it is sorted.

(b) $\langle 2, 1, 4, 3, 6, 5, 8, 7, 10, 9 \rangle$.

(c) Note that

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} = \frac{\sum_{j=i+1}^{i+k-1} A[j]}{k} + \frac{A[i]}{k},$$

and

$$\frac{\sum_{j=i+1}^{i+k} A[j]}{k} = \frac{\sum_{j=i+1}^{i+k-1} A[j]}{k} + \frac{A[i+k]}{k}.$$

Thus, we have

$$\frac{\sum_{j=i+1}^{i+k-1} A[j]}{k} + \frac{A[i]}{k} \le \frac{\sum_{j=i+1}^{i+k-1} A[j]}{k} + \frac{A[i+k]}{k} \iff \frac{A[i]}{k} \le \frac{A[i+k]}{k}. \iff A[i] \le A[i+k].$$

(d) We can use a modification of Randomized-Quicksort that only sorts subarrays with size greater than k. The modified pseudocode is stated below.

```
Randomized-Quicksort-K($A$, $p$, $r$, $k$)
    **if** $r - p \ge k$ **then**
        $q$ = Randomized-Partition($A, p, r$)
        Randomized-Quicksort($A, p, q-1, k$)
        Randomized-Quicksort($A, q+1, r, k$)
```

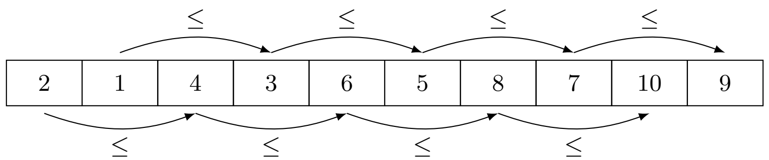
Correctness can be seen as follows. Note that at the start of each recursive call of Randomized-Quicksort-K, every element of the subarray $A[p, \ldots, r]$ is greater than or equal to the elements before the subarray and is smaller than or equal to the elements after the subarray. That is, at the start of each recursive call, the elements of the subarray $A[p, \ldots, r]$ can only be inverted with themselves. Since the base case of the recursion occurs when $r - p < k$, after running Randomized-Quicksort-K we can ensure that every inversion (i, j) has $j - i < k \to j < i + k$, which implies

$$A[i] \le A[i+k],$$

for all $i = 1, 2, \ldots, n-k$. From item (c), this property implies that the array is correctly k-sorted.

As for the running time, we can use a similar argument as the one used on question 7.4-5. Thus, the expected running time of Randomized-Quicksort-K is $O\left(n \lg \frac{n}{k}\right)$.

(e) A k-sorted array is composed of k 1-sorted subarrays where the ith subarray, $i = 0, 1, \ldots, k-1$, is formed by the elements with index j such that $j \bmod k = i$. For instance, the following 2-sorted array has two 1-sorted subarrays:



The Merge-Lists-Min-Heap algorithm (Question 6-5.9) can be used to merge these k subarrays in $O(n \lg k)$-time.

(f) Item (e) shows that a k-sorted array has k 1-sorted subarrays. Note that each of these subarrays must have at least $\lfloor n/k \rfloor$ elements. Thus, since k is a constant, the lower bound to sort each of these subarrays is

$$\Omega\left(\frac{n}{k} \lg \frac{n}{k}\right) = \Omega\left(\frac{n}{k} \lg n - \frac{n}{k} \lg k\right) = \Omega(n \lg n),$$

which implies that $\Omega(n \lg n)$ is also the lower bound to k-sort the whole array.

8-6 ***Lower bound on merging sorted lists***

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine MERGE in Section 2.3.1. In this problem, we will prove a lower bound of $2n - 1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items.

First we will show a lower bound of $2n - o(n)$ comparisons by using a decision tree,

a. Given $2n$ numbers, compute the number of possible ways to divide them into two sorted lists, each with n numbers.

b. Using a decision tree and your answer to part (a), show that any algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons.

Now we will show a slightly tigher $2n - 1$ bound.

c. Show that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

d. Use your answer to the previous part to show a lower bound of $2n - 1$ comparisons for merging two sorted lists.

(a) Note that every list of elements only have one permutation that is in sorted order. Thus, this problem can be seen as counting the number of ways to pick n numbers out of $2n$ numbers to form the first list, and using the remaining numbers to form the second list. We can count that with the binomial notation

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}\left(1 + O\left(\frac{1}{n}\right)\right),$$

in which the approximation is proved in (C.1-13).

(b) The input of the merge algorithm are the two sorted lists and we know from item (a) that the number of possible ways to form these lists elements is $\binom{2n}{n}$. Note that these pairs of sorted lists are unique and each of them can be the final merged list if placed side by side. Thus, the number of permutations of the input to form the final merged list is also $\binom{2n}{n}$. We can compute a lower bound on the number of comparisons to solve this problem by determining the height of the decision tree with at least $\binom{2n}{n}$ leaves. Let h be the height of such a decision tree. Thus, we have

$$\begin{aligned} 2^h &\geq \frac{2n}{n} \\ &= \frac{2^{2n}}{\sqrt{\pi n}}\left(1 + O\left(\frac{1}{n}\right)\right), \end{aligned}$$

which implies

$$\begin{aligned} h &\geq \lg\left(\frac{2^{2n}}{\sqrt{\pi n}}\left(1 + O\left(\frac{1}{n}\right)\right)\right) \\ &= \lg(2^{2n}) - \lg(\sqrt{\pi n}) + \lg\left(1 + O\left(\frac{1}{n}\right)\right) \\ &= 2n - o(n). \end{aligned}$$

(c) Let A and B denote the two input sorted lists and assume without loss of generality that their elements are distinct. Let $a \in A$ and $b \in B$ denote two elements that are consecutive in the sorted order. Note that since they are consecutive, the number of elements that are before/after a in the sorted list is equal to the number of elements before/after b in the sorted list. This implies that if we compare a to every element of B except b and compare b to every element of A except a, we will find the two possible positions for them in the sorted list, but we will not be able to determine which one comes first.

(d) In the worst case, every pair of elements in the final list comes from different input lists. There are $2n$ elements in the final list and $2n - 1$ pairs of consecutive elements. Thus, $2n - 1$ is a lower bound on the worst-case number of comparisons to merge two sorted lists.

8-7 ***The 0-1 sorting lemma and columnsort***
Very long statement. Read it on the text book.

Skipped.